Algorithmica (1999) 24: 243–269

Algorithmica

Design and Implementation of a Practical Parallel Delaunay Algorithm[1]

G. E. Blelloch,[2] J. C. Hardwick,[3] G. L. Miller,[2] and D. Talmor[4]

Abstract. This paper describes the design and implementation of a practical parallel algorithm for Delaunay triangulation that works well on general distributions. Although there have been many theoretical parallel 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 find the 3D convex hull of points on a paraboloid. Based on this reduction we developed a variant of the Edelsbrunner and Shi 3D convex hull algorithm, specialized for the case when the point set lies on a paraboloid. This simplification reduces the work required by the algorithm (number of operations) from $O(n \log^2 n)$ to $O(n \log n)$. The depth (parallel time) is $O(\log^3 n)$ on a CREW PRAM. The algorithm is simpler than previous $O(n \log n)$ work parallel algorithms leading to smaller constants.

Initial experiments using a variety of distributions showed that our parallel algorithm was within a factor of 2 in work from the best sequential algorithm. Based on these promising results, the algorithm was implemented using C and an MPI-based toolkit. Compared with previous work, the resulting implementation achieves significantly better speedups over good sequential code, does not assume a uniform distribution of points, and is widely portable due to its use of MPI as a communication mechanism. Results are presented for the IBM SP2, Cray T3D, SGI Power Challenge, and DEC AlphaCluster.

Key Words. Delaunay triangulation, Parallel algorithms, Algorithm experimentation, Parallel implementation.

1. Introduction.

A Delaunay triangulation in R^2 is the triangulation of a set S of points such that there are no elements of S within the circumcircle of any triangle. Delaunay triangulation—along with its dual, the Voronoi Diagram—is an important problem in many domains, including pattern recognition, terrain modeling, and mesh generation for the solution of partial differential equations. In many of these domains the triangulation is a bottleneck in the overall computation, making it important to develop fast algorithms. As a consequence, there are many sequential algorithms available for Delaunay triangulation, along with efficient implementations. Su and Drysdale [1] present an excellent experimental comparison of several such algorithms. Since these algorithms are time and memory intensive, parallel implementations are important both for improved performance and to allow the solution of problems that are too large for sequential machines. However, although several parallel algorithms for Delaunay triangulation have been described [2]–[7], practical implementations have been slower to appear, and are

[1] G. L. Miller and D. Talmor were supported in part by NSF Grant CCR-9505472.
[2] Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, USA. {blelloch, glmiller}@cs.cmu.edu.
[3] Microsoft Research Ltd, 1 Guildhall Street, Cambridge CB2 3NH, England. jch@microsoft.com.
[4] CADSI, 3150 Almaden Expwy Suite 104, San Jose, CA 95118, USA. dafna@cadsi.com.

Received June 1, 1997; revised March 10, 1998. Communicated by F. Dehne.

mostly specialized for uniform distributions [8]–[11]. One reason is that the dynamic nature of the problem can result in significant interprocessor communication. This is particularly problematic for nonuniform distributions. A second problem is that the parallel algorithms are typically much more complex than their sequential counterparts. This added complexity results in low *parallel efficiency*; that is, the algorithms achieve only a small fraction of the perfect speedup over efficient sequential code running on one processor. Because of these problems no previous implementation that we know of has achieved reasonable speedup over good sequential algorithms when used on nonuniform distributions.

Our goal was to develop a parallel Delaunay algorithm that is efficient both in theory and in practice, and works well for general distributions. In theory we wanted an algorithm that for n points runs in polylogarithmic depth (parallel time) and optimal $O(n \log n)$ work. We were not concerned with achieving optimal depth since no machines now or in the foreseeable future will have enough processors to require such parallelism. In practice we wanted an algorithm that performs well compared with the best sequential algorithms over a variety of distributions, both uniform and nonuniform. We considered two measures of efficiency. The first was to compare the total work done to that done by the best sequential algorithm. We quantify the constants in the work required by a parallel algorithm relative to the best sequential algorithm using the notion of α *work-efficiency*. We say that algorithm A is α *work-efficient* compared with algorithm B if A performs at most $1/\alpha$ times the number of operation of B. An ideal parallel algorithm is 100% work-efficient relative to the best sequential algorithm. We use floating-point operations as a measure of work—this has the desirable property that it is machine independent. The second measure of efficiency is to measure actual speedup over the best sequential algorithm on a range of machine architectures and sizes.

Based on these criteria we considered a variety of parallel Delaunay algorithms. The one eventually chosen uses a divide-and-conquer projection-based approach, based loosely on the Edelsbrunner and Shi [12] algorithm for 3D convex hulls. Our algorithm does $O(n \log n)$ work and has $O(\log^3 n)$ depth on a CREW PRAM. From a practical point of view it has considerably simpler subroutines for dividing and merging subproblems than previous techniques, and its performance has little dependence on data distribution. Furthermore, it is well suited as a coarse-grained partitioner, which splits up the points evenly into regions until there are as many region as processors, at which point a sequential algorithm can be used. Our final implementation is based on this idea.

Our experiments were divided into two parts. A prototyping phase used the parallel programming language NESL [13] to experiment with algorithm variants, and to measure their work-efficiency. An optimized coarse-grained implementation of the final algorithm was then written in C and a toolkit based on MPI [14], and was compared with the best existing sequential implementation. For our measurements in both sets of experiments we selected a set of four data distributions which are motivated by scientific domains and include some highly nonuniform distributions. The four distributions we use are discussed in Section 3.1 and pictured in Figure 8.

Our NESL experiments show that the algorithm is 45% work-efficient or better for all four distributions and over a range of problem sizes when applied all the way to the end. This is relative to Dwyer's algorithm, which is the best of the sequential Delaunay algorithms studied by Su and Drysdale [1]. Figure 1 shows a comparison of floating-point

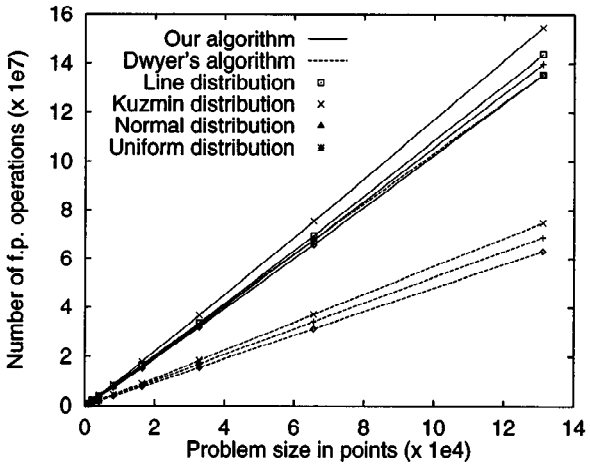


Fig. 1. A comparison of our parallel algorithm versus Dwyer's algorithm, in terms of the number of floating-point operation performed for four input distributions.

operations performed by our algorithm and Dwyer's algorithm for the four distributions (see Section 3.2 for full results). On the highly nonuniform line distribution, Dwyer's cuts bring less savings, and our algorithm is close to 100% work-efficient.

The MPI and C implementation uses our basic algorithm as a coarse-grained partitioner to break the problem into one region per processor. Our experiments show that this approach achieves good speedup on a range of architectures, including the IBM SP2, Cray T3D, SGI Power Challenge, and DEC AlphaCluster. As an example, Figure 2 shows running times for up to 64 processors on the Cray T3D (see Section 3.3 for full results). We note that the implementation not only gives speedups that are within a factor of 2 of optimal (compared with a good sequential algorithm), but allow us to solve much larger problems because of the additional memory that is available across multiple processors. Our algorithm never requires that all the data reside on one processor—both the input and output are evenly distributed across the processors, as are all intermediate results.

1.1. *Background and Choices.*

Theoretical Algorithms. Many of the efficient algorithms for Delaunay triangulation, sequential and parallel, 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 for Voronoi diagrams by Shamos and Hoey [15] and refined for Delaunay triangulation by Guibas and Stolfi [16], 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. [3] first presented a parallel version of the merge phase, which lead

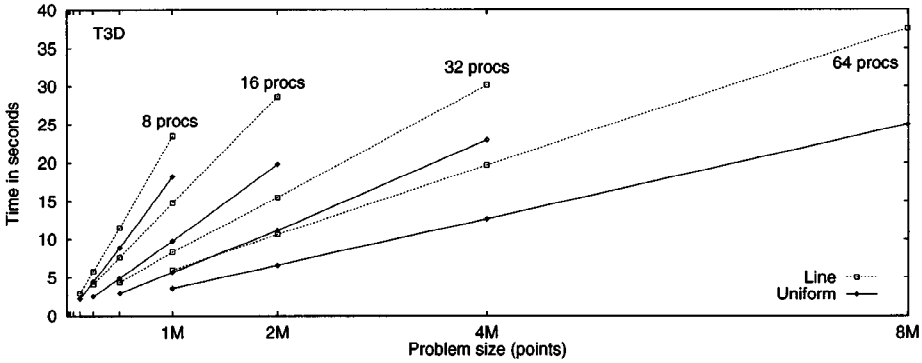


Fig. 2. Scalability of the MPI and C implementation of our algorithm on the Cray T3D, showing the time to triangulate 16k–128k points per processor for a range of machine sizes. For clarity, only the fastest (uniform) and slowest (line) distributions are shown.

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. Cole et al. [4] improved the method and made it work-efficient on the CREW PRAM using the same depth. The algorithm, however, remains hampered by messy data structures, and as it stands can be ruled out as a promising candidate for implementation. We note, however, that there certainly could be simplifications that make it easier to implement.

Reif and Sen [6] 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. The algorithm uses a sample of the points to to split the problem into a set of smaller independent subproblems. The size of the sample ensures even splitting with high probability. The work of the algorithm is concentrated in the divide phase, and merging simply glues the solutions together. Since a point can appear in more than one subproblem, trimming techniques are used to avoid blow-up. A simplified version of this algorithm was considered by Su [11]. He showed that whereas sampling 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 one-sixth work-efficient, and therefore, pending further improvements, is not a likely candidate for implementation. Dehne et al. [17] derive a similar algorithm based on sampling. They show that the algorithm is communication-efficient when $n > p^{3+\varepsilon}$ (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 [12] present a 3D convex hull algorithm based on the 2D algorithm of Kirkpatrick and Seidel [18]. 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 together. The algorithm takes $O(n \log^2 h)$ time, where h

is the number of facets in the solution. When applied to Delaunay triangulation, however, 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 substeps 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 [7] 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.

Implementations. Most of the parallel implementations of Delaunay triangulation use decomposition techniques such as bucketing [8]–[11] or striping [19]. These techniques have the advantage that they can reduce communication by allowing the algorithm to partition points quickly into one bucket (or stripe) per processor and then use sequential techniques within the bucket. However, the algorithms rely on the input dataset having a uniform spatial distribution of points in order to avoid load imbalances between processors. Their performance on nonuniform distributions more characteristic of real-world problems can be significantly worse than on uniform distributions. For example, the 3D algorithm by Teng et al. [10] was up to five times slower on nonuniform distributions than on uniform ones (on a 32-processor CM-5), while the 3D algorithm by Cignoni et al. [9] was up to ten times slower on nonuniform distributions than on uniform ones (on a 128-processor nCUBE).

Even given the limitation to uniform distributions, the speedups of these algorithms over efficient sequential code has not been good. Of the algorithms that quote such speedups the 2D algorithm by Su [11] achieved speedup factors of 3.5–5.5 on a 32-processor KSR-1, for a parallel efficiency of 11–17%, while the 3D algorithm by Merriam [8] achieved speedup factors of 6–20 on a 128-processor Intel Gamma, for a parallel efficiency of 5–16%. Both of these results were for uniform distributions. The 2D algorithm by Chew et al. [20], which can handle nonuniform distributions and solves the more general problem of constrained Delaunay triangulation in a meshing algorithm, achieves speedup factors of 3 on an 8-processor SP2, for a parallel efficiency of 38%. However, this algorithm currently requires that the boundaries between processors be created by hand.

1.2. *Our Algorithm and Experiments*. Previous results do not look promising for developing practical Delaunay triangulation codes. The theoretical algorithms seem impractical because they are complex and have large constants, and the implementations are either specialized to uniform distributions or require partitioning by hand. We therefore developed a new algorithm loosely based on the Edelsbrunner and Shi approach. 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 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 by using components that are not theoretically optimal, as discussed below.

We initially prototyped the algorithm in the programming language NESL [13], a high level parallel programming language designed for algorithm specification and teaching. NESL allowed us to develop the code quickly, try several variants of the algorithm, and run many experiments to analyze the characteristics. For such prototyping NESL has the important properties that it is deterministic, uses simple parallel constructs, has implicit memory management with garbage collection and full bounds checking, and has an integrated interactive environment with visualization tools. We refined the initial algorithm through alternating rounds of experimentation and algorithmic design. We improve the basic algorithm from a practical point of view by using the 2D convex hull algorithm of Chan et al. [21]. This algorithm has nonoptimal theoretical work since it runs in worst case $O(n \log h)$ work instead of linear (for sorted input). However, in practice our experiments show 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 with efficient sequential algorithms over a wide range of distributions. We also observed that the algorithm can be used effectively to partition the problem into regions in which the work on each region is approximately equal.

After running the experiments in NESL and settling on the specifics of an algorithm we implemented it in C and MPI [14] with the aid of the Machiavelli toolkit [22]. This implementation uses our algorithm as a coarse-grained partitioner, as outlined above, and then uses a sequential algorithm to finish the subproblems. Although in theory a compiler could convert the NESL code into efficient C and MPI code, significantly more research is necessary to get the compiler to that stage.

Section 2 describes the algorithm and its implementation in NESL, and its translation into MPI and C. Section 3 describes the experiments we ran on the two implementations.

2. Projection-Based Delaunay Triangulation. In this section we first present our algorithm, concentrating on the theoretical motivations for our design choices, then discuss particular choices made in the NESL implementation, and finally discuss choices made in the C and MPI implementation.

The basic algorithm uses a divide-and-conquer strategy. Figure 3 gives a pseudocode description of the algorithm. 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, composed of Delaunay edges,

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.

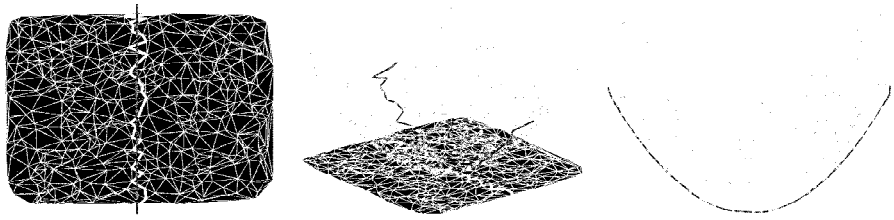
Output: The set of Delaunay triangles of P which are contained within B.

Method:

1. If all the points in P are on the border 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 border). 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:
   - $B^{\mathrm{L}} =$ BORDER_MERGE($B, \mathcal{H}$)
     $B^{\mathrm{R}} =$ BORDER_MERGE($B, \bar{\mathcal{H}}$)
   - $P^{\mathrm{L}} = \{p \in P \mid p \text{ is left of } \mathcal{L}\} \cup \{p' \in P_{\mathcal{H}} \mid p' \text{ contributed to } B^{\mathrm{L}}\}$
     $P^{\mathrm{R}} = \{p \in P \mid p \text{ is right of } \mathcal{L}\} \cup \{p' \in P_{\mathcal{H}} \mid p' \text{ contributed to } B^{\mathrm{R}}\}$
6. Return DELAUNAY($P^{\mathrm{L}}, B^{\mathrm{L}}$) $\cup$ DELAUNAY($P^{\mathrm{R}}, B^{\mathrm{R}}$)

Fig. 3. The projection-based parallel Delaunay triangulation algorithm. Initially B is the convex hull of P. The algorithm as shown 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.

and (2) the set of points P of the region, composed of *internal points* and points on the border. Note that the region may be unconnected. At each call, we divide the region into two regions using a median line cut of the internal points. The set of internal points is subdivided into those to the left and to the right of the median line. The polygonal border is subdivided using a new path of Delaunay edges that corresponds to the median line: the new path separates Delaunay triangles whose circumcenter is to the left of the median line, from those whose circumcenter is to the right of the median line. Once the new path is found, the new border of Delaunay edges for each subproblem is determined by merging the old border with the new path, in the BORDER_MERGE subroutine. Some of the internal points may appear in the new path, and may become border points of the new subproblems. Since we are using a median cut, our algorithm guarantees that the number of internal points is reduced by a factor of at least 2 at each call. This simple separation is at the heart of our algorithm's efficiency. Unlike early divide-and-conquer strategies for Delaunay triangulation which do most of the work when returning from recursive calls [15], [16], [4], this algorithm does all the work before making recursive calls, and trivially appends results when returning from the recursive calls.



(a) The median line $\mathcal{L}$ and the path $\mathcal{H}$.

(b) The points projected onto the paraboloid.

(c) Paraboloid points projected onto the vertical plane.

Fig. 4. Finding a dividing path for Delaunay triangulation. This shows the median line, all the points projected onto 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.

The new separating path of Delaunay edges is a lower convex hull of a simple transformation of the current point set. To obtain this 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 xy plane is $\mathcal{L}$ (see Figure 4). The 2D lower convex hull of those projected points is the required new border path $\mathcal{H}$. In the case of degeneracies, the structure of $\mathcal{H}$ may be more complicated, as is discussed in Lemma 1 below.

This divide-and-conquer method can proceed as long as the subproblem contains internal points. Once the subproblem has no more internal points, it is a set of (possibly pinched) cycles of Delaunay edges. There may be some missing Delaunay edges between border points that still have to be found. To do that, we move to a different strategy which we refer to as the END_GAME. In a theoretical view of the algorithm, to obtain optimal work and run-time bounds, a standard Delaunay triangulation algorithm may be substituted at this point. However, for the implementation we suggest a simple and efficient in practice end-game strategy that takes advantage of the simple structure of the subproblems at this point. This strategy is described in Section 2.1.

Correctness of the Median Splits. The simple partitioning strategy we use is crucial to the efficiency of our algorithm. This strategy is successful because of the relationship between the median line $\mathcal{L}$ used to partition the points, and the Delaunay path $\mathcal{H}$ used to partition the borders. In particular the following lemma 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 $\mathcal{L}$ (see Figure 4(a)).

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

Proof outline. Let Q be a convex body in $\mathbb{R}^3$ with boundary $\bar{Q}$. In our case, $\bar{Q}$ is the 3D paraboloid. Without loss of generality, assume that $\mathcal{L}$ is the line $x = 0$ in the xy plane, and that the median point $q = (0, 0)$. The 3D paraboloid can then be described as the set of points $(x, y, x^2 + y^2)$.

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 the interior of 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 the interior of Q. A point that is both light and dark is said to be on the *silhouette*. The silhouette forms a border between light and dark points. Note that points on the xy plane with a negative x coordinate are mapped using the paraboloid mapping to a dark point.

The set of Delaunay points in the plane, P, is mapped to a set of 3D points $\hat{T}$ on the paraboloid $\bar{Q}$. Let $\bar{T}$ stand for the convex hull of $\hat{T}$, and let T be the interior solid bounded by $\bar{T}$. Clearly, T is a convex body contained in Q. We can classify points on $\bar{T}$ as dark, light, and silhouette as well.

$\bar{T}$ is composed of linear faces, edges, and points. 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. $\mathcal{H}$, the projection of the silhouette of $\bar{T}$ on the plane, is then a simple path.

Note that the points $\hat{T}$ are both on $\bar{T}$ and on $\bar{Q}$. Furthermore, if a point of $\hat{T}$ is dark (light) in $\bar{Q}$, it is also dark (light) in $\bar{T}$. However, if a point is dark in $\bar{Q}$, its 2D projection is left of or on $\mathcal{L}$. If a point is dark in $\bar{T}$, its 2D projection is left of or on the path $\mathcal{H}$. Therefore if a point is left of or on $\mathcal{L}$, then it is left of or on $\mathcal{H}$. This property and hence the statement of the lemma is true in the more general setting in which Q is an arbitrary convex body, $\mathcal{L}$ is the projection of its silhouette onto the plane, $\hat{T}$ is an arbitrary set of points on $\hat{Q}$, and $\mathcal{H}$ is the projection of the silhouette of T onto the plane. □

We now give a simple characterization of when a face on $\bar{T}$ is light, dark, or on the silhouette in term of its circumscribing circle (assuming $\hat{T}$ lie on the paraboloid).

DEFINITION 1. A Delaunay triangle is called a left, right, or 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 strictly dark, strictly light, or on the silhouette if and only if its triangle in the plane is left, right, or middle, respectively.*

PROOF. The supporting plane of face F is of the form $ax + by - z = c$ with normal $n = (a, b, -1)$. Now F is strictly 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. In general we could allow faces on the silhouette in which case $\mathcal{H}$ would could include

triangles (or other shapes if degeneracies are allowed so that faces are not triangles). We could then split the path into one that goes around the triangles to the left ($\mathcal{H}_L$) and one that goes around the triangles to the right ($\mathcal{H}_R$) and use these to merge with the old border for the recursive calls.

Analysis. We now consider the total work and depth of the algorithm, basing our analysis on a 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* [23] *for the* LOWER_CONVEX_HULL *and Cole et al.'s algorithm* [4] *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 axes and compress these ranks when partitioning). This allows us to use Overmars and van Leeuwen's linear-work algorithm [23] for 2D convex hulls on sorted input. Since their algorithm uses divide-and-conquer and each divide stage takes $O(\log n)$ serial 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 partitioning a region is linear, the total work needed to process each level is $O(n)$ and the total work across the levels is $O(n \log n)$. Similarly, the depth is $O(\log^3 n)$.

This is the cost to reduce the problem 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 simpler techniques, such as constructing the triangulation incrementally using point insertion. If the border is large, then the Cole et al. algorithm [4] can be used. □

Comparison with Edelsbrunner and Shi. Here we explain how our algorithm differs from the original algorithm presented by Edelsbrunner and Shi [12]. Our algorithm uses the same technique of partitioning the problem into subproblems using a path obtained by projecting the point set and using a 2D convex hull subroutine. However, 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 simple local operations for merging the borders. In contrast, since Edelsbrunner and Shi are solving the more general problem of a 3D convex hull, they have to (1) find a 4-partition using two 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 complicated than ours, and in the case of step (3) our is asymptotically faster—it runs in linear rather than than $O(n \log n)$—leading to an overall faster algorithm.

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 find a face of the convex hull easily—we just use the median point—and to partition the points simply by using a median line. This avoids the linear programming and point location steps (steps (2) and (5)). Furthermore, because of our median cut lemma (Lemma 1), our partitions can all be made parallel to the x or y axis. This both avoids finding a 4-partition (step (1)) and also allows us in linear time to keep our points sorted along the cut. Using this sorted order we can use a linear work algorithm for the convex hull, as discussed. This is not possible with the Edelsbrunner and Shi algorithms, since it must make cuts in arbitrary directions in order to get the 4-partition needed to guarantee progress. 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. However, our experiments show 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 that 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 [24].

2.1. *NESL Implementation.* This section gives more details on the substeps and outlines the data structures used in the NESL implementation. NESL [13] is a nested data-parallel language that is well suited to expressing irregular divide-and-conquer algorithms of this type. It is very high level and allowed us to prototype, debug, and experiment with variations of the algorithm quickly. The main components of the implementation are the border merge, the convex hull, and the end game. An important result of our experiments was that we noticed that subroutines that are not theoretically optimal sometimes lead to more practical resultes. These subroutines are discussed below.

Border Merge. The border merge consists of merging the new Delaunay path 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. 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 points, i.e., B could contain two triplets with the same middle point (see Figure 6).

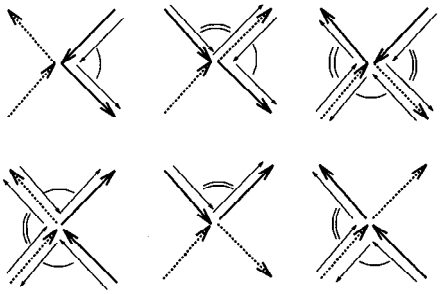


Fig. 5. The six cases for merging the new and old border. The old border is in thick lines, and the partitioning path in dotted lines. The convention in the drawings (and the program) is that the interior lies left of the border when proceeding in the direction of the arrows. The resulting two new borders are in thin lines, with the new left border marked with double arcs, and the new right border with a single arc.

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 endpoints on B.

The border merge can be computed by considering only the local structure of $\mathcal{H}$ and B. Specifically, we intersect 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 5 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 central to our algorithm, and is the most expensive component. We considered three candidates for the convex hull algorithm: (1) Overmars and van Leeuwens' [23], which is $O(n)$ work for sorted points. (2) Kirkpatrick and Seidel's $O(n \log h)$ algorithm [18], and its much simplified form as presented by Chan et al. [21]. (3) A simple worst case $O(n^2)$ quickhull algorithm, as in [25] and [26]. All of these algorithms can be naturally parallelized.

Using the algorithm of Overmars and van Leeuwen [23] the convex hull of presorted points can be computed in serial $O(n)$ work, and the parallel extension is straightforward. Since we can presort the point set, and maintain the ordering through the x and y cuts we perform, using Overmars and van Leeuwen's algorithm will result in linear work for each convex hull invocation.

With the other algorithms we do not make use of the sorted order. The lower bound for finding a 2D convex hull for unsorted input is $O(n \log h)$ work [18] making it seem that

these algorithms should not perform as well given that our output sizes h are typically around $\sqrt{n}$. The lower bound, however, is based on pathological distributions in which all the points are very close to the boundary of the final hull. For more common distributions, such as the ones that appeared in our experiments even with the highly nonuniform input, the algorithms run in what appears to be linear work. This is because the Kirpatrick–Seidel, Chan, and quickhull algorithms all throw away most of the interior points in the first few levels of recursion.

Preliminary experiments showed that quickhull outperformed the Overmars and van Leeuwen algorithm. The experiments also showed, however, that for quickhull some of our distributions resulted in much more costly convex hull phases than the others, in that they seemed to have much higher constants (they still scaled linearly with the input size). Quickhull advances by using the furthest point heuristic, and, for extremely skewed distributions, the furthest point does not always provide a balanced split. The Kirpatrick–Seidel and Chan algorithms guarantee good splits but finding the splits is more expensive making these algorithms slower for the more uniform distributions. We therefore use an algorithm that combines 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 at lower levels 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 routine. 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 6). The border can be split by joining the corners into a linked list and using a list-ranking algorithm (we use pointer jumping). After this step each subproblem is a simple cycle, represented by an ordered list of point indices.

The core of the end game is to find a 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: $\dot{P} = \{(p - q)/\|p - q\| \mid p \in P\}$. 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 7). If

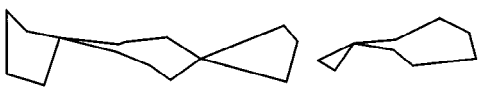


Fig. 6. One of the border subproblems, created by running the basic algorithm until all the points are on the border.

Fig. 7. Starting in (a) with a border subproblem, we look for a new Delaunay edge of a point q by using inversion around q to move to the dual problem of finding convex-hull edges. This is shown using thick lines in (b). 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.

p_d is either p_a or p_b, 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.

2.2. *Implementation in MPI and C.* NESL's current parallel implementation layer assumes an implicitly load-balanced vector PRAM model [27]. This can be efficiently implemented on parallel machines with very high memory and communication bandwidth, but achieves relatively poor performance on current RISC-based multiprocessor architectures, due to the high cost of communication [28]. The final Delaunay triangulation algorithm was therefore reimplemented for production purposes using the Machiavelli [22] toolkit, which has been specifically designed for the efficient implementation of parallel divide-and-conquer algorithms on machines with limited communication bandwidth.

Machiavelli uses the recursive subdivision of asynchronous teams of processors running data-parallel code to implement directly the behavior of a divide-and-conquer algorithm. It obtains parallelism from data-parallel operations within teams and from the task-parallel invocation of recursive functions on independent teams of processors. When a processor has recursed down to a team containing only itself, it switches to a sequential version of the code (or even a more efficient serial algorithm for solving the problem). Machiavelli currently consists of a library of vector communication primitives, based on C and MPI [14], and a small run-time system. The library provides an abstract model of a vector, which is distributed in a block fashion across the processors of a team. It also contains optimized communication functions for many of the idioms that occur in divide-and-conquer algorithms, such as splitting and merging vectors. Local data-parallel operation, such as elementwise mathematical functions on sequences, are implemented as loops over the appropriate section of data on each processor. The run-time system provides memory management and team operations.

The rest of this section describes some of the additional design decisions and optimizations made in the MPI and C implementation, including the layout of data structures and some modifications to subroutines. Most of the optimizations relate to reducing or eliminating interprocessor communication. Analysis of the implementation can be found in Section 3.3.

Data Structures. As in the NESL implementation, sets of points are represented by vectors, and borders are composed of triplets. However, these triplets are not balanced across the processors as the point vectors are, but rather are stored on the same processor as their middle point. A vector of indices is used to link the points in P with the triplets in the borders B and H. Given these data structures, the operations of finding internal points, and projecting the points onto a parabola (see Figure 3), both reduce to simple local loops with no interprocessor communication.

Finding the Median. Initially a parallel version of quickmedian [29] was used to find the median internal point along the x or y axis. Quickmedian redistributes data amongst the processors on each recursive step, resulting in high communication overhead. It was therefore replaced with a median-of-medians algorithm, in which each processor first uses a serial quickmedian to compute the median of its local data, then shares this local median with the other processors in a collective communication step, and finally computes the median of all the local medians. The result is not guaranteed to be the exact median, but in practice it is sufficiently good for load-balancing purposes; this modification decreased the running time of the Delaunay triangulation program for the distributions and machine sizes studied (see Section 3.3) by 4–30%.

Finding the Lower Convex Hull. As in the original algorithm, the pruning variant of quickhull by Chan et al. [21] is used to find the convex hull. This algorithm tests the slope between pairs of points and uses pruning to guarantee that recursive calls have at most three-quarters of the original points. However, pairing all n points and finding the median of their slopes is a significant addition to the basic cost of quickhull. Experimentally, pairing only $\sqrt{n}$ points was found to be a good compromise between robustness and performance when used as a substep of Delaunay triangulation (see Section 3.3 for an analysis). As with the median-of-medians approach, the global effects of receiving approximate results from a subroutine are more than offset by the decrease in running time of the subroutine.

Combining Results. The quickhull algorithm concatenates the results of two recursive calls before returning. In Machiavelli this corresponds to merging two teams of processors and redistributing their results to form a new vector. However, since this is the last operation that the function performs, the intermediate appends in the parallel call tree (and their associated interprocessor communication phases) can be optimized away. They are replaced with a single Machiavelli function call at the top of the tree that redistributes the local result on each processor into a parallel vector shared by all the processors.

Creating the Subproblems. To eliminate an interprocessor communication phase in the border merge step, the two outer points in a triplet are replicated in the triplet structure, rather than being represented by pointers to point structures (which might well be stored on different processors). All the information required for the line orientation tests can thus be found on the local processor. The memory cost of this replication is analyzed in Section 3.3. Additionally, although Figure 3 shows two calls to the border merge function (one for each direction of the new dividing path), in practice it is faster to make a single pass, creating both new borders and point sets at the same time.

End Game. Since we are using the parallel algorithm as a coarse partitioner, the end game is replaced with a standard serial Delaunay triangulation algorithm. We chose to use the version of Dwyer's algorithm that is implemented in the Triangle mesh generation package by Shewchuk [30]. Triangle has a performance comparable with that of the original code by Dwyer, and in addition uses adaptive arithmetic, which is important for avoiding degenerate conditions in nonuniform distributions. Since the input format for Triangle differs from that used by the parallel program, conversion steps are necessary before and after calling it. These translate between the pointer-based format of Triangle, which is optimized for sequential code, and the indexed format with triplet replication used by the parallel code. No changes are necessary to the source code of Triangle.

3. Experiments. In this section we describe the experiments performed on both the NESL and MPI and C implementations. The NESL experiments concentrate on abstract cost measures, and are used to prove the theoretical efficiency of our algorithm. The MPI experiments concentrate on running time, and are used to prove the practical efficiency of our production implementation.

3.1. *Data Distributions.* The design of the data set is always of great importance for the experimentation and evaluation of an algorithm. Our goal was to test our algorithm on distributions that are representative of real-world problems. To highlight the efficiency of our algorithm, we sought to include highly nonuniform distributions that would defy standard uniform techniques such as bucketing. We therefore considered distributions motivated by different domains, such as the distribution of stars in a flat galaxy (Kuzmin) and point sets originating from mesh generation problems. The density of our distributions may vary greatly across the domain, but we observed that the resulting triangulations tend to contain relatively few bad aspect-ratio triangles—especially when contrasted with some artificial distributions, such as points along the diagonals of a square [1]. Indeed, for the case of uniform distribution, Bern et al. [31] provided bounds on the expected worst aspect ratio. A random distribution with few bad aspect-ratio triangles is advantageous for testing in the typical algorithm design cycle, where it is common to concentrate initially on the efficiency of the algorithm, and only at later stages concentrate on its robustness. Our chosen distributions are shown in Figure 8 and summarized here.

- **The Kuzmin distribution:** this distribution is used by astrophysicists to model the distribution of star clusters in flat galaxy formations [32]. It is a radially symmetric distribution whose density falls quickly as r increases, providing a good example of convergence to a point. The accumulative probability function as a function of the radius r is

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

  The point set can be generated by first generating r, and then uniformly generating a point on the sphere of radius r. r itself is generated from the accumulative probability function by first generating a uniform random variable X, and then equating $r = M^{-1}(X)$.

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

Fig. 8. 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).

- **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). We define the 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 in the range $[0, 1]$, then the transformation is

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

  In our experiments, we set $b = 0.001$.
- **Normal distribution:** this consists of 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:** this consists of 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 nonuniform distributions, and also to form common ground for comparison with other relevant work.

3.2. *Experimental Results*: *NESI.* The purpose of these experiments was to measure various properties of the algorithm, including 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 an efficient 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 understand better how the algorithm is affected 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. All these experiments were run using NESL [13] and the measurements were taken by instrumenting the code.

We measured the different quantities over the four distributions, for sizes varying from 2^{10} to 2^{17} points. For each distribution 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 floating-point instructions) and seem to have a strong correlation to running time [1], at least for algorithms with similar structure.

Work. To estimate the work of our algorithm, we compare the floating-point operation counts with those of Dwyer's sequential algorithm [33]. Dwyer's 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 other distributions is at least as efficient as the original algorithm. In a recent paper, Su and Drysdale [1] experimented with a variety of sequential Delaunay algorithms, 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 with. We use the same code for Dwyer's algorithm as used in [1]. The results are shown in Figure 1 in the Introduction.

Our algorithm performance is similar for 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 9 shows the breakdown of floating-point operation 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 (it varies

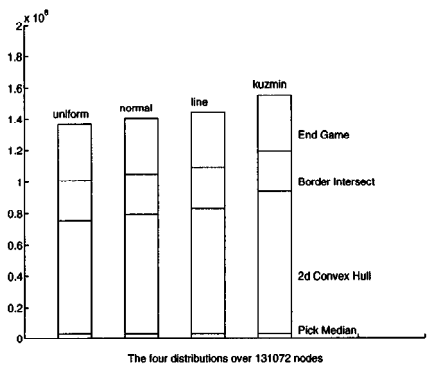


Fig. 9. Floating-point operation counts partitioned according to the different algorithmic parts, for each distribution.

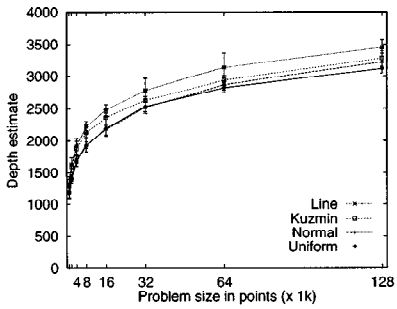


Fig. 10. The depth of the algorithm for the different distributions and problem sizes.

by less than 10%). For the convex hull, the Kuzmin distribution 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 [21], 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 depth over parallel calls and summing depth over sequential calls. Figure 10 shows the depth as a function of problem size for the four distributions. As can be seen, the depth is also not strongly affected 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 11). 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 2 every level, but provides no guarantees for the number of points on the border of each subproblem. Figure 11 shows the total size including the border. As can be seen, the size goes down uniformly. The discontinuity at around level 20 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 cost of the 2D convex hull, can be obtained by looking at the sum of the problem sizes at each level (see Figure 12). Finding the median cut and intersecting the new border with old performs 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

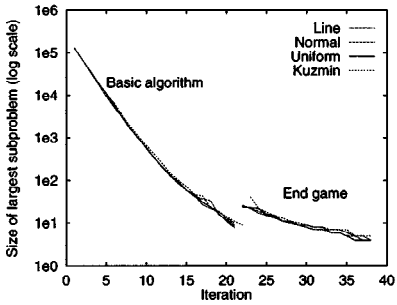


Fig. 11. The maximal piece size as a function of iteration for our algorithm on a representative example of 2^{17} points from each distribution. The piece size includes both internal points and border points. The graph shows that our separator and our end game work quite well for all the distributions.

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 the termination of the basic algorithm at a different iteration.

The Algorithm as a Coarse Partitioner. Finally, we provide support for the contention that the algorithm can be used to divide the problem into a number of subproblems efficiently, and then switch to a sequential algorithm. Specifically, Figure 13 shows the accumulative floating-point operation counts per iteration on 2^{17} points versus the cost of Dwyer's sequential algorithm. In general, at level i the number of subproblems is 2^i. For example, consider using our algorithm for five levels to split the problem into $2^5 = 32$ subproblems and then use a serial algorithm to solve each subproblem (probably on 32 processors). The graph shows that the work involved in the splitting is only about 30% of the total work that is required by Dwyer's algorithm.

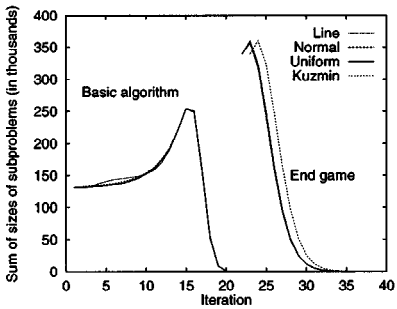


Fig. 12. Sum of the sizes of all the subproblems per iteration, on a representative 2^{17} points example from each distribution.

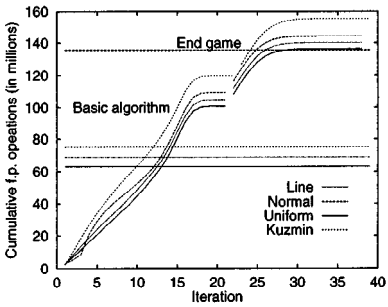


Fig. 13. The cumulative work in floating-point operation counts per iteration, for a representative 2^{17} points example from each distribution. The idea is to see how much work it is to divide the problem into 2^i problems in level i, versus the median cost of the sequential algorithm.

3.3. *Experimental Results*: *MPI and C.* In this section we present experimental results for the MPI and C implementation on four machines, and analyze where the bottlenecks are, the reasons for any lack of scalability, and the effect of some of the implementation decisions presented in Section 2.2 on both running time and memory use. Unlike the previous section which considered work in terms of floating-point operations, this section reports running times and therefore includes all other costs such as communication and synchronization.

To test portability, we used four parallel architectures: a loosely coupled workstation cluster (DEC AlphaCluster) with 8 processors, a shared-memory SGI Power Challenge with 16 processors, a distributed-memory Cray T3D with 64 processors, and a distributed-memory IBM SP2 with 16 processors. To test parallel efficiency, we compared timings with those on one processor, when the program immediately switches to the sequential Triangle package [30]. To test the ability to handle nonuniform distributions we used the four distributions specified earlier. All timings represent the average of five runs using different seeds for a pseudorandom number generator. For a given problem size and seed the input data is the same regardless of the architecture and number of processors.

Speedup. To illustrate the algorithm's parallel efficiency, Figure 14 shows the time to triangulate 2^{17} points on different numbers of processors, for each of the four platforms and the four different distributions. This is the largest number of points that can be triangulated on one processor of all four platforms. Speedup is not perfect because as more processors are added, more levels of recursion are spent in parallel code rather than in the faster sequential code. However, we still achieve approximately 50% parallel efficiency for the distributions and machine sizes tested—that is, we achieve about half of the perfect speedup over efficient sequential code. Additionally, the Kuzmin and line distributions show similar speedups to the uniform and normal distributions, confirming that the algorithm is effective at handling nonuniform distributions as well as uniform ones. Data for the Kuzmin and line singularity distributions are also shown in Table 1. Note that the Cray T3D and the DEC AlphaCluster use the same 150 MHz Alpha 21064 processors, and their single-processor times are thus comparable. However, the T3D's

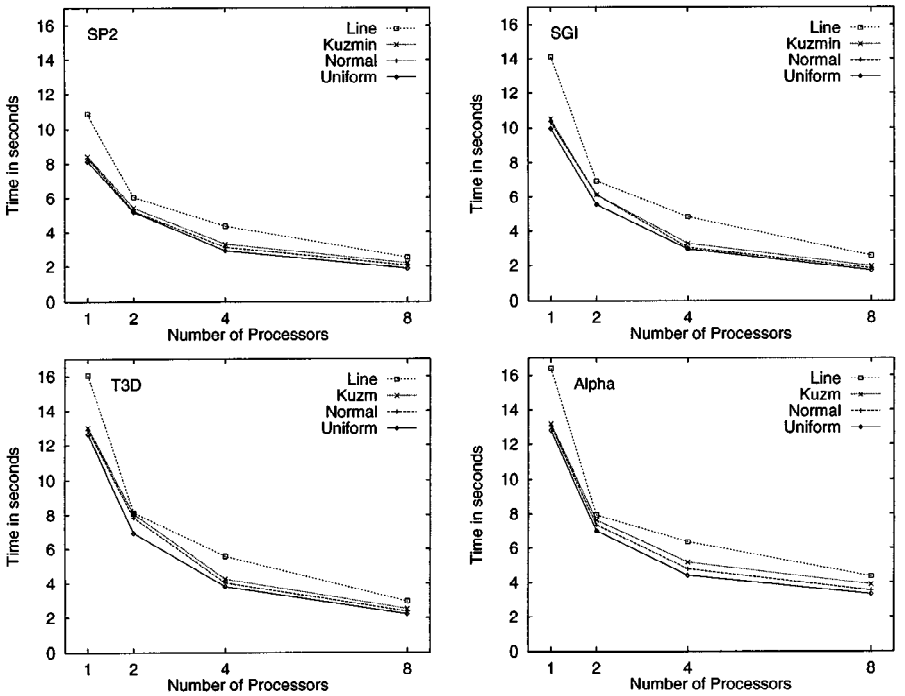


Fig. 14. Speedup of Delaunay triangulation program for four input distributions and four parallel architectures. The graphs show the time to triangulate a total of 128k points as the number of processors is varied. Single processor results are for efficient sequential code. Increasing the number of processors results in more levels of recursion being spent in slower parallel code rather than faster sequential code, and hence the speedup is not linear. The effect of starting with an *x* or *y* cut is shown in the alternately poor and good performance on the highly directional line distribution. IBM SP2 results are for thin nodes, using `xlc -O3` and MPICH 1.0.12. SGI Power Challenge results are for R8000 processors, using `cc -O2` and SGI MPI. Cray T3D results use `cc -O2` and MPICH 1.0.13. DEC AlphaCluster results are for DEC 3000/500 workstations connected by an FDDI Gigaswitch, using `cc -O2` and MPICH 1.0.12.

Table 1. Time taken, and relative speedup, when triangulating 128k points on the four different platforms tested.

Processors	SP2		SGI		T3D		Alpha	
				Kuzmin distribution				
1	8.42s	(1.00)	10.48s	(1.00)	13.02s	(1.00)	13.19s	(1.00)
2	5.42s	(1.55)	6.12s	(1.71)	8.05s	(1.62)	7.63s	(1.73)
4	3.31s	(2.55)	3.28s	(3.20)	4.25s	(3.06)	5.17s	(2.55)
8	2.19s	(3.85)	1.96s	(5.36)	2.53s	(5.16)	3.89s	(3.39)
				Line singularity distribution				
1	10.82s	(1.00)	14.11s	(1.00)	16.05s	(1.00)	16.39s	(1.00)
2	6.04s	(1.79)	6.91s	(2.04)	8.13s	(1.97)	7.93s	(2.07)
4	4.36s	(2.48)	4.84s	(2.92)	5.58s	(2.88)	6.36s	(2.58)
8	2.52s	(4.30)	2.56s	(5.51)	2.96s	(5.43)	4.36s	(3.76)

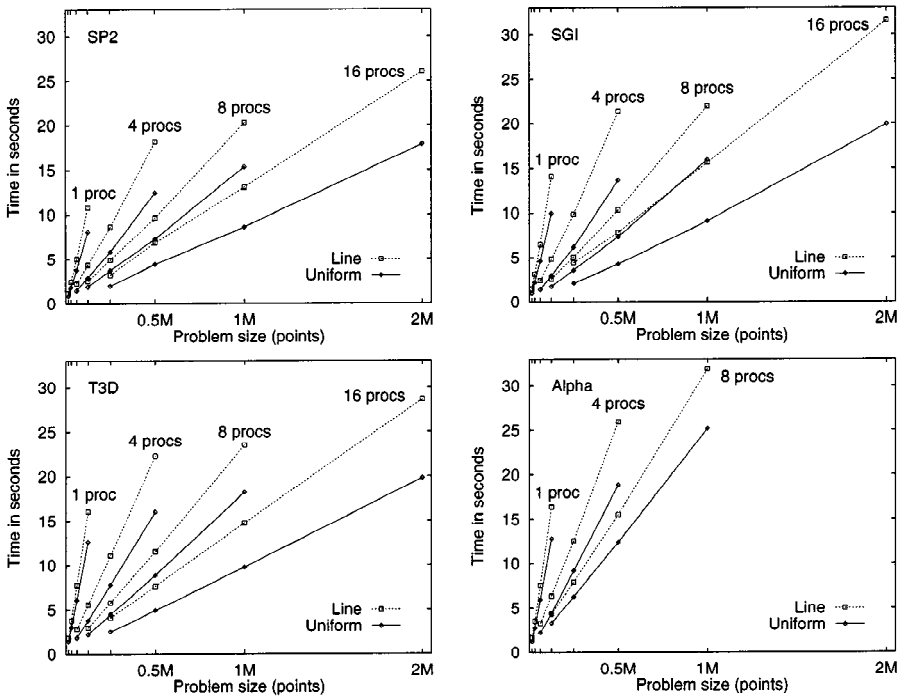


Fig. 15. Scalability of Delaunay triangulation program for two input distributions and four parallel architectures. The graphs show the time to triangulate 16k–128k points per processor as the number of processors is varied. For clarity, only the fastest (uniform) and slowest (line) distributions are shown. Machine setups are as in Figure 14.

specialized interconnection network has lower latency and higher bandwidth than the commodity FDDI network on the AlphaCluster, resulting in better scalability.

To illustrate scalability, Figure 15 shows the time to triangulate a variety of problem sizes on different numbers of processors. For clarity, only the uniform and line distributions are shown, since these take the least and most time, respectively. Again, per-processor performance degrades as we increase the number of processors because more levels of recursion are spent in parallel code. However, for a fixed number of processors the performance scales very well with problem size.

To illustrate the relative costs of the different components of the algorithm, Figure 16(a) shows the accumulated time per substep of the algorithm. The parallel substeps of the algorithm, namely median, convex hull, and splitting and forming teams, become more important as the number of processors is increased. The time taken to convert to and from Triangle's data format is insignificant by comparison, as is the time spent in the complicated but purely local border merge step. Figure 16(b) shows the same data from a different view, as the total time per recursive level of the algorithm. This clearly shows the effect of the extra parallel phases as the number of processors is increased.

Finally, Figure 17 uses a parallel time line to show the activity of each processor when triangulating a line singularity distribution. There are several important effects

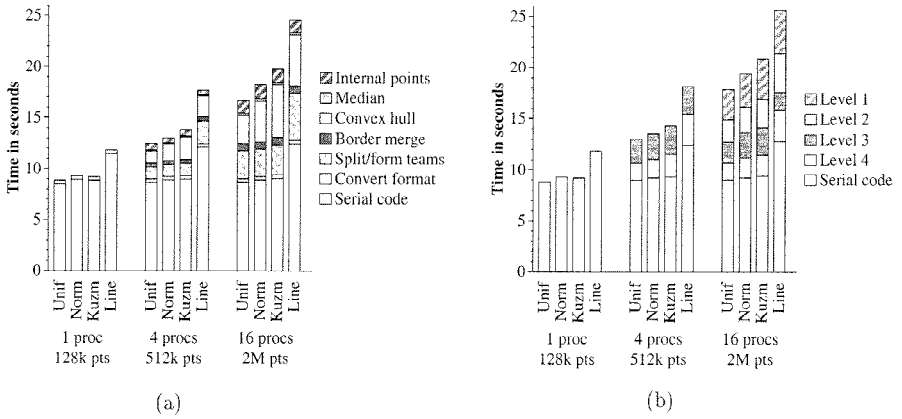


Fig. 16. Two views of the execution time as the problem size is scaled with number of processors (IBM SP2, 128k points per processor). (a) The total time spent in each substep of the algorithm. The time spent in sequential code remains approximately constant, while convex hull and team operations (which includes synchronization delays) are the major overheads in the parallel code. (b) The time per recursive level of the algorithm; note the approximately constant overhead per level.

that can be seen here. First, the nested recursion of the convex hull algorithm within the Delaunay triangulation algorithm. Second, the alternating high and low time spent in the convex hull, due to the effect of the alternating x and y cuts on the highly directional line distribution. Third, the operation of the processor teams. For example, two teams of four processors split into four teams of two just before the 0.94 second mark, and further subdivide into eight teams of one processor (and hence switch to sequential code) just after. Lastly, the amount of time wasted waiting for the slowest processor in the parallel merge phase at the end of the algorithm is relatively small, despite the very nonuniform distribution.

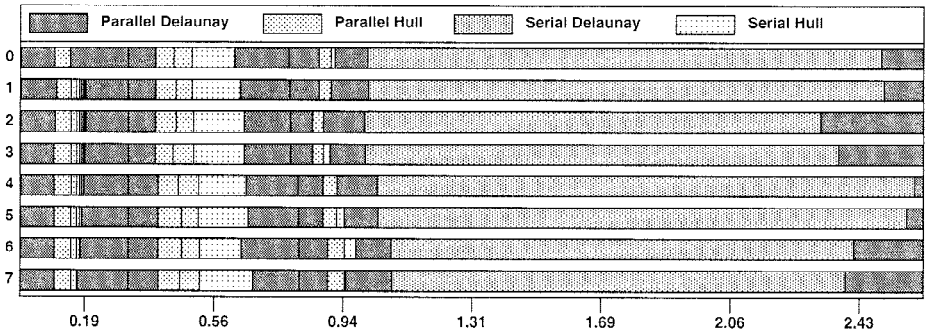


Fig. 17. Activity of eight processors over time, showing the parallel and sequential phases of Delaunay triangulation and its inner convex hull algorithm (IBM SP2, 128k points in a line singularity distribution). A parallel step consists of two phases of Delaunay triangulation code surrounding one or more convex hull phases; this run has three parallel levels. Despite the nonuniform distribution the processors do approximately the same amount of sequential work.

Memory Requirements. As explained in Section 2.2, border points are replicated in triplet structures to eliminate the need for global communication in the border merge step. Since one reason for using a parallel computer is to be able to handle larger problems, we would like to know that this replication does not significantly increase the memory requirements of the program. Using 64-bit doubles and 32-bit integers, a point and associated index vector entry occupies 32 bytes, while a triplet occupies 48 bytes. However, since a border is normally composed of only a small fraction of the total number of points, the additional memory required to hold the replicated triplets is relatively small. For example, in a run of 512k points in a line singularity distribution on eight processors, the maximum ratio of triplets to total points on a processor (which occurs at the switch between parallel and sequential code) is approximately 2000 to 67,000, so that the triplets occupy less than 5% of required storage. Extreme cases can be manufactured by reducing the number of points per processor; for example, with 128k points the maximum ratio is approximately 2000 to 17,500. Even here, however, the triplets still represent less than 15% of required storage, and by reducing the number of points per processor we have also reduced absolute memory requirements.

Performance of Convex Hull Variants. Finally, we investigate the performance of the convex hull variants described in Section 2.2. A basic quickhull algorithm was benchmarked against two variants of the pruning quickhull by Chan et al. [21]: one that pairs all n points, and one that pairs a random sample of $\sqrt{n}$ points. Results for an extreme case are shown in Figure 18. As can be seen, the n-pairing algorithm is more than twice as fast as the basic quickhull on the nonuniform Kuzmin distribution (over all the distributions and machine sizes tested it was a factor of 1.03–2.83 faster). The $\sqrt{n}$-pairing algorithm provides a modest additional improvement, being a factor of 1.02–1.30 faster than the n-pairing algorithm.

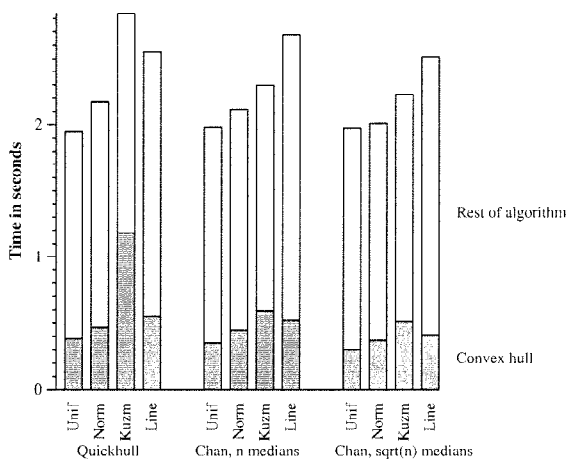


Fig. 18. Effect of different convex hull functions on time to triangulate 128k points on an 8-processor IBM SP2. The pruning quickhull due to Chan et al. [21] has a much better performance than the basic algorithm on the nonuniform Kuzmin distribution; using a variant with reduced sampling accuracy produces a modest additional improvement.

4. Concluding Remarks. In this paper we described an algorithm for Delaunay triangulation that is theoretically optimal in work, requires polylogarithmic depth, and is efficient in practice, at least when using some components that are not known to be asymptotically optimal. In particular the main results are:

- The algorithm runs with $O(n \log n)$ work and $O(\log^3 n)$ depth (parallel time) on a CREW PRAM. This is the best bound that we know for a projection-based algorithm (i.e., one that finds the separating path before making recursive calls).
- Using a variant of our algorithm that uses nonoptimal subroutines the constants in the measured work are small enough to be competitive with the best sequential algorithm that we know of. In particular, depending on the point distribution the algorithm ranges from being 40% to 90% work-efficient relative to Dwyer's algorithm [33].
- The algorithm is highly parallel. The ratio of work to parallel time is approximately 10^4 for 10^5 points. This gives plenty of freedom on how to parallelize the code and was crucial in getting good efficiency in our MPI implementation (the additional parallelism is used to hide message-passing overheads and minimize communication).
- An implementation of the algorithm as a coarse partitioner in MPI and C gets better speedup than previously reported across a variety of machines and nonuniform distributions.

References

[1] P. Su and R. Drysdale. A comparison of sequential Delaunay triangulation algorithms. *Computational Geometry*: *Theory and Applications*, 7:361–386, 1997.

[2] A. Chow. Parallel Algorithms for Geometric Problems. Ph.D. thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, IL, December 1981.

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

[4] R. Cole, M. T. Goodrich, and C. Ó Dúnlaing. Merging free trees in parallel for efficient Voronoi diagram construction. In *Proceedings of the* 17*th International Colloquium on Automata, Languages and Programming*, pages 32–45, July 1990.

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

[6] J. H. Reif and S. Sen. Polling: a new randomized sampling technique for computational geometry. In *Proceedings of the* 21*st Annual ACM Symposium on Theory of Computing*, pages 394–404, May 1989.

[7] M. Ghouse and M.T. Goodrich. In-place techniques for parallel convex hull algorithms. In *Proceedings of the* 3*rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 192–203, July 1991.

[8] M. L. Merriam. Parallel implementation of an algorithm for Delaunay triangulation. In *Proceedings of the First European Computational Fluid Dynamics Conference*, volume 2, pages 907–912, September 1992.

[9] P. Cignoni, D. Laforenza, C. Montani, R. Perego, and R. Scopigno. Evaluation of Parallelization Strategies for an Incremental Delaunay Triangulator in E^3. Technical Report C93-17, Consiglio Nazionale delle Ricerche, Rome, November 1993.

[10] Y. A. Teng, F. Sullivan, I. Beichl, and E. Puppo. A data-parallel algorithm for three-dimensional Delaunay triangulation and its implementation. In *Proceedings of Supercomputing* '93, pages 112–121. ACM, New York, November 1993.

[11] P. Su. Efficient Parallel Algorithms for Closest Point Problems. Ph.D. thesis, PCS-TR94-238, Department of Computer Science, Dartmouth College, Hanover, NH, 1994.

[12] H. Edelsbrunner and W. Shi. An $O(n \log^2 h)$ time algorithm for the three-dimensional convex hull problem. *SIAM Journal on Computing*, 20:259–277, 1991.

[13] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[14] Message Passing Interface Forum. MPI: a message-passing interface standard. *International Journal of Supercomputing Applications and High Performance Computing*, 8(3/4):165–419, March–April 1994.

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

[16] 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.

[17] F. Dehne, X. Deng, P. Dymond, A. Fabri, and A. A. Khokhar. A randomized parallel 3D convex hull algorithm for coarse grained multicomputers. In *Proceedings of the* 7*th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 27–33, July 1995.

[18] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15(1):287–299, February 1986.

[19] J. R. Davy and P. M. Dew. A note on improving the performance of Delaunay triangulation. In *Proceedings of Computer Graphics International* '89, pages 209–226. Springer-Verlag, New York, 1989.

[20] L. P. Chew, N. Chrisochoides, and F. Sukup. Parallel constrained Delaunay meshing. In *Proceedings of the Joint ASME/ASCE/SES Summer Meeting Special Symposium on Trends in Unstructured Mesh Generation*, pages 89–96, June 1997.

[21] T. M. Y. Chan, J. Snoeyink, and C.-K. Yap. Output-sensitive construction of polytopes in four dimensions and clipped Voronoi diagrams in three. In *Proceedings of the* 6*th Annual ACM–SIAM Symposium on Discrete Algorithms*, pages 282–291, 1995.

[22] J. C. Hardwick. An efficient implementation of nested data parallelism for irregular divide-and-conquer algorithms. In *Proceedings of the First International Workshop on High-Level Programming Models and Supportive Environments*, pages 105–114. IEEE, New York, April 1996.

[23] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23(2):166–204, October 1981.

[24] G. L. Miller, D. Talmor, S.-H. Teng, and N. Walkington. A Delaunay based numerical method for three dimensions: generation, formulation and partition. In *Proceedings of the* 27*th Annual ACM Symposium on Theory of Computing*, pages 683–692, May 1995.

[25] F. P. Preparata and M. I. Shamos. *Computational Geometry—an Introduction*. Springer-Verlag, New York, 1985.

[26] G. E. Blelloch and J. 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.

[27] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, MA, 1990.

[28] J. C. Hardwick. Porting a vector library: a comparison of MPI, Paris, CMMD and PVM. In *Proceedings of the* 1994 *Scalable Parallel Libraries Conference*, pages 68–77, October 1994. A longer version appears as Report CMU-CS-94-200, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[29] C. A. R. Hoare. Algorithm 63 (partition) and algorithm 65 (find). *Communications of the ACM*, 4(7):321–322, 1961.

[30] J. R. Shewchuk. Triangle: engineering a 2D quality mesh generator and Delaunay triangulator. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry*: *Towards Geometric Engineering*, volume 1148 of Lecture Notes in Computer Science, pages 203–222. Springer-Verlag, Berlin, May 1996.

[31] M. Bern, D. Eppstein, and F. Yao. The expected extremes in a Delaunay triangulation. *International Journal of Computational Geometry and Applications*, 1(1):79–91, 1991.

[32] A. Toomre. On the distribution of matter within highly flattened galaxies. *The Astrophysical Journal*, 138:385–392, 1963.

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