Parallel d-D Delaunay Triangulations in Shared and Distributed Memory

Daniel Funke [*] Peter Sanders [*]

Abstract

Computing the Delaunay triangulation (DT) of a given point set in $\mathbb{R}^D$ is one of the fundamental operations in computational geometry. In this paper we present a novel divide-and-conquer (D&C) algorithm that lends itself equally well to shared and distributed memory parallelism. While previous D&C algorithms generally suffer from a complex – often sequential – merge or divide step, we reduce the merging of two partial triangulations to re-triangulating a small subset of their vertices using the same parallel algorithm and combining the three triangulations via parallel hash table lookups. In experiments we achieve a reasonable speedup on shared memory machines and compare favorably to CGAL's three-dimensional parallel DT implementation on some inputs. In the distributed memory setting we show that our approach scales to 2048 processing elements, which allows us to compute 3-D DTs for inputs with billions of points.

1 Introduction

The Delaunay triangulation (DT) of a given point set in $\mathbb{R}^D$ has numerous applications in computer graphics, data visualization, terrain modeling, pattern recognition and finite element methods [15]. Computing the DT is thus one of the fundamental operations in geometric computing. Therefore, many algorithms to efficiently compute the DT have been proposed [23] and well implemented codes exist [12, 20]. With ever increasing input sizes, research interest has shifted from sequential algorithms towards parallel ones [2, 4, 6, 9, 11, 15], with shared memory parallelism for algorithms in two dimensions receiving most of the attention. Distributed memory algorithms however – as studied by Cignoni et al. [9] and Lee et al. [17] – are required to cope with triangulations exceeding the memory limitations of one machine.

In this paper we present a novel divide-and-conquer (D&C) DT algorithm for arbitrary dimension that lends itself equally well to shared and distributed memory parallelism and thus hybrid parallelization. Previous D&C DT algorithms suffer from a complex – often sequential – divide or merge step [9, 17]. We reduce the merging of two partial triangulations to re-triangulating a small subset of their vertices using the same parallel algorithm and combining the three triangulations via hash table lookups. All steps required for the merging – identification of relevant vertices, triangulation and combining the partial DTs – are performed in parallel. Only minor modifications are required to adapt our algorithm from a shared memory machine to a message-based distributed memory cluster.

The rest of this paper is structured as follows: we present a problem definition and a survey of related work in Sections 2 and 3. Subsequently, our proposed shared memory algorithm is described in Section 4. The modifications necessary to adapt our algorithm to a distributed memory setting are presented in Section 5. Section 6 highlights some technical details of our implementation. We evaluate our algorithms in Section 7 and close the paper with conclusions and an outlook to future work in Section 8.

2 Definitions

D-simplices are a generalization of triangles ($D = 2$) to D-dimensional space. A D-simplex s is a D-dimensional polytope, i. e., the convex hull of $D + 1$ points. The convex hull of a subset of size $m + 1$ of these $D + 1$ points is called an m-face of s. Specifically, the 0-faces are the vertices of s and the $(D - 1)$-faces are its facets. Given a D-dimensional point set $\mathbf{P} = \{p_1, p_2, \ldots, p_n\}$ with $p_i \in \mathbb{R}^D$ for all $i \in \{1, \ldots, n\}$, a triangulation $T(\mathbf{P})$ is a subdivision of the convex hull of $\mathbf{P}$ into D-simplices, such that the set of the vertices of $T(\mathbf{P})$ coincides with $\mathbf{P}$ and any two simplices of T intersect in a common $D - 1$ facet or not at all. The union of all simplices in $T(\mathbf{P})$ is the convex hull of point set $\mathbf{P}$. A Delaunay triangulation $DT(\mathbf{P})$ is a triangulation of $\mathbf{P}$ such that no point of $\mathbf{P}$ is inside the circumhypersphere of any simplex in $DT(\mathbf{P})$. If the points of $\mathbf{P}$ are in *general position*, i. e., no $D + 2$ points lie on a common D-hypersphere, $DT(\mathbf{P})$ is unique [10].

A note on notation: we refer the set of vertices of simplex s by vertices(s); the individual i-th vertex is denoted vertices$_i(s)$. We employ a similar notation for

[*]Karlsruhe Institute of Technology, Institute for Theoretical Informatics, Algorithmics II. Email: {funke, sanders}@kit.edu

the set of neighboring simplices of s – neighbors(s) – and an individual neighbor i – neighbors$_i(s)$.

3 Related Work

A survey of parallel DT algorithms in two and three dimensions for shared memory is given by Kohout et al. [15]. The proposed algorithms are either based on parallel incremental insertion or a D&C approach. Parallel incremental insertion algorithms are generally bootstrapped with a sequentially obtained initial triangulation of a subset of the input points. Subsequently, the rest of the points can be inserted in parallel by identifying the surrounding simplex for each point, removing it and re-triangulating the resulting cavity with the inserted point and the facets of the surrounding simplices [2, 7, 15]. The Delaunay property of the re-triangulated region is ensured by performing local flips [14, 15]. To avoid simultaneous access to the same simplex during re-triangulation, locks need to be employed. Various locking strategies are studied in [2, 15]. The algorithm of Batista et al. is the basis for the parallel DT algorithm found in the CGAL library [12]. This potential for contention renders parallel incremental insertion very sensitive to the input point distribution and limits the attainable speedup. Moreover, it requires fine-grained communication, which is prohibitive in the distributed memory setting. To avoid communication, Lo [18] partitions the input points into zones. Each zone is further subdivided into cells. Neighboring zones exchange the points of their adjacent cells, to be able to construct the triangulation without synchronization. They report a speedup of ≈ 10 for 12 cores with shared memory for uniformly distributed points, however no comparison with other implementations are given.

A parallel D&C algorithm for DTs was first proposed by Aggarwal et al. [1]. The input points are partitioned into blocks, which are triangulated in parallel. These partial triangulations are stitched together in an expensive merge step, which can only be performed by one processing element, thus limiting speedup. As non-Delaunay simplices might be introduced to the triangulation during stitching, corrective steps are required to restore the Delaunay property. In the worst case, the necessary corrections can spread throughout the entire triangulation [15]. A different approach is pursued by Cignoni et al. for three- and arbitrary-dimensional DTs [8, 9, respectively]. They divide the input by cutting (hyper)planes and firstly construct the simplices of the triangulation crossing those planes. The algorithm continues to build the triangulation in the divided regions in parallel, no further merging is necessary. However, their division step is expensive and sequential and thus limits scalability. Chen [5] improves on that by calculating the *affected zone* comprising the set of simplices that are indeterminate, i. e., a point outside the convex hull of the sub-triangulation may still influence them. The merging of two partial triangulations can then be reduced to merging the affected zones. Using a distributed memory setting, they report speedups of up to 4.5 for clusters of 8 processing elements (PEs) and uniformly distributed points. Further studies on distributed memory machines are presented by Lee et al. [17]. They partition the input according to paths of Delaunay edges obtained from a lower convex hull projection [4]. The individual partitions can then be triangulated without further merging. They report a speedup of ≈ 12 for a machine with 32 PEs and a uniform distribution of input points. To the best of our knowledge no algorithm has been shown to scale well to clusters with hundreds of PEs.

An entirely different approach is proposed by Chen and Gotsman [6]. They *localize* the computation of the DT by computing the Delaunay neighbors for each point individually. This affords for almost linear speedup. It remains to be seen whether their approach generalizes to three and higher dimensions.

Fuetterling et al. [11] present a novel data structure for D&C-based DT algorithms, the *linear floating point quad-tree* (LFQT) based on the Morton codes of the input point coordinates. The geometrical structure of the quad tree allows for efficient subdivision of the input during the recursive descent and its numerical structure minimizes the need for exact arithmetic. Although their data structure should generalize to arbitrary dimension, they only report – very favorable – results for single threaded as well as multi threaded performance for computing the DT in two dimensions.

Table 1 provides an overview of the discussed literature and compares some properties of the proposed algorithms.

4 Shared Memory Algorithm

In this paper we present a novel D&C Delaunay triangulation algorithm with a fully parallelizable merge step. The merging of two partial triangulation relies on re-triangulating a small subset of *border* points of both triangulations with the same parallel DT algorithm. Border points are vertices of simplices that might violate the Delaunay property for some point of the other partition and hence need to be re-triangulated for a valid DT of the combined point set. All steps necessary to identify these points and to combine the two partial triangulations with their border triangulation are fully parallelized. Algorithm 1 outlines our algorithm, which is described in the following. Refer to Figure 1 for a two-dimensional example.

Given the set of input points $\mathbf{P} = \{p_1, \ldots, p_n\}$ and a recursion level r, if the number of points is below a certain

Algorithm	3-D	d-D	shared mem.	dist. mem.	speedup
Kohout et al. [15]	✓	✗	✓	✗	3.7 (4 PEs)
Batista et al. [2]	✓	✗	✓	✗	7 (8 PEs)
Lo [18]	✓	✗	✓	✗	10 (12 PEs)
Aggarwal et al. [1]	✗	✗	✓	✗	theory
Cignoni et al. [8, 9]	✓	✓	✗	✓	3.4 / (16 PEs)
Chen [5]	✓	✗	✗	✓	4.5 (8 PEs)
Lee et al. [17]	✗	✗	✗	✓	12 (32 PEs)
Fuetterling et al. [11]	✗	✗	✓	✗	13 (16 PEs)
Chen and Gotsman [6]	✗	✗	✓	✗	7.5 (8 PEs)
this paper	✓	✓	✓	✓	260 (2048 PEs)

Table 1: Properties of DT algorithms proposed in this paper and related work. The speedup given is the maximum reported by the authors for uniformly distributed points.

threshold or a recursion depth of $\log P$ for P processors has been reached, an efficient sequential DT algorithm is used to solve the base case. Otherwise, our recursive divide-and-conquer algorithm is employed. Firstly, the splitting dimension k is determined following one of various schemes: a) constant, predetermined splitting dimension; b) cyclic choice of the splitting dimension – similar to k-D trees [3]; or c) dimension with largest extend. The input points are then partitioned across the selected dimension k according to the median point. Both partitions are recursively triangulated in parallel, yielding triangulations T_1 and T_2.

Subsequently, the border simplices $\mathbf{B}$ of both triangulations are determined, starting from the convex hull of T_1 and T_2. To identify the convex hull of a triangulation efficiently, Shewchuk [20] introduces a *vertex at infinity* that forms an *infinite* simplex with every facet of the convex hull. We extend this concept by introducing a vertex at infinity for each corner of the bounding box of point set $\mathbf{P}$, thus affording meaningful intersection tests of infinite simplices with a partition's bounding box. The search for border simplices employs a parallel work queue initialized with the infinite simplices of T_1 and T_2. A simplex s belongs to the border of triangulation T_i if its circumsphere intersects with the bounding box of the other triangulation T_j, $i \neq j$, i. e., s might still be influenced by a point in partition j. In that case, s is added to $\mathbf{B}$ and all its neighbors are enqueued for processing. A lock-free marking scheme is used to ensure every simplex is processed at most once. After completion of the algorithm, all simplices of T_1 and T_2 *not* in $\mathbf{B}$ are completely inside their respective partition and are hence not susceptible to change due to points of the other partition. The same criterion is used by Isenburg et al. [13] and later Wu et al. [25] to define *finalized* triangles of a partition in a streaming computation setting, by Chen [5] to determine the *affected zone* and by Lo [18] to determine cells of a zone to be exchanged with neighboring zones. The vertices of all border simplices are collected and recursively triangulated using the D&C algorithm, yielding border triangulation T_B.

The combined triangulation T is composed of simplices from the partial triangulations as well as the border triangulation T_B. Non-border simplices of T_1 and T_2 can be immediately added to T, as no point of the other partition can lie within their circumsphere. The border simplices $\mathbf{B}$ are discarded, since they potentially violate the Delaunay property for some point of the opposite partition. For a simplex $s_B \in T_B$ to be added to T, one of two conditions needs to be fulfilled: a) s_B consists of vertices from both partitions, or b) s_B is contained within one partition but replaces a previously found border simplex. The first condition treats simplices crossing the border of T_1 and T_2, which could not have been found before. As s_B fulfills the Delaunay property with respect to the border point set, it also fulfills it with respect to both partition point sets [5]. The second condition re-adds simplices of the border that have been confirmed to not only fulfill the Delaunay property with respect to their own partition but also with respect to the border point set and hence the other partition [18]. If vertices(s_B) is fully contained in one partition but no simplex with equal vertices has been previously found in the respective partial triangulation, s_B is discarded, as it must violate the Delaunay property for a point of that partition not belonging to the border point set, following the uniqueness of the DT for a point set in general position [10].

Simplices with equal vertices can be efficiently found by using a hash table of the discarded border simplices. The lookup key is a *simplex hash* $h_s(s)$ – the exclusive

Algorithm 1 Delaunay($\mathbf{P}, r$): shared memory parallel D&C algorithm

```
Input:   points $\mathbf{P} = \{p_1, \ldots, p_n\}$ with $p_i \in \mathbb{R}^D$, recursion level $r$
Output:  Delaunay triangulation $T$
 1: if $n < N \vee r = \log P$ then                                          ▷ for $P$ processors
 2:     return $sequentialDelaunay(\mathbf{P})$                               ▷ base case
 3: $k \leftarrow$ splittingDimension($\mathbf{P}$)
 4: $(\mathbf{P}_1 \ \ \mathbf{P}_2) = (p_1 \ \cdots \ p_s \ | \ p_{s+1} \ \cdots \ p_n) \leftarrow$ divide($\mathbf{P}, k$)   ▷ partition points in dim. $k$
 5: $\mathbf{T} = (T_1 \ \ T_2) \leftarrow (\text{Delaunay}(\mathbf{P}_1, r+1) \ \ \text{Delaunay}(\mathbf{P}_2, r+1))$   ▷ in parallel
Border triangulation:
 6: $\mathbf{B} \leftarrow \varnothing$;   $\mathbf{Q} \leftarrow$ convexHull($T_1$) $\cup$ convexHull($T_2$)   ▷ initialize set of border simplices and work queue
 7: parfor $s_{i,x} \in \mathbf{Q}$ do                                       ▷ simplex originating from triangulation $T_i$
 8:     mark($s_{i,x}$)                                                       ▷ only process each simplex once
 9:     if circumsphere($s_{i,x}$) $\cap$ boundingBox($T_j$) $\neq \varnothing$, with $i \neq j$ then
10:         $\mathbf{B} \cup= \{s_{i,x}\}$              ▷ circumsphere intersects other partition $\Rightarrow s_{i,x}$ is a borders simplex
11:         for $s_{i,y} \in$ neighbors($s_{i,x}$) $\wedge \neg$ marked($s_{i,y}$) do   ▷ process all neighbors
12:             $\mathbf{Q} \cup= s_{i,y}$;   mark($s_{i,y}$)
13: $T_B \leftarrow$ Delaunay(vertices($\mathbf{B}$), $r+1$)                  ▷ triangulate points of border simplices
Merging:
14: $T \leftarrow (T_1 \cup T_2) \setminus \mathbf{B}$;   $\mathbf{Q} \leftarrow \varnothing$   ▷ merge partial triangulations stripped from border
15: parfor $s_b \in T_B$ do                                                 ▷ merge simplices from border triangulation
16:     if vertices($s_b$) $\not\subset \mathbf{P}_1 \wedge$ vertices($s_b$) $\not\subset \mathbf{P}_2$ then
17:         $T \cup= \{s_b\}$;   $Q \cup= \{s_b\}$                             ▷ $s_b$ spans both partitions
18:     else
19:         if $\exists s \in \mathbf{B}$ : vertices($s$) = vertices($s_b$) then
20:             $T \cup= \{s_b\}$;   $Q \cup= \{s_b\}$                         ▷ $s_b$ replaces border simplex
Neighborhood update:
21: parfor $s_x \in Q$ do                                                    ▷ update neighbors of inserted simplices
22:     for $d \in \{1, \ldots, D+1\}$ do
23:         if neighbors$_d$($s_x$) $\notin T$ then                           ▷ neighbor not in triangulation anymore
24:             $C \leftarrow \{s_c : f_d(s_x) = f_d(s_c)\}$                   ▷ candidates with same facet hash
25:             for $s_c \in C$ do
26:                 if $|$vertices($s_x$) $\cap$ vertices($s_c$)$| = D$ then
27:                     neighbors$_d$($s_x$) $\leftarrow s_c$;   $Q \cup= s_c$   ▷ $s_c$ is neighbor of $s_x$
28: return $T$
```

or of a hash value $h_v(\cdot)$ of each vertex of s,

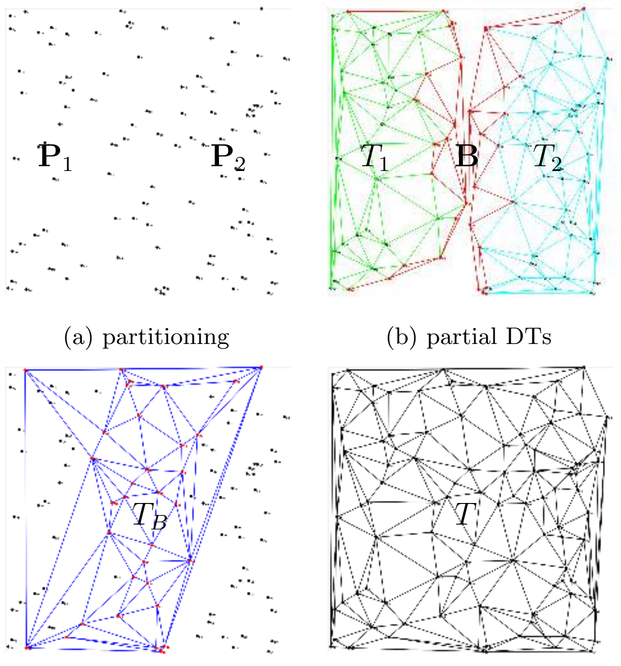
$$h_s(s) := \bigoplus_{i<D+1} h_v(\text{vertices}_i(s)).$$

For a suitable $h_v(\cdot)$, $h_s(\cdot)$ is efficiently computable, commutative and contributions of individual vertices can be easily extracted. The latter two properties are important for the subsequent neighborhood update. Refer to Section 6 for details about the choice of $h_v(\cdot)$.

Finally, the neighborhood-relations of the newly inserted simplices need to be established. For each neighbor $d \in \{1, \ldots, D+1\}$ of a simplex s it is determined whether the currently designated neighbor is valid – i. e., is not some placeholder value and still part of triangulation T – or needs updating. In the latter case, the simplex $s_n \in T$ is determined that shares the facet opposite of vertices$_d(s)$ with s. Simplex s_n is set as the new neighbor and enqueued for updating itself. To efficiently find candidates for neighboring simplices of a given simplex s we employ a *facet hash* – denoted $f_i(s)$ for the facet opposite the i-th vertex. The facet hash must be independent from the order of vertices in s and should be efficiently computable from $h_s(s)$. Thus, we exploit the commutativity of $h_s(s)$ and the involutionarity of exclusive or and let

$$f_i(s) := h_s(s) \oplus h_v(\text{vertices}_i(s)).$$

As only simplices of T_1 and T_2 neighboring the border simplices $\mathbf{B}$ as well as simplices added to T from T_B need to update their neighborhood, we can efficiently maintain



(a) partitioning (b) partial DTs

(c) border DT (d) final DT

Figure 1: Example of a two-dimensional triangulation. Infinite simplices are not drawn for clarity.

a facet lookup table during border detection and merging. The convex hull of T is composed of the convex hull of T_1 and T_2 without the simplices belonging to the border plus the simplices of the convex hull of T_B that have been added to T. The necessary data structures can also be efficiently maintained during merging.

5 Distributed Memory Algorithm

The previously presented divide-and-conquer algorithm can be applied to a distributed memory model with explicit message-based communication. The general idea of the approach remains unchanged, only slight modifications are required to account for the incomplete information each processing element (PE) has about the input points and hence the global triangulation.

Each of the P PEs holds a portion of the input points, $\mathbf{P} = \{\mathbf{P}_1, \ldots, \mathbf{P}_P\}$ with $\mathbf{P}_i = \{p_{i,1}, \ldots p_{i,n_i}\}$. In the following, we assume that a PE i belongs to exactly one partition in each partitioning step, i. e., all its points $\mathbf{P}_i$ are either on one side of the splitting plane or the other. This holds naturally for e. g. geo-spatial data read from pre-tiled files. Data not adhering to this assumption require *one* additional all-to-all communication in the first recursive descent to move the input points to their respective PE.

Algorithm 2 presents the modified D&C algorithm from the viewpoint of PE i. Variables subscripted with $\square_i$ denote variables local to PE i. A partition of nodes is represented by set $\mathbf{C}$, e. g. at recursion level zero $\mathbf{C} = \{1, \ldots, P\}$. At a given recursion level, if the number of input points in the current partition $\mathbf{C}$ lies below a certain threshold or there is only one PE left in the partition the base case Algorithm 3 is invoked to compute the Delaunay triangulation of the points of $\mathbf{C}$. In the recursive case, the local minimum, maximum and median of the input points are computed. These statistics are gathered by all PEs of the partition to compute the global values. The splitting dimension is determined according to the global statistics, following the same schemes as for the shared memory implementation. The partition $\mathbf{C}$ is reduced to $\mathbf{C}'$, containing only PEs on the same side of the splitting plane as PE i. The recursive call with partition $\mathbf{C}'$ yields PE i's local view T on the triangulation of the points in $\bigcup_{j\in\mathbf{C}'} \mathbf{P}_j$ – denoted $DT(\mathbf{C}')$. It holds that $T = \{s \in DT(\mathbf{C}') : |\,\text{vertices}(s) \cap \mathbf{P}_i| \geq 1\}$, i. e., PE i stores every simplex of $DT(\mathbf{C}')$ that contains at least one vertex from the input points of PE i.

Subsequently, the border simplices of the local triangulation T are determined. The algorithm follows the same principle as described in the previous section. A simplex is added to the local border simplex set $\mathbf{B}_i$ if its circumsphere intersects with the bounding box of partition $\mathbf{C} \setminus \mathbf{C}'$, which can be computed with no additional communication. As PE i only has a local view T on $DT(\mathbf{C}')$ and the border detection algorithm starts its search from the convex hull of $DT(\mathbf{C}')$ there might be a simplex $\hat{s} \in T$ which belongs to the border of $DT(\mathbf{C}')$ but is only reachable from the convex hull of $DT(\mathbf{C}')$ via a simplex $\hat{s}'$ of the convex hull only stored at PE j. That is the case if vertices$(\hat{s})$ contains only one vertex from $\mathbf{P}_i$ and vertices$(\hat{s}')$ is fully contained in $\mathbf{P}_j$ and hence $\hat{s}' \notin T$. As $\hat{s}$ and $\hat{s}'$ are neighbors, at least one vertex of $\hat{s}$ is in $\mathbf{P}_j$ and therefore $\hat{s} \in T$ at PE j. Thus, $\hat{s}$ will be identified as border simplex by PE j. To ensure every PE is aware of all of its border simplices, a sparse all-to-all communication within partition $\mathbf{C}'$ is required, yielding the updated set $\mathbf{B}_i = (\cup_{j\in\mathbf{C}'} \mathbf{B}_j) \cap T$.

Only PEs with nonempty set $\mathbf{B}_i$ need to participate in the border triangulation. The border triangulation follows the same algorithm as the main triangulation with the reduced PE set $\mathbf{C}_B$. The merging of T and T_B is extended by the additional condition that a simplex $s_B \in T_B$ is only considered for addition to T if at least one of its vertices lies in $\mathbf{P}_i$. The further conditions are the same as in the shared memory case.

The determination of the neighborhood relations of the newly inserted simplices is also identical to the shared memory case. However, as each PE only possesses a partial view on the triangulation $DT(\mathbf{C})$, not all

Algorithm 2 Delaunay($\mathbf{P}_i, \mathbf{C}$): distributed memory parallel D&C algorithm

```
Input:   point subset $\mathbf{P}_i = \{p_{i,1}, \dots, p_{i,n_i}\}$, PEs of partition $\mathbf{C}$
Output: local view $T$ of Delaunay triangulation $DT(\bigcup_{j\in\mathbf{C}} \mathbf{P}_j)$
 1: if $\Sigma_{j\in\mathbf{C}} n_j < N \vee |\mathbf{C}| = 1$ then                          ▷ base case
 2:     return DelaunayBase($\mathbf{P}_i, \mathbf{C}$)
 3: $S_i \leftarrow$ localVertexStatistics($\mathbf{P}_i$)                 ▷ local min, max and median
 4: $S_{\text{all}} \leftarrow$ allReduce($S_i, \mathbf{C}$)                       ▷ global min, max and median
 5: $k \leftarrow$ splittingDimension($S_{\text{all}}$)
 6: $p_i = \text{median}(S_{i,k}) \geq \text{median}(S_{\text{all},k})$          ▷ PE's side of splitting plane in dim. $k$
 7: $\mathbf{C}' \leftarrow \{j : p_j = p_i \quad \forall j \in \mathbf{C}\}$          ▷ set of all PEs on same side of splitting plane
 8: $T \leftarrow$ Delaunay($\mathbf{P}_i, \mathbf{C}'$)                              ▷ triangulate own partition
 9: $\mathbf{B}_i \leftarrow$ borderSimplices($T, \mathbf{C}, \mathbf{C}'$)                 ▷ simplices across splitting plane
10: $\mathbf{B}_i \leftarrow$ sparseAllToAll($\mathbf{B}_i, \mathbf{C}'$)          ▷ receive border simplices from neighboring PEs
11: $\mathbf{C}_B \leftarrow \{j : \mathbf{B}_j \neq \varnothing \quad \forall j \in \mathbf{C}\}$                ▷ PEs with non-empty border
12: if $i \in \mathbf{C}_B$ then
13:     $T_B \leftarrow$ Delaunay(vertices($\mathbf{B}_i$), $\mathbf{C}_B$)               ▷ triangulate border vertices
14:     $T \leftarrow$ merge($T, T_B, \mathbf{B}_i, \mathbf{P}_i, \mathbf{C}'$)              ▷ slightly modified from Algorithm 1
15:     $U_i \leftarrow$ updateNeighbors($T, Q$)                        ▷ $U_i$ set of performed updates
16:     $U_{\text{all}} \leftarrow$ sparseAllToAll($U_i, \mathbf{C}_B$)                ▷ exchange neighbor updates
17:     parfor $\begin{pmatrix} s_x & k & n \end{pmatrix} \in U_{\text{all}}$ do     ▷ simplex $s_x$, neighbor no. $k$, neighbor simplex $n$
18:         $\text{neighbors}_k(s_x) \leftarrow n$
19: return $T$
```

neighbors of simplex $s \in T$ can be determined by PE i alone. Particularly, if vertices(s) contains only one vertex from $\mathbf{P}_i$, at least one of the neighbors of s will not be stored at PE i. Therefore, each PE keeps track of the updates it performs. In a sparse all-to-all communication, information about the updates to a simplex s is send to every PE that contains s in its local triangulation.

Distributed Base Case: Algorithm 3 details the treatment of the base case in the distributed setting. If there is only PE i left in partition $\mathbf{C}$ the points $\mathbf{P}_i$ are triangulated on PE i using a sequential or shared memory parallel algorithm. By setting the base case threshold N in Algorithm 2 greater than $\max_{j\in\{1,\dots,P\}} |\mathbf{P}_j|$, $|\mathbf{C}|$ will always be one in the first recursive descent and each PE will triangulate its own input points locally. Only in the recursive calls of delaunay($\dots$) for border triangulations can $\mathbf{C}$ contain more than one PE. In that case, the PE in $\mathbf{C}$ with the lowest index is chosen as master and receives the input points of all other PEs in $\mathbf{C}$. The master triangulates the gathered points and broadcasts the simplices among the PEs of the partition. The PEs then discard all simplices with no vertex in their respective input point sets.

6 Implementation Details

This section highlights some aspects of our implementation of the previously proposed algorithms.[1] While our algorithms are implemented for arbitrary dimension, we have only included base case algorithms for two- and three-dimensional DTs at the moment.

The input points $\mathbf{P} = \{p_1, \dots, p_n\}$ with $p_i \in \mathbb{R}^D$ are stored in an array with their D coordinates. A partition of points consists of a list of indices into this array. To ensure globally unique point indices in the distributed setting, each PE i stores the global offset $o_{v,i}$ of its point array; $o_{v,i} = \Sigma_{j<i}|\mathbf{P}_j|$. In addition to the main point array, an auxiliary hash table of points is stored at each PE, that holds the vertices of simplices not entirely contained in $\mathbf{P}_i$ and points received in Algorithm 3. In shared memory, the data structure of Fuetterling et al. [11] seems to be applicable to speedup division of the input points. However, due to unavailability of source code its inclusion remains for future work.

Our triangulation data structure is extended from Shewchuk [20]. Simplices are stored in an array, where each simplex s consists of an ID, the $D + 1$ indices of its points in the point array – vertices(s) – and $D+1$ indices to its neighboring simplices – neighbors(s). The vertices of a simplex are sorted by index; neighbors are stored such

[1]Source code available at `https://github.com/dfunke/ParDeTria`.

Algorithm 3 DelaunayBase($\mathbf{P}_i, \mathbf{C}$): base case for distributed DT algorithm

```
Input:  point subset $\mathbf{P}_i = \{p_{i,1}, \dots, p_{i,n_i}\}$, PEs of partition $\mathbf{C}$
Output: Delaunay triangulation $T_i$ of $\mathbf{P}_i$
1: if $|\mathbf{C}| = 1$ then                                        ▷ base case
2:     return Delaunay($\mathbf{P}_i$)                               ▷ shared memory DT algorithm
3: if $i = \min \mathbf{C}$ then
4:     $\mathbf{P}' \leftarrow$ gather($\mathbf{P}_i, \mathbf{C}$)   ▷ gather points from neighbors
5:     $T' \leftarrow$ Delaunay($\mathbf{P}'$)   broadcast($T'$)     ▷ shared memory DT algorithm
6: else                                                              ▷ all other PEs
7:     send($\mathbf{P}_i$)   receive($T'$)
8: $T \leftarrow \{s \in T' : |\,\text{vertices}(s) \cap \mathbf{P}_i| \geq 1\}$   ▷ filter simplices
9: return $T$
```

that neighbor i intersects s at the facet opposite vertex i. In the distributed case, a PE i sets the upper $\log P$ bits of its simplex IDs to i to obtain globally unique identifiers. A triangulation data structure furthermore stores the indices of the simplices of its convex hull, as these serve as starting point for the border detection algorithm. To allow fast merging of two partial triangulations, simplices are stored in *blocks*. A base case triangulation results in a single block of simplices with consecutive, globally unique IDs. When merging T_1 and T_2 into a combined triangulation T, T stores two pointers to the data blocks of T_1 and T_2, along with their respective minimum and maximum simplex ID in a binary search tree. After k merge steps, this allows for random access to a simplex in $\mathcal{O}(\log k)$ time; scanning is still in $\mathcal{O}(1)$.

The vertex hash function $h_v(\cdot)$ required for the simplex and facet hash function needs to minimize the expected number of collisions while being efficiently computable. In our experiments we found that setting $h_v(v)$ equal to the point index rotated by its lowest byte value to suit our needs. Whether our algorithm would benefit from a more sophisticated hash function, i. e., a provable universal hash function, remains for future work. Concurrent hash table operations are at the heart of the merging and neighborhood update algorithms. The efficient, growable, concurrent hash table by Maier et al. [19] is used in our implementation. We extended their implementation to multisets for facet hash lookups during neighborhood updates.

Intel's Threading Building Blocks (TBB) library[2] is used for shared memory parallelization. Particularly, its concurrent work queue is employed in the border simplex detection and neighborhood update algorithms.

Geometric algorithms need to address the limited floating-point precision of current hardware. Our proposed D&C scheme relies on combinatorial computations on hash values except for the detection of the border simplices of two partial triangulations. We use the fast sphere-box overlap test of Larsson et al. [16] to determine if the (hyper)-circumsphere of a given simplex intersects with the bounding box of the opposite partial triangulation. The test does not suffer from floating-point inaccuracies like the *orientation*- and *inSphere*-tests required by the base case algorithm [21].

Distribution	Points	Simplices	Runtime
	shared memory		
uniform	50 000 000	360 542 380	64 s
normal	50 000 000	361 877 812	83 s
bubble	50 000 000	361 638 812	70 s
ellipsoid	500 000	84 408 498	169 s
lines	10 000	122 396 140	292 s
CuZr	4 000 000	28 927 267	8 s
CuZr	100 000 000	634 926 984	148 s
	distributed memory		
uniform	2 048 000 000	22 112 081 080	92 s
normal	204 800 000	5 861 711 093	50 s

Table 2: Evaluated point sets and their resulting triangulations. Shared memory runtimes are reported for 32 cores, distributed memory runtimes for 2048 cores with 4 cores per MPI process.

7 Evaluation

Batista et al. [2] propose three input point distributions to evaluate the performance of their DT algorithm: N points distributed uniformly a) in the unit cube; b) on the surface of an ellipsoid; and c) on skewed lines. Furthermore, Lee et al. [17] suggest normally distributed input points around d) the center of the unit cube; and e) several points within the unit cube – called "bubbles". All experiments are performed in three-dimensional space.

We furthermore test our algorithm with two real world datasets from material science, where Voronoi

[2]`https://www.threadingbuildingblocks.org/`

analysis is used in simulation studies of liquids, glasses and solids to explore their atomic structure, e. g. the characteristic arrangement of near neighbors of an atom [22]. Amorphous Copper-Zirconium (CuZr) alloys are used as benchmark compound in the field [24]; we evaluate two 50/50 copper/zirconium system consisting of four million and 100 million atoms.

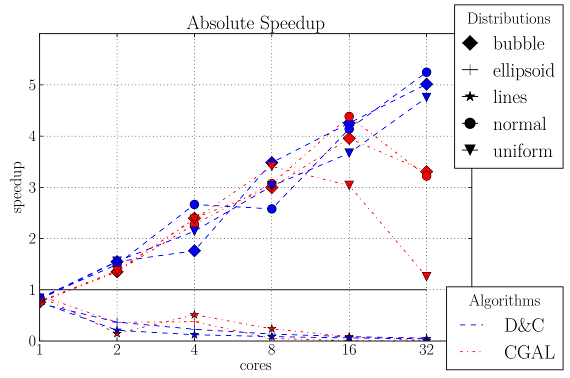
Table 2 gives an overview of all evaluated point sets, along with the size of their resulting triangulation.

The shared memory algorithm was evaluated on a machine with dual Intel Xeon E5-2683 hexadeca-core processors and 512 GiB of main memory. We use the sequential Delaunay triangulation algorithm of CGAL 4.7 as base case in Algorithm 1 and compare our implementation to the parallel DT algorithm of CGAL [12]. In both cases, CGAL is configured to use exact predicates.[3]

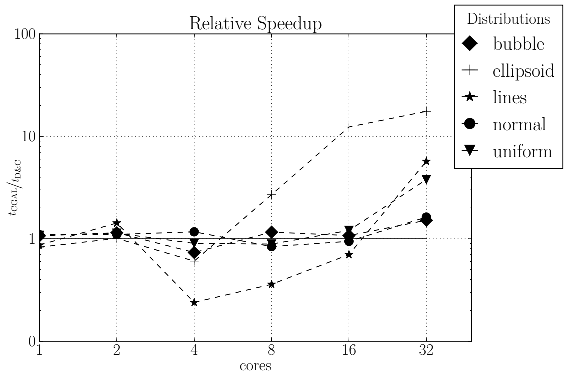
The distributed memory experiments were conducted on InstitutsCluster II at the Steinbuch Centre for Computing at Karlsruhe Institute of Technology. The cluster contains 480 compute nodes with dual Intel Xeon E5-2670 octa-core processors and 64 GiB of main memory, connected by an InfiniBand 4X QDR interconnect. A single job may use up to 128 nodes (≡ 2048 cores). OpenMPI version 1.8.6 was used as message passing library.

7.1 Shared Memory Algorithm Figure 2 shows the performance of our algorithm in comparison to CGAL's sequential and parallel Delaunay triangulation for the aforementioned input distributions. The uniform, normal and bubble distribution show good scaling behavior. The bubble distribution has few points near the border of a partition and thus a low number of vertices in the border triangulation. CGAL's parallel incremental insertion encounters low congestion in its locking, since the vertices of one bubble are mostly handled by a single thread due to spatial sorting [2]. Uniformly distributed points have larger – but compact and even – border triangulations, resulting in good load balancing between the partitions. In contrast, normally distributed points result in larger border triangulations around the center, yet they profit from small cuts in the outer regions. When using only a single socket of our test machine, our algorithm performs on par with CGAL's implementation. For multiple sockets, however, we clearly outperform CGAL. Our algorithm adapts well to the NUMA setting, as – except for the final merge step – the entire algorithm operates exclusively on socket-local data. Contrarily, CGAL's incremental insertion algorithm requires continuous communication between threads of the two sockets.

[3] `CGAL::Exact_predicates_inexact_constructions_kernel`



(a) absolute speedup over sequential CGAL



(b) relative speedup over parallel CGAL

Figure 2: Speedup of our shared memory D&C algorithm and CGAL's parallel DT implementation for various point distributions.

The ellipsoid and skewed line distributions are specifically tailored to be hard inputs for both implementations. The former is hard due to its large convex hull, the latter due to the quadratic number of simplices in the input size. Both parallel implementations fall below the throughput of the sequential reference. Our implementation's performance degrades less than CGAL for the ellipsoid, as only the simplices of the convex hull whose circumspheres intersect the splitting plane contribute to the border triangulation, while CGAL suffers from high congestion on the inner simplices of the ellipsoid. Congestion on the inner simplices also leads to CGAL's bad performance for skewed lines. Our implementations suffers even more, due to almost all simplices intersecting the splitting plane for at least one cutting dimension.

Figure 3 shows the fairly regular structure of the atoms of a CuZr alloy. This results in compact and even cuts between the partitions of the triangulation similar to uniformly distributed points. The scaling behavior is therefore also comparable to this point distribution.

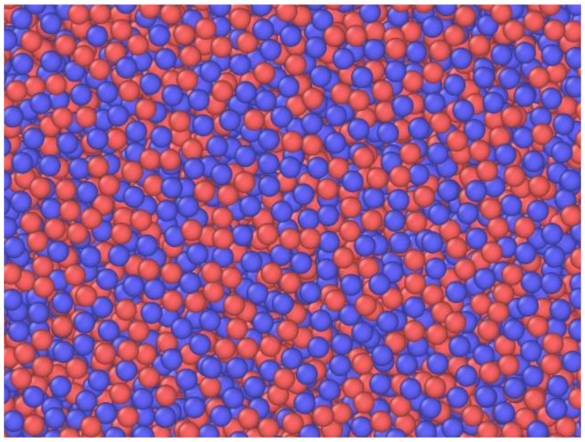
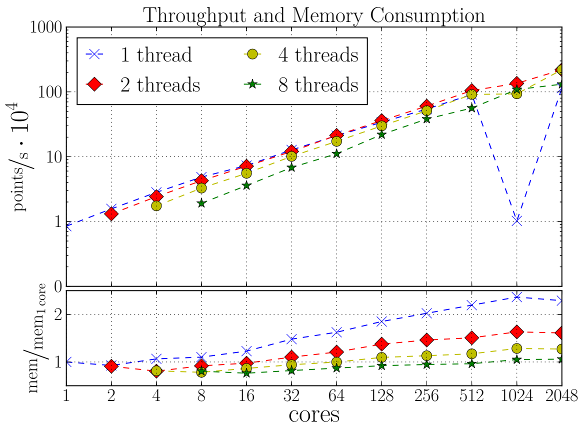


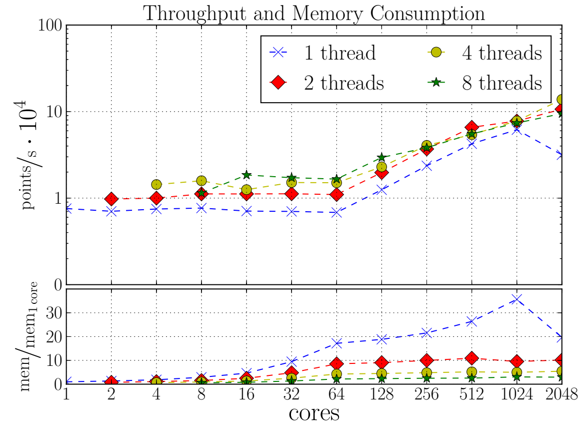
Figure 3: Structure of an amorphous CuZr alloy. Copper atoms are depicted in red, zirconium atoms in blue.

7.2 Distributed Memory Algorithm Figure 4 shows the weak scaling behavior of our distributed memory implementation. In the experiment each core processes one million input points for the uniform distribution and 100k points for the normal distribution, due to memory limitations at the central nodes described below. We show the behavior for different configurations of hybrid parallelization, ranging from one thread per MPI process – i. e., one process per core with sequential base case – to eight cores per MPI process – i. e., one process per socket with our shared memory parallel D&C algorithm as base case in Algorithm 3. We would have expected one process per socket to yield the best results, as this configuration is NUMA aware and requires the least inter-process communication. Nevertheless, for uniformly distributed points, two cores per process show the best performance with a speedup of ≈ 260 for 2048 cores. For the normal distribution, we attain a more modest speedup of 18 for 2048 cores due to the lack of load-balancing in our current implementation. PEs close to the center of the distribution have a much higher workload than others, preventing larger speedup gains. If the input points are not pre-partitioned, we observe a runtime overhead for the additional all-to-all communication of $10 - 15\,\%$ on average.

With increasing number of PEs and input size the recursion depth increases. Thus, more border triangulations are required to produce the global triangulation. Furthermore, each PE needs to store more simplices that are only partially contained in its original input point set. This increases memory consumption per PE. In the uniformly distributed setting, our measurements show that while the total number of processed points grows by three orders of magnitude going from one to 2048 PEs, memory consumption per core only increases by a factor of 2.2. For the normal distribution, the memory increase exceeds a factor of 30. This is due to the large number of points processed by the central PEs. Furthermore, PEs close to the center also have to store many simplices only partially contained in their original point set. Again, load-balancing would mitigate this issue. For both distributions, the benefit of hybrid parallelization is apparent, as more threads per MPI process result in fewer processes, leading to reduced recursion depth in the distributed algorithm.



(a) uniform distribution



(b) normal distribution

Figure 4: Throughput and memory overhead of our distributed memory algorithm in a weak scaling experiment.

8 Conclusions

We present a novel divide-and-conquer algorithm for computing the Delaunay triangulation in arbitrary dimension, that reduces the merging of two subproblems to re-triangulating a small subset of their vertices and using efficient hash table operations to combine the three triangulations into one. All steps of the merging are parallelized. We are able to perform on par with or better than CGAL's parallel three-dimensional DT implementation in shared memory and show good scalability for our approach in distributed memory up to 2048 cores and two billion input points.

Future work will address flexible load balancing and work division strategies aiming at small border sizes. This will yield a more robust algorithm capable of processing large realistic inputs from a variety of fields. Furthermore, many of these inputs require periodic boundary conditions, which need to be handled efficiently by our algorithm.

References

[1] Aggarwal, A., Chazelle, B., Guibas, L.: Parallel computational geometry. Algorithmica 3(1), 293–327 (1988)

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

[3] Bentley, J.: Multidimensional binary search trees used for associative searching. Communications of the ACM 18(9), 509–517 (1975)

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

[5] Chen, M.B.: The Merge Phase of Parallel Divide-and-Conquer Scheme for 3D Delaunay Triangulation. pp. 224–230. IEEE (2010)

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

[7] Chrisochoides, N., Sukup, F.: Task parallel implementation of the Bowyer-Watson algorithm. In: International Conference on Numerical Grid Generation in Computational Fluid Dynamics and Related Fields. pp. 773–782. North-Holland (1996)

[8] Cignoni, P., Montani, C., Perego, R., Scopigno, R.: Parallel 3D Delaunay Triangulation. Computer Graphics Forum 12(3), 129–142 (1993)

[9] Cignoni, P., Montani, C., Scopigno, R.: DeWall: A fast divide and conquer Delaunay triangulation algorithm in E^d. Computer-Aided Design 30(5) (1998)

[10] Delaunay, B.: Sur la sphère vide. A la mémoire de Georges Voronoï. Bulletin de l'Académie des Sciences de l'URSS. Classe des Sciences Mathématiques et Naturelles (6), 793–800 (1934)

[11] Fuetterling, V., Lojewski, C., Pfreundt, F.J.: High-Performance Delaunay Triangulation for Many-Core Computers. In: Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics. The Eurographics Association (2014)

[12] Hert, S., Seel, M.: dD convex hulls and delaunay triangulations. In: CGAL User and Reference Manual. CGAL Editorial Board, 4.7 edn. (2015)

[13] Isenburg, M., Liu, Y., Shewchuk, J., Snoeyink, J.: Streaming Computation of Delaunay Triangulations. ACM Transactions on Graphics 25(3), 1049–1056 (2006)

[14] Joe, B.: Construction of three-dimensional Delaunay triangulations using local transformations. Computer Aided Geometric Design 8(2), 123–142 (1991)

[15] Kohout, J., Kolingerová, I., Žára, J.: Parallel Delaunay triangulation in E2 and E3 for computers with shared memory. Parallel Computing 31(5), 491–522 (2005)

[16] Larsson, T., Akenine-Möller, T., Lengyel, E.: On Faster Sphere-Box Overlap Testing. Journal of Graphics, GPU, and Game Tools 12(1), 3–8 (2007)

[17] Lee, S., Park, C.I., Park, C.M.: An improved parallel algorithm for delaunay triangulation on distributed memory parallel computers. Parallel Processing Letters 11, 341–352 (2001)

[18] Lo, S.: Parallel Delaunay triangulation in three dimensions. Computer Methods in Applied Mechanics and Engineering 237-240, 88–106 (2012)

[19] Maier, T., Sanders, P., Dementiev, R.: Concurrent hash tables: Fast and general?(!). In: Principles and Practice of Parallel Programming (PPoPP). pp. 34:1–34:2. ACM (2016)

[20] Shewchuk, J.: Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. Applied Computational Geometry Towards Geometric Engineering 1148, 203–222 (1996)

[21] Shewchuk, J.: Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. Discrete & Computational Geometry 18(3), 305–363 (Oct 1997)

[22] Stukowski, A.: Structure identification methods for atomistic simulations of crystalline materials. Modelling and Simulation in Materials Science and Engineering 20(4), 045021 (2012)

[23] Su, P., Drysdale, R.L.S.: A comparison of sequential delaunay triangulation algorithms. In: Symposium on Computational Geometry (SCG). pp. 61–70. ACM (1995)

[24] Wang, D., Li, Y., Sun, B.B., Sui, M.L., Lu, K., Ma, E.: Bulk metallic glass formation in the binary Cu–Zr system. Applied Physics Letters 84(20), 4029–4031 (2004)

[25] Wu, H., Guan, X., Gong, J.: ParaStream: A parallel streaming Delaunay triangulation algorithm for LiDAR points on multicore architectures. Computers & Geosciences 37(9), 1355–1363 (2011)