Scalable 3D Hybrid Parallel Delaunay Image-to-Mesh Conversion Algorithm for Distributed Shared Memory Architectures

Daming Feng, Christos Tsolakis, Andrey N. Chernikov, Nikos P. Chrisochoides*

Department of Computer Science, Old Dominion University, Norfolk, Virginia 23569, USA

Abstract

In this paper, we present a scalable three-dimensional hybrid parallel Delaunay image-to-mesh conversion algorithm (PDR.PODM) for distributed shared memory architectures. PDR.PODM is able to explore parallelism early in the mesh generation process because of the aggressive speculative approach employed by the Parallel Optimistic Delaunay Mesh generation algorithm (PODM). In addition, it decreases the communication overhead and improves data locality by making use of a data partitioning scheme offered by the Parallel Delaunay Refinement algorithm (PDR). PDR.PODM supports fully functional volume grading by creating elements with varying size. Small elements are created near boundary or inside the critical regions in order to capture the fine features while big elements are created in the rest of the mesh. We tested PDR.PODM on Blacklight, a distributed shared memory (DSM) machine in Pittsburgh Supercomputing Center. For the uniform mesh generation, we observed a weak scaling speedup of 163.8 and above for up to 256 cores as opposed to PODM whose weak scaling speedup is only 44.7 on 256 cores. The end result is that we can generate 18 million elements per second as opposed to 14 million per second in our earlier work. PDR.PODM scales well on uniform refinement cases running on DSM supercomputers.

*Corresponding author

Email addresses: dfeng@cs.odu.edu (Daming Feng), ctsolakis@cs.odu.edu (Christos Tsolakis), achernik@cs.odu.edu (Andrey N. Chernikov), nikos@cs.odu.edu (Nikos P. Chrisochoides)

The varying size version sharply reduces the number of elements compared to the uniform version and thus reduces the time to generate the mesh while keeping the same fidelity.

Keywords: Parallel Mesh Generation, Varying Size Mesh Generation, Scalability, Image-to-Mesh Conversion

---

1. Introduction

Parallel mesh generation methods decompose the original mesh generation problem into smaller subproblems which are solved (meshed) in parallel using multiple cores (processors). High scalability, quality and fidelity mesh generation is a critical module for the real world (bio-)engineering and medical applications. The quality of mesh refers to the quality of each element in the mesh which is usually measured in terms of its circumradius-to-shortest edge ratio (radius-edge ratio for short) and (dihedral) angle bound. Normally, an element is regarded as a good element when the radius-edge ratio is small [1, 2, 3, 4] and the angles are in a reasonable range [5, 6]. The fidelity is understood as how well the boundary of the created mesh represents the boundary (surface) of the real object. A mesh has good fidelity when its boundary is a correct topological and geometrical representation of the real surface of the object. The scalability can be measured in terms of the ability of an algorithm to achieve a speedup proportional to the number of cores. There is no doubt that the mesh generation algorithms will continue to be critical for many (bio-)engineering applications, such as CFD simulations [7, 8] and image discretization in bioinformatics [9]. In this paper, we present a parallel mesh generation algorithm which is able to deliver high scalability on distributed shared memory (DSM) non-uniform memory access (NUMA) supercomputers that satisfies all of these three important requirements.

The implementation of parallel mesh generation algorithms on supercomputers brings new challenges because of their special memory architecture. Most current mesh generation algorithms are desktop-based, either sequential or parallel, developed for a small number of cores. Such mesh generation algorithms, when run on supercomputers, are either conservative in leveraging available concurrency [10, 11] or depend on the solution of the domain decomposition problem which is still open for three dimensional domains [12, 13]. Implementing an efficient parallel mesh generation algorithm targeting the DSM NUMA architecture will contribute to the understanding

of the challenging characteristics of adaptive and irregular applications on supercomputers consisting of thousands or millions of cores. This will also help the community gain insight into the family of problems characterized by unstructured communication patterns.

The advantage of Delaunay mesh generation over other mesh generation methods is that it can mathematically guarantee the quality of the mesh and the termination of the algorithm [14, 15, 3]. In the previous work [16], our group implemented a parallel Image-to-Mesh (I2M) generation algorithm, the Parallel Optimistic Delaunay Mesh generator (PODM), which uses as input multi-label segmented three dimensional images and creates meshes with quality and fidelity guarantees. PODM introduces low level locking mechanisms, carefully designed contention managers and well-suited load balancing schemes that make it work well for a low core count (less than 128 cores). However, it exhibits considerable performance deterioration for a higher core count (144 cores or more) because of the intensive and multi-hop communication.

Parallel mesh generation algorithms based on the octree structure have exhibited scalability because of the low communication and computation overhead that they introduce. Our group presented an octree-based parallel mesh generation algorithm called Parallel Delaunay Refinement (PDR) [11, 17] that allows multiple point insertions independently, without any synchronization. PDR takes advantage of an octree structure and decomposes the iteration space by selecting independent subsets of points from the set of the candidate points without suffering from rollbacks. The data management and partition approach of PDR improves data locality and at the same time decreases the communication.

The PDR.PODM algorithm proposed in this paper takes advantage of these two legacy approaches. It quickly leverages high parallelism because of the aggressive speculative approach employed by PODM and uses data partitioning offered by PDR to improve data locality and decrease the communication overhead. Experiments performed on Blacklight, a cache-coherent NUMA shared memory machine in the Pittsburgh Supercomputing Center, show that PDR.PODM has a weak scaling speedup of 163.8 for 256 cores and creates 18.02 million tetrahedra every second with high quality. In addition, the surface of the object and the boundaries between materials are well represented. Figure 1 shows an example of a varying size Delaunay mesh created by PDR.PODM. The left figure demonstrates the fidelity of the mesh. The cut-through figure on the right shows the size gradation of the mesh.

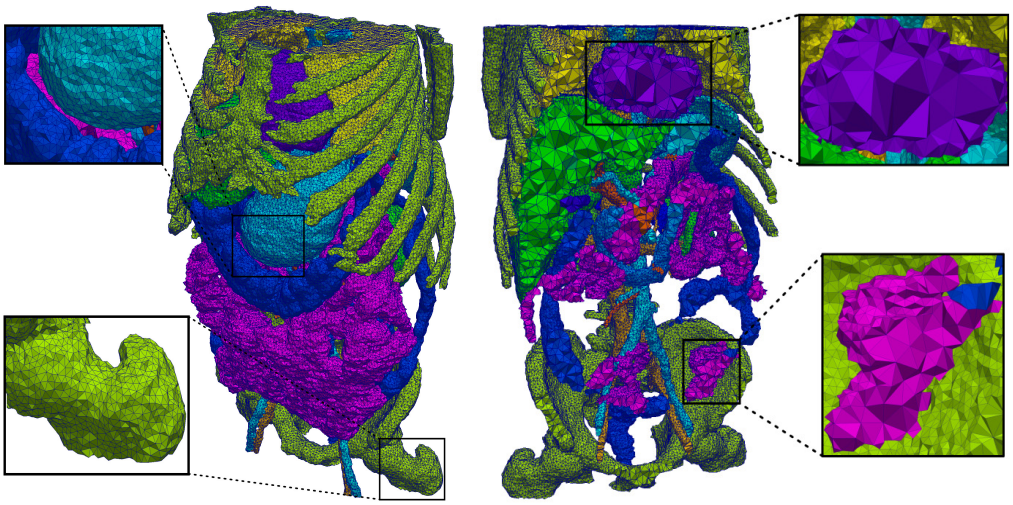


Figure 1: A varying size Delaunay mesh generated by PDR.PODM. The input image is the CT abdominal atlas obtained from IRCAD Laparoscopic Center [18]. The left figure demonstrates the fidelity of the mesh. The surface of the input object is well represented. The cut-through figure on the right shows size variation of the mesh.

In summary, the PDR.PODM method:

- guarantees the quality of the output mesh. The radius-edge ratio of each tetrahedron is smaller than 1.93 and the planar angles of all boundary facets are larger than 30 degrees [3];
- represents the surface of the input object with topological and geometrical guarantees;
- recovers the surface and meshes the volume simultaneously in parallel;
- supports parallel mesh generation with elements of varying size for multi-material objects.

The rest of the paper is organized as follows. Section 2 gives the background for parallel mesh generation algorithms and provides a review on prior work based on Delaunay, Advancing Front, and Octree methods. Section 3 presents the PODM and PDR parallel mesh generation algorithms in detail. Section 4 describes the implementation of the parallel Delaunay mesh generation algorithm PDR.PODM. Section 5 present the experimental results and analysis. Section 6 concludes the paper and outlines our future work.

2. Parallel Mesh Generation Background

Usually, a parallel mesh generation algorithm proceeds as follows:

- Construct an initial mesh (not necessary for some domain-decomposition algorithms);
- Decompose the domain or initial mesh into $N(N >= 2)$ subdomains or submeshes;
- Distribute the subdomains or submeshes to different cores of a multi-core machine, refine the mesh in parallel, and stop when the quality and other criteria are met.

Three of the most popular techniques for parallel mesh generation are Delaunay, Advancing Front and Octree-based. In the following literature review, we do not cover all the approaches of parallel mesh generation and parallel refinement. We list only those that are related to our work.

Delaunay mesh generation algorithms work by inserting additional (often called Steiner) points into an existing mesh to improve the quality of the elements. It is proven [1, 19] that the algorithm terminates after having eliminated all poor quality tetrahedra, and in addition, the termination does not depend on the order of processing the poor quality tetrahedra, even though the structure of the final meshes may vary. The insertion of a point is often implemented according to the well-known Bowyer-Watson kernel [20, 21, 22]. Parallel Delaunay mesh generation methods can be implemented by inserting multiple points simultaneously [10, 12, 11, 16], and the parallel insertion of points by multiple threads needs to be synchronized.

Blelloch et al. [23] solved the problem of creating a Delaunay triangulation for a specified point set in parallel. A divide-and-conquer projection-based algorithm is proposed to construct Delaunay triangulations of a predefined points set. One major limitation of triangulation algorithms [23, 24, 25] is that they only triangulate the convex hull of a given set of points and therefore they provide neither quality nor fidelity guarantees.

Alleaume et al. [26] described a parallel Delaunay mesh generation algorithm that can create large meshes. The method is applied in an out-of-core fashion due to the limited core-memory resources at that time. Okusanya and Peraire [27] proposed a parallel three dimensional unstructured Delaunay mesh generation algorithm that addresses the load balancing problem by distributing the bad elements among the processors using mesh migration.

However, the algorithm exhibits only 30% efficiency on 8 cores. Galtier and George [28] used smooth separators to prepartition the whole domain into subdomains and then distribute these subdomains among different processors for parallel refinement. The drawback of this method is that mesh generation needs to be restarted from the very beginning if the created separators are not Delaunay-admissible. Ivanov et al. [29] proposed a parallel mesh generation algorithm based on domain decomposition that can take advantage of the classic 2D and 3D Delaunay mesh generators for independent volume meshing. It achieves superlinear speedup but only on eight cores.

Chernikov and Chrisochoides [12] proposed the Parallel Constrained Delaunay Meshing (PCDM) algorithm. In PCDM, the edges on the boundaries of submeshes are fixed (constrained). This approach requires the construction of the separators that will not compromise the quality of the final mesh, which is still an open problem for three dimensional domains. The Parallel Delaunay Domain Decoupling PD^3 [13] method is based on the idea of decoupling the individual subdomains so that they can be meshed independently with zero communication and synchronization. A proper decomposition that can decouple the mesh is required to eliminate communication costs. However, the construction of such a decomposition is an equally challenging problem because its solution is based on the Medial Axis [30, 31] which is very expensive and difficult to construct (even to approximate) for complex three dimensional geometries.

Löhner [32] proposed a parallel advancing front grid generation algorithm. The domain-defining grid is partitioned and the elements inside each domain are generated in the first pass. Then, the elements located in inter-domain regions are created through several passes. The scalability is good when generating elements inside each domain but it degrades quickly for the subsequent passes. Ito et al. [33] described a parallel unstructured mesh generation algorithm based on the Advancing Front method. A coarse volume mesh is created and partitioned into a set of subdomains, and each subdomain is refined in parallel using the advancing front method. However, the overall performance of this algorithm is about 6% efficiency on 64 cores. A framework for parallel advancing front unstructured grid generation, targeting both shared memory and distributed memory architectures is proposed by Zagaris et al. [34]. The framework exploits the Master/Worker pattern for the parallel implementation in order to balance the workloads of each task. Because of the low parallelism of the divide and conquer tree, it achieves at best 55% efficiency on 60 cores. It should be mentioned that advancing front

methods cannot guarantee the termination.

Tu et al. [35] implemented a parallel meshing tool called *Octor* for generating and adapting octree meshes. Octor provides a data access interface that can interact with parallel PDE solvers efficiently. However, the fidelity of the meshes is not addressed which is a very important factor that determines the accuracy of the subsequent finite element solver. Burstedde et al. [36] extended the octree method and proposed a parallel adaptive mesh refinement algorithm on forest-of-octrees geometries. Although it exhibits excellent speedup for thousands of cores, the fidelity of the created mesh is not guaranteed. Dawes et al. [37] describe a parallel bottom-up octree mesh generation algorithm. It inverts the process of the top-down octree methods and generate the mesh from the bottom-up, from the finest cells up the tree to the coarser ones. In order to obtain a mesh with good quality, an optimization process is necessary for the exported mesh which takes almost one third of the total execution time.

3. Parallel Delaunay Refinement (PDR) and Parallel Optimistic Delaunay Mesh Generation (PODM)

PDR [17, 11, 10] is based on a theoretically proven method for managing and scheduling the insertion points. This approach is based on an analysis of the dependencies of the inserted points and the resulting Delaunay Independence Criterion: two points p_i and p_j are Delaunay independent with respect to a mesh if and only if (i) their cavites do not have shared elements, and (ii) in the case p_i and p_j have a shared facet on the boundary of their cavites, the sphere that passes through p_i (p_j) and the vertices of the shared facet does not include p_j (p_i). Based on the Delaunay Independence Criterion, PDR breaks the meshing problem of the entire region up into smaller independent subproblems, i.e., partitions the region into subregions in such a way that the circumcenters of the elements belonging to different subregions can be inserted concurrently. Using a carefully constructed octree, the list of the candidate points is split up into smaller lists that can be processed concurrently with the sequential Delaunay refinement code implementated in a Delaunay refinement software, e.g., TetGen [2]. PDR requires neither the runtime checks nor the geometry decomposition and it can guarantee the independence of inserted points and thus avoid the evaluation of data dependencies.

PODM [16] is a tightly-coupled parallel Delaunay mesh generation algorithm. The sequential construction of the initial mesh in PODM only involves the triangulation of the bounding box, i.e., the sequential creation of six tetrahedra. Immediately after the construction of these tetrahedra, the parallel mesh refinement procedure starts. The sequential overhead of constructing the initial mesh is negligible compared to refining millions or even billions of elements in the subsequent parallel procedure. Each thread in PODM maintains its own Poor Element List (PEL) [16] which contains the elements that violate the quality or fidelity criteria [3] and have been assigned to this thread for processing. A global load balancing list is used to spread the work among threads. Each thread has the flexibility to communicate with any other thread during the refinement. This approach works well on a medium number of cores. It scales well up to a relatively high core count compared to other tightly-coupled parallel mesh generation algorithms [38]. However, due to the extensive remote memory accesses, its scalability for Distributed Shared Memory machines is limited from 128 to 256 cores depending on the utilization of the target machine from other users.

In this paper, we combine both approaches in order to take advantage of their best features and propose the PDR.PODM algorithm.

4. Proposed Hybrid Algorithm

In this section, we present a hybrid parallel implementation targeting distributed shared memory architectures. Our parallel mesh generation algorithm proceeds in the following three main steps: (a) parallel initial mesh construction, (b) sequential lattice construction and initial mesh partition, (c) independent subregions (submeshes) scheduling and two-level parallel refinement.

Figure 2 is a high level description of PDR.PODM. A bounding box of the input image is created, the lattice structure is constructed and the buffer zones of each subregion are found and stored (lines 1 to 3). The construction and the distribution of the initial mesh is done in parallel by PODM running on multiple cores (lines 4 to 6). Lines 7 to 29 list the subsequent parallel refinement procedure after the construction of the initial mesh. All the subregions are pushed to a refinement queue Q. Each subregion in Q includes the bad elements that belong to the corresponding subregion. If Q is not empty, a PODM mesh generator that is running on a multi-core computing node gets the bad elements from one subregion to refine. Multiple PODM

PDR.PODM(I,$\bar{r_t}$,$\bar{r_I}$,N,C)

Input: I is the input segmented image;
$\bar{r_t}$ is the circumradius upper bound of the elements in the final mesh;
$\bar{r_I}$ is the circumradius upper bound of the elements in the initial mesh;
N is the number of computing nodes;
C is the number of cores in a computing node.

Output: A Delaunay Mesh $\mathcal{M}$ that conforms to the upper bound $\bar{r_t}$.

```
1:  Create the bounding box of I;
2:  Construct a lattice with subregion size reflecting the initial upper bound r̄_I;
3:  Find the buffer zones of each subregion of the lattice;
4:  Generate an initial mesh that conforms to r̄_I using PODM;
5:  Distribute the initial mesh to subregions based on their circumcenter coordinates;
6:  Push all subregions to a refinement queue Q;
7:  for each computing node in parallel
8:      Create a PODM mesh generator PMG with C threads;
9:      while Q ≠ ∅
10:         Pop one subregion L and its buffer zones B_1 and B_2 from Q;
11:         Get the bad elements in L and add them to the PEL of PMG;
12:         while PEL ≠ ∅ in parallel
13:             Get the first bad element e from PEL;
14:             Check the type of the bad element e;
15:             Refine e based on the refinement rules and create new elements;
16:             Check and classify new elements;
17:             for each newly created element e'
18:                 if e' is a bad element and its circumcenter is inside the current subregion
19:                     add e' to PEL for further refinement;
20:                 else
21:                     add e' to a neighbor subregion;
22:                 endif
23:             endfor
24:         endwhile
25:         for each neighbor subregion L_nei of L that contains bad elements
26:             Push L_nei back to the refinement queue Q;
27:         endfor
28:     endwhile
29: endfor
30: return M
```

Figure 2: A high level description of the PDR.PODM algorithm.

mesh generators that are running on different computing nodes can do the refinement work of different subregions simultaneously. PDR.PODM follows the refinement rules of PODM to create the volume mesh and recover the isosurface. As shown in lines 15 to 23, after creating a new element e', we check whether e' is a bad element or not, and if it is, which refinement rule it violates. Then based on the coordinates of its circumcenter, we add the element either to the current PEL or to the PEL of a neighbor subregion for further refinement. Figure 2 lists only the main steps of PDR.PODM. The actual implementation is more elaborate to support efficient data structures and parallel processing.

4.1. Parallel Initial Mesh Construction

The construction of an initial mesh is the starting point for the subsequent parallel procedure. PDR uses the sequential TetGen [2] algorithm to create the initial mesh, which increases the sequential overhead of the whole parallel algorithm. In order to reduce the sequential overhead, we use the PODM mesh generator to create the initial mesh in parallel. There are two important parameters that will affect the performance of the whole algorithm when we create the initial mesh. The first one is the number of cores that we use to create the initial mesh. Based on the performance of PODM as illustrated in Table 2 and Figure 5a, this value should be neither too small nor too large. If the number of cores is too small, it is not enough to explore the available concurrency. If it is too large, the communication overhead among them is high. Both of these cases make the construction of the initial mesh time-consuming and deteriorate the performance of PDR.PODM. In practice, we found that 64 cores is the optimal value for creating the initial mesh when running PDR.PODM on Blacklight. The second parameter is the circumradius upper bound $\bar{r_I}$ that we use to control the volume upper bound of created elements. This number determines the number of elements of the initial mesh. The larger $\bar{r_I}$ is, the larger the volume of the created tetrahedra are and thus less elements created because the volume of the input object is fixed. If $\bar{r_I}$ is too small, the meshing time of the initial mesh is too long because a large number of elements are created; if it is too large, there is not enough concurrency for the subsequent refinement procedure because there are not enough elements in the intial mesh. In our experiments, we use $\bar{r_I} = 4\bar{r_t}$, where $\bar{r_t}$ represents the target radius upper bound for the final mesh, since it gives the best performance for PDR.PODM among the different values of $\bar{r_I}$ we have tried so far.

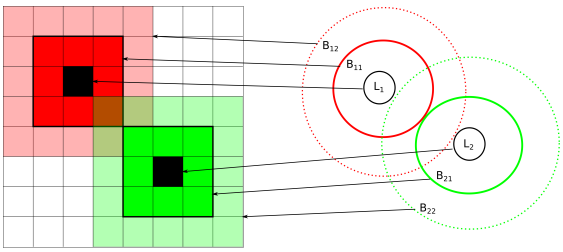


Figure 3: A two dimensional illustration of three dimensional buffer zones. The Venn diagram on the right part demonstrates the logical relations between two subregions and their first and second buffer zones. The left part shows an example with two subregions (the two black subregions), L_1 and L_2, which can be refined independently and simultaneously without synchronization during the PDR.PODM mesh refinement procedure. The dark green and dark red regions around L_1 and L_2 form the first level buffer zones, B_{11} and B_{21}. The light green and light red regions represent the second level buffer zones, B_{12} and B_{22}. The conflict between two PODM mesh generators working on different subregions is eliminated during the refinement.

4.2. Initial Mesh Decomposition and Distribution

We used a simple but efficient way to divide the whole input image into subregions, which consists in partitioning the bounding box into cubes. Then, we assign tetrahedra to different subregions based on the coordinates of their circumcenters. Consider a subregion L_1, the twenty six neighbor subregions form its first level buffer zone B_{11}(the dark red region shown in Figure 3). When subregion L_1 is under refinement, all subregions in the first level buffer zone B_{11} can not be refined by another PODM mesh generator simultaneously. During the refinement procedure, the point insertion operation might propagate to one subregion of its first level buffer zone. Consider a case where L_1 and L_2 are refined simultaneously. If B_{11} and B_{21} are not disjoint, this may result into a nonconforming mesh across B_{11} and B_{21}. Therefore, we use a second level of buffer zones, B_{12} and B_{22} (light red and light green in Figure 3) in order to ensure that B_{11} and B_{21} are disjoint. In our implementation, if one subregion is popped up from the refinement queue during the refinement, all its first and second level buffer neighbors are also popped up. This guarantees that two subregions that are refined simultaneously are at least two layers (subregions) away from each other and thus the aforementioned problems are eliminated.

4.3. Two-level Parallel Mesh Refinement

In distributed shared memory (DSM) systems, memory is physically distributed while it is accessible to and shared by all cores. However, a memory block is physically located at various distances from the cores. As a result, the memory access time varies and depends on the distance of a core from a memory block. Based on a benchmark of the system group of the Pittsburgh Supercomputing Center [39], as shown in Table 1, the memory latency inside one blade (*Computing Node*) is $O(200)$ cycles and it increases when the number of network switches increases. Each extra switch adds about $O(1,500)$ cycles latency penalty.

Table 1: Memory hierarchy and the approximate memory access time (clock cycles) of Blacklight

Level	Memory Module	Size	Access Clock Cycles
1	L1 Cache	32KB per core	4
2	L2 Cache	256-512KB per core	11
3	L3 Cache	1-3 MB per core	40
4	DRAM to a blade	128GB	$O(200)$
5	DRAM to other blades	128GB and more	$O(1500)$

PDR.PODM explores two levels of parallelism: coarse-grain parallelism at the subregion level (which is mapped to a virtual *Computing Node*) and medium-grain parallelism at the cavity level (which is mapped to a single core). A *Computing Node* is a virtual computing unit consisting of a group of cores. A multi-threaded PODM mesh generator is mapped to a computing node. Each PODM thread runs on one core of that computing node. In the implementation, we consider the sixteen cores that are in the same blade as a computing node since they share 128GB local memory on the experimental platform. In the coarse-grain parallel level, the whole region (the bounding box of the input image) is decomposed into subregions and the bad elements of the initial mesh are distributed into different subregions based on the coordinates of their circumcenters. Then, a subset of independent subregions is selected and scheduled to be refined simultaneously. The selection and scheduling of subregions is based on the two level buffer zones presented in subection 4.2, i.e., if one subregion is selected to be refined, all subregions that are in its first and second level buffer zones can not be selected simultaneously to avoid resorting to rollbacks. In the medium-grain parallel level, the threads

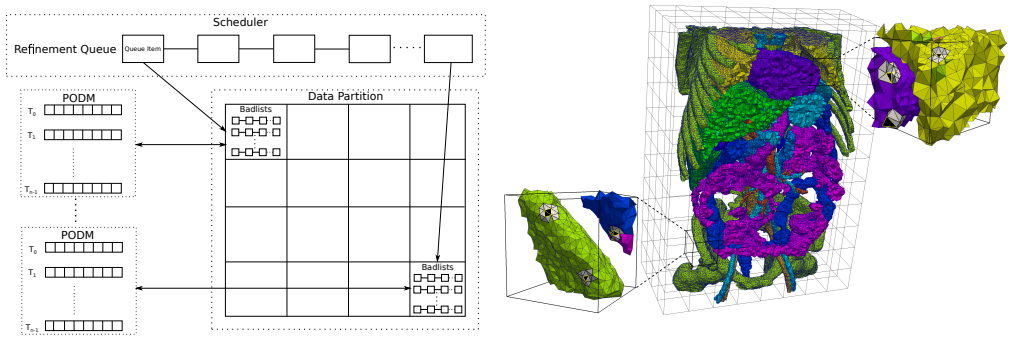


Figure 4: (a) A diagram that illustrates the design of PDR.PODM parallel Delaunay mesh generation algorithm. (b) Two-level parallelism illustration. The selected subregions are refined simultaneously and multiple cavities are expanded concurrently within a single subregion.

running on the cores of a computing node follow the refinement rules of PODM in order to refine the bad elements of each subregion in parallel. The load balance among the cores of each computing node is performed by the load balancing scheme of the PODM mesh generator. The experimental results have shown that over-decomposition [40] of the whole region is a good approach to solve the load-balancing issue among the coarse-grain level computing nodes.

Figure 4a shows a diagram of the PDR.PODM parallel mesh generation implementation design. The boxes that are marked *PODM* represent parallel Delaunay mesh generators. The block *Data Partition* represents the partition of the whole region (the bounding box of the input image). The block *Scheduler* represents the management and distribution of PODM mesh generators on different subregions. *Refinement Queue* is a refinement queue that stores all the subregions. Each *Queue Item* stores a pointer to one subregion of the lattice structure. Figure 4b shows an instance of the two-level parallel mesh refinement. The two subregions are selected to be refined simultaneously, and inside each region multiple points are inserted concurrently.

4.4. Parallel Varying Size Mesh Generation

PDR.PODM is able to create meshes of varying size in parallel based on the application requirements. It divides the elements (tetrahedra) into two

types base on their location: near-boundary tetrahedra and interior tetrahedra. A tetrahedron is regarded as a near-boundary tetrahedron if its circumsphere intersects one (or multiple) of the material boundaries. An interior tetrahedron is the one whose circumcenter lies inside one material. Small elements are generated near the boundaries in order to capture the fine features of the boundaries while in the rest of the region big elements are created to reduce the number of elements generated in the final mesh as shown in Figure 6b. In addition, the users can customize the size function for different regions and pass it as an input parameter. The size function sets an upper bound to the circumradius of the tetrahedra created in different regions and provides the flexibility to generate a dense mesh. PDR.PODM can create a dense uniform mesh inside critical regions while keeping a varying size mesh outside the regions as illustrated in Figure 6c.

5. Experimental Results and Analysis

In this section, we evaluate the performance of PDR.PODM on distributed shared memory architecture. We performed a set of experiments creating uniform meshes to access the weak scaling and strong scaling performance of PDR.PODM. When measuring the weak scaling performance of an algorithm, the problem size (the number of elements created in the case of PDR.PODM) increases proportionally with respect to the number of cores. When measuring the strong scaling performance, the problem size remains the same for all number of cores. We tested both the weak scaling performance of our implementation and PODM on Blacklight using up to 256 cores. Then, another set of experiments were performed to test the strong scaling performance of PDR.PODM when creating uniform mesh. Finally, a set of experiments was conducted to test the performance of PDR.PODM when creating varying size meshes. In all these experiments, the execution time reported includes pre-processing time for loading the image, lattice data structure creation time and the actual mesh refinement time.

5.1. Experiment Setup

The input images we used in our experiment are the CT abdominal atlas from IRCAD Laparoscopic Center [18] and BigBrain [41]. We also show several uniform and varying size meshes using a 3D MRI with brain tumor. Our experimental platform is Blacklight [39], the cache-coherent NUMA shared memory machine in the Pittsburgh Supercomputing Center. Blacklight is a

cc-NUMA shared-memory system consisting of 256 blades. Each blade holds 2 Intel Xeon X7560 (Nehalem) eight-core CPUs, for a total of 4096 cores across the whole machine. The 16 cores on each blade share 128 Gbytes of local memory. One individual rack unit (IRU) consists of 16 blades and 256 cores. A 16-port NL5 router is used to connect blades located internally to each IRU. Each of these routers connects to eight blades within the IRU. The remaining eight ports of the internal router are used to connect to other NL5 router blades [42]. The total 4096 cores have 32 TB memory.

5.2. Weak Scaling Performance of Uniform Meshing

In this subsection, we present the weak scaling performance of PDR.PODM. We also show the weak scaling performance of PODM for comparison. We increase the problem size, i.e., the number of tetrahedra, linearly with respect to the number of cores. The number of tetrahedra created is controlled by the parameter $\bar{r}_t$. This parameter sets a circumradius upper bound on the tetrahedra created. A decrease (increase) of the parameter $\bar{r}_t$ by a factor of m, results in an approximate m^3 times increase (decrease) of the number of tetrahedra created. The number of tetrahedra created increases gradually from 3 million to 745 million when the number of cores increases from 1 to 256. Table 2 shows the weak scaling performance of PODM and PDR.PODM.

5.2.1. Evaluation Metrics

The quality of an element e (tetrahedron or triangle) is measured by its *radius-edge ratio*. Let $r(e)$ and $l(e)$ denote the circumradius and the shortest edge of e respectively. The radius-edge ratio of e is defined as $\rho_e = \frac{|r(e)|}{|l(e)|}$. The radius-edge ratio of each element in the output mesh generated by PDR.PODM is smaller than 1.93 because it utilizes the same refinement rules as PODM [3, 16].

We use the following metrics to evaluate the scalability of parallel mesh generation algorithms [43, 44].

Speedup S: The ratio of the sequential execution time of the fastest known sequential algorithm (T_s) to the execution time of the parallel algorithm (T_p).

Efficiency E: The ratio of speedup (S) to the number of cores (p): $E = S/p = T_s/(pT_p)$.

In the weak scaling case, the number of elements per core (we use one thread per core) remains approximately constant. In other words, the problem size (i.e., the number of elements created) is increased proportionally to

the number of cores. The number of elements generated is approximately equal to 3 million on a single Blacklight core. The problem size increases proportionally from 3 million to 745 million tetrahedra for 1 to 256 cores on Blacklight. In practice, it is difficult to control the problem size exactly while the number of cores is increased to p because of the irregular nature of the unstructured mesh. So we use an alternative definition of speedup which is more precise for a parallel mesh generation algorithm.

We measure the number of elements generated every second during the experiment. Let us denote by $elements(p)$ and $time(p)$ the number of generated tetrahedra and the meshing time respectively, where p is the number of cores. Then we can use the following formula to compute the speedup:

$$S(p) = \frac{elements\_per\_sec(p)}{elements\_per\_sec(1)} = \frac{elements(p) \cdot time(1)}{time(p) \cdot elements(1)} \tag{1}$$

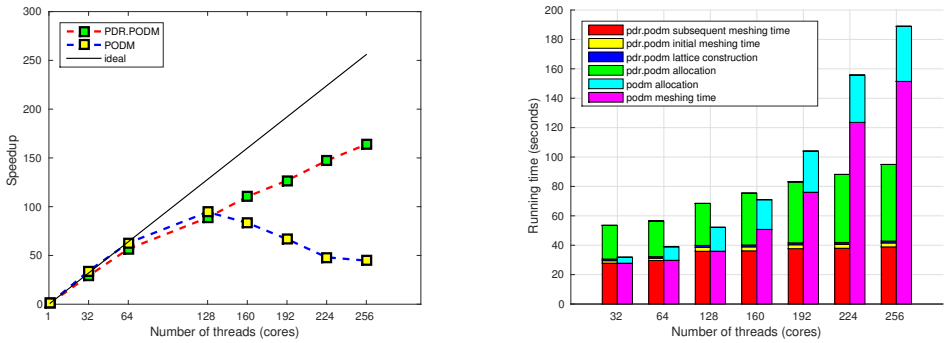
In equation (1), $elements\_per\_sec(p)$ represents the number of elements created per second using p cores while $elements\_per\_sec(1)$ represents the number of elements (tetrahedra) created per second by the best sequential mesh generation algorithm.

5.2.2. Scalability Analysis

Table 2 demonstrates that PODM shows outstanding performance when the number of cores is less than or equal to 64. The speedup using 32 cores and 64 cores is 34.1 and 63.8 respectively, which means that the speedup increases linearly with respect to the number of cores. On 128 cores, PODM achieves a speedup of 94.5 and efficiency of 73.86%. However, the performance of PODM deteriorates when the number of cores is more than 128 on Blacklight. We ran a set of bootstrapping experiments from 128 cores to 256 cores with an increase of two blades (32 cores) each time, to test the performance deterioration of PODM. Each time we increase the number of

Table 2: Performance of PODM & PDR.PODM. The input is the abdominal atlas.

Cores	Elements (millions)	Meshing Time (seconds)		Elements/s (million)		Speedup		Efficiency %	
		podm	pdr.podm	podm	pdr.podm	podm	pdr.podm	podm	pdr.podm
1	3.0	27.18	27.78	0.11	0.11	1.0	1.0	100.00	100.00
32	95.1	26.07	29.74	3.75	3.19	34.1	29.0	106.56	90.71
64	187.3	27.02	30.14	6.91	6.23	63.8	56.6	98.12	88.53
128	375.1	35.93	38.54	10.42	9.76	94.5	88.7	73.86	69.32
160	466.4	50.76	38.86	9.18	12.13	83.5	110.3	52.15	68.92
192	560.3	76.04	40.31	7.35	13.91	66.8	126.5	34.80	66.05
224	657.2	123.57	40.57	5.29	16.19	48.1	147.2	21.47	65.71
256	745.7	151.44	41.53	4.92	18.02	44.7	163.8	17.47	63.99



(a) (b)

Figure 5: (a) Weak scaling speedup of PODM and PDR.PODM on 32 to 256 cores on Blacklight. Three million tetrahedra are created by each thread running on a core. The black line depicts the ideal linear speedup. The red dash line with green markers shows the speedup of PDR.PODM and the blue one with yellow markers is the speedup of PODM. (b) Running time of PDR.PODM and running time of PODM.

cores, the speedup decreases. For example, the speedup on 160 cores is 83.5 which is lower than that of 128 cores and it decreases to only 44.7 for 256 cores. The reason for this performance deterioration of PODM is the increase of communication time due to the large number of remote memory accesses and the congested network. The blue dashed line with yellow markers in Figure 5a shows clearly this performance deterioration of PODM when core count is above 128.

PDR.PODM exhibits better scalability potential when the number of cores is higher than 128 as shown in Table 2. We ran the same set of bootstrapping experiments from 128 cores to 256 cores with a step of two blades (32 cores) each time in order to compare the performance with PODM. We observed that each time we increase the number of cores, the speedup of PDR.PODM increases while the speedup of PODM decreases. For example, the speedup on 128 cores is only 88.7 and it increases to 163.8 for 256 cores. The reason of this performance enhancement is the data partition that PDR offers. As we described before, we partition the whole region into subregions and also we divide all the available cores into groups (computing nodes). Therefore, the communication among different computing nodes is eliminated during the refinement procedure and the runtime checks during the cavity expansion in each subregion involves only a small number of cores.

When the number of cores is less than 128, the performance of PDR.PODM

is lower than that of PODM. As illustrated in Table 2, PODM using 32 and 64 cores creates 3.75 million and 6.91 million elements per second and the speedup is linear with respect to the number of cores while PDR.PODM creates 3.19 million and 6.23 million elements per second respectively and the speedup is only 29.0 and 56.6 respectively. The lower speedup of PDR.PODM is caused by the overhead we introduce to check and distribute newly created elements to the corresponding subregions for further refinement. When the number of cores is small (< 128), the overhead we introduce is more than the overhead that we want to reduce because of remote memory accesses and rollbacks.

Figure 5b depicts the execution time of PODM and PDR.PODM. The execution time of PODM consists of the allocation time for the initialization of the threads and the meshing time. The execution time of PDR.PODM includes the allocation time, the lattice construction time, the initial mesh creation time and the subsequent mesh refinement time. We can see clearly in Figure 5b that the total meshing time of PDR.PODM, i.e., the meshing time to create the initial mesh (the yellow block of the left bar) plus the meshing time in the subsequent refinement procedure (the red block of the left bar), is greater, although by a small amount, than the meshing time of PODM (the light purple bar on the right) when the number of cores is lower than or equal to 128. However, this drawback of PDR.PODM can be overcome easily. What we need to do is to set a threshold on the number of cores. When the number of cores is lower than this threshold, we deactivate the lattice structure and all the related data decomposition and scheduling procedures. Only when the number of cores is higher than this threshold and PODM does not perform well, the PDR.PODM mode is activated to take advantage of its scalability potential.

Table 3: Weak Scaling Performance of PDR.PODM. The input is the BigBrain.

Cores	Elements (million)	Meshing Time (second)	Elements/s (million)	Speedup	Efficiency (%)
1	4.1	41.27	0.10	1.0	100.00
16	66.0	45.88	14.39	14.4	89.95
32	131.9	48.56	27.17	27.2	84.91
64	264.2	50.15	52.68	52.7	82.31
128	526.1	58.83	89.43	89.4	69.86
160	656.7	60.88	107.87	107.9	67.42
192	787.3	62.79	125.39	125.4	65.31
224	917.0	64.09	143.09	143.1	63.88
256	1046.4	66.47	157.43	157.4	61.50

Table 4: Strong Scaling Performance of PDR.PODM. The input is the abdominal atlas.

Cores	Elements (million)	Meshing Time (second)	Elements/s (million)	Speedup	Efficiency (%)
1	376.2	3412.27	0.11	1.0	100.00
16	376.2	236.83	15.88	14.4	90.23
32	376.2	122.26	30.77	27.9	87.41
64	376.2	63.28	59.44	54.0	84.44
128	376.2	38.65	97.32	88.5	69.12
160	376.2	34.64	108.59	98.7	61.70
192	376.2	29.86	125.98	114.5	59.65
224	376.2	26.86	140.06	127.3	56.84
256	376.2	24.92	150.97	137.3	53.61

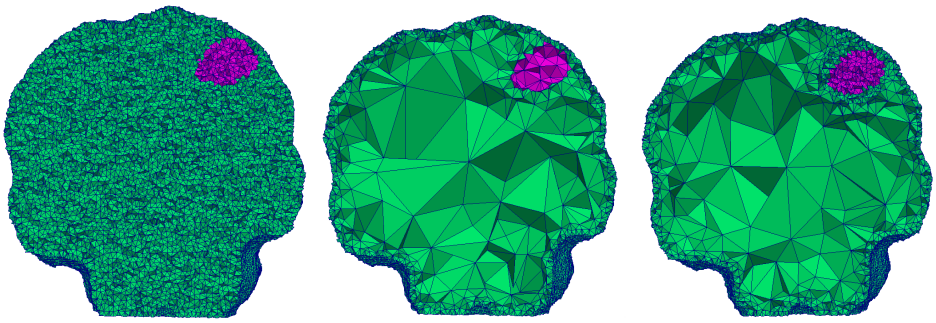
Table 3 shows the weak scaling performance of PDR.PODM for the input BigBrain [41]. Again, we can see similar performance trends for this input.

Table 4 shows the strong scaling performance of PDR.PODM for input image abdominal atlas. In the strong scaling case, the number of elements remains the same. In the experiments, the number of elements is about 376.2 million for all runs from 1 to 256 cores. As demonstrated in Table 4, when the number of cores is large (> 128), the strong scaling speedup is a little lower than the weak scaling speedup shown in Table 2. The underlying reason is that the ratio of useful computation to overhead decreases as the number of cores is increased.

5.3. Comparision: Uniform Meshing and Varying Size Meshing

Figure 6 shows examples of a uniform mesh and two varying size meshes created by PDR.PODM for the brain tumor image. Figure 6a demonstrates a uniform volume mesh. The size upper bound of element is the same (uniform) for both materials. Figure 6b shows a varying size mesh. Small elements are created near the boundaries in order to capture the fine features of the boundaries while in the regions far away from the boundaries the elements are larger. Figure 6c gives an illustration of another varying size mesh that contains a critical region around the tumor specified by the user. A dense uniform mesh is created inside the critical region around the tumor while a varying size mesh is maintained outside.

We also ran a set of experiments from 16 to 256 cores that generated varying size meshes for the multi-material abdominal image. Table 5 shows the experimental results of PDR.PODM creating uniform meshes and varying size meshes. Compared to the uniform meshes, the corresponding varying size meshes have much fewer elements. As a result, the meshing time re-



(a) (b) (c)

Figure 6: Uniform and varying size meshes created by PDR.PODM for the input brain tumor image (two materials). (a) A uniform volume mesh for both materials. (b) A varying size mesh with the same geometric fidelity is created that small elements created near the boundaries. (c) Another varying size mesh with a dense mesh created inside the user-specified region around the tumor.

quired to create the varying size meshes is less than the meshing time for the uniform mesh with the same fidelity. The largest uniform mesh in the experiment contains 745.7 million elements and the meshing time is 41.53 seconds for 256 cores. PDR.PODM created a corresponding varying mesh with the same fidelity in only 5.91 seconds which is almost seven time faster than the uniform meshing. Furthermore, since the number of elements is lower than the uniform mesh, the finite element solver will perform faster using the varying size mesh compared to using the uniform mesh.

Table 5: Comparison of Uniform Meshing and Varying Size Meshing. The input is the abdominal atlas.

Cores	Elements (million)		Meshing Time (second)		Elements/s (million)		Speedup	
	uniform	varying	uniform	varying	uniform	varying	uniform	varying
16	47.2	2.5	27.78	3.60	1.69	0.69	15.4	6.3
32	95.1	5.0	29.74	4.92	3.19	1.02	29.0	9.2
64	187.3	6.5	30.14	5.97	6.23	1.09	56.6	9.9
128	375.1	9.7	38.54	5.62	9.76	1.73	88.7	15.7
160	466.4	12.5	38.86	5.85	12.13	2.14	110.3	19.4
192	560.3	16.1	40.31	5.85	13.91	2.75	126.5	25.0
224	657.2	19.4	40.57	5.99	16.19	3.24	147.2	29.4
256	745.7	23.5	41.53	5.91	18.02	3.98	163.8	36.2

Note: The fidelity of the uniform and the corresponding varying size mesh is the same for the same number of cores.

6. Conclusions and Future Work

In this paper, we present a three dimensional parallel mesh generation algorithm, PDR.PODM, which delivers high scalability on DSM NUMA supercomputers. Taking advantage of the best features of two algorithms we proposed before, PDR.PODM quickly leverages parallelism because of the aggressive speculative approach employed by PODM, and uses data partitioning offered by PDR to improve data locality and decrease the communication overhead. PDR.PODM can create varying size meshes in parallel, i.e., small elements are created near the boundaries or in the critical regions to capture the fine features while larger elements are generated for the rest regions to keep the total number of elements low.

In our current implementation we use the thread model (BoostC++ thread). In our future work, we plan to explore the scalability of PDR.PODM further by running more experiments to evaluate the cache hit ratio for different levels of cache. It should be mentioned that the idea of this paper is also suitable for distributed memory architectures since the communication among computing nodes is low during the parallel refinement procedure. Therefore, one of our future work directions is to extend the idea of this paper to distributed memory architectures. We plan to employ the MPI programming model for the coarse grain parallelism and utilize the MPI+threads mixed programming to explore the idea of PDR.PODM on distributed memory machines.

Acknowledgements

This work in part is funded by NSF grants CCF-1439079 and CC-NIE 1440673, NASA grant NO. NNX15AU39A and DoDs PETTT Special Project PP-CFD-KY07-007. In addition, it utilized resources from the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575. We thank the systems group in the Pittsburgh Supercomputing Center for their great help and support. We especially thank Panagiotis Foteinos and David O'Neal for many insightful discussions. We also thank all the reviewers for their comments. Finally, it is partially supported by the Richard T. Cheng Endowment. The content is solely the responsibility of the authors and does not necessarily represent the official views of the NSF.

References

[1] J. R. Shewchuk, Tetrahedral mesh generation by Delaunay refinement, in: Proceedings of the 14th ACM Symposium on Computational Geometry, 1998, pp. 86–95.

[2] H. Si, Tetgen: A quality tetrahedral mesh generator and a 3D Delaunay triangulator, `http://wias-berlin.de/software/tetgen/` (2013).

[3] P. Foteinos, A. Chernikov, N. Chrisochoides, Guaranteed quality tetrahedral Delaunay meshing for medical images, Computational Geometry: Theory and Applications 47 (4) (2014) 539–562.

[4] CGAL, computational geometry algorithms library, `http://www.cgal.org` (2014).

[5] X. Liang, Y. Zhang, An octree-based dual contouring method for triangular and tetrahedral mesh generation with guaranteed angle range, Engineering with Computers 30 (2) (2014) 211–222.

[6] J. Bronson, J. Levine, R. Whitaker, Lattice cleaving: A multimaterial tetrahedral meshing algorithm with guarantees, Visualization and Computer Graphics, IEEE Transactions on 20 (2) (2014) 223–237.

[7] R. Zhang, K. P. Lam, S. chune Yao, Y. Zhang, Coupled energyplus and computational fluid dynamics simulation for natural ventilation, Building and Environment 68 (0) (2013) 100 – 113. doi:http://dx.doi.org/10.1016/j.buildenv.2013.04.002.

[8] O. C. Zienkiewicz, R. L. Taylor, P. Nithiarasu, The Finite Element Method for Fluid Dynamics, seventh Edition, Butterworth-Heinemann, 2013.

[9] J. Xu, A. Chernikov, Curvilinear Triangular Discretization of Biomedical Images with Smooth Boundaries, in: International Symposium on Bioinformatics Research and Applications, Springer, Norfolk, VA, 2015, to appear.

[10] A. Chernikov, N. Chrisochoides, Three-dimensional Delaunay refinement for multi-core processors, ACM International Conference on Supercomputing (2008) 214–224.

[11] A. Chernikov, N. Chrisochoides, Parallel guaranteed quality Delaunay uniform mesh refinement, SIAM Journal on Scientific Computing 28 (2006) 1907–1926.

[12] A. Chernikov, N. Chrisochoides, Parallel 2D constrained Delaunay mesh generation, ACM Transactions on Mathematical Software 34 (2008) 6–25.

[13] L. Linardakis, N. Chrisochoides, Delaunay decoupling method for parallel guaranteed quality planar mesh refinement, SIAM Journal on Scientific Computing 27 (4) (2006) 1394–1423.

[14] S.-W. Cheng, T. K. Dey, J. Shewchuk, Delaunay Mesh Generation, CRC Press, 2012.

[15] P.-L. George, H. Borouchaki, Delaunay Triangulation and Meshing. Application to Finite Elements, HERMES, 1998.

[16] P. Foteinos, N. Chrisochoides, High quality real-time image-to-mesh conversion for finite element simulations, Journal on Parallel and Distributed Computing 74 (2) (2014) 2123–2140.

[17] A. Chernikov, N. Chrisochoides, Practical and efficient point insertion scheduling method for parallel guaranteed quality Delaunay refinement, in: ACM International Conference on Supercomputing, 2004, pp. 48–57.

[18] Ircad Laparoscopic Center, `http://www.ircad.fr/softwares/3Dircadb` (2013).

[19] L. P. Chew, Guaranteed-quality Delaunay meshing in 3D, in: Proceedings of the 13th ACM Symposium on Computational Geometry, 1997, pp. 391–393.

[20] D. F. Watson, Computing the n-dimensional Delaunay tesselation with application to Voronoi polytopes, Computer Journal 24 (1981) 167–172.

[21] A. Bowyer, Computing Dirichlet tesselations, Computer Journal 24 (1981) 162–166.

[22] N. Chrisochoides, D. Nave, Parallel Delaunay mesh generation kernel, International Journal for Numerical Methods in Engineering 58 (2003) 161–176.

[23] G. E. Blelloch, G. L. Miller, J. C. Hardwick, D. Talmor, Design and implementation of a practical parallel Delaunay algorithm, Algorithmica 24 (3) (1999) 243–269.

[24] D. K. Blandford, G. E. Blelloch, C. Kadow, Engineering a compact parallel Delaunay algorithm in 3D, in: Proceedings of the 22nd Symposium on Computational Geometry, SCG '06, ACM, New York, NY, USA, 2006, pp. 292–300.

[25] V. H. Batista, D. L. Millman, S. Pion, J. Singler, Parallel geometric algorithms for multi-core computers, Computational Geometry 43 (8) (2010) 663–677.

[26] A. Alleaume, L. Francez, M. Loriot, N. Maman, Large out-of-core tetrahedral meshing, in: Proceedings of the 16th International Meshing Roundtable, IMR 2007, October 14-17, 2007, Seattle, WA, USA, 2007, pp. 461–476.

[27] T. Okusanya, J. Peraire, 3D parallel unstructured mesh generation, in: S. A. Canann, S. Saigal (Eds.), Trends in Unstructured Mesh Generation, 1997, pp. 109–116.

[28] J. Galtier, P.-L. George, Prepartitioning as a way to mesh subdomains in parallel, in: Proceedings of the 5th International Meshing Roundtable, Pittsburgh, PA, 1996, pp. 107–121.

[29] E. Ivanov, O. Gluchshenko, H. Andrae, A. Kudryavtsev, Automatic parallel generation of tetrahedral grids by using a domain decomposition approach, Journal of Computational Mathematics and Mathematical Physics 8.

[30] C. Armstrong, D. Robinson, R. McKeag, T. Li, S. Bridgett, R. Donaghy, C. MCGleenan, Medials for meshing and more, in: 4th International Meshing Roundtable, 1995, pp. 277–288.

[31] H. N. Gursoy, N. M. Patrikalakis, An automatic coarse and fine surface mesh generation scheme based on medial axis transform: Part i algorithms, Engineering With Computers 8 (1992) 121–137.

[32] R. Löhner, A 2nd generation parallel advancing front grid generator, in: Proceedings of the 21st International Meshing Roundtable, IMR 2012, October 7-10, 2012, San Jose, CA, USA, 2012, pp. 457–474.

[33] Y. Ito, A. Shih, A. Erukala, B. Soni, A. Chernikov, N. Chrisochoides, K. Nakahashi, Parallel mesh generation using an advancing front method, Mathematics and Computers in Simulation 75 (2007) 200–209.

[34] G. Zagaris, S. Pirzadeh, N. Chrisochoides, A framework for parallel unstructured grid generation for practical aerodynamic simulations, in: 47th AIAA Aerospace Sciences Meeting, Orlando, FL, 2009.

[35] T. Tu, D. R. O'Hallaron, O. Ghattas, Scalable parallel octree meshing for terascale applications, in: Proceedings of the 2005 ACM/IEEE conference on Supercomputing, IEEE Computer Society, Seattle, WA, 2005. doi:http://dx.doi.org/10.1109/SC.2005.61.

[36] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, L. Wilcox, Extreme-scale amr, in: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, 2010, pp. 1–12.

[37] W. Dawes, S. Harvey, S. Fellows, N. Eccles, D. Jaeggi, W. Kellar, A practical demonstration of scalable, parallel mesh generation, in: 47^{th} AIAA Aerospace Sciences Meeting and Exhibit, Orlando, FL, USA, 2009.

[38] D. Nave, N. Chrisochoides, L. P. Chew, Guaranteed-Quality Parallel Delaunay Refinement for Restricted Polyhedral Domains, Computational Geometry: Theory and Applications 28 (2004) 191–215.

[39] Blacklight, a large hardware-coherent shared memory resource, `http://gw55.quarry.iu.teragrid.org/mediawiki/images/0/04` (2010).

[40] A. Chernikov, N. Chrisochoides, K. Barker, Parallel programming environment for mesh generation, in: International Conference on Numerical Grid Generation in Computational Field Simulations, no. 8, Honolulu, HI, 2002, pp. 805–814.

[41] K. Amunts, C. Lepage, L. Borgeat, H. Mohlberg, T. Dickscheid, M.-. Rousseau, S. Bludau, P.-L. Bazin, L. B. Lewis, A.-M. Oros-Peusquens, N. J. Shah, T. Lippert, K. Zilles, A. C. Evans, Bigbrain: An ultrahigh-resolution 3d human brain model, Science 340 (6139) (2013) 1472–1475. doi:10.1126/science.1235381.

[42] SGI Altix UV 1000 System Users Guide, Report (2011).

[43] J. L. Gustanfson, G. R. Montry, R. E. Benner, Development of parallel methods for a 1024-processor hypercube, SIAM Journal on Scientific and Statistical Computing 9 (4) (1988) 609–638.

[44] A. Grama, A. Gupta, G. Karypis, V. Kumar, Introduction to Parallel Computing, Addison Wesley, 2003.