2015

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

Daming Feng
Old Dominion University

Christos Tsolakis
Old Dominion University

Andrey N. Chernikov
Old Dominion University

Nikos P. Chrisochoides
Old Dominion University

Repository Citation

Feng, Daming; Tsolakis, Christos; Chernikov, Andrey N.; and Chrisochoides, Nikos P., "Scalable 3D Hybrid Parallel Delaunay Image-to-Mesh Conversion Algorithm for Distributed Shared Memory Architectures" (2015). *Computer Science Faculty Publications*. 77.
https://digitalcommons.odu.edu/computerscience_fac_pubs/77

Original Publication Citation

Feng, D., Tsolakis, C., Chernikov, A. N., & Chrisochoides, N. P. (2015). Scalable 3D hybrid parallel Delaunay image-to-mesh conversion algorithm for distributed shared memory architectures. *Procedia Engineering, 124*(Supplement C), 18-30. doi:10.1016/j.proeng.2015.10.119

ELSEVIER

Available online at www.sciencedirect.com

ScienceDirect

Procedia Engineering 124 (2015) 18 – 30

Procedia Engineering

www.elsevier.com/locate/procedia

24th International Meshing Roundtable (IMR24)

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

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

[a]*Computer Science Department, Old Dominion University, Norfolk, VA 23508, 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 utilizes an octree structure to decompose the initial mesh and to distribute the bad elements to different octree leaves (subregions). A set of independent subregions are selected and refined in parallel without any synchronization among them. In each subregion, a group of threads is assigned to insert or delete multiple points based on the refinement rules offered by PODM. We tested PDR.PODM on Blacklight, a distributed shared memory (DSM) machine in the Pittsburgh Supercomputing Center, and 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. To the best of our knowledge, PDR.PODM exhibits the best scalability among parallel guaranteed quality Delaunay mesh generation algorithms running on DSM supercomputers.

Keywords: Parallel Mesh Generation, Delaunay Mesh Refinement, 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 [1–4] and the angles are in a reasonable range [5–9]. 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

*Corresponding author. Tel.: +0-000-000-0000 ; fax: +0-000-000-0000.
E-mail address: {dfeng, ctsolakis, achernik, nikos}@cs.odu.edu

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 [10,11] and image discretization in bioinformatics [12]. 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 [13,14] or depend on the solution of the domain decomposition problem which is still open for three dimensional domains [15,16]. Implementing an efficient parallel refinement 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 refinement over other mesh refinement methods is that it can mathematically guarantee the quality of the mesh and the termination of the algorithm [3,17–19]. In the previous work [20,21], our group implemented a parallel Image-to-Mesh (I2M) refinement 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. Our group presented an octree-based parallel mesh generation algorithm called Parallel Delaunay Refinement (PDR) [14,22] 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 decreases communication at the same time.

PDR is conservative in leveraging available concurrency due to the guarantees that require a sufficiently refined mesh before the parallel refinement can be safely started but it decreases communication and improves data locality because of the data partition. PODM, on the other hand, is a tightly coupled meshing approach and it is more aggressive in leveraging parallelism at the cost of run-time checks for data dependencies but the scalability is limited for high core count because of the communication overhead. In this paper, we combine both approaches in order to take advantage of their best features and propose the PDR.PODM algorithm. 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 tissues are well represented. Fig. 1 shows an example of Delaunay mesh created by PDR.PODM. The left figure demonstrates the fidelity of the mesh. The cut-through figure on the right shows the quality 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.
- exhibits the best scalability among three dimensional isosurface based parallel Delaunay mesh generation algorithms.

The rest of the paper is organized as follows. Section 2 presents the background for parallel mesh refinement algorithms and provides a review on prior work based on Delaunay, Advancing Front, and Octree methods. We present the PODM and PDR parallel mesh generation algorithms in detail. Section 3 describes the implementation

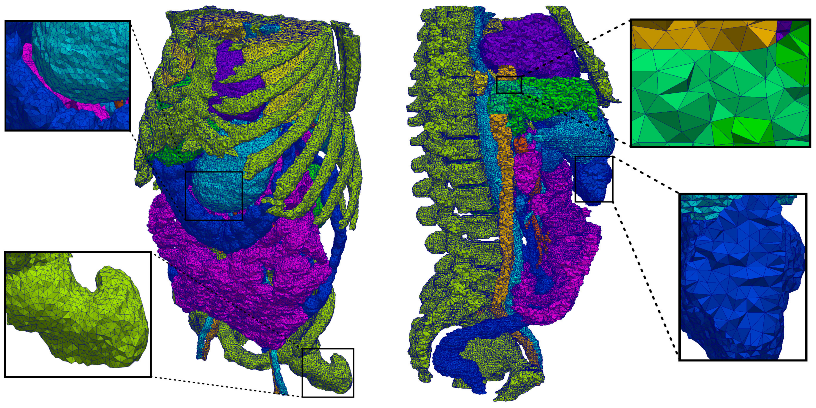


Fig. 1: Delaunay mesh generated by PDR.PODM. The input image is the CT abdominal atlas obtained from IRCAD Laparoscopic Center. The left figure demonstrates the fidelity of the mesh. The surface of the input object is well represented and the zoom-in parts illustrate that the boundaries of all tissues are well recovered. The cut-through figure on the right shows the quality of the mesh.

of our parallel Delaunay mesh generation algorithm, PDR.PODM. Section 4 contains our experimental results and analysis. Section 5 concludes the paper and outlines our future work.

2. Background and Related Work

Three of the most popular techniques for parallel mesh generation are Delaunay, Advancing Front and Octree. 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.

Delaunay refinement algorithms work by inserting additional (often called Steiner) points into an existing mesh to improve the quality of the elements. The basic operation of Delaunay refinement is the insertion and deletion of points, which then leads to the removal of poor quality tetrahedra and of their adjacent tetrahedra from the mesh and the creation of new tetrahedra. The new tetrahedra may or may not be of poor quality, and therefore may or may not require further point insertions. It is proven [1,23] that the algorithm terminates after having eliminated all poor quality tetrahedra, and in addition, the termination does not depend on the order of processing of 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 [24–26]. Parallel Delaunay mesh generation methods can be implemented by inserting multiple points simultaneously [13–15,20,21,27], and the parallel insertion of points by multiple threads needs to be synchronized.

The problem of creating a Delaunay triangulation of a specified point set in parallel has been solved by Blelloch et al. [28]. They describe a divide-and-conquer projection-based algorithm for constructing Delaunay triangulations of pre-defined point sets. Foteinos and Chrisochoides [27] proposed a parallel triangulation algorithm that supports not only point insertions but also point removals that offer new and rich refinement schemes. One major limitation of triangulation algorithms [27–30] is that they only triangulate the convex hull of a given set of points and therefore they guarantee neither quality nor fidelity requirements.

Okusanya and Peraire [31] proposed a parallel three dimensional unstructured Delaunay mesh generation algorithm that allows to distribute bad elements among processors by mesh migration to address the load balancing problem. However, the algorithm exhibits only 30% efficiency on 8 cores. Galtier and George [32] used smooth separators to prepartition the whole domain into subdomains and then to distribute these subdomains to different processors for parallel refinement. The drawback of this method is that mesh generation needs to be restarted form the very beginning if the created separators are not Delaunay-admissible. Ivanov et al. [33] 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.

Linardakis and Chrisochoides [16] presented a two dimensional Parallel Delaunay Domain Decoupling (PD^3) method. The PD^3 method is based on the idea of decoupling the individual subdomains so that they can be meshed independently with zero communication and synchronization by reusing the existing state-of-the-art sequential mesh generation codes. In order to eliminate the communication and synchronization costs, a proper decomposition that can decouple the mesh is required. However, the construction of such a decomposition is an equally challenging problem because its solution is based on the Medial Axis [34,35] which is very expensive and difficult to construct (even to approximate) for complex three dimensional geometries.

The idea of updating partition boundaries when inserted points happen to be close to them, was presented [36] and extended [15] as the Parallel Constrained Delaunay Meshing (PCDM) algorithm. In PCDM, the edges on the boundaries of submeshes are fixed (constrained). If a point which is considered for insertion encroaches upon a constrained edge, then instead of inserting this point another point is inserted in the middle of the edge. As a result, a split message is sent to the core that is responsible for refining the neighboring subdomain, notifying that it also has to insert the midpoint of the shared edge. 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.

Ito et al. [37] describe 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. [38]. 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. [39] 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. [40] 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. [41] 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.

Parallel Delaunay Refinement (PDR) [14,22] is based on a theoretically proven method for managing and scheduling the insertion points. This approach is based on the analysis of the dependencies between the inserted points: if two bad elements are far enough from each other, the Steiner points can be inserted independently. 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. The work has been extended to three dimensions [13]. Using a carefully constructed spatial decomposition tree, the list of the candidate points is split up into smaller lists that can be processed concurrently. The construction of an initial mesh is the basis and starting point for the subsequent parallel procedure. There is a trade-off between the available concurrency and the sequential overhead: the initial mesh is required to be sufficiently dense to guarantee enough concurrency for the subsequent parallel refinement step; however, the construction of such a dense mesh prolongs the low-concurrency part of the computation.

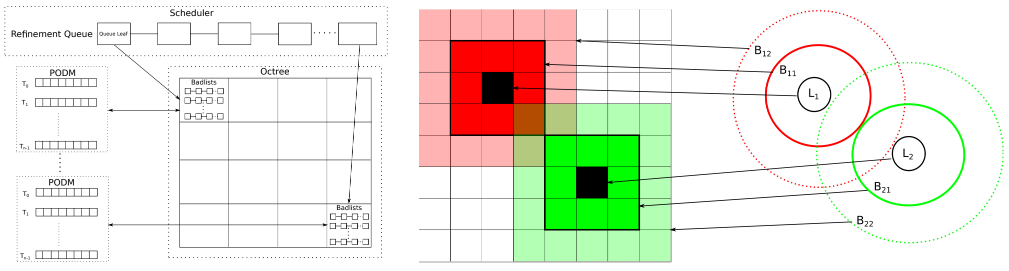
Parallel Optimistic Delaunay Mesh Refinement (PODM) [20,21] is a tightly-coupled parallel Delaunay mesh generation algorithm. The sequential construction of the initial mesh in PODM only involves the triangulation of the bounding box, i.e., the sequential creation of six tetrahedra. Immediately after 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*) [21] 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, and therefore to reduce the idle time of each thread. When a thread runs out of work, it asks for work from another thread. Each thread has the flexibility to communicate with any other thread during the refinement. This approach works well on a medium number of cores and exhibits impressive scalability. It scales well up to a relatively high core count compared to other tightly-coupled parallel mesh generation algorithms [42]. However, when core count is beyond a certain number, 128 on Blacklight for example, the communication overhead becomes the bottleneck that hinders the performance of PODM because it exerts too much pressure on the network routers. Its performance deteriorates for a core count beyond 144 because of the network congestion caused by the communication among a large number of cores.

3. Proposed Hybrid Algorithm

In this section, we present a hybrid parallel implementation targeting distributed shared memory architectures. 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. 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 our implementation, we consider the sixteen cores that are in the same blade as a computing node since they share 128GB local memory on our experimental platform. In the coarse-grain parallel level, we decompose the whole region (the bounding box of the input image) into subregions and distribute the bad elements of the initial mesh into subregions based on the coordinates of their circumcenters. Then, a subset of independent subregions is selected to be refined simultaneously without resorting to rollbacks. In the medium-grain parallel level, the threads 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. Our experimental results have shown that the over-decomposition of the whole region is a good approach to solve the load-balancing issue among the coarse-grain level computing nodes.

Fig. 2a shows a diagram of PDR.PODM parallel mesh generation implementation design. The boxes that are marked *PODM* represent parallel Delaunay mesh generators. The block *Octree* 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 (Octree leaves). *Refinement Queue* is a refinement queue that stores all the octree leaves. Each *Queue Leaf* stores a pointer pointing to one leaf of the octree structure.



(a) (b)

Fig. 2: (a) A diagram that illustrates the design of PDR.PODM parallel Delaunay mesh generation algorithm. (b) 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 leaves), 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.

3.1. 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 1 and Fig. 4a, 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 two 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 that are created in 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 is 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.

3.2. Octree Construction

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}(dark red region shown in Fig. 2b). 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 in Fig. 2b) 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.

3.3. Algorithm

Fig. 3 is a high level description of PDR.PODM. A bounding box of the input image is created, the octree structure is constructed and the buffer zones of each octree leaf are found and stored (lines 1 to 3). Each octree leaf represents a subregion of the bounding box of the input image. 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 octree leaves are pushed to a refinement queue Q. Each leaf 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 leaf to refine. Multiple PODM mesh generators that are running on different computing nodes can do the refinement work of different leaves simultaneously. PDR.PODM follows the refinement rules of PODM to create the volume mesh and recover the isosurface. As shown in lines 15 to 22, 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 we add it to the current PEL or a neighbor subregion for further refinement based on the coordinates of its circumcenter. Fig. 3 lists only the main steps of PDR.PODM. The actual implementation is more elaborate to support efficient data structures and parallel processing.

```
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 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 an octree with leaf size reflecting the initial upper bound $\bar{r}_I$;
 3: Find the buffer zones of each leaf of the octree;
 4: Generate an initial mesh that conforms to $\bar{r}_I$ using PODM;
 5: Distribute the elements of the initial mesh to octree leaves based on their circumcenter coordinates;
 6: Push all octree leaves to a refinement queue $Q$;
 7: for each computing node in parallel
 8:     Create a PODM mesh generator $PMG$ with $C$ threads;
 9:     while $Q \neq \emptyset$
10:         Pop one octree leaf $L$ 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 $PEL \neq \emptyset$ 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 leaf
19:                     add $e'$ to $PEL$ for further refinement;
20:                 else
21:                     add $e'$ to a neighbor leaf;
22:                 endif
23:             endfor
24:         endwhile
25:         if $L$ contains bad elements
26:             Push $L$ back to the refinement queue $Q$;
27:         endif
28:     endwhile
29: endfor
30: return $\mathcal{M}$
```

Fig. 3: A high level description of the PDR.PODM algorithm.

4. Experimental Results and Analysis

In this section, we evaluate the performance of PDR.PODM. A set of experiments were performed on a distributed shared memory architecture supercomputer to assess the weak scaling performance of PDR.PODM, i.e., the problem size (the number of elements created) increases proportionally with respect to the number of cores. We tested both our implementation and PODM on Blacklight using up to 256 cores. In all these experiments, the execution time reported includes pre-processing time for loading the image, octree data structure creation time and the actual mesh refinement time. The time we report does not contain the time for writing the final mesh to disk, since in most applications a parallel finite element solver, which is executed immediately in the subsequent step, will use the created mesh directly.

4.1. Experiment Setup

The input image we used in our experiment, the CT abdominal atlas, was obtained from IRCAD Laparoscopic Center [43]. Our experimental platform is Blacklight [44], 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 compute blades within the IRU. The remaining eight ports of the internal router are used to connect to other NL5 router blades [45]. The total 4096 cores have 32 TB memory.

4.2. 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 [3,21].

We use the following metrics to evaluate the scalability of parallel mesh generation algorithms [46,47].

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 equals approximately 3 million on a single Blacklight core. The problem size proportionally increases 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. Since PODM maintains the best single-core performance compared to other sequential three dimensional mesh generation software, such as Tetgen [2] and CGAL [4], $elements\_per\_sec(1)$ is defined as the number of elements generated per second by single-core PODM.

4.3. Weak Scaling Performance

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 gradually increases from 3 million to 745 million when the number of cores increases from 1 to 256. Table 1 shows the weak scaling performance of PODM and PDR.PODM.

Table 1 demonstrates that PODM shows outstanding performance when the number of cores is less 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 cores, the speedup decreases. For example, the speedup of 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 dash line with yellow markers in Fig. 4a shows clearly this performance deterioration of PODM when core count is above 128.

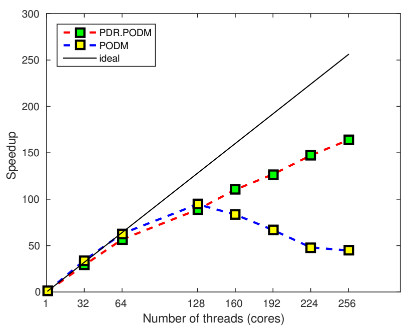
PDR.PODM exhibits better scalability potential when the number of cores is higher than 128 as shown in Table 1. We ran the same set of bootstrapping experiments from 128 cores to 256 cores with a step of two blades (32 cores) each time in order to compare the performance with PODM. We observed that each 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 compared to PODM is because of 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 1, 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 56.6 and 88.7 respectively. The lower speedup of PDR.PODM is caused by the overhead we introduce to check and distribute newly created elements to the corresponding octree leaves 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.

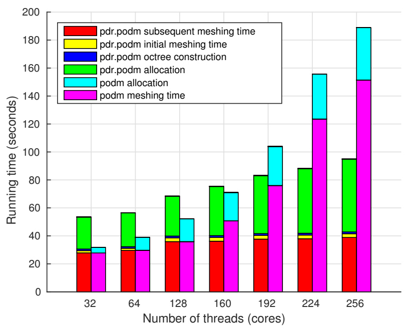
Fig. 4b depicts the execution time of PODM and PDR.PODM. The execution time of PODM consists of the allocation time for the initialization of the threads and the meshing time. The execution time of PDR.PODM includes the allocation time, the octree construction time, the initial mesh creation time and the subsequent mesh refinement time after the creation of the initial mesh. We can see clearly in Fig. 4b 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

Table 1: Weak Scaling Perfomance of PODM & PDR.PODM

Cores	Elements	Meshing Time (seconds)		Elements per second		Speedup		Efficiency %	
	(millions)	podm	pdr.podm	podm	pdr.podm	podm	pdr.podm	podm	pdr.podm
1	3	27.18	27.78	0.11	0.11	1.0	1.0	100.00	100.00
32	97	26.07	29.74	3.75	3.19	34.1	29.0	106.56	90.71
64	187	27.02	30.14	6.91	6.23	63.8	56.6	98.12	88.53
128	375	35.93	38.54	10.42	9.76	94.5	88.7	73.86	69.32
160	466	50.76	38.86	9.18	12.13	83.5	110.3	52.15	68.92
192	560	76.04	40.31	7.35	13.91	66.8	126.5	34.80	66.05
224	657	123.57	40.57	5.29	16.19	48.1	147.2	21.47	65.71
256	745	151.44	41.53	4.92	18.02	44.7	163.8	17.47	63.99



(a)



(b)

Fig. 4: (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.

Table 2: Percentage of different memory access hops of PDR.PODM for 16 to 256 cores

Hops\Cores	16	32	64	128	160	192	224	256
0	100.00%	99.94%	99.95%	98.85%	99.05%	99.20%	99.30%	99.38%
1	0.00%	0.06%	0.02%	0.18%	0.10%	0.08%	0.07%	0.06%
2	0.00%	0.00%	0.02%	0.49%	0.44%	0.37%	0.33%	0.28%
3	0.00%	0.00%	0.02%	0.48%	0.42%	0.35%	0.30%	0.27%
4	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

Table 3: 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	$O(1500)$

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 octree structure and all the related data decomposition and scheduling procedures and PDR.PODM works exactly as PODM. 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.

In order to evaluate the benefits of data partitioning, we calculated the latency of remote memory access in terms of hops. Roughly speaking, the number of hops is a measure of the distance between the core that requests data and the memory module where the data is stored. If the refinement process happens in the memory of one blade, the number of hops is zero since the sixteen cores of one blade share 128GB of local memory on Blacklight. However, if a thread running on a core in one blade requests data located in the memory of another blade, the number of hops increases according to the topology of the memory architecture. It should be noted here that the memory access latency is not the same for different number of hops. Based on a performance benchmark of the system group of the Pittsburgh Supercomputing Center [44] as shown in Table 3, the memory latency in one blade is $O(200)$ cycles and it increases when the number of hops is greater than zero. Each extra hop adds about $O(1,500)$ cycles latency penalty. In Table 2, we can see that 99% of memory accesses involve zero hops. This means that when PDR.PODM runs on Blacklight almost all memory accesses are local. The reason is that the refinement operations involve only the cores of the same blade during the refinement procedure. The rest of the hops happen because of the necessary data migration and distribution between the octree and the computing nodes as shown in lines 11 and 21 in Fig. 3.

Another advantage of PDR.PODM is that it increases the cache hit ratio. In our implementation, we divide the cores into different computing nodes and the sixteen cores of the same blade are in the same computing node. The runtime checks during the cavity expansion in each subregion involve only the cores of the computing node. During the refinement procedure, a newly created element *e* is stored in the local memory. If another thread running on another core of the same computing node runs out of work, it can fetch *e* directly from the cache and refine it. The access latency of cache is approximate 4 to 40 clock cycles as shown in Table 3, which is much less than local memory access. We will run more experiments to evaluate the cache hit ratio in the work to support the theoretically analysis. Since most of the memory accesses are local and the cache hit ratio also increases by the data partitioning scheme, therefore PDR.PODM demonstrates higher scalability than that of PODM.

5. Conclusion and Future Work

In this paper, we present a parallel mesh generation algorithm, PDR.PODM, which delivers high scalability on DSM NUMA supercomputers and creates meshes with quality and fidelity guarantees. PDR.PODM integrates two previous parallel mesh generation algorithms, PODM and PDR. By integrating these two algorithms, 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.

In our current implementation we use the thread model (Pthread and BoostC++ thread). In our future work, we plan to explore the scalability of PDR.PODM further and evaluate its performance with larger number of cores on Blacklight and other shared memory supercomputers. Also, we will run more experiments to evaluate the cache hit ratio for different level 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 combination of MPI and BoostC++ threads to explore the idea of PDR.PODM on distributed memory machines.

Acknowledgements

This work in part is funded by NSF grants CCF-1439079 and CCF-1139864. 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 (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 (2014) 211–222.

[6] A. N. Chernikov, N. P. Chrisochoides, Multitissue tetrahedral image-to-mesh conversion with guaranteed quality and fidelity, SIAM Journal on Scientific Computing 33 (2011) 3491–3508.

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

[8] A. Fedorov, N. Chrisochoides, R. Kikinis, S. K. Warfield, Tetrahedral mesh generation for medical imaging, in: 8th International Conference on Medical Image Computing and Computer Assisted Intervention (MICCAI 2005), 2005.

[9] Y. Liu, P. Foteinos, A. Chernikov, N. Chrisochoides, Mesh deformation-based multi-tissue mesh generation for brain images, Engineering with Computers 28 (2012) 305–318.

[10] R. Zhang, K. P. Lam, S. chune Yao, Y. Zhang, Coupled energyplus and computational fluid dynamics simulation for natural ventilation, Building and Environment 68 (2013) 100 – 113.

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

[12] 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.

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

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

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

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

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

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

[19] A. Chernikov, N. Chrisochoides, Generalized insertion region guides for delaunay mesh refinement, SIAM Journal on Scientific Computing 34 (2012) A1333–A1350.

[20] P. Foteinos, N. Chrisochoides, High quality real-time image-to-mesh conversion for finite element simulations, in: ACM International Conference on Supercomputing, ACM, 2013, pp. 233–242.

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

[22] 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.

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

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

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

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

[27] P. Foteinos, N. Chrisochoides, Dynamic parallel 3D Delaunay triangulation, in: International Meshing Roundtable, 2011, pp. 9–26.

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

[29] 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.

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

[31] 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.

[32] 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.

[33] 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 (2008).

[34] 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.

[35] 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.

[36] L. P. Chew, N. Chrisochoides, F. Sukup, Parallel constrained Delaunay meshing, in: ASME/ASCE/SES Summer Meeting,Special Symposium on Trends in Unstructured Mesh Generation, 1997, pp. 89–96.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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

[42] 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.

[43] Ircad laparoscopic center, `http://www.ircad.fr/softwares/3Dircadb/3Dircadb2`, 2013.

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

[45] SGI Altix UV 1000 System Users Guide, Report, 2011.

[46] 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 (1988) 609–638.

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