Optimal Hypercube Algorithms for Labeled Images

(Preliminary version)

Russ Miller
Department of Computer Science
State University of New York
Buffalo, NY 14260 USA

Quentin F. Stout
Elec. Eng. and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122 USA

Abstract—Optimal hypercube algorithms are given for determining properties of labeled figures in a digitized black/white image stored one pixel per processor on a fine-grained hypercube. A *figure* (i.e., connected component) is a maximally connected set of black pixels in an image. The figures of an image are said to be *labeled* if every black pixel in the image has a label, with two black pixels having the same label if and only if they are in the same figure. We show that for input consisting of a labeled digitized image, a systematic use of divide-and-conquer into subimages of n^c pixels, coupled with global operations such as parallel prefix and semigroup reduction over figures, can be used to rapidly determine many properties of the figures. Using this approach, we show that in $\Theta(\log n)$ worst-case time the extreme points, area, perimeter, centroid, diameter, width and smallest enclosing rectangle of every figure can be determined. These times are optimal, and are superior to the best previously published times of $\Theta(\log^2 n)$.

Index Terms—Parallel algorithms, hypercube computer, convexity, area, perimeter, diameter, smallest enclosing rectangle, image analysis, divide-and-conquer.

1 Introduction

The hypercube computer is a popular parallel architecture that has been used for a wide range of applications. Fine-grained hypercubes are particularly well-suited to solving problems in the field of image analysis, and numerous hypercube algorithms have been developed for this area (c.f., [CySa, CSS87a, KaJa, LAN, MiSt87, MuAb, RaSa]). Using the Gray-code mapping given in Section 2.4, pixels that are neighbors in an image can be mapped to neighboring processors of the hypercube. This is similar to the natural mapping of an image onto a mesh-connected computer, a class of computers often used for pixel-based image processing. In fact, since the hypercube contains the connections of the mesh-connected computer as a subset of its connections, a single step of a mesh-connected computer algorithm can be simulated in constant time on a hypercube, and hence the hypercube can efficiently perform all mesh computer algorithms.

For problems in which communication involves data originating in processors far apart, a hypercube with n processors can send a message between any pair of processors in $O(\log n)$ time, while if many pairs are exchanging information then one can guarantee that all messages arrive in $O(\log^2 n)$ time (see Section 2.2). This permits one to view the hypercube as a pseudo-PRAM (Parallel Random Access Machine), where each PRAM communication step can be simulated by the hypercube in $O(\log^2 n)$ worst-case time.

Other sources of hypercube image algorithms come from simulating algorithms for the mesh-of-trees or pyramid architectures. Both the pyramid and the mesh-of-trees can be embedded into the hypercube so that adjacent pyramid or mesh-of-trees processors are mapped to processors that are separated by no more than two communication links in the hypercube [MiSt87, Sto]. Therefore, any step of a pyramid or mesh-of-trees algorithm can be simulated in constant time on a hypercube, and hence the hypercube can efficiently simulate all image algorithms for these architectures.

While simulation is a powerful technique, it often fails to fully exploit the capabilities of the simulating architecture. We illustrate this by presenting a hypercube algorithm for determining the extreme points of every labeled figure in $\Theta(\log n)$ time. This algorithm does not naturally arise from simulating algorithms written for these other architectures, and it is easy to show that such an algorithm is impossible on all of them except the PRAM and perhaps the mesh-of-trees. It uses a divide-and-conquer approach that could also be used to yield an optimal PRAM algorithm, but our algorithm is more complicated than is needed for an optimal PRAM algorithm. The reason for this complication is that the hypercube is quite sensitive to the amount of data movement required, and we need to insure that the amount of data being moved is restricted to $O(n^{1-c})$, for some fixed c, in order to guarantee that the movement can be completed in $\Theta(\log n)$ time (see Section 2).

We systematically use the same divide-and-conquer approach, combined with efficient global operations for routing, sorting, PRAM simulation, parallel prefix, search, and semigroup reduction over figures, to determine many properties of figures. Algorithms requiring only $\Theta(\log n)$ worst-case time are given for finding the extreme points, area, perimeter, centroid, diameter, width and smallest enclosing rectangle of every figure. Since the solutions to such problems involve combining information from processors arbitrarily far apart, any hypercube algorithm must take $\Omega(\log n)$ time, and therefore all of these algorithms are optimal.

2 Background

In a hypercube with n processors, the processors can be labeled by unique $(\log_2 n)$-bit strings, where two processors have a communication link between them if and only if their labels differ by exactly one bit. We assume that each processor has some fixed number of registers, each of length $\Omega(\log n)$, and that standard operations, including exchanging a word of information with an adjacent processor, take constant time. We assume that each processor initially contains the label of its pixel and any associated data, where the assignment of pixels to processors is given in Section 2.4.

We make the weak assumption that in any time unit a processor can send or receive along only a single communication link. Notice that any pair of processors can communicate in $O(\log n)$ time, if no other communication interferes, since there are no more than $\log_2 n$ communication links in a shortest path between any pair of processors. Further, since there are pairs of processors that are $\log_2 n$ communication links apart, and since every problem considered in this paper may involve communication between arbitrary processors, a worst-case lower bound on the time required to solve any problem considered in this paper is $\Omega(\log n)$. Throughout, all times are worst-case.

2.1 Parallel Prefix

One simple global operation that we will make repeated use of is *parallel prefix* [KRS]. Let $*$ denote an associative binary operation over the values in some set S, and assume that $*$ can be computed in unit time. For example, $*$ may denote maximum over the set of integers. Suppose every processor p of a hypercube with n processors stores a value $v(p) \in S$. Then, using a very simple implementation [Ble], in $\Theta(\log n)$ time every processor p can end up with the value $v(0)*v(1)*\cdots*v(p-1)*v(p-1)$, for all $0 < p \leq n-1$. (Notice that processor 1 ends up with $v(0)$, while processor 0 receives no value.)

Parallel prefix is similar to the "scan" operation in APL, but it can be more flexible. For example, suppose every processor p has a [label, value] pair $[l(p), v(p)]$, where the pairs are ordered so that $l(p) \leq l(p+1)$, and suppose $*$ is defined by

$$[l_1, v_1] * [l_2, v_2] = \begin{cases} [l_1, v_1] & l_1 = l_2 \\ [l_2, v_2] & l_1 \neq l_2 \end{cases} .$$

Then the parallel prefix will result in every processor having the value from the smallest numbered processor with its label. (Notice that processors which are not the smallest numbered with their label obtain this value by the parallel prefix algorithm, while processors which are the smallest numbered with their label already have this value, and may therefore discard the value received during the parallel prefix since the value is attached to the previous label.) By combining a parallel prefix operation with the mirror-image *parallel postfix* operation, one can perform operations such as starting with an ordered set of labels and having every processor determine, say, the sum of all values with its label, all in $\Theta(\log n)$ time.

2.2 Routing and Sorting

While an arbitrary pair of hypercube processors can communicate in $O(\log n)$ time, if many pairs of processors are simultaneously trying to communicate, then the time required for all pairs to finish may be significantly worse. For our purposes, it suffices to consider only routing steps in which every processor sends at most one message, and every processor is the destination of at most one message. Obviously, severe bottlenecks can occur at a single processor if it must either send or receive many messages, but even in our restricted routing situation bottlenecks can occur. Any deterministic oblivious routing scheme (in which the path taken by a message depends only upon its source and destination, and not upon any other messages) must have a worst-case time of $\Omega(n^{1/2}/\log^{1.5} n)$ [BoHo]. Randomized oblivious routing can have an expected time of $\Theta(\log n)$ [Val82], but cannot guarantee good worst-case behavior.

The best way known to guarantee good worst-case routing time is to use sorting. Messages are sorted according to their destination, using pseudo-records with an infinite destination for every processor not sending a message. After the sort, the actual messages are stored in increasing destination order in an initial segment of the processors. These messages can then be routed to their destination in $\log_2 n$ constant-time lock-step stages [MiSt89].

The fastest hypercube sorting algorithm currently known is bitonic sort [Bat], which takes $\Theta(\log^2 n)$ time. Using this will result in routing finishing in $\Theta(\log^2 n)$ time. However, when significantly fewer messages are being sent, faster routing is possible. For any fixed $c > 0$, if there are $O(n^{1-c})$ messages, then they can be sorted in $\Theta(\log n)$ time [NaSa], where the implied constant inside the Θ depends upon c. Using this sort in the routing scheme will result in all messages being delivered in $\Theta(\log n)$ time. Throughout this paper, algorithms are designed so that this faster routing can be achieved.

2.3 PRAM Simulation

A *parallel random access machine (PRAM)* is an idealized parallel model of computation, with a unit-time communication diameter. A PRAM is often described as a machine that consists of a set of identical processors and a global memory, where all processors have unit-time access to any memory location. A *Concurrent Read, Exclusive Write (CREW) PRAM* permits multiple processors to read data from the same memory location simultaneously, but permits only one processor at a time to attempt to write to a given memory location. A *Concurrent Read, Concurrent Write (CRCW) PRAM* permits concurrent reads as above, but allows several processors to attempt writing to the same memory location simultaneously, with some tie-breaking scheme used so that only one of the competing processors succeeds in the write. An *Exclusive Read, Exclusive Write (EREW) PRAM* is the most restrictive version of a PRAM in that only one processor can read and write from a given memory location at a given time.

It is well-known that a combination of sorting and parallel prefix can be used to simulate the ER, CR, EW, and CW PRAM operations on other architectures [MiSt89]. Using this, all of the EW, ER, CW, and CR operations can be simulated on a hypercube with n processors in worst-case $\Theta(\log^2 n)$ time. Further, if there is some fixed $c > 0$ such that no more than $O(n^{1-c})$ processors are reading and writing, then each of these operations can be completed in $\Theta(\log n)$ time.

2.4 Mapping Images onto Hypercubes

Given an $n^{1/2} \times n^{1/2}$ image, where $n^{1/2}$ is a power of 2, there are two principle ways to map the image in a 1-1 fashion onto the processors of a hypercube with n processors. In the *row-major* ordering, pixel (i,j) is mapped to hypercube processor $i \cdot j$, where $\cdot$ indicates the concatenation of the $(0.5\log_2 n)$-bit representations of i and j. While this is quite simple and natural, it has the disadvantage that adjacent pixels, such as (0,1) and (0,2), are not mapped onto adjacent processors.

Mappings which preserve adjacency are based upon Gray codes. The standard binary reflected Gray-code [RND] is a permutation G_d of the integers $\{0,1,2,\ldots,2^d-1\}$, and is defined recursively by $G_1(0)=0, G_1(1)=1$, and

$$G_{d+1}(x) = \begin{cases} 0 \cdot G_d(x) & x < 2^d \\ 1 \cdot G_d(2^{d+1}-1-x) & x \geq 2^d \end{cases} .$$

This mapping has the property that for any integers $d \geq 1$ and $0 \leq x < 2^d$, $G_d(x)$ and $G_d((x \pm 1) \bmod 2^d)$ differ by exactly one bit. By mapping pixel (i,j) to hypercube processor $G_c(i) \cdot G_c(j)$, where $c = 0.5\log_2 n$, pixels sharing edges are mapped to adjacent hypercube processors.

For our purposes, either of these mappings could be used, and from now on we will assume that whenever an image is stored on a hypercube it is stored via one of these mappings. Both of them have the property that if $x \leq n^{1/2}$ is a power of 2, then when the image is partitioned into $x \times x$ subsquares, the subsquares reside in disjoint subcubes of the hypercube. This is the only property we need for our algorithms to work in the times claimed. However, for algorithms that require that every pixel determine the values of its adjacent pixels, where "adjacent" means adjacent in the image, if the row-major mapping is used then the pixels can be moved into the Gray-coded mapping in $\Theta(\log n)$ time.

2.5 Labeling Figures

In a digitized black/white image, two black pixels are said to be *adjacent* if and only if they share an edge. Two black pixels are said to be *connected* if and only if there is a path of adjacent black pixels between them. A *figure* is a maximally connected set of black pixels, which presumably represents some object in the image. By *labeling figures* we mean that every black pixel receives a label, where two pixels receive the same label if and only if they are in the same figure.

The fastest known hypercube algorithms for labeling the figures of a digitized image, distributed one pixel per processor on a hypercube, finish in $\Theta(\log^2 n)$ time. One such algorithm appears in [CSS87a], and is based on simulating the $\Theta(\log n)$ time CRCW PRAM algorithm [ShVi] for labeling the connected components of a graph given as a set of n unordered edges, coupled with the idea of subdividing the image into $n^c \times n^c$ subimages, $c < 1/2$. This approach can also be used to convert the $\Theta(\log n)$ time EREW PRAM algorithm of [LAN, CSS87b] into a $\Theta(\log^2 n)$ algorithm for the hypercube.

We note that since $\Omega(\log n)$ is the largest lower bound known for labeling figures on a hypercube, it is an open question as to the fastest time possible for labeling figures on a hypercube.

3 Extreme Points

The *extreme points* of a figure are the vertices of the smallest convex polygon containing the figure. (See Figure 1.) Extreme points provide a succinct representation of a figure and can be used in efficient algorithms to determine a variety of geometric properties of figures [PrSh, MiSt89]. For several of the algorithms presented in this paper, it will be convenient to have a consistent numbering scheme for the extreme points of a figure. Without loss of generality, we assume the extreme points of a figure are numbered in a counterclockwise fashion, starting with the easternmost point (in case of ties, we start with the southernmost-easternmost point), as shown in Figure 1.

Throughout the rest of the paper, we assume that the hypercube starts with an image in which the figures have been labeled. For determining extreme points, when the algorithm is finished, every processor containing a black pixel will have a Boolean variable "extreme" which is true if and only if its pixel is an extreme point of its figure. Further, every processor containing an extreme point will also contain the coordinates of the preceding and succeeding extreme points, with respect to the counterclockwise numbering of the extreme points.

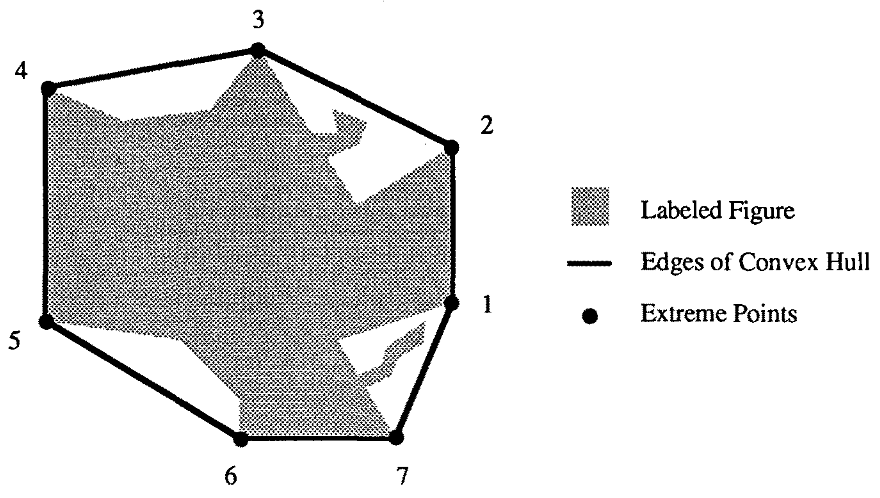

Figure 1: Enumerated Extreme Points.

Our extreme point enumeration algorithm uses a bottom-up divide-and-conquer approach. We partition the image into $n^c \times n^c$ subimages, $c < 1/2$, and find the extreme points of every figure restricted to its subimage, i.e., we temporarily treat each subimage/subcube as if it were an entire image/hypercube. Notice that if p is an extreme point of a figure F, then p is an extreme point of the restriction of F to any subimage containing p. Further, the extreme points of F are the extreme points of the set of extreme points corresponding to all of F's subimages. Therefore, the extreme points of the subimages contain all the information needed to determine the extreme points of the entire image.

When the extreme points of the subimages have been determined, the only figures which are not finished are those that are in more than one subimage. Since figures are connected, every figure in more than one subimage must have a pixel on the border of every subimage that it is in. For each such figure, all of its extreme points from all of the subimages are combined to determine the extreme points of the figure. In every $n^c \times n^c$ subimage, a given figure has only $O(n^{2c/3})$ extreme points [VoKl]. Further, each subimage has only $4n^c - 4$ border pixels, and hence has information pertaining to $O(n^c)$ figures that are not finished. Since there are n^{1-2c} subimages, there are only $O(n^{1-c/3})$ total extreme points from all subimages of all figures which are not finished. A hypercube algorithm for enumerating the extreme points of a set of points in the plane, where the algorithm finishes in time proportional to the time needed to sort, appears in [MiSt88]. Using this algorithm, and the $\Theta(\log n)$ time algorithm to sort a restricted amount of data (Section 2.2), the extreme points of every unfinished figure can now be determined in $\Theta(\log n)$ time.

Theorem 1 *Given a labeled $n^{1/2} \times n^{1/2}$ image stored one pixel per processor in a hypercube with n processors, in $\Theta(\log n)$ time the extreme points of every figure can be determined.*

Proof: To prove that the algorithm finishes in the time claimed, notice that the running time obeys the recurrence

$$T(n) = T(n^{2c}) + \Theta(\log n),$$

which is $\Theta(\log n)$ since $c < 1/2$. □

4 Determining Properties of Figures

By using Theorem 1 to determine the extreme points for every figure, along with the divide-and-conquer approach of using subdivisions of $n^c \times n^c$ subimages, one can determine several properties of figures in logarithmic time. When we say that a property is determined for every figure, we mean that every processor containing a black pixel will know the property with respect to the figure that its pixel is in.

4.1 Semigroup Reduction Over Figures

Let $*$ represent a commutative semigroup operation over some set S, where $*$ can be computed in unit time. For example, $*$ may denote addition over the set of integers. Given that every processor containing a black pixel also stores some data value from S, the following theorem shows that a commutative semigroup operation can be performed over the data values associated with every figure.

Theorem 2 *Given a labeled $n^{1/2} \times n^{1/2}$ image stored one pixel per processor in a hypercube with n processors, suppose every black pixel p has an associated data value $v(p) \in S$. Then in $\Theta(\log n)$ time every black pixel p can determine the result of applying a commutative semigroup operation $*$ to all values associated with the black pixels in its figure.*

Proof: Our algorithm initially uses the bottom-up divide-and-conquer strategy employed in Theorem 1. Suppose that the image has been partitioned into $n^c \times n^c$ subimages, $c < 1/2$, and that in each subimage the result of applying $*$ has been determined for every figure. Since the only figures not finished are those in more than one subimage, a count similar to the one preceding Theorem 1 shows that there are only $O(n^{1-c})$ values to be combined at the final stage. These values can be combined by sorting the restricted number of pieces of data by their figure's label, and then using a parallel prefix operation to combine the values corresponding to the same figure.

If at the end of each bottom-up combination stage a parallel postfix operation is used to inform all pixels involved in the combination as to the combined value for their label, then at the end of the entire bottom-up pass, for every figure the answer is known to those pixels on the borders of the last stage to involve the figure. To disseminate the answer to all pixels in a figure, a final top-down pass is needed. This pass is essentially just the bottom-up pass run in reverse, except that at each stage the border pixels that are acquiring their figure's value merely need to perform a concurrent read from one of the figure's border pixels that is on the border of the next larger subimage. Note that this requires that on the bottom-up pass, a border pixel which had reached the last step that it participated in must also have stored with it the location of a border pixel in the same figure that will participate in later stages, if any such pixel exists. If no such pixel exists then it will have the correct value because it participated in the last stage.

Both the bottom-up and top-down stages satisfy a recurrence of the form

$$T(n) = T(n^{2c}) + \Theta(\log n),$$

and hence finish in $\Theta(\log n)$ time. □

The algorithm of Theorem 2 performs an operation which is sometimes called *semigroup reduction over figures.* This can be viewed as a useful "tool" for constructing parallel algorithms to determine properties of figures, much as parallel prefix is a useful operation for a fairly wide class of parallel algorithms. Simple applications of this operation solve problems such as determining for every figure its area (i.e., the number of pixels in the figure), its perimeter (i.e., the number of pixels along the border of the figure), its centroid (the x-coordinate of the centroid of a figure is the total x-moment divided by the area, and the y-coordinate is the total y-moment divided by the area), and so on. These properties can also be determined with respect to the convex hull of every figure. For example, in order to determine the area of the convex hull of every figure, first determine the extreme points of every figure, then use semigroup reduction over figures so that all extreme points know the extreme point numbered 1 in their figure, use this point to conceptually triangulate the figure, and finally, use semigroup reduction over figures to sum the areas of the triangles for every figure.

Corollary 3 *Given a labeled $n^{1/2} \times n^{1/2}$ image stored one pixel per processor in a hypercube with n processors, in $\Theta(\log n)$ time every black pixel can determine the area, perimeter, centroid, topmost row, bottommost row, leftmost row, and rightmost row of its figure, and the corresponding properties of the convex hull of its figure.* □

4.2 Diameter and Smallest Enclosing Rectangle

The *diameter* of a figure is the maximum distance between any two pixels of the figure. It is straightforward to show that the diameter of a figure F is the maximum distance between two extreme points of F. Further, for any extreme point p, if q is the extreme point farthest from p, then there is some angle $\alpha \in [0, 2\pi)$ such that the halfplane through p at angle α contains the entire figure, and the halfplane through q at angle $(\alpha + \pi) \bmod 2\pi$ contains the entire figure. (See Figure 2.) The angle α must be in the range from the angle of the halfplane through p and its preceding extreme point (and containing the figure), to the angle of the halfplane through p and its following extreme point (and containing the figure). These angles are known as the *angles of support at p.* For example, given an iso-oriented rectangle, the angles of support of the northwest corner are $[\pi, 3\pi/2]$, of the southwest corner are $[3\pi/2, 2\pi) \cup 0$, of the southeast corner are $[0, \pi/2]$, and of the northeast corner are $[\pi/2, \pi]$.

The following search algorithm can be used for every extreme point of a figure to find the distance to a farthest extreme point in its figure. For every extreme point p_i of a figure, with p_{i+1} its succeeding extreme point and p_{i-1} its preceding extreme point, with respect to the counterclockwise ordering of extreme points, we create two *point records* and an *edge record.* The *first point record* contains the label of the figure as major key, the angle of the halfplane (which contains the figure) of hull edge $\overline{p_{i-1}p_i}$ as minor key, with the identity of p_{i-1} and p_i, as well as the ID of the processor creating the record as data.

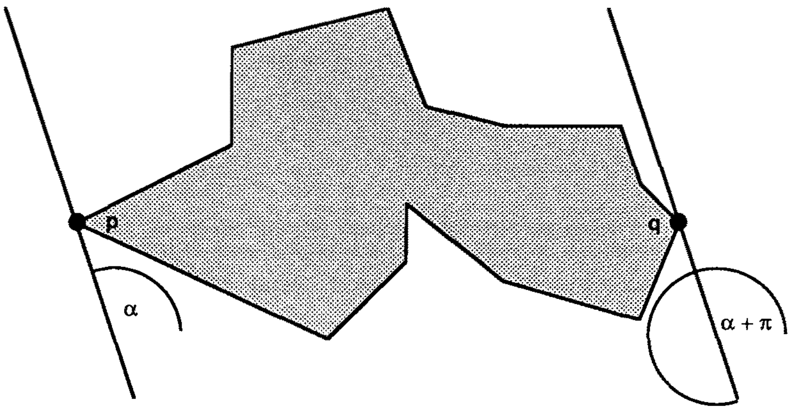

Figure 2: Lines of Support.

The *second point record* contains the label of the figure as major key, the angle of the halfplane (which contains the figure) of hull edge $\overline{p_ip_{i+1}}$ as minor key, with the identity of p_i and p_{i+1}, as well as the ID of the processor creating the record as data. The *edge record* consists of the label of the figure as major key, the angle of the halfplane (which contains the figure) plus π mod 2π of hull edge $\overline{p_ip_{i+1}}$ as minor key, with the identity of p_i and p_{i+1} as well as the ID of the processor creating the record as data.

All point and edge records are sorted together by the major key (figure label), breaking ties by the minor key (angle in $[0, 2\pi)$), and breaking additional ties in favor of a first point record, then an edge record, and finally a second point record. After sorting, an exclusive read is used for every first record to determine the processor index of its associated second record. Within intervals delimited by pairs of point records, perform a semigroup operation to determine the farthest point corresponding to an edge record from the point that determines the interval. A final sort returns all records to the processors that created them. This search is dominated by the time to perform sort, semigroup, prefix, and exclusive read operations. Further, once the search is complete, a simple maximum operation by a semigroup reduction over a figure will determine the diameter of every figure.

To implement this so that every figure determines its diameter, we use a bottom-up procedure similar to that used to find extreme points. As before, if an $n^{1/2} \times n^{1/2}$ image is partitioned into $n^c \times n^c$ subimages, $c < 1/2$, then there are at most $O(n^{1-c/3})$ total extreme points. Therefore the search operation can be completed in $\Theta(\log n)$ time, giving a total time of $\Theta(\log n)$ for the algorithm.

Theorem 4 *Given a labeled $n^{1/2} \times n^{1/2}$ image stored one pixel per processor in a hypercube with n processors, in $\Theta(\log n)$ time the diameter of every figure can be determined.* □

The *width* of a figure can be defined as the minimum of the distances determined by finding for every extreme point p of the figure, the (minimum) distance from a line of support of p to a parallel line of support on the opposite side of the figure. Similarly, a *smallest enclosing rectangle* of a figure, namely a rectangle of minimal area containing the figure, can also be determined using lines of support [FrSh]. Simple modifications to the previous algorithm yield the following.

Corollary 5 *Given a labeled $n^{1/2} \times n^{1/2}$ image stored one pixel per processor in a hypercube with n processors, in $\Theta(\log n)$ time the width of every figure, and a smallest enclosing rectangle for every figure, can be determined.* □

5 Final Comments

We have given several hypercube algorithms which systematically use a divide-and-conquer strategy to determine properties of labeled figures. By dividing an $n^{1/2} \times n^{1/2}$ image into subimages of size $n^c \times n^c$, $c < 1/2$, the amount of data from the subimages which must be combined to finish the entire image is reduced to $O(n^d)$, for some fixed $d < 1$, where d depends upon c and the problem being solved. This enables us to perform global operations such as sorting, routing, or simulating a communication step of a CRCW PRAM, in only $\Theta(\log n)$ time, rather than the $\Theta(\log^2 n)$ time required if $\Theta(n)$ items are being moved. We also note that if our algorithms had used the more standard subdivision into a fixed number of pieces, then the time would have satisfied a recurrence equation of the form

$$T(n) = \Theta(\log n) + T(n/c),$$

which would have a solution of $\Theta(\log^2 n)$.

The use of a divide-and-conquer strategy into data sets of size n^d, $d < 1$, is not new, with its first use in parallel algorithms being for PRAMs [ACGOY, AtGo, Val75]. This strategy was later incorporated into hypercube algorithms, giving the best known algorithms for image component labeling (see Section 2.5) and convex hulls of point sets [CySa, MiSt88]. However, these previous hypercube algorithms require $\Theta(\log^2 n)$ time, as opposed to the $\Theta(\log n)$ time attained here.

Given an image with labeled figures, our approach yields an optimal worst-case $\Theta(\log n)$ time algorithm to enumerate the extreme points of every figure. Further, using this approach, a variety of properties such as area, perimeter, diameter, width, and a smallest enclosing rectangle, can be determined for every figure, or, where appropriate, for every convex hull of a figure, all in $\Theta(\log n)$ time. The final version of this paper will add additional properties to this list, such as deciding for every figure whether or not it is convex. Two useful $\Theta(\log n)$ time global operations for determining such properties are the semigroup reduction over figures, introduced in Theorem 2, and the search operation, introduced in Section 4.2.

Since all of the algorithms presented in this paper involve combining information in processors arbitrarily far apart, and since the communication diameter of the hypercube is $\Theta(\log n)$, all of these algorithms are optimal. This is particularly satisfying since the optimal times to sort, route, label figures, or find extreme points of arbitrary sets of points on a hypercube are not known.

Finally, while we have concentrated on the hypercube because of the widespread interest in such machines, all of the algorithms can be easily modified to run on other machines such as a shuffle-connected computer, cube-connected cycles, or an asynchronous EREW PRAM, so that they still finish in optimal $\Theta(\log n)$ time. Further, the final version of this paper will show that the problems considered here can be solved on these machines in $\Theta(n/p + \log p)$ time, where p is the number of processors. Therefore all of these machines achieve linear speedup for $p \leq n/\log(n)$.

Acknowledgments

This work was partially supported by NSF grants DCR-8608640, IRI-8800514, and DCR-8507851, and by an Incentives for Excellence Award from Digital Equipment Corporation.

References

[ACGOY] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap, "Parallel computational geometry", *Algorithmica* 3 (1988), pp. 293-327.

[AtGo] M.J. Atallah and M.T. Goodrich, "Efficient parallel solutions to some geometric problems", *J. Parallel and Distrib. Comput.* 3 (1986), pp. 492-507.

[Bat] K.E. Batcher, "Sorting networks and their applications", *Proc. AFIPS Spring Joint Comput. Conf.* 32 (1968), pp. 307-314.

[Ble] G. Blelloch, "Scans as primitive parallel operations" *Proc. 1987 Int'l. Conf. Parallel Proc.*, pp. 355-362.

[BoHo] A. Borodin and J.E. Hopcroft, "Routing, merging and sorting on parallel models of computation", *J. Comp. and Sys. Sci.* 30 (1985), pp. 130-145.

[CySa] R. Cypher and J.L.C. Sanz, "Data reduction and fast routing: a strategy for efficient algorithms for message-passing parallel computers", *Algorithmica*, to appear.

[CSS87a] R. Cypher, J.L.C. Sanz, and L. Snyder, "Hypercube and shuffle-exchange algorithms for image component labeling", *Proc. Comp. Arch. Pat. Anal. and Mach. Intel. '87*, pp. 5-10.

[CSS87b] R. Cypher, J.L.C. Sanz, and L. Snyder, "EREW PRAM and Mesh Connected computer algorithms for image component labeling", *IEEE Trans. Pat. Anal. and Machine Intel.*, 11 (1989), pp. 258-262.

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

[KaJa] A.E. Kayaalp and R. Jain, "Parallel implementation of an algorithm for three-dimensional reconstruction of integrated circuit pattern topography using the scanning electron microscope stereo technique on the NCUBE", *Hypercube Multiprocessors 1987*, pp. 438-444.

[KRS] C.P. Kruskal, L. Rudolf, and M. Snir, The power of parallel prefix, *Proc. 1985 Intl. Conf. Parallel Proc.*, pp. 180-185.

[LAN] W. Lim, A. Agrawal, and L. Nekludova, "A fast parallel algorithm for labeling connected components in image arrays", Tech. report NA86-2, Thinking Machines Corp., 1986.

[MiSt87] R. Miller and Q.F. Stout, "Some graph and image processing algorithms for the hypercube", *Hypercube Multiprocessors 1987*, pp. 418-425.

[MiSt88] R. Miller and Q.F. Stout, "Efficient parallel convex hull algorithms", *IEEE Trans. Computers* 37 (1988), pp. 1605-1618.

[MiSt89] R. Miller and Q.F. Stout, *Parallel Algorithms for Regular Architectures*, The MIT Press, 1989.

[MuAb] T.N. Mudge and T.S. Abdel-Rahman, "Vision algorithms for hypercube machines", *J. Parallel and Distrib. Comp.* 4 (1987), pp. 79-94.

[NaSa] D. Nassimi and S. Sahni, "Parallel permutations and sorting algorithms and a new generalized connection network", *J. ACM* 29 (1982), pp. 642-667.

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

[RaSa] S. Ranka and S. Sahni, "Image template matching on SIMD hypercube multicomputers", *Proc. 1988 Intl. Conf. Parallel Proc.*, pp. 84-91.

[RND] E.M. Reingold, J. Nievergelt, and N. Deo, *Combinatorial Algorithms*, Prentice Hall, New York, 1977.

[ShVi] Y. Shiloach and U. Vishkin, "An $O(\log n)$ parallel connectivity algorithm", *J. Algorithms* 3 (1982), pp. 57-67.

[Sto] Q.F. Stout, "Hypercubes and pyramids", *Pyramidal Systems for Computer Vision*, V. Cantoni and S. Levialdi, eds., Sprınger-Verlag, 1986, pp. 75-89.

[Val75] L.G. Valiant, "Parallelism in comparison problems", *SIAM J. Comput.* 4 (1975), pp. 151-162.

[Val82] L.G. Valiant, "A scheme for fast parallel communication", *SIAM J. Comput.* 11 (1982), pp. 350-361.

[VoKl] K. Voss and R. Klette, "On the maximum number of edges of convex digital polygons included into a square", Friedrich-Schiller-Universitat Jena, Forschungsergegnisse, no. N/82/6.