Parallel Hierarchical 2D Unstructured Mesh
Generation with General Cutting

A thesis submitted by Vincent Charles Betro

I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computational Engineering.

We have read this thesis and recommend its acceptance:

Steve L. Karman, Jr.
Research Professor,
Computational Engineering,
Chairperson/Principal Adviser

W. Kyle Anderson
Professor,
Computational Engineering

Daniel G. Hyams
Associate Research Professor,
Computational Engineering

John V. Matthews III
Associate Professor,
Mathematics

Accepted for the Graduate Council:

Deborah E. Arfken
Dean of the Graduate School
PARALLEL HIERARCHICAL 2D
UNSTRUCTURED MESH
GENERATION
WITH GENERAL CUTTING

A Thesis
Presented for the
Master of Science Degree
The University of Tennessee at Chattanooga

Vincent Charles Betro
August 2007
Copyright © 2007 by Vincent Charles Betro.

All rights reserved.
Dedication

This thesis is dedicated to my family and friends, without either of whom I would not be where I am today. My circle of fantastic friends, both within and without the profession, has grown so much over the past two years and both my mind and spirit have grown as a consequence. My family members have always supported my desire to follow my dreams, and they have never wavered in their support for me and their desire to see me succeed.
Acknowledgments

Without the tireless help of Dr. Steve L. Karman, Jr., this thesis would not have been possible. I have learned so much of the world of grid generation and C++ programming that only two years ago I had no idea even existed. It has brought me great joy to work with such an enthusiastic and caring adviser, and I am excited to begin my Ph.D. work with him this fall.

This thesis was also made possible by all the wonderful people at The University of Tennessee SimCenter at Chattanooga, both faculty and fellow graduate students, who have shown me that I am capable of understanding anything that I set my mind to and who have also been willing to help me when I have hit road blocks in my understanding of computational fluid dynamics.
Abstract

In order to solve modern CFD problems, a robust parallel mesh generation process is created using MPI to allow for complicated geometries, rapid generation of meshes, and increased flexibility with spacing and mesh type. The program allows users to generate meshes in two dimensions, which then may be extruded to three dimensions, using any watertight geometry that is created by a CAD program. This procedure uses Quadtree refinement of a Cartesian root cell and general cutting to create both fully triangular, Delaunay, and hybrid meshes. Multiple cases were run to test the robustness of the algorithm including a two-dimensional NACA 0012 airfoil and a two-dimensional mapping of the Gulf of Mexico. Results from these cases are discussed as well as the current and future scalability and load-balancing of P_HUGG2D based on various parallel programming metrics.
Contents

1 Introduction ........................... 1
  1.1 Importance .................................................. 1
  1.2 Types of Meshing .......................... 2
    1.2.1 Extrusion .............................................. 2
    1.2.2 Overset ................................................. 2
    1.2.3 Delaunay ............................................... 3
    1.2.4 Cartesian .............................................. 3
  1.3 Geometries .................................................. 7
  1.4 Chapter Summaries ............................ 8

2 Implementation Terminology and Basic Data Structures 9
  2.1 Cartesian Hierarchical Terminology ................. 9
  2.2 Benefits of Using the Tree Structure ................. 12
  2.3 Data Structures ........................................... 13
  2.4 Memory Allocation ......................................... 15
  2.5 Conventions for Defining the Domain ................. 16

3 Algorithm Implementation .............................. 19
  3.1 Pre-Spawn Bookkeeping and Superstructure Creation ........ 19
  3.2 Spawning to Multiple Processes ...................... 21
  3.3 Refinement .................................................. 24
  3.4 General Cutting ........................................... 29
  3.5 Boundary Element Creation .............................. 40
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6</td>
<td>Final Mesh Creation</td>
<td>45</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Delaunay Output Mesh</td>
<td>47</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Hybrid Output Mesh</td>
<td>48</td>
</tr>
<tr>
<td>3.7</td>
<td>Optimization-based Smoothing</td>
<td>50</td>
</tr>
<tr>
<td>4</td>
<td>Experimental Results</td>
<td>54</td>
</tr>
<tr>
<td>4.1</td>
<td>NACA 0012 Airfoil Results</td>
<td>54</td>
</tr>
<tr>
<td>4.2</td>
<td>Gulf of Mexico Results</td>
<td>69</td>
</tr>
<tr>
<td>4.3</td>
<td>3-D Extrusion Meshes</td>
<td>76</td>
</tr>
<tr>
<td>4.4</td>
<td>Parallel Speedup</td>
<td>76</td>
</tr>
<tr>
<td>5</td>
<td>Conclusions and Future Work</td>
<td>90</td>
</tr>
<tr>
<td>5.1</td>
<td>Conclusions</td>
<td>90</td>
</tr>
<tr>
<td>5.2</td>
<td>Future Work and 3-D Implementation</td>
<td>91</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Multiblock Meshes</td>
<td>91</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Correcting Load-Balancing Issues</td>
<td>92</td>
</tr>
<tr>
<td>5.2.3</td>
<td>64-bit Version</td>
<td>93</td>
</tr>
<tr>
<td>5.3</td>
<td>Memory Allocation Test</td>
<td>99</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Individual Allocation</td>
<td>99</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Chunk Allocation</td>
<td>102</td>
</tr>
<tr>
<td>5.4</td>
<td>Class Header Files</td>
<td>105</td>
</tr>
<tr>
<td>5.4.1</td>
<td>VOXEL Class</td>
<td>106</td>
</tr>
<tr>
<td>5.4.2</td>
<td>CELL Class</td>
<td>110</td>
</tr>
<tr>
<td>5.4.3</td>
<td>NODE Class</td>
<td>113</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Geometry Class</td>
<td>115</td>
</tr>
<tr>
<td>5.4.5</td>
<td>Point Class</td>
<td>117</td>
</tr>
<tr>
<td>5.4.6</td>
<td>Vector Class</td>
<td>122</td>
</tr>
<tr>
<td>5.4.7</td>
<td>Processor-Index Class</td>
<td>128</td>
</tr>
</tbody>
</table>

Bibliography 94

Appendix 98
List of Tables

2.1 Run times for individual voxel object allocation versus group voxel object allocation .................................. 16

4.1 Load balancing eccentricities using round-robin assignment at spawn .......................................................... 85
List of Figures

1.1 Hierarchical Cartesian refinement on an airfoil . . . . . . . . . . . . . . . . . . 4
1.2 Subdivision refinement of a Cartesian cell in two and three dimensions . . 5
1.3 Limitations of Projection Cutting . . . . . . . . . . . . . . . . . . . . . . . 7

2.1 Diagram of Voxel neighbor numbering scheme . . . . . . . . . . . . . . . . . 11
2.2 Diagram of Voxel local node numbering scheme . . . . . . . . . . . . . . . . 11
2.3 Diagram of Voxel children numbering scheme . . . . . . . . . . . . . . . . . 12
2.4 Current mesh file format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

3.1 Super cell creation about a non-square outer boundary . . . . . . . . . . . . 20
3.2 Spawning to four processors . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.3 Effect of user-defined spacing parameters . . . . . . . . . . . . . . . . . . . 25
3.4 Assuring mesh quality . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
3.5 Smooth progression of voxel spacing in a hybrid mesh of the Louisiana coast 27
3.6 Smooth progression of voxel spacing in both a Delaunay mesh and a hybrid
mesh of a NACA 0012 airfoil . . . . . . . . . . . . . . . . . . . . . . . . . . 28
3.7 Ghost voxels within the scope of a given processor . . . . . . . . . . . . . 30
3.8 Uncut NACA 0012 airfoil . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
3.9 Using integer coordinates to alleviate tolerance issues . . . . . . . . . . . . . 34
3.10 Handling collinear geometry edges during general cutting . . . . . . . . . 35
3.11 Handling intersecting geometry edges during general cutting . . . . . . . 38
3.12 Coalescing boundary edges . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
3.13 Creating a boundary element from a cut voxel . . . . . . . . . . . . . . . . . 41
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.14</td>
<td>Splitting a closed loop into sub closed loops</td>
<td>43</td>
</tr>
<tr>
<td>3.15</td>
<td>Improperly marked voxel requires neighbor checking</td>
<td>46</td>
</tr>
<tr>
<td>3.16</td>
<td>Unsmoothed meshes of a NACA 0012 airfoil</td>
<td>52</td>
</tr>
<tr>
<td>3.17</td>
<td>Optimization-based smoothed meshes of a NACA 0012 airfoil</td>
<td>53</td>
</tr>
<tr>
<td>4.1</td>
<td>Shifting a NACA 0012 airfoil from the center of the domain</td>
<td>55</td>
</tr>
<tr>
<td>4.2</td>
<td>Symmetric NACA 0012 airfoil (Delaunay)</td>
<td>57</td>
</tr>
<tr>
<td>4.3</td>
<td>Symmetric NACA 0012 airfoil (hybrid)</td>
<td>60</td>
</tr>
<tr>
<td>4.4</td>
<td>Shifted NACA 0012 airfoil (Delaunay)</td>
<td>63</td>
</tr>
<tr>
<td>4.5</td>
<td>Shifted NACA 0012 airfoil (hybrid)</td>
<td>66</td>
</tr>
<tr>
<td>4.6</td>
<td>Gulf of Mexico (Delaunay)</td>
<td>70</td>
</tr>
<tr>
<td>4.7</td>
<td>Gulf of Mexico (Hybrid)</td>
<td>73</td>
</tr>
<tr>
<td>4.8</td>
<td>Coarse mesh on an x-cross-sectional slice of a 3-D Sea Fighter geometry and</td>
<td></td>
</tr>
<tr>
<td></td>
<td>extrusion into 3-D</td>
<td>77</td>
</tr>
<tr>
<td>4.9</td>
<td>Fine mesh on an x-cross-sectional slice of a 3-D Sea Fighter geometry and</td>
<td></td>
</tr>
<tr>
<td></td>
<td>extrusion into 3-D</td>
<td>78</td>
</tr>
<tr>
<td>4.10</td>
<td>Coarse mesh on a y-cross-sectional slice of a 3-D Sea Fighter geometry and</td>
<td></td>
</tr>
<tr>
<td></td>
<td>extrusion into 3-D</td>
<td>79</td>
</tr>
<tr>
<td>4.11</td>
<td>Fine mesh on a y-cross-sectional slice of a 3-D Sea Fighter geometry and</td>
<td></td>
</tr>
<tr>
<td></td>
<td>extrusion into 3-D</td>
<td>80</td>
</tr>
<tr>
<td>4.12</td>
<td>Mesh on a NACA 0012 airfoil with a 9 degree angle of attack</td>
<td>81</td>
</tr>
<tr>
<td>4.13</td>
<td>Processor configurations for four, eight, and sixteen processors (Delaunay</td>
<td>82</td>
</tr>
<tr>
<td></td>
<td>case)</td>
<td></td>
</tr>
<tr>
<td>4.14</td>
<td>Parallel timing for generating a mesh on a NACA 0012 airfoil</td>
<td>86</td>
</tr>
<tr>
<td>4.15</td>
<td>Parallel speed up for generating a mesh on a NACA 0012 airfoil</td>
<td>87</td>
</tr>
<tr>
<td>4.16</td>
<td>Parallel timing for generating a mesh on the Gulf of Mexico</td>
<td>88</td>
</tr>
<tr>
<td>4.17</td>
<td>Parallel speed up for generating a mesh on the Gulf of Mexico</td>
<td>89</td>
</tr>
<tr>
<td>5.1</td>
<td>Multiple block meshes</td>
<td>92</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Importance

In order to solve modern computational fluid dynamics problems, rapid, efficient, and high quality mesh generation must be a point of emphasis. Due to the complex nature of modern geometries such as aircraft, ships, and land areas, a method to generate an appropriately refined mesh automatically is desirable. Additionally, when using design optimization, the need for re-meshing automatically based on areas of high gradient between flow solver runs becomes paramount [Karman, 2004].

While commercial meshing packages such as GRIDGEN [Pointwise, 2007] and HARPOON [CEI, 2007] exist, these require the user to be highly knowledgeable of the software package and can become cumbersome due to time spent on manual manipulation and the iterative nature of mesh refinement. In order to allow those running simulations to concentrate on modifying the flow solver and interpreting results, the best option is to have a program which generates quality meshes in parallel with little user input.

While Karman has developed a package which generates high quality two-dimensional meshes in serial, HUGG2D, the large size of many modern geometries and the desire to lessen the great amount of time mesh generation takes has led to the need for parallelization. Thus, the current work, P_HUGG2D (Parallel Hierarchical Unstructured Grid Generator 2-D), is a scalable parallel implementation that uses different naming, storage, and cutting conventions more suited to use with the Message Passing Interface (MPI). MPI is the chosen...
method of parallelization because it is not only portable, but unlike OpenMP, it can be used effectively on both distributed memory machines and shared memory machines.

Using all these pieces in conjunction, a large mesh may be generated on the most complex of geometries in a rapid fashion, and this capability allows those working on fluid simulations as well as fuel cell theory, plasmas, and electricity and magnetism simulations to more easily generate large meshes and adapt them to better their solutions.

1.2 Types of Meshing

1.2.1 Extrusion

Extruding a mesh from a given geometry simply means adding layer after layer of cells, working out from a surface mesh generated on the geometry. While this is quite useful in prismatic meshing and viscous layer insertion, in practice it would be very difficult and time consuming to generate an entire mesh in this fashion [Karman, 95 (NASA)]. The difficulties created by stacking cells in tight areas and having to avoid cell edge intersections is among the reasons that this technique is limited to adding small numbers of cells.

However, this technique can be utilized in taking a two-dimensional mesh and extruding a three-dimensional mesh from it. This is analogous to using slices of a three-dimensional object and layering them one after another to re-create the original object.

1.2.2 Overset

Another option for generating meshes on large, complex geometries that may be implemented in parallel is the multi-block overset mesh. Multiple different overlapping structured meshes are created around a given geometry, and boundary information is periodically exchanged through some method of conveyance, such as Chémera interpolation, as the flow solver searches for a solution [Djomehri et al., 2003]. While this can be done very practically, the partitioning system in parallel can be rather difficult and many geometries are not well suited to structured meshes.
1.2.3 Delaunay

Another type of mesh that is relatively easy to generate on large, complex geometries is a Delaunay triangulation. This is defined as a triangulation wherein a circumcircle is placed around each potential element. Then, points are added such that in finality no other point from any other triangle is within the circumcircle of any other triangle [Ruppert, 1995].

The downfall of a Delaunay triangulation is that, in two dimensions, viscous layering requires quadrilaterals to properly obtain solutions at the boundaries, and this requires a hybrid mesh [Karman, 95 (AIAA)]. However, this type of mesh is still implemented in the present work as one option to take the nodes that are generated and determined through cutting to be inside the computational domain and create a purely triangular set of elements on each processor. Additionally, the same technique is used in generating the hybrid mesh in triangulating cut cells and cells with hanging nodes at level changes.

In order to obtain a quality mesh from a Delaunay triangulation, it is often necessary to use a smoothing algorithm before passing the mesh to a flow solver (Section 3.6). This assures quality elements in that it guarantees all triangles have aspect ratios within a certain user-specified range and that the number of elements either does not change or is refined based on adaptation parameters [Ruppert, 1995].

1.2.4 Cartesian

Cartesian meshing provides the most flexibility with the fastest mesh creation [Thompson et al., 1999], and thus it is used here in a parallel implementation. In order to mesh any given geometry, all that needs to be done is to create a supercell and subdivide it recursively until the desired spacing is reached at the finest level of refinement which should be in the vicinity of the geometry. This principle is seen in Figure 1.1, where the most refined cells are in the vicinity of the airfoil itself.

An example of a Cartesian mesh being used to better allow for viscous layering and to smooth the mesh to reduce solver instabilities is when a fluid solver is run on a “clean” aircraft to determine important changes in the flow field. The mesh is then adapted, and a
purely Cartesian mesh is easily generated around a geometry with the weapons bay added to obtain a more exact solution [Welterlen, 2000].

In order to begin generating a Cartesian mesh, the decision must be made as to which method of refinement is best suited to the general case. In Figure 1.2, the five options for refining a square and hexahedron are provided. The method of choice, in order to give the most refined mesh with the least ambiguity and most consistent, easy to traverse tree structure, is isotropic refinement, which is used in the current implementation [Karman, 2004].

Once the overall mesh is created around the geometry, the cells and/or nodes that are not inside the computational domain must be turned off (general cutting) and the leftover partial cells must somehow be converted into acceptable elements. The two methods of achieving this end product that were utilized in the current implementation are fully triangulated and hybrid meshing.

**Fully triangulated** After the basic Cartesian refinement is completed and the cells containing geometry have been cut, what remains is a list of boundary edges (which in parallel must be augmented by a list of inter-processor boundary edges) and a collection of
Figure 1.2: Subdivision refinement of a Cartesian cell in two and three dimensions – (a) Refinement options in three dimensions. The choice used to create the Octree is Isotropic Refinement [Karman et al., 2007]. (b) Refinement options in two dimensions. The choice used to create the Quadtree is Isotropic Refinement.
internal points that have been flagged as inside the computational domain. These two lists are passed to a Delaunay mesh routine that creates a super cell around the input geometry and recursively inserts the internal points and recovers the defined boundary edges. The result is a fully triangular mesh.

**Hybrid**  After the basic Cartesian refinement is completed and the cells containing geometry have been cut, what remains is a collection of intact cells and cut cells inside the computational domain. Here, since both quadrilaterals and triangles may be used, the only areas that must be triangulated are cut cells that have more than three sides and inside cells with hanging nodes. Once each of these is individually split into triangular cells, the remainder of the mesh is stored as quadrilateral cells. This type of mesh is appropriate for the addition of viscous layers, since it is already set up to take quadrilaterals and triangles [Karman, 2004].

It should be noted here that the general cutting method used in P_HUGG2D is very different from the projection cutting method used in Karman’s original three-dimensional serial code HUGG, but similar to the style used in his two-dimensional code, HUGG2D. In HUGG, the tree is traversed to check for intersections between the geometry facets and each cell. Boundary cells are then created by projecting the exposed Cartesian faces to the geometry. Those new projected faces are matched up with the defined geometry parts to identify the mesh boundaries [Karman, 95 (NASA)]. While this is more rapid than general cutting, it presents the problem seen in Figure 1.3, which is intended to be the view from the top of seven geometry facets coming together at a vertex. The top face of the cell highlighted in yellow is at the vertex of the seven different geometry parts, six of which have an equal projection area. Mesh edges in the vicinity should be moved to line up with the geometry edges. However, there is no point in the mesh that can have more than six edges emanating from itself. Another problem occurs if a cell is smaller than the spacing between two geometry segments and lies between them, it is not guaranteed to stretch and fill the gap. Thus, while the projection cutting process can be very rapid compared with general cutting [Karman, 95 (AIAA)], it is a process with definite limitations in three dimensions, so the
Figure 1.3: Limitations of Projection Cutting – The top face of the hexahedral cell to be distorted to fit into the scope of the light blue geometry facet is highlighted in yellow. However, the vertex at the center still cannot be modeled by Cartesian elements since a given point can have at most six edges emanating from itself.

decision was made not to use this method in preparation for the need for general cutting in the three-dimensional code.

1.3 Geometries

P_HUGG2D is equipped to handle any geometry, usually supplied by some Computer Aided Drafting (CAD) package. The only stipulation is that the geometry must be watertight (no holes) and the segments must be stored such that if they are interpreted as vectors from the tail node to the head node, the inside of the computational domain is to the left.

One interesting feature of P_HUGG2D is that, regardless of the spacing of the original geometry, the user can define a spacing that is more refined or less refined. Cells are generated around the geometry using this spacing, and when the boundary segments are merged within a cell, the resolution of the geometry can be downgraded based on the spacing supplied by the user. The exception to this is when segments from distinct boundaries meet inside a cell; they are not merged and the resolution of the original geometry is preserved.
In this thesis, two geometries are used for major examples. One, a simple NACA 0012 airfoil, is used for all initial testing on P_HUGG2D. It is then shifted upward inside the square outer domain (which also serves as the super cell in this case) in order to attempt to resolve as many aspects of the parallelization as possible. The second, a rendition of the coastline of the Gulf of Mexico, is used as a more difficult test case in order to ensure that P_HUGG2D can resolve a high resolution, complex geometry. While it presents many extra challenges, it also sheds light on the robustness of the current two-dimensional code and shows it usefulness in real-world problem solving.

1.4 Chapter Summaries

In Chapter 2, the terminology this thesis uses is explained and any terms specific to this implementation are defined. Additionally, the data structures that are used to aid in the parallel implementation and to increase the ease of use for P_HUGG2D are identified.

In Chapter 3, the algorithms for refinement, general cutting, and mesh file creation are outlined. Other options for node identification, cutting, and mesh style are also defined and rationales for why they are or are not used in P_HUGG2D are given.

In Chapter 4, resulting meshes about a NACA 0012 airfoil (in several positions), the Gulf of Mexico, and a Sea Fighter boat are displayed. The usefulness of these meshes is explained, and an extension into three-dimensions is also given.

In Chapter 5, a summary of the usefulness of P_HUGG2D is outlined. Future additions to the current code as well as uses of the current code in creating a robust three-dimensional version are also stated.

In the Appendix, the source code for memory allocation tests, along with a brief explanation, is provided. Also, the header files for each of the class structures implemented in P_HUGG2D are given.
Chapter 2

Implementation Terminology and Basic Data Structures

2.1 Cartesian Hierarchical Terminology

In order to create a mesh around any given geometry a root cell must be constructed. The essence of Cartesian hierarchical meshing is simply subdividing this root cell until a desired number of child cells is reached, and this desired number varies in different areas of the mesh depending on local refinement criteria supplied by the user. Before it is sensible to discuss mesh quality issues, it is advisable to discuss the data structure that is used to create this root cell: a voxel.

While the concept of a voxel is most often used in discussing computer graphics rendering, it is an appropriate term for Cartesian hierarchical meshing as well. The word comes about by combining the words volume and pixel, which here contains all the information about one specific portion of the mesh and its surroundings, down to a given resolution, just like a pixel. More importantly, the term voxel is used since the voxel itself knows nothing of its physical coordinates; rather, it simply contains information regarding its relational position to other voxels [Voxel, 2007]. The specific pieces of information the VOXL data structure contains are described next.
Each voxel contains the index of its mother voxel, so in a Quadtree structure with isotropic refinement, there are always four voxels with the same mother, and the root voxel is the only voxel without an initialized mother. It also contains its level, which begins with the root voxel at level one and increments each time a new generation of voxels is spawned. It contains an inside flag which is later used to determine whether the voxel should participate in cutting based on whether it is fully in the computational domain, fully outside the computational domain, or intersecting the geometry edges. Initially, this flag is used as a holding spot for different pieces of temporary information to more fully utilize the memory allocated in the class constructor. Also, it contains a new class variable called a \( p_i \) pair, which consists of the processor on which the voxel is located and its local index on that processor. This is a crucial addition to the parallel version of this type of mesh generation since communication can be made much more efficient and compact by knowing these pieces of information intrinsic to the class structure.

In addition to each voxel having information about its origin, it also has information about those voxels surrounding it. Each voxel has a list of neighboring voxels in its structure, and the neighbors are always ordered as shown in Figure 2.1. Using this knowledge and the fact that each neighbor is numbered according to its relative position gives the ability to determine information about which processors are on what edge of the voxel and how the nodes on the corresponding edges are related, which is very important in both communication and general cutting. Also, each voxel has a \( c_n \) list of the nodes of which it is comprised. These nodes also have a specific numbering, so as to be able to relate one voxel’s physical node to the same physical node shared on another voxel by both of them having the same processor-local node number as well as consistent voxel-local node numbering as seen in Figure 2.2. The indexing system shown in Figure 2.2 is based on the one-off indexing of the “C” programming language and its application to the current CGNS standard for node indexing \[CGNS, 2007\].

Finally, each voxel object contains information about its offspring and the geometry that may eventually cut through it. All voxels have \( \text{kids} \) arrays, which hold their four offspring the next level down in the Quadtree. These are also ordered in a consistent fashion (see Figure 2.3), and with the exception of ghost voxels, there is always a full complement of
Figure 2.1: Diagram of Voxel neighbor numbering scheme – The location in the neighbor array of each neighbor of a voxel is done using the consistent counterclockwise numbering scheme seen above.

Figure 2.2: Diagram of Voxel local node numbering scheme – The location in the voxel-local to processor-local node number connectivity array of each node of a voxel is done using the consistent counterclockwise winding seen above.
Figure 2.3: **Diagram of Voxel children numbering scheme** – The location in the children array of each child of a voxel is done using the consistent counterclockwise numbering scheme seen above.

four children. This consistent numbering helps not only with communication and checking the status of the parent voxel but is also absolutely necessary in determining neighbors using a tree structure instead of a recursive structure that is much more time consuming. Additionally, all voxels have a `bnd_lst` list that contains the numbers of the boundary edges, if any, that are created in that voxel during cutting. They also have a `loop` list of nodes which is allocated as necessary and contains each of the closed loops that can be formed from a given cut voxel.

### 2.2 Benefits of Using the Tree Structure

The most important aspect of the tree structure is the choice to use the isotropic Quadtree. This assures equal refinement in all directions, and the consistent numbering of children generated makes traversing the tree consistent. This is opposed to only refining in one direction and allowing that direction to switch, which causes an extra element to have to be added to the `VOXEL` object, namely, the orientation of the children as top-to-bottom or side-by-side.

By using the Quadtree structure when generating Cartesian meshes, the simplicity of subdividing squares repeatedly and keeping track of their relationships to each other using the mother-child construct alleviates the need to initially create physical coordinates and
thus necessitate tolerances to be observed to avoid duplication. This also lends itself to an open approach such as using integer coordinates since physical coordinates are not necessary to traverse the mesh until the final refinement is done and cutting is begun.

Also, using the tree structure makes inter-processor communication simple in that all processors have the full pre-spawn tree-structure; therefore, assigning ownership of voxels and nodes as well as creating ghost nodes is as simple as traversing back to the mother cell and duplicating its ownership information. More importantly, this simple traversal is used when recombining each processor’s final mesh into a global mesh file since each new voxel is simply an imprint upon the mother voxel, which is shared by all processors.

The tree structure may also be used to allow the ability to restart adaptive refinement at a higher level in the tree, avoid having to regenerate the initial levels on the whole mesh, and confine the refinement to only certain branches of the tree.

2.3 Data Structures

Along with the voxel object class there are four other heavily utilized class structures, three of which are instantiated with a voxel object (p_i.h, NODE.h, CELL.h) and the fourth with the instantiation of a geometry object (geometry.h). These are important because they contain a great deal of information based on certain conventions, which are discussed here. To view the header files for these classes as well as two utility classes (Point.h and Vector.h), please see Appendix 5.3.

The first class that is instantiated with a voxel object is the p_i class, which stands for processor-index pair. This convention is used in naming nodes, which are the physical coordinates in the mesh. It is also used in naming voxels and their counterpart cells, which are the final triangular or quadrilateral versions of the inside and cut voxels. If a node or voxel is shared between two processors, its p_i pair contains the lower of the two processors by which it is owned. Cells are the final units to be made into a mesh file, are only created after all refinement is complete, and are never shared between processors.

This naming convention comes into play after spawning off the children to each root process, since the spawn level children are all named based on the processor they are being
sent to, but the parents retain the designation of being from the root process (processor zero). This is because that is where the voxels are created, and it is the lowest processor on which they reside (the whole tree of voxels is sent to each processor at spawn). Using the \( p_i \) pair is even more important with nodes, since those created before spawn are remapped to the processor to which they are sent, but those on the border between processors are designated by the lower processor.

This concern for ownership of voxels, nodes, and cells is so important since it is the only definition of these elements that is unchanged regardless of location. In essence, it is the database key for these elements, since the local index used by each processor may vary. The index part of the \( p_i \) pair is the index of the node or voxel local to the process by which it is owned. This comes into play when, for instance, there is a ghost voxel, which has a local index on the processor on which it is a ghost as well as a different index to be used in communicating with the processor by which it is owned. The same goes for nodes, except that if a new node is created on a border, it is initially given a local index on each processor. Then, after communicating with all processors about new nodes, it is given the entire \( p_i \) pair of the node on the lower processor.

The second class instantiated with a voxel object is the \texttt{NODE} class. It contains only the \( p_i \) pair of the node and the physical coordinates of each node in the form of a \texttt{Point} class object, which allows for doing basic mathematical operations on components of physical coordinates and finding distances using class functions. This allows the voxel object to be true to its definition and not have direct knowledge of its physical coordinates, and it allows for the easy change of indices for the coordinate pairs without directly having to access them.

The third class instantiated with a voxel object is the \texttt{CELL} class. This class is limited to triangular and quadrilateral elements, and is used to store the final elements in a hybrid mesh. It contains only the cell’s \( p_i \) pair, its type (triangle or quadrilateral), and the \( c_n \) list (cell-to-node connectivity) for the cell, which is initialized to the proper length based on cell type.

When a geometry object is instantiated, the object contains the number of distinct boundaries, the number of geometry nodes on the boundaries, and the number of edges.
created by these nodes. Additionally, arrays of the beginning and ending nodes of each boundary, the spacings between the nodes and edges for each boundary, the length of each boundary, the high and low points of each boundary (to create a bounding box), the actual physical nodes on each boundary, and a list of edges are created. These pieces of information are then processed, and the user is told what the minimum and maximum spacings are on the boundary. This allows the user to determine how fine of a resolution must be observed in refining the voxels along the boundary and how coarse of a resolution may be used to define refinement parameters away from the boundary.

2.4 Memory Allocation

Something important to notice about the method of memory allocation in P_HUGG2D is that a combination of malloc and new is used to allocate objects. While new is the preferred “C++” allocation scheme which makes use of the constructor and destructor in the class object, it is not conducive to reallocation. Thus, while user-defined classes are used, artificial construct and destruct routines exist in order to utilize malloc allocation so that the necessary reallocations can be performed without wiping out memory or having to move massive quantities of memory into temporary storage locations while the old location is deleted and then allocated again.

In order to determine which allocation scheme is best for allocating voxel objects, a timing test is run wherein either 100,000 or 10,000,000 voxel objects are instantiated and destructed and the elapsed time is checked (see Table 2.1). The first test is to allocate the given number of objects one at a time, with some artificial calculation introduced in between to simulate a realistic situation within P_HUGG2D. While this proves to be rapid, the time it takes to do the allocation increases by almost two orders of magnitude from the small case to the large case. The second test is to allocate the memory in chunks of one or ten objects at a time. While the chunk size of one proves to be slower than the individual allocation (due to the extra processing required for the logic involved in allocating in chunks), the chunk size of 10 proves to be significantly more rapid than the individual allocations as well as more scalable, since the elapsed time only grows by one order of magnitude between the small
Table 2.1: **Run times for individual voxel object allocation versus group voxel object allocation** – The above timings (in seconds) reflect the elapsed time to run a driver program which only allocated voxel objects and destructed them upon completion. Based on these results, the determination is made to allocate voxels in chunks, as opposed to individually.

and large cases. This trade off of using a small amount of extra memory over spending extra computational time in allocation makes attempting to be truly memory efficient difficult. The decision is made to use extra memory since the two dimensional problems rarely run into memory issues, even in a 32-bit operating environment. For more information as well as the source code for these tests, please see Appendix 5.4.

### 2.5 Conventions for Defining the Domain

Not only must the voxels have a specific numbering scheme for the local node numbers, neighbors, and children but also the geometry must be watertight and wound in the proper direction. For this reason, the same convention was chosen for the boundary windings that was chosen for the voxels: all edges must be numbered such that the inside of the computational domain is on the left. This is not only useful in determining the “in” and “out” status of nodes using a vector approach but it is also helpful in the eventual creation of multi-block meshes (see Section 5.2). Currently, **P_HUGG2D** can only create single block meshes since if a second block is located inside the main computational mesh it would be considered to be outside the computational domain (see Figure 5.1).

In order for mesh creation to be of any use, a mesh file must be produced that can be read into the given solver program. The format used here yields a file structure as seen in Figure 2.4. It lists the number of nodes and their physical coordinates, the ordering of which is used to determine the node number (to be referenced later in the reading process). Then, the number of blocks, triangles and quads are each followed by listings of the node numbers.

<table>
<thead>
<tr>
<th>Number of Voxel Objects (using malloc)</th>
<th>Elapsed Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Individual</td>
<td>Chunk = 1</td>
</tr>
<tr>
<td>100,000</td>
<td>0.06</td>
</tr>
<tr>
<td>10,000,000</td>
<td>1.46</td>
</tr>
</tbody>
</table>
that create each element. Finally, the geometry is broken up into multiple boundaries, and the number of segments on each boundary is followed by the listing of the node numbers that correspond to the endpoints of each segment. This is made more difficult in parallel, since all of the processor-local node numbering must be remapped to global node numbering, and each element must be remapped to the new numbering as well. Currently, P_HUGG2D only produces a single mesh file for the entire domain that must be decomposed again for the solution to be computed in parallel; however, parallel and multi-block versions are in development.
Figure 2.4: **Current mesh file format** – Generic mesh file format produced by the meshing program and read by the flow solver.
Chapter 3

Algorithm Implementation

3.1 Pre-Spawn Bookkeeping and Superstructure Creation

In order to begin meshing a geometry once it is read into the geometry object, a root cell must be created around the given geometry. This is accomplished using two functions of the geometry class called `min_point()` and `max_point()`. These use the high and low points stored for each boundary (see Section 2.3) and discern the lowest and highest x and y values for the whole outer boundary.

These are then converted into the four corners of the root cell (numbered counterclockwise from 0 to 3) as such:

- lower left corner (node 0): \((low_x, low_y)\)
- lower right corner (node 1): \((hi_x, low_y)\)
- upper right corner (node 2): \((hi_x, hi_y)\)
- upper left corner (node 3): \((low_x, hi_y)\).

If the outer boundary is already square, these nodes coincide with the four existing corners of the outer boundary. If it is not, it creates a superstructure cell outside of the outer boundary, and any voxels created outside of the outer boundary but within the superstructure cell are deleted during the general cutting procedure (see Figure 3.1).

The nodes that compose the superstructure cell are stored as members of the `NODE` class and the superstructure cell is stored as the first, motherless voxel, with a level of one.
Figure 3.1: Super cell creation about a non-square outer boundary – In order to begin recursive refinement, a Cartesian super cell is created around the existing geometry, unless the outer boundary is initially square, in which case the super cell and the outer boundary are coincident and there will be no external voxels to be turned off during cutting.
3.2 Spawning to Multiple Processes

Once the root voxel is created, it is subdivided into four children through the creation of the mid-voxel node (which always has index 8, local to the voxel) and the four mid-edge nodes (locally indexed as 4, 5, 6, and 7 and shown in Figure 2.2). These children are subsequently divided in the same fashion, with new voxels and nodes being added each time until there are enough voxels at the finest level to either be equal to or supersede the number of processors requested. Figure 3.2 shows what the spawn map looks like when four processors are requested.

At this point, the voxels are mapped to a given processor, from least to greatest index. If there are more voxels than there are processors (e.g., the number of processors being requested is not a power of four and thus the voxels cannot be evenly distributed), the remaining voxels are arbitrarily mapped to share a processor from the lowest numbered processor to the highest. This is called “round-robin” assignment and while this is not a perfect load balancing scheme, it is not requisite for the rest of the implementation to work, thus room for improvement exists.

Upon spawning, each processor is sent the full tree up to the spawn level. Thus, each processor has all nodes and neighboring voxels, as well as parents, and uses this information later to place the appropriate ghost cells after each refinement. The \( p_i \) pairs of the voxels at the finest level are re-initialized based off the spawn map that is created to reflect ownership by the processor to which they are being sent while leaving their index intact. This is because the same number of voxels are sent to each processor, and thus the global and local index is, initially, the same; however, once recursive refinement continues, this is no longer the case.

After each processor receives all the nodes and voxels from the root, the nodes are remapped based on the \( p_i \) pair of the voxels in which they reside to share the processor index of the lowest process claiming ownership of the node. This is relevant when doing the flood fill, since all the initial nodes being owned by the root process necessitates extra communication and creates a scenario where there are nodes on a process that do not lie on the finest level, yet need to be mapped and are too disjoint to gather the right information.
Figure 3.2: **Spawning to four processors** – When four processors have been requested, once the super cell has been refined into four voxels, each processor receives one voxel, shown in different colors in the above figure.
This causes not only faulty flood filling but also an infinite loop to occur since these points do not reside on a voxel that will ever be considered in the flood filling algorithm.

Both before the spawn and after the spawn, the neighbors of each voxel must be determined so that the connectivity of the voxels is intact. As was discussed in Section 2.1, this knowledge of position based on neighboring voxels is the very essence of using the voxel construct.

Two options exist for accomplishing this neighbor update: one utilizes the tree structure and the other relies on pairing edge nodes based on the counterclockwise winding. While neither is complicated, the tree structure approach proves to be not only twice as fast empirically but also much easier to use without the need for extra parallel communication.

The first type of neighbor search cycles though each voxel and then executes another loop that looks through the voxels that are neighbors of the parent and the other children of the parent itself. Each of these voxels is then tested to determine if it contains the two nodes of one edge of the voxel in question (but in reverse order). If so, it is set to be the neighbor of that voxel, and it is indexed based upon the local indices of the nodes that are found to be in common. This presents a problem once ghost voxels are implemented in that it requires extra communication about the refinement of the ghost voxel to be transmitted beyond whether it is refined or not, and what the new voxel indices are.

On the other hand, the use of the tree structure simply uses the fact that any child voxel automatically has two of its neighbors as other children of the same parent voxel, and their location is consistent due to the chosen numbering scheme. Then, so long as communication has occurred in the refinement process for voxels on a processor border, all voxels at the finest level are known, and the neighbor on the same side of the parent is the new neighbor of the child voxel or an appropriately numbered child of that original neighbor is the new neighbor.

One small piece of information to note here is that in either method, if a voxel does not refine and its neighbor does (but only by one level since the mesh quality routine would not allow any more difference than that between adjacent voxels), the unrefined voxel only knows of neighbors on the same level or lower. Thus, there is a one-way connection between
smaller voxels and larger neighbors and care must be exercised in searching neighbors for mid-edge nodes generated due to such a refinement.

### 3.3 Refinement

Since there is a tree structure created as refinement occurs, it is fairly straightforward to keep track of which voxels need to be refined. After each refinement and the creation of physical nodes to accompany the new voxels, a check occurs on each voxel at the lowest level to determine if it is larger than the desired maximum spacing, and if so, it is set to be refined. Additionally, each voxel is checked for geometry intersections. If it has geometry inside its domain, it is either refined in an attempt to attain the minimum spacing of the geometry it contains or, if it is smaller than the geometry spacing, the user-defined minimum spacing. Figure 3.3 shows how changing the user-defined spacing affects the resolution of a mesh.

This refinement does not occur instantaneously, since the only way to refine a voxel is to subdivide it into four children and create the accompanying physical nodes. Then, the algorithm must check to see if this process needs to be repeated until the desired spacing is reached both in voxels containing geometry and those less-refined voxels in regions of the mesh where the solution gradients are anticipated to be small.

Throughout all this, mesh quality is enforced, such that the situations in Figure 3.4 are avoided. Thus, two refined voxels cannot be neighboring one voxel on opposite edges without the voxel in question being refined, as can be seen in Figure 3.4(a). Also, one edge will not encompass more than two levels, as can be seen in Figure 3.4(b). In enforcing these constraints, the mesh spacing changes smoothly in refinement from the large spacing of the far-field to the tight spacing of the voxels containing geometry as can be seen in Figures 3.5 and 3.6. Also, it ensures that there is at most one hanging node (or a node that splits an edge in two but has no other segment emanating from it within the given voxel) per voxel edge, which greatly aids in the creation of a hybrid mesh, wherein these voxels are triangulated.
Figure 3.3: **Effect of user-defined spacing parameters** – (a) This four processor mesh on the Gulf of Mexico uses a minimum spacing of 0.0001 units and a maximum spacing of 10,000 units. This allows refinement to occur at a very high resolution around the geometry (which has a minimum spacing of 0.579746 units and a maximum spacing of 1187.32 units) but only at the bare minimum resolution in the empty center of the mesh. (b) This four processor mesh on the Gulf of Mexico uses a minimum spacing of 0.001 units and a maximum spacing of 1,000 units. This allows refinement to occur at a slightly lower resolution around the geometry (which has a minimum spacing of 0.579746 units and a maximum spacing of 1187.32 units) and at a slightly higher resolution than the bare minimum in the empty center of the mesh.
Figure 3.4: **Assuring mesh quality** – (a) The voxel that is shaded in grey is surrounded by two refined voxels on opposite edges, and thus leads to marking the shaded voxel for refinement. (b) The red edge connects multiple voxels that are not within one level of each other, and thus the lowest level voxel is marked for refinement.

Additionally, a carry over from HUGG2D is that the user is allowed to input a cell size gradation parameter, between 1 and 2, which enforces the relative size of adjacent voxels. If during refinement, it is detected that an adjacent voxel is more than the given cell size gradation parameter times larger than the neighbor, the adjacent voxel is refined [Domel et al., 2000]. This tends to force the voxels to be on the same level for four to five voxels in either direction in P_HUGG2D, where a cell size gradation parameter of 1.5 is used.

At this point, an explanation of ghost voxels is necessary. In order for voxels on the border of each processor to check for mesh quality, a system exists that utilizes the fact that the initial, pre-spawn refinement and all of the voxels it created is passed to each processor at spawn. The voxels that are not owned by the processor in question are given local numbering while retaining their original $p_i$ pair, and thus they can easily be distinguished as ghost voxels by simply checking $\text{voxel}[i].\text{proc}$ and noting that its index is not the same as that of the current processor.

At the end of each refinement pass, the voxels on processor borders communicate to check for children having been created on other processors. If children exist and/or mesh quality is violated, further refinement occurs.

These ghost voxels are streamlined by only storing progeny that border the current processor directly (this includes caddy-corner borders for checking edge refinement quality, as in Figure 3.7). Thus, a ghost voxel only knows as much about its real self as is necessary
Figure 3.5: Smooth progression of voxel spacing in a hybrid mesh of the Louisiana coast – Here, the smooth progression of spacing can be seen from the large scale far-field mesh in the depths of the Gulf of Mexico to the small scale mesh around the coast of Louisiana.
Figure 3.6: Smooth progression of voxel spacing in both a Delaunay mesh and a hybrid mesh of a NACA 0012 airfoil – The smooth progression of spacing can be seen from the large scale far-field mesh in the free-stream away from the NACA 0012 airfoil to the small scale mesh around the edges of the airfoil itself. (a) The progression of both quadrilateral and triangular elements (to rectify hanging nodes) in a hybrid mesh. (b) The progression of triangular elements in a Delaunay mesh, where most triangles are simple subdivisions of the original quadrilateral voxels along their diagonals.
to determine mesh quality and rename nodes. This comes into play when there is refinement on both sides of a processor border, and a mid-edge node is created on both. In this case, processors communicate to determine if both have a common node, and if so, it is reassigned ownership of the lower process, as well as the index on that processor. It retains the local index on the non-owning process for computations in that scope.

Occasionally, the need for refinement locally is not extant, so this process also allows for refinement to happen on only one side of a processor border and the resulting hanging node to be properly indexed, even if the refinement occurred on the higher-indexed processor.

The Quadtree structure implemented here also speeds the refinement process greatly, since the tree can quickly be traversed to come to the most highly refined level, and then a simple check through the neighbor tables lets the program know if there is the need for refinement anywhere in the given area. If none is needed, no extra computations are done to any surrounding voxel, thus utilizing the tree to not only move quickly through the mesh but also to eliminate areas that no longer need refinement.

### 3.4 General Cutting

Once the hierarchical mesh has been generated, what remains to be done is to limit the nodes to the computational domain and generate appropriate cells along the geometry. Since the geometry cuts through voxels, the entire voxel is not located within the computational domain, and the remnants of the voxel inside the computational domain must be compiled and basic element types must be created from the leftover space. This is accomplished in the general cutting procedure. Figure 3.8 shows the hierarchical mesh that has been generated over the NACA 0012 airfoil before the nodes inside the airfoil have been turned off and the portions of the voxels external to the airfoil have been re-triangulated into valid cells. To see the results once this cutting and re-triangulation has occurred, refer to previous Figure 3.6.

In Karman’s three-dimensional HUGG code, the projection method is used instead of general cutting (see Section 1.2.4). At this point it seems prudent to mention that another inferior aspect of the projection cutting method is the manner in which it determines the...
Figure 3.7: **Ghost voxels within the scope of a given processor** – The voxels shaded in orange are in the upper left corner of a given processor. The voxels shaded in green are the finest level ghost voxels used in the neighbor tables on that processor. The voxels shaded in blue are the ghost parents of ghost voxels, but only contain the children directly bordering the processor in question.
Figure 3.8: **Uncut NACA 0012 airfoil** – The NACA 0012 airfoil above has had the proper mesh generated over it based on the spacing parameters supplied by the user and is ready to have nodes outside of the computational domain (inside the airfoil) turned off and the portions of voxels inside the computational domain re-triangulated.
in and out status of nodes. In general cutting, the closed loops are used not only to
generate boundary elements but also to determine the inside or outside status of each node
simultaneously, thus saving time in the flood fill routine. Karman’s method determined
the in or out status of nodes using an intersection test where the tree is traversed to find
the number of intersections with the geometry that occur between a given cell and a point
fully outside the domain. If the result is odd, then the cell is inside the computational
domain, and if it is even, then the cell is outside the computational domain [Karman,
2004]. While this determines inside and outside well for single block domains, it does not
have the robustness that utilizing the direction of the boundary segments does when dealing
with multi-block domains. Also, the closed loops can be preserved, reversed, and reused
while creating each block of the mesh, but the projection cutting process must be repeated
for each block.

General cutting is used not only to avoid the pitfalls of the projection method but also
to make the best use of the objects that are created by using the innate windings. The
basic principle of general cutting is to discover where the geometry intersects voxels and
create nodes in those locations as well as connectors, which will hereafter be called boundary
edges (b_edges) that follow the path of the geometry through these newly created nodes.
These connectors are then stored in their own array, and those within each voxel are made
reference to in that object’s bnd_lst list.

There are two major considerations that make this the most difficult part of the whole
meshing process. First, if the geometry cuts through an existing node (or within a given
tolerance of that node), a new node need not be created. Also, if a geometry segment cuts
through the existing edge of a voxel, or part of it, it must be snapped to that edge. Creating
the tolerance to determine when an edge is truly collinear with a geometry segment or an
intersection point is truly coincident with an existing node is quite a difficult proposition
indeed.

In this program, the tolerance is one one-thousandth of the length of a voxel edge on
the finest level. Through running several cases with a fine mesh spacing on the order of
$10^{-9}$, this particular value showed itself to be the best fit in two dimensions. This is due
to the fact that a smaller value would make the tolerance on the order of $\epsilon_{\text{machine}}$ and a larger value would allow points to float along the edges of the smallest voxels arbitrarily.

Another option for defining the physical coordinates that lessens the effect of tolerance discrepancies is to use integers as physical coordinates. This may be done after the mesh is generated using the voxel construct, thus alleviating the need for physical coordinates until the cutting procedure, by determining the size of the smallest voxel and deciding to subdivide it into 100 partitions on each edge, for example. This is now the base unit of measure; since all voxels are subdivided as squares, this unit may be easily extrapolated into all voxels, and integer physical coordinates are now defined [Thompson et al., 1999]. Figure 3.9 shows how this transition from the scale on the finest voxel may be extrapolated to the next level up recursively using the fact that each edge of the parent-sized voxel is exactly twice as long.

The positive effect this has is to automatically snap all points to an integer coordinate using simple rounding during the cutting stage. While this can be very helpful and may be used in future implementations, there is a limit to the number of 64-bit, floating-point representations of positive integers that can exist on each axis (negative integers are used elsewhere as placeholders). The largest integer that can be represented in double precision machine numbers before counting begins to skip every other integer is $2^{53}$. Thus, if the increment chosen on the smallest voxel was 100, the most levels that could be created would be $2^{53}/2$, which seems sufficiently large. If this becomes a point of contention, the interval could be shrunk; however, this coarsening results in similar errors to those experienced with floating point coordinates when the snapping of cut nodes is implemented.

The first part of general cutting is to progress to the bottom of the Quadtree and only cut the voxels on the finest levels, depending on location in the mesh. This is accomplished through a recursive function call with one geometry edge being sent down the tree until the finest voxel it intersects is reached.

Next, a check for collinearity of the geometry edge with the voxel edge is performed using the dot product of normalized vectors created from their endpoints as a check for parallelism. These vectors can be seen in Figure 3.10 as the blue edge and a black edge in between any two nodes, respectively.
Figure 3.9: **Using integer coordinates to alleviate tolerance issues** – Since the child voxel is exactly half the size of the parent voxel, there are exactly twice as many integer “spaces” in the parent voxel as there are in the child voxel.
Figure 3.10: Handling collinear geometry edges during general cutting – (a) A collinear geometry edge which results in a boundary edge with the lower point and the upper left voxel node as its endpoints. (b) A collinear geometry edge which results in two boundary edges with the lower point and the mid-edge node as two endpoints and the mid-edge node and the upper left voxel node as the other endpoints. (c) A collinear geometry edge which results in a boundary edge with the lower point and the upper left voxel node as its endpoints. (d) A collinear geometry edge which results in a boundary edge with the snapped to mid-edge node and the upper left voxel node as its endpoints. (e) A collinear geometry edge which results in a boundary edge that is identical to the original geometry edge.
If the segments are found to be parallel (a dot product of 1 or -1), then a check is initiated for whether or not the edges are coincident. This is accomplished by determining if the normal vector to the geometry edge is orthogonal (within a tolerance) to the vector created by the tails of the geometry edge vector and the voxel edge vector. Essentially, this is checking that a vector between two points on truly coincident segments is not at an angle to a normal emanating from one of them.

The final placement of the boundary edge endpoints is decided by using the parametric coordinates obtained by perturbing the tail of the voxel edge vector as seen in Equation 3.1 [Thompson et al., 1999].

\[ \vec{p} = \vec{v}_1 + \alpha (\vec{v}_2 - \vec{v}_1) \] (3.1)

In Equation 3.1, vectors are used to represent coordinate pairs, and \( \vec{p} \) is the boundary segment point generated from the cutting, \( \vec{v}_1 \) and \( \vec{v}_2 \) are the original voxel edge nodes, and \( \alpha \) is the relative distance from \( \vec{p} \) to \( \vec{v}_1 \).

In Equation 3.1, there is an \( \alpha \) for each endpoint of the boundary edge. \( \alpha \) is the dot product of vectors constructed from the tail of the voxel edge to the head and tail of the geometry edge. These are then dotted with the normalized voxel edge vector to determine a fractional relative distance between the tail of the voxel edge and the endpoints of the geometry edge, and the result is normalized with the length of the voxel edge vector.

If both of the resulting dot products are below zero or above one, the edge is strictly below or above, respectively, the edge in question. This is often the case if a voxel edge has a mid-edge node and the geometry is actually coincident with the other half of the edge (see Figures 3.10(c) and 3.10(e)).

However, if the relative distance for one geometry edge endpoint is between 0 and 1, this lies on the voxel edge and becomes a new endpoint for the boundary edge at the location given by Equation 3.1. If \( \alpha \) is above one or below zero, \( \alpha \) becomes one or zero, respectively, and the endpoint is snapped to head or the tail, respectively, of the original voxel edge vector. This is practiced both in the situation of a geometry edge extending beyond a current voxel.
edge node (Figures 3.10(a), 3.10(b), 3.10(c)) or in the case of the endpoint being within the
given tolerance of the location of a current voxel edge node (Figure 3.10(d)).

If a geometry edge is not collinear, a test for intersection is conducted. If it is determined
to intersect a voxel edge, both endpoints are added to the voxel’s \( c_n \) list and the edge is
added to the voxel’s \( bnd_lst \), whether one or both points intersect the edge. If a geometry
edge is found to be wholly inside the voxel, the same additions are made. These situations
can be seen in Figure 3.11.

Once an edge has been added, an attempt is made to merge the edge with any other
geometry edge which has a common endpoint and is on the same geometric boundary.
This is time consuming since each boundary edge within a given voxel must be tested each
time; however, it saves a great deal of memory by not storing extra segments. The new
boundary segment is added to the older boundary segment, the shared node is removed
from the voxel’s \( c_n \) list and the two boundary edges now only take up one slot in both the
overall \( b_edges \) list and the voxel’s \( bnd_lst \) list. Figure 3.12 shows part of a NACA 0012
airfoil with the boundary edges not coalesced as well as what the same boundary segments
look like after being combined. Notice that while the portions of the geometry have been
straightened out and thus some of the airfoil’s curvature is lost, this can be improved (if
necessary) by simply refining the mesh more until each of the original boundary segments
is contained within a single voxel.

One nice feature of this combining boundary edges is that it straightens out the segments
that must be used to form the new cut cell boundaries, which is discussed next, and thus
makes the closed loop creation faster. It also requires the user to set a spacing parameter
that gives the necessary resolution (or lack thereof) and allows a great deal of control in
mesh size. Also, the need to remap the nodes due to the combining of segments as well
as remap the \( b_edges \) due to the occasional combining of two pre-created \( b_edges \) is not
relevant until after cutting, and thus it may be done only once as opposed to continuously
as \( b_edges \) are created.
Figure 3.11: Handling intersecting geometry edges during general cutting – (a) A geometry edge, in red, intersects a voxel in two places, seen in blue. The segment between the two blue nodes is stored in the boundary list of the voxel as a pair of nodes. (b) A geometry edge, in red, intersects a voxel in one place, seen in blue. The segment between the blue node and the internal red node is stored in the boundary list of the voxel as a pair of nodes, assuming it is a separate boundary and is not coalesced with another geometry segment. (c) A geometry edge, in red, is completely contained within a voxel. The segment is stored in the boundary list of the voxel as a pair of nodes, assuming it is a separate boundary and is not coalesced with another geometry segment.
Figure 3.12: **Coalescing boundary edges** – (a) Multiple boundary edges can be seen within one voxel, which allows for a refined geometry regardless of the refinement of the mesh. (b) All boundary edges that are on the same geometric boundary and in the same voxel have been coalesced, which requires the user to define an appropriately fine mesh to get the resolution desired.
3.5 Boundary Element Creation

Once a voxel is cut, it has the agglomerated boundary edges added to its `bnd_lst` list and the nodes of which they are comprised added to its `c_n` list. Using this information, the cut cells are obtained, and thereby which nodes are in the computational domain and which are outside of it can be determined.

What needs to be done now is twofold. First, a `List` must be created for each loop contained in each voxel to hold the node numbers corresponding to a path around what will be the outside of the cut cell. Additionally, a map array of all the nodes on a given processor is created such that as we add nodes to the path they are mapped to positive one (inside the computational domain). Any node that is not part of a given path will be marked as negative one, or out, and though it is not deleted from memory, it will be turned off and not included in the remapped node numbers.

A novel approach is used in determining the boundaries of the cut cell. The cut voxel seen in Figure 3.13(a) has three distinct boundary edges, since the two leftmost boundary segments are on different boundaries and are not agglomerated. A temporary list of all these edges is created, along with a list of the edges that comprise the voxel itself, as can be seen by the numbered segments in Figure 3.13(b). The voxel has essentially been decomposed into two lists of segments, all of which are oriented such that the inside of the computational domain is to their left. Now, beginning with a randomly chosen boundary edge (here boundary edge 1 is chosen as an example), an attempt is made to find another edge with the same node number at its tail as is at the head of the previous edge. The algorithm first looks to find another boundary edge with the proper node, and failing that, it looks to continue the loop with either segment 7 or segment 8. Since the segments have been oriented such that the inside of the computational domain is toward their left, the only segment with the proper node number as its tail is segment 7, and it is added to the closed loop `List` as two ordered node numbers, and both of these are marked as in. Again, the algorithm will look to find a boundary edge which has a tail that matches the head of the previous segment. Failing that, it will find segment 6, and add it to the list as well as mark its nodes as inside. Now, the fact that it looks to boundary edges first comes into play...
Figure 3.13: **Creating a boundary element from a cut voxel** – (a) Cut voxel, with boundary edges shown in blue. (b) Edges have been separated and numbered so that a path may be chosen around the cut cell. (c) Boundary element (shaded) with boundary segment winding such that the inside is to the left.
with segment 5, since it attempts to connect to it instead of segment 11. This algorithm continues until the original tail from segment 1 is reached.

One important facet of this algorithm is that if no loop can be created with a boundary edge, it is ignored and discarded from the given voxel’s `bnd_lst` list. This situation occurs when a boundary edge is coincident with an edge joining two voxels; in this case, only one voxel will need it to create a closed loop. Instead of deleting these boundary edges at a later time or later splitting up separable loops which traveled through the same vertex multiple times and causing excess computational time and memory to be wasted, these loops are placed in separate arrays from the outset, which allows for any geometric configuration within a voxel to be handled and separately either broken down into triangles or stored as the resulting triangle.

The `Linked_List` class could be used to avoid the need that arises in the `List` class to create pre-allocated, pre-sized lists that grow in chunks as well as have a simple method to move through the closed loops, since only the nodes’ order needs to be preserved. Since the closed loops can be of varying lengths and the goal is to be completely scalable, this saving of memory seems prudent. However, the fact that the `List` class allows for actual indices on the elements of the loop makes searching through loops for subloops possible. These broken down subloops make boundary element creation more general and thus have proven to be more useful than the memory saved. Namely, if a non-endpoint of a closed loop occurs more than twice, the list can be split up further into sub closed loops (see Figure 3.14) and easily reassigned from within the `List` class. This also eliminates the need to work through all boundary edges and discard lone segments at a later time.

Two other methods for creating boundary elements are discussed in the “Handbook of Grid Generation” [Thompson et al., 1999]. The first is to use binary face codes (based on which voxel each boundary element node is in) to make the determination if a node is in or out and create a list of in nodes that later must be cycled through to create boundary element lists. While this is a simpler method, it does not incorporate the windings and combining of boundary elements on the fly that the closed loop allows for, and thus causes extra work to be performed later instead of simultaneously. The second option is polygon clipping, which is very similar to the closed loop creation implemented in `P_HUGG2D` in that
Figure 3.14: **Splitting a closed loop into sub closed loops** – (a) A closed loop that begins at the green dot, but actually needs to be split into two cut cells, since the crimp occurs at a voxel edge node. (b) Detecting that the shared node is on a voxel edge (since it will occur twice in the list but not as a loop endpoint) and splitting the closed loop into two closed loops.
each side of a voxel is cycled through, intersections are checked, and edges are created. These are simultaneously strung into a list of boundary element nodes in the same manner that is implemented in P_HUG2D. The only detraction from this method is that it does not combine boundary elements on the fly; however, since the initialization of the algorithm is so similar, it could easily be changed to do so. This method even uses the same parametric coordinates (Equation 3.1) to determine the final placement of boundary segment nodes based on their location on the edge of the original voxel [Thompson et al., 1999].

Now that all the interior nodes are marked in on each cut voxel and the remaining nodes are marked out on a voxel-by-voxel basis, all that remains is to flood fill the voxels that do not intersect the geometry. Two additional complications are added in parallel. First, since processors that do not contain any geometry can exist, these do not have any basis for flood filling. Second, there are ghost nodes that must be ignored in the determinations of in or out, since they may exist in a cut voxel without having the benefit of having gone through the cutting process or having the boundary edges in their structure and thus could give errant results.

The flood fill process loops through the uncut voxels on a given processor. If all the nodes are marked as zero, the unknown state, the voxel is skipped. If at least two nodes are known to be inside the computational domain (based on their being shared with a cut neighbor), each node in the entire voxel is mapped to the inside state. If any node is marked out, then each node in the voxel is marked outside the computational domain. This is done recursively until all voxels are fully marked, and then a communication is initiated to determine if any processor has not been able to initiate this process. If so, the neighboring processors are polled, boundary nodes are exchanged, and the processor that is unfilled uses the results to determine its status and map all of its nodes. It is important to note that this node marking is not intended to finalize the state of the voxels themselves (due to the situation seen in Figure 3.15), but rather to have a simple, quick method of marking the nodes themselves through their relationships with each voxel.

Upon completion of node mapping, each voxel is tagged as inside, outside, cut, ghost, or parent for easier access later in the program. This is done by filtering out ghost, cut, and parent voxels by looking at $p_i$ pairs, bnd_lst lists, and kids, respectively, since they are...
not needed in the final mesh. Then, the inside and outside voxels are determined by testing
the nodes in the uncut voxel, and if any is marked as outside, the voxel is outside. This is
requisite since a boundary edge could pass through the corner node of a voxel, causing this
node to be mapped as inside, and yet the voxel itself is outside the computational domain.
Additionally, since the case seen in Figure 3.15 can occur if the refinement is not set fine
enough by the user, a check is run on all cells upon the full completion of cell creation to
delete any cell that has edges not shared by a neighbor or defined as boundary edges. This is
important since in Figure 3.15, the cell would be marked inside (without the aforementioned
check) due to all its nodes lying on boundary edges, yet the cell itself has no neighbors and
is not in the computational domain.

3.6 Final Mesh Creation

In order for this mesh generation to be useful in determining CFD solutions, a method of
conveyance is created such that a flow solver can read in the mesh and perform calculations
on the given domain. To view the format of the mesh file that P_HUGG2D generates, see
Section 2.5.

This capability to generate a mesh and either output parallel or global mesh files is
integral in moving the mesh generation field in the direction of automatic mesh creation.
For instance, in terms of adaptive meshing, one creates the initial mesh, smooths it, runs
it through a flow solver, and smooths, refines, and/or coarsens it again in order to produce
a better solution in the next iteration of the aforementioned cycle. The only method of
communication between the mesh generator, the optimizer, and the flow solver is the global
mesh file. The flowfield values needed for the adaptive optimizer to compute gradients are
added to the end of the current mesh file format as rows representing the values of the
primitive variables at each node in the order the nodes are listed at the beginning of the
mesh file.

Another important note about the mesh files that are created is that there is no implicit
order in which the boundary segments must be given, other than the convention that for the
given piece of the boundary (usually numbered reflecting the original boundary numbering
Figure 3.15: **Improperly marked voxel requires neighbor checking** – In the above figure, the dark grey signifies parts of cut voxels inside the computational domain and the light grey signifies parts of cut voxels outside the computational domain. The voxel which is striped in light and dark grey would be marked improperly initially, since all of its nodes are on boundary edges and thus marked inside; however, after finding that there are no cells which share its edges and none of its edges are boundary edges, it would be marked outside the computational domain.
in the geometry) all the segments must be oriented such that if they are treated as vectors, the cross product of a vector created from the tail of the boundary segment to a nearby node on the inside of a convex computational domain must be positive. In other words, the inside of the computational domain is on the left of the boundary edge. Also, neither nodes nor elements have an implicit order, other than the fact that the indices on the nodes must correspond to the proper indices in each element and boundary segment list.

3.6.1 Delaunay Output Mesh

At this point, a collection of boundary edges as well as a list of internal nodes for each processor exists locally. Before generating the final mesh, all the internal nodes in the computational domain are remapped and this mapping used to update the node numbers on the boundary segments. The nodes are remapped to new node numbers since the Delaunay mesher requires an array of nodes which must be contiguous. Since the nodes being sent already exist in an array that also includes other nodes outside the computational domain, a simple map array is created, dimensioned to the original number of nodes, and initialized to −1. Then, each node that falls inside the computational domain is remapped to 1, and upon completion of the cutting process, a loop is started at 0 and the new nodes are indexed incrementally until all nodes in the computational domain have node numbers greater than or equal to zero. This is extremely important so that the boundary segments can be mapped to the new numbering in the contiguous array to be sent to the Delaunay mesher, and yet the information about the physical coordinates can be called up based on the pre-cutting node numbering scheme (since the nodes are not deleted, but instead simply turned off).

Having remapped all the nodes inside the computational domain to new node numbers, a complete set of inputs for a Delaunay mesher is available, assuming that a parallel communication is completed wherein all nodes and boundary segments are remapped to a global set of node numbers (thus, the nodes on the processor boundaries have only one name and the nodes are numbered sequentially beginning with those on processor zero). While this option exists in the current code, it is not necessarily the best option. First of all, if a mesh is large enough to require a parallel generation to be carried out in the first place, it is likely that sending the full set of boundary segments and nodes to a Delaunay mesh
routine would cause the routine not only to run slowly but also possibly run out of physical memory or, in the 32-bit case, necessitate the allocation of over 2 GB of data at one time. For instance, each voxel object requires at least 12 words of memory. Thus, if one mesh has approximately 167 million elements the allocation of just the voxel objects would fail, not to mention the 1 GB per processor memory limitations of the machines on the SimCenter cluster, bigfrog. Considering that each voxel that makes up the tree must be stored, this number is not unthinkable, since a mesh with even 100 million elements at the finest level may have enough parent and grandparent elements that this threshold is exceeded.

Thus, the Delaunay mesher is run on a processor-by-processor basis. This requires that the routine receive not only the boundary segments from the cutting and the inside nodes but also the processor boundaries to create a watertight geometry. Using the ghost cells as harbingers of being on a processor boundary and their \( p_i \) pair to aid in separating the boundaries into one per processor interface, it can be determined which processors border the processor in question. Then, the geometric boundary segments are strung together into one portion of the boundary and the processor boundaries are defined as boundary segments to close the loop. If there is no geometry on a processor, the loop then consists of only processor boundaries.

Finally, the routine outputs a triangle connectivity array, its nodes are remapped to global numbering, and the mesh file is generated. Since there is only one type of element in this situation, there is no need to take the current connectivity and create \texttt{CELL} objects from it, since that only serves to waste memory by duplicating a connectivity that is already in place. Once this is completed and the Delaunay mesher is run, it only remains to create a global mesh file by remapping all the nodes to global node numbers, updating the boundary segments, and using interprocess communication to avoid duplicate nodes and write out the mesh file.

3.6.2 Hybrid Output Mesh

In the case of a hybrid mesh, the need to create interprocessor boundaries is alleviated by the fact that our final mesh will consist of quadrilaterals and triangles, and the majority of the mesh is already in that format. Also, the geometry is already approximated to the
given level of resolution as boundary edges. For those voxels that are already squares and have no hanging nodes, it becomes as simple as storing these nodes in a cell object, the details of which can be found in Chapter 2.3. Additionally, if a cut voxel is a triangle, it can also be immediately stored.

The problem of finalizing the mesh lies in two types of voxels: cut voxels with more than three sides, and voxels with hanging nodes. These must be given their own individual Delaunay triangulation, and the resulting triangles are then stored as cell objects [Karman, 2004]. Even though some of the cut voxels may be quadrilaterals and thus acceptable in a hybrid mesh, in the interest of not having concave elements, they are triangulated.

Initially, the edges of a voxel with hanging nodes are stored as a temporary boundary array and the nodes from that voxel are also remapped to a connectivity for just that voxel. This is simple since the nodes are already numbered assuming the winding of inside to the left. Once the sides that actually have hanging nodes are determined, the edges are connected up through a switch, the resulting remapped nodes and boundary segments are sent to the Delaunay mesher, and the resulting triangles are stored as CELL objects of type 3 (TRI2D).

Due to the method used to create the cut voxels and store their boundary segments, the List that stores the loops in each cut voxel can be accessed to provide not only the number of segments (for allocation purposes) but also the order in which they are processed for the Delaunay mesher. Again, the same remapping of the nodes to a numbering scheme local to the voxel is used, and the results of the Delaunay mesher are stored as CELL objects of type 3 (TRI2D). The exception to this is when a cut cell is left with only three sides. In this case it can be immediately stored as a CELL object of type 3 (TRI2D).

Finally, the remaining inside, uncut voxels are stored as CELL objects of type 4 (QUAD2D), which makes testing for which category (quadrilaterals or triangles) the element fits into simplified and fast. Now, it only remains to create a global mesh file by remapping all the nodes to global node numbers, updating the boundary segments’ indices, sorting the cells by type, and using interprocess communication to avoid duplicate nodes and write out the mesh file.
3.7 Optimization-based Smoothing

Once a mesh has been generated, often sliver cells, or cells with very high aspect ratios and very small areas, are generated through the general cutting process. These are not optimal for use with most flow solvers, so the use of an optimization-based smoother is necessary to stretch these cells, giving them more reasonable areas and lower aspect ratios [Sahasrabudhe, 2006]. This routine determines whether or not to move a node based on a cost function, $C$, derived from the areas of the triangles which surround it. The cost function for each individual triangle is actually a conditional, as can be seen in Equation 3.2.

\[
C_t = \begin{cases} 
1 - J, & \text{for } J < 0 \\
\left(1 - \frac{1}{CN}\right)^2, & \text{for } J \geq 0 
\end{cases}
\]

(3.2)

In the above equation, $J$ stands for the Jacobian of an element, which is negative if the element is inverted, and it is simply the cross product of two vectors emanating from one vertex and having as their heads each of the other two triangle vertices. The reason for having a separate cost function for negative Jacobians is that these elements must have priority in movement since a final product with inverted elements is highly undesirable.

In the second piece of the above equation, $CN$ stands for the weighted condition number of the triangle, the make up of which can be seen in Equation 3.3. Notice that this is not the area-weighted condition number, since a negative area (inverted element) causes the previously negative condition number to become positive, thus negating the effect of growing the cost function and possibly causing movement that is undesirable.

\[
CN = \frac{||AW^{-1}|| \cdot ||WA^{-1}||}{2}
\]

(3.3)

where $||\cdot||$ is the Froebenius norm, $A = \begin{bmatrix} e_{1x} & e_{2x} \\
e_{1y} & e_{2y} \end{bmatrix}$, $e_{ix}$, $e_{iy}$ are the edge vectors emanating from a corner on the given triangle, and $W = \begin{bmatrix} 1 & \frac{1}{2} \\
0 & \frac{\sqrt{3}}{2} \end{bmatrix}$, which are the scaling factors for an ideal triangle.
Once the cost functions for each triangle surrounding the node are completed, the cost function for the node is computed using piecewise Formula 3.4 and the maximum and average cost functions from triangles surrounding the node itself.

\begin{equation}
\begin{aligned}
C_n &= (1 - J)^2, \text{ for } c_{\text{max}} > 1 \\
C_n &= \text{blend} \cdot c_{\text{max}} + (1 - \text{blend}) \cdot c_{\text{avg}}, \text{ for } c_{\text{max}} \leq 1
\end{aligned}
\end{equation}

(3.4)

where \(c_{\text{max}}\) is the maximum cost, \(c_{\text{avg}}\) is the average cost, and \(\text{blend} = \frac{\tanh(5 \cdot (c_{\text{max}} - 1)) + 1}{2}\).

There are three types of movement around a node that are utilized in these perturbations (the cost is analyzed between each type of movement). The first is to move the node in the direction of the unit normal of its opposite edge, the second is to move the node along each of the edges in which it is included, and the third is to move the node a prescribed perturbation in both the x and y directions if neither of the previous methods improves the cost function. If the movement of the node has improved (lowered) the nodal cost function from the last iteration, the movement is accepted; if not, the movement is rejected.

In Figures 3.16 and 3.17, both a hybrid mesh and a fully triangular mesh can be viewed in their pre-optimization and post-optimization states. While the mesh is no longer guaranteed to be a true Delaunay triangulation after optimization-based smoothing is applied, this does not denigrate the quality of the mesh; in fact, all of the relevant metrics (aspect ratio, skewness, condition number, etc.) have been improved dramatically. Notice that the sliver cells along the leading and trailing edges of the airfoil have been smoothed out, thus creating a more stable mesh without poorly formed elements. While the mesh connectivity has not changed, the actual volumes of the cells have, making it more conducive to flow solver calculations. Currently, the optimizer only runs in serial, so it must have a global mesh file; one future goal is to have the optimizer running in parallel as well, which prevents issues with having insufficient memory from which to allocate for the incoming mesh when a very large mesh (i.e., the Gulf of Mexico) is needing optimization.
Figure 3.16: Unsmoothed meshes of a NACA 0012 airfoil – (a) A hybrid mesh of a NACA 0012 airfoil before optimization-based smoothing. Notice the sliver cells in the inset image. (b) A fully triangular mesh of a NACA 0012 before optimization-based smoothing. Notice the sliver cells in the inset image.
Figure 3.17: **Optimization-based smoothed meshes of a NACA 0012 airfoil** – It is important to notice how the spacing near the leading edge of the airfoil has changed from Figure 3.16, due to the expansion of the sliver cells into valid elements. This is best seen by viewing the inset figure and comparing with the inset figures in Figure 3.16. (a) A hybrid mesh of a NACA 0012 airfoil from the HUGG2D optimization-based smoother. (b) A fully triangular mesh of a NACA 0012 airfoil from the HUGG2D optimization-based smoother.
Chapter 4

Experimental Results

4.1 NACA 0012 Airfoil Results

The NACA 0012 airfoil was used predominantly as a test case, although the optimized mesh is being extruded and used for further research on the applicability of P_HUGG2D. Two distinct versions of the same airfoil are used to test the robustness of P_HUGG2D and prepare for the difficult Gulf of Mexico geometry. The difference between these two versions is seen in Figure 4.1, where the light blue lines represent a four processor partitioning of the domain. The airfoil is shifted upward from its central position in the domain to explore the differences in communication needs when node creation, refinement, cutting, and coalescing is not performed in the exact same fashion on either side of a partition.

The first case, seen in Figures 4.2 and 4.3, is the simplest case, wherein the geometry (and thus the procedure needed to cut and recreate boundary elements) is symmetric about each half of the processors used. The second case, seen in Figures 4.4 and 4.5, where the airfoil is shifted up in the same outer boundary, is used to assure that the same refinement, cutting, and boundary element creation can be performed when the airfoil is limited to two processors instead of four and the interface is asymmetric. Additionally, the manner in which the flood filling and cell creation is completed for processors without geometry is honed and the proper method of communication is developed with this case.

Most importantly, Figures 4.2 and 4.3 show that the same mesh is generated regardless of how many processors are used, thus proving that the parallel implementation loses nothing
Figure 4.1: **Shifting a NACA 0012 airfoil from the center of the domain** – The blue lines represent a partitioning of the domain with four processors. The airfoil is shifted upwards by one unit in order to demonstrate the differences in communication needs for a non-symmetric partitioning of the main geometry.
in not only the extra communication and different naming conventions but also the use of general cutting and hybridization.
Figure 4.2: **Symmetric NACA 0012 airfoil (Delaunay)** – The one processor Delaunay case of the symmetric NACA 0012 airfoil. This Delaunay mesh on a symmetric airfoil is the simple case, where the airfoil is symmetric about the processor division. Notice that the mesh remains the same regardless of processor count, which implies that the communication is appropriate.
Figure 4.2 (continued). The two processor Delaunay case of the symmetric NACA 0012 airfoil.

Figure 4.2 (continued). The four processor Delaunay case of the symmetric NACA 0012 airfoil.
Figure 4.2 (continued). The eight processor Delaunay case of the symmetric NACA 0012 airfoil.

Figure 4.2 (continued). The sixteen processor Delaunay case of the symmetric NACA 0012 airfoil.
Figure 4.3: **Symmetric NACA 0012 airfoil (hybrid)** – The one processor hybrid case of the symmetric NACA 0012 airfoil. This hybrid mesh on a symmetric airfoil is the simple case, where the airfoil is symmetric about the processor division. Notice that the mesh remains the same regardless of processor count, which implies that the communication is appropriate.
Figure 4.3 (continued). The two processor hybrid case of the symmetric NACA 0012 airfoil.

Figure 4.3 (continued). The four processor hybrid case of the symmetric NACA 0012 airfoil.
Figure 4.3 (continued). The eight processor hybrid case of the symmetric NACA 0012 airfoil.

Figure 4.3 (continued). The sixteen processor hybrid case of the symmetric NACA 0012 airfoil.
Figure 4.4: **Shifted NACA 0012 airfoil (Delaunay)** – The one processor Delaunay case of the shifted NACA 0012 airfoil. This Delaunay mesh on a shifted airfoil shows the need for handling non-symmetric cases when communicating between processors. Notice that the mesh remains the same regardless of processor count, which implies that the communication is appropriate.
Figure 4.4 (continued). The two processor Delaunay case of the shifted NACA 0012 airfoil.

Figure 4.4 (continued). The four processor Delaunay case of the shifted NACA 0012 airfoil.
Figure 4.4 (continued). The eight processor Delaunay case of the shifted NACA 0012 airfoil.

Figure 4.4 (continued). The sixteen processor Delaunay case of the shifted NACA 0012 airfoil.
Figure 4.5: **Shifted NACA 0012 airfoil (hybrid)** – The one processor hybrid case of the shifted NACA0012 airfoil. This hybrid mesh on a shifted airfoil shows the need for handling non-symmetric cases when communicating between processors. Notice that the mesh remains the same regardless of processor count, which implies that the communication is appropriate.
Figure 4.5 (continued). The two processor hybrid case of the shifted NACA 0012 airfoil.

Figure 4.5 (continued). The four processor hybrid case of the shifted NACA 0012 airfoil.
Figure 4.5 (continued). The eight processor hybrid case of the shifted NACA 0012 airfoil.

Figure 4.5 (continued). The sixteen processor hybrid case of the shifted NACA 0012 airfoil.
4.2 Gulf of Mexico Results

The mesh generated for the Gulf of Mexico is a very difficult two-dimensional case. Due to the large number of geometric boundaries (134), fine spacing between geometry segments, and a single watertight mainland boundary accompanied by 133 other small internal watertight boundaries (islands), P_HUG2D now has been augmented with many extra features that the airfoil test case does not require.

This geometry was used to solve the Shallow Water Equations for the wetting and drying that currents cause in the Gulf of Mexico [Sivakumar, 2006]. Figures 4.6 and 4.7 show that the same mesh is generated regardless of how many processors are used, thus proving that the parallel implementation loses nothing in not only the extra communication and different naming conventions but also the use of general cutting and hybridization, except in the sixteen processor case, which is discussed next.

In both the hybrid and Delaunay cases run on sixteen processors, the difficulty arises from the fact that while mesh quality is enforced at processor boundaries, the cell size gradation parameter does not take effect on processors with no geometry, due to its implementation as a factor of the geometric spacing. Thus, at certain internal processor borders, there is one less level of refinement, which is not extremely problematic due to its location in the mesh.
Figure 4.6: **Gulf of Mexico (Delaunay)** – The one processor case of the fully triangular Gulf of Mexico. This Delaunay mesh of the Gulf of Mexico puts cutting to the test with all the islands and small scale geometry. Notice that the mesh remains the same regardless of processor count, which implies that the communication is appropriate, except in the sixteen processor case, which is discussed in Section 4.2.
Figure 4.6 (continued). The two processor case of the fully triangular Gulf of Mexico.

Figure 4.6 (continued). The four processor case of the fully triangular Gulf of Mexico.
Figure 4.6 (continued). The sixteen processor case of the fully triangular Gulf of Mexico. Due to load balancing issues, only 14 of the 16 processors are used beyond cutting, since processors 4 and 5 are outside the computational domain.
Figure 4.7: **Gulf of Mexico (Hybrid)** – The one processor case of the hybrid Gulf of Mexico. This hybrid mesh of the Gulf of Mexico puts cutting to the test with all the islands and small scale geometry. Notice that the mesh remains the same regardless of processor count, which implies that the communication is appropriate, except in the sixteen processor case where the per processor spacing parameters give one less level of refinement in certain spots toward the middle of the computational domain, which is desirable.
Figure 4.7 (continued). The two processor case of the hybrid Gulf of Mexico.

Figure 4.7 (continued). The four processor case of the hybrid Gulf of Mexico.
Figure 4.7 (continued). The sixteen processor case of the hybrid Gulf of Mexico. Due to load balancing issues, only 14 of the 16 processors are used beyond cutting, since processors 4 and 5 are outside the computational domain.
4.3 3-D Extrusion Meshes

Another use for the current two dimensional implementation is to take cross sections of complex three-dimensional geometries, generate a two-dimensional mesh around them, and use a utility developed by Karman to extrude them into three-dimensions. Essentially, the given layer is reproduced two more times, connectors are generated between identical nodes on each layer mesh, and a thin, flat, three-dimensional mesh is the result. Examples of a coarse and fine mesh around x and y cross-sectional slices of the Sea Fighter geometry and their three-dimensional versions can be seen in Figures 4.8, 4.9, 4.10 and 4.11. Notice in the fine and coarse mesh that the spacing near the geometry is nearly identical due to the fact that similar parameters are used to force the minimum spacing. However, in the far field, the parameters used to force the maximum spacing differed by an order of magnitude, and thus if fewer elements are desired in these low-interest areas, P_HUGG2D makes this attainable. It is also interesting to note that several very fine versions of these meshes were too large to run in serial and thus the parallel implementation became necessary.

Additionally, in Figure 4.12, a NACA 0012 airfoil with a 9 degree angle of attack is meshed for use in Dr. Robert Wilson’s published research [Wilson et al., 2007]. This mesh is particularly difficult because the angle of rotation caused the flood fill issues discussed in Section 3.5. Using this mesh, the desired fluid solution results are obtained, and further work of this type is planned.

4.4 Parallel Speedup

Figure 4.13 shows the configuration of processors for the four, eight, and sixteen processor NACA 0012 airfoil cases, both hybrid and Delaunay. The processors are color-coded such that the interfaces can be plainly detected and the scenarios to be discussed in this section regarding disjoint areas being shared by the same processor can be more clearly visualized.

The ideal goal of parallelization is to utilize each processor such that it is computing something at all times and idle time is kept to a minimum. This perfect load balancing guarantees nearly linear speedup as more processors are applied to the same large problem. While the goal of using each processor equally is in sight, the current implementation simply
Figure 4.8: **Coarse mesh on an x-cross-sectional slice of a 3-D Sea Fighter geometry and extrusion into 3-D** – (a) An optimization-based smoothed coarse mesh on a x-cross-sectional slice of the 3-D Sea Fighter geometry. (b) A three-layer extrusion of the mesh seen in (a) which allows 3-D analysis on a section of the Sea Fighter.
Figure 4.9: **Fine mesh on an x-cross-sectional slice of a 3-D Sea Fighter geometry and extrusion into 3-D** – (a) An optimization-based smoothed fine mesh on a x-cross-sectional slice of the 3-D Sea Fighter geometry. (b) A three-layer extrusion of the mesh seen in (a) which allows 3-D analysis on a section of the Sea Fighter.
Figure 4.10: Coarse mesh on a y-cross-sectional slice of a 3-D Sea Fighter geometry and extrusion into 3-D – (a) An optimization-based smoothed coarse mesh on a y-cross-sectional slice of the 3-D Sea Fighter geometry. (b) A three-layer extrusion of the mesh seen in (a) which allows 3-D analysis on a section of the Sea Fighter.
Figure 4.11: **Fine mesh on a y-cross-sectional slice of a 3-D Sea Fighter geometry and extrusion into 3-D** – (a) An optimization-based smoothed fine mesh on a y-cross-sectional slice of the 3-D Sea Fighter geometry. (b) A three-layer extrusion of the mesh seen in (a) which allows 3-D analysis on a section of the Sea Fighter.
Figure 4.12: **Mesh on a NACA 0012 airfoil with a 9 degree angle of attack** – (a) An optimization-based smoothed fine mesh on a NACA 0012 Airfoil with a 9 degree angle of attack, viewed from the top. (b) A three-layer extrusion of the mesh seen in (a) which allows 3-D analysis on the airfoil, where the layers can be seen through the “hole” in the domain.
Figure 4.13: **Processor configurations for four, eight, and sixteen processors (Delaunay case)** – (a) Processor configuration for four processors. Notice that each has a portion of the airfoil and the outer boundary. (b) Processor configuration for eight processors. Notice that these are assigned round robin, since the number of voxels at spawn must be a power of four. Also, the load balancing is flawed since some have more geometry than others (since some of the disjoint partitions have no geometry paired with only a small amount of geometry from the outer boundary). (c) Processor configuration for sixteen processors. Notice that the load balancing is flawed since some have more geometry than others (and some have no geometry).
partitions based on a “round-robin” scheme, wherein each spawnable voxel is assigned in a counterclockwise fashion from the lowest indexed processor to the highest. When the number of available processors is exhausted and there are still more voxels, they are simply assigned to the lowest numbered processor and on down the list again. This is also done with no consideration for how much geometry may or may not be in a given voxel and often allows a processor to be working on completely disjoint sections of the mesh. This lack of consideration causes a need for excess communication (also something that must be avoided in parallel programming, since it is even slower than excess computation), even for activities that are normally relegated to one processor, since disjoint areas on a mesh being owned by the same voxel greatly increase the surface area. Simply put, this means that while a processor without disjoint pieces of the mesh might have to routinely communicate with eight other processors, one with a disjoint subset of the mesh as its domain might have to communicate with up to sixteen other processors. However, this initial implementation is not designed with perfect load balancing in mind; rather, it is intended to be used as a tool to generate large meshes in a manner faster than serial processing allows and to determine the logic needed to handle the general case. Clearly, speed up was exhibited and the algorithm can be expected to perform with a very high efficiency given a better load balancing scheme.

While the code has the ability to deal with many of the issues created by disjoint areas being owned by a single processor (since this is a possibility even when using the equidistribution of geometry as the load balancing condition), it is still fallible in cases that are blatantly load imbalanced. Table 4.1 gives a better picture of why certain cases not only show sub-linear speedup but also why the eight processor case on the Gulf of Mexico does not run to its natural conclusion. Due to the fact that the eight processor case has so many combinations of disjoint areas on the same processors and processors with little or no geometry, processor five has no geometry and half is inside the computational domain and half is outside the computational domain.

When a given processor owns two disjoint areas of the mesh that include no geometry and are not both in the computational domain, this causes a problem since, in order to save excess communication, one node from each inter-processor border of a processor that
has no geometry is sent to the adjacent processor to check for flood fill status. The results are then tallied, and if some come back inside and others come back outside, the program deadlocks. This could be rectified by allowing the logic to send separate messages for each spawn level voxel and then flood fill based on that information, which currently seems like the slower but better option. Also, a full list of all inter-process boundary points could be sent to each adjacent processor (at the cost of extra communication) and then a traditional, slow flood fill could be performed on each processor without geometry. Lastly, a partitioning scheme where the mesh is refined until each partition contains an equal number of geometry segments could be implemented and may result in better load balancing.

The rationale behind including the “Outer Boundary Only” category in Table 4.1 under the two airfoil cases (this does not exist in the Gulf of Mexico case, where the geometry is the outer boundary) is that while these processors technically have geometry, it is the most trivial case and requires significantly less computational time to generate these “cut” cells since they are always inside the computational domain, are not re-triangulated in the hybrid case, and form at most one closed loop per voxel (the most trivial closed loop). While these are more balanced than processors with no geometry, in the future the outer boundary would likely be given to one processor due to the rapidity with which it moves through the general cutting procedure compared to that of a complex geometry.

It is also important to notice that the shifted airfoil case is created not to extol the virtues of the current load balancing scheme (the symmetric airfoil tests much better under the current load balancing system). Rather, it is used to provide insight into adjustments that need to be made to allow for geometry-free processors and more complicated closed loops due to the non-symmetric straddling of two processors created by dividing the airfoil vertically instead of horizontally and vertically as is done in the symmetric case.

By looking at columns four, five, six, and seven of Table 4.1, it becomes clearer why in Figures 4.14 and 4.15 the sixteen processor case not only does not exhibit speedup but actually regresses to taking even longer than the one processor case. The large amount of time fourteen (or twelve for the symmetric case) of the sixteen processors spend waiting on
Table 4.1: **Load balancing eccentricities using round-robin assignment at spawn** –

In the above table, the asterisk marks cases that do not run to conclusion due to serious load balancing errors. If a processor has disjoint areas of the mesh and one area has geometry while the other does not, it is not counted as having no geometry (N.G.). However, if part of a processor is in the computational domain and the other disjoint area is outside the computational domain (O.D.), it is counted as being outside the computational domain (since this is not a situation well handled by the code). If a processor has the airfoil geometry as well as outer boundary geometry, it is not counted as outer boundary only (O.B.O.).

On the other hand, the Gulf of Mexico runs do exhibit speedup (Figures 4.16 and 4.17) as a result of the parallelization; however, the simple round-robin load balancing makes the speedup far from linear. The scalability of the algorithm thus depends not only on the special methods of memory allocation for large cases and the special constructs used to make communication less intensive but also the initial load balancing scheme.
Figure 4.14: Parallel timing for generating a mesh on a NACA 0012 airfoil – The times taken to generate a mesh around both a symmetric and shifted NACA 0012 airfoil show clearly why the speed-up curve in Figure 4.15 is not only non-linear but also in fact turns back in on itself for the sixteen processor case.
Figure 4.15: **Parallel speed up for generating a mesh on a NACA 0012 airfoil** – The speed up gained by doing parallel mesh generation is not as noticeable on the airfoil due to not only the small nature of the problem but also the current lack of a good load balancing scheme for a problem with such a small geometry and a far-field boundary with such a large area. In fact, the speed up (albeit a small one) seen for 2, 4 and 8 processors is completely negated when 16 processors are used. This also varies between the hybrid and Delaunay cases and even more so between the shifted case and the symmetric case, due to the placement of the airfoil within the supercell and the lack of consideration for this in spawning to multiple processors.
Figure 4.16: **Parallel timing for generating a mesh on the Gulf of Mexico** – The times taken to generate a mesh around the Gulf of Mexico show clearly why the speed-up curve in Figure 4.17 is non-linear. However, it is also apparent that the addition of more processors did indeed speed the process.
While the speed up obtained using multiple processors is not completely linear, as would be desired from a perfectly scalable application, it is evident for both the hybrid mesh and the Delaunay mesh. This could be improved by a load balancing scheme that takes more of the geometry into account, since currently it is common to have processors with significantly less (or no) geometry as well as whole processors outside the computational domain.

Figure 4.17: Parallel speed up for generating a mesh on the Gulf of Mexico — While the speed up obtained using multiple processors is not completely linear, as would be desired from a perfectly scalable application, it is evident for both the hybrid mesh and the Delaunay mesh. This could be improved by a load balancing scheme that takes more of the geometry into account, since currently it is common to have processors with significantly less (or no) geometry as well as whole processors outside the computational domain.
Chapter 5

Conclusions and Future Work

5.1 Conclusions

In order to solve modern CFD problems, a robust parallel mesh generation process that uses the “C++” programming language and MPI to allow for complex geometries, rapid generation of meshes, and increased flexibility with spacing and mesh type is implemented in P_HUGG2D. The program allows users to generate large, fine meshes in two dimensions using any watertight geometry that is created by a CAD program such that the inside of the computational domain is to the left of the boundary segments or facets. This procedure uses Quadtree refinement of a Cartesian root cell until a desired set of spacing and mesh quality parameters is reached. Upon completion, general cutting is performed, allowing for the creation of closed loops that become cut cells directly. These can then either be assumed back into the mesh and yield a fully triangulated mesh, or they can be triangulated individually to create a collection of quadrilaterals and triangles for a hybrid mesh.

In order to implement this in parallel, communication between processors regarding node ownership and mesh quality is accomplished using ghost voxels, and results garnered on different sets of processors yield the same, or similar, meshes. Multiple cases test the robustness of the algorithm including a two-dimensional NACA 0012 airfoil and a two-dimensional mapping of the Gulf of Mexico. The meshes on the NACA 0012 airfoil and the Gulf of Mexico are all generated in a much faster manner than the serial implementation, and they still retain the spacing, resolution, and element-typing of the serial version.
Additionally, all these meshes, whether hybrid or fully Delaunay, can have viscous layers inserted very easily and can be optimized and adapted to work with any flow solver.

Thus, the parallel implementation of a hierarchical mesher is important in that it removes any limit that may have been placed on a geometry’s size due to the limitations in speed and memory on a serial machine, no matter how large. The addition of general cutting and closed loops being used to create boundary elements makes the mesher robust enough to handle even the most complex of geometries and possibly even the least ideal processor configuration.

5.2 Future Work and 3-D Implementation

While a two-dimensional mesh is not always directly useful in solving three-dimensional problems, relevant two-dimensional problems do exist (Section 4.2) and two dimensional meshes (such as the NACA 0012 airfoil) can be extruded into three-dimensions (see Section 4.3). More importantly, P_HUGG2D not only serves as a learning tool to help discover pitfalls before they became harder to find in the three-dimensional implementation being written by Karman (P_HUGG) but also contributes certain routines that are applied to faces in P_HUGG. Along with P_HUGG2D and its three-dimensional counterpart P_HUGG, an interface between Computer Aided Drafting (CAD) software and these meshing programs is being developed here at the SimCenter.

5.2.1 Multiblock Meshes

An additional area of interest is multi-block meshes. It is becoming more common to have multiple computational domains which contain different parameters or even different fluids. Quite often, simply determining flow solutions on one domain at a time is not only inefficient but also incorrect when interactions at the interfaces are occurring. One example of this phenomenon is work currently being undertaken at the University of Tennessee SimCenter at Chattanooga with fuel cells, where interactions between gases in channels require that the solution take into account information from two very different domains simultaneously,
Figure 5.1: **Multiple block meshes** – (a) A theoretical single block mesh for a gas channel in a fuel cell. (b) A theoretical multiple block mesh for the same gas channel, wherein the inner boundary has been reversed and a new computational domain has been created, allowing the user to perform solution calculations where interactions between boundaries occur (as is the case when two gases interact).

since the interactions at the interface preclude simple boundary conditions and affect the behavior of the fluid in the adjoining domain in real time.

Figure 5.1 shows an example of the usefulness of multi-block meshes. By iteratively changing the direction of each boundary, one can generate a mesh by using the same conventions as a single block mesh code that covers the obverse area from the original block, and thus creates a second computational block. This situation is allowed for in the current standard mesh file, and implementation simply requires a user to define how many blocks and which geometry pieces to invert to create the alternate computational domains.

### 5.2.2 Correcting Load-Balancing Issues

Also in the future, P_HUGG2D will implement a partitioning strategy that utilizes a cost function based on the geometry contained in each voxel to assure that each processor, while receiving different numbers of voxels, will receive the same amount of geometry. Since the general cutting and boundary element creation are the most computationally intensive part of generating the mesh, the amount of geometry cutting through voxels on each processor is directly proportional to how much computational time is used on each processor. This
redistribution of voxels at spawn creates a situation where each process spends nearly the same amount of time in each routine and thus idle at MPI_BARRIER and MPI_WAITALL statements is negligible.

The goal of attaining perfect load-balancing serves the purpose of not only optimizing the performance of the code but also alleviating certain problems with processors that have no geometry in their domains. One issue that load balancing could resolve is the lack of information shared with processors that do not have geometry about the cell size gradation parameter. Since the goal is to write general code that does not break under even the poorest load-balancing conditions, P_HUGG has been corrected by allowing the cell size gradation parameter loop to be run through regardless of the processor’s orientation to the geometry. Since in the future, the geometry may not be passed to each spawned processor, utilizing the geometry given to each processor to set the spacings on interior processors (even if the geometry segments are distant from the cells) is not the most conducive approach. Most importantly, the algorithm itself is sound, and a better load-balancing scheme is all that is necessary to improve performance. This will be a conglomeration of round robin and geometry segment based partitioning to find the middle ground between decreasing surface area and yet keeping idle, non-cutting processor domains to a minimum.

5.2.3 64-bit Version

The ability to use the 64-bit implementation of P_HUGG2D allows for more than 2 GB to be stored in one allocation, which can come in handy for very large meshes. The 32-bit implementation is used to generate the results for this thesis, due to current limitations with the 64-bit implementation of mpich2 on the bigfrog cluster at the University of Tennessee SimCenter at Chattanooga.
Bibliography
Bibliography


Appendix
Appendix

5.3 Memory Allocation Test

The following two routines, individualtest.cpp and chunktest.cpp, were used to determine the most efficient method for allocating voxel objects. It was determined that allocating in chunks of voxel objects was significantly faster than allocating for the objects individually, as needed. For more information, see Section 2.3.

5.3.1 Individual Allocation

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "Cell.h"
#include "Point.h"
#include "voxel_obj.h"

// run only as check to programmed version
// run with usr/bin/time exe
// this constructor determines what goes into cell_mem and CELL
Voxel_obj::Voxel_obj(Point p0, Point p1)
{
    printf("\nConstructing voxel mesh object.");
}
n_cells = 0;
cell=0;
cell_dim = 0;
Cell_mem(1);
// Generate root quad for this object
cell[0].init(-1);
root_lo = p0;
root_hi = p1;
n_cells = 1;
root_cell = 0;
}

// this is the corresponding destructor
Voxel_obj::~Voxel_obj()
{
    printf("\nDestructing voxel mesh object.");
    Cell_mem(-1);
}

// this allows us to actually allocate cell mem
void Voxel_obj::Cell_mem(int cell_size)
{
    if (cell_size > 0 && cell_size != cell_dim)
    {
        if (cell != 0)
            cell=(CELL*)realloc((void*)cell,cell_size*sizeof(CELL));
        else
            cell=(CELL*)malloc(cell_size*sizeof(CELL));
        if (cell == 0)
        {
            fprintf(stderr,"\nCell_mem(): Unable to reallocate memory for cells!");
        }
fprintf(stderr, "\n Current dimension -> %d", cell_dim);
fprintf(stderr, "\n Requested dimension -> %d", cell_size);
exit(0);
}
cell_dim = cell_size;
} else if (cell_size < 0)
{
    if (cell != 0)
        free(cell);
    cell = 0;
    n_cells = cell_dim = 0;
}
}
int main()
{
    // now, we wish to find start time of memory allocation
    time_t tm;
    char *t_char;
    // we also wish to store this in seconds from Jan 1, 1970
    double initial = time(&tm);
    t_char = ctime(&tm);
    printf("\nRun started at %s", t_char);
    // counter
    int c;
    // now, we allocate an array of cell pointers
    CELL **carray;
    carray = new CELL*[100000];
    if (carray < 0)
    {
        printf("\nMemory allocation failed.");
    }
exit(0);
}

// now, we allocate them individually
for (c = 0; c < 100000; c++)
{
    carray[c] = new CELL(); // since mother cells, automatically designated 0
}

// now, we wish to find end time of memory allocation
// we also wish to store this in seconds from Jan 1, 1970
double end = time(&tm);
t_char = ctime(&tm);
printf("\nRun completed at %s",t_char);
// now we wish to find the elapsed time
double elapsed = end - initial;
printf("\nThe elapsed time is %lf seconds\n",elapsed);
return 0;
}

5.3.2 Chunk Allocation

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "Cell.h"
#include "Point.h"
#include "voxel_obj.h"
// run only as check to programmed version
// run with usr/bin/time exe
// this constructor determines what goes into cell_mem and CELL
Voxel_obj::Voxel_obj(Point p0, Point p1)
{
    printf("\nConstructing voxel mesh object.");
    n_cells = 0;
    cell=0;
    cell_dim = 0;
    Cell_mem(1);
    // Generate root quad for this object
    cell[0].init(-1);
    root_lo = p0;
    root_hi = p1;
    n_cells = 1;
    root_cell = 0;
}
// this is the corresponding destructor
Voxel_obj::~Voxel_obj()
{
    printf("\nDestructing voxel mesh object.");
    Cell_mem(-1);
}

// this allows us to actually allocate cell mem
void Voxel_obj::Cell_mem(int cell_size)
{
    if (cell_size > 0 && cell_size != cell_dim)
    {
        if (cell != 0)
            cell=(CELL*)realloc((void*)cell,cell_size*sizeof(CELL));
        else
            103
cell=(CELL*)malloc(cell_size*sizeof(CELL));

if (cell == 0)
{
    fprintf(stderr, "Cell_mem(): Unable to reallocate memory for cells!" );
    fprintf(stderr, "\n Current dimension -> %d", cell_dim );
    fprintf(stderr, "\n Requested dimension -> %d", cell_size );
    exit(0);
}

    cell_dim = cell_size;
} else if (cell_size < 0)
{
    if (cell != 0)
        free(cell);
    cell = 0;
    n_cells = cell_dim = 0;
}
}

int main()
{
    // now, we wish to find start time of memory allocation
    time_t tm;
    char *t_char;
    //we also wish to store this in seconds from Jan 1, 1970
    double initial = time(&tm);
    t_char = ctime(&tm);
    printf("\nRun started at \%s", t_char);
    //counter
    int c;
    // now, we create a voxel object to allocate cells inside of
    Point p0 = Point(0.0,0.0);
Point p1 = Point(1.0, 1.0);
Voxel_obj *test;
test = new Voxel_obj(p0, p1);
// now, we allocate memory in chunks
int chunk = 10;
for (c = chunk; c <= 100000; c += chunk)
{
    test->Cell_mem(c);
}
// now, we wish to find end time of memory allocation
// we also wish to store this in seconds from Jan 1, 1970
double end = time(&tm);
t_char = ctime(&tm);
printf("\nRun completed at %s\t, t_char);
// now we wish to find the elapsed time
double elapsed = end - initial;
printf("\nThe elapsed time is %lf seconds\n", elapsed);
return 0;
}

5.4 Class Header Files

The following header files, VOXEL.h, CELL.h, NODE.h, and geometry.h, reveal information about the method used to set up the class structure of P_HUGG2D as well as the three-dimensional version, P_HUGG. The Vector.h, Point.h, and p_i.h header files were implemented in both the two and three dimensional versions to allow for consistent use of common point and vector operations. The SPACE toggle is set through a flag at compile time, thus making these files very versatile.
#include <stdio.h>
#include "p_i.h"
#include "space.h"
#include "List.h"
#include "Util.h"
#ifndef VOXEL_h
#define VOXEL_h
#define NKIDS 4
#define NNABORS 4
#define NPTS 9
#else
#define NKIDS 8
#define NNABORS 6
#define NPTS 27
#endif
#define VOXEL_CHUNK 10000

class VOXEL
{
    public:
        void print(FILE *outf)
        {
            fprintf(outf,"\nMother = %d",mom);
            fprintf(outf,"\nLevel = %d",level);
            fprintf(outf,"\nInside = %d",inside);
            fprintf(outf,"\nMe = %d - %d",me.proc,me.index);
        
}
int j;
fprintf(outf, "\nCell kids:");
for (j=0; j < NKIDS; j++)
    if (kids[j] >= 0)
        fprintf(outf, "\nKid %d = %d", j, kids[j]);
fprintf(outf, "\nCell neighbors:");
for (j=0; j < NNABORS; j++)
    if (nabors[j] >= 0)
        fprintf(outf, "\nNeighbor %d = %d", j, nabors[j]);
fprintf(outf, "\nCurrent cell contains vertices:");
for (j=0; j < NPTS; j++)
    if (c_n.list[j] >= 0)
        fprintf(outf, "\nNode %d = %d", j, c_n.list[j]);
//b_edges contained in my voxel
if (bnd_lst.max != 0)
    for (j=0; j < bnd_lst.max; j++)
        fprintf(outf, "\nElement %d = %d", j, bnd_lst.list[j]);
fprintf(outf, "\n");
}
void construct()
{
    int j;
    mom = -1;
    level = -1;
    inside = -1;
    me = p_i(-1,-1);
    for (j=0; j < NKIDS; j++)
        kids[j] = -1; //allows us to see it has not been assigned yet
    for (j=0; j < NNABORS; j++)
        nabors[j] = -1; //allows us to see it has not been assigned yet
c_n.list=0;
c_n.Redimension(0);
bnd_lst.list=0;
bnd_lst.Redimension(0);
ldim=0;
nl=0; //no list to begin with
loop=0;
}
void destruct()
{
    int j;
mom = -1;
level = -1;
inside = -1;
me = p_i(-1,-1);
for (j=0; j < NKIDS; j++)
    kids[j] = -1; //allows us to see it has not been assigned yet

for (j=0; j < NNABORS; j++)
    nabors[j] = -1; //allows us to see it has not been assigned yet

    c_n.Redimension(0);
bnd_lst.Redimension(0);
for (j = 0; j < ldim; j++)
    delete loop[j];
my_mem(ldim, nl, &loop, 0, 0);
ldim=0;
nl=0;
}
void init(int m, int l, p_i c)
{
int j;
mom = m;
level = 1;
inside = -1;
me = c;
for (j=0; j < NKIDS; j++)
    kids[j] = -1; //allows us to see it has not been assigned yet

for (j=0; j < NNABORS; j++)
    nabors[j] = -1; //allows us to see it has not been assigned yet

c_n.Redimension(NPTS);
for (j=0; j < NPTS; j++)
    c_n.list[j] = -1; //allows us to see it has not been assigned yet

c_n.max = NPTS; // to assure a list of size at least NPTS

// NO neeed to init bnd_lst list since grows ONLY as we need it
//set up loop lists
for (j = 0; j < ldim; j++)
    delete loop[j];
my_mem(ldim, nl, &loop, 0, 0);
ldim=0;
nl=0;
}

int kids[NKIDS];
int nabors[NNABORS];
int mom;
p_i me; // if real (not ghost) p is current and i is value of c
int inside;
int level;
List c_n;
List bnd_lst;
List **loop;
int nl; //number of closed loops in each voxel
int ldim;
};

#endif

5.4.2 CELL Class

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include "Util.h"
#include "List.h"
#include "p_i.h"
#include "space.h"
#define MIN(x,y) ((x) <= (y) ? (x) : (y))
#define MAX(x,y) ((x) >= (y) ? (x) : (y))
#ifndef CELL_h
#define CELL_h
#define CELL_h
#if SPACE == 2
#define NCNABORS 4
#define TRI2D 3
#define QUAD2D 4
#else
#define NCNABORS 6
#define TET3D 4
#define PYRA3D 5

110
```c
#define PENTA3D 6
#define HEXA3D 8
#endif
#define CELL_CHUNK 10000

class CELL
{
    public:

    void print(FILE *outf)
    {
        fprintf(outf, "Me = %d - %d", me.proc, me.index);
        // 3 for tri or 4 for quad..use defines
        fprintf(outf, "Type (num sides) = %d", type);
        int j;
        //print out nabors
        fprintf(outf, "Cell neighbors:");
        for (j=0; j < NCNABORS; j++)
            if (nabors[j] >= 0)
                fprintf(outf, "Neighbor %d = %d", j, nabors[j]);
        //print out nodes
        fprintf(outf, "Current cell contains vertices:");
        for (j=0; j < type; j++)
            if (c_n.list[j] >= 0)
                fprintf(outf, "Node %d = %d", j, c_n.list[j]);
        fprintf(outf, "\n");
    }

    void construct()
    {
        int j;
        type = -1;
        me = p_i(-1,-1);
    }
};
```
for (j=0; j < NCNABORS; j++)
    nabors[j] = -1; //allows us to see it has not been assigned yet

c_n.list=0;
c_n.Redimension(0);
}

void destruct()
{
    int j;
type = -1;
    me = p_i(-1,-1);
    for (j=0; j < NCNABORS; j++)
        nabors[j] = -1; //allows us to see it has not been assigned yet
    c_n.Redimension(0);
}

void init(p_i c, int t)
{
    int j;
type = t;
    me = c;
    for (j=0; j < NCNABORS; j++)
        nabors[j] = -1; //allows us to see it has not been assigned yet
    c_n.Redimension(type);
    for (j=0; j < type; j++)
        c_n.list[j] = -1; //allows us to see it has not been assigned yet
    c_n.max = type; // to assure a list of size at least NPTS
}

int nabors[NCNABORS]; //NCNABORS is max for quad/hex
//unused nabors for tri marked -1 (as if no nabor there due to bd)
    p_i me; // if real (not ghost) p is current and i is value of c
int type; //3 for tri 4 for quad
5.4.3 NODE Class

#include <stdio.h>
#include <stdlib.h>
#include "Point.h"
#include "p_i.h"
#include "space.h"
#ifndef NODE_h
#define NODE_h
#define NODE_CHUNK 10000
// allows us to init arrays of blank nodes
class NODE
{
   public:
   #if SPACE == 2
   NODE(Point p=Point(0.0,0.0), p_i m=p_i(-1,-1))
   { vert = p; me = m; }
   #else
   NODE(Point p=Point(0.0,0.0,0.0), p_i m=p_i(-1,-1))
   { vert = p; me = m; }
   #endif
   ~NODE() {}}
   #if SPACE == 2
   void construct()
{  
    vert = Point(0.0,0.0,0.0);
    me = p_i(-1,-1);
}
void destruct()
{
    vert = Point(0.0,0.0);
    me = p_i(-1,-1);
}
#endif
void construct()
{
    vert = Point(0.0,0.0,0.0);
    me = p_i(-1,-1);
}
void destruct()
{
    vert = Point(0.0,0.0,0.0);
    me = p_i(-1,-1);
}
#endif
void print(FILE *outf)
{
#if SPACE == 2
    fprintf(outf,\"Physical Node at ( %lf , %lf\",vert[0],vert[1]);
#else
    fprintf(outf,\"Physical Node at ( %lf , %lf , %lf\",vert[0],vert[1],vert[2]);
#endif
    fprintf(outf,\"Actual Node on Processor %d Index %d\",me.proc,me.index);
}
p_i me; // proc and voxel node belongs to (if real, proc=my_rank)
Point vert; // gives us physical location

#endif

5.4.4 Geometry Class

#include <stdio.h>
#include "Point.h"
#include "Vector.h"
#include "Edge.h"
#include "List.h"
#include "Util.h"
#include "Geom_Edge.h"
#ifndef sharp_point_h
#define sharp_point_h
#define sharp_point_h
class Sharp_Point
{
  public:
  Sharp_Point()
  { node = -1; mat[0] = mat[1] = 0; }
  void print()
  {
    printf("\n Node = %d",node);
    printf("\n Material 1 = %d",mat[0]);
    printf("\n Material 2 = %d",mat[1]);
  }
  int node;
int mat[2];
);
#endif

#ifndef geometry_h
#define geometry_h
class geometry
{
    public:
    geometry()
    {
        ngb = 0;
        n_gnodes = 0;
        n_gedges = 0;
        n_begin = 0;
        n_end = 0;
        g_vert = 0;
        g_edge = 0;
        blo = 0;
        bhi = 0;
    }
    ~geometry()
    {
        free(g_vert); free(n_begin); free(n_end); ngb=n_gnodes=0;
        free(g_edge);
    }
    //constructor fine since instatiated with new like normal C++ obj
    void read_geom(int ngf, char* fnames[]);
    int Identify_sharp_points(Sharp_Point *sharp);
    int Intersect(Point pa, Point pb, Point &p);
Point min_point();
Point max_point();
Point closest(Point p, int &b, Vector &nrm);
void Plotfile();
double Spacing(double x, double y, double smn);
int ngb, n_gnodes, n_gedges;
int *n_begin, *n_end, *layers;
double *g_space, *n_space, *length;
Point *g_vert, *blo, *bhi;
Geom_Edge *g_edge;
};
#endif

5.4.5 Point Class

#include <stdio.h>
#ifndef SPACE
#include "space.h"
#endif
#ifndef Point_h
#define Point_h
class Point
{
    public:
    #if SPACE == 2
    inline Point(double x=0.0, double y=0.0)
    {
        pos[0] = x; pos[1] = y;
    }
    #else
    inline Point(double x=0.0, double y=0.0, double z=0.0)
    {
    }
#endif
};
#endif
{ pos[0] = x; pos[1] = y; pos[2] = z; }
#endif

~Point() { }
#if SPACE == 2
void print(FILE *outf)
{
    fprintf(outf,"\nPoint (x,y) = (%.12g,%.12g)",pos[0],pos[1]);
}
#else
void print(FILE *outf)
{
    fprintf(outf,"\nPoint (x,y,z) = (%.12g,%.12g,%.12g)",pos[0],pos[1],pos[2]);
}
#endif

inline Point operator + (const Point &) const;
inline Point operator - (const Point &) const;
inline Point &operator += (const Point &);
inline Point &operator -= (const Point &);
inline Point &operator += (double);
inline Point &operator -= (double);
inline Point &operator *= (double);
inline Point &operator /= (double);
inline Point operator * (double) const;
inline Point operator / (double) const;
inline double & operator () (int);
inline double operator () (int) const;
inline double & operator [] (int);
double pos[SPACE];
};
inline double Point::operator () (int i) const
{ return pos[i]; }
inline double & Point::operator [] (int i)
{ return pos[i]; }
#if SPACE == 2
    inline Point Point::operator * (double other) const
    { return Point(pos[0]*other, pos[1]*other); }
#else
    inline Point Point::operator * (double other) const
    { return Point(pos[0]*other, pos[1]*other, pos[2]*other); }
#endif
#if SPACE == 2
    inline Point Point::operator / (double other) const
    {
        #ifdef _DEBUG_
        if (fabs(other) <= 1e-20) throw new mException (__LINE__,__FILE__);  
        #endif
        return Point(pos[0]/other, pos[1]/other); }
#else
    inline Point Point::operator / (double other) const
    {
        #ifdef _DEBUG_
        if (fabs(other) <= 1e-20) throw new mException (__LINE__,__FILE__);  
        #endif
        return Point(pos[0]/other, pos[1]/other, pos[2]/other); }
#endif
#if SPACE == 2
inline Point Point::operator + (const Point &other) const
{ return Point(pos[0]+other.pos[0], pos[1]+other.pos[1]); }

inline Point Point::operator - (const Point &other) const
{ return Point(pos[0]-other.pos[0], pos[1]-other.pos[1]); }
#endif
inline Point Point::operator + (const Point &other) const
{ return Point(pos[0]+other.pos[0], pos[1]+other.pos[1], pos[2]+other.pos[2]); }
#endif
inline Point Point::operator - (const Point &other) const
{ return Point(pos[0]-other.pos[0], pos[1]-other.pos[1], pos[2]-other.pos[2]); }
#endif
inline Point& Point::operator += (const Point &other)
{
    for (int i = 0; i < SPACE; i++)
    {
        pos[i]+=other.pos[i];
    }
    return *this;
}
inline Point& Point::operator -= (const Point &other)
{
    for (int i = 0; i < SPACE; i++)
    {
        pos[i]-=other.pos[i];
    }
    return *this;
}
inline Point& Point::operator += (double other)
{
for (int i = 0; i < SPACE; i++)
{
    pos[i]+=other;
}
return *this;
}

inline Point& Point::operator -= (double other)
{
for (int i = 0; i < SPACE; i++)
{
    pos[i]-=other;
}
return *this;
}

inline Point& Point::operator *= (double other)
{
for (int i = 0; i < SPACE; i++)
{
    pos[i]*=other;
}
return *this;
}

inline Point& Point::operator /= (double other)
{
#ifdef _DEBUG_
if (fabs(other) <=1e-20) throw new mException (__LINE__,__FILE__);
#endif
for (int i = 0; i < SPACE; i++)
{
    pos[i]/=other;
}
5.4.6 Vector Class

#include <stdio.h>
#include <math.h>
#include "space.h"
#include "Point.h"
#ifndef Vector_h
#define Vector_h
class Vector
{
    double vec[SPACE];
public:
    #if SPACE == 2
    inline Vector(double dx=0.0, double dy=0.0)
    { vec[0] = dx; vec[1] = dy; }
    #else
    inline Vector(double dx=0.0, double dy=0.0, double dz=0.0)
    { vec[0] = dx; vec[1] = dy; vec[2] = dz; }
    #endif
    inline Vector(const Point &from, const Point &to);
    #if SPACE == 2
    void print(FILE *outf)
    { fprintf(outf,\"\nVector (x,y)= (%.12g,%.12g)\",vec[0],vec[1]);}
    #else
void print(FILE *outf)
{
    fprintf(outf,"\nVector (x,y,z)= (%.12g,%.12g,%.12g),vec[0],vec[1],vec[2]);
}
#endif

inline Vector operator + (const Vector &v);
inline Vector operator - (const Vector &v);
inline Vector &operator += (const Vector &v);
inline Vector &operator -= (const Vector &v);
inline Vector &operator += (const double &s);
inline Vector &operator -= (const double &s);
inline Vector &operator *= (const double &s);
inline Vector &operator /= (const double &s);
#if SPACE == 2
inline double operator * (const Vector &v)
{
    return double(vec[0]*v.vec[0]+vec[1]*v.vec[1]);
}
#else
inline double operator * (const Vector &v)
{
    return double(vec[0]*v.vec[0]+vec[1]*v.vec[1]+vec[2]*v.vec[2]);
}
#endif
#if SPACE == 2
inline double operator % (const Vector &v);
#else
inline Vector operator % (const Vector &v);
#endif
inline Vector operator * (const double &s);
inline Vector operator / (const double &s);
inline double operator () (int i) const;
inline double &operator () (int i);
inline double &operator [] (int i);
inline double magnitude();
inline void normalize();
inline Vector::Vector(const Point &from, const Point &to)
{
    for (int i = 0; i < SPACE; i++)
    {
        vec[i] = to(i) - from(i);
    }
}

inline double &Vector::operator () (int i)
{ return vec[i]; }
inline double Vector::operator () (int i) const
{ return vec[i]; }
inline double &Vector::operator [] (int i)
{ return vec[i]; }
#if SPACE == 2
inline double Vector::operator % ( const Vector &v)
{
    return (vec[0]*v.vec[1]-vec[1]*v.vec[0]);
}
#else
inline Vector Vector::operator % ( const Vector &v)
{
    return Vector(vec[1]*v.vec[2]-vec[2]*v.vec[1],
                  vec[2]*v.vec[0]-vec[0]*v.vec[2],
                  vec[0]*v.vec[1]-vec[1]*v.vec[0]);
}
#endif
#if SPACE == 2

124
inline Vector Vector::operator +( const Vector &v)
{ return Vector(vec[0]+v.vec[0], vec[1]+v.vec[1]); }
inline Vector Vector::operator -( const Vector &v)
{ return Vector(vec[0]-v.vec[0], vec[1]-v.vec[1]); }
inline Vector Vector::operator *( const double &s)
{ return Vector(s*vec[0], s*vec[1]); }
inline Vector Vector::operator /( const double &s)
{ return Vector(vec[0]/s, vec[1]/s); }
inline double Vector::magnitude()
{ return double(sqrt(vec[0]*vec[0] + vec[1]*vec[1])); }
#endif
inline Vector &Vector::operator += (const Vector &v)
{
    for (int i = 0; i < SPACE; i++)
    {
        vec[i]+=v.vec[i];
    }
    return *this;
}
inline Vector &Vector::operator -= (const Vector &v)
{
    for (int i = 0; i < SPACE; i++)
    {
        vec[i] -= v.vec[i];
    }
    return *this;
}

inline Vector &Vector::operator += (const double &s)
{
    for (int i = 0; i < SPACE; i++)
    {
        vec[i] += s;
    }
    return *this;
}

inline Vector &Vector::operator -= (const double &s)
{
    for (int i = 0; i < SPACE; i++)
    {
        vec[i] -= s;
    }
    return *this;
}

inline Vector &Vector::operator *= (const double &s)
{
    for (int i = 0; i < SPACE; i++)
    {
        vec[i] *= s;
    }
}
return *this;
}

inline Vector &Vector::operator /= (const double &s)
{
    for (int i = 0; i < SPACE; i++)
    {
        vec[i]/=s;
    }
    return *this;
}

// if mag less than machine eps, vector is 0 anyways, so no normalization needed
// if we divided nearly 0 by zero, we would get a large, useless answer
#if SPACE == 2
inline void Vector::normalize()
{
    double mag = sqrt(vec[0]*vec[0]+vec[1]*vec[1]);
    if (mag > 1.0e-20) { vec[0] = vec[0] / mag; vec[1] = vec[1] / mag; }
}
#else
inline void Vector::normalize()
{
    double mag = sqrt(vec[0]*vec[0]+vec[1]*vec[1]+vec[2]*vec[2]);
}
#endif
#if SPACE == 3
inline double scalar_triple_product(Vector a, Vector b, Vector c)
{

double d;
d = a[0]*b[1]*c[2]+a[1]*b[2]*c[0]+a[2]*b[0]*c[1]-
a[2]*b[1]*c[0]-a[1]*b[0]*c[2]-a[0]*b[2]*c[1];
return(d);
}
#endif
#endif

5.4.7 Processor-Index Class

#include <stdio.h>

#ifndef p_i_h
#define p_i_h
class p_i
{
    public:
        p_i (int p = -1, int i = -1)
        {
            proc = p;
            index = i;
        }
    ~p_i(){}
    void print(FILE *outf)
    {
        fprintf(outf,"
Processor= %d, Index= %d",proc,index);
    }
    int proc;
}
int index;
};

inline bool operator==(p_i a, p_i b)
{
    return (a.proc == b.proc && a.index == b.index);
}

inline bool operator!=(p_i a, p_i b)
{
    return (a.proc != b.proc || a.index != b.index);
}

#endif
Vita

Vincent Charles Betro was born in Salem, Ohio, on June 28, 1980. After attending Bartlett High School in Bartlett, TN, he pursued an undergraduate degree in Secondary Mathematics at The University of Tennessee at Chattanooga, where he was a member of the University Honors program. After completing his degree in December 2002, he taught mathematics at Ooltewah Middle School and Meigs County High School before returning to UTC in January 2005 to accept a position as adjunct instructor in the Developmental Mathematics Department. At UTC, he worked with the Web Homework System (www.mathclass.org) in conjunction with The University of Kentucky and UTC professors Dr. Stephen Kuhn and Dr. Terry Walters. He also resumed his studies in August 2005. Vincent currently continues to teach as an adjunct in the Mathematics Department and will begin work on his Ph.D. in the fall semester of 2007.