Partnership for Advanced Computing in Europe

Parallel Mesh Generation, Migration and Partitioning for the Elmer Application

Yusuf Yılmaz, Can Özturan*, Oğuz Tosun, Ali Haydar Özer, Seren Soner

Dept. of Computer Engineering, Bogazici University, Istanbul, Turkey

Abstract

The main goal of this project is to develop a parallel tetrahedral mesh generator based on existing sequential mesh generation software. As sequential mesh generation software, the Netgen mesh generator was used due to its availability as LGPL open source software and its wide user base. Parallel mesh generation routines were developed using the MPI libraries and the C++ language. The parallel mesh generation algorithms developed proceed by decomposing the whole geometry into a number of sub-geometries sequentially on a master node at the beginning and then mesh each sub-geometry in parallel on multiple processors. Three methods were implemented. The first decomposes the CAD geometry and produces conforming surface sub-meshes that are sent to other processors for volume mesh generation. The second and third methods are refinement based methods that also make use of the CAD geometry information. Advantages and disadvantages of each method are discussed. Parallel repartitioning also need to be done in the first method. To facilitate distributed element movements in parallel, a migration algorithm that utilizes "owner updates" rule is developed. Timing results obtained on the Curie supercomputer are presented. In particular, results show that by using a refinement based method, one can generate over a billion element meshes in under a minute.

1. Introduction and Goals of the Project

Elmer is an open source multi-physic simulation software developed by CSC - IT Center for Science (CSC) [1]. Elmer employs finite element method to solve partial differential equations associated with, for example, physical models of fluid dynamics, structural mechanics, electromagnetics, heat transfer and acoustics. Elmer has a simple built-in unstructured mesh generator which can do mesh generation on *simple* geometries. Support for parallel mesh generation and mesh partitioning was requested by Elmer developers. The main objective of this project 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 the Elmer solver especially for complex geometry problems since generation of a massive unstructured mesh is a required pre-processing step of finite element methods. Achievement of these objectives require a scalable parallel mesh generator which is the topic addressed in this project.

---

* Corresponding author. *E-mail address*: ozturaca@boun.edu.tr.

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

In the rest of the paper, a survey of related work is given in Section 2. Section 3 presents the methods that were used to generate mesh in parallel using the sequential Netgen. After the mesh generation, repartitioning and migration of mesh to final destination needs to be carried out; Section 4 addresses these topics. Performance results of the developed parallel mesh generation routines and migration are reported in Section 5. Finally, the paper concludes with a discussion of results in Section 6.

2. Survey of Related Work

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. Delanuay based approach of [3] directly parallelizes the mesh generation at all levels on multicore SMT-based architectures. D3D [4] 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. [5] is also available. ParTGen [6] 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 PartTgen have demonstrated scalability only up to 32 processors. The implementation in this project that is described in Section 3 uses a similar approach as that of ParTgen.

Interoperable Technologies for Advanced Petascale Simulations (ITAPS) [7] is a recent U.S. based initiative involving various laboratories and universities. ITAPS forms one of the centers 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 provided by ITAPs can be traced to the SCOREC's parallel mesh database (PMDB) [8] that implemented an owner updates rule based mesh migration routines 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 that are used in Elmer.

Mesh partitioning algorithms partition the mesh in such a way that load balancing is achieved while communication is minimized. Sequential partitioners Metis [9] and Scotch [10] as well as their parallel versions ParMetis and PT-Scotch are widely used in the parallel computing community. In this

project, a repartitioning of the distributed mesh needed to be done after the mesh generation. In order to do this, ParMetis was used. Mesh migration is needed after repartitioning in order to migrate the tetrahedra to their final destinations provided by the partitioners. Zoltan [11] is another tool that provides parallel partitioning, load balancing and data management services for unstructured meshes.

3. Parallel Mesh Generation Algorithms

The input to the developed parallel mesh generator is a geometry file in .geo format. One can use, for example, the open source Gmsh [12] which has a built-in graphical CAD engine to prepare models in this format. The mesh generated in parallel is output in Elmer mesh file format. Internally, the parallel mesh generator stores mesh data in a similar structure as that of Elmer's mesh file format. Figure-1 (a) shows the (compact) mesh data organization that is used in this project. Tetrahedron-to-vertex adjacency relation is stored for the whole mesh. Face-to-tetrahedron and Face-to-vertex relations are stored only for faces on geometry and partition boundaries. In other related work such as [8], the rich relations shown in Figure-1(b) are used on the whole mesh at the expense of heavy memory usage.

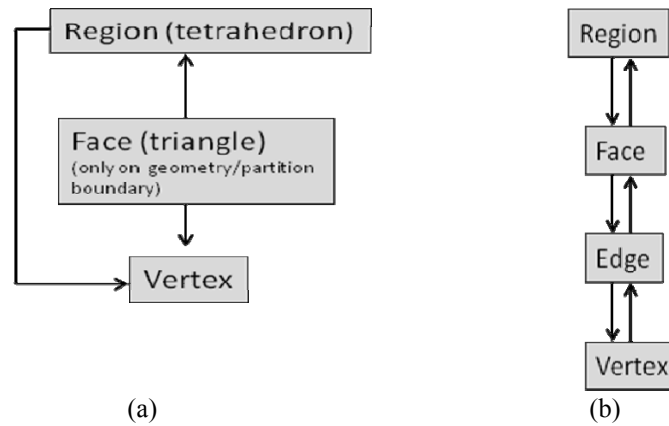(a)   (b)

Figure-1: Mesh entity adjacencies stored in this project (a) and in other related
work [8]

Three methods were used in order to parallelize the mesh generation process. The first method (Method 1), given in Figure-2, 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. match). 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 6. 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 coarse tetrahedra which is roughly balances the load that each processor will encounter while generating the large fine volume mesh. Figure-3 demonstrates execution of Method 1 on the torus geometry.

	Method 1
	On processor 0 **do**
1	• Generate course mesh sequentially by Netgen
2	• Decompose geometry by cutting planes obtained from BSP-Type method like KD-Tree
3	• Generate fine surface mesh of the decomposed geometry
4	• Partition the surface mesh (partition boundaries are conforming)
5	• Distribute the partitioned surface meshes to other processors
	On all processors **do**
6	• Generate volume mesh from the partitioned surface meshes in parallel
7	• Perform Global ID matching
8	• Repartition the mesh in parallel using ParMETIS
9	• Migrate the distributed mesh according to ParMETIS's output
10	• Save the distributed mesh in Elmer's parallel mesh format

Figure-2: Algorithm for Method 1

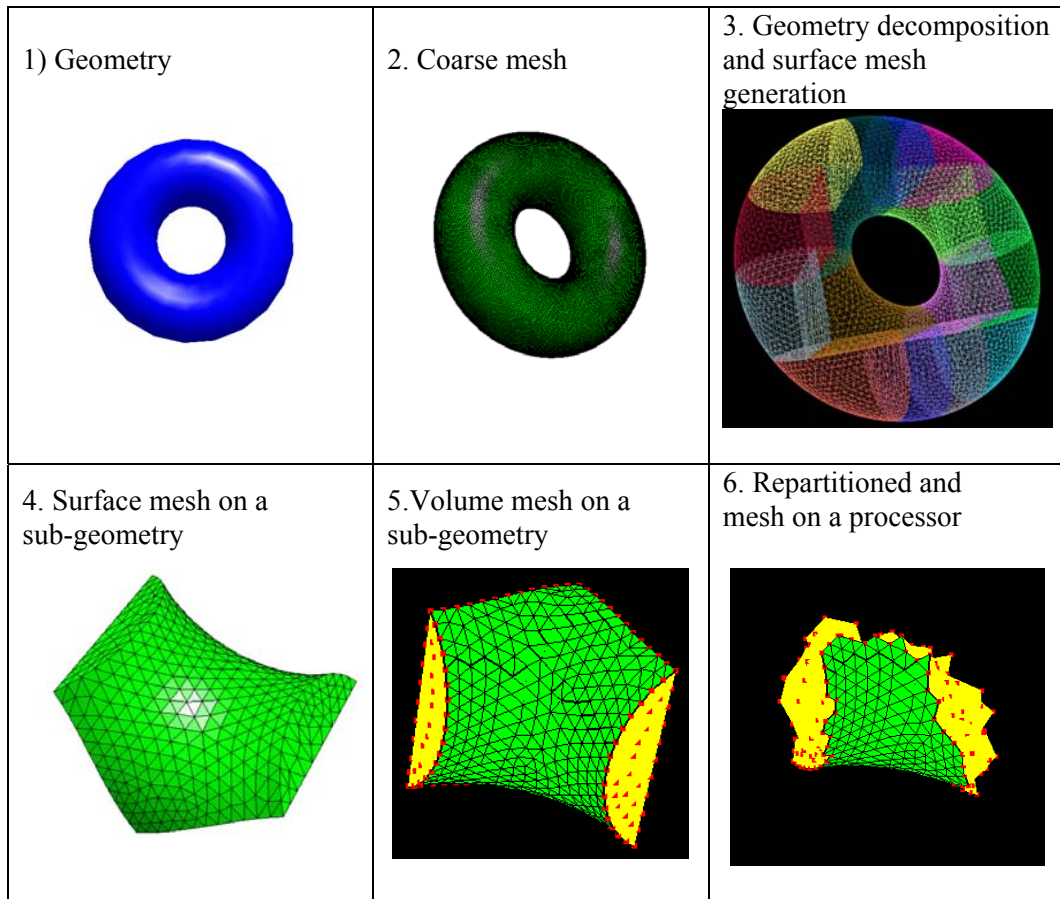| 1) Geometry	2. Coarse mesh	3. Geometry decomposition and surface mesh generation
4. Surface mesh on a sub-geometry	5.Volume mesh on a sub-geometry	6. Repartitioned and mesh on a processor

Fig-3: Mesh generation by Method 1 (geometry decomposition)

Method 2 given in Figure-4 employs a parallel refinement method that uses the geometric information from the CAD model. Larger meshes can be created without much extra computer power by refinement. Netgen refinement process uses a simple 8-way split for tetrahedrons and 4-way split for faces. The refinement of surface elements happens without any processor interaction. Surface and

volume elements are exclusively used by the processor they reside on. Surface elements even do not hold any global information. Volume elements have global ids which are important. By a traversal of volume elements, edges having both of their vertices shared with other processors, can be found. The newly generated shared vertices will reside on these edges. There are three types of new vertices: non-shared vertices, owned shared vertices and non-owned shared vertices. Each vertex, even though it may exist in duplicates on multiple processors, is owned only by a single processor. The concept of entity ownership and how it is determined is explained in detail in Section 4. 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 the assigned global id.

	Method 2
	On processor 0 **do**
1	• Generate a course volume mesh
2	• Partition the mesh sequentially using METIS
3	• Generate global IDs
4	• Distribute the sub-meshes to each processor
	On all processors **do**
5	• Perform parallel refinement of the volume mesh repeatedly until the needed target size is reached
6	• Save the distributed mesh in Elmer's parallel mesh format

Figure-4: Algorithm for Method 2

Method 3 given in Figure-5 is a combination of Methods 1 and 2. It also uses refinement, but this time only the surface mesh is refined. Normally, when refinement occurs in Method 2 (especially when using direct geometric methods like 4-way or 8-way split), the low quality of some elements are passed to their descendants. To make use of Netgen's volume mesh generation optimizations, just the surface mesh can be refined and in the last step Netgen can recreate the whole volume mesh inside the partitions. In this way, the quality of the mesh is expected to be better than the simple volume refinement of Method 2.

	Method 3
	On processor 0 **do**
1	• Generate course volume mesh
2	• Partition the mesh sequentially using METIS
3	• Compute partition boundaries and create a surface mesh
4	• Synchronize Global IDs for nodes and surface elements
5	• Distribute the mesh to other processors
	On all processors **do**
6	• Perform parallel refinement of the surface mesh repeatedly until the needed target size is reached
7	• Generate volume mesh from the surface meshes in parallel
8	• Save the distributed mesh in Elmer's parallel mesh format

Figure-5: Algorithm for Method 3

4. Parallel Partitioning and Migration of Distributed Mesh

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 [9] has been used to repartition the mesh in parallel. ParMetis provides routines

5

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. We can classify distributed mesh entities (faces, edges and vertices) making up a tetrahedron as follows:

1. *Interior entities*: that uniquely reside on a partition (processor), i.e. non-shared entities.

2. *Partition boundary entities:* that are shared by multiple partitions. Since we have a distributed memory model, these entities are replicated on the processors that own the tetrahedron containing these entities.

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 [8] was used in this project 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-6 shows a mesh that has been partitioned into four. Even though three-dimensional meshes are generated in this 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-6, 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.

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. Another approach can be 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 project 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 methods 2 and 3, 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 taken. Call these ids *idv1* and *idv2*. The holders list on each processor that holds these vertices are also taken and intersected. Denote this intersection list of processor ids in sorted order as $(p_{i1}, p_{i2} \ldots, p_{in})$. Then, each processor computes the owner by using the formula:

$$Owner\ id\ index\ in\ the\ list\ =\ (idv1+\ idv2)\ mod\ n$$

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. Note that while doing this, one needs to take care of a subtle special case in which a processor may hold vertices with *idv1* and *idv2*, but does not hold a tetrahedron that uses the edge formed by these vertices. In the future, the method of current owner picking the next owner from among the list of holders randomly can be implemented.

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 hundreds of processors. Figure-7 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 *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 *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.

- Vertex *c*, which is an interior entity, becomes partition boundary entity on destination processors 1 and 2. The current owner, i.e. processor 0, need to form the holders list (1,2) and send it to processors 1 and 2.

- Vertex *a*, which is partition boundary entity, is migrated by both processors 0 and 2 to processor 1. The holders list (0,1,2,3) must be updated on all holders. Holders that will migrate *a* (in this case 0 and 2) inform the owner processor 1 about the destination. The new holders list (1,3) is formed by owner processor 1, and holders (processors 1 and 3) need to be updated.

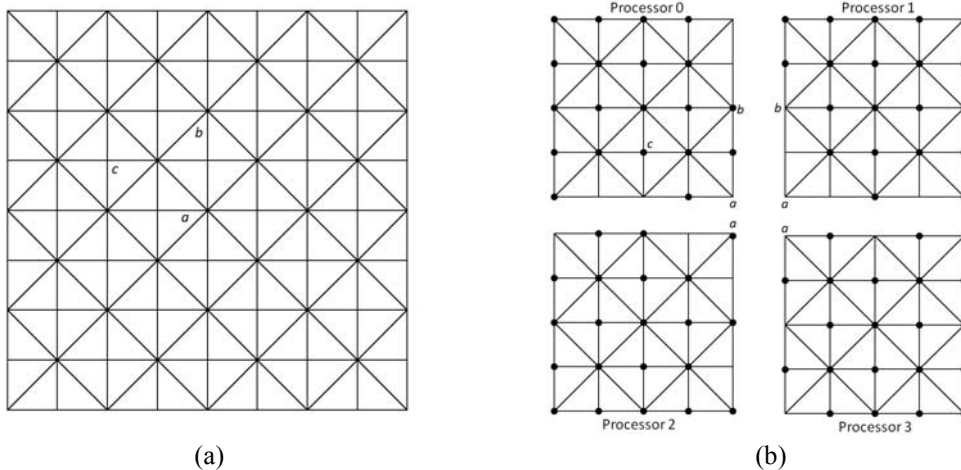(a)   (b)

Figure-6: Mesh (a) partitioned into 4 parts (b) with processor ownership of vertices (bold vertices are owned by the processor holding the vertex)
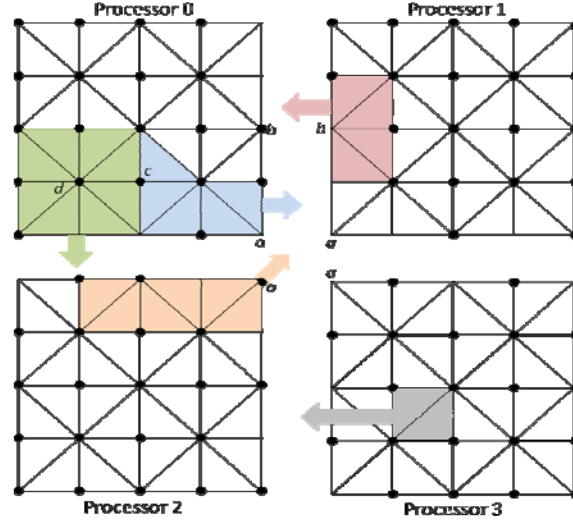
Figure-7: Mesh migration example

The "owner updates" based mesh migration algorithm given in [8] 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 this project, since global ids are used, this algorithm has been further simplified and optimized as described for the cases (*a,b,c,d*) above. The performance of the migration algorithm implemented is reported in the next section.

5. Tests and Results Obtained

The parallel mesh generator was tested on the Curie supercomputer at CEA in France. The 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 methods 2 and 3, 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-8 shows a picture of a mesh generated on the shaft complex geometry. In the rest of the section, we report various performance results.

In Table-1, we show the timing results of parallel mesh generation using geometry decomposition (i.e. Method 1). Method 1 was developed first in our project. It was realized however that it has some problems as exemplified in Figure-9. When the geometry is decomposed by cutting planes, it is possible to have cases like the one shown in Figure-9. 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 is highly 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 a small number of processors as shown in Table-1.
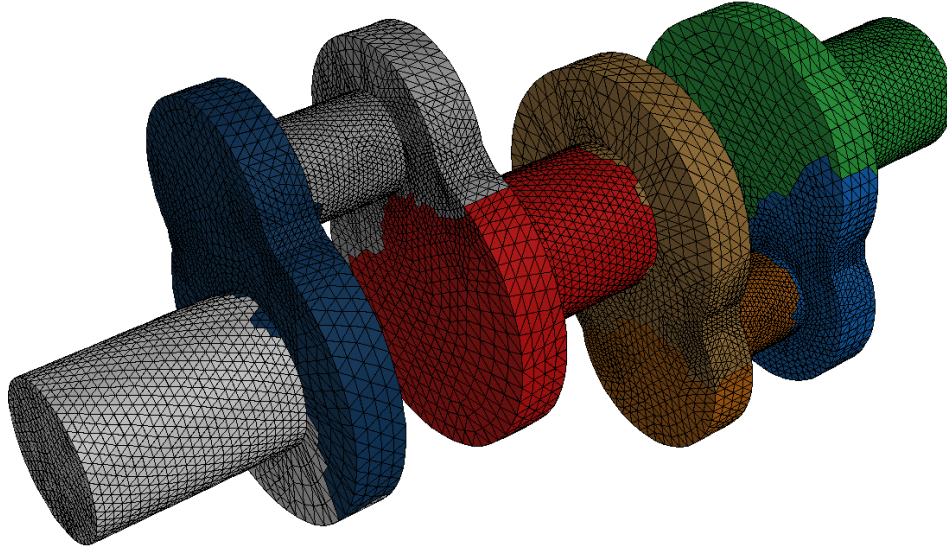
8

Figure-8: Mesh of the complex shaft geometry generated by the parallel mesh generator
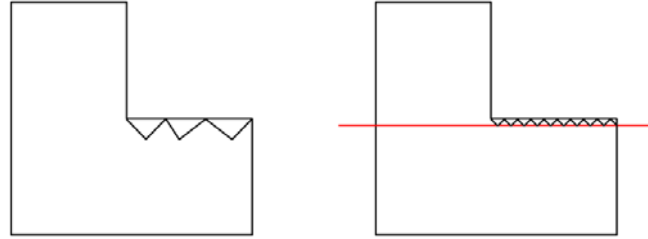

Figure-9: Difficult of making the right cut in Method 1

   In Figure-10, 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. At 1024 processors, the parallel part is about 10 seconds. Increasing the number of processors beyond this point is 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 energy consumption, which is a major issue in the current supercomputers, will be reduced.

Table-1: Timings of mesh generation using geometry decomposition (Method 1)

| No. of Cores	Geometry	Surface Mesh Size (no. of Faces)	Volume Mesh Size (no. of tetrahedra) in	Time Taken (s)				
				Geometry Decomposition	Parallel Volume Mesh Generation	Parallel Repartitioning	Migration	Total
16	Cube	1.3 M	14 M	13	401	19	6	439
16	Torus	1 M	7.2 M	12	224	8	3	247
16	Sphere	1M	11 M	11	387	17	5	420
32	Cube	425K	5 M	5	107	2	0	114
32	Torus	1.08 M	8 M	14	146	6	2	168
32	Sphere	1.15 M	11 M	15	217	8	3	243
64	Cube	2 M	37 M	21	224	10	5	260
64	Torus	1.5 M	12.5 M	20	162	7	3	192
64	Sphere	0.55 M	4.4 M	9	47	1	1	58
64	Cube	8 M	330 M	84	2328	97	58	2567
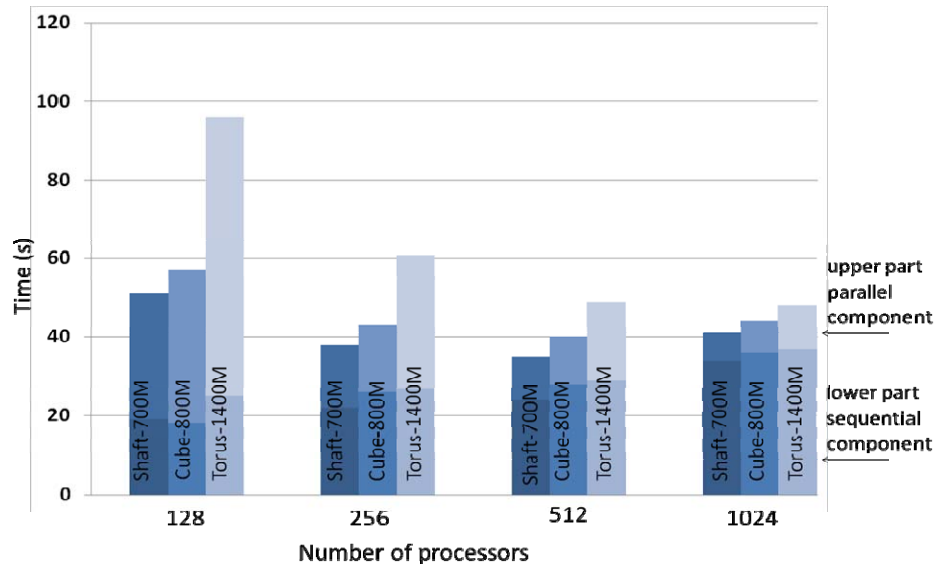
Figure-10: Timings of mesh generation using refinement based method 2

Further performance tests of the migration algorithm were also carried out. Table-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 shows that migration of about 1.5 million elements took under a minute. Considering that migration is to be performed once after repartitioning, these results are acceptable.

Even though Method 3 was implemented, it was not exhaustively tested on a large number of processors due to running out of allocation time on Curie. Some preliminary results are listed in the Table-3. It is expected that Method 3 will be similar cost-wise to Method 1, taking long running times, but should scale to larger number of processors producing higher quality meshes than Method 2.

Table-2: Scalability Test of Mesh Migration

No. of Processors	Time Taken For Migration (s)	Total No. of Tetrahedra
16	13	100 M
32	9	100 M
64	8	100 M
128	8	100 M
512	14	100 M
128	50	800 M
512	57	800 M

Table-3: Preliminary timings of mesh generation using Method 3

No. of Cores	Geometry	Surface Mesh Size (no. of Faces)	Volume Mesh Size (no. of tetrahedra) in	Time Taken (s)		
				Initial sequential phase	Parallel Volume Mesh Generation	Total
16	Cube	0.2 M	3 M	3	109	112
16	Cube	0.8 M	23 M	2	756	758
64	Cube	0.2 M	3.4 M	3	76	79

6. Discussion and Conclusions

It can be concluded that the main goal of this project, that is, the generation of large unstructured meshes with sizes in the hundreds of millions range on complex geometry, has been achieved. Moreover, a mesh with size 1.4 billion could be generated in under a minute in Method 2. These capabilities were not available to the Elmer users who usually used 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 are not suitable for large numbers of processors. It also has problems when geometry is to be decomposed. Automation of geometry decomposition is difficult.

- For refinement based methods, mesh generation of billion(s) of elements are possible on distributed machines.

- Since the size of the mesh that can be initially generated sequentially dictates the size and the quality of the mesh final fine mesh, a large memory fat node is required for the initial sequential coarse mesh generation component.

- Rather than generate a mesh once and save it to files, it may be possible to generate the mesh every time the application runs. In this way, a lot of expensive I/O can be saved.

- 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. Such a malleable job model in which thousands of processors will be acquired later will be much more

efficient in terms of energy consumption on supercomputers. To be able to run such malleable applications, however, job scheduler support for malleability is needed.

In the future, the source code of the parallel mesh generation routines will be made available at address: http://code.google.com/p/parallel-netgen/.

Acknowledgements

References

[1] Elmer, Open Source Finite Element Software for Multiphysical Problems, CSC - IT Center for Science, Finland, http://www.csc.fi/english/pages/elmer/.
[2] N. Chrisochoides, A Survey of Parallel Mesh Generation Methods, Scientific Computing Group Reports, ID: 2005-9, Brown University, Providence RI - 2005.
[3] F. Blagojevic, A. Chernikov and D. Nikolopoulos, A multigrain Delaunay mesh generation method for multicore SMT-based architectures, Journal of Parallel and Distributed Systems, Vol. 69 (7), pages 589-600, 2009.
[4] D3D mesh generator, http://mech.fsv.cvut.cz/~dr/d3d.html
[5] MeshSim, Simmetrix Inc., http://www.simmetrix.com.
[6] E. Ivanov, O. Gluchshenko, H. Andrä, A. Kudryavtsev, Parallel software tool for decomposing and meshing of 3d structures, Fraunhofer-Institut für Techno- und Wirtschaftsmathematik, 2007, http://www.itwm.fraunhofer.de/fileadmin/ITWM-Media/Zentral/Pdf/Berichte_ITWM/2007/bericht110.pdf.
[7] ITAPS, Interoperable Technologies for Advanced Petascale Simulations, http://www.itaps.org/.
[8] M.S. Shephard, J.E. Flaherty, H.L. de Cougny, C. Ozturan, C.L. Bottasso, M. W. Beall, Parallel Automated Adaptive Procedures for Unstructured Meshes, AGARD-FDP-VKI Special Course on "Parallel Computing in CFD", held at the VKI, Rhode-Saint-Genese, Belgium, May 1995 and at NASA Ames, USA, October 1995, AGARD Report R-807. http://www.rto.nato.int/Pubs/rdp.asp?RDP=AGARD-R-807.
[9] G. Karypis and V. Kumar, MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, University of Minnesota, Minneapolis, MN, http://www.cs.umn.edu/~metis , 2009.
[10] C. Chevalier and F. Pellegrini , PT-SCOTCH: A tool for efficient parallel graph ordering. Parallel Computing, 34(6-8), pp 318-331, 2008. http://www.labri.fr/perso/pelegrin/scotch/.
[11] Zoltan: Parallel Partitioning, Load Balancing and Data-Management Services, http://www.cs.sandia.gov/Zoltan/Zoltan.html.
[12] Gmsh, http://geuz.org/gmsh/