TIPP: Parallel Delaunay Triangulation for Large-Scale Datasets

Cuong Nguyen  
University of Mississippi  
University, MS  
cnguyen1@go.olemiss.edu

Philip J. Rhodes  
University of Mississippi  
University, MS  
rhodes@cs.olemiss.edu

ABSTRACT

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 paper, we introduce a novel algorithm – Triangulation of Independent Partitions in Parallel which divides the domain into many independent partitions that can be triangulated in parallel. In contrast to stitching methods, merging our partition triangulations into a single result is easily done, and satisfies the Delaunay criteria.

We use C/C++ and MPI (Message Passing Interface) to implement and evaluate our algorithm on a cluster. Experimental results show that our parallel implementation outperforms our serial implementation by roughly 27x for 1 billion triangles. Lastly, we believe we have generated the largest Delaunay mesh to-date, at 16 billion triangles.

CCS CONCEPTS
- Computing methodologies → Parallel computing methodologies; Distributed computing methodologies;

KEYWORDS
Delaunay Triangulation, Big Data, Distributed Computing, MPI

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

1 INTRODUCTION

Delaunay triangulation [18] 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, but we must 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 [1, 35], physics [33], and mechanical engineering [45]. However, triangulations are quite general, finding applications outside CFD, including chemistry [15], communications [28], and computer vision [17].

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 [27]. 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 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 [5, 10, 11, 14, 21, 24, 29–32]. 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 [13, 19] 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.

In this paper, 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 4) 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.

We make several significant contributions in this paper:  
§ We devise a novel algorithm for partitioning unstructured meshes and
distributing the partitions to nodes in a cluster. 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. Because of these improvements, our algorithm scales
to a billion triangles and beyond. § We prove that our algorithm
generates triangulations that satisfy the Delaunay criteria. § We
propose a method of improving performance by reducing the scope
of triangle search within partitions. Taken together, these contribu-
tions make possible the triangulation of extremely large datasets,
with billions of triangles.

The rest of the paper is organized as follows: sections 2 and 3
give important background and establish some key claims required
by the TIPP algorithm. Sections 4 and 5 describe our algorithm in
detail. Sections 6 and 7 describe our implementation and present
empirical results, while sections 8, 9 and 10 describe related work
and final thoughts.

2 BACKGROUND

The Bowyer–Watson algorithm [6, 44], also known as an incremen-
tal 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 1: Triangles T in cavity C = {v1, v2, ..., v5} (the shaded region) have circumcircles enclosing point p.](image1.png)

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. Lemma 1 establishes that only the set
of triangles T with circumcircles intersecting a partition S could be
deleted by triangulating points contained in S. Lemma 2 establishes
that if we add a new point p to the triangulation, where p ∈ S, 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

3 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 par-
allel. 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 [30–32, 43].

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,
where partitions are chosen to be independent, meaning their trian-
gulations will not influence each other. However, we allow triangles
along partition borders to be continually refined as the algorithm
progresses, meaning that they belong to the final Delaunay Trian-
gulation, and no stitching phase is required.

![Figure 2: Triangles T in Polygon {V1, V2, ..., V7} have circumcircles that intersect with rectangular partition S. The circum-
circle of triangle △V1V2V3 does not intersect square S, and is
a member of the set T′ described in lemma 1.](image2.png)
say two partitions are independent if there is no triangle $t$ with a circumcircle intersecting both partitions.

**Lemma 1.** Let $T$ be the group of triangles with circumcircles that intersect with a partition $S$, then any Delaunay triangle outside $T$ will not be deleted by the Delaunay triangulation of any new vertex inserted within partition $S$.

**Proof.** As shown in figure 2, let $T \subset DT$ be a set of triangles with circumcircles that intersect with a partition $S$, where $DT$ is the Delaunay Triangulation of the points inserted so far. Let $T' = DT - T$. If $S$ 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 group of triangles with circumcircles that intersect with a partition $S$. If a point $p \in S$ 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 $t'$ 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 1). However, $p \in S$, so $t'$ cannot be in $T'$, which contradicts the assumption. □

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. Lemma 3 establishes that once all points contained within a partition have been inserted into the triangulation, there is a set of 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 $S$, the interior set will fit exactly with the rest of the triangulation without modification.

After triangulation of a partition $S$, there is a set of boundary triangles belonging to $S$ that cannot yet be finalized, and must instead be rejoined with the unprocessed triangles 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 $S$, meaning they will fit perfectly 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 3, we define the interior of a partition $S$ as the set of triangles $I = \text{interior}(S)$ that intersect $S$, and also have circumcircles wholly contained in $S$, implying the triangles of $I$ are also wholly inside $S$. Let $R$ be the set of points that remain to be inserted into the triangulation. If all points $p \in R$ are outside of $S$, then the set $I$ remains unchanged after the points of $R$ are inserted.

**Proof.** From the discussion in section 2, 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 $S$, and $R$ contains only points outside $S$. 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 $S$ 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 $S$ contains $t$, since $I$ consists only of triangles with circumcircles wholly within $S$. This contradicts the assumption that $R$ contains only points outside of $S$.

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

**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 $S$ are finalized, they will still fit the interior of $S$.

**Proof.** We define the set $B = \text{boundary}(S)$ of a partition $S$ to be the set of triangles with circumcircles that intersect, but are not wholly contained by $S$. As shown in figure 3, the boundary set of $S$ is hollow, because it can be formed by removing the interior($S$) from the set of triangles with circumcircles that intersect with $S$. We say the boundary has an inside perimeter formed by the edges surrounding this hollow region.

We assume that the interior($S$) was finalized when no points $p \in S$ remained, but that the boundary set may have changed due to further triangulation. Consider an edge $e$ of a triangle $t \in \text{boundary}(S)$
that lies on the inside perimeter of \( \text{boundary}(S) \), as shown in figure 5. Edge \( e \) is also an edge of some triangle \( t' \in \text{interior}(S) \). If \( e \) is deleted by triangulating a point \( p \) via the process described in section 2, it must also be true that \( t' \) has a circumcircle extending beyond \( S \), since only points \( p \notin S \) remain. This violates the definition of \( t' \) as an interior triangle, so there can be no such edge \( e \). Since no edge of the inner perimeter of \( \text{boundary}(S) \) changes between the time when \( \text{interior}(S) \) is finalized and when \( \text{boundary}(S) \) is finalized, the two sets of triangles will fit together perfectly. □

**Lemma 5.** The outside perimeter of the set of boundary triangles \( B \) of partition \( S \) will not change during the triangulation of \( S \).

**Proof.** As before, we define the set \( B = \text{boundary}(S) \) of a partition \( S \) to be the set of triangles with circumcircles that intersect, but are not wholly contained by \( S \). We also define the exterior triangles of \( S \) as the set of triangles \( E = \text{exterior}(S) \) with circumcircles that do not intersect \( S \). 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 \( t \in \text{boundary}(S) \) that lies on the outside perimeter of \( \text{boundary}(S) \), as shown in figure 4. Edge \( e \) is also an edge of some triangle \( t' \in \text{exterior}(S) \).

If \( e \) (and therefore \( t' \)) is deleted by triangulating a point \( p \in S \) via the process described in section 2, it must also be true that \( t' \) has a circumcircle intersecting \( S \), 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 \( p \in S \). We conclude that no edge of the outside perimeter of \( S \) and no triangle \( t' \in \text{exterior}(S) \), will be deleted during the triangulation of \( S \).

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

We conclude that during the triangulation of \( S \), no edge of the outside perimeter is deleted, and \( E = \text{exterior}(S) \) is unchanged. Therefore, the outside perimeter of \( \text{boundary}(S) \) is not changed. □

4 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.

4.1 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.

**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 which do not intersect the partition.

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

Algorithm 1: Overview of TIPP

Input: set of partitions \( P_i \) in domain \( D \), Points \( \in D \)
Output: Delaunay Triangulation \( DT \) for Points \( \in D \)

1. Prework Compute partitioned points
   \( Points_p = points_{s_1}, \ldots, points_{s_n} : points_{s_i} \in P_i \)
2. Prework Compute initial points \( Points_{s1} \) (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.)
3. Let \( P = P_i \)
4. Step 1: generate the initial mesh (IM) using initial points \( Points_{s1} \) and two super-triangles constructed from the four corners of the domain and additional points along the edge of the domain (see figure 8)
5. Let \( T \) be all triangles in IM.
6. while \( P \neq \emptyset \) do
7.   \( Step \) 2: generate the set of independent partitions \( IP \) based on the intersection between partitions and the circumcircles of triangles in \( T \).
8.   \( Step \) 3: for each partition \( p \in IP \) collect triangles in \( T \) that belong to \( P \), using the circumcircle intersections computed in step 2.
9.   \( Step \) 4: Triangulate each partition \( p \in IP \) in parallel on the worker nodes, and update \( T \) accordingly. If the number of available processes \( np \) is less than \( |IP| \), we schedule partitions in shifts of size \( np \).
10. Step 5: write finalized triangles to external storage and update \( P \)
11.   \( let \) \( T_f \subset T \) be the finalized triangles.
12.   \( T \leftarrow T - T_f \).
13.   \( Finalize(T_f) \).
14.\( P \leftarrow P - IP \).
15. end
16. \( Step \) 6: Remove any triangles that use artificial vertices added in step 1.

Algorithm 2: Determining independent partitions.

Input: Mesh \( M = \{ ID, Points \} \), 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. \( \forall p \in P, CP(p) \leftarrow \emptyset \)
3. for each triangle \( t \in M \) do
4.   Determine a set of partitions
5.   \( P_i = \{ p_1, \ldots, p_n \} : circumcircle(t) \cap p_i \neq \emptyset \).
6.   for each partition \( p_i \in P_i \) do
7.     if \( p_i \neq p \) then
8.       \( CP(p_i) \leftarrow CP(p_i) \cup p_j \).
9.     end
10. end
11. end
12. end
13. \( p_i \rightarrow CP(p_i) \) remove duplicates
14. Let \( SP \) be a list of \( p_i \in P \), sorted according to \( |CP(p_i)| \), in increasing order.
15. \( SP \leftarrow Sort(P) \) 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. \( C \leftarrow \emptyset \)
19. \( IP \leftarrow \emptyset \)
20. while \( C \cup IP \neq P \) do
21.   let \( p = head(SP) \) (the partition in \( SP \) with the fewest conflicts)
22.   if \( p \notin C \) then
23.     \( IP \leftarrow IP \cup p \)
24.     \( C \leftarrow C \cup CP(p) \)
25. end
26. end

4.2 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 6.

There are two prework tasks. First, points in the domain are distributed to the partitions that geometrically contain them (line 1 in Algorithm 1). Second, we construct an initial mesh IM, using a small number of points (line 2 in Algorithm 1). We begin the mesh by adding 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.) We then randomly select some initial points from each partition for insertion into the initial mesh. Selecting points in this fashion reduces the occurrence of silver triangles, as described in section 5. 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 DT.

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 IP such that triangulation of a partition \( P_j \in IP \) will not affect any other partition \( P_j \in IP \). Beginning with the initial mesh (shown in figure 6(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 to both partitions. Algorithm 2 shows in detail how we generate 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 \( P_1 \) was chosen as a member of IP, then those edge and vertex neighbor partitions that conflict with \( P_1 \) must not be in IP. This explains the checkerboard appearance of some of the diagrams in figure 6. However, TIPP does not depend on this
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.

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.

Four groups of triangles have been triangulated in parallel. The circumcircles of gray triangles are contained wholly within their partition and do not affect other partitions. These gray triangles are finished, and ready to be finalized.

The grey triangles from figure 6(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.

We must still handle the inactive triangles from figure 6(a). These triangles did not have circumcircles intersecting our active partitions.

The boundary triangles from figure 6(d) are joined with the inactive triangles from figure 6(e). This new set of triangles is ready for the next stage, beginning with the selection of a new set of active partitions.

Figure 6: Parallel Delaunay Triangulation — example of Triangulation of Independent Partitions in Parallel with 1000 points.

observation, and rigorously checks for triangles that actually cause a conflict between partitions.

For example, in figure 6(b), partitions 4, 5, 6, 8, 10, 12, 13, and 14 conflict with partition 9, meaning that if partition 9 is chosen as a member of IP, then the aforementioned partitions will not be placed in IP.

After determining IP, partitions that are not in IP cannot be active in the upcoming processing stage, and will have to wait for a future round. For example, in figure 6(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 inside the partition. Recall from section 3 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 6(c) presents four active partitions after triangulation. More triangles (shown in 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 6(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 6(e) shows the global mesh before the boundary sets from partitions 0, 3, 9, and 15 are added. Notice that the boundary sets fit cleanly in the cavities surrounding the finalized partitions, producing a new global mesh, shown in figure 6(f).

We now repeat the process, beginning with identifying active partitions using the triangles from the global mesh, which has just been updated. 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 in partitions 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 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 can also add missing edges of the convex hull to repair the triangulation. This is a topic for future work.

5 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.

5.1 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 number of triangles 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 7(a) 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 6(a) is roughly 640; hence, the number of active partitions is much improved (e.g. four active partitions in figure 6(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.

Figure 7: A worst case and ideal case of Parallel Delaunay Triangulation algorithm.

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 8(a). Unfortunately, this introduces many slivers with edges $AB$, $BC$, $CD$, or $DA$. These triangles directly intersect many partitions. Worse still, such triangles have very large circumcircles, which introduces more dependencies between partitions, and also increases search costs. Finally, 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 8(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 $AB$, $BC$, $CD$, and $DA$.

Figure 8: Super triangles introduce sliver triangles.

5.2 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 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 7.

Figure 9: 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.

We further enhance performance within each partition by using the sweep line algorithm [20] to process vertices in order sorted by \( x \) coordinate. This allows us to determine finished triangles, delete them from the triangle list, and write them to external storage. In figure 9, 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.

6 IMPLEMENTATION

We implemented TIPP using C/C++ and MPI (Message Passing Interface)[22]. 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 [37], OpenMP [16], and pthreads [34] do not easily extend over multiple nodes.

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

6.1 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 circumsphere. Second, given a partition, we need a list of triangles with circumspheres that intersect the partition.

We could choose either an array-based implementation or a dynamic data structure. Arrays are convenient for GPU computing [36], a possible future extension of TIPP. Some researchers [8, 38, 39, 41] were able to use arrays for elements (triangles or tetrahedra) because the maximum number of elements could be calculated in advance [27]. 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 [23], 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 5.2, the benefits of a tree implementation are uncertain.

6.2 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[42].

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.

7 EXPERIMENTAL RESULTS

We tested TIPP on a cluster running on the Chameleon Cloud Testbed [9]. 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 [2, 3]. 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 7.2 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 10: 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.

Figure 11: Speed up of Delaunay triangulation with different domain sizes (1 Million, 10M, 100M, and 1000M triangles). The $X$ axis is logarithmic.

Figure 12: TIPP Execution times for increasing numbers of nodes.

7.1 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 10 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 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 10 shows TIPP performance for different partitioning granularities and dataset sizes. The execution time due to the master node is shown as 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 11) between Parallel TIPP running on ten nodes with 16 cores apiece and a separate serial implementation. The best performance of parallel TIPP from figure 10 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 12 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 10. 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.

7.2 Processing a Large Dataset

Qhull [2, 3] 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 × 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.

8 RELATED RESEARCH

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 [14], Hardwick [24], Chen [10, 11], Blelloch [5], Futterling [21], and Lin [29] 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 section 1. 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. [30–32] 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. [43] 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. [40] describes an incremental insertion algorithm which does not assign a region to each processor; instead, they use multiple threads via OpenMP [16] 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. [4] 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. Chrischoides [13, 19] attempted to reduce the communication latency between processors by load-balancing and minimizing the border between sub-meshes as they are generated.

Other parallel DT methods have also been reported. Chen [12] presents a method for localizing DT computation and describes an implementation that takes advantage of this work. While the serial implementation is only half as fast as other methods, the parallel implementation gives excellent speedup compared to the serial implementation. 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 [25] reports a DT method which can process a very large number of triangles (up to many billions) with relatively 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.

Similar to Isenburg’s method, Kohout [26] also divides the set of points in domain into many smaller regions using a space-filling Hilbert curve [7] for better locality. The initial mesh is recursively split into partitions using a paraboloid to project points on the vertical plane. Partitions are then sequentially triangulated and stored to files. The final step is to merge all partitions into the result triangulation by updating the connectivity between triangles of adjacent partitions. This step appears to be a stitching problem, but the authors did not describe their approach in detail.

In contrast, our method does not need to merge after triangulation for each partition in the domain and guarantees all triangles are Delaunay throughout the whole domain. Since the active partitions can be assigned different processors in a distributed environment, the performance improves significantly.
9 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.

9.1 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 avoiding 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 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 current 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 prevalence of large triangles with circumcircles spanning several partitions.

Another approach is to raise master performance by employing a parallel implementation for the partitioning process. Additionally, we could reduce network and I/O contention by replacing NFS with a more directly managed MPI mechanism for distributing and collecting data.

Table 1: Number of active partitions based on the total number of partitions

<table>
<thead>
<tr>
<th>Number of partitions active during first phase</th>
<th>4</th>
<th>16</th>
<th>61</th>
<th>251</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of partitions</td>
<td>16</td>
<td>64</td>
<td>256</td>
<td>1024</td>
<td>4096</td>
</tr>
</tbody>
</table>

9.2 Load balancing
We used qhull utilities [2, 3] 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.

10 CONCLUSIONS AND FUTURE WORK
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. An important avenue of future work is the parallelization of master node tasks, which will boost performance, efficiency, and the size of datasets feasible for scientific researchers.

11 ACKNOWLEDGMENTS
Some results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation.

REFERENCES


