A Hybrid Parallel Delaunay Image-to-Mesh Conversion Algorithm Scalable on Distributed-Memory Clusters

Daming Feng, 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 parallel Delaunay image-to-mesh conversion algorithm. A nested master-worker communication model is used to simultaneously explore process- and thread-level parallelization. The mesh generation includes two stages: coarse and fine meshing. First, a coarse mesh is constructed in parallel by the threads of the master process. Then the coarse mesh is partitioned. Finally, the fine mesh refinement procedure is executed until all the elements in the mesh satisfy the quality and fidelity criteria. The communication and computation are separated during the fine mesh refinement procedure. The master thread of each process that initializes the MPI environment is in charge of the inter-node MPI communication for data (submesh) movement while the worker threads of each process are responsible for the local mesh refinement within the node. We conducted a set of experiments to test the performance of the algorithm on distributed memory clusters and observed that the granularity of coarse level data decomposition, which affects the coarse level concurrency, has a significant influence on the performance of the algorithm. With the proper value of granularity, the algorithm is scalable to 45 distributed memory compute nodes (900 cores).

Keywords: Hybrid Programming, Parallel Mesh Generation, Image-to-Mesh Conversion, Two-Level Parallelization

*Corresponding author

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

1. Introduction

1.1. Motivation

Most of the current supercomputer architectures consist of clusters of nodes, each of which contains multiple cores that share the in-node memory. A hybrid parallel programming model, which utilizes message passing (MPI) for the parallelization among distributed memory compute nodes and uses thread-based libraries (Pthread or OpenMP) to exploit the parallelization within the shared memory of a node, seems to be an excellent solution to take advantage of the resources of such architectures. This leads to a trend to write hybrid parallel programs that involve both process level and thread level parallelization. However, writing new hybrid programming codes or modifying existing codes for parallel mesh generation algorithms that are suitable to the supercomputer architectures brings new challenges because of the data dependencies and the irregular and unpredictable behavior of mesh refinement. It is a challenging task for the image-to-mesh conversion of the ultrahigh-resolution three dimensional images, such as BigBrain [1], considering the memory space and the execution time. In the future we will need to mesh even higher resolution images with level of detail up to the structure of individual neurons. We need a fine mesh that contains a hundred trillion of elements to model a human brain which might have more than 100 billion neurons with 1000 elements for each neuron. In order to accomplish this in a reasonable amount of time, it is necessary to develop and exploit exascale parallel mesh generation algorithm which scales well on supercomputers. In this paper, we present a three dimensional hybrid MPI and Threads parallel mesh generation algorithm which exploits the two levels of parallelization by mapping processes to nodes and threads to cores and is able to deliver high scalability on such supercomputer architectures.

Scalable, stable and portable parallel mesh generation algorithms with quality and fidelity guarantees are demanding for the real world (bio-)engineering

and medical applications. The scalability can be measured in terms of the ability of an algorithm to achieve a speedup proportional to the number of cores. The portability is the capability of an algorithm that it can be applied to different platforms without or with only a few minor modifications. The stability refers to the fact that the parallel algorithm can create the meshes that retain the same quality and fidelity as the meshes created by the sequential generator it utilizes. 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 [2, 3, 4, 5] and the angles are in a reasonable range [6, 7, 8]. 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. Delaunay mesh refinement is a popular technique for generating triangular and tetrahedral meshes for use in finite element analysis and interpolation in various numeric computing areas because it can mathematically guarantee the quality (the radius-edge ratio) of the mesh [9, 10, 11, 4].

We present a Hybrid MPI and Threads parallel Delaunay Image-to-Mesh (I2M) Conversion Isosurface-based algorithm that conforms to all of the above requirements. The isosurface-based algorithms assume that the object to be meshed is known only through an implicit function $f : R^3 \rightarrow R$ such that points in different locations evaluate f differently [4]. The isosurface is a level set of a continuous function whose domain is 3D-space. In other words, it is a surface that represents points of a constant value (e.g. pressure, temperature, velocity, density) within a volume of space. Taking a multi-material segmented image and a user-specified size function as inputs, the hybrid MPI and Thread mesh generation algorithm recovers the isosurface of the object and meshes the volume simultaneously. The surface is constructed using the Voronoi filtering surface reconstruction method proposed by Amenta and Bern [12] and adapted by Foteinos and Chrisochoides [4]: the facets of the Delaunay Triangulation

which are restricted to the surface is a correct topological and geometric representation of the surface. Therefore, our method is able to produce high-quality elements respecting at the same time the exterior and interior boundaries of tissues based on the refinement rules proposed previously [4, 13].

1.2. Contributions

In summary, the contributions of this paper are as follows.

- The algorithm proposed in this paper is the first hybrid MPI and Threads parallel mesh generation algorithm which takes complex 3D multi-labeled images as input directly.
- The algorithm is stable and creates meshes with the same quality and fidelity guarantees as the meshes created within the shared memory node. It uses the same refinement rules presented in our previous work [4, 13].
- The algorithm explores two levels of parallelization: process level (which is mapped to a node with multiple cores) and thread level (each thread is mapped to a single core in a node).
- We proposed a nested master-worker model to handle the inter-node MPI communication and intra-node local mesh refinement separately in order to overlap the communication (task request and data movement) and computation (parallel mesh refinement). The master thread that initializes the MPI environment is in charge of the inter-process MPI communication for inter-node data movement and task request. The worker threads of each process do not make MPI calls and are only responsible for the local mesh refinement work in the shared memory of each node.

This article is an improved and extended version of the conference paper [14]. The improvements can be summarized as follows:

- New functionality (a user-specified customized quality criteria) is added in the new version implementation of the algorithm. The users can customize the quality criteria to control the mesh quality in different regions.

- The procedure on how the coarse mesh is created is given and the parameters that are used to control the quality of tetrahedra are explained.
- The fine level parallel mesh refinement is explained in details in additional to the coarse level parallelization.
- We analyzed the quality of the meshes created by the algorithm and compared it with a tightly-coupled Parallel Optimistic Delaunay Mesh generation algorithm (PODM) [13] to show that the algorithm is stable and able to create meshes of the same quality as PODM.
- We tested the performance of the hybrid MPI and Threads algorithm and compared it with other available implementations to demonstrate the scalability of the algorithm. The experiments demonstrate that the algorithm is scalable to 45 distributed memory compute nodes (900 cores) with a efficiency more than 50%.

The rest of the paper is organized as follows. Section 2 presents the background of Delaunay mesh refinement and reviews the related prior work; Section 3 describes the implementation of the hybrid MPI and Threads algorithm; Section 4 presents experimental results and performance of our approach; Section 5 concludes the paper and outlines the furture work.

2. Related Work

Delaunay mesh refinement works by inserting additional (often called Steiner) points into an existing mesh to improve the quality of the elements (triangles in two dimension and tetrahedra in three dimension). The basic operation of Delaunay refinement is the insertion and deletion of points, which then leads to the removal of bad quality elements and of their adjacent elements from the mesh and the creation of new elements. If the new elements are of bad quality, then they are required to be refined by further point insertions. One of the nice features of Delaunay refinement is that it mathematically guarantees the termination after having eliminated all bad quality elements [2, 15]. In addition, the

termination does not depend on the order of processing of bad quality elements, 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 [16, 17]. Parallel Delaunay mesh generation methods can be implemented by inserting multiple points simultaneously [18, 19, 13], and the parallel insertion of points by multiple threads needs to be synchronized.

Blelloch et al. [20] proposed an approach to create a Delaunay *triangulation* of a specified point set in parallel. They describe a divide-and-conquer projection-based algorithm for constructing Delaunay triangulations of predefined point sets. Kohout [21] proposed a parallel Delaunay triangulation algorithm based on the randomized incremental insertion. The algorithm works on a shared memory workstation up to eight cores. The difference between triangulation algorithms [20, 22, 23, 24] and mesh algorithms is that the former only triangulate the convex hull of a given set of points and therefore they guarantee neither quality nor fidelity.

Andra et al. [25] 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. Galtier and George [26] described an approach of parallel mesh refinement. The idea is to prepartition the whole domain into subdomains using smooth separators 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 beginning if the created separators are not Delaunay-admissible. A parallel three-dimensional unstructured Delaunay mesh generation algorithm [27] was proposed which addresses the load balancing problem by distributing bad elements among processors through mesh migration. However, the efficiency of the algorithm is only 30% on 8 cores.

Linardakis [28] 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 se-

quential 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 Medial Axis [29, 30] which is very expensive and difficult to construct (even to approximate) for complex three dimensional geometries.

An idea of updating partition boundaries when inserted points happen to be close to them was presented [31] and extended [32] as a Parallel Constrained Delaunay Meshing (PCDM) algorithm. In PCDM, the edges on the boundaries of submeshes are fixed (constrained), and if a new point encroaches upon a constrained edge, another point is inserted in the middle of this edge instead. As a result, a split message is sent to the neighboring core, 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.

Foteinos and Chrisochoides [33, 13] proposed a tightly-coupled Parallel Optimistic Delaunay Mesh generation algorithm (PODM). This approach works well on a NUMA architecture with 144 cores and exhibits near-linear scalability. PODM scales well up to a relatively high core count compared to other tightly-coupled parallel mesh generation algorithms [34]. However, it suffers from communication overhead caused by a large number of remote memory accesses, and its performance deteriorates for a core count beyond 144 because of the network congestion caused by the communication among threads. The best weak scaling efficiency for 176 continuous cores is only about 49% on Blacklight, a cache-coherent NUMA distributed shared memory (DSM) machine in the Pittsburgh Supercomputing Center.

In our previous work [35], we described a three-dimensional locality-aware parallel Delaunay image-to-mesh conversion algorithm. The algorithm employed a data locality optimization scheme to reduce the communication overhead caused by a large number of remote memory accesses. An over-decomposed block-based partition approach was proposed to alleviate the load balancing

problem and to make LAPD ensure both data locality and load balance. However, it scaled to only about 200 cores on a distributed shared memory architecture.

Parallel Delaunay Refinement (PDR) [19, 36] is 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 [18]. 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 a coarse 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 coarse 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.

We presented a scalable three dimensional parallel Delaunay image-to-mesh conversion algorithm (PDR.PODM) for distributed shared memory architectures [37]. PDR.PODM combined the best features of two previous parallel mesh generation algorithms, the Parallel Optimistic Delaunay Mesh generation algorithm (PODM) and the Parallel Delaunay Refinement algorithm (PDR). PDR.PODM is able to explore parallelization early in the mesh generation process because of the aggressive speculative approach employed by PODM. In addition, it decreases the communication overhead and improves data locality by making use of a data partitioning scheme offered by PDR. Although it shows nice scalability up to about 300 cores, the performance deteriorated when more cores are applied for the tests performed on a distributed shared memory architectures as shown in Fig. 8d.

A number of other parallel mesh generation algorithms have been published,

which are not the Delaunay-based algorithms. De Cougny, Shephard and Ozturan [38] proposed an algorithm in which the parallel mesh construction is based on an underlying octree. Lohner and Cebral [39], and Ito et al. [40] developed parallel advancing front schemes. Globisch [41, 42] presented a parallel mesh generator which uses a sequential frontier algorithm.

Ibanez et al. [43] proposed a hybrid MPI-thread parallelization of adaptive mesh operations. They presented an implementation of non-blocking inter-thread message passing from which they built non-blocking collectives and phased message passing algorithm. A variety of operations for handling adaptive unstructured meshes are implemented based on these message passing capabilities and the phased communication and migration times are directly determined by neighborhood sizes. They presented new workflows enabled by the ability to vary the number of threads during runtime. The algorithm shows a good speedup in the experiment within a node of two processes and sixteen cores.

Gorman et al. [44] presented an optimisation based mesh smoothing algorithm for anisotropic mesh adaptivity. The smoothing kernel they proposed solved a non-linear optimisation problem by differentiating the local mesh quality with respect to mesh node position and employing hill climbing to maximise the quality of the worst local element. It is shown that this approach is effective at raising globally the minimum element quality of the mesh. The algorithm is able to reduce the cost while maintaining its effectiveness in improving overall mesh quality. The method was parallelised using a hybrid OpenMP/MPI programming method and graph coloring to identify independent sets. The experimental results shown that the smoothing kernel is very effective at raising the minimum local quality of the mesh and it achieves a good scaling performance within a shared memory compute node.

3. MPI and Threads Implementation

In this section, we present a hybrid MPI and BoostC++ Threads parallel image-to-mesh conversion implementation for distributed memory clusters. The

algorithm explores two levels of concurrency: coarse-grain level concurrency among subregions and medium-grain level concurrency among cavities. As a result, the implementation of our algorithm exploits two levels of parallelization: process level parallelization (which is mapped to a node with multiple cores) and thread level parallelization (which is mapped to a single core in a node).

In the coarse-grain parallel level, the master process first creates an coarse mesh in parallel using all its threads. Then it decomposes the whole region (the bounding box of the input image) into subregions and assigns the bad elements of the coarse mesh into subregions based on the coordinates of their circumcenters. Finally, the master process uses a task scheduler to manage and schedule the tasks (subregions) to worker processes through MPI communication. In subsection 3.2, we describe a method how to select and schedule a subset of independent subregions to multiple processes, which can be refined simultaneously without synchronization. In the medium-grain parallel level, the process of each compute node launches multiple threads that follow the refinement rules of PODM in order to refine the bad elements of each subregion in parallel by inserting multiple points simultaneously. Fig. 2 and Fig. 3 give a high level description of our hybrid MPI and Threads parallel mesh generation algorithm.

3.1. Parallel Coarse Mesh Generation

The coarse mesh is created in parallel by the multiple threads of the master process that is running on master node. First, the master thread of the master process sequentially creates the initial mesh, i.e., the six tetrahedra that tessellate the bounding box of the input image. This is the only sequential meshing step in our algorithm. Then, this initial mesh is refined in parallel by all threads of the master process to a certain level and the coarse mesh is created [4, 13]. After that it decomposes the whole region (the bounding box of the input image) into subregions and assigns the bad elements of the coarse mesh into subregions based on the coordinates of their circumcenters. If the circumcenter of a tetrahedron falls inside a subregion, the tetrahedron is assigned to the bad element list of that subregion. The most important parameter is the circumradius up-

per bound $\bar{r}_I$ which is used to control the volume upper bound of tetrahedra in the inital mesh. This number determines the number of elements of the coarse mesh. The larger $\bar{r}_I$ is, the larger 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 coarse 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 coarse 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. The other parameter is the circumradius to shortest edge ratio (radius-edge ratio for short) and is set to 2.

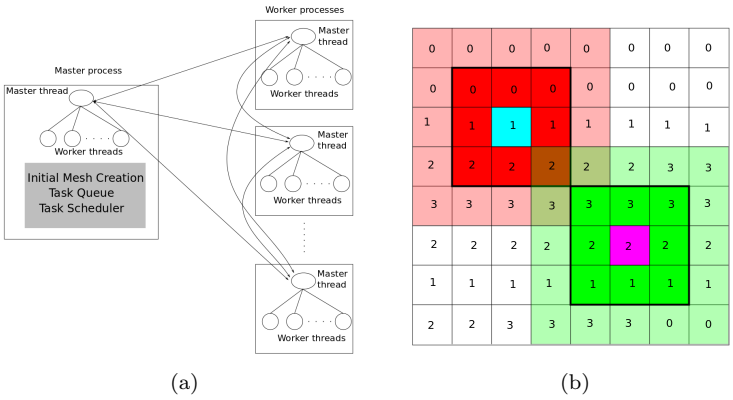


Figure 1: (a) A diagram that illustrates the design of nested master-worker model. (b) A two dimensional illustration of three-dimensional buffer zones. It is an example with two subregions (the cyan and magenta subregions), which can be refined independently and simultaneously. The dark green and dark red regions around these two subregions form their first level buffer zones respectively. The light green and light red regions represent the second level buffer zones. The conflict between two multi-threaded processes working on different subregions is eliminated during the refinement. Each subregion has an integer flag that represents the process rank (node ID) where the actual data (submesh) inside each subregion is stored.

3.2. Coarse Level Data Decomposition and Task Scheduler

We used a simple but efficient way to decompose the whole input image into subregions, which consists in partitioning the bounding box into cubes. Each cube is considered as a subregion and the elements inside one cube constitute the submesh of a subregion. Then, we assign tetrahedra to different subregions

based on the coordinates of their circumcenters. Each subregion has a queue of elements that are inside the subregion. We use a two-level buffer scheme to select and schedule independent subregions to multiple processes, which can be refined simultaneously without synchronization. Consider a subregion and the twenty six neighbor subregions form its first level buffer zone (dark red or dark green region shown in Fig. 1b). When a subregion is under refinement, the point insertion operation might propagate to one neighbor subregion of its first level buffer zone. This leads to the update operation of the element queue of that neighbor subregion. If two subregions that are under refinement by two processes have a common neighbor subregion, the synchronization is needed in the neighbor subregion. We use a second level buffer zone (light red and light green regions in Fig. 1b) in order to ensure that the first level buffer zones of two subregions under refinement are not overlapping and the synchronization is eliminated.

A subregion is considered as a task that can be dealt with by one process and the subregions in its second level neighbors are considered as dependent tasks. A subregion which is outside the second level neighbors is an independent task and can be refined by another process concurrently. We used a task queue and task scheduler to schedule the independent tasks that can be refined by multiple processes simultaneously based on the two level buffer zones. The idea of the task scheduler is straightforward: if one task (subregion) is popped up from the task queue during the refinement, all its dependent tasks, i.e., its first and second level buffer neighbors are also popped up. This guarantees that two subregions that are scheduled to be refined simultaneously are at least two layers (subregions) away from each other and independent. During the refinement procedure, the point insertion operation might propagate to one subregion of its first level buffer zone. Therefore, if the submesh of one subregion was scheduled to one worker process for refinement, the submeshes of its first level neighbors also need to move to the local memory of the worker process. Each subregion has an integer flag that represents the process rank (node ID) where the actual data (submesh) inside each subregion is stored as shown in Fig. 1b. The worker

process sends data request messages to collect the submeshes of one subregion and its first level neighbor subregions from other workers based on the integer flags (lines 6 to 18 in Fig. 3).

3.3. Nested Master-Worker Model

We propose a nested master-worker model in order to take advantage of the two level parallelization on multicore distribute clusters. Fig. 1a is a diagram that illustrates the design of nested master-worker model. The master process running on a node (called master node) creates the coarse mesh, manages and schedules the taskes (subregions) and the worker processes running on other nodes (worker nodes) communicate with each other and master process for task request and data migration. Within each node, the process is multithreaded and each thread runs on one core of the node.

In the implementation, the MPI communication and local shared memory mesh refinement is separated in order to overlap the communication and computation. The master thread of each process that runs on each compute node initializes the MPI environment. Then it creates new worker threads and pins each worker thread on one core of the compute node. Therefore, the number of threads of each process (master and worker threads) is equal to the number of cores of each node. The master thread initializes the MPI environment and communicates with the master thread of other processes that run on other nodes for data movement and task requests. The worker threads of each process do not make MPI calls and are only responsible for the local mesh refinement work in the shared memory of each node.

Fig. 2 and Fig. 3 list the main steps of master process and worker process of the nested master-worker model respectively. In the algorithm, each subregion is considered as a task and the submesh inside the subregion is the actual data. If we denote P_0 as the master process and P_i, P_j as worker processes, the main steps of the algorithm can be summarized as follows: (i) the master process P_0 creates the coarse mesh, decomposes the coarse mesh and initializes the task queue and scheduler (lines 1 to 5 in Fig. 2); (ii) a worker process P_i sends a

```
MASTER PROCESS(P_0)(I, δ̄_t, δ̄_I, g)
Input: I is the input segmented image;
       δ̄_t is the circumradius upper bound of the elements in the final mesh;
       δ̄_I is the circumradius upper bound of elements in the coarse mesh;
       g is value of granularity;
Generate an coarse mesh in parallel that conforms to δ̄_I;
Use a uniform octree to decompose the whole region into subregions based on g;
Find the buffer zones of each subregion of the octree;
Distribute the coarse mesh to octree leaves based on their circumcenter coordinates;
Push all octree leaves to a task queue Q;
while (1)
    Probe the message;
    if The message is a task (subregion) request message from a worker process P_i;
        if Q! = ∅
            Receive message from P_i;
            Get one subregion L from task queue Q;
            Send L and its neighbors' submeshes Location information to P_i;
            //Location is an array that contains the process ranks,
            //which hold the submesh of L or its first level neighbors.
            Set process P_i to status HAS_WORK;
        else if Q == ∅ && at least one worker process's status is HAS_WORK;
            Receive message from P_i;
            Put P_i to waiting task list WTL;
            Send a message to P_i with status WAIT_IN_LIST;
        else if Q == ∅ && all worker processes' statuses are NO_WORK;
            Send termination message to P_i;
            Send termination message to every process that is waiting in the WTL;
            if the number of terminated workers == the number of workers
                break;
            endif
        endif
    endif
    if The message is a data (submesh) request message from P_i
        Receive the message from P_i;
        Pack data (submesh);
        Send data (submesh) to P_i;
    endif
    if The message is a feedback from P_i that just finished the refinement work
        Receive the message from P_i;
        Set P_i to status NO_WORK;
        Update task queue Q based on the feedback message from P_i;
        while Q! = ∅ && waiting task list WTL is not empty
            Get one subregion from Q;
            Pop one process P_j from waiting task list WTL;
            Send subregion and neighbors Location information to P_j;
            Set P_j to status HAS_WORK;
        endwhile
    endif
endwhile
```

Figure 2: A high level description of Master Process's (P_0) work.

```
WORKER PROCESS(P_i)()
 1: while (1)
 2:     Send a task (subregion) request to master process P_0;
 3:     Probe the message;
 4:     if The message is a task message from master process P_0;
        //The message contains a task L and its first level neighbors.
 5:         Receive the message from P_0;
 6:         Send data (submesh) request to each process P_k in Location array;
 7:         while (1)
 8:             Probe the message;
 9:             if The message is a data (submesh) request message from a process P_j;
10:                 Receive the message from P_j;
11:                 Send data (local submesh) to P_j;
12:             else if The message contains data (submesh) from a process P_k
13:                 Receive the message from P_k;
14:                 number of submeshes received += number of submeshes P_k holds;
15:                 if number of submeshes received == number of submeshes needed
16:                     break;
17:             endif
18:         endwhile
19:         Pass the submesh to worker threads for mesh refinement;
20:         while the worker threads are doing the mesh refinement
21:             Probe the message;
22:             if The message is a data (submesh) request message from a process P_j;
23:                 Receive the message from P_j;
24:                 Send data (local submesh) to P_j;
25:             endif
26:         endwhile //Local Mesher has finished the refinement work;
27:         Send feedback message with mesh refinement information to P_0;
28:     else if The message is a message from P_0 with status WAIT_IN_LIST
29:         while P_i is waiting for new task
30:             Probe the message;
31:             if The message is a data (submesh) request message from a process P_j;
32:                 Receive the message from P_j;
33:                 Send data (local submesh) to P_j;
34:             endif
35:         endwhile
36:     else if The message is a termination message from P_0
37:         break;
38:     endif
39: endwhile
```

Figure 3: A high level description of a Worker Process's (P_i) work.

task (subregion) request to master process P_0 (line 2 in Fig. 3); (iii) P_0 receives the task request from P_i, pops one task (subregion L) and its dependent tasks (neighbors) from the task queue and sends the subregion and its neighbors' submeshes *Location* information to P_i (lines 8 to 13 in Fig. 2); (iv) P_i sends data request to each process P_j who has the submeshes that P_i needed (lines 4 to 18 in Fig. 3); (v) after getting all the submeshes of L and its neighbors, the worker threads of P_i start the mesh refinement; (vi) P_i sends feedback message to master process P_0 and P_0 updates the task queue based on the feedback message (lines 31 to 41 in Fig. 2); (vii) if all the refinement work is done, P_0 sends termination message to each worker process P_i and master process exits after all worker processes terminate (lines 18 to 24 in Fig. 2).

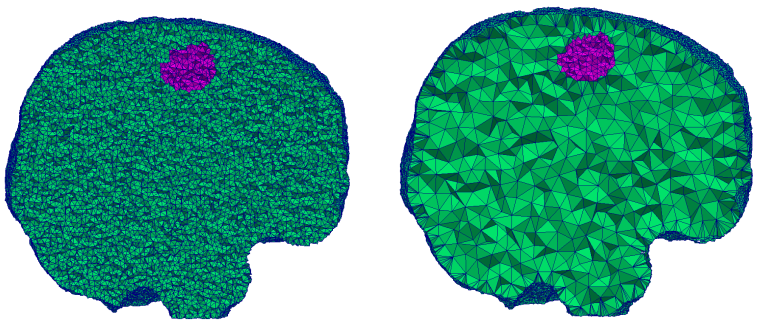
A worker process does not send the submeshes of a subregion and neighbors back to master process after it has finished the refinement work. Instead, it sends a feeback message that only contains the number of bad elements of each subregions to the master process. The master process updates the task queue based on the feedback message to decide whether a subregion needs to be pushed back to the task queue for further refinement. A data (submeshes) collection operation is needed when a worker process gets a task (subregion) to refine. The worker process sends data request to other worker processes who hold the submeshes in their local memories. A worker process is likely to send data requests to other worker processes and receive data requests from these worker processes simultaneously. In order to handle the interleaving messages among the worker processes and avoid deadlocks, non-blocking MPI communication and a message polling approach are used when the master thread of a worker process try to collect the submeshes it needs from other worker processes (lines 6-18 in Fig. 3). When the worker threads of a process are doing the refinement work, the master thread is still able to receive and response to the data requests from other workers (lines 20-26 in Fig. 3) since the communication and computation are separated.

3.4. Fine Level Parallel Mesh Refinement

Once a process is assigned a task (a subregion), the worker threads of this process start to refine the submesh of the subregion in parallel. In the algorithm, each worker thread T_i of a process maintains its own Bad Element List (BEL) which contains the elements that violate the quality or fidelity criteria [4] and have been assigned to this thread for further refinement.

The basic operation of Delaunay refinement is the insertion of a point p to eliminate a bad element. The cavity is a set of elements that violate the Delaunay property because their circumsphere contains the inserted point p. The elements in the cavity are deleted and p is connected to the vertices of the boundary of the cavity to create new elements. The cavity is constructed and re-triangulated according to the well known Bowyer-Watson kernel [16, 17]. If a vertex of an element is already locked by another thread during the cavity expansion, a rollback occurs [13, 33]. This happens when two threads try to refine bad elements that are close to each other in a submesh. If a rollback happens, the operation is stopped and the changes are discarded. When a rollback occurs, the thread moves on to the next bad element in its Bad Element List (BEL).

One of the challenges is that the algorithm needs to maintain load balance in two levels: the coarse-grain level load balance among processes and fine-grain level load balance among threads within a process. In the coarse-grain level, we utilized the method called over-decomposition [45] to deal with the load balancing problem among processes. We over-decomposed the whole region so that the number of subregions is much larger then the number of processes to ensure that each process has enough subregions to refine. This method introduces some overhead but it does help to alleviate the load balancing problem among processes. A study of optimal load balancing strategies among processes, while keeping the overhead and communication cost small, is part of our future work. In the fine-grain level, a load balancing list is used to spread the elements among threads of a process. Each thread has the flexibility to communicate with other threads that belong to the same process during the refinement. A worker thread



(a) (b)

Figure 4: (a) A uniform mesh that the tetrahedra in the mesh meet the same quality criteria. (b) A non-uniform mesh that a dense and better quality mesh is created in the critical region.

T_i pushes back its $Thread\_ID$ to the load balancing list, if the bad element list BEL_i of a thread T_i does not contain any elements. Then, T_i goes to sleep and is able to be awaken by another thread T_j when T_j produces some work for T_i. After a thread T_j completes a Delaunay insertion operation, it checks all the newly created elements and puts the ones that are regarded as bad elements to the BEL of the first thread T_i found in the load balancing list. T_j also removes T_i from the load balancing list.

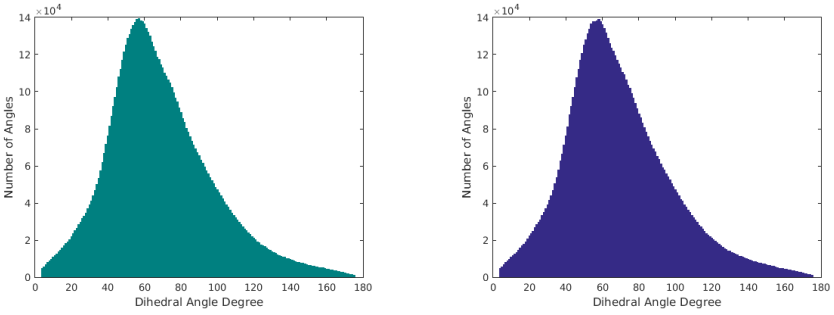
The hybrid MPI and Thread algorithm can generate uniform mesh as demonstrated in Fig. 4a. In this case, the tetrahedra in all regions meet the same quality criteria (element size bound, radius-edge ratio and dihedral angle). It is also able to create non-uniform mesh, i.e., the elements in different region meet different quality criteria, as shown in Fig. 4b. In this case, only the quality of elements in a certain region (often called critical region) is required while the mesh quality of other regions are not required. In the algorithm, the users can customize the quality criteria for different regions and pass these criteria as input parameters to control the mesh quality in different regions. If the applications only require the mesh quality improvement in a certain region, the algorithm is able to create a dense and high-quality mesh inside the critical region while keeping a coarse mesh outside the region.

4. Performance

4.1. Experimental Platform, Inputs and Evaluation Metrics

We have conducted a set of experiments to assess the performance of the hybrid MPI and Threads parallel mesh generation algorithm. The experimental platform is the Turing cluster computing system at High Performance Computing Center of Old Dominion University. We tested the performance of our implementation on Turing with its two subclusters: Phi cluster and Ed-Main cluster. Phi cluster contains 9 Intel Xeon Phi nodes each with 2 Xeon Phi MIC cards and 20 cores. Ed-Main cluster of Turing contains 190 multi-core compute nodes each containing between 16 and 32 cores and 128 Gb of RAM. We have used two 3D multi-tissued images as inputs in the experiments: (i) the CT abdominal atlas obtained from IRCAD Laparoscopic Center [46], and (ii) the knee atlas obtained from Brigham & Women's Hospital Surgical Planning Laboratory [47]. We performed the experiments on Phi cluster using up to 180 cores and on Ed-Main cluster up to 900 cores (45 compute nodes, the maximum number of nodes available for us in the experiments). We use the **Weak Scaling 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)) and **Weak Scaling Efficiency *E*** (The ratio of speedup (S) to the number of cores (p): $E = S/p = T_s/(pT_p)$) to evaluate the scalability of parallel mesh generation algorithms [48, 49].

In the weak scaling case, the number of elements per core remains approximately constant. In other words, the problem size (i.e., the number of elements created) increases proportionally to the number of cores. For example, with the input image abdominal atlas, the number of elements generated equals about 6.64 million on a single core. It increases approximately to 1.17 billion tetrahedra for 180 cores on Phi cluster and up to 3.86 billion tetrahedra for 1200 cores on Ed-Main cluster. In the experiments, it is impossible to control the problem size increased exactly by p times when the number of cores is increased from 1 to p because of the irregular nature of the unstructured tetrahedra mesh.



(a) (b)

Figure 5: (a) Dihedral angle distribution of the final mesh created by the hybrid MPI and Thread method. (b) Dihedral angle distribution of the final mesh created by the PODM.

Therefore, an alternative definition of speedup is used which is more precise for a parallel mesh generation algorithm. We measure the number of elements generated every second during the experiment. Then the speedup can be calculated as $S(p) = \frac{elements\_per\_sec(p)}{elements\_per\_sec(1)}$.

4.2. Mesh Quality Analysis

In this subsection, we analyze the quality of the meshes created by our algorithm. The quality of mesh refers to the quality of each element in the mesh which is measured by Delaunay methods in terms of its circumradius-to-shortest edge ratio (radius-edge ratio for short) and (dihedral) angle bound. The radius-edge ratio of a tetrahedron is defined as the ratio of its circumradius to the length of its shortest edge. The radius-edge ratio of the created mesh is theoretically guaranteed by the Delaunay refinement method and the actual

Table 1: Mesh quality comparison of our method and PODM. The two input images are abdominal atlas and knee atlas.

	Abdominal Atlas		Knee Atlas	
	MPI+Threads	PODM	MPI and Thread	PODM
number of elements	1,355,131	1,352,737	832,569	831,674
number of vertices	244,066	244,027	185,752	185,703
edge-radius ratio bound	2	2	2	2
min dihedral angle	4.89°	4.78°	4.96°	4.94°
max dihedral andle	174.21°	174.47°	174.89°	175.11°

edge ratio bound in our implementation is less than 2 in our implementation. Because of the potential nearly flat tetrahedra, a more useful measure is the dihedral angle. We compared the quality of meshes that were created by our method with the quality of meshes that were created by PODM method on the two multi-material 3D input images (abdominal atlas and knee atlas) as shown in Table 1. We also showed the dihedral angle distribution of the final meshes created by our algorithm and PODM in Fig. 5a and Fig. 5b. The goal of such comparisons is to illustrate that the hybrid MPI and Threads is able to generate meshes with the same quality guarantees as PODM.

4.3. Scalability, Granularity and Concurrency

In this subsection, we present the weak scaling performance of the implementation on Phi cluster up to 180 cores (9 compute nodes) with different data decomposition granularities. The number and size of subregions into which a problem is decomposed determines the granularity of the decomposition. In the implementation, we used a uniform octree to decompose the whole image and the depth of the octree determines the number of leaves (subregions) of the decomposition. The number of subregions is $N_{sub} = 8^d$, where d is the depth of the octree. In the algorithm, we pass the depth of the octree as an input parameter to control the granularity of the coarse level data decomposition. We performed the experiments on Phi cluster with two different data decomposition granularities:

- $d = 3$ represents the octree was split to depth 3 with 512 subregions.
- $d = 4$ represents the octree was split to depth 4 with 4096 subregions.

The problem size, i.e., the number of tetrahedra, increases linearly with respect to the number of cores. The number of tetrahedra created gradually increases from 6.64 million to 1.17 billion for the input image abdominal atlas, and from 6.33 million to 1.14 billion for knee atlas when the number of cores increases from 1 to 180. Table 2 and Table 3 show the weak scaling perfor-

mance of the algorithm for the two input images, abdominal atlas and knee atlas respectively.

Table 2: Weak scaling performance of data decomposition with different granularities. The input is abdominal atlas.

Cores	Elements (million)	Running Time (s)		million elements/s		Speedup		Efficiency%	
		depth3	depth4	depth3	depth4	depth3	depth4	depth3	depth4
1	6.64	64.70	64.70	0.10	0.10	1.00	1.00	100.00	100.00
20	133.93	76.47	76.47	1.75	1.75	17.07	17.07	85.35	85.35
40	261.54	102.63	177.09	2.55	1.49	24.84	14.50	62.10	36.25
60	390.61	110.35	156.09	3.54	2.51	34.50	24.50	57.50	40.83
80	520.31	115.02	141.44	4.52	3.69	44.09	35.96	55.11	44.95
100	650.16	125.96	133.79	5.16	4.87	50.31	47.45	50.31	47.45
120	780.13	137.03	129.54	5.69	6.03	55.49	58.77	46.24	48.97
140	905.07	149.64	125.00	6.05	7.25	58.95	70.64	42.11	50.46
160	1034.34	158.47	120.28	6.53	8.60	63.62	83.85	39.76	52.41
180	1167.95	177.24	119.56	6.57	9.74	64.01	94.89	35.56	52.72

Table 3: Weak scaling performance of data decomposition with different granularities. The input is knee atlas.

Cores	Elements (million)	Running Time (s)		million elements/s		Speedup		Efficiency%	
		depth3	depth4	depth3	depth4	depth3	depth4	depth3	depth4
1	6.33	64.27	64.27	0.10	0.10	1.00	1.00	100.00	100.00
20	126.24	76.20	76.20	1.66	1.66	16.83	16.83	84.14	84.14
40	257.01	83.16	179.44	3.09	1.44	31.39	14.60	78.48	36.51
60	383.76	97.56	158.04	3.93	2.43	39.96	24.67	66.59	41.12
80	508.18	109.02	149.37	4.66	3.40	47.35	34.52	59.18	43.15
100	633.73	127.67	139.03	4.96	4.55	50.42	46.19	50.42	46.19
120	762.09	141.62	136.83	5.38	5.55	54.66	56.39	45.55	46.99
140	886.62	152.78	132.96	5.80	6.64	58.95	67.49	42.10	48.21
160	1014.09	174.60	127.26	5.81	7.93	59.00	80.60	36.87	50.37
180	1142.34	199.07	125.32	5.74	9.07	58.29	92.14	32.38	51.19

As demonstrated in Table 2 and Table 3, the algorithm gets near-linear weak scaling performance for both of the two inputs when the number of cores is less than or equal to 20. The efficiency with 20 cores is about 85%. The reason is that the refinement work was done inside one compute node with shared memory and no core was dedicated to MPI communication in this case. Therefore, no inter-node communication overhead was introduced. The algorithm shows better weak scaling performance with $d = 3$, i.e., the coarse mesh and underlying image is partitioned into 512 subregions than that with $d = 4$, i.e., the coarse mesh is partitioned into 4096 subregions when the number of cores is less than or equal to 100 (5 nodes). There are two reasons. First, the decrease of granularity with more subregions does not necessarily lead to the increase of degree of concurrency because the maximum number of tasks (subregions) that can be executed (refined) simultaneously is limited by the number of available cores. Second, the decrease of granularity, which increases the number of subre-

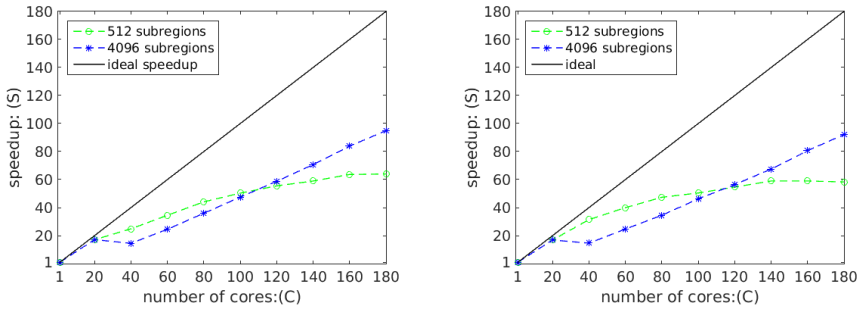


Figure 6: (a) Weak scaling speedup comparison of two different granularities for the input image abdominal atlas. (b) Weak scaling speedup comparison of two different granularities for the input image knee atlas.

gions, introduces more overheads. As demonstrated in Fig. 7a and Fig. 7b, the communication overhead (the red part) with 4096 subregions (the right bar) is always higher than the communication overhead with 512 subregions (the left bar). The large overhead leads to the speedup of 40 cores (2 nodes) even lower than that of 20 cores with 512 subregions as demonstrated in Fig. 6a and Fig. 6b. The algorithm exhibits better scalability when the octree depth is 4 (4096 subregions) and the number of cores is more than 120 (6 nodes) as shown in Table 2 and Table 3 . We observed that each time we increase the number of cores, the efficiency of experiment with 4096 subregions increases while the efficiency with 512 subregions decreases. Take the experimental result of input abdominal as an example, the efficiency with 512 subregions on 40 cores is 62.10% and it decreases to 35.56% for 180 cores. In contrast, the efficiency with 4096 subregions on 40 cores is 36.25% and it increases to 52.72% for 180 cores. Fig. 6a and Fig. 6b illustrate the speedup comparison with two different granularities for two input images respectively. For 512 subregions, the gradient of speedup becomes smaller and smaller with the number of cores (nodes) increasing and the speedup with 180 cores is almost the same as that with 160 cores. In contrast, for 4096 subregions, the speedup increases almost linearly compared to the speedup with 40 cores.

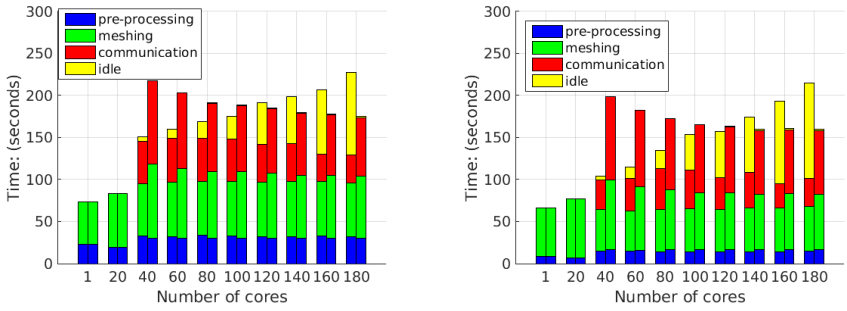


Figure 7: The breakdown of the running time of two different granularities. The left bar in each bar graph is the time breakdown with 512 subregions and the right one is the time breakdown with 4096 subregions. (a) The breakdown of the running time of experiments for abdominal atlas. (b) The breakdown of the running time of experiments for knee atlas.

Fig. 7a and Fig. 7b show the breakdown of the total running time for the experiments with the two images respectively. The running time consists of four parts: (i) the pre-processing time is the time that the master process spends on loading an image from disk, constructing an octree, creating the coarse mesh, assigning the elements of the coarse mesh to subregions and creating subthreads; (ii) the meshing time is the time that a process (more precisely, the multiple worker threads of a process) spends on mesh refinement; (iii) the communication time is the time that a process spends on task requests and data movement; (iv) the idle time is the time that a process waits in the waiting list and does not perform any mesh refinement work. Each bar is the sum of the time that a process spends on each part for each iteration (In each iteration, the process requests a subregion and refines the submesh inside the subregion). We calculate the average time of each part for all processes. As demonstrated in Fig. 7a and Fig. 7b, the idle time with large granularity (512 subregions) keeps on increasing from 40 cores (2 nodes) to 180 cores (9 nodes). It becomes the major overhead that deteriorates the performance of the algorithm when more than 5 nodes are used because of the low degree of concurrency. In this case, a finer decomposition is required although it introduces more overhead. In addition, the communication overhead (the red part) with small granularity (the right

bar) is always higher than the communication overhead with large granularity (the left bar). In fact, we can see clearly the basic tradeoffs in parallel computing between granularity and concurrency: we have to decrease the granularity in order to increase the concurrency, which introduces more overhead. If there are not enough processing units to exploit the maximum degree of concurrency, a finer data decomposition with smaller granularity deteriorates the performance of the algorithm because of the higher overhead it introduces.

The strong scaling performance of the algorithm is determined by the problem size. If the problem size is very large, i.e., creating a very large mesh, in the experiments, the algorithm will demonstrates good strong scaling performance; On the contrary, if the problem size is very small, the strong scaling performance will deteriorate. This issue has been pointed out and analyzed by J. L. Gustanfson [48]. As he stated: on ensemble (distributed) computers, fixing the problem size creates a severe constraint, since for a large ensemble (with the small fixed problem size) it means that a problem must run efficiently even when the problem occupies only a small fraction of available memory. Therefore, we only demonstrated the weak scaling performance of our algotithm to show its ability to solve large problem utlizing large number of cores.

4.4. Performance Evaluation and Comparison

We ran a set of experiments on Ed-Main cluster of Turing cluster system up to 900 cores (45 nodes) to test the scalability of the algorithm. We use the same two input images, abdominal atlas and knee atlas, as the test expreiments we ran on the Phi cluster. Based on the analysis of the subsection above, we ran the experiments with the optimal value of octree depth, i.e. $d = 3$ when the number of nodes is less than or equal to 5 (100 cores) and $d = 4$ when the number of nodes is between 6 to 30 (120 to 900 cores). Fig. 8a demonstrates the weak scaling speedup of the two input images up to 900 cores (45 nodes) on Ed-Main cluster of Turing.

We compared the performance of hybrid MPI and Threads algorithm with other three shared memory algorithms, PODM [13], LAPD [35] and PDR.PODM

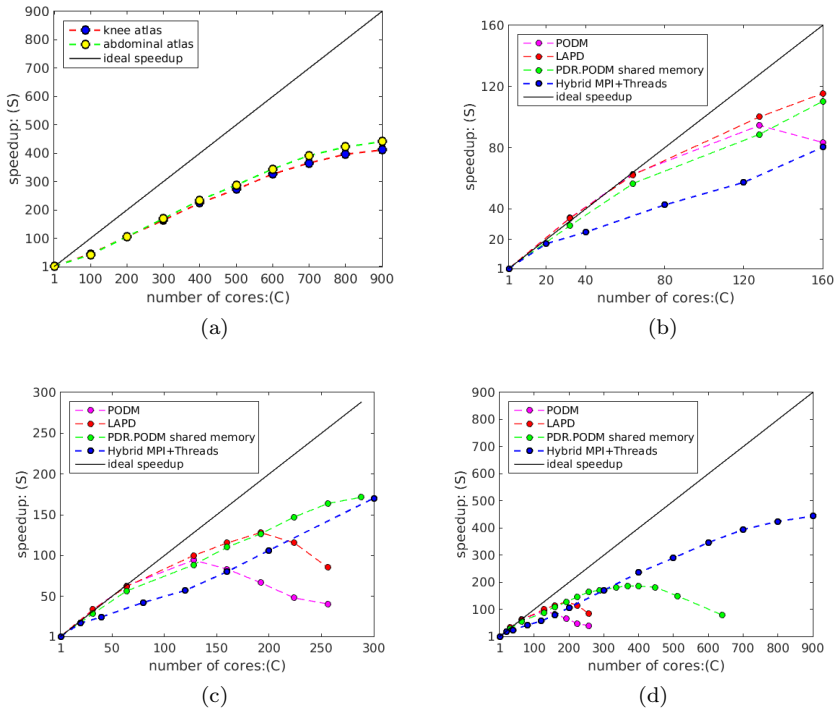
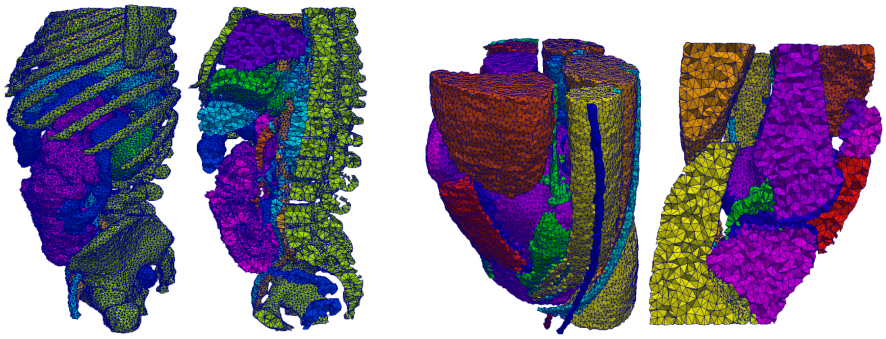


Figure 8: (a) The overall weak scaling speedup of the two input images up to 900 cores (45 nodes) on Ed-Main cluster of Turing. (b), (c), (d) The weak scaling speedup comparison of hybrid MPI and Threads implementation with other three shared memory algorithm implementations and ideal speedup for upto 160, 300 and 900 cores.

[37]. The main characteristics of these algorithms are listed in Table 4. The input image we use in all experiments is the abdominal atlas. Fig. 8b shows the speedups of the four implementations upto 160 cores. As we can see, The locality-aware parallel mesh generation algorithm LAPD has the best performance because it increases the data locality during the parallel mesh refinement.

Table 4: Comparison of parallel Delaunay image-to-mesh conversion algorithms.

Methods	Max Cores	Elements Per Second	Platform	Main Characteristics
PODM	128	10.42 million	Shared Memory	First parallel image-to-mesh conversion algorithm.
LAPD	192	14.12 million	Shared Memory	Improving the locality during refinement.
PDR.PODM	256	18.02 million	Shared Memory	Taking advantages of two previous algorithms [13, 36] to improve the scalability.
MPI+Threads	900	45.25 million	Distributed Memory	Hybrid MPI and Threads algorithm involves two level parallelization.



(a) (b)

Figure 9: (a) A mesh example created by the algorithm with input abdominal atlas. (b) A mesh example created by the algorithm with input knee atlas.

In fact, all the other three methods have better performance than that of hybrid MPI and Threads algorithm when the number of cores is small (less than 300). The reason is that the hybrid methods introduces the process level data migration and movement, which introduces additional communication overhead through MPI routines compared to the pure shared memory parallel mesh refinement. However, when the number of cores increases, the scalability potential of the hybrid method becomes more and more obvious. When the expreiments are performed with more than 300 cores, the MPI and Thread methods has the best performance compared with the other three methods because of the two level of parallelization it utilizes. Fig. 9a and Fig. 9b show two Delaunay meshes created by the algorithm with input images abdominal atlas and knee atlas.

5. Conclusion and Future Work

We presented a scalable hybrid MPI and Threads image-to-mesh conversion algorithm on distributed memory clusters with multiple cores. The algorithm is able to create meshes with quality guarantees. First, a coarse mesh is constructed and decomposed. Then, the fine mesh refinement procedure starts and runs until all the elements in the mesh satisfy the quality criteria. The algorithm explores two levels of parallelization: process level parallelization (which is mapped to a node with multiple cores) and thread level parallelization (each

thread is mapped to a single core in a node). We proposed a nested master-worker model in order to take advantage of the two-level parallelization on multicore distributed memory clusters. In order to overlap the communication (task request and data movement) and computation (parallel mesh refinement), the inter-node MPI communication and intra-node local mesh refinement are separated. The master thread initializes the MPI environment and communicates with the master threads of other processes that run on other nodes for data movement and task requests. The worker threads of each process do not make MPI calls and are only responsible for the local mesh refinement work in the shared memory of each node.

We compared the performance of the algorithm with our previous algorithms as detailed in Section 4.4. The experimental results demonstrated that the hybrid MPI and Threads algorithm proposed in this paper is quite suitable to the hierarchy of distributed memory clusters with multiple cores and shows so far the best scalability. We conducted the experiments on up to 900 cores (45 nodes, the maximum number of nodes available for us so far) and the speedup is increasing almost linearly with the number of nodes as illustrated in Fig. 8a. This trend is likely to continue to higher number of cores (nodes) based on the speedup shown in Fig. 8a. Therefore, our future work includes assessing the performance of the hybrid algorithm with a larger number of cores. The communication overhead takes a certain portion in the total running time. One of our future tasks is to reduce the communication overhead and improve the data locality to further improve the performance of the algorithm.

Acknowledgements

This work is in part funded by NSF grants CCF-1439079, CC-NIE 1440673, NASA grant NO. NNX15AU39A, DoD's PETTT Project PP-CFD-KY07-007 and the Richard T. Cheng Endowment. The content is solely the responsibility of the authors and does not necessarily represent the official views of the government agencies that support this work. We thank the systems group especially

Min Dong and Terry R. Stilwell in the High Performance Computing Center of Old Dominion University for their great help. We thank all the reviewers for their detailed comments which helped us improve the manuscript.

References

[1] K. Amunts, C. Lepage, L. Borgeat, H. Mohlberg, T. Dickscheid, M.-É. 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.

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

[3] H. Si, Tetgen, a Delaunay-based quality tetrahedral mesh generator, ACM Trans. Math. Softw. 41 (2) (2015) 11:1–11:36.

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

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

[6] 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.

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

[8] 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.

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

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

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

[12] N. Amenta, M. Bern, Surface reconstruction by voronoi filtering, Discrete & Computational Geometry 22 (4) (1999) 481–504.

[13] 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.

[14] D. Feng, A. N. Chernikov, N. P. Chrisochoides, A hybrid parallel delaunay image-to-mesh conversion algorithm scalable on distributed-memory clusters, Procedia Engineering 163 (2016) 59 – 71, 25th International Meshing Roundtable.

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

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

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

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

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

[20] 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.

[21] J. Kohout, I. Kolingerová, Parallel delaunay triangulation in e3: make it simple, The Visual Computer 19 (7) (2003) 532–548.

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

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

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

[25] H. Andra, G. Gluchshenko, E. Ivanov, A. Kudryavtsev, Automatic parallel generation of tetrahedral grids by using a domain decomposition approach, Computational Mathematics and Mathematical Physics 48 (8) (2008) 1367–1375.

[26] 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.

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

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

[35] D. Feng, A. N. Chernikov, N. P. Chrisochoides, Two-level locality-aware parallel delaunay image-to-mesh conversion, Parallel Computing 59 (Supplement C) (2016) 60 – 70, theory and Practice of Irregular Applications.

[36] 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.

[37] D. Feng, C. Tsolakis, A. Chernikov, N. Chrisochoides, Scalable 3D hybrid parallel Delaunay image-to-mesh conversion algorithm for distributed shared memory architectures, in: 24th International Meshing Roundtable, Austin, Texas, 2015.

[38] H. L. de Cougny, M. S. Shephard, C. Ozturan, 3rd national symposium on large-scale structural analysis for high-performance computers and workstations parallel three-dimensional mesh generation, Computing Systems in Engineering 5 (4) (1994) 311 – 323.

[39] R. Löhner, J. R. Cebral, Parallel advancing front grid generation, in: Proceedings of the 8th International Meshing Roundtable, South Lake Tahoe, CA, 1999, pp. 67–74.

[40] Y. Ito, A. M. Shih, A. K. Erukala, B. K. Soni, A. Chernikov, N. P. Chrisochoides, K. Nakahashi, Parallel unstructured mesh generation by an advancing front method, Mathematics and Computers in Simulation 75 (5) (2007) 200 – 209.

[41] G. Globisch, Parmesh – a parallel mesh generator, Parallel Computing 21 (3) (1995) 509–524.

[42] G. Globisch, On an automatically parallel generation technique for tetrahedral meshes, Parallel Computing 21 (12) (1995) 1979–1995.

[43] D. Ibanez, I. Dunn, M. S. Shephard, Hybrid MPI-thread parallelization of adaptive mesh operations, Parallel Computing 52 (C) (2016) 133–143.

[44] G. Gorman, J. Southern, P. Farrell, M. Piggott, G. Rokos, P. Kelly, Hybrid OpenMP/MPI anisotropic mesh smoothing, Procedia Computer Science 9 (2012) 1513 – 1522, proceedings of the International Conference on Computational Science, {ICCS} 2012.

[45] L. Linardakis, N. Chrisochoides, Graded delaunay decoupling method for parallel guaranteed quality planar mesh generation, SIAM Journal on Scientific Computing 30 (4) (2008) 1875–1891.

[46] Ircad Laparoscopic Center, http://www.ircad.fr/softwares/3Dircadb/3Dircadb2 (2013).

[47] J. Richolt, M. Jakab, R. Kikinis, Surgical Planning Laboratory, https://www.spl.harvard.edu/publications/item/view/1953 (2011).

[48] 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.

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