Summer 2019

Scalable Parallel Delaunay Image-to-Mesh Conversion for Shared and Distributed Memory Architectures

Daming Feng
Old Dominion University, dmfeng8898@gmail.com

SCALABLE PARALLEL DELAUNAY IMAGE-TO-MESH CONVERSION FOR SHARED AND DISTRIBUTED MEMORY ARCHITECTURES

by

Daming Feng
B.S. June 2006, Harbin University of Science and Technology, China
M.S. June 2009, Soochow University, China

A Dissertation Thesis Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
August 2019

Approved by:

Nikos Chrisochoides (Co-Director)

Andrey Chernikov (Co-Director)

Danella Zhao (Member)

Nail Yamaleev (Member)

ABSTRACT

SCALABLE PARALLEL DELAUNAY IMAGE-TO-MESH CONVERSION FOR SHARED AND DISTRIBUTED MEMORY ARCHITECTURES

Daming Feng
Old Dominion University, 2019
Co-Directors: Dr. Nikos Chrisochoides and Dr. Andrey Chernikov

Mesh generation is an essential component for many engineering applications. The ability to generate meshes in parallel is critical for the scalability of the entire Finite Element Method (FEM) pipeline. However, parallel mesh generation applications belong to the broader class of adaptive and irregular problems, and are among the most complex, challenging, and labor intensive to develop and maintain. In this thesis, we summarize several years of the progress that we made in a novel framework for highly scalable and guaranteed quality mesh generation for finite element analysis in three dimensions. We studied and developed parallel mesh generation algorithms on both shared and distributed memory architectures. In this thesis we present a novel two-level parallel tetrahedral mesh generation framework capable of delivering and sustaining close to 6000 of concurrent work units (cores). We achieve this by leveraging concurrency at two different granularity levels by using a hybrid message passing and multi-threaded execution model which is suitable to the hierarchy of the hardware architecture of the distributed memory clusters. An end-user productivity and scalability study was performed on up to 6000 cores, and indicated very good end-user productivity with about 300 million tets per second and about 3600 weak scaling speedup. Both of these results suggest that: compared to the best previous algorithm, we have seen an improvement of more than 7000 times in performance, measured in terms of speed (elements per second) by using about 180 times more CPUs, for geometries that are by many orders of magnitude more complex.

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest appreciation to my advisors Dr. Nikos Chrisochoides and Dr. Andrey Chernikov for all the ways they helped me for the last six years. Nothing would be possible without their insightful guidance. Their wisdom, patience and encouragement helped me not only on my research but also on my future career. I thank Dr. Danella Zhao, Dr. Yaohang Li and Dr. Nail Yamaleev for their help on my study and my life. I thank all my research collaborators for their helpful discussions.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 MOTIVATION

Scalable, stable, and portable parallel mesh generation algorithms with quality and fidelity guarantees are important for real world (bio-)engineering and medical applications. Their scalability can be measured in terms of the ability of an algorithm to achieve a speedup, proportional to the number of cores. Portability is the capability of an algorithm to be executed on different platforms, with or without only a few minor modifications. Stability refers to the fact that the parallel algorithm can create meshes that retain the same quality and fidelity as the meshes created by the sequential generator that 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 its radius-edge ratio is small [1, 2, 3, 4] and when the angles are in a reasonable range [5, 6, 7]. 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 [8, 9, 10, 3].

Most current mesh generation algorithms are desktop-based, are either sequential or parallel, and have been developed for a small number of cores. The previous parallel unstructured mesh generation and refinement algorithms are not suitable for NUMA DSM architecture supercomputers. These algorithms rely on irregular communication patterns and lack data locality, due to the large number of remote memory accesses. Oliker and Biswas [11] concluded that the performance of unstructured mesh refinement deteriorates, for some cases, on just a 4-core cc-NUMA

architecture. Chowdhury et al. [12] presented multicore-oblivious algorithms and a run-time scheduler for several fundamental problems, including matrix transposition, FFT, sorting, Gaussian elimination, and others; however, mesh generation has not yet been addressed. Mesh generation algorithms, when run on supercomputers, are either conservative in leveraging available concurrency [13, 14, 15] or require the solution of the domain decomposition, which is still an open problem for three-dimensional domains [16, 17]. There is no doubt that the scalability of mesh generation algorithms will continue to be critical for many engineering applications, such as CFD simulations. In order to solve this problem and to create high-quality meshes of the desired resolution, it is necessary to apply a supercomputer-based parallel mesh generation algorithm which scales well on modern non-uniform (i.e., distributed) memory machines. To this end, we propose a three-dimensional locality-aware parallel Delaunay image-to-mesh conversion algorithm that (1) applies a three dimensional *data decomposition* instead of *domain decomposition* according to the circum-centers of tetrahedra, and (2) employs a data locality optimization scheme to reduce the communication overhead caused by a large number of remote memory accesses.

The Parallel Delaunay Refinement algorithm (PDR) [15, 14, 13] is based on a theoretically proven method for managing and scheduling the insertion points. Based on the Delaunay Independence Criterion, PDR breaks the meshing problem of the entire region up into smaller independent subproblems, i.e. it partitions the region into subregions in such a way that the circumcenters of the elements belonging to different subregions can be inserted concurrently. Using a carefully constructed octree, the list of the candidate points is split up into smaller lists that can be processed concurrently with the sequential Delaunay refinement code implemented in a Delaunay refinement software, e.g., TetGen [2]. PDR requires neither the runtime checks nor the geometry decomposition, and it can guarantee the independence of inserted points and thus avoid the evaluation of data dependencies. The Parallel Optimistic Mesh Generation algorithm (PODM) [18] is a tightly-coupled parallel Delaunay mesh generation algorithm. The sequential construction of the initial mesh in PODM only involves the triangulation of the bounding box, i.e., the sequential creation of six tetrahedra. This approach works well on a medium number of cores. It scales well up to a relatively high core count, compared to other tightly-coupled parallel mesh generation algorithms [19]. However, due to extensive remote memory accesses, its

scalability for Distributed Shared Memory machines is limited, depending on the utilization of the target machine from other users. We take advantage of the legacy mesh approaches and propose an algorithm that quickly leverages high parallelism. The algorithm utlizes the aggressive speculative approach employed by PODM and data partitioning offered by PDR to improve data locality and to decrease the communication overhead.

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 parallelization among distributed memory compute nodes and uses thread-based libraries (Pthread or OpenMP) to exploit 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 the trend of writing hybrid parallel programs that involve both process-level and thread-level parallelization. Implementation of the hybrid MPI+Thread parallel mesh generation algorithms is challenging because of the data dependencies and the irregular and unpredictable behavior of mesh refinement. We have proposed 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. It is able to deliver considerable scalability on supercomputer architectures. Preliminary results show that the major overhead of current implementation is the format translation overhead between the distributed memory apaces of parallel processes. This is a commonly reported problem when importing or migrating a shared memory application to distributed memory architectures. We are exploring a method that helps to reduce the overhead. The idea is that we introduce a customized layer on top of the current shared memory implementation. This customized layer allows a thread to transparently get either pointer mesh structure (for mesh refinement) or index mesh structure (for data migration), based on the operations that it will perform. In addition, most of the distributed memory clusters consist of heterogeneous nodes (nodes with different number or type of cores), which makes the computing ability of each node different from the other nodes, depending on the number and type of cores that a node has. Developing parallel applications and software that efficiently utilizes such heterogeneous and hierarchical computing and communication resources is a challenging and open question [20]. Furthermore, a supercomputer is usually open to the public or to a certain community, and it is shared by all authorized users,

which means that many jobs, from different users, may run simultaneously. Due to management and scheduling, a few cores on a node may have already been occupied by other users' jobs in practice. Therefore, the parallel mesh generation algorithm that we have proposed is able to efficiently work on heterogeneous supercomputers while scaling it towards exascale, and while maintaining correctness.

1.2 CONTRIBUTIONS

In summary, the contributions of this dissertation thesis are as follows:

- We propose a three dimensional two-level Locality-Aware Parallel Delaunay image-to-mesh conversion algorithm (LAPD) [21]. The algorithm exploits two levels of parallelism at different granularities: coarse-grain parallelism at the region level (which is mapped to a node with multiple cores), and medium-grain parallelism at the cavity level (which is mapped to a single core). We employ a data locality-aware mesh refinement process to reduce the latency caused by the remote memory access. We evaluated LAPD on Blacklight, a cache-coherent NUMA distributed shared memory (DSM) machine in the Pittsburgh Supercomputing Center, and we observed a weak scaling efficiency of almost 70% for roughly 200 cores, compared to only 30% for the previous algorithm PODM.

- A parallel mesh generation algorithm for distributed shared memory architecture which takes advantage of two legacy approaches, i.e., the Parallel Optimistic Delaunay Mesh generation algorithm (PODM) [18] and PDR [15, 14, 13]. We present a scalable three-dimensional hybrid parallel Delaunay image-to-mesh conversion algorithm (PDR.PODM) for distributed shared memory architectures [22]. PDR.PODM is able to explore parallelism early in the mesh generation process because of the aggressive speculative approach employed by PODM. In addition, it decreases the communication overhead and it improves data locality by making use of a data partitioning scheme offered by PDR. PDR.PODM supports fully functional volume grading by creating elements of varying size. Small elements are created near the boundary or inside the critical regions in order to capture the fine features, while big elements are created in the rest of the mesh. We tested PDR.PODM on Blacklight, a distributed shared memory (DSM) machine in the Pittsburgh Supercomputing Center. For

the uniform mesh generation, we observed a weak scaling speedup of 163.8 and above for up to 256 cores, as opposed to PODM, whose weak scaling speedup is only 44.7 on 256 cores. The end result is that we can generate 18 million elements per second, as opposed to the 14 million per second that we were able to generate in our earlier work. PDR.PODM scales well on uniform refinement cases running on DSM supercomputers. The varying size version sharply reduces the number of elements, compared to the uniform version, and thus, reduces the time required to generate the mesh, while keeping the same fidelity.

- A scalable three dimensional hybrid MPI+Threads parallel Delaunay image-to-mesh conversion algorithm on distributed memory clusters [23], which simultaneously explores process-level parallelization and thread-level parallelization: inter-node parallelization using MPI and inter-core parallelization inside one node using threads. 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 threads. In order to overlap the communication (task request and data movement) and computation (parallel mesh refinement), the inter-node MPI communication and the 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 the Old Dominion University High Performance Computing Center, and we 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 up to 6000 cores (the maximum number of nodes available to us in the experiments), and indicated very good end-user productivity with about 300 million tets per second and about 3600 weak scaling speedup.

CHAPTER 2

BACKGROUND

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, 24]. 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 [25, 26]. Parallel Delaunay mesh generation methods can be implemented by inserting multiple points simultaneously [13, 14, 18], and the parallel insertion of points by multiple threads needs to be synchronized.

Blelloch et al. [27] 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 [27, 28, 29, 30] 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. [31] 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 [32] 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 [33] 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 [34, 18] 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 [19]. 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.

Parallel Delaunay Refinement (PDR) [14, 15] 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 [13]. Using a carefully constructed spatial decomposition tree, the list of the candidate points is split up into smaller lists that can be processed concurrently. The construction of an initial mesh is the basis and starting point for the subsequent parallel procedure. There is a trade-off between the available concurrency and the sequential overhead: the initial mesh is required to be sufficiently dense to guarantee enough concurrency for the subsequent parallel refinement step; however, the construction of such a dense mesh prolongs the low-concurrency part of the computation.

A number of other parallel mesh generation algorithms have been published, which are not the Delaunay-based algorithms. De Cougny, Shephard and Ozturan [35] proposed an algorithm in which the parallel mesh construction is based on an underlying octree. Lohner and Cebral [36], and Ito et al. [37] developed parallel advancing front schemes. Globisch [38, 39] presented a parallel mesh generator which uses a sequential frontier algorithm. A more detailed review of many more methods appears in [40].

Ibanez et al. [41] 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. [42] 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.

Dhairya Malhotra and George Biros [43] presented a Fast Multipole Method (FMM) for computing volume potentials and use them to construct spatially adaptive solvers. They used space-filling curves, locally essential trees and a hypercube-like communication scheme for distributed memory parallelization. They discussed the distributed-memory tree construction and explained the partitioning of the domain across processes. They also discussed the parallel 2:1 balance algorithm on this distributed octree. For a distributed octree, the interacting source octants may belong to different processes. Therefore, A local essential tree is built by communicating the ghost octants needed by a process for the downward pass.

Lashuk et al. [44] described a parallel fast multipole method (FMM) for highly nonuniform distributions of particles. The method employs both distributed memory parallelism (via MPI) and shared memory parallelism (via OpenMP and GPU acceleration) to rapidly evaluate two-body nonoscillatory potentials in three dimensions on heterogeneous high performance computing architectures. The communication cost for the hypercube communication scheme is discussed.

Hu et al. [45] proposed a scalable fast multipole methods on distributed heterogeneous architectures. The FMM is a divide-and-conquer algorithm that performs a fast N-body sum using a spatial decomposition and is often used in a time-stepping

or iterative loop. The implementation of the Fast Multipole Method (FMM) on a computing node with a heterogeneous CPU-GPU architecture with multicore CPU(s) and one or more GPU accelerators, as well as on an interconnected cluster of such nodes.

Ibanez et al. [41] 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. However, the authors did not show the overall performance and the scalability (speedup or efficiency) of their algorithm.

Gorman et al. [42] 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.

Dreher and Grauer [46] described Racoon, a framework that offers a grid-based environment for the mesh-adaptive solution of conservative systems and related systems. Racoon is a hybrid strategy of multi-threading and inter-process communication through MPI and POSIX-multithreading, which exploits both shared and distributed memory architectures parallelization. It supports mesh refinement, regridding, load balancing and distribution. During mesh refinement, bands of ghost cells are created and updated around the individual grid blocks.

Remacle et al. [47] proposed the Parallel Algorithm Oriented Mesh Database (PAOMD). PAMOD is the extension of the Algorithm Oriented Mesh Database (AOMD) in order to support distributed meshes. It provides a general parallel mesh management framework in which mesh representation can be adapted to different types of applications. Each partition is assigned to a processor, and the local mesh is represented by a serial AOMD mesh. Ghosting are not supported. Instead, mesh entities that are classified on partition boundaries must exist in the parallel data structure and may be shared with other partitions.

The Mesh-Oriented datABase (MOAB) [48] is a component for representing and

evaluating mesh data. MOAB can store structured and unstructured mesh, consisting of elements in the finite element polygons and polyhedra. It supports common parallel mesh operations like parallel import and export and general sending and receiving of mesh and metadata between processors.

Rodriguez et al. [49] described ViennaMesh, a parallel mesh generation approach for multi-core and distributed computing environments based on the generic meshing library ViennaMesh and on the Advancing Front mesh generation algorithm. The approach is based on the Advancing Front meshing technique, in which the algorithm preserves the input hull mesh during the volume meshing process. Therefore, the communication overhead is minimized, as interface changes do not have to be communicated through the parallelized meshing environment. The ViennaMesh library offers a unified interface to various mesh related tools.

PMSH [50] is a parallel mesh generation algorithm is based on Netgen. The mesh generation algorithm proceeds in five main stages: (i) generation of a coarse volume mesh, (ii) partitioning of the coarse mesh to get submeshes, each of which will be processed by a processor, (iii) extraction and refinement of coarse surface submeshes to produce fine surface submeshes, (iv) remeshing of each fine surface submesh to get the final fine volume mesh, and (v) matching of distributed duplicate partition boundary vertices followed by global vertex numbering.

Burstedde et al. [51] uses ghost layers for parallel adaptive mesh refinement and coarsening in ALPS (Adaptive Large-scale Parallel Simulations) which is a library for parallel octree-based dynamic mesh adaptivity and redistribution. One processor maintains entities from other parts located on other processors to avoid excessive communication with other parts. These read-only copies are often referred as ghost entities. Ghost entities are read-only copies of remote part entities.

CHAPTER 3

PARALLEL DELAUNAY IMAGE-TO-MESH CONVERSION ALGORITHMS FOR SHARED MEMORY ARCHITECTURES

In this chapter, we analyze the remote memory access on NUMA shared memory architecure and proposed the LAPD [21] and PDR.PODM [22] algorithms.

3.1 REMOTE MEMORY ACCESS ON NUMA ARCHITECTURE

In distributed shared memory (DSM) systems, memory is physically distributed, while it is accessible to and shared by all cores. However, a memory block is physically located at various distances (hops) from the cores. As a result, the memory access times vary, and this depends on the distances (hops) from a core to a memory block. When the parallel applications are running on such NUMA machines, a thread running on a core might access its own local memory or its non-local (remote) memory (memory local to another node). The experimental platform, Blacklight [52], is a cc-NUMA shared-memory system that consists of 256 blades. Each blade holds two Intel Xeon X7560 (Nehalem) eight-core CPUs, for a total of 4096 cores across the whole machine. The 16 cores on each blade share 128 Gbytes of local memory. The dashline rectangle in Fig. 1. shows one individual rack unit (IRU) of 16 blades and 256 cores on Blacklight. Each rectangle represents a blade. The 16-port NL5 router is used to connect blades that are located internally to each IRU. Each of these routers connects to eight compute blades within the IRU. The remaining eight ports of the internal router are used to connect to other NL5 router blades [53]. The total 4096 cores have 32 TB of memory. Each thread that runs on a core in PODM maintains its own Poor Element List (PEL) [18]. The PEL contains the poor elements that violate the quality criteria [3] and are assigned to be processed by this thread. If PEL_i is not empty, thread T_i will get the first element in the list, compute the cavity, delete the tetrahedra in the cavity, and create new tetrahedra

according to the Bowyer-Watson kernel [25, 26]. Additionally, a global load balancing list stores the IDs of threads whose poor element list is empty. When a thread T_i runs out of work (when its PEL is empty), it will push back its ID to the global load balancing list and start to wait. If another thread T_j creates new elements, it will check whether the load balancing list is empty. If the list is empty, it means all of the other threads are busy. Thread T_j adds the newly created elements to its own PEL. If the list is not empty, thread T_j adds the newly created elements to the PEL of the first thread. Suppose it is thread T_i, in the load balancing list. The new poor elements in thread $T_i's$ poor element list created by thread T_j still reside in thread $T_j's$ local memory. When T_i needs to refine these poor elements, it needs to access thread $T_j's$ local memory to fetch them. In the case that these two threads run on cores that belong to the same blade, the ask-for-work operation between them is a local memory access, since all of the cores in the same blade share memory, without any switches. However, If thread T_i and thread T_j are not in the same blade, one thread needs to fetch a poor element from the local memory of another thread. This leads to a remote memory access. Moreover, if they are not in the same IRU, the time latency is much longer, because of the increase in hops (about a 2000 cycle latency penalty on Blacklight for each hop) and the traffic contention. Fig. 1. shows one possible case in which we reserve 64 cores on Blacklight. It illustrates that the maximum number of hops between two blades of the same IRU is three, and that of different IRUs is five.

As with most DSM supercomputers, the experimental platform, Blacklight, is shared by many users. The scheduling and the reservation of cores (blades) is managed by the system. A user has no mechanism to decide which blades he can get to run his job. The system determines which blades are given to the user's job based on the available blades. In most cases, the job will get several non-adjacent blades, among all of the blades in the system. For a data communication intensive application, such as parallel mesh generation, performance will suffer on high core counts because of a large number of remote memory accesses. The performance of PODM was indeed poor when the allocated cores (>128) were non-consecutive. The ideal case is to make a thread finish all of its work on its local memory. However, this is almost impossible for unstructured parallel mesh generation because of the irregular and unpredictable communication during run-time. In this paper, we describe a two-level locality-aware parallel Delaunay mesh refinement algorithm, LAPD. It

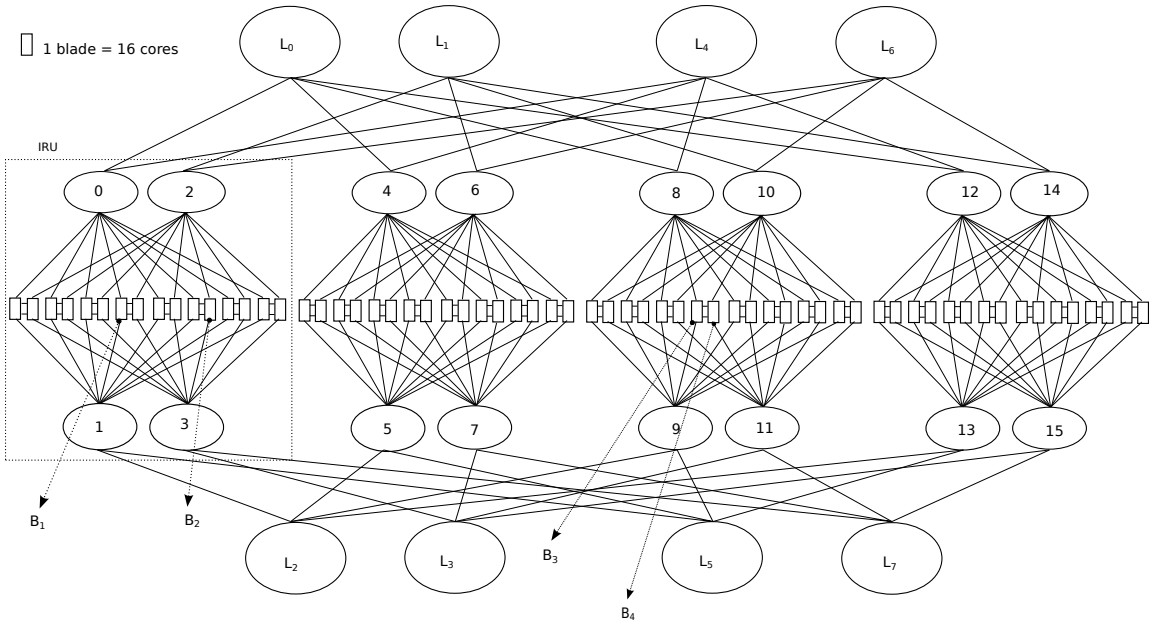


FIG. 1.: One possible allocation of 4 blades (64 cores) on Blacklight. The dashline rectangle represents the individual rack unit (IRU). Each IRU includes 16 blades and each blade has 16 cores that share 128 GB local memory.

divides the image into subregions, and each subregion is refined by a parallel mesh generator (PODM, in our case). In each of the subregions, a thread of a PODM mesh generator has the flexibility to communicate with any other thread in the same PODM mesh generator, in order to maximize the concurrency of this PODM mesh generator. The communication between different PODM mesh generators is confined and only happens when the poor element is near the partition boundary. This two-level locality-aware parallel strategy eliminates a large number of remote memory accesses, as well as alleviates the pressure of the network routers caused by the intensive communication among threads.

3.2 LOCALITY-AWARE PARALLEL DELAUNAY MESH GENERATION

In this section, we present the Locality-Aware Parallel Delaunay (LAPD) mesh generation algorithm in details.

3.2.1 TWO-LEVEL PARALLEL MESH REFINEMENT

The algorithm explores concurrency at two levels of granularity: coarse-grain parallelism at the subregion level (which is mapped to a node with multiple cores) and medium-grain parallelism at the cavity level (which is mapped to a single core).

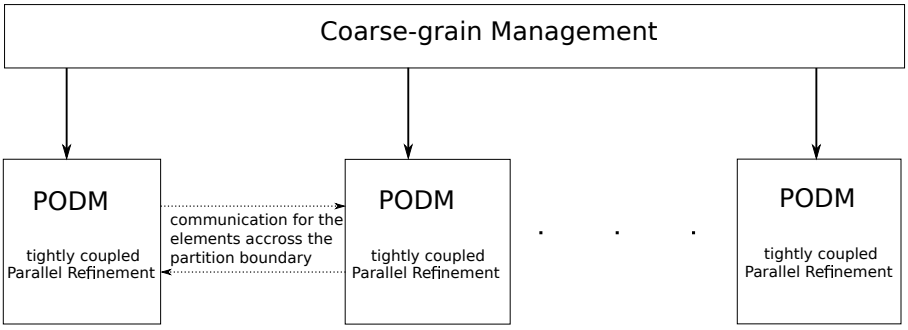


FIG. 2.: A diagram that illustrates the design of the two-level parallel Delaunay mesh generation algorithm.

In the coarse-grain parallel implementation, the input image (domain) is decomposed into subregions and each node is responsible for the refinement work of one subregion. The communication between two adjacent subregions happens only near the partition boundary. In the medium-grain parallel implementation, the threads running in the cores of a node follow the refinement rules of PODM to insert or delete multiple points in parallel. The work load balancing among the threads of each node is performed by the load balancing scheme of the PODM mesh generator. Fig. 2. illustrates a diagram of the two-level parallel mesh generation design.

In the two-level parallel locality-aware refinement algorithm, we combine two different communication types in two granularity levels, with partially coupled communication at the node level and tightly coupled communication at the core level. This algorithm quickly leverages high concurrency due to the aggressive speculative approach employed by the tightly coupled PODM of each subregion, uses the partially coupled communication to ensure conformity of elements across the partition boundary, and employs data partitioning to improve data locality gradually during the mesh refinement procedure. Fig. 3. depicts the pseudo-code of the algorithm.

3.2.2 DATA LOCALITY-AWARE IMPLEMENTATION

For illustration purposes, in this subsection, we exhibit a simplified two dimensional example that shows the locality-aware property of the LAPD algorithm. The parameter r_i is the circumradius upper bound that we use to control the size and the

```
1  Algorithm: LAPD (I,r,n,b)
   Input  : I is the input segmented image,
            r is the circum-radius upper bounds vector of length n
            /* r_i in the vector r defines the circum-radius upper bounds of elements created in step i.  */
       2    b is the number of blades.
   Output: A Delaunay Mesh M that is conforming to the size upper bound r_n.
3  Generate Initial Mesh that is conforming to the size upper bound r_1;
4  for i = 2 to n do
5       M_i = StepMesh(r_i, i, M_{i-1}, b);
6  end

7  Algorithm: StepMesh (r̄,i,M, b)
   Input  : i is the current step
            r̄ is the current size upper bound,
            M is the mesh created in the previous step,
            b is the number of blades.
   Output: A Delaunay Mesh M' that is conforming to the current size upper bound r̄.
8  Divide the 2^{i-2} subregions of the previous step into 2^{i-1} subregions by the bisection plane of the previous subregion along
   one dimension;
9  Assign the elements of M to subregion PELs based on the circumcenter coordinates;
10 Divide the b blades into 2^{i-1} nodes based on their physical locations;
11 Assign each node a PEL of a subregion;
12 for each node do
13      SM = GenerateMesh(r̄, PEL);
14 end

15 Algorithm: GenerateMesh (r̄,PEL)
   Input  : r̄ is the size upper bound of the current step,
            PEL is the poor element list that need to be refined.
   Output: A submesh SM of a subregion.
16 while PEL! = NULL do
17      Get the first poor element e in PEL;
18      Refine e and create new elements;
19      for each newly created element e' do
20           if e' is a poor element according to the size upper bound r̄ and fidelity bounds then
21                add e' to PEL;
22           else
23                add e' to output SM of this subregion;
24           end
25      end
26 end
```

FIG. 3.: Pseudocode of the locality-aware parallel Delaunay mesh algorithm.

number of elements that are created in step i. The smaller r_i is, the more elements are created. Empirically, we found that setting $r_i = r_{i-1}/2$ leads to the best balance between available concurrency and overheads.

In the beginning of the mesh process, an appropriate isosurface is recovered and an initial tetrahedral mesh is constructed and refined in parallel using the PODM mesh generator, until all of the elements satisfy the user-defined size upper bound r_1 determined by the theory that we developed in the previous work [14, 15, 13]. The elements in this initial mesh are distributed among all memory blocks, since PODM uses all of the available threads to jump-start the computation with maximum concurrency. Fig. 4.a exhibits a two-dimensional illustration of an initial mesh and its distribution in memory after the first step. We use four different colors to distinguish

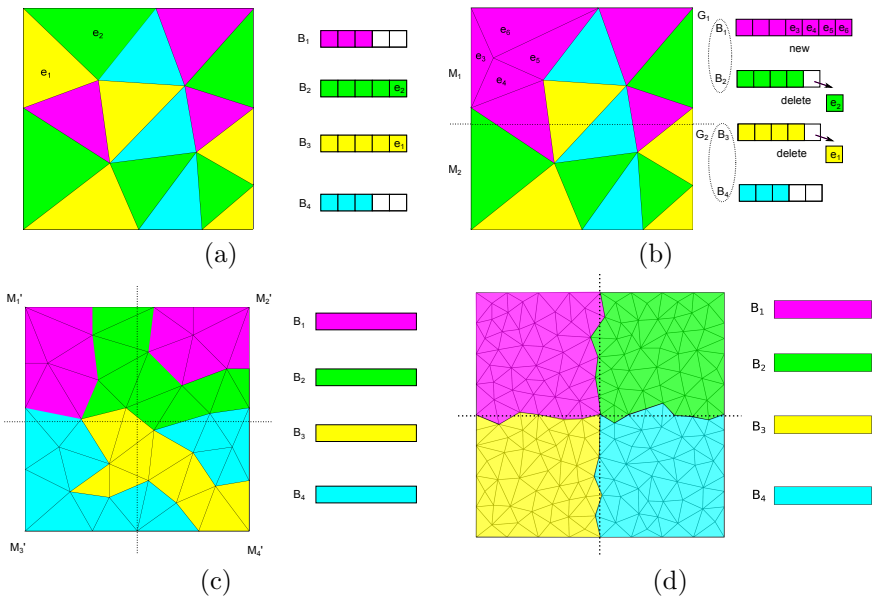


FIG. 4.: A simplified two dimensional illustration of LAPD method.

the elements in the different memory of the four blades. The left subfigure shows the mesh, and the right subfigure illustrates the element distribution on four different memory blocks. The different colors represent different memory blocks in which elements are stored.

In the second step, the whole region (image) is divided into two subregions, and the initial mesh is divided into two sub-meshes, based on the coordinates of centers of element circum-spheres. A PODM mesh generator refines the sub-mesh of each subregion. The threads of the PODM mesh generator will work in the subregion and will refine the sub-mesh in parallel to get the mesh that is conforming to size upper bound r_2, using similar criteria to that used before. As illustrated in Fig. 4.b, the whole region, i.e., the bounding box and the image, is divided by its bisection line (the bisection plane in a three-dimensional space) along one dimension, and the initial mesh is divided into two sub-meshes, M_1 and M_2, according to the coordinates of the circum-centers of elements in the mesh. In this example, there are two nodes in this step. We assume that node G_1 contains blades B_1 and B_2, while node G_2 contains blades B_3 and B_4. Node G_1 is only responsible for refining the elements belonging to the top half subregion, and node G_2 is responsible for the elements in the bottom half subregion.

During the refinement, B_1 gets the poor element e_1 that was stored in $B_3's$ local memory and adds it to its own poor element list. Element e_1 is triangulated and deleted. To keep the Delaunay property, element e_2 is part of the cavity and is also triangulated. The new elements e_3, e_4, e_5 and e_6 will be stored in the local memory of B_1, because the new elements will be stored in the local memory of the blade that created the elements, i.e. B_1, as shown in Fig. 4.b. The same process is used to refine the other elements in the mesh. The elements of the first subregion, i.e. sub-mesh M_1, will be refined by node G_1, and the newly created elements will be in the local memory of G_1, i.e. in the local memory of the blades of this node. Those elements of M_2 will be refined by G_2, and the newly created elements will be in the local memory of G_2.

Therefore, after the second step, a new mesh that conforms to size upper bound r_2 is generated. Fig. 4.c shows the mesh and its storage configuration in memory. The elements that are on the top half subregion were created by blades B_1 and B_2 of node G_1 and are stored in the local memory of B_1 and B_2, while those that are on the bottom half subregion were created by blades B_3 and B_4 of node G_2 and are stored in the local memory of B_3 and B_4.

In the third step, the whole region is divided into four subregions, as shown in Fig. 4.c. Each of the four sub-meshes will be assigned to one node to refine. Each blade in node G_1 of the second step, i.e. B_1 or B_2, can only be assigned a subregion that is in the top half part, because the elements (the green and pink ones) of this part were stored in the local memory of B_1 and B_2. Similarly, B_3 or B_4 can only be assigned a subregion that is in the bottom half part, because the elements (the green and pink ones) of this part were stored in the local memory of B_3 and B_4. Under this assignment, during the refinement of the third step, blades B_1 and B_2 can only refine the elements stored in either $B_1's$ or $B_2's$ local memory, while blades B_3 and B_4 can only refine the elements stored in either $B_3's$ or $B_4's$ local memory. The communication happens between subegions if, and only if, a thread in one subregion wants to refine the elements that are adjacent to the elements of the other subregion along the partition boundary.

After the third step, a mesh is created and its storage configuration in memory is shown in Fig. 4.d. The newly created elements of each subregion will be only stored in the local memory of the blade that is responsible for refining the subregion.

Finally, we go on to the next step. In this step, each blade will only need to

access its own local memory in order to finish the refinement work, except for the elements on the boundary shown in Fig. 4.d. The refinement continues until all of the elements in the mesh are Delaunay and are conforming to the target size upper bound r_t.

In this data locality-aware mesh refinement algorithm, we reduce the number of remote memory accesses by controlling the inter-node communication in each step. A blade can only ask for work or give work to the blades that are in the same node; B_1 can only communicate with B_2 in the second step shown in Fig. 4.b because they are in the same node and are physically close to each other. In the last step, there will be only a few remote memory accesses (when the elements are near the partition boundary between subregions) because most of the poor elements that one blade needs to refine are in the local memory of this blade. In this subsection, we describe an over-decomposed block-based partition approach to alleviate the load balancing problem.

The over-decomposed block-based partition proceeds in three main stages: (1) over-decompose the bounding box that overlaps the input image and the coarse mesh (i.e., the number of blocks is much greater than the number of cores), (2) mark the blocks that contain elements as active blocks, (3) partition the active blocks into N subregions to make each of the subregions contain a roughly equal number of blocks, where N is the number of basic computing nodes that share the local memory (for example, a blade of 16 cores that share 128GB memory is a basic computing node on Blacklight). In this case, each subregion ends up with a roughly equal number of elements and the refinement is well balanced among the multicore PODM mesh generators working on each subregion.

Fig. 5. gives a demonstration of how this static work load partition strategy makes LAPD ensure both data locality and load balance. We use the static block-based partition approach to roughly balance the load among different subregions at each step. For each subregion, we utilize the load balancing approach of PODM. Taking advantage of this two-level load balancing scheme, LAPD ensures the data locality during the parallel refinement procedure and does not suffer from severe load imbalance for the complex input images. The case of the dynamic load balancing problem in the context of adaptive mesh refinement is out of the scope of this paper. We developed a run time system [54] to address this problem.

3.2.3 EXPERIMENTAL EVALUATION

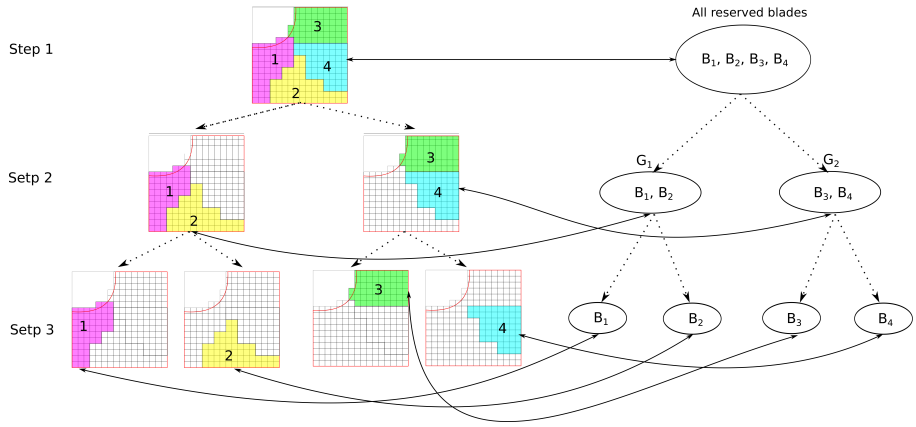


FIG. 5.: Block partition and computing node mapping illustration of LAPD

In this section, we present the weak scaling performance of the two-level locality-aware parallel mesh generation algorithm, LAPD, as well as that of PODM, for comparison. The input images that we used in the experiments are the 3D CT abdominal atlas obtained from IRCAD Laparoscopic Center [55] and the 3D Brain atlas [56]. We tested both LAPD and PODM on Blacklight, using up to 192 cores. See Table 4. and Table 2. for detailed results of both approaches.

Performance Evaluation Metrics

We used the following metrics to evaluate and study the performance of parallel mesh generation algorithms [57, 58]. **Speedup *S*:** The ratio of the serial execution time of the fastest known serial algorithm (T_s) to the parallel execution time of the parallel algorithm (T_p). **Efficiency *E*:** The ratio of speedup (S) to the number of cores (p): $E = S/p = T_s/(pT_p)$.

In the weak scaling case, the number of elements per thread (we used one thread per core) remained approximately constant. In other words, the problem size (i.e., the number of elements created) was increased proportionally to the number of threads. The number of elements generated equalled approximately three million on a single Blacklight core. The problem size gradually increased from three million to 559 million tetrahedra for 1 to 192 cores on Blacklight. In practice, because of the irregular nature of the unstructured mesh, it was impossible to control the problem size (the number of elements) exactly, while the number of cores was increased by

p times, so we used an alternative definition of speedup which is more precise for a parallel mesh generation algorithm.

We measured the number of elements (tetrahedra) generated every second in the experiment. Let us denote by $Elements(p)$ and $Time(p)$ the number of tetrahedra generated and the meshing time respectively, where p is the number of threads. Then, we can use the following equation to compute the speedup:

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

In equation (2), $elements\_per\_sec(p)$ represents the number of elements (tetrahedra) created per second using p threads (cores); $elements\_per\_sec(1)$ represents the number of elements (tetrahedra) created per second by the best sequential mesh generation algorithm. Since the PODM maintains the best single-threaded performance compared to other sequential three dimensional mesh generation software, such as Tetgen [59] and CGAL [4], we used the sequential rate of PODM, i.e. the number of elements generated per second by single-threaded PODM, as a reference when we computed the speedup and presented the performance of the multi-threaded LAPD.

Experimental Results and Analysis

Table 4. shows the weak scaling performance of PODM and the two-level locality-aware parallel mesh generation algorithm, LAPD, respectively. The input image is the 3D abdominal atlas. We observed that both PODM and LAPD performed well on Blacklight for up to 64 cores. However, the speed-up of PODM deteriorated significantly for 128 or more cores. In fact, the speed-up on 144 cores was about 80.8, which was smaller than that on 128 cores (about 94.5), and it was down to only 62.9 and 54.8 for 176 and 192 cores, respectively. The blue line in Fig. 10.a shows this speed-up clearly.

The main reason for this performance deterioration of PODM is the increase of communication time due to the large number of remote memory accesses and due to the congested network. In PODM, each thread has the flexibility to communicate with any other thread during the refinement. This approach works well on a medium number of cores (threads) and exhibits impressive scalability. However, when the core count is beyond a certain number (128 on Blacklight, for example), the communication overhead becomes the bottleneck that hinders the performance of PODM,

TABLE 1.: Weak scaling performance comparison of PODM and LAPD. The input image is a 3D abdominal atlas. The number of elements remains approximately linear with respect to the number of threads in both PODM and LAPD.

(a) Weak scaling performance of PODM

Threads	1	32	64	128	144	160	176	192
Elements (millions)	3.09	96.96	186.80	374.09	419.65	467.01	513.81	559.20
Time(s)	27.78	26.07	27.02	35.97	47.24	57.24	74.20	92.77
Elements per second (millions)	0.11	3.75	6.91	10.41	8.89	8.14	6.92	6.03
Speedup	1.0	34.1	62.8	94.5	80.8	74.0	62.9	54.8
Efficiency	1.00	1.06	0.98	0.74	0.56	0.46	0.36	0.29

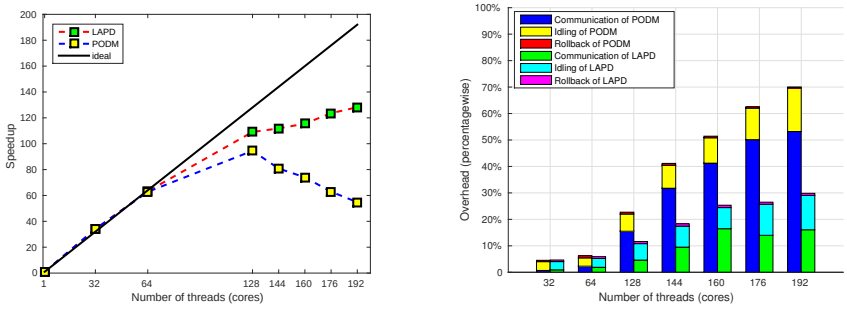
(b) Weak scaling performance of LAPD

Threads	1	32	64	128	144	160	176	192
Elements (millions)	3.09	96.96	186.80	374.09	419.65	467.01	513.81	559.20
Time(s)	27.81	26.18	26.90	31.26	34.04	36.27	37.82	39.63
Elements per second (millions)	0.11	3.73	6.95	12.01	12.30	12.70	13.61	14.12
Speedup	1.0	33.9	63.2	109.1	111.8	115.5	123.6	128.2
Efficiency	1.00	1.06	0.99	0.85	0.78	0.72	0.70	0.67

because it exerts too much pressure on the network routers.

In the LAPD algorithm, we confine the tightly coupled communication among the cores in each node, i.e. the threads running on the cores of a node have the flexibility to communicate with each other in order to maximize the concurrency of that node. As we explain below, the communication between the two nodes happens when one node needs to refine an element across the partition boundary between these two subregions and at least one of element is in the local memory of the other node during the cavity expansion. In order to guarantee the conformity of the created mesh, this inter-node communication is necessary and unavoidable. In other words, the inter-node communication is forbidden, unless it is unavoidable.

Table 1.b shows that the speed-up and efficiency of LAPD on 128 Blacklight cores is 109.1 and 85% respectively, which is better than those of PODM, 94.5 and 74% respectively. The red line in Fig. 10.a illustrates that each time we increase the number of cores by 16 (a blade), the approach gains some speedup increase, for up to 192 cores. The efficiencies of LAPD on the 176 and 192 cores are 1.9 and 2.3



(a) (b)

FIG. 6.: Speedup and overhead comparison of PODM and LAPD for 3D abdominal image. (a) Weak scaling speedup of PODM and LAPD upto to 192 cores on Blacklight. Three million tetrahedra are created by each thread running on a core. The black line depicts the ideal linear speedup. The red dash line with green markers shows the speedup of LAPD and the blue one with yellow markers is the speedup of PODM. (b) Overhead percentage of PODM and LAPD. The left stacked bar shows the overhead percentage of PODM and the right one shows the overhead of LAPD.

times better than those of PODM. The previous image-to-mesh conversion algorithm, PODM, scales well up to only 128 cores on Blacklight. The locality-aware approach, LAPD, scales well up to 192 cores on Blacklight.

The overhead time of both PODM and LAPD mainly consists of three parts:

- Rollback overhead time: this is the time that threads spend on completing partial cavity expansions before they detect a conflict with some other thread and discard the expansion.
- Idling time overhead: this is the total time that threads have no poor elements to refine, and they are idling and waiting for more work from other threads.
- Communication overhead time: this is the total time that threads spend on fetching elements that are not in the local memory.

See Fig. 6.b for the details of the overhead time percentages of PODM and LAPD. The figure shows that the total overhead time of LAPD is less than that of PODM for all numbers of cores, from 64 to 192. We observe that, for core counts beyond 128, the communication overhead time (the blue bar in Fig. 6.b) contributes the main part of the total overhead time. The communication overhead time takes a higher percentage of the total overhead time with the increase in the number of cores. The percentage

TABLE 2.: Weak scaling performance comparison of PODM and LAPD. The input image is a 3D Brain atlas. The number of elements remains approximately linear with respect to the number of threads in both PODM and LAPD.

(a) Weak scaling performance of PODM

Threads	1	32	64	128	144	160	176	192
Elements (millions)	2.01	64.32	128.61	257.30	289.46	321.72	353.78	385.94
Time(s)	18.29	17.73	18.72	22.88	29.91	39.73	49.69	57.78
Elements per second (millions)	0.11	3.62	6.87	11.24	9.68	8.10	7.12	6.68
Speedup	1.0	33.0	62.5	102.2	87.97	73.60	64.72	60.72
Efficiency	1.00	1.03	0.98	0.80	0.61	0.46	0.37	0.32

(b) Weak scaling performance of LAPD

Threads	1	32	64	128	144	160	176	192
Elements (millions)	2.01	64.32	128.61	257.30	289.46	321.72	353.78	385.94
Time(s)	18.29	18.48	18.83	21.66	24.01	25.18	27.08	28.48
Elements per second (millions)	0.11	3.48	6.83	11.88	12.06	12.77	13.06	13.55
Speedup	1.0	31.6	62.1	109.6	111.8	116.1	118.8	123.2
Efficiency	1.00	0.99	0.97	0.84	0.76	0.73	0.68	0.65

of communication overhead on 144 cores is about 32%, while this number is already increasing to 50% for 176 cores and to 53% for 192 cores. Since the problem size increases linearly with respect to the number of threads (cores), the communication traffic per network router increases during the refinement process. Besides the risk of contention with other users' the number of jobs running on Blacklight also increases as the core count increases. Because of these overheads, the performance of PODM deteriorates on Blacklight for a high core count.

The green bar in Fig. 20.b illustrates that the communication overhead time of LAPD is less than half that of PODM. The idling time and the rollback overhead time of each thread in LAPD stay approximately at the same percentage as those in PODM. Since the communication overhead time is the main part of the total overhead time in PODM, the total overhead time in LAPD is reduced by a large percentage after the communication overhead time is reduced by the locality-aware optimization.

Table 2. shows the performance comparison of PODM and LAPD algorithms for

the 3D Brain atlas. Again, we can see clearly the significant performance improvement of LAPD compared to that of PODM.

3.3 SCALABLE 3D PARALLEL DELAUNAY IMAGE-TO-MESH CONVERSION FOR DISTRIBUTED SHARED MEMORY ARCHITECTURES

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

PDR.PODM(I,$\bar{r_t}$,$\bar{r_I}$,N,C)
Input: I is the input segmented image;
$\bar{r_t}$ is the circumradius upper bound of the elements in the final mesh;
$\bar{r_I}$ is the circumradius upper bound of the elements in the initial mesh;
N is the number of computing nodes;
C is the number of cores in a computing node.
Output: A Delaunay Mesh $\mathcal{M}$ that conforms to the upper bound $\bar{r_t}$.

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

FIG. 7.: A high level description of the PDR.PODM algorithm.

Figure 7. is a high level description of PDR.PODM. A bounding box of the input image is created, the lattice structure is constructed, and the buffer zones of each

subregion are found and stored (lines 1 to 3). The construction and the distribution of the initial mesh are done in parallel by PODM running on multiple cores (lines 4 to 6). Lines 7 to 29 delineate the subsequent parallel refinement procedure after the construction of the initial mesh. All of the subregions are pushed to a refinement queue Q. Each subregion in Q includes the bad elements that belong to the corresponding subregion. If Q is not empty, a PODM mesh generator that is running on a multi-core computing node gets the bad elements from one subregion to refine. Multiple PODM mesh generators that are running on different computing nodes can do the refinement work of different subregions simultaneously. PDR.PODM follows the refinement rules of PODM to create the volume mesh and recover the isosurface. As shown in lines 15 to 23, after creating a new element e', we check whether e' is a bad element or not, and if it is, we check which refinement rule it violates. Then, based on the coordinates of its circumcenter, we add the element either to the current PEL or to the PEL of a neighbor subregion for further refinement. Figure 7. lists only the main steps of PDR.PODM. The actual implementation is more elaborate, in order to support efficient data structures and parallel processing.

3.3.1 PARALLEL INITIAL MESH CONSTRUCTION

The construction of an initial mesh is the starting point for the subsequent parallel procedure. PDR uses the sequential TetGen [2] algorithm to create the initial mesh, which increases the sequential overhead of the whole parallel algorithm. In order to reduce the sequential overhead, we used the PODM mesh generator to create the initial mesh in parallel. There are two important parameters that will affect the performance of the whole algorithm when we create the initial mesh. The first one is the number of cores that we use to create the initial mesh. This value should be neither too small nor too large. If the number of cores is too small, it will not be enough to explore the available concurrency. If it is too large, the communication overhead among them is high. Both of these cases make the construction of the initial mesh time-consuming and deteriorate the performance of PDR.PODM. In practice, we found that 64 cores is the optimal value for creating the initial mesh when running PDR.PODM on Blacklight. The second parameter is the circumradius upper bound $\bar{r_I}$ that we use to control the volume of the upper bound of created elements. This number determines the number of elements of the initial mesh. The larger $\bar{r_I}$ is, the larger the volume of the created tetrahedra are, and thus, fewer elements are created,

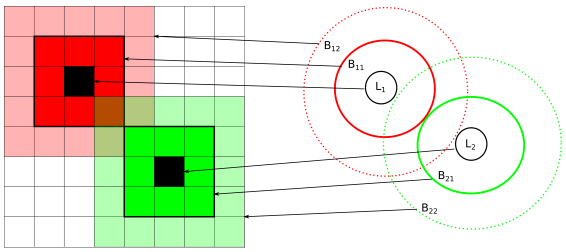


FIG. 8.: A two dimensional illustration of three dimensional buffer zones. The Venn diagram on the right part demonstrates the logical relations between two subregions and their first and second buffer zones.

because the volume of the input object is fixed. If $\bar{r_I}$ is too small, the meshing time of the initial mesh is too long because a large number of elements are created; if it is too large, there is not enough concurrency for the subsequent refinement procedure because there are not enough elements in the initial mesh. In our experiments, we used $\bar{r_I} = 4\bar{r_t}$, where $\bar{r_t}$ represents the target radius upper bound for the final mesh, since it gave the best performance for PDR.PODM among the different values of $\bar{r_I}$ that we have tried, so far.

3.3.2 INITIAL MESH DECOMPOSITION AND DISTRIBUTION

We used a simple, but efficient, way to divide the whole input image into subregions; this consists of partitioning the bounding box into cubes. Then, we assigned tetrahedra to different subregions based on the coordinates of their circumcenters. Consider a subregion L_1. The 26 neighbor subregions form its first level buffer zone B_{11}(the dark red region shown in Figure 12.b). When subregion L_1 is under refinement, all of the subregions in the first level buffer zone B_{11} cannot be refined by another PODM mesh generator simultaneously. During the refinement procedure, the point insertion operation might propagate to one subregion of its first level buffer zone. Consider a case where L_1 and L_2 are refined simultaneously. If B_{11} and B_{21} are not disjoint, this may result ina nonconforming mesh across B_{11} and B_{21}. Therefore, we used a second level of buffer zones, B_{12} and B_{22} (light red and light green in Figure 12.b) in order to ensure that B_{11} and B_{21} are disjoint. In our implementation, if one subregion is popped up from the refinement queue during the refinement, all of its first and second level buffer neighbors were also popped up. This guarantees that two subregions that are refined simultaneously are at least two layers (subregions)

away from each other; thus, the aforementioned problems are eliminated.

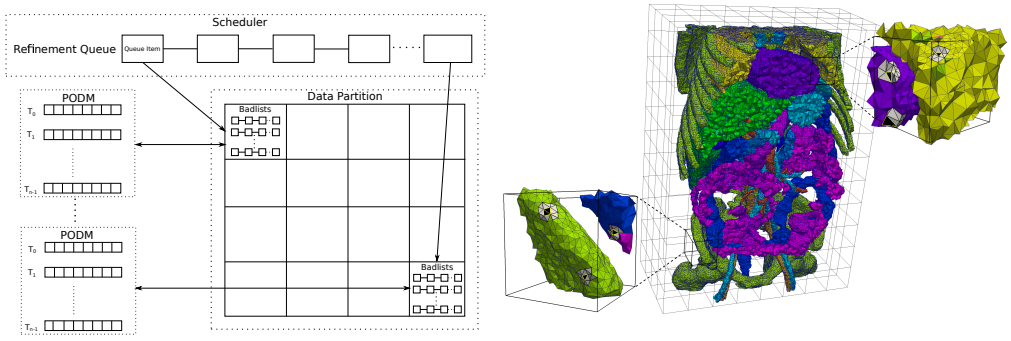
3.3.3 TWO-LEVEL PARALLEL MESH REFINEMENT

In distributed shared memory (DSM) systems, memory is physically distributed while it is accessible to, and shared by, all of the cores. However, a memory block is physically located at various distances from the cores. As a result, the memory access time varies, and it depends on the distance of a core from a memory block. Based on a benchmark of the system group of the Pittsburgh Supercomputing Center [60], as shown in Table 3., the memory latency inside one blade (*Computing Node*) is $O(200)$ cycles and it increases when the number of network switches increases. Each extra switch adds about $O(1,500)$ cycles of latency penalty.

TABLE 3.: Memory hierarchy and the approximate memory access time (clock cycles) of Blacklight

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

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



(a) (b)

FIG. 9.: (a) A diagram that illustrates the design of PDR.PODM parallel Delaunay mesh generation algorithm. (b) Two-level parallelism illustration. The selected subregions are refined simultaneously and multiple cavities are expanded concurrently within a single subregion.

level, the threads running on the cores of a computing node follow the refinement rules of PODM, in order to refine the bad elements of each subregion in parallel. The load balance among the cores of each computing node is performed by the load balancing scheme of the PODM mesh generator.

Figure 9.a shows a diagram of the PDR.PODM parallel mesh generation implementation design. The boxes that are marked *PODM* represent parallel Delaunay mesh generators. The block *Data Partition* represents the partition of the whole region (the bounding box of the input image). The block *Scheduler* represents the management and distribution of PODM mesh generators on different subregions. *Refinement Queue* is a refinement queue that stores all the subregions. Each *Queue Item* stores a pointer to one subregion of the lattice structure. Figure 9.b shows an instance of the two-level parallel mesh refinement. The two subregions are selected to be refined simultaneously, and inside each region multiple points are inserted concurrently.

3.3.4 EXPERIMENTAL RESULTS AND ANALYSIS

In this section, we evaluate the performance of PDR.PODM on distributed shared memory architecture. We performed a set of experiments, creating uniform meshes to access the weak scaling and strong scaling performance of PDR.PODM. When

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

Experiment Setup

The input images we used in our experiment are the CT abdominal and brain atlas from IRCAD Laparoscopic Center [55]. We also show several uniform and varying size meshes using a 3D MRI with brain tumor. Our experimental platform is Blacklight [60], the cache-coherent NUMA shared memory machine in the Pittsburgh Supercomputing Center. Blacklight is a cc-NUMA shared-memory system consisting of 256 blades. Each blade holds 2 Intel Xeon X7560 (Nehalem) eight-core CPUs, for a total of 4096 cores across the whole machine. The 16 cores on each blade share 128 Gbytes of local memory. One individual rack unit (IRU) consists of 16 blades and 256 cores. A 16-port NL5 router is used to connect blades located internally to each IRU. Each of these routers connects to eight blades within the IRU. The remaining eight ports of the internal router are used to connect to other NL5 router blades [53]. The total 4096 cores have 32 TB memory.

Weak Scaling Performance of Uniform Meshing

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

created increases gradually from 3 million to 745 million when the number of cores increases from 1 to 256. Table 4. shows the weak scaling performance of PODM and PDR.PODM.

Evaluation Metrics

The quality of an element e (tetrahedron or triangle) is measured by its *radius-edge ratio*. Let $r(e)$ and $l(e)$ denote the circumradius and the shortest edge of e respectively. The radius-edge ratio of e is defined as $\rho_e = \frac{|r(e)|}{|l(e)|}$. The radius-edge ratio of each element in the output mesh generated by PDR.PODM is smaller than 1.93 because it utilizes the same refinement rules as PODM [3, 18].

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

$$S(p) = \frac{elements\_per\_sec(p)}{elements\_per\_sec(1)} = \frac{elements(p) \cdot time(1)}{time(p) \cdot elements(1)} \tag{2}$$

In equation (2), $elements\_per\_sec(p)$ represents the number of elements created per second using p cores while $elements\_per\_sec(1)$ represents the number of elements (tetrahedra) created per second by the best sequential mesh generation algorithm.

Scalability Analysis

Table 4. demonstrates that PODM shows outstanding performance when the number of cores is less than or equal to 64. The speedup, using 32 cores and 64 cores, is 34.1 and 63.8 respectively, which means that the speedup increases linearly

TABLE 4.: Performance of PODM & PDR.PODM. The input is the abdominal atlas.

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

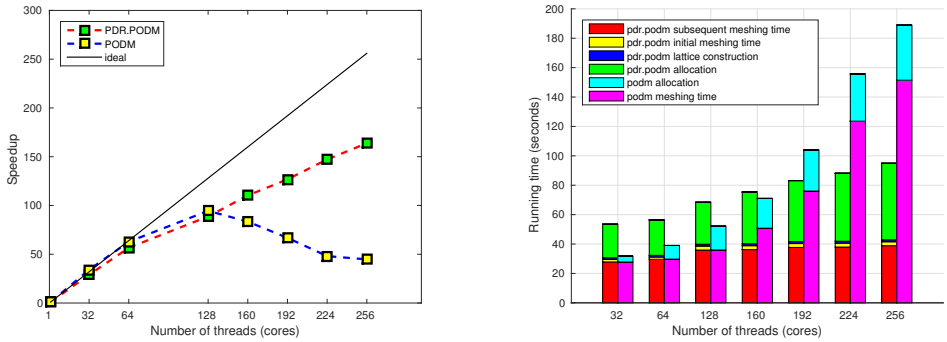
with respect to the number of cores. On 128 cores, PODM achieves a speedup of 94.5 and an efficiency of 73.86%. However, the performance of PODM deteriorates when the number of cores is more than 128 on Blacklight. We ran a set of bootstrapping experiments, from 128 cores to 256 cores, with an increase of two blades (32 cores) each time, to test the performance deterioration of PODM. Each time we increased the number of cores, the speedup decreased. For example, the speedup on 160 cores is 83.5, which is lower than that of 128 cores, and it decreased to only 44.7 for 256 cores. The reason for this performance deterioration of PODM is the increase of communication time due to the large number of remote memory accesses and the congested network. The blue dashed line with yellow markers in Figure 10.a shows this performance deterioration of PODM when the core count is above 128 quite clearly.

PDR.PODM exhibits better scalability potential when the number of cores is higher than 128, as shown in Table 4.. We ran the same set of bootstrapping experiments from 128 cores to 256 cores with a step of two blades (32 cores) each time, in order to compare the performance with PODM. We observed that, each time we increased the number of cores, the speedup of PDR.PODM increased, while the speedup of PODM decreased. For example, the speedup on 128 cores was only 88.7 and it increased to 163.8 for 256 cores. The reason for this performance enhancement is the data partition that PDR offers. As we described before, we partition the whole region into subregions, and we also divide all of the available cores into groups (computing nodes). Therefore, the communication among different computing nodes is eliminated during the refinement procedure, and the runtime checks during the cavity expansion in each subregion involve only a small number of cores.

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

TABLE 5.: Weak Scaling Performance of PDR.PODM. The input is the BigBrain.

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



(a) (b)

FIG. 10.: (a) Weak scaling speedup of PODM and PDR.PODM on 32 to 256 cores on Blacklight. Three million tetrahedra are created by each thread running on a core. The black line depicts the ideal linear speedup. The red dash line with green markers shows the speedup of PDR.PODM and the blue one with yellow markers is the speedup of PODM. (b) Running time of PDR.PODM and running time of PODM.

is lower than that of PODM. As illustrated in Table 4., PODM using 32 and 64 cores creates 3.75 million and 6.91 million elements per second, respectively, and the speedup is linear with respect to the number of cores, while PDR.PODM created 3.19 million and 6.23 million elements per second respectively, and the speedup is only 29.0 and 56.6 respectively. The lower speedup of PDR.PODM is caused by the overhead that we introduced to check and distribute newly created elements to the corresponding subregions for further refinement. When the number of cores is small (< 128), the overhead that we introduced was more than the overhead that we want to reduce, because of remote memory accesses and rollbacks.

Figure 20.b depicts the execution time of PODM and PDR.PODM. The execution time of PODM consists of the allocation time for the initialization of the threads and the meshing time. The execution time of PDR.PODM includes the allocation time, the lattice construction time, the initial mesh creation time, and the subsequent mesh refinement time. We can see clearly in Figure 20.b that the total meshing time of PDR.PODM, i.e. the meshing time to create the initial mesh (the yellow block of the left bar) plus the meshing time in the subsequent refinement procedure (the red block of the left bar), is greater, although by a small amount, than the meshing time of PODM (the light purple bar on the right) when the number of cores is lower than or

TABLE 6.: Strong Scaling Performance of PDR.PODM. The input is the abdominal atlas.

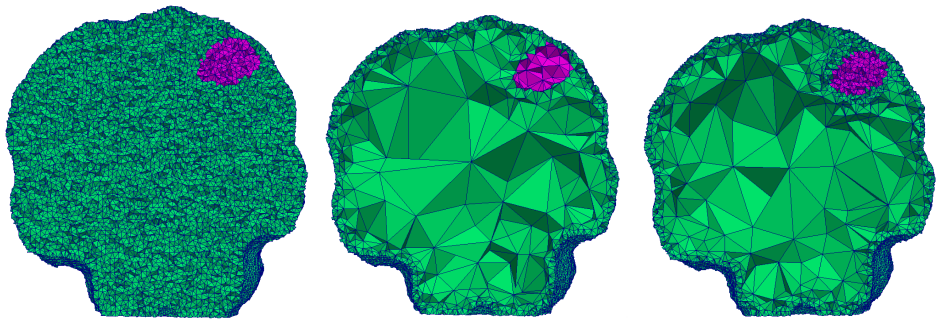
Cores	Elements (million)	Meshing Time (second)	Elements/s (million)	Speedup	Efficiency (%)
1	376.2	3412.27	0.11	1.0	100.00
16	376.2	236.83	15.88	14.4	90.23
32	376.2	122.26	30.77	27.9	87.41
64	376.2	63.28	59.44	54.0	84.44
128	376.2	38.65	97.32	88.5	69.12
160	376.2	34.64	108.59	98.7	61.70
192	376.2	29.86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

equal to 128. However, this drawback of PDR.PODM can be easily overcome. What we need to do is set a threshold on the number of cores. When the number of cores is lower than this threshold, we deactivate the lattice structure and all of the related data decomposition and scheduling procedures. Only when the number of cores is higher than this threshold and PODM does not perform well is the PDR.PODM mode activated to take advantage of its scalability potential.

Table 6. shows the strong scaling performance of PDR.PODM for an input image abdominal atlas. In the strong scaling case, the number of elements remains the same. In the experiments, the number of elements is about 376.2 million for all runs from 1 to 256 cores. As demonstrated in Table 6., when the number of cores is large (> 128), the strong scaling speedup is a little lower than the weak scaling speedup shown in Table 4.. The underlying reason is that the ratio of useful computation to overhead decreases as the number of cores is increased.

Comparision: Uniform Meshing and Varying Size Meshing

Figure 11. shows examples of a uniform mesh and two varying size meshes created by PDR.PODM for the brain tumor image. Figure 11.a demonstrates a uniform volume mesh. The size upper bound of the element is the same (uniform) for both materials. Figure 11.b shows a varying size mesh. Small elements are created near the boundaries in order to capture the fine features of the boundaries while, in the regions far away from the boundaries, the elements are larger. Figure 11.c gives an illustration of another varying size mesh that contains a critical region around the tumor specified by the user. A dense uniform mesh is created inside the critical region around the tumor while a varying size mesh is maintained outside.



(a) (b) (c)

FIG. 11.: Uniform and varying size meshes created by PDR.PODM for the input brain tumor image (two materials). (a) A uniform volume mesh for both materials. (b) A varying size mesh with the same geometric fidelity is created that small elements created near the boundaries. (c) Another varying size mesh with a dense mesh created inside the user-specified region around the tumor.

We also ran a set of experiments from 16 to 256 cores that generated varying size meshes for the multi-material abdominal image. Table 7. shows the experimental results of PDR.PODM in creating uniform meshes and varying size meshes. Compared to the uniform meshes, the corresponding varying size meshes have many fewer elements. As a result, the meshing time required to create the varying size meshes is less than the meshing time for the uniform mesh with the same fidelity. The largest uniform mesh in the experiment contains 745.7 million elements, and the meshing time is 41.53 seconds for 256 cores. PDR.PODM created a corresponding varying mesh with the same fidelity in only 5.91 seconds, which is almost seven times faster

TABLE 7.: Comparison of Uniform Meshing and Varying Size Meshing. The input is the abdominal atlas.

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

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

than the uniform meshing. Furthermore, since the number of elements is lower than the uniform mesh, the finite element solver will perform faster using the varying size mesh, compared to using the uniform mesh.

CHAPTER 4

PARALLEL DELAUNAY IMAGE-TO-MESH CONVERSION ALGORITHMS FOR DISTRIBUTED MEMORY CLUSTERS

In this chapter, 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.

4.1 ALGORITHM

The algorithm has two stages: the pre-processing stage and the two level parallel refinement stage. In the pre-processing stage, the algorithm first uses the scheduling information from the system to create processes and threads. The scheduling information includes the hostname (which also can be called the node name) and the number of available cores (and their IDs) of each node. Based on the information, the algorithm creates an MPI process on each node. Each process creates threads based on the number of available cores of each node. The number of threads of each process (master and worker threads) is equal to the number of available cores

of each node, and each is different from each other. The node with maximum number of available cores is chosen as the master node, and the process running on it is the master process. Then, a coarse mesh is constructed by the multiple threads of the master process, and the whole region (the bounding box of the input image) is decomposed into subregions. Next, the bad quality elements of the coarse mesh are assigned into subregions based on the coordinates of the circumcenters of the elements. If the circumcenter of an element is inside a subregion, the element is assigned to that subregion.

The second stage is the two-level parallel mesh refinement stage. The algorithm simultaneously explores process-level parallelization and thread-level parallelization: inter-node communication using MPI and inter-core communication inside one node using threads. In process level parallelization, the master process uses a task scheduler to schedule the tasks (subregions) to worker processes, through MPI communication. Each subregion is considered as a task, and the submesh inside the subregion is the data. The worker processes communicate with the master process and with each other for task requests and data migration. In the thread-level parallelization, the process of each computing node creates multiple threads that follow the refinement rules of the Parallel Optimistic Delaunay Mesh generation algorithm (PODM) [34, 18] to refine the bad elements of each subregion in parallel by inserting multiple points simultaneously. The parallel mesh refinement terminates, until all of the bad elements in the coarse mesh are eliminated according to the user-specified quality criteria.

The MPI communication and the local shared memory mesh refinement are separated, in order to overlap the communication and computation in this two-level parallelization model. The master thread of each process that runs on each computing node initializes the MPI environment, and creates worker threads based on the number of available cores of each computing node. It 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 in the shared memory of each node. One thing that we should mention is that the number of threads of each process should be at least two (a master thread for communication and a worker thread for mesh refinement), in order to separate the MPI communication and the local mesh refinement. Therefore, we set a threshold for the number of available cores and we

discarded the node that had only one available core.

4.2 MPI+THREADS IMPLEMENTATION

In this section, we present a hybrid MPI + BoostC++ Threads parallel image-to-mesh conversion implementation for distributed memory clusters [23]. 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 it 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 to schedule the tasks (subregions) to worker processes, through MPI communication. We describe a method of 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. 13. and Fig. 14. give a high level description of our hybrid MPI+Threads parallel mesh generation algorithm.

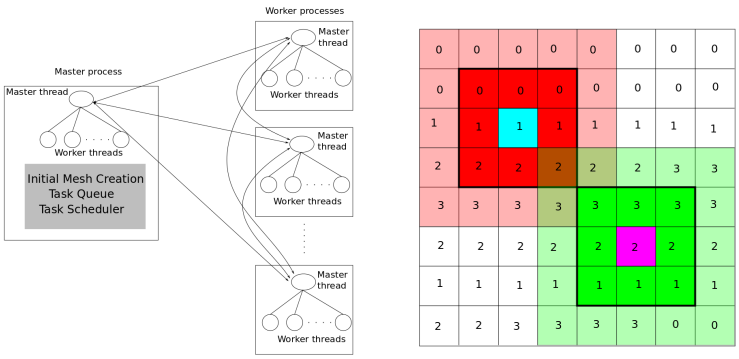


FIG. 12.: (a) A diagram that illustrates the design of nested master-worker model. (b) A two dimensional illustration of three-dimensional buffer zones.

4.2.1 COARSE LEVEL DATA DECOMPOSITION AND TASK SCHEDULER

We used a simple but efficient way to decompose the whole input image into subregions, which consisted of partitioning the bounding box into cubes. Then, we assigned tetrahedra to different subregions, based on the coordinates of their circumcenters. We used a two-level buffer scheme to select and schedule independent subregions to multiple processes, which could then be refined simultaneously without synchronization. Consider that a subregion and the twenty six neighbor subregions form its first level buffer zone (dark red or dark green region shown in Fig. 12.b). When a subregion is under refinement, all of the subregions in the first level buffer zone cannot be refined simultaneously, because the point insertion operation might propagate to one or several subregions of its first level buffer zone. Consider a case in which two subregions are refined simultaneously. If their first level buffer zones are not disjoint, this may result in a nonconforming mesh in the intersection subregions of their first level buffer zones. Therefore, we use a second level buffer zone (the light red and light green regions in Fig. 12.b) in order to ensure that the first level buffer zones of the two subregions 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 it can be refined by another process, concurrently. We used a task queue and task scheduler to schedule the independent tasks that could 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 of its dependent tasks, i.e. its first and second level buffer neighbors, are also popped up. This guarantees that the 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 were scheduled to one worker process for refinement, the submeshes of its first level neighbors would 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. 12.b. 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. 14.).

4.2.2 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. 12.a is a diagram that illustrates the design of the nested master-worker model. The master process, running on a node (called the master node), creates the initial mesh, and manages and schedules the tasks (subregions), while the worker processes, running on other nodes (worker nodes), communicate with each other and with the 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 the local shared memory mesh refinement are separated, in order to overlap 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 it pins each worker thread on one core of the compute node. Therefore, the number of threads (master and worker threads) of each process is equal to the number of cores of each node. The master thread initializes the MPI environment and communicates with the master thread of the 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. 13. and Fig. 14. list the main steps of the master process and the 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 initial mesh, decomposes the initial mesh, and initializes the task queue and scheduler (lines 1 to 5 in Fig. 13.); (ii) a worker process P_i sends a task (subregion) request to the master process P_0 (line 2 in Fig. 14.); (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. 13.); (iv) P_i sends a data request to each process P_j that has the submeshes that P_i needed (lines 4 to 18 in Fig. 14.); (v) after

```
MASTER PROCESS($P_0$)($I$, $\bar{\delta_t}$, $\bar{\delta_I}$, $g$)
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;
 1: Generate an initial mesh in parallel that conforms to δ̄_I;
 2: Use a uniform octree to decompose the whole region into subregions based on g;
 3: Find the buffer zones of each subregion of the octree;
 4: Distribute the initial mesh to octree leaves based on their circumcenter coordinates;
 5: Push all octree leaves to a task queue Q;
 6: while (1)
 7:     Probe the message;
 8:     if The message is a task (subregion) request message from a worker process P_i;
 9:         if Q! = ∅
10:             Receive message from P_i;
11:             Get one subregion L from task queue Q;
12:             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.
13:             Set process P_i to status HAS_WORK;
14:         else if Q == ∅ && at least one worker process's status is HAS_WORK;
15:             Receive message from P_i;
16:             Put P_i to waiting task list WTL;
17:             Send a message to P_i with status WAIT_IN_LIST;
18:         else if Q == ∅ && all worker processes' statuses are NO_WORK;
19:             Send termination message to P_i;
20:             Send termination message to every process that is waiting in the WTL;
21:             if the number of terminated workers == the number of workers
22:                 break;
23:             endif
24:         endif
25:     endif
26:     if The message is a data (submesh) request message from P_i
27:         Receive the message from P_i;
28:         Pack data (submesh);
29:         Send data (submesh) to P_i;
30:     endif
31:     if The message is a feedback from P_i that just finished the refinement work
32:         Receive the message from P_i;
33:         Set P_i to status NO_WORK;
34:         Update task queue Q based on the feedback message from P_i;
35:         while Q! = ∅ && waiting task list WTL is not empty
36:             Get one subregion from Q;
37:             Pop one process P_j from waiting task list WTL;
38:             Send subregion and neighbors Location information to P_j;
39:             Set P_j to status HAS_WORK;
40:         endwhile
41:     endif
42: endwhile
```

FIG. 13.: A high level description of Master Process's (P_0) work.

getting all of the submeshes of L and its neighbors, the worker threads of P_i start the mesh refinement; (vi) P_i sends a feedback message to the master process P_0 and P_0 updates the task queue based on the feedback message (lines 31 to 41 in Fig. 13.); (vii) if all of the refinement work is done, P_0 sends a termination message to each worker process P_i and master process exits after all worker processes terminate (lines 18 to 24 in Fig. 13.).

A worker process does not send the submeshes of a subregion and neighbors back to the master process after it has finished the refinement work. Instead, it sends a feedback message that only contains the number of bad elements of each subregion 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

```
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;
 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
```

FIG. 14.: A high level description of a Worker Process's (P_i) work.

needed when a worker process gets a task (subregion) to refine. The worker process sends a data request to other worker processes which hold the submeshes in their local memories. A worker process is likely to send data requests to other worker processes and to receive data requests from these worker processes, simultaneously. In order to handle the interleaving messages among the worker processes and to avoid deadlocks, non-blocking MPI communication and a message polling approach are used when the master thread of a worker process tries to collect the Submeshes that it needs from other worker processes (lines 6-18 in Fig. 14.). When the worker threads of a process are doing the refinement work, the master thread is still able to receive and respond to the data requests from other workers (lines 20-26 in Fig. 14.), since the communication and the computation are separated.

4.2.3 OPTIMIZATIONS

The first optimization is to reduce the format translation overhead caused by the STD map utilized in the previous implementation. The time complexity of the lookup operation in the std::map is logarithmic in size. Therefore, for each time of data mapping from the pointer to integer index, the time is approximately $O(nlgn)$, where n is the number of elements that need to exchange among the processes. Obviously, a constant mapping with the time complexity of $O(1)$, instead of the std::map, will reduce the format translation overhead. The new implementation introduced and maintained the index-based mesh structure during the mesh refinement procedure. After introducing the index-based mesh structure during the mesh refinement procedure, the std::map could be replaced by one-to-one direct mapping. The lookup time of each element will, then, be constant instead of logarithmic, which reduces the overhead caused by the explicit mesh format translation.

The second optimization is to reduce the delay caused by the communication, the packing and unpacking, and the waiting time of both the master and the worker threads. In a previous implementation, the master thread was responsible for the communication, the data packing and unpacking, and the mesh structure format translation. It caused the following problems: (i) It increased the waiting time (idle time) of the worker threads of the local mesher. The local mesher could not do any mesh refinement work in the required subregion until it got all of the data (the submeshes of the level one neighbors). (ii) It increased the response time of the master thread(s) of other process(es) which sent data requests to ask for data to current-process. The process was only able to perform one operation at a time: either the data processing (the unpacking and translation) or the response to the MPI message of other processes. The delay in the data processing of the current process caused a delay in response to the data requests messages from other processes, and vice versa. In order to solve the problem, we offloaded some work from the master thread to the worker threads, to speed up the data processing step. The master thread was now only responsible for communication and data migration. The data processing (unpacking and translation) work was split to the worker threads.

The third optimization is to maximize the resource utilization. Most of the current supercomputer architectures consist of clusters of nodes that are used by many clients (users). A user wants his/her job submitted in the job queue to be scheduled promptly. However, the resource sharing and job scheduling policies that are used

in the scheduling system to manage the jobs are usually beyond the control of users. Therefore, in order to reduce the waiting time of their jobs, it is becoming more and more crucial for the users to consider how to implement the algorithms that are suitable to the system scheduling policies and are able to effectively and efficiently utilize the available resources of the supercomputers. We proposed a hybrid MPI+Threads parallel mesh generation algorithm on distributed memory clusters with efficient core utilization. The algorithm takes the system scheduling information into account and is able to utilize the nodes that have been partially occupied by the jobs of other users. In the previous work [61], we proposed a parallel Delaunay image-to-mesh conversion algorithm which is the first three-dimensional hybrid MPI+Threads parallel meshing algorithm that involves both distributed memory parallelization and shared memory parallelization. However, the execution of the algorithm has very high resource requirements. It only works on clusters with homogeneous nodes and needs exclusive accesses of these nodes. In other words, it cannot execute on the nodes which have been partially occupied by the jobs of other users. As a result, the waiting time of the algorithm in the queue is large in order to get the resources it needs. The optimization overcomes the limitations of the previous parallel meshing algorithm [61] which only works on clusters with homogeneous nodes and needs exclusive accesses of these nodes. As a result, it is up to 12.74 times faster than the previous algorithm without efficient core utilization for 400 cores and 2.58 billion element mesh as illustrated in Table 12..

4.2.4 LOAD BALANCE

In order to alleviate the load balance problem during the parallel mesh refinement procedure, a semi-dynamic load balance scheme was introduced. It deals with the load balance issue of the algorithm on two levels: the coarse-grain level load balance among processes, and the fine-grain level load balance among threads within a process.

Coarse Level Load Balance

On the coarse-grain level, over-decomposition [62] is utilized to deal with the load balance problem among processes. We over-decomposed the whole region so that the number of subregions was much larger than the number of processes, to ensure that each process would have enough subregions to refine. This method introduced some

overhead, but it helped to alleviate the load balance problem among the processes. A study of optimal load balance strategies among processes, while keeping the overhead and communication cost small, is part of our future work.

Fine Level Load Balance

On the fine-grain level, a load balance list was used to spread the elements among the 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 T_i pushes back its $Thread\_ID$ to the load balance 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 awakened by another thread T_j when T_j produces some work for T_i. After a running thread T_j completes a Delaunay insertion operation, it checks all of the newly created elements and puts the ones that are regarded as bad elements on the BEL of the first thread T_i, found in the load balance list. T_j also removes T_i from the load balance list.

4.3 PERFORMANCE

4.3.1 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 was the Turing cluster computing system at the High Performance Computing Center of Old Dominion University. We tested the performance of our implementation on Turing with its two subclusters: the Phi cluster and the Ed-Main cluster. The Phi cluster contained nine Intel Xeon Phi nodes each, with two Xeon Phi MIC cards and 20 cores. The Ed-Main cluster of Turing contained 190 multi-core compute nodes each, containing between 16 and 32 cores and 128 Gb of RAM. We used two 3D multi-tissued images as inputs in the experiments: (i) the CT abdominal atlas obtained from the IRCAD Laparoscopic Center [55], and (ii) the knee atlas obtained from Brigham & Women's Hospital Surgical Planning Laboratory [63]. We performed the experiments on the Phi cluster using up to 180 cores, and on the Ed-Main cluster using up to 900 cores (45 compute nodes, the maximum number of nodes available

for us in the experiments). We used the **Weak Scaling Speedup** $\boldsymbol{S}$, (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** $\boldsymbol{E}$ (the ratio of the speedup (S) to the number of cores (p): $E = S/p = T_s/(pT_p)$) to evaluate the scalability of the parallel mesh generation algorithms [57, 58].

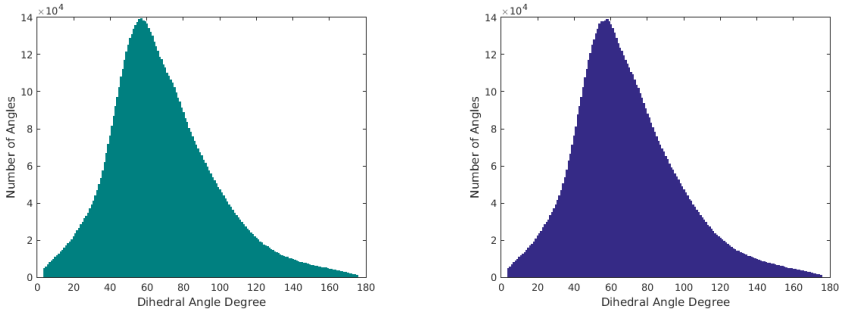
In the weak scaling case, the problem size (i.e., the number of elements created) increased proportionally to the number of cores. Because of the irregular nature of the unstructured tetrahedra mesh, it was impossible to control the problem size, which increased exactly by p times when the number of cores was increased from 1 to p. Therefore, an alternative definition of speedup was used, which was more precise for a parallel mesh generation algorithm. We measured the number of elements generated every second during the experiment. Then the speedup could be calculated as $S(p) = \frac{elements\_per\_sec(p)}{elements\_per\_sec(1)}$.

4.3.2 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 (its radius-edge ratio, for short) and its (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 edge ratio bound in our implementation was less than 2, in our implementation. Because of the potential nearly flat tetrahedra, a more useful measure was the dihedral angle. We compared the quality of

TABLE 8.: 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°



(a) (b)

FIG. 15.: (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.

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 (an abdominal atlas and a knee atlas), as shown in Table 8.. We also showed the dihedral angle distribution of the final meshes created by our algorithm and PODM in Fig. 15.a and in Fig. 15.b. The goal of such comparisons is to illustrate that the hybrid MPI+Threads is able to generate meshes with the same quality guarantees as PODM.

4.3.3 SCALABILITY, GRANULARITY AND CONCURRENCY

In this subsection, we present the weak scaling performance of the implementation on the Phi cluster up to 180 cores (9 compute nodes) with different data decomposition granularities. The number and size of the 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 determined the number of leaves (subregions) of the decomposition. The number of subregions was $N_{sub} = 8^d$, where d was the depth of the octree. In the algorithm, we passed the depth of the octree as an input parameter, in order to control the granularity of the coarse level data decomposition. We performed the experiments on the Phi cluster with two different data decomposition granularities:

- $d = 3$ represents the octree split to depth 3, with 512 subregions.
- $d = 4$ represents the octree 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 increased from 6.64 million to 1.17 billion for the input image abdominal atlas, and from 6.33 million to 1.14 billion for the knee atlas, when the number of cores increased from 1 to 180. Table 9. and Table 10. show the weak scaling performance of the algorithm for the two input images (abdominal atlas and knee atlas), respectively.

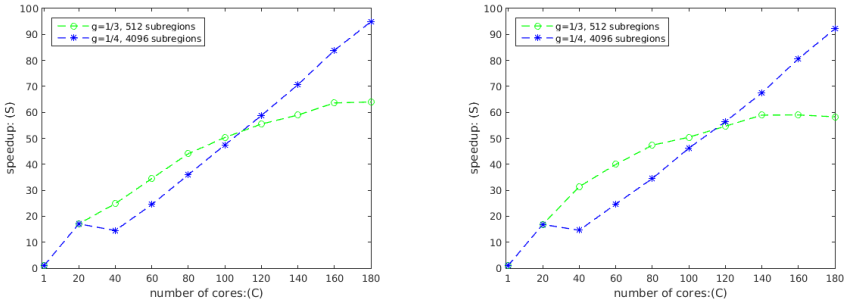
TABLE 9.: Weak scaling performance of data decomposition with different granularities. The input is abdominal atlas.

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 10.: 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 9. and Table 10., the algorithm gets near-linear weak scaling performance for each 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, when the coarse mesh and underlying image is partitioned into 512 subregions, than that with $d = 4$, i.e. when 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 the degree of concurrency, because the maximum number of tasks (subregions) that can be executed (refined)

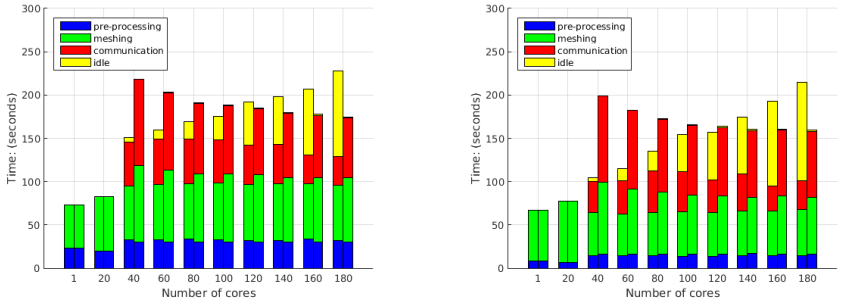


(a) (b)

FIG. 16.: (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.

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. 17.a and Fig. 17.b, 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. 16.a and Fig. 16.b. 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 9. and Table 10.. We observed that, each time we increased the number of cores, the efficiency of experiment with 4096 subregions increased, while the efficiency with 512 subregions decreased. Take the experimental result of the input abdominal as an example: the efficiency with 512 subregions on 40 cores was 62.10% and it decreased to 35.56% for 180 cores. In contrast, the efficiency with 4096 subregions on 40 cores was 36.25% and it increased to 52.72% for 180 cores. Fig. 16.a and Fig. 16.b 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; 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.

Fig. 17.a and Fig. 17.b show the breakdown of the total running time for the



(a) (b)

FIG. 17.: 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.

experiments with the two images, respectively. The running time consists of four parts: (i) the pre-processing time: the time that the master process spends on loading an image from the disk, constructing an octree, creating the coarse mesh, assigning the elements of the coarse mesh to subregions, and creating subthreads; (ii) the meshing time: the time that a process (more precisely, the multiple worker threads of a process) spends on mesh refinement; (iii) the communication time: the time that a process spends on task requests and data movement; and (iv) the idle time: 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 calculated the average time of each part for all processes. As demonstrated in Fig. 17.a and Fig. 17.b, the idle time with large granularity (512 subregions) continued to increase, from 40 cores (2 nodes) to 180 cores (9 nodes). It became the major overhead that deteriorated the performance of the algorithm when more than five nodes were used, because of the low degree of concurrency. In this case, a finer decomposition was required, although it introduced 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

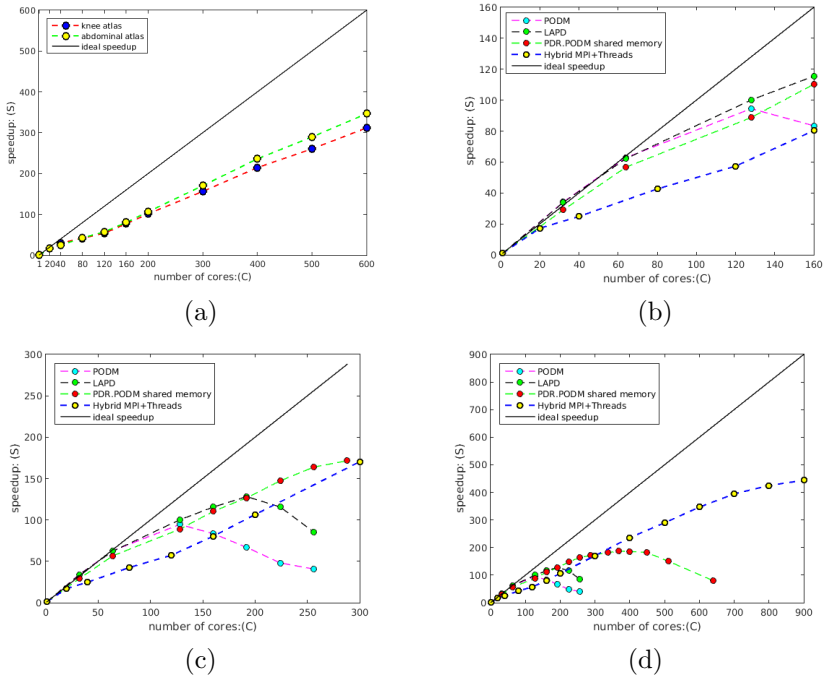


FIG. 18.: (a) The overall weak scaling speedup of the two input images up to 900 cores (45 nodes) on Ed-Main cluster of Turing. (b), (c), (d) The weak scaling speedup comparison of hybrid MPI and Threads implementation with other three shared memory algorithm implementations and ideal speedup for upto 160, 300 and 900 cores.

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.

4.3.4 PERFORMANCE COMPARISON

We ran a set of experiments on the Ed-Main cluster of the Turing cluster system up to 900 cores (45 nodes) to test the scalability of the algorithm. We used the same two input images, abdominal atlas and knee atlas, as our test experiments 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 was less than or equal to 5 (100 cores), and $d = 4$ when the number of nodes was between 6 and 30 (120 to 900 cores). Fig. 18.a demonstrates the weak scaling speedup of the

TABLE 11.: 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 [18, 15] to improve the scalability.
MPI+Threads	900	45.25 million	Distributed Memory	Hybrid MPI and Threads algorithm involves two level parallelization.

two input images up to 900 cores (45 nodes) on the Ed-Main cluster of the Turing.

We compared the performance of the hybrid MPI+Threads algorithm with three other shared memory algorithms, PODM [18], LAPD [21], and PDR.PODM [22]. The main characteristics of these algorithms are listed in Table 11.. The input image that we used in all of the experiments is the abdominal atlas. Fig. 18.b shows the speedups of the four implementations, up to 160 cores. As we can see, the locality-aware parallel mesh generation algorithm, LAPD, had the best performance, because it increased the data locality during the parallel mesh refinement. In fact, all of the other three methods had better performance than that of the hybrid MPI+Threads algorithm when the number of cores was small (less than 300). The reason is that the hybrid method 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 experiments are performed with more than 300 cores, the MPI+Threads method has the best performance, compared with the other three methods, because of the two levels of parallelization that it utilizes.

4.3.5 PERFORMANCE ON HETEROGENEOUS CLUSTERS

In this section, we shows the experimental results of the algorithm on heterogeneous clusters with the disruption of other users' jobs. We compared the performance of the new implementation with the previous implementation which only works on cluster with homogeneous nodes.

The total time of the algorithm in the experiments has two parts: the execution time and the waiting time. The execution time is the time that the job executes on

TABLE 12.: The performance comparison of the heterogeneous algorithm and the homogeneous algorithm. The input is abdominal atlas.

Cores	Nodes		Elements	Execution Time (s)		Waiting Time (s)		Total Time (s)		Total Time Ratio
	homo	hetero	(million)	homo	hetero	homo	hetero	homo	hetero	homo:hetero
1	1	1	6.64	64.70	64.35	0	0	64.70	64.35	1:1
20	1	1	133.93	76.47	77.27	0	0	76.47	77.27	1:1
40	2	3	261.54	102.63	107.09	0	0	102.63	107.09	0.96:1
60	3	4	390.61	110.35	115.22	0	0	110.35	115.22	0.96:1
80	4	6	520.31	115.02	121.44	0	0	115.02	121.44	0.95:1
100	5	10	650.16	125.96	204.79	0	0	125.96	204.79	0.62:1
200	10	22	1292.20	139.33	249.87	731	0	870.33	249.87	3.48:1
300	15	26	1937.00	131.53	231.44	3674	181	3805.53	412.44	9.23:1
400	20	36	2577.91	127.65	280.35	18622	1191	18749.65	1471.35	12.74:1

the cluster. The waiting time is the time that the job spends on waiting in the SGE job queue to get the resources from the scheduling system. The waiting time depends on the job scheduling of the system and the resources that a job requests, such as the number of cores and the amount of memory. In the experiments, the problem size (i.e., the number of elements created) in the experiments increases proportionally to the number of cores and the number of elements per core remains approximately constant. The comparison of the total time of the heterogeneous algorithm and the previous homogeneous algorithm is demonstrated in Table 12..

The execution time of the heterogeneous algorithm is a bit longer than that of the homogeneous algorithm because more computing nodes involve in the communication and the execution of the job is disturbed by the jobs of other users running on the same node. As shown in Table 12., the execution time of the homogeneous algorithm of 100 cores is 125.95 seconds and five 20-core node are used while the execution time is 204.79 seconds of the heterogeneous algorithm and ten partially occupied nodes are used in order to get 100 cores. However, the heterogeneous algorithm reduces the waiting time which is the dominating part of the total time when the number of cores is large. As a consequence, the total time of the algorithm is much smaller than that of the homogeneous algorithm. As shown in Table 12., the waiting time for the homogeneous algorithm of 400 cores is about 18622 seconds in order to get the computing resources (the homogeneous nodes with exclusive accesses). In contrast, it only takes about 1191 seconds for the new algorithm waiting in the queue before it executes. As a result, the total time of the previous algorithm is 12.74 times more than that of the heterogeneous algorithm for the experiments of 400 cores.

4.3.6 STRONG SCALING PERFORMANCE

We also tested the strong scaling performance of the algorithm. The input image that we used in the expreiments is the abdominal atlas. In the strong scaling case, the problem size is fixed across all the runs with different number of cores. The scalability of the algoritm in the strong scaling case is far more worse than that of the weak scaling case. It only scales up to 800 cores in the test experiments. The reason can be explained as follows. 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 demonstrate 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 [57]. 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.

TABLE 13.: Weak scaling performance up to 6000 cores. The input image is abdominal atlas.

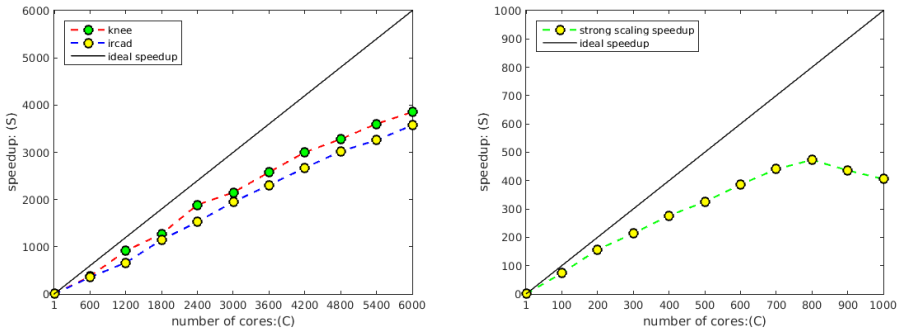
#Cores	1	2400	3000	3600	4200	4800	5400	6000
#Nodes	1	106	114	144	171	192	213	247
# Elements [a]	6.64 M[b]	16.85 B[c]	21.03 B	25.18 B	29.43 B	33.63 B	37.80 B	41.98 B
Running Time(s)	79.70	132.41	129.64	130.99	132.38	133.68	139.29	141.17
Elements/second(M)	0.83	127.24	162.25	192.23	222.31	251.55	271.36	297.38
Speedup	1.00	1527.71	1948.10	2308.04	2669.21	3020.21	3258.11	3570.51
Efficiency	1.00	0.65	0.64	0.63	0.63	0.61	0.59	0.62

[a] # Elements represents the number of elements.
[b] M: million.
[c] B: billion.

TABLE 14.: Weak scaling performance up to 6000 cores. The input image is knee atlas.

#Cores	1	2400	3000	3600	4200	4800	5400	6000
#Nodes	1	78	104	135	156	191	213	247
# Elements [a]	6.33 M[b]	14.97 B[c]	18.67 B	22.44 B	26.16 B	29.86 B	33.59 B	37.32 B
Running Time(s)	67.27	87.98	96.62	96.64	96.79	101.16	103.55	107.62
Elements/second(M)	0.90	170.11	193.53	232.17	270.24	295.15	324.37	346.79
Speedup	1.00	1808.52	2057.48	2468.42	2873.13	3137.9	3448.72	3687.03
Efficiency	1.00	0.75	0.69	0.69	0.68	0.65	0.63	0.61

[a] # Elements represents the number of elements.
[b] M: million.
[c] B: billion.



(a) (b)

FIG. 19.: (a) Weak scaling speedup up to 6000 cores of two input images. (b) Strong scaling speedup up to 1000 cores with input ircad atlas.

4.3.7 PERFORMANCE FOR LARGE NUMBER OF CORES

We decomposed the whole domain into 32768 regions. The number of tetrahedra created gradually increased from 6.64 million to 41.98 billion for the input image abdominal atlas, and from 6.33 million to 37.32 billion for the knee atlas, when the number of cores increases from 1 to 6000.

Table 13. and Table 14. show the weak scaling performance of the hybrid parallel mesh generation algorithm of the two input images, respectively. We ran the experiments using from 1 up to 6000 cores (the maximum number of available cores on Turing). As we mentioned before, in the weak scaling case, the number of elements increased proportionally to the number of cores. The number of elements increased linearly from 6.64 million using a single core to 41.98 billion using 6000 cores for the input image abdominal atlas, and from 6.33 million to 37.32 billion for the input image knee atlas. The implementation of the algorithm scaled well up to 6000 cores. On 6000 cores, it achieved a speedup of 3570.51 with an efficiency of 62% with the input abdominal atlas. This was the best result for parallel mesh generation algorithms running on a distributed memory clusters. One can note that the speedup and efficiency of the algorithm for the same number of cores has some perturbation.

4.3.8 OVERHEAD ANALYSIS FOR LARGE NUMBER OF CORES

We analyzed the communication overhead for the experiments, up to 6000 cores. The breakdown of the total running time consists of four parts: the pre-processing

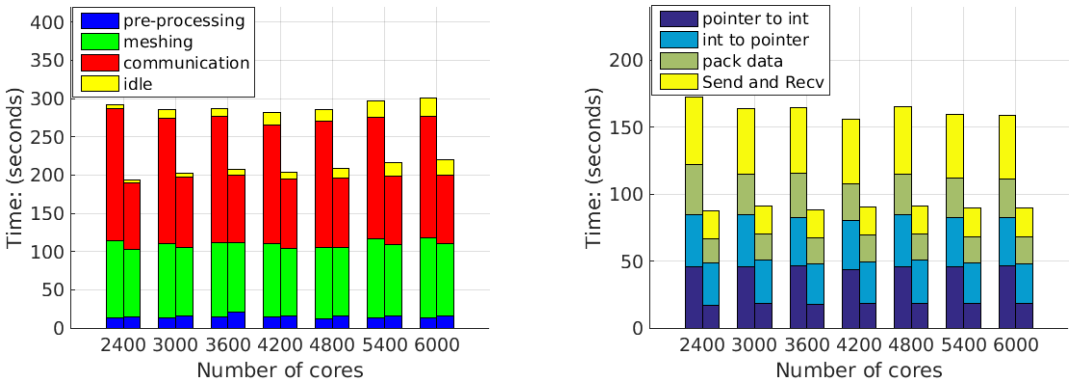


FIG. 20.: (a) Running time breakdown comparison. The left bar and right bar represent the running time of implementation before and after optimization. (b) Overhead Comparison. The left bar and right bar represent the overhead of implementation before and after optimization.

time, the meshing time, the communication time, and the idle time. The pre-processing time is the time that the master process spends loading an image from disk, constructing an octree, creating the coarse mesh, assigning the elements of the coarse mesh to subregions and creating subthreads. The meshing time is the time that a process (more precisely, the multiple worker threads of a process) spends on mesh refinement. The communication time is the time that a process spends on task requests and data movement. The idle time is the time that a process waits in the waiting list and does not perform any mesh refinement work. In the experiments, we counted 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). Then, we calculated the average time of each part for all of the processes in one experiment.

Fig. 20.b shows the communication overhead breakdown. The communication overhead consists of four parts: the pointer mesh structure to integer index mesh structure (pointer to int) time, the integer index mesh structure to pointer mesh structure (int to pointer) time, the packing and unpacking data time, and the sending and receiving time. We reduced almost 50% of the total overhead (the red part of left bar in Fig. 20.a) in the new implementation. As a result, we generated a decrease of about 25% of the total running time and we improved the performance in the new

implementation by about 25%.

CHAPTER 5

CONCLUSION AND FUTURE WORK

In this thesis, we summarize several years of the progress that we made in a novel framework for highly scalable and guaranteed quality mesh generation for finite element analysis in three dimensions. We present three parallel Delaunay mesh generation algorithms on shared and distributed memory supercomputers: (i) A three dimensional Locality-Aware Parallel Delaunay image-to-mesh conversion algorithm (LAPD), which employs a data locality-aware mesh refinement process to reduce the latency caused by the remote memory access. (ii) A parallel mesh generation algorithm for distributed shared memory architecture, which takes advantage of two legacy approaches, i.e. the Parallel Optimistic Delaunay Mesh generation algorithm (PODM) and the Parallel Delaunay Refinement algorithm (PDR). It quickly leverages high parallelization because of the aggressive speculative approach employed by PODM, and it uses data partitioning offered by PDR to avoid the runtime checks and to decrease the communication overhead. (iii) A scalable three-dimensional hybrid MPI+Threads parallel Delaunay image-to-mesh conversion algorithm on distributed memory clusters, which simultaneously explores process-level parallelization and thread-level parallelization: inter-node parallelization using MPI, and inter-core parallelization inside one node using threads. We implemented a nested master-worker model to handle the inter-node MPI communication and the intra-node local mesh refinement separately, in order to overlap the communication (the task request and data movement) and computation (the parallel mesh refinement). We achieved this by leveraging concurrency at two different granularity levels, using a hybrid message passing and multi-threaded execution model which is suitable to the hierarchy of the hardware architecture of the distributed memory clusters. An end-user productivity and scalability study was performed on up to 6000 cores, and indicated very good end-user productivity, with about 300 million tetrahedra per second and about 3600 weak scaling speedup. A set of experiments were conducted at the Pittsburgh Supercomputing Center and at the High Performance Computing Center of Old Dominion University in Norfolk, Virginia. The experimental results demonstrated that the novel framework that we proposed for scalable mesh generation is suitable to the

hierarchy of distributed memory clusters with multiple cores, and it showed, so far, the best scalability. We analyzed the communication overhead for the experiments, up to 6000 cores, and found that the communication overhead takes a certain portion in the total running time. Therefore, there is still space for improvement in terms of the scalability of Delaunay-based isotropic grid generation codes. Tasks for the future are to reduce the communication overhead and to improve data locality, in order to further improve the performance of the algorithm.

REFERENCES

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

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

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

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

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

[6] 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.

[7] 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.

[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] L. Oliker, R. Biswas, Parallelization of a dynamic unstructured algorithm using three leading programming paradigms, IEEE Transactions on Parallel and Distributed Systems (TPDS) 11 (9) (2000) 931–940.

[12] R. A. Chowdhury, F. Silvestri, B. Blakeley, V. Ramachandran, Oblivious algorithms for multicores and networks of processors, Journal of Parallel and Distributed Computing 73 (7) (2013) 911–925.

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

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

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

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

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

[18] 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.

[19] 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.

[20] A. Lastovetsky, Heterogeneous parallel computing: from clusters of workstations to hierarchical hybrid platforms, Supercomputing frontiers and innovations 1 (3).

[21] D. Feng, A. Chernikov, N. Chrisochoides, Two-level locality-aware parallel Delaunay image-to-mesh conversion, Parallel Computing10.1016/j.parco.2016.01.007.

[22] D. Feng, C. Tsolakis, A. Chernikov, N. Chrisochoides, Scalable 3d hybrid parallel delaunay image-to-mesh conversion algorithm for distributed shared memory architectures, Computer-Aided Design-Http://dx.doi.org/10.1016/j.cad.2016.07.010.

[23] D. Feng, A. Chernikov, N. Chrisochoides, A hybrid parallel delaunay image-to-mesh conversion algorithm scalable on distributed-memory clusters, in: International Meshing Roundtable, 2016.

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

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

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

[27] 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.

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

[29] 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.

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

[31] 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.

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

[33] 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.

[34] 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.

[35] 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. doi:http://dx.doi.org/10.1016/0956-0521(94)90014-0.

[36] 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.

[37] 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.

[38] G. Globisch, Parmesh – a parallel mesh generator, Parallel Computing 21 (3) (1995) 509–524. doi:http://dx.doi.org/10.1016/0167-8191(94)00085-O.

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

[40] N. P. Chrisochoides, A survey of parallel mesh generation methods, Tech. Rep. BrownSC-2005-09, Brown University, also appears as a chapter in Numerical Solution of Partial Differential Equations on Parallel Computers (eds. Are Magnus Bruaset and Aslak Tveito), Springer, 2006 (2005).

[41] D. Ibanez, I. Dunn, M. S. Shephard, Hybrid MPI-thread parallelization of adaptive mesh operations, Parallel Computing 52 (C) (2016) 133–143. doi:10.1016/j.parco.2016.01.003.
URL http://dx.doi.org/10.1016/j.parco.2016.01.003

[42] 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.

[43] D. Malhotra, G. Biros, Algorithm 967: A distributed-memory fast multipole method for volume potentials, ACM Trans. Math. Softw. 43 (2) (2016) 17:1–17:27.

[44] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, G. Biros, A massively parallel adaptive fast multipole method on heterogeneous architectures, Commun. ACM 55 (5) (2012) 101–109.

[45] Q. Hu, N. A. Gumerov, R. Duraiswami, Scalable fast multipole methods on distributed heterogeneous architectures, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, ACM, New York, NY, USA, 2011, pp. 36:1–36:12.

[46] J. Dreher, R. Grauer, Racoon: A parallel mesh-adaptive framework for hyperbolic conservation laws, Parallel Comput. 31 (8+9) (2005) 913–932.

[47] J.-F. Remacle, O. Klaas, J. E. Flaherty, M. S. Shephard, Parallel algorithm oriented mesh database, Engineering with Computers 18 (3) (2002) 274–284.

[48] MOAB: Mesh-oriented database, `http://sigma.mcs.anl.gov/moab-library/` (2016).

[49] J. Rodríguez, J. Weinbub, D. Pahr, K. Rupp, S. Selberherr, Distributed High-Performance Parallel Mesh Generation with ViennaMesh, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 548–552.

[50] S. Soner, C. Ozturan, Generating multibillion element unstructured meshes on distributed memory parallel machines, Scientific Programming 2015.

[51] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, L. Wilcox, Extreme-scale amr, in: 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, 2010, pp. 1–12.

[52] Blacklight, https://portal.xsede.org/web/xup/psc-blacklight (2013).

[53] Sgi altix uv 1000 system users guide, Report (2011).

[54] K. Barker, A. Chernikov, N. Chrisochoides, K. Pingali, A load balancing framework for adaptive and asynchronous applications, IEEE Transactions on Parallel and Distributed Systems 15 (2) (2004) 183–192.

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

[56] Multi-modality MRI-based Atlas of the Brain, http://www.spl.harvard.edu/publications/item/view/2037 (04 2015).

[57] 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.

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

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

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

[61] D. Feng, A. Chernikov, N. Chrisochoides, A hybrid parallel delaunay image-to-mesh conversion algorithm scalable on distributed-memory clusters, in: International Meshing Roundtable, 2016.

[62] 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.

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

VITA

Daming Feng
Department of Computer Science
Old Dominion University
Norfolk, VA 23529

SUMMARY

Daming Feng has broad interested in mesh generation, machine learning and high performance computing. He has solid background in C++ programming, software toolkit development and parallel programming. For his Ph.D. thesis, he was doing research on parallel mesh generation which is very challenging topic on parallel computing.

EDUATION

- Sep. 2012 - Aug. 2019, Ph.D. of Computer Science, Old Dominion University
- Sep. 2006 - May 2009, M.S. of Computer Science, Soochow Univerisy
- Sep. 2002 - May 2006, B.E. of Computer Science, Harbin University of Science and Technology

PROFESSIONAL EXPERIENCES

- Developed a visualization software for the visualization of three dimensional mesh generation and medical image registration based on QT, VTK and ITK.
- Proposed a novel framework for developing highly scalable mesh generation for distributed-shared memory architectures.
- Proposed a hybrid MPI+Threads multi-layered parallel Delaunay mesh generation algorithm, which simultaneously explores process-level parallelization and thread-level parallelization.