Advances in Parallelization For Large Scale Oct-Tree Mesh Generation

Matthew O'Connell [*]

NASA Langley Research Center, Hampton, Virginia, 23681

Steve L. Karman [†]

University of Tennessee at Chattanooga: SimCenter, Chattanooga, Tennessee, 37403

Despite great advancements in the parallelization of numerical simulation codes over the last 20 years, it is still common to perform grid generation in serial. Generating large scale grids in serial often requires using special "grid generation" compute machines that can have more than ten times the memory of average machines. While some parallel mesh generation techniques have been proposed, generating very large meshes for LES or aeroacoustic simulations is still a challenging problem. An automated method for the parallel generation of very large scale off-body hierarchical meshes is presented here. This work enables large scale parallel generation of off-body meshes by using a novel combination of parallel grid generation techniques and a hybrid "top down" and "bottom up" oct-tree method. Meshes are generated using hardware commonly found in parallel compute clusters. The capability to generate very large meshes is demonstrated by the generation of off-body meshes surrounding complex aerospace geometries. Results are shown including a one billion cell mesh generated around a Predator Unmanned Aerial Vehicle geometry, which was generated on 64 processors in under 45 minutes.

I. Introduction

Automated grid generation techniques for Computational Fluid Dynamics simulation have seen a rise in popularity in recent years. Generating large scale meshes with these methods has, up to this point, largely been relegated to generating the mesh on a "grid generation" machine. To handle large meshes, the "grid generation" machines can be equipped with more than 10 times the memory of average machines. Large meshes often take hours or even several days to generate. The power of next generation compute clusters will enable simulations using exceptionally large grids. There has been a growing interest in parallelizing the mesh generation process to harness the power of the next generation of computing hardware.

Hexagrid[1] has seen increased performance with the addition of shared and distributed memory parallelism through use of the building cube method.[2] Distributed memory parallelization has also been implemented in Boxer Mesh[3] and PHUGG.[4] These methods have been used to generate meshes on the order of tens to hundreds of millions of cells. In addition to these recent advancements, higher grid resolutions are needed to support high fidelity problems such as Large Eddy Simulations (LES) and aeroacoustic simulations. In these problem domains, meshes approaching or exceeding billions of control volumes will be needed for complex geometries.

Many common automatic mesh generation techniques are based on hierarchical trees for off-body grid generation combined with a near body gridding technique. Some techniques include cut-cell methods such as in Cart3D,[5] SPLITFLOW,[6] and PHUGG. Other techniques, such as Hexagrid, project nodes from the off-body mesh onto the geometry.[7] FASTAR[8] was developed using a type of binary tree called a split-tree. It uses tetrahedral meshing to combine a hierarchical technique for off-body grids to a body conforming viscous mesh generated using extrusion.

Hierarchical mesh generation methods are generally implemented as either top down or bottom up methods. In top down methods, the mesh is generated and the tree is produced, while traversing down the tree structure level by level. In a bottom up mesh generator, data at the lowest level of the tree is generated first. The remaining mesh is generated as the tree data structure is made level by level progressing up the tree. Originally, top down methods were common, though recently bottom up methods have been popularized by Dawes.[3]

Rather than use a strictly top down or bottom up method, Betro introduced a hybrid method that begins as a top down approach and finishes as bottom up.[8] That work was done using a specialized binary tree called a split tree. In

[*]Intern Employment Program Research Trainee, Computational AeroSciences Branch, Mail Stop 128, AIAA Student Member.
[†]Professor, Computational Engineering Department, AIAA Associate Fellow.

recent years a number of researchers have developed dynamically load balanced parallel oct-tree methods. Octor[9] and p4est[10] are both examples of parallel oct-tree codes, which use space-filling curves to repartition and dynamically load balance during the mesh generation process.

This work presents a parallel, hybrid top down and bottom up approach for use in off-body mesh generation using oct-trees. This is in contrast to other parallel grid generation techniques, which are either strictly top down or bottom up. This work differ from those methods by using both top down and bottom up tree traversals during the generation process. The technique begins by generating a coarse mesh by subdividing elements in an oct-tree based on an input geometry. The oct-tree is only refined until it reaches an intermediate depth creating a coarse mesh. The coarse mesh is then partitioned before refinement continues top down, in parallel, to create full resolution spacing around the geometry. Finally, a smooth variation in cell size is achieved by enforcing quality rules in parallel by traversing the oct-tree level by level beginning at the bottom.

This paper focuses on parallelization of an off-body mesh generation technique. Off-body mesh generation must be combined with a near body technique to provide a complete solution for high fidelity CFD problems. There are many options for a near body technique including cut-cell, immersed boundary, and creating a body conforming mesh through projection. The near wall approach will be the subject of a future paper.

II. Hierarchical Mesh Generation

Hierarchical meshing techniques use spatial trees whose elements are voxels. The term voxel, which stands for volumetric pixel, comes from the field of computer graphics. Each voxel contains only a Cartesian bounding box, referred to as an extent box, the index of the parent voxel, and the indices of any of the voxel's children. No unique identifiers are stored for the nodes of the mesh as they are not needed by the mesh generation process, but they can be generated when the mesh is exported to comply with file format standards. The voxels are implemented in this work using a very simple data structure, which consumes only 84 bytes of memory.

The root element of an oct-tree encompasses the entire spatial domain spanned by the oct-tree. Children of a voxel are obtained by dividing a parent voxel into eight equal octants. The child of a voxel has the same aspect ratio as its parent. The work presented here requires each voxel to be isotropic. The root voxel is initialized as isotropic making all voxels isotropic. Requiring all voxels to be isotropic allows for the size of a voxel to be measured as simply the length of one of its edges. This length is referred to as the characteristic length of the voxel. The characteristic length of a voxel is compared to user defined spacing to determine whether or not a voxel should subdivide. Voxels that do not have children are leaves in the tree and become cells in the mesh.

Though isotropic elements have ideal quality, they are not suited for all regions of the simulation. Most notably, isotropic elements are not suited in boundary layers of high Reynolds number flow. Using isotropic elements to resolve a boundary layer would be very costly. Hierarchical grid generation is usually used only in the off-body region where isotropic elements are better suited

The oct-tree based methods lack precise control over grid spacing. Instead, these tools favor automation and speed over precision. However, it can be ensured that the spacing in the mesh will be at least as small as the user requested. Figure 1 demonstrates this in 2D using a quad-tree. In this example, small spacing is requested inside the Cartesian aligned region outlined as the dashed blue line. Regions that are Cartesian aligned can be described using just two coordinates specifying the location of two opposite corners of a box. They will be referred to as extent boxes. Subdivision occurred inside the marked extent box, but this subdivision also decreased spacing outside the requested region. It is also not possible to achieve precisely sized spacing. This is illustrated in Fig. 2. For example, a requested spacing of 0.33 is incompatible with a mesh where the root voxel had a characteristic length of 1. Subdividing the root voxel would yield spacings of 0.5, 0.25, 0.125, and so on.

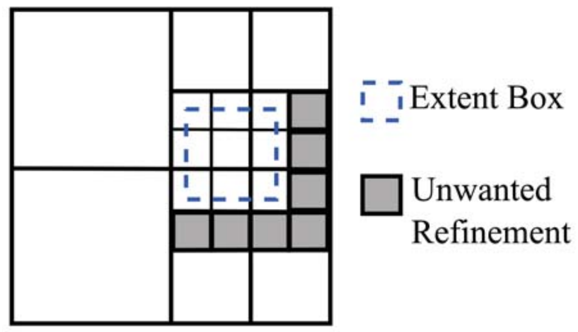


Figure 1. Extent box used to define a region of refinement in the mesh. Notice that due to the quad-tree data structure, additional voxels outside the extent box were also refined.

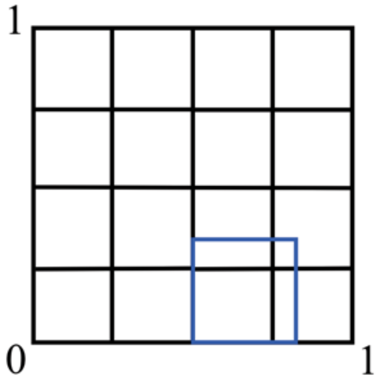


Figure 2. A root voxel of unit length cannot produce the user defined spacing of 0.33 represented by the blue square. The mesh is subdivided until the spacing is at least as small as the user requested. In this case the mesh subdivided creating voxels of length 0.25.

A. Geometry Input

The mesh generation process begins by reading in a stereo lithography file. Stereo lithography files follow a simple format to store a collection of triangles called facets. The file contains no adjacency information and acts as a lowest common denominator format for faceted surface information. Stereo lithography files make no guarantee that the geometry surface is manifold. In fact, these files often contain gaps or overlapping triangles. Gaps can be handled so long as they are smaller than the local mesh spacing. Any gaps larger than the local mesh spacing are identified as geometry features. The mesh generator must remove elements of the mesh, which are not in the flow region. With external flow cases this means removing elements that are inside the geometry. It is possible that overlapping triangles could make it difficult to categorize regions of the mesh as inside or outside the geometry. In practice this problem has not been encountered. Furthermore, geometry facets do not need to have a specified orientation or even uniform orientation.

B. Mesh Generation

The mesh generation process begins with initializing the root voxel to be large enough to encompass the entire mesh domain. Once the root voxel is initialized, geometry facets are then used to recursively subdivide voxels until they reach a minimum size threshold. Each facet is processed beginning at the root voxel. The extent box of the voxel is compared to the facet. If any part of the facet is inside the voxel, it is considered as crossing that voxel. If a voxel's spacing is larger than the spacing set by the user, it subdivides creating 8 new child voxels. This subdivision occurs only once per voxel. If another facet later is determined to be inside the voxel, it does not trigger a second subdivision. After new child voxels are created, those children are compared to the facet and the algorithm repeats recursively down the tree. These simple rules are applied each time a voxel is compared to a facet. Each recursive call ends when the facet is compared to a voxel, which has a characteristic length the same size or smaller than the user requested spacing. The process then repeats for each geometry facet creating an oct-tree with different sized voxels for each depth in the tree.

Cells that are completely contained within a geometry are outside the computational domain and are deleted, but first these cells must be identified. Any cell that contains a geometry facet is labeled as crossing the geometry during the initial subdivision phase. All the remaining cells are not labeled. These cells are either inside the geometry and will be deleted or they are outside the geometry and will be part of the final mesh. The cells tagged as crossing the geometry separate cells inside the geometry from those outside. The collection of cells tagged as crossing the geometry will defined a closed region in the mesh as long as the input geometry does not contain large gaps.

A flood fill algorithm is used to mark all cells that are outside the geometry. The flood fill algorithm begins with a cell that is known to be outside the geometry. This cell is chosen by selecting a voxel in the mesh that does not contain geometry as follows. Six rays are cast from the centroid of the voxel, one in each of the cardinal directions. If all the rays pass through an even number of geometry facets, then the cell is outside the geometry. If the all rays pass through an odd number of geometry facets, then the cell is inside the geometry and another cell must be selected. Overlapping triangles or gaps in the input geometry may cause rays to not agree on inside or outside status. If this happens the cell is discarded and another is selected. The method continues testing voxels until it finds a cell, which is outside the geometry. Once a cell is found, which is outside the geometry, the flood fill progresses recursively to neighboring cells. The flood fill does not progress to cells, which are marked as containing geometry. After the flood fill terminates, each cell is either marked as containing geometry, marked as outside the geometry, or left unmarked. The user can select to generate an internal or external grid. For external cases all unmarked cells are implicitly identified as internal to the geometry, and these cells are deleted. For an internal grid, all external and outside cells are deleted. However, when generating internal cases, each cell must be determined either inside or outside the geometry. Then outside cells must be deleted. The resulting mesh is referred to as the base mesh.

C. Gradation Enforcement

An example of a base mesh can be seen in Fig. 3. Notice that the base mesh has large cells adjacent to much smaller cells. The nodes that make up corners of small cells are sometimes also corner nodes of larger cells. There are also corner nodes of small cells, that are instead on a side of a larger cell and not its corner. These nodes are commonly called hanging nodes. The more hanging nodes a side has the larger the cell size jump across the face. This quick cell gradation creates problems for many simulation techniques and so, for the final mesh, restrictions are placed on the growth of cell sizes. These qualities are not enforced on the base mesh; they are only enforced on the parallel mesh.

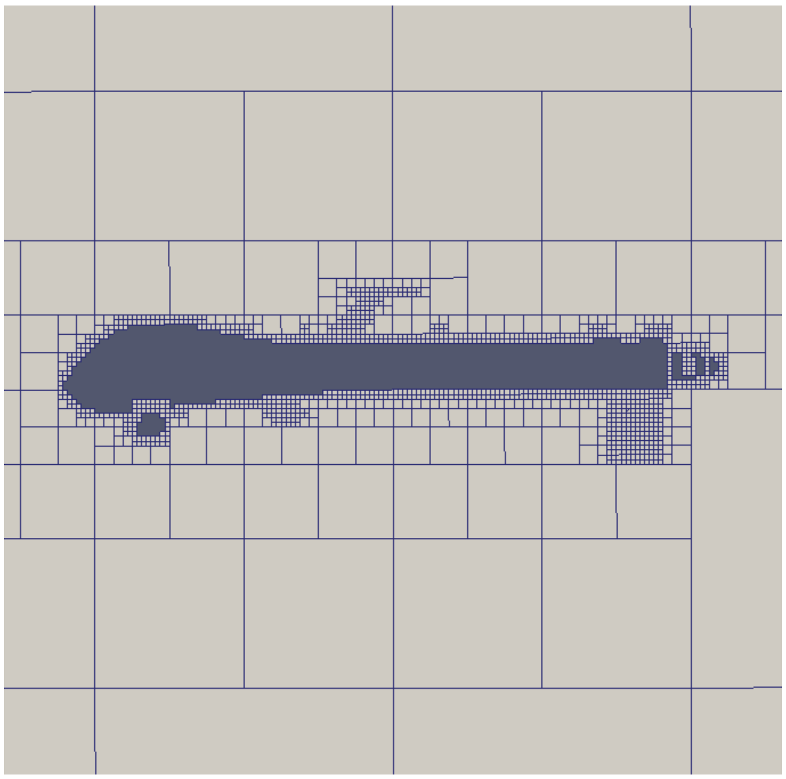
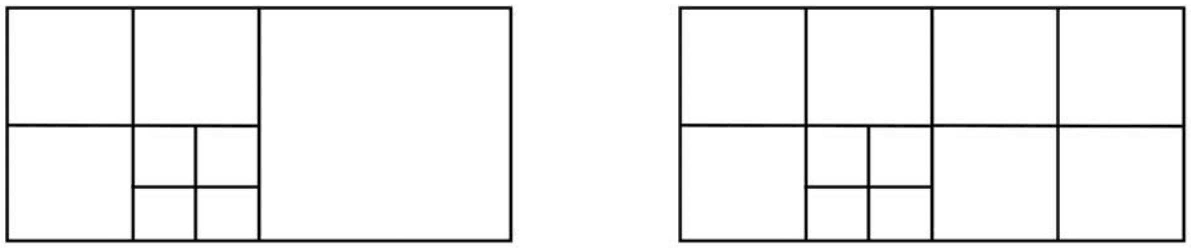


Figure 3. A base mesh for a Predator geometry. Many large cells have several hanging nodes.

Smooth cell size gradation is enforced with two quality characteristics. The first characteristic is that no cell has more than one hanging node per edge. This restriction is common among hierarchical mesh generators and is referred to as a 4:1 neighbor gradation (2:1 in 2D), or sometimes simply the balancing rule. A violation of the balancing rule is illustrated in Fig. 4. The large voxel on the right in Fig 4a has three neighboring voxels on the same side. The large voxel's left side has two hanging nodes. The balancing rule requires that each voxel side has at most only one hanging node. Figure 4b shows the same set of voxels after a subdivision refinement has been performed on the largest voxel. Now each voxel has at most one hanging node on any given side and the balancing rule is now being satisfied.

The second quality characteristic that is enforced is a minimum thickness layer. A region of a mesh with the same size cells is referred to as a layer. Every cell of a given size needs to be adjacent to other cells of the same size such that there are at least n cells of a particular size before cell size changes. The parameter n is set by the user and describes the minimum thickness of a layer. Typical values of the thickness parameter are between 4 and 10. Similar to the minimum spacing parameter, a specific thickness of a layer cannot be guaranteed. Layers are guaranteed to be at least

n cells thick. However, because of how voxels can subdivide, layers may be thicker in some regions. An example of layer thickness is illustrated in Fig. 5.



(a) A violation of the balancing rule. **(b) The balancing rule enforced by subdivision.**

Figure 4. The balancing rule requires that no cells have more than one hanging node. The left image shows a large cell with two hanging nodes violating the balancing rule. On the right, the balancing rule is followed after subdividing the large cell.

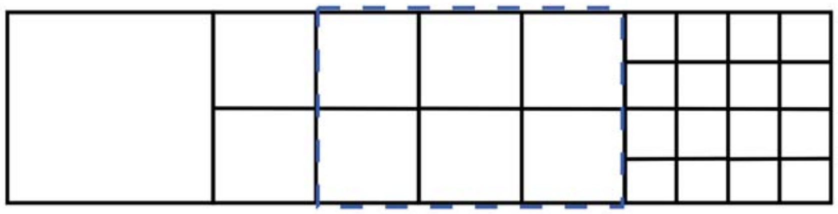


Figure 5. Thickness parameter of three. The layer between small and large cell size is four cells thick due to the oct-tree data structure.

The enforcement of both the balancing rule and layer thickness occurs simultaneously and is handled in a bottom up sweep. The technique begins at the deepest level of the oct-tree corresponding to cells of the smallest spacing. An extent box is created around each of these cells. Each extent box is enlarged using the gradation parameter n described above. Once enlarged, all extent boxes are used for refinement until all cells inside the extent boxes match the smallest spacing. This step creates a layer of the smallest cells.

After all the cells in the smallest layer have been created, the balancing rule is enforced on all voxels adjacent to the newly created layer. This is done by enlarging extent boxes for all cells at the smallest spacing. When enforcing the balancing rule, extent boxes are only enlarged enough to capture neighboring cells. Neighboring cells are subdivided until they satisfy the 4:1 requirement. Now, there are no cells at the smallest spacing that are adjacent to any cell that is greater than 8 times its size, and each layer of cells of the same size has the proper thickness. The process is repeated at each depth of the tree beginning at the deepest level of the tree and proceeding to the highest.

III. Parallel Implementation

Parallel mesh generation is the primary goal of this work. This implementation uses MPI to run on distributed memory machines. Unlike common serial mesh generation methods, the parallel implementation requires no special hardware beyond common cluster compute nodes. There is no need for a "grid generation" machine with exceptionally large memory. The parallel grid generation process is described below.

Every processor reads in the geometry and generates the same base mesh in parallel. The minimum cell spacing for the base mesh is determined by a user defined maximum depth. The maximum depth of a tree is the maximum number of times the root voxel is subdivided. The base mesh is generated down to this maximum depth. For the results presented in the next section, a typical depth of 7 was used. After the base mesh is generated down to the maximum depth, cells identified as internal to the geometry are removed to save memory. The leaf voxels in the base mesh are then assigned to partitions using METIS.[11] Since each processor generates the same base mesh, no communication is required to partition the mesh. Each processor receives the same partitioning from METIS.

Other parallel meshing techniques also generate base meshes down to a user desired maximum depth; however, these techniques enforce quality rules in the base mesh.[4, 8] These techniques can be described as top down methods since they begin at the top of the tree and generate the mesh as they traverse down. The technique proposed here differs from those methods by generating only the smallest possible tree structure that contains pieces of the geometry. Notice in Fig. 3 the cells in the mesh with several hanging nodes. In the base mesh, these hanging nodes are allowed since the base mesh is not used as the computational mesh for the CFD solver. It is only after the geometry has been resolved by a top down generation of the base mesh that the quality rules are enforced, this time by travelling up the tree. This technique can be described as a hybrid method that begins with a fast "top down" sweep to build the skeleton of the tree followed by a "bottom up" sweep to fill the tree.

The base mesh serves two purposes. One copy of the base mesh is used as a map describing what processor is responsible for each region of the domain and a second copy of the base mesh is used as a starting point for parallel mesh generation. Each processor retains a copy of the base mesh and the partitioning information. Using this information, processors can determine what other partitions need to refine even with large extent boxes that span several partitions.

Each processor must maintain the tree structure that supports the region of space it is assigned, but any other sections of the mesh can be removed. Figures 6 – 8 represent a one dimensional base mesh generated with a binary tree and the steps taken to partition this mesh on two processors. The mesh contains 10 cells corresponding to the 10 leaves in the tree in Fig. 7. By partitioning the leaf voxels of the base mesh, each processor is assigned spatial regions of the computational domain. In this case, the space occupied by the left 5 voxels is identified as being owned by one partition, the space occupied by the right 5 voxels to another. In Fig. 7, voxels in the tree that are marked as blue will be stored strictly on the processor responsible for the blue leaf voxels while red voxels will need to be stored only on the other processor. Voxels colored purple have both red and blue descendants and will need to be stored on both processors. Any part of the tree that is not needed by the host processor is deleted. This is illustrated in Fig. 8.

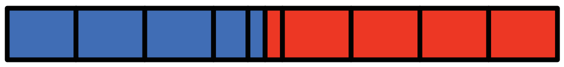


Figure 6. A one dimensional base mesh. Cells colored blue are assigned to one partition, red cells are assigned to another.

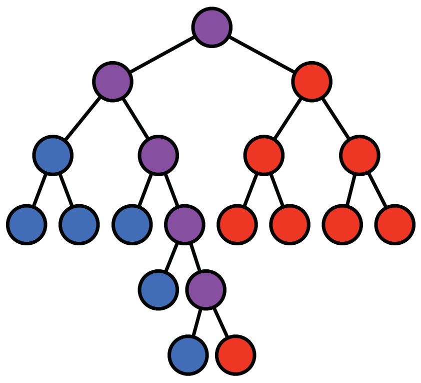


Figure 7. A representation of the tree for the base mesh partitioned onto two processors. Blue voxels only need to be stored on the processor responsible for blue cells. Red voxels only need to be stored on the processor responsible for red cells. Purple voxels have both red and blue descendants and must be stored on both processors.

After removing the unnecessary voxels, every processor has a coarse mesh covering the domain that the processor is assigned. The coarse meshes will be refined to create the partitioned components of the final mesh. Every processor again uses geometry facets to refine its mesh. During base mesh generation, cells containing geometry are refined to a maximum depth. After partitioning, cells containing geometry are refined until they match the global minimum spacing requested by the user. Cells internal to the geometry must be deleted. A flood fill algorithm is again used to mark these internal cells.

No communication is used for the flood fill; instead it is performed by each processor, The flood fill does not progress to neighboring partitions. Each processor may contain multiple disjoint regions of the domain; both internal and external cells must be identified. When all cells have been identified as either inside of, outside of, or crossing the geometry, the cells outside the flow domain are deleted.

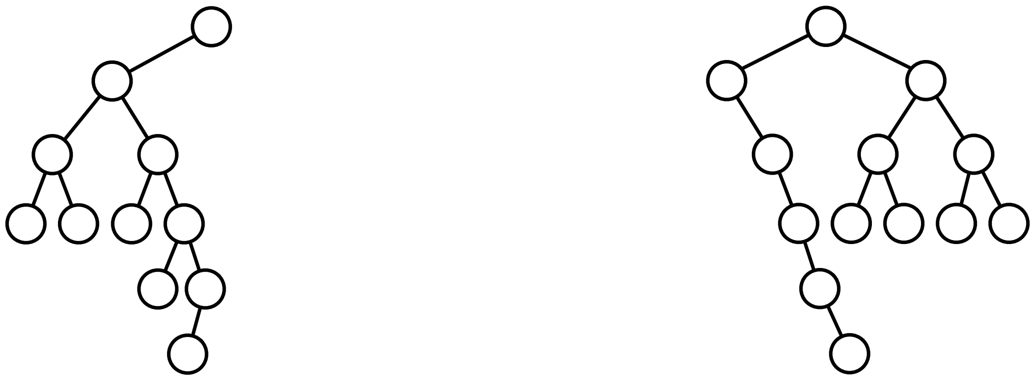


Figure 8. After partitioning, each processor removes unused voxels from their local tree.

For many near body techniques, such as those based on cut-cells or immersed boundary, only strictly internal cells need to be deleted. However, for other techniques, such as those that project off body elements, cells that cross the geometry also must be deleted. If the layer parameter is less than 3, deleting crossing cells may create cells in the off-body mesh that do not meet the user specified minimum spacing. To ensure that cells of the minimum size are exposed to the geometry, an extra layer of minimum spacing cells is created. Extent boxes are created from all cells that contain the geometry. The extra layer of minimum spacing cells only needs to be one cell width thick, so the extent boxes are enlarged enough to cross into neighboring cells. Cells inside these extent boxes are subdivided until they have characteristic lengths that meet the minimum spacing. Some of the extent boxes will cross into other partitions. Processors can identify if any extent boxes cross into regions of the domain owned by other processors by using a saved copy of the base mesh and the partitioning information. Any extent box that crosses into the domain owned by another processor is sent to that processor along with the requested spacing. The receiving processor refines the mesh making all cells within the extent box match the required spacing.

The last step in the mesh generation process is to enforce the balancing rule and to create the proper thickness in spacing layers. Enforcement of these gradation qualities begins at the maximum depth across all processors. Spacing layers are created by using extent boxes from cells at the smallest spacing. These extent boxes are enlarged and used to subdivide all the cells inside the extent box down to the smallest spacing. If an extent box crosses into domains owned by other processors, the extent box and the requested spacing is communicated to those processors. Once each processor completes the refinement for the deepest level, the process repeats for the next deepest and continues until the top of the tree is reached. Figures 9 and 10 illustrate the gradation phase. Figure 9 shows that the layer of smallest spacing cells has been created. Figure 10 is the final mesh. All the cell layers have a minimum thickness of 4 cells.

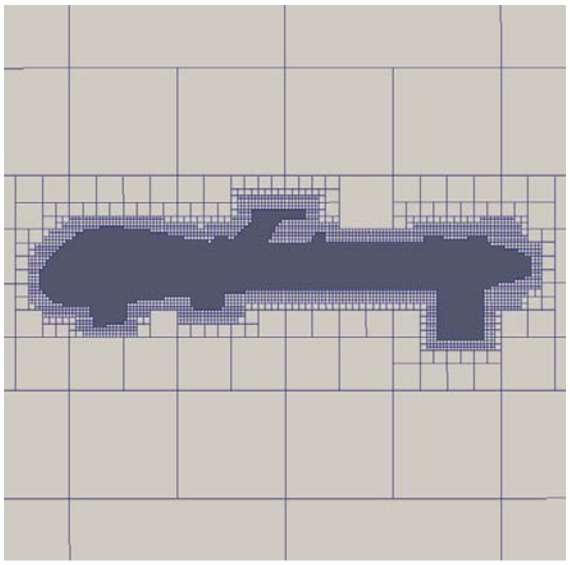


Figure 9. The smallest spacing cells have the proper sized layer based on the user requested size.

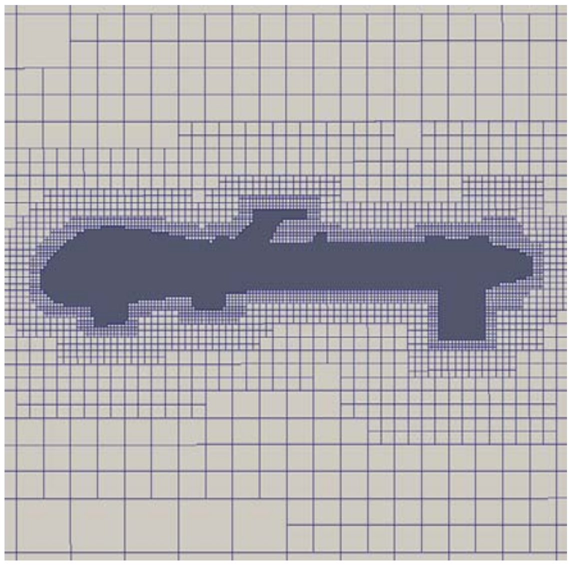


Figure 10. The final off-body mesh with all the quality enforcements.

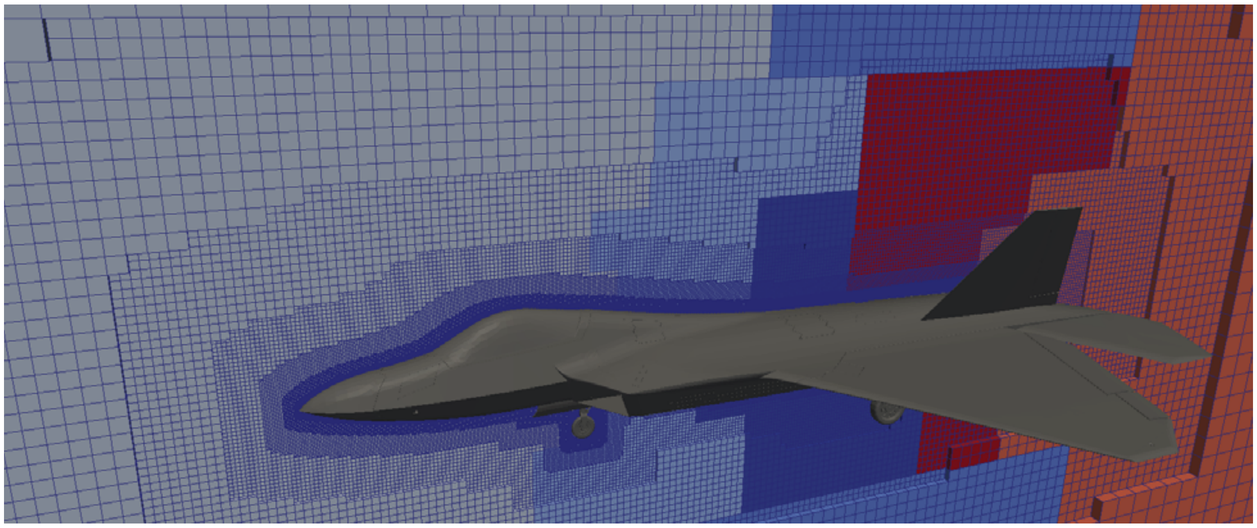


Figure 11. A 61 million cell mesh for a fighter aircraft generated on 8 processors with partitions colored.

IV. Grid Generation Examples

This section presents some meshing examples generated using this parallel mesh generation technique. The example meshes are generated using either an F-22 fighter aircraft geometry, a Predator Unmanned Aerial Vehicle (UAV) geometry, or a simple pipe geometry. Both the F-22 and the UAV geometry files can be obtained from www.GrabCAD.com.

A. Fighter Aircraft

The first example mesh was generated around an F-22 fighter aircraft geometry. The mesh is relatively small, consisting of only 61 million cells. This example was generated using 8 Intel X5355 CPUs each with 4GB of memory. Each CPU ran one MPI process. An overview image of the mesh and input geometry is shown in Fig. 11. The off-body grid generator was configured to delete crossing cells and create a small gap region between the geometry and the off-body mesh. There are just under one million facets, which make up this geometry and the off-body technique has no problems with this scale of complexity. Additional slices are shown in Figs. 12 and 13, including close up images of the landing gear. Notice that the gap region does not have a uniform size. The surface is discrete and can have large gaps near concave regions as can be seen for the case in Figure 13.

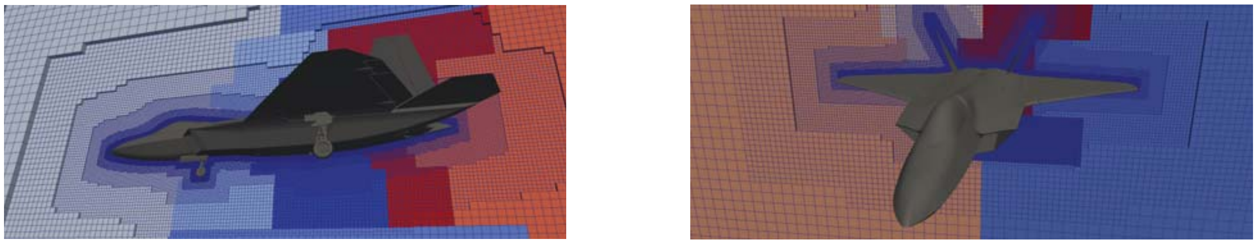


Figure 12. F-22 geometry with off-body mesh generated on 8 processors. Each partitioned mesh domain is assigned a different color.

B. Unmanned Aerial Vehicle

The technique was also used to generate a one billion cell mesh for a Predator UAV geometry. The case completed in under 45 minutes using 64 Intel X5355 CPUs each with 4GB of memory. Each CPU ran one MPI process. A slice of one domain is shown in Fig. 14. This case illustrates some of the problems with very large meshes, it was quite challenging to visualize the mesh. The full mesh could not be loaded onto a standard workstation computer. It requires over 20GB of memory to store just the coordinate locations of the points in the mesh. To overcome this limitation,

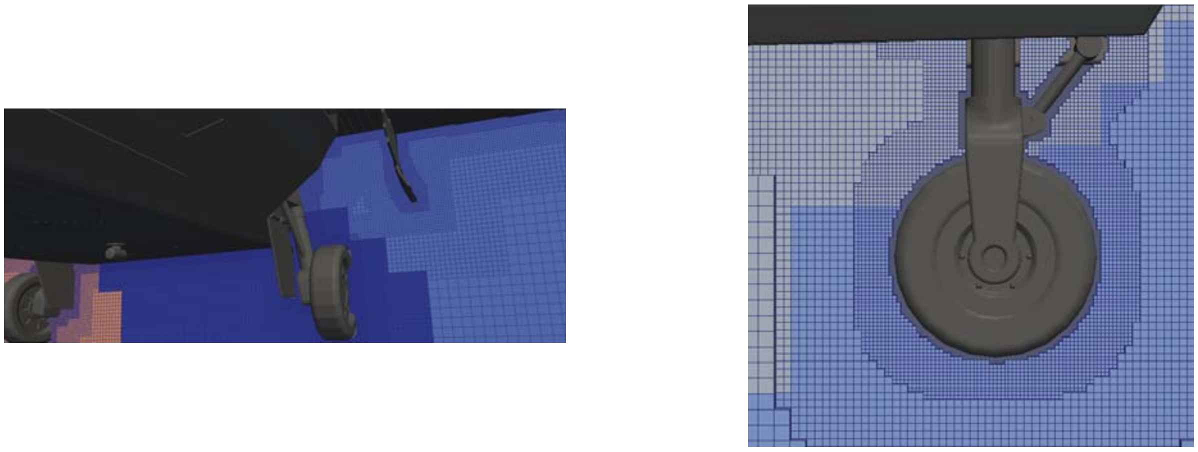


Figure 13. The mesh near the landing gear.

each processor wrote out a separate visualization file for the mesh in its partition. These files could then be loaded one at a time into paraview. Figure 14 is a slice of one of those partitions.

A 1.6 billion cell mesh was generated for this geometry by increasing the thickness of each spacing layer. On the same hardware, this mesh took 2 hours and 12 minutes to complete.

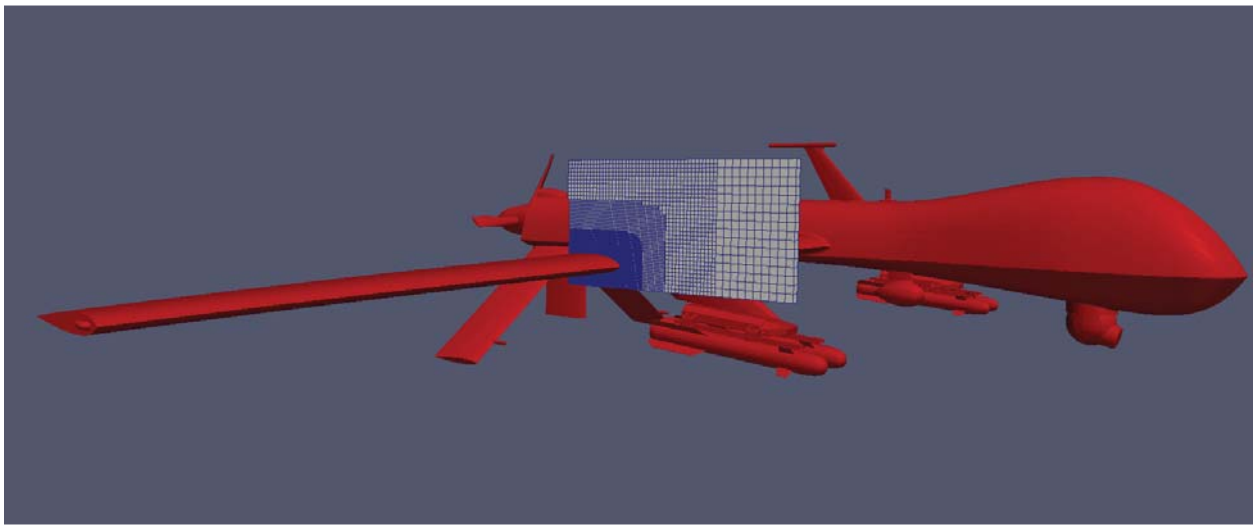


Figure 14. A slice of a billion cell mesh generated for a Predator UAV geometry.

C. Internal Curved Pipe

Internal grid cases are also supported by removing cells marked as external rather than internal during the flood fill step. Figures 15 and 16 show an internal off body mesh of a curved pipe where the external and crossing cells have been removed. Figure 15 shows just the interior mesh, and 16 shows a cut away of the mesh along with the geometry shown in blue. Notice that the internal grid has similar qualities as the external grid. The gap region between the mesh and the geometry is jagged as the discrete mesh approximates the smooth curve of the pipe geometry.

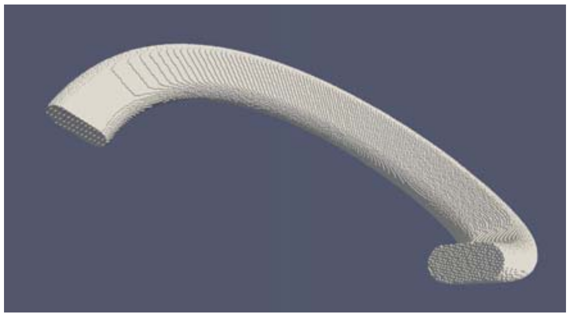


Figure 15. A mesh inside a curved pipe. The external cells and cells that cross the pipe have been removed.

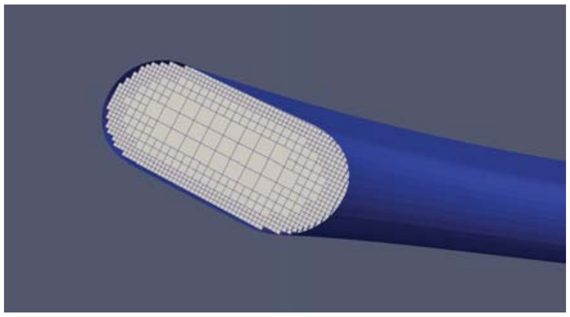


Figure 16. A cut away of a curved pipe geometry (blue) and the interior mesh generated inside of it.

V. Performance Results

One of the most important performance metrics is how the method performs with increasing cell count. There are two tunable parameters to increase the cell count. The first parameter is the minimum cell size requested by the user defining the spacing of cells near the geometry. The second parameter that will increase cell count is the number of cells in a refinement layer. Figure 17 shows the runtime required when increasing the cell count by decreasing the minimum spacing. Figure 18 shows runtime required when increasing the cell count by increasing the number of cells requested in a refinement layer. The method scales linearly when increasing cell count by decreasing the minimum cell size, but the method does not scale linearly when increasing the thickness of refinement layers. Decreasing minimum cell size is a local operation requiring only local subdivision. However, increasing the thickness of refinement layers is a non-local operation. There is additional work for each cell in a refinement layer. Each cell at a given depth creates an extent box where spacing of the mesh must be enforced.

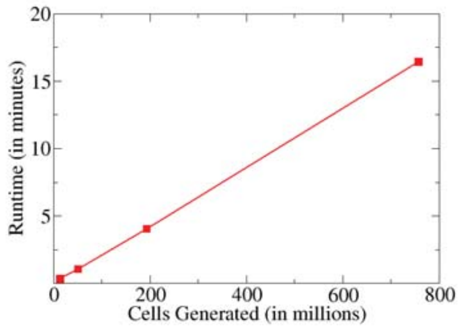


Figure 17. Runtime scalability for increasing cell count by decreasing the minimum cell size.

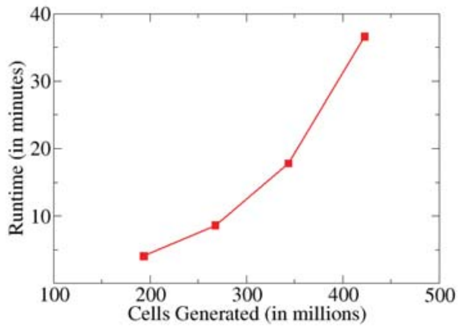


Figure 18. Runtime scalability for increasing cell count by increasing the thickness of refinement layers.

Another important aspect of performance is the parallel scalability. It is desirable for parallel algorithms to have scalable performance to make efficient use of compute clusters. Figure 19 shows parallel speedup for a strong scaling case as the number of CPUs is increased. The mesh generated contained 50 million cells around the UAV geometry. This 50 million cell mesh is representative of the largest mesh that could be achieved on one processor using the hardware mentioned above. It took 22 minutes to generate this mesh on a single processor. While runtime decreased to 117 seconds using 32 processors, the parallel performance is not ideal. Factors limiting performance are discussed later in the section.

Figure 20 shows scalability for a 100 million cell mesh generated with between 2 and 64 processors. Speedup was computed by comparing the parallel run times to the serial mesh generation. While running in serial, the algorithm does not need to perform partitioning of the base mesh. Partitioning the base mesh requires building up connectivities needed to call METIS, calling METIS to generate the partition and then deleting voxels from the local tree that are owned by non-local processors. This series of steps accounts for about 16% of the total run time while running in parallel.

Parallel performance is also hindered by poor load balancing since the initial base mesh is coarse. All cells in the

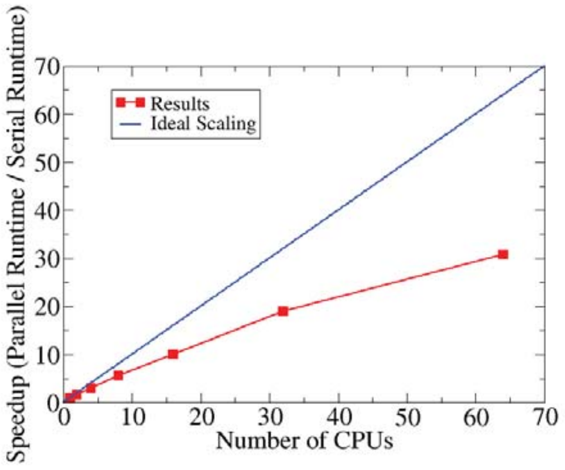


Figure 19. Strong scalability results for generating a 50 million cell mesh.

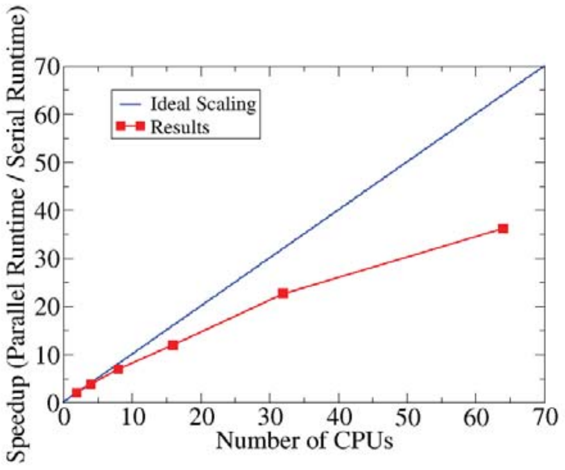


Figure 20. Strong scalability results for generating a 100 million cell mesh only comparing parallel runs.

base mesh are given the same weight when the base mesh is partitioned. However, each cell in the base mesh does not represent an equal amount of work to be performed during the parallel grid generation. Figures 21 and 22 show the amount of time spent working while generating a 100 million cell mesh around the UAV geometry. It is clear that the partitioning did not yield ideal load balancing. Generating the base mesh took 2 seconds for each case while the full mesh using two processors took just over 50 minutes. Redundant generation of the base mesh is not the limiting factor for parallel scalability. It is the poor load balancing that comes from the base mesh that is the limiting factor.

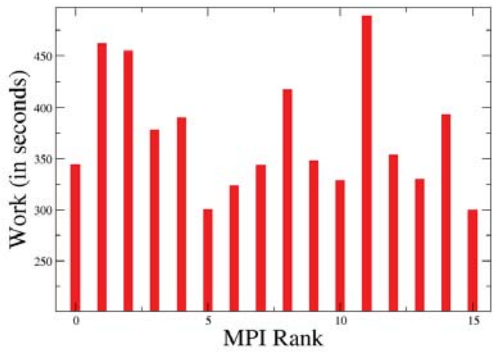


Figure 21. The time spent working while generating a 100 million cell mesh while running on 16 cores.

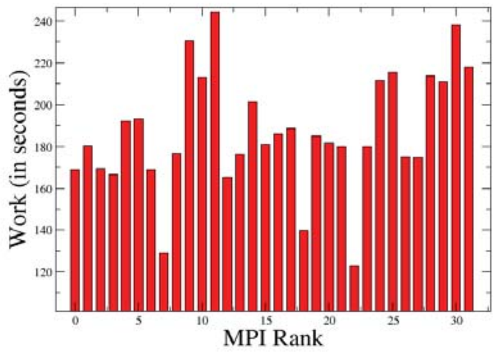


Figure 22. The time spent working while generating a 100 million cell mesh while running on 32 cores.

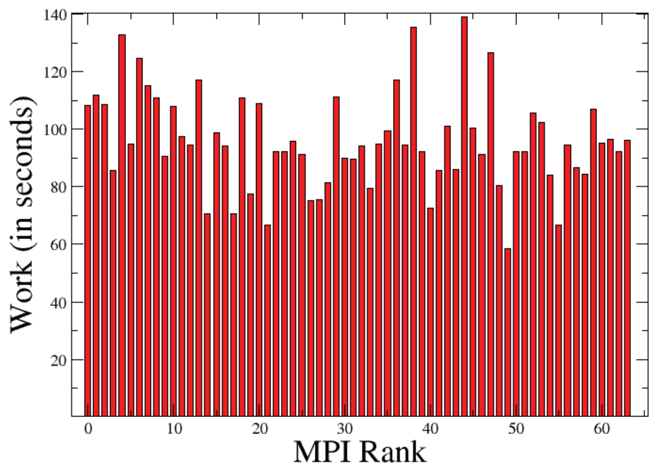


Figure 23. Work load for each of the 64 processors while generating a 100 million cell mesh.

It may be possible to estimate the amount of work required for each cell in the base mesh and generate a better initial partitioning, but a more robust solution would be to increase scalability by using dynamic load balancing. Additionally, parallel speedup is not always the primary goal of parallel grid generation. Sometimes a grid generator must be parallel to generate large meshes where the memory requirements grow so large they cannot be met within a standard single machine.

The results discussed here only pertain to the parallel performance of the off body implementation. For high fidelity simulations a near body technique must also be applied. Repartitioning may be required to achieve high levels of parallel performance for the near body technique. The partitioning scheme used here treats all cells equally and makes no special consideration for cells, which are adjacent to geometry. Because of this, it is likely that the load balancing of cells adjacent to the geometry will be poor. However, it should be possible to separate the near body algorithm from the off-body and run the near body technique on a partition better suited for the chosen near body technique.

VI. Conclusion

This paper has presented a parallel grid generation technique for off-body mesh generation. This technique combines both top down and bottom up approaches to oct-tree based grid generation to produce all hexahedral meshes. Results were presented demonstrating high quality off-body meshes for aerospace applications with minimal user input. Parallel performance results were presented. The method is capable of running on standard compute cluster hardware. Despite lacking ideal parallel scalability, the method is found to be fast. The practicality of the method was demonstrated by generating one billion cell meshes on complex geometries in under an hour on 64 processors.

References

[1]Ishida, T., Takahashi, S., and Nakahashi, K., "Fast Cartesian Mesh Generation for Building-Cube Method using Multi-Core PC," AIAA Paper 2008-0919, 2008.

[2]Ishikawa, N., Sasaki, D., and Nakahashi, K., "Large-scale Distributed Computation Using Building-Cube Method," AIAA Paper 2011-0754, 2011.

[3]Dawes, W., Harvey, S., Fellows, S., Eccles, N., Jaeggi, D., and Kellar, W., "A Practical Demonstration of Scalable, Parallel Mesh Generation," AIAA Paper 2009–0981, 2009.

[4]Karman, Jr., S. L. and Betro, V. C., "Parallel Hierarchical Unstructured Mesh Generation with General Cutting," AIAA Paper 2008–0918, 2008.

[5]Aftosmis, M. J., Berger, M. J., and Melton, J. E., "Robust and Efficient Cartesian Mesh Generation for Component-Based Geometry," AIAA Paper 1997–196, 1997.

[6]Karman, Jr., S. L., "SPLITFLOW: A 3D unstructured Cartesian/prismatic grid CFD code for complex geometries," AIAA Paper 1995–0343, 1995.

[7]Lahur, P. R., "Automatic Hexahedra Grid Generation Method for Component-Based Surface Geometry," AIAA Paper 2005–5242, 2005.

[8]Betro, V. C., "Fully Antisotropic Split-Tree Adaptive Refinement Mesh," AIAA Paper 2011–0895, 2011.

[9]David, T. T., Ohallaron, D. R., and Ghattas, O., "Scalable Parallel Octree Meshing For Terascale Applications," *in SC2005*, 2005.

[10]Burstedde, C., Wilcox, L. C., and Ghattas, O., "`p4est`: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees," *SIAM Journal on Scientific Computing*, Vol. 33, No. 3, 2011, pp. 1103–1133.

[11]George, K. and Kumar, V., "A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, Vol. 20, No. 1, 1999, pp. 359–392.