Algorithmica (1992) 7: 51–75

Algorithmica

Efficient Convexity and Domination Algorithms for Fine- and Medium-Grain Hypercube Computers[1]

Ed Cohen,[2] Russ Miller,[2] Elias M. Sarraf,[3] and Quentin F. Stout[4]

Abstract. This paper gives hypercube algorithms for some simple problems involving geometric properties of sets of points. The properties considered emphasize aspects of convexity and domination. Efficient algorithms are given for both fine- and medium-grain hypercube computers, including a discussion of implementation, running times and results on an Intel iPSC hypercube, as well as theoretical results. For both serial and parallel computers, sorting plays an important role in geometric algorithms for determining simple properties, often being the dominant component of the running time. Since the time required to sort data on a hypercube computer is still not fully understood, the running times of some of our algorithms for unsorted data are not completely determined. For both the fine- and medium-grain models, we show that faster expected-case running time algorithms are possible for point sets generated randomly. Our algorithms are developed for sets of planar points, with several of them extending to sets of points in spaces of higher dimension.

Key Words. Hypercube, Parallel algorithms, Convex hull, Domination, Computational geometry.

1. Introduction. Computational geometry is a rapidly growing field, with applications to robotics, VLSI design, plant layout, and many other areas. Numerous serial algorithms have been developed for a large number of geometric problems, forming a somewhat organized body of results [PS]. However, parallel algorithms for problems in computational geometry have lagged behind serial algorithms, with most of the parallel algorithms appearing in the last few years [AG], [ACG*]. In this paper we consider hypercube algorithms for a few simple problems involving sets of points in two-dimensional Euclidean space, and provide efficient algorithms for problems involving convexity and domination (defined below). We also indicate some extensions to data in higher dimensions. While the algorithms are designed explicitly for hypercube computers, they borrow ideas used in geometric algorithms for other parallel computers, and they in turn can be utilized on other parallel computers.

[1] The research of E. Cohen, R. Miller, and E. M. Sarraf was partially supported by National Science Foundation Grant ASC-8705104. R. Miller was also partially supported by National Science Foundation Grants DCR-8608640 and IRI-8800514. Q. F. Stout's research was partially supported by National Science Foundation Grant DCR-85-07851, and an Incentives for Excellence Grant from the Digital Equipment Corporation.

[2] Department of Computer Science, State University of New York at Buffalo, Buffalo, NY 14260, USA.

[3] Amherst Systems Inc., 30 Wilson Road, Buffalo, NY 14221, USA.

[4] Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109, USA.

Received April 21, 1988; revised May 9, 1989. Communicated by Alok Aggarwal.

Algorithms are given for both fine- and medium-grain hypercubes. In a *fine-grain* machine, such as the 65,536 processor Connection Machine manufactured by Thinking Machines, Inc., each processor is quite simple, typically bit-serial with a small amount of memory. For the problems we consider, we assume that each processor in a fine-grain machine starts with one data point.

In a *medium-grain* machine there are many microprocessor-like processors, each with a substantial amount of memory. For our geometric problems, we assume that each processor in a medium-grain machine starts with many points, perhaps thousands, and that there may be a thousand processors. Hypercubes from NCUBE, Intel, and FPS are examples of medium-grain machines, though apparently only NCUBE has installed a thousand-processor machine so far.

Throughout this paper, an n-cube denotes an n-dimensional binary hypercube with 2^n processors, labeled with n-bit binary strings. These labels may be interpreted as binary representations of numbers from 0 to $2^n - 1$, where two processors are connected if and only if their labels differ by exactly one bit. Note: we use lg to mean $\log_2$, and ln to mean $\log_e$.

The best serial and parallel solutions for many geometric problems depend upon sorting, and in fact sorting is often the slowest part of the algorithm [PS]. For hypercubes this causes a problem since it is not clear that good sorting algorithms are known with respect to all possible input. A sorting algorithm with the current best worst-case time on fine-grain hypercubes is Bitonic Sort [Ba], which sorts 2^n items stored one per processor in an n-cube in $\Theta(n^2)$ time. We define $Sort(2^n)$ to be the time to sort 2^n items in an n-cube. Since the largest known lower bound for sorting on an n-cube is $\Omega(n)$, it is not clear that Bitonic Sort is optimal. If an expected-case sorting algorithm can be tolerated, then the algorithm of [RV] can be used to sort 2^n items stored one per processor on an n-cube, in expected $\Theta(n)$ time.

For medium-grain hypercubes, the best sorting algorithms are found in [CS2] and [P]. These algorithms sort 2^{cn} items per processor in an n-cube (for a total of $2^{n(1+c)}$ items) in $\Theta(n2^{cn}/c)$ time. For any fixed $c > 0$ this achieves linear speedup, but requires that the number of items per processor grows as a power of the number of processors. It is also not known whether the constants involved are low enough to make these algorithms of practical interest. Practical algorithms with good observed performance are known [W], achieving linear speedup with far fewer items per processor, but it is not clear how low the item/processor ratio can go and still attain linear speedup. Further, these practical algorithms can exhibit poor worst-case behavior.

Because of the dependence on sorting, and the uncertain status of sorting times, several of our algorithms are first developed for presorted data. Notice that given a presorted algorithm with running time $T(2^n)$, an arbitrary input version of the algorithm will solve the same problem in $T(2^n) + Sort(2^n)$ time. We also analyze algorithms for data sets consisting of random points, chosen independently and uniformly from a variety of unit-area regions, such as a square or circle. In this situation many of the points can be quickly eliminated from further consideration without any sorting, which has the effect of rapidly reducing the problem to one involving far fewer points. Although our algorithms are optimized for

randomly distributed data, each algorithm works correctly no matter what the distribution.

In Section 2 we define the various geometric properties for which we develop algorithms. In Section 3 some global parallel operations are defined and their implementation times on the hypercube are given. These form the building blocks of our algorithms. In Section 4 we give algorithms for fine-grain machines, using worst-case analyses. Section 5 contains efficient algorithms for sets of random points, using expected-case analyses. In that section we also evaluate alternative algorithms by analyzing the tradeoffs that occur. One important point in such an analysis is that we need to deal with the expected maximum completion time, rather than the expected completion time, when a task is broken into parallel subparts. Finally, Section 6 contains additional comments.

2. Geometric Properties. Throughout this paper, *point* without further modifiers means a point of Euclidean 2-space, given via Cartesian coordinates. To avoid various definitional and algorithmic complications of little interest, we assume that no set contains duplicate points. Given two points (x_1, y_1) and (x_2, y_2), we say that (x_1, y_1) *dominates* (x_2, y_2) if $x_1 \geq x_2$ and $y_1 \geq y_2$. Given a set S of points, a point p in S is *maximal* if it is not dominated by any other point. This definition extends to higher dimensions in the obvious manner. By *determining the maximal points of* S we mean that there is a Boolean flag associated with each point in S, and at the end of the algorithm a point's flag is true if the point is a maximal point of S, otherwise the flag is false. Figure 1 illustrates maximal points. We use *maximal* (S) to denote the set of maximal points of S. Notice that no two maximal points can have the same x-coordinate, and no two can have the same y-coordinate. Also notice that if the maximal points are ordered by decreasing x-coordinates, then their y-coordinates are increasing.

The convex hull of a set S of points, denoted *hull*(S), is defined to be the minimum convex set containing S. A point $p \in S$ is an extreme point of S if $p \notin hull(S - \{p\})$. For finite sets, p is an extreme point if and only if it is a vertex of the smallest convex polygon containing all points of S. *Determining the extreme points of* S is

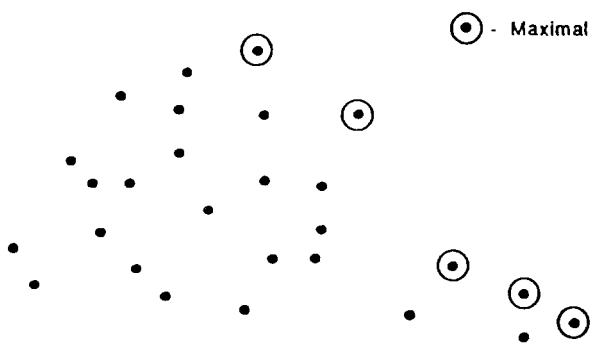

Fig. 1. Maximal points.

defined analogously to determining the maximal points of S. We use *extreme*(S) to denote the set of extreme points of S.

Two properties that are used repeatedly are

$$\text{maximal}(S \cup T) \subseteq \text{maximal}(S) \cup \text{maximal}(T)$$

and

$$\text{extreme}(S \cup T) \subseteq \text{extreme}(S) \cup \text{extreme}(T),$$

for any sets S and T.

Besides determining the extreme points of a set, we often need to determine the edges of the convex hull. To do so, it is convenient to *number the extreme points* in counterclockwise order, starting with the topmost point. (If there are two topmost points, then choose the rightmost one of these as the starting point.) If a point has number i, then it is an endpoint of edges to points numbered $i - 1$ and $i + 1$, where these indices are interpreted modulo the number of extreme points.

Notice that any set can have at most two topmost extreme points, and similarly has at most two bottommost, two leftmost, and two rightmost extreme points. Also notice that in a counterclockwise traversal of extreme points from the rightmost extreme point (topmost in the case of a tie) to the topmost one (rightmost in the case of a tie), the extreme points occur in decreasing x-coordinate order, and all are maximal. However, not all maximal points are extreme points.

One technique for reducing the amount of data to be combined in finding extreme points of a set S is first to find all the maximal points of S (call this set M_1). Next, find all the maximal points M_2 using the revised definition that (x_1, y_1) dominates (x_2, y_2) if $x_1 \geq x_2$ and $y_1 \leq y_2$, then, all maximal points M_3 where domination is revised to $x_1 \leq x_2$ and $y_1 \leq y_2$, and, finally, all maximal points M_4 where domination is revised to $x_1 \leq x_2$ and $y_1 \geq y_2$. Since

$$\text{extreme}\,(M_1 \cup M_2 \cup M_3 \cup M_4) = \text{extreme}(S),$$

if maximal points can be found rapidly and if few points are maximal under one of these four definitions, then the problem of determining extreme points of the original set has been rapidly reduced to finding extreme points of a much smaller set.

Several properties of a point set can be quickly determined from its extreme points. For example, the *diameter* of a set is the maximal (Euclidean) distance between any pair of points, and it is easy to see that the diameter of a set is the diameter of the extreme points of the set. A *smallest enclosing rectangle* of a set S is a rectangle of minimal area containing S. Such a rectangle must have one side along an edge of the convex hull of S, with all four sides touching the convex hull [FS].

Many of the geometric techniques and facts used in this paper are also used for serial algorithms. A fairly comprehensive overview of serial algorithms for compu-

tational geometry appears in [PS], containing extensive references to most of the simple facts mentioned here without attribution.

3. Global Operations. Global operations form the foundation of our algorithms. These include *broadcast*, in which one processor sends a value to all others, and *report*, in which all processors start with a value and there is a commutative semigroup operation such as maximum or sum which is applied to the values, resulting in a single value arriving at a designated processor. (Throughout this paper we assume that all relevant semigroup operations can be computed in constant time). Well-known hypercube algorithms can be used to perform these operations in $\Theta(n)$ time on an n-cube.

Another common operation is parallel *prefix*, in which every processor i starts with a value v_i and ends up with $v_0 * \cdots * v_{i-1}$, where $*$ is some associative operation. The postfix operation is defined analogously. Both prefix and postfix can be computed in $\Theta(n)$ time on an n-cube in a straightforward manner. On a medium-grain n-cube, if each processor starts with m values, then report, prefix, and postfix can be accomplished in $\Theta(n + m)$ time.

Routing on a parallel computer, namely delivering messages from source nodes to their destinations, can be accomplished via a fixed number of sorting steps [MS3], so arbitrary routing on a fine-grain n-cube can be accomplished in $\Theta(n^2)$ time if each processor initiates a fixed number of messages and receives a fixed number of messages. If there is a constant $c < 1$ such that only 2^{cn} items are being routed in an n-cube, then the routing can be accomplished in $\Theta(n)$ time, where the implied constant depends upon c, by using the sorting algorithm in [NS]. While the best worst-case time to sort on a fine-grain n-cube is still unknown, it was shown in [Ba] that *merging* two sorted sets stored one item per processor can be completed in $\Theta(n)$ worst-case time.

For all of the $\Theta(n)$-time operations it is easy to see that the times are optimal in the worst case since information starting in one processor of an n-cube might need to travel at least n communication links to reach its bit-complemented destination processor.

An interesting use of parallel prefix and merging can be used to solve a variety of search problems. A simple example follows. Suppose $R = \{r_1, \ldots, r_N\}$ is an ordered set of real numbers such that

$$-\infty = r_0 < r_1 < \cdots < r_N < r_{N+1} = +\infty,$$

and S is a set of M or fewer real numbers. Suppose for each $s \in S$ we want to find the elements $r_i, r_{i+1} \in R$ such that $r_i < s \leq r_{i+1}$. If R and S are sorted and stored one per processor in an $\lg(N + M)$-cube, then, by merging them, each $s \in S$ will fall between the appropriate elements of R. Using a prefix operation to find the largest element of R occurring earlier in the sorted order, and a postfix operation to find the least element of R occurring later, the problem is solved in $\Theta(\log(N + M))$ time. Variations of this operation have been called a *grouping operation* or a *merge search* [MS2], [St]. Here it is referred to as *ordered search.*

4. Fine-Grain Algorithms. Throughout this section we assume that points are initially stored one per processor in a fine-grain hypercube. Let Sort(N) denote the best worst-case time to sort N items stored one per processor in an lg(N)-cube. Because several algorithms involve sorting, and Sort(N) is only known to be bounded between $\Omega(\log N)$ and $O(\log^2 N)$, we state the times of some algorithms in terms of Sort(N). In all cases, the algorithms, statements of the results, and analyses remain correct if time is interpreted as expected-case time and Sort(N) is the best expected-case time to sort.

4.1. Domination. In [AG] it was shown that a simple "sweep" algorithm can be used to find maximal points efficiently on almost any parallel computer. First sort the points by decreasing x-coordinate (breaking ties by decreasing y-coordinate). Then use a prefix computation to determine, for each point, the largest y-coordinate of any preceding point. Note that a point p is maximal if and only if p's y-coordinate is greater than this value. This is a direct parallelization of a natural serial solution to the problem, having the nice property that it yields optimal algorithms on a variety of parallel computers of interest. For the hypercube it gives the following.

PROPOSITION 1. *Given a set of no more than N points stored one point per processor in an* lg(N)*-dimensional hypercube, if the points have been presorted by x-coordinate, then the maximal points can be determined in $\Theta(\log N)$ time. If the points have not been presorted, then the maximal points can be determined in $\Theta(\text{Sort}(N))$ time.*

The number of points dominated by each maximal point can be determined by counting the number of points a maximal point does not dominate. If a maximal point (x, y) does not dominate (x', y'), then it must be that either $x < x'$ or $y < y'$, but not both since (x', y') does not dominate (x, y). The number of dominated points with greater x-coordinate is the position of (x, y) when the set is sorted in decreasing x-coordinate order, minus the number of preceding points with the same x-coordinate. The number of preceding points with greater y-coordinate can be similarly determined. The number of points with the same x-coordinate can be determined by a prefix operation. Note that having presorted data will not help the asymptotic running time since this algorithm uses two sorts.

THEOREM 2. *Given a set of no more than N points stored one point per processor in an* lg(N)*-dimensional hypercube, for every maximal point, the number of points it dominates can be determined in $\Theta(\text{Sort}(N))$ time.*

Next we consider the problem of determining for every point (not just maximal points) the number of points it dominates. We exploit a multidimensional divide-and-conquer approach [Be] to solve this problem. The sort- or merge-dominated algorithm is based on the following observation. Suppose point sets S and T are such that, for every point $(x, y) \in S$ and $(x', y') \in T$, either $x < x'$ or $x = x'$ and $y < y'$. Then no point in S dominates any point in T, and a point $p \in T$ dominates each point in S with y-coordinate no greater than p's. The number of points that

$p \in T$ dominates in S can be determined by sorting the two sets by y-coordinate, breaking ties by placing smaller x-coordinates first, and then taking p's position minus the number of points in T that precede p. (If S and T were initially sorted, then this reduces to a simple use of merging.) A generalization of the result follows.

THEOREM 3. *Fix $d \geq 2$. Given a set S of no more than N d-dimensional points stored one per processor in an $\lg(N)$-dimensional hypercube, determining the number of points every point dominates, or determining the number of points dominating every point, can be accomplished in $\Theta(\log^d N)$* time.

4.2. Convexity. While extreme points are similar to maximal points, the differences can also be quite significant. For example, if the points are sorted by x-coordinate, then no simple information from points lying on one side of a point will provide enough information to decide whether or not the point is an extreme point. Using a divide-and-conquer approach first applied to PRAM algorithms, Cypher and Sanz [CS1] developed a $\Theta(\log^2 N)$-time algorithm, and Miller and Stout [MS1] developed a $\Theta(\text{Sort}(N))$-time algorithm, for determining the extreme points of a set of N points on an $\lg(N)$-dimensional hypercube. The [MS1] algorithm has the additional property that it needs only $\Theta(\log N)$ time if the data is presorted. The divide-and-conquer algorithm given in [MS1] finds *lines of support* between linearly separable convex hulls based on the assumption that the points are presorted so that there are $N^{1/4}$ groups of $N^{3/4}$ points, where the ith group is stored in the ith subhypercube. This yields a running time recurrence of the form $T(N) = T(N^{3/4}) + \Theta(\log N)$, which is $\Theta(\log N)$.

PROPOSITION 4. *Given a set S of no more than N points stored one per processor in sorted order in an $\lg(N)$-dimensional hypercube, in $\Theta(\log N)$ time the extreme points can be determined where the counterclockwise numbering and the coordinates of the preceding and following extreme points are known for every extreme point.*

Given the extreme points and order information, several other properties of the convex hull can be easily determined by triangulating the convex hull so that the first point in the counterclockwise ordering is a vertex of every triangle. Using this, we obtain

COROLLARY 5. *Given a set of no more than N points stored one per processor in sorted order in an $\lg(N)$-dimensional hypercube, in $\Theta(\log N)$ time the area, centroid, and perimeter of the convex hull of the points can be determined.*

To determine a smallest enclosing box for a set S, first determine for each hull edge e_i a smallest enclosing box B_i containing an edge collinear with e_i. Then a smallest enclosing box of the set is a minimum area box of any of these smallest boxes. For each edge, we need to find the "opposite" point with a support line parallel to the edge, and also the points with perpendicular support lines. These are just search problems where each extreme point has an interval of support angles, and the edges form searching elements looking for the points defining the

interval containing their support angle. (*Support angles* correspond to supporting half-planes and are in the range $[0, 2\pi)$, where $0 \neq \pi$.) On the hypercube this can all be accomplished in logarithmic time using the ordered search operation discussed in Section 3. This approach has been previously used on mesh computers [MS2], and seems to be an efficient approach on almost any parallel computer. The diameter can be found similarly since the diameter is the largest distance between pairs of points with angles of support differing by π.

COROLLARY 6. *Given a set of no more than N points stored one per processor in sorted order in an* $\lg(N)$*-dimensional hypercube, in* $\Theta(\log N)$ *time a smallest enclosing box and the diameter can be determined.*

5. Algorithms for Random Point Data Sets. For very large data sets, sorting may involve a considerable amount of time. However, for sets of random points, it may be possible to avoid sorting and solve geometric problems more efficiently. This section concentrates on fine- and medium-grain hypercube algorithms to solve the domination and convex-hull problems for random point data input. Following some definitions and the discussion of the relationships between random data and efficient expected-case running times, Section 5.1 describes domination algorithms on medium-grain machines, with experimental results being given in Section 5.2. Sections 5.3 and 5.4 cover algorithm descriptions and experimental results for the convex-hull problem, respectively. Section 5.5 discusses algorithms for random points on fine-grain machines. Finally, algorithms for higher-dimensional data are described in Section 5.6.

Random points can be defined in a wide variety of ways. For simplicity we assume that the points have been independently chosen from a unit square or unit circle using a uniform distribution, though many of our techniques work with much more general distributions. The phrase *set of random points* denotes a data set generated in this manner. The fact that convexity and domination algorithms can have faster expected times on sets of random points is well known for serial algorithms. In [PS] it is shown that for the random data sets we consider in this section (described in detail later), the expected running time to determine maximal points for N d-dimensional points is $\Theta(N)$. Furthermore, it is shown in [PS] that for two-dimensional data of the form considered in this section, the extreme points can be determined with an expected running time of $\Theta(N)$. The use of randomization for faster parallel algorithms is also well known, but so far there has been little work on exploiting random data sets on parallel algorithms for geometric problems. (See, however, [Sto].)

Given a set of N random points chosen from the unit square using a uniform distribution, it can be shown that the expected number of maximal points is $H_N = \sum_{i=1}^{i=N}(1/i) = \ln(N) + \gamma + \Theta(1/N)$, where γ is Euler's constant [K]. Further, H_N has a standard deviation of $\Theta(\log^{1/2} N)$. It is also known that for the same distribution the expected number of extreme points is $(\frac{8}{3})[\ln(N) + \gamma] + O(1)$ [RS].

Since the expected number of maximal or extreme points for the unit square distribution is relatively small, we can hope to eliminate rapidly most of the points

from consideration. Given a set of N random points from a unit square, a point (x, y) can be expected to dominate $(N-1)xy$ points. Therefore, a point P which maximizes xy is expected to dominate the most points. Notice that P is necessarily maximal.

Once P is detected, every point dominated by P can be eliminated from further consideration as a maximal point. It can be shown that this single elimination step will remove all but $\Theta(N^{1/2})$ points, on average, from consideration as maximal points. The remaining points form two sets, namely those with the x-coordinate larger than P and those with the y-coordinate larger than P. In each set we can recursively apply the procedure of choosing the point which is expected to dominate the most points in the respective set. Since each application finds one new maximal point, the number of applications is the number of maximal points. In this section we consider results for random points chosen from the unit square and random points chosen from regions that do not allow for such a simplistic reduction so quickly.

5.1. Random Point Domination on Medium-Grain Machines. There are two natural ways to implement a domination algorithm for random data chosen from a unit square on a medium-grain machine:

(1) The *global* version is a recursive routine where at each step of the recursion, a globally expected best point is found and used to eliminate as many points as possible. (As described above, the first step of the algorithm would choose P, a point with maximum xy value.)
(2) The *local* algorithm first solves the domination problem independently on each processor. After finding the maximal points restricted to each processor, these points are combined to find the maximal points of the entire set. Since we expect relatively few locally maximal points, there are several reasonable techniques for combining local solutions to obtain the global solution. One natural way is to use a report procedure, where at each step every processor merges its candiate maximal points with the candidates received and eliminates those in either group that are dominated by a point from the other group.

Various blends of these two methods are also possible. To determine which of these approaches to use, we analyze the first stage of each. Given p processors, one stage of the global algorithm will take $c_1(N/p)$ time for each processor to find its best candidate, $c_2 \lg(p)$ time for the processors to determine the globally best candidate, and $c_3(N/p)$ time for each processor to eliminate dominated points, for some constants c_1, c_2, and c_3. At the end of the stage there are on average $c_4 N^{1/2}$ points remaining for c_4 a constant.

One stage of the local algorithm will take $c_1(N/p) + c_3(N/p)$ time, and will leave a total of $c_4 p(N/p)^{1/2} = c_4 p^{1/2} N^{1/2}$ points. Therefore, by spending $c_2 \lg(p)$ time on communication, the global algorithm reduces the average number of points remaining from $c_4 p^{1/2} N^{1/2}$ to $c_4 N^{1/2}$, which will reduce the time of the next stage.

Once these constants have been measured for a particular implementation, for any given values of N and p we can then predict which of these two approaches would be best for the first stage. As the stages are repeated, the communication

time required by the global algorithm increases, since at stage i there are $O(2^i)$ points being detected. Therefore, after an implementation-dependent number of global reduction stages, local reduction is used, followed by a combination step to determine the global solution.

Note that the maximum number of points remaining in any processor needs to be taken into consideration since the time of a stage depends on the slowest processor to complete that stage. This maximum grows with p as well as N/p, though the increase with an increase in p is quite slow.

Another factor that needs to be examined is how many stages of reduction should be performed for the local algorithm, since the number of "best" reduction candidates doubles at each stage. Experiments have shown that, at some point, it is faster to perform the "sweep" domination technique, instead of doing additional reduction passes. The number of reduction stages (prior to performing "sweep" domination) will depend on the number of points reduced each time and the overhead costs of performing the reduction.

5.2. Experimental Results for Domination on a Medium-Grain Machine. In this section we give results of parallel domination algorithms on a 16-node Intel iPSC1 hypercube at SUNY, Buffalo. Restricting our attention for the moment to the global approach, we obtain the following asymptotic result as a point of reference.

PROPOSITION 7. *Given N random points chosen from a unit square and distributed N/p per processor in an $\lg(p)$-dimensional hypercube, the maximal points can be determined in $\Theta(N/p + \log(p)\ \log(N))$ expected time by the global approach.*

Our results show that beyond the first one or two global stages there is little additional load-balancing benefit from the global approach since the number of points remaining is extremely small. Due to the fact that global stages use extra communication, we have found it better to switch to a local approach after a few stages and then combine the results. Extensive testing was performed and results verify our analyses of time and expected number of points remaining.

Table 1. The number of points remaining in a cube after each global pass.*

Number of passes	Points remaining in unit square	Points remaining in tilted unit square
0	100,000	100,000
1	615	24,599
2	523	12,278
3	328	6,177
4	220	3,144
5	218	1,673
6	216	872
Number of maximal points	122	289

* As the number of remaining points approaches the number of maximal points, the reduction passes become less useful.

Table 2. Running times for Domination program with unit square point distribution.*

Number of processors	Points per processor	Running time in milliseconds	Speedup
1	100,000	37,625	1.00
2	50,000	18,375	2.05
4	25,000	9,075	4.15
8	12,500	4,575	8.22
16	6,250	2,455	15.36

* The program performed one global reduction and two local reductions, befor performing a local "sweep" algorithm. Extensive testing showed this to be the most efficient algorithm. The data were randomly generated each time accounting for the apparent superlinear speedups.

Table 1 shows that the first few global passes rapidly reduce the number of points to be considered. In our tests, the time for the initial $\Theta(N/P)$-time reduction dominated the minimal $\Theta(\log p \log N)$ time for communication and combination steps, giving us near linear speedup over the data tested. (See Table 2.)

Since our reduction techniques rapidly reduce the number of points to consider for random points chosen from a unit square, we consider additional random point distributions which have a greater expected percentage of maximal points. Distributions in the shape of a unit circle and of a 45° tilted unit square were examined, and the results from the tilted unit square distribution are discussed here. It can be shown that points chosen independently from a tilted unit square using a uniform distribution will have approximately order $n^{1/2}$ expected maximal points. Notice that no point can dominate more than 75% of the tilted unit square. Therefore, the percentage of points remaining after each reduction stage for tilted square distributions is expected to be higher than for unit square distribution. This is confirmed by our experiments as can be seen in Table 1.

The method we used for choosing reduction points for the tilted square distribution was different than for a nontilted square, and involves three stages. The first stage chooses the point with the maximum xy product, as before. The second stage chooses two points, one with maximum x value and one with maximum y value. For the third and subsequent stages, we chose points closest to the center of the northeast wall for a given group of points. (See Figure 2.) Using this reduction technique, the average percentage of points eliminated was 75 for the first stage and 50 for each following stage. For the tilted square it was found that the best performance was achieved by performing several global reduction passes followed by a local "sweep" algorithm and a final combination step.

We considered three ways of performing the final combination. The most efficient of those is now described. First, every node locally sorts its candidate maximal points. Then, at each step of the report process, the points are merged (maintaining their sorted order), while dominated points are eliminated by keeping track of the current maximal y-value. This is simply a modified version of the "sweep" technique. Using this technique, dominated points are eliminated at each step, and no further work is needed when the points reached the final node.

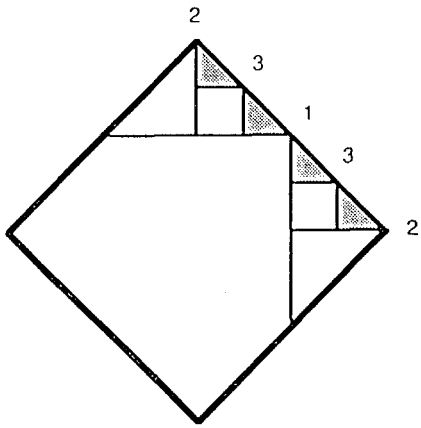

Fig. 2. Unshaded areas indicate the points dominated by the various reduction points (number 1 indicates the first reduction point, number 2 indicates the second two reduction points, etc.).

Analysis of the expected running time of the algorithm shows that we may be able to achieve near linear speedup, provided the number of iterations and number of processors used is reasonable for the problem. The global reduction takes $\Theta((N/p) + (2^{i-1} \log p))$ time, where the (N/p) term dominates while the number of iterations i is small. The local reduction takes $\Theta(N/p)$ time. The local domination takes time proportional to the serial sort time with respec to the number of points left to consider. For the random point distribution on the tilted square, the expected local domination time (i.e., the "sweep" algorithm) is

$$\Theta(N^{1/2}/p \log(N^{1/2}/p)).$$

Finally, the time for the report is $\Theta(N^{1/2})$. Therefore, the expected running time for the tilted square distribution is $\Theta(N/p + 2^{i-1} \log p + N^{1/2})$. Our tests were performed with 100,000 random points distributed evenly across the nodes, and the number of local and global reduction iterations varying from one to five. In all cases, the first term dominated and speedups approached linear. (See Table 3.)

Table 3. Running times for Domination with tilted square point distribution, with six global passes.

Number of processors	Points per processor	Running time in milliseconds	Speedup
1	100,000	64,740	1.00
2	50,000	32,825	1.97
4	25,000	16,640	3.89
8	12,500	8,610	7.52
16	6,250	5,050	12.82

Other reduction techniques were examined and are briefly described here. For the tilted square, notice that by omitting the "maximum x and maximum y" reduction stage and proceeding immediately to the "closest to the center" reduction strategy, the expected number of points reduced during the second stage of the algorithm would be approximately 63%. This compares favorably with the 50% expected reduction achieved by using the "maximum x and maximum y" reduction as a second stage. However, experiments on the iPSCl have shown that including "maximum x and maximum y" reduction as the second stage improves the total running time of the algorithm. We attribute this to the fact that the tests performed in the "maximum x and maximum y" reduction are somewhat different from the tests performed in the "closest to the center" reduction. The "maximum x and maximum y" reduction uses two simple comparisons ($y > max\_y$ and $x > max\_x$) in determining the points of interest, while the "closest to the center" reduction uses a single distance comparison $((x - x\_current)^2 + (y - y\_current)^2 < (distance\_current)^2)$ to determine the point of interest. Therefore, although using "maximum x and maximum y" reduction as the second reduction stage of the algorithm reduces fewer points on average than proceeding immediately to the "closest to the center" reduction, the "maximum x and maximum y" reduction is used as a second stage reduction since it reduces the total running time of our algorithm on the iPSCl.

We examined other combinations of these reduction strategies, as well. For example, after observing the results just described, we considered using the "maximum x and maximum y" reduction as the first stage, since it is also expected to reduce 75% of the points. However, experiments again showed that the original method described is faster (about 5%) than using the "maximum x and maximum y" reduction as the first reduction stage. We should point out, however, that since these results have been determined experimentally on an Intel iPSCl, variations of the algorithm or changes in the low-level implementation details, such as the ones we have described, might be more efficient on other hypercubes.

The analysis for unit square distribution is similar, except that the expected number of maximal points is smaller, and gives a running time of $\Theta(N/p + 2^{i-1} \log p + \log N)$. (See Table 2.)

5.3. Random Point Convex Hull on Medium-Grain Machines. The same approach used in determining the maximal points of a random set of data can be used to find the extreme points. Given random input chosen from a uniform distribution on the unit square, first locate a point closest to each of the four corners of the square, and then eliminate any points in the quadrilateral determined by these four points. After this elimination, there are four sets of points remaining outside the quadrilateral, one corresponding to each side. During the second step of the algorithm, locate the point the furthest distance from the side of each of the four sets. This creates a situation as in Figure 3, where the points inside the triangles are eliminated and the remaining points are subdivided into two regions. The algorithm extends recursively within regions containing potential extreme points. As a serial algorithm this approach is called "Quickhull" in [PS], and has apparently been independently discovered by several authors.

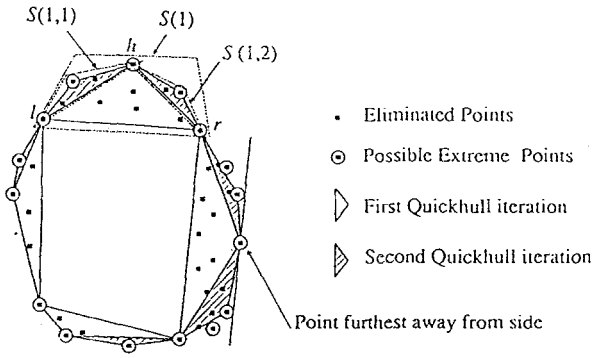

Fig. 3. Quickhull reduction. Points inside the triangles are eliminated. Points l, r, and h subdivide the set $S(1)$ into sets $S(1, 1)$ and $S(1, 2)$.

In a parallel convex-hull algorithm, a *global* Quickhull approach can be used as a data reduction technique to eliminate most points from consideration. This technique is used in [MM] to solve the convex-hull problem for digitized picture input, and follows the approach taken in the global domination algorithm of the previous section. After a number of global Quickhull iterations, each processor will attempt further data reduction by performing a local Quickhull reduction. The final solution can then be determined by combining the local results using a report operation. It should be noted that experimentation indicates that a *global* Quickhull technique should be used initially as a data reduction technique to reduce the number of points per processor, since a *local* Quickhull reduction leaves a much larger number of points in at least one processor. Two combine steps were implemented. The *global-combine* step uses a report procedure to collect the points remaining into a single processor. This is followed by a serial convex-hull algorithm (Graham scan [Se]) on the (remaining) data to determine the final solution. Conversely, the *local-combine* step uses the Graham scan hull algorithm to eliminate points during each iteration of the report step.

5.4. Experimental Results for Convex Hull on Medium-Grain Machine. In this section we give results of convex-hull algorithms that were implemented on a 16-node Intel iPSCl hypercube at SUNY, Buffalo. Both the *global-combine* and the *local-combine* versions of the algorithms were implemented for a variety of iterations of Quickhull. The input to these algorithms is a set S of N random points. The set S is distributed throughout the p nodes of the hypercube so that each node has N/p points. In this section we explore algorithms based on two different input sets. The first is a set of random points chosen from a uniform distribution on the unit square. For this set of input, [PS] has shown that there are $\Theta(\log N)$ expected extreme points. The second, which has $\Theta(N^{1/3})$ expected extreme points [PS], is a set of random points chosen from a uniform distribution on the unit circle. Notice that tilted square distribution has the same expected number of extreme points as square distribution.

Since the expected number of extreme points is relatively small for both sets of inputs, we hope to eliminate quickly most of the points from consideration. The first iteration of the Quickhull algorithm takes $\Theta(N/p)$ time for each processor to find local corner points, $\Theta(\log p)$ time to find and distribute the global corner points, and $\Theta(N/p)$ time to eliminate local points enclosed in the quadrilateral formed by these four global points. For random data on the unit square, the expected number of points remaining after the first iteration of Quickhull is $\Theta(N^{1/2})$. For random data on the unit circle, the expected number of remaining points after the first iteration of Quickhull is $N(1 - 2/\pi) + o(N)$. Notice that at every stage of the parallel convex-hull algorithm for random data on the unit circle there are more points remaining for consideration than for random data on the unit square. Therefore, it is expected that the running times of algorithms for the circular distribution will be longer than for the square distribution.

We chose to implement the Graham scan as the serial convex-hull algorithm since it is efficient and has a guaranteed worst-case $\Theta(N \log N)$ running time. Other serial algorithms, such as "package wrapping" [Se], have a good running time on random points, but the worst-case running time is $\Theta(N^2)$, which occurs when all the points are on the hull. Since the Quickhull algorithm is used for data reduction, the Graham scan algorithm will be more efficient than "package wrapping" during the final combine step since most of the remaining points will be on the convex hull.

The running time of our algorithm is influenced by the number of iterations of the Quickhull algorithm. Let $T_s(N)$ be the running time of the algorithm for N random points chosen from a uniform distribution on the unit square on a hypercube with p processors. Then

$$T_s(N) = \Theta(N/p + \log p) + T'_s(N^{1/2}), \tag{1}$$

where $\Theta(N/p + \log p)$ is the cost to run the first iteration of Quickhull and $T'_s(N^{1/2})$ is the time to run the algorithm on the remaining points. Since each additional iteration of Quickhull on data from a square distribution reduces the number of points by a fraction, the cost of the ith such iteration, for small i, is approximately determined by

$$T'_s(N^{1/2}/c^{i-1}) = T'_s(N^{1/2}/c^i) + \Theta(N^{1/2}/(pc^{i-1}) + 2^{i+1} \log p), \tag{2}$$

where c is the reduction factor. Therefore, the number of iterations needed to reduce the data to its extreme points is approximately

$$x = \lg(N^{1/2}/\lg N)/\lg c. \tag{3}$$

So, it is expected that after $\lceil x + 1 \rceil$ Quickhull iterations the data will be reduced to the set of extreme points.

The analysis as presented would be correct if a uniform reduction of points occurred in all intervals. Unfortunately, this is not a valid expected-case assumption as imbalances in the number of points distributed to regions may arise

throughout the algorithm. Therefore, some regions will require a slightly larger number of iterations, in the expected case, to reduce its set of points to its extreme points. However, the value x can be computed at the outset of the algorithm, and subsequently used to determine a checkpoint for the algorithm. That is, after some number of iterations, dependent on x, a global decision can be made as to how many more iterations of Quickhull will be performed, based on the number of points in intervals, before another checkpoint is made. While the algorithm must terminate eventually, these global checkpoints can also be used to determine a point at which to initiate a local reduction algorithm (for the outstanding intervals), followed by a global combination of these outstanding intervals. This would avoid the situation of a few (bad) intervals resulting in a large number of additional iterations of Quickhull. We observed that after running the algorithm for $\lceil x + 1 \rceil$ iterations, all of the extreme points have been determined.

It should also be noted that it is possible for minor load balancing problems to arise due to imbalances in the number of points that each processor is responsible for as the algorithm progresses. However, these balancing problems are minor since the running times of the algorithm during the later stages are dominated by communication costs.

For random data chosen from a uniform distribution on the unit circle, each Quickhull iteration reduces the number of points by a fraction. The cost of the ith iteration, for small i, is

$$T_c(N/k^{i-1}) = T'_c(N/k^i) + \Theta(N/(pk^i) + 2^{i+1}), \tag{4}$$

where k is the reduction factor. The number of iterations used in the algorithm is based on the expected number of Quickhull iterations and the expected number of extreme points. The number of Quickhull iterations to reduce the data to the set of extreme points can be determined to be

$$y = (2 \log N)/(3 \log k). \tag{5}$$

As in the case of the distribution on the unit square, this analysis is based on a flawed uniform reduction assumption. Nevertheless, both of these analyses can be useful for implementation purposes, as was done for some the results presented below.

Figure 4 gives sample running times for variations of the convex-hull algorithm for 100,000 random points chosen from a uniform distribution on the unit square. The significant reduction in the running time of the algorithm after the first iteration is due to the fact that, for random data on a unit square, the Quickhull data reduction technique reduces the expected number of points from N to $\Theta(N^{1/2})$. Figure 5 is a plot of the number of points remaining versus the number of iterations of the global Quickhull reduction. As shown in Figure 5, the first Quickhull iteration eliminated in excess of 99% of the points, leaving, approximately 750 points to consider from the original 100,000 points. It is interesting to note that since the Graham scan algorithm on each processor runs in $\Theta(N/p \log(N/p))$ time, the first Quickhull iteration itself lead to an 83% reduction in the running time

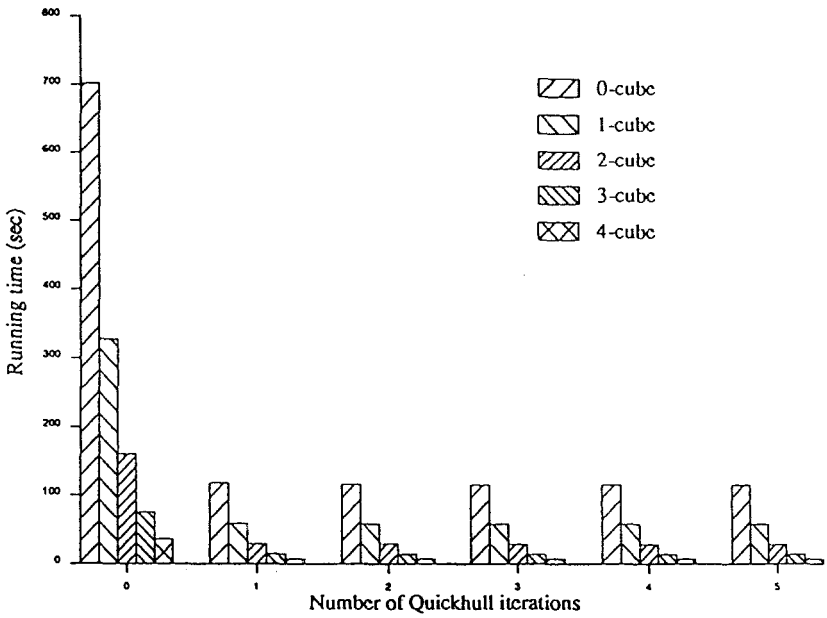

Fig. 4. Convex-hull algorithm on random data from a uniform distribution on the unit square.

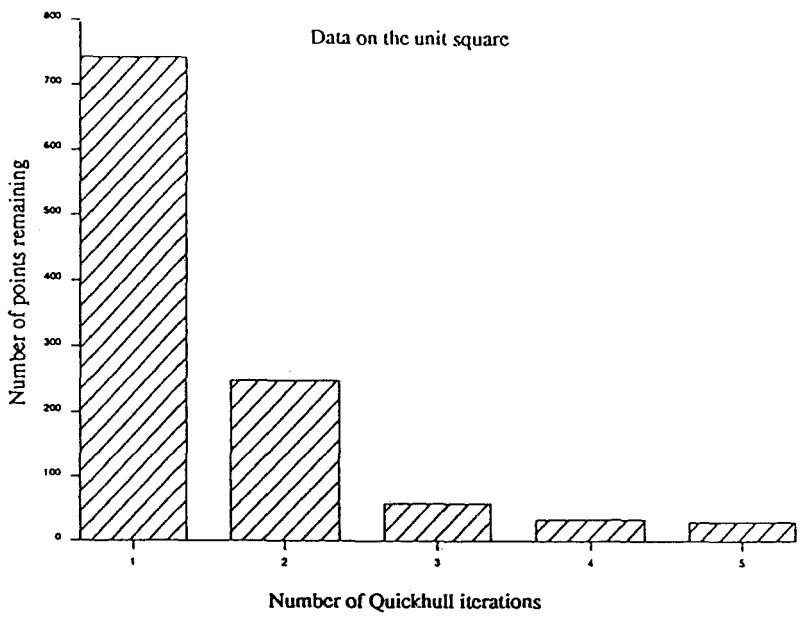

Fig. 5. Number of points remaining after Quickhull iterations from an input of 100,000 random point on a unit square.

Table 4. Speedup of convex-hull algorithm for each iteration of Quickhull for random data on the unit square.

Number of processors	Speedup per iterations of Quickhull					
	0	1	2	3	4	5
0	1.00	1.00	1.00	1.00	1.00	1.00
1	2.14	1.99	1.99	1.99	1.99	1.99
2	4.38	3.96	3.95	3.96	3.96	3.96
3	9.18	7.86	7.89	7.92	7.92	7.92
4	18.99	15.04	15.22	15.46	15.50	15.38

on a single node. After the fifth iteration of the Quickhull algorithm, the data was completely reduced to the set of extreme points. Therefore, additional iterations or a local Quickhull algorithm will only serve to increase the running time of the algorithm.

Table 4 presents the speedups of the algorithm for different numbers of iterations of Quickhull versus the number of processors for random data on the unit square. As can be seen from Figure 4 and Table 4, near linear speedup was achieved with respect to the number of nodes tested. Notice also that this speedup was attained with respect to a given number of iterations of the Quickhull algorithm. For the data on the unit square, it was noticed that after the first iteration of Quickhull, the running times are dominated by the local processing times of the additional Quickhull iterations of the algorithm. The Quickhull iterations run in $\Theta(M + \log p)$ time, where M is the maximum number of points left in any node, $\Theta(M)$ is the cost of local processing, and $\Theta(\log p)$ is the cost of communication. If the number of points left in every node is approximately the same, then it can be expected that doubling the number of nodes will reduce the processing cost by half while adding a linear cost for communication. As long as the local processing is expensive compared with the communication costs, good speedup will be obtained. Eventually, as more nodes are added, the local processing will become less expensive compared with the communication cost, resulting in a situation where adding more nodes will degrade the performance of the algorithm. As can be seen in Figure 4 and Table 4 the speedups generally decrease as the number of processors increases.

The algorithm that we described works well for random data on the unit square, due to the fact that the first Quickhull iteration eliminates most of the points. Next, we consider random data on the unit circle. This is a more interesting situation since the expected number of extreme points is far greater then the expected number of extreme points for random data on the unit square.

Figure 6 shows sample running times for 100,000 random points chosen from a uniform distribution on the unit circle. Notice that the running times without any Quickhull data reductions are approximately the same as the running times of the algorithm on a square distribution of data. This is due to the fact that the running time of the Graham scan algorithm is largely independent of the number

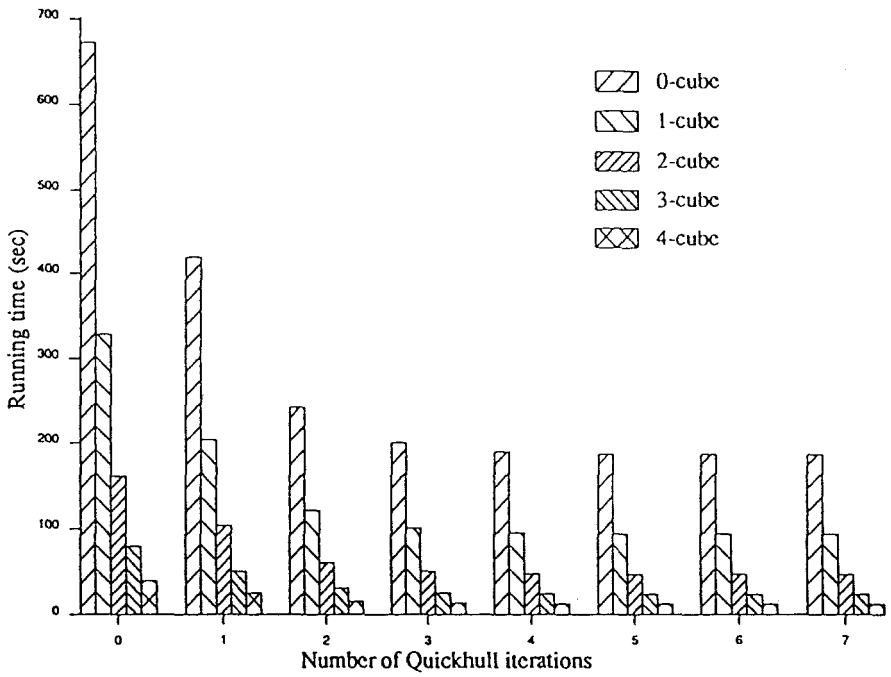

Fig. 6. Convex-hull algorithm for random data from a uniform distribution on the unit circle.

of extreme points. Figure 6 also confirms our expectation that for a circular distribution of data, the first Quickhull iteration will not reduce the running time as significantly as in the case of data on the unit square. The reduction in the running time of the algorithm after the first quadrilateral data reduction step for the circular data was approximately 55% compared with over 80% reduction in the running time of the algorithm for the square data. This is mainly due to the fact that the Quickhull reduction technique does not eliminate points as fast for the circular distribution when compared with the square distribution. As shown in Figure 7, after one iteration of Quickhull on the circular data, approximately 40,000 points remained, where for the square data Figure 5 shows that only 750 points remained. Notice that both of these numbers confirm our earlier analysis concerning the expected number of points remaining after the first Quickhull iteration. It took seven iterations of the Quickhull algorithm to reduce the circular data to its extreme points, as opposed to the five iterations that it took for the square data. Therefore, more work is required to eliminate points in a uniform distribution on the unit circle.

As shown in Figure 6, each of iterations 2 and 3 of the Quickhull algorithm resulted in approximately 40% reduction in the running times from the previous iteration. Figure 8 shows that for data in a circular pattern, the second and third iteration of the Quickhull algorithm will eliminate a large number of points. As can be seen in Figure 7, approximately 9800 and 2550 points remained after iterations 2 and 3 of the Quickhull algorithm, respectively. By analyzing the

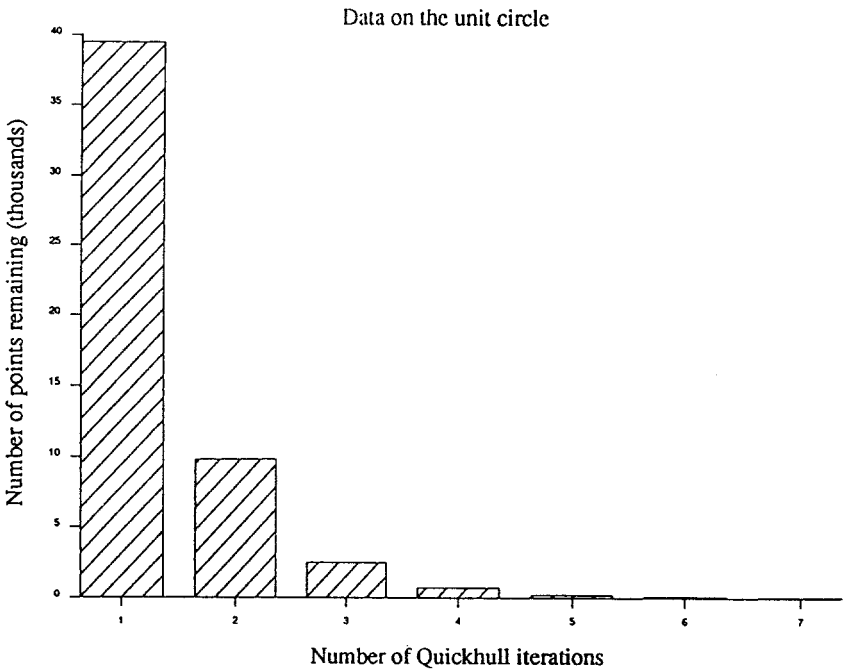

Fig. 7. Number of points remaining after Quickhull iterations from an input of 100,000 random point on a unit circle.

geometric properties of the square and triangles created by the Quickhull technique inside the circle we determined a reduction factor of 3.6 and 4.5 for iterations 2 and 3, respectively. This compares favorably to the results presented in Figure 7. Table 5 shows the speedups of the algorithm with respect to the number of Quickhull iterations versus the size of the cube for random data on the unit circle. Figure 6 and Table 5 show that the parallel convex-hull algorithm resulted in excellent speedups for small-dimensional hypercubes. But, as in the case of the

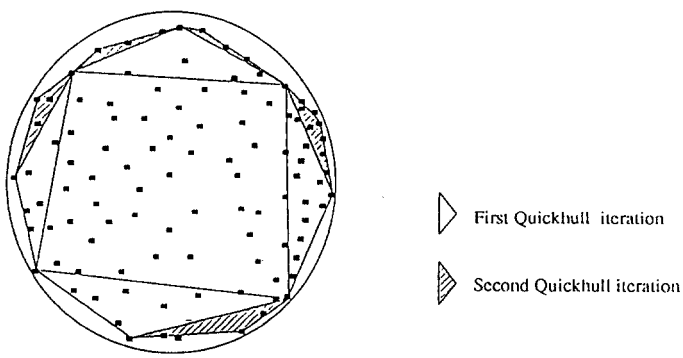

Fig. 8. Quickhull reduction of data on the unit circle.

Table 5. Speedup of convex-hull algorithm for each iteration of Quickhull for random data on the unit circle.

Number of processors	Speedup per iterations of Quickhull							
	0	1	2	3	4	5	6	7
0	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1	2.05	2.06	1.99	1.98	2.00	1.99	1.99	1.99
2	4.17	4.06	3.97	3.88	3.92	3.92	3.94	3.93
3	8.38	8.17	7.73	7.58	7.58	7.62	7.70	7.70
4	6.80	16.24	14.98	14.24	14.14	14.54	14.70	14.66

unit square distribution, the speedups fell below linear for increased iterations of Quickhull and larger-dimensional hypercubes because the local processing became less expensive relative to the cost of communication between the nodes.

Figures 9 and 10 compare the running times of the *local-combine* version of the convex-hull algorithm with the *global-combine* version of the convex-hull algorithm, for square and circular distribution, respectively. Figure 9 presents the results for runs of three, four, and five iterations of Quickhull for square distribution and Figure 10 presents the results for runs of five, six, and seven iterations of Quickhull for circular distribution. As can be seen from Figures 9 and 10, the *global-combine* version ran slightly faster than the *local-combine* version for

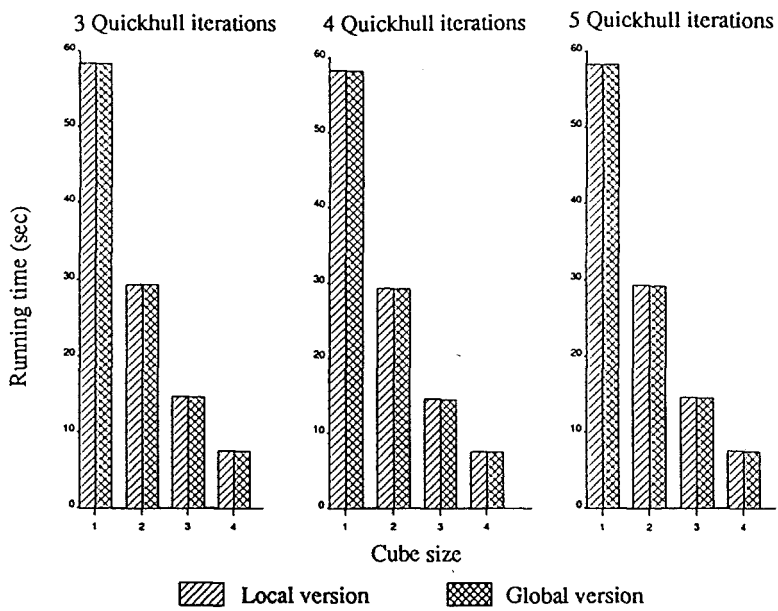

Fig. 9. Local combine version versus global combine version of convex-hull algorithm for random data points on the unit square.

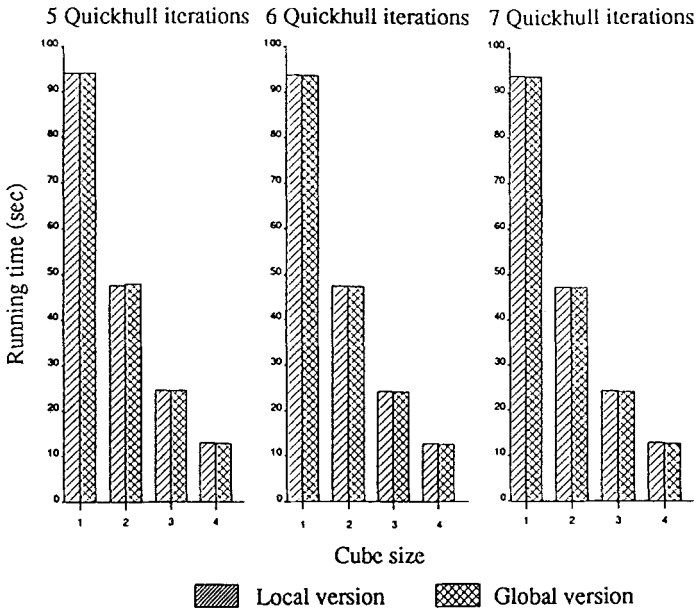

Fig. 10. Local-combine version versus global-combine version of convex-hull algorithm for random data points on the unit circle.

iterations 3, 4, and 5 for the square distribution and iterations 6 and 7 for the circular distribution. However, for fewer iterations of Quickhull, the *local-combine* version is faster than the *global-combine* version. This is due to the fact that for a relatively large number of points that need to be combined, the local Graham scan algorithm during the report operation is more efficient than allowing the points to mount up in node 0 and then running the Graham scan algorithm on all those points. Therefore, we conclude that when the number of points needing to be combined is relatively small, the *global-combine* version of the parallel convex-hull algorithm is desirable.

5.5. Fine-Grain Machines. Algorithms for sets of random points can also be developed that are more efficient than worst-case algorithms on fine-grain machines. For instance, given a set of N random points chosen from a unit square distributed one per processor on a hypercube with N processors, using the first step of the global Quickhull approach reduces the number of points to $\Theta(N^{1/2})$. These $\Theta(N^{1/2})$ points can then be sorted in $\Theta(\log N)$ time [NS], and then the algorithm for presorted data sets can be utilized. This combination of approaches gives the following.

THEOREM 8. *Given a set of no more than N random points chosen from a unit square stored one per processor in an $\lg(N)$-dimensional hypercube, in $\Theta(\log N)$ expected time*

(1) *the maximal points can be determined*;

(2) *the extreme points can be determined, each having its counterclockwise numbering and the coordinates of the following and preceding extreme point*;
(3) *the area, centroid, and perimeter of the convex hull can be determined*; *and*
(4) *a smallest enclosing box and the diameter can be determined.*

5.6. Higher-Dimensional Data. The basic idea of identifying points which eliminate many other points can be extended to points chosen from the unit cube in d-dimensional space, for any fixed d. The expected number of maximal and extreme points is $\Theta(\log^{d-1} N)$ [BKST], and for finding maximal points we can show that the expected number remaining after the best candidate has eliminated points is $\Theta(N^{(d-1)/d})$. The same basic global approach for determining maximal points works essentially as before.

Determining extreme points is somewhat more complicated, but can be approached by first finding 2^d sets of "maximal" points obtained by using all possible choices of $\leq$ and $\geq$ for each dimension, as in Section 2. The extreme points must be a subset of these generalized maximal points, and for fixed d there are only $O(\log^{d-1} N)$ points remaining, on average. For these points, a slower, more complicated algorithm can be utilized, and as long as it runs in polynomial time in the number of points, it will yield a time polylogarithmic in N. One simplistic possibility is to take each point and see if it is contained in the hull of $d+1$ other remaining points, for all possible choices of the $d+1$ points. This requires $\Theta(x^{d+2})$ calculations if there are x points remaining, which can be performed in $\Theta(x^{d+2}/p)$ time by p processors. This is $O(N/p)$, so in O-notation the extra difficulty in determining extreme points does not appear.

THEOREM 9. *For any fixed d, given N points chosen independently from a uniform distribution on the unit d-dimensional cube, distributed N/p per processor in an* $\lg(p)$*-dimensional hypercube, in* $\Theta(N/p + \log p \log^{d-1} N)$ *expected time*

(1) *the maximal points can be determined*;
(2) *the extreme points can be determined, each with its counterclockwise numbering and the coordinates of the following and preceding extreme point*; *and*
(3) *the area, centroid, and perimeter of the convex hull can be determined.*

THEOREM 10. *For any fixed d, given a set of no more than N points chosen independently from a uniform distribution on the unit d-dimensional cube, stored one per processor in an* $\lg(N)$*-dimensional hypercube, then in* $\Theta(\log N)$ *expected time*

(1) *the maximal points can be determined*;
(2) *the extreme points can be determined, each with its counterclockwise numbering and the coordinates of the following and preceding extreme point*;
(3) *the area, centroid, and perimeter of the convex hull can be determined*; *and*
(4) *a smallest enclosing box and the diameter can be determined.*

6. Comments. We have given parallel algorithms for a hypercube to determine geometric properties of sets of points. Because of the dependence upon sorting,

we have also analyzed the time needed if the points were appropriately presorted, or if they were randomly chosen from a uniform distribution. In both situations the times are faster than if sorting is performed using the sorting algorithm currently having the best worst-case sorting time. For medium-grain machines any sorting algorithm must take $\Omega(N \log(N)/p)$ time, and hence when $N/p = \Omega(\lg p)$ the medium-grain algorithms given here for random points will be faster than any general-purpose algorithm for unsorted data. It should be noted that although some of the algorithms we have presented have been optimized for specific distributions of points, each of these works correctly regardless of the distribution.

While we have treated the medium- and fine-grain machines separately, it is possible to combine the approaches and improve upon the results for random points on medium-grain machines when $N = O(p^2)$. In this case, after the first global stage of determining maximal points the expected number of points remaining is less than the number of processors. Redistributing these points results in a configuration suitable for a fine-grain algorithm for nonrandom data. Using this approach shows that each of the problems noted in Proposition 7 and Theorem 8 can be solved in $\Theta(N/p + \log p)$ time, which is optimal for all values of p and N. It is also optimal for any parallel computer with p processors without concurrent write or without concurrent read operations.

References

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

[AG] M. J. Atallah and M. T. Goodrich, Efficient parallel solutions to geometric problems, *J. Parallel Distrib. Comput.* **3** (1986), 492–507.

[Ba] K. E. Batcher, Sorting networks and their applications, *Proc. AFIPS Spring Joint Computer Conf.* (1968), pp. 307–314.

[Be] J. L. Bentley, Multidimensional divide-and-conquer, *Comm. ACM* **23** (1980), 214–229.

[BKST] J. L. Bentley, H. T. Kung, M. Schkolnick, and C. D. Thompson, On the average number of maxima in a set of vectors, *J. Assoc. Comput. Mach.* **25** (1978), 536–543.

[CS1] R. E. Cypher and J. L. C. Sanz, Data reduction and fast routing: a strategy for efficient algorithms for parallel message-passing computers, *Algorithmica*, this issue, 77–89.

[CS2] R. E. Cypher and J. L. C. Sanz, Optimal sorting on feasible parallel computers, in preparation.

[FS] H. Freeman and R. Shapira, Determining the minimal-area encasing rectangle for an arbitrary closed curve, *Comm. ACM* **18** (1975), 409–413.

[K] D. E. Knuth, *The Art of Computer Programming*, Vol. 1, Addison-Wesley, Reading, MA, 1968.

[MM] R. Miller and S. E. Miller, Image processing on hypercube multiprocessors, *Proc. SPIE* **939**, 156–166.

[MS1] R. Miller and Q. F. Stout, Efficient parallel convex hull algorithms, *IEEE Trans. Comput.* **37**(12) (1988), 1605–1619.

[MS2] R. Miller and Q. F. Stout, Mesh computer algorithms for computational geometry, *IEEE Trans. Comput.* **38**(3) (1989), 321–340.

[MS3] R. Miller and Q. F. Stout, *Parallel Algorithms for Regular Architectures*, MIT Press, Cambridge, MA, 1989.

[NS] D. Nassimi and S. Sahni, Parallel permutation and sorting algorithms and a new general interconnection network, *J. Assoc. Comput. Mach.* **29** (1982), 642–667.

[P] C. G. Plaxton, Load balancing, selection and sorting on the hypercube, *Proc. 1st ACM Symp. on Parallel Algorithms and Architecture* (1989), to appear.

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

[RV] J. H. Reif and L. G. Valiant, A logarithm sort for linear size networks, *J. Assoc. Comput. Mach.* **34** (1987) 60–76.

[RS] A. Renyi and R. Sulanke, Uber die konvexe Hulle von n zufallig gewahlten Punkten, *Z. Wahrsch. Verw. Gebiete* **2** (1963), 75–84.

[Se] R. Sedgewick, *Algorithms*, Addison-Wesley, Reading, MA, 1983.

[St] I. Stojmenovic, Personal communication.

[Sto] Q. F. Stout, Constant-time geometry on PRAMs, *Proc. 1988 Internat. Conf. on Parallel Processing*, vol. III, pp. 104–107.

[W] B. Wagar, Hyperquicksort: a fast sorting algorithm for hypercubes, in *Hypercube Multiprocessors*, SIAM, Philadelphia, PA, 1987, pp. 292–299.