University of Mississippi

eGrove

Electronic Theses and Dissertations | Graduate School

1-1-2020

Efficient Generating And Processing Of Large-Scale Unstructured Meshes

Cuong Manh Nguyen

Follow this and additional works at: https://egrove.olemiss.edu/etd

Recommended Citation

Nguyen, Cuong Manh, "Efficient Generating And Processing Of Large-Scale Unstructured Meshes" (2020). *Electronic Theses and Dissertations*. 1872.
https://egrove.olemiss.edu/etd/1872

This Thesis is brought to you for free and open access by the Graduate School at eGrove. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of eGrove. For more information, please contact egrove@olemiss.edu.

EFFICIENT GENERATING AND PROCESSING OF LARGE-SCALE UNSTRUCTURED MESHES

A Dissertation
presented in partial fulfillment of requirements
for the degree of Doctor of Philosophy
in the Department of Computer and Information Science
The University of Mississippi

by

CUONG M. NGUYEN

May 2020

Copyright Cuong M. Nguyen 2020
ALL RIGHTS RESERVED

ABSTRACT

Unstructured meshes are used in a variety of disciplines to represent simulations and experimental data. Scientists who want to increase accuracy of simulations by increasing resolution must also increase the size of the resulting dataset. However, generating and processing a extremely large unstructured meshes remains a barrier.

Researchers have published many parallel Delaunay triangulation (DT) algorithms, often focusing on *partitioning* the initial mesh domain, so that each rectangular partition can be triangulated in parallel. However, the common problems for this method is how to merge all triangulated partitions into a single domain-wide mesh or the significant cost for communication the sub-region borders. We devised a novel algorithm – *Triangulation of Independent Partitions in Parallel (TIPP)* to deal with very large DT problems without requiring inter-processor communication while still guaranteeing the Delaunay criteria. The core of the algorithm is to find a set of *independent* partitions such that the circumcircles of triangles in one partition do not enclose any vertex in other partitions. For this reason, this set of independent partitions can be triangulated in parallel without affecting each other.

The results of mesh generation is the large unstructured meshes including vertex index and vertex coordinate files which introduce a new challenge - locality. Partitioning unstructured meshes to improve locality is a key part of our own approach. Elements that were widely scattered in the original dataset are grouped together, speeding data access. For further improve unstructured mesh partitioning, we also described our new approach *Direct Load* which mitigates the challenges of unstructured meshes by maximizing the proportion of useful data retrieved during each read from disk, which in turn reduces the total number of read operations, boosting performance.

ACKNOWLEDGEMENTS

I express my deepest gratefulness to my advisor, Dr. Philip J. Rhodes, who has given me the opportunity to be his research assistant, my first crucial step towards my career goal. I am very grateful for his earnest and generous support and help during my graduate study. I feel I am quite fortunate to have this considerate and helpful advisor, who constantly helps me to grow professionally and academically. I thank my committee members, Dr. Gregory L. Easson, Dr. Byunghyun Jang and Dr. Feng Wang for their time and valuable comments, making my dissertation more solid and more complete.

I appreciate the support and the assistantship provided by the Department of Computer and Information Science at the University of Mississippi. I am very thankful to other colleagues and professors at Olemiss, who have helped me in my teaching and my research. Lastly, this thesis is dedicated to my wife and my two sweet children.

TABLE OF CONTENTS

ABSTRACT ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES v

LIST OF TABLES viii

INTRODUCTION AND CONTRIBUTIONS 1

PARALLEL DELAUNAY TRIANGULATION FOR LARGE-SCALE DATASETS . . 8

PARALLEL DELAUNAY TRIANGULATION FOR LARGE-SCALE DATASETS USING TWO-LEVEL PARALLELISM 38

ACCELERATING RANGE QUERIES FOR LARGE-SCALE UNSTRUCTURED MESHES 58

LOAD BALANCING FOR A LARGE-SCALE UNSTRUCTURED MESHES 83

RELATED RESEARCH 94

CONCLUSION 99

BIBLIOGRAPHY 101

VITA 110

LIST OF FIGURES

2.1 Delaunay Triangulation 9

2.2 Set of triangles belong to a partition P 10

2.3 Interior and boundary triangles of a partition P 12

2.4 Exterior, Interior and Boundary triangles of a partition P 14

2.5 Boundary and interior triangles of the partition P 15

2.6 Independent and conflicting partitions 18

2.7 Parallel Delaunay Triangulation — example of TIPP with 1000 points. 21

2.7 Parallel Delaunay Triangulation — example of TIPP with 1000 points. 22

2.8 A worst and ideal case of Parallel Delaunay Triangulation 26

2.9 Super triangles introduce sliver triangles. 28

2.10 Triangulation with vertical sweep line 29

2.11 Parallel Delaunay Triangulation performance with four different datasets . . . 32

2.12 Speed up of Delaunay triangulation with different domain sizes 33

2.13 TIPP Execution times for increasing numbers of nodes. 33

3.1 Two level partitioning with uniform and non-uniform point distribution. . . . 39

3.2 Multi-master TIPP — An example with 10,000 points. 41

3.2 Multi-master TIPP — An example with 10,000 points. 42

3.3 The running of multi-master TIPP 46

3.4 TIPP performances with uniform point distribution 48

3.5 Performance of multi-master TIPP relative to single-master TIPP 49

3.6 The scaling behaviors of multi-master and single-master TIPP 50

3.7 An example of a ADCIRC Mesh with non-uniform point distribution 53

3.8 TIPP performance with uniform and non-uniform point distributions 54

3.9 The scaling behavior between multi-master TIPP and Tess 56

4.1 Data structure for unstructured meshes 59

4.2 Owner and borrower cells 60

4.3 The proportion of *load*, *process*, and *update* 63

4.4 Direct Load method 65

4.5 Data organization before and after loading data using DL or LRU of 2D datasets 68

4.6 Owner, Borrower, Borrower Index arrays in the GPU memory 71

4.7 2D range query 72

4.8 LRU execution times for 2D dataset 76

4.9 2D Load performance using *LRU* and *Direct Load* 77

4.10 Partitioning process performance of GPU over CPU 78

4.11 Overall partitioning performances between *Simple* and *Advanced* cases. 79

4.12 The speedup of GPU range query compared to CPU on 2D datasets 81

5.1 Two layers access pattern 84

5.2 Mesh partitioning with graph-based method 86

5.3 Recursive partitioning North Carolina meshes with Morton curve 89

5.4 Parallel partitioning the unstructured meshes with Z-order 91

LIST OF TABLES

2.1 Number of active partitions based on the total number of partitions 36

3.1 Execution time (seconds) for master and worker processes of single and multi-master TIPP for 10 billion points (or 20 billion triangles) on 10 nodes, 256 processes. 51

4.1 Range query time of domain for 2D and 3D datasets 64

4.2 Number of cache misses ($\times 10^6$) and load time for loading 2D vertex file in two cases: (A) block Number = 512 and (B) block Number = 256×2^{10}, warm cache system 75

CHAPTER 1

INTRODUCTION AND CONTRIBUTIONS

Delaunay triangulation[31] is a fundamental problem for many fields including terrain modeling, scientific data visualization, surface construction, finite element analysis, and computational fluid dynamics. Scientists in these fields demand ever-increasing resolution and spatial scale, which in turn increases the number of points that must be triangulated. The increasing availability of large scale parallelism over the last several decades make larger triangulations possible. Three main topics will be covered in this chapter including triangulation, I/O, and load balancing.

1.1 Large-Scale Parallel Delaunay Triangulation

The demand of high mesh resolution together with the widely available high performance hardware motivates to develop new triangulation algorithms that take advantage of parallelism. Unlike regular grids, a triangulation may place points where they are most needed in order to accurately represent rapidly changing attributes. That is, the resolution of the triangulation is easily varied over the domain according to the needs of the application. *Computational Fluid Dynamics (CFD)* is a classic application of Delaunay Triangulation, and appears in a variety of fields, including biology and medicine [63, 6], physics [58], and mechanical engineering [98]. However, triangulations are quite general, finding applications outside CFD, including chemistry [27], communications [51], and computer vision [30].

Any triangulation of a set of points $P = \{p \in \mathbb{R}^2\}$ produces a set of triangles with vertices taken from P. Edges of the triangles do not cross, and triangles do not overlap [50]. The *Delaunay* Triangulation (DT) of P has additional properties that are particularly desirable for science and engineering applications.

Computing a DT for a set of points is computationally expensive due to exhaustive search. For example, the incremental insertion approach described in section 2.1 requires us to find the set of triangles with *circumcircles* that enclose a newly added point. Smaller meshes can be calculated on a single machine, but as demand for larger meshes increases, we must find ways to distribute the computational cost over several machines.

Researchers have published many parallel DT algorithms, often focusing on *partitioning* the initial mesh domain, so that each rectangular partition can be triangulated in parallel [26, 40, 23, 22, 13, 35, 52, 53, 54, 55]. As a result of this parallel strategy, triangulation performance improves significantly. However, a common problem for this method is how to merge all triangulated partitions into a single domain-wide mesh. The triangles in the border of a partition have to connect to triangles in adjacent partitions. Any newly generated triangles in the borders of partitions should (preferably) satisfy the Delaunay properties. This merging step is sometimes referred to as "stitching", and is notoriously difficult, especially when guaranteeing a result that satisfies the Delaunay criteria. One way to avoid the stitching problem is to communicate between partitions during triangulation. However, this is especially expensive in a distributed environment, and when working with very large datasets.

Other studies [25, 33] divide the initial domain into many sub-regions with arbitrary shapes, allowing load balancing by choosing shapes with roughly the same number of triangles. However, performance is somewhat reduced due to communication and contention along the sub-region borders.

We develop a novel algorithm – *Triangulation of Independent Partitions in Parallel (TIPP)* to deal with very large DT problems without requiring inter-processor communication while still guaranteeing the Delaunay criteria. The core of the algorithm is to find a set of *independent* partitions (see section 2.3) such that the circumcircles of triangles in one partition do not enclose any vertex in other partitions. For this reason, this set of independent partitions can be triangulated in parallel without affecting each other and reduced signif-

icantly the communication between workers while still globally guaranteeing the Delaunay criteria throughout the triangulation time.

1.2 Delaunay Triangulation of Large-Scale Datasets Using Two-Level Parallelism

With the *TIPP* algorithm, we can deal with a very large DT problems in parallel with uniform data distribution. However, there are some issues that need to improve. First, the bottleneck seems to be happen at master node as increasing dataset size. Indeed, the original TIPP algorithm relies upon a single master node to identify the set of partitions that can be scheduled simultaneously. We evaluated performance using synthetic datasets containing billions of points with a uniform distribution over the domain. Results indicated that when the number of partitions is large, the master node becomes a bottleneck, causing worker nodes to wait for new work. Second, the load unbalancing is another factor that significantly affects the triangulation performance. With non-uniform data distribution, the computing capability is wasted by the idle time of some worker processes. Some processes have done their tasks while others got heavy jobs since the data is not balanced.

We improve the original TIPP algorithm, producing *multi-master* TIPP, which shares the master's burden over several sub-masters. Sub-masters can also receive worker results in parallel, so workers receive the next task sooner. We measure TIPP performance using a real-world dataset with a non-uniform distribution of points across the domain, observing the impact of uneven load on overall performance. We modify TIPP to better address the challenges of uneven point distribution. To evaluate the new method, we also analyze and examine the scalability behavior of our multi-master TIPP versus *Tess* [72, 60, 59] which is a distributed-memory with high scalability Delaunay and Voronoi parallel algorithm.

We show that our improvements allow us to triangulate even larger datasets, with 20 billion triangles or more using commodity machines. As unstructured datasets continue to grow in scale and resolution, tools such as TIPP will help to extend the reach of scientific inquiry.

1.3 Accelerating Range Queries for Large-scale Unstructured Meshes

The size of modern scientific datasets has been steadily increasing into the terabyte and petabyte range, presenting challenges for storage and processing resources. The problem is even more acute for *unstructured meshes*, because these datasets have especially poor locality, which significantly hampers performance.

Unstructured meshes are used in a variety of disciplines to represent simulations and experimental data. Scientists who want to increase accuracy of simulations by increasing resolution must also increase the size of the resulting dataset. It is therefore important that the size of datasets used by scientists should not be dictated by the capacity of a local machine, disk, or memory. Recently, we have applied a partitioning technique to the storage and processing of large unstructured datasets throughout the memory hierarchy [79, 1, 3]. By allowing unstructured meshes (or grids) to be processed in a piecewise fashion, storage and memory capacity no longer constrain the practical size of datasets. However, this partitioning process is computationally expensive, as is the interpolation process that is performed during dataset access. Both processes can benefit from a parallel implementation.

With the integration of hundreds to thousands of processing cores, Graphics Processing Units (GPUs) [62] present powerful computational capabilities and have become a general-purpose computing platform for data-intensive applications. However, GPUs have limited memory capacity, which requires careful management of data access when processing very large datasets. Data management is further complicated by the poor locality exhibited by unstructured grids, which especially hurts performance when accessing a spinning disk.

Dataset partitioning (or *chunking*) has been an active area of research for decades [10, 68, 82, 84]. However, much of this work does not address the unique challenges of unstructured meshes, while other work does not take advantage of the bandwidth and computing power of modern GPUs. For example, Akande et al. report that the partitioning problem can take hours for a 25GB data volume [2, 1, 3]. Though they demonstrate significant performance gains by improving I/O behavior, scaling to even larger data volumes

must also address computation.

The performance of our GPU implementation improves drastically over the serial version, achieving a 4× speedup for the partitioning process, and over 100× for the range query. These results reflect not only the computational advantages of GPU computing, but also our efforts to keep the GPU busy by efficiently reading data from disk. In this paper, we will describe a new approach to the I/O problem called *Direct Load*. This method mitigates the challenges of unstructured meshes by maximizing the proportion of useful data retrieved during each read from disk, which in turn reduces the total number of read operations, boosting performance.

1.4 Contributions

- We devise a novel algorithm– Triangulation of Independent Partitions in Parallel (TIPP) for partitioning unstructured meshes and distributing the partitions to nodes in a cluster. The core of TIPP algorithm is to identify the independent partitions which can be assigned to processes run independently without communication between them. Hence, reduce the overall performance of Delaunay triangulation.

- We prove that our algorithm generates triangulations that globally satisfy the Delaunay criteria in entire Delaunay triangulation time. Specifically, we make two claims that require formal verification. First, we establish that the effect of triangulating points contained within a partition is localized, implying that an appropriate set of partitions can be triangulated in parallel. We call this property *independence*. The second claim requiring verification is that our approach does not require a stitching process in order to join the triangles of partitions once they have been processed. We refer to this property as *fit*. For these reasons, we do not have to merge all partitions after triangulations.

- We also analysis the tradeoff between the parallelism and the workload of master process to prepare initial triangles for other partitions. Large number of initial triangles

in the domain means that more number of independent partitions will be generated. Thus, increase the parallelism. However, it would cause more workload for the master process.

Taken together, these contributions make possible the triangulation of extremely large datasets, with billions of triangles. As a result, we significantly improve performance by reducing the scope of the searches each node must perform, and by allowing nodes to work in parallel. However, the original TIPP algorithm relies upon a single master node to identify the set of partitions that can be scheduled simultaneously. We evaluated performance using synthetic datasets containing billions of points with a uniform distribution over the domain. Results indicated that when the number of partitions is large, the master node becomes a bottleneck, causing worker nodes to wait for new work.

– For above reasons, we further develop the TIPP algorithm to employ multiple master processes, distributing computational load across several machines. This new design improves both performance and scalability. Since, the workload for master process is distributed to multiple sub-master processes, the idle time of independent partition preparation reduces considerably.

– Load unbalancing is another issue that affect performance. We measure TIPP performance using real-world dataset with a non-uniform distribution of points across the domain, observing the impact of uneven load on overall performance. We propose two ways to mitigate the process idle time during the triangulation. First, we partition the non-uniform point distribution dataset in two-level partitions to reduce the point differences between minor partitions. Then, base on number of points that assigns for sub-master, we dynamically distribute number of processes to sub-masters accordingly.

– We also develop a new method named *Direct Load* to address the bad locality problem of unstructured mesh instead of using LRU. The Direct Load is working much better

LRU in the case when the dataset size is larger than main memory. We parallelize the partitioning process for unstructured meshes as well as range queries using the GPU.

CHAPTER 2

PARALLEL DELAUNAY TRIANGULATION FOR LARGE-SCALE DATASETS

Because of the importance of Delaunay Triangulation in science and engineering, researchers have devoted extensive attention to parallelizing this fundamental algorithm. However, generating unstructured meshes for extremely large point sets remains a barrier for scientists working with large scale or high resolution datasets. In this chapter, we introduce a novel algorithm – *Triangulation of Independent Partitions in Parallel* which can triangulate a large number of points in parallel and guarantee all triangles are Delaunay globally at any time during the triangulation.

2.1 Background

The *Bowyer–Watson* algorithm [14, 95], also known as an *incremental insertion* algorithm, is a method for computing the Delaunay triangulation of a finite set of points. The triangulation begins with *super triangles* large enough to contain all points. The points are added to the domain, one at a time. For each point p, the triangles whose circumcircles contain p are deleted, leaving a star-shaped polygonal *cavity*. The cavity is then retriangulated using p and the vertices of the cavity boundary.

Figure 2.1 presents the basic idea of the Bowyer–Watson method, in which p is the new inserted point. Triangles $\triangle v_1 v_2 v_5$, $\triangle v_2 v_4 v_5$, and $\triangle v_2 v_3 v_4$ are removed because their circumcircles contain p, which introduces a cavity (the shaded region). New triangles are created from the vertices $v_1 ... v_5$ and p, which covers the cavity.

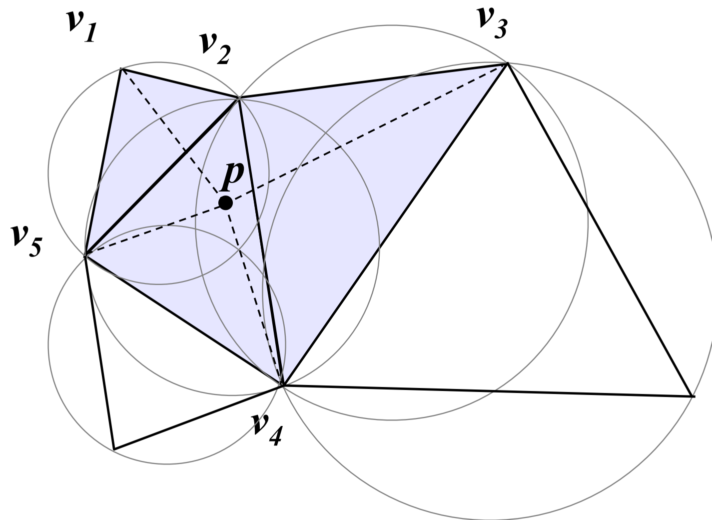


Figure 2.1. Triangles T in cavity $C = \{v_1, v_2, ..., v_5\}$ (the shaded region) have circumcircles enclosing point p.

2.2 Foundations

Delaunay triangulation becomes challenging when dealing with a large number of triangles because the triangle search procedure is so time-consuming. The common method to solve this problem is to divide the domain into regions which are then triangulated in parallel. However, merging these separate triangulations into a single result mesh requires "stitching" the triangles together along region boundaries. It is particularly difficult to stitch regions together while also satisfying the Delaunay criteria, and many existing approaches relax the Delaunay constraint [53, 54, 55, 89].

The work described here divides the domain into rectangular partitions, rather than arbitrarily shaped regions. Like the works cited above, TIPP can triangulate multiple partitions in parallel. However, we allow triangles along partition borders to be refined as the algorithm progresses, so the resulting triangles belong to the final Delaunay Triangulation, and no stitching phase is required.

We make two claims that require formal verification. First, we establish that the effect of triangulating points contained within a partition is localized, implying that an appropriate set of partitions can be triangulated in parallel. We call this property *independence*. The

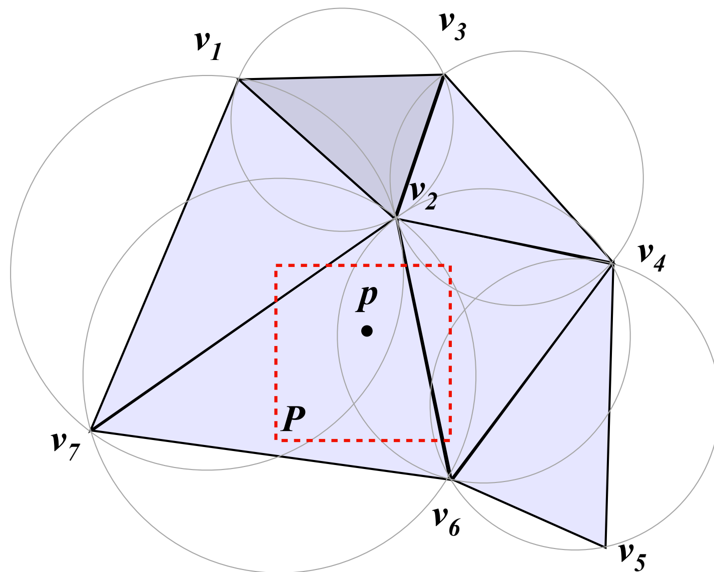


Figure 2.2. Triangles T in polygon $\{V_1, V_2, ..., V_7\}$ have circumcircles that intersect with partition P. The circumcircle of triangle $\triangle V_1V_2V_3$ does not intersect partition P, and is a member of the set T' described in lemma 1.

second claim requiring verification is that our approach does not require a stitching process in order to join the triangles of partitions once they have been processed. We refer to this property as *fit*. The next two sections formally describe and establish these properties.

2.2.1 Independence

Definition 1. *Two partitions P_1 and P_2 are independent iff the points within each partition can be triangulated simultaneously, producing results identical to those produced when the partitions are triangulated one at a time.*

Lemma 1 establishes that only the set of triangles T with circumcircles intersecting a partition P could be deleted by triangulating points contained in P. Lemma 2 establishes that if we add a new point p to the triangulation, where $p \in P$, we will not need to add an edge (and therefore triangle) involving a vertex outside the triangles in T. Taken together, these two lemmas formally establish the possibility of *independent* partitions that can be triangulated without disturbing each other. Specifically, we say two partitions are independent

if there is no triangle t with a circumcircle intersecting both partitions.

Lemma 1. *Let T be the set of triangles with circumcircles that intersect with a partition P, then any triangle $t \notin T$ will not be deleted by the Delaunay triangulation of any new vertex inserted within partition P.*

Proof. As shown in figure 2.2, let $T \subset DT$ be a set of triangles with circumcircles that intersect with a partition P, where DT is the Delaunay Triangulation of the points inserted so far. Let $T' = DT - T$. If P contains a new point p, there is no element of T' with a circumcircle containing p. Therefore, no element of T' will be deleted by the new triangulation made by inserting p. □

Lemma 2. *Let T be a set of triangles with circumcircles that intersect with a partition P. If a point $p \in P$ is added to the triangulation, no edge between p and a vertex outside of T will be generated.*

Proof. Assume there exists an edge connecting p with some vertex v' of a triangle $t' \in T'$, where $T' = DT - T$. The Delaunay algorithm would only construct such an edge if the circumcircle of t' contains p (see figure 2.1). However, $p \in P$, so t' cannot be in T', which contradicts the assumption. □

Theorem 1. *Two partitions P_1 and P_2 are independent iff no triangle t exists such that the circumcircle of t intersects both P_1 and P_2.*

Proof. Let $T(P_1)$ be a set of triangles with circumcircles that intersect with partition P_1. Let $R \subset P_1$ be the set of points that remain to be triangulated.

We assume that no triangle in $T(P_1)$ has a circumcircle that intersects with $T(P_2)$.

By lemma 1, we know that inserting any point $p \in R$ will not cause any triangle outside $T(P_1)$ to be deleted. By lemma 2, we know that inserting any point $p \in R$ will not generate any edge using a vertex outside of $T(P_1)$. By the assumption, $T(P_1)$ does not contain any triangles with circumcircles that intersect $T(P_2)$, and furthermore cannot

have any triangles in common with $T(P_2)$. Therefore, the further triangulation of points in $R \subset P_1$ can neither delete any triangle in $T(P_2)$ or add an edge to any vertex in P_2. We can also make a symmetric claim about the triangulation of P_2. □

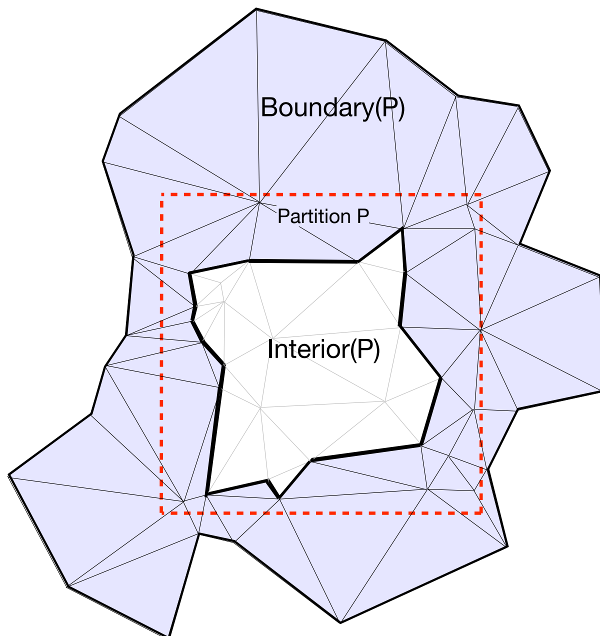


Figure 2.3. Interior and boundary triangles of a partition P, which is drawn as a dashed square. The interior triangles, shown in very light gray, all have circumcircles (not shown) contained wholly inside P. In contrast, the boundary triangles (darker gray) have circumcircles that intersect, but are not wholly inside P.

2.2.2 Fit

Definition 2. *Consider two sets of triangles T_1 and T_2, and the set of edges E_1 and E_2 taken from T_1 and T_2 respectively. We say that T_1 and T_2* fit *if $T_1 \cap T_2 = \varnothing$ and (in the general case) $C = E_1 \cap E_2$ forms a circular path* [1].

This definition implies that one set of triangles wholly surrounds the other, and that when the two sets are joined, there are no holes in the interior of the joined sets, making any "stitching" process unnecessary.

[1]When T_1 or T_2 are adjacent to a domain boundary, $C = E_1 \cap E_2$ forms a single connected path with endpoints on the domain boundary, rather than a circular path.

Lemma 3 establishes that once all points contained within a partition have been inserted into the triangulation, there is a set of *interior* triangles within that partition that can be *finalized*, meaning they can be written to disk, and removed from memory. This not only saves memory, but also reduces the cost of future triangle searches.

Lemma 4 establishes that although the triangles surrounding the interior triangles of a partition may change during triangulation of points outside of partition P, the interior set will fit exactly with the rest of the triangulation without modification. After triangulation of a partition P, there is a set of *boundary* triangles belonging to P that cannot yet be finalized, and must instead be rejoined with triangles of the global mesh in order to be further processed as members of another partition. Lemma 5 demonstrates that the perimeter of this set of triangles will not change during the triangulation of P, meaning they will fit perfectly when rejoined with the global mesh.

Taken together, these three lemmas establish that the results of Delaunay Triangulation within partitions can be merged together without resorting to a stitching process that artificially creates new edges or triangles along partition boundaries.

Lemma 3. *As shown in figure 2.3, we define the* interior *of a partition P as the set of triangles $I = interior(P)$ that have circumcircles wholly contained in P, implying the triangles of I are also wholly inside P. Let R be the set of points that remain to be inserted into the triangulation. If all points $p \in R$ are outside of P, then the set I remains unchanged after the points of R are inserted.*

Proof. From the discussion in section 2.1, we know that only triangles with circumcircles containing the new point will be deleted. No interior triangle has a circumcircle containing $p \in R$, because the circumcircles of these triangles are entirely inside P, and R contains only points outside P. Therefore no interior triangle will be deleted as the points in R are inserted into the triangulation.

We now consider the possibility of adding triangles to the interior of P as the points in R are triangulated. If the insertion of some point $p \in R$ results in a new triangle $t \in I$,

then P contains t and its three vertices, since I consists only of triangles with circumcircles wholly within P. This contradicts the assumption that R contains only points outside of P.

Since no triangle in I is deleted by the triangulation of R, and no triangle is added to I, the set I remains unchanged. □

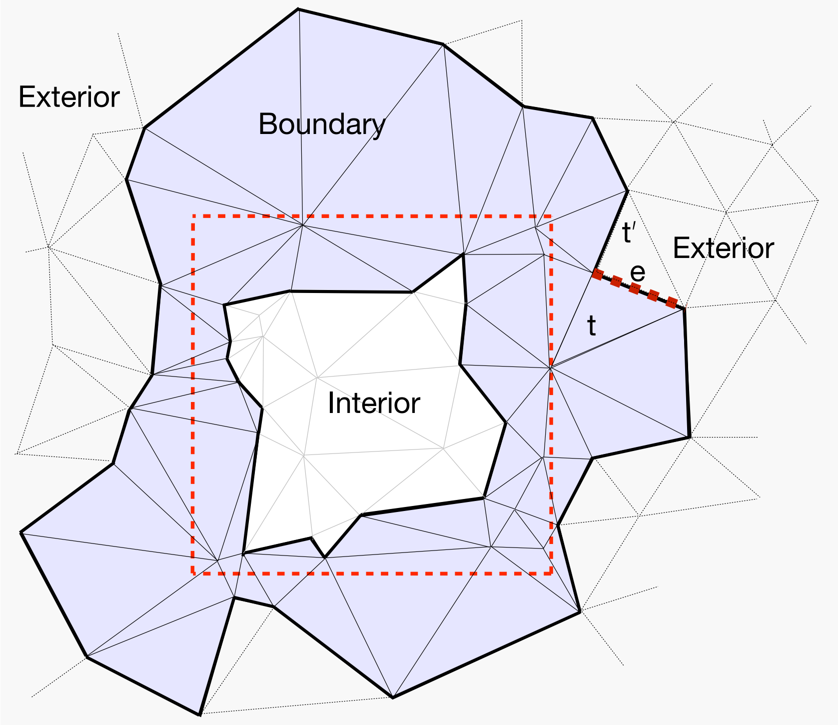


Figure 2.4. Exterior, Interior and Boundary triangles of a partition P. Triangle t belongs to boundary(P), while triangle t' belongs to exterior(P). Circumcircles of exterior triangles do not intersect P.

Lemma 4. *Once all the points within a partition have been inserted into the triangulation, the interior triangles for that partition are finalized. However, the* boundary *triangles of the partition may change during the triangulation of the remaining points. When these boundary triangles B of partition P are finalized, they will still fit the interior of P.*

Proof. We define the set $B = boundary(P)$ of a partition P to be the set of triangles with circumcircles that intersect, but are not wholly contained by P. As shown in figure 2.3, the boundary set of P is hollow, because it can be formed by removing $I = interior(P)$ from

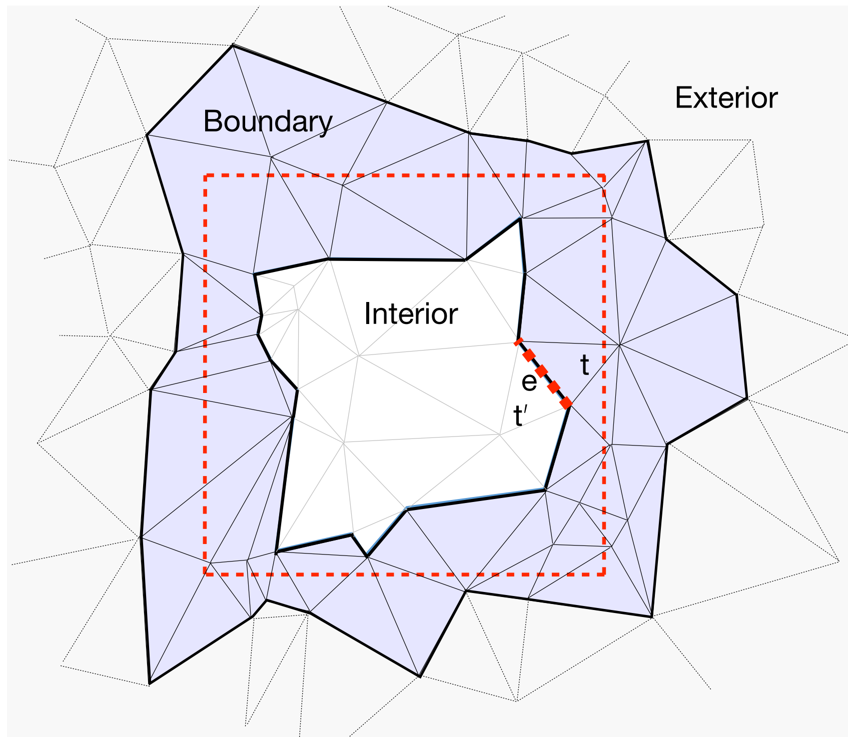


Figure 2.5. Boundary and interior triangles of the partition P when the boundary is ready to be finalized. Notice that although the boundary set has changed compared to the previous figure, the border between the interior and boundary sets remains unchanged.

the set of triangles with circumcircles that intersect with P. We say the boundary has an *inside perimeter* formed by the edges surrounding this hollow region.

We assume that I was finalized when no points $p \in P$ remained, but that the boundary set may have changed due to further triangulation. Consider an edge e of a triangle $t \in B$ that lies on the inside perimeter of B, as shown in figure 2.5. Since the inside perimeter of B is also the perimeter of I, edge e is also an edge of some triangle $t' \in I$. If e is deleted by triangulating a point p via the process described in section 2.1, then both t and t' must have been deleted. If t' has been deleted, it must also be true that t' has a circumcircle extending beyond P, since only points $p \notin P$ remain. This violates the definition of t' as an interior triangle, so we conclude that no edge e on the inner perimeter of B will be deleted by triangulation of points $p \notin P$.

Between the time when I is finalized and when B is finalized, lemma 3 implies that

no new edges will be added to the perimeter between I and B, since no triangles (or edges) will be added to I. We have now also shown that during that same time, no edge of the inner perimeter of B is deleted, implying the perimeter is unchanged. Since this perimeter between B and I is unchanged by triangulation of points $p \in P$, B and I will still fit after this process concludes. □

Lemma 5. *The outside perimeter of the set of boundary triangles B of partition P will not change during the triangulation of P.*

Proof. As before, we define the set $B = boundary(P)$ of a partition P to be the set of triangles with circumcircles that intersect, but are not wholly contained by P. We also define the *exterior* triangles of P as the set of triangles $E = exterior(P)$ with circumcircles that do not intersect P. We say the boundary has an *outside perimeter* consisting of edges between triangles in B and triangles in E.

Consider an edge e of a triangle $t \in boundary(P)$ that lies on the outside perimeter of $boundary(P)$, as shown in figure 2.4. Edge e is also an edge of some triangle $t' \in exterior(P)$.

If e (and therefore t') is deleted by triangulating a point $p \in P$ via the process described in section 2.1, it must also be true that t' has a circumcircle intersecting P, since only such triangles are deleted by that process. This contradicts the definition of t' as an exterior triangle, so neither t' nor e will be deleted by the triangulation of points $p \in P$. We conclude that no edge of the outside perimeter of $boundary(P)$, and no triangle $t' \in exterior(P)$, will be deleted during the triangulation of P.

More broadly, no new triangle $n \in exterior(P)$ will be introduced by the triangulation of P, since n would have to be incident to some point $p \in P$, and there are no such triangles in $exterior(P)$.

During the triangulation of P, no edge of the outside perimeter of $boundary(P)$ is deleted, and $E = exterior(P)$ is unchanged. Therefore, the outside perimeter of $boundary(P)$ is not changed during the triangulation of P. □

Theorem 2. *Immediately after the Delaunay Triangulation of partition P is complete, $boundary(P)$ fits $interior(P)$ and $exterior(P)$.*

Proof. For some partition P, the boundary set $B = boundary(P)$ shares an outer perimeter with $E = exterior(P)$ and an inner perimeter with $I = interior(P)$. Assuming all points $p \in P$ have already been inserted into the triangulation, we know from Lemma 3 that I remains unchanged when triangulating additional points outside partition P. Lemma 4 shows that B will fit I even when the *entire* triangulation is complete, despite changes in B due to triangulation of points outside of P. Lemma 5 shows that B fits E immediately after the triangulation of P.

Therefore, B will fit both I and E immediately after the triangulation of all points in P. Furthermore, B will continue to fit I even when the entire triangulation is complete. □

There are two important implications of Theorem 2. First, once triangulation of P is completed, we can put aside $interior(P)$, since it will not be changed by triangulation of points not in P, and will always fit $boundary(P)$. Second, we can completely avoid a stitching phase if we rejoin $boundary(P)$ with the remainder of the global mesh immediately after triangulation of P is finished. At that time, $boundary(P)$ will fit $exterior(P)$ (and the global mesh) without stitching.

2.3 Triangulation of Independent Partitions in Parallel

Processing very large Delaunay Triangulations in parallel is challenging because we must identify regions that can be processed independently. When a new point is inserted in the domain, some triangles whose circumcircles enclose that point will be removed, and new triangles in the resulting cavity are generated. If more than one point is inserted at the same time, the multiple insertions must not interfere with each other.

Our solution is to divide the domain into partitions, and carefully identify the partitions that can be processed simultaneously while still guaranteeing correctness. Partitioning the domain has the additional benefit of reducing triangle search costs, since the number of

triangles in each partition is vastly reduced compared to the overall number of triangles in the domain.

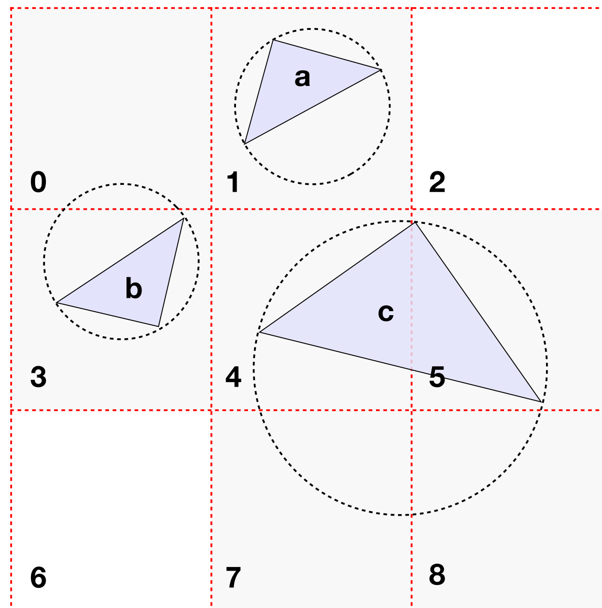


Figure 2.6. Partition 0 conflicts with partition 3 due to the circumcircle of triangle B, so this pair cannot be triangulated simultaneously. Partition 0 is independent of all other partitions, however. Similarly, partition 4 conflicts with partitions 5, 7, and 8 due to triangle C, but could be processed in parallel with partitions 0,1,2,3, or 6. The circumcircle of triangle A stays wholly inside partition 1, and causes no conflict.

2.3.1 Terminology

For clarity, we provide a brief glossary of terms:

Initial points: set of points collected from each partition and used for generating the initial mesh. The number of initial points is much smaller than the total number of points in the domain.

Initial mesh: the Delaunay triangles which are generated from the initial points.

Global mesh: the Delaunay triangles generated at a given time across the entire domain.

Partitions: rectangular regions resulting from dividing domain axes into sections, forming a rectilinear grid that covers the domain. Each partition manages its own set of points and triangles.

Independent partitions: Partitions are *independent* if their interior triangles can be triangulated simultaneously, producing the same result as a serial implementation.

Conflicting partitions: two partitions *conflict* if there exists a triangle whose circumcircle intersects both partitions, meaning they are not independent.

Active partitions: the set of partitions that are currently being triangulated. We must ensure these partitions are independent.

Interior triangles of a partition: The set of triangles that have circumcircles wholly contained inside the partition.

Boundary triangles of a partition: The set of triangles with circumcircles that intersect, but are not wholly contained by the partition.

Exterior triangles of a partition: The set of triangles that have circumcircles that do not intersect the partition.

Active triangles: The triangles belonging to currently active partitions.

Finalized partitions: A partition P can be *finalized* if all points $p \in P$ have been triangulated. Interior triangles of finalized partitions can be written to storage and removed from memory.

Finalized triangles: Triangles that have been written to non-volatile storage.

Major partition: an element of the coarse top-level partitioning used with multi-master TIPP, as described in section 3.1.

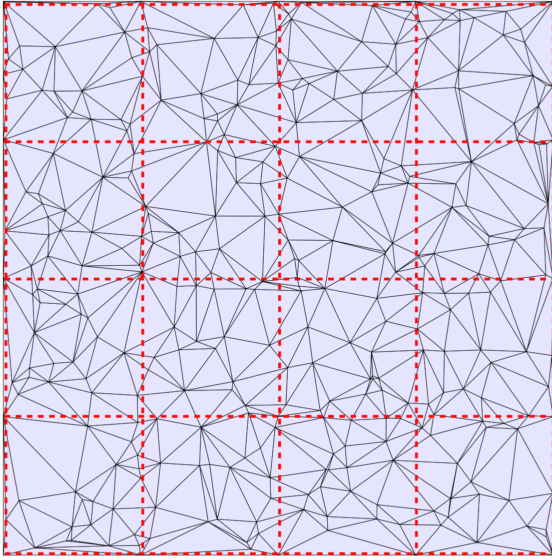
Minor partition: an element of the fine second-level partitioning used with multiple-master TIPP, as described in section 3.1.

2.3.2 The TIPP algorithm

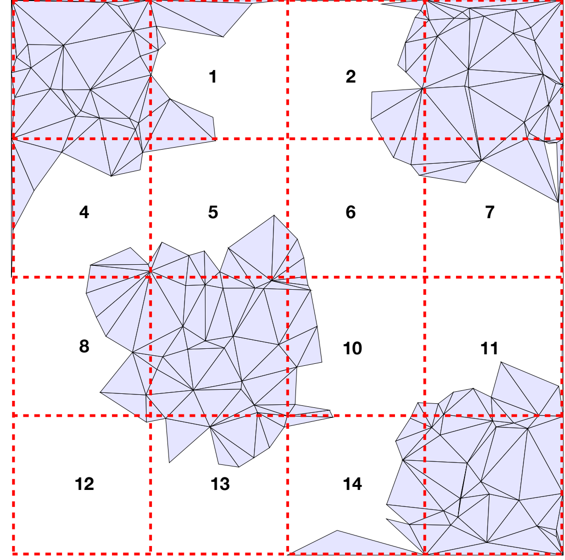
In this section we present a novel algorithm to generate very large Delaunay Triangulations in parallel, as shown in Algorithm 1, and depicted in figure 2.7.

There are three prework tasks. First, points in the domain are distributed to the partitions that geometrically contain them. Second, for each partition, points are sorted

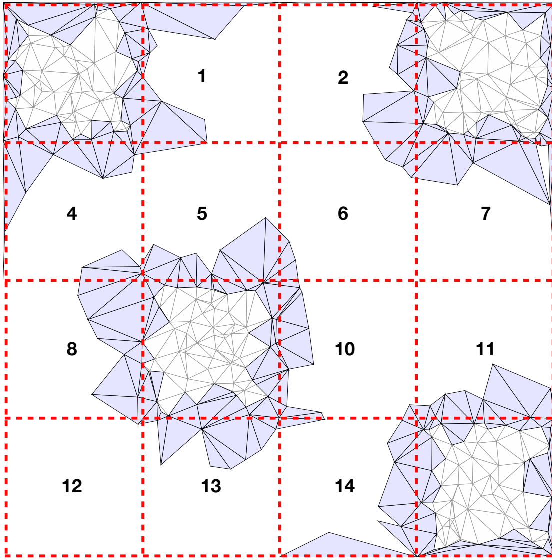
according to x coordinate. Third, we randomly select a set of k points from each partition to form a set of *initial points*. Selecting points in this fashion reduces the occurrence of *sliver* triangles, as described in section 2.4.



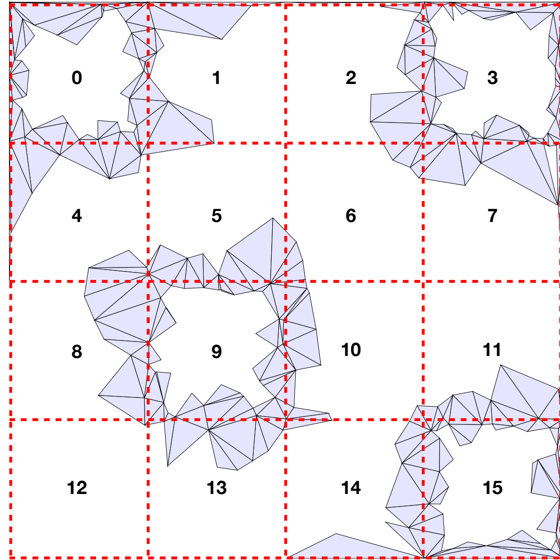
(a) The initial mesh is generated from the set of initial points taken equally from all partitions. The intersections are calculated between triangle circumcircles and partitions.



(b) We choose four active partitions numbered 0, 3, 9, and 15. The active triangles belonging to these partitions form four disjoint areas that are independent and will be assigned to processors for further triangulation.

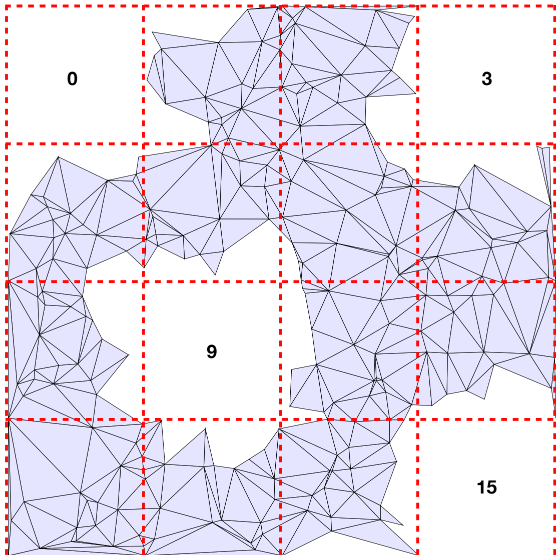


(c) Four groups of triangles have been triangulated in parallel. The circumcircles of light gray triangles (the interior set) are contained wholly within their partition and can be finalized. The dark gray triangles (the boundary set) will require further processing.

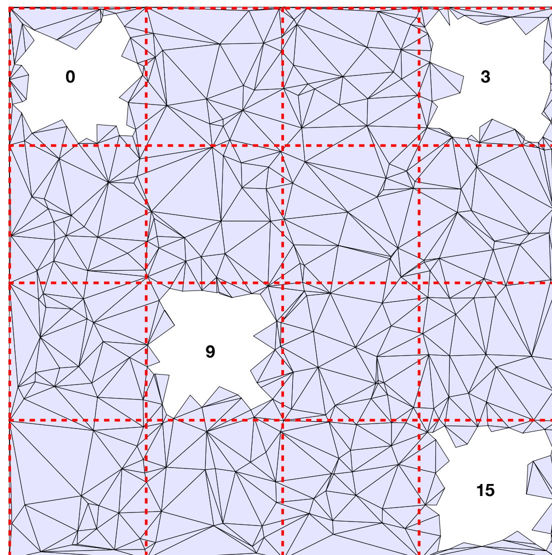


(d) The interior triangles from figure 2.8(c) have been finalized (stored to disk). The boundary triangles of partitions 0, 3, 9, and 15 will later be assigned to other partitions for further triangulation.

Figure 2.7. Parallel Delaunay Triangulation — example of TIPP with 1000 points.



(e) The inactive triangles from figure 2.8(a) did not have circumcircles intersecting our active partitions, but will be further refined by the next stage.



(f) The boundary triangles from figure 2.8(d) are joined with the inactive triangles from figure 2.8(e). The resulting set is ready for the next stage, beginning with the selection of new active partitions.

Figure 2.7. Parallel Delaunay Triangulation — example of TIPP with 1000 points.

We begin the main algorithm by constructing the initial mesh IM from the initial points and two *supertriangles* constructed from the four corners of the domain. (These corner points can be removed after the algorithm finishes, along with other artificially added points.) Since we choose only a small number of points from each partition, the number of triangles in the initial mesh is much smaller than the set of triangles in the final triangulation.

Following lemmas 1 and 2, we assign a triangle to a partition P_i if the circumcircle of the triangle intersects P_i. We say the triangle *belongs* to P_i. A triangle may belong to more than one partition.

In order to triangulate in parallel we need to find a set of independent partitions $\mathcal{IP}$ such that triangulation of a partition $P_i \in \mathcal{IP}$ will not affect any other partition $P_j \in \mathcal{IP}$. Beginning with the initial mesh (shown in figure 2.8(a)), we determine the partitions that *conflict*, meaning they are not independent. More specifically, two partitions conflict if there exists a triangle t with a circumcircle that intersects both partitions, meaning that t belongs

Algorithm 1: Overview of TIPP

Input: set of partitions $\mathcal{P} = P_0 \dots P_m$ covering domain D, and $Points \in D$.

1 ***Prework:*** Compute partitioned points $Points_p = points_0 \dots points_n : points_i \in P_i$

2 ***Prework:*** For each P_i, sort $points_i$ according to x coordinate.

3 ***Prework:*** Compute initial points $Points_I$ (Points in the domain are distributed to partitions based on their coordinates. From each partition, we randomly select k points to form the set of initial points).

Output: *All triangles in T are Delaunay.*

4 **Step 1**: generate the initial mesh (IM) (see figure 2.8(a)) using initial points ($Points_I$) and two super-triangles (see figure 2.9(a)) constructed from the four corners of the domain and additional points along the edge of the domain

5 $T \leftarrow IM$

6 **while** $\mathcal{P} \neq \varnothing$ **do**

7   **Step 2**: generate the set of independent partitions $\mathcal{IP} \in \mathcal{P}$ based on the intersection between partitions and the circumcircles of triangles in T.

8   **Step 3**: for each partition $P \in \mathcal{IP}$ collect triangles in T that belong to P, using the circumcircle intersections computed in step 2.

9   **Step 4**: Triangulate each partition $P \in \mathcal{IP}$ in parallel on the worker nodes, and update T accordingly. If the number of available processes np is less than $|\mathcal{IP}|$, we schedule partitions in shifts of size np.

10   **Step 5**: write finalized triangles to external storage and update $\mathcal{P}$:

11   let $T_f \subset T$ be the finalized triangles.

12         $T \leftarrow T - T_f$;

13   Finalize(T_f);

14         $\mathcal{P} \leftarrow \mathcal{P} - \mathcal{IP}$;

15 **end**

16 **Step 6**: (Optional) Remove any triangles that use artificial vertices added in step 1.

to both partitions. Algorithm 2 shows in detail how we generate $\mathcal{IP}$.

The edge and vertex neighbors of P_i are those partitions that share either an edge border or a corner vertex with P_i. These partitions are almost certain to have triangles belonging to both P_i and to themselves. If P_i was chosen as a member of $\mathcal{IP}$, then those edge and vertex neighbor partitions that conflict with P_i must not be in $\mathcal{IP}$. This explains the checkerboard appearance of some of the diagrams in figure 2.7. However, TIPP does not depend on this observation, and rigorously checks for triangles that actually cause a conflict between partitions.

For example, in figure 2.8(b), partitions 4, 5, 6, 8, 10, 12, 13, and 14 clearly con-

Algorithm 2: Determining independent partitions.

Input: Mesh $M = (triangles, points)$, set of partitions P in domain
Output: set of independent partitions IP

```
1  Let CP(p) be the set of partitions that conflict with a partition p.;
2  ∀p : p ∈ P, CP(p) ← ∅
3  for each triangle t ∈ M do
4  |  Determine a set of partitions P_t = {p_0 ... p_n} : circumcircle(t) ∩ p_i ≠ ∅.;
5  |  for each partition p_i ∈ P_t do
6  |  |  for each partition p_j ∈ P_t do
7  |  |  |  if p_i ≠ p_j then
8  |  |  |  |  CP(p_i) ← CP(p_i) ∪ p_j;
9  |  |  |  end
10 |  |  end
11 |  end
12 end
13 ∀p_i : p ∈ P, Unique(CP(p_i)) remove duplicates;
14 Let SP be a list of p_i ∈ P, sorted according to |CP(p_i)|, in increasing order.;
15 SP = Sort(P) according to |CP(p_i)|, in increasing order. ;
16 Let IP be the set of independent partitions.;
17 Let C be the set of partitions that conflict with some element of IP ;
18 C ← ∅;
19 IP ← ∅;
20 while (C ∪ IP) ≠ P do
21 |  let p = head(SP) (the partition in SP with the fewest conflicts);
22 |  if p ∉ C then
23 |  |  IP ← IP ∪ p ;
24 |  |  C ← C ∪ CP(p) ;
25 |  end
26 end
```

flict with partition 9, meaning that if partition 9 is chosen as a member of $\mathcal{IP}$, then the aforementioned partitions will not be placed in $\mathcal{IP}$.

After determining $\mathcal{IP}$, partitions that are not in $\mathcal{IP}$ cannot be active in the upcoming processing stage, and will have to wait for a future round. For example, in figure 2.8(b), partitions 1, 2, 4, 5, 6, 7, 8, 10, 11, 12, 13, and 14 are inactive, while active partitions 0, 3, 9, 15 are being processed in parallel. Notice that these four active partitions have triangles associated with them that are not wholly inside the partition. Recall from section 2.2 that the boundary set of a partition is the set of triangles with circumcircles that cut the partition

border. Because each partition only processes points that fall within its borders, the effect of triangulating these points can extend no further than the boundary set.

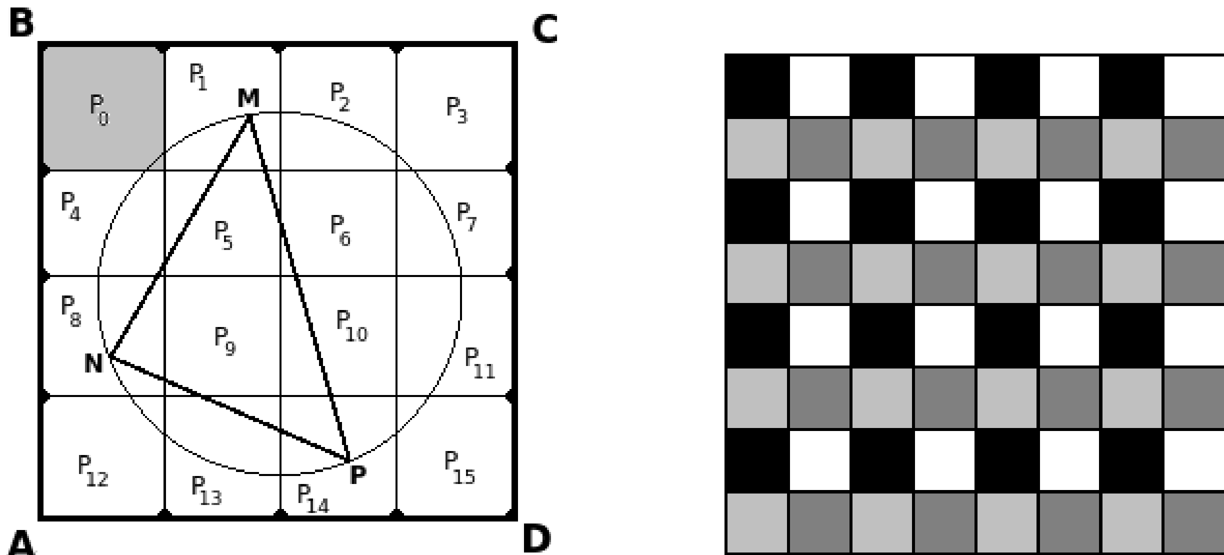
Figure 2.7.c presents four active partitions after triangulation. More triangles (shown in light grey) have been added to the interior set of the partitions. Recall that the interior set of a partition is those triangles with circumcircles wholly contained within the partition. Because all points within each active partition have now been inserted into the triangulation, we can finalize the interior set of each partition, writing these triangles to disk and removing them from memory. This not only conserves memory, but also reduces triangle search costs for the remainder of the triangulation.

Partitions 0, 3, 9, and 15 in figure 2.8(d) have been finalized, but the boundary sets of these partitions must continue to be processed, since they can be affected by the insertion of points in nearby partitions, because their circumcircles intersect with them. We simply give these triangles to nearby partitions that intersect their circumcircles for further processing. Figure 2.8(e) shows the global mesh before the boundary sets from partitions 0, 3, 9, and 15 are added. Notice that (due to lemma 5) the boundary sets fit cleanly in the cavities surrounding the finalized partitions, producing a new global mesh, shown in figure 2.8(f).

We now repeat the process, beginning with identifying active partitions using the triangles from the updated global mesh. These partitions are processed as before, resulting in newly finalized interior triangles and a global mesh that is updated with boundary triangles.

When all partitions are finalized, all interior triangles have been finalized, and all points have been inserted. We can now finalize any remaining boundary triangles. Lemma 4 shows that the outside border of a partition's interior triangles and the inside border of the partition's boundary triangles will fit, even when the boundary triangles have been updated. This means that interior and boundary triangles are Delaunay and there is no need for stitching or a complex merging process. The pieces all fit perfectly together.

For many applications, we will want to remove the artificial points introduced during step 1 of Algorithm 1, since they are not genuine sample points. They do not have data



(g) A worst case for the TIPP algorithm in which the circumcircle of a large triangle $\triangle MNP$ intersects with all partitions in the domain, effectively serializing the triangulation. Only one partition can be processed at a given time.

(h) An ideal case for an 8×8 partitioning. No circumcircles span more than two partitions, allowing triangulation in four stages of 16 partitions each. Stages are shown in different shades of gray.

Figure 2.8. A worst case and ideal case of Parallel Delaunay Triangulation algorithm.

values associated with them, so triangular cells that use such points as vertices cannot be used for interpolation. One solution is to simply remove the triangles that are incident to these artificial points. Since the removed triangles may leave a ragged border, we could also add missing edges of the convex hull [9] to repair the triangulation. Davy, et al.[29] report an algorithm with upper bound $O(n^2)$, the same as serial Bowyer-Watson triangulation, but note that in practice the complexity is closer to the lower bound of $\Omega(n_b)$, where n_b is the number of points on the convex hull. The quickhull algorithm [9] has worst case $O(n\ log\ v)$, where n is the number of input points, and v is the (much smaller) number of points on the hull.

2.4 Improving Performance of TIPP

We can improve the performance of our algorithm by increasing the degree of parallelism, and also by optimizing the search operation for the Delaunay process on the worker nodes.

2.4.1 Increasing Parallelism

The TIPP algorithm is not fully parallel. The degree of parallelism depends on the number of active partitions, which in turn depends on the number of triangles. A small triangle count increases the presence of large triangles with circumcircles that intersect more than one partition. Such triangles introduce dependencies between partitions, decreasing opportunities for parallelism.

For example, figure 2.8(g) consists of a large triangle MNP with a circumcircle that intersects all partitions in the domain $ABCD$. In this case all partitions ($P_0 ... P_{15}$) are conflicting partitions. This means only a single partition can be active at one time. In this special case, the algorithm degrades into sequential triangulation. On the other hand, the number of initial triangles in figure 2.8(a) is roughly 320, so the number of active partitions is much improved (e.g. four active partitions in figure 2.8(b)).

For this reason, an initial mesh with many triangles will improve performance during parallel processing. However, this mesh is computed serially, so we should be careful to implement this task efficiently. We choose the initial points so that each partition contributes the same number of points, which improves the resulting mesh. Large triangles or long, thin *slivers* are far less likely, increasing performance for a mesh of a given size.

Delaunay implementations often use one or more *supertriangles* that contain the entire domain, and are then removed from the final triangulation. We chose to use two supertriangles ABC and ACD, as shown in figure 2.9(a). Unfortunately, this introduces many slivers with edges $\overline{AB}$, $\overline{BC}$, $\overline{CD}$, or $\overline{DA}$. These triangles directly intersect many partitions. Even worse, such triangles have very large circumcircles, which introduces more dependencies between partitions, and also increases search costs. Worse still, once the long edges of slivers are introduced into the triangulation, they provide opportunity for further slivers to be created, and performance degrades substantially.

The solution for this problem is to introduce more points along the exterior edges of the supertriangles. Figure 2.9(b) shows the addition of points $A_1 ... A_3$, $B_1 ... B_3$, $C_1 ... C_3$, and

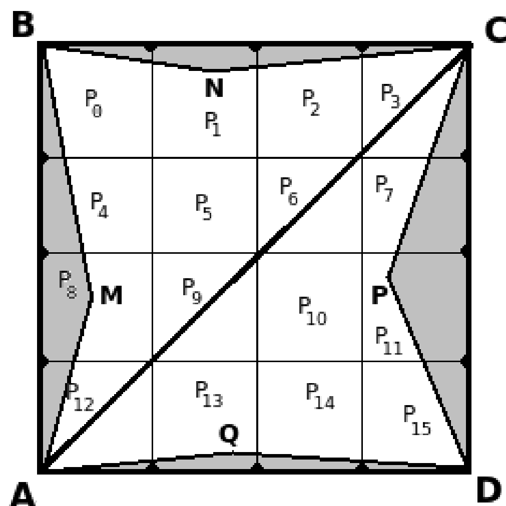
$D_1...D_3$ along edges $\overline{AB}$, $\overline{BC}$, $\overline{CD}$, and $\overline{DA}$.

2.4.2 Improving Triangle Search

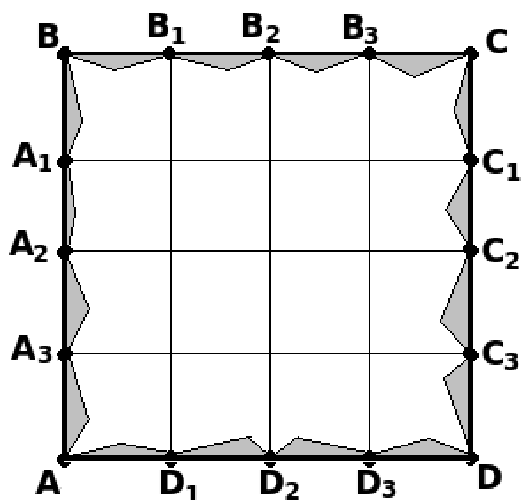
The most expensive step of Delaunay triangulation is to find the triangles whose circumcircles enclose a newly inserted vertex. These triangles later will form a polygon or a *cavity* (shaded triangles in figure 2.1) for further triangulation. In a large-scale domain, the number of triangles in a domain could reach millions or billions. At this scale, triangle search becomes very time consuming.

Our partitioning approach addresses this bottleneck in two ways. First, the triangle list for each partition is much shorter than in the serial case, since these lists only record triangles relevant to each partition. Second, partition triangle lists can be searched independently in parallel. Both factors contribute to the performance improvements observed in section 3.3.

We further enhance performance within each partition by using the sweep line algorithm [34] to process vertices in order sorted by x coordinate. This allows us to determine



(a) Two super triangles $\triangle ABC$ and $\triangle ACD$ that cover all points in the domain. Four sliver triangles $\triangle AMB$, $\triangle BNC$, $\triangle CPD$, and $\triangle AQD$ degrade DT performance.



(b) Addition of points $A_1...A_3$, $B_1...B_3$, $C_1...C_3$, and $D_1...D_3$ along edges to reduce sliver problems and improve the parallelism.

Figure 2.9. Super triangles introduce sliver triangles.

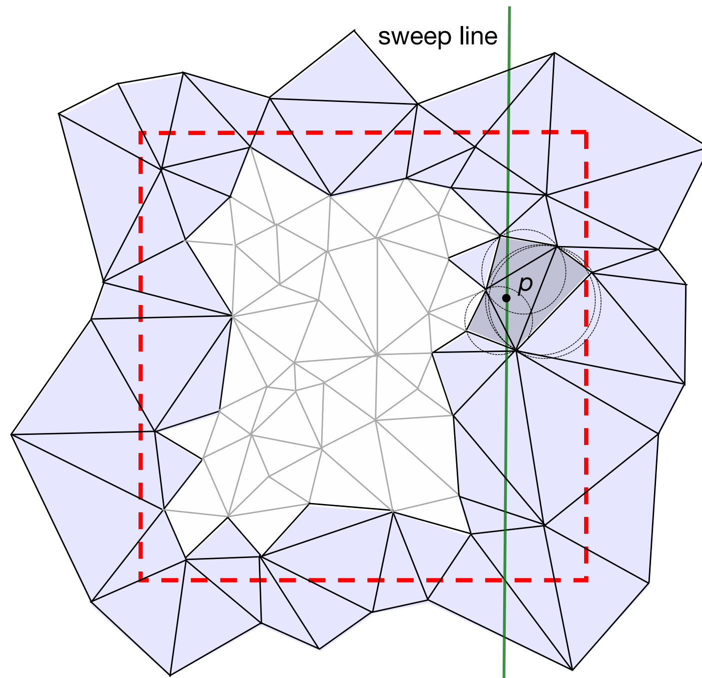


Figure 2.10. An active partition during triangulation. The new inserted vertex P is on the vertical sweep line. The light gray triangles are stored to external memory because their circumcircles can not reach the sweep line or partition border (square). The shaded triangles form a cavity to be retriangulated. The boundary triangles (drawn in black) will be further processed as members of other partitions.

finished triangles, delete them from the triangle list, and write them to external storage. In figure 2.10, circumcircles of light grey triangles do not enclose point p, and their largest x coordinate is smaller than that of the sweep line. Therefore, vertices to the right of the sweep line will not be enclosed by any circumcircle of the grey triangles. We can consider them to be part of the final triangulation, and write them to storage. Since they are also removed from the partition's triangle list, future triangle search operations will benefit from the shorter list.

2.5 Implementation

We implemented TIPP using C/C++ and MPI (Message Passing Interface)[38]. The MPI programming model greatly simplifies development of code that runs in parallel over a cluster of ordinary machines, and also allows an arbitrary number of tasks to be scheduled

on each machine. In contrast, shared memory models such as *TBB* [73], *OpenMP*[28], and *pthreads* [61] do not easily extend over multiple nodes.

This section describes some of the choices made during the implementation of TIPP.

2.5.1 Data Structure

The intersection between circumcircles of triangles in the initial mesh and partitions in the domain requires a data structure that can answer two types of query. First, given a triangle, we need a list of partitions that intersect with that triangle's circumcircle. Second, given a partition, we need a list of triangles with circumcircles that intersect the partition.

We could choose either an array-based implementation or a dynamic data structure. Arrays are convenient for GPU computing [65], a possible future extension of TIPP. Some researchers [81, 18, 75, 74] were able to use arrays for elements (triangles or tetrahedra) because the maximum number of elements could be calculated in advance [50]. This is harder for our work, due to the partitioning mechanism. Also, an array implementation cannot easily free the memory used by deleted triangles, which is an important feature of TIPP. For these reasons, we chose a linked list to hold triangles, since we can efficiently reclaim memory when triangles are finalized.

We have not yet investigated the possibilities for dynamic tree-like spatial data structures, such as r-trees [39], and have instead used our own linked list implementation that is convenient for our application. Because the scope of triangle search is already greatly reduced by partitioning, and also by the sweep-line algorithm described in section 2.4.2, the benefits of a tree implementation are uncertain.

2.5.2 Parallelism

Our distributed system includes multiple worker nodes and a master node, which can also serve as a worker node. The computation is distributed, and the data access is centralized. The entire dataset is stored in one directory on the master node, made available to all worker nodes via NFS[87].

The master node is responsible for selecting active partitions based on the initial mesh. Each active partition will then be assigned to an MPI process for further triangulation on a worker node. However, the number of active partitions may overwhelm the number of available processes. In this case, we schedule partitions onto available processes in groups until all partitions have been triangulated.

When worker nodes complete their jobs, they use NFS to store the finalized triangles and also return the remaining (boundary) triangles to the master. The boundary triangles will be assigned to non-finalized partitions by the master as it prepares the next stage of active partitions. This set of boundary triangles is quite small compared to the set of interior triangles that were finalized, which helps TIPP scale to datasets with billions of triangles.

2.6 Experimental Results

We tested TIPP on a cluster running on the Chameleon Cloud Testbed [20]. Our storage node with two Intel® Xeon® E5-2650 v3 processors @ 2.30GHz, 64GB RAM and 2T HDD served data via NFS and was also both a worker and master node. Nine additional worker nodes running 64 bit Linux had two Intel® Xeon® E5-2670 v3 @ 2.30GHz processors (16 cores total), 128G RAM and a 250GB HDD.

We use 2D datasets that are generated from the *qhull* utility [8, 9]. Point coordinates are generated on the range $[0, 1]$ and converted from text into binary representation for speed and storage efficiency. The point coordinates are separated into partitions in the domain and sorted based on x coordinate. The coordinate files belonging to partitions will later be joined into a single large file. The size of the datasets ranges from 500,000 points (roughly one million triangles) up to 500 million points (around one billion triangles). Section 3.3.2 also describes performance for an 8 billion point triangulation. In all cases, there are no obstructions such as airplane wings or similar objects, and points are evenly distributed throughout the domain.

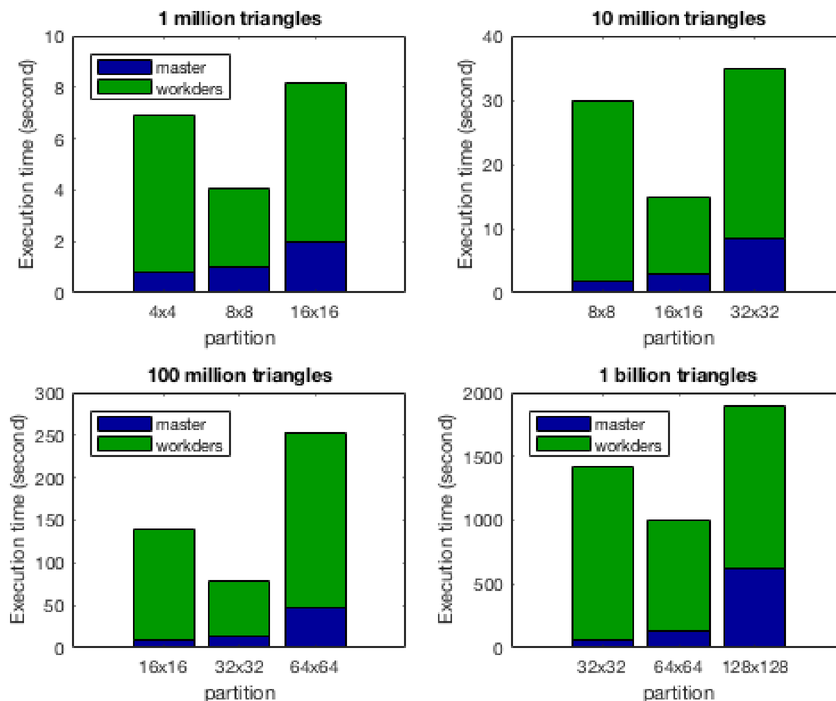


Figure 2.11. Execution times of Parallel Delaunay Triangulation with four different datasets (500K, 5M, 50M, and 500M points) or (1M, 10M, 100M, 1B triangles) while changing partitioning granularity. Each bar represents the execution times of master (lower) and worker nodes (upper). $n \times n$ *is the number of partitions.*

2.6.1 Performance

The performance of TIPP depends on the dataset size (number of points in the domain), number of the partitions, number of compute nodes, and the performance of each node. The domain is divided into $n \times n$ partitions where n is in the range (4, 8, 16, 32, 64, 128). Figure 3.4 shows the execution times of TIPP with different numbers of partitions. The best performance falls in the middle of partition granularity. Indeed, there is a trade-off between the granularity of the partitioning and performance. Coarse–grained partitionings improve the speed of the master, since fewer partitions mean fewer triangle–partition intersection tests are required. However, coarse partitionings hurt triangulation performance on the workers because large partitions are not as effective in reducing the scope of point-circumcircle search. Conversely, fine–grained partitionings burden the master node while

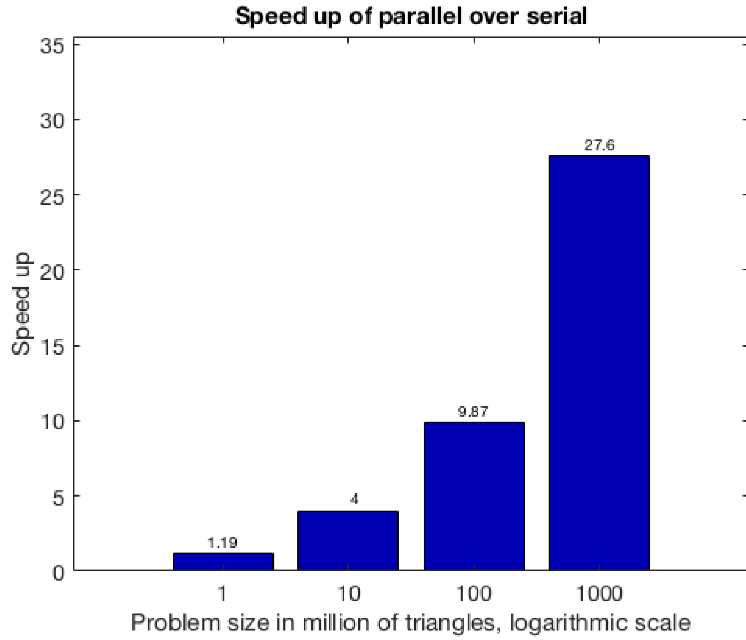


Figure 2.12. Speed up of Delaunay triangulation with different domain sizes (1 Million, 10M, 100M, and 1000M triangles). The X axis is logarithmic.

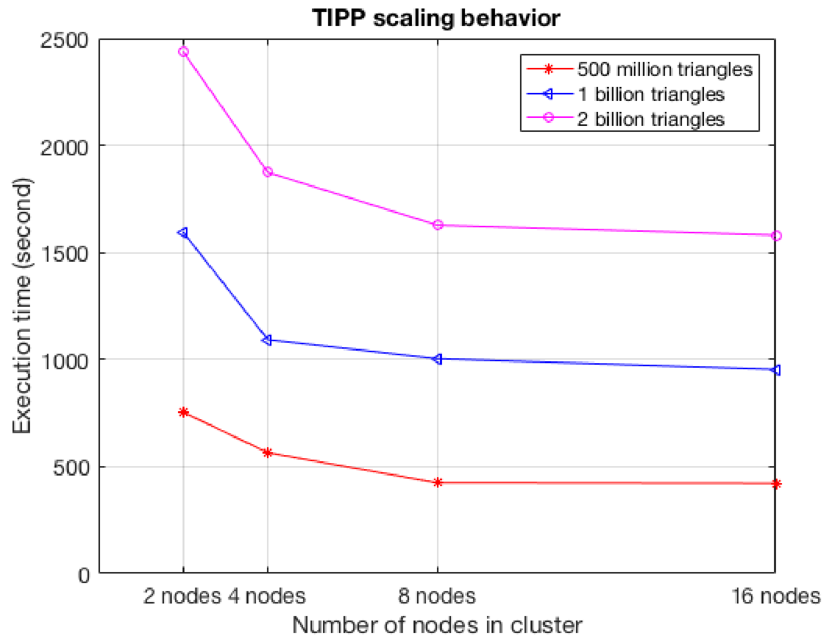


Figure 2.13. TIPP Execution times for increasing numbers of nodes.

improving computational performance on the workers.

However, computational performance on the workers is only part of worker performance. I/O is another important factor. Fine–grained partitionings imply a large number

of small communications between master and worker nodes, which will incur larger overall latency costs compared to a coarser partitioning. For this reason, partitionings that are too fine–grained can increase overall worker execution time.

Figure 3.4 shows TIPP performance for different partitioning granularities and dataset sizes. The execution time due to the master node is shown as a the darker region at the base of each bar. As partitionings become more fine–grained, this cost increases. Worker execution costs are lowest in the middle case for all datasets, but are higher for the most fine-grained partitionings, due to I/O costs.

We also measure the speedup (figure 2.12) between Parallel TIPP running on ten nodes with 16 cores apiece and a separate serial implementation. The best performance of parallel TIPP from figure 3.4 is compared with serial TIPP for four different dataset sizes. As expected, speedup improves as the dataset size increases, since overhead costs become proportionally less significant.

Figure 2.13 shows the effect of an increasing number of compute nodes on overall execution time for different dataset sizes. For each size, we chose the partitioning with best performance from figure 3.4. The current implementation is not able to take advantage of more than about eight nodes, probably due to I/O contention and load on the master node. The flattening of the curves past eight nodes indicates that performance is dominated by non-parallelizable costs, likely involving our single master node. That we see similar behavior for all three dataset sizes indicates that these non-parallelizable costs are not $O(1)$, but instead dependent on data volume.

2.6.2 Processing a Large Dataset

Qhull [8, 9] can generate at most two billion points in the domain. However, in order to produce even larger datasets, we can replicate points in this domain to adjacent domains, but must check the resulting dataset to make sure there are no duplicate points. In our experiment, we make a 4×4 domain with 8 billion points and generate roughly

16 billion triangles, which may be the largest existing 2D mesh. The domain is divided into $128 \times 128 = 16384$ partitions. In the first stage of TIPP, almost one fourth of these partitions are independent and ready for triangulation. However, the number of active partitions processed simultaneously depends on the number of MPI process slots available. For this experiment, each of our 10 nodes was set to 24 slots, using a round-robin scheduling policy to assign processes to nodes. Execution took 19988 seconds overall, with 1478 seconds initially taken by the master node to distribute partitions.

2.7 Discussion

The TIPP algorithm is designed to work in a distributed environment in which the master node prepares the independent (active) partitions and sends them to worker nodes for triangulation in parallel. With this in mind, we analyze both the scalability and load balancing issues in the algorithm.

2.7.1 Scalability

The TIPP algorithm is able to handle very large datasets because the processing and memory requirements are split across the nodes of the cluster, partially escaping the limits of a single machine approach. When dealing with a very large number of points, the domain is simply divided into many more partitions.

The number of active partitions given to worker nodes depends on the total number of partitions in the domain, with roughly 25% being processed during each stage of the algorithm regardless of the granularity of the grid (see table 2.1). Therefore, to take advantage of more worker nodes, we can simply increase the number of partitions by using a finer grid over the domain.

However, the TIPP implementation relies upon a single master node to identify the next active set of partitions. This is a computationally expensive process, requiring a large number of rectangle-triangle and rectangle-circumcircle intersection tests. Worker nodes

access their assigned partitions via an NFS server running on the master, which places further load on this single machine.

To alleviate the congestion, we envision several approaches. First, we could reduce the number of triangles and points in the initial mesh, reducing the cost of determining active partitions. Unfortunately, this approach would reduce parallelism because the number of active partitions in the first stages of TIPP would also be reduced, due to the prevalance of large triangles with circumcircles spanning several partitions.

Another approach is to raise master performance is to share master's workload to several sub-master processes. Instead of triangulation a large amount of points to prepare independent partitions as problem size increasing, master process simply triangulate a coarse-grained initial meshes at the first level. The second level of triangulation will further carry out the fine-grained triangulation. The detail of how to triangulate in two-level parallelism will be described in next chapter.

Table 2.1. Number of active partitions based on the total number of partitions

Number of partitions active during first phase	4	16	61	251	1020
Total number of partitions	16	64	256	1024	4096

2.7.2 Load balancing

We used qhull utilities [8, 9] to generate point datasets uniformly distributed over the domain. Since partitions are same–sized rectangles, the number of points in each partition is roughly similar, implying similar execution times for each node. For non-uniform point distribution, some nodes may take much longer than others, all else being equal. Happily, we expect to greatly reduce this load imbalance by using a finer grid, when appropriate. With smaller partitions, the number of points in each partition will be closer. Also, since smaller partitions take less time to compute, idle workers will not wait as long for more work.

2.8 Chapter Summary

We have developed a novel algorithm named TIPP to generate Delaunay triangulations for very large scale datasets. We have shown that the dataset domain can be decomposed into independent partitions that can be processed in parallel. TIPP also improves performance by identifying sets of triangles that can be finalized early, removing those triangles from memory, and reducing the cost of triangle search. The result is a distributed algorithm able to generate roughly 16 billion triangles.

We have also shown that the results of triangulated partitions fit perfectly together, preserving the Delaunay criteria. We do not require a stitching process that would introduce non-Delaunay triangles between partitions.

TIPP significantly improves the performance of Delaunay triangulation, bringing extreme-scale meshes into the realm of the feasible through its distributed approach. However, there is still room for improvement. We would like to implement a method of repairing the mesh boundary after removing artificial vertices and their triangles. Also, because we use partitions of uniform size, we do not yet perform load balancing between worker nodes. A third issue is the master node. When the number of partitions is large, the master node becomes a bottleneck, causing worker nodes to wait for new work. A solution for this problem is to parallelize the master process tasks, which will boost performance, efficiency, and the size of datasets feasible for scientific researchers.

CHAPTER 3

PARALLEL DELAUNAY TRIANGULATION FOR LARGE-SCALE DATASETS USING TWO-LEVEL PARALLELISM

In the previous chapter, we developed a novel algorithm – *Triangulation of Independent Partitions in Parallel (TIPP)* to deal with very large DT problems without requiring communication between workers while still guaranteeing the Delaunay criteria. However, the original TIPP algorithm relies upon a single master node to identify the set of partitions that can be scheduled simultaneously. We evaluated performance using synthetic datasets containing billions of points with a uniform distribution over the domain. Results indicated that when the number of partitions is large, the master node becomes a bottleneck, causing worker nodes to wait for new work. In this chapter, we further develop the TIPP algorithm to employ multiple master processes, distributing computational load across several machines. This new design improves both performance and scalability.

3.1 Multiple Masters

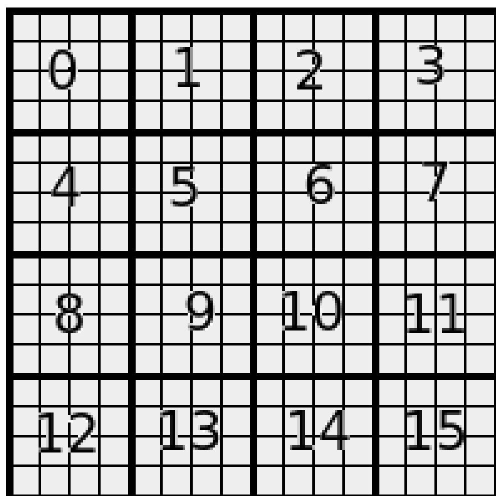
While TIPP worker processes are tasked only with triangulation, the master process must repeatedly generate the next set of independent partitions that are assigned to workers. Because this job requires circumcircle-partition intersection tests, it is computationally expensive. This burden increases significantly when finer-grained partitionings are employed to improve parallelism with larger datasets. In our previous chapter, we found that using only a single master process became a performance bottleneck, due to both computational and I/O costs.

3.1.1 Multiple Masters for Uniform Point Distribution

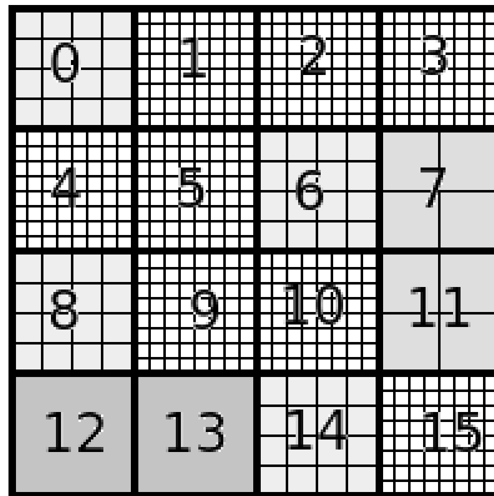
Our solution for this problem is to share the master's burden over several *sub-master* processes. We divide the worker processes into groups (actually *MPI Communicators*) that are each served work by a single sub-master. A single master process remains, but its job is now to assign partitions to sub-master processes, so they may in turn piece out tasks to the worker processes in their group. Because the master now works with a coarse-grained partitioning, its burden is much reduced. However, workers can be given much smaller partitions, due to our two-level partitioning scheme.

As shown in figure 3.1(a), multi-master TIPP uses a relatively coarse *major partitioning* to assign work to sub-masters. Each major partition is further subdivided using a *minor partitioning*, allowing sub-masters to assign fine-grained tasks to workers for improved parallelism. Our notation for two-level partitionings is $[n \times n] \times [m \times m]$ where $[n \times n]$ is the number of major partitions and $[m \times m]$ is the number of minor partitions.

In effect, this two-level partitioning forms a very shallow tree, where each tree node



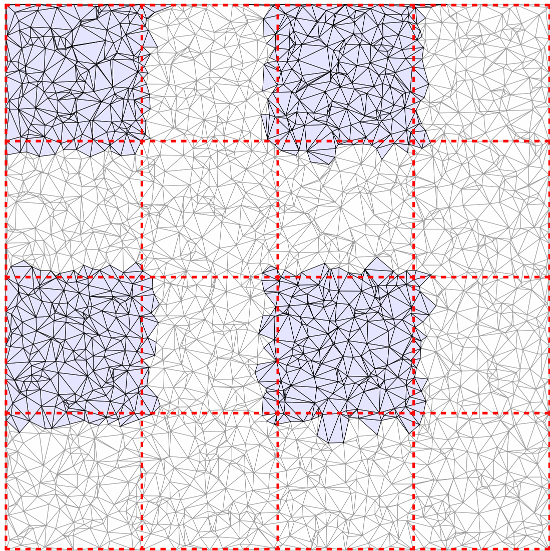
(a) Two level partitions with uniform point distribution. The domain consists of 16 major partitions, each of which includes 16 minor partitions.



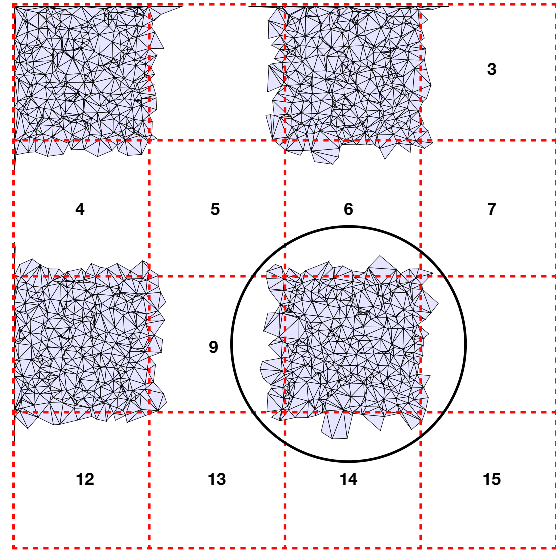
(b) Two level partitions with non-uniform point distribution. The domain consists of 16 major partitions, each subdivided into a number of minor partitions depending on point distribution.

Figure 3.1. Two level partitioning with uniform and non-uniform point distribution.

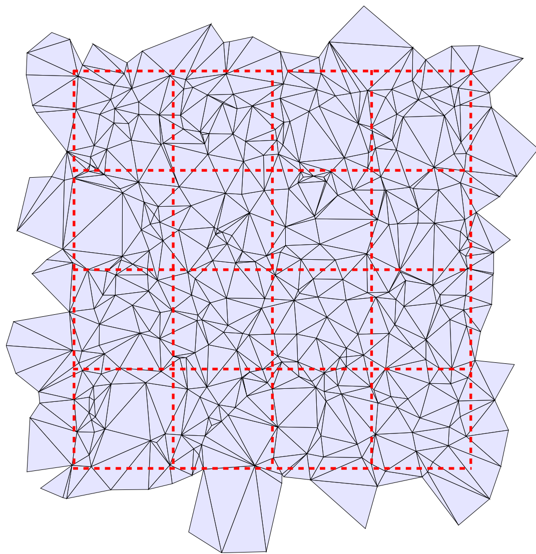
has a very large number of children. Using only two levels, we address the scaling problems of single-master TIPP while retaining the O(1) complexity of regular partitionings.



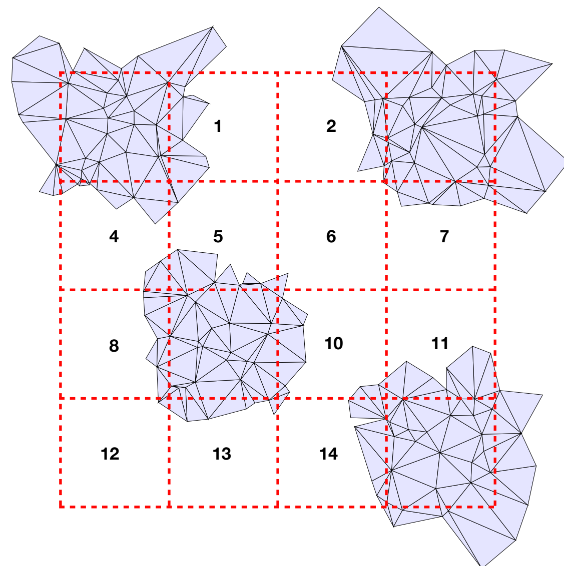
(a) The initial mesh, including active triangles (darker gray) which belong to four independent major partitions (0, 2, 8, 10), which are about to be triangulated.



(b) Independent major partitions are refined with some additional points. Each major partition is divided into minor partitions for Delaunay triangulation in parallel.

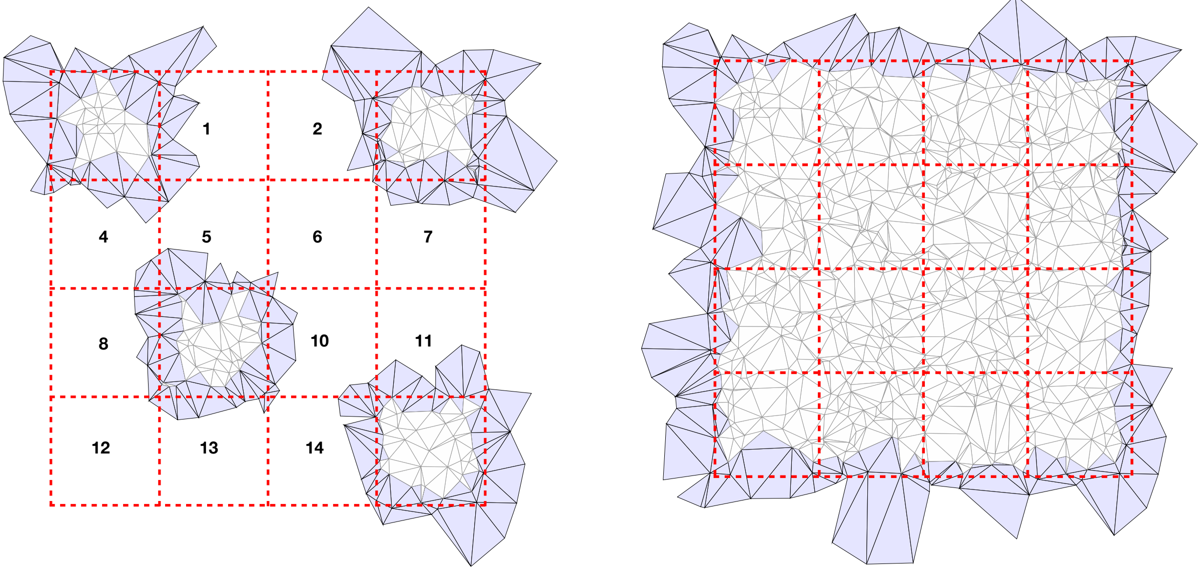


(c) Detail of major partition 10, taken from figure 3.2(b). Major partition 10 is subdivided into 16 minor partitions.



(d) Triangulation in progress. We show four independent minor partitions (0, 3, 9, 15) chosen for parallel refinement.

Figure 3.2. Multi-master TIPP — An example with 10,000 points.



(e) The four independent minor partitions are triangulated. Darker gray triangles are boundary triangles, while light gray triangles form the interior of the minor partitions.

(f) After all minor partitions in major partition 10 are triangulated, the minor partition results are assembled into the final result for the major partition, including interior (light grey) and updated boundary triangles (dark gray).

Figure 3.2. Multi-master TIPP — An example with 10,000 points.

Multi-master TIPP determines independent partitions in a manner similar to the single-master case described in section 2.3, but applied to two levels, as shown in figure 3.2. Algorithm 3 describes the two-level parallelism (multi-master TIPP). The master first constructs a coarse initial mesh, and uses it to identify independent major partitions. Independent major partitions are then chosen (figure 3.3(a)) and sent to the sub-master processes for further triangulation. Each sub-master process then refines an initial mesh for their major partition, and uses it to determine a set of active minor partitions to be given to worker processes in the sub-master's group. Major partition 10, circled in figure 3.3(b), is detailed in the subsequent figures. When workers have completed triangulation of this partition, the responsible sub-master sends the boundary set to the master, while interior triangles are finalized to disk. The entire procedure is then repeated.

Algorithm 3: Overview of multi-master TIPP

Input: A set of partitions $\mathcal{P} = P_0 \dots P_{m-1}$ covering domain D. Partition $P_i \in \mathcal{P}$ is divided into multiple minor partitions $P_{i0} \dots P_{i(n-1)}$

Output: *All partitions $P_i \in \mathcal{P}$ are triangulated and Delaunay*

1 Step 1(on master): generate the initial mesh for the domain (fig. 3.3(a)).
2 **while** $\mathcal{P} \neq \varnothing$ **do**
3 Step 2(on master): generate the set of independent major partitions $\mathcal{IP} \subset \mathcal{P}$ (fig. 3.3(a)).
4 Step 3(on master): master sends active major partitions to sub-masters
5 Step 4: triangulate all partitions $P_i \in \mathcal{IP}$ in parallel:
6 **for each** $P_i \in \mathcal{IP}$ **do**
7 Step 4.1(on sub-masters): refine initial mesh for partition P_i (fig. 3.3(b)).
8 Step 4.2(on sub-masters): Assign triangles $t \in P_i$ to all minor partitions $P_{ij} \in P_i$ that intersect the circumcircle of t.
9 **let** $\mathcal{P}_{minor}$ be the set of minor partitions $P_{ij} \in P_i$
10 **while** $\mathcal{P}_{minor} \neq \varnothing$ **do**
11 Step 4.3(on sub-masters): generate independent minor partitions $\mathcal{IP}_{minor} \subset \mathcal{P}_{minor}$ (fig. 3.3(d)).
12 Step 4.4 sub-master sends $\mathcal{IP}_{minor}$ to workers
13 Step 4.5(on workers): triangulate $\mathcal{IP}_{minor}$ (fig. 3.3(e)).
14 Step 4.6 worker sends interior and boundary triangles to sub-master
15 Step 4.7(on sub-masters): finalize the interior triangles of $\mathcal{IP}_{minor}$
16 Step 4.8(on sub-masters):
17 $\mathcal{P}_{minor} \leftarrow \mathcal{P}_{minor} - \mathcal{IP}_{minor}$.
18 Step 4.9(on sub-masters): reassign boundary triangles to new minor partitions that are members of $\mathcal{P}_{minor}$
19 **end**
20 **end**
21 Step 5(on master): master collects boundary triangles from $\mathcal{IP}$ and updates $\mathcal{P} \leftarrow \mathcal{P} - \mathcal{IP}$.
22 Step 6(on master): reassign boundary triangles to new major partitions that are members of $\mathcal{P}$
23 **end**

3.1.2 Multiple Masters for Non-Uniform Point Distribution

Both the single and multi-master algorithms described so far are designed only for datasets with uniform point distributions. However, many important real-world datasets show great density variation across the domain, due to features such as coastlines (see figure

3.7) or concentration of sample points in areas of interest. To handle these non-uniform cases efficiently, we must augment TIPP to balance load between processes.

As shown in figure 3.1(b), the solution for this problem is to choose a minor partitioning for *each* major partition that assigns roughly the same number of points to each minor partition. Since each minor partition is assigned to a single process, the result is relatively balanced load.

Let threshold T be the number of points in a minor partition, and N_i be the number of points in major partition i The number of minor partitions in a major partition i will be MP_i:

$$MP_i = \begin{cases} 0, & \text{if } N_i = 0 \\ 1, & \text{if } 0 < N_i \leq T \\ \sqrt{\dfrac{N_i}{T}}, & \text{if } N_i > T \end{cases} \tag{3.1}$$

Figure 3.1(b) is an example of 4×4 major partitions. Major partitions 1-5, 9, 10 and 15 have more points than others, so they are more finely subdivided. The remaining major partitions have fewer points, so they are partitioned more coarsely.

3.2 Implementation

We implemented TIPP using C/C++ and MPI (Message Passing Interface)[38]. The MPI programming model greatly simplifies development of code that runs in parallel over a cluster of ordinary machines, and also allows an arbitrary number of tasks to be scheduled on each machine. In contrast, shared memory models such as *TBB* [73, 77], *OpenMP*[28], and *pthreads* [66] do not easily extend over multiple nodes.

This section describes some of the choices made during the implementation of TIPP.

3.2.1 Parallelism for Uniform Point Distribution

TIPP includes worker, master, and sub-master processes. The computation is distributed via MPI, while the *Ceph* distributed filesystem [96] handles data I/O for compute

nodes across the cluster.

We implemented both single-master (described in previous chapter) and multi-master TIPP. The first version consists of one master process and many worker processes. Master process execution time for single-master TIPP can be significant when processing a large number of partitions. For this reason, we designed the multi-master version described in section 3.1. Figure 3.3 illustrates process behavior across several steps.

In the top row of figure 3.3, the master process is preparing the set of active partitions for the sub-master processes, which must wait until the master is done. Although the sub-masters are idle, the coarseness of the major partitioning means the wait will be brief. In the bottom row of figure 3.3 the sub-masters now prepare active partitions for the worker processes in their group.

We add two more details to this description. First, the number of worker processes in each group is adjusted dynamically in response to the number of active major partitions. At the tail end of the calculation, the number of active major partitions is often smaller, so group membership is adjusted accordingly.

The second point relates to the gathering of results. Once worker processes in a group have completed their triangulations, the group sub-master collects the boundary set. Boundary triangles are needed by the sub-master in order to determine the next set of active minor partitions assigned to the group.

At the major partition level, TIPP's handling of interior triangles differs for the single and multi-master variants. In single-master TIPP, the worker nodes write interior triangles directly to the Ceph filesystem, not requiring sub-master involvement. In contrast, multi-master TIPP benefits from aggregating writes at the sub-master, reducing the load on Ceph. Once a group has finished its computation, the sub-master sends the boundary set to the master, which uses the boundary set to determine the next set of active major partitions.

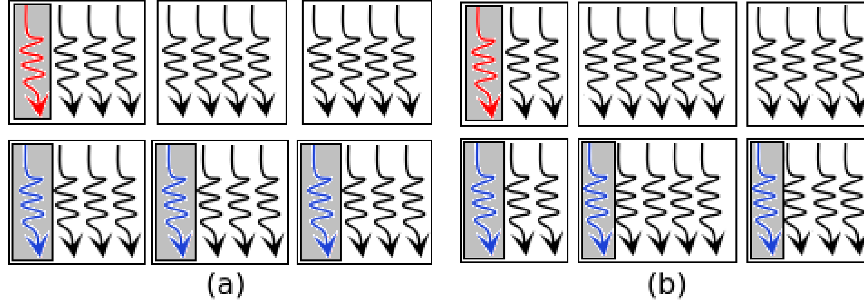


Figure 3.3. The running of multi-master TIPP for uniform (a) and non-uniform (b) point distribution. 12 processes are divided into three groups with a sub-master in each group (blue) and one master (red).

3.2.2 Parallelism for Non-uniform Point Distribution

In section 3.2.1, all processes are assigned into equally-sized groups of communicators because each group is responsible for a roughly similar workload. However, with non-uniform point distribution, the number of processes in each group should depend upon the number of points each group is responsible for. Figure 3.3(b) illustrates this case.

Let N_P be the number of major partitions, and $|P_i|$ be the number of points in some major partition P_i. Let $|world|$ be the total number of all MPI processes. Let $|group_i|$ be the number of processes assigned to a group (or *communicator*) working on a major partition P_i . $|group_i|$ should be:

$$|group_i| = \frac{|P_i|}{\left[\sum_{j=0}^{N_P-1} |P_j|\right]} \times |world| \tag{3.2}$$

The non-uniform point distribution problem can also be solved by applying the traditional producer-consumer model to single-master TIPP. In a given stage of Delaunay triangulation, the master process waits for worker requests, responding with new work items until no more work is available. Workers that finish partitions quickly are able to request more work from the master immediately, rather than wait for their busier peers to finish. The master node is responsible for keeping track of all independent partitions available for processing.

3.3 Experimental Results

We tested TIPP on a cluster running on the Chameleon Cloud Testbed [20]. Our storage node had two Intel® Xeon® E5-2650 v3 processors @ 2.30GHz, 64GB RAM and a 2TB HDD. Dedicated worker nodes had two Intel® Xeon® E5-2670 v3 @ 2.30GHz processors (16 cores total), 128G RAM and a 250GB HDD. All nodes ran 64-bit Linux.

We use 2D datasets that are generated from the *Qhull* utility [8, 9]. Point coordinates are generated on the range $[0, 1]$ and converted from text into binary representation for speed and storage efficiency. The point coordinates are separated into partitions in the domain and sorted by x coordinate. The size of the datasets ranges from 250 million points (roughly 500 million triangles) up to 2000 million points (around four billion triangles). Section 3.3.2 also describes performance for a 10 billion point triangulation. In all cases, there are no obstructions such as airplane wings or similar objects, and points are evenly distributed throughout the domain, except where otherwise noted.

3.3.1 Performance

The performance of TIPP depends on the dataset size (number of points in the domain), number of partitions, number of compute nodes, and the performance of each node. Figure 3.4 shows the execution times of TIPP with different numbers of partitions. The execution time of the the master node is shown as the darker region at the base of each bar. The middle bar in each graph shows the best performance, while finer and coarser partitionings perform less well overall.

Indeed, there is a trade-off between the granularity of the partitioning and the overall performance. Coarse–grained partitionings improve the speed of the master, because fewer partitions means fewer triangle–partition intersection tests are required. However, coarse partitionings hurt triangulation performance on the worker side because large partitions are not as effective in reducing the scope of point-circumcircle search. Coarse partitionings also exacerbate the effects of poor load balancing. Conversely, fine–grained partitionings burden

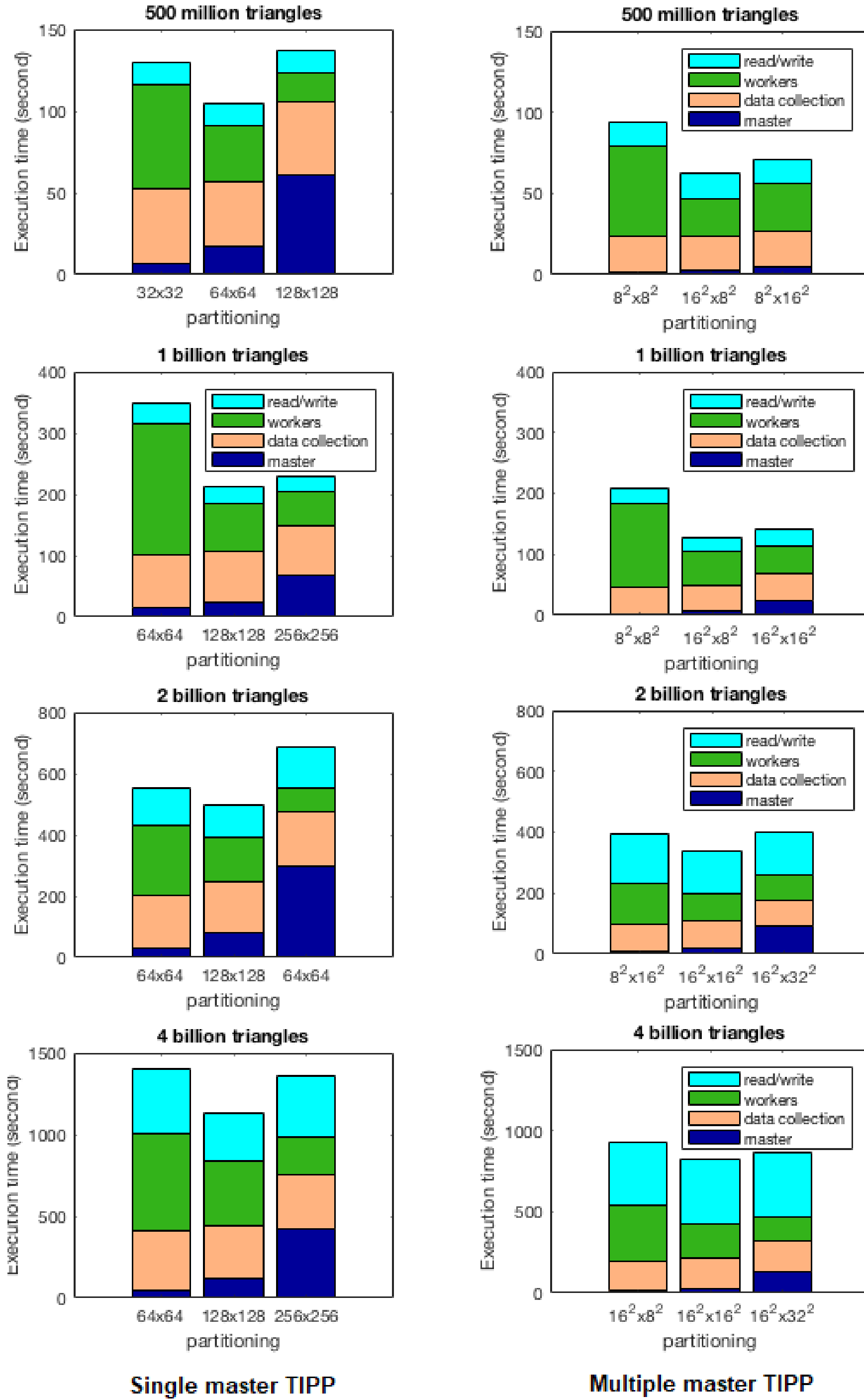


Figure 3.4. Execution times for 10 nodes executing 256 processes. The left column shows results for single-master TIPP while the second column shows multi-master results. For both implementations, four different datasets (250M, 500M, 1000M, and 2000M points) were used, each having a uniform point distribution. For each dataset, we show results for different partitioning granularity. For the multi-master case we write $n^2 \times m^2$, where n^2 denotes the major partitioning, and m^2 denotes the minor partitioning. Each bar represents the execution times of the master computation (lower), worker and sub-master nodes (upper), and master write (middle).

the master node while improving computational performance on the workers.

However, computation on the workers is not the only factor affecting performance. The results shown in figure 3.4 were generated using TIPP implementations in which the master node is solely responsible for reading and writing data files. Data must be distributed to the workers and results collected once complete, so they can be written to disk. The *read/write* and *data collection* costs depicted in figure 3.4 grow quickly as the problem size increases, hurting scalability. For this reason, we chose to employ *Ceph*[96], a well-known distributed filesystem suited for scientific computation. Results shown in the figures following figure 3.5 reflect performance of TIPP leveraging Ceph for file I/O.

Figure 3.5. Performance of multi-master TIPP relative to single-master TIPP, excluding I/O. (a) Speedup for both master and workers (256 total processes). (b) The percentage of total execution time due to the master for 1B triangles, for both multi and single-master TIPP.

From the execution times presented in figure 3.4, we also measure the speedup (see figure 3.5.a) of multi-master over single-master TIPP over 4 different dataset sizes, separating results for master and worker. Speedups were computed by comparing the best cases of multi-master TIPP with the best cases of the single-master version. The speedup ranges from 3.4 to 4.8 for master performance. Since we divide the domain into two levels of partitions in multi-master TIPP, the master is burdened by a much smaller number of partitions, reducing

master execution time.

The worker processes in multi-master TIPP also demonstrate superior performance. The reason is that the update procedure in the multi-master version is performed in parallel, while single-master TIPP must perform this task in serial fashion if the number of independent partitions is greater than the number of processes. The update procedure includes collecting boundary triangles and preparing active minor partitions for the next stage, so it has significant communication and computational components. On the right side of figure 3.5, we compare the percentage of total execution time spent on master tasks for both single and multi-master TIPP. This percentage grows much more rapidly in the single-master case, causing reduced performance not only because of resource contention on the master, but because workers sit idle waiting for work.

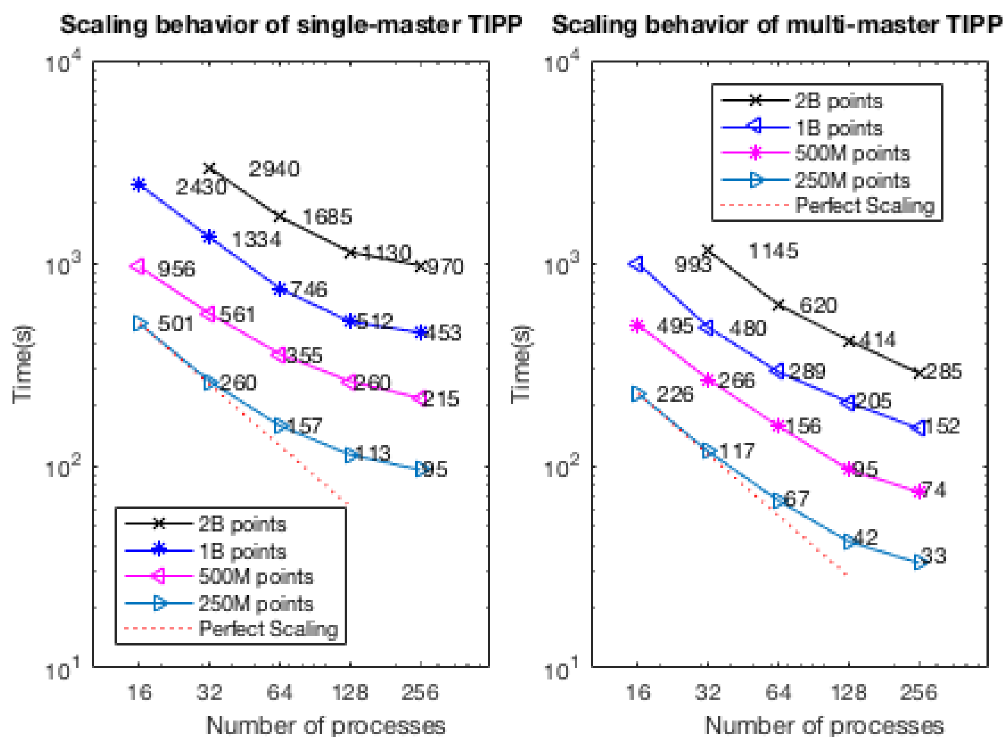


Figure 3.6. The scaling behaviors of multi-master and single-master TIPP for five datasets from 500 million to 4 billion triangles on 10 compute nodes. (a) The execution times of single-master TIPP (b) The execution times of multi-master TIPP. (The horizontal and vertical axis are logarithmic).

Figure 3.6 shows the effect of an increasing number of processes on overall execution

Table 3.1. Execution time (seconds) for master and worker processes of single and multi-master TIPP for 10 billion points (or 20 billion triangles) on 10 nodes, 256 processes.

TIPP versions	master	worker	total
single-master TIPP	718	2897	3615
multi-master TIPP	110	1461	1571

time for different dataset sizes in both single-master and multi-master TIPP versions. For each dataset size, we choose the partitioning with best performance from figure 3.4. The results indicate that multi-master TIPP is faster for a given problem size, and improves scalability compared to the single-master version, since the master node is no longer a bottleneck.

3.3.2 Processing a Large Dataset

With Qhull [8, 9] we can generate at most two billion points in the domain. However, in order to produce even larger test cases, we can replicate points in a sub-domain to adjacent sub-domains. All sub-domains are joined together into one large domain. Similarly, we also generate a large number of random points in a domain by generating random points in a partition, then replicating these points to all partitions in the domain. We check the resulting dataset to make sure there are no duplicate (or very close) points.

In our experiment, we triangulate a domain with 10 billion points and generate roughly 20 billion triangles on a Chameleon cluster with 10 nodes. The domain is divided with a $[256 \times 256]$ partitioning for the single-master TIPP version and $[32 \times 32] \times [16 \times 16]$ for the multi-master version. We ran the experiment using a cluster with 10 nodes and 256 processes.

Table 3.1 presents the execution times of single-master and multi-master TIPP versions. The speedup of the master process in the second version over the first version is roughly 6.5 times. A major reason for the improvement is the coarseness of the partitioning used by the master process. Single-master TIPP uses a 256×256 partitioning on the master node, compared to a 16×16 major partitioning for multi-master TIPP. Because the initial

mesh is built by selecting a similar number of points from each partition, the coarse partitioning used by multi-master TIPP results in a coarser initial mesh. In turn, the coarser mesh reduces the cost of identifying independent partitions, leaving a larger portion of work to be parallelized over the sub-master processes.

The total execution time is under half an hour for 20 billion triangles using only 10 nodes. We expect multi-master TIPP to make even larger meshes feasible even on commodity machines.

3.3.3 Load Balancing

In this section we conduct an experiment on non-uniform point distribution from a real-world dataset used for real-time storm surge prediction on the North Carolina coast [64] (see figure 3.7). The original mesh consists of 31,435 points, but we added points to the dataset while maintaining a distribution of points that is similar to the original. We produced four different sizes, resulting in 125, 250, 500 and 1000 million triangles. All tests were run with 10 nodes.

Figure 3.8 presents TIPP performance with uniform and non-uniform point distribution using 10 nodes.

For the uniform case, figure 3.8(a) shows that the multi-master implementation outperforms the single-master version by nearly 3× due to the distribution of master load onto several sub-masters. More generally, the trend as data size increases implies that multi-master TIPP scales better than the single-master version with regard to data size. We found similar behavior in figure 3.6.

For the non-uniform case shown in figure 3.8(b), multi-master TIPP shows much less improvement over the single-master version, due to the lack of effective load balancing. Figure 3.7 shows significant empty areas in North and South America in which there is no work to perform, while other areas have a much denser concentration of points. The figure also shows the relative size of the (major) partitions used by each version. Since

the major partitioning is uniform, some partitions contain huge numbers of points, while others have none. Since multi-master TIPP uses the same number of processes per major partition, load balancing is clearly poor. It is even worse than the single-master case for two reasons. First, the use of a coarse major partitioning increases the difference in load between partitions. Second, because the allocation of processes is the same for all major partitions, lightly loaded partitions finish very quickly, while heavily loaded partitions take much longer. Early finishers must wait until the other sub-masters have finished in order to

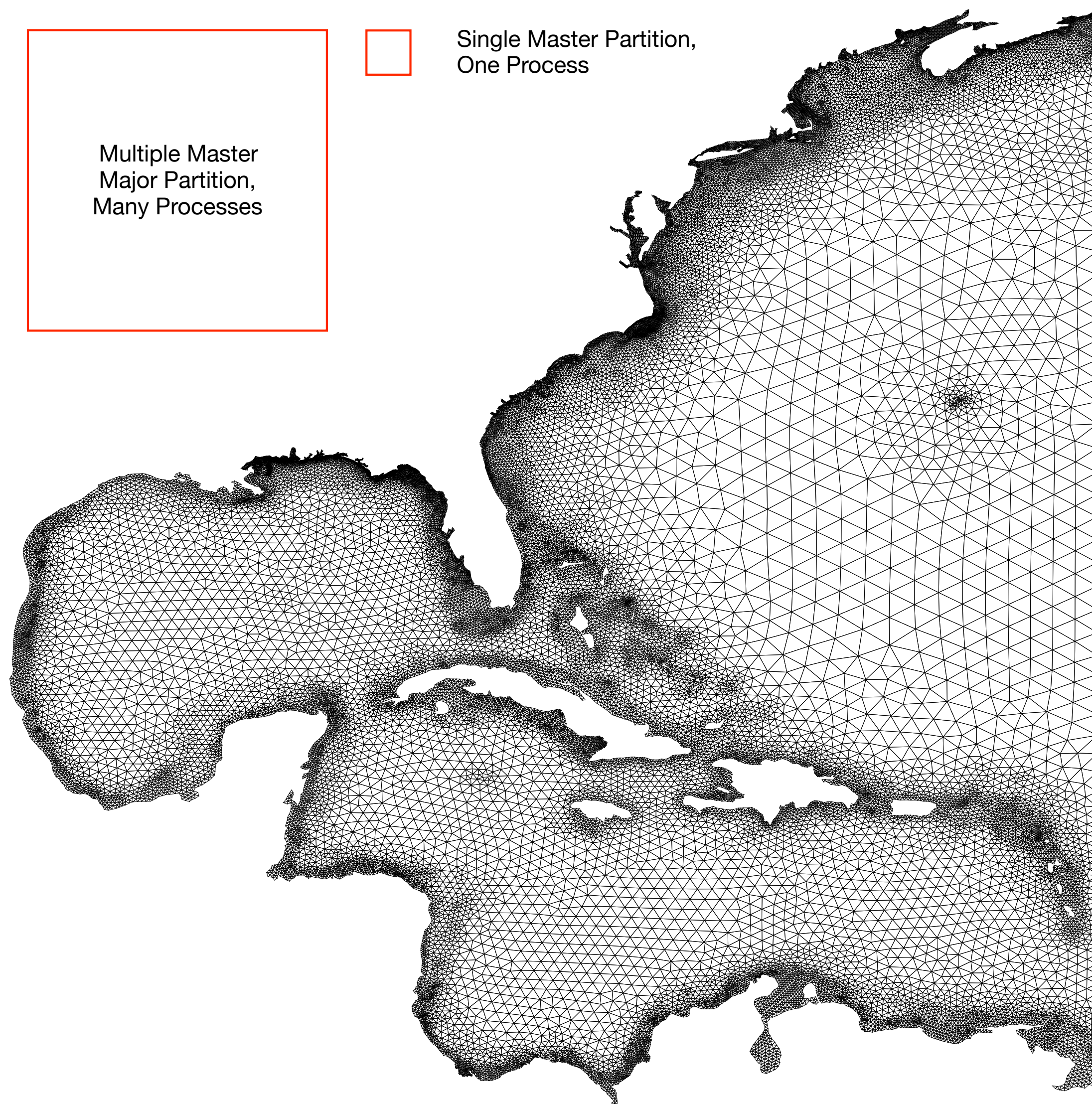


Figure 3.7. The ADCIRC Mesh. An example of a dataset with non-uniform point distribution.

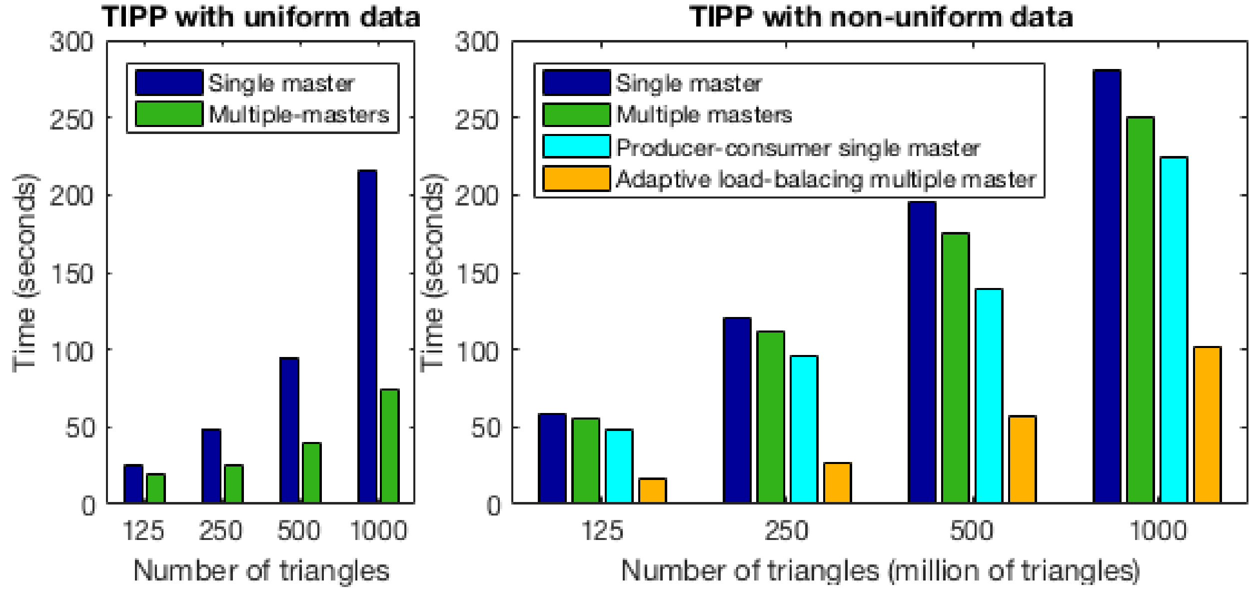


Figure 3.8. TIPP performance using 10 nodes, 256 processes (a) uniform and (b) non-uniform point distributions. (The horizontal axis is logarithmic).

contribute further to the triangulation.

To address this problem of load-imbalance, we implemented the adaptive version of the multi-master code, as described in section 3.2.2. Figure 3.8(b) shows a dramatic improvement in execution time. As shown in figure 3.1, this method varies the resolution of the minor partitionings within each major partition. Major partitions with a large number of points get subdivided into finer sub-partitionings, meaning they are assigned a larger number of processes. Conversely, lightly loaded partitions will be assigned only a small number of processes. The overall effect is to apply computational resources proportional to load in different areas of the domain. Although this method is adaptive, it is still a static load balancing strategy, since the load is divided up before starting the computation itself.

We also evaluated the effectiveness of dynamic load balancing in the form of a simple producer-consumer implementation with a single master, as described in section 3.2.2. The motivation is to address the problem of "stragglers" when processing partitions in shifts. As described in section 2.3.2, when the number of independent partitions is greater than the number of processes, we must schedule partitions in several shifts on the available processes.

If one partition takes longer than the rest of the shift, the plain single-master implementation waits until that straggler is finished before giving other processes in the shift more work. The producer-consumer implementation allows individual processes in the shift to request more work without waiting for stragglers to finish. Results in figure 3.8(b) show modest improvement over the plain single-master case. The adaptive static approach proved to be much more effective.

3.3.4 The Tess System

Like our own work, the *Tess* system performs distributed Delaunay tessellations, and can handle extremely large input sets [72, 60, 59]. Tess is targeted to 3D tetrahedralizations running on supercomputer hardware such as IBM's *BlueGene/Q* and the *Cray XC30*. Tess can decompose the input set using either a regular partitioning for balanced point distributions or a kd-tree for unbalanced datasets. In either case, the points belonging to each block are tessellated using an existing tool, either *qhull* [8] or *CGAL* [19]. After each block has performed an initial tessellation, points that may affect neighboring blocks are exchanged. Local tessellations are then updated with the newly received points. This process is repeated until no blocks exchange any more points.

The result of the initial tessellation is a set of disjoint tessellations, rather than a single global tessellation that spans the domain. Subsequent rounds will refine the tessellation until the global mesh satisfies the Delaunay property. During this process, Tess keeps mesh data in memory on the compute nodes, assuming that the aggregate memory available in a supercomputing environment is sufficient to hold the dataset. In contrast, TIPP can process partitions in shifts, so it does not the require aggregate capacity necessary to hold the complete dataset.

Tess and TIPP differ in their approach to communication. TIPP sends a few large messages between (sub)master and worker. Tess workers communicate directly, sending relatively small messages when exchanging points with neighboring blocks, avoiding the

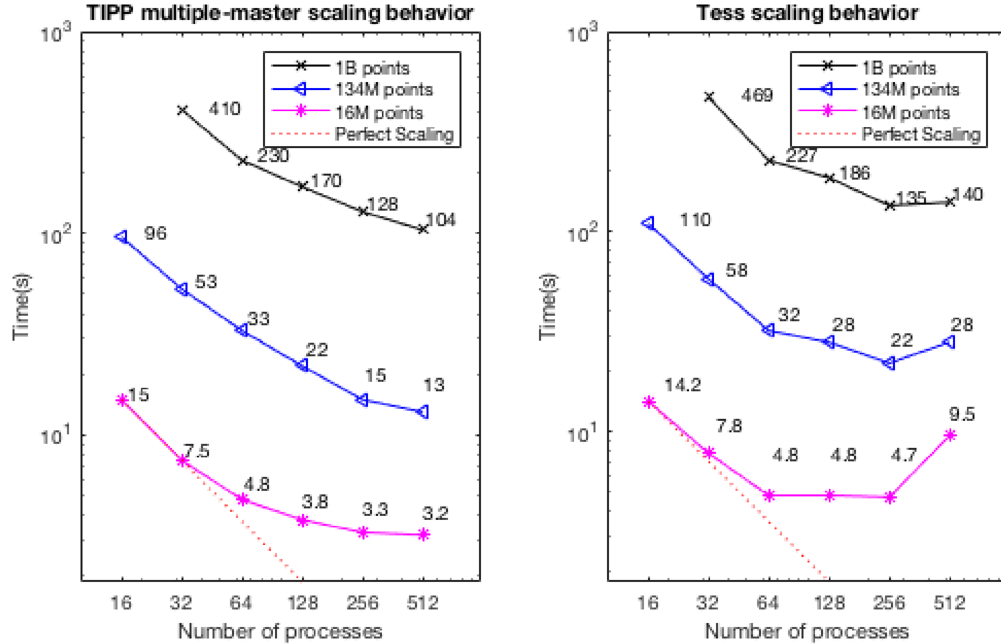


Figure 3.9. The scaling behavior of total execution time, including tessellation and data I/O costs for (a) multi-master TIPP and (b) Tess[60] on 32 compute nodes. (The horizontal and vertical axes are logarithmic).

contention seen in single-master TIPP. Supercomputing environments such as BlueGene have very high performance interconnects, so latency costs due a large number of communications are minimized. However, clusters of commodity machines connected via less exotic networks are far more common in institutional and cloud computing. In these environments, the Tess design may suffer from network contention and increased latency. On BlueGene/Q, the Tess authors report a strong scaling efficiency of 32% for the tessellation phase (no I/O), using a regular decomposition of 1 billion points. We repeated this test on a 32 node cluster connected via 1Gbps Ethernet, provided by the Chameleon Testbed [20]. We found strong scalability of only 21%, reflecting the higher overhead costs of the cluster environment. On the same platform, we observed strong scaling efficiency for TIPP's tessellation phase to be 45%, but since Tess works in 3D, the numbers are not directly comparable. A 3D implementation of TIPP is an avenue for future work.

Figure 3.9 shows the results of a scalability study for both TIPP and Tess, running

on a cluster hosted on the Chameleon testbed consisting of 32 compute nodes and 9 Ceph storage nodes. TIPP results rely on Ceph, while Tess handles I/O on its own. However, the cause of the poor scalability in figure 3.9b is likely not I/O, but communication overhead. On commodity hardware, network latency is simply too high to support many small messages. TIPP scalability fairs better with high process count for two reasons: the 2D case inherently requires less communication than 3D, and the TIPP design aggregates the communication that is required.

3.3.5 Chapter Summary

In this paper, we improved the original single-master TIPP algorithm (present in previous chapter) by introducing parallelism at two levels, distributing the burden of the master node across several sub-masters. Multi-master TIPP shows significant gains in performance over the previous version.

We also implemented both static and dynamic load-balancing strategies to address non-uniform point distribution. Although a producer-consumer implementation shows improvement over the original TIPP, a static adaptive strategy demonstrated the most significant gains over all.

One avenue for future work is to augment the existing implementation to produce additional information convenient for topological navigation of the computed mesh. Although our geometric approach has proven convenient for generating the mesh in parallel, as well as geometric queries, support for topological queries would be a useful addition.

Along these lines, we envision a distributed query engine that would take advantage of the triangulated mesh. Such a system might not gather triangulation results back to the master nodes, but instead leave this data on the workers, making it available to answer queries on unstructured datasets at unprecedented scale.

CHAPTER 4

ACCELERATING RANGE QUERIES FOR LARGE-SCALE UNSTRUCTURED MESHES

Unstructured meshes are essential to certain fields of engineering and science, but they present special challenges for efficient access and processing. The work described in this chapter accelerates range queries for very large unstructured meshes using the GPU. Prior work in the area introduced a preprocessing phase that *partitions* large unstructured meshes in order to improve locality in storage and memory.

Here, we apply the computational power and bandwidth of GPUs to the partitioning problem, significantly reducing preprocessing time. In order to keep the GPU busy, we have to overcome the poor locality of the original unstructured mesh. Toward this end, we developed our own approach to unstructured mesh I/O, called *Direct Load*. We show that Direct Load significantly outperforms a typical LRU cache. Our ultimate goal is to accelerate range queries. Our preprocessing steps allow us to parallelize range query processing with relatively simple GPU code.

4.1 Background

Spatial scientific data is a general term that often is classified into two types of data: *structured* and *unstructured meshes*. Structured meshes follow a regular pattern throughout the dataset domain, while unstructured meshes contain points that are arbitrarily distributed in space. Because of their irregular nature, unstructured meshes are very flexible, but also much more difficult to process efficiently.

4.1.1 Data Structures

Unstructured meshes are a collection of non-overlapping cells, usually triangles (in 2D) or tetrahedra (in 3D). Such cells are called simplices or simplicial cells, meaning that they use the fewest number of vertices to form a closed region in space.

The data structure of unstructured meshes can be organized into several files. The first file, known as a *cell file* consists of tuples of indices into the vertex file. One triangular cell require three vertexIDs while a tetrahedron needs four. The *vertex file* contains all data associated with vertices, including tuples of coordinates. There can be many attributes of a vertex, for example, in a weather application the attributes might be temperature, humidity, pressure, etc. In figure 4.1, cell 200 has three vertexIDs (1, 300, 2999) pointing to three coordinates (0.75, 0.53), (0.25. 0.95), and (0.45, 0.78) respectively.

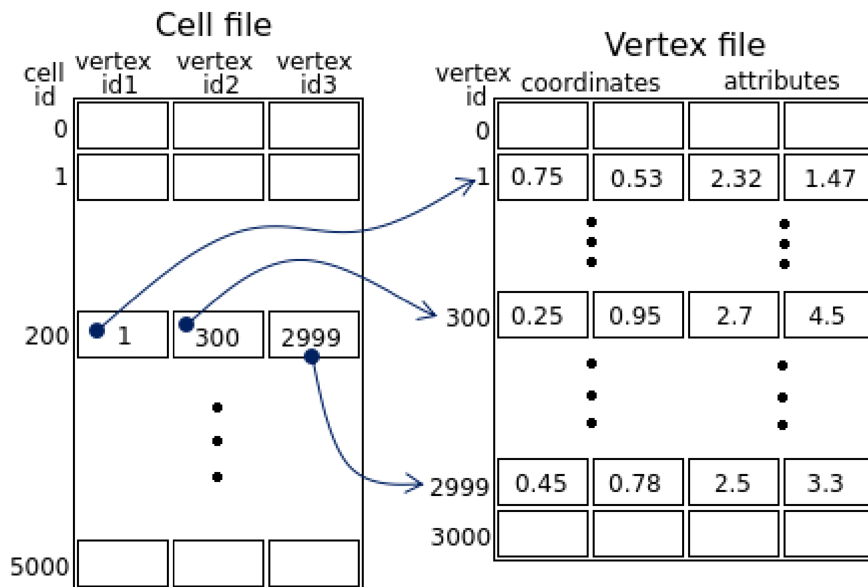


Figure 4.1. Data structure for unstructured meshes. Cell 200 has three indexes 1, 300, and 2999 in vertex file

4.1.2 Owner and borrower cells

Partitioning datasets into *partition elements* is an effective general-purpose technique for improving access to unstructured datasets. However, one of the problems of partitioning unstructured grids is cells that span partition element boundaries, causing duplication of

cell data [79, 1, 3, 2]. To solve this duplication problem, Akande et al. [1] used the *owner-borrower scheme* to store attributes of a cell uniquely. In this scheme, a cell can intersect with one or more partition elements in the domain space, but only one partition is chosen as the owner, while other intersecting partitions are the borrowers. Figure 4.2 presents an example of 8 partition elements with 4x4 partitioning resolution. The empty and shaded triangles are owner and borrower cells respectively.

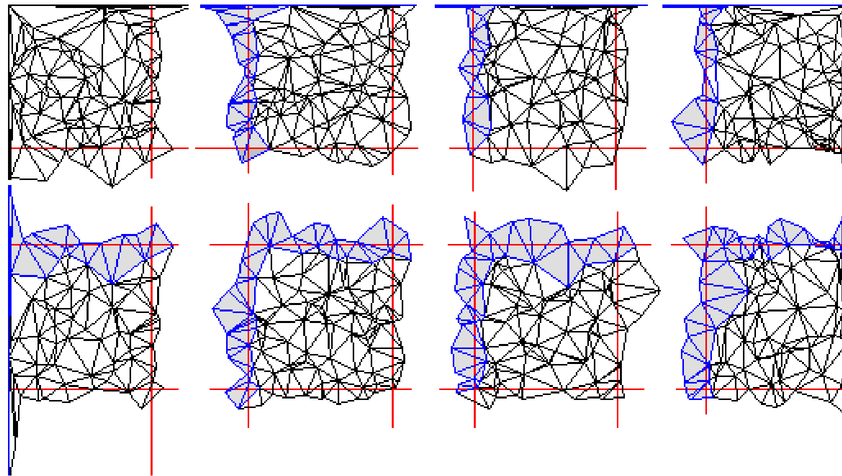


Figure 4.2. 2D partition element 0 to 7 of 4x4 partitioning for 2000 cells. Black and blue triangles are owner and borrower cells. There is no borrower cell at partition element 0.

4.1.3 The partitioning process

Partitioning spatial datasets is a method to split the domain space into many smaller partition elements. Figure 4.2 shows 8 partition elements out of a total of 16. Based on the partition element positions in domain space, cells that are nearby and located inside a partition element will be grouped together and stored to a file on disk. Partitioning methods have been applied widely for decades [83, 82, 84, 102, 80] as effective techniques for solving big data problems, especially for multidimensional data. After reorganization, the partitioned datasets will have significantly improved locality, and therefore gain overall performance. Moreover, partitioning is a solution for breaking down the large datasets and distributing them over a set of cloud or cluster machines.

4.1.4 Caching

Caching is a well-studied technique for accelerating access to data. Caches maintain a collection of cache blocks that each hold a copy of some subset of a larger dataset. Because the cache is quicker to access than the underlying dataset, query performance is enhanced whenever the cache contains the requested data.

Caches work by exploiting two different types of locality in an access pattern. *Spatial locality* refers to multiple accesses to data items that are nearby in the dataset. An access pattern with good spatial locality benefits from a cache because the same cache block can satisfy several nearby accesses. *Temporal locality* refers to multiple accesses to the same data item over time. As long as the block containing that data item remains in the cache, an access pattern that requests the same item repeatedly will benefit from the cache's quick response time.

An access pattern with poor temporal locality will cause the cache to discard a block before it is reused. Correspondingly, an access pattern with poor spatial locality will fill the cache with blocks that are widely separated in space, and are not reused before being discarded. In either case, the result is cache *pollution* [103], meaning that the cache is filled with information that will not actually be used.

The degree of cache pollution is partially dependent on factors such as the size of the cache blocks, and also on the *replacement policy*, which governs which block is evicted when more space is needed in the cache. A very common example is the *Least Recently Used (LRU)* policy [88], which discards the cache block that was accessed least recently. In any case, changing the replacement policy or cache block size cannot help in cases where the access pattern has little or no locality. In such cases, pollution is inevitable, and caching may even hurt, rather than help, performance.

Cache pollution results from the assumption that future accesses can be predicted from a current access to the cache. The cache loads data "on demand", in response to a query, in the hope that the extra data read into a cache block will be used in the near future.

For access patterns with poor locality, this gamble does not pay off, and the cache only increases the volume of data read from file.

One of the goals of this paper is to demonstrate an alternative to straightforward caching that takes better advantage of available information to handle the challenge of reading from the vertex file of an unstructured mesh. Since this file has extremely poor spatial locality, we should use the information in the cell file to help manage how vertex data is accessed. In section 4.3.1, we describe Direct Load, our proposed solution to the I/O problems of unstructured meshes. Direct Load is not a caching method, and therefore has no need for a replacement policy. Most importantly, it makes few assumptions about the access pattern used when reading data from the vertex file.

4.2 Motivations

Our goal is to accelerate both the partitioning process and the response to range queries made by end users. Because we focus on large datasets, both tasks present performance challenges, which we further explain in this section.

4.2.1 Partitioning Performance Breakdown

The partitioning process consists of three phases: *load*, *process*, and *update*. The load phase reads cells and corresponding coordinate data from files into memory. The process phase distributes cells and their corresponding data to the appropriate partition elements. The update phase simply stores cells, coordinates and attributes into partition files. Figure 4.3 shows the proportions of these three phases to the overall execution time of the partitioning process in which we use the best results of LRU for the loading phase. The particulars of the testing environment are described in section 4.5.

Figure 4.3 presents the execution times of phases in which the load phase proportion increases as the dataset size becomes larger. The load phase is therefore an obvious candidate for improvement. Load often has a relatively high cost due to the scattered locations of cell vertices in the vertex file. Since cell vertices are often far apart in the file, reading cell

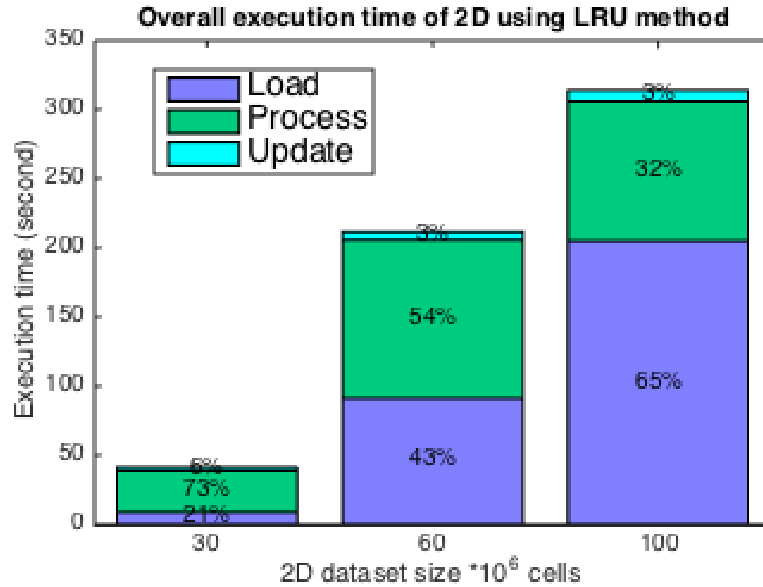


Figure 4.3. The proportion of load, process, *and* update *execution time with different 2D dataset sizes*

vertices requires several separate read operations, not just one. To address this problem, we first tried using an LRU cache to store and reuse data loaded from the vertex file. We also developed a new approach, Direct Load, which shows substantial improvement over the LRU implementation. Section 4.3 describes this method in detail.

The other major partitioning cost is the *Process* phase, in which cells are associated with the partitions with which they intersect. This requires triangle-rectangle (2D) or tetrahedron-cuboid (3D) intersection tests, which are expensive. However, we can use a cheap rectangle-rectangle or cuboid-cuboid test with the cell's bounding box to vastly reduce the number of expensive intersection tests. Nonetheless, the sheer volume of cells is a natural candidate for the bandwidth and computational power of the GPU.

4.2.2 Range Queries

Scientists often query attribute data in an arbitrary rectangular area or cuboid volume (i.e. *range*) within the domain space. Instead of only collecting attribute data from cell vertices, it is often useful to *resample* the data at regular intervals within the specified range (see figure 4.7). For each point in the resampling, we must first determine which cell

contains the point using an intersection test, and then interpolate new data values from the relevant cell. Both the intersection test and the interpolation contribute significantly to the execution time of the range query. The performance results in table 4.1 show that range query execution times are sensitive to the number of query points, indicating that a highly parallel GPU implementation of range queries will be fruitful.

Table 4.1. Range query time of domain for 2D and 3D datasets

Number of query points in a partition	25	36	64	144
Range query on CPU, 2D (seconds)	42.84	57.88	94.05	200.89

Number of query points in a partition	1	8	27	64
Range query on CPU, 3D (seconds)	39.1	128.02	396.4	879.4

For large range queries that are too large to be processed in-core, we must process the data in a piecewise fashion. Fortunately, the original dataset has been restructured into convenient partition elements that significantly simplify I/O. Range query performance is further improved by the parallelism of the GPU, but a major motivation of our work is to allow the query code to be as simple as possible. After all, the partitioning process is performed once, but users will present queries repeatedly. Other geometric queries for scientific mesh data, such as those listed by Lee, et al. [49], should also benefit from the data reorganization, though we focus only on range queries in this paper.

4.3 Preprocessing datasets

The unstructured datasets used by scientists can be very large, often with hundreds of millions of cells. Since we cannot read the entire dataset into memory for partitioning, we instead load subsets of the cell file into memory in a piecewise fashion. We call this subset a *chunk*. For each cell vertex index in a chunk, we must also read its coordinates from the vertex file. Each vertex index in the chunk points to a coordinate record in the vertex file (see figure 4.1 or 4.4.a). The cells in each chunk can then be further processed for classifying

into the partitions to which they belong.

4.3.1 Direct Load

Loading vertex data presents a performance challenge because of poor locality. The vertexIDs of a cell may refer to coordinates that are far apart in the vertex file, which will likely hurt performance by incurring separate read operations for each cell vertex. In this case, cache methods (such as LRU) do not work effectively. Large numbers of blocks that would have been reused are discarded, while non-reused blocks are retained.

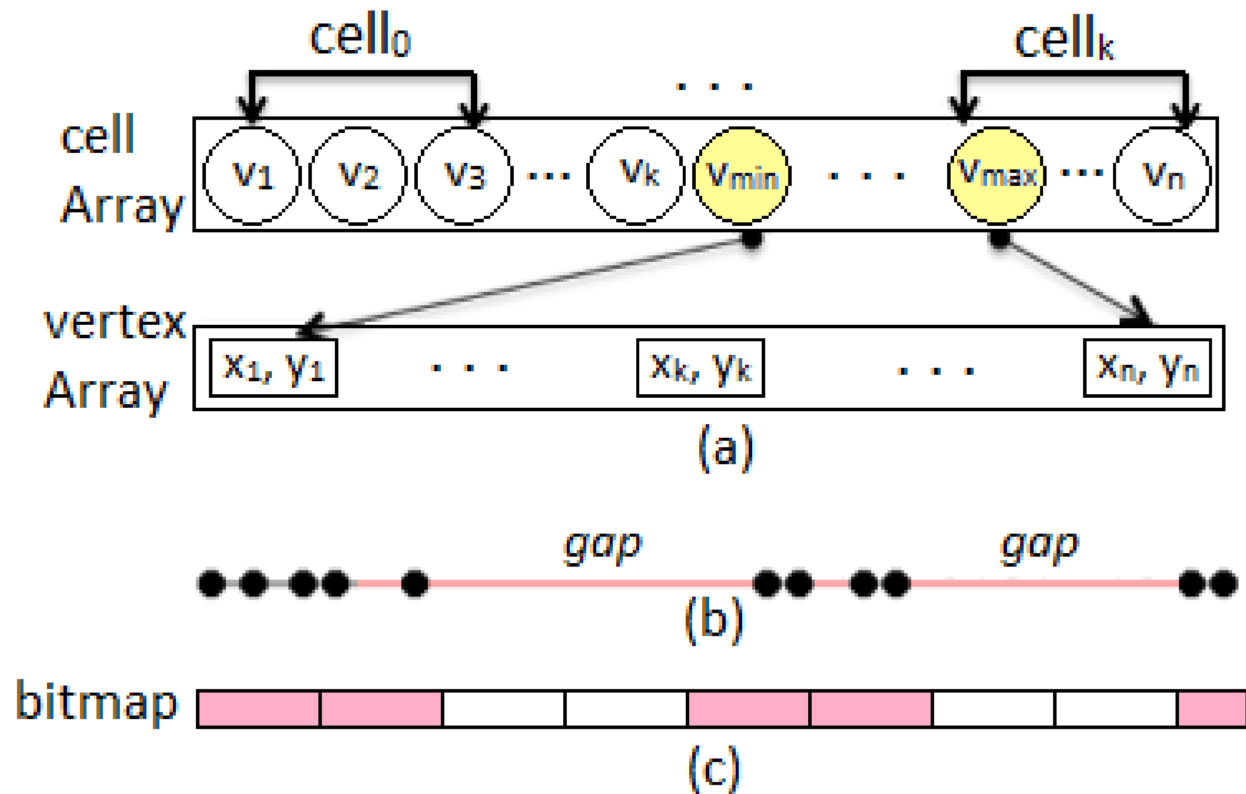


Figure 4.4. Direct Load method. Each item in cell array is vertexID, three vertexIDs form a cell sequentially. The vertex array consists of coordinate and attribute data for vertexIDs. (a) Read vertex array from v_{min} and v_{max} indices. (b) Gaps are unneeded segments where their distances are larger than vertex memory threshold M. (c) bitmap contains active (shaded rectangles) and inactive segments.

In this section, we developed a new approach to the locality problem called *Direct Load (DL)* which reduces the number of read operations made to disk. DL is not a caching algorithm, and does not require a replacement policy. Instead, we see DL as a prefetching

algorithm that avoids reading unnecessary vertex data using prior knowledge extracted from the cell file.

The main idea is to read as much data as possible from vertex files and reduce the number of read operations. Because not all of these coordinates are required by the cells, we copy out only those that are needed. The final result is a *flattened* version of the original cell file subset, in which each vertex index has been replaced with the corresponding coordinate values (see figure 4.5.b). Figure 4.4.a presents DL in the simplest case, where enough memory is available to read the vertex data for an entire chunk of cells. However, if memory is scarce, or the gaps between vertexIDs are too large to fit into the available memory, we must perform more than one read, rather than one large one.

In order to improve the performance in this case, we can increase the amount of useful coordinate data contained in each read by skipping over large regions (at least M vertices) that contain no required vertices.

Figure 4.4.b shows the case where a set of vertices are separated by significant gaps of unneeded data. To skip unneeded segments in the vertex file, we design a bitmap array (*bitmapArr*) to reduce the number of read operations. Each bit of the bitmap refers to a corresponding *segment* of contiguous vertices in the vertex file. The bitmap labels each segment as either *active* or *inactive*. A segment is called active if it contains coordinate data needed by a cell while inactive segments will not have any references pointing from *vertexIDs* in *cellArr* (i.e. a chunk). Inactive segments are skipped to save reading time. Figure 4.4.c shows several skipped regions of inactive segments.

The Delaunay triangulation algorithm reads segments of coordinate data from the vertex file and copies needed data to the result array (*coorArr*). Because only needed segments are read and the number of read operations is far less, DL performance is much faster than LRU. Algorithm 4 presents in detail how DL works. First, we load a subset $[vertexID_{min} \ldots vertexID_{max}]$ of vertices from the vertex file into temporary buffers (*buff*) using one or more reads (see figure 4.4). Then, we copy needed coordinate data from the

Algorithm 4: DIRECT LOAD read a subset of coordinates from vertex file based on vertexIDs in a chunk (*cellArr*)

Input: *cellArr* is the array of *vertexIDs*, M is the number of vertices threshold or memory threshold, verFile is vertex file
Output: *coorArr* (coordinates Array)

```
1  verId_min ← min(cellArr);
2  verId_max ← max(cellArr);
3  chunkRange = verId_min ... verId_max;
4  if ||chunkRange|| ≤ M then
5      buf ← readFile(verFile, chunkRange);
6      for i ← 0 ... ||cellArr|| − 1 do
7          coorArr[i] ← buf[cellArr[i] − verId_min];
8      end
9  else
10     bitmapSize ← ⌈||chunkRange||/M⌉;
11     Generate bitmapArr[bitmapSize] where each item is corresponding to a segment
         with length M
12     segmentIDs ← 0 ... bitmapSize − 1;
13     for each segID ∈ segmentIDs do
14         if bitmapArr[segID] is active then
15             firstVerId ← segID × M + verID_min;
16             lastVerId ← (segID + 1) × M + verID_min;
17             verRange ← {firstVerId ... lastVerId};
18             buf ← readFile(verFile, verRange);
19             for i ← 0 ... ||cellArr|| − 1 do
20                 verId ← cellArr[i];
21                 if verID ∈ verRange then
22                     coorArr[i] ← buf[verId − firstVerId];
23                 end
24             end
25         end
26     end
27 end
```

buffers to the coordinate array (*coorArr*). If the subset to be read from the vertex file is too large to fit into the available memory, we divide it into a number of segments of size M and read each segment one at a time. The advantage of this method is that we do not read a segment more than one time, so segment memory can be reused. Moreover, the segment size can be as large as the vertex memory threshold M. Therefore, the number of read operations

is reduced significantly.

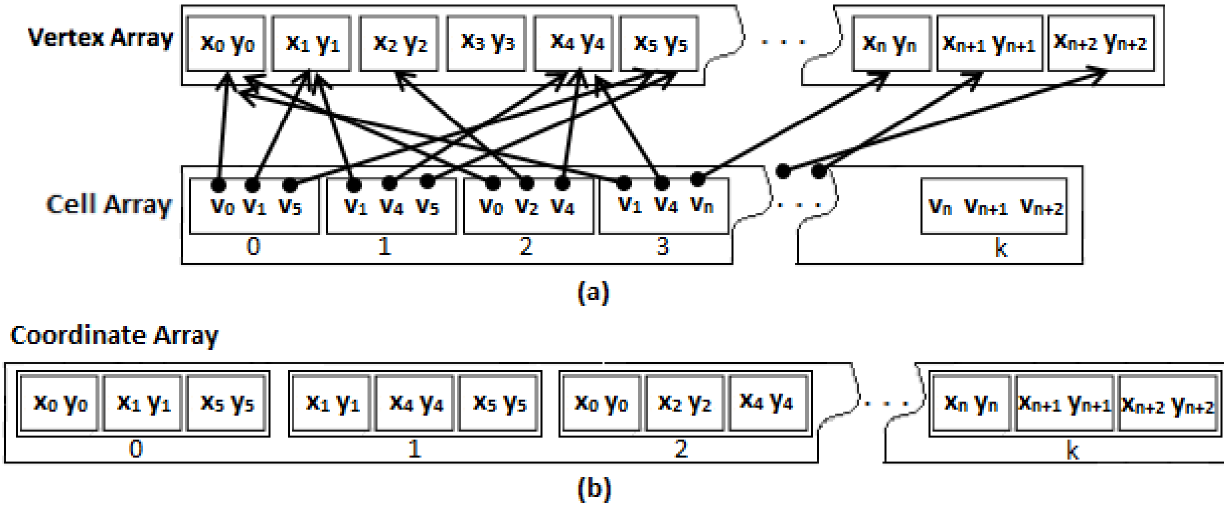


Figure 4.5. Data organization before and after loading data using DL or LRU of 2D datasets. (a) The original data organization of cell and vertex arrays. The values v_i in rectangles of cell array are vertexIDs, and the indices outside rectangles are cell indices. (b) Coordinate array after loading data using either DL or the LRU cache. The values inside small rectangles are coordinates, and the values below the rectangles are cell indices. The coordinate array now contains all coordinates needed for each cell in a very localized manner.

The key difference between LRU and DL is the number of read operations from the vertex file. In case the vertex data is too large to fit in the available memory, the DL method only reads $\|chunkRange\|/M$ times to load coordinate data for a chunk *cellArr*. Since M is relatively large, the number of reads is quite small. On the other hand, in the LRU method, the number of reads depends on cache block size, the number of blocks, and the number of gaps in the vertex file. The number of reads increases if cache size is limited. We will give details of our LRU implementation and the performance of both methods in section 4.5.A.

4.3.2 Partitioning operations on the GPU

The *Owner-borrower* representation of unstructured meshes divides an unstructured mesh into a set of partitions that vastly improve locality [79, 2, 1, 3]. Adapting this partitioning scheme to run efficiently on a GPU presents challenges, largely due to the GPU's SIMD (single instruction multiple data) architecture. We do not know ahead of time how

many cells will be given to each partition, and the GPU cannot dynamically allocate memory, so predetermining adequate memory space is a particular concern. Along similar lines, we must accommodate a potentially different number of cells for each partition, but cannot use the dynamic data structures available on an ordinary CPU. In this section, we present a GPU version of the original owner-borrower algorithm.

Algorithm 5 loads successive *chunks* of cells from the cell file, processing the file in a piecewise fashion. If a chunk is larger than GPU global memory, that chunk will be split up into sub-chunks and each one processed on the GPU sequentially. For the cells in each chunk, we must determine the partitions they intersect with, and also which of those partitions is the cell's owner, and which are borrowers. Each cell has exactly one owner, but may have several borrowers.

Having loaded a chunk of cells, we next use the GPU to compute the bounding box for each cell, and then map that bounding box to a set of intersecting partitions. Because the bounding box is axis-aligned, this is an arithmetically simple operation. It is also embarrassingly parallel and well suited to the GPU. For each cell, we count the number of partitions the cell's bounding box intersects with. After all cells in the chunk have been processed, we sum the counts to produce N, the total number of cell-partition intersections found within the chunk. Along with K, the number of cells in the chunk, we now have enough information to allocate space for two GPU arrays: the *owner* array requires K elements, since each cell in the chunk has exactly one owner, while the *borrower* array requires no more than $N - K$ elements. We know this because K of the N intersections must correspond to owner partitions, since each cell has one owner. Therefore $N - K$ intersections will either correspond to borrower partitions, or perhaps to partitions that intersect with a cell's bounding box, but not with the cell itself (i.e. false intersections).

The next task is to eliminate false intersections by doing a true cell-partition intersection test for each cell in the chunk, and also determine the owners and borrowers for each cell. This test is much more expensive than the bounding box intersection and bene-

Algorithm 5: Partitioning unstructured meshed into partition elements

Input: *cellFile, coorFile (coordinate), set of partitions P*

Output: *Owned and borrowed cells for partitions in P*

```
1  for each chunk C_i ⊂ cellFile do
2      On CPU:
3      coorArr ← loadDirect(C_i, coorFile);
4      On GPU:
5      for each cell c_j ∈ C_i do
6          n_j ← count(map(P, boundingBox(c_j)));
7      end
8      On CPU:
9      N ← ∑ n_j;
10     K ← count(C_i); //number of cells in C_i
11     Allocate ownArr[K], borrArr[N-K], borrId[K]
12     borrId[0] ← 0;
13     for each cell c_n ∈ C_i, except c_0 do
14         borrId[c_n] ← borrId[c_n − 1] + n_j − 1;
15     end
16     On GPU:
17     for each cell c_k ∈ C_i do
18         P_b ← map(P, boundingBox(c_k));
19         for each partition p_l ∈ P_b do
20             if c_k.containedBy(p_l) then
21                 ownArr[c_k] ← p_l;
22             else
23                 if c_k.intersects(p_l) then
24                     if ownArr[c_k] == ∅ then
25                         ownArr[c_k] ← p_l;
26                     else
27                         borrArr[borrId[c_k]] ← p_l;
28                         borrId[c_k] ← borrId[c_k] + 1;
29                     end
30                 end
31             end
32         end
33     end
34     On CPU:
35     for each partition p_m ∈ P do
36         Extract cells, coordinates, and attributes to data_m from ownArr, borrArr,
            coorArr
37         append(data_m, partitionFile_m);
38     end
39 end
```

fits significantly from parallel execution. Because each cell may have an arbitrary number of borrowers, we use a *borrowerIndex* array that records the index in the *borrower* array where the list of borrowers for a given cell begin (see figure 4.6). Because we know how many bounding box intersections involved each cell, we can initialize the *borrowerIndex* array to provide enough space for each cell's intersecting partitions (see line 13 in algorithm 5).

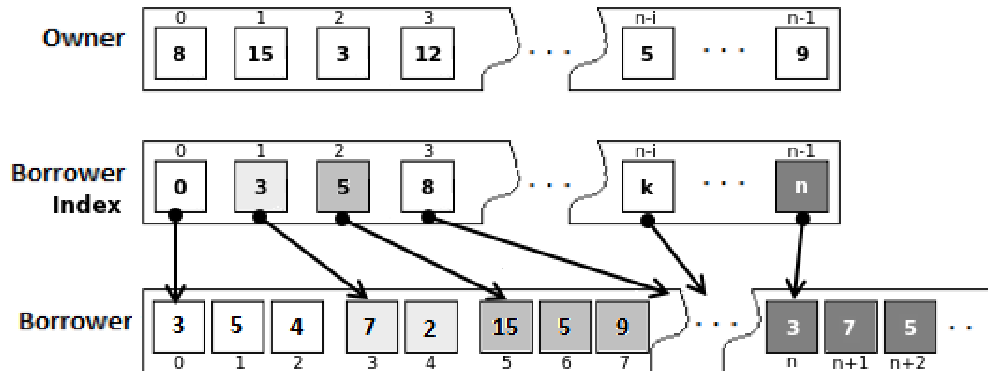


Figure 4.6. Owner, Borrower, Borrower Index arrays in the GPU memory. The numbers inside the squares of the Owner and Borrower arrays are partition IDs. The numbers inside squares of the Borrower Index array are indices into the Borrower array, keeping track of Borrower data for each cell.

The GPU next iterates through the set of partitions that intersect each cell bounding box, and determines each such partition to be either the owner, a borrower, or a false intersection. The owner is always the first truly intersecting cell encountered during the iteration, and is recorded in the *owner* array. Other intersecting cells are recorded in the *borrower* array via the *borrowerIndex* described above.

It now remains to write the owner and borrower information for each partition out to file. However, the GPU arrays are organized by cell, rather than partition. Beginning at line 35, algorithm 5 reorganizes the owner and borrower arrays, and then writes data corresponding to each partition out to the corresponding files.

The GPU portions of algorithm 5 run with one thread per cell, and no communication or synchronization between threads is required. For best performance, threads should each process a similar number of cell-partition intersections because of the lockstep execution

model of SIMD. Otherwise, all threads in a group will have to wait for the busiest thread to complete. For our algorithm, if the partition size is reasonably large compared to cell size, most cells fall within the interior of the partitions, so corresponding threads have a similar workload (see line 20 of algorithm 5).

4.4 Range Queries

A user can specify a range query by giving an axis-aligned bounding box (AABB) and delta values for each dimension. We express the bounding box as two points in the domain, corresponding to lower and upper corners of the box. Delta values are used to step through the volume described by the bounding box, generating a *grid* of sample points along the way. In effect, these delta values allow the user to specify the resolution of the range query.

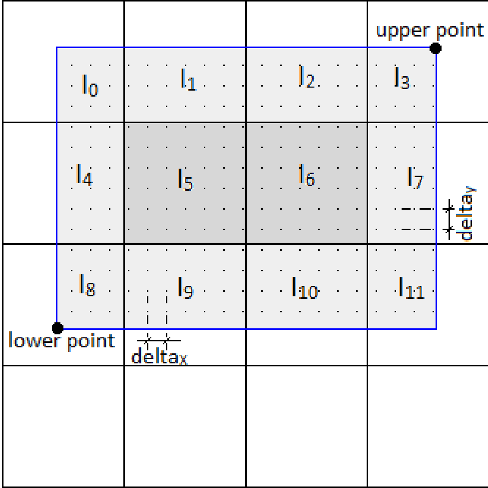


Figure 4.7. 2D range query. The shaded rectangle is the bounding box. The intersections are from I_0 to I_{11}

Figure 4.7 shows the relationship between the dataset partitioning and the range query itself. We can easily map the query bounds to the partitioning, finding the partition elements that intersect the bounds. In some cases, the partition elements are entirely contained in the query (I_5 and I_6 in figure 4.7), while in others the query bounds intersect only a portion of the partition. In either case, we compute the set of sample grid points corresponding to each partition, load the relevant data, and launch a GPU kernel to perform the bulk of the

computation. The data relevant to each partition includes not only the attribute data for cells *owned* by a partition, but also cells *borrowed* by a partition.

Finding the cell that contains a sample point is an essential part of satisfying a range query. To find this cell, we must examine each cell in a partition using a moderately expensive geometric test [48, 4]. Once the containing cell has been determined, we must compute interpolated attribute values from the data associated with the cell vertices. Because this process must be performed for every sample point in the grid, range queries are particularly expensive. Table 4.1 shows the range query running times of 2D and 3D datasets. The range query times increase significantly when the number of sample points increases.

We expect that the number of cells will typically be greater than the number of grid points, so we parallelize over the set of cells instead of the set of points. Each thread iterates through the given grid of sample points, and tests whether each point is contained by the cell it is responsible for.

4.5 Experimental Results

We ran our experiments on a 64 bit Linux server with 8 AMD cores running at 4Ghz and 32G RAM. Our AMD GPU has 3GB of global memory and 2048 lightweight cores running at 925 MHz.

We use 2D and 3D datasets that are generated from the *qhull* utility [8]. For both 2D and 3D datasets, point coordinates are generated on the range $[0, 1]$, and then grouped into cells using qhull's Delaunay triangulation (or tetrahedralization for 3D). The results include a cell file, containing the vertexIDs for each cell, and a vertex file containing the coordinates and attributes for each vertex. The vertexIDs in the cell file are indices into the vertex file. We convert both files from text to a binary representation for speed and storage efficiency.

The 2D dataset contains over 30 million triangular cells while the 3D data consists of 67 million tetrahedral cells. We evaluated both an LRU cache and our own DL method. For fair comparison between the two approaches, we restrict the number of vertices held in

memory to be no greater than a vertex memory threshold M. For LRU, this value determines the size of the cache used, while for DL, it limits the size of the buffer used for reading from the vertex file.

We also consider the effect of the *file system cache* on performance. This cache is maintained by the operating system, and is most effective with sequential access patterns, but still somewhat affects performance in our experiments. *Cold Cache* results were obtained by explicitly clearing the filesystem cache of data between runs. *Warm Cache* experiments allowed the filesystem cache to accumulate data between runs.

When evaluating the performance of the partitioning step, we choose 4x4 partitions.

4.5.1 LRU Read Performance

In this section, we describe the performance of an LRU cache while reading a large mesh. The LRU cache is implemented using both a hash table and a doubly linked list. The list is ordered by access time, simplifying maintenance of the LRU replacement policy, while the hash table allows $O(1)$ access to cache blocks.

To gain insight into LRU performance, we explored the effect of changing some parameters. Figure 4.8 visualizes loading time for the entire 2D mesh with various combinations of M and the number of blocks used for the cache. Note that increasing the number of blocks also decreases their size for a given value of M. We ran this experiment with both a warm and cold filesystem cache.

The hump along the rear plane of figure 4.8 shows that when the number of cache blocks is relatively small, loading time is particularly sensitive to M, the overall size of the cache. Case A of Table 4.2 yields further insight. The number of cache misses decreases as M increases, due to the corresponding increase in block size. The competing factor is the cost of reading larger blocks. The hump in the middle occurs because this increased I/O cost overwhelms the benefit of a lower miss rate. It is only when blocks are much larger that the cache is able to demonstrate a benefit (due to spatial locality) in spite of the increased

I/O cost.

The front of the surface in figure 4.8, corresponding to 256×1024 cache blocks behaves much more uniformly. Case B in Table 4.2 shows that the number of cache misses behaves similarly to case A, but the loading time is dramatically smaller, and monotonically decreasing with increasing M. This contrast with case A is due to the large number of cache blocks in case B, which consequently has much smaller blocks. Each block holds data for only a handful of vertices. That such a fine-grained cache would outperform other configurations indicates that the access pattern has little spatial locality, and performance is best when the I/O cost for reading a block is minimized. Indeed, it is likely that the LRU cache is often more a hindrance than a help.

Table 4.2. Number of cache misses ($\times 10^6$) and load time for loading 2D vertex file in two cases: (A) block Number = 512 and (B) block Number = 256×2^{10}, warm cache system

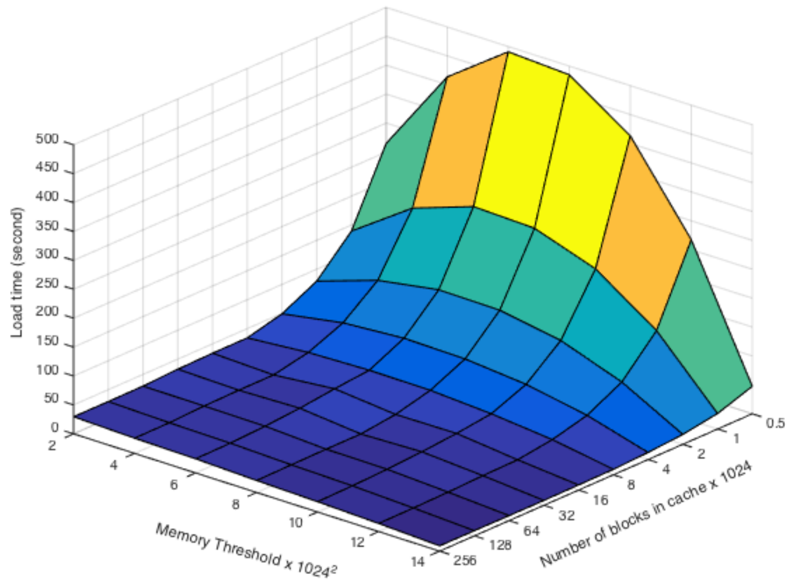
Thrshld M ($\times 2^{20}$)	2	4	6	8	10	12	14
# misses (case A)	14.1	11.79	9.49	7.01	4.84	2.59	0.34
Load time (second)	264.8	413.7	490.9	484.9	413.2	267.2	47.49
# misses (case B)	13.2	11.10	8.99	6.89	4.79	2.69	0.58
Load time (second)	29.53	26.3	24.4	20.7	17.8	14.1	10.64

4.5.2 Direct Load Read Performance

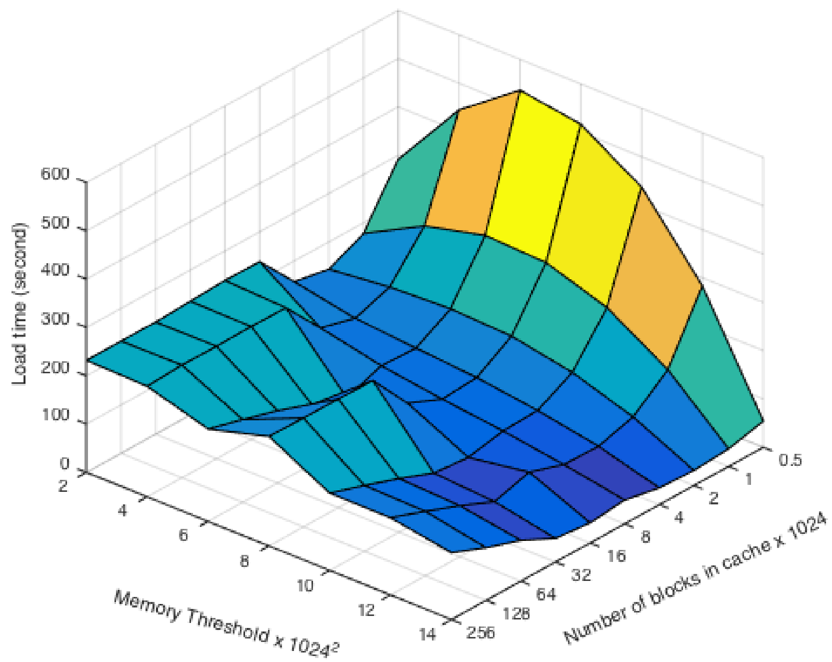
Given that the access pattern presented to the vertex file has very little spatial locality, we developed our own DL method to take particular advantage of *temporal* locality without incurring the costs associated with ineffective caching.

As described in section 4.3.1, DL reads consecutive chunks of cell data from the cell file, where each cell consists of several vertex indices pointing into the vertex file. We can then determine for each chunk which vertices (actually segments of the vertex file) are needed. Some vertices are needed repeatedly, but DL reads them only once for each chunk, thus taking advantage of temporal locality.

This process starts over again for each successive chunk, so vertex data is not retained

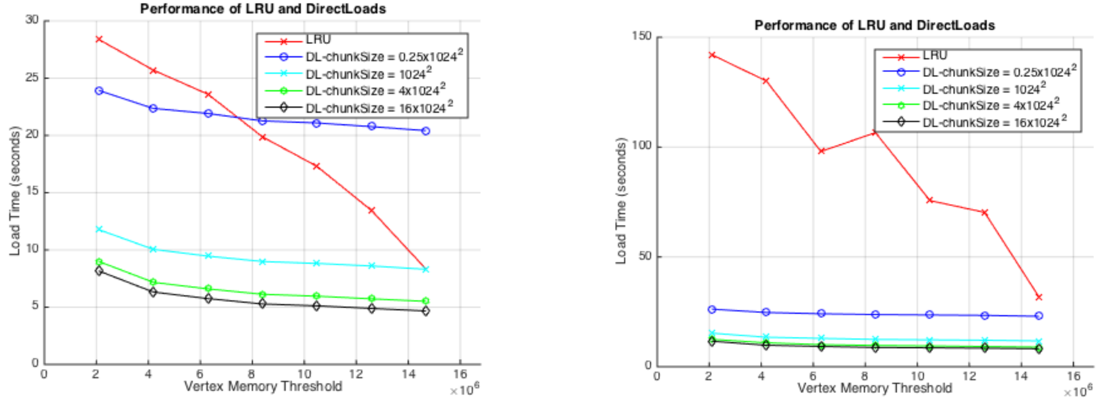


(a) Warm system cache



(b) Cold system cache

Figure 4.8. LRU execution times for 2D dataset with chunkSize = 128×1024 *vertices, 2D. Lower values are better.*



(a) Load time of LRU and DLs, warm system cache

(b) Load time of LRU and DLs, cold system cache

Figure 4.9. 2D Load performance using LRU and Direct Load

across chunks. However, DL's ability to exploit temporal locality increases as the chunk size is increased. Figure 4.9 shows DL performance for several different chunk sizes and varying values of M. Larger chunk sizes significantly improve performance. The figure also shows the performance of LRU in the best cases, which are clearly slower than DL except for the smallest chunk size. Specifically, DL with $M = 8\ million$ vertices and $chunksize = 16\ million$ vertices is about 4 times faster than the best LRU performance for the same value of M.

DL achieves this significant improvement by significantly reducing the number of seeks and reads because DL reads M vertices at once, while the LRU cache divides this same volume of memory into a large number of small blocks which must each be read separately. At the same time, DL is able to skip over segments that contain no required vertices using the bitmap described in section 4.3.1.

4.5.3 Performance of partitioning processes on CPU and GPU

After loading data from CPU, all vertex data (coordinates and attributes) for cells are copied into a coordinate array ($coorArr$ or $chunk$) with a relatively good locality (figure 4.5.b). This coordinate array will be processed, distributing cells to different partition elements. In this experiment, we compare partitioning process performance on both CPU and

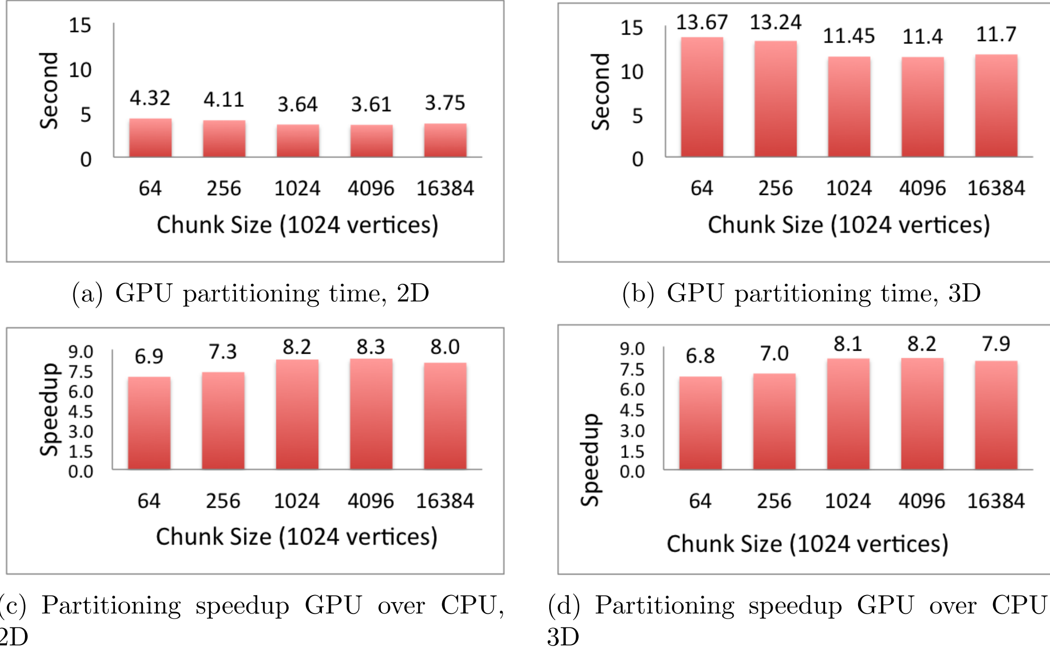


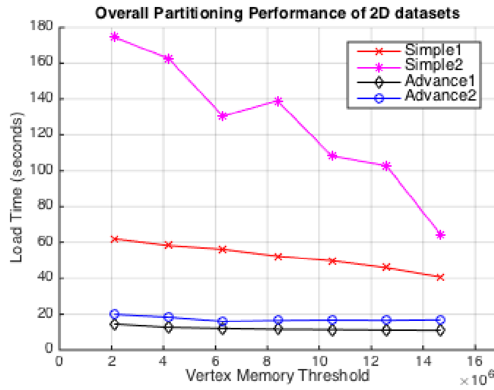
Figure 4.10. Partitioning process performance of GPU over CPU (without read performance). 2D CPU time was approximately 30 seconds, and 3D time was about 93 seconds.

GPU. First, we evaluated the single-CPU partitioning process with different *chunkSizes*, and found results to be very similar. This is because partitioning execution time only depends on the number of cells. In all cases, the execution time for the partitioning process is about 30 and 93 seconds for 2D and 3D datasets respectively.

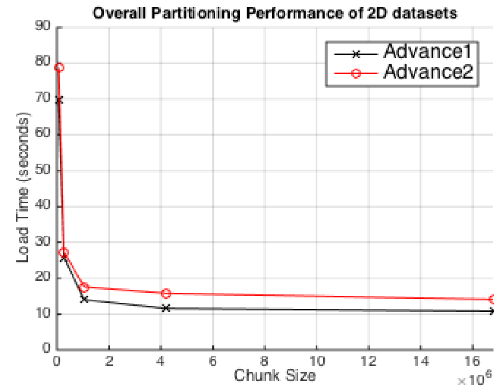
We also implemented the partitioning process on the GPU. Since the GPU organizes thousands of threads running in parallel, the partitioning performance improves drastically. Figures 4.10.a and 4.10.b show the execution times of the partitioning process for both 2D and 3D datasets on the GPU. We test with different *chunkSizes* from 64×2^{10} to 16×2^{20}. The best case happens when *chunkSize* is 4096×2^{10}, but performance is not very sensitive to *chunkSize*, as expected. Figures 4.10.c, 4.10.d present speedups from 6× to 8× for the partitioning process on the GPU compared to the CPU.

4.5.4 Overall partitioning performance

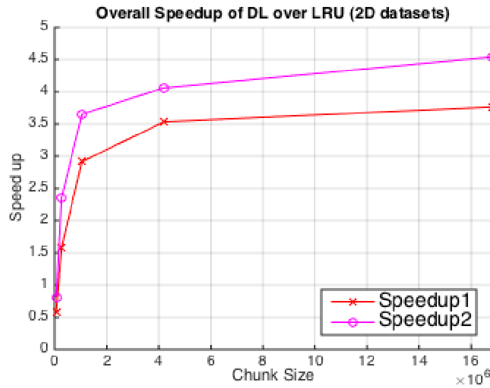
In this experiment we combine all execution times of Load, Partition and Update phases into one value. We compare the performance of two cases: *Simple* and *Advanced*.



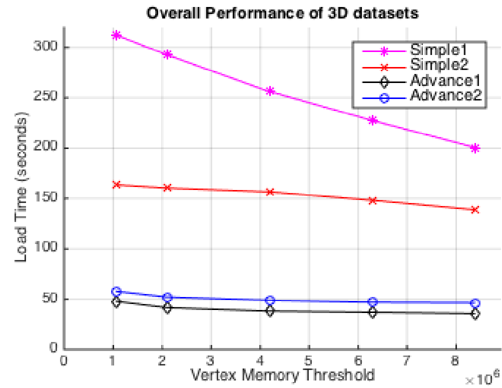
(a) Simple1, Simple2: loading data by LRU method and partitioning cells on CPU. Advance1, Advance2: loading data by DL method and partitioning cells on GPU with $M = 16 \times 2^{20}$. (1 and 2 are hot and cold system cache)



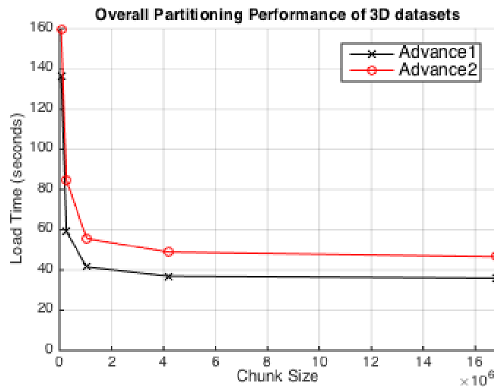
(b) Advance1, Advance2: loading data by DL method and partitioning cells on GPU with $M = 14 \times 2^{20}$. (1 and 2 are warm and cold system cache)



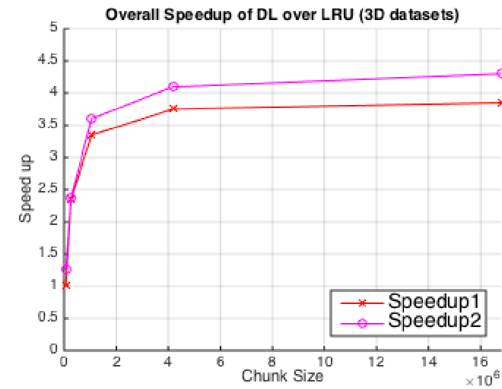
(c) Speedup1, Speedup2: Overall Partitioning speedup use DL for loading data and partitioning by GPU over the best case of Normal1, Normal2.



(d) 3D results corresponding to (a)



(e) 3D results corresponding to (b)



(f) 3D results corresponding to (c)

Figure 4.11. Overall partitioning performances between Simple *and* Advanced *cases.*

Simple uses LRU for loading and partitions the cells using the CPU. *Advanced* used DL for loading, and runs the GPU version of the partitioning code.

Figures 4.11.a,.d compare the overall performance of *Simple* and *Advanced.* The performance of *Simple* is sensitive to both the vertex memory threshold M and *chunkSize* while *Advanced* performance depends somewhat on *chunkSize*, and very little on M.

Figures 4.11.b,.e show the execution time of *Advanced* over a range *chunkSizes* in two cases: warm and cold system cache. The best values of *Simple1*, *Simple2* are compared with *Advanced1*, *Advanced2* of figure 4.11.b,.e to get speedups in figure 4.11.c,.f

4.5.5 Range Queries

We performed several range query experiments on CPU and GPU using 2D datasets. The range query running time is related to the number of cells and number of grid points in a partition element. The complexity of the interpolation function also affects the query performance significantly. In this experiment, we choose a range query bounding box covering the whole domain space, thus, each intersection is a partition element in the domain.

We use the 2D datasets, containing over 30 million triangular cells. We test the performance of range queries based on various numbers of grid points in a partition element and various numbers of partition elements. The experimental results in figure 4.12 show significant speedup over the serial version.

Assuming a fixed number of cells within the domain space, we analyze the speedup based on the number of grid points and the number of partition elements. The CPU implementation uses a loop to iterate over the set of grid points, checking cells for containment, and moving to the next point when the containing cell is found. The GPU implementation has no such loop, since the computation is parallelized over the set of cells. The GPU therefore does more work than the CPU version, since all cells are checked simultaneously with respect to each point. Nevertheless, the GPU takes advantage of the power of 2048 cores running simultaneously, and the range query performance improves drastically.

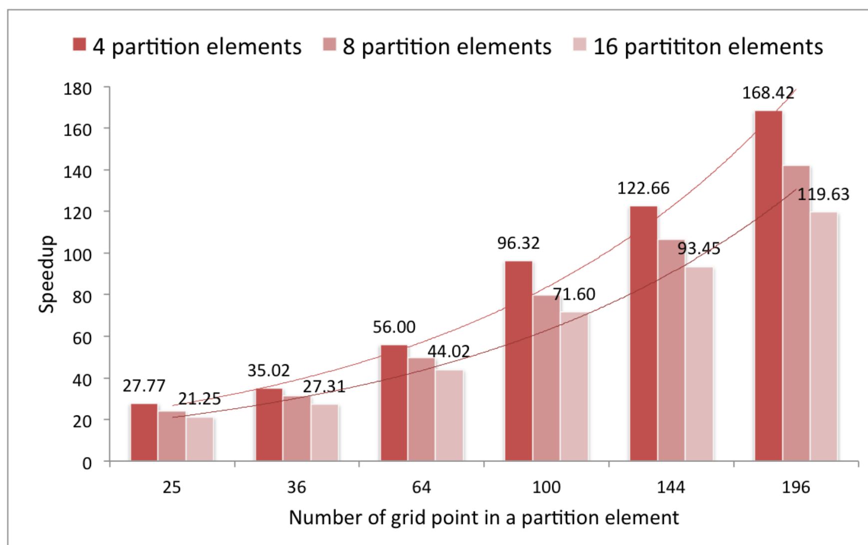


Figure 4.12. The speedup of GPU range query compared to CPU on 2D datasets

In figure 4.12, for the same number of grid points in a partition element, the performance slightly degrades as the number of partition elements intersected by the query bounds increases. This is because we invoke a kernel for each partition element, meaning all partition elements in the bounding box are processed sequentially on the GPU. This problem can be solved by making the GPU workload as large as possible. With the current implementation, we can do this by choosing a coarse partitioning in which each partition matches the GPU's capacity. In future, we also hope to evaluate performance with multiple kernels processing partition elements at the same time. This scheme will allow us to decouple the partitioning granularity from GPU characteristics.

4.5.6 Chapter Summary

We have accelerated range queries for very large unstructured mesh datasets via the GPU, producing very significant speedup over the serial version. This improvement requires not only parallelization of the range query itself, but also reorganization of the original dataset to improve locality both when accessing files and memory. We also accelerated the reorganization process by applying GPU power to the computationally expensive intersection tests required to assign cells to the appropriate partitions. At the disk level, we compared

LRU cache performance with our own DL method for reading the vertex coordinates required by a collection of cells. DL was found to be dramatically faster.

CHAPTER 5

LOAD BALANCING FOR A LARGE-SCALE UNSTRUCTURED MESHES

Performance of mesh applications running on parallel system is significantly influenced by the quality of a partitioning unstructured meshes. There are variety partitioning approaches which are classified into two categories, *topological* and *geometric* approach. Choosing a partitioning approach is highly specific to the unstructured mesh applications. In this chapter, we analyze two common approaches of mesh partitioning for a range query application and design a new algorithm for partitioning a large-scale unstructured meshes.

5.1 Background

Unstructured meshes contain points that are arbitrarily distributed in space. Because of their irregular nature, unstructured meshes are very flexible, but also much more difficult to process efficiently. Organizing the unstructured mesh file system and manipulating the access pattern influence the range query performance.

5.1.1 Data Structures

The data structure of unstructured meshes can be organized into several files. The first file, known as a *cell file* consists of tuples of indices into the vertex file. One triangular cell require three vertexIDs while a tetrahedron needs four. The *vertex file* contains all data associated with vertices, including tuples of coordinates and many attributes of a vertex, for example, in a weather application the attributes might be temperature, humidity, pressure, etc. Figure 5.1.1 presents the structure of two files.

The size of *cell file* is relatively far different from the *vertex file.* While *cell file* only contains the indices, *vertex file* consists of all important data including coordinates and

attributes of vertices. Assume each data size of index, coordinate, and attribute is 8 bytes. If N is the number of attributes of a vertex, then the proportion between *cell file* over *vertex file* is 1/(2+N) in 2D and 1/(3+N) in 3D domain. In reality, number of attributes of a vertex can be up to twenty, therefore, with large-scale mesh data, the *vertex file* is very large (terabytes).

5.1.2 Unstructured mesh access pattern

The unstructured mesh dataset basically organized into two files, cell file and vertex file (see figure 4.1). The cell file contains only references to the second file that includes all point coordinates and attributes. When loading a partition including multiple mesh elements, the cell file should be read first to collect elements in term of indices. Based on the array of indices from reading cell file, the second read will collect coordinates and attributes of points belong to the partition. When a domain is partitioned into multiple partitions, the only cell file is divided into multiple files.

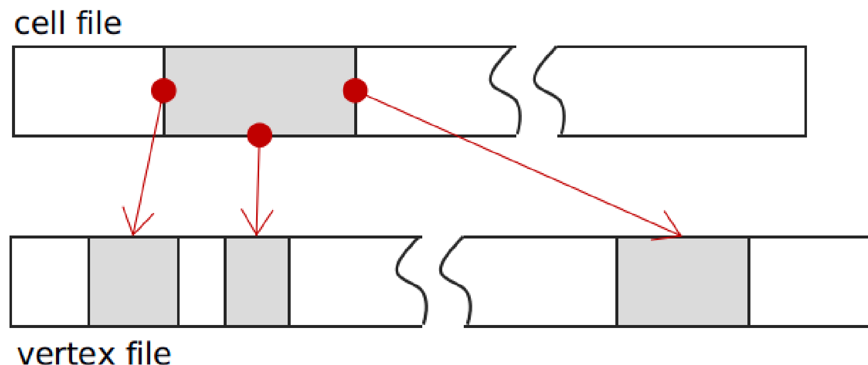


Figure 5.1. Two layers access pattern of an unstructured mesh files. The coordinates and attributes data of a group of cell file *may not be continuously in one range.*

5.1.3 Ceph- a distributed file system

The unstructured mesh data files are organized into two files, in which the *vertex file* can be very large (discussed in 5.1.1) so that the storage space of *vertex file* can surpass the storage capacity of a normal node in a cluster. In this case, *Ceph* [96], a distributed filesystem can be an acceptable solution.

Ceph handles data I/O for compute nodes across the cluster. It provides excellent performance, reliability, and scalability. One of important function of Ceph is CRUSH [97]. It is scalable data distribution function designed for distributed object-based storage systems.

5.1.4 Range query

Range query unstructured meshes is a task to collect attribute data in an arbitrary rectangular area or cuboid volume (i.e. *range*) within the domain space or to *resample* the data at regular intervals within the specified range. Figure 4.7 is an example for a 2D range query. The size of query's rectangular area can be small, therefore, in this case, there is no need to load full mesh files to memory. It means that, the mesh data should be designed in a way that the system can only read related mesh data for a small area of range query.

5.2 Partitioning unstructured meshes

Partitioning unstructured meshes into *partition elements* is a work to split the meshes into multiple regions to reduce the overall computation time in some mesh applications on parallel environment. The unstructured mesh domain should divided in such a way to satisfy two desirable requirements, load balancing and low communication. The workload is evenly distributed so that the load on all processors is balance. Furthermore, the communication cost between regions should be reduced as possible. Many partitioning algorithms have been proposed and they can be categorized into two general classes: geometric and topological. Choosing to use geometric or topological method is highly specific to the unstructured mesh applications.

5.2.1 Topological methods

Topological methods partition unstructured meshes based on the connectivity of the elements in the domain. In general, the meshes are modeled as a graph $G(V, E)$ in which the set of vertices V are mesh elements and set of edges E are the adjacency between elements. The unstructured mesh partitioning is a method to split the graph into many regions where

each region has mostly the same amount of elements. Beside, the communication between regions is also minimized. It means that the boundary triangles between regions are limited. These characteristics of graph partitioning method satisfy two requirements of mesh partitioning. The amount of elements for each region is mostly the same can reduce the waiting time between processors because they mostly finish running region meshes at the same time. Moreover, the less communication between region boundaries also reduce the communication between processors. For these reasons, the graph-based method can be able to generate the high quality of mesh partitioning.

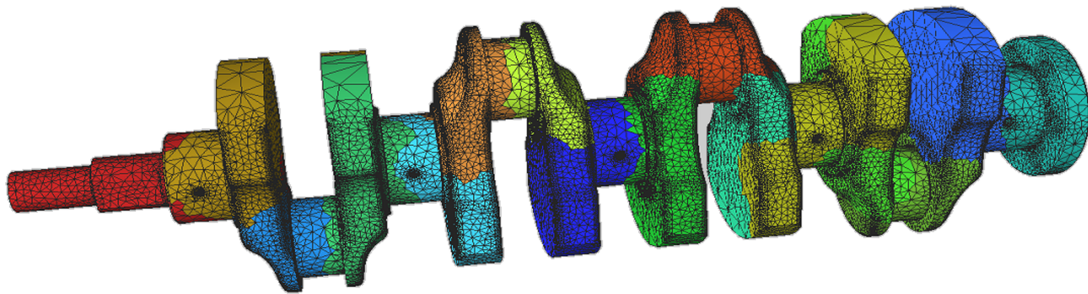


Figure 5.2. Mesh partitioning with graph-based method $(http : //www.simmetrix.com/index.php/technologies/parallel - meshing)$.

Since graph-based method can be able to partition a mesh domain in very high quality, it is widely used in many applications including fluid dynamic simulation, finite element computation, aerospace dynamic, etc. In visualization, it used to partition complex objects (see figure 5.2). A popular tool widely used for many unstructured mesh partitioning is ParMetis [46, 45] which uses graph-based algorithms. ParMetis is MPI-based parallel library, and works efficiently with a large-scale unstructured meshes. All partitioning regions are well balance, however, graph-based methods do not well apply all mesh applications. Specifically, for the applications that focus on rectangular areas in the domain such as unstructured mesh query [65, 69, 92], the query regions are commonly represented as a rectangular shape (see section 4.4 for more detail). On the other hand, the partitioning regions, using graph-based methods, have arbitrary shapes which would not work very well in a small range query case, meaning that reading the whole mesh files for a small rectangular area of range query would

degrade performance significantly.

5.2.2 Quadtree

The quadtree [5] is a data structure that organizes data in term of tree structure, in which each node has 4 children nodes in form of 2×2 squares of North-West (NW), North-East (NE), South-West (SW), and South-East (SE).

Quadtree can be considered as a hierarchical spatial structure which each child level corresponds to a further subdivision of the domain space. Based on the threshold argument (highest number of mesh elements in a child quadrant) and the element belongs to a quadrant, it can be recursively subdividing it into four quadrants or regions.

5.2.3 Geometric methods

Instead of exploiting the connectivity of topological method, geometric methods partition the unstructured meshes by dealing with the location of the elements in the domain. Many geometric approaches [37, 42, 70, 7, 100, 99, 41, 7, 101, 85, 16] have been proposed. Space-filling or Morton curves [43, 71, 7, 41, 85, 16] are useful methods to map elements in multi-dimensional data into one dimension and still guarantees the spatial locality characteristic. It means that neighboring elements of an element in the domain are also stay close together in one dimensional file. Morton curves (or Z-order), on the other hand, partition with recursively splitting in four quadrants density areas in the domain into an array partitions which can results from a depth-first traversal of a quadtree. Basically, both methods are relatively similar to the way of reorganization multi-dimensional dataset into one dimension. The common advantage geometric methods is easy to implementation, however, the partitioning quality is low. It means that the number of elements in each partition may be far different, especially for the non-uniform data distribution. Some partitioning applications gain high quality of partitioning in term of well balancing such as binary decomposition [11], or KD-tree [101], yet, they seem to have more interface boundary with larger influences to communications. It may have a trade-off between mitigating the waiting time and the

communication time.

To satisfy the partitioning requirements (load balancing and less communication), several improvements should be made. Since the load is imbalance, fine-grained partitioning would mitigate the load imbalance. The problem now is simplified to solve how to distribute the workloads to the group of processors such that the total elements for each processor would be similar [100, 99, 41]. However, fine-grained partitioning may cause the communication between processors becomes worse. In particular, the boundary elements (triangles) on a partition should be sent to neighbored partitions for the their computations.

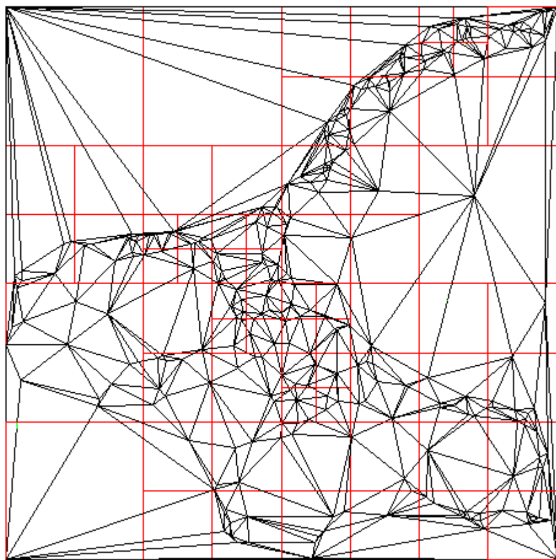
To reduce the boundary element communications during the computations, our solution is to duplicate the boundary triangles for all subdivisions. This may increases the data size, especially, it is even severely with large-scale data size and fine-grained partitioning. One solution for saving storage is called *Owner and Borrower* [79, 1, 3, 2] which dealing with the span partition element boundaries without duplication storing the boundary elements. This can save storage space to unstructured mesh data. Nevertheless, the price to pay for this saving is number of times of reading mesh data. To load mesh data for a partition, instead of read once to mesh elements that belong to a partition, it has to make several reads. The first read is to collect the *Owner* data, then the next several reads to get all *Borrower* data which are located in other partitions. For instances, it would be extra three reads in 2D dataset, and seven with 3D case. Figure 4.2 the shows the *Owner and Borrower* elements in the domain.

This is the trade-off between storage and computation that many computer engineers would sacrifice the storage space for better execution time. In some cases, the execution time would be the better option. This solution can be acceptable for two reasons, first, the hardware storage is cheaper recently, and second, the duplication seems to less heavy because of the unstructured mesh file structures (see subsection 5.1.2). In particular, we only duplicate the reference (indices to points) or elements (triangles) related to boundary interface (boundary triangles) between partitions. It means that the some parts of cell

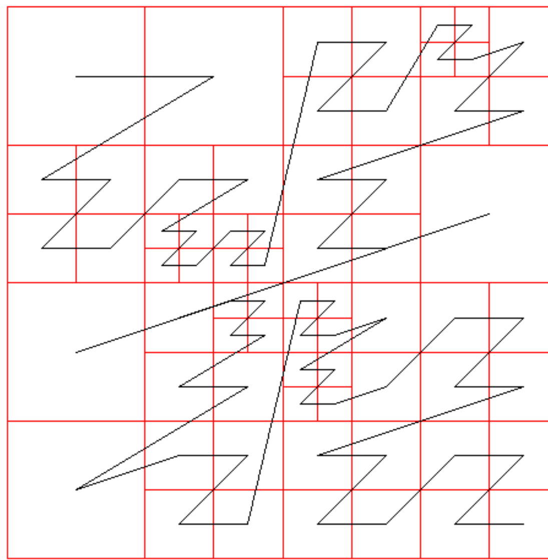
file are duplicated, and the vertex file does not change. As describing in 5.1.1, the *vertex file* stores most important information of unstructured meshes data, while *cell file* only contains references, therefore, *vertex file* is very large for the big unstructured mesh. Hence, duplicating some parts in *cell file* will not affect much.

5.3 Load balancing

A large-scale mesh applications desires a high level of parallelism to gain an acceptable performance. The efficiency of mesh applications mostly depends on the partitioning approaches for distribution workloads across processors. In this section, we will address how to partitioning a large-scale mesh dataset in parallel and schedule partitions to processors with good load balacing. As discussed in section 5.3.1, we would choose geometric method to decompose the mesh domain such as Morton curve with quadtree (see figure 5.3).



(a) Partitioning with recursive method.



(b) Morton curves

Figure 5.3. Recursive partitioning the small mesh sample of North Carolina coast. The sample have 300 elements, and threshold = 50.

5.3.1 Partitioning a large-scale unstructured meshes with quadtree

Partitioning an unstructured mesh domain is a method to split the domain into multiple partitions. In particular, we use Morton curve and quadtree data structure to divide the mesh domain into many square shape partitions. A root node containing entire mesh domain will be recursively partitioned into multiple tree levels in which each node contains four quadrant nodes. Each node in the quadtree consists of bounding box that contains mesh elements. The traditional algorithm of mesh partitioning is to decompose the condensed quadtree node that has more number of elements than a threshold. Finally, mesh partitions that are subdivided will stay in the leaf nodes of quadtree. The middle nodes contains only empty bounding box.

The algorithm of subdivision of an unstructured mesh with quadtree is relatively simple, however, it needs completely mesh loading to memory for the decision of decomposition. This would become impractical for a very large unstructured meshes, which often have hundreds of millions to billions of cells. This overflow problem can be solved with external memory by swapping data in and out of main memory. Yet, it would dramatically degrade performance.

In the big-data era, the power of computation for large datasets has been beyond the serial computation. The current trend of designing and processing unstructured meshes is moving toward parallelization. We will presents a new algorithm to partitioning a large-scale unstructured meshes in a cluster with multiple processes running at a same time.

The main idea of the algorithm is to share the workload to many processes and let them to recursively partition by themselves. Several steps are presented in algorithm 6. Step 1 is recursive partitioning in parallel. At first, each process reads an equally amount of elements from mesh domain which is stored in two data files (cell and vertex files) with function $readElements(E, \#process)$). After worker processes get their elements ($setElements$), all processes run $recursivePartitioning$ function to partition in parallel to generate their own sub quadtrees qt. This function works recursively based on the number of elements, the bounding

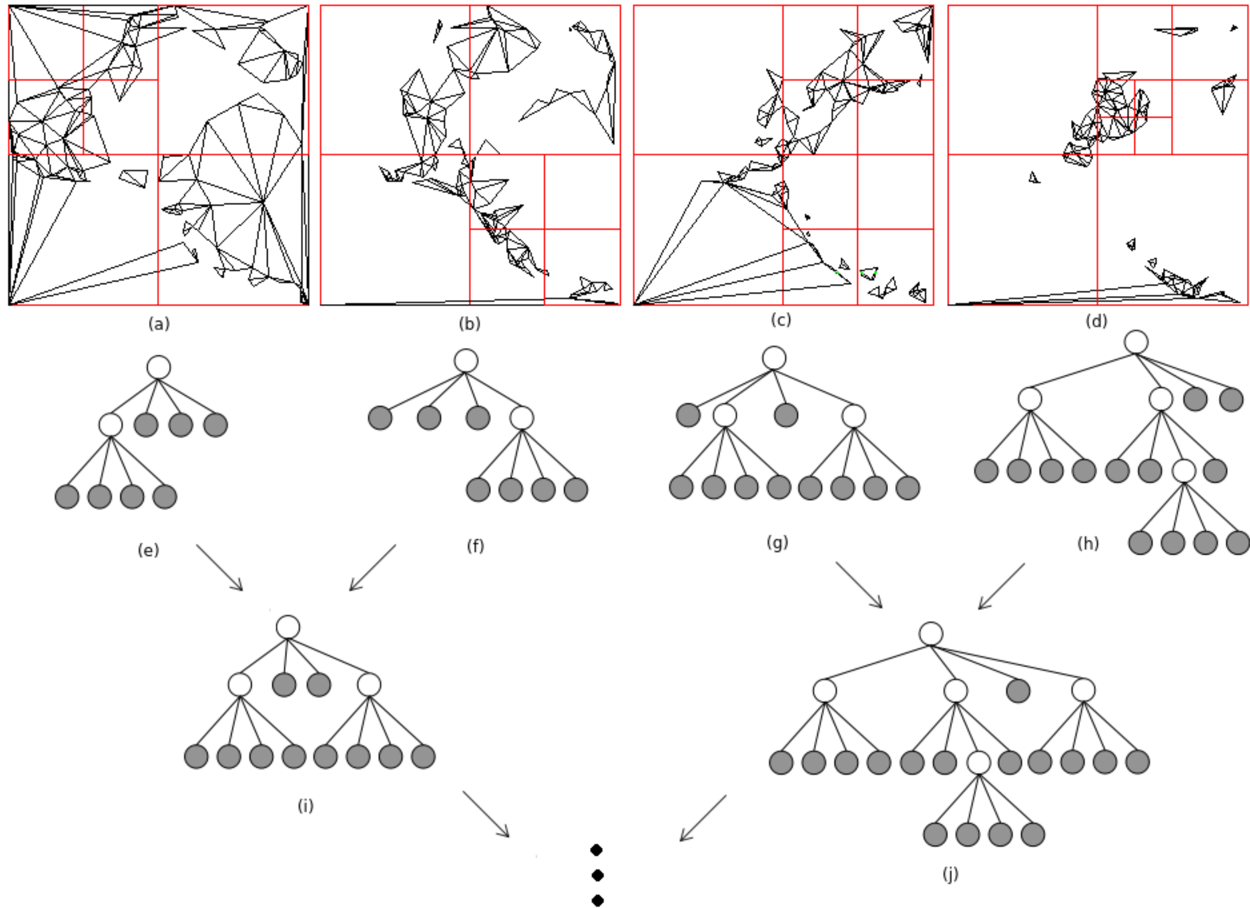


Figure 5.4. Partitioning the unstructured meshes with Z-order. (a), (b), (c), (d) are four processes. Each process partitions one fourth of meshes. (e), (f), (g), (h) are sub-quadtrees that are generated from processes. (i), (j) are first level merged sub-quadtrees. The last task is to merge all sub-quadtrees into the final result.

box in the current node of the quadtree until $setElements$ is smaller than threshold T.

Since each process partitions on a subset of total elements E, their sub quadtrees are relatively shallow. In order to have a final result, all sub quadtrees in worker processes should merge into a large one. This would repeat the overflow problem if all sub quadtrees in worker processes are gathered to master process. The solution for this issue is to merge the structure of sub quadtrees. It means that we will not gather elements in sub quadtrees of worker processes, instead, we merge quadtree structures only. The merge is similar to the parallel reduction problems. Instead of using basic operators and normal numbers of the operands for the reduction, the operator in this case is merging and operands are quadtree

Algorithm 6: Partitioning unstructured meshed into partitions using quadtree in parallel with MPI

Input: *All elements E (cells/triangles), threshold T, domain bounding box B, number of processes #process.*

Output: *A quadtree contains all mesh partitions in leaf nodes.*

1 Step 1: recursively partitioning with quadtree in parallel. Leaf nodes contain elements that intersect the bounding boxes (see figure 5.4):
2 $setElements \leftarrow readElements(E, \#process))$
3 $qt \leftarrow recursivePartitioning(setElements, B, T)$
4 Step 2: merge quadtree structure of qt (merge quadtree structure only, no elements involve) in all processes into a large quadtree bigQt and update to qt in processes:
5 $bigQt \leftarrow reduceQuadTreeStructure(qt)$
6 $bigQt \leftarrow mergeQuadTree(qt, bigQt)$
7 $bigQt \leftarrow moveElements(bigQt)$
8 Step 3: update number of elements in all leaf nodes of the quadtree in master process based on leaf nodes of quadtrees of all processes:
9 Step 3: update number of elements in all leaf nodes of the quadtree in master process based on leaf nodes of quadtrees of all processes:
10 $stop \leftarrow false$
11 **while** $stop == false$ **do**
12   **for** *each* $leafNode_i \in getLeafNodes(bigQt)$ **do**
13         $leafNode_i.\|element\| \leftarrow gatherElementsFromProcesses(leafNode_i)$
14   **end**
15     $stop \leftarrow true$
16   **for** *each* $leafNode_i \in getLeafNodes(bigQt)$ **do**
17   **if** $leafNode_i.\|elements\| > T$ **then**
18             $leafNode_i \leftarrow decompose(leafNode_i)$
19             $stop \leftarrow false$
20   **end**
21   **end**
22 **end**

structures. The function *reduceQuadTreeStructure* merges all sub quadtree structures in processes into one big structure (*bigQt*) at master process. Figure 5.4 (i) and (j) shows the first merged level. After one merged level, number of processes joining for the merging reduces a half. The reduction would continue until the last level which has only one last process. This process normally scheduled as master.

Other processes should have the same structure as *bigQt* for further updating, there-

fore, function $mergeQuadTree(qt, bigQt)$ should be used to merge $bigQt$ to sub quadtree qt in each process. Eventually, all quadtrees in worker processes have the same quadtree structure with $bigQt$. After merging, some internal nodes may have elements that should moved to leaf nodes. The *moveElements* function does the job based on the intersection between triangles and the bounding boxes.

After step 2, quadtree structures in all processes are the same, the only difference is the number of elements in their leaf nodes. The final step should combine elements in leaf nodes in processes, specifically, for a specific leaf node of $bigQT$, we collect a number of elements ($\|elements\|$) in the same leaf node but different $bigQT$ of processes. Again, we do not collect all elements, just find the total number elements in a leaf node because if it is greater than threshold T, then the leaf node should be decompose one more time. The task will repeat until no leaf nodes should be decomposed. Figure 5.3 is a example of partitioning unstructured meshes of the North Carolina ocean with 300 elements and threshold is 50.

CHAPTER 6

RELATED RESEARCH

In this chapter, we are going to review the previous research related to large-scale parallel Delaunay triangulation and unstructured meshes accessing.

6.1 Parallel Delaunay Triangulation

There has been extensive research on Parallel Delaunay Triangulation as well as out-of-core Delaunay Triangulation. However, the combination of both parallelism and out-of-core triangulation is rare. Parallel approaches run on a single machine rather than over a cluster.

Cignoni [26], Hardwick [40], Chen [23, 22], Blelloch [13], Fuetterling [35],and Lin [52] employed parallel Delaunay algorithms based on the divide-and-conquer method in which an initial mesh is recursively divided into sub-regions and each of them assigned to a processor. These regions are further triangulated simultaneously, and later joined into one domain. The method achieves a certain level of parallelism; however, the joining of separated regions is challenging. This is the “stitching problem” mentioned in chapter 2.2. Since the sub-regions have independently processed triangulation, each region must have its own convex hull. After triangulation for each region, the triangles in the sub-regions have been updated. The merging of sub-region triangles to form a final mesh can be quite difficult, especially if preserving the delaunay criteria.

Some research also divides the domain into smaller regions without recursion. Instead, they directly divide the domain into many areas. Lo, et al. [53, 54, 55] divided the rectangular/cuboid domain into many partitions and triangulated partitions simultaneously.

In this case, the corners of the domain are used as the convex hull for each partition's triangulation. Since the triangulation has been processed in parallel, the performance improves significantly. However, the joining of partitions into the final mesh still presents a stitching problem. Smolik et al. [89] used partitioning and triangulation methods similar to Lo, et al., but address the stitching problem differently. However, the final mesh is not delaunay, which is an important property for many applications, since it promotes accurate interpolation of data values.

Conversely, the research from Remacle et al. [78] describes an incremental insertion algorithm which does not assign a region to each processor; instead, they use multiple threads via *OpenMP* [28] to run DT simultaneously in a shared memory environment. The performance suffers if the cavities created by threads intersect with each other. In this case, only one thread can update the triangles. To minimize the conflict, Remacle sorts vertices using a Hilbert curve, and distributes points to each thread such that points from threads are not geometrically close. Blandford et al. [12] also use OpenMP to generate a very large 3D DT using a 64 core SMT machine. They also improve DT performance using a special data structure for facilitating DT operations. While both methods hold the dataset in shared memory, our own work scales well beyond the limits of a single machine because of our distributed approach.

Various parallel algorithms for incremental insertion avoid conflicts by assigning a region of the mesh to each processor. Chrisochoides et al. [33] triangulate independent sub-regions in parallel. The sub-regions are selected and scheduled to processors from a refinement queue such that they do not share the same boundary tetrahedra. Hence, the communication latency between processors is eliminated, improving performance. The algorithm works well for uniform data, but suffers from poor load balancing in the non-uniform case.

Chen et al. [24] present a method for localizing DT computation. While the serial implementation is only half as fast as other methods, the high efficiency of the parallel

implementation yields performance that exceeds several competing tools. However, this method runs on a single multi-core machine, and is not applied to very large meshes.

Aside from parallelism, many studies have focused on large dataset triangulation using out-of-core methods. Isenburg [44] reports a DT method which can process a very large number of triangles (up to many billions) with relativey good performance. The domain is divided into small regions and loads new partition to triangulate after the previous one is done. There are not so many current triangles in the memory because when the last point in a region is inserted and triangulated, a significant number of triangles are finalized. Since these triangles do not affect the DT of other regions, they are stored in the external memory. However, this method is limited to a single machine, with no parallel implementation.

Several researchers employ a space-filling Hilbert curve to improve locality [15, 47, 57]. For example, Kohout et al. recursively split the initial mesh into regions with the help of a lifting transformation [36, 76]. Partitions are then sequentially triangulated and stored to files. The final step is to merge the partitions into the final result by updating the connectivity between triangles of adjacent partitions. This step appears to be a stitching problem, but details are not given.

In contrast, as TIPP adds points to the triangulation, it maintains the Delaunay property over the entire global mesh, so stitching is unnecessary. However, TIPP does this without requiring worker processes to coordinate while triangulating, which avoids communication and contention along the partition boundaries.

6.2 Load balancing

Load balancing is an important consideration for any parallel mesh algorithm, especially when applied to non-uniform point distributions. Like our own work, the PLUM system [67] uses a coarse initial mesh to inform the load balancing process. Many researchers have addressed this problem using geometric partitioning methods. Campbell and Remacle [17, 78] use a space filling curve to map points to rectangular units such that the number of

points for each unit is roughly the same. Other studies [56, 11] use *Quadtree* or *Octree* data structures to fairly divide the computational load.

6.3 Accessing unstructured mesh

Although the challenges of *big data* have recently been the focus of intense interest, comparatively little work has been done on spatial scientific data or unstructured meshes.

Thakur et al. [94, 93] improve I/O latency by aggregating multiple small, non-contiguous transactions into a few, large, contiguous transactions. The I/O performance improves significantly because they make as few read operations as possible. Taking advantage of parallel I/O, their ROMIO model uses MPI-IO to read and write all data with a single I/O function request.

Some work on reorganizing unstructured grids has been reported. Rhodes and Akande [79, 1, 2, 3] use a partitioning method to reorganize data for locality improvement. They use the owner-borrower scheme to solve the spanning problem for reducing duplicated data between partition elements. However, their implementation is not parallelized.

There have been many contributions to the field of graph and unstructured mesh partitioning. Devine et al. [32] have developed the *Zoltan* library based on recursive bisection, space-filling curves, and graph partitioning algorithms. The Zoltan library supports parallel partitioning and load balancing. Schloegel et al.[86] describe unstructured mesh partitioning based mainly on graph partitioning. Graph based partitioning methods usually yield high quality results [86], but they are usually in-core techniques and are therefore hard to scale to very large datasets. In contrast, our own partitioning method uses a piecewise approach that is applicable to datasets much larger than available memory. Similarly, our range query implementation collects only those partition elements belonging to the bounding box of the range, greatly reducing memory requirements and speeding performance.

Papadomanolakis et al. [21] uses a space filling curve method to reorganize the tetrahedral meshes for improving locality access to the datasets. Specifically, simplicial cells

located closed to each other are also stored together on disk. With this reorganization, data access is improved significantly. However, that work predates mainstream use of GPU Computing. Other research [82, 84, 68] addresses chunking or partitioning problems for multidimensional data, but not the unstructured meshes that are the focus of this dissertation.

There have been some efforts to implement unstructured mesh applications on GPUs. Solano-Quinde et al. [91, 90] uses iterative method until converges to compute values of solution points in the cells. Also, memory access pattern has been mentioned in detail, but not analysis the impacts from dataset pattern.

CHAPTER 7

CONCLUSION

We have developed a novel algorithm named TIPP to generate Delaunay triangulations for very large scale datasets. We have shown that the dataset domain can be decomposed into independent partitions that can be processed in parallel. TIPP also improves performance by identifying sets of triangles that can be finalized early, removing those triangles from memory, and reducing the cost of triangle search. The result is a distributed algorithm able to generate roughly 20 billion triangles.

We have also shown that the results of triangulated partitions fit perfectly together, preserving the Delaunay criteria. We do not require a stitching process that would introduce non-Delaunay triangles between partitions.

TIPP significantly improves the performance of Delaunay triangulation, bringing extreme-scale meshes into the realm of the feasible through its distributed approach. However, there is still room for improvement. We would like to implement a method of repairing the mesh boundary after removing artificial vertices and their triangles. Also, because we use partitions of uniform size, we do not yet perform load balancing between worker nodes. A third issue is the master node. When the number of partitions is large, the master node becomes a bottleneck, causing worker nodes to wait for new work. Empirical evaluation exposed some drawbacks to the algorithm that degrade performance for extremely large datasets. Specifically, when the number of partitions is large, master node performance becomes a bottleneck, causing worker processes to wait for new work. Furthermore, the original TIPP algorithm has no load-balancing mechanism, an essential feature for the non-uniform point distributions exhibited by real scientific datasets.

We improved the original single-master TIPP algorithm by introducing parallelism at two levels, distributing the burden of the master node across several sub-masters. Multi-master TIPP shows significant gains in performance over the previous version.

We also implemented both static and dynamic load-balancing strategies to address non-uniform point distribution. Although a producer-consumer implementation shows improvement over the original TIPP, a static adaptive strategy demonstrated the most significant gains over all.

In addition, we developed a new algorithm for a very large-scale unstructured mesh spatial-based partitioning in parallel.

One avenue for future work is to augment the existing implementation to produce additional information convenient for topological navigation of the computed mesh. Although our geometric approach has proven convenient for generating the mesh in parallel, as well as geometric queries, support for topological queries would be a useful addition.

Along these lines, we envision a distributed query engine that would take advantage of the triangulated mesh. Such a system might not gather triangulation results back to the master nodes, but instead leave this data on the workers, making it available to answer queries on unstructured datasets at unprecedented scale.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] O. O. Akande and P. J. Rhodes. Iteration aware prefetching for unstructured grids. In *Big Data, 2013 IEEE International Conference on*, pages 219–227. IEEE, 2013.

[2] O. O. Akande and P. J. Rhodes. Multilevel partitioning of large unstructured grids. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 317–322. IEEE, 2014.

[3] O. O. Akande and P. J. Rhodes. Towards an efficient storage and retrieval mechanism for large unstructured grids. *Future Generation Computer Systems*, 45:53–69, 2015.

[4] T. Akenine-Möller. Fast 3d triangle-box overlap testing. In *ACM SIGGRAPH 2005 Courses*, page 8. ACM, 2005.

[5] A. Angelo. A brief introduction to quadtrees and their applications. In *Style file from the 28th Canadian Conference on Computational Geometry*, 2016.

[6] L. Antiga, B. Ene-Iordache, L. Caverni, G. P. Cornalba, and A. Remuzzi. Geometric reconstruction for computational mesh generation of arterial bifurcations from ct angiography. *Computerized Medical Imaging and Graphics*, 26(4):227–235, 2002.

[7] M. Bader. *Space-filling curves: an introduction with applications in scientific computing*, volume 9. Springer Science & Business Media, 2012.

[8] B. Barber and H. Huhdanpaa. Qhull. *The Geometry Center, University of Minnesota, http://www. geom. umn. edu/software/qhull*, 1995.

[9] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)*, 22(4):469–483, 1996.

[10] M. Barrena, J. Hernández, J. Martínez, A. Polo, P. de Miguel, and M. Nieto. Multidimensional declustering methods for parallel database systems. In *Euro-Par'96 Parallel Processing*, pages 866–871. Springer, 1996.

[11] M. J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, (5):570–580, 1987.

[12] D. K. Blandford, G. E. Blelloch, and C. Kadow. Engineering a compact parallel delaunay algorithm in 3d. In *Proceedings of the twenty-second annual symposium on Computational geometry*, pages 292–300. ACM, 2006.

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

[14] A. Bowyer. Computing dirichlet tessellations. *The computer journal*, 24(2):162–166, 1981.

[15] K. Buchin. Incremental construction along space-filling curves. *EuroCG*, 5:17–20, 2005.

[16] C. Burstedde and T. Isaac. Morton curve segments produce no more than two distinct face-connected subdomains. *CoRR, abs/1505.05055 v2*, 2015.

[17] P. M. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and J. D. Teresco. Dynamic octree load balancing using space-filling curves. *Williams College Department of Computer Science, Tech. Rep. CS-03-01*, 2003.

[18] T.-T. Cao, A. Nanjappa, M. Gao, and T.-S. Tan. A gpu accelerated algorithm for 3d delaunay triangulation. In *Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 47–54. ACM, 2014.

[19] CGAL, Computational Geometry Algorithms Library.

[20] Chameleon project web site. http://www.chameleoncloud.org.

[21] L. T. Chen, R. Drach, M. Keating, S. Louis, D. Rotem, and A. Shoshani. Efficient organization and access of multi-dimensional datasets on tertiary storage systems. *Information Systems*, 20(2):155–183, 1995.

[22] M.-B. Chen. The merge phase of parallel divide-and-conquer scheme for 3d delaunay triangulation. In *Parallel and Distributed Processing with Applications (ISPA), 2010 International Symposium on*, pages 224–230. IEEE, 2010.

[23] M.-B. Chen, T.-R. Chuang, and J.-J. Wu. Efficient parallel implementations of 2d delaunay triangulation with high performance fortran. *Proceedings of 10th SIAM Conference on Parallel Processing for Scientific Computing*, 2001.

[24] R. Chen and C. Gotsman. Localizing the delaunay triangulation and its parallel implementation. In *Voronoi Diagrams in Science and Engineering (ISVD), 2012 Ninth International Symposium on*, pages 24–31. IEEE, 2012.

[25] N. Chrisochoides and D. Nave. Parallel delaunay mesh generation kernel. *International Journal for Numerical Methods in Engineering*, 58(2):161–176, 2003.

[26] P. Cignoni, C. Montani, R. Perego, and R. Scopigno. Parallel 3d delaunay triangulation. *Computer Graphics Forum*, 12:129–142, 1993.

[27] C. M. Cortis and R. A. Friesner. An automatic three-dimensional finite element mesh generation system for the poisson–boltzmann equation. *Journal of computational chemistry*, 18(13):1570–1590, 1997.

[28] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[29] J. R. Davy and P. Dew. *Removing Local Concavities from Delaunay Triangulations*. University of Leeds, School of Computer Studies, 1994.

[30] L. De Floriani, B. Falcidieno, and C. Pienovi. Delaunay-based representation of surfaces defined over arbitrarily shaped domains. *Computer Vision, Graphics, and Image Processing*, 32(1):127–140, 1985.

[31] B. Delaunay. Sur la sphère vide. *Bulletin of Academy of Sciences of the USSR*, pages 793–800, 1934.

[32] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science & Engineering*, 4(2):90–96, 2002.

[33] D. Feng, C. Tsolakis, A. N. Chernikov, and N. P. Chrisochoides. Scalable 3d hybrid parallel delaunay image-to-mesh conversion algorithm for distributed shared memory architectures. *Computer-Aided Design*, 85:10–19, 2017.

[34] S. Fortune. A sweepline algorithm for voronoi diagrams. *Algorithmica*, 2(1-4):153, 1987.

[35] V. Fuetterling, C. Lojewski, and F.-J. Pfreundt. High-performance delaunay triangulation for many-core computers. In *High Performance Graphics*, pages 97–104, 2014.

[36] Geometry: Combinatorics and algorithms. http://geometry.inf.ethz.ch.

[37] J. R. Gilbert, G. L. Miller, and S.-H. Teng. Geometric mesh partitioning: Implementation and experiments. *SIAM Journal on Scientific Computing*, 19(6):2091–2110, 1998.

[38] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[39] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 47–57. ACM, 1984.

[40] J. C. Hardwick. Implementation and evaluation of an efficient parallel delaunay triangulation algorithm. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 239–248. ACM, 1997.

[41] D. F. Harlacher, H. Klimach, S. Roller, C. Siebert, and F. Wolf. Dynamic load balancing for unstructured meshes on space-filling curves. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1661–1669. IEEE, 2012.

[42] M. T. Heath and P. Raghavan. A cartesian parallel nested dissection algorithm. *SIAM Journal on Matrix Analysis and Applications*, 16(1):235–253, 1995.

[43] D. Hilbert. Über die stetige abbildung einer linie auf ein flächenstück. In *Dritter Band: Analysis· Grundlagen der Mathematik· Physik Verschiedenes*, pages 1–2. Springer, 1935.

[44] M. Isenburg, Y. Liu, J. Shewchuk, and J. Snoeyink. Streaming computation of delaunay triangulations. *ACM transactions on graphics (TOG)*, 25(3):1049–1056, 2006.

[45] G. Karypis and K. Schloegel. Parallel graph partitioning and sparse matrix ordering. *University of Minnesota, Department of Computer Science and Engineering*, 2013.

[46] G. Karypis, K. Schloegel, and V. Kumar. Parmetis. *Parallel graph partitioning and sparse matrix ordering library. Version*, 2, 2003.

[47] J. Kohout and I. Kolingerová. Acut: Out-of-core delaunay triangulation of large terrain data sets. In *VMV*, pages 181–190, 2007.

[48] P. B. Laval. Mathematics for computer graphics-barycentric coordinates. 2003.

[49] B. S. Lee, R. R. Snapp, L. Chen, and I.-Y. Song. Modeling and querying scientific simulation mesh data, 2002.

[50] D.-T. Lee and B. J. Schachter. Two algorithms for constructing a delaunay triangulation. *International Journal of Computer & Information Sciences*, 9(3):219–242, 1980.

[51] J. Liebeherr, M. Nahas, and W. Si. Application-layer multicasting with delaunay triangulation overlays. *IEEE Journal on Selected Areas in Communications*, 20(8):1472–1488, 2002.

[52] J. Lin, R. Chen, C. Yang, Z. Shu, C. Wang, Y. Lin, and L. Wu. Distributed and parallel delaunay triangulation construction with balanced binary-tree model in cloud. In *Parallel and Distributed Computing (ISPDC), 2016 15th International Symposium on*, pages 107–113. IEEE, 2016.

[53] S. Lo. Parallel delaunay triangulation – application to two dimensions. *Finite Elements in Analysis and Design*, 62:37–48, 2012.

[54] S. Lo. Parallel delaunay triangulation in three dimensions. *Computer Methods in Applied Mechanics and Engineering*, 237:88–106, 2012.

[55] S. Lo. 3d delaunay triangulation of 1 billion points on a pc. *Finite Elements in Analysis and Design*, 102:65–73, 2015.

[56] P. MacNeice, K. M. Olson, C. Mobarry, R. De Fainchtein, and C. Packer. Paramesh: A parallel adaptive mesh refinement community toolkit. *Computer physics communications*, 126(3):330–354, 2000.

[57] C. Marot, J. Pellerin, and J.-F. Remacle. One machine, one minute, three billion tetrahedra. *International Journal for Numerical Methods in Engineering*, 117(9):967–990, 2019.

[58] D. J. Mavriplis. Adaptive mesh generation for viscous flows using triangulation. *Journal of computational Physics*, 90(2):271–291, 1990.

[59] D. Morozov and T. Peterka. Block-parallel data analysis with diy2. In *2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 29–36. IEEE, 2016.

[60] D. Morozov and T. Peterka. Efficient delaunay tessellation through kd tree decomposition. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 728–738. IEEE, 2016.

[61] F. Mueller. Pthreads library interface. *Florida State University*, 1993.

[62] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg. *OpenCL programming guide.* Pearson Education, 2011.

[63] J. R. Nanduri, F. A. Pino-Romainville, and I. Celik. Cfd mesh generation for biological flows: Geometry reconstruction using diagnostic images. *Computers & Fluids*, 38(5):1026–1032, 2009.

[64] nc_inundation_v6c.grd from ADCIRC.org, UNC-Chapel Hill. http://adcirc.org/products/grids/.

[65] C. Nguyen and P. J. Rhodes. Accelerating range queries for large-scale unstructured meshes. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 502–511. IEEE, 2016.

[66] B. Nichols, D. Buttlar, J. Farrell, and J. Farrell. *Pthreads programming: A POSIX standard for better multiprocessing.* O'Reilly Media, Inc., 1996.

[67] L. Oliker and R. Biswas. Plum: Parallel load balancing for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 52(2):150–177, 1998.

[68] E. J. Otoo, D. Rotem, and S. Seshadri. Optimal chunking of large multidimensional arrays for data warehousing. In *Proceedings of the ACM tenth international workshop on Data warehousing and OLAP*, pages 25–32. ACM, 2007.

[69] S. Papadomanolakis, A. Ailamaki, J. C. Lopez, T. Tu, D. R. O'Hallaron, and G. Heber. Efficient query processing on unstructured tetrahedral meshes. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 551–562, 2006.

[70] A. Patra and D. Kim. E cient mesh partitioning for adaptive hp finite element meshes. Technical report, Technical report, Dept. of Mech. Engr., SUNY at Bu alo, 1999.

[71] G. Peano. Sur une courbe, qui remplit toute une aire plane. *Mathematische Annalen*, 36(1):157–160, 1890.

[72] T. Peterka, D. Morozov, and C. Phillips. High-performance computation of distributed-memory parallel 3d voronoi and delaunay tessellation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 997–1007. IEEE Press, 2014.

[73] C. Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[74] M. Qi, T.-T. Cao, and T.-S. Tan. Computing 2d constrained delaunay triangulation using graphics hardware. *Technical Report/National University of Singapore, School of Computing.*, 3, 2011.

[75] M. Qi, T.-T. Cao, and T.-S. Tan. Computing 2d constrained delaunay triangulation using the gpu. *IEEE transactions on visualization and computer graphics*, 19(5):736–748, 2013.

[76] V. Rajan. Optimality of the delaunay triangulation in rd. *Discrete & Computational Geometry*, 12(2):189–202, 1994.

[77] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[78] J.-F. Remacle. A two-level multithreaded delaunay kernel. *Computer-Aided Design*, 85:2–9, 2017.

[79] P. J. Rhodes. *Granite: a scientific database model and implementation*. University of New Hampshire, 2004.

[80] P. Ridley. Guide to partitioning unstructured meshes for parallel computing, 2010.

[81] G. Rong, T.-S. Tan, T.-T. Cao, et al. Computing two-dimensional delaunay triangulation using graphics hardware. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 89–97. ACM, 2008.

[82] D. Rotem, E. J. Otoo, and S. Seshadri. Chunking of large multidimensional arrays. *Lawrence Berkeley National Laboratory*, 2007.

[83] F. Rusu and Y. Cheng. A survey on array storage, query languages, and systems. *arXiv preprint arXiv:1302.0103*, 2013.

[84] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *Data Engineering, 1994. Proceedings. 10th International Conference*, pages 328–336. IEEE, 1994.

[85] A. Sasidharan, J. M. Dennis, and M. Snir. A general space-filling curve algorithm for partitioning 2d meshes. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 875–879. IEEE, 2015.

[86] K. Schloegel, G. Karypis, and V. Kumar. *Graph partitioning for high performance scientific simulations*. Army High Performance Computing Research Center, 2000.

[87] S. Shepler, M. Eisler, D. Robinson, B. Callaghan, R. Thurlow, D. Noveck, and C. Beame. Network file system (nfs) version 4 protocol. *Network*, 2003.

[88] A. J. Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[89] M. Smolik and V. Skala. Highly parallel algorithm for large data in–core and out–core triangulation in e2 and e3. *Procedia Computer Science*, 51:2613–2622, 2015.

[90] L. Solano-Quinde, B. Bode, and A. K. Somani. Techniques for the parallelization of unstructured grid applications on multi-gpu systems. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 140–147. ACM, 2012.

[91] L. Solano-Quinde, Z. J. Wang, B. Bode, and A. K. Somani. Unstructured grid applications on gpu: performance analysis and improvement. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, page 13. ACM, 2011.

[92] F. Tauheed, L. Biveinis, T. Heinis, F. Schurmann, H. Markram, and A. Ailamaki. Accelerating range queries for brain simulations. In *2012 IEEE 28th International Conference on Data Engineering*, pages 941–952. IEEE, 2012.

[93] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi. Passion: Optimized i/o for parallel applications. *Computer*, 29(6):70–78, 1996.

[94] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective i/o in romio. In *Frontiers of Massively Parallel Computation, 1999. Frontiers' 99. The Seventh Symposium on the*, pages 182–189. IEEE, 1999.

[95] D. F. Watson. Computing the n-dimensional delaunay tessellation with application to voronoi polytopes. *The computer journal*, 24(2):167–172, 1981.

[96] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[97] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 31–31. IEEE, 2006.

[98] R. V. Wilson, F. Stern, H. W. Coleman, and E. G. Paterson. Comprehensive approach to verification and validation of cfd simulations, part 2: Application for rans simulation of a cargo/container ship. *Journal of fluids engineering*, 123(4):803–810, 2001.

[99] K. Zhai, T. Banerjee, D. Zwick, J. Hackl, R. Koneru, and S. Ranka. Dynamic load balancing for a mesh-based scientific application. *Concurrency and Computation: Practice and Experience*, page e5626, 2020.

[100] K. Zhai, T. Banerjee, D. Zwick, J. Hackl, and S. Ranka. Dynamic load balancing for compressible multiphase turbulence. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 318–327, 2018.

[101] J. Zhang, H. Guo, F. Hong, X. Yuan, and T. Peterka. Dynamic load balancing based on constrained kd tree decomposition for parallel particle tracing. *IEEE transactions on visualization and computer graphics*, 24(1):954–963, 2017.

[102] Y. Zhou and L. Jiang. Hilbert curve based spatial data declustering method for parallel spatial database. In *Remote Sensing, Environment and Transportation Engineering (RSETE), 2012 2nd International Conference on*, pages 1–4. IEEE, 2012.

[103] X. Zhuang and S. L. Hsien-hsin. Reducing cache pollution via dynamic data prefetch filtering. *IEEE Transactions on Computers*, 56(1):18–31, 2007.

VITA

Mr. Cuong M. Nguyen worked for ten years in the Nha Trang University, Vietnam. Mr. Cuong received his Bachelor of Science in Computer Science in 1995 from National University in HCM city and his Master degree of Computer Science at Asian Institute of Technology (AIT) in 2002.

At the University of Mississippi, as a teaching assistant, Mr. Cuong have worked with Dr. Philip J. Rhodes on Efficient Generating and Processing of Large-scale Unstructured meshes. Meanwhile, he taught three courses as a graduate instructor, including Matlab.

His research interests include large-scale scientific data in distributed computing, High-Performance Computing, distributed and parallel programming, GPGPU computing, spatial dataset load balancing.