Implementation and Evaluation of an Efficient 2D Parallel Delaunay Triangulation Algorithm

Jonathan C. Hardwick

April 1997

CMU-CS-97-129

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

An earlier version of this paper appears in "Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures", June 1997 [23].

This work was supported in part by ARPA Contract Number DABT-63-96-C-0071, "Hierarchical, Automated Formal Hardware Verification: From Transistors to Abstract Protocols", and in part by the National Center for Supercomputing Applications, who provided the use of an SGI Power Challenge under grant number ACS930003N, and in part by the Pittsburgh Supercomputer Center, who provided the use of a Cray T3D and a DEC AlphaCluster under grant number SCTJ8LP, and in part by the Phillips Laboratory, Air Force Material Command, USAF, through the use of an IBM SP2 at the Maui High Performance Computing Center under cooperative agreement number F29601-93-2-0001.

The view and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of ARPA, the Phillips Laboratory, or the U.S. Government.

Keywords: Two-dimensional Delaunay triangulation, parallel algorithm, divide-and-conquer, nested data parallel, Machiavelli, MPI

Abstract

This paper describes the derivation of an empirically efficient parallel two-dimensional Delaunay triangulation program from a theoretically efficient CREW PRAM algorithm. Compared to previous work, the resulting implementation is not limited to datasets with a uniform distribution of points, achieves significantly better speedups over good serial code, and is widely portable due to its use of MPI as a communication mechanism. Results are presented for a loosely-coupled cluster of workstations, two distributed-memory multicomputers, and a shared-memory multiprocessor. The Machiavelli toolkit used to transform the nested data parallelism inherent in the divide-and-conquer algorithm into achievable task and data parallelism is also described and compared to previous techniques.

1 Introduction

Delaunay triangulation represents an important substep in many computationally-intensive applications, including pattern recognition, terrain modelling, and mesh generation for the solution of partial differential equations. Delaunay triangulations and their duals, Voronoi diagrams, are among the most widely-studied structures in computational geometry. Voronoi diagrams have also appeared in many other fields under different names [2]; *domains of action* in crystallography, *Wigner-Seitz zones* in metallurgy, *Thiessen polygons* in geography, and *Blum's transforms* in biology. This paper assumes that the reader is familiar with the basic definition of Delaunay triangulation in R^2, namely the unique triangulation of a set S of points such that there are no elements of S within the circumcircle of any triangle.

There are many well-known serial algorithms for Delaunay triangulation. The best have been extensively analyzed [17, 36], and implemented as general-purpose libraries [4, 33]. 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 serial machines. However, although several parallel algorithms for Delaunay triangulation have been described [1, 32, 13, 27, 20], practical implementations have been slower to appear. One reason is that the dynamic nature of the problem can result in significant inter-processor communication. Performing key phases of the algorithm on a single processor (for example, serializing the merge step of a divide-and-conquer algorithm, as in [38]) reduces this communication, but introduces a serial bottleneck that severely limits scalability in terms of both parallel speedup and achievable problem size. The use of decomposition techniques such as bucketing [28, 11, 37, 35], or striping [14] can also reduce communication, but relies on the input dataset having a uniform spatial distribution of points in order to avoid load imbalances between processors. Unfortunately, while most real-world problems are not this uniform, few authors report the performance of their implementations on non-uniform datasets. Of those that do, the 3D algorithm by Teng et al [37] was up to 5 times slower on non-uniform datasets than on uniform ones (on a 32-processor CM-5), while the 3D algorithm by Cignoni et al [11] was up to 10 times slower on non-uniform datasets than on uniform ones (on a 128-processor nCUBE). The 2D algorithm by Ding and Densham [15] is designed to be able to handle non-uniform datasets, but has only been demonstrated to scale to 2 processors.

A second problem is that the parallel algorithms are typically much more complex than their serial counterparts. This added complexity results in low *parallel efficiency*; that is, they achieve only a small fraction of the perfect speedup over good serial code running on one processor. Again, direct comparison is difficult because few authors quote speedups over good serial code. Of those that do, the 2D algorithm by Su achieved speedup factors of 3.5–5.5 on a 32-processor KSR-1 [35], for a parallel efficiency of 11–17%, while the 3D algorithm [28] by Merriam 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 datasets. The 2D algorithm by Chew et al [10] (which solves the more general problem of constrained Delaunay triangulation in a meshing algorithm) achieves speedup factors of 3 on an 8-processor SP2, but currently requires that the boundaries between processors be created by hand.

Blelloch, Miller and Talmor recently developed a CREW PRAM algorithm that does not rely on bucketing and hence can efficiently handle non-uniform datasets [8]. It is divide-and-conquer in style but uses a "marriage before conquest" approach to eliminate the expensive merge step that has hindered previous parallel algorithms. Additionally, when prototyped in the nested data-parallel language NESL [7], the algorithm was found to perform only twice as many floating-point operations as a good serial algorithm.

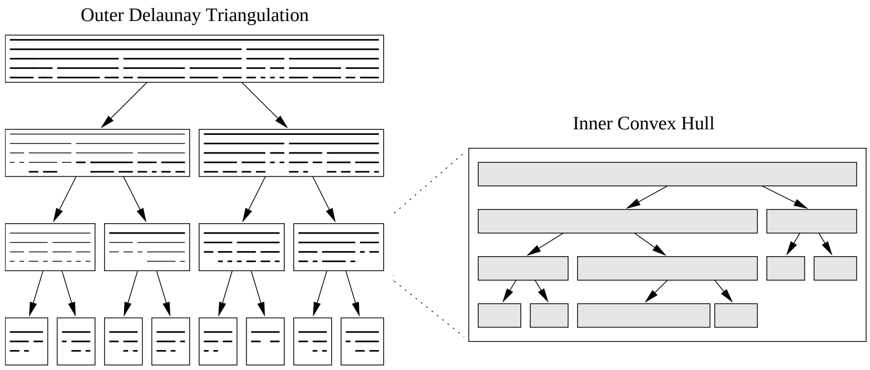


Figure 1: Nested recursion in Delaunay triangulation algorithm by Blelloch et al [8]. Each recursive level of the outer divide-and-conquer triangulation algorithm, which has a perfect split, uses as a substep a divide-and-conquer convex hull algorithm which may have a much more irregular structure.

This paper describes a practical parallel Delaunay triangulation program which uses the algorithm by Blelloch et al as a coarse parallel partitioner, switching to an efficient implementation of Dwyer's serial algorithm provided by the Triangle package [33] at the leaves of the recursion tree. The program was parallelized using the Machiavelli toolkit [24], which has been designed both for the direct implementation of parallel divide-and-conquer algorithms (as in this case), and as an implementation layer for nested data-parallel languages. It is particularly well-suited to exploiting the nested divide-and-conquer nature of the algorithm by Blelloch et al, which uses an inner quickhull algorithm as a substep, as shown in Figure 1.

Since Machiavelli uses MPI [18] as a communication mechanism, the resulting Delaunay triangulation program is more portable than most previous implementations, which have used vendor-specific message-passing libraries [14, 28, 11, 37] or shared memory directives [35]. We present results for a loosely-coupled DEC AlphaCluster, a distributed-memory IBM SP2, a distributed-memory Cray T3D, and a shared-memory SGI Power Challenge. Experimentally, the program achieves three times the parallel efficiency of previous implementations, and is at most 1.5 times slower on non-uniform datasets than on uniform ones.

The rest of the paper is arranged as follows. Section 2 describes the Machiavelli toolkit, and Section 3 outlines the parallel algorithm by Blelloch et al. Section 4 discusses key implementation decisions. Section 5 presents and analyzes performance results for a range of input distributions. Finally, Section 6 concludes the paper.

2 The Machiavelli Toolkit

Many of the most efficient and widely used computer algorithms use a divide-and-conquer approach to solving a problem. Examples include binary search, quicksort [26], and fast matrix multiplication [34]. The subproblems that are generated can typically be solved independently, and hence divide-and-conquer algorithms have long been recognized as presenting a potential source of parallelism. This has resulted in many architectures and parallel programming languages being designed specifically for the implementation of divide-and-conquer algorithms (see [3] for a survey). However, previous parallel divide-and-conquer models have typically been limited to *regular* algorithms, in which the subproblems are of equal size. This excludes a very useful class of algorithms;

for example, quicksort, selection, and many computational geometry algorithms all have an irregular divide-and-conquer structure. *Concatenated parallelism* is a notable recent exception that can handle irregular divide-and-conquer algorithms, but can only outperform a task-parallel approach when the communication cost of redistributing the data is significant compared to the computational cost of subdividing the task [19]. The alternative approach of using a more general language model to handle irregular algorithms runs the risk of hiding divide-and-conquer parallelism that would otherwise be easy to exploit. For example, although nested data-parallel languages such as NESL [7] and Proteus [31] are well-suited for expressing irregular divide-and-conquer algorithms, their current implementation layer assumes a vector PRAM model [6]. This can be efficiently implemented on vector processors with high memory bandwidth, but it is harder to do so on current RISC-based NUMA multiprocessor architectures, due to the higher relative costs of communication and poor data locality [21].

Machiavelli [24] is a new parallel toolkit for divide-and-conquer algorithms that is intended to alleviate some of these problems. It is designed to be usable both as an implementation layer for languages such as NESL, and as a programmer's toolkit for the direct implementation of efficient parallel programs. There were three main goals in Machiavelli's development. First, it must be capable of handling divide-and-conquer algorithms that are both irregular and *nested*; that is, algorithms that use a different divide-and-conquer algorithm as a substep. Second, it must be portable across parallel architectures. Finally, it must be efficient; that is, it must result in parallel programs that show significant speedup over good serial programs.

To achieve the first goal, Machiavelli uses recursive subdivision of asynchronous teams of processors running SPMD code to directly implement the behavior of a divide-and-conquer algorithm (this can be seen a generalized version of the technique used by Barnard's spectral bisection algorithm on the Cray T3D [5]). To achieve the second goal, Machiavelli is implemented using C and MPI (the standard Message Passing Interface [18]). To achieve the third goal, Machiavelli obtains parallelism from both data-parallel operations within teams and from the task-parallel invocation of recursive functions on different teams, and uses good serial code where possible.

The Machiavelli toolkit currently consists of a library of vector primitives and a small run-time system. The library implements the basic communication and team functions in terms of MPI; as an example, Figure 2 shows source code for the team-splitting phase of the Delaunay triangulation program. The run-time system adds the ability to perform dynamic load-balancing for irregular algorithms. Specifically, it can ship a recursive serial function call to an idle processor in order to redistribute computation [22].

A Machiavellian divide-and-conquer program consists of both serial and SPMD parallel code. The parallel code operates at the upper levels of recursion, and uses calls to the library to redistribute data and subdivide the problem. Initially, data is distributed in block fashion across all the processors. The processors act as a data-parallel team, synchronizing with each other only when necessary to exchange data. At the first recursive call, the initial team splits into two new teams, with the relative number of processors in each new team being chosen to approximate the relative cost of each of the sub-problems. The teams repeat the process, recursing down to smaller and smaller teams, which run asynchronously with respect to each another. When a team contains a single processor, that processor switches to serial code. Note that this can be a more efficient serial algorithm, rather than just a serial translation of the parallel algorithm. When the serial code finishes, parallel teams are reformed and results are combined on the way back up the recursion call tree.

Within the Machiavelli toolkit, data is expressed as private scalars and distributed vectors. For communication, teams are mapped to MPI groups, and user-defined types are mapped to MPI's

```
vec_point parallel_DT (team T, vec_point P, vec_border B)
{
  team T_new;
  vec_point P_L, P_R, P_new, result;
  vec_border B_L, B_R, B_new;

  ... /* Compute P_L, P_R, B_L, B_R */ ...

  /* Create two new teams according to ratio of
   * subproblem sizes, join one of them */
  T_new = split_teams (T, P_L->len, P_R->len);

  /* Split P_L/P_R and B_L/B_R between the teams */
  P_new = split_vec_point (P_L, P_R, T, T_new);
  B_new = split_vec_border (B_L, B_R, T, T_new);
  free (P_L); free (P_R); free (B_L); free (B_R);

  /* Is my new team just a single processor? */
  if (T_new->nproc == 1) {
    /* Then run serial Delaunay triangulation */
    result = serial_DT (P_new, B_new);
  } else {
    /* Else recurse in parallel code in new team */
    result = parallel_DT (T_new, P_new, B_new);
  }
  /* Teams rejoin here */
  return result;
}
```

Figure 2: Team-splitting in Delaunay triangulation program (see Figure 3), expressed as SPMD C code with calls to Machiavelli library.

derived datatypes [18]. A purely data-parallel operation (such as an elementwise mathematical function on a vector) is implemented as a loop over the appropriate section of data on each processor. More complex functions, such as finding the largest element in a vector, involve both a local loop and an MPI reduction operation within the current team. Finally, vector communication functions are implemented using MPI's all-to-all communication primitives. Optimized library functions are provided for many of the idioms that occur in divide-and-conquer algorithms, such as sending subsets of a vector to sub-teams, and merging data from sub-teams into a single vector.

3 Delaunay Triangulation Algorithm

This section briefly outlines use of the parallel Delaunay triangulation algorithm by Blelloch, Miller, and Talmor as a coarse partitioner. The algorithm uses the well-known reduction of two-dimensional Delaunay triangulation to the problem of finding the three-dimensional convex hull of points on a sphere or paraboloid. The resulting algorithm is divide-and-conquer in nature but uses a "marriage before conquest" approach, similar to the DeWall triangulation algorithm [12], which enables it to avoid an expensive merge step. See [8] for more details, and `http://web.scandal.cs.cmu.edu/cgi-bin/demo` for an interactive demonstration.

Algorithm: ParDel(P, B, T)

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, and T, a team of processors.

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

Method:

1. If $|T| = 1$, return Serial(P, B).
2. Find the point q that is the median along the x axis of all internal points (that is, points in P that are not on the boundary B). Let $\mathcal{L}$ be the line $x = q_x$.
3. Let $P' = \{(p_y - q_y, ||p - q||^2) \mid (p_x, p_y) \in P\}$, derived from projecting the points P onto a $3D$ paraboloid centered at q, and then 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 on the path $\mathcal{H}$, and $\bar{\mathcal{H}}$ be the path $\mathcal{H}$ traversed in the opposite direction.
5. Create the input for left and right subproblems:

$$\begin{array}{rcl}
B^L & = & \textsc{Border_Merge}(B, \mathcal{H}) \\
B^R & = & \textsc{Border_Merge}(B, \bar{\mathcal{H}}) \\
P^L & = & \{p \in P \mid p \text{ is left of } \mathcal{L}\} \cup \{p' \in P_{\mathcal{H}} \mid p' \text{ contributed to } B^L\} \\
P^R & = & \{p \in P \mid p \text{ is right of } \mathcal{L}\} \cup \{p' \in P_{\mathcal{H}} \mid p' \text{ contributed to } B^R\} \\
T^L & = & \text{subset of } T \text{ of size } |T||P^L|/(|P^L| + |P^R|) \\
T^R & = & T - T^L
\end{array}$$

6. Return ParDel$(P^L, B^L, T^L) \cup$ ParDel(P^R, B^R, T^R).

Figure 3: Recursive divide-and-conquer SPMD pseudocode for Delaunay triangulation, using the algorithm by Blelloch et al [8] as a coarse partitioner (a correction has been made to step 5). Although the algorithm uses alternating x and y cuts, for simplicity only the x cut is shown. The three subroutines Serial, Lower_Convex_Hull, and Border_Merge are described in the text.

Pseudocode for the algorithm is shown in Figure 3. It has three important substeps:

Serial Delaunay: Although any serial Delaunay triangulation algorithm can be used for the base case, Dwyer's [16] is recommended since it has been shown experimentally to be the fastest [36, 33].

Lower convex hull: The lower half of the convex hull of the projected points is used to find a new path $\mathcal{H}$ that divides the problem into two smaller problems. Since the projection is based on the median point, the division is perfect, as shown in Figure 1.

Border merge: This routine takes the old border B and merges it with the newly-found dividing path $\mathcal{H}$ to form a new border for a recursive call. The new border is computed based on an inspection of the possible intersections of points in B and $\mathcal{H}$.

3.1 Theoretical Performance

The full Delaunay triangulation algorithm by Blelloch et al requires $O(n \log n)$ parallel work and $O(\log^3 n)$ parallel depth on a CREW PRAM (i.e., a total time of $O((n \log n)/P + \log^3 n)$ on P processors). The work complexity is optimal for Delaunay triangulation, and the depth complexity is practical for parallelization purposes.

Note that these complexities assume that the lower convex hull substep is solved using a linear-work algorithm, which is possible since we can store the points in sorted order [29]. However, Blelloch et al found experimentally that a simple quickhull [30] was faster than a more complicated convex hull algorithm that was guaranteed to take linear time. Furthermore, using a point-pruning version of quickhull that limits possible imbalances between recursive calls [9] reduces its sensitivity to non-uniform datasets.

With these changes, the parallel Delaunay triangulation algorithm was found to perform about twice as many floating-point operations as Dwyer's algorithm [16]. Furthermore, the cumulative floating-point operation count was found to increase uniformly with recursion depth, indicating that the algorithm should be usable as a partitioner without loss of efficiency.

However, as implemented in NESL, the algorithm was an order of magnitude slower on one processor than a good serial algorithm. It was chosen as a test case for Machiavelli due to its recursive divide-and-conquer nature, the natural match of the partitioning variant to Machiavelli's ability to use efficient serial code, and its nesting of a recursive convex hull algorithm within a recursive Delaunay triangulation algorithm, as shown in Figure 1.

4 Implementation

This section describes several implementation decisions and optimizations that affect the performance of the final program, including using the right data structures, improving the performance of specific algorithmic substeps, and using a general optimization that can be applied to many Machiavelli programs. Most of the optimizations relate to reducing or eliminating interprocessor communication. Further analysis can be found in Section 5.

Data structures The basic data structure used by the code is a *point*, represented using two double-precision floating-point values for the x and y coordinates, and two integers, one serving as a unique global identifier and the other as a communication index within team phases of the program. Points are stored in vectors, which are distributed in a block fashion across the processors of the current team. A border is composed of *corners*, each of which represents the triplet of points corresponding to two segments in a path. Corners are not balanced across the processors as points are, but rather are stored on the same processor as their "middle" point. A vector of indices I links the points in P with the corners in the borders B and H. Given these data structures, the operations of finding internal points, and projecting points onto a parabola (see Figure 3), both reduce to simple local loops.

Serial triangulation For the serial base case we use the optimized version of Dwyer's algorithm that is provided by the Triangle mesh generation package [33]. 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 serial code, and the indexed format with replication used by the parallel code. No changes are necessary to the source code of Triangle.

Finding the median Initially a parallel version of quickmedian [25] 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 contributes this local median in a collective communication step, and finally computes the median of all the medians. The result is not guaranteed to be the exact median, but in practice it is sufficiently good for load-balancing purposes; this modification increased the speed of the overall Delaunay triangulation program for the datasets and machine sizes studied (see Section 5) by 4–30%.

Finding the lower convex hull As in the original algorithm, a variant of quickhull [30] is used to find the convex hull, resulting in two nested recursive algorithms as shown in Figure 1. However, in contrast to using a different serial Delaunay triangulation algorithm, the serial code of quickhull implements the same algorithm as the parallel code.

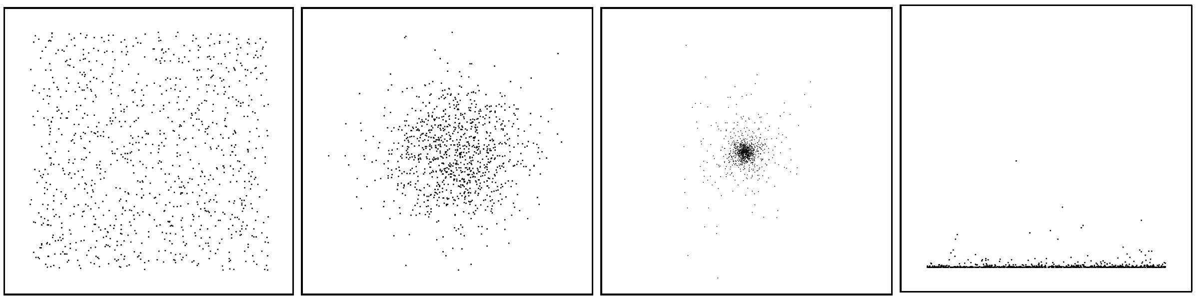
The basic quickhull algorithm tends to pick extreme "pivot" points when operating on non-uniform point distributions, resulting in a poor division of data and a consequent lack of progress. Chan et al [9] describe a variant that tests the slope between pairs of points and uses pruning to guarantee that recursive calls have at most $3/4$ 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 give better performance when used as a substep of the Delaunay triangulation program (see Section 5.4 for an analysis). As with the median-of-medians approach, the global effects of receiving approximate results from an algorithm substep are more than offset by the decrease in running time of the substep.

Combining results The quickhull algorithm concatenates the results of two recursive calls before returning. Using 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 call to Machiavelli at the top level that redistributes the serial result from each processor into a parallel vector shared by all the processors. This general Machiavelli optimization is also applied to merging the results of the Delaunay triangulation algorithm itself.

Creating the subproblems The border merge step intersects the current border B with the dividing path $\mathcal{H}$, creating two new borders B^L and B^R and two new point sets P^L and P^R. This requires a series of line orientation tests to decide how to merge corners. To eliminate an interprocessor communication phase in this step, the two outer points represented by a corner are replicated in the corner structure. 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 5.3). 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.

5 Experimental Results

The goal of this section is to validate the claims of portability, good parallel efficiency, and the ability to handle non-uniform datasets. We also analyze where the bottlenecks are, the reasons for



Uniform distribution. Normal distribution. Kuzmin distribution. Line singularity.

Figure 4: Examples of 1000 points in each of the four test distributions (taken from [8]; for clarity, the Kuzmin distribution is shown zoomed on the central point). Parallelization techniques that assume uniform distributions, such as bucketing, suffer from poor performance on the Kuzmin and line distributions.

any lack of scalability, and the effect of some of the implementation decisions presented in Section 4 on both running time and memory use.

To test portability, we used three 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 to those on one processor, when the program immediately switches to the serial Triangle package [33]. To test the ability to handle non-uniform datasets we used four different distributions taken from [8]:

Uniform distribution: The coordinates x and y are chosen at random within the unit square.

Normal distribution: The coordinates x and y are chosen as independent samples from the normal distribution.

Kuzmin distribution: This is an example of convergence to a point, and is used by astrophysicists to model the distribution of star clusters within galaxies. It is radially symmetric, with density falling rapidly with distance r from a central point. The accumulative probability function is

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

Line singularity: This is an example of convergence to a line, resulting in a distribution that cannot be efficiently parallelized using techniques such as bucketing. It is defined using a constant b (set here to 0.001) and a transformation from a uniform distribution (u, v) of

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

Examples of these distributions are shown in Figure 4. All timings represent the average of five runs using different seeds for a pseudo-random number generator. For a given problem size and seed the input data is the same regardless of the architecture or number of processors.

Processors	SP2		SGI		T3D		Alpha	
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)

Processors	SP2		SGI		T3D		Alpha	
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)

Table 1: Time taken, and relative speedup, when triangulating 128k points on the four different platforms tested. The times shown are for the irregular Kuzmin (top) and line singularity (bottom) distributions.

5.1 Analysis

To illustrate the algorithm's parallel efficiency, Figure 5 shows the time to triangulate a relatively small problem (128k points) on different numbers of processors, for each of the three platforms and the four different datasets. Speedup is not perfect because as more processors are added, more levels of recursion are spent in parallel code rather than in the more efficient serial code. However, we still achieve approximately 50% parallel efficiency for the datasets and machine sizes tested—that is, we achieve about half of the perfect speedup over good serial code. Additionally, the Kuzmin and line distributions show similar speedups to the uniform and normal distributions, suggesting that the algorithm is effective at handling non-uniform datasets 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 150MHz Alpha 21064 processors, and their single-processor times are thus comparable. However, the T3D's 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 6 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 well with problem size, as we would expect from an $O(n \log n)$ algorithm.

To illustrate the relative costs of the different components of the algorithm, Figure 7a 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 7b 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 8 uses a parallel time line to show the activity of each processor when triangulating a line singularity dataset. There are several important effects that can be seen here. First, the

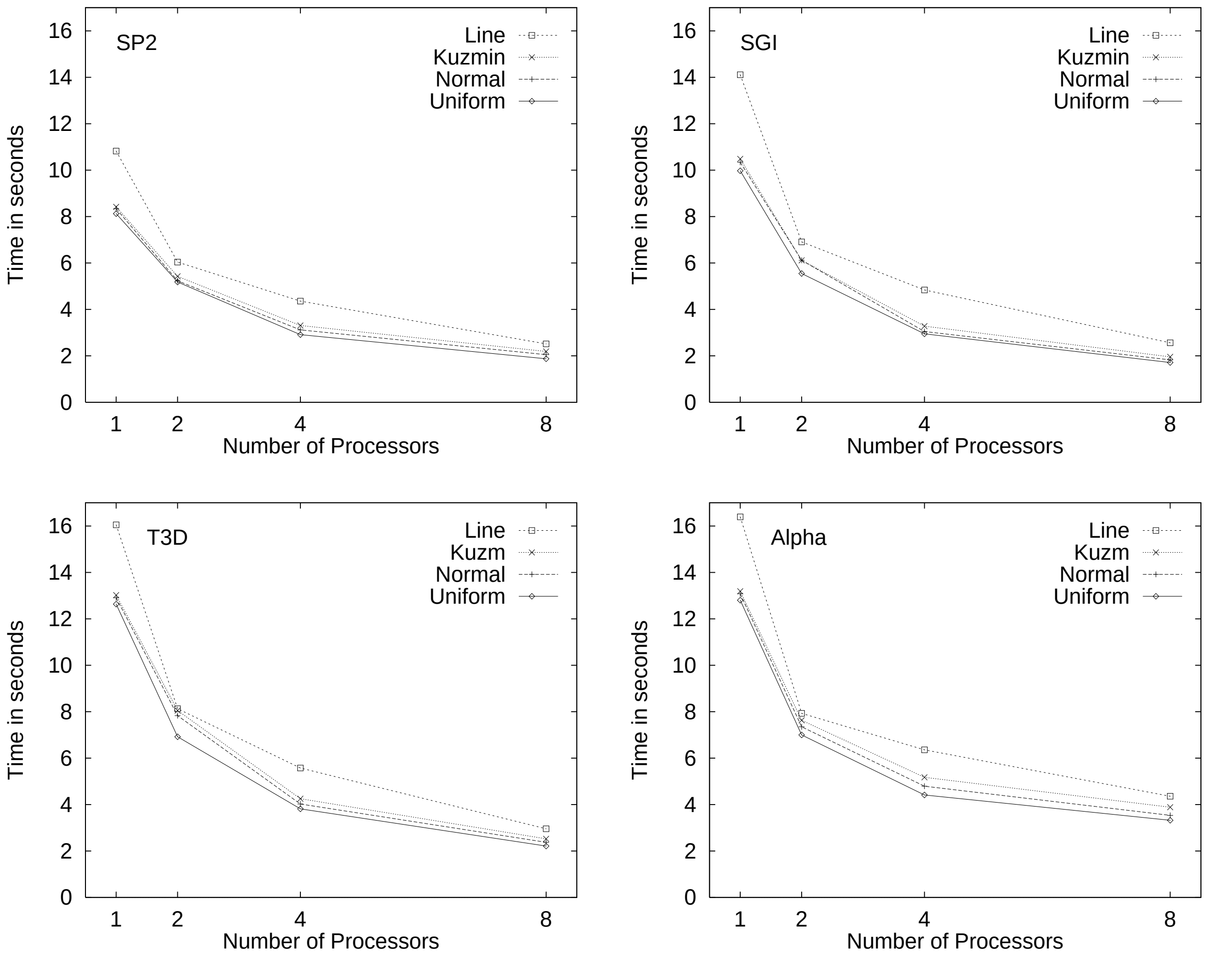


Figure 5: Speedup of Delaunay triangulation program for four input distributions and four parallel architectures. The graphs show the time to triangulate a total of 128k points as the number of processors is varied. Single processor results are for good serial code (Triangle [33]). Increasing the number of processors results in more levels of recursion being spent in slower parallel code rather than faster serial 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.

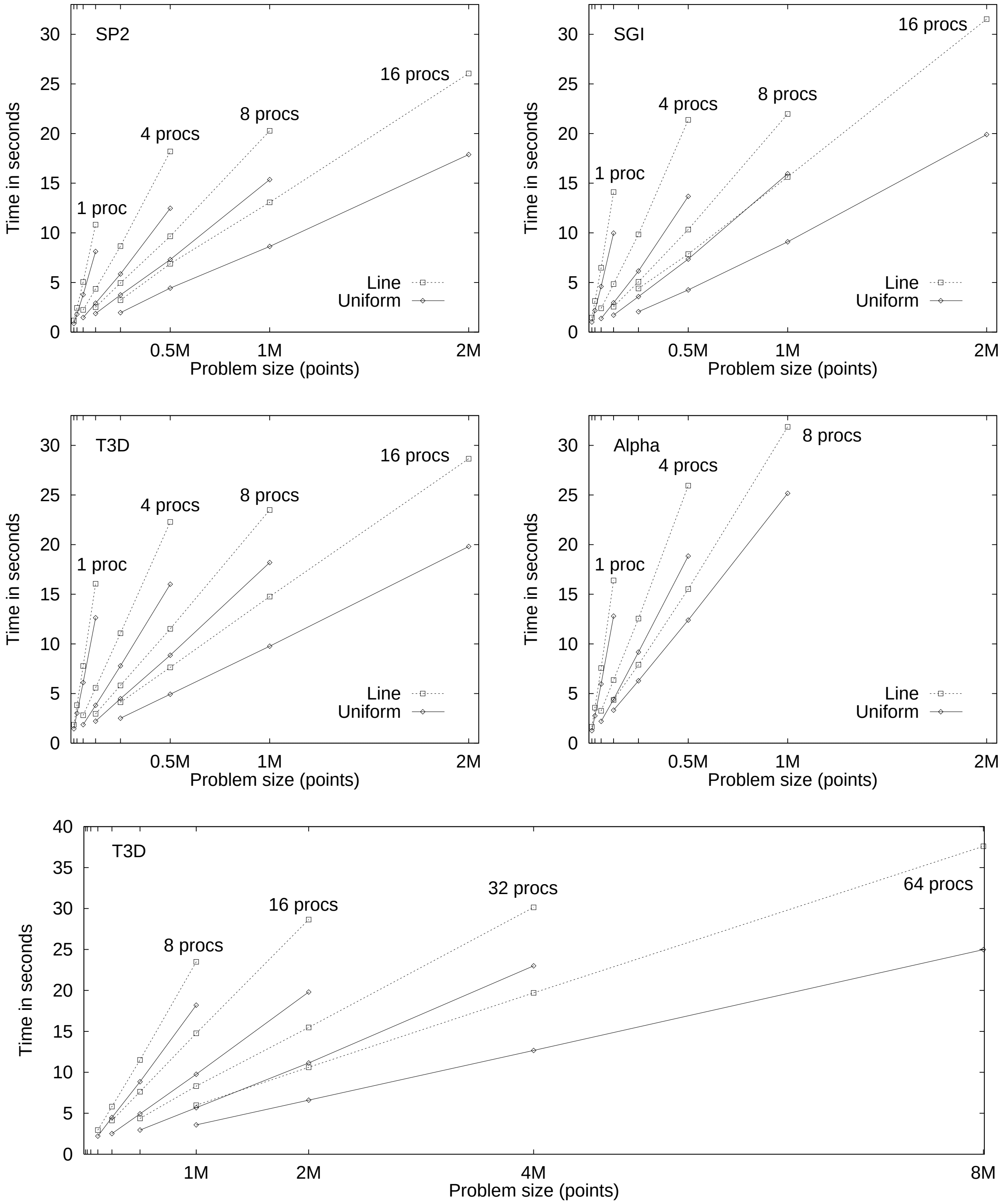
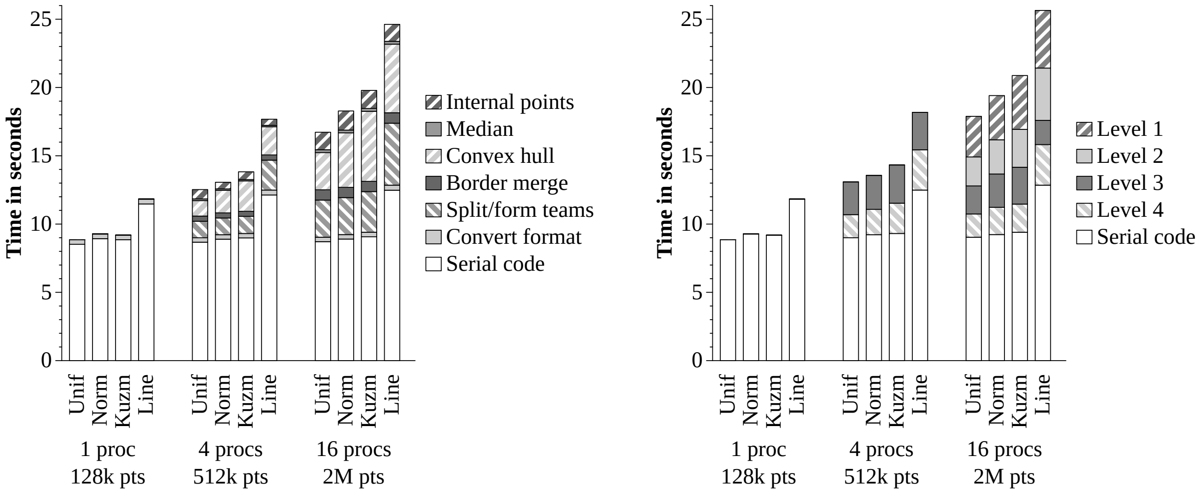


Figure 6: Scalability of Delaunay triangulation program for four input distributions and four parallel architectures. The graphs show the time to triangulate 16k-128k points per processor as the number of processors is varied (for clarity, only the fastest and slowest distributions are shown for a given number of processors). Machine setups are as in Figure 5. An additional graph for the Cray T3D shows scalability up to 64 processors.



(a) Total time in each substep (b) Time in each recursive level

Figure 7: Two views of the execution time as the problem size is scaled with number of processors (IBM SP2, 128k points per processor). (a) shows the total time spent in each substep of the algorithm. The time spent in serial code remains approximately constant, while convex hull and team operations (which includes synchronization delays) are the major overheads in the parallel code. (b) shows the time per recursive level of the algorithm; note the approximately constant overhead per level.

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 serial 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 non-uniform dataset.

5.2 Performance Prediction

The total running time of the program for a problem of size n on P processors can be calculated as the sum of the serial and parallel components per processor. The serial time (that is, the time spent in Triangle) is $O((n \log n)/P)$. The number of parallel phases is equal to the logarithm of the number of processors, and the time spent in each phase is again $O((n \log n)/P)$, so that the parallel time is $O((n \log n)(\log P)/P)$. Combining these into an equation to predict running time, we have:

$$T(n, P) = \frac{(n \log_2 n)(k_1 + k_2 \lceil \log_2 P \rceil)}{P}$$

Note that this simplified formula ignores overheads dependent only on P, since for problem sizes of interest these overheads are insignificant compared to the combined effects of n and P (in practical terms, the time taken to send and receive large amounts of data far outweighs the fixed latency of the messages). We can then measure the parameters k_1 and k_2 for a specific distribution and architecture in order to be able to predict the running time for a given problem size on a

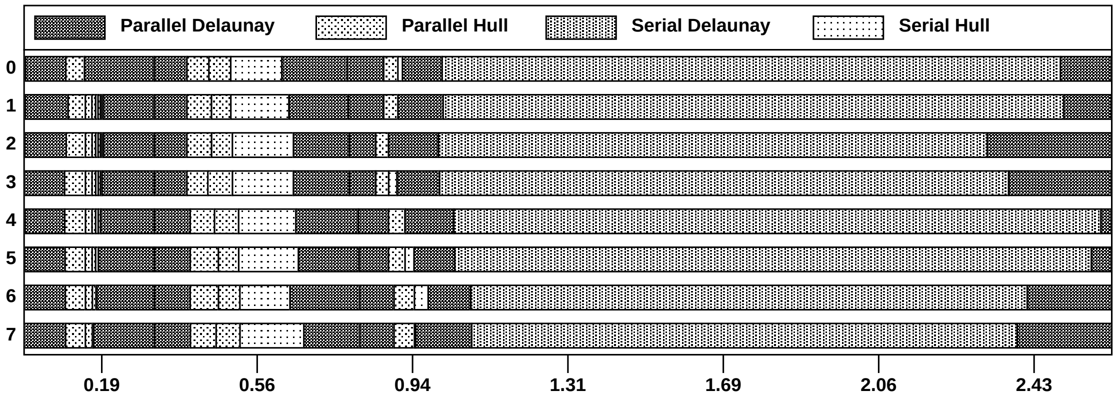


Figure 8: Activity of eight processors over time, showing the parallel and serial phases of Delaunay triangulation and its inner convex hull algorithm (IBM SP2, 128k points in a line singularity distribution). A parallel step level of two phases of Delaunay triangulation code surrounding one or more convex hull phases; this run has three parallel levels. Despite the non-uniform dataset the processors do approximately the same amount of serial work.

given number of processors. For example, for a uniform distribution on the SGI Power Challenge, $k_1 = 4.4 \times 10^{-6}$ s. and $k_2 = 6.0 \times 10^{-7}$ s. predicts running times to within 7%.

5.3 Memory Requirements

As explained in Section 4, border points are replicated in corner structures to eliminate the need for global communication in the border merge step. An obvious objection to this approach is that it increases the memory requirements of the program. Assuming 64-bit doubles and 32-bit integers, a point (two doubles and two integers) and associated index vector entry (two integers) occupies 32 bytes, while a corner (two replicated points) occupies 48 bytes. However, since a border is composed of only a small fraction of the total number of points, the additional memory required to hold the corners is relatively small. For example, in a run of 512k points in a line singularity distribution on eight processors, the maximum ratio of corners to total points on a processor (which occurs at the switch between parallel and serial code) is approximately 2,000 to 67,000, so that the corners 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 2,000 to 17,500. Even here, however the corners still represent less than 15% of required storage, and by reducing the number of points per processor we have also reduced absolute memory requirements.

5.4 Performance of Convex Hull Variants

Finally, we investigate the performance of the convex hull variants described in Section 4. The final $\sqrt{n}$-pair pruning quickhull was benchmarked against both a basic quickhull and the original n-pair pruning quickhull by Chan et al [9]. Results for an extreme case are shown in Figure 9. As can be seen, the n-pair algorithm is more than twice as fast as the basic quickhull on the non-uniform Kuzmin dataset (over all the datasets and machine sizes tested it was a factor of 1.03–2.83 faster). The $\sqrt{n}$-pair algorithm provides a modest additional improvement, being a factor of 1.02–1.30 faster than the n-pair algorithm.

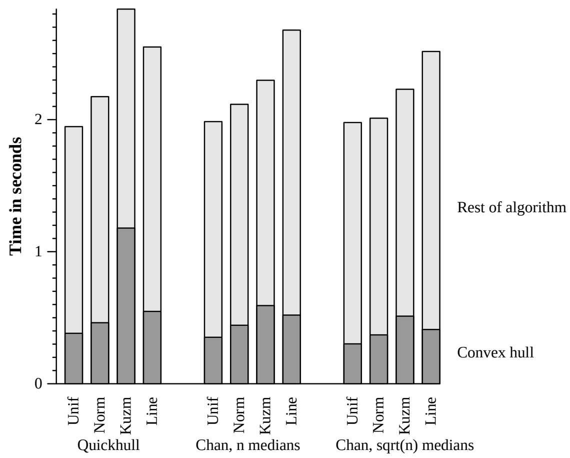


Figure 9: Effect of different convex hull functions on time to triangulate 128k points on an 8-processor IBM SP2. The pruning quickhull due to Chan et al [9] has much better performance than the basic algorithm on the non-uniform Kuzmin distribution; reducing its sampling accuracy produces a modest additional improvement.

6 Conclusions

This paper has described the use of the Machiavelli toolkit to produce a fast and practical parallel two-dimensional Delaunay triangulation algorithm. The code was derived from a combination of a theoretically efficient CREW PRAM parallel algorithm and existing optimized serial code. The resulting program has three advantages over existing work. First, it is widely portable due to its use of MPI; it achieves similar speedups on three machines with very different communication architectures. Second, it can handle datasets that do not have a uniform distribution of points with a relatively small impact on performance. Specifically, it is at most 1.5 times slower on non-uniform datasets than on uniform datasets, whereas previous implementations have been up to 10 times slower. Finally, it has good absolute performance, achieving a parallel efficiency (that is, percentage of perfect speedup over good serial code) of greater than 50% on machine sizes of interest. Previous implementations have achieved at most 17% parallel efficiency.

In addition to these specific results, there are two more general conclusions that are well-known but bear repeating. First, constants matter: simple algorithms are often faster than more complex algorithms that have a lower complexity (as shown in the case of quickhull). Second, quick but approximate algorithmic substeps often result in better overall performance than slower approaches that make guarantees about the quality of their results (as shown in the case of the median-of-medians and $\sqrt{n}$-pair pruning quickhull).

7 Acknowledgements

Dafna Talmor wrote the original NESL code and provided invaluable help in understanding the algorithm. Guy Blelloch gave guidance and suggestions throughout the work. Jonathan Shewchuk wrote the Triangle package and gave feedback on an earlier version of the paper.

References

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

[2] Franz Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, September 1991.

[3] Tom H. Axford. The divide-and-conquer paradigm as a basis for parallel language design. In *Advances in Parallel Algorithms*, chapter 2. Blackwell, 1992.

[4] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software*, 22(4):469–483, December 1996.

[5] Stephen T. Barnard. PMRSB: Parallel multilevel recursive spectral bisection. In *Proceedings of Supercomputing '95*. ACM, December 1995.

[6] Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, 1990.

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

[8] Guy E. Blelloch, Gary L. Miller, and Dafna Talmor. Developing a practical projection-based parallel Delaunay algorithm. In *Proceedings of the 12th Annual Symposium on Computational Geometry.* ACM, May 1996.

[9] 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 *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 282–291, 1995.

[10] L. Paul Chew, Nikos Chrisochoides, and Florian Sukup. Parallel constrained Delaunay meshing. In *Proceedings of the Joint ASME/ASCE/SES Summer Meeting Special Symposium on Trends in Unstructured Mesh Generation*, June 1997. To appear.

[11] 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, November 1993.

[12] P. Cignoni, C. Montani, R. Perego, and R. Scopigno. Parallel 3D Delaunay triangulation. In *Proceedings of the Computer Graphics Forum (Eurographics '93)*, pages 129–142, 1993.

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

[14] 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, 1989.

[15] Yuemin Ding and Paul J. Densham. Dynamic and recursive parallel algorithms for constructing Delaunay triangulations. In *Proceedings of the Sixth International Symposium on Spatial Data Handling*, pages 682–696. Taylor & Francis, 1994.

[16] 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 2nd Annual Symposium on Computational Geometry*, pages 276–284. ACM, May 1986.

[17] Steven Fortune. Voronoi diagrams and Delaunay triangulations. In Ding-Zhu Du and Frank Hwang, editors, *Computing in Euclidean Geometry*, pages 193–233. World Scientific, 1992.

[18] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputing Applications and High Performance Computing*, 8(3/4), 1994.

[19] Sanjay Goil, Srinivas Aluru, and Sanjay Ranka. Concatenated parallelism: A technique for efficient parallel divide and conquer. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing '96*, pages 488–495, 1996.

[20] Sumanta Guha. An optimal mesh computer algorithm for constrained Delaunay triangulation. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 102–109. IEEE, April 1994.

[21] Jonathan 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.

[22] Jonathan 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*, April 1996.

[23] Jonathan C. Hardwick. Implementation and evaluation of an efficient parallel Delaunay triangulation algorithm. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1997. To appear.

[24] Jonathan C. Hardwick. *Practical Parallel Divide-and-Conquer Algorithms*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997. To appear.

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

[26] C.A.R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–15, 1962.

[27] J. Andrew Holey and Oscar H. Ibarra. Triangulation, Voronoi diagram, and convex hull in k-space on mesh-connected arrays and hypercubes. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume III, Algorithms & Applications. CRC Press, August 1991.

[28] Marshal L. Merriam. Parallel implementation of an algorithm for Delaunay triangulation. In *Proceedings of Computational Fluid Dynamics*, volume 2, pages 907–912, September 1992.

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

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

[31] Jan Prins and Daniel Palmer. Transforming high-level data-parallel programs into vector operations. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 119–128. ACM, May 1993.

[32] John H. Reif and Sandeep Sen. Polling: A new randomized sampling technique for computational geometry. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 394–404, 1989.

[33] Jonathan Richard Shewchuk. Triangle: Engineering a 2D quality mesh generator and delaunay triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996.

[34] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(3):354–356, 1969.

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

[36] Peter Su and Robert L. Drysdale. A comparison of sequential Delaunay triangulation algorithms. In *Proceedings of the 11th Annual Symposium on Computational Geometry*, pages 61–70. ACM, June 1995.

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

[38] N.A. Verhoeven, N.P. Weatherill, and K. Morgan. Dynamic load balancing in a 2D parallel Delaunay mesh generator. In *Parallel Computational Fluid Dynamics*, pages 641–648. Elsevier Science Publishers B.V. (North–Holland), June 1995.