A PARALLEL MESH GENERATOR BASED ON SEQUENTIAL NETGEN

by

Yusuf Yılmaz

B.S, Computer Engineering, Boğaziçi University, 2009

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2013

ACKNOWLEDGEMENTS

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-211528 and FP7-261557. The work was achieved using the PRACE Research Infrastructure resources (the Curie supercomputer at CEA in France). I thank Prof. Oğuz Tosun, Assoc. Prof. Ali Haydar Özer and Seren Soner for their various contributions. I also thank the Elmer application developers Peter Raback and Mika Malinen from CSC, Finland for their help on Elmer related questions and testing of the generated meshes.

I would also like to thank my thesis supervisor Can Özturan for his guidance and support and patience throughout the preparation of my thesis.

As last I would like to thank everyone working in the video game industry. Without them I would never find the ambition and motivation to become a computer scientist.

ABSTRACT

A PARALLEL MESH GENERATOR BASED ON SEQUENTIAL NETGEN

A parallel tetrahedral mesh generator is developed using the existing sequential NETGEN mesh generator. Mesh generation algorithms developed decompose the geometry or volume mesh into multiple sub-geometries or sub-meshes sequentially on a master node and then create fine volume meshes from those sub-geometries and sub-meshes in parallel on multiple processors. Three methods are implemented. The first decomposes the geometry and produces conforming surface sub-meshes from which volume meshes can be generated in parallel. The second and third methods which are refinement based also make use of the CAD geometry information. A scalable mesh migration algorithm that utilizes "owner updates" rule is implemented. Results show that using refinement based methods; a mesh with over a billion volume elements can be generated in under a minute. Our developed software is also distributed freely as open source code at the address http://code.google.com/p/parallel-netgen/.

ÖZET

SIRALI NETGEN BAZLI PARALEL GRİD ÜRETİMİ

Paralel tetrahedral grid üreticisi mevcut sıralı NETGEN grid üreticisi kullanılarak geliştirilmiştir. Geliştirdiğimiz grid oluşturma algoritmaları bir ana düğüm üzerinde geometriyi veya hacimsel gridi alt geometrilere ve alt-gridlere ayrıştırıyor ve sonrasında birden çok işlemci üzerinde paralel olarak her alt-geometri veya alt-grid üzerinde daha ayrıntılı hacimsel grid üretimi gerçekleştiriyor. Temel olarak üç yöntem uygulanmaktadır. Birinci yöntemde geometri parçalanır ve yüzeysel alt-gridler üretilir. Daha sonra bu alt-gridlerden ayrıntılı hacimsel grid paralel olarak üretilir. Grid inceltme bazlı yöntemlerimizde (Metod 2. ve 3. yöntemlerinde) ayrıca CAD geometri bilgileri kullanır. "sahip günceller" kuralı kullanan grid göç algoritması da uygulanmıştır. Sonuçlar, inceltme tabanlı yöntemler kullanarak, bir dakika altında bir milyardan fazla elemandan oluşan bir gridin elde edilebileceğini göstermektedir. Geliştirilen yazılım ücretsiz açık kaynaklı kod olarak http://code.google.com/p/parallel-netgen/ adresinden temin edilebilir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS/ABBREVIATIONS

2D	Two Dimensional
3D	Three Dimensional
API	Application Programming Interface
BSP	Binary Space Partitioning
CAD	Computer Aided Design
CEA	French Government Funded Research Organization for Nuclear and Alternative Energy Sources
CFD	Computational Fluid Dynamics
FEA	Finite Element Analysis
FEM	Finite Element Method
FMDB	Flexible Distributed Mesh Database
GPU	Graphics Processing Unit
HPC	High Performance Computing
ITAPS	Interoperable Technologies for Advanced Petascale Simulations
LGPL	Lesser General Public License
METIS	A Sequential Mesh Partition Software
MPI	Message Passing Interface
MeshSim	Commercial Mesh Generation Software
NETGEN	Open Source Sequential Mesh Generator
OpenMP	Open Multiprocessing
PCA	Principle Component Analysis
PMDB	Parallel Mesh Database
PRACE	Partnership for Advanced Computing in Europe
PT-Scotch	A Parallel Mesh Partitioner
ParMETIS	A Parallel Mesh Partitioner
ParTGen	A Parallel Mesh Generator
SCOREC	Scientific Computation Research Center
SMT	Simultaneous multithreading

STL	Standard Template Library
SciDAC	Scientific Discovery through Advanced Computing
UFO	Unidentified Flying Object

1. INTRODUCTION

The main objective of this work is to develop a software to generate large unstructured meshes with sizes in the hundreds of millions range on complex geometry. With sequential mesh generators, memory on a single node becomes a serious bottleneck allowing generation of only a few tens of millions elements. While generating huge meshes, another objective is to reduce the mesh generation time drastically. These objectives need to be achieved in order to petascale unstructured mesh solvers especially for complex geometry problems since generation of a massive unstructured mesh is a required pre-processing step of the finite element methods. Achievement of these objectives requires a scalable parallel mesh generator, which is the topic addressed in this work.

To quote Chrisochoides [1]: "It takes about 10-15 years to develop the algorithmic and software infrastructure for sequential industrial strength mesh generation libraries". Therefore, given a limited time frame on our part, the following approach is followed: Take an existing sequential mesh generator and parallelize it by decomposing the geometry for parallel mesh generation. As sequential mesh generation software, NETGEN [2,3] mesh generator is used due to its availability as LGPL open source software and its wide user base. NETGEN was developed mainly by Joachim Schöberl at the Johannes Kepler University Linz. Our parallel mesh generation routines are implemented using the MPI communication libraries and the C++ language.

Our work was funded by PRACE. During the discussions we became aware of the need for mesh generators that could create meshes with over a billion elements. One of the problems other scientists mentioned was that to do that many of them had to use some kind of refinement library in conjunction with sequential mesh generators and mesh repartitioning tools. The resulting meshes should also satisfy the condition that vertices at the boundaries should be exactly on the geometry. We wanted to address this problem and see whether those could be integrated in a way so that it would be possible for a scientist to generate the required mesh with a single API call.

The main contributions of the thesis can be listed as follows:

- Meshes with billion elements can be generated using one of the methods we implemented in around a minute time. As mentioned earlier due to memory limitations it is practically impossible to generate tetrahedral meshes with around a billion volume elements in currently available hardware. Only some fat node configurations may enable that and even in this case a sequential mesh generator would take a very long time to generate a mesh with that size due to the non-linear complexity of the mesh generation problem. So currently, this can be achieved by using parallel refinement. To the extent of our knowledge, currently, there is no open source solution to generating large meshes in a short time in a single step. Most scientists do use different mesh generation, mesh partitioning, migration and mesh refinement libraries in various combinations to generate meshes required for their research. This also adds additional labour time to the process.
- The generated meshes do match the geometry. This means that during the refinement steps, new vertices are generated to snap at the original geometry. Since the mesh generator library is integrated with our refinement code, during refinement steps we do have access to the original geometry. This allows us to generate vertices on the boundaries according to the underlying geometry. Higher quality elements at the boundary are generally needed by physical simulators like FEM Solvers. In case the mesh is used in computer graphics this problem is even more important. In that case practically refinement without matching geometry is useless and does not bring anything to the render.
- Robust and scalable mesh migration code is implemented. This allows us to move the generated partitioned meshes around if needed. For different reasons the researcher might want to repartition the generated mesh. He might have non-standard hardware, or his solver may not scale very well, yet he would still want a large mesh in a short amount of time. This can be achieved by using ParMETIS with our integrated migration function easily.
- Our work integrates one of the most widely used sequential mesh generator and a parallel mesh partitioner which are available as open source.

- The generated meshes are well balanced. Due to the nature of the refinement algorithm if we start with a well balanced mesh, we end up with a well balanced mesh. This is also the case in our work. This would mean that the user of the program would not need to post-process the generated mesh and directly use it as it is in his/her research. Well balanced meshes do also work well in rendering jobs like for example ray tracers (more or less equal sized terminal nodes with the same amount of processing requirement), or video games (sustained fps levels which are a very important feature for end-users).
- By writing interface code to the required application one can use the program like API calls. This would enable the researcher to use his valuable time for something else rather than manually trying to create a mesh in desired size and time. This way the researcher can use slightly different geometric models, with variable processor numbers in an efficient way.

In the rest of the thesis, a survey of related work is given in Chapter 2. Chapter 3 briefly describes the mesh data structures used in our different implemented methods. Chapter 4 presents the methods that were used to generate a mesh in parallel using the sequential NETGEN. After the mesh generation, repartitioning and migration of the mesh to final destination needs to be carried out; Chapter 5 addresses these topics. Performance results of the developed parallel mesh generation routines and migration are reported in Chapter 6. Finally, the thesis concludes with a discussion of results in Chapter 7.

2. BACKGROUND WORK

In this chapter, we first present background work on mesh generation. Then, we review supercomputer architectures and their programming environment. Finally, we review the literature work on parallel mesh generation.

2.1. Mesh Generation

Mesh generation is generating polyhedral meshes that approximate geometry. Mostly they are used for rendering purposes in various entertainment products like animations, movies, video games, etc. They are also extensively used by scientists in physical simulations like FEA or CFD. For rendering purposes 3D surface meshes generally suffice, yet for simulations volumetric meshes are used. There are many different methods to generate meshes, but mostly the following methods are used.

Delaunay based mesh generation is one method to generate meshes. This method uses Delaunay triangulations which maximize the minimum angle of the triangles created. And this is generally a good idea if you do not want to generate triangles (in case of 2D meshes) or tetrahedron (in case of 3D meshes) with very small angles [21]. This method works by constructing Delaunay triangulations and inserting new vertices at locations preferably far from existing ones to prevent short edges. Most delaunay based methods create higher quality elements far from the boundaries [21]. Triangle [24] and Gmsh [25] are mesh generators utilizing this method.

Advancing front methods are another way to create meshes. This method adds elements one by one. It starts from the boundaries and advances inward. As long as there is unmeshed territory the front continues advancing, and updating itself while creating new elements along the way. Higher quality elements are generally found near the boundary [22]. The sequential mesh generator we are using called NETGEN implements both the advancing front and delaunay techniques.

Another method is using quadtrees and octrees to generate meshes in 2D and 3D respectively. In this method a background grid (octree) is placed on the geometry and triangulation occurs in each of the cells of the grid. Generally this method may create graded meshes and therefore a further optimization step is needed. Camino [26] is an example of this type of mesh generator.

The website at [23] has a very extensive list of public domain and commercial mesh generators of different types.

2.2. Supercomputer Architectures and Their Programming

Since our program is designed to run on supercomputers, it would be a good idea that we give some information about it.

We can briefly describe supercomputers as systems with very large number of processors. They generally define the standard for peak processing capacity in today's computers. Mainly we can divide them into two categories; grids and clusters. In our work we used a cluster supercomputer called Curie. Compute clusters are systems where a large number of processors are located in a large room, and connected by high-speed local area networks. Processors in newer supercomputer clusters are contained in nodes which mostly contain (but not limited to) 4-64 cores and a large shared memory within the node. Memories between nodes are not shared and require communication. Thus this kind of memory and network architecture requires programmers to use message passing library like MPI for node-to-node communication and multithreading libraries/tools like Pthreads and OpenMP within the processors that reside in the same node. In our work, we have only used MPI with C++. MPI basically utilizes functions that enable communication between processors with generally distributed memories in a wide variety of different cluster topologies. There are some free and some commercial implementations of it. We used mpich2, which is one free implementation. Utilizing OpenMP might be an additional future work for us. Curie has 16 cores in a thin node which we used and utilizing the shared memories within the nodes might give an additional performance bump to our program.

As previously mentioned, we used the Curie supercomputer in France for making test runs of our software. Curie currently has three different kinds of nodes as shown in Figure 2.1. Mainly fat nodes have a bigger memory per node; thin nodes have adequate memory per node for jobs that do not require a single node to have a heavier memory load like pre or post-processing data. There are also hybrid nodes that utilize the new GPU based systems [18]. Curie had at the time we did the work the 9th raw processing power in the world and the best in Europe, and currently it is ranked at 11th place according to the top 500 list [19].

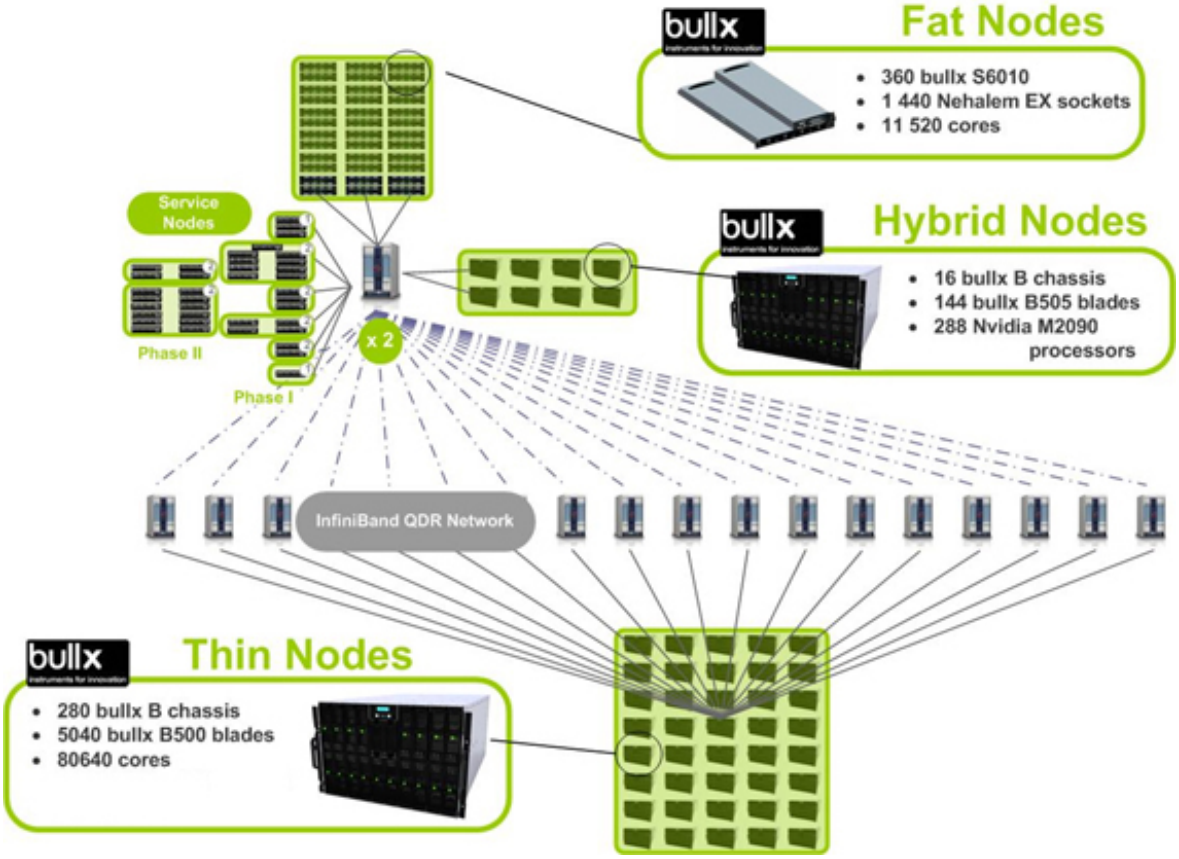

Figure 2.1. The Curie supercomputer architecture used in this thesis [18].

Our work is made possible with funding from the PRACE project. Most part of this work is done for PRACE. PRACE is short for Partnership for Advanced Computing in Europe and its basic goals are improvement of the HPC infrastructure in Europe and provision of easy access to education, training and high-cost hardware to improve scientific research in Europe [20].

Table 2.1. June 2011 Top 10 most powerful supercomputers at the time of our work.

Rank	**Site**	**Manufacturer**	**Computer**	**Country**	**Total Cores**	**Linpack (Tflops)**	**Peak (Tflops)**
1	RIKEN Advanced Institute for Computational Science (AICS)	Fujitsu	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect	Japan	548352	8162000	8773630
2	National Supercomputing Center in Tianjin	NUDT	NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C	China	186368	2566000	4701000
3	DOE/ SC/ Oak Ridge National Laboratory	Cray Inc.	Cray XT5-HE Opteron 6-core 2.6 GHz	United States	224162	1759000	2331000
4	National Supercomputing Centre in Shenzhen (NSCS)	Dawning	Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU	China	120640	1271000	2984300
5	GSIC Center, Tokyo Institute of Technology	NEC/HP	HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows	Japan	73278	1192000	2287630
6	DOE/ NNSA/ LANL/ SNL	Cray Inc.	Cray XE6 8-core 2.4 GHz	United States	142272	1110000	1365810
7	NASA/ Ames Research Center/NASA	SGI	SGI Altix ICE 8200EX/8400EX, Xeon HT QC 3.0/Xeon 5570/5670 2.93 Ghz, Infiniband	United States	111104	1088000	1315330
8	DOE/ SC/ LBNL/ NERSC	Cray Inc.	Cray XE6 12-core 2.1 GHz	United States	153408	1054000	1288630
9	**Commissariat a l'Energie Atomique (CEA)**	**Bull SA**	**CURIE, Bull bullx super-node S6010/S6030**	**France**	**138368**	**1050000**	**1254550**
10	DOE/NNSA/LANL	IBM	BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband	United States	122400	1042000	1375780

2.3. Parallel Mesh Generation

Parallel mesh generation is difficult to implement because it involves complicated tight integration of many components, such as those from computational geometry, unstructured distributed mesh data structures, load balancing and mesh partitioning. Since the whole geometry needs to be considered for quality meshing, a sequential or tightly coupled component that operates on the whole geometry is necessary which in turn forms a bottleneck. Mesh generation can be parallelized using two approaches: one is to directly parallelize the mesh generation algorithm and the other is to decompose the geometry so that sequential mesh generators can be used on each sub-domain. Delaunay based approach of [4] directly parallelizes the mesh generation at all levels on multicore SMT-based architectures. D3D [5] is another direct approach that is implemented in MPI and that employs octree based mesh generation. A commercial multi-threaded and distributed parallel mesh generator MeshSim by Simmetrix Inc. [6] is also available. ParTgen [7] is a parallel mesh generator that utilizes the second approach, i.e., that of a geometry decomposition based approach to decouple the meshing process into parallel mesh generation tasks. The code of ParTgen is not open source. Also, performance results of ParTgen have demonstrated scalability only up to 32 processors. Our implementation that is described in Chapter 3 uses a similar approach to that of ParTgen.

Interoperable Technologies for Advanced Petascale Simulations (ITAPS) [8,9] is a recent U.S. based initiative involving various laboratories and universities. ITAPS forms one of the centres in the U.S. Department of Energy's Scientific Discovery through Advanced Computing (SciDAC) program. ITAPS aims to develop interoperable and interchangeable mesh, geometry, and field manipulation tools. In particular, the roots of Flexible Distributed Mesh Data Base (FMDB) tool [10] provided by ITAPs can be traced to the SCOREC's parallel mesh database (PMDB) [11] that included an owner updates rule based mesh migration routines [12] and tested it on the legacy parallel systems in mid 1990s. In this project, an improved version of this algorithm was implemented that operates on the tetrahedron-to-vertex and boundary face-to-tetrahedron connectivity data structures.

Mesh partitioning algorithms partition the mesh in such a way that load balancing is achieved while communication is minimized. Sequential partitioners Metis [13] and Scotch [14] as well as their parallel versions ParMetis and PT-Scotch are widely used in the parallel computing community. In this work, a repartitioning of the distributed mesh needs to be done after the mesh generation. In order to do this, ParMetis is used. Mesh migration is needed after repartitioning in order to migrate the tetrahedra to their final destinations provided by the partitioners. Zoltan [15] is another tool that provides parallel partitioning, load balancing and data management services for unstructured meshes.

3. DATA STRUCTURES USED

For this project we used mainly two different mesh data structures. For the first method that uses geometric decomposition we used BSP-Trees. This mesh structure is being used in a tree structure, so as to let efficient access to its elements during tree traversal. Although it contains a vertex, triangle and tetrahedron list, the tetrahedron list is mainly used after the geometric decomposition is done and volume meshes are generated from the partitioned surface meshes. We will explain the structure in Chapter 3.2 and also how, where and when it is used in Chapter 4.

The second mesh structure is used in both the second and third methods. A format similar to Elmer partitioned meshes is used. We needed minimal communication among the processors during the refinement steps and also wanted the independent part of the mesh refinement to be as fast as possible. Therefore, a minimalistic structure design that only contains the essential data is selected. For that purpose edge elements are left out, and mostly one-way element to element relations are kept.

3.1. Mesh Data Structure

The input to the developed parallel mesh generator is a geometry file in NETGEN's .geo format. As output, we use Elmer mesh file format [16], since the mesh generated is intended to be used by the Elmer multi-physics simulation software. Internally also, the parallel mesh generator stores mesh data in a similar structure as that of Elmer's mesh file format. Figure 3.1a shows the compact mesh data organization that is used. Tetrahedron-to-vertex adjacency relation is stored for the whole mesh as an array. Arrays of face-to-tetrahedron and face-to-vertex relations are kept only for faces on geometry and partition boundaries. In other related work such as [11], the rich relations shown in Figure 3.1b are used on the whole mesh at the expense of heavy memory usage.

We can classify distributed mesh entities (faces and vertices) making up a tetra-

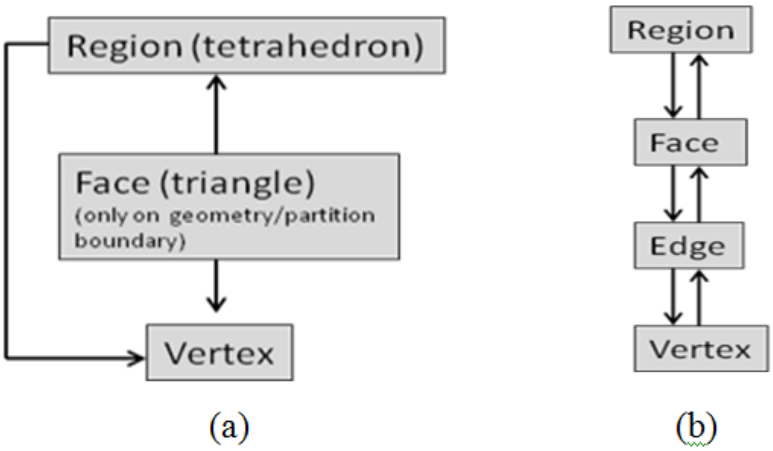

Figure 3.1. Mesh entity adjacencies stored in this project (a) and in other related work [11] (b).

hedron as follows:

- Interior entities: entities that uniquely reside on a partition (processor), i.e. non-shared entities.
- Partition boundary (shared) entities: entities that are shared by multiple partitions. Since we have a distributed memory model, these entities are replicated on the processors that own the tetrahedra containing these entities.

As in [11,12], we designate one of the processors that hold replicated shared entities as the owner of that entity. Since the ownership information is used in mesh migration, we discuss the details of how partition boundary entities are kept and how their ownerships are determined in the mesh migration chapter (Chapter 5).

3.2. Tree Data Structure

In this part we will explain the tree data structure and the mesh structure that resides in it. What we actually did, is take a standard BSP-Tree and use a modified

computer graphics mesh as its primitives. A standard computer graphics mesh generally contains a vertex and face list. We added a tetrahedron list to it that enabled us to use it not just for surface meshes, but also volume meshes. Also, we modified the vertex, face and tetrahedron elements to hold a processor list that they belong to. Since we also matched the processor count with the number of terminal nodes of the tree, this way, we could think of each terminal node of the tree as holding a sub-mesh which will later reside in its matching processor. The processor lists that are embedded into the element structures are used to determine whether they are shared and help in calculating the partition boundary between processors. Figure 3.2 illustrates the changes to a standard computer graphics mesh structure. For the purpose of achieving this following classes were implemented.

- The TreeNode: The tree node contains a bounding box, depth level info, splitting plane, splitting coordinate, a list of primitives, and right and left node pointers. In addition to that, we also use a partition ID variable and partition name only on end nodes (leaf nodes) to match their data with processors for parallel processing.
- MeshVertex Class: This holds mesh vertex information. It has position and normal information and additionally its index in the vertex list of the parent mesh. It also holds a global ID of itself, and a list of processor ID's that the element will be in. The standard get and set functions for its private variables are also implemented. In addition to a standard tree mesh vertex class it also has a function to add processor ID's to its processor list and also a boolean function that checks whether the vertex is shared or not.
- MeshTriangle: This holds information for a mesh triangle. Pointer to three vertices, pointer to its parent mesh, its ID, and a list of processor ID's are its elements. It has a get bounding box function which finds the minimum and maximum x, y, z coordinates of its vertices. In addition to a standard tree mesh triangle class it also has a function to add processor ID's to its processor list and also a boolean function that checks whether the vertex is shared or not. In our case, since we are only interested in boundary face elements and not all face elements, this function is not actually used.

- MeshTetrahedron: This holds information for a mesh tetrahedron. Pointer to four vertices, pointer to its parent mesh, its ID, and a list of processor ID's are its elements. It has a get bounding box function which finds the minimum and maximum x, y, z coordinates of its vertices. In addition to a standard tree mesh tetrahedron class it also has a function to add processor ID's to its processor list and also a boolean function that checks whether the tetrahedron is shared or not. In our case since we partition the meshes by volume elements and not by actual planes, again this function is not used. (This does not mean that we do not use the cut planes, but means that we determine which child node the element goes not by its bounding box, but by the position of its centroid (mean value of its vertices x, y, z and w).
- Mesh Class: Contains a mesh object. It has a vertex, triangle and a tetrahedron list as variables. Get, set functions and add functions are present.

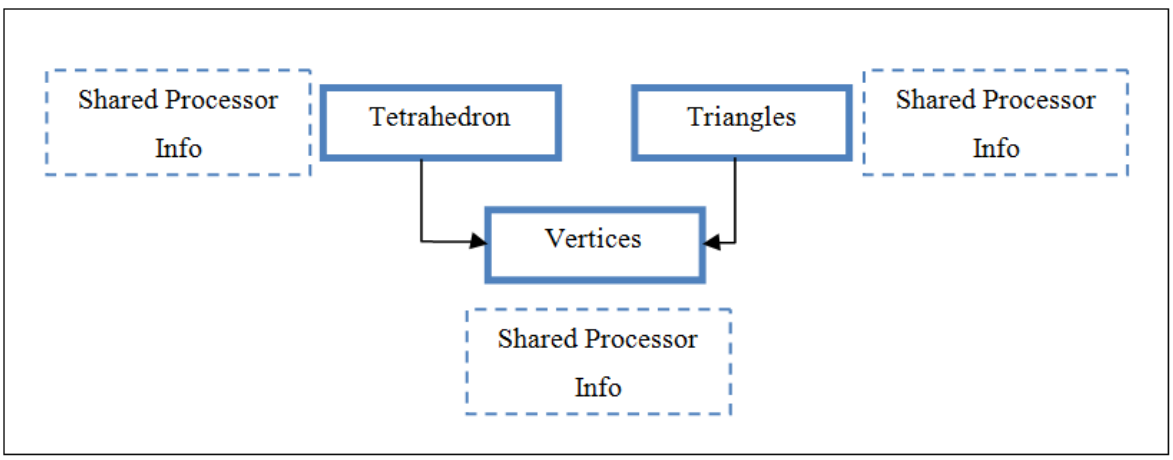

Figure 3.2. Mesh class entity relations that are used in the Tree class.

4. PARALLEL MESH GENERATION ALGORITHMS

Three main methods are used in order to parallelize the mesh generation process. The first of them utilizes geometry decomposition and the other two methods use parallel refinement. With geometry decomposition not the mesh, but its geometry is partitioned, and then from these sub-geometries meshes are generated in parallel. At the end of the algorithm since the generated meshes should be part of a larger mesh, the global ID's of its elements are to be reassigned to make up a partitioned conforming mesh. This is explained more thoroughly in Chapter 4.1. The second and third methods that were implemented make use of parallel refinement techniques. The second uses volume mesh refinement, whereas the third one uses surface mesh refinement. At the beginning of both these methods a coarse volume mesh is partitioned using ParMETIS, and in the case of the third method the mesh parts are further skinned, which means that the volume elements are gotten rid of. The second method ends after multiple (as many as needed) refinement steps. Since the third method uses surface meshes for refinement, at the end of the refinement steps, a volume mesh generation of the surface mesh parts is done. The details of these methods can be found accordingly in Chapters 4.2 and 4.3.

4.1. Mesh Generation Using Geometry Decomposition

The first method (Method 1), given in Figure 4.1, decomposes the geometry into multiple sub-geometries sequentially by using binary space partitioning (BSP) trees (like the KD-tree partition planes) on a coarsely generated mesh, and then uses NETGEN to generate fine volume meshes from these sub-geometries in parallel. When a mesh is generated for each sub-geometry by each processor, the partition boundary mesh faces of adjacent sub-geometries need to be conforming (i.e. matching). This is achieved during the initial sequential phase on processor 0 which generates conforming surface mesh for sub-geometry faces. These surface meshes are sent to each processor in step 5. Load balancing is achieved by the use of the coarse mesh generated in step 1. BSP tree partitions the coarse mesh into parts with equal number of tetrahedra

which roughly balances the load that each processor will encounter while generating the large fine volume mesh. Figure 4.2 demonstrates execution of Method 1 on the torus geometry.

Steps	**Method 1**
	On processor 0 **do**.
1	Generate course mesh sequentially by NETGEN.
2	Decompose the geometry by cutting planes obtained from a BSP-Type method like KD-Tree.
3	Generate fine surface mesh of the decomposed geometry.
4	Partition the surface mesh.
5	Distribute the partitioned surface meshes to other processors.
	On all processors **do**.
6	Generate volume mesh from the partitioned surface meshes.
7	Perform global ID assignment and matching.
8	Repartition the mesh using ParMETIS.
9	Migrate the distributed mesh according to ParMETIS output.
10	Save the distributed mesh.

Figure 4.1. Algorithm for Method 1.

The parallel steps of Method 1 are carried out as follows: In step 6, we just use a NETGEN method which accepts a face list (list containing triangles representing a closed surface) to generate a volume mesh. Then, in step 7 global ID assignment and matching is done. Global IDs are assigned to tetrahedra and vertices. Faces do not have global IDs. The owner processor is responsible for making the global ID assignment. By global ID matching, we mean the following: Since NETGEN generates a new volume mesh for each closed surface mesh which resides in each processor; it also assigns its own IDs to tetrahedra, faces and vertices. What we do is just create a map which relates the global vertex IDs to the local IDs since the closed surface mesh we pass to the NETGEN method has the vertices locally numbered from 1 to N. After Netgen creates the volume mesh, we just swap the local IDs of the vertices which are

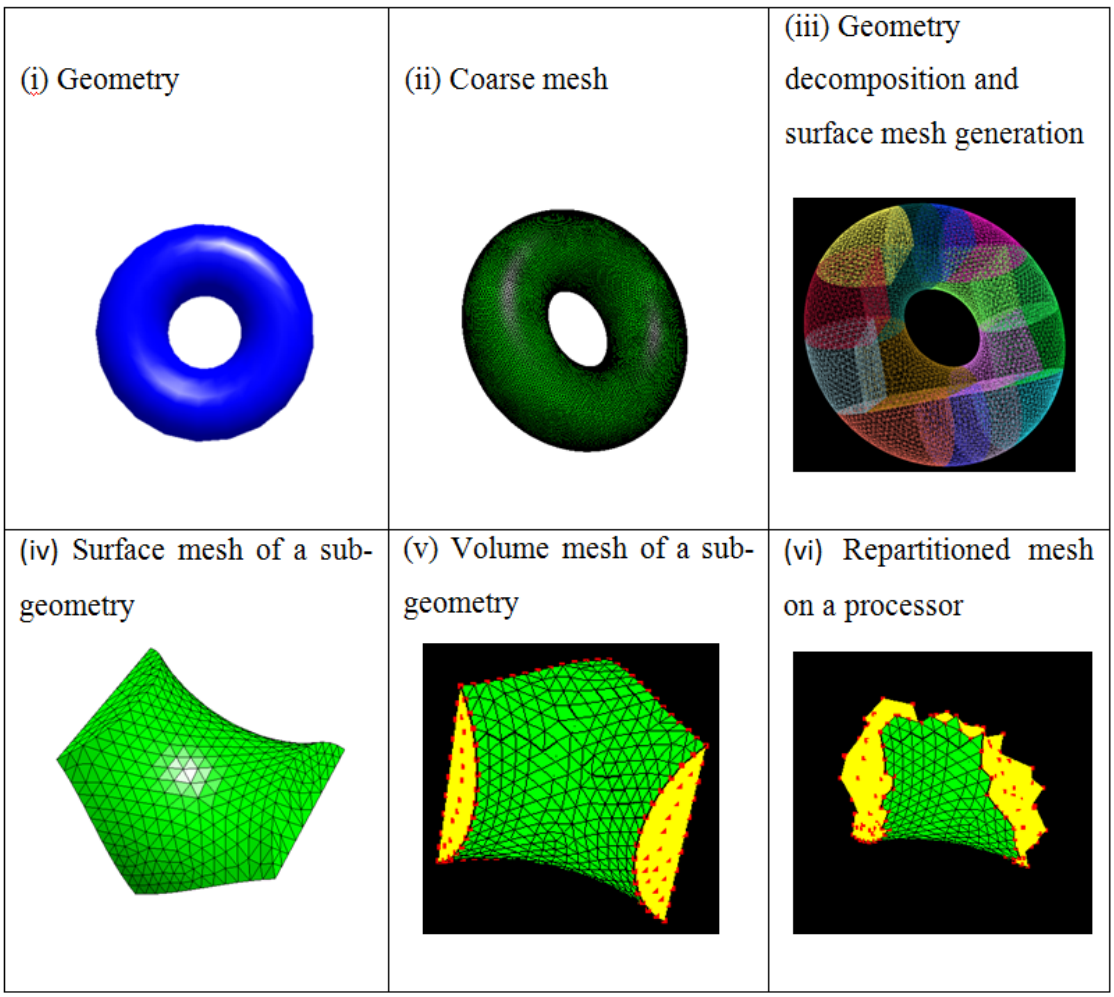

Figure 4.2. Mesh generation by Method 1 (geometry decomposition).

on the surface of the newly created volume mesh according to the map. Step 8 does parallel repartitioning with ParMetis. Step 9 migrates tetrahedra using the algorithm given in Chapter 5.

4.1.1. Geometry Decomposition

The methods we used for testing are Octree, KD-Tree, BSP-Tree using PCA, BSP-Tree using random selection. The following can be said about them.

- Octree, KD-Tree and PCA are known methods.
- With the last method (BSP using random selection) we mean the following. (The decision of the cut plane is made as follows)
  (i) While (End Criteria)
  (ii) Select a random triangle from the current node.
  (iii) Use the normal of the triangle as the cut plane normal.
  (iv) Calculate the number of elements left and right of the plane.
  (v) End criteria may be a certain equality rate (49%-51%) or best split up to N steps (10-20).
- The method is pretty fast and robust and yet not perfect (KD-Tree and BSP with PCA calculate a near perfect %50-%50 splits).
- With the exception of the Octree method, which is inferior for this application the other three algorithms perform similarly quality-wise.
- Since PCA is a costly method and does not bring much in this case, for test runs we used KD-Trees and BSP-Trees with random selection.

In this part, we will explain the process of with which the BSP-Tree is implemented. We will not discuss the selection of cut-planes, so this part is actually the same for all the above four mentioned algorithms.

In general we could divide the tree class into two main parts, one of it is the tree building functions, and the other is traversal functions.

The building process can be further divided into 4 parts.

(i) Main build function: This function starts the building process. It calculates the bounding box (from now on called BB) for the whole mesh. And then calls the main recursive function.

(ii) Recursive Function: The algorithm of the main recursive function is as follows.
- Check for end condition. If reached return. Although different end conditions can be implemented, here we used two things. The function ends if the expected depth level or a minimum number of elements we want on a node is reached.
- Create the left and right nodes
- Call the add elements into child nodes function.
- Call the compute splitting plane position function for the left and right nodes.
- Recursively split the left node.
- Recursively split the right node.

(iii) Compute Splitting Plane Position: This one returns the splitting plane position and normal. The only part that changes between Octree, KD-Tree and BSP-Tree implementations is this part. Octree splits the node on its geometric center and the planes are parallel to the main axis (x, y and z). KD-Tree still uses the axis-aligned planes to split, but the position of the split-planes are determined so that the left and right hand sides of the split plane contain equal number of volume elements. In the case of BSP-Trees additionally the split plane is not necessarily aligned with the main axis.

(iv) Add Elements into Child Nodes: According to the splitting position and the split plane puts each element into left or right node's element list. To achieve this tree-vertex and tree-face intersection functions are implemented.

In addition to these build functions; we also have the following traversal functions.

(i) Partitioned .geo file creation: This function creates the new .geo file to feed

to NETGEN which contains the split planes that are calculated during the mesh building step. This function calls the recursive function that writes the cut planes to the .geo file.

(ii) Write Cut Planes to File: This function recursively traverses the tree and writes each node's cut plane. It uses breadth first search while doing so, so that the cut planes could be written in correct order.

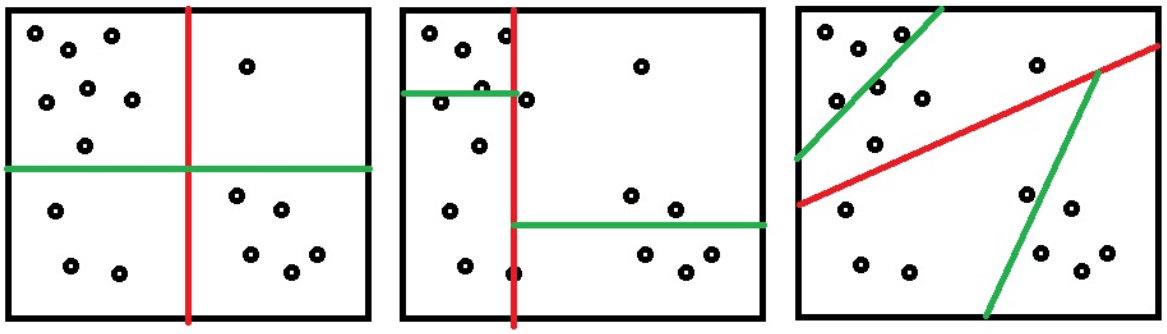

Figure 4.3. Different Tree Methods. a) Octree b) KD-Tree c) BSP-Tree. Circles denote elements, red lines the first level partition planes, green lines the second level partition planes.

4.1.2. Surface Mesh Partitioning

The same tree that we used to create the new geometry is also used to partition the surface mesh. The elements on the nodes of three are at this point not relevant, since they came from the coarse mesh. The NETGEN mesh generation function is called again with the modified .geo file which contains the split planes that were calculated. The newly generated fine surface mesh is used to fill the leaf nodes of the tree. The recursive face-tree intersection methods that were implemented are used here. A good thing with this is that the vertices already have global ID's and for our implementation there is no need for boundary face ID's to be global. The Tree also contains an additional list<int>* for vertices which contain partition node IDs.

4.1.3. Distribution of the Partitioned Surface Mesh

Vertices and faces are sent to each processor according to the partition nodes. The contents of each node will be sent to the related processor. This process can both be through MPI communication of strings or through files. Only leaf (terminal) nodes do any actual file or string write operation. For example a 4 level tree will have 8 leaf nodes each of which will contain one surface partition. Let's denote L as left and R as right nodes. In this case processor 0 will get the surface mesh from LLL node, processor 1 from LLR node, processor 2 from LRL node, etc.

The format of the string or file that is created to be sent by each terminal node to the corresponding processors has the following structure:

First the vertex count is written and then each vertex is written line by line. The vertex ID is followed by the coordinates of the vertex and then size of the list of processors that should have that vertex at the creation of sub-meshes on each processor. At the end the processor list is also filled. The list of processors mostly is one entry long (so numprocs is mostly 1), yet for shared vertices it may be 2, 3 or even more.

```
VertexCount
VertexID1 x1 y1 z1 numprocs proc1-ID proc2-ID ... procN-ID
VertexID2 x2 y2 z2 numprocs proc1-ID proc2-ID ... procN-ID
...
...
VertexIDN xN yN zN numprocs proc1-ID proc2-ID ... procN-ID
```

Figure 4.4. Vertex part of the format of the distributed string.

Then the boundary face information is written to the string or file. This part is straightforward. In these lines faceTriangle denotes the id of the geometric face the triangle is on. The number of processors and the processor list is the same as the vertex information.

TriangleID1 x1 y1 z1 faceTriangle1 numprocs proc1-ID proc2-ID ... procN-ID
TriangleID2 x2 y2 z2 faceTriangle2 numprocs proc1-ID proc2-ID ... procN-ID
...
...
TriangleIDN xN yN zN faceTriangleN numprocs proc1-ID proc2-ID ... procN-ID

Figure 4.5. Boundary face part of the format of the distributed string.

4.1.4. Global ID Assignment

After the volume meshes are generated from the partitioned surface meshes in step 6 of Figure 4.1, the global ID assignment and matching takes place (Figure 4.1 step 7). The global ID's for volume elements are created as follows:

Assume that:

- Proc 0 has created 100 volume elements,
- Proc 1 has created 120 volume elements,
- Proc 2 has created 90 volume elements,
- Proc 3 has created 110 volume elements.

Then ID's of the volume elements will be assigned as:

- Proc 0 will assign ID's between 1-100
- Proc 1 will assign ID's between 101-220
- Proc 2 will assign ID's between 221-310
- Proc 3 will assign ID's between 311-420

There is no need for global ID's for boundary faces.

The global ID's for vertices are created as follows:

Assume that:

- Global max vertex id was 430
- proc 0 has 100
- proc 1 has 120
- proc 2 has 90
- proc 3 has 110 newly created vertices.

Then ID's of the newly created vertices will be:

- proc 0 will assign new global ID's between 431-530
- proc 1 will assign new global ID's between 531-650
- proc 2 will assign new global ID's between 651-740
- proc 3 will assign new global ID's between 741-850

By global ID matching we mean the following. Since Netgen generates a new volume mesh for each closed surface mesh which reside in each processor, it also assigns its own ID's to the elements, faces and vertices. What we do is just create a map<int> which relates the global vertex ID's to the local ID's before the volume mesh generation since the closed surface mesh we pass to the Netgen method has the vertices numbered 1 to N and not the numbered the same as the our current surface mesh. After Netgen creates the volume mesh, we just swap the local ID's of the vertices which are on the surface of the newly created volume mesh according to the map. The local internal vertex ID's are also assigned their global counterparts. And their ID's in the volume element data and face data are also swapped. Shared vertex data does not change, because no new vertices are created on the boundary faces and the shared vertex data is already filled with the global ID's.

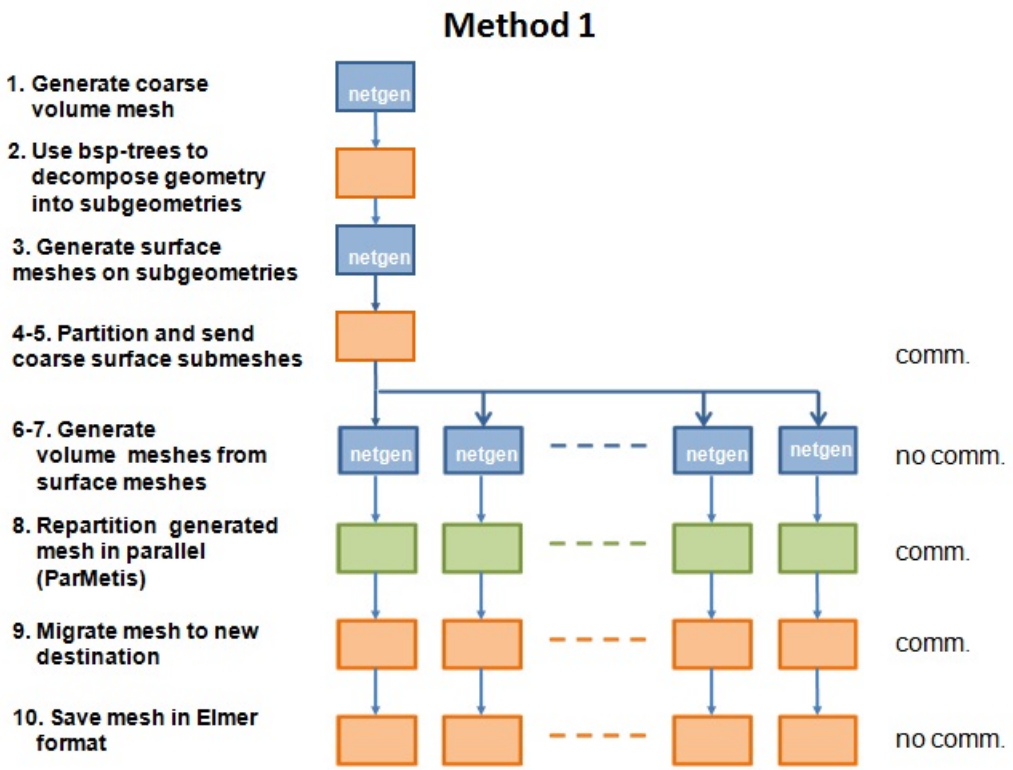

Figure 4.6. Steps of Method 1. Blue colour is netgen, green ParMETIS, orange our implementation.

4.2. Mesh Generation Using Parallel Refinement

Method 2 given in Figure 4.7 employs a parallel refinement method that uses the geometric information from the CAD model. Method 2 is easier to implement than Method 1. Larger meshes can be created without much extra computer power by refinement. Our refinement process uses a simple 8-way split for tetrahedrons and 4-way split for faces. The refinement of surface elements is carried out in parallel without any processor interaction (step 5). After the refinement, global ids need to be generated for newly generated entities in step 6. By traversing volume elements, edges having both of their vertices shared with other processors can be found. Newly generated shared vertices will reside on these edges. There are three types of new vertices: non-shared vertices in the interior and owned shared and non-owned shared vertices on the partition boundaries. Each vertex, even though it may exist in duplicates on multiple processors, is owned only by a single processor. Each processor is responsible for assigning the global IDs to its non-shared and owned vertices. For their owned vertices they also have to report to the other holder processors that they do own the vertex with an assigned global ID. Note that while doing this, one needs to take care of a subtle special case in which a processor may hold two vertices with idv1 and idv2, but does not hold a tetrahedron that uses the edge formed by these vertices. This is illustrated in Figure 4.8 with a 2D example mesh. Here, although the light blue, purple and light brown coloured partitions share the vertices a and b, only the light blue coloured one should create the vertex c. This subtle case does not arise in the data structures used in [11,12] as shown in Figure 3.1b, since edge information is explicitly stored in that work. Finally, we note that unlike Method 1, here the refined mesh is already balanced and hence no repartitioning is needed.

4.2.1. Global ID Generation

Global ID generation actually is straightforward. In our implementation face elements do not need global ID's. Only vertices and volume elements have global ID's. Since the mesh is generated using NETGEN on a single processor sequentially we can use the same ID's that are created for our program. The only additional thing that we

Steps	Method 2
	On processor 0 **do**.
1	Generate a course volume mesh.
2	Partition the mesh sequentially using METIS.
3	Generate global ID's.
4	Distribute the sub-meshes to each processor.
	On all processors **do** repeatedly until the needed mesh size is reached.
5	Perform parallel refinement of the volume mesh.
6	Perform global ID assignment to new entities.
7	Save the distributed mesh.

Figure 4.7. Algorithm for Method 2.

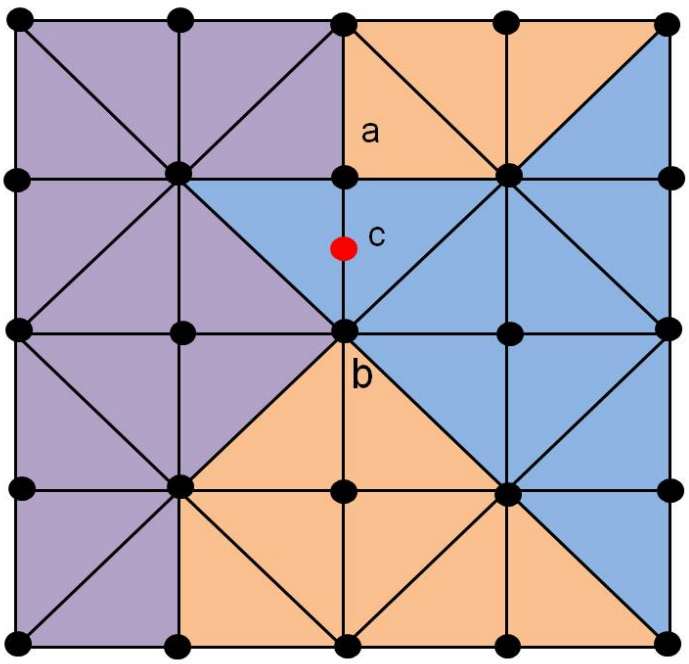

Figure 4.8. A problem that may arise during refinement step. a and b belong to all partitions, but only the blue partition should create vertex c.

do in this step is actually broadcasting the total number of volume elements to each processor. This is used to assign the new ID's for the vertices that are created after the refinement process. So Table 4.1 illustrates this part simply.

Table 4.1. Global ID generation steps.

PROC 0	**PROC 1 to N**
Generate Volume Mesh using NETGEN	—
Broadcast number of volume elements.	Broadcast number of volume elements.

4.2.2. Distribution of Sub-Meshes

After the generated mesh is partitioned on the first processor, we need to send each part to the other processors and while doing so ensure that the shared parts are conforming. What this means in our case is the following:

- The shared vertices should contain the same processor list on all the sharing processors.
- The shared vertices should have the same global ID ('global' already implies that) on all the sharing processors.

Table 4.2 illustrates what happens during the distribution of sub-meshes phase at each processor.

At step 1 the sequential partitioning of the mesh using Metis takes place. A good thing with this method is that the vertices and volume elements already have global ID's since they are created at the same processor. For steps 2 and 3 what needs to be determined is actually only the calculation of shared vertices and the parent ID's for faces. Shared vertices are created by looping through volume elements and using the Metis output to assign each vertex of each volume element to processors. To find the parent volume element ID's for faces we use an STL multimap of (Index3, int). Index3

Table 4.2. Sub-mesh distribution steps.

Steps	**PROC 0**	**PROC 1 to N**
1	Setup and call Metis	
2	Calculate vertex and volume data. Calculate shared vertex data. Find parent ID's for faces.	
3	Fill those to the strings to send to other processors.	Setup string to receive data.
4	Send the number of elements, faces and vertices.	Receive number of elements, faces, vertices
5		Resize receive string.
6	Send the strings that contain the data.	Get the string
7	Create the mesh part using the data in the received string.	Create the mesh using data from the received string.

is just a structure which holds 3 integer values. Then we loop over the face elements (not that this list only contains boundary face elements and not every face inside the mesh) and find their corresponding parent ID using this multimap. After all these are calculated they are packed into strings in a format that will be given in the following part of this chapter. Then these strings are sent to each processor using steps 4, 5 and 6. At the end the received strings are used to create the conforming sub-meshes at each processor (step 7).

Now we are going to describe the format of the string that is created to be sent to other processors during step 6 in Table 4.2. First the vertices are written with the information that they are shared or not. If isShared is equal to 0 this means this vertex is internal, if it is 1 this means this vertex is shared by multiple processors and another line is added for passing this shared processor list information. Numshared field that exists in this new line is the number of processors the vertex is shared among. Figure 4.9 illustrates the format for that.

```
VertexID1 x1 y1 z1 isShared(1)
        Numshared proc1-ID proc2-ID ... procN-ID
VertexID2 x2 y2 z2 isShared(0)
VertexID3 x3 y3 z3 isShared(1)
        Numshared proc1-ID proc2-ID ... procN-ID
VertexID4 x4 y4 z4 isShared(0)
...
...
...
VertexIDN xN yN zN isShared(0)
```

Figure 4.9. Vertex part of the format of the distributed string.

Then the boundary face information is written to the string. This part is straightforward. In Figure 4.10 faceTriangle denotes the id of the geometric face the triangle

is on and ParentID denotes the ID of the parent volume element.

```
TriangleID1 faceTriangle1 parentID1 x1 y1 z1
TriangleID2 faceTriangle2 parentID2 x2 y2 z2
...
...
...
TriangleIDN faceTriangleN parentIDN xN yN zN
```

Figure 4.10. Boundary face part of the format of the distributed string.

As the last thing the volume elements are filled to the string. As also shown in Figure 4.11 volume elements have ID's and the vertex ids that they contain.

```
VolElemID1 x1 y1 z1 w1
VolElemID2 x2 y2 z2 w2
VolElemID3 x3 y3 z3 w3
...
...
...
VolElemID1 xN yN zN wN
```

Figure 4.11. Volume element part of the format of the distributed string.

4.2.3. Parallel Refinement

In this part we will describe how a single refinement step takes place. Actually the whole process takes place in four main parts. First the new vertices will be created at each edge. Then to ensure the conformity between processors the shared vertices that already exist and that are newly created will be communicated among the processors.

The last two steps are pretty straightforward. First the volume elements and then the boundary face elements are refined using the information from the previous steps. In the following part these four parts will be discussed in detail.

4.2.3.1. Creation of New Vertices. For each edge (except boundary edges) we will create a new vertex in the middle of it. So we find all the edges by traversing the volume elements. To make it clear we define the following:

- Shared Edge: If the vertices that make an edge are "both" shared vertices, we will call this a shared edge.
- Boundary Edge: If the vertices that make an edge are on a boundary face, we will call this a boundary edge.

During the creation process the following rules are used.

- Global ID's for the newly created vertices are to be assigned by their "owner" processor.
- Each processor owns all the non-shared (internal) vertices and part of the shared vertices whose owner it is (in very rare cases it may own all or none of the shared vertices).
- The owner of the newly created shared vertex is not random and is determined by the following rule. (This allows every processor to know the new owner of shared vertices without any communication.)
  (i) X and Y are ID's of the two vertices that make the shared edge.
  (ii) Z = (Set of processors who own X) $\cap$ (Set of processors who own Y)
  (iii) Order Z by processor ID.
  (iv) ind = (X+Y) % size(Z)
  (v) Owner is Z[ind]

After these steps the global maximum vertex ID is calculated. Let's call this "globalmax". MPI max function is used on maximum vertex ID's of each processor.

This ID used in the following way. Each processor gives its owned and newly created vertices sequential global ID's starting with (globalmax + 1).

To illustrate this with an example:

- If global max vertex id is 430
  (i) proc 0 has 100
  (ii) proc 1 has 120
  (iii) proc 2 has 90
  (iv) proc 3 has 110 newly created vertices
- Then ID's of the newly created vertices will be
  (i) proc 0 will assign new global ID's between 431-530
  (ii) proc 1 will assign new global ID's between 531-650
  (iii) proc 2 will assign new global ID's between 651-740
  (iv) proc 3 will assign new global ID's between 741-850

4.2.3.2. Communication of Shared Vertex Information to Ensure Conformity. After assigning global ID's to vertices, the owners will need to send this information to the other processors in case they are shared.

- The data to be sent is as follows: id1 and id2 denote the global ID's of the vertices that make the shared edge and glid is the global ID set by the owner processor for the newly created vertex in the "middle" of this shared edge.
  (i) id1 id2 glid
- The processors that will receive this data are determined as follows:
  (i) X and Y are ID's of the two vertices that make the shared edge.
  (ii) Z = (Set of processors who own X) $\cap$ (Set of processors who own Y)
- Each processor sets the global ID of his newly created shared but non-owned vertices according to this data received.

At this step we have two main problems:

(i) There is some excess data that are sent. Owners wrongly assume that every processor who has both the vertices of a shared edge actually has that edge.

(ii) There is some missing data. The assumed owner may not actually own the shared edge. So it did not send anything to the others.

4.2.3.3. Refinement of Volume Elements. This part is straightforward. 8-way tetrahedral refinement is used. Contingent Global ID's are given in a similar fashion that we did with the vertices.

To illustrate this with an example:

- Proc 0 has created 100 volume elements,
- Proc 1 has created 120 volume elements,
- Proc 2 has created 90 volume elements,
- Proc 3 has created 110 volume elements.
- Then ID's of the volume elements will be assigned as:
  (i) Proc 0 will assign ID's between 1-100
  (ii) Proc 1 will assign ID's between 101-220
  (iii) Proc 2 will assign ID's between 221-310
  (iv) Proc 3 will assign ID's between 311-420

Since every volume element that is created is new and no volume element from before the refinement stays, it is okay to assign completely different new global ID's to the newly created volume elements.

4.2.3.4. Refinement of Boundary Elements. This part is also straightforward. 4-way triangular refinement is used. Boundary elements do not have global ID's. They have local ID's. So there is no communication needed between processors to assign ID's to boundary face elements.

At the end of the refinement we update the parent volume element ids of the

boundary elements since they are new.

4.2.3.5. Solution to Problem 1. The processors that wrongly get additional new vertex global ID information, send the following data to the owner processor. In this data id1 and id2 denotes the vertex ID's that make up the shared edge and procID is the processor ID of the processor that got the wrong data.

id1 id2 procID

The owner recreates the shared vertex data using these. (Deletes the processor ID's from the processor list of the shared vertex). Then the owner sends to the remaining processors that information, so that they can update it.

4.2.3.6. Solution to Problem 2. The processors that did not get the global id data from the assumed owner, sends the following data to the assumed owner. In this data id1 and id2 denote the vertex ID's that make up the shared edge and procID is the processor ID of the processor that did not get the global id data.

id1 id2 procID

The assumed owner combines this information from different processors and creates shared vertex information. The global ID's at this step are assigned starting from the last maximum global vertex ID (which we know from the vertex creation process). As the last step, the assumed owner sends both the global ID and the updated shared vertex information to the related processors.

Let's illustrate this solution on the problem that is shown in Figure 4.8. Let's define the following:

- The purple processor is called processor 0.
- The light blue processor is called processor 1.

- The orange processor is called processor 2.

In this case, the vertices a and b are both shared by all three processors. Let's assume that the owner calculation yields the processor 0 as the owner. Since this processor actually does not have the edge (a,b) it will not create c and will not send any information to the other processors. Processor 2 also does not create c, so it is not a problem for him. But processor 1 thinks that processor 0 will create c and assign a global ID to it and send that information to him. Yet he does not get that info. And as our solution it sends "id1 id2 procID" (in this case a,b,1) to the assumed owner which is processor 0. Processor 0 gets a,b,1 from processor 1 and creates a shared vertex information from it. Note that he does not use this data himself, but sends it to all processors that wanted to get the global ID for the newly created vertex on edge (a,b). In this case the newly assigned global ID is sent to only processor 1. On more complex examples the same information could be sent to every processor that creates the new vertex from the edge.

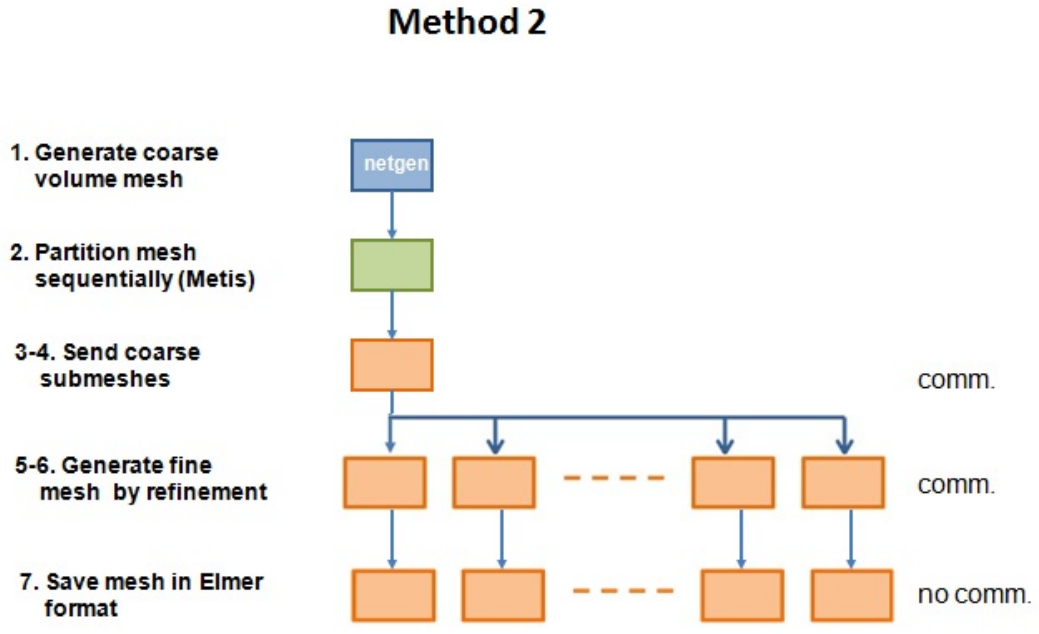

Figure 4.12. Steps of Method 2. Blue colour is Netgen, green METIS, orange our implementation.

4.3. Mesh Generation Using Parallel Surface Refinement

Method 3 given in Figure 4.13 employs a parallel surface refinement method that uses the geometric information from the CAD model. Method 3 is also easier to implement than Method 1. In general the steps are similar to Method 2, yet the refinement takes place on a surface mesh instead a volume mesh. At the end of the refinement a volume mesh generation takes place (step 6). Similarly larger meshes can be created without much extra computer power by refinement. This time the refinement process uses only a 4-way split for faces. The refinement process in step 5 is similar to the Method 2, yet in addition to geometric boundary faces, the partition boundary faces that were calculated during mesh skinning (step 4) are also refined. By traversing geometric and partition boundary elements, edges having both of their vertices shared with other processors can be found. Newly generated shared vertices will reside on these edges. Just like Method 2 there are three types of new vertices: non-shared vertices in the interior and owned shared and non-owned shared vertices on the partition boundaries. Each vertex, even though it may exist in duplicates on multiple processors, is owned only by a single processor. Each processor is responsible for assigning the global IDs to its non-shared and owned vertices. For their owned vertices they also have to report to the other holder processors that they do own the vertex with an assigned global ID. The special case in which a processor may hold two vertices with idv1 and idv2, but does not hold a boundary or partition face that uses the edge formed by these vertices also arises. Just like Method 2, the refined mesh is already somewhat balanced and hence no repartitioning is needed unless in extreme cases.

4.3.1. Skinning the Volume Mesh into a Surface Mesh

The faces that are created in this step are either boundary faces or partition faces. We create maps for each of these called boundfromvol and facefromvol. Boundfromvol will contain the faces from the geometric boundary and facefromvol will contain faces from the partition boundary. The algorithm to skin the partitioned volume meshes is shown in Figure 4.14.

Steps	Method 3
	On processor 0 **do**.
1	Generate a course volume mesh.
2	Partition the mesh sequentially using METIS.
3	Generate and distribute the sub-meshes to each processor.
	On all processors **do**.
4	Skin the partitioned volume meshes in parallel so that they are just surface meshes.
5	Perform parallel refinement of the surface mesh repeatedly until the needed target size is reached .
6	Perform volume mesh generation on the refined surface meshes in parallel.
7	Save the distributed mesh.

Figure 4.13. Algorithm for Method 3.

4.3.2. Volume Mesh Generation on the Refined Surface Meshes

In this part we will create a volume mesh from the closed surface mesh that is defined by the boundary and partition faces. To do this, we will utilize the Ng_Mesh data structure, which is the default Mesh structure that Netgen uses. Since we use a different data structure to represent our meshes, the mesh contents are filled from our mesh object to the Netgen mesh object while holding a map for the element ids, so that the sequential volume meshing can be done by Netgen.

4.3.3. Parallel Refinement of the Surface Mesh

One parallel refinement step is shown in Figure 4.16. Since the newly created vertices will be created in the middle of the edges, we will need to create a temporary edge list that normally is not included in our data structure. Then the shared and non-shared vertices are determined. For the shared vertices the owner is determined

First fill the boundfromvol with faces from the geometric boundary.

For each face at each volume element do the following.

If it exists in boundfromvol.

Do not add the face. (It is boundary face and not partition face).

If it exists in facefromvol.

Then delete that face from there. (This means the face exists on two different volume elements. So it is an inside face and not a partition face).

If it does not exist in facefromvol.

Then add that face there.

In the end of this for loop we will have only faces which are not on the boundary and yet belong only to one volume element of the partition. So this means they are partition faces (they are on the partition boundary).

For each face at each volume element do the following.

If the face is in facefromvol.

then add it to the partition boundary face list.

Now the partition faces together with the geometric boundary faces form a closed surface that can be volume meshed.

Figure 4.14. Algorithm to skin the volume mesh into a surface mesh.

Create an Ng_Mesh element.
Initialize total number of face elements.
Create a set for vertex ID's that are present in this surface mesh and fill it by looping through both the boundary and parition face elements.
Create a temporary vertex ID map.
Fill the Ng_Mesh element with the existing vertices.
Fill the Ng_Mesh element with the face data according to the temporary vertex map.
Generate Volume Mesh.

Figure 4.15. Algorithm for volume mesh generation from refined surface mesh.

with the same formula as in Method 2. The same excess data and absent data problems are also encountered and solved like in Method 2.

The edge list creation process that is mentioned in step 1 of Figure 4.16 works as follows: First we fill the edge list that does not exist in our data structure. The edge list is a map of Index2 to integer. For each edge of each geometric boundary face element, the edge is added to the edge list if it does not already exist. For each edge of each partition boundary face element, the edge is again added to the edge list if it does not already exist. After these steps we have found all the edges. We also have calculated during the loop whether each edge is fully shared or not. Fully shared edge is an edge that both its vertices are shared. Additionally we also have calculated the maximum global vertex ID that exists and how many new vertices each processor will have.

In step 2 of Figure 4.16 for each edge the following is done. If it is non-shared we do nothing. If it is shared, then we compare the shared processor ID's of the two vertices that make up the edge and find the intersection set of those two. From this intersection set the owner is determined just like in Method 2. At this point, four possible outcomes exist. These are:

Steps	Surface Mesh Refinement
1	Create the edge list.
2	Calculate the number of shared vertices for each processor.
3	Assign global ID's to newly created, non-shared vertices.
4	Calculate the size of the data each processor will communicate with each other.
5	Calculate the global ID's for newly created, shared and owned vertices and send this information to non-owner processors.
6	Update newly created, non-owned, shared vertex global ID's.
7	Create the new vertices, geometric boundary and partition boundary elements.

Figure 4.16. Parallel Refinement of the Surface Mesh.

- If the processor owns the new vertex and the intersection set contains more than one processor the data size to send to other processors is incremented.
- Else if the processor owns the new vertex and the intersection set only contains this processor, the vertex is obviously an internal vertex so, we set the new vertex as an internal vertex.
- Else if the processor does not own the new vertex and the intersection set only contains this processor we do nothing. Yet, this should never evaluate to true.
- Else data to receive from the owner is filled.

At this point we already have calculated the number of shared vertices at each processor during the edge loop. We should start assigning the ID's of the new vertices and send the global ID's of the owned vertices to the other processors if necessary. To do that, we again loop the edge list. If it is shared and the processor is not the owner, it does nothing. If it is shared and the processor is the owner, it calculates the intersection processor set again and increments send count to each processor in the set. If it is non-shared a global ID is assigned to it. During this loop we also calculate the size of the data each processor will send and receive from each other processor.

As mentioned in step 5 in Figure 4.16 now we need to assign global ID's to the newly created, shared vertices and send this information from the owner processors to the non-owner processors. At this point we already know which processor will send what, which will get what and how many from whom. We just need to fill the data to be sent. Again we loop through the edge list. At this step the processor only does something if the looped edge is shared and the processor is the owner. What is actually done is that the owner processor assigns a global ID to the shared and newly created vertex. Again the owner calculates the non-owner yet sharing processor set and fills the following data to send:

"vertexID1, vertexID2, globalID" where vertexID1 and vertexID2 are the ID's of the vertices that make the edge, and globalID is the global ID of the vertex that is newly created in the middle of that edge.

After the data are filled, it is sent to the receiving processors. At this point we process the received data (step 6 in Figure 4.16).

For each data received, each processor checks if the information about the edge that is sent exists. If it exists, it updates the global ID of the newly created vertex on that edge with the received data. If it does not exist in its own edge list, then it adds the edge information to an excess list which will be used later. Then for each edge the processor checks whether all the missing global ID's in its own edge list are updated. If they are not updated which means the supposedly owner processor did not send the global ID of the new vertex, then the processor adds that edge information to an absent list which will be used later. First we call the function that solves the absent problem and update the missing global ID data with the information that is returned. How this function works was already described during Method 2. Then we call the function that solves the excess problem. This function is also explained during Method 2. The shared vertex information should be updated with the received data afterwards.

The last step in Figure 4.16 works in the following order. First the vertex list is

resized. For each edge new shared vertex information is created. Also the actual new vertex is created and its global ID is assigned from the edge list and its coordinate data is calculated. Now we update first the absent data related shared processor info errors and then the excess data related shared processor info errors. As the last remaining steps the new geometric boundary and partition boundary elements are created.

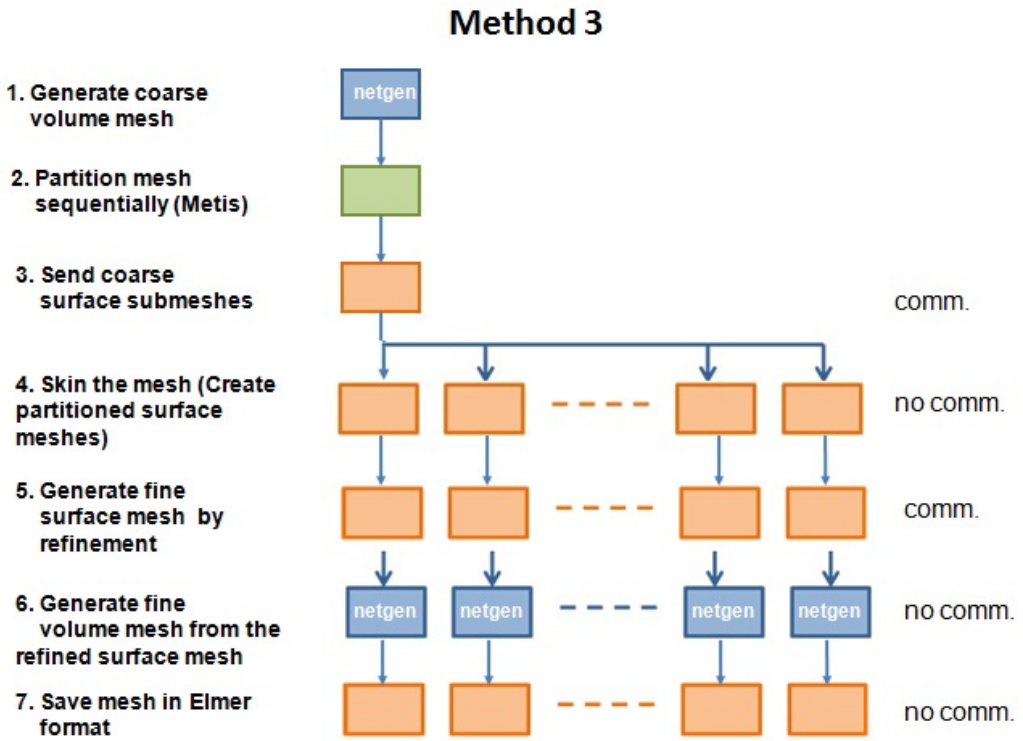

Figure 4.17. Steps of Method 3. Blue colour is netgen, green ParMETIS, orange our implementation.

5. MESH MIGRATION

When the mesh has been generated, especially by using Method 1, the sizes of the sub-meshes on processors may be imbalanced. Therefore, the distributed mesh needs to be repartitioned in parallel. To do this, ParMetis [13] has been used to repartition the mesh in parallel. ParMetis provides routines that allow direct input of the mesh. After completion, it returns the destination ID of the processor on which each element needs to reside at the end. As a result, we are faced with a nontrivial distributed mesh migration and data structure update problem. Even though packing of tetrahedra information (vertices that make up tetrahedra) and sending the packed mesh is straightforward, the entities such as vertices that are shared by elements introduce complications, especially at the partition boundaries.

When the mesh is migrated, it is possible that interior entities become partition boundary entities and vice versa. As a result, when moving, in particular, a partition boundary entity to other processors, holders of this partition boundary entity need to be notified of where the entity migrated. Owner updates paradigm that was used in [12] is used in our mesh generator to implement these distributed notifications. In this paradigm, an owner processor of each entity is designated. This owner is known by all duplicate (shared) entities of that entity. The new destinations can be collected in this owner and the information about the new holders (holding processors) of duplicate entities can be updated.

Figure 5.1 shows a mesh that has been partitioned into four. Even though three-dimensional meshes are generated in our project, a two dimensional example mesh has been used in this figure for simplicity of presentation. Bold dotted vertices are owned by the processor holding them. Non-bold vertices are not owned by the holding processor. Interior entities which exist as a single copy are owned by the processor holding them. For example, owner of vertex c is processor 0. Partition boundary vertices which exist in duplicates have only one owner. For example, the owner of vertex ‘a’ is processor 2 and the owner of vertex ‘b’ is 0.

For partition boundary entities a list of holders is stored on each processor. For example, in Figure 5.1b, the list (0,1,2,3) will be stored on processors 0,1,2 and 3 for vertex a. The list (0,1) will be stored on processors 0 and 1 for vertex b. No list will be stored for c since it is an interior vertex. Note that the lists are stored in a hash table indexed by the global ID of the entity.

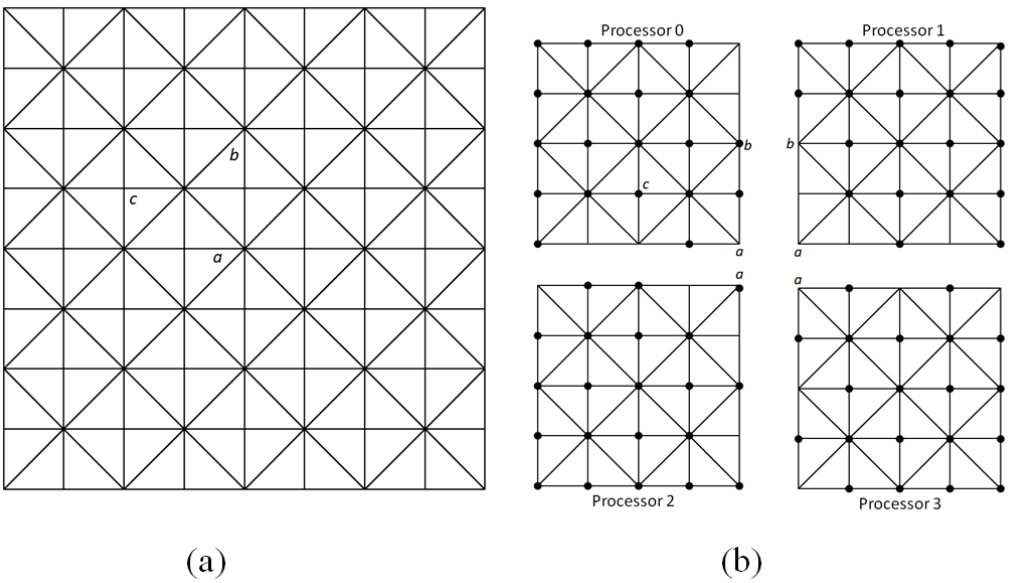

Figure 5.1. Mesh partitioned into 4 parts (a), and with processor ownership of vertices (bold vertices are owned by the processor holding the vertex) (b).

There are different ways in which the ownership of entities can be determined. A simple strategy is to let the holder (processor) with the lowest or largest ID among the holders to be the owner of the entity. Even though this method does not require any communication, it has the disadvantage that imbalanced distribution of owned entities among processors can occur. For example in case of using the lowest ID to determine the owner, the processor with the smallest ID would own all of its shared vertices and the one with the largest ID would own none of its shared vertices. Another approach that is used in [12] is to randomly pick the owner from among the list of holders. This, however, requires communication of the owner to the holders. The method that was implemented in this work is as follows: In method 1, since the surface mesh is determined during the sequential phase, no new vertices are created in the parallel

phase. Therefore, this problem does not arise. In Method 2, when a new vertex is going to be created as a result of refinement of an edge, the global ids of the vertices (which already exist) are used to calculate its owner. Call these IDs idv1 and idv2. The holders list on each processor that holds these vertices are also used and intersected. Denote this intersection list of processor ids in sorted order as (pi1, pi2. . . , pin). Then, each processor computes the owner by using the formula:

$$\textit{Index to owner id in the list} = (\textit{idv1} + \textit{idv2}) \bmod n$$

For example, suppose vertex ids are 12 and 13 and the intersected holder list is (4,5,7), then the owner id index in the list is: (12+13) mod 3 = 1 which gives the second entry in the list which is processor 5. This scheme does not require communication and assigns a somewhat randomized ownership assignment to newly created vertices. With increasing number of processors this approach would yield to better (more equal in number) distribution of shared vertices among processors.

Mesh migration must be implemented in such a way that it will work in general for any pattern of movement. It should have low communication cost so that it can scale to thousands of processors. Figure 5.2 shows the previous example mesh where some elements are to be migrated to some other destination processors. This example illustrates what may happen to interior and partition boundary entities and points out to some cases for the migration algorithm to take care of:

- Vertex a, which is partition boundary entity, is migrated by both processors 0 and 2 to processor 1. Holders that will migrate a (in this case 0 and 2) inform the owner processor about the destination. The new holders list (1,3) is formed by owner processor, and holders (processors 1 and 3) need to be updated about this.
- Vertex b, which is a partition boundary vertex, becomes an interior vertex on processor 0. The owner can update the holders list without the need to notify the other holders.
- Vertex c, which is an interior entity, becomes partition boundary entity on des-

tination processors 1 and 2. The current owner, i.e. processor 0, needs to form the holders list (1,2) and send it to processors 1 and 2.

- Vertex d, which is an interior entity, is still interior on the destination processor, so no update operation is done since no holders list is formed.

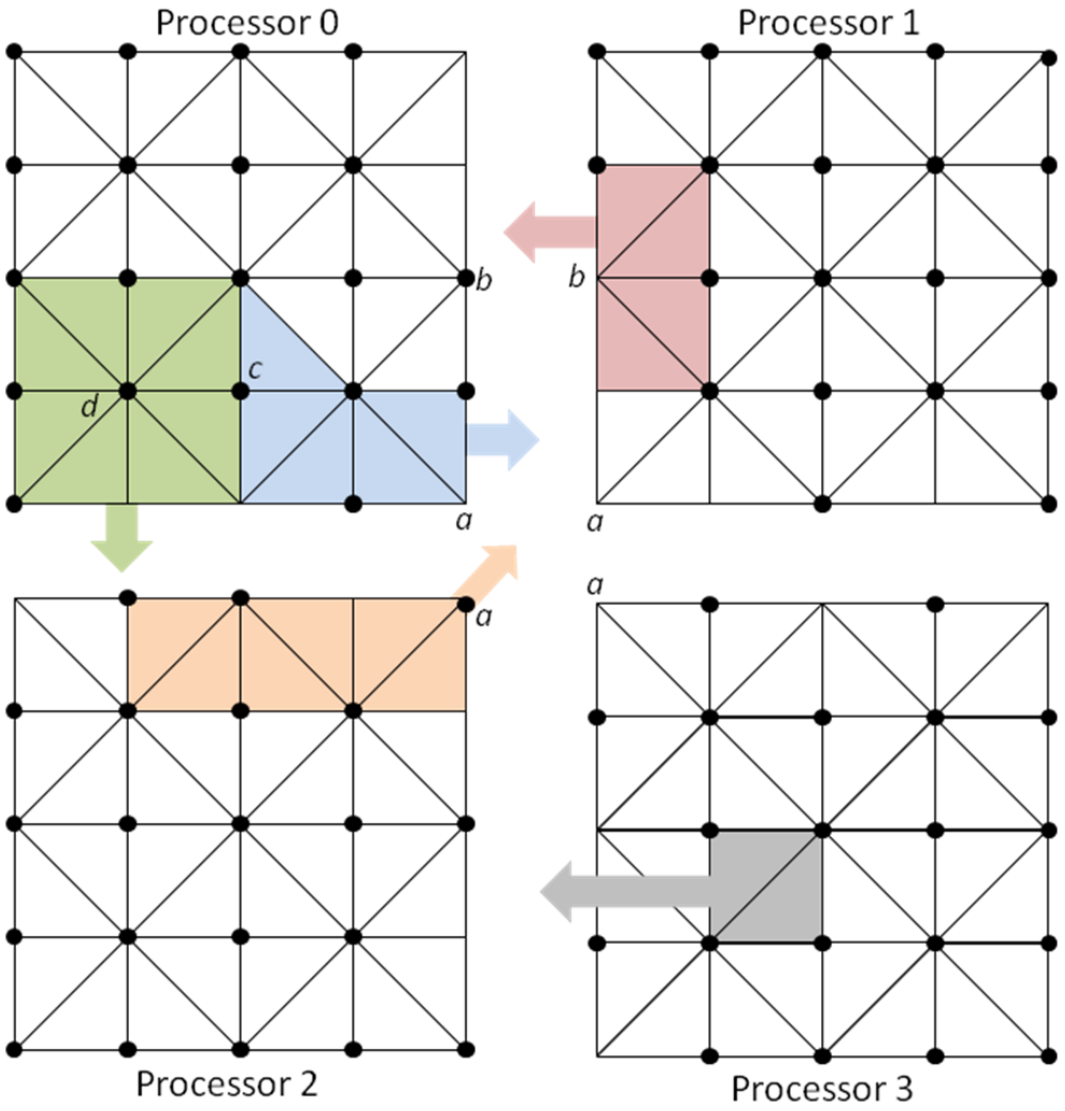

Figure 5.2. Mesh migration example.

The "owner updates" based mesh migration algorithm given in [12] does not use global IDs as entity identification. It uses processor ID and entity pointer tuple and proceeds in three phases: (i) senders send migrated sub-meshes to receivers; (ii) senders and receivers send updates to owners; and (iii) owners send updates to the

affected processors. In our work, since global IDs are used, this algorithm has been further simplified and optimized. Figure 5.4 shows the detailed steps of the mesh migration routine we have implemented.

Steps 1 through 4 are basically about packing and sending of sub-meshes to be migrated to destination processors. Step 5 is concerned about the sending of the shared vertex data and the updating of the holders lists. In particular, step 5.d works as follows: Since vertices are either internal or shared, the shared information data is determined and sent to the owner processors. In steps 5.(e-i), the add/delete instructions are processed by the owner and the new holder lists are sent to the shared vertex holders. The scalability performance of the migration algorithm implemented is reported in the next chapter.

As also previously mentioned, vertices in our implementation are either internal or shared. The "shared information data" that is mentioned in the above paragraph is filled as follows:

For Internal Vertices:

- If the vertex is sent to only one processor do nothing.
- If the vertex is sent to more than one processor (1 means "add", 0 means "delete")
  (i) If the vertex will no longer be here. Send: "vertex id, myprocessorID, (0)" and "vertex id, sentProcessorID, (1)"
  (ii) If the vertex will still be here. Send: "vertex id, myprocessorID, (1)" and "vertex id, sentProcessorID, (1)"

For Shared Vertices:

- Find the current owner processor id, which is the first entry in the processor list of the shared vertex.
- If the vertex is sent to more than one processor (1 means "add", 0 means "delete")
  (i) If the vertex will no longer be here. Send: "vertex id, myprocessorID, (0)"

and "vertex id, sentProcessorID, (1)" foreach processor it is sent.

(ii) If the vertex will still be here. Send only: "vertex id, sentProcessorID, (1)"

In both cases the data are sent to the owner processor. This corresponds to: In case of internal nodes, the data are sent to itself, which is implicit. In case of shared nodes the data are sent to the owner processor. The owner of the vertex before the migration process (not the new owner) will handle this data and make sure it will be conforming after the migration.

The new "holder list" that is created by the previous owners (Step 5.f of Figure 5.4) of the vertices have the form as follows:

"-1 vertexid procid1 procid2 procid3 ... procidN"

This same information is send to each of the processors from procid1 to procidN. Note that N in procidN is not the number of processors used to run the program, but the number of processors that share the vertex in question.

-1 is used to denote the start of new data block, since it is unknown to the processors that will receive the data how long each part is. In this way we also reduce communication cost.

When this data is received by the processors that share the vertex it is used as follows:

- If the vertexid already exists in the shared vertex data.
  (i) If the size of the processor list is greater than one update the shared vertex data. (Just overwrite)
  (ii) If the size of the processor list is 1, this means the shared vertex has become internal. So delete shared vertex data.
- If the vertexid doesn't exist in the shared vertex data.
  (i) If the size of the processor list is greater than one create a new shared vertex

data.

(ii) If the size of the processor list is 1, this means the supposedly shared vertex is actually internal. So do nothing.

5.1. Example

Now we are going to give an example of how the process takes place for the vertices a,b,c,d in Figure 5.2 between the 4 processors. The result is displayed in Figure 5.3. Note that the actual data is much larger, but we are only interested in the four vertices we selected for demonstration purposes.

We first start with how this vertices are represented in the processor list.

- Vertex a's list is (2,0,1,3). 2 is the first element since the owner processor is processor #2.
- Vertex b's list is (0,1)
- For vertex c and d we do not have any lists since they are currently internal.

The next step is the "shared information data" that is created by each processor.

- Processor 0 has all four vertices and will create:
  (i) Vertex a is sent to processor 1, but its data (a,0,0 and a,1,1) is sent to processor 2 since it is the owner. A,0,0 means processor 0 will no longer possess vertex a and a,1,1 means processor 1 will have vertex a after the process.
  (ii) Vertex b is not send anywhere and so no data is created.
  (iii) Vertex c is sent to both processor 1 and 2. Since the owner is processor 0, the data (c,0,0 and c,1,1 and c,2,1) is sent to itself. Again c,0,0 means that vertex c will no longer be here. And c,1,1 and c,2,1 mean that c will be sent to both processor 1 and 2.
  (iv) Vertex d is sent to processor 2, but since the owner is again processor 0, the

data (d,0,0 and d,2,1) will be sent to itself. D,0,0 again stands for the loss of the vertex and d,2,1 means the vertex d is sent to processor 2.

- Processor 1 has the vertices a and b.
  (i) Vertex a is not sent anywhere so no data is created.
  (ii) Vertex b is sent to processor 0. The data (b,1,0 and b,0,1) is also sent to processor 0 since it is the owner of vertex b. Since the vertex will no longer be here, b,1,0 is sent. B,0,1 means processor 0 will get vertex b after the process.
- Processor 2 has the vertex a.
  (i) Vertex a is sent to processor 1, Since the owner is processor 2, it will send the data (a,2,0 and a,1,1) to itself. A,2,0 means processor 2 will no longer have vertex a, and a,1,1 means vertex a is sent to processor 1.
- Processor 3 has the vertex a.
  (i) Vertex a is not sent anywhere so no data is created.

So the following is what each processor got from the others or itself.

- Processor 0
  (i) c,0,0
  (ii) c,1,1
  (iii) c,2,1
  (iv) d,0,0
  (v) d,2,1
  (vi) b,1,0
  (vii) b,0,1
- Processor 1
- Processor 2
  (i) a,0,0
  (ii) a,1,1
  (iii) a,2,0
  (iv) a,1,1

- Processor 3

As expected each processor only got data for vertices it is the owner of. Now each processor will create the new "holder list". At the end of this step the owner processor also sets the new owner of the vertex randomly from the "holder list" to ensure a more or less equal distribution of shared vertices among processors.

- Processor 0
  (i) Vertex b's new holder list was (0,1). B,1,0 will delete the 1 and b,0,1 will add 0 to this list. So the new list will be just (0)
  (ii) Vertex c had no holder list since it was internal. C, 0, 0 will delete 0 from the list, and c1,1 and c,2,1 will add 1 and 2 to the list. The new list will be (1,2)
  (iii) Vertex d also had no holder list. d,0,0 will delete 0 and d,2,1 will add 2 to the new list. So the new list will be just (2)
- Processor 2
  (i) Vertex a's holder list was (0,1,2,3). A,0,0 and a,2,0 will delete 0 and 2 from this list. a,1,1 and a,1,1 will add 1 to the new list. So the new list will be (1,3).

In this process deletions take place before the additions. This ensures that in a case where the sender thinks he will no longer have a vertex and sends delete information to the owner which may get the same vertex as part of another tetrahedron from another processor, the newly created data will not be problematic. Also the code handles duplicate additions in a way that ignores the consequent additions by checking the list formed so far.

Another issue here is that, for example, processor 0 and 2 will not get new holder's list for vertex a. After the whole migration takes place, each processor deletes any shared vertex information for vertices that are no longer present.

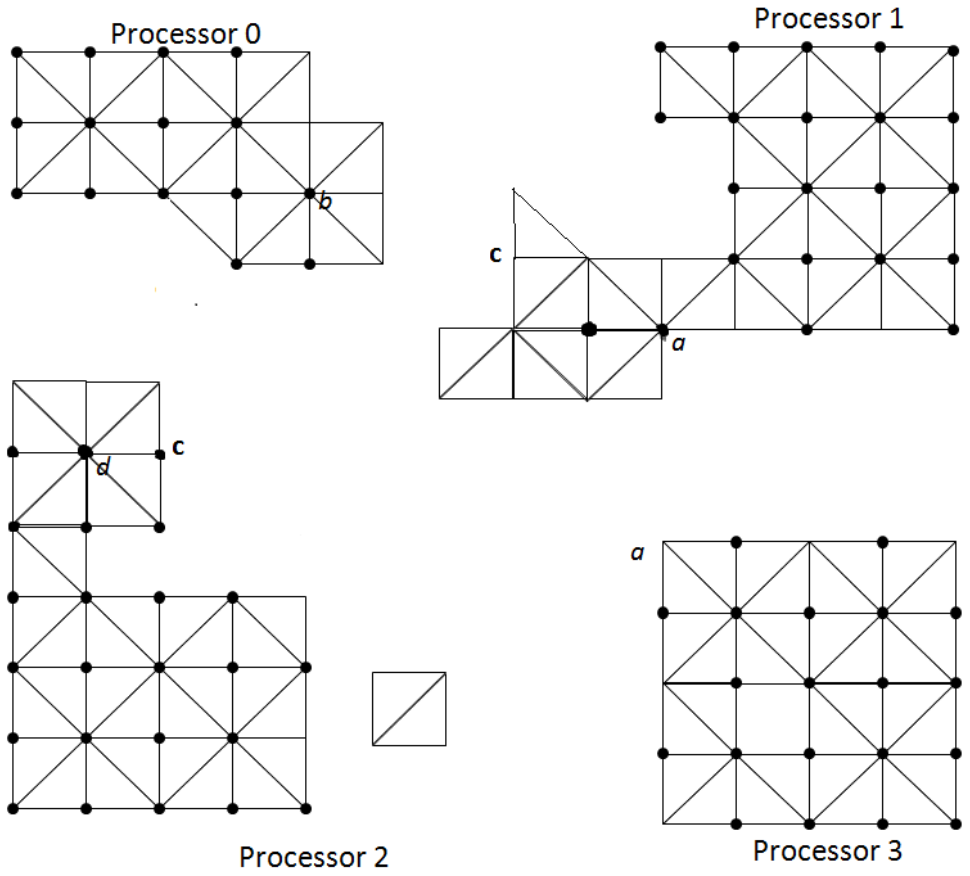

Figure 5.3. Mesh migration example.

- Find parent element (tetrahedra) ids of the boundary faces if not already available.
- Send Boundary Faces:
  - (i) Find total no. of boundary faces to be sent to other processors by checking whether their parent elements are in the list containing the volume elements to be migrated.
  - (ii) Resize migration boundary list accordingly.
  - (iii) Find how many boundary faces are to be sent to each destination processors. Send this information to destination processors.
  - (iv) Now, each processor knows how much to send to and receive from each processor.
  - (v) Send the data of the faces to be migrated.
- Send Volume Elements:
  - (i) Count total number of elements to send and resize migration element list accordingly.
  - (ii) Find how many volume elements are to be sent to each other processor. Notify this information to destination processors. Prepare a list of vertex ids that need to be sent.
  - (iii) Now, each processor knows how much to send to and receive from each processor.
  - (iv) Send the data of volume elements.
- Send Vertices:
  - (i) The vertex id list built in the previous step has all the information about how many vertices each processor sends to the other one. Send this information to other processors.
  - (ii) Each processor knows how much to send to and receive from each processor.
  - (iii) Calculate shared vertex information use counts. Send this information to relevant processors.
  - (iv) Fill in shared vertex information data.
  - (v) Delete vertices with zero use counts.
  - (vi) Resize vertex list according to the deleted vertices and the newly added ones.
  - (vii) Send the vertices data.
  - (viii) Delete duplicate vertices
  - (ix) Update use count of vertices.
- Send Shared Vertex Information (add/delete holder instructions)
  - (i) Send the shared vertex information data already filled in the previous step.
  - (ii) Create a vector array to hold the updated shared vertex data.
  - (iii) Fill vertex array is with already existing shared vertex data (i.e. data from before migration)
  - (iv) Add or delete newly received data according to the shared vertex information data.
  - (v) Delete shared vertices which are no longer present (which are not in the vertex list anymore).
  - (vi) Create new shared vertex information data ("holder list").
  - (vii) Send each processor the size of data they are going to receive.
  - (viii) Send the shared vertex data.
  - (ix) Update the shared vertex data accordingly.

Figure 5.4. Detailed steps of mesh migration.

6. TESTS AND RESULTS OBTAINED

The parallel mesh generator was tested on the Curie supercomputer at CEA in France. Curie is in the Top500 (http://www.top500.org/) supercomputer sites list. Curie system has nodes with 16 cores and 4 GB per core of memory. In Method 1, since the size of the surface mesh on processor 0 dictates the size of the final mesh to be generated in parallel, a fat node configuration (128 GB) is used for processor 0 for the initial sequential phase. For Method 2, since refinement is performed in parallel, we do not have this restriction. However, the bigger the initial mesh, the better the quality of the final mesh will be. Figure 6.1 shows a picture of an example mesh generated on the shaft complex geometry. In the rest of the chapter, we report various performance results.

In Table 6.1, we show the timing results of parallel mesh generation using geometry decomposition (i.e. Method 1). Timings of each component are reported in the table. Method 1 was developed first in our work. It was realized, however, that it had some problems as exemplified in Figure 6.2. When the geometry is decomposed by cutting planes, it is possible to have cases like the one shown in Figure 6.2. Here, an inappropriate cut location will result in thinly sliced regions and cause small elements to be generated where large elements are expected. This problem becomes more probable as the number of sub-domains increases. Because of this difficulty, Method 1 is not appropriate when used on a large number of processors. Therefore, this method was run only with small cases on small number of processors as shown in Table 6.1.

In Figure 6.3, timings obtained from Method 2 are plotted. The lower part of the bar chart corresponds to the sequential initial phase on processor 0, whereas the upper part corresponds to the parallel phase. As expected, the execution time of the sequential part increases with the number of processors, since the overhead associated with the partitioning and distributing of the initial mesh increases. The timings of the parallel component (the upper part) decreases as the number of processors increase, since, for a fixed mesh size, the size of the sub-mesh on each processor decreases.

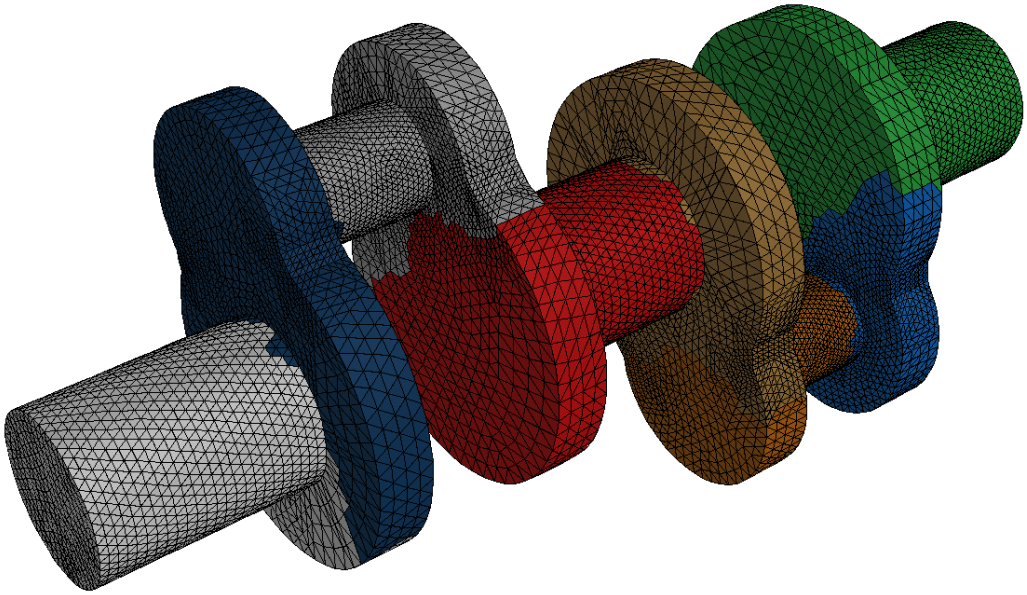

Figure 6.1. Mesh of the complex shaft geometry.

Table 6.1. Mesh generation timings using geometry decomposition (Method 1 steps in Figure 4.1).

		Mesh size in millions		Time in seconds taken by each step				
No. of Cores	**Geometry**	**Surface mesh**	**Volume mesh**	**1-5**	**6-7**	**8**	**9**	**Total**
16	Cube	1.3	14	13	401	19	6	439
16	Torus	1	7.2	12	224	8	3	247
16	Sphere	1	11	11	387	17	5	420
32	Cube	0.425	5	5	107	2	0	114
32	Torus	1.08	8	14	146	6	2	168
32	Sphere	1.15	11	15	217	8	3	243
64	Cube	2	37	21	224	10	5	260
64	Torus	1.5	12.5	20	162	7	3	192
64	Sphere	0.55	4.4	9	47	1	1	58
64	Cube	8	330	84	2328	97	58	2567

At 1024 processors, the parallel part is about 10 seconds. Increasing the number of processors beyond this point is currently not beneficial because even if the parallel 10 seconds is decreased further, the sequential component will increase. We are better off if we use a smaller number of processors wherever possible, since usage of smaller number of processors means less energy consumption by our mesh generator, which is a major issue facing the current supercomputers.

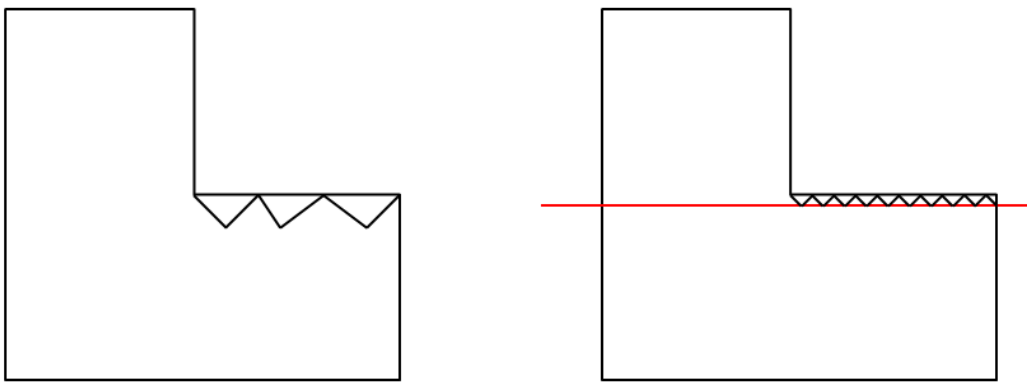

Figure 6.2. Difficulty of making the right cut in Method 1.

The scalability problem we see here at processor size around 1024 for a mesh size of 1.4 billion volume elements is also due to the size of the mesh. As can be seen from the Figure 6.3, the problem arises even earlier for smaller size meshes. For the shaft and cube geometries (having 700M and 800M elements respectively) we see that the scalability problem catches between 256 and 512 processors. This means that if we would further increase the mesh size, we could also increase the processor number without losing much efficiency. Unfortunately we did not foresee this problem and did implement our software to work on 32 bit integers. This would mean that many things like global ID's for volume elements are limited up to around 4 billion. This could be remedied in a future work.

Further performance tests of the migration algorithm were also carried out. Table 6.2 shows timings obtained when 10% of the elements at each processor are migrated to 10 different random destination processors. The last entry for the case of 512 processors

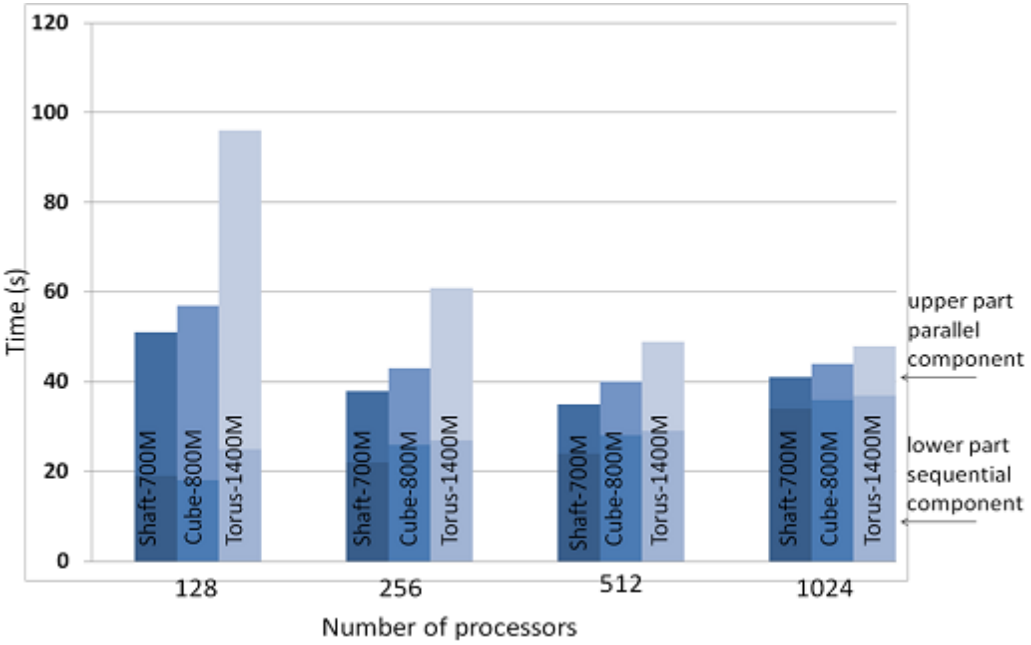

Figure 6.3. Mesh generation timings using refinement based Method 2.

shows that migration of about 80 million elements took under a minute. Considering that migration is to be performed only once after repartitioning, these results are acceptable.

Table 6.2. Mesh migration performance.

No. of Cores	**Time in seconds taken for migration**	**Total no. of tetrahedra (in millions)**
16	13	100
32	9	100
64	8	100
128	8	100
512	14	100
128	50	800
512	57	800

Table 6.3 shows the results for the same setup, but this time migration takes place to 20 random processors instead of 10.

Table 6.3. Mesh migration performance.

No. of Cores	**Time in seconds taken for migration**	**Total no. of tetrahedra (in millions)**
512	43	800
128	44	800
512	16	100
128	9	100

7. DISCUSSIONS, CONCLUSIONS AND FUTURE WORK

We have developed mesh generation and migration routines based on the open source and freely available NETGEN code. Our developed software is also distributed freely as open source code at the address:

http://code.google.com/p/parallel-netgen/

It has been noted by [17] that FMDB consumes too much memory. The data structures we use are very compact – in fact, the same as the data structures used by the open source Elmer solver. Our mesh migration routines operate on the compact data structures and are optimized for shared vertex updates. We also note that even though FMDB source code is available as open source, MeshSim parallel mesh generator [6] that is closely associated with it is commercial. The approach for parallel mesh generation as reported in [6], is to generate the mesh on 64 processors and then load balance the mesh to thousands of processors.

Our main goal in our work was to generate large unstructured meshes with sizes in the hundreds of millions range on complex geometry. As our test results showed, a mesh with 1.4 billion elements could be generated in under a minute using Method 2. These capabilities were not available to the intended Elmer users who usually use sequential mesh generators to generate meshes. For the mesh generation algorithms developed in this project the following conclusions can be drawn:

- Geometry decomposition methods that use decoupled sequential mesh generators may not be suitable when run on large numbers of processors. This is mainly due to the difficulty of automating geometry decomposition. On the other hand, if a proper geometric decomposition is made, then a high quality mesh generation can be done. Therefore, Method 1 can be used on a small number of processors, perhaps with user assisted geometry decomposition in a semi-automatic manner.
- For refinement based methods, mesh generation of billion(s) of elements are pos-

sible on distributed machines.

- Since the size of the mesh that can be initially generated sequentially dictates the size and the quality of the final fine mesh, a large memory fat node is required for the initial sequential coarse mesh generation component.
- A malleable job model can be appropriate for applications in which mesh generation is followed by a solver phase. A smaller number (a thousand or so) of processors can be used to generate the mesh. Then the generated mesh can be migrated to a large number of processors (tens of thousands) and the solver can be invoked using the large number of processors.

We have also implemented a third method, which is a combination of methods 1 and 2. This method combines the advantages of each method by generating surface meshes on sub-geometries and refining them before volume mesh generation. Unfortunately due to time constraints we were unable to debug this method and could not get test results for it on a supercomputer.

Finally we summarize the methods that are used in this project by comparing them on Table 7.1.

In this next part we will discuss some of the possible future works. We did achieve the goal we set ourselves, mainly generating a mesh containing billion elements in a reasonable amount of time. We can improve our work in the following ways:

(i) Making the Third Method More Robust: The third method currently is not running robustly. Due to time constraints, it was not thoroughly tested. This is probably the first thing that should be done.

(ii) Support for More Input Types: Currently the software supports only a few types of geometry files. In the first method this number is only one. Since the main goal of the project was to demonstrate the billion sized mesh generation capability of the software, we did not focus on accepting different types of geometry files as input. The geo file format which is one of the simplest geometry formats to parse and edit is used for this purpose. Yet if we want to use the software in

Table 7.1. Comparison of Methods.

	Method 1	**Method 2**	**Method 3**
Memory Requirements	Processor 0 needs a fat node	Evenly distributed by processors	Evenly distributed by processors
Speed	Slow	Very Fast	Fast
Scalability	Up to 16 processors	Up to 1024 processors	Up to 1024 processors
Geometry Format Support	Only .geo	Several including .stl	Several including .stl
Load Balance		Evenly Balanced	Might need repartitioning with increasing number of refinement steps.
Mesh Size Limit	Up to around 4 billion but limited by the main node memory size and processing power	Up to around 4 billion	Up to around 4 billion
MPI communication	Low	High	Medium to High
Mesh Degradation Problems	Increasing with processor size	None	None

actual projects it is probably a good idea to support at least the standard .obj and .stl formats that many model design software products are able to import and export. In case the user of our program has access to such software he/she could easily convert any geometry format to another after we implement .obj or .stl support. The second and third methods do support those file formats, since they are available to use by NETGEN. Another thing that can be implemented is support for partitioned geometry types. For various reasons the geometry may be already in separate files. The user could have modelled different parts of an object separately, yet may want to use them together in a simulation.

(iii) Support for More Output Types: Currently the software has two different output formats. One is the Elmer partitioned mesh format, and the other is geomview partitioned mesh format. Geomview format is selected for ease of implementation and Elmer was our target solver product. Yet both of these formats are very simple mesh formats and don't generally have two-way relations between elements. For example face elements have information on which vertices it has, but vertices don't have information of which face elements they are on. Edges don't exist at all. Many mesh formats that are currently used in the industry yet require that information. So it would be a good idea to implement a partitioned mesh format with full information.

(iv) Extending to 64-bit Architecture: Another important future work is rewriting the program to support 64-bit data types. Because of this limitation currently we are only able to generate meshes of at most 4 billion volume elements. This also affects the scalability of our program. Solving this problem would enable our program to harness the power of next generation supercomputers which will be beyond petascale. Much larger meshes could be created and bigger number of processors could be used this way.

(v) Using Mesh Generators other than NETGEN: Currently we did use NETGEN as the sole sequential mesh generator. It is possible to use other mesh generators and compare their efficiency. Using other mesh generators would also probably expand the input geometry formats.

(vi) Using Partitioners other than ParMETIS: We did use ParMETIS as the only

mesh partitioner. There are other Open Source mesh generators. PT-Scotch is supposedly a faster mesh partitioner and it would be a good idea to compare their performances, and if possible also slightly speed-up our mesh generator along the way.

(vii) Integration of the software with a FEM Solver: Parallel file read operations do take a lot of time. The output of our program is mainly targeted to be used by FEM Solver's. Currently our program writes the partitioned mesh on to the file system and the solver should read it again from the file system. As can be seen there is a need for a large disc write and disc read operation. Considering the meshes that contain billions of elements have large file sizes, this operation would take a considerable time. There might be two available options to integrate the mesh generator and the solvers. One is directly implementing the mesh generator inside the solver, and the other one is deciding on a protocol and implementing interface functions. Since through these options the read write operations will take place on the memory instead of the file system, we could expect a considerable speed-up in the process. It would also automate some of the manual work that would be done by the scientist.

APPENDIX A: ELMERVIEWER PROGRAM

We also have implemented a program to visualize partitioned meshes at different processors. This program basically uses the partitioned Elmer output files as input and visualizes one sub-mesh that resides at a single processor while enabling functionality to highlight different parts and/or elements of a single sub-mesh. The user of the program has the option to display/hide the following:

- Volume Elements
- Geometric Boundary Face Elements
- Partition Boundary Face Elements
- Vertices
- Shared Vertices

In addition to that it also has the option to show the mesh in grid mode, where only the lines that make up the volume elements are shown.

This program uses the OpenGL library and its implementation details are as follows:

A.1. Data Structures that are used by ElmerViewer Class

- Vertex3D - contains x,y,z (double) coordinates of a vertex.
- Triangle - contains the ids(int) of the four vertices that make a tetrahedron.
- Boundary - contains the ids(int) of the three vertices that make a boundary face.

A.2. ElmerViewer Class Private Members

- (int) numVertex - number of Vertex3D elements.
- (int) numTriangle - number of Tetrahedron elements.
- (int) numBoundary - number of Boundary Face Elements.

- (int) numShared - number of Shared Vertices.
- (int) numParts - number of Partition Boundary Face Elements.

- map<int,int> vertexMap - map between ids of Vertex3D elements and their position in the Vertex3D array.
- Vertex3D* vertexList - Vertex3D array

- map<int,int> triangleMap - map between ids of Tetrahedron elements and their position in the Tetrahedron array.
- Triangle* triangleList - Tetrahedron array

- map<int,int> boundaryMap - map between ids of Boundary Face elements and their position in the Boundary Face array.
- Boundary* boundaryList - Boundary Face array

- map<int,int> sharedMap - map between ids of shared vertices and their position in the shared vertex array.
- int* sharedList - Shared Vertex array

- map <int,int> partMap - map between ids of Partition Boundary Face elements and their position in the Partition Boundary Face array.
- Boundary* partList - Partition Boundary Face array

A.3. ElmerViewer Class Public Members

- ElmerViewer() - Default Constructor
- ˜ElmerViewer() - Default Destructor

- void readMesh(char* filename) - calls the following six read file functions in order to store the mesh from file to memory.

- void readHeaderFile(char* filename) - The format is as follows:
  (i) numVert numelem numbound
  (ii) numofLines
  (iii) typeofelems numofelems
  (iv) boundarytype1 numbound1
  (v) boundarytype2 numbound2
  (vi) ....
  (vii) sharednum.

- void readVertexFile(char* filename) - reads the vertex information from file to memory. The format is as follows:
  (i) ID '-1' xval yval zval

- void readTriangleFile(char* filename) - reads the tetrahedron information from file to memory. The format is as follows:
  (i) ID '1' Type Val1 Val2 Val3 Val4 .....

- void readBoundaryFile(char* filename) - reads the boundary face information from file to memory. The format is as follows:
  (i) LineNum '?' '?' '?' Type Val1 Val2 .....

- void readPartFile(char* filename) - reads the partition boundary face information from file to memory. The format is as follows:
  (i) a1 a2 a3

- void readSharedFile(char* filename) - reads the shared vertex information from file to memory. The format is as follows:
  (i) VertexNo TotalDistributedProcessors Proc1No Proc2No .....

- void drawVertices() - uses the vertexList to draw OpenGL points.

- void drawTriangles(bool isfull) - uses triangleList, vertexMap, vertexList to draw a tetrahedron. If isfull is true the tetrahedron is drawn as four filled OpenGL triangles. Else it is drawn as 6 OpenGL lines. This means isfull is used to show it either as a filled figure or as a grid-like figure.

- void drawParts() - uses partList, vertexMap, vertexList to draw partition boundary faces as 3 OpenGL lines.

- void drawBoundary() - uses boundaryList, vertexMap, vertexList to draw boundary faces as 3 OpenGL lines.

- void drawShared() - uses sharedList, vertexMap, vertexList to draw shared vertices as OpenGL points but this time 5 times larger than normal vertices and with a different colour to make them stand out.

A.4. Main Function that uses ElmerViewer Class

This contains a simple glut window that initializes an ElmerViewer instance with the passed argument as filename. The following variables can be manipulated to see different element types of the mesh:

- bool drawElements - whether to draw the tetrahedron or not.
- bool drawBoundary - whether to draw the boundary face elements or not.
- bool drawPoints - whether to draw each vertex or not.
- bool drawShared - whether to draw shared vertices or not.
- bool isfull - whether to draw the tetrahedron grid-like or filled.
- bool drawpart - whether to draw partition boundary faces or not.

APPENDIX B: INSTRUCTIONS TO RUN THE SOFTWARE

B.1. Running the Parallel Mesh Generator

- Download ng.zip and gl.zip by browsing from source page. `http://code.google.com/p/parallel-netgen/source/browse/`
- First install the following preliminaries.
  (i) sudo apt-get install build-essential tcl8.5 tcl8.5-dev tk8.5 tk8.5-dev tix tix-dev libtogl1 libtogl-dev glutg3 glutg3-dev libxmu-dev liblapack-dev
- Download the source files for NETGEN from `http://sourceforge.net/projects/netgen-mesher/files/` and extract it as follows
  (i) tar xfs netgen-4.9.11.tar.gz
  (ii) cd netgen-4.9.11
- Replace nglib.cpp and nglib.h files with the ones contained in ng.zip
- Install it as follows.
  (i) ./configure –with-tcl=/usr/lib/tcl8.5/ –with-tk=/usr/lib/tk8.5/ –with-togl=/usr/lib/Togl1.7/
  (ii) Make
  (iii) Sudo make install
- export LD_LIBRARY_PATH=/usr/lib/Togl1.7:/opt/netgen/lib
- download Parmetis version 3.2 from `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/download` and install it using make and make install
- unzip gl.zip and locate the TreeBuilder.h main.cpp and MeshMig.h. Change the netgen path's inside these files.
- Open parmetisbin.h and change the parmetis paths according to your installation.
- compile the program using g++.
- call the program as follows: "mpirun -n 16 a.exe cube.geo 3 100000 1 1 1" where the parameters are
  (i) a.exe – the compiled program

(ii) filename.geo – Name of the input geo file.

(iii) 3 – Method name 1,2 or 3

(iv) The following parameters are for method 2 and 3

(v) 100000 – Min size of the starting coarse mesh.

(vi) 3 – Number of refinement steps. Increase to get higher # of elements.

(vii) 1 – Get geomview output. 0 for no. 1 for yes.

(viii) 1 – Get Elmer output. 0 for no. 1 for yes.

B.2. Running ElmerViewer

Just compile with g++ and run like this: "./ElmerViewer filename"

'Filename' should be the name of the Elmer output file that is created by our program. Note that if the created files are 'shaft.header', 'shaft.nodes', 'shaft.elements', 'shaft.boundary', 'shaft.shared', 'shaft.part' then the filename to be entered is only 'shaft'. The program takes care of the file extensions.

APPENDIX C: NETGEN INTERFACE MODIFICATIONS

C.1. Topology Functions Added

Ng_SetPointIndex, Ng_SetSurfaceElementIndex, Ng_SetVolumeElementIndex - These functions set the index of vertices, surface and volume elements.

GetMyPoint - Returns the coordinates of a vertex at a specified index.

GetMyVolumeElement - Returns the index of vertices of a volume element at a specified index.

GetMySurfaceElement - Returns the index of vertices of a surface element at a specified index.

GetBoundaryID - Returns the id of the geometric face the specified surface element resides on.

SetBoundaryID - Sets the geometric face id of the specified surface element.

GetDomIn, GetDomOut - Gets the inside and outside geometric domain ids of a surface element.

Ng_GetParentElement2 - Gets the parent element id of the specified surface element. (In this case the id of a volume element)

C.2. Meshing Functions Added

- Ng_CSG_LoadGeometry - Loads a geo file into a mesh geometry structure.
- Ng_CSG_FindPoints - Interface function.
- Ng_CSG_GenerateMesh - Function to generate a coarse volume mesh.

- Ng_CSG_GenerateSurfaceMesh - Function to generate a surface mesh.

Ng_CSG_GenerateVolumeMesh -This function is overloaded 3 times and the following are achieved. The first function generates a volume mesh from the .geo files and returns both the number of elements created and the elements themselves. The second function is used to just get information about the number of vertices, surface elements and volume elements to be created during the meshing process. The third function is used to generate a volume mesh but this time only returns the elements themselves.

Ng_CSG_ProjectMesh - This function takes the coordinates of a list of vertices and which geometric face they are to be projected. It is used to make sure that at each refinement step the newly created vertices at boundaries snap to the actual geometry.

APPENDIX D: TEST SCREENS

In this part, we can see some screenshots taken from the outputs of our program. Due to visualization limits, the meshes shown here contain at most 20 million elements. Since it is not the scope of this project, parallel visualization software is not implemented. There are in total 3 different objects shown here. First the whole object with coloured partitions is shown. Then the surface mesh that resides just on one partition is shown.

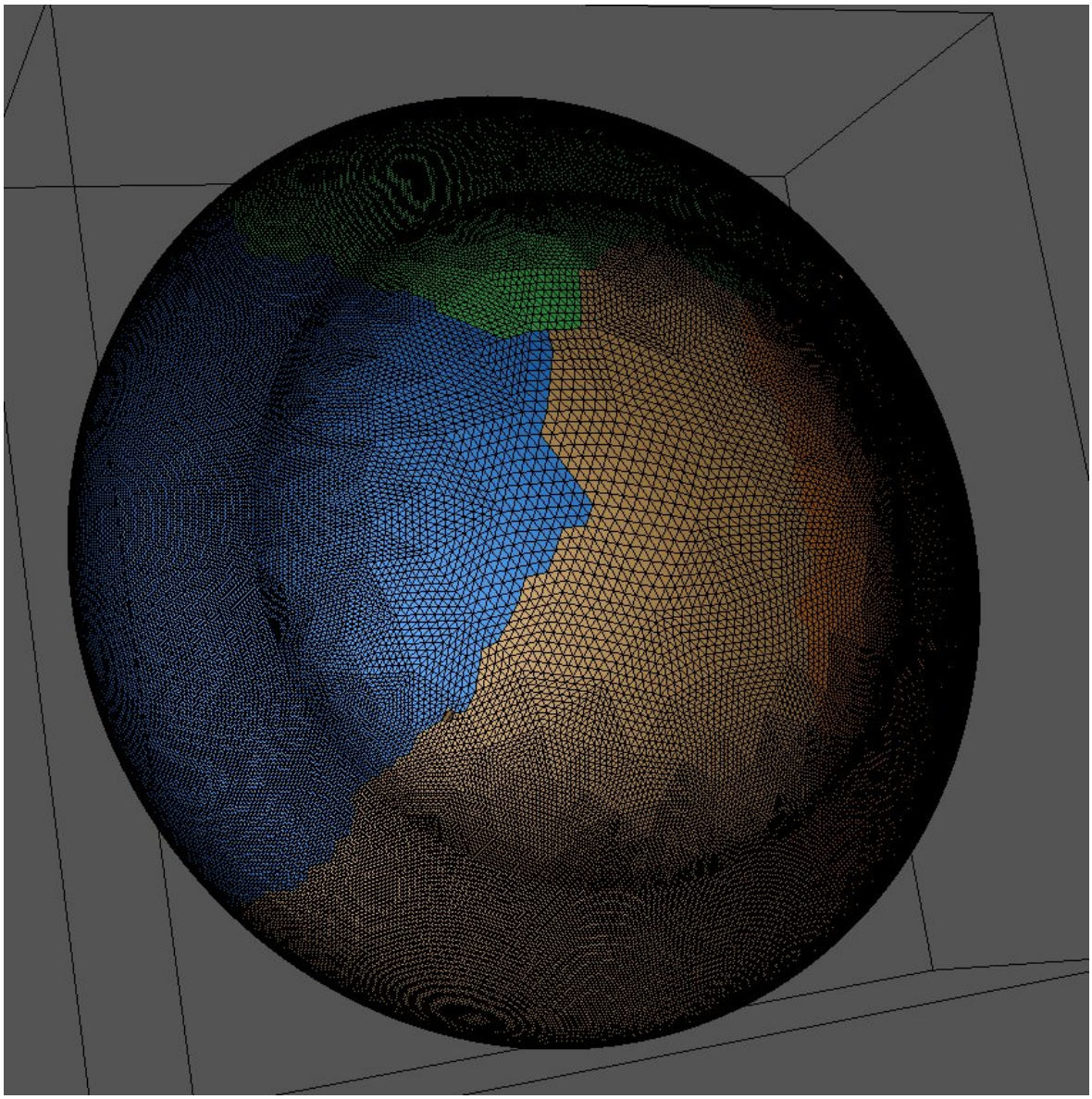

Figure D.1. Sphere with holes (4 parts).

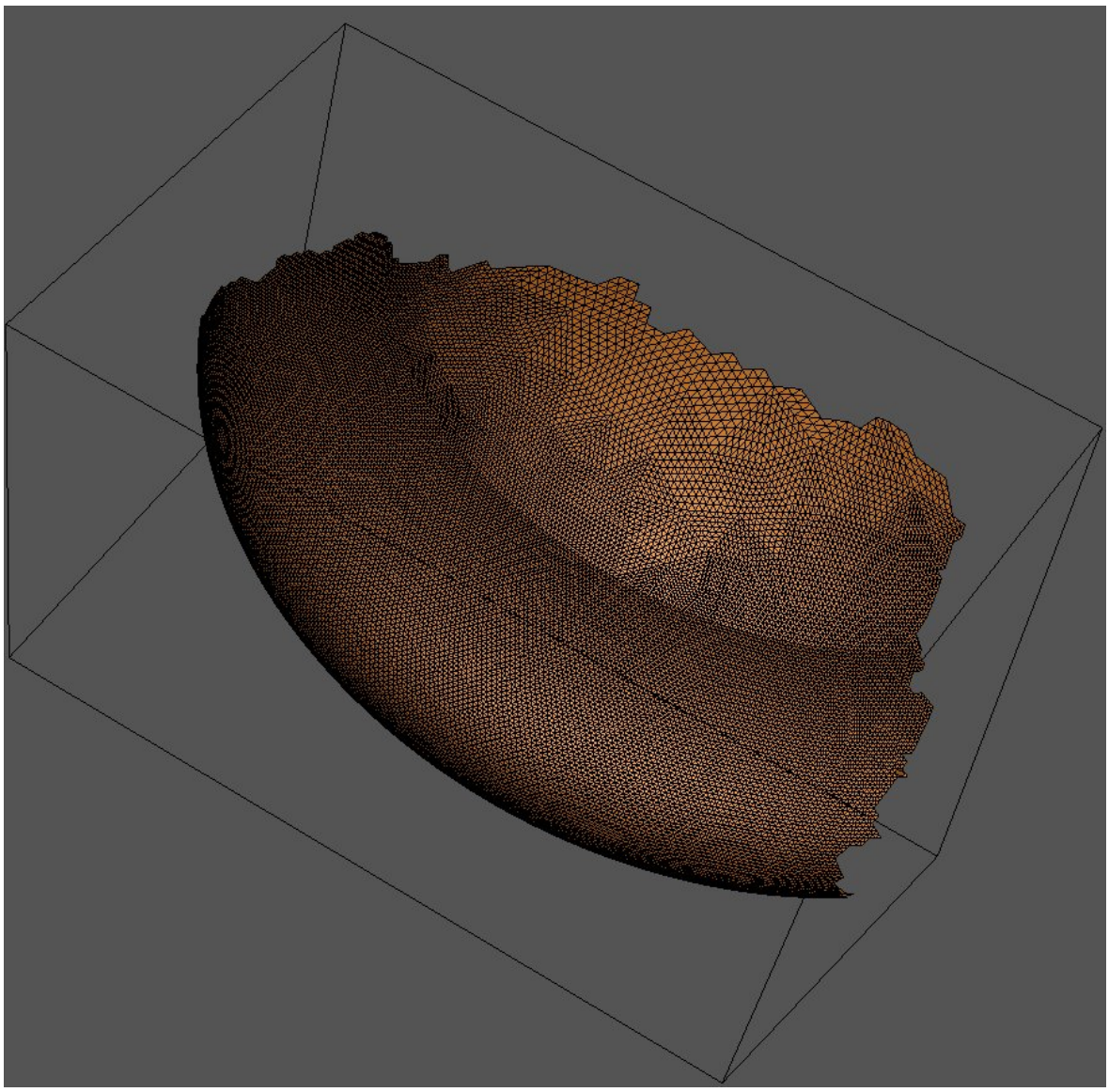

Figure D.2. Part of Sphere with holes.

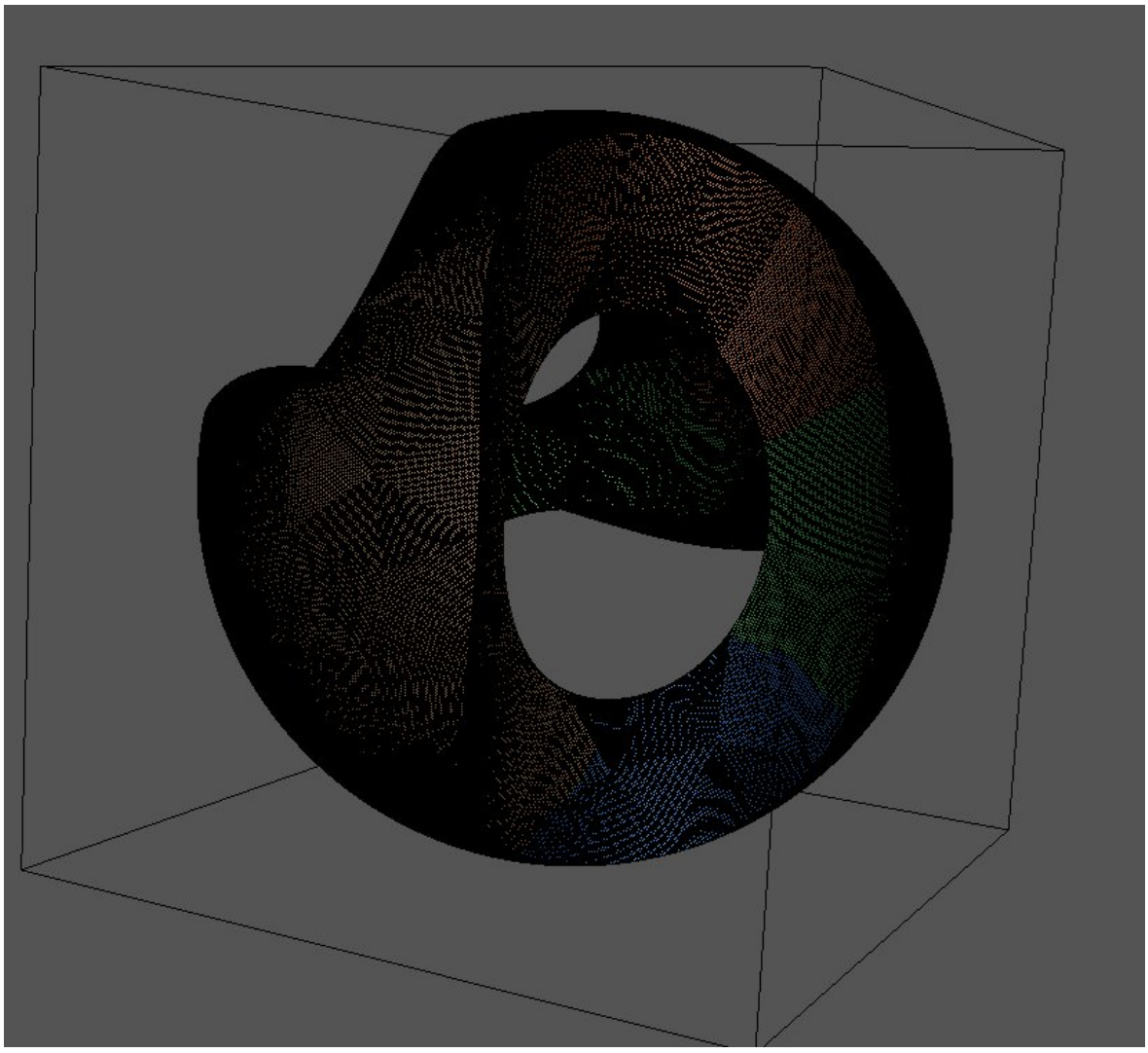

Figure D.3. A different shape (16 parts).

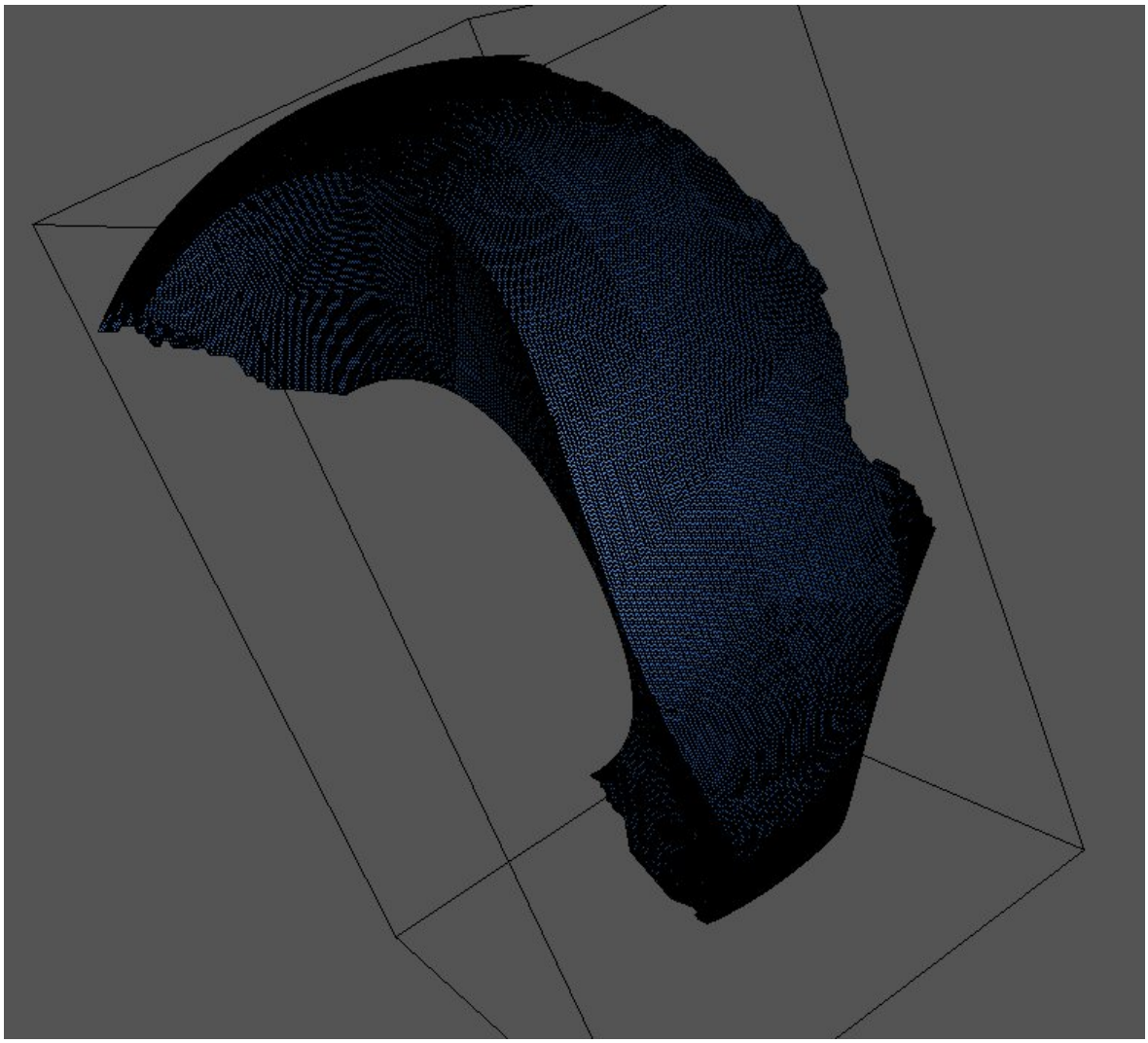

Figure D.4. Part of a different shape.

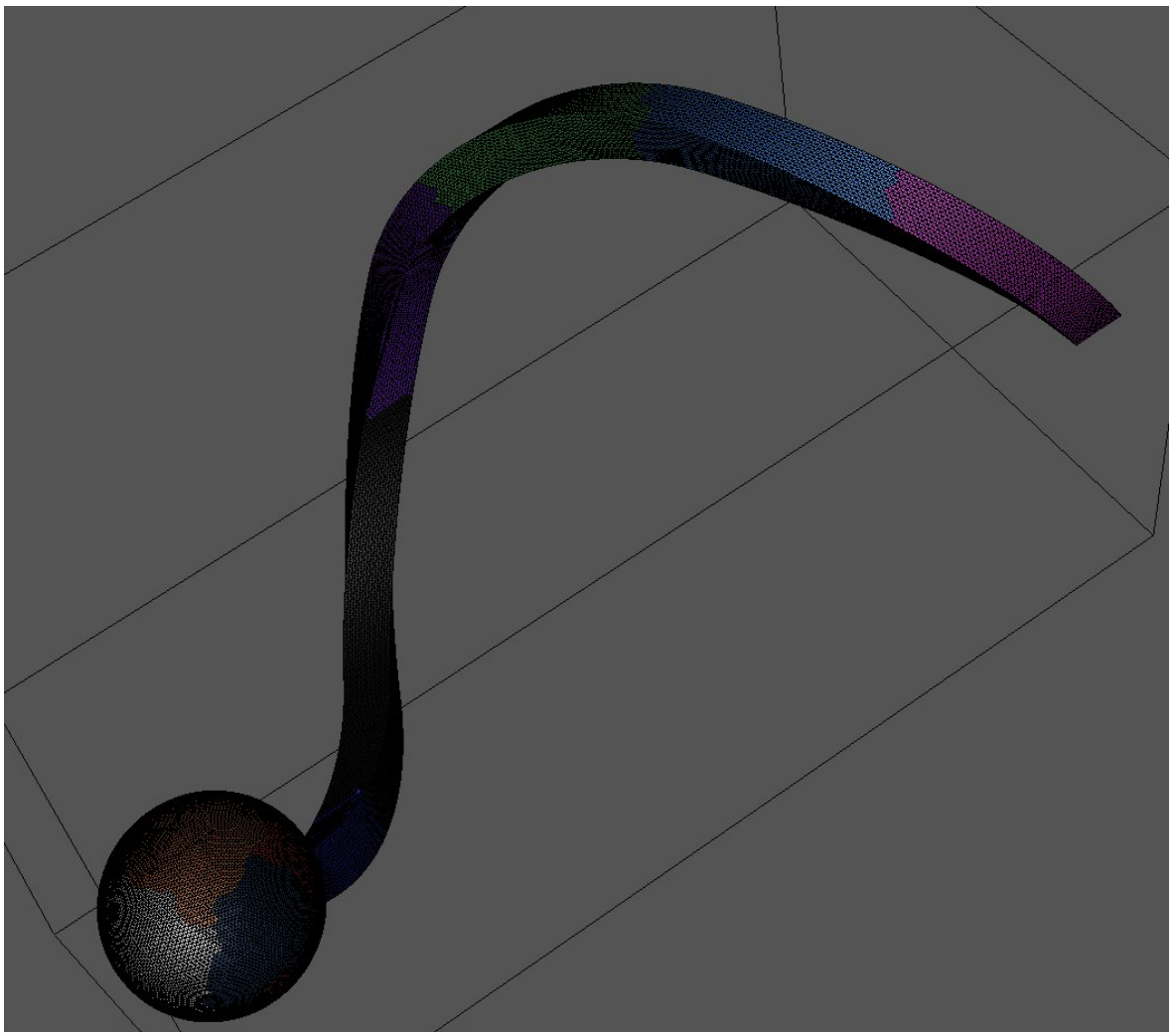

Figure D.5. UFO like shape (4 parts).

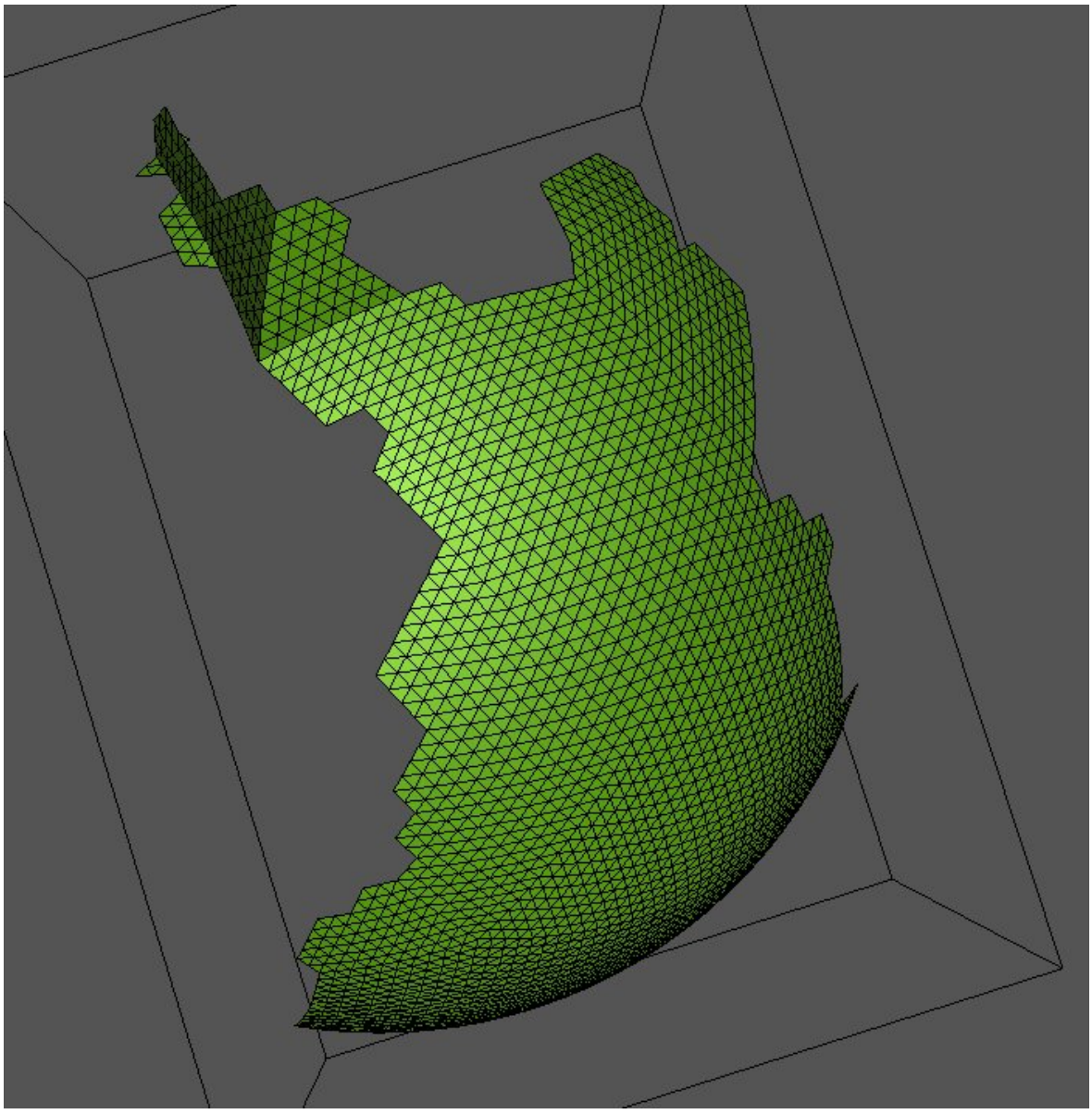

Figure D.6. Part of UFO like shape.

REFERENCES

1. Chrisochoides, N., "A Survey of Parallel Mesh Generation Methods", *Scientific Computing Group Reports*, 2005.

2. NETGEN Automatic Mesh Generator, 2003, `http://www.hpfem.jku.at/netgen/`, accessed at February 2013.

3. Schöberl, J., "An Advancing Front 2D/3D-Mesh Generator Based on Abstract Rules", *Computing and Visualization in Science*, Vol.1, No. 1, pp. 41–52, 1997.

4. Blagojevic, F., A. Chernikov, D. Nikolopoulos, "A Multigrain Delaunay Mesh Generation Method for Multicore SMT-based Architectures", *Journal of Parallel and Distributed Systems*, Vol. 69, No. 7, pp. 589–600, 2009.

5. D3D Mesh Generator, 2001, `http://mech.fsv.cvut.cz/~dr/d3d.html`, accessed at February 2013.

6. Tendulkar, S., M. Beall, M. S. Shephard, K. Jansen, "Parallel Mesh Generation and Adaptation for CAD Geometries", *Proceedings of the NAFEMS World Congress 2011*, 2011, `http://www.scorec.rpi.edu/REPORTS/2011-2.pdf` , accessed at February 2013.

7. Ivanov, E., O. Gluchshenko, H. Andrae, A. Kudryavtsev, "Parallel Software Tool for Decomposing and Meshing of 3D Structures", 2007, `http://www.itwm.fraunhofer.de/fileadmin/ITWM-Media/Zentral/Pdf/Berichte_ITWM/2007/bericht110.pdf`, accessed at February 2013.

8. Chand, K. K., L. Freitag Diachin, X. Li, C. Ollivier-Gooch, E. S. Seol, M. S. Shephard, T. Tautges, H. Trease, "Toward Interoperable Mesh, Geometry and Field

Components for PDE Simulation Development", *Engineering with Computers*, No. 24, pp. 165–182, 2008.

9. Interoperable Technologies for Advanced Petascale Simulations, 2010, `http://www.itaps.org/`, accessed at February 2013.

10. Seol, E. S., M. S. Shephard, "Efficient Distributed Mesh Data Structure for Parallel Automated Adaptive Analysis", *Engineering with Computers*, Volume 22, Issue 3, pp. 197–213, 2006.

11. Shephard, M.S., J.E. Flaherty, H.L. de Cougny, C. Ozturan, C.L. Bottasso, M. W. Beall, "Parallel Automated Adaptive Procedures for Unstructured Meshes", 1995, `http://www.cso.nato.int/Pubs/rdp.asp?RDP=AGARD-R-807`, February 2013.

12. Ozturan, C., "Distributed Environment and Load Balancing for Adaptive Unstructured Meshes", PhD Thesis, Rensselaer Polytechnic Institute, 1995.

13. Karypis, G., V. Kumar, "MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System", 2009, `http://www.cs.umn.edu/~metis` , accessed at February 2013.

14. Chevalier C., F. Pellegrini, "PT-SCOTCH: A Tool for Efficient Parallel Graph Ordering", *Parallel Computing*, Volume 34, Issues 6-8, pp 318–331, 2008, `http://www.labri.fr/perso/pelegrin/scotch/`, accessed at February 2013.

15. Zoltan: Parallel Partitioning, Load Balancing and Data-Management Services, 2000, `http://www.cs.sandia.gov/Zoltan/Zoltan.html`, accessed at February 2013.

16. Elmer, Open Source Finite Element Software for Multiphysical Problems, 1995,

http://www.csc.fi/english/pages/elmer, accessed at February 2013.

17. Ledoux, F., J. C. Weill, Y. Bertrand, "GMDS, Generic Mesh Data Structure", *Research Notes, 17th International Meshing Roundtable*, pp. 31–35, 2008, http://www.imr.sandia.gov/papers/abstracts/Le533.html, accessed at February 2013.

18. Curie Supercomputer, 2010, http://www-hpc.cea.fr/en/complexe/tgcc-curie.htm, accessed at February 2013.

19. Top 500 Supercomputer Sites, 2011, http://www.top500.org/list/2011/06/, accessed at February 2013.

20. PRACE, 2010, http://www.prace-ri.eu/PRACE-in-a-few-words, accessed at February 2013.

21. Shewchuk, J. R., "Lecture Notes on Delaunay Mesh Generation", 2012, http://www.cs.berkeley.edu/~jrs/meshpapers/delnotes.pdf, accessed at February 2013.

22. Zhang, P., "Advancing Front Mesh Generation", 2004, http://www10.informatik.uni-erlangen.de/~pflaum/pflaum/SeminarGrid_04/Refs/AFT_Zhang_28-06-04_Grid.pdf, accessed at February 2013.

23. Schneiders, R., Public Domain and Commercial Mesh Generators List, 1996, http://www.robertschneiders.de/meshgeneration/software.html, accessed at February 2013.

24. TRIANGLE, A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator, 2003, http://www-2.cs.cmu.edu/~quake/triangle.html, accessed at February 2013.

25. GMSH, A Three-Dimensional Finite Element Mesh Generator, 2009, http://www.geuz.org/gmsh/, accessed at February 2013.

26. Chen, T., CAMINO, 1995, http://www-tcad.stanford.edu/tcad/bios/tchen.html, accessed at February 2013.