Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Engineering 00 (2016) 000–000

Procedia Engineering

www.elsevier.com/locate/procedia

25th International Meshing Roundtable (IMR25)

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

Daming Feng[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 MPI+Threads parallel Delaunay image-to-mesh conversion algorithm. A nested master-worker communication model for parallel mesh generation is implemented which simultaneously explores process-level parallelization and thread-level parallelization: inter-node communication using MPI and inter-core communication inside one node using thread. 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 is separated. The master thread that initializes the MPI environment is in charge of the inter-node MPI communication while the worker threads of each process are only responsible for the local mesh refinement within the node. We conducted a set of experiments to test the performance of the algorithm on Turing, a distributed memory cluster at Old Dominion University High Performance Computing Center 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 expresses impressive performance potential and is scalable to 30 distributed memory compute nodes with 20 cores each (the maximum number of nodes available for us in the experiments).

Peer-review under responsibility of organizing committee of the 25th International Meshing Roundtable (IMR25).

Keywords: Hybrid Programming; Parallel Mesh Generation; Nested Master-Worker Model; Two-Level Parallelization

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

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

Peer-review under responsibility of organizing committee of the 25th International Meshing Roundtable (IMR25).

of 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. In this paper, we present a three dimensional hybrid MPI+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 [1–4] and the angles are in a reasonable range [5–7]. 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 of the mesh [3,8–10]. The hybrid MPI+Threads parallel mesh generation algorithm proposed in this paper is a Delaunay algorithm that conforms to all of these requirements.

1.2. Contributions

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

- The algorithm proposed in this paper is the first hybrid MPI+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 [3,11].
- 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 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.
- The algorithm exhibits impressive scalability. It is scalable to 30 distributed memory nodes, each of which has 20 cores. This is the maximum number of nodes available for us in the experiments and it already exhibits so far the best performance for image-to-mesh conversion algorithms.

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+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 poor quality elements

and of their adjacent elements from the mesh and the creation of new elements. If the new elements are of poor 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 poor quality elements [1,12]. In addition, the termination does not depend on the order of processing of poor 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 [13,14]. Parallel Delaunay mesh generation methods can be implemented by inserting multiple points simultaneously [11,15,16], and the parallel insertion of points by multiple threads needs to be synchronized.

Blelloch et al. [17] 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 pre-defined point sets. One major limitation of triangulation algorithms [17–20] is that they only triangulate the convex hull of a given set of points and therefore they guarantee neither quality nor fidelity.

Ivanov et al. [21] 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 [22] 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 very beginning if the created separators are not Delaunay-admissible. A parallel three-dimensional unstructured Delaunay mesh generation algorithm [23] 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.

Foteinos and Chrisochoides [11,24] proposed a tightly-coupled Parallel Optimistic Delaunay Mesh generation algorithm (PODM). This approach works well on a NUMA architecture with medium number of cores and exhibits considerable scalability. PODM scales well up to a relatively high core count compared to other tightly-coupled parallel mesh generation algorithms [25]. 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 [26], 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) [16,27] 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 [15]. 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.

We presented a scalable three dimensional parallel Delaunay image-to-mesh conversion algorithm (PDR.PODM) for distributed shared memory architectures [28]. 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. 6b.

A number of other parallel mesh generation algorithms have been published, which are not the Delaunay-based algorithms. De Cougny, Shephard and Ozturan [29] proposed an algorithm in which the parallel mesh construction is based on an underlying octree. Lohner and Cebral [30], and Ito et al. [31] developed parallel advancing front schemes. Globisch [32,33] presented a parallel mesh generator which uses a sequential frontier algorithm. A more detailed review of many more methods appears in [34].

Ibanez et al. [35] 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. These operations show good speedup over threads per process. However, the authors did not show the overall performance (speedup or efficiency) of their algorithm. In addition, they did not mention any information about the input that they used in the experiments. The hybrid algorithm we proposed in this paper take complex multi-labeled 3D image as input directly.

Gorman et al. [36] presented an optimisation based mesh smoothing algorithm for anisotropic mesh adaptivity. The method was parallelised using a hybrid OpenMP/MPI programming method and graph colouring to identify independent sets. The algorithm achieved good scaling performance within a shared memory compute node. However, no experiments were conducted on distributed memory clusters to evaluate the inter-node and overall performance.

Mavriplis [37] described the implementation and performance of a parallel unstructured Navier-Stokes flow solver. The solver is parallelized using a hybrid MPI/OpenMP implementation. The performance of two hybrid OpenMP/MPI communication stragegies was found to be generally inferior to the performance of either method used exclusively on their experimental platform. For their implementation, the exclusively MPI-based communication strategy was demonstrated to exhibit good scalability for large processor counts on various machines.

3. MPI+Threads Implementation

In this section, we present a hybrid MPI + 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 initial 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 initial 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 3.1, 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+Threads parallel mesh generation algorithm.

3.1. 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. Then, we assign tetrahedra to different subregions based on the coordinates of their circumcenters. 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, all subregions in the first level buffer zone can not be refined simultaneously because the point insertion operation might propagate to one or several subregions of its first level buffer zone. Consider a case when two subregions are refined simultaneously. If their first level buffer zones are not disjoint, it may result in a nonconforming mesh in the intersection subregions of their first level buffer zones. Therefore, 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 subre-

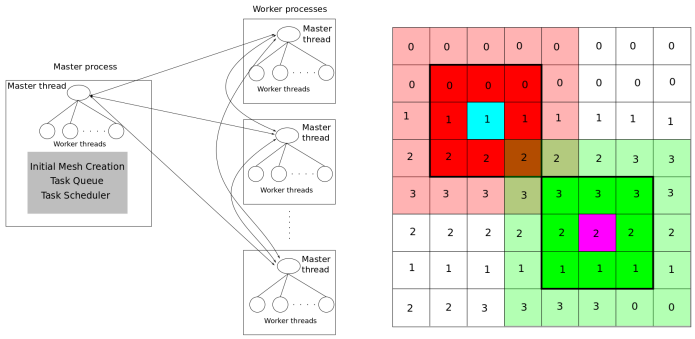


Fig. 1: (a) A diagram that illustrates the design of nested master-worker model. (b) A two dimensional illustration of three-dimensional buffer zones. It is an example with two subregions (the cyan and magenta subregions), which can be refined independently and simultaneously. The dark green and dark 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.

gions under refinement are not overlapping. 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).

3.2. 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 initial 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

```
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 initial mesh;
       g is value of granularity;
Generate an initial 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;
Assign the elements of the initial 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 subregion L and its first level neighbors' submeshes Location information to P_i;
            //Location is an array that contains the process ranks that 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 message from P_i that just finished the refinement work of a subregion
        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
```

Fig. 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 subregion L and its first level neighbors' submeshes Location message from master process P_0;
        //Location is an array that contains the process ranks that hold the submesh of L or 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:                 the number of submeshes received += the number of submeshes P_k holds;
15:                 if the number of submeshes received == the number of submesh needed
16:                     break;
17:             endif
18:         endwhile
19:         Stitch the submesh together and pass it 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
```

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

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 initial mesh, decomposes the initial mesh and initializes the task queue and scheduler (lines 1 to 5 in Fig. 2); (ii) a worker process P_i sends a 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.

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+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. An FDR inifiniband network provides fast cluster communication. We have used two 3D multi-tissued images as inputs in the experiments: (i) the CT abdominal atlas obtained from IRCAD Laparoscopic Center [38], and (ii) the knee atlas obtained from Brigham & Women's Hospital Surgical Planning Laboratory [39]. We performed the experiments on Phi cluster using up to 180 cores and on Ed-Main cluster up to 600 cores (30 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 [40,41].

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 600 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. 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. 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 an 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 define granularity as $g = 1/d$, where d is the depth of the octree. We performed the experiments on Phi cluster with two different data decomposition granularities:

- $g = 1/3$ represents the octree was split to depth 3 with 512 subregions.
- $g = 1/4$ 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 1 and Table 2 show the weak scaling performance of the algorithm for the two input images, abdominal atlas and knee atlas respectively.

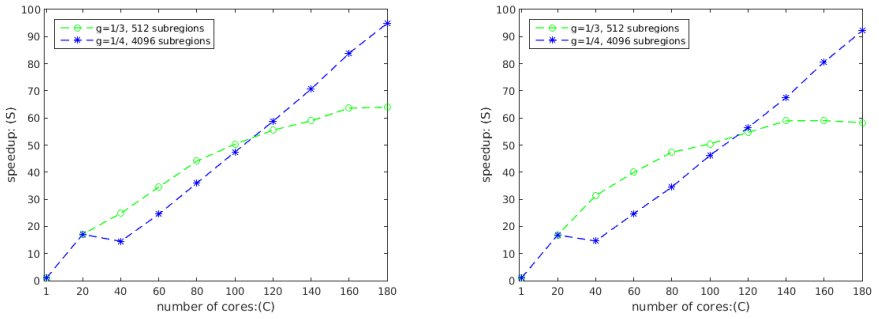
Table 1: Weak scaling perfomance of data decomposition with different granularities. The input is abdominal atlas.

Cores	Elements (millions)	Running Time (s)		million elements/s		Weak Scaling Speedup		Efficiency %	
		g = 1/3	g = 1/4	g = 1/3	g = 1/4	g = 1/3	g = 1/4	g = 1/3	g = 1/4
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 2: Weak scaling perfomance of data decomposition with different granularities. The input is knee atlas.

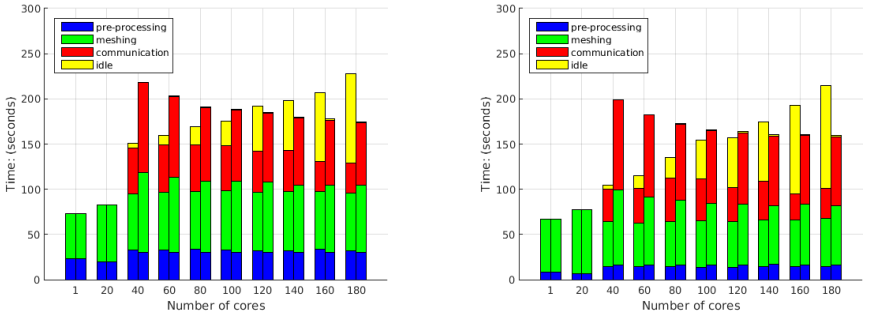
Cores	Elements (million)	Running Time (s)		million elements/s		Weak Scaling Speedup		Efficiency %	
		g = 1/3	g = 1/4	g = 1/3	g = 1/4	g = 1/3	g = 1/4	g = 1/3	g = 1/4
1	6.33	64.27	64.27	0.10	0.10	1.00	1.00	100.00	100.00
20	126.24	76.20	76.20	1.66	1.66	16.83	16.83	84.14	84.14
40	257.01	83.16	179.44	3.09	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 1 and Table 2, the algorithm gets outstanding 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 $g = 1/3$, i.e., the initial mesh and underlying image is partitioned into 512 subregions than that with $g = 1/4$, i.e., the initial 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 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 subregions, introduces more overheads. As demonstrated in Fig. 5a and



(a) (b)

Fig. 4: (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.

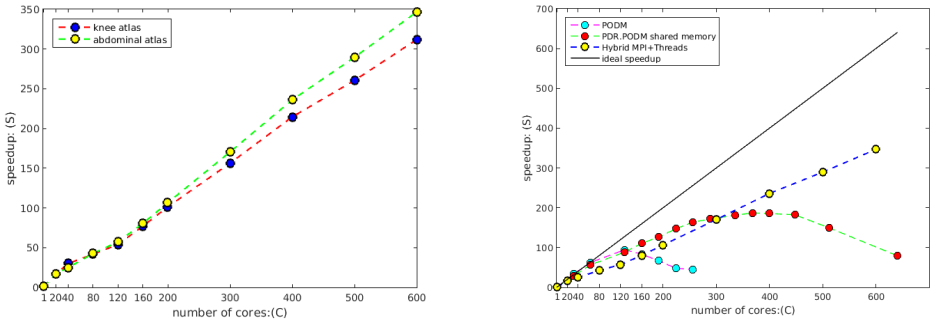


(a) (b)

Fig. 5: The breakdown of the running time of two different granularities. The left bar in each bar graph is the time breakdown with granularity $g = 1/3$ and the right one is the time breakdown with granularity $g = 1/4$. (a) The breakdown of the running time of experiments for abdominal atlas. (b) The breakdown of the running time of experiments for knee atlas.

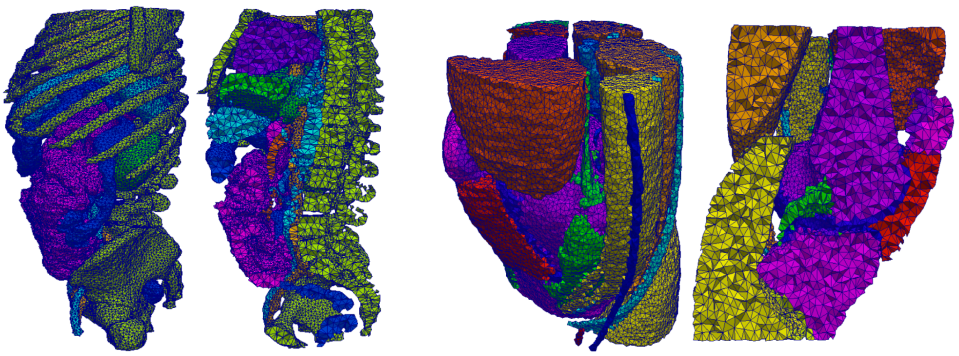
Fig. 5b, the communication overhead (the red part) with granularity $g = 1/4$ (the right bar) is always higher than the communication overhead with granularity $g = 1/3$ (the left bar). The large overhead leads to the speedup of 40 cores (2 nodes) even lower than that of 20 cores with granularity $g = 1/4$ as demonstrated in Fig. 4a and Fig. 4b. The algorithm exhibits better scalability when the granularity is $g = 1/4$ and the number of cores is more than 120 (6 nodes) as shown in Table 1 and Table 2 . We observed that each time we increase the number of cores, the efficiency of experiment with $g = 1/4$ increases while the efficiency with $g = 1/3$ decreases. Take the experimental result of input abdominal as an example, the efficiency with $g = 1/3$ on 40 cores is 62.10% and it decreases to 35.56% for 180 cores. In contrast, the efficiency with $g = 1/4$ on 40 cores is 36.25% and it increases to 52.72% for 180 cores. Fig. 4a and Fig. 4b illustrate the speedup comparison with two different granularities for two input images respectively. For $g = 1/3$ (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 $g = 1/4$ (4096 subregions), the speedup increases almost linearly compared to the speedup with 40 cores.

Fig. 5a and Fig. 5b 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 initial mesh, assigning the elements of the initial 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 perfom 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 subre-



(a) (b)

Fig. 6: (a) The overall weak scaling speedup of the two input images up to 600 cores (30 nodes) on Ed-Main cluster of Turing. (b) The weak scaling speedup comparison of hybrid MPI+Threads implementation with other two shared memory algorithm implementations and ideal speedup.



(a) (b)

Fig. 7: (a) A mesh example created by the algorithm with input abdominal atlas. (b) A mesh example created by the algorithm with input knee atlas.

gion). We calculate the average time of each part for all processes. As demonstrated in Fig. 5a and Fig. 5b, the idle time with granularity $g = 1/3$ 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 granularity $g = 1/4$ (the right bar) is always higher than the communication overhead with granularity $g = 1/3$ (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.

4.3. Overall Weak Scaling Performance

We conducted a set of experiments with the same two input images, abdominal atlas and knee atlas, on Ed-Main cluster of Turing cluster system up to 600 cores (30 nodes) to test the scalability of the algorithm. Based on the analysis of the subsection above, we ran the experiments with the optimal value of granularity, i.e. $g = 1/3$ when the number of nodes is less than or equal to 5 (100 cores) and $g = 1/4$ when the number of nodes is between 6 to 30 (120 to 600 cores). Fig. 6a demonstrates the weak scaling speedup of the two input images up to 600 cores (30 nodes) on Ed-Main cluster of Turing. Fig. 6b compares the speeup of hybrid MPI+Threads implementation with other two shared memory implementations and illustrates that the hybrid MPI+Threads implementation proposed in this paper

shows so far the best scalability. Fig. 7a and Fig. 7b show two Delaunay meshes created by the algorithm with input images abdominal atlas and knee atlas.

5. Conclusion and Future Work

In this paper, we proposed a hybrid MPI+Threads parallel image-to-mesh conversion algorithm. It is the first scalable hybrid MPI+Threads image-to-mesh conversion algorithm on distributed memory clusters with multiple cores. 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 is 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.

A set of experiments have been conducted in High Performance Computing Center of Old Dominion University with its two clusters. The experimental results demonstrated that the hybrid MPI+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 600 cores (30 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. 6a and Fig. 6b. This trend is likely to continue to higher number of cores (nodes) based on the speedup shown in Fig. 6a. 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 funded (in part) by NSF grants CCF-1439079 and CCF-1139864 and by the Richard T. Cheng Endowment. We thank the systems group in the High Performance Computing Center of Old Dominion University for their great help and support. 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] S.-W. Cheng, T. K. Dey, J. Shewchuk, Delaunay Mesh Generation, CRC Press, 2012.
[9] P.-L. George, H. Borouchaki, Delaunay Triangulation and Meshing. Application to Finite Elements, HERMES, 1998.
[10] A. Chernikov, N. Chrisochoides, Generalized insertion region guides for Delaunay mesh refinement, SIAM Journal on Scientific Computing 34 (2012) A1333–A1350.
[11] 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.

[12] L. P. Chew, Guaranteed-quality Delaunay meshing in 3D, in: Proceedings of the 13th ACM Symposium on Computational Geometry, 1997, pp. 391–393.
[13] D. F. Watson, Computing the n-dimensional Delaunay tesselation with application to Voronoi polytopes, Computer Journal 24 (1981) 167–172.
[14] A. Bowyer, Computing Dirichlet tesselations, Computer Journal 24 (1981) 162–166.
[15] A. Chernikov, N. Chrisochoides, Three-dimensional Delaunay refinement for multi-core processors, ACM International Conference on Supercomputing (2008) 214–224.
[16] A. Chernikov, N. Chrisochoides, Parallel guaranteed quality Delaunay uniform mesh refinement, SIAM Journal on Scientific Computing 28 (2006) 1907–1926.
[17] 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.
[18] P. Foteinos, N. Chrisochoides, Dynamic parallel 3D Delaunay triangulation, in: International Meshing Roundtable, 2011, pp. 9–26.
[19] 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.
[20] V. H. Batista, D. L. Millman, S. Pion, J. Singler, Parallel geometric algorithms for multi-core computers, Computational Geometry 43 (2010) 663–677.
[21] 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).
[22] 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.
[23] 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.
[24] 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.
[25] 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.
[26] D. Feng, A. Chernikov, N. Chrisochoides, Two-level locality-aware parallel Delaunay image-to-mesh conversion, Parallel Computing (2016). 10.1016/j.parco.2016.01.007.
[27] 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.
[28] 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.
[29] 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 (1994) 311 – 323.
[30] 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.
[31] Y. Ito, A. Shih, A. Erukala, B. Soni, A. Chernikov, N. Chrisochoides, K. Nakahashi, Generation of unstructured meshes in parallel using an advancing front method, in: International Conference on Numerical Grid Generation in Computational Field Simulations, San Jose, CA, 2005.
[32] G. Globisch, Parmesh – a parallel mesh generator, Parallel Computing 21 (1995) 509–524.
[33] G. Globisch, On an automatically parallel generation technique for tetrahedral meshes, Parallel Computing 21 (1995) 1979–1995.
[34] N. P. Chrisochoides, A Survey of Parallel Mesh Generation Methods, Technical Report BrownSC-2005-09, Brown University, 2005. Also appears as a chapter in Numerical Solution of Partial Differential Equations on Parallel Computers (eds. Are Magnus Bruaset and Aslak Tveito), Springer, 2006.
[35] D. Ibanez, I. Dunn, M. S. Shephard, Hybrid MPI-thread parallelization of adaptive mesh operations, Parallel Computing 52 (2016) 133–143.
[36] 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.
[37] D. J. Mavriplis, Parallel performance investigations of an unstructured mesh navier-stokes solver, International Journal of High Performance Computing Applications 16 (2002) 395 – 407.
[38] Ircad Laparoscopic Center, `http://www.ircad.fr/softwares/3Dircadb/3Dircadb2`, 2013.
[39] J. Richolt, M. Jakab, R. Kikinis, Surgical Planning Laboratory, `https://www.spl.harvard.edu/publications/item/view/1953`, 2011.
[40] 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.
[41] A. Grama, A. Gupta, G. Karypis, V. Kumar, Introduction to Parallel Computing, Addison Wesley, 2003.