Mesh Computer Algorithms for Computational Geometry

RUSS MILLER, MEMBER, IEEE, AND QUENTIN F. STOUT, MEMBER, IEEE

Abstract—**We present asymptotically optimal parallel algorithms for using a mesh computer to determine several fundamental geometric properties of figures. For example, given multiple figures represented by the Cartesian coordinates of n or fewer planar vertices, distributed one point per processor on a two-dimensional mesh computer with n simple processing elements, we give $\Theta(n^{1/2})$ time algorithms for identifying the convex hull and smallest enclosing box of every figure. Given two such figures, we give a $\Theta(n^{1/2})$ time algorithm to decide if the two figures are linearly separable. Given n or fewer planar points, we give $\Theta(n^{1/2})$ time algorithms to solve the all-nearest neighbor problem for points and for sets of points. Given n or fewer circles, convex figures, hyperplanes, simple polygons, orthogonal polygons, or iso-oriented rectangles, we give $\Theta(n^{1/2})$ time algorithms to solve a variety of area and intersection problems. Since any serial computer has worst case time of $\Omega(n)$ when processing n points, our algorithms show that the mesh computer provides significantly better solutions to these problems.**

Index Terms—**Area, computational geometry, convexity, intersection, mesh computer, parallel algorithms, planar point data, proximity.**

I. Introduction

THE GROWING field of computational geometry has provided elegant and efficient serial computer solutions for a variety of problems. Particular attention has been paid to determining geometric properties of planar figures, such as determining the convex hull, and to determining a variety of distance, intersection, and area properties involving multiple figures. Some general references which describe such problems, show some of their uses, and provide some serial algorithms solving them, are [34], [35], and [43].

Compared to the number of serial computational geometry algorithms, the number of parallel algorithms is quite small. Some parallel algorithms were presented which computed geometric properites of digitized pictures [16], [28], [39], but such problems are significantly different from the problems that arise when the figures are represented as sets of points or line segments, as is the norm in most of computational geometry. In the early 1980's, parallel algorithms for convex hull problems using point data began to appear [2], [13], [32]. In 1984, the authors published a preliminary version of this paper which included parallel algorithms for several problems involving geometric properties and distances [27], and Chazelle published a paper using a one-dimensional systolic computer to solve some distance and intersection problems [12]. Subsequently, additional papers with parallel algorithms have appeared [1], [3], [14], [21], [24], and it can be expected that this trend will continue. Parallel computers provide the possibility of substantial improvements in the running time of algorithms, allowing larger problems to be solved in a feasible amount of time.

Elegant serial solutions to many problems are based on being able to efficiently construct the planar Euclidean Voronoi diagram of a set of planar points, or use sophisticated data structures specially designed for geometric problems [37]. However, it is not clear that manipulating data structures or constructing Voronoi diagrams is as useful in parallel computers, since operations such as following a pointer may be very efficient on a serial computer but less so on a parallel one. Our algorithms are for a local-memory parallel computer where information must be exchanged as messages between processors. In such a setting, the distance information must travel becomes a dominant consideration. While the logical arrangement of information in data structures plays a major role in serial algorithms, the physical arrangement provided by data movement operations such as sorting and compression plays a major role in our parallel algorithms.

Our algorithms are designed for a (two-dimensional) mesh computer, defined in Section II. Several large mesh computers have been constructed [8], [7], [15], predominately for use in low-level processing of digitized pictures. This paper demonstrates that meshes are also suitable for geometric problems, much as [5], [4], [36], and [41] showed that they are suitable for graph problems. Compared to other parallel architectures, meshes have the advantage that several already exist, and their simple near-neighbor wiring allows them to be constructed more economically than, say, hypercubes. Furthermore, if the amount of data per processor is sufficiently high, then for a wide variety of applications the mesh interconnection structure is as good, in an asymptotic sense, as any interconnection network [29]. Some supporters of parallel random access machines (PRAM's) advocate designing all parallel algorithms for PRAM's, and then simulating a PRAM on the actual machine. However, the algorithms in this paper are significantly better than those that could be obtained by simulating PRAM algorithms.

Manuscript received August 23, 1986; revised May 22, 1987 and June 3, 1988. This work was supported in part by the National Science Foundation under Grants MCS-83-01019, DCR-85-07851, DCR-8608640, and IRI-8800514, Research Development Fund Award numbers 150-8536G and 150-9532G from the Research Foundation of the State University of New York, and an Incentives for Excellence Award from Digital Equipment Corporation.

R. Miller is with the Department of Computer Science, State University of New York, Buffalo, NY 14260.

Q. F. Stout is with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109.

IEEE Log Number 8824532.

This paper presents optimal algorithms to solve problems that have not previously been discussed for the mesh. The only problem that has been previously discussed is that of determining a minimal distance spanning tree of planar points [24]. The solution we give in Theorem 5.6 is more time and space efficient than the solution presented in [24].

In Section II, we define the mesh and review fundamental mesh algorithms and data movement operations that are used in this paper. For most of the problems in this paper, the input is n for fewer planar points, or pairs of points representing line segments or edges, distributed one per processor in a two-dimensional mesh computer with n processors. Convex figures are represented by the set of their vertices, and simple polygons are represented by the set of their edges. For problems involving multiple figures, each point or edge will have an associated label identifying its figure.

In Section III, an algorithm is given for finding the convex hull of a set of planar points. This algorithm introduces a *parallel binary search* technique for the mesh. In Section IV, algorithms are presented for determining smallest enclosing rectangles of figures. These algorithms introduce a *grouping* technique for solving search problems involving multiple parallel searches. Variations of this technique are used throughout this paper for a variety of search problems. In Section V, algorithms are presented to solve the all-nearest neighbor problem for a collection of points, to find the minimum distance between two sets of points, to solve the all-nearest neighbor problem for collections of point sets, and to generate a minimal distance spanning tree. In Section VI, algorithms are given for finding nearest neighbors of line segments and for deciding whether or not line segments intersect. These algorithms are used to solve several problems involving simple polygons, including deciding if simple polygons intersect and solving the all-nearest neighbor problem for simple polygons if there are no intersections. The algorithms in this section introduce our implementation of *multidimensional divide-and-conquer* [11] for the mesh.

In Section VII, algorithms are given for deciding whether or not convex hulls intersect and for finding intersections of convex polygons and hyperplanes. In Section VIII, algorithms are given for determining area and intersection properties of iso-oriented rectangles, and the results are extended to circles and orthogonal polygons. Section IX discusses extensions to mesh computers of higher dimensions and to input data of higher dimensions.

Except for the extensions in Section IX and Theorem 5.6, every algorithm in this paper finishes in $\Theta(n^{1/2})$ time, which is optimal for a two-dimensional mesh with n processing elements. Section IX points out that straightforward changes produce optimal algorithms for meshes of higher dimensions and for some of the problems when the input is extended to higher dimensions.

II. Preliminaries

We use Θ to mean "order exactly," O is used to mean "order at most," and Ω is used to mean "order at least." That is, given nonnegative functions f and g defined on the positive integers, we write $f = \Theta(g)$ if and only if there are positive constants C_1, C_2, and a positive integer N such that $C_1 * g(n) \leq f(n) \leq C_2 * g(n)$, whenever $n > N$. We write $f = O(g)$ if and only if there is a positive constant C and an integer N such that $f(n) \leq C * g(n)$, for all $n > N$, and we write $f = \Omega(g)$ if and only if there is a positive constant C and an integer N such that $C * g(n) \leq f(n)$, for all $n > N$.

For problems in this paper that involve distances between figures, the term *distance* is used to mean Euclidean distance. It should be noted that in most cases any reasonable metric will suffice. Let $d(x, y)$ denote the distance between points x and y, and define the *distance between two sets S and T* to be $\min\{d(s,t) | s \in S, t \in T\}$.

A. Mesh Computer

The *mesh computer (mesh) of size n* is a machine with n simple *processing elements (PE's)* arranged in a square lattice. To simplify exposition, we assume $n = 4^c$ for some integer c. For all $i, j \in [0, \cdots, n^{1/2} - 1]$, PE $P_{i,j}$, representing the PE in row i and column j, is connected via bidirectional unit-time communication links to its four *neighbors*, PE's $P_{i\pm1,j\pm1}$, assuming they exist. (See Fig. 1.) Each PE has a fixed number of registers (words), each of size $\Omega(\log n)$, and can perform standard arithmetic and Boolean operations on the contents of these registers in unit time. Each PE can also send or receive a word of data from each of its neighbors in unit time. Each PE contains its row and column indexes, as well as a unique identification register, the contents of which is initialized to the PE's row-major index, shuffled row-major index, snake-like index, or proximity order index, as shown in Fig. 2. (If necessary, these values can be generated in $\Theta(n^{1/2})$ time.)

For many of the algorithms presented in this paper, a mesh of size n will be used to simulate a *desired* mesh of size Cn, $C \geq 1$ a constant, where the desired mesh of size Cn is mapped in a natural fashion onto the mesh of size n so that each processor of the mesh is responsible for C processors of the desired mesh. For instance, a desired mesh of size $9n$ is mapped onto a mesh of size n so that each processor of the mesh is responsible for the natural 3×3 subregion of processors from the desired mesh. Under this mapping, an algorithm that runs in time $T(Cn)$ on a mesh of size Cn, $C \geq 1$ a constant, will run in time $dT(Cn)$ on a mesh of size n, where the constant d depends only on C. Therefore, simulating an algorithm for a desired mesh of size Cn, $C \geq 1$ a constant, on a mesh of size n under the described mapping, will not affect the asymptotic running time of the algorithm as measured by O, Θ, or Ω.

B. Initial Conditions

For all problems involving points or sets of points, we assume that initially there are n or fewer planar points, distributed no more than one per PE on a mesh of size n. Points are always represented by Cartesian coordinates. If the input is sets of points, then each point will also have an attached label indicating the set it belongs to. For problems involving convex polygons, the poygons are represented by the set of their vertices. Circles are always represented by their radius and the Cartesian cooridinates of their center, and they

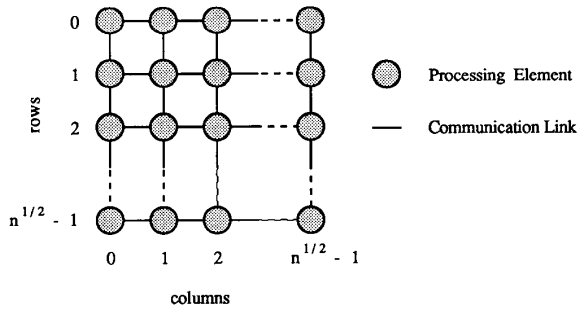


Fig. 1. A mesh-connected computer of size n.

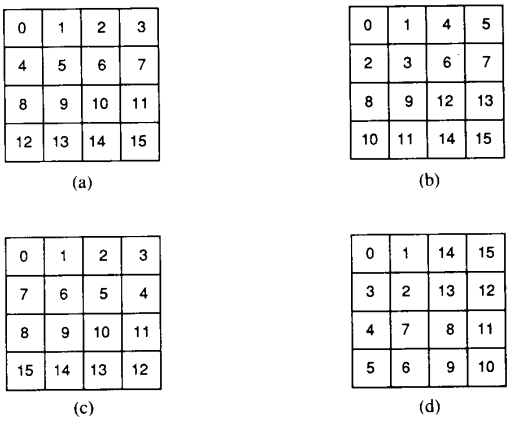


Fig. 2. Indexing schemes for the processors of a mesh. (a) Row major. (b) Shuffled row major. (c) Snake-like. (d) Proximity.

are stored no more than one per PE. Simple polygons (i.e., polygons that do not intersect themselves) are given as line segments represented by the Cartesian coordinates of their endpoints, stored no more than one segment per PE.

For data represented by points, such as point or line segment data, it is assumed that no two distinct points have the same x-coordinate or y-coordinate. It is also assumed that no two endpoints from line segments have the same x-coordinate or y-coordinate, unless they are from line segments that share a common endpoint. These are common assumptions in computational geometry as it simplifies exposition by eliminating special cases. Furthermore, in $\Theta(n^{1/2})$ time, arbitrary input can be rotated to satisfy these assumptions by using sort steps (described in Section II-D) to find the minimum difference in x-coordinates between points with different x-coordinates, the minimum difference in y-coordinates between points with different y-coordinates, the maximum difference in x-coordinates, and the maximum difference in y-coordinates, and then determining a small angle such that rotating by that much will eliminate duplicate coordinates and not introduce new ones.

C. Lower Bounds

For all problems considered in this paper, it is easy to create specific arrangements of data so that the answer cannot be determined faster than the time it takes to combine information starting at opposite corners of the mesh. In a two-dimensional mesh of size n, information starting at opposite corners cannot meet in any PE in less than $n^{1/2} - 1$ time steps. Therefore, all problems considered in this paper must take $\Omega(n^{1/2})$ time on a mesh of size n.

D. Fundamental Operations of the Mesh

Several of the data movement operations in this paper use a proximity ordering of processors (which is based on the concept of space-filling curves). There is no single natural ordering of a two-dimensional mesh, so many orderings of processors have been used. Some of the more useful and popular orderings are given in Fig. 2. Notice that snake-like ordering has the useful property that PE's with consecutive numbers in the ordering are adjacent in the mesh, while shuffled row-major ordering has the property that the first quarter of the PE's form one quadrant, the next quarter form another quadrant, etc., with this property holding recursively within each quadrant. This property of shuffled row-major ordering is useful in many applications of a divide-and-conquer approach. Proximity ordering combines the advantages of snake-like and shuffled row-major order. Given row and column coordinates of a PE P, in $O(\log n)$ time a single processor can compute the proximity order of P by a binary search technique. Similarly, given a positive integer i, the row and column coordinates of the PE with i as its proximity number can be determined in $O(\log n)$ time by a single processor. Given any positive integers $i < j$, the shuffled row-major property of recursively dividing indexes among quadrants gives the property that the distance from PE number i to PE number j is $O((j - i)^{1/2})$, and that a path of length $O((j - i)^{1/2})$ can be achieved using only PE's numbered from i to j. Furthermore, the PE's numbered from i through j contain a subsquare with more than $(j - i)/8$ PE's. The proximity ordering has not yet been widely used in mesh computers (its first appearance being in [40]), but it does seem to combine useful properties of other mesh orderings.

The remainder of this section is devoted to presenting fundamental data movement operations that are used in this paper. Many of these data movement operations will be performed in parallel on items stored in disjoint consecutively numbered (with respect to proximity ordering) processors, which we will refer to as *(ordered) intervals*. It should be noted that ordered intervals may be created by sorting data into proximity order so that related items reside in disjoint consecutively indexed processors. Details of the data movement operations that follow may be found in [30].

1) *Sorting*: [42] showed that n elements, distributed one per PE on a mesh computer of size n, can be sorted in $\Theta(n^{1/2})$ time. While they did not show how to sort into proximity order, one can always sort into proximity order by sorting into snake-like order as defined in [42], computing an item's rank from its current position, computing its destination PE in the proximity order, computing the rank of this destination PE in snake-like order, forming a record with this later rank and the original data, and then sorting these records into snake-like order using rank as the key. For any ordering studied, this entire process can be finished in $\Theta(n^{1/2})$ time. An algorithm

that directly sorts into proximity order can be faster by a multiplicative factor, and such an algorithm would be useful, but this does not affect our analysis. ■

2) *Broadcasting* and *Rotating Data within Intervals:* Suppose every PE contains a record with data, a label, and a Boolean flag called "marked." Furthermore, assume that all PE's containing records with the same value in the label field form an (*ordered*) *interval* with respect to the proximity ordering. Then the data in all records with marked = *true* can be sent to all other PE's holding records with the same label in

$$\Theta(\max\{m(r)+i(r)^{1/2} \mid r \text{ a label}\})$$

time, where $m(r)$ is the number of marked records with label r, and $i(r)$ is the number of records with the label r. This is accomplished by building a breadth-first spaning tree, level by level, within every ordered interval, and then using this spanning tree to perform the desired data movement operation. We first show how to construct the breadth-first spanning tree within every ordered interval and then show how to use it to perform the desired data movement operations.

At time 0, the processor corresponding to the *root* of every tree is identified, with the root of a tree being defined to be at level 0. This is accomplished in $\Theta(1)$ time by having every processor P_i examine the label of processor P_{i-1}, where the indexes are with respect to the proximity order of the processors, and having processor P_i identify itself as the root of the tree for its ordered interval of labels if the label of processor P_{i-1} is different from the label of processor P_i. At time 1, the root of every tree sends a message to all its neighbors with the same label informing them that it is their parent. The root records the identity of these processors as its children, and these neighbors record the identity of the root as their parent, as well as the fact that they are at level 1 of the tree. At time t, processors at level $t-1$ send a message to all neighbors with the same label that have not yet recorded a level. Every processor receiving one or more such messages at time t records the fact that it is at level t in the breadth-first spanning tree of its label. Every processor receiving one or more such messages also picks one of the senders as its parent, records the identity of this chosen parent processor, and sends a message back to the chosen parent processor so that processors at level $t-1$ in the breadth-first spanning tree can record the identity of their children in this tree. Notice that the height of a breadth-first spanning tree for PE's with label r is $\Theta(i(r)^{1/2})$. Therefore, the breadth-first spanning tree for the PE's labeled r is constructed in $\Theta(i(r)^{1/2})$ time, since each step of the level-by-level construction takes $\Theta(1)$ time.

Once the spanning tree is constructed, the marked data are passed up the tree, and when it reaches the root it is passed down, every parent passing it to all its children. Using simple pipelining, the first item reaches all PE's in its interval in $\Theta(i(r)^{1/2})$ time, and each subsequent item follows in $\Theta(1)$ time.

For the situation where one piece of data is circulated within every ordered interval, this operation is referred to as *broadcasting within (ordered) intervals.* For the situation where multiple pieces of data are circulated within ordered intervals, this operation is referred to as *rotating data within (ordered) intervals.* ■

3) *Reporting* and *Semigroup Computation within Intervals:* Suppose every PE has a record with data and a label, and all records with the same label form an ordered interval. Furthermore, suppose a unit-time semigroup operation (such as minimum or summing) is to be applied to all data items with the same label, with all PE's receiving the answer for its data label. Then this can be accomplished in $\Theta(\max\{i(r)^{1/2} \mid r \text{ a label}\})$ time, where $i(r)$ is the number of records with label r. This is performed by forming a breadth-first spanning tree within every ordered interval, followed by having the leaves start passing their values up, where once a PE receives values from all of its children it applies the semigroup operation to these values and its own, and passes up the result. Once the root processor of the spanning tree has computed the answer, the spanning tree is used to broadcast it to all PE's in the interval.

The first phase of the semigroup operation that combines data to the root of the spanning tree within every interval is referred to as *reporting within (ordered) intervals.* Therefore, a *semigroup operation within intervals* is simply a report followed by a broadcast within intervals. ■

4) *Concurrent Read and Concurrent Write:* Two other common data movement operations for the mesh are *concurrent read* and *concurrent write*, also known as *random access read* and *random access write*, respectively. These operations are used to allow the mesh to simulate the concurrent read and concurrent write capabilities of a *Concurrent Read, Concurrent Write Parallel Random Access Machine (CRCW PRAM)*, where multiple processors are permitted to simultaneously read a value associated with a given key (concurrent read), and multiple processors are permitted to simultaneously attempt to update the value associated with a given key (concurrent write). In the case of multiple writes, only one processor succeeds according to some tie-breaking scheme such as minimum data value.

Algorithms for restricted versions of concurrent read and concurrent write were presented in [31]. In this section, we give significantly different algorithms for more general versions of these operations. The concurrent read and concurrent write operations involve two sets of PE's, the *sources* and the *destinations*. Source PE's send a fixed number of records, each consisting of a key and one or more data parts. (A record may also be null.) Destination PE's receive a fixed number of records sent by the source PE's. We allow the possibility that a PE is both a source and a destination.

In a concurrent read, every source PE creates no more than some fixed number of source records, where each source record consists of a key, the data value associated with the key, and some bookkeeping information. It is assumed that no two source records have the same key. The purpose of source records in the concurrent read is to make available the data value associated with every key (assuming that keys are maintained in unique locations throughout the mesh) to any processor that might want to read it. Every destination PE creates no more than some fixed number of destination (*request*) records, each of which specifies the key associated

with a data value it wishes to receive. Several destination records can request a data value associated with the same key. A destination PE may create a destination record for which there is no source record, in which case it receives a null message. An algorithm for the concurrent read is given below.

Concurrent Read:

a) All PE'S create the same fixed number, call it D, of destination (request) records with the desired key and the ID (proximity order index) of the PE that created the record. Some or all destination records created by each PE may be "dummy" records, so as to balance the number of items per processor in subsequent sort steps.

b) All PE's create the same fixed number, call it S, of source records corresponding to those keys for which the PE is responsible. Each record contains the key, the associated data value, and ID of the PE creating the record. Some or all of the source records created by a PE may be "dummy" records, so as to balance the number of items per processor in subsequent sort steps. Furthermore, a key value may be represented by at most one source record somewhere in the mesh.

c) Sort all $(D + S) * n$ source and destination records together into proximity order by key, where ties are broken in favor of source records, and ties between requests are broken arbitrarily.

d) By performing a broadcast operation within ordered intervals with respect to keys, every source record will inform all destination records that requested the value associated with its key as to that value.

e) Sort the $(D + S) * n$ source and destination records by the proximity order index of the PE that originally created them so that they are returned to the PE that originated the request. Notice that the source records are not conceptually needed for the sort step, but are used so as to balance the number of items in each PE during this step. The source records are discarded after the sort is complete.

The concurrent read is accomplished through a fixed number of sort and interval operations, and for fixed constants D and S is completed in $\Theta(n^{1/2})$ time on a mesh of size n. (Notice that throughout most of the algorithm, the mesh of size n simulates a desired mesh of size $(D + S) * n$.)

In a concurrent write, every destination PE creates no more than some fixed number of destination records, consisting of a key and some bookkeeping information, for each of the keys that it maintains and is willing to receive an updated value for. It is assumed that no two destination records contain the same key. At the end of the concurrent write, a destination PE will receive a record corresponding to each of the destination records it created, indicating the new value to be associated with that key. Each key is received by exactly one destination PE, since it is again assumed that data associated with each key is maintained in a unique location in the machine. Every source PE creates no more than some fixed number of source records, each consisting of a key, a data value, and some bookkeeping information. If two or more source PE's send records in an attempt to update the data value associated with the same key, then a destination PE will receive a record containing the minimum such value. (In other circumstances, one could replace minimum with some other tie-breaking mechanism.)

Concurrent Write:

a) All PE's create the same fixed number, call it D, of destination records, corresponding to those keys for which the PE is willing to receive data values. Each record contains a key and the ID (proximity order index) of the PE that created the record. Some or all of the destination records created by a PE may be "dummy" records, so as to balance the number of items per processor in subsequent sort steps. Furthermore, a key value may be represented by at most one destination record somewhere in the mesh.

b) All PE's create the same fixed number, call it S, of source records. Each record contains a key, an associated data value, and the ID of the PE creating the record. Some or all source records created by a PE may be "dummy" records, so as to balance the number of items per processor in subsequent sort steps.

c) Sort all $S * n$ source records by key into proximity order, breaking ties of the same key arbitrarily.

d) Apply the concurrent write tie-breaking mechanism within the ordered intervals. This should be a mechanism computable in $O(n^{1/2})$ time, such as one that can be computed by performing a semigroup operation within ordered intervals. While the only tie-breaker needed in this paper is the minimum, other possibilities are average, product, median, mode, or choosing any value. In every ordered interval, replace the first data value with this new value. This becomes the *representative* for the key.

e) Sort all $(S + D) * n$ source and destination records together by key, where ties are broken in favor of destination records, and ties between source records are broken in favor of the representative record.

f) Using broadcasting within ordered intervals, all destination records obtain their updated value from their representative source record which is stored (if it exists) in the succeeding PE (in proximity order).

g) Sort all $(S + D) * n$ records by the proximity order index of the PE that originally created them so that the destination records take the required data back to the PE responsible for them. Notice that the source records are not conceptually needed for the sort step, but are used so as to balance the number of items in each PE during this step. The source records are discarded after the sort is complete. Like the concurrent read, the concurrent write is accomplished through a fixed number of sort and interval operations, and is completed in $\Theta(n^{1/2})$ time on a mesh of size n. ■

5) *Compression:* Suppose that in a mesh of size n, m pieces of data are randomly distributed one element per PE. Furthermore, suppose that it is desirable to minimize the interprocessor communication time of the PE's that contain these m pieces of data. Then in $\Theta(n^{1/2})$ time this information can be moved to a subsquare of size $\Theta(m)$, where the communication diameter is $\Theta(m^{1/2})$.

Each PE containing one of the m data items creates a record with key 0 and the data item as data, while all other PE's create a record with key 1. These records are then sorted. Each PE receiving an item with key 0 examines the next PE (in proximity order) to see if it too has a 0 key. If not, then this PE computes its proximity rank, which must be m. This value is

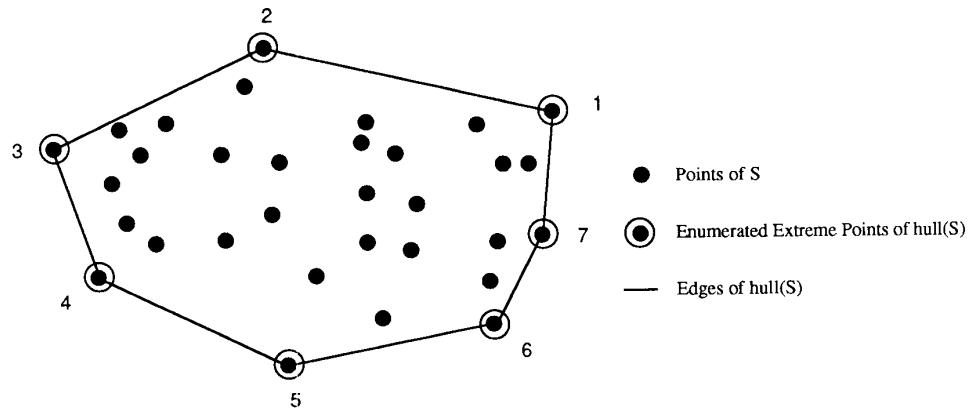


Fig. 3. Convex hull of S.

broadcast to all PE's, which then compute $S = 4^{\lceil \log_4 m \rceil}$. The PE's with a proximity index no greater than S are the desired square, and already have the data. ■

III. Marking the Convex Hull

The convex hull is a geometric structure of primary importance that has been well studied for the serial model of computation [35], [37], [43], [6], [44]. It has applications to normalizing patterns in image processing, obtaining triangulations of sets of points, topological feature extraction, shape decomposition in pattern recognition, and testing for linear separability, to name a few.

In this section, a $\Theta(n^{1/2})$ time algorithm is given for identifying the extreme points that represent the convex hull of a set of n or fewer planar points, initially distributed one point per PE. 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 several of the algorithms presented in this paper, it will be useful to impose an ordering on the extreme points of S. The ordering will be in a counterclockwise fashion, starting with the easternmost point. (Remember from Section II-B, that since the number of points is finite and no two points have the same x-coordinate, there must be a unique easternmost point.) See Fig. 3.

In general, if S is finite, then *hull*(S) is a convex polygon, and the extreme points of S are the corners of this polygon. The edges of this polygon will be referred to as the *edges of the hull*(S). (See Fig. 3.) We say that *the extreme points of S have been identified*, and hence *hull*(S) has been identified, if for each PE P_i containing a point of S, P_i has a Boolean variable "extreme," and extreme is true if and only if the point contained in P_i is an extreme point of S. Furthermore, each PE P_i containing an extreme point of S will contain the position of its point in the counterclockwise ordering, the total number of extreme points, and its adjacent extreme points in the counterclockwise ordering.

Theorem 3.1: Given a set S of n or fewer planar points, distributed no more than one per processor on a mesh computer of size n, the extreme points of S can be identified in $\Theta(n^{1/2})$ time.

Proof: An algorithm to determine the extreme points of *hull*(S) follows. Initially, "extreme" will be set to true for all points. As it is determined that a point is not an extreme point, this flag will be set to false.

1) Sort the points into proximity order using the x-coordinate as the key. Every PE P_i, $1 \le i \le n - 1$, holding a point examines the point in PE P_{i-1} (in proximity order) and if the point in P_{i-1} is the same as the point in P_i, then the point in P_i is viewed as a duplicate and will not be used in steps 2–5.

2) Recursively solve the problems for the points in quadrants A_1, A_2, A_3, and A_4. (See Fig. 4.)

3) From *hull*(A_1) and *hull*(A_2), idenfity *hull*($A_1 \cup A_2$). Call this set of extreme points B_1.

4) From *hull*(A_3) and *hull*(A_4), identify *hull*($A_3 \cup A_4$). Call this set of extreme points B_2.

5) From *hull*(B_1) and *hull*(B_2), identify *hull*($B_1 \cup B_2$). This set of extreme points is *hull*(S).

Notice that in steps 3, 4, and 5, the convex hulls of the two sets of points, say A and B, are used to identify *hull*($A \cup B$). In each of these steps, A and B can be picked so that *hull*(A) lies to the left of *hull*(B), and *hull*(A) does not intersect *hull*(B). This is due to the partitioning of points from step 1. An explanation of how to identify *hull*($A \cup B$) from *hull*(A) and *hull*(B) follows (see Figs. 4 and 5).

Without loss of generality, assume that the points of A are in quadrant A_1, the points of B are in quadrant A_2, and *hull*(A) lies to the left of *hull*(B). The most crucial phase of the algorithm is the identification of points $p, q \in hull(A)$ and $p', q' \in hull(B)$, such that $\overline{pp'}$ and $\overline{qq'}$ represent the top and bottom tangent lines, respectively, between *hull*(A) and *hull*(B). Let $t, u \in hull(A)$ be the westernmost and easternmost extreme points of *hull*(A), respectively. p must lie on or above the line $\overline{tu}$, otherwise $\overline{pp'}$ would intersect *hull*(A). Let $x, y \in hull(A)$ be the extreme points immediately succeeding and preceding (in the counterclockwise ordering of extreme points) p on the *hull*(A), respectively. All points in *hull*(B) must lie below $\overline{xp}$ and some points in *hull*(B) must lie above $\overline{py}$. (Similar remarks can be made about the points p', q, and q'.) A method of identifying the extreme points p and p' by using a *binary search* technique is now described. (A similar technique is used to identify q and q'.)

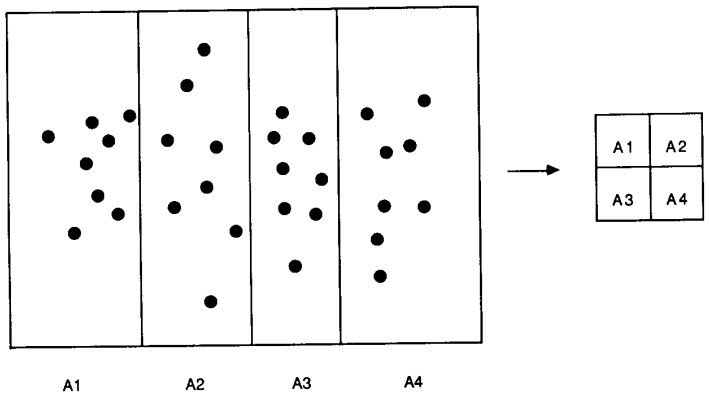


Fig. 4. Mapping points into the proper quadrants.

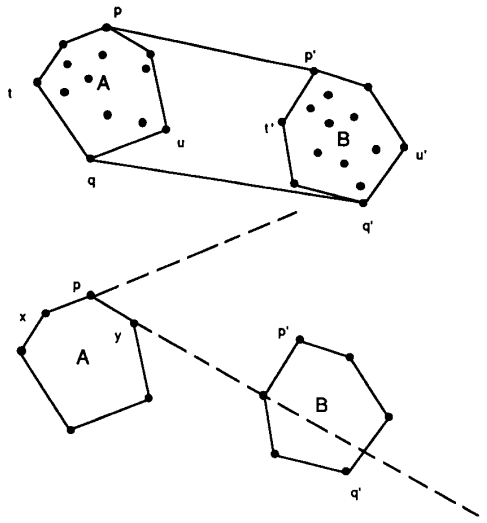


Fig. 5. Stitching convex hulls together.

All PE's of A_1 can *identify* t and u, simultaneously, in $\Theta(n^{1/2})$ time, where identifying t and u means that every PE of A_1 will know the Cartesian coordinates of t and u, as well as the positions of t and u in the counterclockwise ordering of the extreme points of A. This is accomplished by performing two semigroup operations in A_1, where every PE in A_1 containing an extreme point a of *hull*(A) initially creates a record with the x-coordinate of a as key, and the y-coordinate of a and counterclockwise order of a in *hull*(A) as data. The semigroup operations over these records will identify and broadcast the easternmost and westernmost points of A (stored in A_1), as well as their position in the counterclockwise ordering of the extreme points of A, to all PE's of A_1. Without loss of generality, assume that point u is numbered n_0, and point t is numbered n_1, in the counterclockwise ordering of extreme points of A.

Next, every PE in A_1 that contains an extreme point of *hull*(A) decides if its point is above the line $\overline{tu}$. Notice that all such points above the line $\overline{tu}$ have counterclockwise numbers in the set $\{n_0 + 1, n_0 + 2, \cdots, n_1 - 2, n_1 - 1\}$. The PE in A_1 containing the point above $\overline{tu}$ and half way between t and u (i.e., the point numbered $\lceil (n_1 + n_0)/2 \rceil$), identifies this point as p. A $\Theta(n^{1/2})$ time semigroup operation is used to broadcast p to all PE's of A_1. The PE's containing the succeeding and preceding neighbors of p (in the counterclockwise ordering) create the equations of lines $\overline{xp}$ and $\overline{py}$, respectively. Similar computations in A_2 identify p', $\overline{p'x'}$, and $\overline{y'p'}$ for B.

All PE's in A_2 containing a point that is above $\overline{t'u'}$ simultaneously perform a concurrent read to obtain $\overline{xp}$ and $\overline{py}$. Next, these PE's decide if the point that they contain is below $\overline{xp}$ and also if it is above $\overline{py}$. By performing a concurrent write, this information can be sent to the PE in A_1 that contains the point p. This PE can now determine if $\overline{xp}$ is above all of the extreme points in B, and if $\overline{py}$ is below some of the extreme points of B. If both conditions are satisfied, then p, x, and y have been identified. If these conditions are not satisfied, then if $\overline{xp}$ is not above all of the extreme points of B, then assign to u the point x, recompute p as the point half way between t and the new u, compress the data, and iterate the algorithm. If $\overline{xp}$ is above all of the extreme points of B, then assign to t the point y, recompute p, compress the data, and iterate the algorithm. (The corresponding computations for p', q, and q' are similar.)

After $O(\log n)$ iterations, p, p', q, and q' will be identified since each iteration of the binary search for p and p' eliminates half the points in *hull*(A) and half the points in *hull*(B) from further inspection. Each iteration of the binary search operating on data in processors at communication diameter $\Theta(k)$ is dominated by the time required to complete a fixed number of $\Theta(k)$ time data movement operations. Therefore, by compressing the remaining data from *hull*(A) and *hull*(B) jointly into the smallest square set of processors that will hold these data after every iteration of the algorithm, the ith iteration of the algorithm will operate on $\Theta(n/2^i)$ pieces of data at communication diameter $\Theta((n/2^i)^{1/2})$ and finish in $\Theta((n/2^i)^{1/2})$ time. Notice that if the remaining data from *hull*(A) was compressed to the smallest square set of processors in the upper-left corner of A_1 and the remaining data from *hull*(B) was compressed to the smallest square set of processors in the

upper-left corner of A_2, for example, then the during the ith iteration of the algorithm the remaining $O(n/2^i)$ pieces of data would be at communication diameter $\Theta(n^{1/2})$, and hence every iteration of the binary search would take $\Theta(n^{1/2})$ time. With the joint compression of data after each iteration of the algorithm, the time for the binary search to identify the desired points p, p', q, and q' is given by $\sum_{i=0}^{O(\log n)} \Theta((n/2^i)^{1/2})$, which is $\Theta(n^{1/2})$.

Finally, the positions of the extreme points of $hull(A \cup B)$ must be computed. This can be done by all PE's performing a concurrent read for the number of points in $hull(A)$, the number of points in $hull(B)$, and the original positions of p, p, p', q, and q'. Every PE can now compute the correct position of its extreme points in $hull(A \cup B)$, if indeed its point is an extreme point of $hull(A \cup B)$. Using a concurrent read every PE can determine the total number of extreme points and the extreme points that are adjacent to the extreme point that it contains. Therefore, the time to identify $hull(A \cup B)$ from the extreme points of A and B, given so that $hull(A)$ lies to the left of $hull(B)$, $hull(A)$ does not intersect $hull(B)$, and the extreme points are given in a mesh of size n, is dominated by the $\Theta(n^{1/2})$ time for the binary search plus the $\Theta(n^{1/2})$ time used for the data movement operations to compute the final position information, which is $\Theta(n^{1/2})$ time.

From the merge step just described, we know that steps 1, 3, 4, and 5 of the algorithm (refer to the beginning of the proof) each take $\Theta(n^{1/2})$ time. (Notice that steps 3 and 4 can be performed simultaneously.) Since step 2 is a recursive call, the running time of the entire algorithm is given by the recurrence $T(n) = T(n/4) + \Theta(n^{1/2})$, which is $\Theta(n^{1/2})$. ■

The algorithms of Theorem 4.1, Lemma 5.4, and Theorem 6.1 that appear later in this paper will introduce *grouping techniques* that allow for efficient solutions to search problems. As an alternative to the binary search in the algorithm of Theorem 3.1, these may also be used to identify p, p', q, and q' in $\Theta(n^{1/2})$ time.

Suppose that instead of being given a single set S comprised of n or fewer points, the input to the convex hull problem is given as n or fewer *labeled* points representing multiple sets. If there are only a fixed number of labels, say L, then the previous algorithm could be performed L times, once for each labeled set, and still identify the extreme points of every set simultaneously in $\Theta(n^{1/2})$ time. However, the following corollary shows that this restriction is not necessary.

Corollary 3.2: Given n or fewer labeled planar points, distributed no more than one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time the extreme points of every labeled set can be identified.

Proof: The previous algorithm needs to be modified only slightly. Modify the first sentence of step 1 to read, "Sort the n points into proximity order using the label as primary key and the x-coordinate as secondary key." As was noted in Section II-D, if a given label has m points, then those points are now in an interval of processors which contains a square of size greater than $m/8$. The preceding algorithm is then executed inside each such square, where the PE's in such a square simulate 16 PE's of the original algorithm. (This last point ensures that each simulated PE has at most one point.) ■

Once the extreme points of the hull have been identified, several properties of the hull can be quickly determined. The following result is straightforward and the proof will only be sketched.

Corollary 3.3: Given n or fewer labeled points, distributed no more than one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time the area, perimeter, and centroid of the hull of every labeled set can be determined.

Sketch of Proof: The area of the hull of every set of labeled extreme points is computed as follows. Use the algorithm associated with Corollary 3.2 to determine the extreme points of every labeled set of points. Use sorting to gather together all points with the same label, where sorting is performed so that within each labeled set, all extreme points will be stored in counterclockwise fashion before all points interior to the hull. For every set of extreme points, broadcast within ordered intervals the easternmost extreme point of the labeled set, call it p_e, to all PE's in the interval. Every PE in an interval containing extreme point p_i, computes the area of the triangle $p_i p_e p_{i+1}$. See Fig. 6. A semigroup operation within ordered intervals allows every PE to know the total area of the hull of the points with its label, and a concurrent write sends all points back to the PE where they initially resided, along with the total area of the labeled set that the point is a member of.

The perimeter of the hull of each labeled set of points is computed simply by determining the extreme points of every labeled set of points, gathering labeled sets of extreme points together, and then using a semigroup operation within ordered intervals to sum the lengths of the line segments $\overline{p_i p_{i+1}}$, for all extreme points i in the labeled set.

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. To determine the centroid of every hull, form the triangles as in Fig. 6, determine their moments and areas, and then add them to determine the moments and areas of the entire hull. ■

IV. Smallest Enclosing Box

Problems involving smallest enclosing figures have been studied extensively [43], [17], [19]. For certain packing and layout problems it is useful to find a *minimum-area rectangle (smallest enclosing box)* that encloses a set S of planar points. (Notice that while the area of this rectangle is unique, the rectangle itself need not be.) Any enclosing rectangle must clearly enclose $hull(S)$, and [17] has shown that a smallest enclosing box of S must have one side collinear with an edge of the $hull(S)$, and that each of the other three sides must pass through an extreme point of S.

Theorem 4.1: Given a set S of n or fewer planar points, distributed no more than one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time a smallest enclosing box of S can be identified.

Proof: An algorithm for finding a smallest enclosing box of S follows.

1) Find $hull(S)$. Let l represent the number of edges in $hull(S)$.

2) For each edge $e_i \in hull(S)$, $1 \le i \le l$, determine the

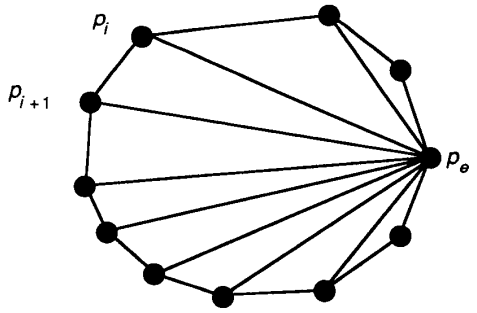


Fig. 6. Computing the area of a convex hull.

minimum-area enclosing box of S that contains a side collinear with e_i. Denote this box as B_i.

3) A smallest enclosing box of S is B_k, where $area(B_k) = \min\{area(B_i) \mid 1 \le i \le l\}$.

The extreme points of S can be identified in $\Theta(n^{1/2})$ time from the algorithm associated with Theorem 3.1. When the algorithm of Theorem 3.1 terminates, every PE containing an extreme point x of S also contains the preceding extreme point w and the succeeding extreme point y, with respect to the counterclockwise ordering of extreme points of S. Each such PE now creates the hull edge $\overline{xy}$ of S. In order to determine the minimum-area rectangle for every edge $\overline{xy}$ of $hull(S)$, each PE containing an edge $\overline{xy}$ needs to know three additional extreme points of S. These points are N, the last extreme point of S encountered as a line parallel to $\overline{xy}$, starting collinear to $\overline{xy}$, passes through the $hull(S)$, W, the last extreme point of S encountered as a line perpendicular to $\overline{xy}$ passes through all of the points of $hull(S)$ from right to left [viewing $\overline{xy}$ as the southernmost edge of $hull(S)$], and E, the last extreme point of S encountered as this perpendicular line passes through $hull(S)$ from left to right. (See Fig. 7.)

Every PE containing an edge $\overline{xy}$ of the $hull(S)$ can find the necessary points, N, W, and E simultaneously in $\Theta(n^{1/2})$ time. Using a semigroup operation, in $\Theta(n^{1/2})$ time determine the easternmost and westernmost extreme points of S, denoted a and b, respectively. The following is a description of how to find the point N for every edge $\overline{xy}$ of the $hull(S)$ that is below $\overline{ab}$ by a *grouping operation*. Each PE P_i that is responsible for an edge $\overline{xy}$ of the $hull(S)$ that is below $\overline{ab}$ creates a *slope record* with the slope of $\overline{xy}$ as key, and i (the proximity order index of the PE) as data. Room is left in this record for a description of the point N that is to be determined. Every processor P_j that is responsible for an extreme point p in $hull(S)$, and contains the preceding extreme point u and the succeeding extreme point v, with respect to the counterclockwise ordering of extreme points of S, determines the slope of $\overline{pu}$. Each such PE P_j for which $\overline{pu}$ is above $\overline{ab}$ creates an *interval record* with the slope of $\overline{pu}$ as key, and p, u, v, and j as data. See Fig. 8.

Sort the slope and interval records together by the key field, with ties broken in favor of interval records. After sorting, every PE P_i that contains an interval record ($slope(\overline{pu})$, p, u, v, j) creates a destination record with v as the key, and a source record with u as the key and the proximity order index i of the PE as data. Perform a concurrent read so that every PE

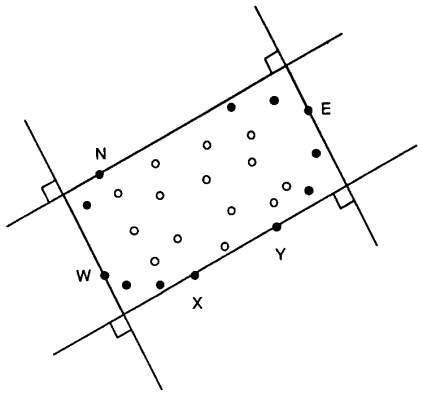


Fig. 7. Determining a smallest enclosing box.

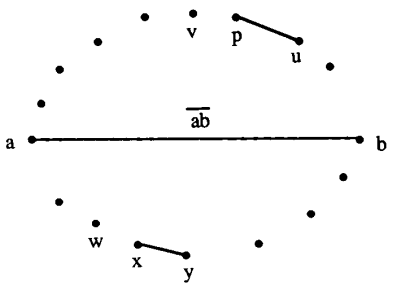


Fig. 8. Creating the slope and interval records.

containing an interval record determines the proximity order index of the next interval record. Notice that every consecutive pair of interval records shares a common extreme point and represents an interval of slopes. Given two consecutive interval records stored in P_i and P_j, with $i < j$, those processors $P_i \cdots P_{j-1}$ form a *group*, and P_i is called the *leader* of the group. Perform a broadcast within every group to notify all slope records as to their desired point N, which is the extreme point common to the pair of interval records that the slope record is between, with respect to the proximity ordering of the processors. This broadcast operation is almost identical to broadcasting within proximity ordered intervals, except that after the leader of a group initializes the creation of the breadth-first spanning tree within its group, data are only passed to those PE's that are in the group. This can be done since the leader knows the index of the first PE (itself) of the group and the last PE of the group (found during the concurrent read).

A concurrent read completes the operation so that PE P_i knows N that corresponds to $\overline{xy}$. Perform the grouping operation a second time with the roles of the slope and interval records reversed so as to find the point N for all hull edges above $\overline{ab}$. By rotating slopes by $\pm \pi/2$, the E and W points can be found in a similar fashion for every edge of $hull(S)$.

In $\Theta(1)$ time, every PE can compute the area of the rectangle formed by its edge $\overline{xy}$ and the three corresponding points, N, W, and E. Once these minimum area rectangles have been determined for every hull edge, a smallest enclosing box of S can be determined by taking a minimum over the area of these

rectangles in $\Theta(n^{1/2})$ time. Hence, the entire algorithm requires $\Theta(n^{1/2})$ time. ■

Simple modifications to the algorithm from Theorem 4.1, as in Corollary 3.2, allow labeled figures to be accounted for in solving the smallest enclosing box problem for multiple figures.

Corollary 4.2: Given n or fewer labeled planar points, distributed one point per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time a smallest enclosing box can be identified for every labeled set. ■

V. Nearest Point Problems

In this section, mesh algorithms are given for problems involving nearest neighbors of planar points. These algorithms include a solution to the minimum spanning tree problem for planar points.

Several standard nearest neighbor problems have been explored for the serial computer (c.f. [37], [43], [22], [9]). One of these problems is the nearest neighbor query. The *nearest neighbor query* requires that a nearest neighbor of a single query point be identified. Given n or fewer points, distributed one point per PE on a mesh of size n, the nearest neighbor query can be solved in $\Theta(n^{1/2})$ time by broadcasting a copy of the query point to all PE's, having every PE compute the distance from its point to the query point, and then taking the minimum over these results.

A more interesting problem is the *all-nearest neighbor problem for points*. That is, given a set of points, find a nearest neighbor from the set for each point of the set. In this section, an optimal $\Theta(n^{1/2})$ time algorithm is presented to solve the all-nearest neighbor problem for points given n or fewer planar points, distributed one point per PE on a mesh of size n. This solution easily yields an optimal mesh solution to the *closest-pair problem for points*, which requires that a closest pair of points from a given set be identified. This can be done on a mesh of size n in $\Theta(n^{1/2})$ time by simply taking the minimum over the all-nearest neighbor distances for every point.

Before solving the all-nearest neighbor problem for points, a lemma is presented which is useful to the algorithms presented later in this section. The nearest-neighbor algorithms described in this section work by finding nearest neighbors in vertical "strips," and then in horizontal "strips." The lemma shows that after these operations, for any rectangular region which is determined by the intersection of a vertical and horizontal strip, there are only a few objects in the region that have not yet found their nearest neighbor. The reader should refer to Fig. 9 during the statement and proof of the following lemma.

Lemma 5.1: Given an arbitrary set S of points in two-dimensional space, and arbitrary real numbers $x_1 < x_2$ and $y_1 < y_2$, let $R = \{(x, y) | x_1 \le x \le x_2 \text{ and } y_1 \le y \le y_2\}$, let $D(p) = \min\{d(p, q) | q \ne p, q \in S\}$, and let $D'(p) = \min\{d(p, q) | q \ne p, x_1 \le x\text{-coordinate of } q \le x_2 \text{ or } y_1 \le y\text{-coordinate of } q \le y_2, q \in S\}$. Then the following hold.

a) If p is any element of $R \cap S$ such that $D(p) < D'(p)$, then there is a corner c of the rectangle R, such that $d(p, c) < D'(p)$.

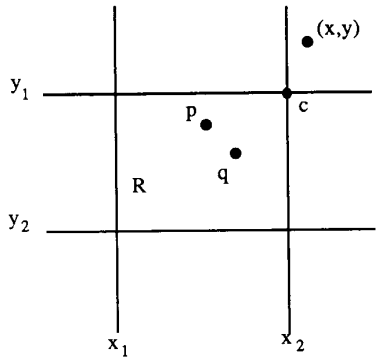


Fig. 9. Nearest neighbor in a corner.

b) There are at most eight elements $p \in R \cap S$ such that there is a corner c where $d(p, c) < D'(p)$.

Proof: To prove a), notice that if $p \in R \cap S$ is such that $D(p) < D'(p)$, then there is an $(x, y) \in S$ such that $D(p) = d(p, (x, y))$, x is not in the interval $[x_1, x_2]$, and y is not in the interval $[y_1, y_2]$. Assume $x > x_2$ and $y > y_2$, with all other cases being identical. In this case, if c is the corner (x_2, y_1), then $d(p, c) < d(p, (x, y)) = D(p) < D'(p)$, as was to be shown.

To show b), let c be any corner, and suppose $p, q \in R \cap S$ are such that $d(p, c) < D'(p)$ and $d(q, c) < D'(q)$. It must be that the angle from p to c to q is at least $\pi/3$ radians, for otherwise the further of p and q would be closer to the other than to c. Therefore, there are at most two points of $R \cap S$ which are closer to c than to any other point in R's vertical or horizontal slab. Since there are only four corners, b) is proven. ■

Theorem 5.2: Given n or fewer planar points, distributed no more than one per processor on a mesh computer of size n, the all-nearest neighbor problem for points can be solved in $\Theta(n^{1/2})$ time.

Proof: The algorithm is recursive in nature. Initially, every PE P_i containing a planar point p_i creates a record with the x-coordinate of p_i as key, and the y-coordinate of p_i and the distance and identity to a nearest point found up to the current iteration of the algorithm as data. The distance is initialized to ∞. Sort the points into proximity order by their x-coordinate. (Recall from Section II-B that no two unique points have the same x-coordinate.) After sorting, let x_1, x_2, x_3, and x_4 be the x-coordinates of the points in processors $P_{n/5}$, $P_{2n/5}$, $P_{3n/5}$, and $P_{4n/5}$ (in proximity ordering), respectively. These values divide the planar points into five vertical *slabs*, namely,

1) $\{p |$ the x-coordinate of $p \le x_1\}$,
2) $\{p | x_1 < x\text{-coordinate of } p \le x_2\}$,
3) $\{p | x_2 < x\text{-coordinate of } p \le x_3\}$,
4) $\{p | x_3 < x\text{-coordinate of } p \le x_4\}$, and
5) $\{p | x\text{-coordinate of } p > x_4\}$.

Recursively solve the all-nearest neighbor problems so that each point finds its nearest neighbor (and associated distance) in its slab. Now repeat the same process for y-coordinates, finding $y_1, \cdots, y_4$ and solving the all-nearest neighbor problems for points within each of the five horizontal slabs.

The planar points can now be thought of as being in at most 25 rectangular regions, determined by $x_1, \cdots, x_4$ and $y_1, \cdots, y_4$. Sort the points by region to create ordered intervals of points corresponding to regions. Within every ordered interval, perform a semigroup operation to determine the, at most, two points (by Lemma 5.1) that are closer to each corner of the region than to their nearest neighbor found so far. All $2 * 4 * 25$ (or fewer) such points are circulated to all n PE's by performing a rotation within the mesh, as described in Section II-D, after which every PE P_i knows the identity and distance from its planar point p_i to a nearest neighbor. (It should be noted that the number of points that actually needs to be circulated can be reduced to 128. Notice that the nine interior squares each have four corners, nine of the exterior squares each have two corners of concern, and the four outer squares each have one corner of concern. This is a total of 64 critical corners, each of which might have two points that need to be involved in the circulating step.)

Sorting and semigroup operation within the ordered intervals corresponding to regions requires $\Theta(n^{1/2})$ time, as does circulating (rotating) a fixed number of points through the mesh. Therefore, the time of the algorithm obeys the recurrence $T(n) = \Theta(n^{1/2}) + 2T(n/5)$, which is $\Theta(n^{1/2})$. ■

Corollary 5.3: Given n or fewer planar points, distributed no more than one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time a closest pair of points can be identified. ■

The next problem considered is the *all-nearest neighbor problem for point sets*. That is, for each set of planar points, find the label and distance to a nearest distinct set of points. When the algorithm terminates, every PE that is responsible for a labeled point will know the nearest neighbor for the set that its point is a member of. It should be noted that a solution to the all-nearest neighbor problem for point sets will not, in general, provide a solution to the problem of detecting for each labeled point a nearest distinctly labeled point. To solve the all-nearest neighbor problem for point sets, the following lemma will be used.

Lemma 5.4: Given n or fewer planar points each labeled either A or B, distributed no more than one per processor on a mesh computer of size n, and given the equation of a line L that separates A and B (i.e, A lies on one side of L and B lies on the other), in $\Theta(n^{1/2})$ time every processor can determine the distance from A to B.

Proof: In $\Theta(n^{1/2})$ time a semigroup operation can determine if either A or B is empty, in which case the answer is infinity. Otherwise, the equation of L, along with a choice of orientation and origin for L, is broadcast to all PE's in $\Theta(n^{1/2})$ time. Suppose $a \in A$ and $b \in B$ are such that $d(a, b)$ equals the distance from A to B. Since A and B are separated by L, the line L must intersect line segment $\overline{ab}$ at some point p. Since $d(a, b)$ is the minimum distance between points in A and points in B, it must be that a is a closest point in A to p, and b is a closest point in B to p. This fact is used in the following solution (see Fig. 10).

1) Partition L into a set of maximal intervals such that for each interval there is a single element of A which is a closest point of A to each point of the interval.

2) Partition L into a set of maximal intervals such that for each interval there is a single element of B which is a closest point of B to each point of the interval.

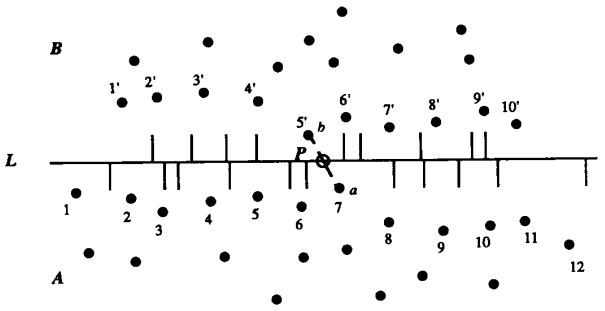


Fig. 10. Partitioning L into maximal intervals. The labeled points correspond to the intervals.

3) Perform an intersection operation on these sets of intervals to determine a closest pair (a_i, b_i), $a_i \in A$, $b_i \in B$, for each interval I_i.

4) Determine $\min \{d(a_i, b_i) | (a_i, b_i)$ is a closest pair of $I_i\}$.

Details of the algorithm follow. First sort the points into sets A and B, on which the partitioning of L proceeds independently and indentically. The partitioning is explained for set A. Intervals will be represented by *interval records* of the form (endpoint1, endpoint2, data), where endpoint1 is the key, and data are the Cartesian coordinates of the point that determines the interval. Every interval of L will be represented twice, once by its left endpoint and once by its right endpoint, where the intervals in the partition overlap only at their endpoints. If A consists of a single point x, then there is only one interval, which is represented by the creation of interval records $(-\infty, \infty, x)$ and $(\infty, -\infty, x)$. If A has more than one point stored in processors numbered 1 through k (in proximity order), then simultaneously and recursively, find the intervals given by the set of points H_1 in processors numbered 1 through $\lfloor k/2 \rfloor$, and those given by the set of points H_2 in processors numbered $\lfloor k/2 \rfloor + 1$ through k.

Notice that an interval from H_1 or H_2 can only shrink or disappear in the final set of intervals for A. Since an interval from H_1 (H_2) may overlap many intervals from H_2 (H_1), the intervals from H_2 (H_1) will be used to determine how much an interval from H_1 (H_2) shrinks. To shrink the intervals, shrink those that came from H_1 first, and then those that came from H_2, as follows.

First, generate interval records that also include a 1 or a 2 to indicate for each record whether it came from H_1 or H_2, respectively. Sort these interval records by their key (which represents an endpoint). In case of ties, a left endpoint of H_1 precedes any endpoint of H_2, and a right endpoint of H_1 follows any endpoint of H_2. For every interval from H_1, the PE's between the representatives of its left and right endpoints form an interval of PE's that is called a *group*. Every PE holding an interval record of H_2 can determine which group the endpoint of the interval is in as follows. All left endpoints of H_1 intervals perform a concurrent read to determine the proximity order index of the processor responsible for the right endpoint of its interval. Viewing each group as an ordered interval, every processor containing a left endpoint of an interval of H_1 (the *leader* of the group) creates a spanning

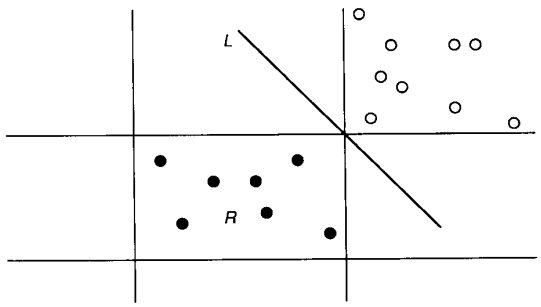
tree in its group, as described in Section II-D and in the algorithm of Theorem 4.1. While the spanning tree is created within every group, all processors representing an interval of H_2 are informed of the point of A that defines the H_1 interval of the group that it is a member of. In a fixed amount of time, every PE containing an interval of H_2 decides whether or not the point corresponding to its interval is closer to *any* of its interval than to the point of H_1 representing the group that it is in. If the answer is affirmative, then the processor can determine which part of the intersection of its interval and the group's interval is closest to its point. By finding a minimum and maximum within intervals of PE's (by computing a semigroup operation within intervals), every group can then determine the final interval (if any) corresponding to the group's point. Finally, repeat the process, interchanging the roles of H_1 and H_2 to determine the final intervals.

Once the partitions corresponding to A and B have been determined, the process of finding a nearest pair between them is similar. First, groups corresponding to intervals of A find the nearest point of any interval of B not properly containing the A interval, and then groups corresponding to intervals of B find the nearest point of any interval of A not properly containing the B interval. Finding the global minimum gives the answer. The running time of the algorithm is given by the recurrence $T(n) = T(n/2) + \Theta(n^{1/2})$, which is $\Theta(n^{1/2})$. ■

Theorem 5.5: Given n or fewer labeled points representing sets of points, distributed no more than one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time the all-nearest neighbor problem for point sets can be solved.

Proof: Every point will try to find the nearest point of a different label, quitting only when it determines that it cannot find a nearer neighbor than some other point can find for its set. The algorithm in Theorem 5.2 is used, resulting in the same conclusion that for each corner of each rectangular region (determined by the intersection of a vertical and horizontal slab) there are points from at most two sets which may be able to find closer points in the direction of the corner. The slight difference is that in each of these regions there may be $O(n)$ points from the same set trying to look in the same direction. For a given rectangular region R, assume that a set A of labeled points is one of the, at most, two closest sets of labeled points to a corner c of the region. A tilted line L through c and tangent to R is a separating line from $A \cap R$ and the points in $S - A$ on the other side of L in the target direction, where S represents the entire set of n labeled points. (See Fig. 11, where the direction is northeast.) Therefore, at most 128 applications of the algorithm associated with Lemma 5.4 are needed. A final concurrent write and concurrent read are used to complete the solution. ■

Given a collection of planar points S, a spanning tree can be constructed by using the points as vertices and straight lines between them as edges. A *minimal distance spanning tree* is a spanning tree of S which minimizes the sum of the Euclidean lengths of the tree edges. A standard approach to building minimal spanning trees is to start off with each point as being its own component with its own label. Then each connected component (as a point set) merges with a nearest neighbor (in case of ties, the one of minimal label is chosen), and an edge



● Points in $A \cap R$

○ Points of S-A in northwest direction from R.

Fig. 11. Solution to the all-nearest neighbor problem for point sets.

corresponding to this minimal distance is added to the edge set of the minimal distance spanning tree. This occurs simultaneously for all components. Every iteration reduces the number of components by at least a factor of two, so at most $\log_2 n$ iterations are needed. Using Theorem 5.2 to find nearest neighbors, and the graph labeling algorithm of [36] to label components, the following is obtained.

Theorem 5.6: Given n or fewer planar points, distributed no more than one per processor on a mesh computer of size n, in $\Theta(n^{1/2} \log n)$ time a minimal distance spanning tree can be constructed. ■

It should be pointed out that this algorithm is not optimal. However, it is superior to the algorithm appearing in [24]. For the same input data, they used a mesh of size $n^{4/3}$, instead of n, and needed $\Theta(n^{2/3} \log n)$ time, instead of $\Theta(n^{1/2} \log n)$. An optimal $\Theta(n^{1/2})$ time mesh algorithm can be derived by using the Voronoi diagram algorithm of [21] coupled with the minimum-weight spanning tree algorithm for graphs appearing in [36].

VI. Line Segments and Simple Polygons

In this section, problems involving line segments and simple polygons are examined. The first problem considered is to determine whether or not there is an intersection among sets of planar line segments. This is a fundamental problem in computational geometry [37], [10], [35]. In fact, [35] conjectures that in order to efficiently solve hidden-line problems, one must first be able to solve basic intersection problems.

The solution to this problem introduces our use of a paradigm known as *multidimensional divide-and-conquer* [11]. In this approach, k-dimensional problems are solved by subdividing them into smaller k-dimensional pieces, plus similar $(k-1)$-dimensional problems. These pieces are solved recursively and are then glued together. When this paradigm is used on parallel computers, the smaller pieces can be solved simultaneously. However, this raises the possibility that an initial object is subdivided into several smaller objects and if the recursion causes this to happen repeatedly there can be an explosion in the amount of data. The algorithms of this section prevent this by ensuring that any initial line segment never has more than two pieces representing it at the start of any level of

recursion, no matter how many levels of recursion have occurred.

Theorem 6.1: Given n or fewer labeled line segments, distributed no more than one per processor on a mesh computer of size n, if no two line segments with the same label intersect other than at endpoints, then in $\Theta(n^{1/2})$ time it can be determined whether or not there are any intersections of line segments with different labels.

Proof: Every PE with a labeled line segment $\overline{ab}$ creates two *line segment records* representing the line segment. One record has the x-coordinate of a as key, with the y-coordinate of a, coordinates of b, and label of $\overline{ab}$ as data, while the other record has the x-coordinate of b as key, with the y-coordinate of b, coordinates of a, and label of $\overline{ab}$ as data. With two keyed line segment records per PE, in $\Theta(n^{1/2})$ time all $2n$ records are sorted into proximity order by the key field.

After sorting, the keys (x-coordinates) of the first record in PE's $P_{n/4}$, $P_{n/2}$, and $P_{3n/4}$ (with respect to proximity order index) are used to partition the plane into four vertical *slabs*. A concurrent read or broadcast provides all PE's with these three values in $\Theta(n^{1/2})$ time. For every record representing the left endpoint a of a line segment $\overline{ab}$, the PE holding the record determines if there are any slabs which the line segment crosses completely. For each such line segment and slab pair, the PE containing the line segment record generates a *spanning line record* equivalent to the line segment record except that the key is the left x-coordinate of the slab. The spanning line records will be used temporarily and then destroyed, which prevents the overaccumulation of data records. Sort the spanning line records by slab, breaking ties arbitrarily, and perform a semigroup operation within every ordered interval corresponding to a slab to enumerate the spanning line segments of the slabs. Finally, using these numbers, a concurrent write is used to send the spanning line records to their slab. This is accomplished in $\Theta(n^{1/2})$ time.

Each slab is now stored in a quadrant of the mesh. Within every quadrant of processors, in $\Theta(n^{1/2})$ time it can be detected if there is an intersection among the line segments that span the slab that is stored in the quadrant. This can be accomplished as follows. Sort the spanning line segments by y-intercept with the left boundary of the slab. This determines for every spanning line segment its position relative to the other spanning line segments with respect to the left boundary of the slab. Repeat the process to find the relative position of every spanning line segment to the other spanning line segments with respect to the right boundary of the slab. If any spanning line segment has different order positions for the left and right boundary, or if there were any ties involving line segments with different labels, then there is an intersection within the slab stored in the quadrant, and the problem is finished.

Otherwise, in every slab the spanning line segments divide the slab into nonoverlapping *regions*. (It should be noted that the property that line segments of the same label can only intersect at their endpoints is used here to guarantee that these regions are nonoverlapping. If arbitrary intersections were allowed among line segments with the same label, then spanning line segments of the same label could cross each other in the interior of the slab.) Any line segment not spanning this slab will intersect spanning line segments if and only if its endpoints lie in different regions or on one of the spanning line segments. (If the line segment does not span, but does extend outside the slab, then one of its endpoints is temporarily treated as being the appropriate y-intercept.)

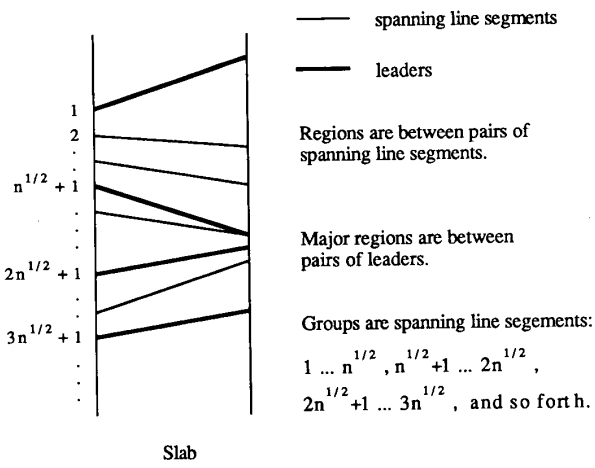


Fig. 12. Spanning line segments, leaders, regions, and major regions.

The following *grouping operation* may be used to determine whether or not spanning line segments are intersected in a given slab. Sort all spanning line segments by the y-intercept of the left boundary. The spanning line segments in every disjoint interval of $n^{1/2}$ PE's form a group, and the first spanning line segment of each group is the *leader* of the group. See Fig. 12. In $\Theta(n^{1/2})$ time, rotate the leaders through the PE's of the slab (stored in a quadrant of the mesh), where the number of processors is $i(r) = \Theta(n)$ and the number of records being rotated is $m(r) = O(n^{1/2})$, in the rotation algorithm of Section II-D. During the rotation, all nonspanning line segment records in the slab determine which *major region* their line segment is in, where a major region is a region of the slab with respect to the leaders' spanning line segments. With respect to the left boundary of the slab, use the y-intercept of the spanning line segments and the y-intercept of the top boundary of the major region for every nonspanning line segment as keys, and sort the spanning line segment and nonspanning line segment records together, with ties broken in favor of spanning line segments. A concurrent read is performed so that the leader of every major region determines the proximity order index of the next leader of a major region, forming a group. Within every group, in $\Theta(n^{1/2})$ time the $O(n^{1/2})$ spanning line segments are rotated to enable each nonspanning line segment record to obtain the identity of the region of the slab that its endpoint is in, as determined by a consecutive pair of spanning line segments. A concurrent read brings the region labels of each endpoint back to the PE responsible for the nonspanning line segments in the quadrant of the mesh maintaining the slab. Every nonspanning line segment now determines if it intersects a spanning line segment, and a semigroup operation determines if any of the spanning lines segments in the slab were intersected.

If there is such an intersection, then the algorithm is done,

while otherwise the spanning line segments are discarded, and in every slab the problem is recursively solved to see if there are any intersections among line segments of different labels with endpoints in the slab.

The running time of a single step of the algorithm is dominated by a fixed number of data movement operations such as sorting, concurrent read, concurrent write, and interval (grouping) operations. Therefore, step i of the algorithm operates on a subsquare of size $k = n/4^i$ and finishes in $\Theta(k^{1/2})$ time. Hence, the running time of the entire algorithm obeys the recurrence $T(n) = T(n/4) + \Theta(n^{1/2})$, which is $\Theta(n^{1/2})$. ■

With minor changes, the above algorithm can be modified to ignore intersections of line segments at common endpoints, or to ignore intersections involving the endpoint of one line segment but the middle of another.

The next problem examined is the *all-nearest neighbor problem for sets of line segments*. That is, for every set of line segments, find the label and distance to a nearest distinct set of line segments. When the algorithm terminates, every PE that is responsible for a labeled line segment will know the nearest neighbor for the set that its line segement is a member of.

Theorem 6.2: Given n or fewer labeled line segments, distributed no more than one per processor on a mesh computer of size n, where line segments intersect at most at their endpoints, in $\Theta(n^{1/2})$ time the all-nearest neighbor problem for sets of line segments can be solved.

Proof: The algorithm combines the ideas of the algorithm in the preceding theorem with that of Theorem 5.5. The plane is divided into five vertical slabs and five horizontal slabs, and nearest neighbors within each are found. To find nearest neighbors in the slabs, first every line segment with an endpoint in the slab finds a nearest spanning line segment, and every spanning line segment finds a nearest neighbor among the spanning line segments and line segments with an endpoint in the slab. The spanning line segments use a concurrent write to report this back to the endpoint that generated them, and are then discarded.

As in Theorem 5.5, after nearest neighbors in slabs have been found, in every rectangular region (determined by the intersection of a vertical and horizontal slab) there are at most eight labels with endpoints of line segments in the region which may not yet have found their nearest neighbor. Lemma 5.4 can be straightforwardly extended to line segments, with the slight difference that k nonoverlapping (except at endpoints) line segments may partition the separating line L into as many $2k - 1$ regions. ■

A simplified algorithm derived from the one in Theorem 6.2, where each line segment finds a nearest neighbor with a different label directly above it will be occasionally useful. Horizontal slabs are not needed, nor is there a final stage involving line segments close to corners of rectangular regions. Since the final stage is eliminated, every line segment can find a nearest neighbor in an upward direction, rather than just finding a nearest neighbor for every label.

A polygon is *simple* if it has the property that every two consecutive edges share only a common endpoint, and no two nonconsecutive edges intersect. While vertices can be used to uniquely represent a convex figure, a more general simple polygon cannot be represented by vertices unless they are given in an enumerated fashion. The input to the problems in this section will be slightly less restrictive than that of enumerated vertices. Simple polygons will be described by line segments that represent their edges.

Some of the algorithms that follow will make use of an efficient solution to the *connected component labeling problem for line segments*, where two line segments are connected if and only if they share a common endpoint. Proposition 6.3 is due to Reif and Stout [36].

Proposition 6.3: Given n or fewer line segments (edges), distributed no more than one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time every PE containing a line segment can know the label of the connected component that its segment is a member of. ■

The first problem of this section involving simple polygons is to determine for every labeled set of line segments whether or not it forms a simple polygon.

Theorem 6.4: Given n or fewer nondegenerate labeled line segments, distributed one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time it can be determined for every set whether or not the line segments form a simple polygon.

Proof: Sort the line segments by sets into proximity order in $\Theta(n^{1/2})$ time. Using the algorithm associated with Proposition 6.3, for every set of line segments simultaneously label all connected components in $\Theta(n^{1/2})$ time. Using a report and broadcast within every set of line segments, in $\Theta(n^{1/2})$ time discard those sets for which not all line segments received the same component label. Next, using a concurrent read within every set, in $\Theta(n^{1/2})$ time mark every line segment that does not satisfy the condition that each of its endpoints intersects exactly one endpoint from a distinct line segment of its component (set). Those components that contain marked line segments do not form simple polygons and are also discarded. For the remaining sets of line segments, apply the intersection algorithm in Theorem 6.1, treating each line segment as having a unique label, and ignoring intersections at common endpoints. Those components that contain intersections are not simple polygons, while the remaining nondiscarded polygons are simple. A final $\Theta(n^{1/2})$ time concurrent read returns the line segments to their original PE's with the solution to the query. ■

For some problems, it is useful to determine if there is an intersection among a set of simple polygons. Before presenting a solution to this problem, a useful result that distinguishes the inside from the outside of each polygon will be given.

Lemma 6.5: Given multiple simple polygons, represented by n or fewer labeled line segments, distributed no more than one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time every processor containing a line segment can determine which side of its line segment is towards the interior of its polygon.

Proof: Every PE that contains a line segment creates a *line segment record* with the polygon label as major key and the x-coordinate of the leftmost of the two endpoints as minor key. Sort the line segment records by polygon labels (key),

with ties broken in favor of minimum x-coordinate. After sorting, the first two line segments of every label intersect at the leftmost point of that polygon. Therefore, their interior angle must be towards the interior of the polygon. For every labeled polygon, select the topmost of these two line segments and conceptually eliminate its link with the line segment at its other endpoint, viewing the line segments as edges between the endpoint vertices. In the remaining graph, select the leftmost point as root, and orient the edges to form an upward directed graph. (Algorithms appearing in [41], [5] do this in the required time.) This graph is a clockwise traversal around the polygon, so for each edge the inside is the right-hand side when going upward (in the tree). A final concurrent read allows every PE to know the orientation of the line segment that it initially contained, with respect to its simple polygon. ■

Theorem 6.6: Given multiple simple polygons, represented by n or fewer labeled line segments, distributed no more than one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time it can be decided whether or not there is an intersection among the polygons.

Proof: From Theorem 6.1, in $\Theta(n^{1/2})$ time it can be determined whether or not there is an intersection between labeled line segments. It only remains to detect if one simple polygon contains another. If there is a containment relationship among some polygons, then there is at least one line segment l for which the closest line segment to l, among the line segments directly above it, is a line segment k of a polygon that contains l. That is, l is on the inside of k, and hence the polygon that l is a member of is contained in the polygon that k is a member of. Furthermore, if no polygons are inside of others, then for every line segment l, the closest line segment k directly above it either belongs to the same polygon as l, or else l is on the outside of k, and hence the polygon that l belongs to is not contained in the polygon that k belongs to.

In $\Theta(n^{1/2})$ time, temporarily give each line segment its own label and use the modified version of the algorithm in Theorem 6.2 to find, for every line segment, the nearest neighbor directly above it (if any). Using Lemma 6.5 to determine orientations, it can then be decided in $\Theta(n^{1/2})$ time if any polygon is contained in another. The algorithm from Theorem 6.1, the modification of the algorithm from Theorem 6.2, and the algorithm from Lemma 6.5 all finish in $\Theta(n^{1/2})$ time. Therefore, the algorithm finishes in the time claimed. ■

The following result is an immediate corollary of Theorem 6.2.

Corollary 6.7: Given multiple nonintersecting simple polygons, represented by n or fewer labeled line segments distributed no more than one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time the all-nearest neighbor problem for simple polygons can be solved. ■

The final result of this section solves the problem of determining for a set of query points and a set of nonintersecting simple polygons, the label of the polygon that each points is in, or the fact that the point is not contained in any polygon.

Corollary 6.8: Given multiple nonintersecting simple polygons, represented by labeled line segments, and given a collection of points, such that there are no more than n segments and points, stored no more than one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time every point can determine the label of a polygon it is in, if any.

Proof: Assign to all points a label that is different from all of the polygons. Then use the modified version of the algorithm in Theorem 6.2 to find the nearest line segment above each point. If the point is on the inside side of this segment the point is in the polygon, while otherwise it is outside of it. ■

VII. Intersection of Convex Sets

This section presents efficient mesh algorithms to determine intersection properties of convex figures. Considerable work for the serial model has been performed on such problems (c.f., [33], [35], [38]), since many of these problems solve classic pattern recognition queries, such as deciding if there is an intersection among the convex hulls of arbitrary sets of planar points. In addition to presenting solutions to intersection problems, this section also presents a solution to the two-variable linear programming problem.

The algorithms in this section make extensive use of the notion of *the angle of a half-plane*, which is in the range $[0, 2\pi)$. To define the angle of a half-plane H, translate the origin so that it lies on the edge of H. The angle of H is the angle α such that H contains all rays from the origin at angles $\alpha + \beta$ for β in $(0, \pi)$. For example, when considering a half-plane determined by the x-axis, the angle of the upper half-plane is 0, while the angle for the lower half-plane is π.

For an extreme point p in a set S, the *angles of support of* p is the interval of angles of half-planes with edges through p which contain S. For example, if S is an iso-oriented rectangle, then the angles of support of the northwest corner are $[\pi, 3\pi/2]$, the angles of support of the southwest corner are $[3\pi/2, 2\pi) \cup 0$, the angles of support of the southeast corner are $[0, \pi/2]$, and the angles of support of the northeast corner are $[\pi/2, \pi]$. For an edge e of the hull of a set S, the *angle of support of* e is the angle of the half-plane containing S with edge containing e.

The first result of this section shows that a mesh computer can be used to efficiently determine whether or not the convex hulls of two arbitrary sets of planar points intersect. A classic related problem is to determine whether or not two sets of planar points S_1 and S_2 are linearly separable [43]. Sets S_1 and S_2 are *linearly separable* if and only if there exists a line in the plane such that all of S_1 lies on one side of the line, and all of S_2 lies on the other side of the line. It is not hard to show that sets of planar points are linearly separable if and only if their convex hulls are disjoint.

Theorem 7.1: Given n or fewer labeled planar points, representing sets S_1 and S_2, distributed no more than one per processor on a mesh computer of size n, it can be determined whether or not *hull*(S_1) *and hull*(S_2) intersect and if they do not intersect then a separating line can be determined, all in $\Theta(n^{1/2})$ time.

Proof: If S_1 and S_2 are separated by a line L, then there are extreme points p in S_1 and q in S_2 such that each of p and q has an angle of support parallel to L, and these lines of support differ by π from each other (see Fig. 13). Furthermore, given extreme points p in S_1 and q in S_2, along with the angles of

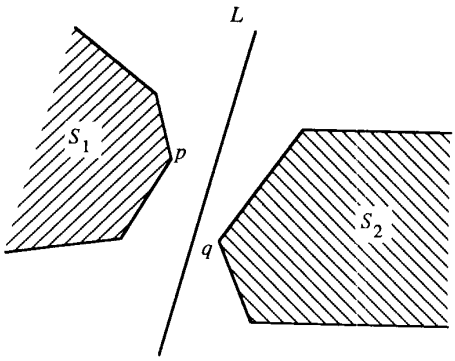


Fig. 13. A separating line L between p and q.

support, in constant time and it can be determined if there is such a separating line, and if there is such a separating line then S_1 and S_2 are linearly separable. To locate a separating line, if one exists, representatives of the angles of support of the extreme points of S_1 and S_2 will use the grouping technique to locate extreme points of the other set with an angle of support differing by π.

Every extreme point p in S_1 creates two records containing p's coordinates and its range of supporting angles, along with an indicator that p is in S_1. One of these records has as its key p's smallest supporting angle, and the other has as its key p's largest supporting angle. Every extreme point of S_2 creates two similar records, adding π (mod 2π) to each angle. Furthermore, to convert the circular ordering of angles of support into a linear ordering, an extreme point of S_1 having 0 as an angle of support, and an extreme point of S_2 having π as an angle of support, creates two additional records having 0 and 2π as keys. All records are then sorted by key, using smallest supporting angles as a secondary key.

Notice that if extreme points p in S_1 and q in S_2 have a separating line, then either an endpoint of q's range of angles of support (plus π) is within the range of p's angles of support, or vice versa, or both. For the first and third cases, if the records are viewed as grouped by intervals of angles of support determined by exteme points in S_1, then by circulating the information about every extreme point in S_1 throughout its interval, every PE holding a record corresponding to an extreme point in S_2 can determine if there is a line separating them. Similarly, viewing the records as grouped by intervals determined by extreme points in S_2 can be used for the third case. ■

The previous theorem demonstrates how to determine if the convex hulls of two sets of planar points intersect. The following theorem shows that the convex hull intersection problem can be solved in the same asymptotically optimal time for multiple sets of planar points.

Theorem 7.2: Given n or fewer labeled planar points, distributed no more than one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time it can be determined if any two labeled sets have convex hulls which intersect.

Proof: Corollary 3.2 gives a $\Theta(n^{1/2})$ time algorithm to enumerate the extreme points of every labeled set. A $\Theta(n^{1/2})$ time concurrent read is used to generate the edges of the convex hulls. Finally, the algorithm of Theorem 6.6 is applied to give the desired result. ■

The next problem examined is that of *constructing* the intersection of multiple half-planes. Serial solutions to this problem appear in [35].

Theorem 7.3: Given the description of n or fewer half-planes, distributed no more than one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time their intersection can be determined.

Proof: Sort the half-planes into proximity order by their angles. Half-planes with the same angle can be intersected into a single half-plane using simple prefix calculations. Then use a simple bottom-up merge technique, where stage i merges 2^i half-planes into their intersection in $\Theta(2^{i/2})$ time. At each stage the result is a perhaps infinite convex figure, and when two figures are being merged the initial sorting guarantees that at most one is noninfinite and either one is contained in the other, they have no intersection, their boundaries intersect in exactly one point, or their boundaries intersect in exactly two points. These cases can easily be determined and solved using, say, the algorithm in Theorem 6.1 to locate the intersections and the algorithm of Theorem 6.6 to determine if there is containment. ■

It was noted in [18] that linear programming can be viewed as an intersection problem, determining the intersection of half-planes and evaluating the objective function at each extreme point. Corollary 7.4 follows directly from Theorem 7.3.

Corollary 7.4: Given n or fewer two-variable linear inequalities, distributed one per processor on a mesh computer of size n, and a unit-time computable objective function to be maximized (minimized), then in $\Theta(n^{1/2})$ time the linear programming problem can be solved. ■

Since a convex polygon is the intersection of the supporting half-planes corresponding to its edges, the problem of *constructing* the intersection of multiple convex polygons is a simple application of the intersection of multiple half-planes. We note that the following corollary can also be obtained by using a bottom-up merging approach which intersects pairs of convex polygons together.

Corollary 7.5: Given multiple labeled convex polygons, represented by n or fewer labeled planar points, distributed no more than one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time the common intersection of the polygons can be constructed. ■

VIII. Iso-Oriented Rectangles and Polygons

Problems involving rectangles have been well studied for the serial model of computation [26], [35], [25], since they are important to many packing and layout problems. An important class of rectangles are the iso-oriented ones, where an *iso-oriented (planar) rectangle* is a planar rectangle with the property that one set of opposite sides is parallel to the x-axis and the other set is parallel to the y-axis. In this section, n or fewer iso-oriented planar rectangles are given, distributed no more than one rectangle per PE on a mesh of size n. It is assumed that each iso-oriented rectangle is described by the Cartesian coordinates of its four planar vertices. To distin-

guish the rectangles during the course of the algorithms, each rectangle can use as its label the index of the PE that contains it. Multidimensional divide-and-conquer, as introduced in Section VI, will be used extensively.

Recall from Section VI that for general simple polygons, it was only possible to detect if an intersection exists. The first theorem of this section shows that when the polygons are restricted to iso-oriented rectangles, then in $\Theta(n^{1/2})$ time, simultaneously for all rectangles, each rectangle can determine whether or not it is intersected by another rectangle.

Theorem 8.1: Given n or fewer iso-oriented planar rectangles, distributed one rectangle per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time every rectangle can determine whether or not it is intersected by another rectangle.

Proof: Each rectangle will have a "left" and "right" representative, corresponding to its two vertical (parallel to the y-axis) edges. Every PE initially holding a rectangle creates the two representatives, one with the x-coordinate of the left side as key, with the rest of the rectangle's description, proximity index of the PE (label of the rectangle), and a flag set to "left" as data, and the other with the x-coordinate of the right side as key, with the description, proximity index of the processor, and "right" as data. After this initialization step there are at most $2n$ representatives. Every PE keeps track of the left and right *limits* of the region under consideration as the algorithm progresses, similar to other slab partitioning algorithms that have been presented in this paper. Initially, every PE sets the left and right limits to $-\infty$ and $+\infty$, respectively.

Sort the representatives by their keys (x-coordinates). The x-coordinates of the second representative in PE's $P_{n/4}$, $P_{n/2}$, and $P_{3n/4}$ (in proximity order) are broadcast to all PE's. This serves to partition the region into four vertical slabs. Every PE holding a representative of a rectangle that spans one or more slabs generates a *special record* describing the rectangle for each slab the rectangle completely crosses. Initially a rectangle can cross at most two vertical slabs, but in latter stages of recursion a rectangle may cross three slabs. The special records are then sent to the quadrant of the mesh holding the spanned slab. This is accomplished by sorting the special records with respect to slabs, performing a semigroup operation within ordered intervals corresponding to slabs to enumerate the special records of each slab, followed by performing a concurrent write to send special records to their appropriate slabs.

In each slab these special records represent spanning rectangles. Notice that a spanning rectangle is intersected by another iso-oriented rectangle in the slab, if and only if their y-coordinates overlap. Therefore, spanning rectangle intersections have been reduced to a one-dimensional intersection problem. First, perform a sort step to eliminate duplicate entries that might have been created by a left and right representative of the same rectangle. Every spanning and nonspanning rectangle creates two records, one corresponding to the y-coordinate of its top edge, and one corresponding to the y-coordinate of its bottom edge. Sort all of these records together. A grouping operation allows every rectangle (spanning or nonspanning) in the slab to determine whether or not it is intersected by a spanning rectangle. The rectangles then report back to the representative that created them, and the spanning rectangles are then discarded.

Next, every PE updates the left and right limits of its slab and the algorithm proceeds recursively. Since the spanning rectangles are discarded before the recursive call, no rectangle can ever have more than two representatives (and up to six spanning rectangle representatives) at any one time. The time of the algorithm satisfies the recurrence $T(n) \leq \Theta(n^{1/2}) + T(n/4)$. Therefore, the algorithm finishes in the time claimed. ■

By applying a similar technique, this result can be extended to circles, assuming that every circle is represented by a record consisting of its center and radius.

Corollary 8.2: Given the descriptions for n or fewer circles, distributed one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time every circle can determine whether or not it is intersected by another circle. ■

The next result proves that the area covered by iso-oriented planar rectangles can be computed in asymptotically optimal time. Notice that if the set of rectangles were nonintersecting, then this would be trivial. No such restrictions are posed on the rectangles.

Theorem 8.3: Given n or fewer iso-oriented planar rectangles, distributed no more than one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time all processors can know the total area covered by the rectangles.

Proof: This algorithm is quite similar to that of Theorem 8.1. At every stage, when the spanning rectangles are sent to each slab, they first determine the measure of the part of the y-axis they cover. The total area covered by the spanning rectangles is this measure times the width of the slab. Every representative of a nonspanning rectangle in the slab now "eliminates" the portion of itself that overlaps spanning rectangles. That is, for a given nonspanning rectangle R with top y-coordinate y_1 and bottom y-coordinate y_2, determine the total measure M_1 of the y-axis covered by spanning rectangles below y_2 and the total measure M_2 of the y-axis covered by spanning rectangles below y_1. The PE responsible for R subtracts M_1 from y_2 and M_2 from y_1. Fig. 14 illustrates this. It has the effect of "cutting out" the spanning rectangles and moving everything else down. The algorithms to determine these measures can be complete in $\Theta(n^{1/2})$ time and are left to the reader. A final semigroup operation will determine the total area covered by the rectangles. ■

Theorem 8.4: Given n or fewer iso-oriented planar rectangles, distributed no more than one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time every processor can know the area which its rectangle covers and which is covered by no other rectangle.

Proof: The algorithm is quite similar to that of Theorem 8.3. Notice that in the algorithm of Theorem 8.3, within every slab each nonspanning rectangle will have its overlap with spanning rectangles eliminated, and hence the area it uniquely covers in that slab will be found by the recursive call. Therefore, it only remains to determine for each spanning rectangle, how much of the slab it uniquely covers. First, each spanning rectangle can "cut out" the part which overlaps with

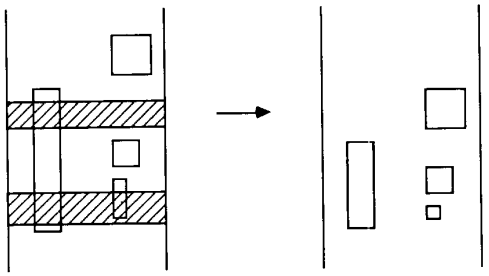


Fig. 14. "Cutting out" the spanning rectangles from a slab.

other spanning rectangles, and each nonspanning rectangle in the slab can similarly "cut out" the part of itself covered by two or more spanning rectangles.

To finish, each nonspanning rectangle creates a temporary representative of itself with the space between the spanning rectangles (i.e., the space not covered by any spanning rectangle) cut out. In each slab, the situation is now that everything is covered by nonoverlapping horizontal bands, and each such band needs to determine the portion of its area covered by nonspanning rectangles. To solve this subproblem within every slab, perform a modification of the algorithm in Theorem 8.3 within the slabs, where the roles of vertical and horizontal are interchanged in the implementation of the algorithm of Theorem 8.3, as follows. First, partition into horizontal slabs the nonspanning rectangles and horizontal bands. In each horizontal slab, the area of a horizontal band that is covered by vertically spanning rectangles is determined by multiplying the total horizontal measure covered by vertically spanning rectangles by the height of the horizontal band (or, for the bands on the upper and lower edges of the horizontal slab, by the height of the band in the slab). Next, the vertically spanning rectangles are eliminated after all remaining rectangles "cut out" the area of themselves that overlap these vertically spanning rectangles. Finally, the problem of determining the remaining area of the horizontal bands covered by those rectangles that have yet to be considered, is solved recursively within the horizontal slabs. ■

By combining the basic technique of multidimensional divide-and-conquer with the use of spanning rectangles, and using the fact that the spanning rectangles have particularly simple properties, the following two results are obtained.

Theorem 8.5: Given n or fewer iso-oriented planar rectangles, distributed no more than one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time every processor can know a nearest neighboring rectangle to the one that it contains. ■

Theorem 8.6: Given a total of n or fewer iso-oriented planar rectangles and planar points, distributed no more than one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time every processor containing a point can determine the number of rectangles containing the point, and every processor containing a rectangle can determine the number of points contained in the rectangle. ■

A minor modification to Theorem 8.5 will yield an optimal mesh solution to the *all-nearest neighbor problem for circles.*

Corollary 8.7: Given n or fewer nonintersecting circles, distributed no more than one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time every processor can know a nearest neighboring circle to the one that it contains. ■

Problems in VLSI layout often involve more than iso-oriented rectangles. Frequently the objects that need to be considered are simple polygons with iso-oriented sides, often called *orthogonal polygons.* It is quite straightforward to add horizontal line segments which decompose each orthogonal polygon into a collection of rectangles overlapping only along their edges, where the number of rectangles is less than the number of initial edges, and where the process takes only $\Theta(n^{1/2})$ time. (See Fig. 15.) Having done this, each of the results in Theorems 8.3, 8.5, and 8.6 can be extended to orthogonal polygons, still requiring only $\Theta(n^{1/2})$ time. The only difference is that Theorem 8.5 must be extended to handle labeled rectangles, finding the nearest neighbor of a different label. This is straightforward and will be omitted.

Theorem 8.8: a) Given multiple simple polygons with iso-oriented sides, represented by n or fewer labeled line segments, distributed no more than one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time the total area covered by the polygons can be determined, a nearest neighbor of each polygon can be determined, and the area uniquely covered by each polygon can be determined.

b) Given a total of n or fewer labeled line segments (representing iso-oriented simple polygons) and planar points, distributed no more than one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time every processor containing a point can determine the number of polygons containing the point, and every processor containing a line segment of a polygon can determine the number of points contained in its polygon. ■

IX. Further Remarks

Given n or fewer planar points, distributed one point per processor on a mesh computer of size n, algorithms have been presented to determine a number of formal geometric structures in $\Theta(n^{1/2})$ time. Since it takes $\Omega(n^{1/2})$ time for data to travel across a mesh computer of size n, all of the algorithms, except the one associated with Theorem 5.6, have optimal worst case times and are significantly faster than the $\Omega(n)$ time required for a serial computer to process $O(n)$ pieces of data. (In fact, many of these problems require $\Omega(n \log n)$ time on a serial computer.) Other than a preliminary announcement of some of these results by the authors [27], the algorithms in this paper represent the first solutions to these problems for the two-dimensional mesh. The one exception is the minimal distance spanning tree problem for point data, which has been previously discussed in [24]. However, the solution that we present in Theorem 5.6 is more space and time efficient than that appearing in [24], and we have described how to solve the minimum spanning tree problem in optimal time and space on a mesh computer.

The algorithms presented in this paper employ different approaches to solving geometric problems than those that have been explored for these problems on a serial computer. A variety of techniques have been introduced for the mesh

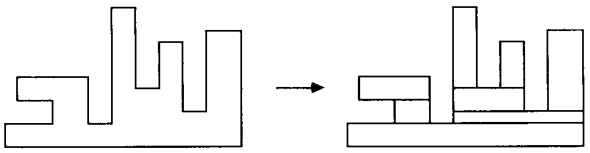


Fig. 15. Decomposing an orthogonal polygon into iso-oriented rectangles.

computer, including *multidimensional divide-and-conquer*, *parallel binary search*, and a variety of interesting *grouping techniques* that seem amenable to most situations where multiple parallel searches are needed. These approaches appear to be well suited to other architecturally related models (e.g., easily providing poly-logarithmic solutions to these problems on a hypercube), and we believe that our algorithms will produce similarly good results for these models. Furthermore, the various grouping techniques can be extended to arbitrary parallel computers through the use of sorting and prefix operations, providing a systematic efficient solution to parallel search problems.

There may be situations in which our algorithms can be slightly modified to produce even faster solutions to these problems on a mesh computer. For instance, when multiple figures exist, each stored in a subsquare of size no more than D, then solutions to many of the problems addressed in this paper may be extended so that the necessary result can be determined simultaneously for all of the figures in $\Theta(D^{1/2})$ time.

In this paper, the concentration has been on two-dimensional meshes since they are the ones most commonly built. A j-dimensional mesh of size n (where n is the jth power of some integer) has n PE's arranged in a j-dimensional cubic lattice. PE $P_{s_1,\cdots,s_j}$ and PE $P_{t_1,\cdots,t_j}$ are connected if and only if $\Sigma_{i=1}^{j}|s_i - t_i| = 1$. In the O-notational analyses of algorithms for j-dimensional meshes it makes sense to consider j as fixed. That is, there is no differentiation between a step needing a constant amount of time and one needing 2^j units. The reason for this is that a PE in a j-dimensional mesh is fundamentally different from one in a k-dimensional mesh when $j \neq k$. A proximity ordering can be defined for a j-dimensional mesh, and all of the data movement operations described in Section II-D can be extended to run in $\Theta(n^{1/j})$ time, which again is optimal. Therefore, all of the optimal two-dimensional mesh algorithms written solely in terms of these data movement operations yield optimal $\Theta(n^{1/j})$ time j-dimensional mesh algorithms. (The algorithm in Theorem 5.6 yields a j-dimensional mesh algorithm requiring $\Theta(n^{1/j} \log n)$ time.) For a few algorithms, values of constants were chosen to make the recurrence yield the desired result. For j-dimensional mesh algorithms, these constants need to be chosen as a function of j. For example, in Theorem 5.2, for two-dimensional meshes five slabs were used in each direction, while for j-dimensional meshes one needs at least $1 + 2^j$.

Bentley [11] has described an algorithm paradigm, called *multidimensional divide-and-conquer*, that has applications to many problems including those in computational geometry. In Section VI, possible pitfalls that exist when one tries to use multidimensional divide-and-conquer naively on a parallel machine were discussed. Furthermore, the algorithms that were presented have avoided these pitfalls when dealing with polygonal figures. For problems involving points, multidimensional divide-and-conquer can be applied more simply on mesh computers than it can be for problems involving polygonal figures. A point p is said to *dominate* a point q if and only if the x and y coordinates of p are greater than the respective x and y coordinates of q. (This definition can be naturally extended to higher dimensional data.) By applying a straightforward multidimensional divide-and-conquer technique, for any fixed j, for n or fewer points on a j-dimensional mesh of size n, dominance problems can be solved in optimal $\Theta(n^{1/j})$ time. These problems include determining for every point how many other points it dominates, and finding for every point whether or not it is a maxima (i.e., not dominated by any point). Serial algorithms for these problems appear in [11] and optimal two-dimensional mesh algorithms appear in [14].

Other problems that are solved by using multidimensional divide-and-conquer to reduce the problem to the same problem in lower dimensions, such as deciding which iso-oriented boxes are intersected by others, can similarly be solved in the same time. Some algorithms naturally yield optimal algorithms for higher dimensional data even though they do not use multidimensional divide-and-conquer. For example, for any fixed dimension j, the all-nearest neighbor problem for points can be solved in $\Theta(n^{1/j})$ time on a j-dimensional mesh by using a straightforward extension of the algorithm in Theorem 5.2. For a few problems, such as finding the convex hull, it should be possible to extend to three-dimensional data in the same time bounds. However, many of the remaining problems seem to either require too much data movement, or the generation of too much data, when the dimension of the input increases. For example, it is known that the convex hull of n points in d-dimensional space may have $\Theta(n^{\lfloor d/2 \rfloor})$ faces, so for $d \geq 4$ any algorithm which generates and keeps all the faces will need $\Omega(n^{d/2})$ PE's to store them, or else the memory available in each PE must be increased.

[29] have shown that the mesh computer is an asymptotically optimal interconnection for a wide variety of problems that consider problems of size n on a mesh with s processing elements, $s \leq n$, where each PE has $\Omega(n/s)$ memory. We are currently studying problems in computational geometry, such as those described in this paper, to see how they would relate to mesh computers of various sizes.

Acknowledgment

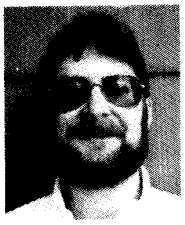
The authors would like to express their appreciation to the following people for their comments on this paper: M. Atallah, L. Boxer, R. Cypher, M. Lu, S. Miller, J. Sanz, I. Stojmenovic, B. Warren, and the anonymous referee. They would also like to thank P. Newport for drawing some of the diagrams.

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap, "Parallel computational geometry," in *Proc. 1985 Symp. Foundations Comput. Sci.*, pp. 468-477.

[2] S. Akl, "Parallel algorithms for convex hulls," Dep. Comput. Sci., Queens Univ., Kingston Ont., Canada, 1983.

[3] M. J. Atallah and M. T. Goodrich, "Efficient parallel solutions to some geometric problems," *J. Parallel Distribut. Comput.*, vol. 3, pp. 492-507, 1986.
[4] M. J. Atallah and S. R. Kosaraju, "Graph problems on a mesh-connected processor array," *J. ACM*, vol. 31, pp. 649-667, 1984.
[5] M. J. Atallah and S. E. Hambrusch, "Solving tree problems on a mesh-connected processor array," in *Proc. 26th Symp. Foundations Comput. Sci.*, 1985, pp. 222-231.
[6] D. Avis, "On the complexity of finding the convex hull of a set of points," Tech. Rep. FOCS 79.2, School of Comput. Sci., McGill Univ., 1979.
[7] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The ILLIAC IV Computer," *IEEE Trans. Comput.*, vol. C-17, pp. 746-757, 1968.
[8] K. E. Batcher, "Design of a massively parallel processor," *IEEE Trans. Comput.*, vol. 29, pp. 836-840, 1981.
[9] J. L. Bentley, B. W. Weide, and A. C. Yao, "Optimal expected-time algorithms for closest point problems," in *Proc. Allerton Conf.*, 1978.
[10] J. L. Bentley and T. A. Ottman, "Algorithms for counting and reporting geometric intersections," *IEEE Trans. Comput.*, vol. 28, pp. 643-647, 1979.
[11] J. L. Bentley, "Multidimensional divide-and-conquer," *Commun. ACM*, vol. 23, pp. 214-229, 1980.
[12] B. Chazelle, "Computational geometry on a systolic chip," *IEEE Trans. Comput.*, vol. C-33, pp. 774-785, 1984.
[13] A. Chow, "A parallel algorithm for determining convex hulls of sets of points in two dimensions," in *Proc. 19th Allerton Conf. Commun., Contr., Comput.*, 1981, pp. 214-233.
[14] F. Dehne, "$O(n^{1/2})$ algorithms for the maximal elements and ECDF searching problem on a mesh-connected parallel computer," *Inform. Proc. Lett.*, vol. 22, pp. 303-306, 1986.
[15] M. L. B. Duff and D. M. Watson, "The cellular logic array image processor," *Comput. J.*, vol. 20, pp. 68-72, 1977.
[16] C. R. Dyer and A. Rosenfeld, "Parallel image processing by memory augmented cellular automata," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. PAMI-3, pp. 29-41, 1981.
[17] H. Freeman and R. Shapira, "Determining the minimal-area encasing rectangle for an arbitrary closed curve," *Commun. ACM*, vol. 18, pp. 409-413, 1975.
[18] S. I. Gass, *Linear Programming*. New York: McGraw-Hill, 1969.
[19] F. C. A. Groen, P. W. Verbeek, N. d. Jong, and J. W. Klumper, "The smallest box around a package," Tech. Rep. Instit. Appl. Phy., Delft Univ. of Technology.
[20] K. Hwang and K-s. Fu, "Integrated computer architectures for image processing and database management," *IEEE Computer*, vol. 15, pp. 51-60, 1982.
[21] C. S. Jeong and D. T. Lee, "Parallel geometric algorithms on a mesh connected computer," Tech. Rep. 87-02-FC-01 (revised), Dep. EECS, Northwestern Univ.
[22] R. J. Lipton and R. E. Tarjan, "Applications of a planar separator theorem," in *Proc. 18th Annu. IEEE Symp. Foundations Comput. Sci.*, 1977, pp. 162-170.
[23] M. Lu, "Constructing the Voronoi diagram on a mesh-connected computer," in *Proc. 1985 Int. Conf. Parallel Processing*, pp. 806-811.
[24] M. Lu and P. Varman, "Solving geometric proximity problems on mesh-connected computers," in *Proc. 1985 Workshop Comput. Architecture Pattern Anal. Image Database Management*, pp. 248-255.
[25] E. M. McCreight, "Priority search trees," Tech. Rep. Xerox PARCL CSL-81-5, 1981.
[26] C. A. Mead and L. Conway, *Introduction to VLSI Systems*. Reading, MA: Addision-Wesley, 1979.
[27] R. Miller and Q. F. Stout, "Computational geometry on a mesh-connected computer," in *Proc. 1984 Int. Conf Parallel Processing*, pp. 66-73.
[28] ——, "Geometric algorithms for digitized pictures on a mesh-connected computer," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. PAMI-7, pp. 216-228, 1985.
[29] ——, "Varying diameter and problem size in mesh-connected computers," in *Proc. 1985 Int. Conf. Parallel Processing*, pp. 697-699.
[30] ——, *Parallel Algorithms for Regular Architectures*. Cambridge, MA: MIT Press, to be published.
[31] D. Nassimi and S. Sahni, "Data broadcasting in SIMD computers," *IEEE Trans. Comput.*, vol. C-30, pp. 101-107, 1981.
[32] D. Nath, S. N. Maheshwari, and P. C. P. Bhatt, "Parallel algorithms for the convex hull in two dimensions," in *Proc. Conf. Anal. Problem Classes Programming Parallel Comput.*, 1981, pp. 358-372.
[33] J. O'Rourke, C.-B. Chen, T. Olson, and D. Naddor, "A new linear algorithm for intersecting convex polygons," *Comput. Graph. Image Processing*, vol. 19, pp. 384-391, 1982.
[34] F. P. Preparata and D. T. Lee, "Computational geometry—A survey," *IEEE Trans. Comput.*, C-33, pp. 1072-1100, 1984.
[35] F. P. Preparata and M. I. Shamos, *Computational Geometry*. New York: Springer-Verlag, 1985.
[36] J. Reif and Q. F. Stout, "Optimal component labeling algorithms for mesh computers and VLSI," to be published.
[37] M. I. Shamos, "Computational geometry," Ph.D. dissertation, Yale Univ., 1978.
[38] M. I. Shamos and D. Hoey, "Geometric intersection problems," in *Proc. Seventh Annu. IEEE Symp. Foundations Comput. Sci.*, 1975, pp. 151-162.
[39] Q. F. Stout, "Broadcasting in mesh-connected computers," in *Proc. 1982 Conf. Inform. Sci. Sys.*, Princeton Univ., pp. 85-90.
[40] ——, "Topological matching," in *Proc. 15th ACM Symp. Theory of Computing*, 1983, pp. 24-31.
[41] ——, Tree-based graph algorithms for some parallel computers," in *Proc. 1985 Int. Conf. Parallel Processing*, pp. 727-730.
[42] C. D. Thompson and H. T. Kung, "Sorting on a mesh-connected parallel computer," *Commun. ACM*, vol. 20, pp. 263-271, 1977.
[43] G. T. Toussaint, "Pattern recognition and geometrical complexity," in *Proc. 5th Int. Conf. Pattern Recognition*, 1980, pp. 1324-1347.
[44] A. Yao, "A lower bound to finding convex hulls," Dep. Comput. Sci., Stanford Univ., 1979.



Russ Miller (S'82-M'85) was born in Flushing, NY, on January 8, 1958. He received the B.S., M.A., and Ph.D. degrees in computer science/mathematics from the Department of Mathematical Sciences, State University of New York at Binghamton.

Since 1985 he has been an Assistant Professor in the Department of Computer Science at the State University of New York at Buffalo. Since 1988 he has also been Associate Director for the graduate group in Advanced Scientific Computing at the University of Buffalo. His primary research interests are parallel algorithms, parallel computing, and parallel architectures. He recently coauthored (with Q. F. Stout) the book *Parallel Algorithms for Regular Architectures* (Cambridge, MA: MIT Press, 1989).

Dr. Miller is a member of the Computer Society, the Association for Computing Machinery, SPIE, and Phi Beta Kappa.

Quentin F. Stout (M'82) received the B.A. degree from Centre College, Danville, KY, and the Ph.D. degree from Indiana University.

Since 1984 he has been an Associate Professor in the Department of Electrical Engineering and Computer Science of the University of Michigan, Ann Arbor. From 1976 to 1984 he was in the faculty of the Mathematical Sciences Department of the State University of New York at Binghamton. His primary research interests are in parallel algorithms, parallel computing and parallel architectures. He recently coauthored (with R. Miller) the book *Parallel Algorithms for Regular Architectures* (Cambridge, MA: MIT Press, 1989).

Dr. Stout is a member of the Association for Computing Machinery, the American Mathematical Society, and the Mathematical Association of America, and serves on the editorial board of the *Journal of Parallel and Distributed Computing*.