In *Proc. 3rd Symp. Frontiers Massively Parallel Computation* (1990), pp. 8–11.

Asymptotically Efficient Hypercube Algorithms for Computational Geometry

Preliminary Version

Philip D. MacKenzie[1] and Quentin F. Stout[2]

Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI

Abstract

We present new hypercube algorithms which solve many fundamental computational geometry problems. These algorithms use decomposition techniques which enable them to asymptotically outperform the fastest previous algorithms for these problems. Previous algorithms all run in $\Theta(\log^2 n)$ time, even when using a sorting method which runs in $o(\log^2 n)$ time. The algorithms herein are able to use a recently discovered $o(\log^2 n)$ time sorting method to improve their asymptotic speed to $o(\log^2 n)$. If sorting runs in $\Theta(\mathrm{Sort}(n))$ time, our algorithms for 2-Set Dominance Counting, 3-Dimensional Maxima, Closest pair, and All Points Nearest Neighbors run in $\Theta(\mathrm{Sort}(n) \cdot \log\log n)$ time, and our algorithms for Triangulation and Visibility from a Point run in $\Theta(\mathrm{Sort}(n))$ time.

1 Introduction

The field of parallel computational geometry has seen rapid growth in recent years, both on distributed memory architectures such as the mesh and the hypercube, and on shared memory computers, or Parallel Random Access Machines (PRAMs). For the mesh and the PRAM optimal algorithms have been developed for many of the fundamental problems in computational geometry. On a square mesh many problems can be solved in $\Theta(\sqrt{n})$ time [7, 10, 11], and since this is the diameter of the mesh, there is no possibility for asymptotic improvement. On the PRAM, by using logarithmic sorting algorithms, most of the fundamental problems in computational geometry can be solved with linear speedup [1, 2, 5]. This is optimal, though the constants are large, so again there is no possibility for asymptotic improvement beyond lowering the constants.

The state of computational geometry on the hypercube, however, has been quite different. The lower bound for computational geometry algorithms on the hypercube has not been the diameter or the serial lower bounds, but the time to sort. In fact, Batcher's $\Theta(\log^2 n)$ time Bitonic Sort algorithm had the best asymptotic time of any hypercube sorting algorithm for many years. Therefore, an algorithm which relied on sorting ran in $\Omega(\log^2 n)$ time on a hypercube. Since many of the fundamental computational geometry problems have very straightforward $\Theta(\log^2 n)$ time hypercube solutions [8, 11, 12], there was little incentive to improve these algorithms.

[1]Research supported by an AT&T Fellowship.

[2]Research partially supported by a joint NSF-DARPA grant.

Recently, however, an asymptotically faster hypercube sorting algorithm called ShareSort was developed by Cypher and Plaxton [6]. The algorithm sorts n elements on an n processor hypercube in $\Theta(\log n(\log\log n)^2)$ time. Unfortunately, simply exchanging ShareSort for Bitonic Sort in the aforementioned algorithms will not produce asymptotically faster algorithms. One could say these algorithms are *inherently* $\Theta(\log^2 n)$ time algorithms. We develop new algorithms for these problems and show that they can use ShareSort to run in $o(\log^2 n)$ time.

We must emphasize that these new algorithms are asymptotically faster, but the constants are very large when compared to the simple, inherently $\Theta(\log^2 n)$ algorithms. Practical hypercube algorithms for these problems appear in [8, 11].

Given two points p and q in d-dimensional space, p is said to *dominate* q if each coordinate of p is larger than the corresponding coordinate of q. We give new hypercube algorithms for the following geometric problems:

2-Set Dominance Counting Given a set S of m points and a set T of k points, find for each point $p \in S$ the number of points in T dominated by p, and find for each point $q \in T$ the number of points in S dominated by q. We assume $n = m + k$.

3-Dimensional Maxima Given a set S of n points in 3-dimensional space, determine all points $p \in S$ such that no other point of S dominates p.

Closest Pair Given a set S of n points in the plane, determine a pair of points in S which are closest to each other in the Euclidean metric.

All Points Nearest Neighbors Given a set S of n points in the plane, for each point $p \in S$ find a point in $S - \{p\}$ that is closest to p in the Euclidean metric.

Triangulation Given a set S of n points in the plane, join the points of S by nonintersecting straight line segments such that every region internal to the convex hull of S is a triangle.

Visibility from a Point Given n line segments (possibly intersecting) and a point p, determine that part of the plane visible from p if all segments are opaque. An interesting special case of this problem is to determine the **Skyline** of a set of n rectangles resting on a horizontal line. In this case, the line segments are the upper edges of the rectangles and the point p is $(0, \infty)$.

The problem of 2-Dimensional Maxima can be solved trivially in $\Theta(\mathrm{Sort}(n))$ time using a sort and a prefix operation, and thus is not considered here. Also, Miller and Stout [9] showed that the convex hull of a set of planar points can be found in $\Theta(\mathrm{Sort}(n))$ time. Slower hypercube algorithms for some of the above problems, taking $\Theta(\log^2 n)$ time, appear in [4, 8, 11, 12]. Optimal PRAM algorithms for these problems appear in [1, 3, 5, 13, 15].

Throughout we assume that we are using an n processor hypercube with the following characteristics. Each processor has a unique $\log_2(n)$-bit ID (a number from 0 to $n - 1$), and two processors are neighbors (share a communication link) if their IDs differ in exactly one bit. In one time step, each processor can either send one word to a neighbor, receive one word from a neighbor, or perform a single operation on its data.

We also assume each processor has a constant amount of memory, except in the algorithm for Visibility from a Point with intersecting line segments, in which we assume each processor has a memory of size $\Omega(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann's function, a very slow growing function of n. This is needed because the output in this case can be of size $\Theta(n\alpha(n))$.

2 Fundamental Algorithms

We will use many known algorithms as subroutines. One is the $\Theta(\log n(\log\log n)^2)$ time ShareSort algorithm mentioned previously. Others, listed below, are simpler and run in $\Theta(\log n)$ time. See [11] for detailed implementations.

Prefix Assume each processor i contains the value a_i. Then the *prefix* operation (*parallel prefix, scan*) will result in processor i containing value $s_i = a_0 \oplus a_1 \oplus \ldots \oplus a_i$, where $\oplus$ is any associative or Π-quasi-valid (defined later) operator. A special type of prefix operation is the *segmented prefix* operation in which the the processors are divided into groups with consecutive IDs and a prefix operation is performed within each group in parallel.

Monotonic Route Assume that m of the processors contain records to be routed, each record contains a destination processor ID, and these destination IDs form a strictly monotonic sequence. Then *monotonic route* will route these records to their destination processors. Some special monotonic routes are *compress*, in which the m records are routed to the first m processors, *distribute*, in which the m records are originally in the first m processors, and *increment*, in which processor i receives the record from processor $i-1$, for all $1 \leq i \leq n-1$.

Merge Assuming all lists are stored one item per processor in consecutively numbered processors, a *merge* operation takes two sorted lists of combined size n and merges them into a single sorted list.

Broadcast A *broadcast* operation takes a value at a single processor and broadcasts it to all other processors. A special type of broadcast is the *segmented broadcast*, in which processors are split groups of consecutively numbered processor, and one processor in each group broadcasts its value to the rest of its group.

3 General Approach

The inherently $\Theta(\log^2 n)$ algorithms mentioned above all use the same basic technique. They split the data into k groups, recursively solve the problem in each group, and then merge solutions in $\Theta(\log n)$ time. Thus the running time of this method is found by solving the recurrence:

$$\begin{aligned} T(n) &= T(n/k) + \Theta(\log n) \\ &= \Theta(\log^2 n) \end{aligned}$$

Since the time to merge two solutions together cannot be decreased (the time is dominated by the hypercube diameter, which is $\log n$), to decrease the time we will use the idea of splitting the input data into $\sqrt{n}$ groups, instead of k groups. This divide-and-conquer variation is common for PRAM algorithms, but has only rarely been applied to hypercube algorithms (one relevant exception is the convex hull algorithm in [9]). Sometimes the subdivision must be done twice, and sometimes once will suffice. After this, solutions will be merged together in $\Theta(\mathrm{Sort}(n))$ time. This method yields running times which can be found through one of the following recurrences:

$$\begin{aligned} T(n) &= T(\sqrt{n}) + \Theta(\mathrm{Sort}(n)) \qquad (1) \\ &= \Theta(\mathrm{Sort}(n)) \end{aligned}$$

or

$$\begin{aligned} T(n) &= 2T(\sqrt{n}) + \Theta(\mathrm{Sort}(n)) \\ &= \Theta(\mathrm{Sort}(n) \cdot \log\log n) \end{aligned} \tag{2}$$

The equations above are valid if $\mathrm{Sort}(n) = f(n)\log n$, where $f(n)$ is a positive, nondecreasing function of n. We assume that any hypercube sorting algorithm will satisfy this property. Technically equation 2 only yields $T(n) = O(\mathrm{Sort}(n) \cdot \log\log n)$ under this assumption, but for all of the sorting algorithms known to us the O can be replaced by Θ.

4 Dominance Operations

From [15], a Π-*quasi-valid response* to a search query is a set which is a superset of the answer to the search query and satisfies a constraint Π, and a binary operator is Π-*quasi-valid* if it takes two Π-quasi-valid responses to search queries and returns a Π-quasi-valid response to the union of the search queries. (Section 4.3 contains an illustrative example.) Suppose we are given a set S of n planar points with associated values where each point $p \in S$ is represented by a point record (p_x, p_y, p_v). Suppose that we are also given a constant-time operation $\oplus$, which is either associative or Π-quasi-valid. Then the goal of the *dominance operation* is to calculate for each $p \in S$ the result of the $\oplus$ operation applied to the values of all the points which are dominated by p. If we call this result $F(p)$ and define $q \prec p$ to mean that q is dominated by p, then

$$F(p) = \bigoplus_{q:q \prec p} q_v.$$

If this operation is to be performed over a subset T of S, it will be denoted by $F_T(p)$. It will be shown that 2-Set Dominance Counting, 3-Dimensional Maxima, Closest Pair, and All Points Nearest Neighbor can all be solved using dominance operations.

4.1 Dominance Operation Procedure

To compute a dominance operation, we are given a set S of n planar points, placed one per processor on an n processor hypercube and for each point $p \in S$, we wish to compute $F(p)$. We will use divide-and-conquer into sets of size $\sqrt{n}$.

First sort the points by x coordinates and take every $\sqrt{n}$'th point. These x *splitters* divide the points into $\sqrt{n}$ columns with $\sqrt{n}$ points in each. Then sort the points by y coordinates and take every $\sqrt{n}$'th point to find the y *splitters* which divide the points into $\sqrt{n}$ rows with $\sqrt{n}$ points in each.

Denote the x splitter of rank i by $\hat{x}_i$ and the y splitter of rank i by $\hat{y}_i$. Define *column* i as the area between $\hat{x}_i$ and $\hat{x}_{i+1}$, and C_i as the subset of points in S which lie in column i. Similarly, define *row* i as the area between $\hat{y}_i$ and $\hat{y}_{i+1}$, and R_i as the subset of points in S which lie in row i. Define *cell* i, j as the intersection of column i and row j, and $Q_{i,j}$ as $C_i \cap R_j$.

We can see from Figure 1 that if $p \in Q_{i,j}$, then $F(p)$ can be found by taking the $\oplus$-sum of three values, namely, $F((\hat{x}_i, \hat{y}_j))$, $F_{R_j}(p)$, and $F_{C_i}((p_x, \hat{y}_j))$. The procedures to find these will be called computing dominance values at intersections, computing dominance values in rows, and computing dominance values in columns, respectively.

Computing Dominance Values at Intersections Sort the points in S by y coordinates so that they will be grouped by rows. Then broadcast the x splitters to each row. Once this is done, we can work in

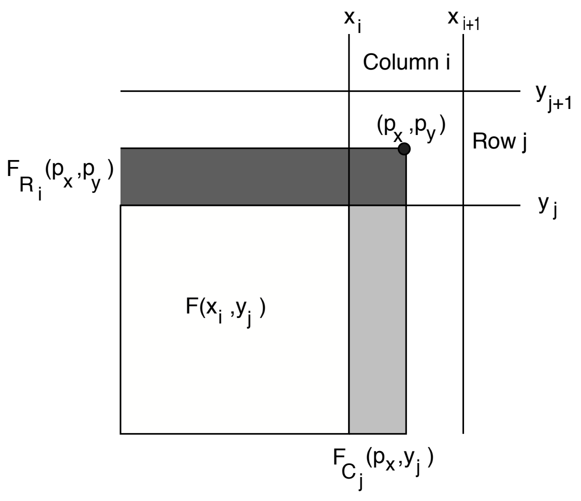

Figure 1: Computing the Dominance Value of a Point in $Q_{i,j}$

parallel on each row separately. Assume we are in row j. First sort the points in R_j by x coordinate, and then merge in the x splitters, giving each x splitter a y coordinate of $\hat{y}_{j+1}$ and a value which is the zero of the $\oplus$ operation. Then perform a prefix using the $\oplus$ operation. The value obtained at x splitter i will be $F_{R_j}(\hat{x}_i, \hat{y}_{j+1})$. To find $F(\hat{x}_i, \hat{y}_j)$, first compress the x splitters in each row, along with the values they obtained from the prefix, and then sort all of them together by two keys, the first being their rank, and the second being the row to which they were broadcast. Now x splitters will be grouped by rank and ordered by row, so performing a prefix operation using the $\oplus$ operation within each rank will correctly compute $F(\hat{x}_i, \hat{y}_{j+1})$ for $\hat{x}_i$ corresponding to row j. Then perform an increment route to give $\hat{x}_i$ in row j the value $F(\hat{x}_i, \hat{y}_j)$.

To handle the case where we are performing a prefix operation on points sorted by x (y) coordinates, and many points have the same x (y) coordinate, we modify prefix as follows. First perform a normal prefix operation. Then each processor i which contains a point with a different x (y) coordinate than processor $i-1$ broadcasts its value to the group (possibly empty) of processors immediately following it which have the same x (y) coordinate. This segmented broadcast will give each processor the correct dominance value.

To inform each point in $Q_{i,j}$ of the value of $F(\hat{x}_i, \hat{y}_j)$, sort the x splitters by two keys, the first being the rows to which they were originally sent, and the second being their ranks. Then merge them (along with the dominance values) into their corresponding rows, and broadcast the value at each x splitter to those points between itself and the next x splitter.

Computing Dominance Values in Rows Sort the points by row and recursively call the dominance procedure within each row.

Computing Dominance Values in Columns To avoid double counting the points in a cell, change p_y to $\hat{y}_j$. Now sort the points by column and recursively call the dominance procedure for each column.

We see that the dominance operation computes a function over the points below and to the left of each point. Equivalently, this dominance operation could be performed over the points above and to the left, above and to the right, and below and to the right.

4.2 Time Analysis

At each stage of recursion, only sorting, merging and prefix operations are performed. These are dominated by the time to sort. The recursive procedure is called twice on the square root of the number of points we are dealing with, leading to recurrence 2 above. Thus we have the following theorem.

Theorem 4.1 *Given n points, a dominance operation can be computed on an n processor hypercube in $\Theta(\mathrm{Sort}(n) \cdot \log\log n)$ time. Using ShareSort gives $\Theta(\log n(\log\log n)^3)$ time.*

4.3 Applications

The 2-Set Dominance Counting problem can be easily solved using the dominance operation. To solve the 3-Dimensional Maxima problem on a set S of points in 3 dimensions, run the upper right dominance operation (where $q \prec p$ if the x and y coordinates of q both exceed the corresponding coordinates of p). For each point $p \in S$ designate p_v as the z coordinate of p, and let $\oplus = \max$. Then for each p, p will be a maximal point if and only if $F(p) < p_v$.

The All Nearest Neighbors problem can be solved as follows. Let $p_v = \{p\}$ for each point $p \in S$ and let $\oplus$ be the following Π-quasi-valid operation defined by Willard and Wee [15]. Let R designate a region, x_0 be the smallest x coordinate of a point in $S \cap R$, and y_0 be the smallest y coordinate of a point in $S \cap R$. Let $r = (x_0, y_0)$. Let $V(r)$ be the points in $S \cap R$ which are as close to r as to any other points in $S \cap R$. Define a query Q_R which returns $V(r)$. The constraints Π which a quasi-valid response A to Q_R must meet will consist of $A \subset S \cap R$ and $|A| \leq 3$. The definition of the Π-quasi-valid operator follows from the definition of the Π-quasi-valid response to the query Q_R.

Run the dominance operation procedure four times (once for each quadrant), finding the most twelve candidates for each point $p \in S$ which could have p as a nearest neighbor. By sorting these candidates and performing a prefix operation, we can find the nearest neighbor for each point. A simple prefix operation will then solve the Closest Pair problem. These facts lead to the following theorem.

Theorem 4.2 *Given n points, the following problems can be reduced to dominance operations, and can thus be solved on an n processor hypercube in $\Theta(\mathrm{Sort}(n) \cdot \log\log n)$ time: 2-Set Dominance Counting, 3-Dimensional Maxima, Closest Pair, and All Nearest Neighbors. Using ShareSort, these problems can be solved in $\Theta(\log n(\log\log n)^3)$ time.*

5 Triangulation

Wang and Tsin [13] give an optimal PRAM algorithm for Triangulation, based on using $\sqrt{n} - 1$ dividing lines to divide the point set S into $\sqrt{n}$ equal sized subsets, solving Triangulation recursively on each subset, and then performing the necessary triangulation outside the convex hulls of each subset. An important part of this last step is to find the common supporting lines between every pair of convex hulls. Their algorithm for this part does not translate directly into an efficient hypercube algorithm, but by using the technique of Miller and Stout [9] one can obtain an efficient algorithm. They show that by dividing S into subsets of size $n^{3/4}$, and recursively finding the convex hull of each, one can find the common supporting lines between every pair of these convex hulls in $\Theta(\log n)$ time. Combining these, the total time required for triangulation is

$$\begin{aligned} T(n) &= T(n^{3/4}) + \Theta(\mathrm{Sort}(n)) \\ &= \Theta(\mathrm{Sort}(n)). \end{aligned}$$

6 Visibility

The third type of problem we consider is visibility from a point. Due to severe space restrictions, we can only make a few comments concerning our visibility algorithms.

When there are no intersections, the basic divide-and-conquer into $\sqrt{n}$ pieces is rather straightforward, and one need only consider how line segments originating in one region may overlap another region. When there are intersections it is harder to keep track of the segments, and there is the additional complication that the answer may have $\Theta(n\alpha(n))$ segments, where $\alpha(n)$ is the inverse of the Ackermann function [14]. To merge pieces together, we use the merge routine described in ShareSort. This merge routine calls a subroutine called SharedKeySort, which sorts together groups of items where each item in a group has the same key. The analysis for our routine will thus depend on $\Theta(\mathrm{SharedKeySort}(m))$ rather than $\Theta(\mathrm{Sort}(m))$. In [6] it is shown that merging can be completed in $\Theta(\mathrm{SharedKeySort}(m) \cdot \log\log m)$ time, and that SharedKeySort(m) can be completed in $\Theta(\log m(\log\log m))$ time.

Theorem 6.1 *Given n line segments, and a point p, the visibility operation on an n processor hypercube can be performed in $\Theta(\mathrm{Sort}(n))$ time if the segments do not intersect, and in $\Theta(\mathrm{SharedKeySort}(n) \cdot \log\log n + \log n(\log\log n)\alpha(n))$ time if there are intersections. (If segments intersect, then we assume each processor has a memory size of $\Omega(\alpha(n))$.) Using ShareSort, the visibility operation can be performed in $\Theta(\log n(\log\log n)^2)$ time, whether or not the segments intersect.*

7 Conclusion

We have presented new hypercube algorithms for 2-Set Dominance Counting, 3-Dimensional Maxima, Closest Pair, All Nearest Neighbors, Triangulation, and Visibility from a Point. All of these algorithms have better asymptotic running times than their hypercube predecessors. The impetus for these faster algorithms was the development of a sorting algorithm which sorts in $o(\log^2 n)$ time. Previously developed hypercube computational geometry algorithms could not use new sorting algorithm to decrease their asymptotic running time below $\Theta(\log^2 n)$.

We note that sorting on a hypercube is still an open problem, with the only known lower time bound being $\Omega(\log n)$. Any asymptotic improvement in sorting time will be reflected by the same asymptotic improvement in the running times of our algorithms. An open question is if the $\log\log n$ factor can be removed from the $\Theta(\mathrm{Sort}(n) \cdot \log\log n)$ time algorithms which we presented.

References

[1] M.J. Atallah, R. Cole, and M.T. Goodrich, "Cascading divide-and-conquer: A technique for designing parallel algorithms", *SIAM J. Computing*, 18(3):499-532, 1989.

[2] M.J. Atallah and M.T. Goodrich, "Efficient parallel solutions to some geometric problems", *J. of Parallel and Distributed Computing*, 3:492–507, 1986.

[3] P. Bertolazzi, S. Salza, and C. Guerra, "A parallel algorithm for the visibility problem from a point", *J. Parallel and Distributed Comp.*, 9(1):11–14, 1990.

[4] L. Boxer and R. Miller, "Dynamic computational geometry on meshes and hypercubes", *Proc. 1988 Intl. Conf. on Parallel Proc.*, 323–330.

[5] R. Cole and M.T. Goodrich, “Optimal parallel algorithms for polygon and point-set problems”, *Proc. 4th ACM Symp. on Comp. Geom.*, 205–214, 1988.

[6] R. Cypher and C.G. Plaxton, “Deterministic sorting in nearly logarithmic time on the hypercube and related computers”, *Proc. 22nd ACM Symp. Theory Comp.*, 1990.

[7] C.S. Heong and D.T. Lee, “Parallel geometric algorithms on a mesh connected computer”, Technical Report 87-02-FC-01, Northwestern University, 1987.

[8] P.D. MacKenzie and Q.F. Stout, “Practical hypercube algorithms for computational geometry”, *Proc. 3rd Frontiers of Massively Parallel Proc.*, 1990.

[9] R. Miller and Q.F. Stout, “Efficient parallel convex hull algorithms”, *IEEE Trans. Comp.*, 37(12):1605–1618, 1988.

[10] R. Miller and Q.F. Stout, “Mesh computer algorithms for computational geometry”, *IEEE Trans. Comp.*, 38(3):321–340, 1989.

[11] R. Miller and Q.F. Stout, *Parallel Algorithms for Regular Architectures*, MIT Press, 1990.

[12] I. Stojmenovic, “Computational geometry on a hypercube”, *Proc. 1986 Intl. Conf. on Parallel Proc.*, 100–103.

[13] C.A. Wang and Y.H. Tsin, “An O(log n) time parallel algorithm for triangulating a set of points in the plane”, *Info. Process. Lett.*, 25(1):55–60, 1987.

[14] A. Wiernik and M. Sharir, “Planar realizations of nonlinear Davenport-Schinzel sequences by segments”, *Discrete Comput. Geometry*, 3:15–47, 1988.

[15] D.E.Willard and Y.C. Wee, “Quasi-valid range querying and its implications for nearest neighbor problems”, *Proc. 4th ACM Symp. on Comp. Geom.*, 34–43, 1988.