Developing a Practical Projection-Based Parallel Delaunay Algorithm

Guy E. Blelloch Gary L. Miller Dafna Talmor
Computer Science Department
Carnegie Mellon University
{blelloch,glmiller,talmor}@cs.cmu.edu

Abstract

In this paper we are concerned with developing a practical parallel algorithm for Delaunay triangulation that works well on general distributions, particularly those that arise in Scientific Computation. Although there have been many theoretical algorithms for the problem, and some implementations based on bucketing that work well for uniform distributions, there has been little work on implementations for general distributions.

We use the well known reduction of 2D Delaunay triangulation to 3D convex hull of points on a sphere or paraboloid. A variant of the Edelsbrunner and Shi 3D convex hull is used, but for the special case when the point set lies on either a sphere or a paraboloid. Our variant greatly reduces the constant costs from the 3D convex hull algorithm and seems to be a more promising for a practical implementation than other parallel approaches. We have run experiments on the algorithm using a variety of distributions that are motivated by various problems that use Delaunay triangulations. Our experiments show that for these distributions we are within a factor of approximately two in work from the best sequential algorithm.

1 Introduction

Delaunay triangulation along with its dual, the Voronoi Diagram, is an important problem in many domains, including imaging, computer vision, terrain modeling, and meshing for solving PDEs. In many of these domains the triangulation is a bottleneck in the computation time, making it important to develop fast algorithms. There are now many sequential algorithms available for Delaunay triangulation along with efficient implementations. Su and Drysdale [23] present an excellent experimental comparison of several such algorithms. The development of parallel algorithms is not as advanced. As a first step researchers have developed many theoretical parallel algorithms [7, 1, 8, 26, 20, 12]. However, there have been very few efficient implementations, and these few depend on having a uniform distribution of points [16, 24, 22]. Attempts to implement the theoretically good algorithms have met with limited success [22]. There are several obstacles to constructing good practical parallel Delaunay triangulation algorithms: (1) the known parallel solutions are highly irregular and dynamic, (2) they require significant inter-processor communication, and (3) they have very large constants in their asymptotical work analysis even if we ignore communication costs.

Our goal was to develop a practical parallel algorithm that works on general distributions. Since there are many different parallel machines with very different characteristic we tried to design our criteria for success in a machine independent way. Also, since reducing the total work executed by a parallel algorithm (point 3 above) is a prerequisite to getting a practical parallel algorithm, we decided that an initial goal should be to construct a parallel algorithm that does little work beyond the best sequential algorithm. To quantify the constants in the work required by an algorithm, we say algorithm A is α *work-efficient* compared to algorithm B if A performs at most $1/\alpha$ times the number of operation of B. For example, the standard tree-algorithm for prefix sums [15] is 50% work-efficient relative to the sequential prefix sums, since for n values the parallel version requires $2n-2$ operations whereas the sequential requires only $n-1$. The goal is to design parallel algorithms that are as close as possible to 100% work-efficient relative to the best sequential algorithm. Some natural measures of work that we considered included: floating point operations, memory references, comparisons, and data movement. We settled on floating point operations as our primary measure of work because it is machine independent and is reasonably correlated with runtime for this class of algorithms.

In most Delaunay algorithms although the asymptotic work for n points is bound by $O(n \log n)$ the actual work can depend significantly on the distribution of the points. Because of this, when considering α work-efficiency we need to state results relative to particular distributions (this was not necessary for the prefix sums example since the work is only dependent on the data size). For the results to be useful it is necessary to select a "representative" set of distributions. Our selection of point distributions was motivated by scientific domains and includes some highly nonuniform distributions. The four distributions we use are discussed in Section 3.1 and pictured in Figure 6.

We considered a variety of parallel Delaunay algorithms with the goal of getting high α work-efficiency on the data set distributions. The most promising algorithm we considered uses a projection based approach, loosely based on the Edelsbrunner and Shi [11] approach for $3D$ Convex Hulls. Our algorithm does $O(n \log n)$ work and has $O(\log^3 n)$ depth (parallel time) on a CREW PRAM if we use Overmars and

Van Leeuwen's linear-work subroutine for 2-d convex hulls on sorted input [18]. However, by using subroutines that are not known to be theoretically optimal we significantly reduce both the experimental work and depth over our data set. We implemented several variants of our algorithm and ran experiments on our distributions to measure operation counts (floating-point operations) and parallel depth. We compared the operation counts to Dwyer's algorithm [10], which is the best of the sequential Delaunay algorithms studied by Su and Drysdale [23]. Our algorithm is approximately 50% work-efficient relative to Dwyer's if it is run all the way to the end. Furthermore, if the algorithm is used as a coarse partitioner to break a problem into components that can then be solved independently on a set of processors using Dwyer's algorithm, then the algorithm is very close to 100% work efficient. Finally, the point distribution has little effect on the total work, suggesting that our algorithm is reasonably robust across a variety of nonuniform distributions.

1.1 Background and choices

Many of the algorithms for Delaunay triangulation, both parallel and sequential, are based on the divide-and-conquer paradigm. These algorithms can be characterized by the relative costs of the divide and merge phases. An early sequential approach developed by Shamos and Hoey [21] (for Voronoi diagrams) and refined by Guibas and Stolfi [13] (for Delaunay triangulation), is to divide the point set into two subproblems using a median, then to find the Delaunay diagram of each half, and finally to merge the two diagrams. The merge phase does most of the work of the algorithm and runs in $O(n)$ time, so the whole algorithm runs in $O(n \log n)$ time. Unfortunately, these original versions of the merge were highly sequential in nature. Aggarwal et al. [1] first presented a parallel version of the merge phase, which lead to an algorithm with $O(\log^2 n)$ depth. However, this algorithm was significantly more complicated than the sequential version, and was not work efficient—the merge required $O(n \log n)$ work. Goodrich, Cole and O'Dunlaing improved the method making it work efficient [8], but it remains hampered by messy data structures, and as it stands can be ruled out as a promising candidate for implementation.[1]

Reif and Sen [20] developed a randomized parallel divide-and-conquer paradigm, called "polling". They solve the more general 3D Convex-hull problem, which can be used for finding the Delaunay triangulation. In their algorithm a sample of the points is used to split the problem into a set of smaller independent subproblems. For this algorithm, the work is concentrated in the divide phase, and merging simply glues the solutions together. The size of the sample ensures even splitting with high probability. A point can appear in more than one subproblem and so to avoid blow-up trimming techniques are used. A simplified version of this algorithm was considered by Su [22]. His findings show that whereas the paradigm does indeed evenly divide the problem, the expansion factor is close to 6 on all the distributions he considered. This will lead to an algorithm that is at best 1/6 work-efficient, and therefore, pending further improvements, is not a likely candidate for implementation. Dehne et al derive a similar algorithm based on sampling [9]. They show that the algorithm is communication efficient when $n > p^{3+\epsilon}$ (only $O(n/p)$ data is sent and received by each processor). The algorithm is quite complicated, however, and it is unclear what the constants in the work are.

Edelsbrunner and Shi [11] present a 3D convex hull algorithm based on the 2D algorithm of Kirkpatrick and Seidel [14]. The algorithm divides the problem by first using linear programming to find a facet of the 3D convex hull above a splitting point, then using projection onto vertical planes and 2D convex hulls to find two paths of convex-hull edges. These paths are then used to divide the problem into four subproblems, using planar point location to decide for each point which of the subproblems it belongs to. The merge phase again simply glues the solutions. The algorithm takes $O(n \log^2 h)$ time where h is the number of facets in the solution. When applied to Delaunay triangulation the algorithm takes $O(n \log^2 n)$ time since the number of facets will be $\Theta(n)$. This algorithm can be parallelized without much difficulty since all the sub-steps have known parallel solutions, giving a depth (parallel time) of $O(\log^3 n)$ and work of $O(n \log^2 h)$. Ghouse and Goodrich [12] showed how the algorithm could be improved to $O(\log^2 n)$ depth and $O(\min(n \log^2 h, n \log n))$ work using randomization and various additional techniques. The improvement in work makes the algorithm asymptotically work-efficient for Delaunay triangulation. However, these work bounds were based on switching to the Reif and Sen algorithm if the output size was large. Therefore, when used for Delaunay triangulation, the Ghouse and Goodrich algorithm simply reduces to the Reif and Sen algorithm.

1.2 Our algorithm and experiments

The complicated subroutines in the Edelsbrunner and Shi approach, and the fact that it requires $O(n \log^2 n)$ work when applied to Delaunay triangulation, initially seems to rule it out as a reasonable candidate for a parallel implementation. We note, however, that by restricting ourselves to a point set on the surface of a sphere or parabola (sufficient for Delaunay triangulation) the algorithm can be greatly simplified. Under this assumption, we developed an algorithm that only needs a $2D$ convex-hull as a subroutine, removing the need for linear programming and planar point location. Furthermore our algorithm only makes cuts parallel to the x or y axis allowing us to keep the points sorted and use an $O(n)$ work $2D$ convex-hull. These improvements reduce the theoretical work to $O(n \log n)$ and also greatly reduce the constants. This simplified version of the Edelsbrunner and Shi approach seemed a promising candidate for experimentation: it does not suffer from unnecessary duplication, as points are duplicated only when a Delaunay edge is found, and it does not require complicated subroutines, especially if one is willing to compromise on non theoretically-optimal components, as discussed below.

Through alternating rounds of experimentation and algorithmic design, we refined this initial algorithm. We improve the basic algorithm from a practical point of view by using the $2D$ convex-hull algorithm of Chan et al [6]. This algorithm leads to a non optimal theoretical work since it runs in worst case $O(n \log h)$ work (instead of linear), but in practice our experiments showed that it runs in linear work, and has a smaller constant than the provably linear work algorithm. Our final algorithm is not only simple enough to be easily implemented, but is also highly parallel and performs work comparable to efficient sequential algorithms over a wide range of distributions. The algorithm can also be used to partition the problems into processors, and solv-

[1] We note, however, that there certainly could be simplifications that make it easier to implement.

ing each subproblem using the sequential algorithm on each processor, with the work for the coarse partitioning being negligible.

The algorithm was implemented and evaluated in the NESL parallel programming language [4]. The emphasis of our study is on measuring the algorithms in a manner as implementation-independent as possible. Hence, we seek to quantify the number of operations, recursion levels, etc., rather than run times.

The rest of the paper is organized as follows: in section 2 we discuss the algorithm, justify our design choices theoretically, and mention some of the details of the implementation, such as the data structures. In section 3 we describe the test-bed, and present the experimental results.

A preliminary version of this work was presented at the MSI Workshop on Computational Geometry [3]. The earlier version only included the basic algorithm. Here we have optimized the basic algorithm, included an optimized end-game, and a systematic comparison of our optimized code with the best sequential code we know of.

2 Projection-based Delaunay

The goal of this section is to present our algorithm, concentrating on the theoretical motivations for our design choices. Our claim is that these choices lead to a parallel algorithm which is not only efficient, but simple to implement as well, and therefore we also present in some detail the data structures we use.

The basic algorithm uses a divide-and-conquer strategy. Each subproblem is determined by a region $\mathcal{R}$ which is the union of a collection of Delaunay triangles. The region $\mathcal{R}$ is represented by the following information: (1) the polygonal border B of the region and (2) the set of points P of the region, composed of *internal points* and points on the border. Note that the region may not be connected. At each call, we divide the region into two regions using a median line cut of the internal points, and a corresponding path of Delaunay edges. The new path separates Delaunay triangles whose circumcenter is to the left of the median line, from those whose center is to the right of the median line. We then determine the new border of Delaunay edges for each subproblem by merging the old border with the new path. Since we are using a median cut, our algorithm guarantees that the number of internal points is reduced by a factor of at least two at each call. This simple separation is at the heart of our algorithm being efficient. Unlike early divide-and-conquer strategies for Delaunay triangulation which do most of the work when returning from recursive calls [21, 13, 8], this algorithm does all the work before making recursive calls.

To find the separating path (which we call $\mathcal{H}$), we project the points onto a paraboloid whose center is on the median line $\mathcal{L}$, then project the points horizontally onto a vertical plane whose intersection with the x-y plane is $\mathcal{L}$ (see Figure 2). The 2D lower convex hull of those projected points, is the required new border path $\mathcal{H}$. In general, the structure of $\mathcal{H}$ may be more complicated. We discuss this below. Figure 1 gives a more detailed description of the algorithm. Once the subproblem has no more internal points, we move to the end-game strategy to be described later in this section.

Correctness of the median splits: We now show the Delaunay path $\mathcal{H}$ is related to the median line $\mathcal{L}$ in the following sense:

Algorithm: DELAUNAY(P, B)

Input: P, a set of points in R^2,
B, a set of Delaunay edges of P which is the border of a region in R^2 containing P.

Output: The set of Delaunay triangles of P which are contained within B.

Method:

1. If all the points in P are on the boundary B, return END_GAME(B).
2. Find the point q that is the median along the x axis of all internal points (points in P and not on the boundary). Let $\mathcal{L}$ be the line $x = q_x$.
3. Let $P' = \{(p_y - q_y, ||p - q||^2) \mid (p_x, p_y) \in P\}$. These points are derived from projecting the points P onto a $3D$ paraboloid centered at q, and then projecting them onto the vertical plane through the line $\mathcal{L}$.
4. Let $\mathcal{H} =$ LOWER_CONVEX_HULL(P'). $\mathcal{H}$ is a path of Delaunay edges of the set P. Let $P_{\mathcal{H}}$ be the set of points the path $\mathcal{H}$ consists of, and $\bar{\mathcal{H}}$ is the path $\mathcal{H}$ traversed in the opposite direction.
5. Create two subproblems:
   - $P^L = \{p \in P | p \text{ is left of } \mathcal{L}\} \cup P_{\mathcal{H}}$
     $P^R = \{p \in P | p \text{ is right of } \mathcal{L}\} \cup P_{\mathcal{H}}$
   - $B^L =$ BORDER_MERGE$(B, \mathcal{H})$
     $B^R =$ BORDER_MERGE$(B, \bar{\mathcal{H}})$
6. Return DELAUNAY$(P^L, B^L) \cup$ DELAUNAY(P^R, B^R)

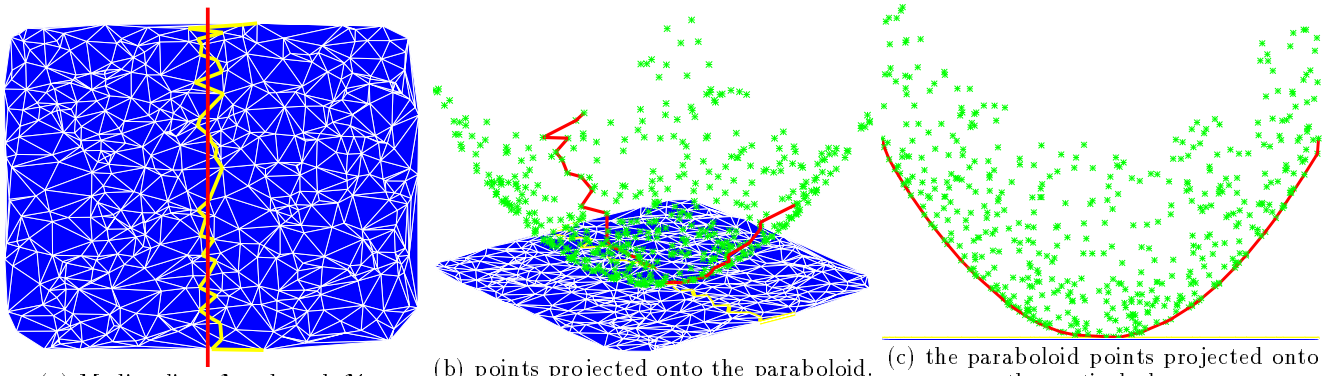
Figure 1: The projection-based parallel algorithm. Initially B is the convex hull of P. The algorithm as described cuts along the x axis, but in general we can switch between x and y cuts, and all our implementations switch on every level. The algorithm uses the three subroutines END_GAME, LOWER_CONVEX_HULL, and BORDER_MERGE, which are described in the text.

Lemma 1 *There is no point in P which is left(right) of the line $\mathcal{L}$, but right(left) of $\mathcal{H}$.*

This implies that we can determine whether a point not on $\mathcal{H}$ is left or right of $\mathcal{H}$ by comparing it against the line $\mathcal{L}$, a simpler computational task.

Proof outline: Lemma 1 is easily seen to be true in a much more general setting. Let Q be a convex body in $\mathbb{R}^3$ with boundary $\bar{Q}$. A point q in $\bar{Q}$ is said to be **light** if it is visible from the x direction, i.e., the ray $\{q + \alpha\hat{x} | \alpha > 0\}$ does not intersect Q, where $\hat{x} = (1, 0, 0)$. We say the point q is **dark** if the ray $\{q - \alpha\hat{x} | \alpha > 0\}$ does not intersect Q. The boundary between light and dark is called the silhouette. In the case when $\bar{Q}$ is our paraboloid then the silhouette is just the image of the line $\mathcal{L}$. In general the **silhouette** is all those points that lie on a supporting plane whose normal is also normal to $\hat{x}$.

If we further assume that the points P are contained in $\bar{Q}$ with convex hull $\bar{P}$ we can define the light, dark, and silhouette points of $\bar{P}$. In the case when $\bar{Q}$ is our paraboloid then the silhouette in general will consist of faces and edges of $\bar{P}$. For ease of exposition, we assume no faces appear on the silhouette. We also assume that the points are in general position, i.e. that all faces are triangular. $\mathcal{H}$ is then a simple



(a) Median line $\mathcal{L}$ and path $\mathcal{H}$. (b) points projected onto the paraboloid. (c) the paraboloid points projected onto the vertical plane.

Figure 2: This figure shows the median line, all points projected on a parabola centered at a point on that line, and the horizontal projection onto the vertical plane through the median line. The result of the lower convex hull in the projected space, $\mathcal{H}$, is shown in highlighted edges on the plane.

path .

Clearly no point $q \in \bar{Q} \cap \bar{P}$ can be light as a point in $\bar{Q}$ but dark as a point in $\bar{P}$ because $\bar{P}$ in contained in Q. This is exactly what Lemma 1 states and thus proves the lemma. □

We now return to the case where the points P lie on the paraboloid. We give a simple characterization of when a face on $\bar{P}$ is light, dark or on the silhouette in term of its circumscribing circle.

Definition 1 *A Delaunay triangle is called a left, right, middle triangle with respect to a line $\mathcal{L}$, if the circumcenter of its Delaunay circle lies left of, right of, or on the line $\mathcal{L}$ respectively*

Lemma 2 *A face F is dark, light, or on the silhouette if and only if its triangle in the plane is left, right, or middle respectively.*

Proof: The face F supporting plane is of the form $ax+by-z=c$ with normal $n=(a,b,-1)$. Now F is dark, light, or on the silhouette if and only if $a<0$, $a>0$, or $a=0$ respectively. The vertical projection of the intersection of this plane, and the paraboloid, is described by $x^2+y^2=ax+by+c$, or by $(x-a/2)^2+(y-a/2)^2=c+(a/2)^2+(b/2)^2$. This is an equation describing a circle whose center is $(a/2,b/2)$ and contains the three points of the triangle. Hence, this is the Delaunay triangle's circumcenter and this circumcenter is simply related to the normal of the corresponding face of the convex hull. □

Therefore the only time that a triangle will cause a face to be on the silhouette is when its circumcenter is on $\mathcal{L}$. For ease of exposition, we assume the following degeneracy condition: No vertical or horizontal line contains both a point and a circumcenter.

Analysis: We now consider the total work and depth of the algorithm. Our bounds are based on an CREW PRAM. The costs depend on the three subroutines END_GAME, LOWER_CONVEX_HULL, and BORDER_MERGE. In this section we briefly describe subroutines that lead to theoretically reasonable bounds, and in the following sections we discuss variations for which we do not know how to prove strong bounds on, but work better on our data sets.

Lemma 3 *Using a parallel version of Overmars and Van Leeuwen's algorithm for the* LOWER_CONVEX_HULL *and Goodrich, Cole and O'Dunlaing's algorithm [8] for the* END_GAME, *our method runs in $O(n \log n)$ work and $O(\log^3 n)$ depth.*

Proof: We first note that since our projections are always on a plane perpendicular to the x or y axis, we can keep our points sorted relative to these axes with linear work (we can keep the rank of each point along both axis and compress these ranks when partitioning). This allows us to use Overmars and Van Leeuwen's linear-work subroutine for 2D convex hulls on sorted input. Since their algorithm uses divide-and-conquer and each divide stage takes $O(\log n)$ sequential time, the full algorithm runs with $O(\log^2 n)$ depth. The other subroutines in the partitioning are the median, projection and BORDER_MERGE These can all be implemented within the bounds of the convex hull (BORDER_MERGE is discussed later in this section). The total cost for partitioning n points is therefore $O(n)$ work and $O(\log^2 n)$ depth.

As discussed earlier, when partitioning a region (P,B) the number of internal points within each partition is at most half as many as the number of internal points in (P,B). The total number of levels of recursion before there are no more internal points is therefore at most $\log n$. Furthermore, the total border size when summed across all instances on a given level of recursion is at most $6n$. This is because $3n$ is a limit on the number of Delaunay edges in the final answer, and each edge can belong to the border of at most two instances (one on each side). Since the work for portioning a region is linear, the total work needed to process each level is $O(n)$ and total work across the levels is $O(n \log n)$. Similarly, the depth is $O(\log^3 n)$.

This is the cost to bring us down to components which have no internal points (just borders). To finish off we need to run the END_GAME. If the border is small (our experiments indicate that the average size is less than 10), it can be solved by any technique, and if the border is large, then the Goodrich, Cole and O'Dunlaing algorithm [8] can be used. □

Border merge: We now discuss how the new Delaunay path is merged with the old border, to form two new borders. The geometric properties of these paths, combined with the data structure we use lead to a simple and elegant $O(n)$ work $O(1)$ time intersection routine.

The set of points is represented using a vector. The border B is represented as an unordered set of triplets.

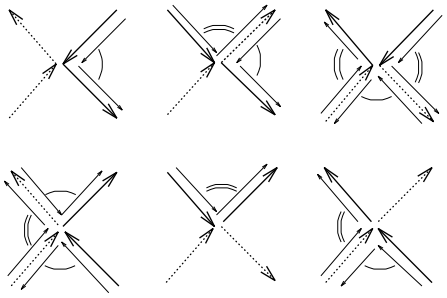


Figure 3: The six cases for merging the new and old border. The old border is in thick lines, and the partitioning path in dotted lines. The convention in the drawings (and the program) is the interior lies left of the border, when proceeding in the direction of the arrows. The resulting two new borders are in thin lines, with the new left border marked with double arcs, and the new right border with a single arc.

Each triplet represents a corner of the border, in the form (i_a, i_m, i_b), where i_m is an index to the middle corner point, i_a is an index to the preceding point on the border, i_b is an index to the following point. Note that the border could have pinch point, i.e. B could contain two triplets with the same middle point (see Figure 4).

The 2D convex-hull algorithm returns a new border path, $\mathcal{H}$, also represented as a set of triplets. $\mathcal{H}$ and B have the following properties:

- No two edges cross since both B and $\mathcal{H}$ are subsets of the set of Delaunay edges of P.
- $\mathcal{H}$ is a simple path anchored at its end points on B.

The border merge can be computed by considering only the local structure of $\mathcal{H}$ and B, that is, by intersecting the pairs of triplets of equal middle index, one representing the shape of the new border near the point, the other representing the shape of the old border. The existence of pinch points in B does not affect this simple procedure, as each triplet belonging to the pinched corner can be intersected independently with the new border. Since the new border has distinct i_m's the number of corner intersections computed is at most the number of corners in the old border.

Figure 3 shows the six different cases for the intersection of the old and new triplets. The core of the border merge is therefore a routine receiving two triplets, identifying which of the six cases they fall into, and returning a set of new left border triplets and right border triplets.

2D convex hull: The 2D convex hull is a central to our algorithm, and is, in fact, the most expensive component. We considered a few candidates for the convex hull algorithm: (1) Overmars' [18], which is $O(n)$ work for sorted points. (2) Kirkpatrick and Seidel's $O(n \log h)$ algorithm [14], and its much simplified form as presented by Chan et al [6]. (3) A simple worst case $O(n^2)$ quickhull algorithm, as in [19, 5].

In [18] it was shown how to compute the convex-hull of pre-sorted points in sequential $O(n)$ work, and the parallel extension is straightforward. Since we can pre-sort the point set, and maintain the ordering through the x and y cuts we perform, using Overmars' will give us a linear runtime for each convex hull invocation. Unfortunately, preliminary experimentation with Overmars' proved it to be quite expensive compared to quickhull. Indeed, we estimate that quickhull (with some additional point pruning heuristics) is of experimental work $O(n)$ and $O(\log n)$ depth over the distributions arising in our Delaunay algorithm and data set.[2]

Using quickhull, our algorithm compared favorably with efficient sequential algorithms, and a preliminary version of this paper contained comparisons based solely on this simple algorithm. Nonetheless, some of our distributions, in particular the Kuzmin distribution, were more costly in the convex-hull phase (seemed to have higher constants) than the others. Quickhull advances by using the furthest point heuristic, and for extremely skewed distributions, the furthest point does not always provide a balanced split. To solve the problem and improve performance, we combined a randomized version of Chan et al's algorithm with the quickhull algorithm. Running the quickhull for a few levels makes quick progress when possible and prunes out a large fraction of the points. Switching to the Chan et al algorithm, guarantees balanced cuts which might be necessary for point sets that did not yield to quickhull.

The end-game: Once the subproblems have no internal points, we switch to the end-game strategy. The basic form of the end-game is quicksort in flavor, since at each iteration a random point is picked, which then gives a Delaunay edge that partitions the border in two. As with quicksort, the partition does not guarantee that the subproblems are balanced. For the end-game we first need to decompose the border into a set of simple cycles, since the borders can be disjoint and have pinch points (see Figure 4). The border can be split by joining the corners into a linked list and using a list-ranking algorithm (we currently use pointer jumping). After this step each subproblem is a simple cycle, represented by an ordered list of point indices.

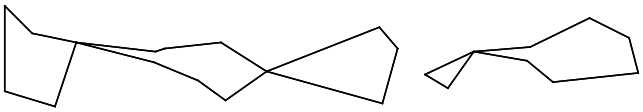
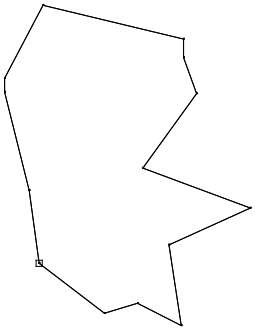


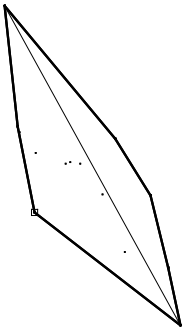
Figure 4: One of the border subproblems, created by running the basic algorithm until all the points are on the border

The core of the end-game is to find one new Delaunay edge, and use this edge to split the cycle into two new simple cycles. We find this edge using an efficient $O(n)$ work $O(1)$ time routine. We use a well known duality: the set of Delaunay neighbors of a point q is equal to the set of points on the 2D convex hull after inverting the points around q. The inversion takes the following form: $\dot{P} = \{(p-q)/||p-q|| \quad | \quad p \in P\}$. Since we are looking for one Delaunay neighbor only, rather than the full set, we do not need to compute the convex hull, but rather just pick an extremal point. For example, if q belongs to a convex corner (p_a, q, p_b), we can draw a line between p_a and p_b and find the furthest point p_d from that line, which will be on the convex hull (See Figure 5). If p_d is either p_a or p_b, then then (p_a, p_b) must be a new Delaunay edge, otherwise (q, p_d) is a new Delaunay edge. For concave corners we can proceed similarly.

[2] We note that this work may seem to violate the lower bound of $O(n \log h)$ for finding a 2-d convex hull [14], given that our output sizes h are typically on the order of $O(\sqrt{n})$. This bound, however, is a worst-case analysis based on artificial distributions.



(a) A border piece. The point highlighted, q, is the one singled for expansion in the end-game.



(b) The same set of points after inversion around q.

Figure 5: Starting in (a) with a border subproblem, we look for a new Delaunay edge of a point q. We do that by using inversion around q to move to the dual problem of finding convex-hull edges, as in (b), drawn in thick lines. The new Delaunay edge we pick is to the point farthest from the line between the points preceding and following q, shown in a thin line.

Comparison to Edelsbrunner and Shi Here we explain how our algorithm differs from the original algorithm presented by Edelsbrunner and Shi. We partition the points and border by finding a median, computing a 2D convex hull of a simple projection of the points, and then using some simple local operations for merging the borders. Since Edelsbrunner and Shi are solving the more general problem of 3D convex hull, they have to (1) find a 4-partition using 2 intersecting lines in the XY plane, (2) use linear programming to find the face of the convex-hull above the intersection, (3) compute two convex hulls of projections of the points (4), Merge the borders, and (5) use point location to determine which points belong to which partition. Each of these steps is more complex than ours.

We can get away with the simpler algorithm since in our projection in 3D, all points lie on a surface of a parabola. This allows us to easily find a face of the convex-hull (we just use the median point), and to simply partition the points using a median line. Furthermore, our partitions can all be made parallel to the x or y axis, which allows us to keep our points sorted along the cut for the convex-hull. With this we can use a linear work algorithm for the convex hull, as discussed. This is not possible with the Edelsbrunner and Shi algorithms since in their algorithm it is necessary to make cuts in arbitrary directions in order to get the 4-partition. On the down side, our partition is not as good as the 4-partition of Edelsbrunner and Shi since it only guarantees that the internal points are well partitioned (the border could be badly partitioned). This means we have to switch algorithms when no internal points remain. Our experiments show, however, that for all our distributions the average size of components when switching is less than 10, and the maximum size is rarely more than 50 (this assumes we are alternating between x and y cuts).

We note that proving the sufficiency of the median test for the divide phase was the insight that motivated our choice of the projection-based algorithm for implementation. This work was also motivated by the theoretical algorithms presented in [17].

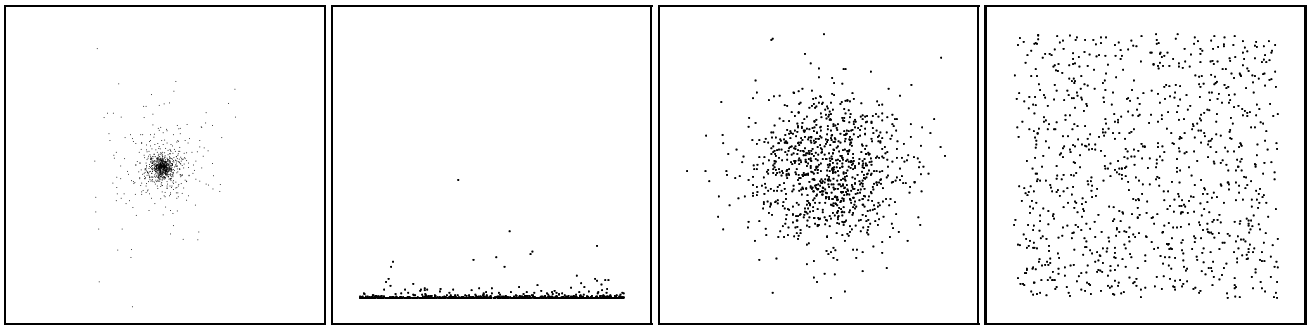
3 Experiments

To evaluate our algorithm, we implemented it and instrumented the implementation to measure several quantities. The quantities we measure include the total work (floating-point operations), the parallel depth, the number and sizes of subproblems on each level, and the relative work of the different subroutines. The total work is used to determine how work-efficient the algorithm is compared with a good sequential algorithm, and the ratio of work to parallel-depth is used to estimate the parallelism available in the algorithm. We use the other measures to better understand how the algorithm is effected by the distributions, and how well our heuristic for splitting works. We have also used the measures extensively to improve our algorithm and have been able to improve our convex-hull by a factor of 3 over our initial naive implementation.

We measured the different quantities over four data set distributions, for sizes varying from 2^{10} to 2^{17}. For each distribution, each data set, and each size we ran five instances of the distribution (seeded with different random numbers) The results are presented using median values and intervals over these experiments, unless otherwise stated. Our intervals are defined by the outlying points (the minimum and the maximum over the relevant measurements). We defined a floating point operation as a floating point comparison or arithmetic operation, though our instrumentation contains a break down of the operations into the different classes. The data files are available upon request if a different definition of work is of interest. Although floating-point operations certainly do not account for all costs in an algorithm they have the important advantage of being machine independent (at least for machines that implement the standard IEEE float instructions) and seem to have a strong correlation to running time [23].

3.1 Data Distributions

The design of the data set is always of great importance for the experimentation and evaluation. Our goal is to test our algorithm on distributions that are non uniform, and yet representative of real-world problems. We therefore picked distributions from different domains, such as the distribution of stars in a flat galaxy (Kuzmin) or point sets originating from the mesh generation domain, where Delaunay triangulation plays an important role. Many of these distributions fall under the class of Lipschitz distributions—that is, the density function's derivative is bounded by a constant. This allows for arbitrary refinements of the triangles



(a) The Kuzmin distribution. (b) The line distribution. (c) The normal distribution. (d) The uniform distribution.

Figure 6: Our test-suite of distributions for 1000 points. For the Kuzmin distribution, the figure shows a small region in the center of the distribution (otherwise almost all points appear to be at one point at the center).

sizes, but allows for certain probabilistic restrictions on the aspect ratios of the triangles and number of neighbors of points. An analysis of the expected extremes for the uniform distribution can be found in a paper by Bern, Eppstein and Yao [2]. Lipschitz distributions can be interesting enough to defy standard uniform distribution handling techniques such as bucketing. Our distributions are shown in Figure 6 and summarized here.

- **The Kuzmin distribution:** this distribution is used by astrophysicists to model the distribution of star clusters in flat galaxy formations [25]. This is a radially symmetric distribution, whose density falls quickly as r increases, providing us with a good example of convergence to a point. The accumulative probability function, as a function of the radius r is:

$$M(r) = 1 - \frac{1}{\sqrt{1+r^2}}$$

- **Line singularity:** this distribution was defined by us as an example of a distribution that has a convergence area (points very densely distributed along a line segment) and yet the density function is Lipschitz. We define this probability distribution using a constant $b \geq 0$, and a transformation from the uniform distribution. Let u and v be two independent, uniform random variables, then the transformation is:

$$(x, y) = (\frac{b}{u - bu + b}, v)$$

  in our experiments, we set $b = 0.001$.

- **Normal distribution:** points (x, y) such that x and y are independent samples from the normal distribution. The normal distribution is also radially symmetric, but its density at the center is much smaller than in the Kuzmin distribution.

- **Uniform distribution:** points picked at random in a unit square. It is important to include a uniform distribution in our experiments for two reasons: to contrast the behavior of the algorithm over the uniform distribution and the non-uniform distributions, and also to form common ground for comparison with other relevant work.

3.2 Experimental Results

Work: To estimate the work of our algorithm, we compare the floating-point operation counts to Dwyer's sequential algorithm [10]. This algorithm is a variation of Guibas and Stolfi's divide-and-conquer algorithm, which is careful about the cuts in the divide-and-conquer phase so that for quasi-uniform distributions the expected run time is $O(n \log\log n)$, and on the rest is at least as efficient as the original algorithm. In a recent paper, Su and Drysdale [23] have experimented with a variety of sequential Delaunay algorithm, and Dwyer's algorithm performed as well or better than all others across a variety of distributions. It is therefore a good target to compare to. We use the same code for Dwyer's algorithm as used in [23].

Figure 7 shows a comparison of float-counts of our algorithm and Dwyer's for all our distributions. The values are median value for each size, the error intervals are too small to be noticeable. For the uniform, normal and Kuzmin distributions our algorithm is about 50% work efficient relative to Dwyer's (does a factor of about 2 more work). For the line distribution, Dwyer's cuts bring less savings, and our algorithm is close to 100% work efficient.

Our algorithm performs almost uniformly on the line, normal and uniform distribution, but the Kuzmin distribution is slightly more expensive. To understand the variation among the distributions we studied the breakdown of the work into the components of the algorithm—finding the median, computing 2D convex hulls, intersecting borders, and the end-game. Figure 8 shows the breakdown of float counts for a representative example of size 2^{17}. These represent the total number of floating-point operations used by the components across the full algorithm. As the figure shows, the work for all but the convex-hull is approximately the same across distributions (varies by less than 10%). For the convex-hull, the Kuzmin data set requires about 25% more work than the uniform distribution. Our experiments show that this is because after the paraboloid lifting and projection, the convex-hull removes fewer points in early steps and therefore requires more work. In an earlier version of our implementation in which we used a simple quickhull instead of the balanced algorithm [6], Kuzmin was 75% more expensive than the others.

Depth: We now consider the depth (parallel time) of the algorithm. The depth was determined by measuring the total depth of the call tree (always taking the maximum

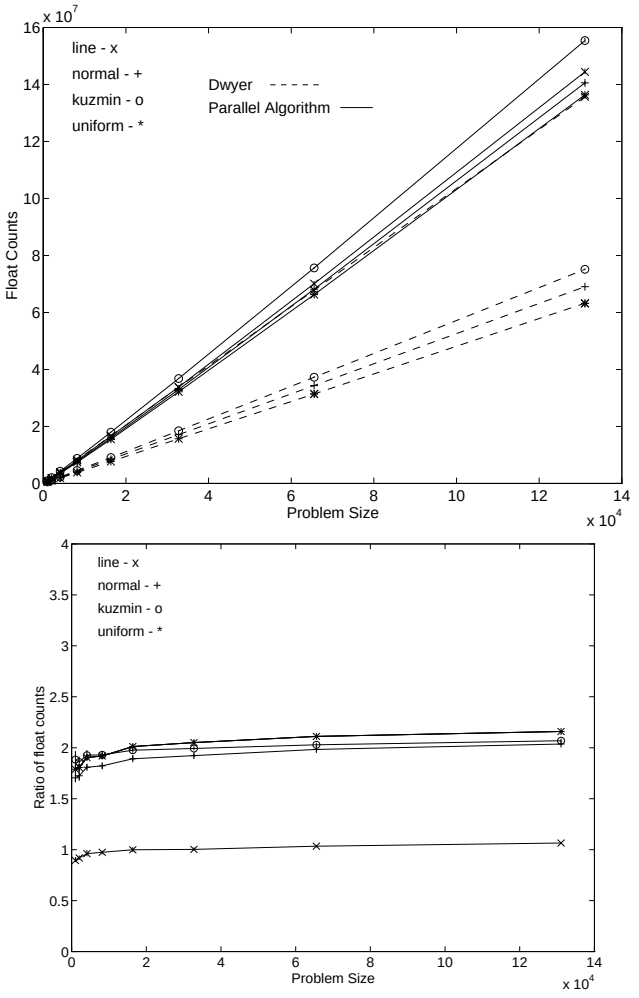


Figure 7: A comparison of the parallel algorithm vs. Dwyer's algorithm in terms of the number of float operation performed, for varying distributions and problem sizes.

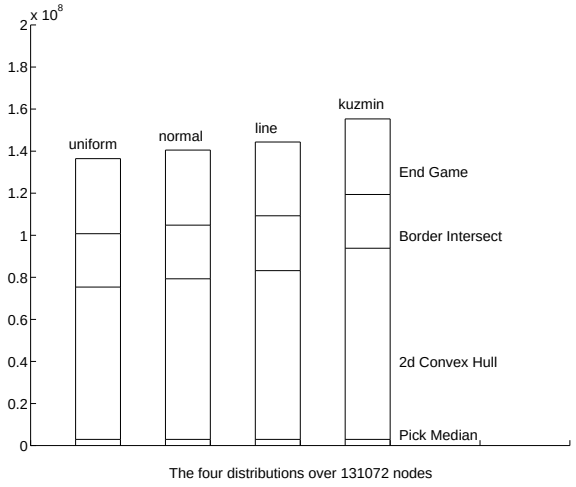


Figure 8: Float counts partitioned according to the different algorithmic parts, for each distribution.

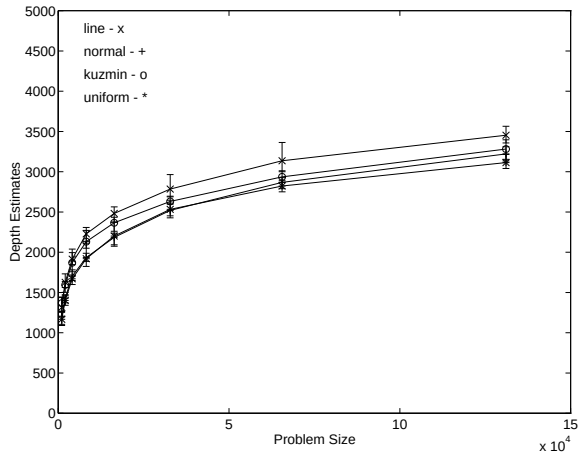


Figure 9: The depth of the algorithm for the different distributions and problem sizes.

depth over parallel calls, and summing depth over sequential calls). Figure 3.2 shows the depth as a function of problem size for the four distributions. As can be seen, the depth is also not strongly effected by the distribution. As some of the constants in the depth calculation for the different parts are estimated, the figure should be studied for the trends it shows.

Effectiveness of our divide: To investigate the divide-and-conquer behavior of the algorithm and how well our heuristic for separating point sets works, we look at the size of the maximal subproblem at each level (see Figure 10). A parallel algorithm should quickly and geometrically reduce the maximal problem size. As mentioned earlier, the theory tells us that the number of internal points is decreased by a factor of at least two every level, but provides no guarantees for the number of points on the border of each subproblem. Figure 10 shows the total size including the border. As can be seen, the size goes down uniformly. The discontinuity at around level twenty represents the move from the basic algorithm to the end game strategy.

Another estimate of work: A different kind of work estimate, which abstracts away from the 2D convex-hull costs, can be obtained by looking at the sum of the problem sizes at each level, see Figure 11. Finding the median cut and intersecting the new border with old perform work linear in this quantity.

This figure also provides a way to estimate the quality of the cuts we are performing, since the points expanded by each cut are duplicated, and thus counted twice in the sum of problem sizes. For example, a simple median cut of the line distribution taken in the wrong direction can cause more points to be exposed as border points (and thus duplicated) early on. The line distribution does indeed seem to be slightly more expensive, but the agreement on the work among the distributions is surprisingly good. The curves corresponding to the end game work are even more similar, though shifted from each other. The shift occurs because each distribution reached termination of the basic algorithm at a different iteration.

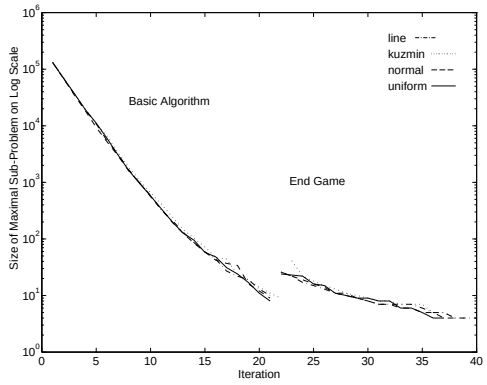


Figure 10: The maximal piece size as a function of iteration for our algorithm on a representative example of 2^{17} points from each distribution. The piece size includes both internal points and border points. The graph shows that our separator and our endgame work quite well for all the distributions.

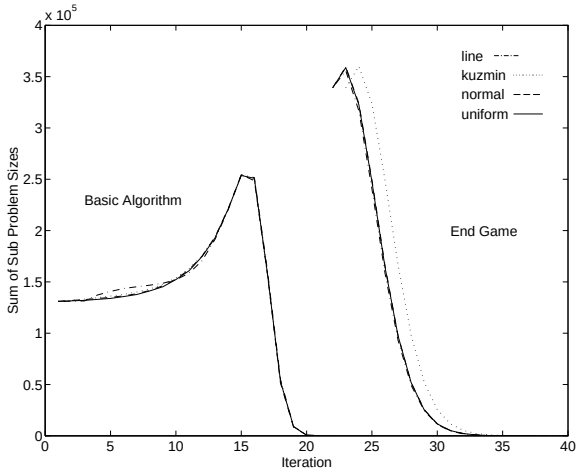


Figure 11: Sum of the sizes of all the subproblems per iteration, on a representative 2^{17} points example from each distribution.

The algorithm as a coarse partitioner: Finally, we provide support to the fact that the algorithm can be used very efficiently to divide the problem into a number of subproblems, and then switch to a sequential algorithm. To do that, we present the accumulative float counts (work) per iteration on 2^{17} points, versus the cost of the sequential algorithm, see Figure 12. In general, at level i the number of subproblems is 2^i.

4 Concluding Remarks

In this paper we described the development of an experimentally work efficient parallel Delaunay triangulation algorithm. Our starting point was a divide and conquer algorithm, resembling Edelsbrunner and Shi's [11] 3D convex-hull approach specialized to the Delaunay triangulation case. Our simpler algorithm relies on two subroutines: to compute lower 2D convex-hull, and to compute the Delaunay triangulation of simple edge cycles. Depending on our choices for

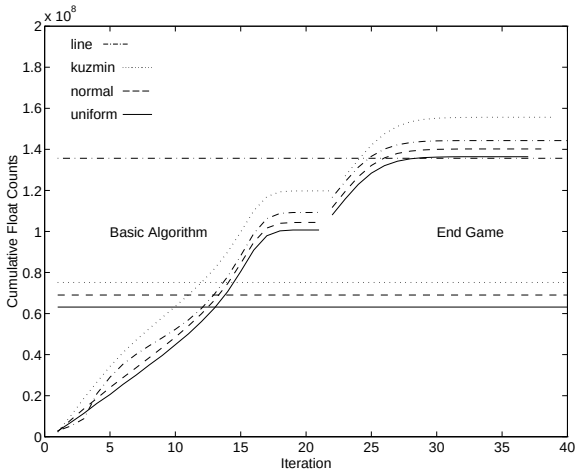


Figure 12: The cumulative work in float counts per iteration, for a representative 2^{17} points example from each distribution. The idea is to see how much work is it to divide the problem into 2^i problems in level i, versus the median cost of the sequential algorithm.

these two subroutines, we can obtain either an asymptotycally optimal algorithm, or an experimentally work efficent algorithm. Investigating the trade-offs between the two is the theme of this research.

The main results are:

- Our experimentation demonstrates that the constants of our algorithm are small enough for the measured work to be competitive with the best sequential algorithm we know of. In particular, depending on the point distribution we range from being .4 to .9 work-efficient relative to Dwyer's algorithm.
- We provide a variant of our algorithm which is of $O(n \log n)$ work. This is contrast with the upper bound of $O(n \log^2 n)$, given by the Edelsbrunner and Shi [11] for their algorithm applied to the Delaunay triangulation case.
- Our algorithm is highly parallel. The ratio of work to parallel time is approximately 10^4 for 10^5 points. For a machine with a modest number of processors (*eg.* 100), this should give plenty of options of how to parallelize the algorithm to reduce communication.
- Our algorithm is well suited for partitioning a data set into a set of subproblems that each can then be solved independently, since it is a divide and conquer algorithm with a trivial conquer phase, consisting of simply gathering the solutions of the smaller subproblems, much like quicksort. We show partitioning into a small set of subproblems does not require much work, and our algorithm used as a coarse partitioner would be close to 100%-efficient.

Our experiments and measurements showed that the algorithm performs almost equally well across a set of very different and highly non-uniform distributions. The variations that were observed across the distributions can be attributed to the performance of the 2D convex-hull subroutine used. This provides strong motivation for a study of different parallel 2D convex-hull implementations.

5 Acknowledgments

This research was sponsored in part by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. It was also supported in part by NSF, under Grant numbers CCR-9258525 (Blelloch) and CCR-9505472 (Miller and Talmor). The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Wright Laboratory or the U. S. Government.

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, 3(3):293–327, 1988.

[2] Marshall Bern, David Eppstein, and Frances Yao. The expected extremes in a Delaunay triangulation. *Inter. J. of Computational Geometry and Appl.*, 1(1):79–91, 1991.

[3] G. Blelloch, G.L. Miller, and D. Talmor. Parallel Delaunay triangulation implementation. In *MSI workshop on Computational geometry*, Cornell, Oct 1994.

[4] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[5] Guy E. Blelloch and James J. Little. Parallel solutions to geometric problems in the scan model of computation. *Journal of Computer and System Sciences*, 48(1):90–115, February 1994.

[6] Timothy M. Y. Chan, Jack Snoeyink, and Chee-Keng Yap. Output-sensitive construction of polytopes in four dimensions and clipped voronoi diagrams in three. In *Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 282–291. ACM-SIAM, 1995.

[7] A. Chow. *Parallel Algorithms for Computational Geometry.* PhD thesis, University of Illinois, 1980.

[8] R. Cole, M. T. Goodrich, and C. O'Dunlaing. Merging free trees in parallel for efficient voronoi diagram construction. In *International Colloquium on Automata, Languages and Programming*, pages 32–45, July 1990.

[9] F. Dehne, X. Deng, P. Dymond, A. Fabri, and A. Khokhar. A randomized parallel 3d convex hull algorithm for coarse grained multicomputers. In *Proc. ACM Sympos. Parallel Algorithms Architectures.*, pages 27–33, 1995.

[10] R.A. Dwyer. A simple divide-and-conquer algorithm for constructing delaunay triangulations in $O(n \log\log n)$ expected time. In *2nd Symposium on Computational Geometry*, pages 276–284, 1986.

[11] Herbert Edelsbrunner and Weiping Shi. An $O(n \log^2 h)$ time algorithm for the three-dimensionaal convex hull problem. *SIAM J. Computing*, 20:259–277, 1991.

[12] M. Ghouse and M. T. Goodrich. In-place techniques for parallel convex hull algorithms. In *Proc. 3rd ACM Sympos. Parallel Algorithms Architect.*, pages 192–203, 1991.

[13] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, 1985.

[14] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM J. Comput.*, 15:287–299, 1986.

[15] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the Association for Computing Machinery*, 27(4):831–838, October 1980.

[16] M. L. Merriam. Parallel implementation of an algorithm for delaunay triangulation. In *First European Computational Fluid Dynamics Conference*, pages 907–912, September 1992.

[17] Gary L. Miller, Dafna Talmor, Shang-Hua Teng, and Noel Walkington. A Delaunay based numerical method for three dimensions: generation, formulation and partition. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, 1995.

[18] Mark H. Overmars and Jan Van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.

[19] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry An Introduction.* Texts and Monographs in Computer Science. Springer-Verlag, 1985.

[20] J. Reif and S. Sen. Polling: A new randomized sampling technique for computational geometry. In *Proceedings of the 21th Annual ACM Symposium on Theory of Computing*, 1989.

[21] M.I. Shamos and D. Hoey. Closest-point problems. In *16th Annual Symposium on Foundations of Computer Science*, pages 151–162. IEEE, Oct 1975.

[22] Peter Su. *Efficient parallel algorithms for closest point problems.* PhD thesis, Dartmouth College, 1994. PCS-TR94-238.

[23] Peter Su and Robert L. Drysdale. A comparison of sequential delaunay triangulation algorithms. In *Proceedings of the eleventh Annual Symposium on Computational Geometry*, pages 61–70, Vancouver, June 1995. ACM.

[24] Y. Ansel Teng, Francis Sullivan, Isabel Beichl, and Enrico Puppo. A data-parallel algorithm for three-dimensional Delaunay triangulation and its implementation. In *SuperComputing 93*, pages 112–121. ACM, 1993.

[25] A. Toomre. On the distribution of matter within highly flattened galaxies. *The astrophysical journal*, 138:385–392, 1963.

[26] B. C. Vemuri, R. Varadarajan, and N. Mayya. An efficient expected time parallel algorithm for Voronoi construction. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 392–400, June 1992.