AN ALGORITHM FOR PARALLEL UNSTRUCTURED MESH GENERATION AND FLOW ANALYSIS

by

Tolulope O. Okusanya

B.Sc. Mechanical Engineering
Rutgers University, New Jersey, 1994

SUBMITTED TO THE DEPARTMENT OF
AERONAUTICS AND ASTRONAUTICS
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF SCIENCE
at the
Massachusetts Institute of Technology

February 2, 1996

Signature of Author _____
Department of Aeronautics and Astronautics
February 2, 1996

Certified by _____
Professor Jaime Peraire, Thesis Supervisor
Department of Aeronautics and Astronautics

Accepted by _____
Professor Harold Y. Wachman, Chairman
Department Graduate Committee

PARALLEL UNSTRUCTURED MESH GENERATION AND FLOW ANALYSIS

by

Tolulope O. Okusanya

Submitted to the Department of Aeronautics and Astronautics

on February 2, 1996

in partial fulfillment of the requirements for the degree of

Master of Science in Aeronautics and Astronautics

Abstract

A strategy for the parallel generation of unstructured meshes is proposed. A distributed unstructured mesh generation environment is presented and this is coupled with a time dependent compressible Navier-Stokes equations solver. The mesh generation schemes developed in the serial context are extended for parallel execution. Dynamic load-balancing and mesh migration are incorporated to ensure even work distribution and for refinement analysis. Anisotropic mesh refinement is also incorporated for flow analysis involving anisotropic gradients. The flow solver is linked to the mesh generation environment as a test of the parallel mesh generation capabilities for flow analysis and several well defined test cases are modeled for comparison to experimental results.

Acknowledgements

First of all, I would like to thank my thesis advisor, Professor Jaime Peraire for his invaluable support and encouragement over the past year. He has shown me new and interesting views and was always a constant source of ideas. I look forward to working with him in the future and wish him the best in his new role as a father.

A very special thanks goes to Dr. Andréa Froncioni and Professor Richard Peskin who started me off in the field of CFD. It was a real pleasure working with them at Rutgers. Here is to "Andy" in the hopes of his settling down long enough for a few bouncing baby bambinos.

Thanks goes to all my friends at CASL, GTL and SSL who have moved on and are still here. To those who endured the nightmares of taking the qualifiers and/or writing a thesis, pray you do not have to do such again. Things would simply not be the same without mentioning Folusho, Karen, Graeme, Ray, Jim, Carmen, Guy (Go RU !), Angie, Ed and K.K. (Remember guys, that was no ordinary chicken !!). Ali deserves special mention because like it or not, our fates are tied together somehow.

I also wish to thank the "arcade" group (that's right, you know yourselves !) especially the other members of Team Megaton Punch for at least dragging me out of the lab (kicking and screaming) to go have fun. What would life have been without hearing phrases like "Feel the power little guy"?

Of course, I wish to thank my parents for their unconditional love and support without which I would never have been here in the first place. Lastly, I wish to thank God because there is only so far you can go by yourself.

私 の しゅうろん は おわり だ!

まかい へ の 門 を ひらけ!

ふたたび よ に カオス を おころ!

- Hakai no Tenshi

Contents

List of Figures

List of Tables

List of Symbols

Ω	Computational domain
Ω_k	Global element k (Serial)
$\Omega_{j,k}$	Local element k in subdomain j (Parallel)
$\mathrm{R^d}$	Computational dimension
$\{\mathbf{V}_i\}$	Voronoï regions for point set $\{\mathbf{x}_i\}$
$\mathrm{R_f}$	Rebay factor
$\mathrm{S_i}$	Minimum distance from point i to boundary
δ_0	Minimum anisotropic refinement spacing
$\mathrm{r_g}$	Anisotropic geometric growth ratio
$\mathrm{G_f}$	Growth limiter factor
$\mathrm{N_p}$	Number of processors
$\mathrm{T_j^i}$	Global element identification tag
Ω^m	Migrated element set
${\Omega_C}^m$	Contiguous subset of migrated element set
${n_x}^B$	Number of X bands
${n_y}^B$	Number of Y bands
${}^G N_e$	Global number of elements
${}^G N_p$	Global number of vertices
$\mathrm{^G N_{pf}}$	Global number of processor fronts
$B_x[i]$	Required average number of elements in X band (i)
$B_y[i,j]$	Required average number of elements in Y band (i,j)

$\{\Omega_i{}^r\}$	Initial element set before remote element request
$\{\Omega_f{}^r\}$	Final element set after remote element request
N_r	Number of remote request processors
$\{\Omega^t\}$	Granted remote element set for request transfer
$\{F_i{}^r\}$	Initial front set before remote element request
$\{F_f{}^r\}$	Final front set after remote element request
p	Pressure
ρ	Density
u, v	2D velocity components
E	Total energy
H	Total enthalpy
$\vec{U}$	State vector
$\vec{F}, \vec{G}$	Flux components
γ	Ratio of specific heats
Pr	Prandtl number
Re	Reynolds number
c	Local speed of sound
T	Local temperature
$\tau_{xx}, \tau_{yy}, \tau_{xy}, \tau_{yx}$	2D stress components
μ	Viscosity coefficient
$w_x, w_y, \bar{w}_x, \bar{w}_y$	Edge based weights
S_P	Pressure switch
Δt	Local time step
CFL	Courant-Friedrichs-Lewy condition
S_f	Safety factor
$\mathcal{F}_s(\mathbf{v_i})$	Shared vertex mapping

Chapter 1

Introduction

1.1 Overview

Unstructured mesh methods for computational fluid dynamics have experienced a rapid growth 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 [21, 25, 40]. 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 mesh generation. Currently, the serial mesh generator within the unstructured mesh system FELISA [25] generates grids at an approximate constant rate of 3 million elements per hour on a high end workstation. It is conceivable that for the large grids required for viscous turbulent flow analysis, mesh generation may become a problem. More important however, are the memory requirements. At present, the size of a computation is determined by the ability to generate the grid on a serial machine. For large applications, the memory requirements can be a

problem. For instance, the generation of an unstructured grid containing 8 million cells requires a serial machine with 512Mb of core memory using the FELISA system [25] . Once generated, this grid needs to be partitioned and sent to the parallel processors to carry out the flow solution. Hence, a strategy capable of generating a grid in parallel mode and such that when the grid generation process is finished, the grid is already partitioned and in place for the flow solution, is necessary.

This thesis addresses the development of a parallel unstructured mesh generator designed to operate on parallel MIMD machines with distributed memory. Serial mesh generation is a prerequisite to any form of parallel mesh generation since this is essentially mesh generation on a single processor. The serial mesh generation algorithms are detailed in chapter 2 and these deal with the algorithms, data structures and mesh representation. Chapter 3 deals with the extension of isotropic mesh generation to anisotropic mesh generation for analysis involving anisotropic gradients. Parallel systems based on architecture and memory organization is discussed in chapter 4. This chapter also discusses the programming model adopted for the purpose of parallel mesh generation. The development of a parallel mesh generation system is discussed in chapter 5 and this deals with the topics of mesh migration, load balancing and the parallel extension of the serial mesh generation algorithms. The results of the serial and parallel mesh generation systems are presented in chapter 6. The mesh generation system can only be truly tested under an actual CFD application and the formulation of an explicit 2D compressible Navier-Stokes equation solver is presented in chapter 7. The results for several test configurations is presented in this chapter with conclusions and recommendations for future work in chapter 8.

Chapter 2

Mesh Generation Algorithms

2.1 Preliminaries

A general description of a computational mesh in R^d may be given as a set of unique global coordinate vertices, $\{\mathbf{x}_i \mid i = 1..M\}$ along with some form of connectivity information regarding this set of points. Computational grids can be broadly subdivided into two major classes: *structured* and *unstructured* although mixed or hybrid grids also exist. Structured meshes in general do not require the connectivity information as this may be deduced from the point set distribution. Unstructured meshes require this connectivity due to the lack of a logical structure to the point set distribution. This may be thought of in the sense that the number of connectivity paths meeting at a point varies from point to point. The meshes considered here will be such that the connectivity information associated with any given grid satisfies the following constraints.

- The connectivity information is contained in a set of closed polyhedral elements Ω_k which span the domain.
- Any given pair of polyhedral elements in R^d may share up to n global points.
- Any given set of (d-1) points which are coplanar and form an element face can be shared by at most two elements. In the case of boundary faces, these are shared by exactly one element for external boundaries and possibly two elements for internal

boundaries.

Given this set of constraints, we observe that the polyhedral elements (Ω_k) have been defined in a way such that for any given domain Ω, the elements satisfy

$$\Omega = \bigcup_k \Omega_k \tag{2.1}$$

which implies that the elements are non-overlapping and tile the entire domain. The polyhedral elements considered for grid generation are triangles in 2D and tetrahedra in 3D.

2.1.1 Delaunay Triangulation

A triangulation of a given set of points in R^d may be effected by means of *Dirichlet tessellation*. The Dirichlet tessellation of a point set $\{\mathbf{x}_i\}$ is defined as the pattern of convex regions $\{\mathbf{V}_i\}$ which is formed by assigning to each point $\mathbf{x}_i$, a region $\mathbf{V}_i$ which represents the space closer to point $\mathbf{x}_i$ than any other point. These regions satisfy the property

$$\mathbf{V}_i = \{\mathbf{x}_i : |\mathbf{x} - \mathbf{x}_i| < |\mathbf{x} - \mathbf{x}_j|\} \quad \forall\, j \neq i \tag{2.2}$$

The resultant convex polyhedra are called Voronoï regions and cover the entire domain. The Delaunay triangulation of a point set is then formed by considering any point pair with a common Voronoï boundary segment and joining them together. This results in a triangulation of the convex hull of the point set $\{\mathbf{x}_i\}$ and is also referred to as the *dual* of the Voronoï diagram of the point set.

If Delaunay triangulation is considered in R^2, we observe that since each line segment of the Voronoï diagram is equidistant from the two points it separates, then each

vertex of the Voronoï diagram must be equidistant from the three nodes which form the Delaunay triangle that encloses the vertex. This is to say that the vertices of the Voronoï diagram are the *circumcenters* of the Delaunay triangles. In R^3, this translates to the vertices of the Voronoï diagram being the *circumcenters* of the Delaunay tetrahedra.

The Delaunay triangulation satisfies a number of properties in R^2 not all of which have extensions to R^3. These include

1. *Completeness.* The Delaunay triangulation covers the convex hull of all points.

2. *Uniqueness.* The Delaunay triangulation is unique except for degeneracies such as 4 or more cocircular points (in R^2) and 5 or more cospherical points (in R^3).

3. *Circumcircle/Circumsphere criteria.* A triangulation of N points is Delaunay if and only if every circle passing through the three vertices of a triangle in 2D (sphere passing through the four vertices of a tetrahedra in 3D) does not contain any other point.

4. *Edge circle property.* A triangulation of N points in 2D is Delaunay if and only if there exists some circle passing through the endpoints of every edge which is point-free. This can be extended to 3D for element faces.

5. *Equiangularity property.* The Delaunay triangulation of a given set of points maximizes the minimum angle of the triangulation. Hence the Delaunay triangulation is also called the MaxMin triangulation. This only holds in R^2.

6. *Minimum Containment Circle(Sphere).* The Delaunay triangulation minimizes the maximum containment circle(sphere) over the entire triangulation. The containment circle(sphere) is defined as the smallest circle(sphere) enclosing the vertices of a triangle(tetrahedron).

7. *Nearest neighbor property.* An edge formed by connecting a vertex to the nearest neighbor is always an edge of the Delaunay triangulation.

8. *Minimal roughness.* Given an arbitrary set of data f_i defined on the vertices of the mesh such that f_i varies as a piecewise linear function over the elements. For every possible triangulation, the Delaunay triangulation minimizes the functional

$$I = \int_{\Omega} \nabla f . \nabla f \, d\Omega \tag{2.3}$$

2.1.2 Constrained Delaunay Triangulation

For a given point set $\{\mathbf{x}_i \mid i = 1..M\}$, the boundary of the Delaunay triangulation of the point set is the convex hull of the point set. However, if the Delaunay triangulation of a point set with respect to a prescribed set of fixed edges which bounds the domain is considered, this is referred to as a *Constrained Delaunay triangulation.* This prescribed set of edges may represent the domain boundary or interior boundaries. In most practical applications, we will be interested in constrained triangulations.

2.2 Delaunay Triangulation Algorithms

For the purposes of mesh generation, *Incremental Insertion* algorithms are considered. These algorithms are designed in such a way that given a point set, the points are considered in sequence and inserted into the triangulation in such a way as to ensure that the grid is always locally Delaunay. Due to the localized nature of these algorithms, global Delaunay cannot always be guaranteed. Several incremental insertion strategies exist such as the Bowyer algorithm [1] and Randomized algorithms [30]. However for the purpose of computational simplicity, we consider only the Watson algorithm [12]

and the Green-Sibson algorithm [43]. These are outlined in 2D but are extendable to 3D.

2.2.1 Watson Point Insertion Algorithm

The Watson point insertion strategy is based on the circumcircle property of the Delaunay triangulation. Given a point Q to be inserted into the current triangulation, the *root* element is defined as any element whose circumcircle contains the point. From the root element, a *Breadth First Search* (BFS) is performed to locate all elements whose circumcircle contains the given point. The BFS is a tree search which is accomplished by checking the neighbors of all the currently identified elements which violate the circumcircle test and considering only those elements which have not been tested. This may be done recursively or with the help of supporting data structure and is always guaranteed to terminate. This generated set of elements is always independent of the element selected as root. This search method is particularly suited to constrained triangulations where an element may not be visible to the point Q due to a prescribed edge. For constrained triangulations however, one required modification is to make sure that the root element is on the "same" side of the prescribed edge as the point Q by testing if the centroid of the root element is on the "same" side as Q. Deletion of the set of elements which violate the circumcircle condition results in a polygonal cavity surrounding the point Q. This cavity is then retriangulated by connecting the point Q to the vertices of the polygonal cavity. This guarantees that the triangulation is always locally Delaunay around the point Q. The implementation steps of the algorithm are listed below.

1. Insert new point Q into existing point set.

2. Locate root element with circumcircle containing point.

3. Perform tree search to obtain element set which violates circumcircle property.

4. Construct polygonal cavity edges and delete element set.

5. Connect cavity vertices to point Q and update Delaunay data structure.

This is the most general form of implementation for a given point set. In the process of mesh generation with constrained Delaunay triangulation, due consideration must be given to boundary violations also. Hence new points which are to be inserted into the triangulation are only inserted if there are no boundary or proximity violations with respect to the identified element set associated with the point Q. Proximity violations as defined in this context means that the new point Q may not be closer to any existing vertex than some specified tolerance.

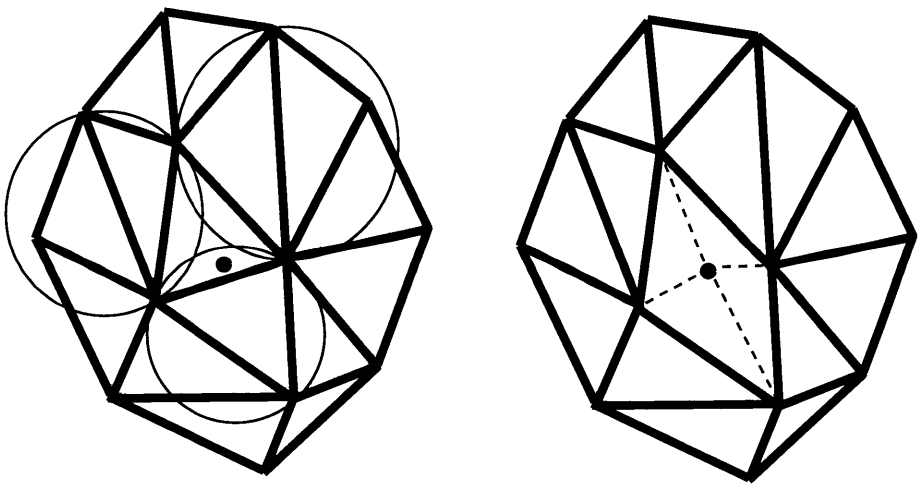


Figure 2.1: Watson Point Insertion Strategy

2.2.2 Green-Sibson Point Insertion Algorithm

The Green-Sibson point insertion strategy is based on the circumcircle property of the Delaunay triangulation. The difference is that local edge transformations (edge swapping) are employed to reconfigure the triangulation. Given the point Q to be inserted into the triangulation, the *root* element is defined as the element that encloses the point Q. Upon the location of the root element, three new edges and elements are created by connecting Q to the vertices of the root element and deleting the root element. If the point lies on an edge, the edge is deleted and four edges are created connecting the point Q to the vertices of the quadrilateral formed. For the purpose of computational simplicity in mesh generation, if the point lies on an edge, then it is moved by small ΔL along the normal to the edge into the root element. Based on

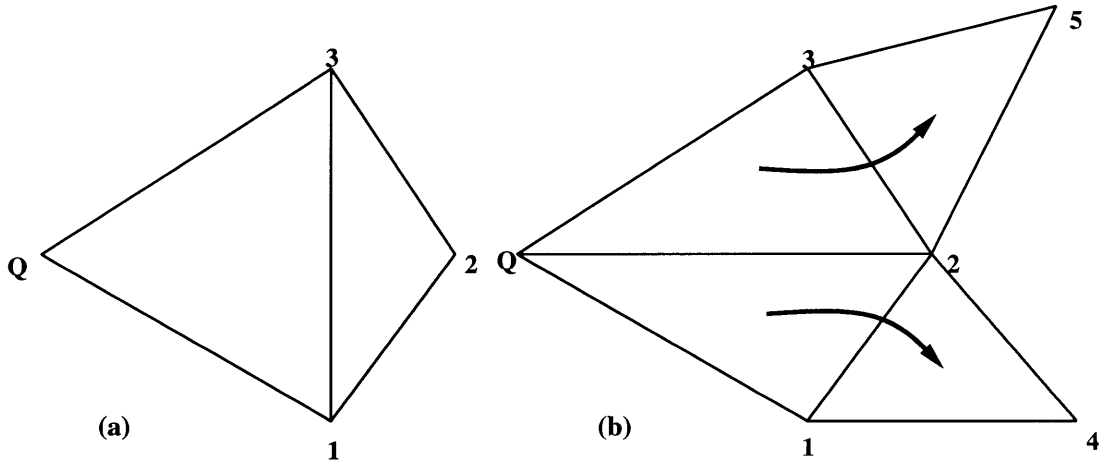


Figure 2.2: Edge Swapping with Forward Propagation

the circumcircle criteria, the newly created edges will be Delaunay. However, some of the original edges have now been rendered invalid and such, all edges which are termed *suspect* must be located. This is done by considering a "suspect" edge as the diagonal of the quadrilateral formed from the two adjacent elements. The circumcircle

test is applied to either of the adjacent elements such that if the fourth point of the quadrilateral is interior to the circumcircle, then the edge is swapped. This creates two more "suspect" edges which need to be tested. The process terminates when all the "suspect" edges pass the circumcircle test. The nature of the Delaunay triangulation guarantees that any edges swapped incident to Q will be final edges of the Delaunay triangulation. This implies that *forward propagation* [55] need be considered as depicted in figure 2.2. This may be done recursively or with the help of a stack data structure. The implementation of the algorithm is listed below.

1. Insert new point Q into existing point set.
2. Locate root element which encloses point.
3. Insert point and connect to surrounding vertices.
4. Identify "suspect" edges.
5. Perform edge swapping on "suspect" edges failing circumcircle test and identify new "suspect" edges.
6. If new "suspect" edges, go to Step 4.

As mentioned before, the problems associated with constrained Delaunay triangulation must also be considered with this method.

2.2.3 Alternative Point Insertion Algorithms

The primary point insertion algorithms implemented are as described above. However, it is sometimes necessary to generate non-Delaunay triangulations based on other criteria.

To this effect, we consider a modification of the Green-Sibson algorithm such that the circumcircle test is replaced by other criteria. Possible options are

1. *Minimization of maximum angle.* This is referred to as a MinMax triangulation and considers the maximum angle for both the unswapped and swapped edge configuration. If the maximum angle for the swapped configuration is less than that of the unswapped configuration, the edge is swapped.

2. *Minimization of skewness.* This is referred to as a MinSkew triangulation which attempts to minimize the skew parameter for an element. The skewness parameter as defined by Marcum [13] is proportional to the element area divided by the circumcenter radius squared in 2D and to the element volume divided by the circumcenter radius cubed in 3D.

2.3 Implementation Issues

A number of issues need to be addressed for the implementation of the mesh generation algorithms outlined above. These are directly related to the problems associated with mesh control, data structure formats and geometric searching.

2.3.1 Mesh Control

Mesh control mechanisms must be included in mesh generation implementations to allow for control over the spatial distribution of points such that a grid of the desired density is produced. This can be accomplished by means of a *background mesh* and a *source distribution.*

Background mesh

A background mesh, implemented as tetrahedral elements in both 2D and 3D, is provided such that desired mesh spacings are specified at the vertices of the elements. The background mesh must cover the entire domain and is defined such that the mesh spacing (or element size distribution) at any point in the domain interior is computed by linear interpolation on the background element which encloses the point. This provides a convenient method of specifying linearly varying or constant mesh spacings over the entire domain.

Source distribution

For complex geometries, specification of a background mesh can lead to a large number of background elements. This can be remedied by specifying sources at specific regions in the computational domain.

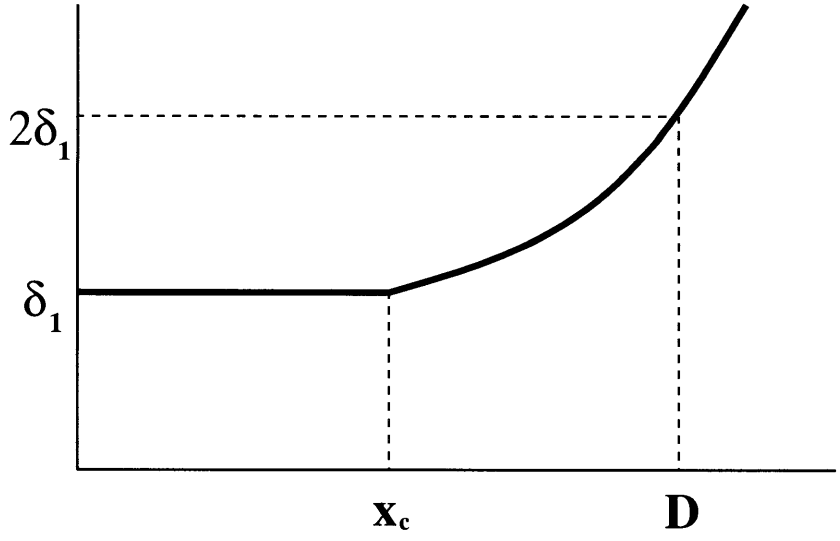


Figure 2.3: Source Element

The mesh spacings for the sources are then defined as an isotropic spatial distribution

which is a function of the distance from a given point to the source. The functional form of the mesh spacing specified by a point source is given by

$$\delta(x) = \begin{cases} \delta_1 & \text{if x} \leq x_c \\ \delta_1 e^{\left|\frac{x-x_c}{D-x_c}\right| \log 2} & \text{if x} \geq x_c \end{cases} \tag{2.4}$$

where the mesh parameters are represented by

1. x_c: Distance over which mesh spacing is constant.
2. δ_1: Constant mesh spacing over distance x_c.
3. D: Distance at which mesh spacing doubles.

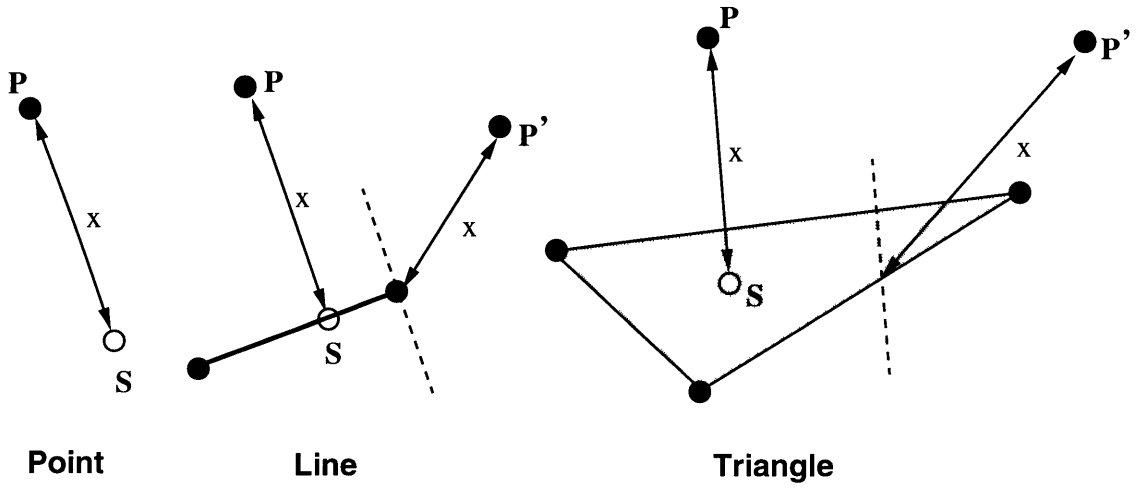


Figure 2.4: Point, Line and Triangle Sources

The sources may take the form of point, line or triangle sources as depicted in figure 2.4. The mesh spacing is defined for line and triangle sources as the spacing based on the closest point on the line segment or triangle This point is chosen as a point source with the mesh parameters x_c, δ_1 and D linearly interpolated from the nodal values on the line or triangle source. Hence for any given point in the computational domain, the spacing is given as the minimum spacing defined by all the source spacings and that specified by the background mesh.

2.3.2 Data Structures

The choice of data structures to be employed in the mesh generation implementation plays an important role due to the nature of the algorithms. Efficient, compact and well structured data types are to be used to ensure fast execution. Some of the data structures as presented in Löhner [46] and Morgan [29] were implemented. The primary data structures involved include

1. **Boundary Information:** The constrained Delaunay triangulation involves a prescribed set of boundary edges. The information regarding a boundary edge is stored in a data structure termed a *front segment* which is to be distinguished from the *front* in regards to the Advancing Front Method for mesh generation e.g [25]. A front segment contains the information regarding the vertex identification indices (VID) of the boundary edge vertices, the attached element and any other marker information regarding the edge.

2. **Element Adjacency:** Element adjacency or neighbor information is also stored and explicitly updated. This involves increased memory usage but greatly reduces computation time. For a given front segment attached to an element, the element adjacency is modified to take this into account.

3. **Dynamic Heaps:** Several point creation algorithms used to determine the coordinate of the next generated point make use of some geometric property (such as the circumcircle radius or element area) of the current elements to generate the point. This usually involves some sort of sorting of the elements to find the seed element from which the point is to be generated. The *Heap Sort* algorithm [11, 50] was chosen due to the $\bigcirc$(log N) efficiency for insertion and deletion operations.

2.3.3 Geometric Searching

The problem of determining the members of a set of n points or elements in R^d which lie inside a prescribed subregion or satisfy some proximity criteria is known as geometric searching. Several algorithms involving $\bigcirc$(log N) operations have been put forward [24, 23, 33] to solve this problem and other equivalent problems. The selected algorithm is the Alternating Digital Tree (**ADT**) algorithm which is an extension of the binary tree search methods. It provides for a fast and efficient method to perform geometric searches as presented in [22]. Examples of the use of ADT include point, edge and face proximity searches and root element location as in the case of the Green-Sibson point insertion algorithm.

2.4 Mesh Generation Process

Before the actual mesh generation can proceed, a preprocessing stage is necessary. Hence, the mesh generation process is divided into two phases with the preprocessor phase separate and taking place independent of the actual mesh generation.

2.4.1 Preprocessing

The mesh generation procedure begins with a geometric description of the computational domain Ω based on CAD/CAGD geometric models. The bounding curves and surfaces of the domain are modeled by creating a geometric description based on Ferguson cubic splines and bicubic patches [18]. The curve segments are discretized based on the mesh spacing specified by the background mesh from which the surfaces are created. The point creation schemes which are to be considered require an initial triangulation of the

computational domain. This is performed as described in [55].

2.4.2 Interior Point Creation

The mesh generation aspect involves the actual determination and creation of mesh vertex coordinates. This is a sequential procedure for the generation of new points in the domain based on some geometric property of the current triangulation and subsequent insertion based on the constraint imposed by the physical boundary edges. Several schemes have been put forward and are currently implemented in the literature. These include such algorithms as the Advancing Front Algorithm [25] and Circumcenter Algorithm [41]. In this work, three different algorithms have been implemented and these are outlined briefly below.

Rebay

This algorithm as proposed by Rebay [53] is a variant of the Advancing Front algorithm which combines aspects of the Advancing Front algorithm with those of the Bowyer point insertion algorithm. The algorithm considers the division of the created elements into two groups which are tagged *accepted* and *unaccepted*. The *accepted* elements consist of those whose circumradii is less than a factor multiple (Rebay Factor R_f) of the desired element size defined as the mesh spacing at the circumcenter of the element. The algorithm proceeds by considering the maximal non-accepted element, defined as the element with the largest circumradius, which is adjacent to an accepted element as shown in figure 2.5.

The Voronoï segment which joins the circumcenters of the two elements is perpendicular to the common face between the two elements. The new point X is then inserted

on the Voronoï segment in the interval between the midpoint M of the common face and the circumcenter C of the non-accepted element. In the 2D context, let p be half the length of the common edge $\overline{PQ}$ and q be the length of $\overline{CM}$. Given the desired circumcircle radius f_M and defining

$$R = \min\left[\max(f_M, p), \frac{p^2 + q^2}{2q}\right] \tag{2.5}$$

Rebay's algorithm inserts the new point X on the interval between M and C at a distance of

$$d = R + (R^2 - p^2)^{\frac{1}{2}} \tag{2.6}$$

It is proven by Baker [54] that in 2D, the elements in the interior tend to equilateral triangles with possible distortions on the boundary.

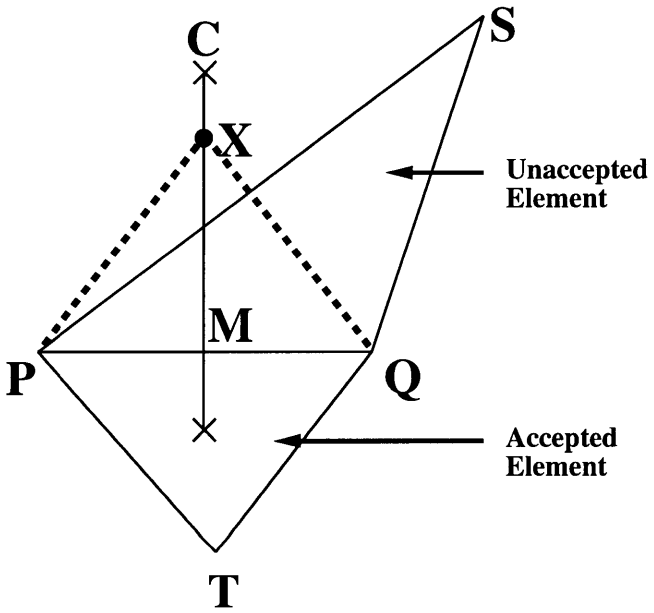


Figure 2.5: Rebay Point Insertion Algorithm

Circumcenter Algorithm

This is a relatively fast and inexpensive algorithm which has been reported by Chew [42] and Ruppert [26]. The algorithm considers the maximal element defined as above and

generates the new point X at the circumcenter of the element. Chew proved that the elements generated by this algorithm have a minimum bound of 30° except for boundary effects.

Centroid Algorithm

This is another relatively fast and inexpensive algorithm reported by Weatherhill [41] which also considers the maximal element defined as above and generates the new point at the centroid of the element.

All the above considered point generation algorithms allow for a constrained triangulation with respect to the domain boundary edges. A common feature between them is that they consider the maximal element. This implies that the elements need to be sorted. This is the reason why a heap sort algorithm was implemented with supporting dynamic heap data structure. The mesh generation process simply consists of point creation and insertion until there are no more elements in the dynamic heap to consider. The facilitation of the entire mesh generation process may be done by having each point creation algorithm define a specified set of functions which will be invoked during execution such that any of the given algorithms may be chosen at will. This is implemented by currently defining a data structure of functions as in Appendix A. This function set is sufficient within the serial mesh generation context to provide enough functionality for any of the point creation algorithms.

Mesh generation systems for unstructured grid must be able to deal with the above mentioned issues. One issue which has not been mentioned is support for meshes generated on *non-manifold* models. In a non-manifold representation, the surface area around a given point on a surface might not be flat in the sense that the neighborhood

of the point need not be a simple two dimensional disk. An example of such a mesh is shown in figure 2.6.

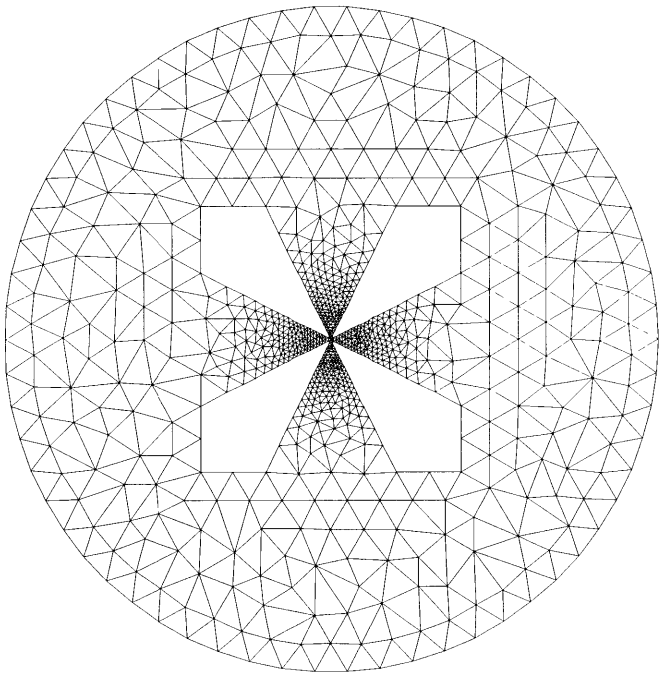


Figure 2.6: Non-manifold mesh example

Chapter 3

Anisotropic Grid Generation Extension

This chapter discusses the extension of the presented isotropic mesh generation algorithms to anisotropic meshes. Such meshes are required for many applications such as the computation of viscous flows which exhibit anisotropic gradients. The quality of the mesh in the regions of sharp gradients has to be such that the solution features are resolved. This may be done by increased refinement in the desired regions in an isotropic manner but this leads to a large number of elements. An alternative is to consider stretched triangulations in these regions as shown in figure 3.1 used for viscous flow calculation about a flat plate.

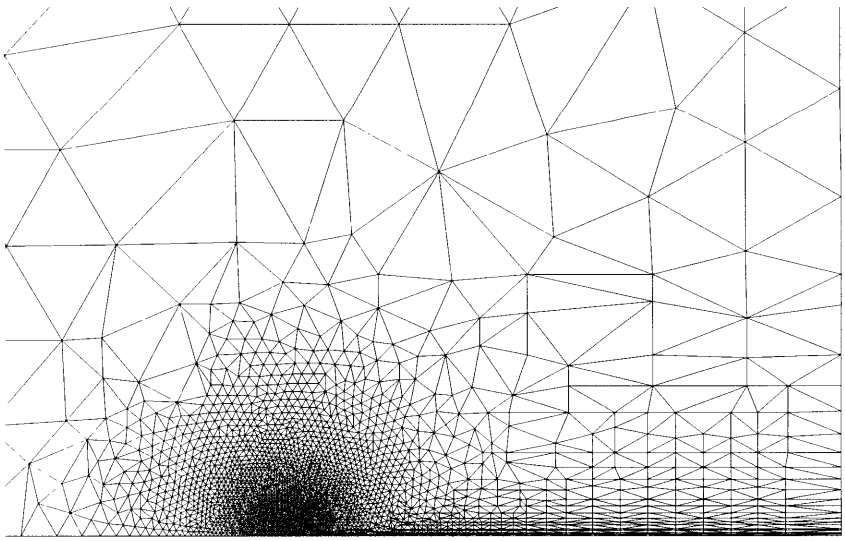


Figure 3.1: Viscous Flat Plate Mesh

3.1 Anisotropic Refinement

Anisotropic grids are produced by refining an existing initially isotropic grid [55]. This is based on the *Steiner* triangulation which is defined as any triangulation that adds additional points to an existing triangulation to improve some measure of grid quality. The stretching or refinement criteria is based on the minimum Euclidean distance S from the interior vertices of the elements to the boundary segments which have been marked for refinement as depicted in figure 3.2.

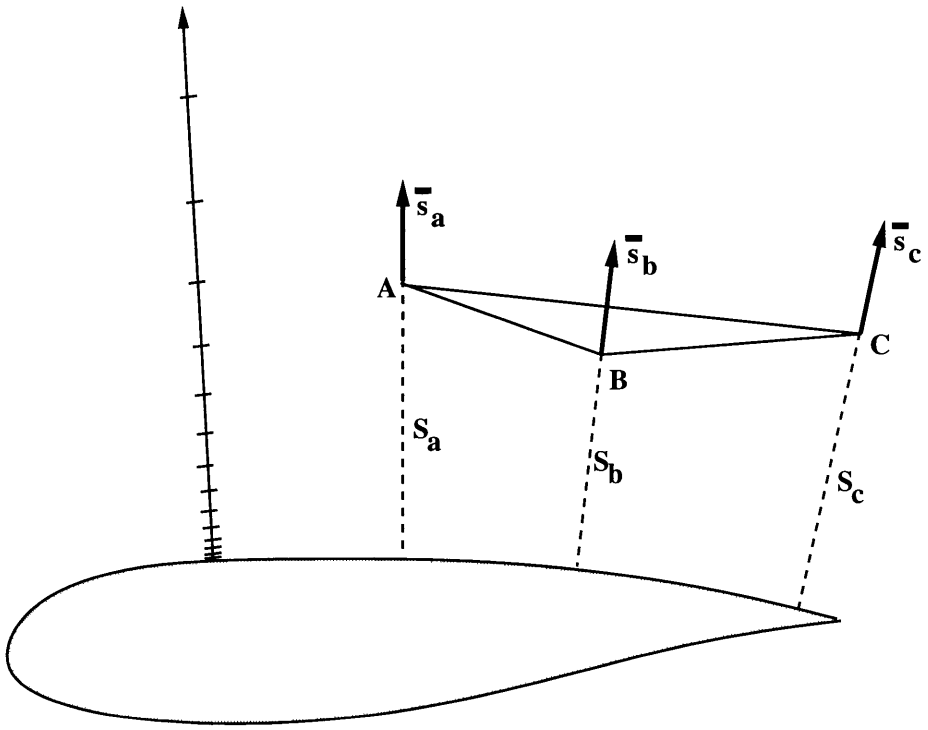


Figure 3.2: Anisotropic Refinement Parameters

For each boundary curve which has been marked for refinement, two parameters δ_0 and r_g which represent the minimum spacing and the geometric growth ratio are specified. A dynamic heap structure is maintained which sorts the "active" elements according to the minimum Euclidean distance of their vertices. An active element is

defined as one which has at least one vertex from which a new point can be generated. For each vertex in the triangulation, a unit vector $\hat{s}$ from the closest point on the closest boundary segment to the vertex is computed. The element measure chosen which determines if an element is active or not is based on the maximum dimension of the element projected along the vector $\hat{s}$ at the vertices of the element. For each vertex on an element, consider the element edges attached to that vertex. If the projection of any of the edge vectors along the unit vector $\hat{s}$ associated with that vertex is larger than the specified spacing at the vertex, then the element is classified as active.

The candidate Steiner points associated with an active element are obtained at the vertices which correspond to the maximum and minimum Euclidean distances S_{max} and S_{min} for the element. At the maximal vertex, a new point X_{max} is inserted at a distance d_{max} along the unit normal $\hat{s}_{max}$ associated with the vertex and away from the closest boundary segment point. At the minimal vertex, a new point X_{min} is inserted at a distance d_{min} along the unit normal $\hat{s}_{min}$ associated with the vertex and towards from the closest boundary segment point. This is as depicted in figure 3.3. The generation distances are chosen such that positions of the new points coincide with the points at which a point would have been created based on the geometric growth associated with the boundary segment such that the new points are created in uniform layers over the boundary segment. The procedure follows the following form

1. Given the current active element, the distances S_{min} and S_{max} for the element and δ_0 and r_g for the closest boundary segment are obtained.

2. Generate distances based on the geometric growth sequence

$$g_n = g_{n-1} + \delta_0 {r_g}^{n-1} \tag{3.1}$$

until

$$g_{\bar{n}} - S_{min} \geq \mathrm{G_f}\delta_0 r_g^{\bar{n}-1} \tag{3.2}$$

where G_f represents a growth limiter factor usually taken to be 0.5.

3. Generate set of N distances d_i based on geometric growth starting from $g_{\bar{n}}$ until the condition below is satisfied.

$$S_{max} - d_i \leq \mathrm{G_f}\delta_0 r_g^{i+\bar{n}-1} \tag{3.3}$$

The distances d_{max} and d_{min} are computed from these N generated distances using

$$d_{min} = d_1 \tag{3.4}$$

$$d_{max} = S_{max} - S_{min} - d_N \tag{3.5}$$

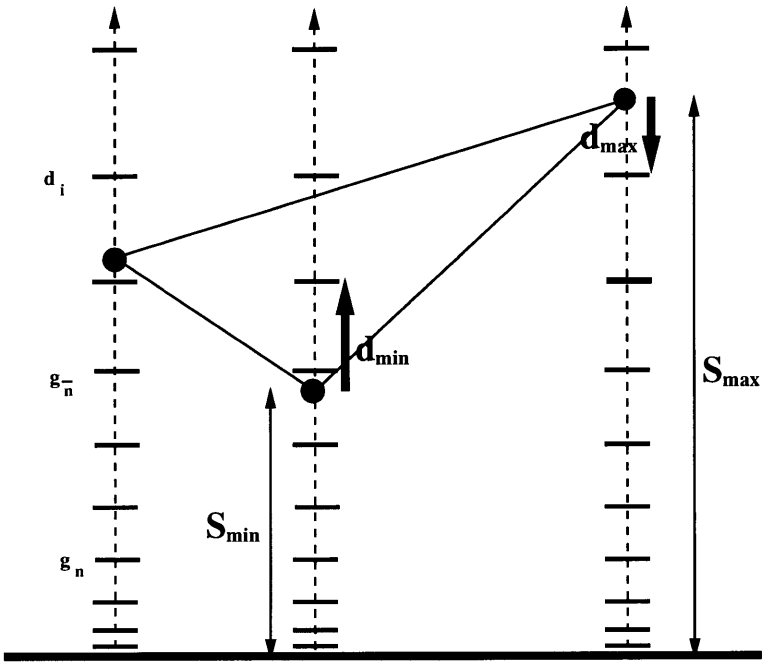


Figure 3.3: Anisotropic Point Creation

The candidate Steiner points are inserted into the triangulation only if there exists

no other point in the triangulation which lies at a distance closer than the generation distance (d_{max} or d_{min}) associated with that point. The proximity query is performed by making use of the ADT structure of the current element set. The refinement is guaranteed to converge since a point is reached for all the elements such that either every element is refined or the geometric growth thickness at any point is greater than the mesh spacing at that point. Figures 3.4 and 3.5 show a mesh generated about a 5 element Boeing 737 airfoil by this strategy in which the minimum wall spacing was specified at 0.001% of the chord length.

3.2 Wake Path Generation

In viscous flow computations about airfoils, the wake path which is created by the airfoil is a region of interest. To properly resolve this region, the above technique may also be applied to create a stretched triangulation of the wake path.

To perform this, an initial guess must be made for the shape of the path. It should be noted that the procedure described here takes place in the preprocessing stage of the mesh generation. The wake path is initially modeled by making use of Ferguson cubic splines and discretized based on the background mesh. The discrete points on the wake path are inserted into the boundary triangulation. After insertion, the wake segments are treated as physical boundaries for subsequent constrained triangulation. After the isotropic mesh is generated, the Steiner triangulation proceeds as described above.

Figures 3.6 and 3.7 show the effect of wake path generation. The figures show a generated mesh about a NACA-0012 airfoil with both wake path generation and boundary refinement.

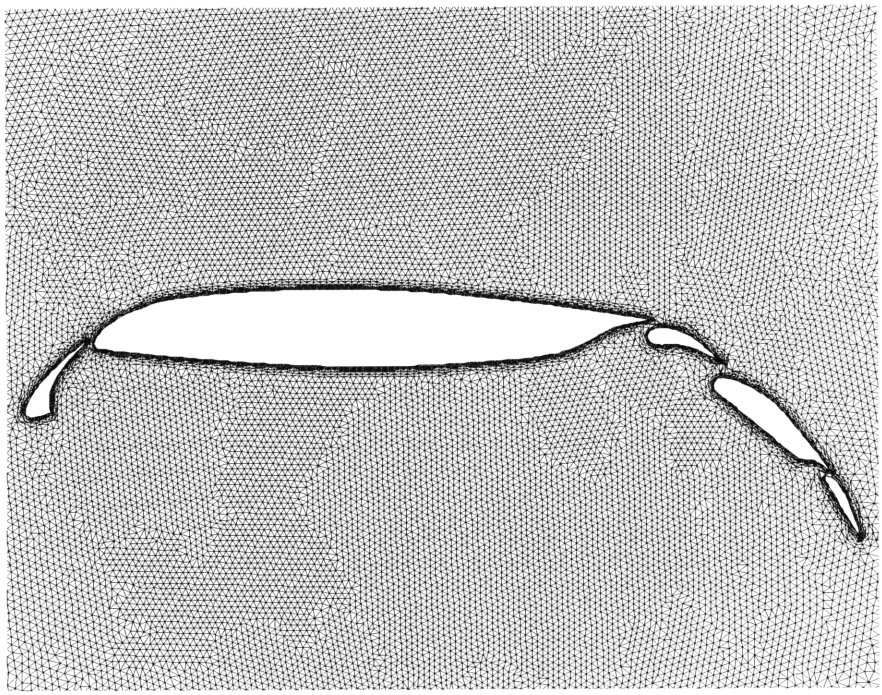


Figure 3.4: Viscous Mesh Generation on 737 Airfoil

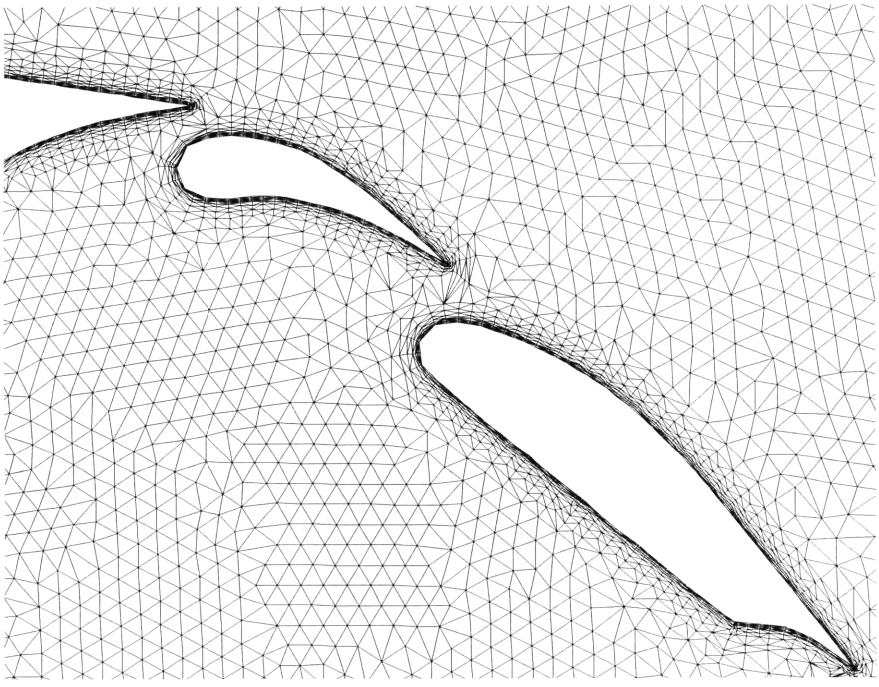


Figure 3.5: Trailing Edge Closeup of 737 Airfoil

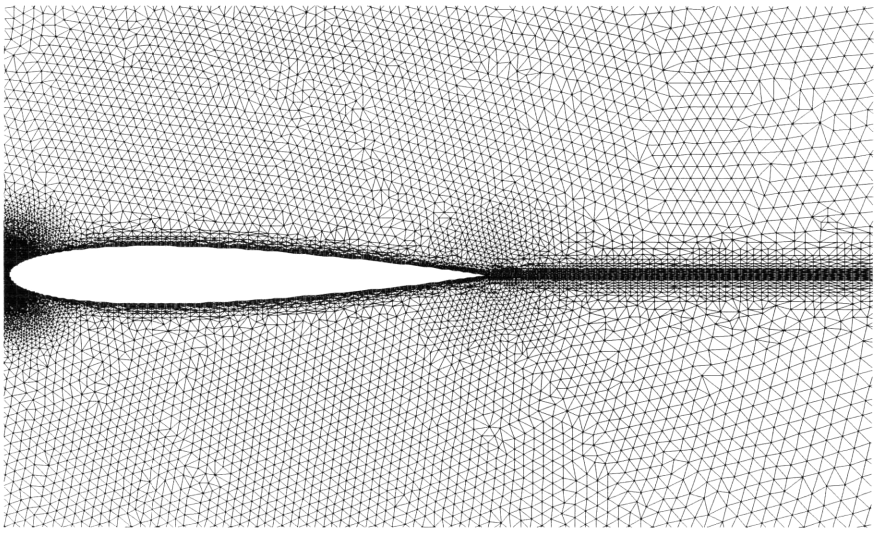


Figure 3.6: Wake Path Generation on NACA-0012 Airfoil

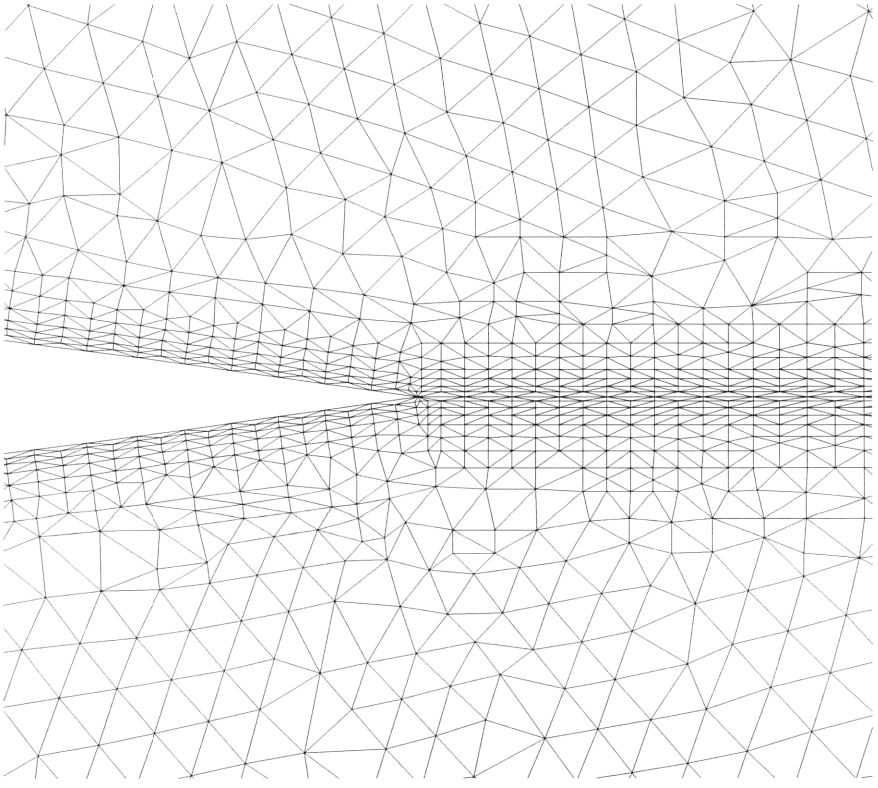


Figure 3.7: Trailing Edge Closeup of NACA-0012 Airfoil

Chapter 4

Parallel Systems and Model Overview

4.1 Parallel Systems

The last few years have witnessed a marked shift in the scale and complexity of problems which have been simulated numerically. This is due in part to the increased technology which has enabled significant advances in both computer software and hardware. However, despite these improvements, numerous problems exist today which cannot still be solved with conventional uniprocessor computers. This is where supercomputers and parallel machines come into play and provide a chance at actually solving some of these problems.

The architectural classification of modern parallel systems falls into two major classes based on instruction execution and data stream handling.

4.1.1 SIMD architecture

The SIMD acronym stands for Single Instruction, Multiple Data and describes systems which are composed of a large number of (simple) processing units, ranging from 1K up to 64K, such that all may execute the same instruction on different data in lock step. Hence, a single instruction manipulates several data items in parallel. Examples of such systems include the MasPar MP-2 and the CPP DAP Gamma. It should be noted that

the SIMD architecture was once quite successful and is quickly disappearing. This is due in part to applications relevant to this type of architecture such as image processing, which is characterized by highly structured data sets and data access patterns.

Another subclass of SIMD systems are *Vector processors* and these incorporate special hardware or vector units which perform operations on arrays of similar data in a pipelined manner. These vector units can deliver results with a rate of one, two - and in special cases - three per internal clock cycle. Hence, from the programmer's point of view, vector processors operate on data in an SIMD fashion when executing in vector mode. Some examples of these systems in use include the Cray Y-MP, C90, J916 and T90 series and the Convex C-series.

4.1.2 MIMD architecture

The MIMD acronym stands for Multiple Instruction, Multiple Data and describes systems composed of (relatively) few number of processors executing different instructions independently, each on an independent localized data set. In general, the incorporated hardware and software for these systems are highly optimized so that the processors can cooperate efficiently. Examples of such systems include the Cray T3D and the IBM SP2. The majority of modern parallel systems fall under this class. The full specifications and descriptions of current and some past systems may be found in [57]. Parallel systems, in particular the MIMD based systems, may also be classified based on the organization of memory.

4.1.3 Shared Memory Systems

Shared memory systems are characterized by all the processors having access to a common global memory. Hence it is possible to have all the processors operating on the same common array, though care has to be taken to prevent accidental overwrites. As discussed in [57], the major architectural problem is that of connection of the processors to global memory (or memory modules) and to each other. As more processors are added, the collective bandwidth to the memory should ideally scale linearly with the number of processors, N_p. Unfortunately, full interconnectivity is expensive, requiring $\bigcirc(N_p{}^2)$ connections. Hence a number of creative alternative networks have been developed as shown in figure 4.1.

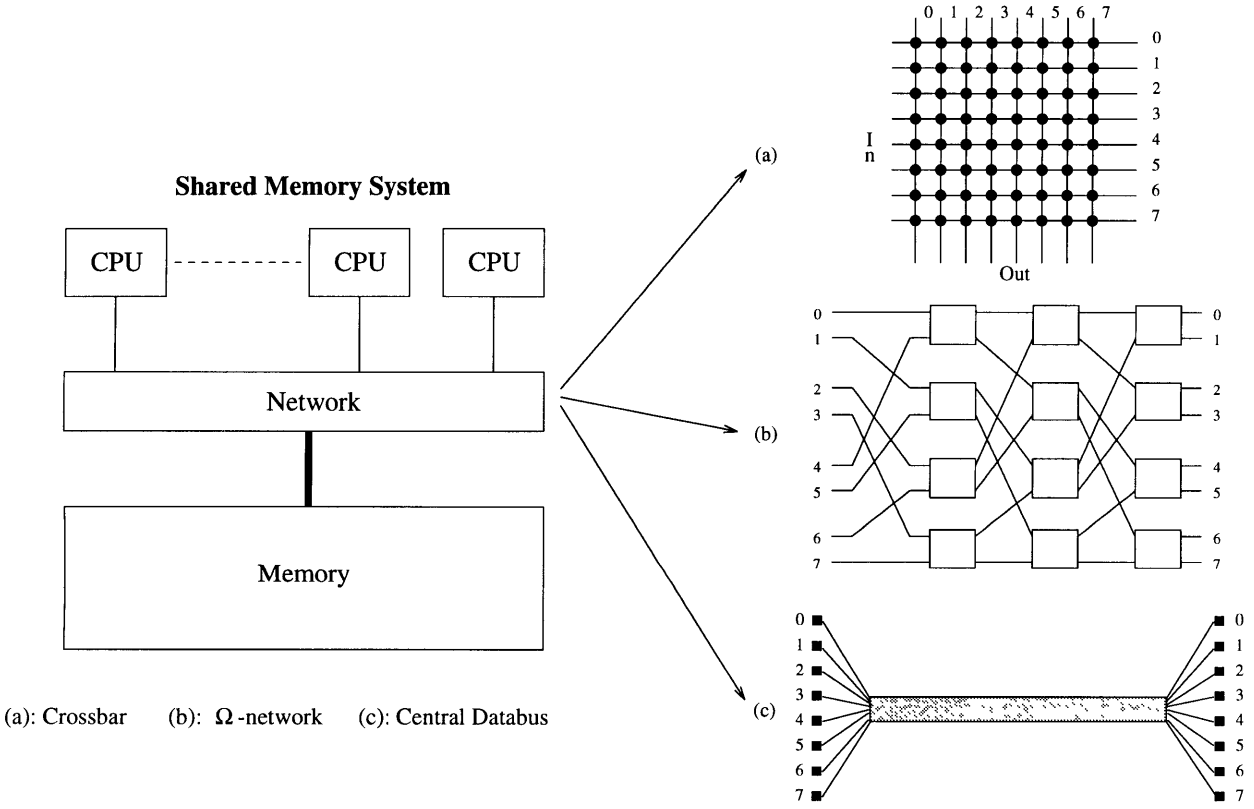


Figure 4.1: Possible connectivities for Shared Memory MIMD systems

The crossbar network makes use of exactly $N_p{}^2$ connections and Ω network makes use of $N_p \log_2 N_p$ connections, while a central bus represents only one connection. Due

to the limited capacity of the interconnection network, share memory parallel computers are not very scalable to a large number of processors. The Ω network topology is made use of in such commercial systems as the IBM SP2.

4.1.4 Distributed Memory Systems

Distributed memory MIMD system consist of a processor set $\{P_i \mid i = 1..N_p\}$ each with its own local memory interconnected by a communication network. The combination of a processor and its local memory is often defined as a *processor node* such that each processor node is actually an independent machine. The communication between the processor nodes may only be performed by message passing over the communication network. The communication network for distributed memory systems is also of importance. Full interconnectivity is not a feasible option, hence the processor nodes are arranged in some interconnection topology.

Hypercube Topology

The hypercube topology is a popular choice and is incorporated in such systems as the nCUBE series. The hypercube topology as shown in figure 4.2

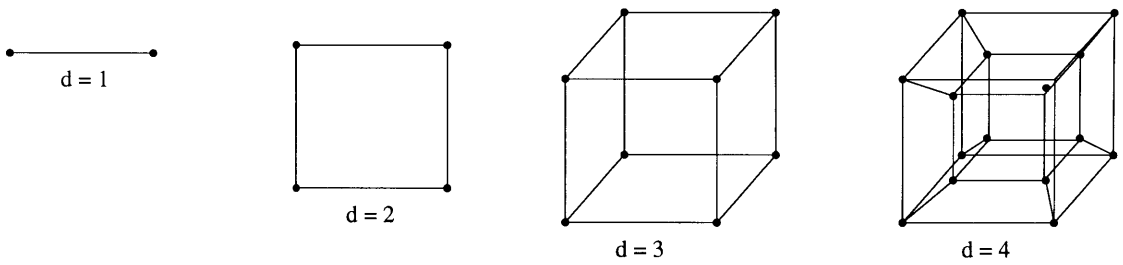


Figure 4.2: 1D, 2D, 3D and 4D hypercube connectivity

has the nice feature such that for $N_p = 2^d$ processor nodes, the maximum number

of links between any two nodes (network diameter) is d. Hence, the diameter grows logarithmically with the number of nodes. Also, several other topologies such as rings, trees, 2-D and 3-D meshes may be mapped unto the hypercube topology since these are all subsets of the hypercube topology.

Torus Topology

The *torus* or mesh topology is implemented on such systems as the Cray T3D and the 3D analog is shown in figure 4.3

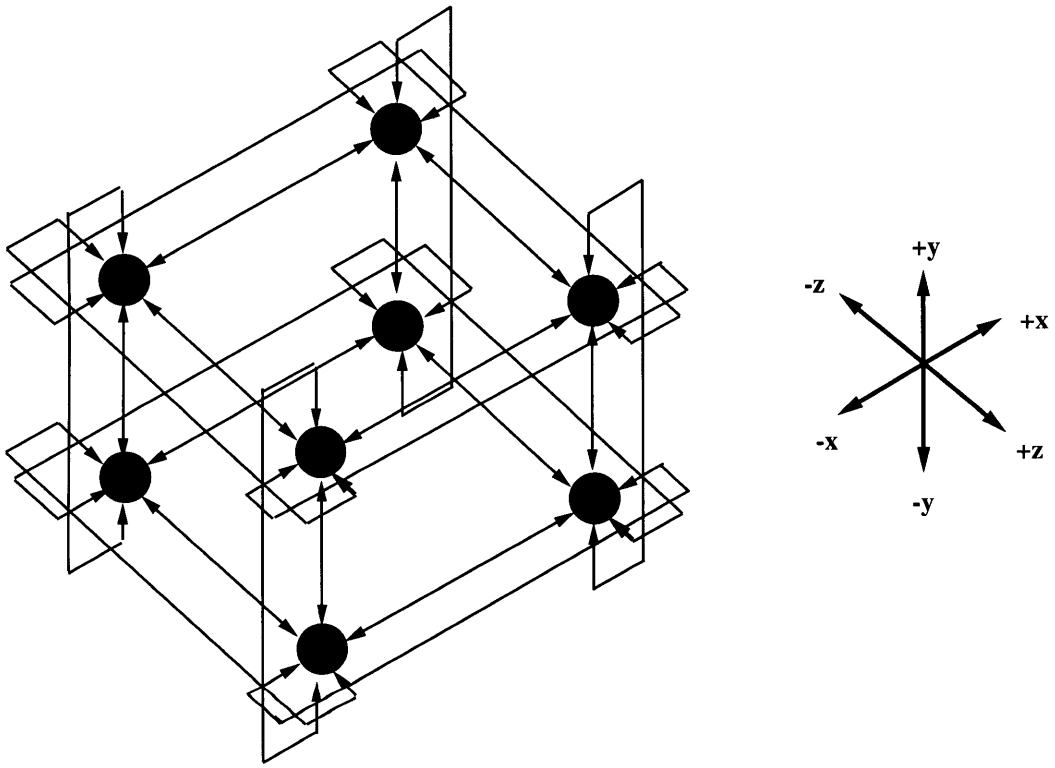


Figure 4.3: 3D torus connectivity

The torus topology for a given dimension essentially consists of a periodic processor array which exhibits "wrapping". The rationale behind this is that most large-scale physical simulations can be mapped effectively on this topology and that a richer interconnection structure hardly pays off.

Multi-Stage Topology

Multi-stage networks such as the Ω network shown in figure 4.1 are characterized by small numbers of processor nodes connected in clusters with each cluster connected to other clusters at several levels. It is thus possible to connect a large number of processor nodes through only a few switching stages. Multi-stage networks have the advantage that the *bisection bandwidth* can scale linearly with the number of processors while maintaining a fixed number of communication links per processor. The *bisection bandwidth* of a distributed memory system is defined as the bandwidth available on all communication links that connect one half of the system ($\frac{N_P}{2}$ processors) with the second half.

The major advantage of distributed memory systems over shared memory systems is that the architecture suffers less from the scalability problem associated with the connectivity bandwidth. However, the major disadvantage is that the communication overhead incurred in message passing is significantly higher. Another problem may occur if the system is not heterogeneous (i.e the individual CPUs are not identical) such that mismatch in communication and computation speeds may occur.

Due to the clear overall advantages of the distributed memory model and the MIMD architecture, this was the choice for the programming model for the implementation of the parallel grid generation system. A general comparison based on performance of several benchmarking tests on several distributed memory MIMD systems is given by Bokhari [52].

4.2 Distributed memory programming model

The distributed memory programming or message passing model is characterized by each processor node having its own local memory. Hence data which resides off-processor can only be accessed by having the processor on which the data resides send the data across the network. The implemented programming model is SPMD (Single Program, Multiple Data) in which one program is executed across all the processors executing on different parts of the same data set. This implies that the data must be distributed across the processor set. This partitioning is the basis of the parallel grid definition.

Implementation of message passing systems implies the existence of a communication library. The basic operation of the communication routines is data passing between arbitrary processor nodes and possibly the ability to check for the existence of messages in the message buffer. Higher level operations involve global operations such as broadcasting in which one processor node sends the same message to all other nodes, global sum, global minimum and global maximum of distributed data, and also global synchronization between a subset or possibly all of the nodes. In general, most distributed MIMD systems come with a native communication library usually written in C or FORTRAN such as on the IBM SP2 (MPL) or the nCUBE (NCUBE). A number of commercial efforts have been made to develop machine independent communication libraries and these include PVM [59], MPI [14], PARMACS [45] and CHARM [31].

The program structure is based on a message driven slave/master paradigm which allows for a highly concurrent implementation of the procedures involved. Other than the two operations of dynamic load balancing and work allocation, the slave processors are relatively independent of the master processor. Hence this model does not present a bottleneck. Based on the design philosophy of portability to distributed memory

MIMD systems, a communications library to provide a common interface for message passing has been developed for several major systems and is still undergoing additions. This library contains a base set of standard parallel communications routines which are outlined in Appendix B. This set of routines provide a minimal base of functions which enable a standard interface to parallel communication. Due to the peculiar nature of the distributed system, the parallel send and receive routines are *blocking* as opposed to *non-blocking*. A blocking operation does not return control to the processor until the message has been fully sent or received. However, options exist for specifying a non-blocking mode. Currently, this library has been implemented for the nCUBE and IBM SP2 native communication libraries as well as for the PVM and MPI message passing libraries.

Routines for operations which need to take place in a global fashion are also provided. These routines include

- Element location which involves the location of the element and subdomain tuple which encloses a point or satisfies some sort of geometric criterion.
- Point location which involves the location of the point and subdomain tuple which satisfies some sort of geometric criterion, possibly with respect to other points.
- Global identifier tag creation for elements and points.

Chapter 5

Parallel Automatic Mesh Generation

5.1 Introduction

This section introduces the parallel implementation of the mesh generation algorithms described previously. In general, efficient parallel algorithms require a balance of work between the processors while maintaining interprocessor communication to a minimum. A major key to determining and distributing the work load is based on the knowledge of the nature of the type of analysis being performed. Parallel adaptive finite element analysis [9, 7, 35] is thus significantly affected due to imbalance in the work load after adaptivity unless load balancing is performed. Parallel mesh generation is much more difficult to control since the only knowledge available at the beginning of the process is the initial structure of the geometric model which has little or no relationship to the work required to generate the mesh. This lack of ability to predict the work load during the meshing process leads to the selection of the parallel mesh generation implementation.

The parallel mesh generation algorithms are an extension of the serial algorithms which are executed when operating in parallel mode. The 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 a better distribution of the work load.

5.2 Previous Efforts

The current literature on aspects of mesh generation in parallel is sparse and this indicates that this is a relatively new field which has not been explored. Early attempts at parallel mesh generation include Weatherhill [39], Löhner *et al* [47] and Saxena *et al* [34]. Weatherhill implemented what is essentially a "stitching" technique whereby the subdomains are meshed individually on separate processors and cosmetic surgery is performed on the processor interfaces. Löhner *et al* [47] parallelized a 2-D advancing front procedure which starts from a pre-triangulated boundary. The approach is also similar to Weatherhill [39] in that the domain is partitioned among the processors and the interior of the subdomains is meshed independently. The inter-subdomain regions are then meshed using a coloring technique to avoid conflicts. Saxena *et al* [34] implement a parallel Recursive Spatial Decomposition (RSD) scheme which discretizes the computational model into a set of octree cells. Interior and boundary cells are meshed by either using templates or element extraction (removal) schemes in parallel such that the octant level meshes require no communication between the octants.

Software systems which include compiler primitives such as on the Thinking Machines CM2 and runtime systems such as the PARTI primitives [27] were earlier designed for generating communication primitives for mesh references. Hence opportunities for parallelization are recognized in the code and automatically created.

Distributed software systems have also been developed which implement the management of distributed meshes in such operations as load balancing and mesh adaptivity. These include the Distributed Irregular Mesh Environment (DIME) project by Williams [51] and the **Tiling** system by Devine [28]. However to date, a more general attempt at parallel unstructured mesh generation has been made by Shephard *et al* [10, 36]. Shep-

hard *et al* discuss the development of a parallel three-dimensional mesh generator [10] which is later coupled to an adaptive refinement system and a parallel mesh database based on the serial SCOREC mesh database into the distributed mesh environment, Parallel Mesh Database (PMDB) [36]. PMDB hence incorporates mesh generation, adaptive refinement, element migration and dynamic load-balancing which are four of the defining characteristics of any fully operational distributed mesh environment. The parallel mesh generator described by Shephard *et al* is an octree-based mesh generator [37, 61, 38] in which a variable level octree is used to bound the computational domain from which mesh generation takes place. Full data spectrum for all geometric entities are stored in this implementation and hence may be very expensive in terms of computational resources.

The development of a functional 2-D parallel unstructured mesh system is now discussed. The design philosophy of the system is based on

1. Independent interior mesh generation with in-process interprocessor communication for processor boundary exchange.
2. Portability to distributed memory MIMD parallel system.
3. Scalability to any number of processors. Speedup considerations are however not as important as ability to generate grids more massive previously attempted.
4. Minimal data structure to enable maximal memory capacity but with ability to create necessary data as needed.
5. Provision of a dynamic load balancing routine to ensure even distribution of work.
6. Ability to generate massive meshes of order 10 million elements.

5.3 Requirements

5.3.1 Data Structure

The distributed mesh needs to be defined for implementation of parallel mesh generation. Implicit to this discussion is the assumption that subdomains bear a one-to-one mapping to processors such that their usage is interchangeable. For the element set $\{\Omega_{i,j}\ \{\{j = 1..N_e{}^i\}, i = 1..N_p\}\}$ where $N_e{}^i$ represents the number of local elements in subdomain i and N_p is the number of subdomains, the mesh is partitioned across processors such that each element belongs to a single processor. Interprocessor information is dealt with by having edges common to two adjacent subdomains shared (duplicated). This is not considered wasteful since for typical meshes considered, the surface area to volume ratio are usually quite small. The mesh generation procedures operate on this model of distributed grid representation.

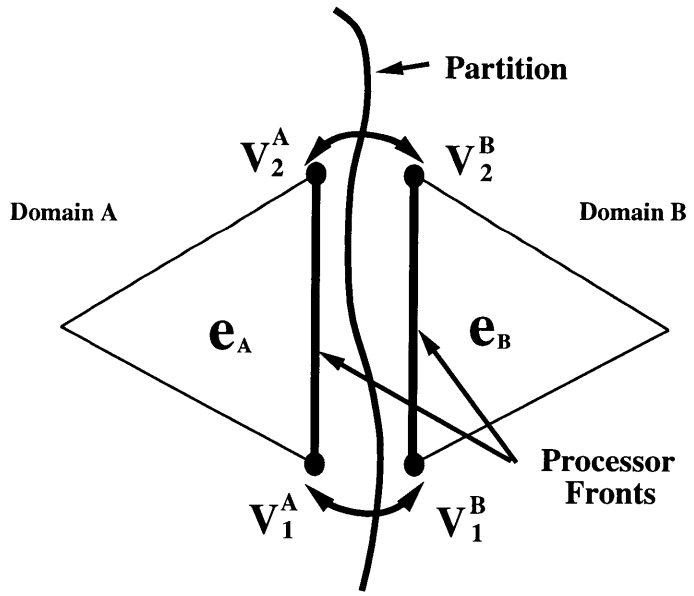


Figure 5.1: Interprocessor Front Segment

The concept of edge and point sharing is embodied in the *processor front segment*

data structure which is defined for each shared interprocessor edge and duplicated on each sharing subdomain. This is depicted in the 2-D context in figure 5.1. For a given interprocessor interface shared by two processors as shown in figure 5.1, the associated processor front segment on each sharing subdomain contains information regarding

1. The remote processor *Unique Identifier* (UID).
2. The remote identifier of the remote processor front segment.
3. The local vertex identifiers of the local front segment vertices, and the corresponding remote vertex identifiers of the remote front segment vertices.

This is incorporated into a modified *front* boundary structure described in chapter 2. From the interprocessor front segments, a list of all neighbouring subdomains is extracted such that local and global subdomain graphs may be created.

5.4 Parallel Grid Generation

5.4.1 Element Migration

Element migration is an essential aspect of parallel grid generation as it is provides the mechanism for the load balancing and element request procedures to be discussed. Element migration consists of the transfer of all information regarding a subset of elements (Ω^m) between arbitrary processors such that the global mesh properties are still valid. As depicted in figure 5.2, the element migration philosophy involves the transfer of boundary, topological and geometric information about the elements to be transferred.

The process of element migration from one processor node to another node is carried

out in three stages. The first stage involves verification that the elements to be migrated can actually be transferred to the destination processor. This requires that a *front lock* must be made on all other processors (excluding the sender and receiver) on all the front segments which will be affected by the migration. The second stage involves the transfer of mesh entities between the sender and receiver. The third stage involves the shared information update of front segments attached to the migrated elements on affected processors which share these front segments with the receiver. In this last stage, the receiver processor also issues appropriate messages to unlock the locked front segments on the remote processors which were originally locked by the sender. The migration procedure is outlined in Appendix C and explained in detail below.

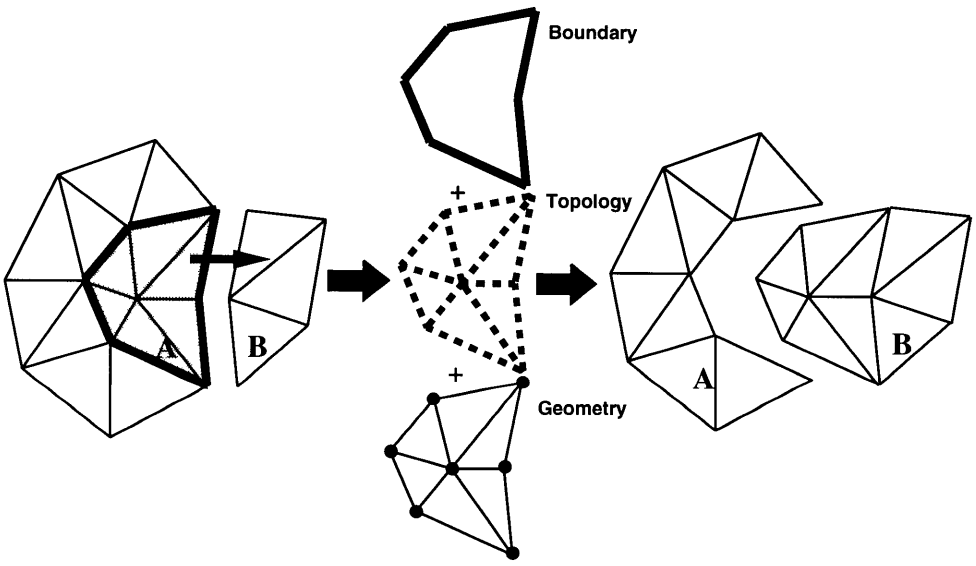


Figure 5.2: Element Migration Philosophy

1. Lock Verification

This stage is actually a preliminary stage which is carried out independently of the actual mesh migration. The reason for lock validation is due to the fact that each processor has a local copy of all front segments shared by other processors. A situation which involves a simultaneous transfer of adjacent elements in different subdomains cannot guarantee the proper update of the shared information. Front locks prevent such occurrences and if the proper locks cannot be made, the element migration routine cannot be invoked.

2. Data Transfer

This stage is involved in the actual transfer of mesh entity information between the sender and receiver processors. The basic idea behind this is to pack the raw element data into message packets which can be easily decoded and unpacked on the receiver end. This involves packing vertex identifiers, neighbor adjacency, vertex coordinate and front information about the transferred mesh entities.

The first step is to create a hash table to be able to quickly determine if a given element is in Ω^m. This will be used in the process of packing the neighbor information. Next, the vertex ids of the element vertices in Ω^m which are shared by the receiver are assigned the remote vertex ids. For a given vertex, this is done by checking the processor front segments attached to the elements in Ω^m for the remote processor UID. If the UID matches the receiver UID, the vertices of all the elements to be transferred which share this vertex are assigned the remote vertex id associated with this vertex. This is performed by a fast BFS algorithm. In the case of a vertex V as exemplified in figure 5.3, care must be taken. Assuming elements e_1 and e_2 are to be migrated from subdomains A to C in the given context, vertex V poses a problem since the

proper information regarding the remote vertex id of the vertex can only be obtained from the processor front segments attached to element e_3. Hence, in the case of any boundary vertex (where boundary is defined in both the sense of front segments and elements which currently neighbor Ω^m and are not to be migrated), a quick $\bigcirc$(log N) ADT tree search is performed on the list of processor front segments. This justifies the maintainance of a list of all the boundary vertices of Ω^m as well as the assigned vertex ids as this stage progresses.

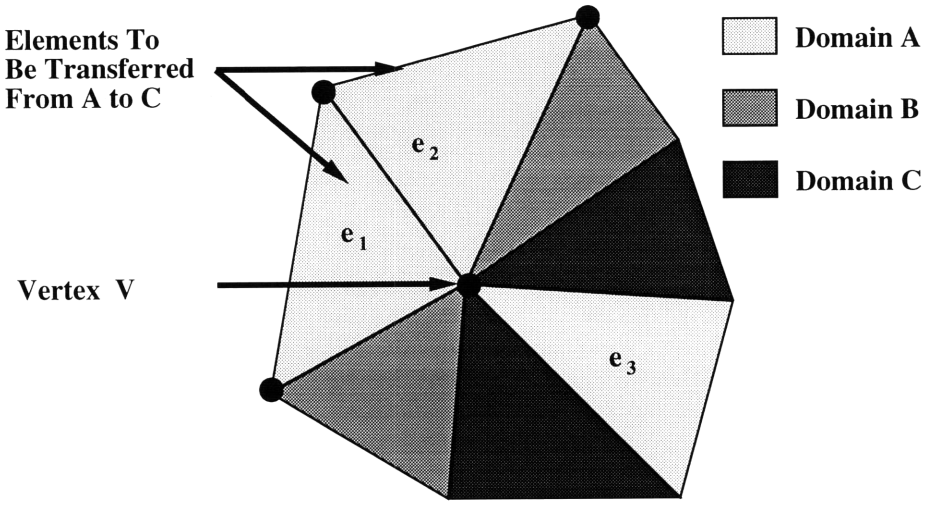


Figure 5.3: Element Migration Example

The element set Ω^m may be fragmented in the sense that there may exist at least one element of Ω^m which does not have a neighbor in Ω^m. In order to remedy this situation, the elements are packed by making use of a hybrid BFS/Greedy algorithm which picks the first element in Ω^m which has not been packed as a seed element to generate a subset of contiguous elements ${\Omega_C}^m$.

The final step is performed to extract and pack the neighbor adjacency information of the elements in ${\Omega_C}^m$. A quick check of the neighboring elements is performed for all elements $\{e_i\}$ in ${\Omega_C}^m$. For any given element in Ω^m, three cases may arise for any of

the elements $\{e_n\}$ adjacent to $\{e_i\}$ which are

1. **Neighbor is in receiver domain:**

   This implies a processor front segment is attached to the element. The element neighbor is set to the remote front segment id in the receiver domain.

2. **Neighbor is neither in sender nor receiver domain:**

   This also implies a processor front segment is attached to the element. A valid front segment is created and packed.

3. **Neighbor is in sender domain:**

   Two cases arise here which can be easily determined by means of the afore mentioned hash table.

   (a) $e_n \in \Omega^m$:

   The element neighbor is simply set to e_n.

   (b) $e_n \notin \Omega^m$:

   A valid front segment is created regarding this newly created interface since this constitutes a processor boundary after the migration is complete.

The packed data is then sent to the receiver and since no further reference will be made to the migrated mesh entities, these are immediately deleted by the sender. The sender then receives the completed front information from the receiver about any newly created front segment between the sender and the receiver.

The receiver receives, unpacks and decodes the raw data regarding the migrated mesh elements. The vertex coordinates are first unpacked and in the case of a non-manifold configuration of the migrated elements, a check must be made to ensure that the coordinates are not duplicated on the boundary. This is performed by an $\bigcirc$(log

N) ADT tree search on the list of processor front segments. Even though this is the currently implemented procedure, future optimizations will include a faster and less expensive operation for determining this condition. Any duplicated point is simply assigned the vertex id of the matching point while the non-duplicated ones are added on to the coordinate list. The elements are then unpacked with the vertex ids set to the proper local ids and then added on to the element list. Finally the front segments are unpacked with the information contained in them set to the proper local and remote values. The last decoding to be performed is the adjacency data on the elements.

3. Remote Update

To complete the procedure, an update needs to be made on all subdomains which contain processor front segments affected by the migration. The necessary information regarding data such as the local vertex ids, remote processor and remote front segment ids are packed and sent to the affected processors.

5.4.2 Load Balancing

The dynamically evolving nature of the unstructured mesh during the mesh generation process creates a load imbalance among the processors such that it is critical to maintain some level of balance in the work load. This is done by means of *partitioning* and *dynamic redistribution* of the elements. The current methods to achieving this for unstructured meshes generally fall into three main categories.

(1) *Recursive Bisection* (RB): This class of techniques involves recursive subdivision of the mesh into two submeshes. This implies that the number of subdomains must be an integral power of 2 i.e $N_p = 2^z$. *Coordinate RB* techniques are based on the

bisection of the element set based on some property of the spatial coordinates. To preserve good surface-to-volume ratios (low communication overhead), a cyclic change of coordinates is usually employed. If the axis of bisection is Cartesian, then this is called *Orthogonal RB* [20, 15, 60]. If the axes are chosen to be along the principal axis of the moment of inertia matrix, then this is called *Inertial or Moment RB*. *Spectral RB* techniques exploits the properties of the *Laplacian Matrix* $\mathcal{L}$ of the mesh connectivity graph and bisects the mesh according to the eigenvector corresponding to the smallest non-zero eigenvalue of the Laplacian $\mathcal{L}$ [4]. *Cuthill-McKee RB* method involves renumbering the elements in a subdomain using the Cuthill-McKee bandwidth minimization algorithm based on the *dual* of the mesh graph [16] and dividing the elements by choosing the separator at the median of the ordering.

(2) *Probabilistic Methods*: These techniques are the least popular choice for mesh partitioning. *Simulated annealing* attempts to achieve optimal load balancing by randomly assigning elements to domains in order to achieve some local minima [15]. Other examples include *genetic algorithms*. These methods are in general very expensive and require many iterations.

(3) *Iterative Local Migration*: This class of techniques involves the exchange of elements based on subdomain adjacency [28] or processor adjacency [17] to improve the load balance and/or communication overhead. These methods are particularly suited to dynamically evolving meshes which change incrementally. This allows for incremental balancing operations which can result in few elements being transferred.

Some disadvantages of the common implementations of the RB techniques are that the entire mesh resides on a single processor on which the partitioning is done and also, after the new load allocation is achieved, a global reordering of data is required. Also, one disadvantage of iterative local migration techniques is that several iterations may be required to regain global balance and hence elements may reach their final destination

after many local transfers rather than directly. The proposed algorithm for the dynamic load balancing of the evolving mesh is a geometry-based procedure. This is based on a modified Coordinate Bisection algorithm. The developed algorithm has the crucial property that the destination of the elements are predetermined such that migration iterations are not required. This results in substantial reductions in the time for load balancing to a given tolerance. The algorithm is also such that the partition separators for a given load balance at level n are usually reasonably close to those at level (n-1) for a small evolution of the grid. This results in fewer elements transferred and consequently reduces the load balance time.

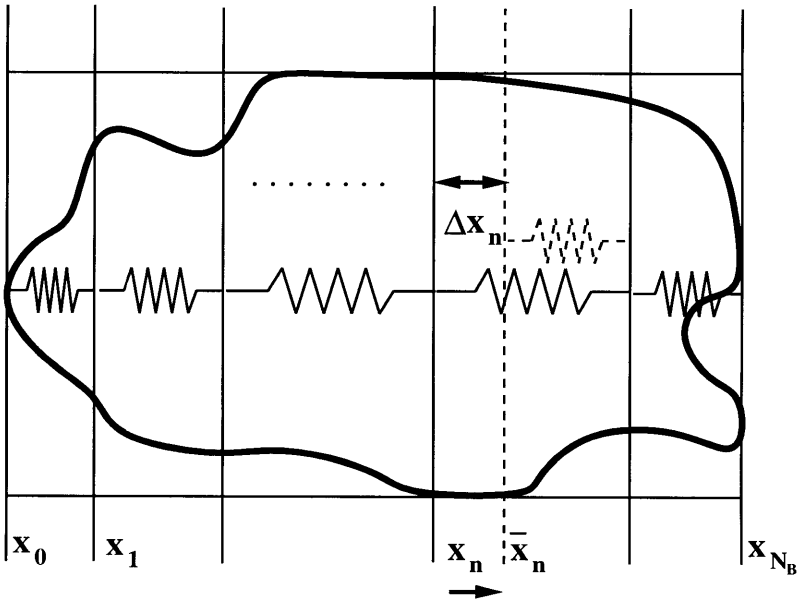


Figure 5.4: Linear 1D Load Balance System

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 partition separators are obtained by iteratively solving a linear system of equations derived from a spring analogy. In the 2D context, each block is represented as two linear springs in the x and y directions such that in each direction, the "spring constant" is proportional to the ratio between the number

of elements in the block and block dimension along that direction.

Consider the 1D system depicted in figure 5.4 which shows the computational domain divided into blocks with respect to one axis. The defining parameters are

1. $\mathbf{x}_i$, $\overline{\mathbf{x}}_i$: i^{th} partition coordinate before and after relaxation.
2. $n_i{}^e$, $n_{av}{}^e$: number of elements in block i and expected average number of elements per block.
3. K_i, $\overline{K}_i$: i^{th} block stiffness and inter-block stiffness coefficients.
4. N_B: Number of blocks.

An analysis of this linear system using propagation along the axis and taking into consideration the relative motion of the partitions results in the iterative procedure outlined in Appendix D to determine the partition locations. The procedure is executed in parallel by the master processor and the slave processors by initially creating the partition locations based on either an even element distribution assumption or making use of the previous partition locations on the master processor. The master then goes into a loop in which iterative sweeps in the X and Y axes are performed. At the beginning of the loop, the current partition locations are communicated to the slaves such that for all processor blocks which are represented by these partitions, the number of local elements within each block is computed by the slaves. This counting operation is based on the centroid location of each local element. Earlier attempts at making use of the number of local vertices contained in the blocks was made but experience showed that cleaner partitions occur based on the centroid. The next step in the loop is the axis sweep. At this point, it is necessary to define another term called a *band* which is simply the space between any two adjacent partitions along a given axis direction. This

may be thought of by dividing the computational domain into blocks by partition planes in the y-z plane into X bands. Y bands are then created by dividing a given X band into smaller blocks by partition planes in the x-z plane. Let ${n_x}^B$ and ${n_y}^B$ represent the number of bands along the X and Y axes respectively such that $N_p = {n_x}^B {n_y}^B$ and let ${}^G N_e$ be the global sum of the number of elements across all domains. The required average number of elements for a band in X ($B_x[i]$) or Y ($B_y[i,j]$) is computed from

$$B_x[i] = \frac{{}^G N_e}{{n_x}^B} \tag{5.1}$$

$$B_y[i,j] = \frac{B_x[i]}{{n_x}^B {n_y}^B} \tag{5.2}$$

Currently in the 2D context, the implementation on the choice of ${n_x}^B$ and ${n_y}^B$ is done to ensure that they are as close to each other as possible. This is essentially the same as making the processor block array as topologically close to a square as possible. Sweeps are made in the X and then Y directions to determine the new coordinates for the partitions. For a sweep in X, the band "spring constants" $\{K_i \mid i = 1..{n_x}^B\}$ are computed from the number of elements in each X band and the partition spacings in X. For a given displacement of a band, the number of elements in the adjacent band changes such that some measure of this needs to be added to the number of elements in the adjacent band. This is done by making use of inter-band "spring constants" $\{\overline{K}_i \mid i = 1..{n_x}^B\}$. Several options are open as to the computation of these values and tested forms include

1. **Average:** $\overline{\mathbf{K}}_i = \frac{\mathbf{K}_i + \mathbf{K}_{i-1}}{2}$
2. **Absolute Minimum:** $\overline{\mathbf{K}}_i = \mathrm{MIN}(\mathbf{K}_i, \mathbf{K}_{i-1})$
3. **Absolute Maximum:** $\overline{\mathbf{K}}_i = \mathrm{MAX}(\mathbf{K}_i, \mathbf{K}_{i-1})$
4. **Weighted Minimum:** $\overline{\mathbf{K}}_i = \frac{2\mathbf{K}_i \mathbf{K}_{i-1}}{\mathbf{K}_i + \mathbf{K}_{i-1}}$

5. **Weighted Maximum:** $\overline{\mathbf{K}_i} = \frac{\mathbf{K}_i^2 + \mathbf{K}_{i-1}^2}{\mathbf{K}_i + \mathbf{K}_{i-1}}$

It was found that in general, the weighted maximum form gave the best results. Next, testing must be done to determine if a band contains the required average number of elements. If a band contains the specified number of elements to within a prescribed tolerance, the band is fixed if and only *all bands behind or ahead are already fixed* (exceptions are the end bands). This implies that convergence of the partition locations occurs from the boundaries into the interior. The positional variation algorithm in Appendix D is then applied to all non-fixed bands while sweeping along the positive X axis. For the sweep along the Y axis, only Y bands located in a fixed X band are actually operated on.

This algorithm thus described provides a relatively fast incremental mechanism which is fitting to the evolving nature of the parallel mesh generation process. The fact that the old partition locations are used as the initial conditions helps in the convergence rate of the algorithm. Also, the provision of a prescribed tolerance for the number of elements within the bands means that full convergence does not necessarily have to be obtained while in the process of grid generation. This algorithm is in some respects similar to the "Strip" partitioning algorithm developed by Vidwans *et al* [58].

A pathological case which is not dealt with easily by this method is a mesh which contains regions of densely clustered elements. The implicit assumption that the number of elements varies "relatively" smoothly such that linear springs could be considered breaks down. In this case, the iterative procedure leads to an oscillatory solution which may require an unreasonable number of iterations to converge. This was dealt with by introducing band limiters which limit the minimum and maximum displacements for a partition. Hence for a sweep, say in X, the band limiters are applied to the first

non-fixed band because this band provides the base for the partition displacements. Assuming the lower and upper bounds for a given band i are $\mathbf{x_i}^{\mathbf{L}}$ and $\mathbf{x_i}^{\mathbf{U}}$ respectively,

1. $\bar{\mathbf{x}}_i \geq \mathbf{x_i}^{\mathbf{U}}$: $\bar{\mathbf{x}}_i = \frac{\alpha_1 \mathbf{x}_i + \alpha_2 \mathbf{x_i}^U}{\alpha_1 + \alpha_2}$

2. $\bar{\mathbf{x}}_i \leq \mathbf{x_i}^{\mathbf{L}}$: $\bar{\mathbf{x}}_i = \frac{\alpha_3 \mathbf{x}_i + \alpha_4 \mathbf{x_i}^L}{\alpha_3 + \alpha_4}$

where α_1, α_2, α_3 and α_4 are chosen such that $\alpha_2 \geq \alpha_1$ and $\alpha_4 \geq \alpha_3$ to provide weighting towards the bounds. After each update when the actual number of elements within the bands has been determined, the lower and upper bounds are set accordingly.

Upon the final determination of the partition locations, the final processor destination of the each element is assigned after which a migration schedule is set up to redistribute the elements. The actual simultaneous transfer of elements between the processors poses a unique problem which is directly related to the shared data formulation as described in the section on element migration. This implies that some ordering must be set up between the processors in order to maintain mesh consistency. This task is controlled by the master. The first step is the creation of an element transfer graph for the slaves such that an (i, j) entry is the number of elements to be transferred between processors i and j. This determines the possibility of element migration between any two processors. The slave processors create a list of all processor front segments attached to the elements to be migrated such that for each affected subdomain, the corresponding front segment list for that subdomain is communicated to the processor. For each received front segment id, a check is made on the front segment to test if the attached element is to be migrated. If the element is to be migrated and the UID of the remote processor across the front segment is numerically less than the processor UID, the front segment will be locked which means that it will not be transferred under any circumstance until a message to unlock the front segment is received. The last stage of

the migration schedule is a message driven loop on both the master and slave processors.

Master Processor Schedule

In addition to the transfer graph matrix, the master processor maintains a matrix of the state of each processor relative to the others such that an (i, j) entry determines if elements were actually transferred between processors i and j. The master also maintains a status list which at any given time determines if the processor is idle and free for migration or in the process of migration as either a sender or a receiver. The master processor then enters into a scheduling loop in which the transfer of elements between the processors is determined. The processors which are ready for migration are determined and notified with the UID of a processor ready to receive from them. This is done for a given idle processor i such that for any other idle processor j, the (i, j) entry of the transfer graph must be non-zero. In this case, if the transfer state between these two processors is True, then processor j is designated as a receiver for processor i. If the relative state is False, then the next idle processor with a relative state of True is selected. Provided that an (i, j) pair is determined, a RECV message is written to processor j and a SEND message is written to processor i. The status of processor i is upgraded to a send status while that of processor j is upgraded to a receive status. A polling for messages from any slave processor is then made such that if a message exists from a slave, it must be of one of the three following types which are

1. STATE: This means that the relative state of the slave has changed with respect to another processor. The UID of the other processor is read and the transfer state matrix is updated.

2. SEND: This is a handshake signal after the node p_A has transferred elements to another processor. The UID of this other processor (p_B) is read as well as the number of elements left to transfer from p_A to p_B. The transfer status of p_A is reduced to idle and if the relative state between these two processors did not change based on the number of element left to transfer and the transfer graph entry, the relative state matrix (p_A, p_B) entry is set to False else it is set to True. The transfer graph (p_A, p_B) entry is then updated.

3. RECV: This is received from any processor p_A which has just received elements. The list of processors affected by the migration is read and each processor is notified of possible incoming front updates after which the status of processor p_A is reduced to idle.

Slave Processor Schedule

This consists of a polling loop from both the master and other peer processors. A message from the master may be one of three types which are

1. SEND: The UID of the receiver processor is read and from the given list of elements to be migrated to the sender, the subset which can actually be transferred is extracted. This subset is determined by checking if any front segment attached to any element in the set has been locked. If there exists any processor front segment which has not been locked, is attached to any of the elements and satisfies the conditions that

    - The remote processor UID is numerically greater than the processor UID.
    - The remote element across the front segment is to be transferred.

then the remote front segment id is added to a list which represents a list of remote front segments to be unlocked. The element subset is transferred and the remote front segment ids are communicated to the receiver for unlocking.

2. RECV: The UID of the sender processor is read and elements are received. The list of processors affected by the migration isthen communicated to the master. The remote front segment ids described above are then received and for each affected processor, the corresponding subset of front segment ids to be unlocked is communicated.

3. FRONT: This is a signal from the master that there might be possible incoming data regarding an update in the front information.

A peer message from any other slave processor can only be of type UNLOCK from which the list of front segments to be unlocked are read. For each remote processor in the list of unlocked processor front segments, the relative state between the processor and the remote processor has changed such that this is communicated to the master. This essentially means that the processor now has a set of elements which it can migrate to any of these remote processors.

This concludes the entire load balancing procedure and it is seen that this provides a relatively simple and quick way of obtaining a partitioning of the computational domain. It is apparent that the quality of the partitioning may not be as good as perhaps Spectral RB. However, the motivation in mind was to obtain an algorithm which provides quick load balancing with scaling.

5.4.3 Element Request

The described point insertion procedures in chapter 2 (Watson and Green-Sibson) need to be modified to take into account the nature of the parallel mesh generation process. This involves the inclusion of a procedure to obtain the global element set which will be affected by the insertion of a newly created point [12, 43]. This is required for completion of the point insertion algorithms in order to guarantee the mesh is locally Delaunay. 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. As a means to accomplishing this, the modification of the base set of functions described in chapter 2 for facilitation of the mesh generation process is necessary and this is reflected in Appendix A.

Given an initial set of local elements (Ω_i^r) affected by a newly created point, the first step in any of the modified point insertion algorithms is the determination of wether remote elements might be required. This is done by searching for any processor front segments attached to Ω_i^r. If any of these front segments has been locked by any other remote processor, the request is automatically *denied*. A denied request is treated by reducing the seed element which was popped off the dynamic heap to the base state as described in chapter 2 and placing the element in a *denial queue*. If there are no attached processor front segments, no request can be made and the point insertion continues as in the serial context. At this stage, the two point insertion algorithms differ in the way the request is handled and this is described in detail below. As a means of notation, let P^r represent the request processor and $\{P_i \mid i = 1..N_r\}$ represent the set of processors from which the request is made.

Modified Watson Algorithm

The request algorithm for the Watson point insertion method is initiated by compiling the list of remote processors which are to be contacted $\{P_i \mid i = 1..N_r\}$ and then communicating to these processors, the coordinates of the new point. The request processor then enters a loop waiting for messages from peer processors.

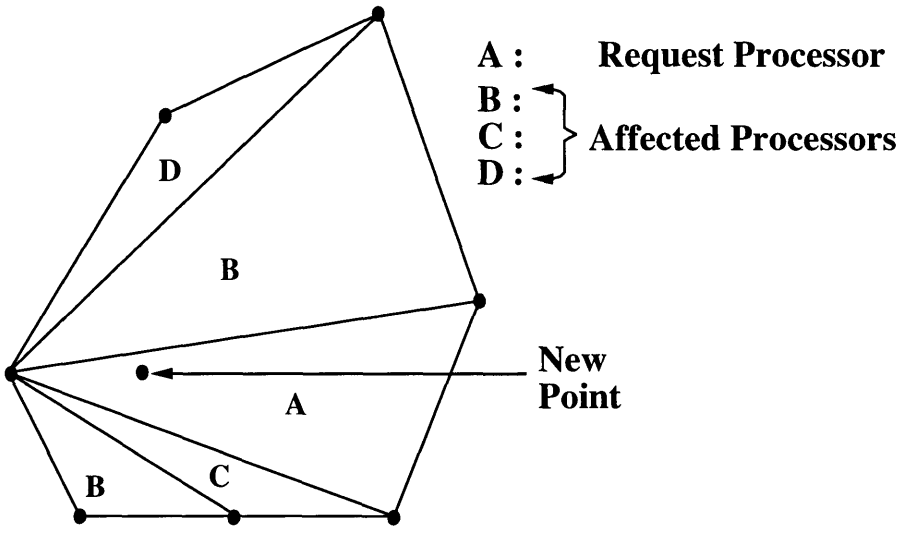


Figure 5.5: Remote Element Request Configuration Example

On any processor P_i in the request processor set, the coordinates of the newly created point are read and the request function (**ElementRequest**) described before is invoked. The algorithm behind the request function is an ADT tree search on the local element tree of all elements $\{\Omega^t\}$ which satisfy the circumcircle criterion. This is to be preferred to locating a seed element which satisfies the circumcircle criterion and performing a BFS search for the affected elements. As depicted in figure 5.5, consider processor B as a member of $\{P_i \mid i = 1..N_r\}$ such that two elements need to be transferred to processor A. If either of the two elements is identified as a seed element, a BFS search

will fail to detect the other. Thus, a complete ADT tree search is necessary to deal with fragmentation.

The next step in the process is to determine if there exists any overlap between any topological entitiy in use and $\{\Omega^t\}$. If this is the case, then the request processor (P^r) is denied. The next step is to request a lock on all processor front segments attached to $\{\Omega^t\}$ excluding the processor front segments which are shared by P^r. This is as described in the section under *element migration*. In the case that any of these front segments have been locked by another processor, the request processor (P^r) is automatically denied. If any remote processor could not grant the front locks perhaps due to the fact that they may have been locked by another processor, then the request processor (P^r) must be denied. In this case, all front locks which were actually granted by other processors must be unlocked by communicating a front unlock message to these processors as well as the front segments which must have been locked locally.

If all the front locks were granted, $\{\Omega^t\}$ is migrated to the request processor and the granted front locks which are characteristically remote front segment id and remote processor UID tuples are then transferred to the request processor. The request processor receives the remote elements and concatenates them to $\{\Omega_i{}^r\}$ to form a final element set $\{\Omega_f{}^r\}$. The boundaries of this transferred element set is checked for attached processor front segments such that if a processor from which a request has not been made lies across this front segment, a request is then made. This is depicted in figure 5.5 where processor B migrates the necessary affected elements but affected remote elements also exist in subdomain D. The transferred front locks are also received such that after any remote front update may have been made, the remote front segments associated with these front locks are then unlocked. The request processor (P^r) continues to poll until all affected processors have been accounted for.

The final element set $\{\Omega_f{}^r\}$ so created may not represent the true set of elements which are affected by the newly created point. This may happen as depicted in figure 5.6 in which processor A makes a request from processor B for the new point. Due to the way in which the request is handled in which an ADT tree search is done, this results in elements B_1 and B_2 migrated. The physical boundary separating element B_1 from subdomain A is part of the constrained triangulation and so this element cannot be a member of the set of elements affected by the new point. Hence, the proper set of elements must be recreated using the BFS algorithm described in chapter 2 using the original seed element. The Watson algorithm simply continues from this point as in the serial context.

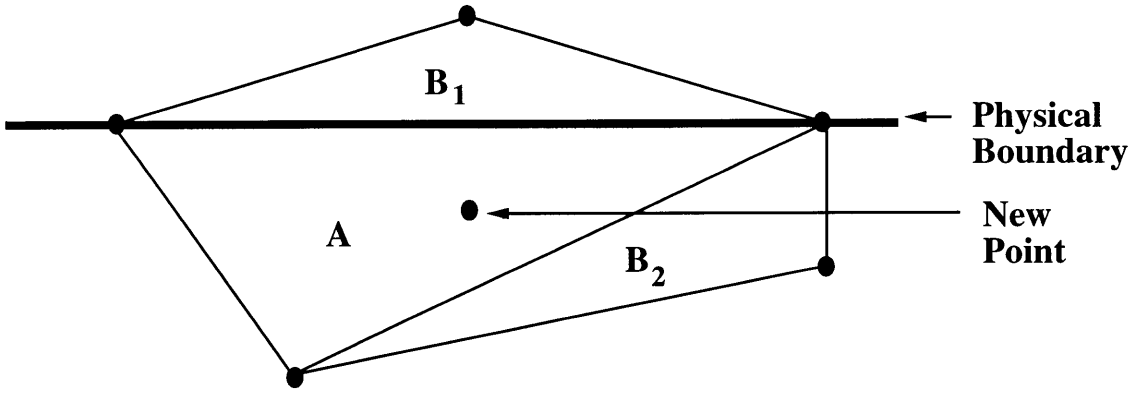


Figure 5.6: Constrained Triangulation Violation

Modified Green-Sibson Algorithm

The request algorithm for the Green-Sibson point insertion method is initiated by compiling the list of front segments $\{F_i{}^r\}$ attached to the initial element set $\{\Omega_i{}^r\}$. These front segments are stored and a marker index is created for each one to determine if the front segment is active. An active front segment is defined to be a candidate for forward propagation as described in chapter 2 under the Green-Sibson point insertion algorithm.

As described before, any existing lock on any of these front segments automatically terminates the request process and the request processor is denied. The front segments are initially made active after which a loop is entered. The design philosophy behind this algorithm differs from the Watson request algorithm in the sense that processors are contacted individually instead of broadcasting to all the affected processors. Due to the propagative nature of the Green-Sibson algorithm, it is necessary to ensure that a given request can be satisfied at all. This gives rise to the idea that the affected processors are contacted individually.

The identification of a candidate processor is made by considering the first active front segment encountered in $\{F_i^r\}$. For the corresponding remote processor across the front segment, all active front segments which are shared by the same processor are extracted and made inactive. The coordinates of the newly created point as well as the remote ids of the extracted front segments are communicated to the remote processor. The handling of the request from the point of view of the remote processor is similar to that based on the Watson request algorithm. The point coordinates as well as the front segment ids (local with respect to the remote processor) are read and if any of the specified front segments are locked, then the request processor is denied. For each of these front segments, the set of elements affected by the new point based on forward propagation from the element to which the front segment is attached is obtained. The union is made of these sets to obtain the complete list of elements $\{\Omega^t\}$ affected by the point. As described in chapter 2, the criterion for determining if an element is affected by a point does not necessarily have to be based on the circumcircle criterion.

The next step is to determine the topological entity overlap condition as described before. The handling of the overlap condition differs from the Watson request algorithm due again to the propagative nature of the Green-Sibson algorithm. As the affected

global element set grows during forward propagation, it is envisioned that when the overlap condition between the sets $\{\Omega^t\}$ and $\{\Omega_i{}^r\}$ of the affected remote processor is violated, then some kind of compromise must be reached between the request processor and the affected remote processor. This compromise is chosen such that if the UID of the request processor is the numerical minimum of the corresponding UIDs of both processors, then the request processor will be granted the request. If the overlap violation is due to element intersection i.e

$$\{\Omega^t\} \bigcap \{\Omega_i{}^r\} \neq \emptyset \tag{5.3}$$

then the remote processor must be denied since it no longer possesses the elements required to complete the point insertion. Also, if the seed element which generated $\{\Omega_i{}^r\}$ on the remote processor is a member of $\{\Omega^t\}$, then this must also be taken into account because this element would later have to be reduced to base state due to the denial. Since it no longer exists locally, it cannot be reduced. Front lock requests are then made from all processors (excluding the request processor) for all front segments attached to $\{\Omega^t\}$ as described before. If the locks are granted, the elements are migrated to the request processor and the front locks are also transferred to the request processor. If not, the same procedure for this case as described previously is performed.

The loop mentioned with respect to the request processor now consists of message testing from any peer processor. Requests from other processors are handled such that if the UID of the remote requesting processor is numerically less than that of the current request processor, then the request is handled. This strategy however can cause a deadlock for a unique configuration termed a *cyclic ring deadlock* (CRD). Consider figure 5.7 which depicts a request sequence created by a set of processors. Under normal circumstances, this ring will be broken if the processor with UID 3 receives the request message from the processor with UID 1 before that with UID 4. In the case where this

is reversed, processor 3 only "sees" the request message from processor 4 such that the linear chain between processors 1 through 4 will not be broken. This scenario may be corrected by actually checking for request messages from all processors with UIDs which are numerically less that the request processor UID.

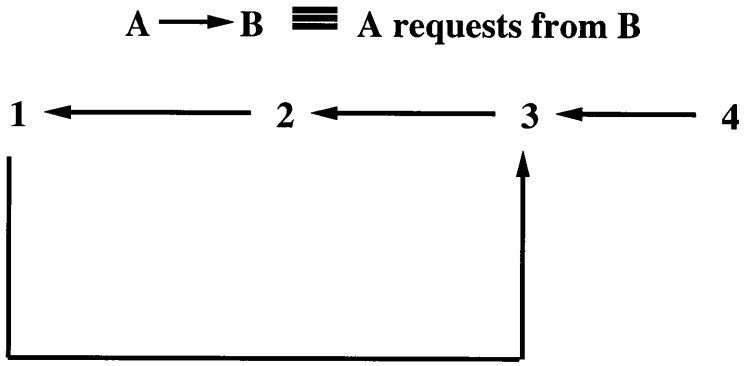


Figure 5.7: Cyclic Ring Deadlock

The request processor receives the remote elements and concatenates them to $\{\Omega_i{}^r\}$ to form a final element set $\{\Omega_f{}^r\}$. The boundaries of this transferred element set is checked for attached processor front segments such that if the remote processor across the segment is not the processor from which the elements were just received then these front segments are added to the set $\{F_i{}^r\}$ and tagged active. In the case of a local configuration as depicted in figure 5.8, the remote processor transfers the requested elements. However element E in the subdomain associated with the request processor is adjacent to the one of these elements and is not in the element set $\{\Omega_f{}^r\}$. Assuming that this element actually satisfies the criteria for inclusion in $\{\Omega_f{}^r\}$ due to forward propagation, this implies that forward propagation must be applied to this element to obtain any elements which need to be included in $\{\Omega_f{}^r\}$. All possibly attached processor fronts are extracted and included in $\{F_i{}^r\}$ and tagged active. The polling loop is terminated when either all the members of $\{F_i{}^r\}$ are inactive or the request was

denied. The Green-Sibson algorithm simply continues from this point as in the serial context.

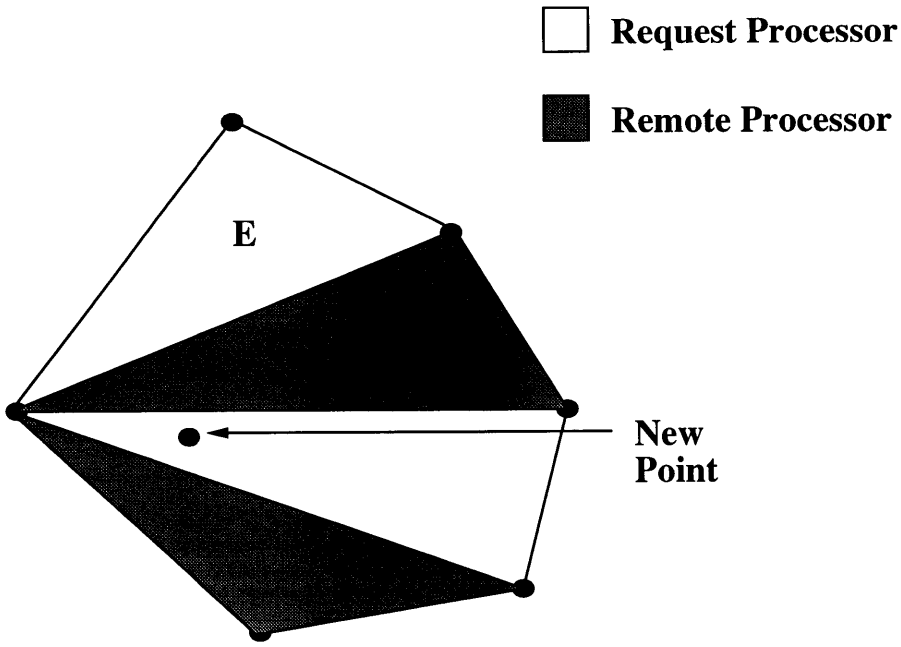


Figure 5.8: Green-Sibson Element Configuration Exception

5.4.4 Procedure

The mesh generation procedure in the parallel mode is initialized in as in the serial mode. The preprocessing stage begins with a CAD/CAGD description of boundary from which the boundary is discretized according to the background mesh and triangulated as described in chapter 2. This initial triangulation needs to be partitioned and two approaches have been tested. In the first one, the initial boundary triangulation is partitioned using the Greedy algorithm [6, 5]. This is preferred to coordinate bisection methods or the load balancing algorithm described for the initial boundary triangulation. The second approach is to have the entire initial boundary mesh reside on one processor such that after several points have been inserted, a load balancing is performed to distribute the mesh across all the processors. It was found that due to the fact that the initial elements are rather slender such that the insertion of a new

point affects a large number of elements, many requests and denials take place initially for the first approach. Hence the second approach is preferable. Figure 5.9 depicts the operation for the slave processors in which the slaves are allocated work by the master in element blocks. The slave operation then goes into a point insertion mode where new points and elements are created until a specified number of elements have been created for the given cycle. The currently implemented form for the total number of elements after a given cycle n is empirically given by a power law

$$^{G}N_e(n+1) = C_0 N_p [C_1]^n + {}^{G}N_e(n) \tag{5.4}$$

When the given number of elements has been created, load balancing is performed. Let the total number of processor fronts be represented by $^{G}N_{pf}$. The implemented element tolerance within each domain is given empirically by

$$\Delta E_{tol} = 2\frac{^{G}N_{pf}}{N_p} \tag{5.5}$$

The cycle is repeated as in figure 5.9 until the entire mesh is complete.

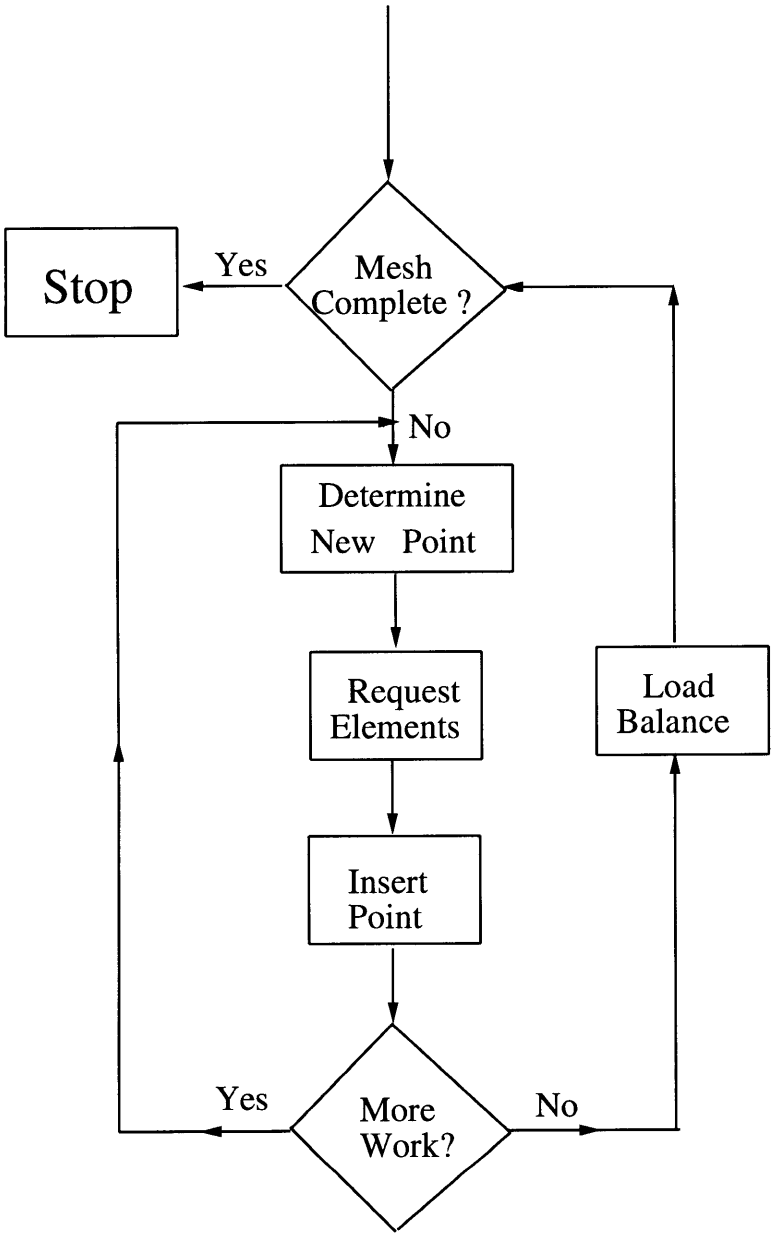


Figure 5.9: Slave Processor Operation

Chapter 6

Grid Generation Results

This chapter is devoted to an analysis of the different phases of the parallel mesh generation. The presentation of the mesh generation results will be based on mesh migration, load balancing and mesh generation timings. Mesh statistics such as grid quality are not discussed as these have been discussed extensively in the literature [42, 53, 41, 55]. The current platform used for parallel computation is the IBM SP2 situated in M.I.T which consists of 12 processor nodes with approximately 96 Mb of core memory each.

6.1 Mesh Migration Results

The performance of the mesh migration algorithm was tested by considering the throughput i.e number of elements transferred as a function of CPU time. This was done with two processors by initially creating the mesh on one processor completely. Several randomly chosen elements are then migrated to the second processor to give an approximate simulation of actual conditions during mesh generation. The remaining elements on the first processor are then migrated completely to the second processor and the CPU time required for this second migration is obtained. Table 6.1 gives a tabular representation of the results. The test was run three times to take into account network fluctuations and user contention. Both average and rms deviations based on three separate runs are shown in table 6.1. Comparison is also made for both the MPI and PVM message

passing libraries. Based on the raw data, the expected trend of an approximate linear variation is seen and this is better represented in figure 6.1. An interesting feature of

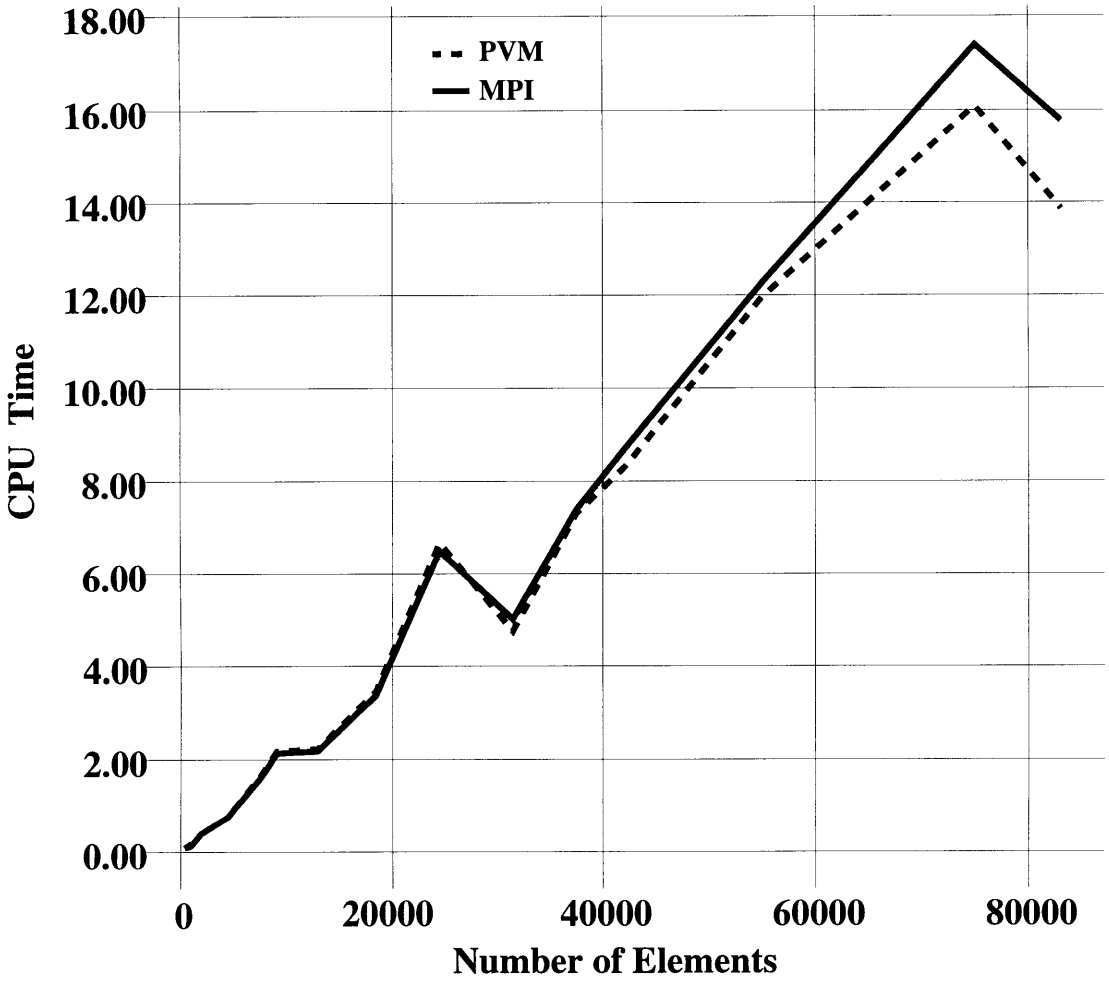


Figure 6.1: Mesh Migration Throughput

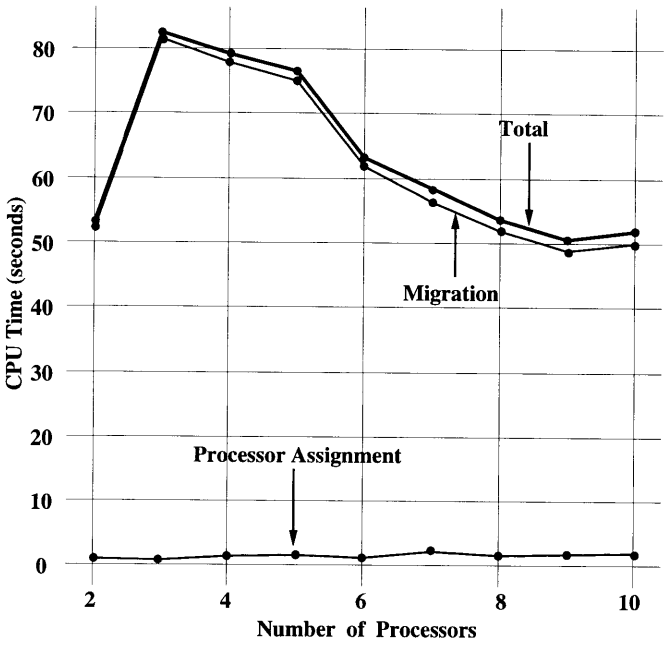
figure 6.1 is the unexpected dips which occur in the transfer times. This would normally not be expected but based on the analysis of Bokhari [52] for the IBM SP2, the time taken for a data packet to be sent over the network is sensitive to the size of the data packet. This would explain the inflections in the plot. For this same test, Shephard *et al* [36] obtain an average throughput of about 640 elements per second. This is to be compared with the current figures which average around 5000 elements per second.

Elements	Total CPU Time (secs) MPI	Total CPU Time (secs) PVM
500	0.07172 ± 0.00007	0.08683 ± 0.00373
1000	0.15401 ± 0.00016	0.16265 ± 0.00012
2000	0.41141 ± 0.00110	0.42495 ± 0.00536
4500	0.74668 ± 0.00084	0.76976 ± 0.00106
7500	1.61619 ± 0.00127	1.65675 ± 0.00131
9000	2.14673 ± 0.00046	2.19074 ± 0.00177
13000	2.16303 ± 0.00909	2.22293 ± 0.00172
18500	3.36195 ± 0.00114	3.45860 ± 0.00598
24500	6.49787 ± 0.00369	6.64235 ± 0.00369
31500	5.01537 ± 0.21958	4.73825 ± 0.00063
37500	7.36107 ± 0.05521	7.28208 ± 0.00286
42000	8.70766 ± 0.16631	8.28938 ± 0.00268
55000	12.25635 ± 0.05215	11.94889 ± 0.00746
75000	17.37797 ± 0.00624	16.08902 ± 0.60615
83000	15.77546 ± 0.00322	13.86933 ± 0.00361

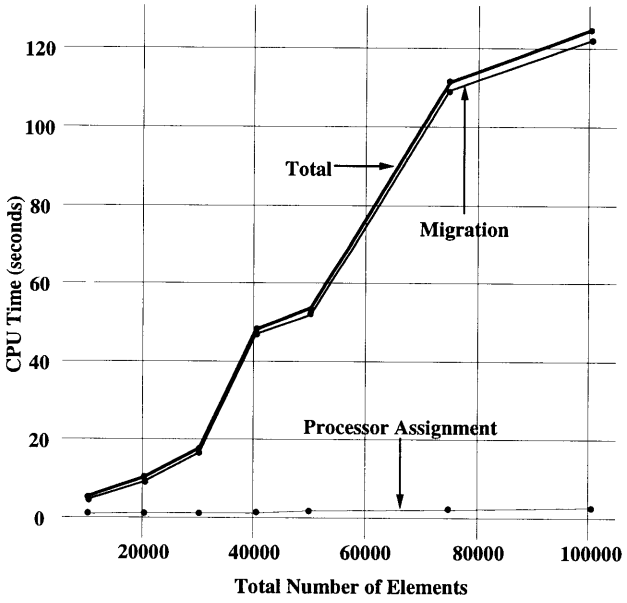
Table 6.1: Mesh Migration Throughput Comparison for PVM and MPI

6.2 Load Balancing Results

Benchmarking of the load balancing algorithm is done 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 since this affects the time required for the migration of the elements to the destination processors. Figure 6.2a shows the CPU time required to balance 50000 randomly distributed elements as a function of the number of processors and figure 6.2b shows the CPU time required to balance randomly distributed elements on 8 processors as a function of the number of elements. The total CPU times depicted have been decomposed into processor assignment times and element migration times. Figure 6.2a shows the expected trend of a decrease in the load balancing time with the number of processors and indicates scalability. The magnitude of the slope could perhaps be explained by noting that as the number of processors increase, the total communication length also increases. Hence, even though the average number of elements per processor decreases, this is offset by the increased communication. The load balancing algorithm however appears to be slightly sensitive to the subdomain partitioning as indicated by the 5 processor (1×5) and 10 processor (2×5) data points of figure 6.2a. The low times required for the two processor case may be explained by noting that the migration algorithm simplifies considerably when only two processors are involved.



(a) Random Assignment of 50000 Elements



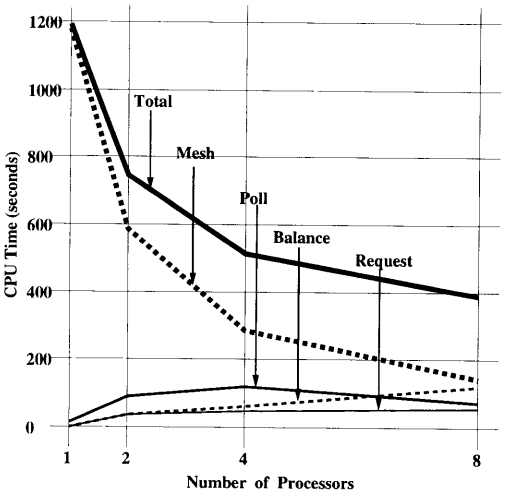
(b) Random Assignment for 8 Processors

Figure 6.2: Load Balancing Results for Random Element Assignment

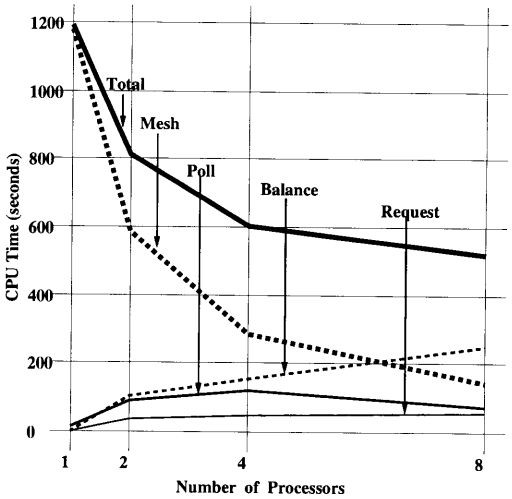
6.3 Mesh Generation Results

The above described algorithms are incorporated into a parallel mesh generation code and the performance obtained for a typical case is depicted in figure 6.3(a-d). This example uses the Green/Sibson point insertion and the Rebay point creation algorithms. Figures 6.3a and 6.3c show the mesh generation timings based on average and maximum processor CPU times respectively as a function of the number of processors for 1 million elements without a final fine-resolution load balance. Figures 6.3b and 6.3d show the mesh generation timings based on average and maximum CPU times respectively for 1 million elements 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 idle waiting for messages. All figures show the expected trend for the total generation times and show linear scaling for the actual time spent in mesh generation. As seen from the figures, the time for actual mesh generation decreases with the number of processors but the load balancing time increases with the number of processors. Hence some optimal processor size (element granularity) exists which minimizes the total mesh generation time for a fixed number of elements.

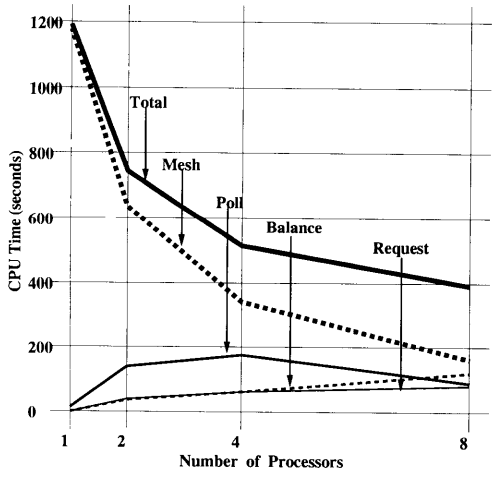
Figures 6.4(a-d) show a 4 processor example of load balancing during the process of mesh generation. Figures 6.4a and 6.4b depict an intermediate stage in the mesh generation about a NACA-0012 airfoil before and after load balancing respectively. Figures 6.4c and 6.4d show the final grid before and after load balancing respectively.



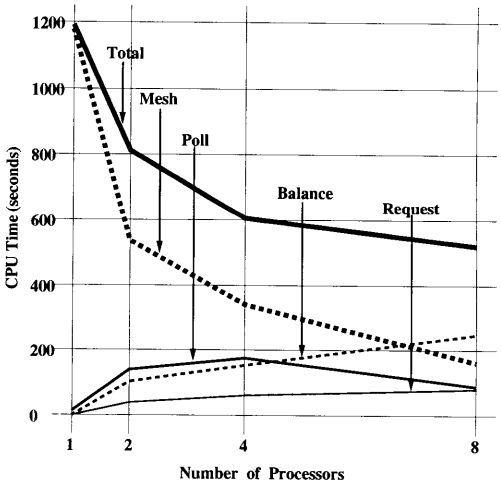
(a) Avg w/o Final Balance



(b) Avg w/ Final Balance

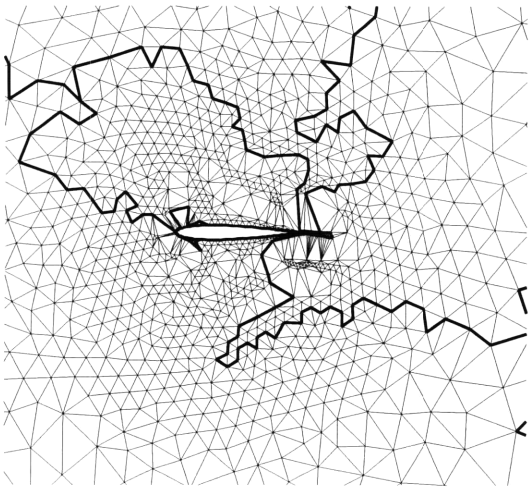


(c) Max w/o Final Balance

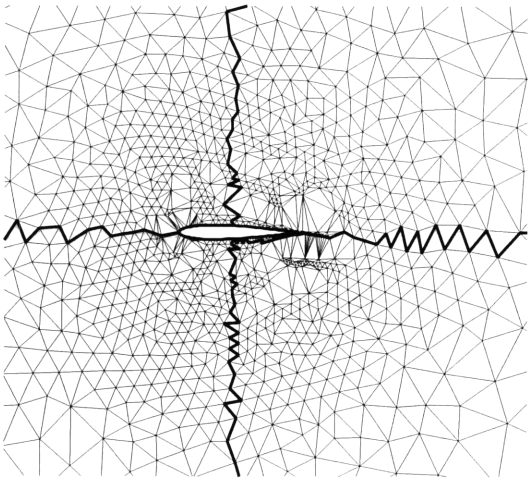


(d) Max w/ Final Balance

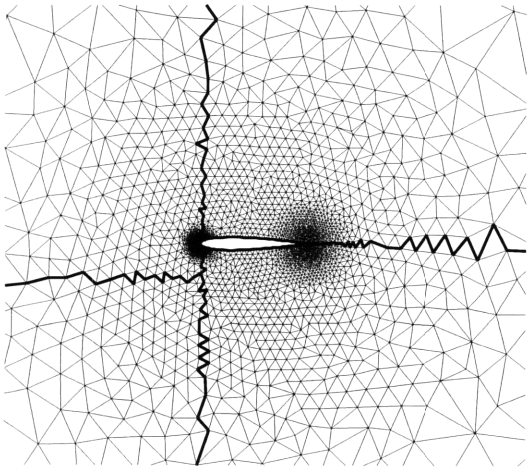
Figure 6.3: Mesh Generation of 1 Million Elements



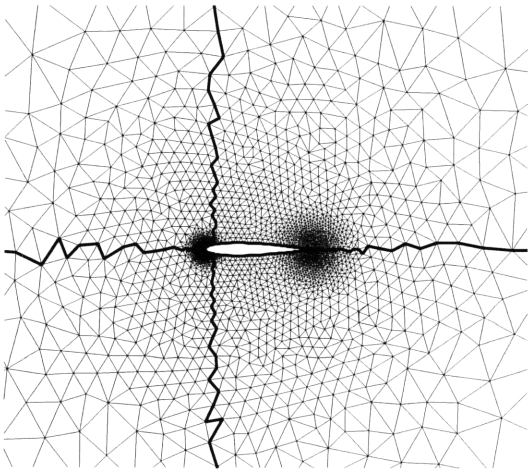
(a) Midway Unbalanced



(b) Midway Balanced



(c) Final Unbalanced



(d) Final Balanced

Figure 6.4: 2D Parallel Mesh Generation - Four processor illustrative example

Chapter 7

CFD Application

7.1 Preamble

The recent rapid developments of solution algorithms in the field of computational fluid mechanics means that it is now possible to attempt the numerical solution of a wide range of practical problems such that coupled with the power of parallel computers should allow for simulations on scales previously impossible.

The numerical solution of the Navier-Stokes equations should provide an accurate prediction of the location and strength of shock waves as well as provide an accurate representation of the physical flow conditions. The current implemented solution method makes use of the finite volume formulation of Jameson *et al* [3] that uses a Runge-Kutta multistep time marching scheme which has been shown to be second order accurate in space and fourth order accurate in time for RK4. The efficiency of the method is enhanced by artificial damping/dissipation to provide stability and resolution in shock regions. One major problem in solving flow problems over complex geometries is the generation of smoothly varying meshes about body configurations to enable flow variable computations with sufficient accuracy. The current implementation used to solve this is the use of unstructured grids which allow great flexibility in fitting complex geometries.

7.2 Governing Equations

The flow variables to be determined are the pressure, density, Cartesian velocity components, total energy and total enthalpy denoted by p, ρ, u, v, E and H, respectively. The conservative form of the compressible Navier-Stokes equations is given by

$$\frac{\partial \vec{U}}{\partial t} + \frac{\partial \vec{F}}{\partial x} + \frac{\partial \vec{G}}{\partial y} = 0 \tag{7.1}$$

$$\vec{F} = \vec{F}_{inviscid} + \vec{F}_{viscous} \tag{7.2}$$

$$\vec{G} = \vec{G}_{inviscid} + \vec{G}_{viscous} \tag{7.3}$$

$$\vec{U} = [\rho,\ \rho u,\ \rho v,\ \rho E]^T \tag{7.4}$$

$$\vec{F}_{inviscid} = \left[\rho u,\ \rho u^2 + p,\ \rho uv,\ \rho uH\right]^T \tag{7.5}$$

$$\vec{G}_{inviscid} = \left[\rho v,\ \rho uv,\ \rho v^2 + p,\ \rho vH\right]^T \tag{7.6}$$

$$\vec{F}_{viscous} = \begin{pmatrix} 0 \\ -\tau_{xx} \\ -\tau_{xy} \\ -u\tau_{xx} - v\tau_{xy} + q_x \end{pmatrix} \tag{7.7}$$

$$\vec{G}_{viscous} = \begin{pmatrix} 0 \\ -\tau_{yx} \\ -\tau_{yy} \\ -u\tau_{yx} - v\tau_{yy} + q_y \end{pmatrix} \tag{7.8}$$

Based on the assumption of a Newtonian fluid coupled with the Stokes assumption

$$\tau_{xx} = \frac{2}{3}\mu\left(2\frac{\partial u}{\partial x} - \frac{\partial v}{\partial y}\right) \tag{7.9}$$

$$\tau_{xy} = \mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right) \tag{7.10}$$

$$\tau_{yy} = \frac{2}{3}\mu\left(2\frac{\partial v}{\partial y} - \frac{\partial u}{\partial x}\right) \tag{7.11}$$

$$\tau_{yx} = \tau_{xy} \tag{7.12}$$

The heat fluxes q_x and q_y are expressed as

$$q_x = -K\frac{\partial T}{\partial x} \tag{7.13}$$

$$q_y = -K\frac{\partial T}{\partial y} \tag{7.14}$$

$$K = \mu\frac{C_p}{Pr} \tag{7.15}$$

$$T = \frac{p}{(\gamma-1)\rho C_v} \tag{7.16}$$

where

1. μ : Reference viscosity.
2. C_p : Specific heat at constant pressure.
3. C_v : Specific heat at constant volume.
4. Pr : Prandtl number.
5. T : Temperature.
6. γ : Ratio of C_p to C_v.

Closure to the above system of equations is provided by the perfect gas relations

$$E = \frac{p}{(\gamma-1)\rho} + \frac{1}{2}(u^2 + v^2) \tag{7.17}$$

$$H = E + \frac{p}{\rho} \tag{7.18}$$

7.3 Non-Dimensionalization

Non-dimensionalization of the Navier-Stokes equations [8] is done with respect to

1. L_R : Reference length.

2. U_R : Reference velocity.

3. ρ_R : Reference density.

4. μ_R : Reference viscosity.

such that

$$x^* = \frac{x}{L_R} \tag{7.19}$$

$$u^* = \frac{u}{U_R} \tag{7.20}$$

$$t^* = \frac{tU_R}{L_R} \tag{7.21}$$

$$p^* = \frac{p}{\rho_R {U^2}_R} \tag{7.22}$$

$$T^* = \frac{TC_p}{{U^2}_R} \tag{7.23}$$

$$\mu^* = \frac{\mu}{\mu_R} \tag{7.24}$$

$$\tau^* = \frac{\tau}{\rho_R {U^2}_R} \tag{7.25}$$

$$q^* = \frac{q}{\rho_R {U^2}_R} \tag{7.26}$$

$$\rho^* = \frac{\rho}{\rho_R} \tag{7.27}$$

The Navier-Stokes equations retain the same form with the introduction of the following auxilary equations

$${\tau_{xx}}^* = \frac{1}{Re}\left[\frac{2}{3}\mu^*\left(2\frac{\partial u^*}{\partial x^*} - \frac{\partial v^*}{\partial y^*}\right)\right] \tag{7.28}$$

$${\tau_{xy}}^* = \frac{\mu^*}{Re}\left(\frac{\partial u^*}{\partial y^*} + \frac{\partial v^*}{\partial x^*}\right) \tag{7.29}$$

$${\tau_{yy}}^* = \frac{1}{Re}\left[\frac{2}{3}\mu^*\left(2\frac{\partial v^*}{\partial y^*} - \frac{\partial u^*}{\partial x^*}\right)\right] \tag{7.30}$$

$${\tau_{yx}}^* = {\tau_{xy}}^* \tag{7.31}$$

$${q_x}^* = -\frac{\mu^*}{PrRe}\frac{\partial T^*}{\partial x^*} \tag{7.32}$$

$${q_y}^* = -\frac{\mu^*}{PrRe}\frac{\partial T^*}{\partial y^*} \tag{7.33}$$

$$T^* = \frac{\gamma}{\gamma - 1}\frac{p^*}{\rho^*} \tag{7.34}$$

$$Re = \frac{\rho U L}{\mu} \tag{7.35}$$

7.4 Spatial Discretization

The unstructured grid for the domain discretization is based on the distributed mesh which has been constructed in parallel as described in chapter 5. Consideration of a finite volume formulation of the Navier-Stokes equations results in

$$\int_\Omega \frac{\partial U}{\partial t} d\Omega + \int_\Omega \left(\frac{\partial F}{\partial x} + \frac{\partial G}{\partial y} \right) d\Omega = 0 \tag{7.36}$$

By considering the semi-discrete form of the above equation and applying Gauss's divergence law to the second integral

$$\int_\Omega \frac{dU}{dt} d\Omega + \int_{\partial\Omega} (F n_x + G n_y)\, d\Gamma = 0 \tag{7.37}$$

7.4.1 Interior Nodes

Considering the control volume Ω_o in figure 7.1 at the cell vertex O comprised of all the triangles sharing vertex O, application of the mean value theorem results in

$$\Omega_o \frac{d\bar{U}}{dt} + \sum_j \left[\left(\bar{F} {n_x}^j + \bar{G} {n_y}^j \right) l^j \right] = 0 \tag{7.38}$$

$$\bar{F}^j = \frac{1}{2}(F_I + F_{I+1}) \tag{7.39}$$

$$\bar{G}^j = \frac{1}{2}(G_I + G_{I+1}) \tag{7.40}$$

- $[{n_x}^j, {n_y}^j]^T$: Unit normal to edge j
- l^j : Length of edge j

- $\bar{F}, \bar{G}$: Average fluxes over edge j

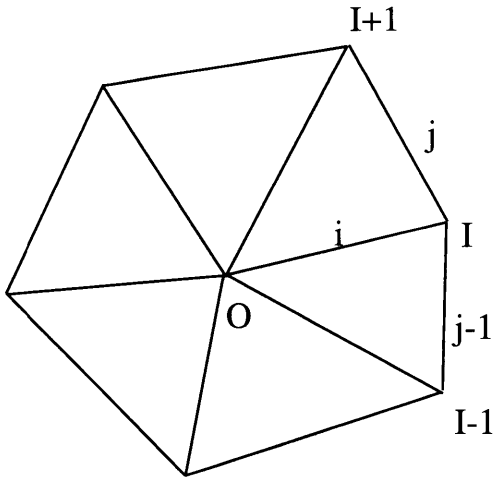


Figure 7.1: Interior Node Control Volume

Grouping terms on the RHS gives

$$\Omega_o \frac{d\bar{U}}{dt} + \sum_i \left[\frac{F_I}{2} \left({n_x}^{j-1} l^{j-1} + {n_x}^j l^j \right) + \frac{G_I}{2} \left({n_y}^{j-1} l^{j-1} + {n_y}^j l^j \right) \right] = 0 \qquad (7.41)$$

Hence weights may be defined for an edge i to be

$$w_x{}^i = \frac{1}{2} \left({n_x}^{j-1} l^{j-1} + {n_x}^j l^j \right) \qquad (7.42)$$

$$w_y{}^i = \frac{1}{2} \left({n_y}^{j-1} l^{j-1} + {n_y}^j l^j \right) \qquad (7.43)$$

The weight vector $({w_x}^i, {w_y}^i)$ is normal to the line segment (I-1,I+1) with a modulus proportional to the length of (I-1,I+1) as depicted in figure 7.2. The final form of the discretization is thus

$$\Omega_o \frac{d\bar{U}}{dt} + \sum_i \left({w_x}^i F_I + {w_y}^i G_I \right) = 0 \qquad (7.44)$$

where the sum is taken over all the edges which share node O. It may be verified that for an interior node O connected to node I along edge i,

$$\sum_i {w_x}^i = 0 \qquad (7.45)$$

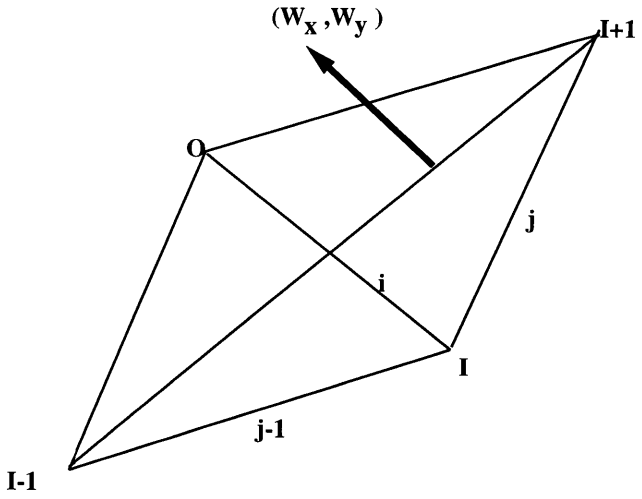


Figure 7.2: Edge Weight Vector

$$\sum_i {w_y}^i = 0 \tag{7.46}$$

$$({w_x}^i, {w_y}^i)_{OI} = -({w_x}^i, {w_y}^i)_{IO} \tag{7.47}$$

The scheme may thus be written as

$$\Omega_o \frac{d\bar{U}}{dt} + \sum_i \left[{w_x}^i (F_I + F_O) + {w_y}^i (G_I + G_O) \right] = 0 \tag{7.48}$$

where the term in brackets represents the flux contribution from node I to equation O. Due to the antisymmetry condition above, the flux contribution from node O to equation I along edge i will be

$$- \left[{w_x}^i (F_I + F_O) + {w_y}^i (G_I + G_O) \right]$$

Hence flux contributions may be computed by looping over the edges of the grid, computing the edge flux and sending the flux to node O and the negative of the flux to node I.

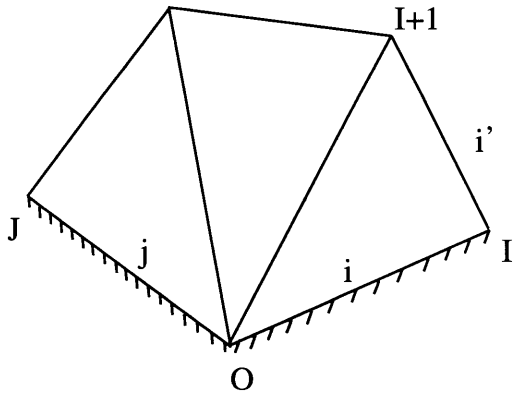


Figure 7.3: Boundary Point Control Volume

7.4.2 Boundary Nodes

The current definition for the control volume does not apply to boundary points since the volume is not closed. To obtain closure, we consider the boundary edges as part of the control volume boundary as depicted in figure 7.3. Considering the new control volume Ω_O comprised of edges {O, I, I+1, ..., J, O}

$$\begin{aligned}\Omega_o \frac{d\bar{U}}{dt} &+ \sum_{i'} \left[\frac{1}{2} {w_x}^{i'} \left(F_I + F_{I+1}\right) + \frac{1}{2} {w_y}^{i'} \left(G_I + G_{I+1}\right)\right] \\ &+ \left[\frac{1}{2}\left(F_J + F_O\right) n_x l + \frac{1}{2}\left(G_J + G_O\right) n_y l\right]_{JO} \\ &+ \left[\frac{1}{2}\left(F_O + F_I\right) n_x l + \frac{1}{2}\left(G_O + G_I\right) n_y l\right]_{OI} = 0\end{aligned}$$

To compute the contribution from node I to node O, we write

$$\begin{aligned}\frac{1}{2} F_I \left({n_x}^i l^i + {n_x}^{i'} l^{i'}\right) &+ \frac{1}{2} G_I \left({n_y}^i l^i + {n_y}^{i'} l^{i'}\right) \\ &= \\ F_I \left[\frac{1}{2}\left(\frac{1}{2}{n_x}^i l^i + {n_x}^{i'} l^{i'}\right) + \frac{1}{4}{n_x}^i l^i\right] &+ G_I \left[\frac{1}{2}\left(\frac{1}{2}{n_y}^i l^i + {n_y}^{i'} l^{i'}\right) + \frac{1}{4}{n_y}^i l^i\right]\end{aligned}$$

By defining ${w_x}^i$ and ${w_y}^i$ for boundary edges as

$${w_x}^i = \frac{1}{4}{n_x}^i l^i + \frac{1}{2}{n_x}^{i'} l^{i'}$$

$$w_x{}^i = \frac{1}{4} n_y{}^i l^i + \frac{1}{2} n_y{}^{i'} l^{i'}$$

the scheme may be written as

$$\begin{aligned} \Omega_o \frac{d\bar{U}}{dt} &+ \sum_i \left(w_x{}^i F_I + w_y{}^i G_I \right) \\ &+ \frac{1}{4} n_x{}^i l^i \left(F_I + 2F_O \right) + \frac{1}{4} n_y{}^i l^i \left(G_I + 2G_O \right) \\ &+ \frac{1}{4} n_x{}^j l^j \left(F_J + 2F_O \right) + \frac{1}{4} n_y{}^j l^j \left(G_J + 2G_O \right) = 0 \end{aligned}$$

By defining new weights for boundary edges only

$$\bar{w}_x^i = \frac{1}{4} n_x{}^i l^i \tag{7.49}$$

$$\bar{w}_y^i = \frac{1}{4} n_y{}^i l^i \tag{7.50}$$

the scheme may now be written for boundary nodes as

$$\Omega_o \frac{d\bar{U}}{dt} + \sum_i \left(w_x{}^i F_I + w_y{}^i G_I \right) + \sum_i \left[\bar{w}_x^i \left(F_I + 2F_O \right) + \bar{w}_y^i \left(G_I + 2G_O \right) \right] = 0$$

It may be shown that for a boundary node O

$$\sum_i w_x{}^i + 3 \sum_i \bar{w}_x^i = 0 \tag{7.51}$$

$$\sum_i w_y{}^i + 3 \sum_i \bar{w}_y^i = 0 \tag{7.52}$$

$$(\bar{w}_x^i, \bar{w}_y^i)_{OJ} = -(\bar{w}_x^i, \bar{w}_y^i)_{JO} \tag{7.53}$$

Hence the scheme may be rewritten as

$$\Omega_o \frac{d\bar{U}}{dt} + \sum_i \left[w_x{}^i \left(F_I + F_O \right) + w_y{}^i \left(G_I + G_O \right) \right] + \sum_i \left[\bar{w}_x^i \left(F_I + 5F_O \right) + \bar{w}_y^i \left(G_I + 5G_O \right) \right] = 0$$

This consists of the sum of a symmetric and an antisymmetric part. Hence the flux update for the nodes on a boundary edge involves sending the positive and negative antisymmetric flux to the two nodes on the edge plus the addition of the symmetric flux to both nodes. The derivative terms are computed simply by the use of the edge

weights. To compute $\frac{\partial U}{\partial x}$ for example,

$$\begin{aligned} \int_\Omega \frac{\partial U}{\partial x} d\Omega &= \int_{\partial\Omega} U n_x dS \\ \frac{\partial U}{\partial x}\Omega &\approx \int_{\partial\Omega} U n_x dS \\ \Rightarrow \frac{\partial U}{\partial x} &= \frac{1}{\Omega}\int_{\partial\Omega} U n_x dS \end{aligned}$$

This simply reduces to a summation over the interior and boundary edges which share the given point and computing flux values for the gradients based on the weights.

7.5 Artificial Dissipation

Most discrete approximations to the Navier-Stokes equations require some form of incorporated artificial dissipation to overcome two major problems.

1. Dissipation of high wave number oscillations which lead to numerical instability.
2. Shock capturing where it is needed to suppress/limit overshoot.

In order to accomplish this, the fluxes are modified to include second and fourth order dissipative terms of the form

$$\Omega_o \frac{d\bar{U}}{dt} + \sum Flux = D_O \tag{7.54}$$

$$D_O = \left(\frac{\Omega_o}{\Delta t}\right)[\nu_2 S(U_I - U_O) - \nu_4(U_{I'} - 3U_I + 3U_O - U_{O'})] \tag{7.55}$$

where O' and I' are extrapolated points on edge O-I and ν_2 and ν_4 are the second and fourth order artificial viscosity coefficients. Let $\vec{\sigma}$ be the local coordinate vector from node O to node I. Then the following relationships hold

$$\vec{U}_{I'} = \vec{U}_O + 2\left(\nabla\vec{U}_I.\vec{\sigma}\right) \tag{7.56}$$

$$\vec{U}_{O'} = \vec{U}_I - 2\left(\nabla\vec{U}_O.\vec{\sigma}\right) \tag{7.57}$$

The variable S_P is an edge based pressure switch which is computed as the maximum of the pressure switches computed at nodes O and I. The functional form for S_P is given by

$$S_P{}^O = \frac{|P_I - 2P_O + P_{O'}|}{(1-\epsilon)(|P_I - P_O| + |P_O - P_{O'}|) + \epsilon(P_I + 2P_O + P_{O'})} \tag{7.58}$$

This was the most successful form for the pressure switch as it properly accounted for strong shocks. This construction of the pressure switch ensures that $0 \leq S_P \leq 1$, with $S_P \approx 1$ in the vicinity of discontinuities and $S_P \approx 0$ in smooth regions of the flow. A better estimate of the $\left(\frac{\Omega_o}{\Delta t}\right)$ term may be obtained by replacing it with the maximum eigenvalue of the Roe matrix [44] such that

$$\left(\frac{\Omega_o}{\Delta t}\right)_e \sim [(|u_n| + c)\, l]_e \tag{7.59}$$

- u_n : Evaluated normal velocity to edge
- c : Local sound speed
- l : Length of edge

7.6 Temporal Discretization

The explicit time-marching was implemented by means of a general multistep Runge-Kutta scheme with frozen smoothing which involves only one evaluation of the artificial dissipation per iteration used in all the stages of the multistep. This is done for the sake of computational efficiency. For an N stage scheme

$$\begin{aligned} U_0^n &= U^n \\ U_1^n &= U^n - \frac{\Delta t_j}{N-0}\left(\sum_j Flux^0 - D^0\right) \end{aligned}$$

$$
\begin{aligned}
U_2^n &= U^n - \frac{\Delta t_j}{N-1}\left(\sum_j Flux^1 - D^0\right) \\
&\vdots \\
U_{i+1}^n &= U^n - \frac{\Delta t_j}{N-i}\left(\sum_j Flux^i - D^0\right) \\
&\vdots \\
U^{n+1} &= U_N^n
\end{aligned}
$$

The local timestep Δt_j employed at each node with control volume area Ω_j is computed according to an energy stability analysis by [32] and [49] using the relation

$$
\Delta t_j = 2\ \mathrm{CFL}\ \mathrm{S_f}\ \frac{{\Omega_j}^2}{\sum_e \left(\Omega_j \left\{|u_n|_e + c\right\} + D\frac{\mu_j}{\rho_j}\mathrm{MAX}\left(\frac{4}{3}, \frac{\gamma}{\mathrm{Pr}}\right)\right) l_e} \tag{7.60}
$$

where the summation is taken over all edges which share node j and

- CFL : Courant number
- S_f : Safety factor ~ 0.9
- c : Local sound speed
- l_e : Length of edge
- D : Diffusion factor

7.7 Boundary Conditions

For the computations involved, four different types of boundary conditions were specified as described below.

7.7.1 Wall Boundary Conditions

Due to viscous interactions on a wall, the no-slip condition must be satisfied such that u = v = 0. Also, if a thermally insulated wall is assumed, then the condition $\frac{\partial T}{\partial n} = 0$ must also be satisfied.

7.7.2 Symmetry Boundary Conditions

On a line of symmetry, the only condition needed to be satisfied is $\vec{U}.\hat{n} = 0$ where "." represents the scalar product.

7.7.3 Farfield Boundary Conditions

The farfield boundary conditions are described in [3] and [2], and outlined here briefly. On these boundaries, one wants to minimize the reflection of outgoing disturbances such that the use of Riemann Invariants is applicable. The Riemann Invariants are defined by

$$r1 = J^+ = u_n + \frac{2c}{\gamma - 1} \tag{7.61}$$

$$r2 = J^- = u_n - \frac{2c}{\gamma - 1} \tag{7.62}$$

$$r3 = v = u_t \tag{7.63}$$

$$r4 = s = ln(p) - \gamma ln(\rho) \tag{7.64}$$

- u_n : Normal velocity component to boundary tangent.
- u_t : Tangential velocity component to boundary normal.
- c : Local sound speed.

If the flow is subsonic at infinity, there will be three incoming characteristics (r1, r3, r4)where there is an inflow across the boundary and one outgoing characteristic (r2) corresponding to the possibility of escaping acoustic waves. Where there is an outflow, on the other hand, there will be three outgoing characteristics (r1, r3, r4) and one incoming characteristic (r2). Hence according to the theory of Kreiss [19], three conditions may therefore be specified at the inflow and one at the outflow, while the remaining conditions are determined by the solution. If the flow is supersonic at infinity, all characteristics will be incoming at the inflow and outgoing at the outflow. Hence four infinity conditions are specified at the inflow and the outflow is left untouched.

7.7.4 Boundary Layer Boundary Conditions

This is a special boundary condition applied to a boundary layer such that the static pressure was specified as $p = p_\infty$, while ρ , u and v were extrapolated from inside the domain.

7.8 Parallelization

The parallelization of the code is built around the distributed mesh as described in chapter 5. The implementation is based on the construction of a single layer halo of elements on the processor boundaries of the subdomains as in [56]. This is depicted in figure 7.4. The single layer halo is necessary to perform first order reconstruction of the solution in the vicinity of the processor boundaries. Parallelization is achieved by considering two aspects of the CFD algorithm outlined above.

7.8.1 Edge Weight Parallelization

The weights associated with the edges which lie on the processor boundaries need to be modified due to the fact that update of the flow variables on the vertices shared by remote processors needs to be done to ensure that the values at these local vertices correspond to the correct values on the corresponding global vertices. The edge weights are computed as described previously for non-processor boundary edges. For the processor boundary edges, each processor sharing that edge receives exactly half the associated edge weight values. Hence, computations involving processor boundary edges are done properly since each of the two sharing processors across the face will contribute exactly half the required value for the edge.

7.8.2 Parallel Variable Update

The update of a given variable defined on the vertices of the grid as mentioned above may be done only if the mapping between a given local boundary vertex and the tuple consisting of the set of processors and the remote vertex ids which share that boundary vertex (i.e the set of all remote boundary vertices which have the same global identifier) is known. Hence, an update of vertex based variables may be done by simply communicating the local contribution to the remote processors sharing the processor boundary vertices. This shared vertex mapping ($\mathcal{F}_s(\mathbf{v_i})$) is provided by the parallel unstructured mesh library on request.

Variable update is done by broadcasting the local contribution of the shared vertices to the affected remote processors along with the remote vertex ids of these vertices in the corresponding remote subdomain. Each received contribution is simply added to the corresponding local vertex value. Variable updates are required for five main procedures

which are

1. Viscous and inviscid flux updates.
2. Flow variable gradient updates.
3. Artificial dissipation updates.
4. Control volume computation.
5. Local and global timestep updates.

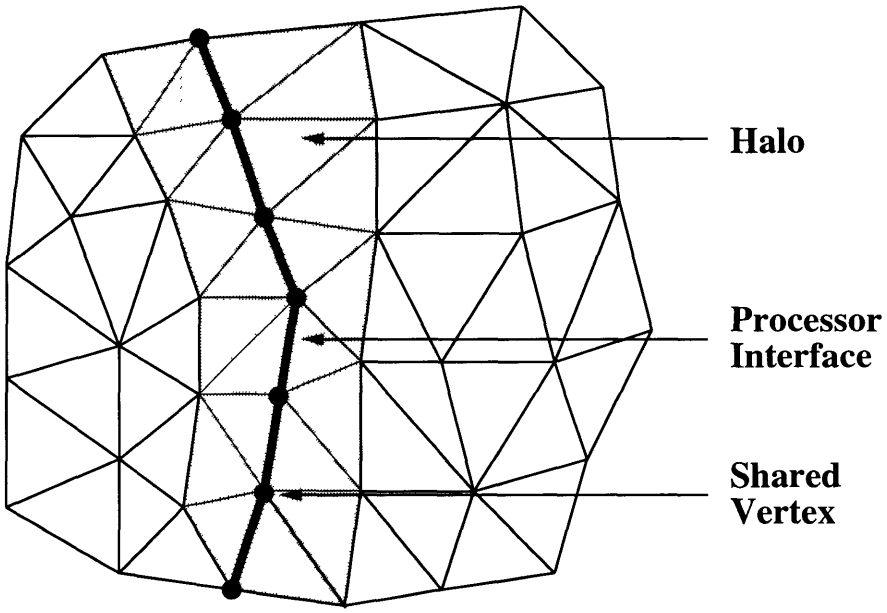


Figure 7.4: Single Layer Halo

7.9 Results

Two examples are included to illustrate the performance of the parallel CFD scheme described above. Both examples are based on steady state solutions and were selected to depict the different ranges of fluid dynamics.

The first example considered is supersonic inviscid flow over a 4% bump with a free stream Mach number of 1.4 as described by Ni [48]. The simulation was done on 4 processors with a grid consisting of 50000 elements. The Mach number contours are shown in figure 7.5 and these compare with those reported by Ni. However, the greater resolution of the grid resolved flow details which Ni missed such as the *Mach reflection* of the shock off the top wall as well as the intersection of this reflected shock with the shock formed at the trailing edge of the bump.

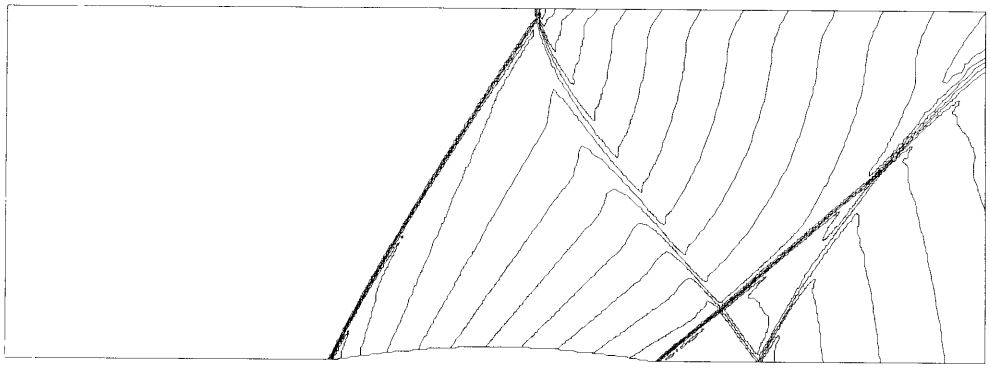


Figure 7.5: Mach Number Contours for Supersonic Bump

The second test case considered is the classic viscous flow over a flat plate for which analytic results exist. The free stream Mach number selected was 0.1 so that the flow could be considered to be essentially incompressible. The simulation was done on 2 processors with a grid consisting of 19000 elements. Anisotropic mesh refinement was employed to resolve the mesh in the regions of the boundary layer. The Mach number contours are as shown in figure 7.6 and these show Blasius boundary layer profile.

Theoretical values of the displacement thickness δ and momentum thickness δ^* can be calculated using the Blasius solution for incompressible flow over a flat plate. The

expressions are

$$\delta = \frac{5x}{\sqrt{Re_x}} \tag{7.65}$$

$$\delta^* = \frac{1.72x}{\sqrt{Re_x}} \tag{7.66}$$

For the case considered, $Re = 1000$ and $x = 1.0$ at the outflow, we obtain $\delta = 0.1581$ and $\delta^* = 0.05439$. The numercal results obtained for this case are $\delta = 0.1597$ and $\delta^* = 0.05513$. These results are in close agreement to the theoretical values with the discrepancies attributable to compressibility and streamline thickening effects.

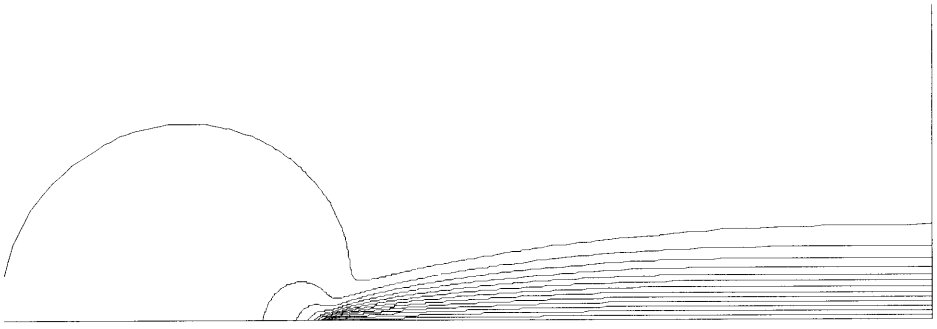


Figure 7.6: Mach Number Contours for Flat Plate

Chapter 8

Conclusions

An approach to parallel unstructured 2D mesh generation has been developed for operation on distributed memory MIMD computers. The developed algorithms include parallelization of the point insertion schemes with 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 but are easily portable to other platforms supporting standard communication libraries. Scalability of the mesh generation algorithm has been illustrated based on timings for different processor and mesh sizes.

An algorithm for extension of the isotropic mesh generation to anisotropic mesh refinement has been incorporated for analysis involving anisotropic gradients. An explicit finite volume solver has also developed for parallel solution of the unsteady Navier-Stokes equations.

The algorithms have been developed for 2D and will be extended to 3D to model practical applications. Current efforts are now focused on improving several aspects of the parallel mesh generation system such as inclusion of adaptive refinement capabilities as well as development of a more sophisticated load balancing strategy. Recommendations for improvement of the developed flow solver include conversion of the scheme from explicit to implicit with inclusion of GMRES. Also turbulence modeling needs to be included to be able to deal with high speed turbulent flows.

Appendix A

Mesh Generation Function Set

Serial Context

1. **InsertPoint:** Determines which point insertion algorithm to employ. The current choices are between Watson and Green-Sibson.

2. **InitElement:** Initializes a given set of elements by performing any preprocessing procedures and determines the subset which qualifies to be put on the dynamic heap.

3. **StatElement:** Actual function which determines if a given element belongs to the dynamic heap.

4. **CreatePoint:** Pops the head of the dynamic heap and creates a new point based on the geometrical properties of the popped element.

5. **CheckBound:** Determines boundary/geometric violations of a newly created point with respect to a set of elements.

6. **ReduceElement:** Reduces a set of elements to base state by deleting the elements from the dynamic heap and dereferencing any other associated data.

7. **GetElementSet:** Obtains the set of elements affected by a newly created point based on a seed element parameter. This is used in conjunction with the point insertion algorithms to determine the element set associated with a new point to

be inserted.

Parallel Extension

8. **ElementRequest**: Determines the set of elements which need to be migrated to a requesting processor based on some geometric object communicated from the request processor.

9. **ElementQueue**: In the case where the request cannot be completed due to a denial from a remote processor, the seed element is placed on a *denial queue* for later insertion attempts.

Appendix B

Parallel Communication Function Set

1. **ReadFromNode** (message, tag, node):

   Receive a message with given message tag from specified node.

2. **ReadFromAnyNode** (message, tag, node):

   Receive any message with given message tag from any node.

3. **WriteToNode** (message, tag, node):

   Send a message with given message tag to specified node.

4. **ReadFromMaster** (message, tag):

   Receive a message with given message tag from master node.

5. **WriteToMaster** (message, tag):

   Send a message with given message tag to master node.

6. **PollFromNode** (tag, node):

   Test for any message with message tag from specified node.

7. **PollFromAnyNode** (tag):

   Test for any message with message tag from any node.

8. **MultiCast** (message, tag):

   Broadcast message with message tag to slave nodes.

9. **ParallelSum** (variable):

   Perform a parallel sum over all slave nodes.

10. **ParallelMin** (variable):

    Perform a parallel minimum over all slave nodes.

11. **ParallelMax** (variable):

    Perform a parallel maximum over all slave nodes.

12. **ParallelSync** ():

    Perform a synchronization over all slave nodes.

Appendix C

Element Migration Algorithm

subroutine MigrateElements(Ω^m, P_d)

input : Ω^m: Element set to be migrated.

P_d : Destination processor

begin

Sender to Receiver

1. Secure lock on processor fronts associated with Ω^m from affected processors.
2. Set remote vertex ids of boundary points to be transferred and shared by receiver.
3. Pack element entities to be transferred into element, vertex coordinate and front groups. Front group automatically contains migrated boundary.
4. Delete all references to the migrated entities, send packed submesh and receive front update information from receiver.

Receiver from Sender

5. Receive and unpack packed submesh. Set proper local vertex ids on points which may have been duplicated on boundary as in the case of non-manifold mesh.
6. Insert unpacked submesh into local mesh.
7. Update fronts on affected processors and unlock remote locked fronts.

Affected from Receiver

8. Receive front update and unlock information. Update front.

Appendix D

Partition Location Algorithm

subroutine SolvePartition($\{\mathbf{x}_i \mid \text{i=0..}N_B\}$, ${n_{av}}^e$)

input : $\{\mathbf{x}_i\}$: Initial partition coordinates.

${n_{av}}^e$: Average block element count

$$\overline{K}_0 = 0$$

$$\overline{\mathbf{x}}_0 = \mathbf{x}_0$$

begin

For i = 1 to (N_B-1)

{

$$\Delta {n_{i-1}}^e = \overline{K}_{i-1}(\mathbf{x}_{i-1} - \overline{\mathbf{x}}_{i-1})$$

$${n_i}^e = {n_i}^e + \Delta {n_{i-1}}^e$$

$$\overline{\mathbf{x}}_i = \overline{\mathbf{x}}_{i-1} + \frac{{n_{av}}^e}{{n_i}^e}(\mathbf{x}_i - \overline{\mathbf{x}}_{i-1})$$

}

end

Bibliography

[1] A.Bowyer. Computing Dirichlet Tessellations. *The Computer Journal*, pages 162–166, 1981.

[2] A.Jameson and T.J.Baker. Solution of the Euler Equations for Complex Geometries. *Proceedings of AIAA 6^{th} Computational Fluid Dynamics Conference, New York*, pages 293–302, 1983.

[3] A.Jameson, W.Schmidt, and E.Turkel. Numerical Solution of the Euler Equations by Finite Volume Methods Using Runge Kutta Time Stepping Schemes. *AIAA*, 1981.

[4] A.Pothen, H.D.Simon, and K.P.Liou. Partitioning Sparse Matrices with Eigenvectors of Graphs. *SIAM J. Matrix Anal. Appl.*, pages 430–452, 1990.

[5] C.Farhat. A Simple and Efficient Automatic FEM Domain Decomposer. *Comput. and Struct.*, pages 579–602, 1988.

[6] C.Farhat and F.X.Roux. Implicit Parallel Processing in Structural Mechanics. *Computational Mechanics Advances*, pages 1–124, 1994.

[7] C.Ozturan, H.L. de Cougny, M.S.Shephard, and J.E.Flaherty. Parallel Adaptive Mesh Refinement and Redistribution on Distributed Memory Machines. *Comp. Meth. Appl. Mech. Engg.*, pages 123–137, 1994.

[8] D.A.Anderson, J.C.Tannehill, and R.H.Pletcher. *Computational Fluid Mechanics and Heat Transfer.* Taylor & Francis, 1984.

[9] H.L. de Cougny, 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*, pages 157–182, 1994.

[10] 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.

[11] D.E.Knuth. *The Art of Computer Programming. Vol. 1. Fundamental Algorithms.* Addison-Wesley,Reading,MA, 1969.

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

[13] D.L.Marcum. Control of Point Placement and Connectivity in Unstructured Grid Generation Proceedures. *Proceeds of the Ninth Intnl. Conf. on Finite Elements In Fluids-New Trends and Applications*, pages 1119–1128, 1995.

[14] D.Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, pages 657–673, 1994.

[15] D.Williams. Performance of Dynamic Load Balancing Algorithms for Unstructured Grid Calculations. Technical Report C3P913, California Institute of Technology, 1990.

[16] E.Cuthill and J.McKee. Reducing the Bandwidth of Sparse Symmetric Matrices. *Proc. ACM Nat. Conf., New York*, pages 157–172, 1969.

[17] E.Leiss and H.Reddy. Distributed load balancing: Design and performance analysis. Technical Report Vol. 5, W.M. Kerk Research Computation Laboratory, 1989.

[18] Farin Gerald. *Curves and Surfaces for CAGD.* Academic Press, Inc., 1988.

[19] H.O.Kreiss. Initial Boundary Value Problems for Hyperbolic Systems. *Comm. Pure Appl. Math.*, pages 277–298, 1970.

[20] H.Simon. Partitioning of Unstructured Problems for Parallel Processing. Technical Report RNR-91-008, NASA Ames, 1991.

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

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

[23] J.Boris. A Vectorized Algorithm for Determining the Nearest Neighbours. *J. Comp. Physics*, 1986.

[24] J.L.Bentley and J.H.Friedman. Data Structures for Range Searching. *Computing Surveys*, 1979.

[25] 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.

[26] J.Ruppert. *Results on Triangulation and High Quality Mesh Generation.* PhD thesis, Univ. Cal. Berkeley, 1992.

[27] J.Saltz, R.Crowley, R.Mirchandaney, and H.Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, pages 303–312, 1990.

[28] 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.

[29] K.Morgan, J.Peraire, and J.Peiró. Unstructured Grid Methods for Compressible Flows. *AGARD*, 1988.

[30] L.J.Guibas, D.E.Knuth, and M.Sharir. Randomizing Incremental Construction of Delaunay and Voronoi Diagrams. *Algorithmica*, pages 381–413, 1992.

[31] L.V.Kale. The Design Philosophy of Chare Kernel parallel Programming System. Technical Report UIUCDCS-R-89-1555, University of Illinois, Department of Computer Science, 1989 (Available via ftp from a.cs.uiuc.edu dir: /pub/CHARM).

[32] M.Giles. Energy Stability Analysis of Multistep Methods on Unstructured Meshes. Technical Report CFDL-TR-87-1, Computational Fluid Dynamics Laboratory/M.I.T, 1987.

[33] M.I.Shamos and D.Hoey. Geometric Intersection Problems. *17^{th} Annual Symposium on Foundations of Computer Science, IEEE*, 1976.

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

[35] 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. *Recent Developments in Finite Element Analysis*, pages 205–214. Int. Center for Num. Meth. in Engg., Barcelona, Spain, 1994.

[36] 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.

[37] M.S.Shephard and M.K.Georges. Automatic Three-Dimensional Mesh Generation by the Finite octree Technique. *Int. Journ. Numer. Meth. Engg.*, pages 709–749, 1991.

[38] M.S.Shephard and M.K.Georges. Reliability of 3-D Mesh Generation. *Comp. Meth. Appl. Mech. Engg.*, pages 443–462, 1992.

[39] N.P.Weatherhill. The Delaunay Triangulation - From The Early Work in Princeton. *Frontiers of Computational Fluid Dynamics*, pages 83–100, 1994.

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

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

[42] 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.

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

[44] P.L.Roe. Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes. *Journal of Computational Physics*, pages 357–372, 1981.

[45] R.Calkin, R.Hempel, H.C.Hoppe, and P.Wypior. Portable programming with the PARMACS message-passing library. *Parallel Computing*, pages 615–632, 1994.

[46] R.Löhner. Some Useful Data Structures for the Generation of Unstructured Grids. *Communications in Applied Numerical Methods*, pages 123–135, 1988.

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

[48] R.Ni. A Multiple-Grid Scheme for Solving the Euler Equations. *AIAA*, 1982.

[49] R.Radespiel. A Cell-Vertex Multigrid Method for the Navier-Stokes Equations. Technical Report 101557, NASA, 1989.

[50] R.Sedgewick. *Algorithms.* Addison-Wesley,Reading, 1988.

[51] R.Williams. Supercomputing Facility, California Institute of Technology.

[52] S.H.Bokhari. Communication Overhead on the Intel Paragon, IBM SP2 & Meiko CS-2. Technical report, ICASE Interim Report No. 28, NASA, 1995.

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

[54] T.J.Baker. Triangulations, mesh generation and point placement strategies. *Frontiers of Computational Fluid Dynamics*, pages 101–115, 1994.

[55] T.J.Barth. Aspects of unstructured grids and finite-volume solvers for the Euler and Navier-Stokes equations. Lecture Series 1994-05, von Karman Inst. for Fluid Dynamics, March 1994.

[56] T.J.Barth. Parallel CFD Algorithms on Unstructured Meshes. *Parallel Computing in Computational Fluid Dynamics, AGARD-FDP-VKI*, 1995.

[57] A.J. van der Steen. Overview Of Recent Supercomputers. Technical report, NCF Report, Stichting Nationale Computer Faciliteiten, Gravenhage, the Netherlands, 1994 (Available via ftp from ftp.cc.ruu.nl, dir: /pub/BENCHMARKS/reports).

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

[59] V.S.Sunderam, G.A.Geist, J.Dongarra, and R.Manchek. The PVM concurrent computing system: Evolution, experiences and trends. *Parallel Computing*, pages 531–746, 1994 (Available via ftp from netlib2.cs.utk.edu).

[60] V.Venkatakrishnan, H.D.Simon, and T.J.Barth. A MIMD Implementation of a Parallel Euler Solver for Unstructured Grids. Technical Report RNR-91-024, NASA Ames, 1991.

[61] W.J.Schroeder and M.S.Shephard. A Combined Octree/Delaunay Method for Fully Automatic 3-D Mesh Generation. *Int. Journ. Numer. Meth. Engg.*, pages 37–55, 1990.