Localizing the Delaunay Triangulation and its Parallel Implementation

Renjie Chen
Technion
Haifa, Israel
renjie.c@gmail.com

Craig Gotsman
Technion
Haifa, Israel
gotsman@cs.technion.ac.il

***Abstract*—We show how to *localize* the Delaunay triangulation of a given planar point set, namely, bound the set of points which are possible Delaunay neighbors of a given point. We then exploit this observation in an algorithm for constructing the Delaunay triangulation (and its dual Voronoi diagram) by computing the Delaunay neighbors (and Voronoi cell) of each point independently. While this does not lead to the fastest serial algorithm possible for Delaunay triangulation, it does lead to an efficient parallelization strategy which achieves almost perfect speedups on multicore machines.**

Delaunay triangulation; Voronoi diagram; parallel computation

I. INTRODUCTION

The Delaunay triangulation (DT) of a set of planar point sites and its dual, the Voronoi diagram (VD), are among the most fundamental structures in computational geometry. DT is the triangulation of the sites such that each triangle satisfies the empty circumcircle property, i.e. its circumcircle contains none of the other sites, thus, intuitively, the DT has the "fattest" triangles among all possible triangulations of the sites. The VD is a partition of the plane into (possibly unbounded) convex polygonal cells, one per site, such that the points inside each cell are closer to the site corresponding to that cell than any other site. Due to their many desirable properties, DT and VD are widely used in many fields of science.

Because of the duality relationship between them, DT and VD can be converted to each other in linear time. The classical algorithms for computing DT and VD in O($n$log$n$) time are Dwyer's divide and conquer algorithm [5], Fortune's plane sweep algorithm [6], incremental construction [19] and variations based on randomization [4], lifting to three dimensions and computing the convex hull [29]. Su et al. [2] provide a thorough survey and comparison of these algorithms.

Most of these serial algorithms achieve $O(n\log n)$ time complexity. When the input is drawn from a uniform spatial distribution of sites, more efficient algorithms are possible. Bentley et al. [1] first proposed a linear expected time algorithm for the VD based on the idea of finding the Voronoi cell of each site independently. Using a cell data structure and a spiral search [1, 3] technique, the algorithm finds all other sites in the vicinity of the given site, and builds its Voronoi cell from these. A rough estimate for the region that contains all the possible neighboring Voronoi sites of each interior site is given. One of the traditional algorithms is used both for finding the VD of the periphery of the point set, and also as a last resort for the rare cases where the algorithm fails on an interior point. Using the same data structure, Maus [3] proposed another linear expected time algorithm for the DT of sites drawn from a uniform distribution by greedily finding all the Delaunay edges starting from an initial Delaunay edge list. In addition to the sites, this algorithm requires the convex hull of the sites as input, which serves as the initial Delaunay edge list.

Due to the increasing need to rapidly construct the DT in many applications which may involve millions of points, there has been much research on computing DT in parallel. Many of these [16-28] are based on the state-of-the-art serial algorithms, such as divide and conquer and incremental construction. They achieve parallelism by partitioning the set of sites into smaller subsets and using parallel processing to construct the DT of each subset separately. The separate triangulations are then combined by edge flipping where needed. These algorithms make use of a "master" processor which assigns tasks to "slave" processors, attempting to balance well the load between processors. Thus the master becomes a bottleneck at some stage, and the algorithms do not scale well with the number of processors. Furthermore, these algorithms do not fit well into the current model of multi-core processors and general purpose graphics processing units (GPU), in which no master process should be present.

In this paper, we present a new algorithm to construct the VD and DT. This is achieved by running an incremental half-plane intersection method to compute the Voronoi cell and Delaunay neighbors of each site independently. A key Locality Lemma, which may be of independent interest, allows us to limit the candidate set of the Delaunay neighbors to be considered for each site, thus we drastically reduce the $O(n^2\log n)$ time complexity of the naïve half-plane intersection algorithm [30]. For sets of uniformly distributed sites, the complexity is $O(1)$ per site. The algorithm is extremely simple and easy to implement. Although this does not result in a serial algorithm which is any faster than state-

of-the-art serial algorithms, it does, as opposed to other algorithms, lend itself to easy and efficient parallelization. With an extra (quite straightforward) optimization as described in Section VI, the resulting parallel implementation achieves almost perfect speedups.

II. Related Work

Serial algorithms for computing the VD and the dual DT are well known, so we will not survey them here. Instead, we will concentrate on the lesser-known parallel algorithms.

Rong et al. [10] proposed a parallel algorithm to compute the DT using the GPU and CPU in tandem. With the GPU they compute the DT of a modified point set constructed by snapping each original input point to the nearest grid point ("pixel"). After computing the DT of the grid point set using the GPU, they move each point back to its original coordinates and repair the triangulation by edge flipping where necessary. Because of its inherent serial nature, the edge flip step is done on the CPU, thereby making the complete algorithm only partially parallel, with limited speedup over serial algorithms. Very recently, Qi et al. [7] improved this algorithm by implementing also the edge flipping on the GPU, thus making the entire algorithm GPU-based. They also extend the algorithm to a constrained DT.

It is well known that the DT contains the nearest neighbor graph as a subgraph. Maus and Drange [9] generalized this property to the k nearest neighbors, namely, they prove that for any point x in the point set X with k nearest neighbors $\{b_1, b_2 \ldots b_k\}$ (b_i are sorted by their distances to x), the j'th closest neighbor b_j is a neighbor of x in the DT of X if it is not contained in any of circles having the segment xb_i $\{i=1, 2,\ldots j\text{-}1\}$ as its diameter. Based on this and the nearest neighbor graph property, they presented two algorithms for constructing the DT in parallel. With the nearest neighbor graph and k-nearest neighbor graph as starting points, they use an incremental algorithm [3] and constrained DT algorithm to find the Delaunay neighbors of each point independently in the two algorithms respectively. However in both algorithms, a serial algorithm is employed to compute the convex hull of the point set, which is necessary for their algorithm to construct the DT, therefore making these algorithms also only partially parallel.

Very recently, Reem [15] adapted his ray-shooting-based parallel algorithm for computing the approximate VD in general settings [14] (general sites, and general normed space) to compute the exact VD by carefully utilizing the information along the rays. A formal proof is given to show that the algorithm will always terminate with the correct result within a finite number of steps. Experimental results show that this algorithm, equipped with appropriate spatial data structures for the sites, achieves almost linear expected time complexity for uniform distributions. However, since the VD is clipped to a rectangular domain, an important component of the DT - the convex hull of the point set - will be incomplete when transforming the VD into a DT.

Shewchuk [8] proposed the *Star Splaying* algorithm for transforming a triangulation which is nearly DT into a DT. The algorithm seeks to adjust the stars, the *candidate* Delaunay one-rings of all the vertices, so that they agree with each other and therefore form a DT. *Star Splaying* is akin to the Delaunay edge flip algorithm, and it requires (near DT) connectivity in addition to the point set as input. In this paper, we propose an algorithm which also seeks to find the Delaunay one-rings for all the vertices. However, it does not check the relation between different Delaunay one-rings, rather computes the Delaunay one-rings of the vertices independently of each other, making it inherently parallelizable.

III. Localizing the Delaunay Triangulation

First some terminology.

Delaunay edge: an edge xy is a *Delaunay edge* if it is contained in the DT.

Delaunay neighbor: a vertex x is a *Delaunay neighbor* of y if xy is a Delaunay edge.

Delaunay one-ring: the *Delaunay one-ring* of a vertex x is the set of all Delaunay neighbors of x.

Voronoi vertex: a vertex of a Voronoi cell boundary

Half-plane: the half-plane between two points c and v is the bisector of the points.

The terms point, vertex and site are interchangeable through the paper.

In this section, we present a key Lemma, illustrated in Fig. 1, that leads to the main algorithm of this paper. In general, Delaunay edges are short, because they connect a site to other sites in close proximity. However, there is no strict upper bound on the length of Delaunay edges, and in some extreme cases, edges can span the entire point set. Furthermore, there is no easy rule of thumb that can predict which sites exactly will be the Delaunay neighbors of a given site. Thus, there is value in a rule which localizes the Delaunay triangulation, namely, strictly bounds the set of possible Delaunay neighbors of a given site.

Local Delaunay Lemma

Let X be a set of points in the plane. If the ordered subset $P=\{p_1, p_2,\ldots, p_n\}\subseteq X$ forms a simple polygon containing $c\in X$, then the Delaunay neighbors of c are contained in the union of the circumcircles of the n triangles formed by c and every two consecutive points of P (irrespective of the triangle orientation).

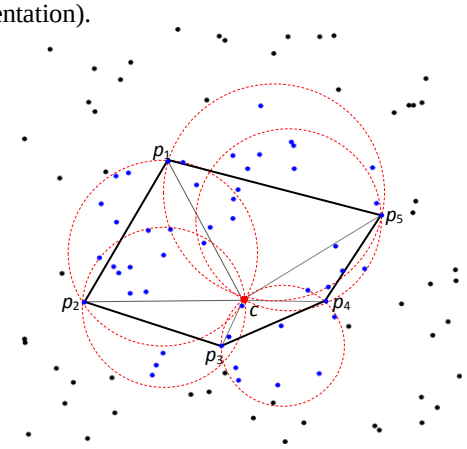


Figure 1. The Local Delaunay Lemma. Only the blue sites, inside or on the red circumcircles, can be the Delaunay neighbors of the site c.

Proof: Let $CC=\bigcup_i cc_i$, where cc_i is the circumcircle of $\triangle cp_ip_{i+1}$.

For any point $v \notin CC$, since P is closed and contains c, v must be contained in some (at least one) closed sector defined by c and an edge on P, say p_ip_{i+1}. The sector is defined as the unbounded region inside the angle $\angle p_i c p_{i+1}$, and a closed sector includes the two defining rays, cp_i and cp_i, as shown in Figure 2.

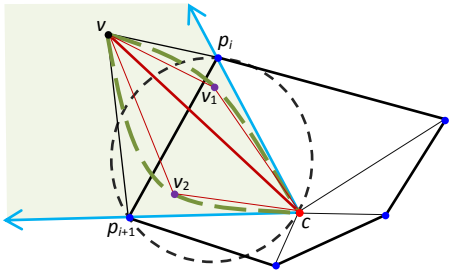


Figure 2. *Proof of the Local Delaunay Lemma.*

Assume cv is a Delaunay edge. This implies that there exists a circle through c and v which does not contain any other point of X, including p_i and p_{i+1} [11]. Obviously, cv cannot be at the boundary of the sector, as this would imply that either p_i or p_{i+1} is inside this circle. Therefore p_i and p_{i+1} are on opposite sides of the chord cv that divides this circle into two arcs, as shown in Fig. 2. So, on the one hand, for any point v_1 on the arc which is on the same side of cv as p_i, we have $\angle cv_1v \geq \angle cp_iv$. Similarly for any point v_2 on the complementary arc, we have $\angle cv_2v \geq \angle cp_{i+1}v$. Therefore

$$\angle cv_1v + \angle cv_2v \geq \angle cp_iv + \angle cp_{i+1}v$$

Since v is outside the circumcircle of $\triangle cp_ip_{i+1}$, we have

$$\angle cp_{i+1}v + \angle cp_iv > \pi$$

which leads to

$$\angle cv_1v + \angle cv_2v \geq \angle cp_iv + \angle cp_{i+1}v > \pi$$

On the other hand, since v_1 and v_2 are on the two complementary arcs of the chord cv:

$$\angle cv_1v + \angle cv_2v = \pi$$

which is a contradiction. Thus cv cannot be a Delaunay edge. □

Note that by including the point at infinity, the Local Delaunay lemma can be generalized to the case that the polygon P is unbounded, as is the case for points on the convex hull. Consider the "closed" polygon $P \cup \infty$, any point inside the unbounded sector of P falls inside either or both of the circumcircles of the two infinite triangles, which are essentially two halfspaces.

We also note an important special case of the Local Delaunay Lemma. If P is exactly the set of Delaunay neighbors of c, then the union of the circumcircles will contain no other points, as expected. The Local Delaunay Lemma allows us to significantly limit the number of points we need to consider when searching for the Delaunay neighbors of a point x. Indeed, none of the points outside the union of the circumcircles of triangles incident to c can be a Delaunay neighbor of c. Therefore it suffices to consider only the points inside this union.

IV. Delaunay Triangulation

Based on the Local Delaunay Lemma, we now outline an algorithm for computing the Delaunay neighbors (and Voronoi cell) of a given point c.

Algorithm 1.

1. Find a (simple) polygon $P_0=\{p_1, p_2,.., p_k\}$, containing c.
2. Initialize c's candidate Delaunay one-ring: $P=P_0$.
3. Initialize c's candidate Voronoi cell: $Q=\{q_1, q_2,.., q_k\}$, where q_i is the circumcenter of $\triangle cp_ip_{i+1}$.
4. Construct the list of Delaunay neighbor candidates
   $$V = \{x \in X : \exists i \ \|x - q_i\| < \|c - q_i\| \ \}$$
5. for each $v \in V$
6. Compute the half-plane H_v defined by the bisector of v and c, containing c.
7. $Q \leftarrow H_v \cap Q$
8. Update P, based on Q (See Algorithm 2)
9. end

To find the complete DT of a point set X, Algorithm 1 is run for each point c in X.

The core of the algorithm is half-plane intersection in the loop described in Steps 5-9. Note that the candidate Voronoi cell changes (actually, shrinks) between iterations, therefore the halfspace corresponding to a vertex in V may not intersect it, thus not change it. This can be checked by inspecting whether the candidate vertex v is inside any of the circumcircles defined by the current P (or, equivalently, the current Q). Actually, for each candidate v, we can find the sector $p_ic p_{i+1}$ that v resides in, then by comparing the distance $\|q_i - v\|$ with the distance $\|q_i - c\|$, we can tell if v is inside the current union of circumcircles. Since the vertices in P are ordered (CCW), we can find the sector containing v in $O(\log d)$ time, where d is the length of P. The intersection with half-plane H_v is now done easily, since, starting from the sector, we can find the two edges on Q that H_v intersects in constant time, as shown in Fig. 3. Then we can simply keep the vertices of Q that are closer to c than v is, and replace the other vertices of Q with the centers of the new circumcircles.

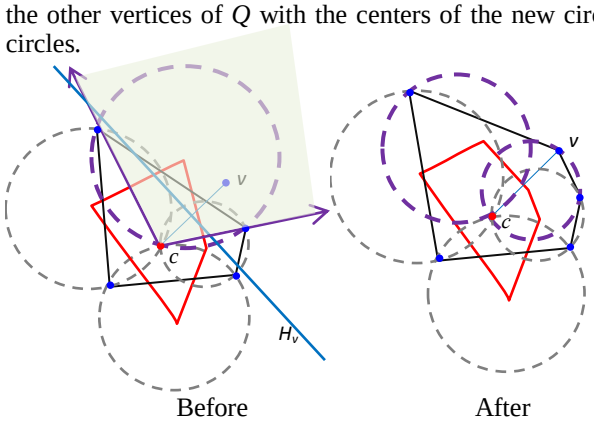


Figure 3. Intersecting the (solid red) candidate Voronoi cell of c with (blue) halfplane H_v associated with the candidate v. After the intersection, the previous (purple) circumcircle is replaced with two new (purple) circumcircles, and v is added to the (blue) candidate Delaunay one-ring of c. The relevant sector is shaded.

Some of the initial triangles incident to c could be very skinny and have large circumcircles, which results in the list of Delaunay candidates V constructed in Steps 1-4 containing many points. Since the Local Delaunay Lemma holds for any polygon containing c, we can optimize the routine by implementing the loop of Algorithm 1 in an incremental manner. Starting from the initial candidate Delaunay polygon P, in each iteration we deal with one of its edges, p_ip_{i+1}. We check whether the circumcircle of the corresponding triangle $\triangle cp_ip_{i+1}$ contains a point. Only if it does do we run the half-plane intersection routine and update P and Q. Algorithms 2 shows the pseudo-code for the incremental update of P and Q.

Algorithm 2.

```
1.  LocDT(c, P, Q)
2.  i = 1;
3.  while i <= length(P)
4.      V = {x ∈ X : ||x − q_i|| < ||c − q_i|| }
5.      if V is empty
6.          i++
7.      else /* half-plane intersection */
8.          v = any vertex in V
9.          j = index of first vertex of Q inside H_v
10.         m =index of last vertex of Q inside H_v
11.         o_1 = circumcenter(p_{m+1}, v, c)
12.         o_2 = circumcenter(v, p_{j-1}, c)
13.         Q = { q_1,...q_m, o_1, o_2, q_{j-1},...q_end}
14.         P = { p_1,...p_{m+1},v, p_{j-1},...p_end}
15.         i = m+1
16.     end
17. end
```

In Algorithm 2, m+1 and j-1 are computed modulus length(P).

For the algorithm to have good performance, we need a data structure that supports efficient disk range queries on a set of points. We use the standard cell/bucket data structure proposed by Bentley et al. [1] and Maus [3]: the domain, the bounding box of the point set, is partitioned into boxes of the same size, and an index array is used to store the indices of the points inside each box. The points inside each box can be retrieved in constant time, and the index of the box containing any point can also be computed in constant time. For point sets containing n points, we partition the domain into $\sqrt{n} \times \sqrt{n}$ boxes. Then each box contains a single point on the average, and we found that for uniformly-distributed point sets it takes less than 10 half-plane intersections to find the exact Voronoi polygon and Delaunay one-ring. To make the following operations simpler, we scale the (square) domain to be the unit square.

It remains to provide the details of Steps 1-3 in Algorithm 1, i.e. building an initial candidate Delaunay polygon P_0 and candidate Voronoi cell Q_0 for c. Obviously, we would like Q_0 to be as *tight* as possible. Also, these steps should be as fast as possible.

We use the "spiral" search technique to find a fixed number, say 6, of non-empty cells around c, and then sort all the points inside these cells in CCW order around c to obtain the initial polygon P_0. The dark red spiral in Fig. 4 shows the procedure of the "spiral" search. However in some cases, this initial polygon will not contain c. Worse still, when c is on the convex hull of the point set, there exists no polygon containing it at all. Luckily, as mentioned in the previous section, we can always include the infinite point into P_0, and make it "closed".

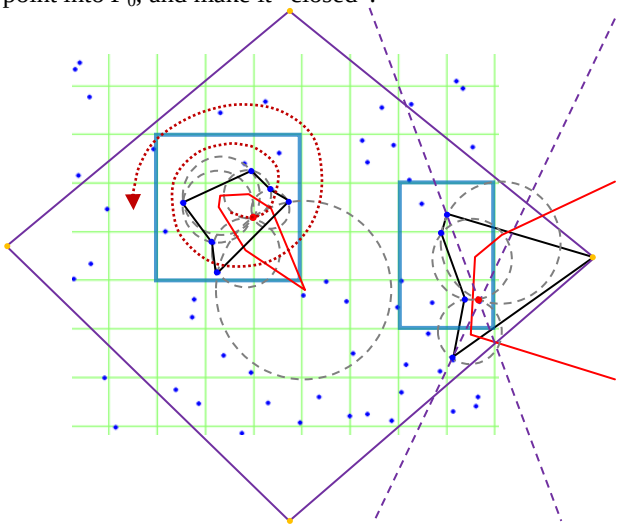


Figure 4. (Black) Candidate Delaunay one-ring polygon P and corresponding (red) dual candidate Voronoi cell Q constructed from the points inside the cells inside the blue rectangles. The purple quadrilateral is the initial candidate Delaunay polygon formed by 4 virtual "infinite" points. The right portion shows a candidate Delaunay one-ring containing one virtual "infinite" point. Circumcircles involving the "infinite" point are actually (dashed purple) halfplanes.

The initial candidate Voronoi cell Q_0 can be constructed as the "dual" of the site c by taking the i'th vertex of Q_0 to be the circumcenter of $\triangle cp_ip_{i+1}$ (the circumcenter of infinite triangle is the infinite point). Unfortunately, this does not always result in a simple polygon, as the triangulation inside the polygon P_0 is not always Delaunay itself. This will interfere with the later half-plane intersecting procedure, since the initial candidate Voronoi cell must be valid (simple and convex) for it to be correct. Thus we must prune the polygon P_0 in order to make Q_0 valid. Obviously, this can be achieved by intersection of all the half-planes defined by c and all the vertices of P_0. To simply this process, we first construct P_0 to be the square formed by 4 virtual points, {(-1/2, -1/2), (3/2, -1/2), (3/2, 3/2), (-1/2, 3/2)}, outside the domain (the unit square), and take the candidate Voronoi cell Q_0. to be the dual of P_0.Then for each vertex of P_0 (without the infinite point), we run the same half-plane intersection routine as in Step 5-9 of Algorithm 1, to update P and Q.

After this step, we will usually be left with a very tight containing polygon P. This will rule out the majority of the point set from the Delaunay neighbor candidate list V, In fact, quite a few of the circumcircles are already empty, as shown in Fig. 4, therefore the core of Algorithm 1, Steps 5-9, needs to process only a very small number of candidate neighbors.

Thanks to the introduction of virtual (infinite) points, we do not need to take special care of the points on the convex hull. However, note that the circumcircles of the triangles containing the virtual points are actually halfspaces, therefore the Delaunay neighbor candidates list V should be constructed slightly differently. The point-in-circumcircle test in Algorithm 1 should also be replaced with a point-in-halfplane test.

The top row of Fig. 7 shows the evolution of P and Q in a typical scenario.

Note that our algorithm does not take degenerate cases into account. When such situations exist in the given point set, i.e. more than 3 points form an empty circle, then the Delaunay one-rings of these points, as found by our algorithm, may not agree with each other, rendering the complete DT invalid. In such cases, we can slightly modify the algorithm by perturbing each point randomly so that the degeneracy disappears while the DT is preserved.

V. Parallel Delaunay Triangulation

The Delaunay triangulation algorithm can be parallelized in a straightforward manner. In fact, since the same procedure is applied to each point, and the processing of each point is independent of the others, we can simply parallelize the loop applying Algorithm 1 to all input points. Before running the main algorithm, we need to partition the point set into uniform cells/buckets and build the data structure. Due to its regularity and simplicity, this can also be parallelized using standard thread synchronization techniques, such as atomic operations. In any case, this preprocessing accounts for less than 0.1% of the serial processing time.

A. Avoiding redundancy

Each Delaunay triangle features in each of the three Delaunay one-rings of its vertices, therefore simply applying Algorithm 1 to each point independently will compute each Delaunay triangle three times. A similar analysis reveals that each Delaunay edge will be computed four times. When running the serial version of the algorithm, some of this can be saved in an obvious manner by updating the Delaunay neighbor information for each p_i after finding P - the Delaunay one-ring of c - and then skipping the Delaunay neighbors already found when applying Algorithm 1 on p_i. Alas, it is difficult to apply this simple strategy when running the algorithm in parallel, as this requires too much coordination between processors when updating the Delaunay neighbor information.

Fortunately, it is still possible to reduce the redundant computation also in the parallel case. Since a triangle is always intersected by one of the three vertical lines through its vertices, we can construct the entire DT by computing *only the two Delaunay triangles that intersect the vertical line through each point*. As shown in Fig. 5, only the two gray Delaunay triangles need to be found for point c. To implement this optimization, we modify Algorithm 1 and 2 accordingly. In Algorithm 1, we build the initial candidate Delaunay one-ring using only four points (including virtual points if necessary); one in each of c's four quadrants. In Algorithm 2 we process only the two edges of the Delaunay polygon which intersect the vertical through c. The bottom row of Fig. 7 shows the evolution of P and Q in this optimized version of the DT algorithm, which may be compared to the evolution in the serial version of the algorithm in the top row of that figure.

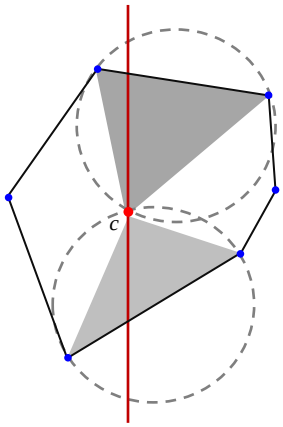


Figure 5. Reducing the DT computation by computing only the two Delaunay triangles incident on c that intersect the red vertical through c.

B. Load balancing

To achieve the best performance of a parallel algorithm, it is important to balance the workload of the parallel tasks, since the overall performance is determined by the slowest processor. However, Algorithm 1 performs quite differently for interior points and for boundary points, even in a uniformly distributed point set. For most interior points, Algorithm 1 has constant time complexity w.r.t. n, the size of the point set, while for points on the convex hull and some interior points nearby, the time complexity is $O(\sqrt{n})$. This is because the Delaunay polygon P contains the infinite point, which indicates that the circumcircle of some triangle is a half-space, meaning that $O(\sqrt{n})$ cells of points must now be checked. So although the serial version of the algorithm treated all points equally, the parallel version must be wary of points on the convex hull.

Since it is difficult to tell apriori which points are on the convex hull, we adopt a strategy which disguises these points as interior points. This is done by using a *periodic* DT [36], which is the DT of a point set which is replicated in tiles over the plane. Thus each point in the periodic DT may be considered an interior point and the time complexity of Algorithm 1 will then always be constant. Based on this, we may adjust Step 1 and 4 in Algorithm 1 by replacing the virtual (infinite) points with replicas of c in different periods and build the Delaunay neighbor candidate list in periodic space. Fig 6. shows an example of a periodic DT.

It remains to describe a method to transform a periodic DT into a regular DT in linear time. First we remove the triangles crossing the boundary of the domain in the periodic DT, and find the resulting triangulation boundary vertices by checking whether their Delaunay one-ring is closed. Since no new triangles are introduced, all the existing triangles stay Delaunay, and we need only to find the Delaunay edges between the triangulation boundary and the convex

hull. The latter can be traced from the boundary vertices in time linear in the number of boundary vertices [31]. Then, as shown in Fig. 6, the region between the (blue) boundary and the (red) convex hull is the union of simple closed polygons, which may be identified by "walking" along the boundary. A simplified version of our Algorithm 1 may be used to triangulate these polygons by running on their vertices in parallel. For each vertex c on any of these simple polygons G, we replace $c \in G$ with the infinite point, to obtain the initial candidate Delaunay one-ring P, and construct the Delaunay neighbor candidate list V as all the vertices of P.

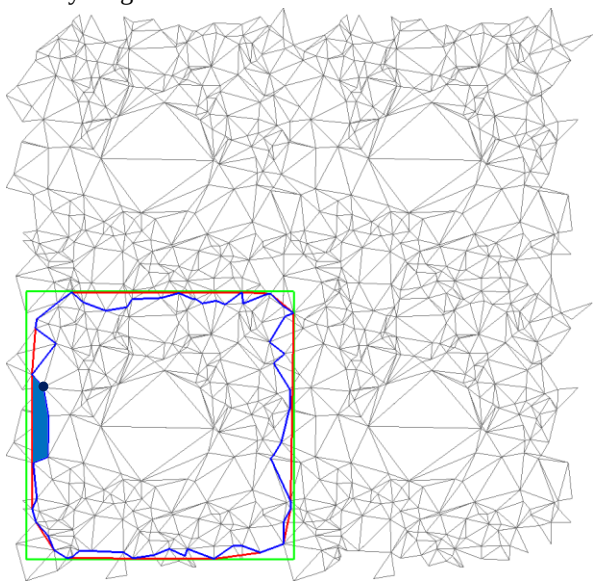


Figure 6. Transforming a periodic DT to a regular DT. The green square marks the original domain of the input point set and the red polygon its convex hull. Four periods are shown. The blue polygon marks the boundary of the DT after removing all triangles of the periodic DT crossing the original (green) domain boundary. Only the region between it and the (red) convex hull, which consists of the union of simple polygons, such as the blue polygon, need to be triangulated.

VI. Experimental Results

In this section, we demonstrate the efficiency of our DT algorithm, both serial and parallel, for point sets drawn from a uniform distribution, and analyze the complexity of the algorithm. Our experiments were run on a PC with an Intel i7-i2720QM@2.2 GHZ 4-core CPU and 8GB RAM.

For a uniformly distributed point set, the algorithm takes constant average time to compute the Voronoi cell and the Delaunay one-ring for most interior points and $O(\sqrt{n})$ for each point on the convex hull and a very few points near the convex hull. Since the number of points on the convex hull is $O(\log n)$ on the average [33], the overall time complexity of the algorithm is $O((n\text{-}\log n)+ \sqrt{n}\log n) = O(n)$. Our serial implementation confirms this. As for the space complexity, our algorithm needs to build and use the cell/bucket data structure, which takes $O(n)$ space. As discussed in Section V, we only need to output two triangles for each vertices, therefore we need $O(n)$ space to store the results. For each parallel thread, we need to maintain both the candidate Delaunay one-ring polygon and the candidate dual Voronoi cell for current vertex. Let the largest valence of the DT be k and the number of parallel threads be p, then the overall space complexity is $O(n)+O(n)+O(kp)=O(n+kp)$.

Fig. 8 shows the runtime of our DT algorithm with 1 to 4 CPU cores in comparison with Qhull [37], CGAL [35] and Triangle [32] - the best (and most popular) serial algorithms that we are aware of - and GPU-DT [8], for uniformly distributed point sets of different sizes (between 10^5 and 10^6 points). Triangle and CGAL have similar performance, and the serial implementation of our DT algorithm is approximately 2-2.5 times slower.

The DT algorithm was parallelized on a multi-core CPU using OpenMP [34]. The atomic directive is used to build the cell data structure in parallel. Only the point set and corresponding spatial data structure are shared among all the threads. Fig 8. shows that the parallel DT gives an almost perfect speedup over the serial version, thus our parallel implementation on 3 cores and above outperforms Triangle and CGAL, and the same implementation on 4 cores outperforms GPU-DT. Fig 9. shows the speedup using different numbers of CPU cores for different point sets. This particular experiment was run on a Linux server containing two Intel Xeon E5420@2.5 GHZ 4-core CPUs with 32GB RAM. As can be seen, our parallel implementation gives an almost perfect speedup over the serial version for point sets under either uniform or non-uniform distribution.

Our algorithm is designed primarily for uniformly distributed point sets. Although it can also be used for non-uniform distributions, its performance will not be as good. Table I shows the timing of the algorithm run on some point sets having irregular distributions. For point sets having "reasonable" distributions, such as the Gradient and Leaf examples, our algorithm still achieves reasonable performance. However for point sets having extreme distributions, such as the Ring example, our algorithm performs poorly. The reason is that most cells in the underlying grids are either empty or very dense, causing the Delaunay neighbor candidate sets V constructed in Algorithm 1 to be either very large or empty. This significantly damages the load balance, increasing the complexity of Algorithm 1.

VII. Conclusion

We have presented a Local Delaunay lemma which allows to localize the Delaunay triangulation, namely, bound the points in a set which may be Delaunay neighbors of a given point. This localization may be used to design an algorithm to construct the Delaunay triangulation and Voronoi diagram, which may easily be parallelized, since the Delaunay neighbors of any point may be found independently and relatively quickly by process of elimination. Our experiments show that speedup is linear in the number of processors, which means that Delaunay triangulations may be computed arbitrarily quickly by adding computing power.

Future work includes implementation on modern graphic hardware (GPU), extending the algorithm to 3D space, optimizations for point sets with non-uniform distributions and generalization to power diagrams and regular triangulations.

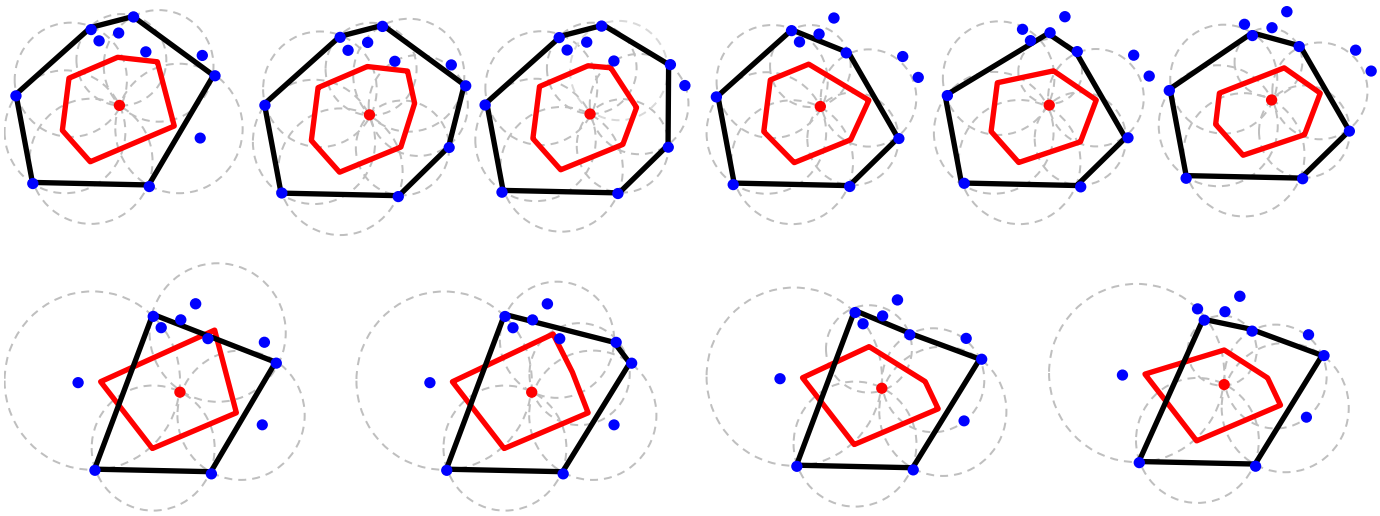


Figure 7. Evolution of the (black) Delaunay one-ring *P* and (red) Voronoi cell *Q* as Algorithms 1 and 2 are running. (Top) Serial version. (Bottom) Parallel version optimized to eliminate redundancy.

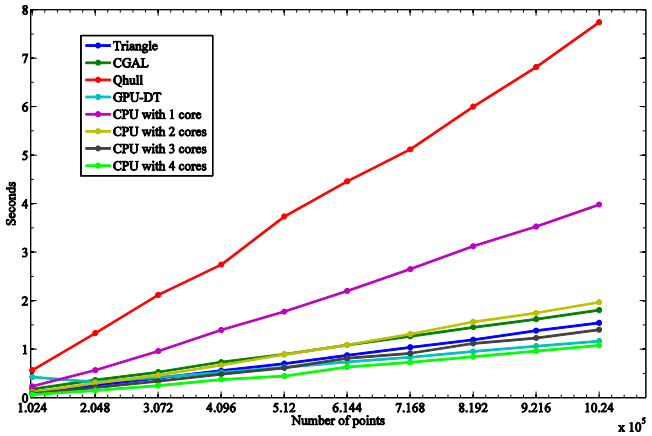


Figure 8. Runtime of parallel DT with different configurations compared to Qhull [37], CGAL [35] and Triangle [32] - the state-of-the-art serial algorithms, and GPU-DT [7].

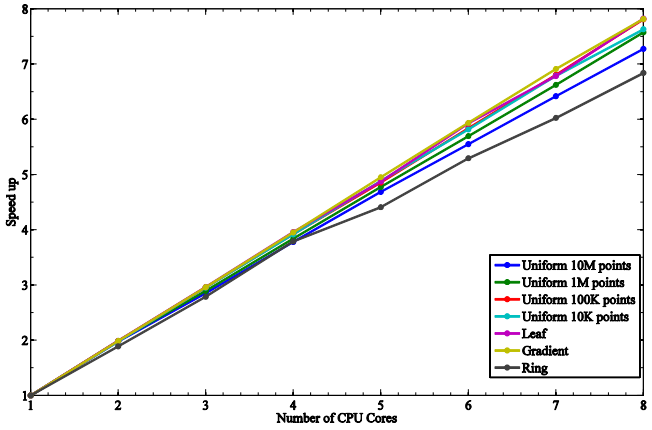
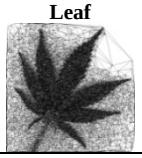


Figure 9. Speedup of parallel DT using multi-core CPU.

TABLE I. RUNTIME (SEC) OF PARALLEL DT ON NON-UNIFORM POINT SETS.

Point set (51,200 points)	**Ring** 	**Gradient** 	**Leaf** 
Parallel DT on 4 cores	0.516	0.036	0.048
CGAL [35]	0.091	0.089	0.091
Triangle [32]	0.055	0.055	0.057

ACKNOWLEDGMENT

This research project was financially supported by the state of Lower-Saxony and the Volkswagen Foundation, Hannover, Germany. R. Chen is partially supported by the Ali Kaufmann postdoctoral fellowship at the Technion.

REFERENCES

[1] J. L. Bentley, B. W. Weide, and A. C. Yao. 1980. Optimal expected-time algorithms for closest point problems. ACM Trans. Math. Softw. 6(4), 563-580.

[2] P. Su and R. L. Scot Drysdale. 1995. A comparison of sequential Delaunay triangulation algorithms. In Proc. SoCG '95. 61-70.

[3] A. Maus. 1984. Delaunay Triangulation and the convex hull of n points in expected linear time. BIT, 24:151-163.

[4] L. J. Guibas, D. E. Knuth, and M. Sharir. 1990. Randomized incremental construction of Delaunay and Voronoi diagrams. In Proc. ICALP '90, 414-431.

[5] R. A. Dwyer. 1987. A faster divide-and-conquer algorithm for constructing Delaunay triangulations. Algorithmica 2, 137-151

[6] S. Fortune. 1986. A sweepline algorithm for Voronoi diagrams. In Proc. SoCG '86, 313-322.

[7] M. Qi, T-T Cao and T-S Tan. 2012. Computing 2D constrained Delaunay triangulation using the GPU. In Proc. Symp. Interactive 3D Graphics and Games (I3D '12). 39-46.

[8] J. Shewchuk. 2005. Star splaying: an algorithm for repairing Delaunay triangulations and convex hulls. In Proc. SoCG ’05:237-246.

[9] A. Maus, J. M. Drange. 2010. All closest neighbors are proper Delaunay edges generalized, and its application to parallel algorithms. Proceedings of Norwegian informatikkonferanse. 1-12.

[10] G. Rong, T-S Tan, T-T Cao, and Stephanus. 2008. Computing two-dimensional Delaunay triangulation using graphics hardware. In Proc. Symp. Interactive 3D Graphics and Games (I3D '08). 89-97.

[11] M. de Berg, O. Cheong, M. van Kreveld and M. Overmars. 2008. Computational Geometry: Algorithms and Applications. Springer-Verlag

[12] S. Lee, C-I Park, and C-M Park. 1997. An improved parallel algorithm for Delaunay triangulation on distributed memory parallel computers. In Proc. Advances in Parallel and Distributed Computing Conf. (APDC '97).

[13] J. Kohout and I. Kolingerov. 2003. Parallel Delaunay triangulation based on circumcircle criterion. In Proc. Spring Conf. Computer Graphics (SCCG '03), 73-81.

[14] D. Reem. 2009. An algorithm for computing Voronoi diagrams of generators in general normed spaces. Proc. International Symp. Voronoi Diagrams in Science and Engineering (ISVD 2009), 144-152.

[15] D. Reem. 2011. On the possibility of simple parallel computing of Voronoi diagrams and Delaunay graphs. Preprint, 2011.

[16] P. Cignoni, C. Montani, R.Perego and R. Scopigno. 1993. Parallel 3D Delaunay triangulation. In Proc. Eurographics 93.

[17] G. E. Blelloch, G. L. Miller, and D. Talmor. 1996. Developing a practical projection-based parallel Delaunay algorithm. In Proc. SoCG ’96, 186-195.

[18] S. Lee, C-I Park, C-M Park. 1996. An efficient parallel algorithm for Delaunay triangulation on distributed memory parallel computers. In Proc. PDPTA '96, 169-177

[19] P. J. Green and R. Sibson. 1977. Computing Dirichlet tessellation in the plane. Comput. J. 21, 168-173

[20] N. M. Amato, M. T. Goodrich and E. A. Ramos. 1994. Parallel algorithms for higher-dimensional convex hulls. In Proc. FOCS ‘94, 683-694

[21] G. E. Blelloch, J. C. Hardwick, G. L. Miller and D. Talmor. 1999. Design and implementation of a practical parallel Delaunay algorithm. Algorithmica 24, 243-269

[22] N. Dadoun and D. G. Kirkpatrick. 1989. Parallel construction of subdivision hierarchies. J. Comput. Syst. Sci. 39, 153-165

[23] H. Meyerhenke. 2005. Constructing higher-order Voronoi diagrams in parallel. EWCG. 123-126

[24] J. H. Reif and S. Sen. 1992. Optimal parallel randomized algorithms for three dimensional convex hulls and related prblems. SIAM J. Comput. 21, 466-485

[25] O. Schwarzkopf. 1989. Parallel computation of discrete Voronoi diagrams. LNCS 349, 193-204

[26] D. A. Spielman, S.-H. Teng and A. Ungor. 2007. Parallel Delaunay refinement: Algorithms and analyses. International Journal of Computational Geometry and Applications 17, 1-30

[27] C. Trefftz and J. Szakas. 2003. Parallel algorithms to find the Voronoi diagrams and the order-k Voronoi diagram. In Proc. PODC ’03.

[28] B. C. Vemuri, R. Varadarajan and N. Mayya. 1992. An efficient expected time parallel algorithm for Voronoi construction. In Proc. SPAA ’92, 392-401.

[29] C. B. Barber. 1993. Computational geometry with imprecise data and arithmetic. Ph.D. Thesis, Princeton.

[30] A. Okabe, B. Boots, K. Sugihara and S.N. Chiu. 2000. Spatial tessellations: Concepts and applications of Voronoi diagrams, 2nd Ed. Wiley Series in Probability and Statistics, John Wiley & Sons Ltd.

[31] D. McCallum and D. Avis. 1979. A linear algorithm for finding the convex hull of a simple polygon. Information Processing Letters, 9(5):201-206.

[32] J. Shewchuk. 1996. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In Applied Computational Geometry: Towards Geometric Engineering, vol. 1182 of LNCS, Springer, 203-222.

[33] S. Har-Peled. 1997. On the expected complexity of random convex hulls. Technical Report.

[34] OpenMP API Specifications for Parallel Programming. http://openmp.org

[35] CGAL - Computational Geometry Algorithms Library. http://www.cgal.org

[36] C. Manuel and T. Monique. 2008. On the computation of 3D periodic triangulations. In Proc. SoCG ’08:222-223.

[37] C. B. Barber, D. P. Dobkin and H. T. Huhdanpaa. The Quickhull algorithm for convex hulls. ACM Trans. on Mathematical Software, 22(4):469-483.