Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 437480, 10 pages
http://dx.doi.org/10.1155/2015/437480

Research Article

Generating Multibillion Element Unstructured Meshes on Distributed Memory Parallel Machines

Seren Soner and Can Ozturan

Department of Computer Engineering, Bogazici University, 34342 Istanbul, Turkey

Correspondence should be addressed to Can Ozturan; ozturaca@boun.edu.tr

Received 30 June 2014; Revised 2 March 2015; Accepted 10 May 2015

Academic Editor: Damian Rouson

We present a parallel mesh generator called PMSH that is developed as a wrapper code around the open source sequential Netgen mesh generator. Parallelization of the mesh generator is carried out in five stages: (i) generation of a coarse volume mesh; (ii) partitioning of the coarse mesh; (iii) refinement of coarse surface mesh to produce fine surface submeshes; (iv) remeshing of each fine surface submesh to get a final fine mesh; (v) matching of partition boundary vertices followed by global vertex numbering. A new integer based barycentric coordinate method is developed for matching distributed partition boundary vertices. This method does not have precision related problems of floating point coordinate based vertex matching. Test results obtained on an SGI Altix ICE X system with 8192 cores confirm that our approach does indeed enable us to generate multibillion element meshes in a scalable way.

1. Introduction

We developed a tool called PMSH for facilitating fast generation of unstructured multibillion element tetrahedral meshes (grids) on complex geometries for the OpenFOAM computational fluid dynamics (CFD) package [1]. PMSH is developed as a C++ wrapper code around the popular open source sequential Netgen mesh generator [2]. OpenFOAM provides a mesh generator called blockMesh for simple geometries. The blockMesh utility is a multiblock mesh generator that generates hexahedral meshes from a text configuration file. For complex geometries OpenFOAM also provides a mesh generation utility called snappyHexMesh which generates hexahedral meshes. The snappyHexMesh utility works more like a mesh sculptor rather than a generator. It takes an existing mesh such as the one produced by blockMesh and chisels out a mesh on a complex geometry that is given in STL format. The snappyHexMesh utility has advanced features like the ability to run in parallel and being able to redistribute the mesh so as to perform automatic load balancing. Both utilities, snappyHexMesh and blockMesh, are not as advanced as other commercial or open source mesh generator packages for producing quality tetrahedral meshes on complex geometries. Therefore, there is a great need in the OpenFOAM community for tools that will enable researchers to generate massive tetrahedral meshes on complex geometries.

Löhner states in [3] that "*grid sizes have increased by an order of magnitude every 5 years and that presently, grids in of the order of 10^9 elements are commonly used for leading edge applications in the aerospace, defense, automotive, naval, energy and electromagnetics sectors.*" Löhner also mentions that "*for applications where remeshing is an integral part of simulations, for example, problems with moving bodies or changing topologies, the time required for mesh regeneration can easily consume a significant percentage of the total time required to solve the problem.*" Geometry and mesh generation related concerns have also been voiced in a recent Applied Mathematics for Exascale Report [4] by the US based Exascale Mathematics Group. Therefore, as we move towards exascale computing, the ability to generate massive multibillion tetrahedral meshes especially on complex geometries will be in more demand in the future.

This paper is organized as follows: Section 2 reviews the previous work done in the area of mesh generation and refinement. Section 3 presents the details of our algorithms

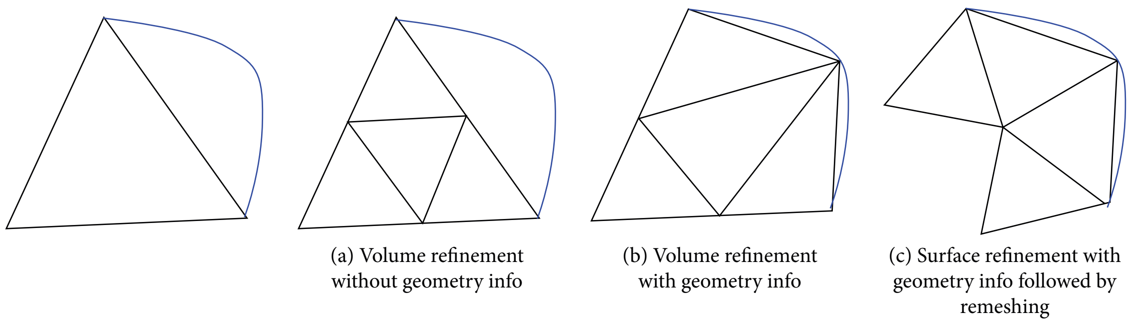

(a) Volume refinement without geometry info
(b) Volume refinement with geometry info
(c) Surface refinement with geometry info followed by remeshing

Figure 1: Given coarse mesh volume refinements of the mesh without (a) or with (b) geometry information and remeshing after surface refinement with geometry information (c).

and data structures used. Section 4 lists a few programming issues and the modifications that needed to be done in order to fix some bugs in Netgen. Section 5 presents the results of various parallel mesh generation tests we have carried out on an SGI Altix ICE X system with 8192 cores. Finally, conclusions and future work are presented in Section 6.

2. Previous Work

Parallel mesh generation has been addressed in several reported works: Dawes et al. [5] present bottom up octree based parallel mesh generation routines. Antonopoulos et al. [6] parallelize Delaunay based mesh generation on multicore SMT-based architectures. D3D [7] is an MPI based parallel mesh generator that employs octree based mesh generation. MeshSim by Simmetrix Inc. is a commercial multithreaded and distributed parallel mesh generator [8]. ParTGen [9] is a parallel mesh generator that uses a geometry decomposition based approach to decouple the meshing process into parallel mesh generation tasks. Performance tests of PartTgen have demonstrated scalability only up to 32 processors. The approach taken in PMSH is similar to that of ParTGen.

The recent work by Löhner [3] involves geometry decomposition and the use of advancing front technique on multiple subdomains to generate meshes in parallel. It is reported that a billion-element sized mesh has been generated in roughly forty minutes on a 512-core SGI ITL machine.

Pamgen [10] is another parallel mesh generation library within the Trilinos Project that produces hexahedral or quadrilateral meshes of simple topologies. It is stated in its FAQ page that the number of elements PAMGEN can generate is limited to just above two billions (because of 32-bit integer limitation).

CGAL is a computational geometry algorithms library developed at INRIA. Pion et al. present 3D parallel Delaunay triangulation work in [11]. CGAL also provides a parallel mesh generation capability based on TBB (Threading Building Blocks) library [12].

Figure 1 illustrates refinement based approaches that can be used to generate massive meshes. Uniform mesh refinement without the use of geometric information, as shown in (a), works very fast in parallel, is quite scalable to tens of thousands of processors, and can be used to produce meshes with tens of billions of elements. The works by Houzeaux et al. [13] and Kabelikova et al. [14, 15] can be shown as examples of uniform refinement. These approaches, however, cannot be used on problems involving complex geometries with curved boundaries. In such cases, they cannot accurately approximate the boundary of the geometry. An alternative method is to do uniform refinement but also to make use of geometric information as shown in Figure 1(b). In [16], mesh multiplication technique with surface correction and volume node movement is presented. This approach is also taken by Yilmaz et al. [17]. In this case, in addition to the mesh entities, one needs to have access to the geometry. Such an implementation can be achieved by utilizing mesh generation libraries as, for example, Netgen which is used in [17]. One and a half billion element meshes have been generated in this way in one- to two-minute time on one thousand cores. Yilmaz et al. also provide results for a method involving decomposition of geometry in the flavour of Figure 1(c), where meshing of decomposed geometry is performed. However, geometry decomposition into a large number of subgeometries introduces problems caused by thinly cut subdomains. Using this method, meshes of about a few hundred million elements could be accomplished on up to 64 cores on simple geometries. A preliminary implementation was also reported in [17], which generates a surface mesh from a geometry, refines the surface mesh, and generates a fine mesh from it in the style of Figure 1(c). This implementation had some problems and some preliminary results involving generation of a few million element meshes on few tens of cores were reported.

Table 1 summarizes the mesh generation methods and the degree to which they use the geometric information. In our current PMSH work, the approach shown in Figure 1(c) involving generation of a coarse mesh, extraction of a surface mesh, and remeshing from a refined and projected fine surface mesh has been completely redesigned using new data structures and has been successfully implemented. The resulting implementation is robust and enabled us to generate multibillion meshes on up to eight thousand processors (note that 8K processor is not a limitation of our PMSH but rather it was the maximum number of cores for the maximum allowable memory we could get on the specific supercomputer that we used for our tests). Also, note that both our PMSH and

Table 1: Comparison of parallel mesh generation approaches.

Approach	Quality of modelling geometry
Mesh multiplication (uniform refinement) [13–15]	Worst
Mesh multiplication with geometry correction [16, 17]	↑
Geometry partitioning [9] and PMSH	↓
Fully parallel meshing [3, 5–8, 12]	Best

Löhner's [3] work employ a similar approach with Löhner using his own internal mesh generator and PMSH using the external Netgen mesh generator augmented with extra PMSH data structures, routines, and modified Netgen libraries.

Finally, we note that when new mesh entities are created in parallel, duplicate entities on the partition boundaries result which need to be matched. A somewhat similar problem occurs in the OpenFOAM reconstructParMesh utility. We quote Piscaglia et al. [18] on this problem: "*. . .the point, face and cell processor addressing lists, generated at decomposition time, are no longer valid once topological changes have occurred. As a consequence, the only way to reconstruct a parallel case involving dynamic mesh with topological changes consist of using an algorithm based on geometry instead than topology. . .*." This geometric approach, however, operates on floating point numbers and also requires a specified tolerance. To resolve this complicated problem, we contribute a new integer barycentric coordinates based approach for parallel duplicate entity matching. Our approach is actually a simple solution whose simplicity makes it even more valuable, since it drastically reduces the effort required for parallel programming of unstructured mesh applications. This approach is explained in detail in Section 3.

3. PMSH Parallel Mesh Generation Algorithm

Our mesh generation algorithm proceeds in five main stages: (i) generation of a coarse volume mesh, (ii) partitioning of the coarse mesh to get submeshes, each of which will be processed by a processor, (iii) extraction and refinement of coarse surface submeshes to produce fine surface submeshes, (iv) remeshing of each fine surface submesh to get the final fine volume mesh, and (v) matching of distributed duplicate partition boundary vertices followed by global vertex numbering.

Given the list of symbols and their definitions in Abbreviations, Algorithm 1 presents the detailed steps of our parallel mesh generation algorithm. Steps (2)–(4) basically correspond to stage (i) that involves generation of an initial coarse mesh. Firstly, the geometry description Ω is loaded. Geometry file can be in STL or Netgen's CSG (.geo) format. A coarse mesh is then generated from this geometry. The coarse mesh basically enables us to subdivide our domain into P subdomains in a load balanced way.

In step (5), corresponding to stage (ii), version 5.1.0 of METIS [19] partitioner is used to partition the coarse mesh in a face-connected way. The coarse volume mesh and its partitioning are available on all the processors and hence allow each processor to compute processor adjacencies independently. In our current implementation, we let each processor generate the coarse volume mesh and its partitioning redundantly. Even though energy and resources can be saved if a dynamic malleable job model is used (i.e., starting with a single processor and increasing the number of allocated processors after step (5)), there are some complications involving reconstruction of Netgen's internal data structures on dynamically spawned processors. The malleable job model will be considered in the future.

Steps (6)–(9) correspond to stage (iii), which involves extraction and refinement of the coarse surface submesh to produce a fine surface submesh. Faces which are on the geometric boundary and faces which are shared by elements that are on different processors are extracted and a coarse surface submesh is formed. This surface submesh is then refined uniformly r times to form a fine submesh. Refinement is done by our own code and not by Netgen code. A queue of faces to be refined is maintained. A face to be refined is pushed into this queue. After refinement, if it is to be refined again (i.e., $r-1$ more times), the newly created faces are also pushed into the queue. Using Netgen's library, newly formed vertices are projected onto the actual geometry boundary as part of step (9). Currently projection works only when meshing Netgen's CSG (.geo) geometry files. It does not work properly for STL geometries because sometimes Netgen's projection routine returns incorrect projections for some points.

Since we need to match new surface vertices with their counterparts on the neighbouring processors later on, we need to keep track of the addresses of newly created vertices. Our own refinement code helps us to do this tracking. We know the IDs of the coarse mesh vertices, since the same coarse mesh is present on all processors. The newly created vertices, however, have local IDs on each processor and these are different from their counterparts on other processors. Even though one can always use x, y, z coordinates of these vertices to match them, such a method uses floating point comparisons and can be sensitive to precision related errors. The generation of new vertices is done in different orders on different processors. Since multibillion element meshes are to be generated, precision related problems are more likely to happen due to very small element sizes. As a result, we wanted to come up with a new integer based vertex identification system.

The new vertex identification system we came up with makes use of integer barycentric coordinates. All processors know the IDs of the coarse mesh vertices. Hence, given a coarse surface mesh face that is defined with vertices having indices i, j, and k, the address of this face can be identified easily by employing a map that maps the key (i, j, k) to the corresponding local face address on each processor. However, when new faces and vertices are created as a result of refinement, their orders of creation are different on each processor. There can be different connectivity types as shown in Figure 2. Vertex connectivity is easy to resolve since the vertex connectivities of partitions can be established by the coarse mesh vertices which are known by all processors. The newly created vertices on edges and faces introduce

```
(1) **On all** processors $i = 0 \cdots P - 1$ **do**
(2)   Load geometry $\Omega$
(3)   Generate coarse surface mesh $\partial\mathcal{T}$ for $\Omega$
(4)   Generate coarse volume mesh $\mathcal{T}$
(5)   Partition the coarse volume mesh $\mathcal{T}$ into $P$ parts $\mathcal{T}_i$, $i = 0 \cdots P - 1$
(6)   Extract coarse surface mesh faces $\partial\mathcal{T}_i^f$ for all partitions $i = 0 \cdots P - 1$
(7)   Construct surface mesh $\partial\mathcal{T}_i'$ from $\partial\mathcal{T}_i^f$
(8)   Perform $r$-level refinement of surface mesh $\partial\mathcal{T}_i'$ to get fine surface mesh $\partial\mathcal{T}_i''$ and record barycentric global IDs of each
      newly created vertex
(9)   Project new surface vertices in $\partial\mathcal{T}_i''$ to the geometric boundary $\Omega$
(10)  Compute partition boundary vertex adjacencies to adjacent processors by using a map that uses barycentric global IDs
      as keys
(11)  Generate fine volume mesh $\mathcal{T}_i''$ from the fine surface mesh $\partial\mathcal{T}_i''$
(12)  Compute owners of partition boundary vertices held by adjacent processors (called holders)
(13)  Compute global integer IDs of owned vertices
(14)  Inform global integer IDs of the owned vertices to the holder processors by using barycentric global IDs as keys
      to locate the corresponding vertices on adjacent holder processors
(15)  Output fine volume mesh $\mathcal{T}_i''$ in OpenFOAM format
(16) **enddo**
```

Algorithm 1: PMSH parallel mesh generation algorithm.

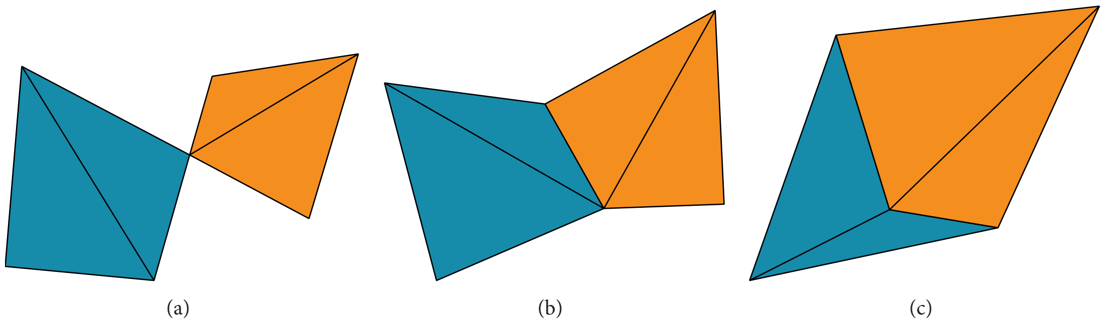
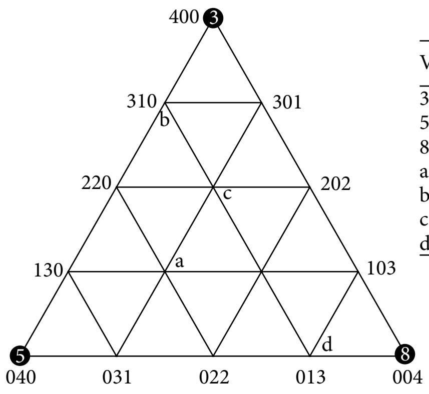


Figure 2: Connectivity types: vertex (a), edge (b), and face (c) connectivities.

complications because their orders of creation are different. Integer based barycentric coordinates help us to address all three types of connectivities and identify all types of vertices (coarse or newly created) by a single mechanism. We define barycentric global IDs as a sorted sequence of three (*vertex id*, *integer barycentric coordinate*) pairs, that is, as $[(i, \alpha), (j, \beta), (k, \gamma)]$. Here, i, j, k are either 0 or indices of vertices from $\partial\mathcal{T}^v$ satisfying $i < j < k$ and α, β, and γ are integer barycentric coordinates with $\alpha + \beta + \gamma = 2^r$. Figure 3 shows an example face defined by coarse mesh vertices 3, 5, and 8 and the newly created vertices as a result of 2-level refinement. The table in Figure 3 shows the barycentric global IDs of some vertices.



Vertex	Global barycentric IDs
3	[(0,0), (0,0), (3,4)]
5	[(0,0), (0,0), (5,4)]
8	[(0,0), (0,0), (8,4)]
a	[(3,1), (5,2), (8,1)]
b	[(0,0), (3,3), (5,1)]
c	[(3,2), (5,1), (8,1)]
d	[(0,0), (5,1), (8,3)]

Figure 3: Integer based barycentric coordinates and global barycentric IDs when two levels of refinement are done on coarse mesh face with vertices 3, 5, and 8.

Note that even though there are six pieces of information in a barycentric global ID compared with three pieces of information in x, y, z coordinates, storage-wise barycentric global IDs are more advantageous. x, y, z coordinates require three doubles (24 bytes). A barycentric coordinate (α or β or γ) can be represented using $r + 1$ bits. The initially generated coarse mesh is small, one or two million at most. A 32-bit unsigned integer is more than enough to fit both the $r + 1$ bits as well as the vertex ID in the coarse mesh. Since we have 3 pairs, 3 unsigned integers, that is, a total of 12 bytes, are actually sufficient.

We also note that, in the literature, applications of barycentric coordinates can be found in many areas such as finite element analysis and computer graphics. Recently, in particular, in the area of computer graphics, Boubekeur and Schlick [20, 21] made use of barycentric coordinate based interpolation in generic and adaptive mesh refinement kernels for GPUs. Our use of barycentric global IDs for distributed vertex matching in PMSH is yet another novel application of barycentric coordinates.

In order to compute vertex adjacencies of processors, adjacent processor IDs are inserted into a map whose keys are barycentric global IDs. This is done in step (10). After this step, the fine volume submesh is generated from the fine surface submesh in step (11). OpenFOAM mesh format uses global vertex IDs. Therefore, we need to assign global numbers to all vertices. In steps (12)-(13), the global numbers of partition boundary vertices are generated by first assigning an owner to one of the processors holding a copy of the vertex. Given a sorted list L of processors that hold a vertex with barycentric global ID $[(i, \alpha), (j, \beta), (k, \gamma)]$ the pth processor in the list is calculated as follows:

$$p = (i + \alpha + j + \beta + k + \gamma) \mod |L| . \quad (1)$$

Here, $|L|$ is the number of holder processors of the vertex. This method of determining owners is similar to our earlier approach given in [17]. Once owner processors are determined, owner processors generate global numbers for each vertex they own. A prescan operation is carried out among processors to compute prefix counts of owned vertices on each processor. From these counts the starting point of global numbering on each processor can be determined. Once global numbers for partition boundary vertices are determined, the owner processors then send these numbers to holder processors in step (14). In this way, all integer global IDs for all vertices (whether they are internal or on partition boundary) are determined. Finally, in step (15), the mesh is output in OpenFOAM format using these integer global IDs.

As data structures, Standard Template Library (STL) maps are used to store various surface mesh data. The main ones are for the storage of geometry and partition boundary faces, global barycentric IDs of partition boundary vertices, and processor adjacencies.

4. Programming Issues

In this section, we consider programming issues that arose while developing PMSH. While PMSH runs, each processor meshes a different subgeometry in parallel during fine mesh generation. Since we are on a distributed memory machine, each processor has a different address space. The new fine mesh entities have completely different addresses and different orders of creation. As a result, matching these entities to their correspondents on other processors is nontrivial. This situation does not arise in mesh multiplication approaches, since in these approaches all tetrahedrons are refined uniformly and hence one can predict the addresses of newly created entities. Our barycentric coordinates approach helps us to resolve this programming issue in a simple way.

On a petascale machine, one wants to minimize expensive communication costs. In particular, one wants to reduce the number of messages. Large numbers of messages put on the communication network, besides introducing large latency costs, cause contention on the network (by also interfering with other users packets). As a result, one of our major objectives has been to reduce the number of messages. We do this by performing all levels of refinement before doing any communication. This, however, introduces a complication, because large numbers of new vertices are created on faces and edges and their orders of creation and addresses are completely different on neighboring processors. Again, our simple barycentric coordinates approach enables us to agglomerate multiple communication stages of refinement levels into a single communication stage, hence reducing the number of messages by a factor equal to the number of refinement levels.

4.1. Bug Fixes and Modifications Made in Netgen 5.1 Code. During the development of this project, Netgen 5.1 was the most recent version. Our modifications and fixes have been made on this version. When we generated fine surface meshes after multiple levels of refinement and passed them to Netgen for fine volume generation (step (11) in the Algorithm 1), we encountered Netgen crashes on a few processors. This happened repeatedly and prevented us from obtaining massive meshes. We examined Netgen code and traced the bugs. A number of fixes as well as some other modifications that let us access some functions within Netgen were made to Netgen 5.1 code; these are described in the PMSH website at https://code.google.com/p/pmsh/.

The modifications enabled our tests to run successfully. Afterwards, we were able to generate multibillion element meshes. The bugs we discovered were also reported to the Netgen developer.

5. Tests and Results

We have meshed Onera M6 wing, sphere, and shaft geometries shown in Figure 4 on NTNU's Vilje system. Sphere is a simple geometry whereas Onera M6 wing and shaft are moderately complex geometries. Vilje is an SGI Altix ICE X system that has 936 nodes available to academic users. Each node has 2 Intel Xeon E5-2670 (Sandy Bridge) processors with a total of 16 cores and 28 GB of memory. Table 2 shows the results of the mesh generation tests we have performed. On Vilje, due to restrictions, a job can be allocated at most 14.3 TB total memory on up to 512 nodes. Therefore, we submitted tests that would satisfy this memory limit. If this limit was higher, we would have been able to run our tests on higher numbers of cores and generate even bigger meshes than what is shown in Table 2.

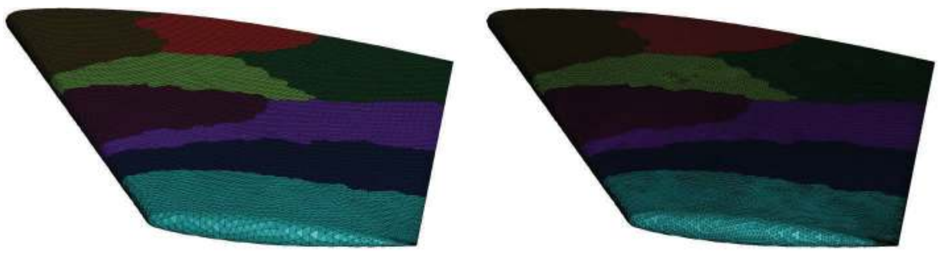
The columns times taken by coarse mesh $\mathscr{T}$ and fine mesh $\mathscr{T}''$ show the timings of steps (3)-(4) and steps (5)–(14), respectively. Total timing is the summation of these two timings. Geometry input and mesh output are not included in the timings. Note that coarse mesh $\mathscr{T}$ is the initial mesh generated on each processor whereas fine mesh $\mathscr{T}''$ is the global mesh over all processors (i.e., union of all $\partial\mathscr{T}_i''$).

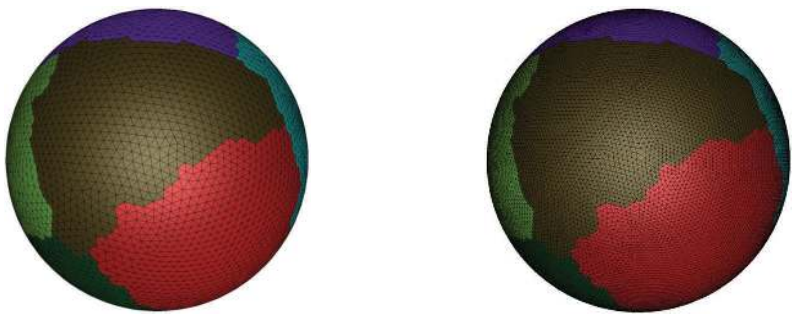
Table 2: Mesh generation results.

Geometry	Number of cores	Allocated job memory (TB)	Refinement levels (r)	Mesh size (M)		Time taken by (s)			Estimated efficiency (%)	Estimated speedup
				Coarse $\mathcal{T}$	Fine $\mathcal{T}''$	Coarse $\mathcal{T}$	Fine $\mathcal{T}''$	Total		
onera m6.stl	1024	7.2	4	0.2	156.1	78.95	96.08	175.03	54.94	563
	2048	14.3	4	0.2	174.8	78.93	95.38	174.31	54.74	1121
	4096	14.3	4	0.2	183.1	85.67	55.16	140.83	39.18	1605
	8192	14.3	4	0.2	189.6	81.97	51.27	133.24	38.49	3153
	1024	7.2	5	0.2	1014.7	78.96	715.3	794.26	90.07	922
	2048	14.3	5	0.2	1114.3	78.89	521.68	600.57	86.87	1779
	4096	14.3	5	0.2	1123.3	82.73	316.04	398.77	79.26	3246
	8192	14.3	5	0.2	1152.7	82.14	152.66	234.8	65.02	5327
	1024	7.2	4	1.2	929.5	483.15	700.91	1184.06	59.24	607
	2048	14.3	4	1.2	970.9	482.69	331.96	814.65	40.78	835
	4096	14.3	4	1.2	909.1	537.04	202.96	740	27.44	1124
	8192	14.3	4	1.2	1028.9	510.94	152.93	663.87	23.05	1888
	2048	14.3	5	1.2	7030.7	483.16	2719.29	3202.45	84.92	1739
	4096	14.3	5	1.2	6393.5	538	1831.23	2369.23	77.3	3166
	8192	14.3	5	1.2	7209.8	510.59	713.66	1224.25	58.3	4776
sphere.geo	1024	7.2	4	0.2	146.7	43.88	69.72	113.6	61.41	629
	2048	14.3	4	0.2	176.7	44.01	32.32	76.33	42.37	868
	4096	14.3	4	0.2	198.7	47.02	23.6	70.62	33.43	1369
	8192	14.3	4	0.2	217	45.04	10.8	55.84	19.35	1585
	1024	7.2	5	0.2	1048.4	43.95	488.25	532.2	91.75	940
	2048	14.3	5	0.2	1278.4	44.02	230.87	274.89	83.99	1720
	4096	14.3	5	0.2	1367.6	47	138.46	185.46	74.66	3058
	8192	14.3	5	0.2	1476.7	46.74	92.01	138.75	66.32	5433
	1024	7.2	5	0.3	1625.5	181.18	796.93	978.11	81.49	835
	2048	14.3	5	0.3	1773.3	184.42	583.59	768.01	76	1556
	4096	14.3	5	0.3	2134.4	242.52	204.96	447.48	45.82	1877
	8192	14.3	5	0.3	2163.3	232.15	133.28	365.43	36.48	2988
sphere.stl	1024	7.2	4	0.4	296.4	94.13	114.04	208.17	54.83	561
	2048	14.3	4	0.4	301.6	93.81	69.17	162.98	42.47	870
	4096	14.3	4	0.4	365.4	97.91	45.35	143.26	31.67	1297
	8192	14.3	4	0.4	417.4	96.31	40.41	136.72	29.57	2422
	1024	7.2	5	0.4	2120	94	924.97	1018.97	90.78	930
	2048	14.3	5	0.4	2131.1	93.76	525.56	619.32	84.87	1738
	4096	14.3	5	0.4	2584.5	97.82	287.31	385.13	74.61	3056
	8192	14.3	5	0.4	2840.7	96.59	166.04	262.63	63.23	5180
	1024	7.2	4	1.4	829.5	286.32	319.88	606.2	52.81	541
	2048	14.3	4	1.4	931.4	286.96	201.49	488.45	41.28	845
	4096	14.3	4	1.4	900.4	299.03	150.31	449.34	33.47	1371
	8192	14.3	4	1.4	1001.5	294.21	105.18	399.39	26.34	2158
	1024	7.2	5	1.4	6392.3	286.51	2938.7	3225.21	91.13	933
	2048	14.3	5	1.4	6847.5	286.88	2212.16	2499.04	88.53	1813
	4096	14.3	5	1.4	6283.1	298.83	912.7	1211.53	75.34	3086
	8192	14.3	5	1.4	7053.6	293.56	513.41	806.97	63.63	5212

Table 2: Continued.

Geometry	Number of cores	Allocated job memory (TB)	Refinement levels (r)	Mesh size (M)		Time taken by (s)			Estimated efficiency (%)	Estimated speedup
				Coarse $\mathcal{T}$	Fine $\mathcal{T}''$	Coarse $\mathcal{T}$	Fine $\mathcal{T}''$	Total		
shaft.geo	1024	7.2	4	0.2	139.6	37.37	52.55	89.92	58.48	599
	2048	14.3	4	0.2	166.8	37.4	28.61	66.01	43.37	888
	4096	14.3	4	0.2	182.5	39.01	22.11	61.12	36.19	1482
	8192	14.3	4	0.2	221.4	38.42	25.73	64.15	40.12	3286
	1024	7.2	4	0.4	281.4	75.21	110.85	186.06	59.62	610
	2048	14.3	4	0.4	295.6	75.7	55.71	131.41	42.42	869
	4096	14.3	4	0.4	351.7	78.87	40.67	119.54	34.04	1394
	8192	14.3	4	0.4	382.2	77.74	36.23	113.97	31.8	2605
	1024	7.2	5	0.2	948.4	37.38	515.95	553.33	93.25	955
	2048	14.3	5	0.2	1139.9	37.47	221.87	259.34	85.56	1752
	4096	14.3	5	0.2	1191.5	39	125.79	164.79	76.34	3127
	8192	14.3	5	0.2	1340.3	38.42	82.55	120.97	68.24	5591
	1024	7.2	5	0.4	1991.3	75.21	920.31	995.52	92.45	947
	2048	14.3	5	0.4	2081.9	75.37	486.93	562.3	86.6	1774
	4096	14.3	5	0.4	2475.1	78.81	260.88	339.69	76.81	3146
	8192	14.3	5	0.4	2568.2	77.84	168.31	246.15	68.38	5602


(a)


(b)

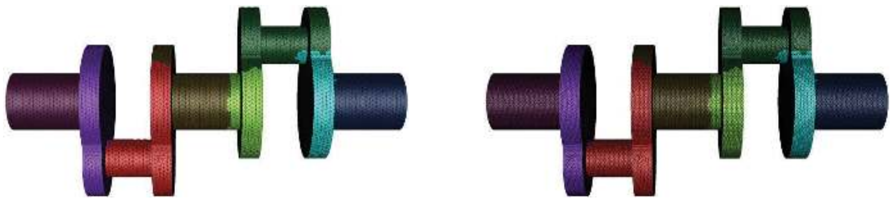

(c)

Figure 4: Test geometries and meshes used: onera-m6.stl (a), sphere.geo (b), and shaft.geo (c). The top picture is the coarse mesh and the bottom one is the (one-level refined) fine mesh.

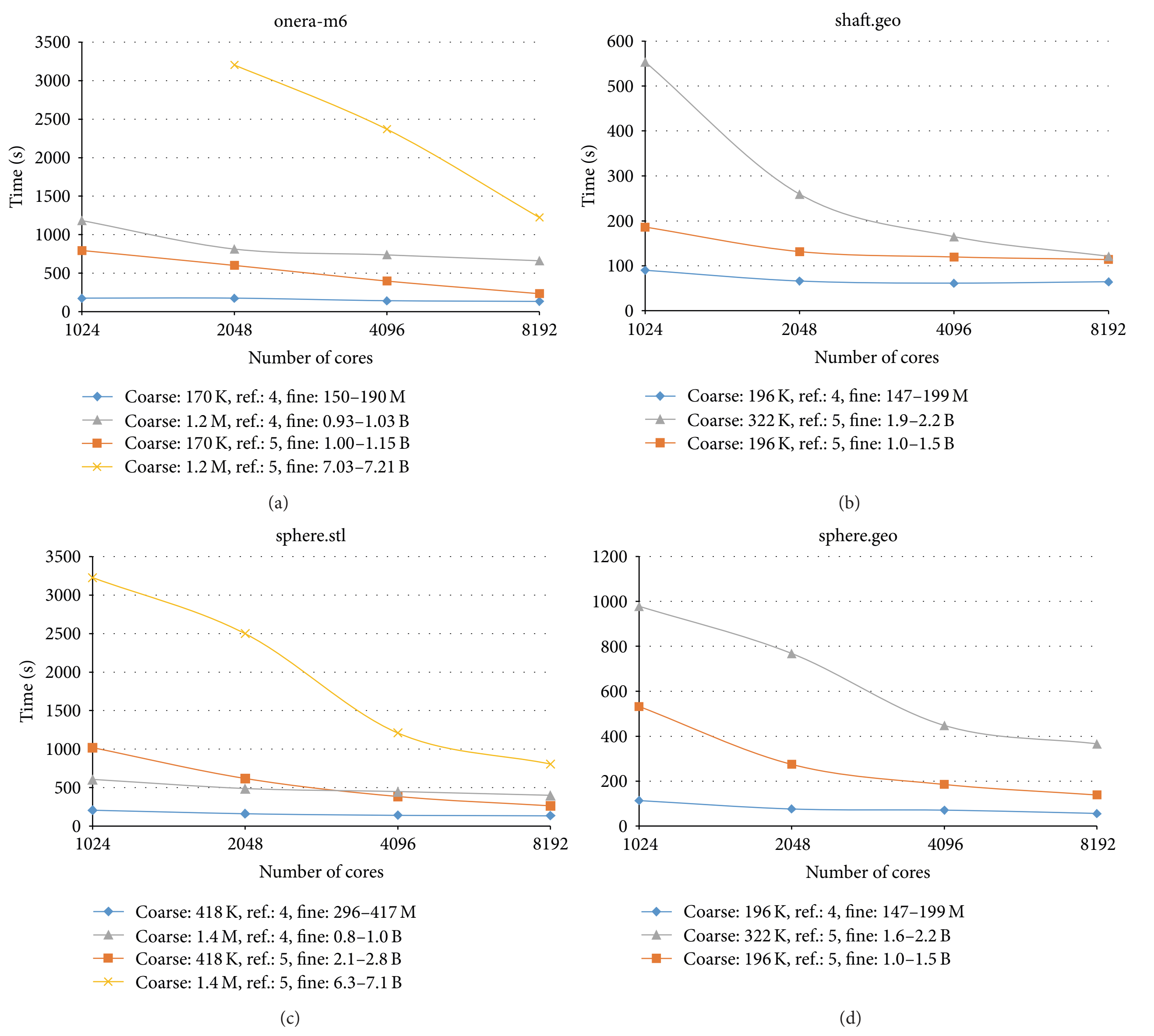


Figure 5: Parallel mesh generation timings; for each geometry timings obtained for ranges of ($|\mathcal{T}|, |\mathcal{T}''|$) pairs are drawn as separate curves.

When looking at *sequential* mesh generation times of the coarse mesh, we see that on average 0.23 million elements/minute rate is achieved by Netgen. In [3], Löhner states that *"typical speeds for the complete generation of a mesh (surface, mesh, improvement) on current Intel Xeon chips with 3.2 GHz and sufficient memory are of the order of 0.5–2.0 million elements/minute."* Netgen's meshing rate on a single 2.6 GHz core is lower than this, but still it is a reasonable rate.

When looking at the parallel *multibillion* element mesh generation results, we see that *within the 1K–8K core range* scalability is achieved. Figure 5 plots mesh generation time curves for ($|\mathcal{T}|, |\mathcal{T}''|$) pairs. $|\mathcal{T}|$ and $|\mathcal{T}''|$ are the sizes of initial coarse and the final fine mesh, respectively. The inputs to our program are the following parameters: (i) maximum element edge length in the mesh, (ii) mesh grading factor between 0.0 and 1.0, and (iii) number of levels of refinement. Parameters (i)-(ii) dictate the size of the coarse mesh and (i)–(iii) dictate the size of the final fine mesh. We can treat coarse and fine mesh sizes, that is, the pair ($|\mathcal{T}|, |\mathcal{T}''|$), as our problem size. Since, when changing the number of cores (partitions), the submesh sizes can change (since a new mesh is generated), we have defined ranges on ($|\mathcal{T}|, |\mathcal{T}''|$) and plotted mesh generation times for ranges of ($|\mathcal{T}|, |\mathcal{T}''|$). Hence, given a roughly fixed problem size, that is, a given range of the pair ($|\mathcal{T}|, |\mathcal{T}''|$), we see from Table 2 that execution times decrease as the number of cores is increased. Of course, as the ratio $|\mathcal{T}''|/P$ gets smaller and smaller, any further increase in the number of cores will lead to smaller and smaller reductions in execution time and will approach the time it takes to generate and partition the coarse mesh. The plot in Figure 6 shows the dependence of speedup on the number of processors and coarse mesh size to fine mesh size ratio.

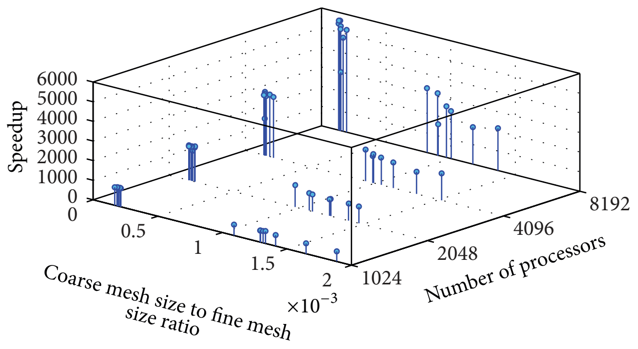


Figure 6: The dependence of speedup on number of processors and coarse mesh size to fine mesh size ratio.

Since we are not able to generate massive meshes sequentially (because they do not fit into the memory of a single node), we cannot calculate speedup and efficiency values based on actual sequential program execution. To get an idea about these, we estimated sequential running time as summation of the time taken to generate the coarse mesh plus the time taken to generate P fine meshes on one core; that is, $T_{\text{seq}} = \text{time}(\mathcal{T}) + P \cdot \text{time}(\mathcal{T}'')$. Using T_{seq}, we calculated the speedup as sequential time divided by total parallel time, that is, $T_{\text{seq}}/\text{Total}$, and efficiency as $T_{\text{seq}}/(P \cdot \text{Total})$. The speedup and efficiency columns are shown in Table 2.

6. Conclusions

We have developed a wrapper tool called PMSH around the existing sequential Netgen mesh generator that enabled us to generate multibillion element tetrahedral meshes for the OpenFOAM applications. The mesh generation runs on 1K through 8K cores with 14.3 TB total memory limit show our PMSH implementation scales. Seven billion element meshes that were generated on 8K cores for the Onera M6 wing and the sphere geometries took about 21 minutes and 14 minutes, respectively. With a sequential rate of 0.23 million elements/minute, such a mesh would need roughly 21 days to be generated sequentially. It is also interesting to note that by choosing smaller initial coarse meshes roughly billion element sized meshes can be generated in 2–4 minutes. PMSH tool code is maintained at https://code.google.com/p/pmsh/.

GUI (graphical user interface) version of Netgen offers some additional capabilities such as specification mesh sizes on specific regions of the geometry (by inputting a mesh size file). We have not been able to locate an API in the non-GUI Netgen library to do this. Therefore, currently this feature is not supported in our PMSH. When an API is made available in future versions of Netgen, this will be incorporated into PMSH. Netgen can generate prismatic boundary layer consisting of prismatic cells. This is again available in the GUI version of Netgen and hence not supported in PMSH. Finally, anisotropic meshing is not supported.

Our work described in this paper concentrated mainly on setting up distributed data structures and solving the entity matching problem in a scalable way without resorting to floating point number comparisons. We believe it is because of these issues (data structures and entity matching problems) that massive mesh generation results at tens of thousands of cores scale are absent in the literature. Our approach simplifies our programming and enables us to generate multibillion element meshes. Having solved these fundamental problems, we can now concentrate on the next stage. Our future work will focus on the following:

(i) Development of a malleable parallel mesh generation system;

(ii) Implementation of more advanced schemes for projecting vertices on geometric boundaries;

(iii) Support for specification of mesh sizes for specific regions;

(iv) Support for generation of boundary layer;

(v) Support for anisotropic meshing.

Abbreviations

P:	Number of processors (subdomains)
Ω:	Closed geometric domain to be meshed
$\partial\Omega$:	Boundary (surface) of geometry Ω
$\mathcal{T}$:	Tetrahedral mesh
$\mathcal{T}^f$:	Faces of tetrahedral mesh
$\mathcal{T}^v$:	Vertices of tetrahedral mesh
$\partial\mathcal{T}$:	Boundary of tetrahedral mesh
$\mathcal{T}_i$:	Tetrahedral submesh (partitioned mesh) on processor i
$\partial\mathcal{T}_i$:	Boundary of tetrahedral submesh (partitioned mesh) on processor i
$\mathcal{T}_i$:	Faces of tetrahedral submesh (partitioned mesh) on processor i
r:	Number of refinement levels
$[(i,\alpha),(j,\beta),(k,\gamma)]$:	Integer based global barycentric IDs where i, j, k are either 0 or indices of vertices from $\partial\mathcal{T}^v$ with $\alpha+\beta+\gamma = 2^r$ and $i < j < k$.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work was financially supported by the PRACE-3IP project funded in part by the EUs 7th Framework Programme (FP7/2012–2014) under Grant Agreement no. RI-312763. The authors thank Peter Stadelmeyer for initial discussions that motivated the development of this tool. They also acknowledge greatly Vilje supercomputer allocation provided to them by NTNU and the always prompt help they received from Bjorn Lindi on the use of Vilje system. Finally, they thank Mustafa Dindar for providing them with Onera M6 wing geometry in STL format.

References

[1] OpenFoam, http://www.openfoam.org/.

[2] J. Schöberl, "NETGEN An advancing front 2D/3D-mesh generator based on abstract rules," *Computing and Visualization in Science*, vol. 1, no. 1, pp. 41–52, 1997.

[3] R. Löhner, "Recent advances in parallel advancing front grid generation," *Archives of Computational Methods in Engineering*, vol. 21, no. 2, pp. 127–140, 2014.

[4] J. Dongarra, J. Hittinger, J. Bell et al., *Applied Mathematics Research for Exascale Computing: Report*, Mathematics Working Group, 2014.

[5] W. N. Dawes, S. A. Harvey, S. Fellows, N. Eccles, D. Jaeggi, and W. P. Kellar, "A practical demonstration of scalable, parallel mesh generation," in *Proceedings of the 47th AIAA Aerospace Sciences Meeting & Exhibit*, pp. 5–8, January 2009.

[6] C. D. Antonopoulos, F. Blagojevic, A. N. Chernikov, N. P. Chrisochoides, and D. S. Nikolopoulos, "A multigrain Delaunay mesh generation method for multicore SMT-based architectures," *Journal of Parallel and Distributed Computing*, vol. 69, no. 7, pp. 589–600, 2009.

[7] D3D mesh generator, http://mech.fsv.cvut.cz/~dr/d3d.html.

[8] MeshSim, Simmetrix inc, http://www.simmetrix.com.

[9] E. Ivanov, O. Gluchshenko, and H. Andrä, *Parallel Software Tool for Decomposing and Meshing of 3d Structures*, 2007.

[10] Pamgen Mesh Generation, 2014, http://trilinos.sandia.gov/packages/pamgen/.

[11] S. Pion, J. Singler, V. Batista, and D. Millman, "Parallel geometric algorithms for multi-core computers," Tech. Rep. RR-6749, 2008.

[12] Cgal 4.5.2—3d mesh generation, 2014, http://doc.cgal.org/latest/Mesh_3/index.html#title12.

[13] G. Houzeaux, R. de la Cruz, H. Owen, and M. Vázquez, "Parallel uniform mesh multiplication applied to a Navier–Stokes solver," *Computers & Fluids*, vol. 80, pp. 142–151, 2013.

[14] P. Kabelikova, A. Ronovsky, and V. Vondrak, "Parallel mesh multiplication for code saturne," PRACE Project FP7/2007-2013, 2011.

[15] A. Ronovsky, P. Kabelikova, V. Vondrak, and C. Moulinec, "Parallel mesh multiplication and its implementation in code saturne," in *Proceedings of the 3rd International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineers*, Civil-Comp Press, 2013.

[16] E. Yilmaz and S. Aliabadi, "Application of surface conformed linear mesh and data subdivision technique to a spinning projectile," *Computers & Fluids*, vol. 88, pp. 773–781, 2013.

[17] Y. Yilmaz, C. Ozturan, O. Tosun, A. H. Ozer, and S. Soner, "Parallel mesh generation, migration and partitioning for the elmer application," PRACE Project FP7/2007-2013, 2011.

[18] F. Piscaglia, A. Montorfano, and A. Onorati, "Development of fully automatic parallel algorithms for mesh handling in the openfoam-2.2.x technology," in *Proceedings of the International Multidimensional Engine Modeling Users Group Meeting*, 2013.

[19] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[20] T. Boubekeur and C. Schlick, "Generic mesh refinement on GPU," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pp. 99–104, ACM, 2005.

[21] T. Boubekeur and C. Schlick, "A flexible kernel for adaptive mesh refinement on GPU," *Computer Graphics Forum*, vol. 27, no. 1, pp. 102–113, 2008.