Algorithmica (1988) 3: 293–327

Algorithmica

Parallel Computational Geometry[1]

A. Aggarwal,[2] B. Chazelle,[3] L. Guibas,[4] C. Ó'Dúnlaing,[5,6] and C. Yap[5]

Abstract. We present efficient parallel algorithms for several basic problems in computational geometry: convex hulls, Voronoi diagrams, detecting line segment intersections, triangulating simple polygons, minimizing a circumscribing triangle, and recursive data-structures for three-dimensional queries.

Key Words. Parallel algorithms, Computational geometry, Data structures.

1. Introduction. Computational geometry addresses algorithmic problems in diverse areas such as VLSI design, robotics, and computer graphics. Since 1975 there has been a wide development of sequential algorithms for geometric problems, but until 1985 there was little published about developing parallel algorithms for such problems. A notable exception was the Ph.D. research of Chow (1980) which seems to have been the pioneering work in the field but unfortunately only a portion of it has appeared in the open literature.

This paper contributes some parallel algorithms for solving geometric problems:

(1) Convex hulls in two and three dimensions.
(2) Voronoi diagrams and proximity problems.
(3) Detecting segment intersections and triangulating a polygon.
(4) Polygon optimization problems.
(5) Creating data structures in two and three dimensions to answer some standard queries.

It is seldom obvious how to generate parallel algorithms in this area since popular techniques such as contour tracing, plane sweeping, or gift wrapping involve an explicitly sequential (iterative) approach (see Preparata and Shamos (1985) for more details). In this paper we exploit data-structures that are simple enough to compute efficiently in parallel while being powerful enough to answer

[1] The work of C. Ó'Dúnlaing and C. Yap was supported by NSF Grants DCR-84-01898 and DCR-84-01633.
[2] IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, USA.
[3] Department of Computer Science, Princeton University, Princeton, NJ 08544, USA.
[4] Digital Equipment Corporation Systems Research Laboratories, 130 Lytton Avenue, Palo Alto, CA 94301, USA, and Computer Science Department, Stanford University, Stanford, CA 94305, USA.
[5] Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012, USA.
[6] School of Mathematics, Trinity College, Dublin 2, Republic of Ireland.

Received January 8, 1987; revised November 30, 1987. Communicated by Jeffrey Scott Vitter.

queries efficiently. However, the queries may be answered slightly less efficiently than would be possible if the data-structures were constructed sequentially. "Efficient" here means polylogarithmic in parallel time.

All the algorithms presented here will be *NC*-algorithms, i.e., algorithms executable in polylog depth on polynomial-size circuits. We caution that the circuits here are slightly different from those used in machine-based complexity theory in that each node of our circuit can compute an infinite precision arithmetic operation. Furthermore, these algorithms shall all be described as if they were implemented on a *CREW PRAM*—a concurrent-read, exclusive-write parallel random access machine. Such a machine is conceived as having a large number of processors with common access to a common memory. Any number of processors can read the same memory cell simultaneously in $O(1)$ steps, and any processor can write to a memory cell in $O(1)$ steps, but if two processors attempt to write to the same cell simultaneously then the machine enters an undefined state.

We should mention here that Anita Chow's dissertation pays close attention to the model of computation involved, and in several cases she provides algorithms for implementation on two models of parallel computation: the CREW PRAM model and the cube-connected cycles (CCC) network. As mentioned before, in the CREW model, all the processors share the same memory; however, in the CCC network, each processor has its own memory, and is connected to at most four other processors. For details on the CCC model, see Chow (1980).

It follows from Kozen and Yap (1985) that most common computational geometry problems including all the problems considered in this paper have, in principle, *NC*-algorithms. Recall that they combined the techniques of Ben-Or *et al.* (1984) and Collins (1975) to give parallel algorithms for computing cell decompositions with adjacency information; for fixed dimensions, these algorithms are in *NC*. The consequences of the cell decomposition algorithm for parallel computational geometry are explored more carefully in Yap (1987). Since any appeal to Kozen and Yap (1985) involves a reduction to Tarski's language for real closed fields, and the methods are too general to be of much practical use, this furnishes an existence proof for *NC* algorithms rather than furnishing practical *NC* algorithms.

Many of the problems considered here are known to have $\Omega(n \log(n))$ lower bounds in the algebraic computation tree model [Ben-Or (1983)]. Therefore the goal in our research in such cases is to aim for $\Omega(\log(n))$ parallel time when $O(n)$ processors are available. Only in some instances do we present such optimal algorithms, and the algorithms may require a preprocessing step which involves sorting. While $O(\log^2(n))$ sorting networks have been known for many years, to ensure optimality we may need to invoke an optimal parallel sorting method ($\log(n)$ parallel time). The AKS sorting network achieves these bounds but with utterly unacceptable overhead (at the present state of knowledge: see Ajtai *et al.* (1983) and Leighton (1984)). More recently an optimal PRAM algorithm has been described which has much lower overhead [Cole (1986)]. In our algorithms to compute the Voronoi diagram for a planar point set and convex hull in three dimensions, we are able to avoid sorting even though, for the latter problem, we only match the complexity bounds of Anita Chow.

We also pay some attention to th
algorithm cannot be considered co
description of how the various tasks
of processors. Such considerations u
taken to complete a parallel step i
processors involved, since processor
logarithmic parallel time.

Two standard methods of paralle
throughout this paper: computing
ranking. To compute a running sum
array of n numbers. This is easily
processors, by imagining a balanced
at each internal node the sum of t
computed (in an upward parallel sw
sums can be computed in a seco
linked list and determining the ra
This can be accomplished in log
technique; it is discussed briefly a

2. Definitions and Terms. As sta
here all belong to the class *NC*, bu
on a CREW machine. The notati

indicates the class of algorithms
halting in $O(\log^k(n))$ steps. We
those using a linear number of pr
to distinguish our model from t
See Yap (1987).

We let E^2 (resp. E^3) denote th
this paper we restrict our attent
set S of points in E^2 or E^3, t
containing all points in S. Some
hull with the polygon (resp. pol
the notation $\partial H(S)$ to denote t

Given a finite nonempty set
defined as follows. For each p
of points z in E^2 such that z
that the Voronoi cells are a
boundaries and two cells can
the interior of every cell is di
Vor(S) is defined as the planar
of all the Voronoi cells; it is
edges are straight line-segme
at most one edge in common

We also pay some attention to the processor allocation problem: a PRAM algorithm cannot be considered completely described until we have a clear description of how the various tasks are to be allocated among the available pool of processors. Such considerations usually become unimportant when the time taken to complete a parallel step is logarithmic (or worse) in the number of processors involved, since processor allocation is usually easy to accomplish in logarithmic parallel time.

Two standard methods of parallel algorithm design will be exploited freely throughout this paper: computing a running sum ("parallel prefix") and list-ranking. To compute a running sum means to compute the n initial sums of an array of n numbers. This is easily computed in logarithmic parallel time with n processors, by imagining a balanced binary tree whose leaves are the array entries: at each internal node the sum of the entries of all its leaf descendants can be computed (in an upward parallel sweep), and from these internal sums all initial sums can be computed in a second phase. List-ranking involves traversing a linked list and determining the rank (distance from the end) of all its entries. This can be accomplished in logarithmic parallel time by a pointer-jumping technique; it is discussed briefly at the end of Section 3.

2. Definitions and Terms. As stated in the Introduction, the algorithms treated here all belong to the class NC, but we express them as PRAM algorithms running on a CREW machine. The notation

$$NC_k^+(f(n))$$

indicates the class of algorithms running on a PRAM using $f(n)$ processors and halting in $O(\log^k(n))$ steps. We are mainly interested in the class $NC_k^+(n)$, i.e., those using a linear number of processors. Note: we use "NC^+" instead of "NC" to distinguish our model from the circuit model of machine-based complexity. See Yap (1987).

We let E^2 (resp. E^3) denote the Euclidean plane (resp. Euclidean 3-space). In this paper we restrict our attention to E^2 or E^3. Given a (finite and nonempty) set S of points in E^2 or E^3, the *convex hull* $H(S)$ is the smallest convex set containing all points in S. Sometimes by abuse of notation we confuse the convex hull with the polygon (resp. polyhedron) which constitutes its boundary. We use the notation $\partial H(S)$ to denote the polygonal boundary of $H(S)$.

Given a finite nonempty set S of points in E^2, its *Voronoi diagram* $\mathrm{Vor}(S)$ is defined as follows. For each point p in S, define the *Voronoi cell* $V(p)$ as the set of points z in E^2 such that z is as close to p as to any other point in S. Recall that the Voronoi cells are all nonempty closed convex sets with polygonal boundaries and two cells can meet only along their common boundaries (i.e., the interior of every cell is disjoint from all other cells). The *Voronoi diagram* $\mathrm{Vor}(S)$ is defined as the planar point-set formed from the union of the boundaries of all the Voronoi cells; it is planar graph with $O(n)$ edges and vertices, all the edges are straight line-segments (perhaps unbounded), and two cells can have at most one edge in common.

3. Optimal Parallel Convex Hull in the Plane. Fast parallel algorithms for the planar convex-hull problem have been considered in the recent literature [Nath *et al.* (1981), Chow (1981), Akl (1983), Chazelle (1984)]. The algorithms typically use a simple divide-and-conquer technique, recursively computing the convex hull of n points by solving two problems of half the size. It seems that any such technique requires $\Omega(\log^2(n))$ time. The main result in this section is the following:

THEOREM 1. *The problem of computing the convex hull of a set of points in the plane and listing its corners in cyclic order is in* $NC_1^+(n)$.

We subsequently learned that Atallah and Goodrich (1985), and Wagener (1985, 1987), have independently used a similar technique to derive the same result.

From the well-known fact that sorting can be reduced to the planar convex-hull problem, or using the lower bounds for the algebraic computation tree model in Ben-Or (1983), it follows that the $\log(n)$ time is optimal for n processors. Actually the algorithm is optimal in a stronger sense: it follows from the work of Cook and Dwork (1982) that $O(\log(n))$ time is the best possible *even* if we allow arbitrarily many processors. To identify the points on the convex hull can be done in $O(1)$ CRCW time [Akl (1983)].

Let S be the given set of n points. Use the pair of points on the convex hull $H(S)$ with maximum and minimum x-coordinate to partition $H(S)$ into two parts in the natural way: the *upper* and the *lower chains*. By symmetry, it is sufficient to show how to compute the upper chain in $O(\log(n))$ time. First sort S according to the x-coordinate of the points, using $O(\log(n))$ time with n processors [Ajtai *et al.* (1983), Leighton (1984), Cole (1986)]. At this stage, covertical sets of points can be detected and all but the highest from each set discarded in $O(\log(n))$ steps. Our algorithm divides S into $\sqrt{n}$ sets of $\sqrt{n}$ points, recursively computes the upper convex chain of each set, and then merges all these chains together in $O(\log(n))$ time. The details of merging are nontrivial.

Let us see how to merge $\sqrt{n}$ upper chains in $O(\log(n))$ steps with n processors. First consider two upper chains C and C'. Since we divide S by the x-coordinate, this ensures that the x-coordinates of any two chains do not overlap. Granted that their edges are presented in an array in sorted order, a single processor can compute in $O(\log(n))$ time the unique line which is tangent to both chains, together with the two points of tangency. This was shown by Overmars and Van Leeuwen (1981). Since there are at most $\binom{\sqrt{n}}{2}$ common tangents to be computed, n processors can compute all such tangents in time $O(\log(n))$. Note that processor allocation raises no difficulty here since all the chains have a natural sequence (according to their x-coordinates) and there is a natural sequence therefore on the set of all pairs of chains.

Let G denote the (combinatorial) graph whose vertices are the points in S and whose edges are the edges in the upper chains, together with the line-segments defined by the tangent lines just discussed. There are at most $n-1$ edges in all the upper chains, and they can be compressed into contiguous locations in an array A by using a running-sum technique (in $O(\log(n))$ parallel time). The extra

$\binom{\sqrt{n}}{2} < n$ edges may be appended to these edges in the upper chains. The array A contains two directed edges (x_1, x_2) and (x_2, x_1) for each undirected edge $\{x_1, x_2\}$. Using an optimal parallel sorting method, sort these edges according to

$$\text{(slope of edge } (x_1, x_2)\text{) within (earlier } x\text{-coordinate } (x_1)\text{).}$$

The slope of an edge is the angle it makes with the x-axis, measured from $-\pi/2$ to $3\pi/2$ (with $\pi/2$ corresponding to the vertical upward direction). Thus the array is partitioned into adjacency lists for the vertices of G.

From G we can identify vertices of the upper chain of S using the following observation: consider any point v in S, which is neither leftmost nor rightmost in S. If v belongs to the upper chain of S, let u and w be the vertices immediately before and after v along the chain. Clearly, the slopes of the edges (v, u) and (v, w) are adjacent among all the edges out of v, and the angle uvw (measured clockwise from u to w) is $\geq \pi$. Clearly, for each vertex v there can be at most one pair of edges (v, u) and (v, w) with this property, and such vertices v can be detected in constant time with one processor per edge in the array A. This criterion identifies all edges and vertices on the upper chain, and perhaps some others, except for the first and last edges on the upper chain: these are, of course, the edges with maximum and minimum slopes, respectively, leaving the leftmost and rightmost vertices. (Since S is presorted by x-coordinate it is easy to identify these two vertices.)

For all these triples u, v, w which indicate possible vertices of the upper chain, let us call v a "candidate" and w its "successor." If v is not a candidate its successor is undefined, and the leftmost vertex has as its successor the next vertex on the upper chain. As indicated above this is easily identified. This successor relation defines a forest of trees, in which the upper chain is represented by a branch connecting the leftmost vertex to the rightmost. This branch can be identified in logarithmic parallel time, one processor per vertex, using the parallel ranking technique described below.

To complete our inductive step, we must now collect the vertices of the upper chain of S into consecutive locations in an array. This is easily accomplished using parallel prefix, i.e., computing a running sum on an array $C(v)$ whose entry is 1 if v is in the upper chain and 0 otherwise. This completes the merge process of our algorithm.

Since the ranking problem is so basic and will be used again, it bears repeating here. Here we consider the successor relation as defining a forest of trees directed from child to parent, and the rank of a node in this forest is its distance from its root ancestor. In view of a later application, in addition to the rank information, we compute an additional $O(\log(n))$ pointers for each vertex:

(*) In $\log(n)$ parallel time, each node can learn its rank and, also for each $i \leq \log(n)$, it also holds a pointer to its 2^ith successor, if it has such a successor.

This is done in two phases, each phase taking $\log(n)$ parallel time. In the first phase, by the well-known doubling technique, every vertex determines for all $i \leq \log(n)$ whether it has a 2^ith successor, and, if so, which. In the second phase, at stage i (there are $\log(n)$ stages) all vertices with rank between 2^i and $2^{i+1}-1$ learn their exact rank. It is easy to implement all this on a CREW PRAM in logarithmic time.

The runtime $T(n)$ of this convex-hull algorithm, for n points with n processors, satisfies the relation

$$T(n) \leq c \log(n) + T(\sqrt{n})$$

which implies $T(n)$ is $O(\log(n))$.

4. Voronoi Diagram and Proximity Problems. One of the fundamental data structures of computational geometry is the Voronoi diagram [Shamos (1977)]. The first optimal sequential algorithm to compute the Voronoi diagram was given by Shamos and Hoey (1975) and it takes $O(n \log n)$ time. In this section we show

THEOREM 2. *There is an $NC_2^+(n)$ algorithm for the planar Voronoi diagram of a set of points. Furthermore, the algorithm only assumes an $NC_2^+(n)$ sorting algorithm.*

Shamos (1977) also pointed out that the Voronoi diagram can be used as a powerful tool to give efficient algorithms for a wide variety of other geometric proximity problems. From the Voronoi diagram we can easily obtain the closest pair of sites (points), or the closest neighbor of each site, or the Euclidean minimum spanning tree of the n sites, or the largest point-free circle with center inside their convex hull, etc. Efficient reductions for these problems are possible in our parallel computation model, so our polylog result for the Voronoi also implies polylog algorithms for all these problems, using a linear number of processors. Note that there is a well-known reduction of the Voronoi diagram of a set of points in the plane to the convex hull of a set of points in 3-space [Brown (1979), Guibas and Stolfi (1983)]. Also, in her thesis, Chow (1980) shows that the three-dimensional convex hull can be computed in $NC_3^+(n)$, and exploits this to give an $O(\log^3(n))$ algorithm to compute the diagram. However, her methods frequently involve optimal-time parallel sorting,[7] which the following direct method avoids. In our FOCS extended abstract we gave an $NC_3^+(n)$ algorithm which avoids optimal-time parallel sort. In the present paper we improve this to $NC_2^+(n)$ by exploiting a simple observation suggested by Goodrich.

[7] The runtime bound given in Chow (1980) differs from the bound quoted here by an additional factor of $\log\log(n)$. These discrepancies arise wherever sorting is involved. The $O(\log(n)\log\log(n))$ parallel-sorting algorithm in Valiant (1975) has since been superseded by theoretically optimal algorithms.

Fig. 1.

The present algorithm for c
the sequential method first pr
divide-and-conquer method in
sets P and Q of points (where
vertical line L which has all of
in linear time to a diagram fo
path (the "contour" or "seam
tracing a line and seems inher
into the set of points closer to
point or points in S are all i
illustrates the contour betwee
Our goal is to parallelize thi
terminology. The convex hull
are defined as the half-lines exte
to the sides incident to those
These normals partition the co
are alternately *slices* (V-shape
two incident normals). Figure
tion of $\mathrm{Vor}(S)$ with the conve
Let U be the strip defined by
we call the *e-promontory*. We a
collinear, no four concyclic).
this implies that the promont
leaves at the cutpoints and su
from e. Beyond the root of t
the strip.

Point location inside strips
modify a suggestion of Good

LEMMA 3. *Suppose a strip h*
is a k processor $O(\log(k))$-tim

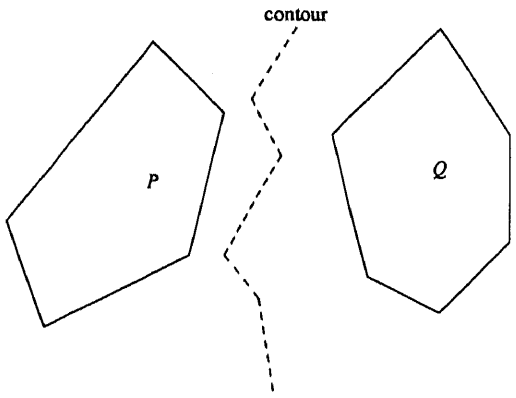

Fig. 1. Merging by contour tracing.

The present algorithm for computing the Voronoi diagram is modeled after the sequential method first presented by Shamos and Hoey (1975). This is a divide-and-conquer method in which two Voronoi diagrams for the respective sets P and Q of points (where these sets are *vertically separated*, i.e., there is a vertical line L which has all of P to its left and all of Q to its right) are merged in linear time to a diagram for $P \cup Q$. The merging is defined by a polygonal path (the "contour" or "seam") separating P from Q; the process is akin to tracing a line and seems inherently sequential. The contour separates the plane into the set of points closer to P than to Q (i.e., the set of points x whose closest point or points in S are all in P) and those closer to Q than to P. Figure 1 illustrates the contour between two hypothetical sets P and Q.

Our goal is to parallelize this contour tracing. Let us introduce some suitable terminology. The convex hull $H(S)$ has already been defined. The *normals* of S are defined as the half-lines extending from the corners of $H(S)$ and perpendicular to the sides incident to those corners: two normals extend from each corner. These normals partition the complement of $H(S)$, $E^2 \sim H(S)$, into *sectors*, which are alternately *slices* (V-shaped) and *strips* (bounded by a side of $H(S)$ and the two incident normals). Figure 2 clarifies this terminology. The points of intersection of $\mathrm{Vor}(S)$ with the convex polygon $\partial H(S)$ we call the *cutpoints* (Figure 3). Let U be the strip defined by an edge e of the convex hull. The set $U \cap \mathrm{Vor}(S)$ we call the *e-promontory*. We assume that S is in general position (no three points collinear, no four concyclic). Together with the convexity of the Voronoi cells, this implies that the promontory has the natural structure of a binary tree with leaves at the cutpoints and such that the parent of a node is always more distant from e. Beyond the root of this binary tree extends an infinite ray that bisects the strip.

Point location inside strips is easily done in $O(\log^2(n))$ serial time. We now modify a suggestion of Goodrich to show that $O(\log(n))$ serial time suffices.

LEMMA 3. *Suppose a strip has an e-promontory with k vertices and edges. There is a k processor* $O(\log(k))$*-time parallel algorithm for constructing a data-structure*

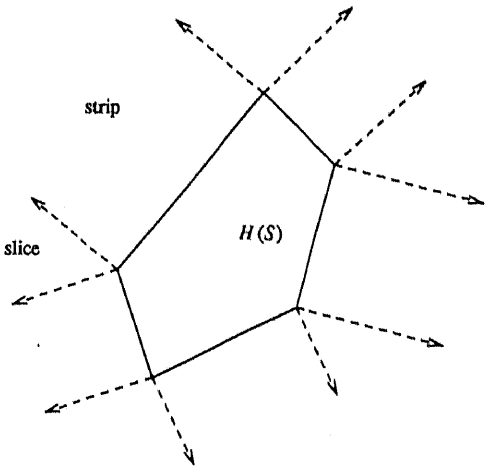

Fig. 2. Normals and sectors (slices and strips).

for the plane partition defined by the e-promontory so that in $O(\log(k))$ time a single processor can answer a point location query.

PROOF. The e-promontory is a binary tree laid out in the plane. It is complete, i.e., each internal node has two children. We assume that each leaf is on the x-axis, and for every node v all its descendants have a smaller y-coordinate. For any node v of the promontory, let T_v denote the subtree of the e-promontory rooted at v. Each internal node v of the e-promontory tree may be associated with a triangular region R_v with v as one corner of the triangle and the two other corners the extreme cutpoints below the subtree at v. By convexity of Voronoi cells, the entire subtree T_v is contained in R_v. Suppose p is a query point inside the strip. Let r be the root of the e-promontory. The interesting case is when p lies inside R_r, and this can be checked in constant time. We can now do a binary search inside R_r by locating a vertex v such that T_v contains between one-third

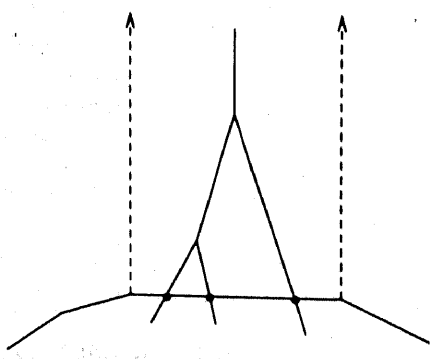

Fig. 3. Promontory: cutpoints are indicated by (•).

and two-thirds of the vertices of T_r. If p is inside R_v, we can recursively do point location for p inside this subtree. Otherwise, we next search inside a modified version of T_r: the modified tree T'_r consists of replacing T_v by three vertices corresponding to the corners of the triangle R_v, together with the two edges from v to the other two corners. This recursive search can be carried out in $O(\log(k))$ serial time.

It remains to construct a search structure D_r for the tree T_r. The search structure D_r consists of a binary tree whose internal nodes are associated with triangular regions of the form R_v which we use for comparisons: if we are at some internal node z of D_r, we next go to the right (resp. left) child of z if query point p lies inside (resp. outside) the triangular region of z. To construct D_r associated with T_r we do an important preprocessing step where for each node v we determine its preorder rank $\text{rank}(v)$ in the tree and, from this, we easily deduce $\text{maxrank}(v)$, the maximum rank of its descendants. This can be done using k processors and $O(\log(k))$ time [Tarjan and Vishkin (1985)].[8] We deduce two important pieces of information from these computed values. First, the number $\text{desc}(v)$ of descendants of a node v is given by $1+\text{maxrank}(v)-\text{rank}(v)$. Second, we can decide in constant time if any given vertex u is a descendant of another v. More precisely, u is a descendant of v iff

$$\text{rank}(v) \leq \text{rank}(u) \leq \text{maxrank}(v).$$

Inductively, suppose we have to construct a search structure D_r for the subtree T_r rooted at some node r. We assume that $\text{desc}(v)$ is precomputed for each v in T_r. In particular, let $k=\text{desc}(r)$ be the number of nodes in T_r. Assigning one processor to each vertex of T_r, in $O(1)$ steps, we can locate the (unique) deepest node of T_r with at least $2k/3$ descendants, and choose the node v_0 to be that one of its children with the larger number of descendants (breaking ties arbitrarily). It is easy to see that v_0 will have between $k/3$ and $2k/3$ descendants. The root of D_r will be associated with the region R_r. The right subtree of (the root of) D_r will be recursively constructed from T_{v_0}: note that the precomputed information $\text{desc}(v)$ for each node of T_{v_0} is available with no further work. Let T'_r be the modified tree obtained by pruning T_{v_0} as described above. We want to construct recursively the left subtree of D_r from T'_r. But first we must revise the precomputed information $\text{desc}(v)$ for each node for T'_r. To do this we only need to update the value of $\text{desc}(v)$ by subtracting $\text{desc}(v_0)-2$ from $\text{desc}(v)$ for all v that are ancestors of v_0. But this is easy since with one processor per vertex, each vertex can in constant time deduce if it is an ancestor of v_0 (use the preorder numbering of nodes as shown above). This process involves changing the structure of T_r, so it should be done with a copy of T_r. The search structure D_r will be a binary tree of logarithmic height, with a description of the triangle R_r at its root, plus children representing T'_r and T_{v_0}, respectively. □

[8] Their CRCW parallel RAM algorithm can be adapted for our CREW version.

The above data-structure helps us form the contour efficiently. Let us introduce some simple notation. Recall that a recursive step in the algorithm will involve separating a given set S by a vertical line L into two sets $P \cup Q$ (where P is to the left of L and Q to its right), recursively constructing $\mathrm{Vor}(P)$ and $\mathrm{Vor}(Q)$ separately, and merging the two along the contour. We speak of the edges of $\mathrm{Vor}(P)$ and $\mathrm{Vor}(Q)$ as PP edges and QQ edges. In the combined Voronoi diagram the contour edges can be called PQ edges in an obvious extension of this notation, and the vertices can be classified as PPP, PPQ, PQQ, and QQQ, depending on whether the associated three points of S (assuming general position so no four are cocyclic) involve three, two, one, or no points of P.

We need therefore to identify the PPQ and PQQ vertices. Instead of computing them directly, we solve the following simpler problem: to identify the PP and QQ edges that intersect the contour. We postpone till later determining precisely where they meet the contour. To carry this out, we need one more preliminary computation. This is necessitated by the fact that a Voronoi edge may intersect the contour twice.[9] However, we can break up each Voronoi edge into at most two "semiedges" such that the contour meets each semiedge at most once: let e be any edge of $\mathrm{Vor}(P)$ determined by the two points u, v in P. If the horizontal ray emanating leftward from u or v intersects e at some point x (this happens with at most one of the points u and v) then we split e into two *semiedges* at x. If neither ray intersects e then the entire edge e constitutes a semiedge. Similarly, if e were an edge of $\mathrm{Vor}(Q)$, we would split e into at most two semiedges using the horizontal rays emanating rightward from u or v.

LEMMA 4. *Each semiedge intersects the contour at most once.*

PROOF (SKETCH). Let e be an edge separating points u and v of P in $\mathrm{Vor}(P)$. Since any intersection of the contour with e results in a PPQ vertex, which will be an endpoint of an edge, call it e', separating u and v in the combined diagram, we conclude that the contour cannot intersect e more than twice. Suppose that it intersects e twice. Then e' together with the contour would enclose one of these points, u say, completely, and since the contour is monotonic in the y-direction it is easy to see that the horizontal line extending leftward from u intersects the edge e'. Thus each intersection point is on a different semiedge of e. □

Call any semiedge that intersects the contour *relevant*. Note that a semiedge in $\mathrm{Vor}(P)$ is relevant if and only if one endpoint is closer to P and the other endpoint is closer to Q. To decide if a Voronoi vertex (or other point bounding a semiedge) v of $\mathrm{Vor}(P)$ is closer to P than to Q, we have two cases. (i) If v is left of L then we determine the slice or strip of $H(Q)$ containing v. If v is in a slice, then v is closest to a unique point q in Q; we can easily decide if v is closer to q than to the closest points of P. If v is in a strip we reduce the problem

[9] We are indebted to Hubert Wagener (private communication) for pointing this out to us.

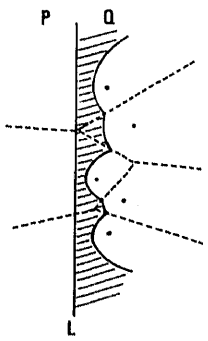

Fig. 4. The beachline (the beachhead B is shaded).

to point location inside a strip; once we have located v in the appropriate cell of Vor(Q) we can compare distances as in the case of the slice. (ii) If v is right of L it may initially appear that we need a general point location data-structure for that part of Vor(Q) lying to the right of L. We show how to avoid this using another idea.

We need some new terminology. Consider the set of points x to the right of L such that the circle centered at x and tangential to L passes through some point $q \in Q$ but where the interior of the circle contains no other points of Q. The set of such points will be called the *beachline* (of Q). (Alternatively, we could have defined the beach to be the "contour" separating L from Q.) For any point q not in L, let the *q-parabola* be the parabola focused at q with directrix L. (See Figure 4.)

LEMMA 5.

(a) *The beachline is a simple curve, disjoint from L and projecting monotonically onto L. Thus the beachline divides the half-plane to the right of L into two connected regions. The region that is adjacent to L is called the beachhead and denoted by B. The beachhead is characterized as the set of those points in the right half-plane of L that are closer to L than to any point in Q.*

(b) *The beachline is composed of a finite sequence of parabolic arcs, each such parabola being a portion of some q-parabola, $q \in Q$. Each such q is said to be near L. The point where two parabolic arcs meet is called a transition point.*

(c) *Let C be the contour defined by the partition $S = P \cup Q$. Then C lies entirely to the left of the beachline.*

(d) *The Voronoi cells of* Vor(Q) *that intersect B are precisely those belonging to points q that are near L in the sense of* (b). *Each connected component in the restriction of* Vor(Q) *to B has the structure of a free tree embedded in the plane whose leaves are on the boundary of B: at most one leaf is on L. The intersection of a cell of* Vor(Q) *with the beachhead may have several connected components. Each such component is called a B-cell. The B-cells are linearly ordered along the beachline.*

(e) *Let x be a transition point. Then there are two points q, $q' \in Q$ such that x lies on an edge of* Vor(Q) *separating the cells of q and q'. Moreover, the two parabolic arcs that meet at x are portions of the q- and q'-parabolas, respectively.*

PROOF. (a) To see that the beachline projects monotonically onto e, for each point p on L, draw the horizontal ray R emanating rightward from p. The beach line intersects R at a unique point x such that the points on R closer to L than to Q are precisely those lying between p and x.

(b) The points x of the beachline where the circle centered at x and tangent to L is incident to two or more points of Q are finitely many in number. These define the transition points. For nontransition points x on the beachline, the circle at x touches a unique $q \in Q$. Such an x lies on the parabola focused at q. Thus the beachline is composed of a finite number of parabolic arcs that are joined together at transition points.

(c) Let x be a point on the contour, and let D be the largest disk centered at x whose interior does not contain any point of $P \cup Q$. Then D touches points both in P and Q since x is on the contour; therefore its interior intersects L. Were x to the right of the beachline it would follow that D contained points of Q in its interior, a contradiction; hence x is on or to the left of the beachline.

(d) Clearly, a point in Q is near L if and only if its cell in Vor(Q) intersects B. Since every point in Q is obviously to the right of the beachline, every B-cell must intersect the beachline and therefore the B-cells are linearly ordered by their intersections with the beachline. For the same reason each connected component of $B \cap$ Vor(Q) is acyclic and has the structure of a (free) tree whose leaves are on the boundary of B. If such a component met the line L twice then there would be a chain of edges connecting these two points within B and therefore there would exist some B-cells not meeting the beachline: hence at most one leaf is on L.

(e) Choose q and q' to be the points that define the parabolic arcs that meet at x. □

LEMMA 6. *In $O(\log(n))$ time with $O(n)$ processors, we can construct the beachline of Q and a suitable data-structure to do the following query: for any point v, to determine in $O(\log(n))$ serial time, whether v lies in the beachhead B and, if so, which B-cell contains v.*

PROOF. Using one processor per edge of Vor(Q), we can compute all the transition points (at most two) on that edge. To see this, let e be an edge separating the Voronoi cells of q and q' in Vor(Q). Then each point x (at most two) where the q-parabola intersects e is a transition point. Once all the transition points are computed, we may sort them by their y-coordinates (using list-ranking) to form the beachline as well as determining all the B-cells.

Next consider the point location problem: by Lemma 5(a), we can use binary search to decide if a query point p lies in the beachhead in $O(\log(n))$ time. It is slightly more involved to locate the B-cell containing p in the beachhead.

Recall that Vor(Q) $\cap B$ is composed of a collection of free trees with leaves

on the beachline and perhaps one on L. We can assign a root to each such free tree: if it reaches L, let its root be the vertex connected to L; otherwise, let its root be the highest leaf vertex, say. All the edges can then be directed toward the root using a trick of Tarjan and Vishkin's (1985): form an Eulerian directed path beginning at the root and visiting each edge twice, in which case v is an ancestor of w if and only if v is first visited before w and last visited after w; thus the edges can be directed using list ranking. We can form regions R_T that enclose each individual tree T: if T reaches the line L then the region R_T has three corners formed from the root of the tree and the two extreme leaves of the tree. If T does not reach L then let u and v be its top and bottom intersections with the beachline. The line joining these two points cannot be crossed by any edge of T since otherwise T would either meet L or meet the beachline outside the range of these points; hence all of T is to the right of this line and we define R_T as that region of B to the right of this line in this case. (Unbounded regions raise a few special cases which are easily dealt with.) Each connected region of the beachhead exterior to all the regions R_T belongs to a unique B-cell. Thus using binary search we can determine the region R_T (if any) that contains the query point p. If p lies outside all the R_T then we are done. Otherwise, we use the same technique as for point location in an e-promontory. □

We are now ready to give the overall algorithm in five steps (A)-(F). Step (A) is done once and the rest are recursively called. The complexity $O(\log^2(n))$ follows since steps (B)-(F) each take $O(\log(n))$.

(A) First sort S according to its x-coordinate (an $O(\log^2(n))$ sorting method is enough here). Steps (B)-(F) involve partitioning S evenly into $P \cup Q$ (using a vertical line L) and recursively computing Vor(P), Vor(Q) and also the convex hulls $H(P)$, $H(Q)$.

(B) Construct the beachline for Q and the corresponding search structure for point location within the beachhead. Similarly, construct the data structure for point location within each strip emanating from the convex hull of Q. This takes $O(\log(n))$ time with n processors. Do the same for P.

(C) For each point v bounding a semiedge of Vor(P) determine whether it lies closer to Q than to P, using the two techniques shown above: if v lies to the left of L, determine the sector (strip or slice) containing v and if a strip, do a point location to determine the cell containing v. If v lies to the right of L then check if v lies within the beachhead, and if not we know that v is closer to Q than to P. Otherwise, locate the cell containing v. In either case, we can decide if v is closer to Q than to P. This can be accomplished in $O(\log(n))$ time with one processor per vertex. Repeat the step with the roles of P and Q reversed.

(D) At this point we know which semiedges of Vor(P) (and by a symmetrical operation, Vor(Q)) meet the contour. Using this information we next compute a linear sequence of triangular subcells of cells of Vor(P), whose union covers the contour and such that the contour visits each triangular subcell exactly once, and visits the cells in the given linear order. We call this

structure the *P-conduit.* The contour will finally be determined by combining the P-conduit with the Q-conduit (a merging process).

It will simplify things if all the geometric constructions are bounded. This can be ensured by finding a rectangle which contains on its inside all points in Vor(P) and Vor(Q), and also the top and bottom contour vertices. Since the beachlines on both sides are enclosed by the vertical sides of such a rectangle, we conclude that the contour can only cross its horizontal sides, and no essential information is lost. In our discussion of the conduits we assume that all the cells and edges are bounded, having been "clipped" to lie inside this rectangle. It is relatively easy to construct such a rectangle in logarithmic parallel time. Details are left to the interested reader.

Consider a relevant point p of P, i.e., one whose cell meets the contour. The contour first enters the p-cell (in Vor(P), perhaps clipped) through an upper semiedge, possibly leaves and returns by several pairs of semiedges (each pair being two halves of the same edge), and finally leaves the cell for ever. The direction of travel within the p-cell is clockwise around p and all parts of the contour within the cell are, of course, visible from p. Let e and f be a contiguous pair of semiedges in this clockwise sequence where e is an "entering" semiedge and f a "leaving" semiedge. Consider these two semiedges together with the list of all edges bounding the (clipped) p-cell included between them in clockwise order around p. Connecting the endpoints in this ordered list of edges to p (including e and f), we obtain an ordered list of triangles with the property that the contour enters the first and leaves the last (to enter a different cell) and enters and leaves each intermediate triangle exactly once. This list of triangles forms part of the P-conduit mentioned above. It is relatively straightforward to combine all these lists together for all relevant points p, in the appropriate linear sequence, by assigning the appropriate number of processors to each point p: namely, one to each edge of the clipped p-cell, thereby locating, counting, and sequencing the lists of triangles around p: to link these lists together in the correct way is easily done using one processor per semiedge. The time to do this is $O(\log(n))$. Thus the P-conduit can be formed as a linked list and we can use list-ranking to store an appropriate representation of this structure in an indexed array.

(E) We construct the contour by merging the P-conduit with the Q-conduit. Say that two triangles T_P and T_Q from the respective conduits *interact* if the bisector separating the respective points p and q (apexes of the respective triangles) meets the intersection of the two triangles. Clearly, these triangles interact if and only if the contour contains an edge separating the two points and this edge intersects the common intersection of both triangles. Each conduit triangle interacts with at least one on the other side. Thus the contour can be constructed as follows. Let T' be the median triangle in the Q-conduit. Assigning one processor to each triangle in the P-conduit we can identify the set S of all triangles in the P-conduit which interact with T'. This set is an interval of one or more contiguous triangles in the P-conduit; let T_1 and T_2 be the first and last in the list (these may be the same triangle). Then only T_1 and the triangles above it in the P-conduit can interact with

triangles above T' in the Q-conduit, with a corresponding observation connecting T_2 with the triangles below T'. Thus all pairs of interacting triangles can be identified recursively in $O(\log(n))$ parallel steps. Using this information it is straightforward to finish constructing the contour in logarithmic parallel time.

(F) Finally construct the cells of Vor(S) by fixing up the cells of Vor(P), Vor(Q) as appropriate.

5. Intersections, Partitions, and Related Problems. We consider the following problems: (i) detecting whether a collection of line segments contains any intersection, (ii) triangulating a simple polygon, (iii) balanced recursive partition of a polygon (so the structure of the partition is a logarithmic depth binary tree), and (iv) computing an optimal placement of watchguards in an art gallery. We prove:

THEOREM 7. *The above problems* (i)–(iv) *are all in* $NC_1^+(n \log(n))$. *For problems* (iii) *and* (iv), *if we begin with a triangulated polygon* (*or art gallery*), *then the problem is in* $NC_1^+(n)$.

Note that using straightforward emulation [Brent (1974)], the $NC_1^+(n \log(n))$ algorithms above can be converted into $NC_2^+(n)$-solutions. These time-bounds have recently been improved to $O(\log(n))$ [Atallah and Goodrich (1986), Atallah *et al.* (1987)].

The solutions to all these problems share a common basis, which is the construction of the so-called (*vertical*) *trapezoidal map* of a set of line segments. Let V be a collection of n line segments, no two of which intersect except perhaps at their endpoints. Following Lipski and Preparata (1981) we define the trapezoidal map of V as follows: for each endpoint p of a segment in V, let L_p denote the maximal vertical line segment through p that does not *intersect properly* any segment of V. (We say that two segments intersect properly if their relative interiors intersect.) As is apparent from Figure 5, the finite regions of the trapezoidal map are trapezoids.

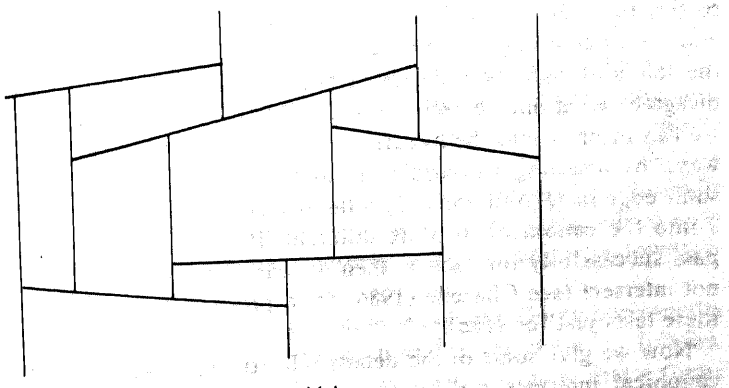

Fig. 5. Trapezoidal map.

It is natural to combine the problem of segment intersection with the problem of computing the trapezoidal map: we give an algorithm that, on input V, detects if there are intersections among the segments of V and, if not, it computes the trapezoidal map of V. For brevity in exposition, we assume no two endpoints have the same x-coordinate and in particular no segment of V is vertical.

Let T be the segment tree associated with the projection of V on the x-axis [Bentley (1977), Bentley and Wood (1980)]. Recall that T is a balanced binary tree of $2n+1$ leaves. For each k between 1 and $2n+1$, the kth leaf of T from the left is associated with a *canonical interval* on the x-axis: this interval is delimited by the kth smallest x-coordinate in V and the $(k+1)$st smallest one (default $\pm\infty$ as appropriate). Similarly, the *canonical interval* of every internal node of T is defined recursively as the union of its two children's canonical intervals. Let $s_1, \ldots, s_n$ be the segments of V. Each node v of the segment tree is assigned a set of indices, $L(v)$, called its *node-set*, and defined as follows: let $I(s_i)$ be the projection of segment s_i on the x-axis. We say that a node of T *covers* $I(s_i)$ if its canonical interval is contained in $I(s_i)$ but the canonical interval of its father is not contained in $I(s_i)$ (or else the node has no father). We define the node-set $L(v)$ of v to be the set of all indices i such that v covers $I(s_i)$. It is straightforward to see that each index appears in $\leq 2\log(n)$ nodes (at most two nodes per level) and thus that the storage requirement of the segment tree is $O(n\log(n))$.

Let us return to our original problem. Define the *canonical strip* of a node v of T to be the infinite vertical slab formed by the Cartesian product of its canonical interval and $(-\infty, +\infty)$. The intersection of the strip with the segments of V whose indices are in $L(v)$ forms the so-called *hammock* of v, denoted $H(v)$ [Chazelle (1986)]. If V is free of proper intersections then the hammock consists of noncrossing line segments joining two vertical lines. For each node v, define the set $W(v)$ to consist of segments of the form $s \cap S_v$ where S_v is the canonical strip of v and s intersects the interior of S_v and s has at least one endpoint in the closure of S_v. Again, note that each s occurs in at most $2\log(n)$ sets $W(v)$.

First we consider the main idea of the algorithm. The problem of constructing the trapezoidal map of V amounts to computing the segment immediately above (below) each endpoint in V. We can detect if the segments in $H(v)$ intersect by sorting the segments in $H(v)$ in two ways: once according to their left endpoint and once according to their right endpoint. (The endpoints lie, respectively, on the left and right boundaries of the canonical strip.) If these sorting orders disagree, we at once detect intersection. Otherwise, for each segment s of $W(v)$, we can insert s into the linearly ordered *cells* of the canonical strip of v in two ways: by inserting s according to its two endpoints. Again, $s \in W(v)$ intersects some edge in $H(v)$ if and only if the two cells obtained by both ways of inserting s into the canonical strip are different. If these two tests for $H(v)$ and $W(v)$ pass successfully for each v, then we can conclude that the segments in V do not intersect (see Chazelle (1986) for a proof). Note that it is insufficient to do these tests just for $H(v)$.

Now we give some of the details related to sorting. The tree T as well as the canonical intervals can be computed in $O(\log(n))$ time, using sorting on n

processors. Next we assign one processor to each segment of V and compute the nodes of T covering each segment in $O(\log(n))$ time (using binary search in the tree). At this point we can assume that we have one list of size $O(n \log(n))$, each list item containing a pair of the form

$$(i, v) = (\textit{segment index, covering node}).$$

By sorting the pairs (i, v) lexicographically so that v is most important, we get the pairs having a common v into consecutive locations of an array. We may regard the consecutive locations as sublists of pairs corresponding to the above set $H(v)$. We may sort the edges of $H(v)$ in the two ways described above and verify that no intersection occurs. Next, the sets $W(v)$ are similarly constructed with $n \log(n)$ processors in logarithmic time. We may then insert each segment s of $W(v)$ into the sorted sublists of $H(v)$: again we do this in two ways and with $O(n \log(n))$ processors and logarithmic time. So now we may assume that we have the quadruples

$$(p, v, s_1, s_2),$$

where p is an endpoint of some segment s in V, $W(v)$ contains a portion of the segment s which terminates in p, and, finally, the segment s_1 (resp. s_2) comes from the hammock edges directly above (resp. below) p in the canonical strip. We sort again to collect in consecutive locations the quadruples involving a fixed endpoint p. By computing the minimum distance among the s_1 (resp. s_2) from p, we obtain the edge of V directly above (resp. below) p. This completes our algorithm for the trapezoidal map. To prove that the algorithm is correct, it suffices to observe that the segment immediately above (resp. below) an endpoint p will necessarily contribute a hammock edge which lies immediately above (or below) the endpoint in question.

Note that the data-structure can be used for planar point location. Better yet, it allows us to determine the segment immediately above a query point in $O(\log^2(n))$ time, using $O(n \log(n))$ space.

In the next two subsections we show two applications of the trapezoidal map.

5.1. Triangulating a Simple Polygon. To triangulate a simple n-gon P, we first compute its trapezoidal map. We invoke Chazelle and Incerpi (1984) in the following. Our goal is first to partition P into monotone polygons. To do this, at each reflex corner c that lies in the relative interior of a vertical edge of a trapezoid, we introduce a diagonal edge from c to the corner of P on the opposite vertical edge. Such diagonals can be constructed in constant time. This now partitions P into a collection of horizontally monotone polygons. A horizontally monotone polygon consists of an *upper* and a *lower chain* of edges, which meet at the two extremal (leftmost, rightmost) corners. A chain (upper or lower) is *trivial* if it consists of a single edge.

We show how to partition a horizontally monotone polygon Q into horizontally monotone subpolygons with a trivial upper or trivial lower chain, as follows. Allocate one processor to each edge e of Q. By symmetry, assume that e forms

Fig. 6. Horizontally monotone polygon with trivial upper chain.

the trivial upper chain. If the region of the plane sufficiently close to e and below e is outside polygon P then do nothing. Otherwise, denoting by u_L and u_R the left and right endpoints of e, we (conceptually) drop the vertical line segments from u_L and u_R (resp.) to the boundary of P. Let these line segments meet the boundary of P at v_L and v_R, respectively. Introduce diagonals from u_L (u_R) to the leftmost (rightmost) vertex of P between v_L and v_R. If u_L (resp. u_R) is the leftmost (resp. rightmost) vertex of the lower chain then $u_L = v_L$ (resp. $u_R = v_R$), in which case we do not introduce a diagonal. Compressing the vertices of each monotone polygon into a single array by parallel prefix, this produces the desired partition of Q in $O(\log(n))$ time using q processors, where q denotes the number of vertices of Q (Figure 6).

So now we may assume that $Q = \{w_1, \ldots, w_q\}$ is a horizontally monotone polygon with a trivial upper chain $w_1 w_q$, and lower chain given by the polygonal line $w_1, \ldots, w_q$. We break up the lower chain into $\sqrt{q}$ contiguous segments of equal size and compute the *upper hull* of each piece (that is, the upper chain of its convex hull). This is similar to our convex hull algorithm in Section 3. These hulls form a continuous path that partitions Q into a connected upper portion adjacent to $w_1 w_q$ and at least $\sqrt{n}$ portions below the hulls. Then we recursively triangulate the parts of Q below each upper hull (Figure 7).

As in the analysis in Section 3, if we can triangulate the part of Q above these upper hulls in $O(\log q)$ time using $O(q)$ processors then the complete triangulation of Q will have the same asymptotic complexity. Once again by assigning one processor per pair of upper hulls we can compute all pairwise tangents in logarithmic time (see Section 3). Next we build up a partial triangulation of the region of Q between these upper hulls and the trivial upper chain: (1) pair up

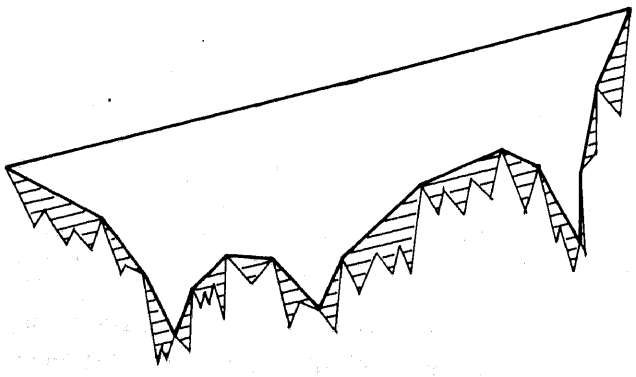

Fig. 7. Recursively triangulate below each of the $\sqrt{n}$ upper hulls.

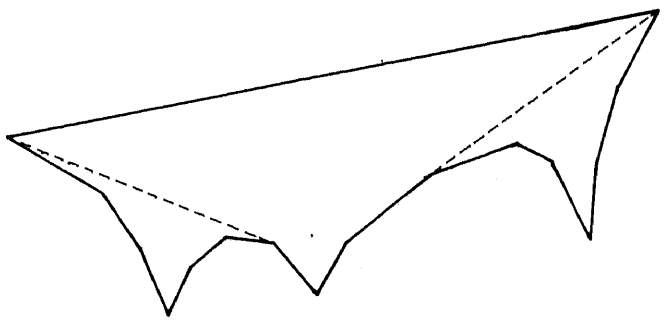

Fig. 8. Partial triangulation of region above the upper hulls.

consecutive upper hulls (the first and second, the third and fourth, etc.) and for each pair determine the unique tangent between the two upper hulls, adding it into the triangulation; (2) iterate on this process until there are only two upper hulls left. Step (1) can be implemented in constant time: to see this, assign one processor per vertex, per current upper hull, and per precomputed tangent. At the beginning of each new iteration we assume that the processors assigned to vertices which are no longer on the current upper hulls are deactivated. Then in $O(1)$ time the vertices of the current upper hulls are each labeled with a number unique to the hull. In this way each (tangent) processor can then decide immediately if it is relevant, that is, if it connects two hulls in a given pair. If so, then the tangent is added into the triangulation and the processor of every vertex lying strictly below the tangent is deactivated. Tangents which cross the newly added edges of the triangulation also have their processors deactivated. The iteration can then proceed. When the iterations terminate, we have the situation of Figure 8.

It is easy to see that the partial triangulation leaves behind polygonal regions with very simple characteristics: such polygons $Q' = \{u_1, \ldots, u_k\}$ are horizontally monotone with a trivial upper chain $u_1 u_k$, and there is a unique lowest vertex u_l $(1 \leq l \leq k)$ such that the y-coordinate of u_i is decreasing as i increases from 1 to l, and the increasing as i increases from l to k. Furthermore, the subpolygons $\{u_1, \ldots, u_l\}$ and $\{u_l, \ldots, u_k\}$ are also horizontally monotone with trivial lower chains $u_1 u_l$ and $u_l u_k$, respectively (Figure 9). Since they are also convex, it is now an easy exercise to triangulate each Q' using k processors and $O(\log(k))$ time.

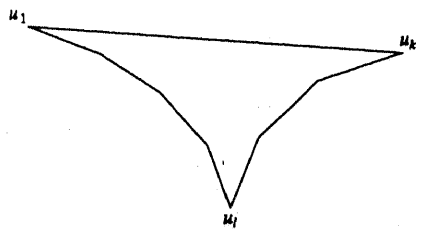

Fig. 9. Special monotonic polygons after partial triangulation.

In conclusion, we have given an algorithm for triangulating a simple n-gon in $O(\log(n))$ time, using n processors if the vertical trapezoidal map is available, or using $O(n \log(n))$ processors otherwise.

5.2. Polygon-Cutting, Recursive Decompositions, and Applications. The polygon-cutting theorem states that any simple polygon has a diagonal which *cuts* it into two roughly equal pieces. In its simplest form, it says that an n-gon can always be cut up into polygons of at most $2(n/3)+1$ vertices. As was shown in Chazelle (1982) this property, though quite simple, is of great importance, because like other separator theorems it sets the stage for divide-and-conquer algorithms. We illustrate this with the problem of placing guards in an art gallery. First we review how the polygon-cutting theorem is used.

In many problems on simple polygons, it is useful to divide a polygon roughly evenly for separate processing, and to proceed recursively. This amounts to constructing a "decomposition tree" that is balanced (i.e., logarithmic depth), where each node v stores a diagonal of the polygon (representing a partition) and the left and right subtrees below v represent the two polygons defined by this partition; the leaves represent triangles. Note that the set of all diagonals stored in the decomposition tree forms a triangulation of the polygon. The polygon-cutting theorem implies the converse: every triangulation can be organized into a balanced decomposition tree. The graph-theoretic dual of the triangulation is a tree whose nodes and edges are in bijective correspondence with the triangles and the diagonals, respectively. If a diagonal partitions the boundary of P into two polygonal lines of respectively p and $n-p$ edges then removing the corresponding edge from the tree will leave two trees of respectively $p-1$ and $n-p-1$ nodes. What is now needed is an efficient parallel algorithm for computing the size of the subtrees adjacent to every edge in the tree. This can be reduced to computing the number of descendants of every node in a rooted tree, which we have already used in our algorithm for Voronoi diagrams (Section 4). From this we conclude that balanced decomposition trees can be implemented in $NC_1^+(n)$ if the polygon is already triangulated, and in $NC_1^+(n \log(n))$ otherwise.

In Chvátal (1975) it is shown that as long as P is simple it is always possible to place $\lfloor n/3 \rfloor$ watchguards inside P to keep the whole polygon ("art gallery") in check. This means that with $\lfloor n/3 \rfloor$ carefully chosen points every point in P is visible from at least one of the chosen points. A simple proof follows from the property that any triangulation of P is 3-colorable. Being a tree, the dual graph of a triangulation of P has at least one leaf. This means that if P has more than three vertices, its triangulation contains at least one triangle abc adjacent to the rest of P through ab only. Removing this triangle allows us to 3-color the remainder recursively, which then leaves one color available for vertex c. A solution to the art-gallery problem then results from placing a guard at each vertex colored with the least-used color.

To overcome the inherently sequential nature of this algorithm, we first assume that a decomposition tree T for the polygon is available. Our goal is to 3-color a simple polygon P. A 3-coloring of P is represented by storing at each edge of

T a permutation of the colors c_1, c_2, c_3, and at the leaves of T, an arbitrary 3-coloring of the vertices of the triangle. Note that a vertex of P may appear in more than one triangle but we make no assumption that the color assigned to its various appearances at the leaves are consistent. The consistency question is resolved by the permutations at the edges. Let us describe how the color of any vertex v of P is obtained in this coding scheme. Say v appears in the triangle at a leaf l of the decomposition tree T. If v is "locally" assigned a color c at l, then the color of v in the "global coloring" is obtained by applying the permutations found along the path from l to the root of T. We must ensure consistency, i.e., different leaves containing v will lead to the same global color for v. This is simple. Let us say that a node u of T is *globally consistent* if for all vertices v of P, whenever two leaves below u assign local colors to v, then the path from these two leaves to u will assign the same global color to v. Assume that all nodes at distance greater than k from the root are globally consistent. Let u be a node at distance k from the root, and u_L, u_R be the two children of u. It is easy to assign permutations to the two edges from u to its children so that u is globally consistent. (To see this, note that the possible cause for inconsistency is at the two vertices incident to the diagonal of P at u.) Starting from the nodes furthest away from the root, in $O(\log(n))$ steps, we can compute the permutations. In another $O(\log(n))$ steps, with one processor per leaf of T, we can assign the global color to each v in P.

6. Polygon Optimization Using a Back-and-Forth Technique. There has been considerable interest recently in optimization problems where the solution space is a suitable class of polygons (see Chang (1986)). One such class of problems is the following, for fixed $k \geq 3$. Given a convex n-gon P, determine the minimum-area (resp. maximum-area) k-gon that circumscribes (resp. is inscribed in) P. First consider the circumscribing problem. In the sequential setting, O'Rourke *et al.* (1986) give a linear-time algorithm for $k=3$ while Aggarwal *et al.* (1985) and Chang (1986) give an $O(n^2 \log(n) \log(k))$ algorithm for $k \geq 4$. Recently, a factor of $\log(n)$ has been shaved off this result using a general technique in Aggarwal *et al.* (1986). Since the solution for $k \geq 4$ is at least quadratic time, we cannot have an $NC_k^+(n)$ solution without improving the sequential result. Thus we focus on the minimum circumscribing triangle problem. Our main result is given next: the stated runtime is achieved using what we have dubbed "back-and-forth subdivision," a technique new in computational geometry.

THEOREM 8. *The problem of computing a minimum circumscribing triangle for a convex n-gon is in* $NC_1^+(n)$.

For convenience we refer to the n edges of a given convex polygon P by their indices in clockwise order from 0 to $n-1$. Also, given an edge i, we denote by $\alpha(i)$ the last edge in clockwise order from i which forms a positive angle with the line through i: put another way, $\alpha(i)$ is the unique edge such that its second terminal (in clockwise order) supports the tangent to P parallel to i and opposite i (there is an easy special case when this tangent lies along an edge) (Figure 10).

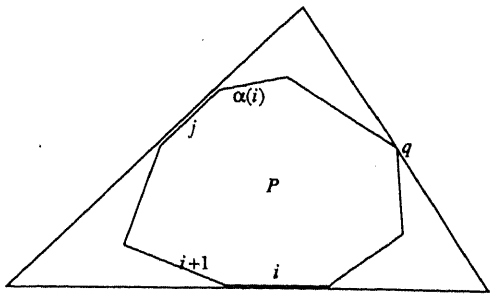

Fig. 10. The edges $\{i, i+1, \ldots, \alpha(i)\}$ of P and $T(i, j)$.

We exhibit a $\sqrt{n}$ divide-and-conquer strategy (cf. Section 3) which relies on the following geometric characterization developed in Klee and Laskowski (1985), and O'Rourke *et al.* (1986).

We use the following known properties:

(i) Among the minimum circumscribing triangles of P, at least one such triangle has two of its sides "flush" (where a side is flush if it lies along an edge of P).

(ii) The midpoint of each side of any minimum circumscribing triangle of P must touch P.

(iii) Given i and j, where $i \leq j \leq \alpha(i)$, let $T(i, j)$ denote the triangle with two sides flush with i and j such that the third side has its midpoint q touching P and such that i, j, and q occur in clockwise order on the boundary of P. We write $i \leq j \leq k$ to mean j is in the range $i \cdots k \pmod n$. We write $m(i, j)$ for the edge of P that touches q. In general the third side of $T(i, j)$ is not flush so there will be two choices for $m(i, j)$; we will conventionally choose the larger index (mod n). For each i, there are at most two indices $j^*(i)$ such that the midpoint of the second side of $T(i, j^*(i))$ (i.e., the side that is flush with $j^*(i)$) actually touches P. If there are two choices (necessarily adjacent) for $j^*(i)$, we choose the edge that has the larger index (mod n). Let $m^*(i)$ refer to the edge $m(i, j^*(i))$, and $T^*(i)$ refer to $T(i, j^*(i))$. We say edge $j^*(i)$ (resp. $m^*(i)$) is *left critical* (resp. *right critical*) with respect to i. We call triangle $T^*(i)$ a *locally minimum* triangle with respect to edge i. Note that the midpoint of the edge of $T^*(i)$ flush with i need not touch P.

(iv) Monotonicity property. An edge $j \in P$ $(i \neq j)$ is called *low* (resp. *high*) with respect to i if j precedes $j^*(i)$ (resp. j succeeds $j^*(i)$) in a clockwise traversal of P. By definition then, for each i, the sequence of edges $(i, i+1, \ldots, \alpha(i))$ consists of a sequence of low edges, followed by a left critical edge, followed by a sequence of high edges. The monotonicity property is as follows. Suppose that p is the midpoint of the side of $T(i, j)$ lying along j and p is not on j: if p and the side containing i are on the same side of j then j is high, and if they are on opposite sides then j is low, with respect to i. (Note that this deduction is made without knowledge of the left or right critical edges of i.)

(v) Interspersing property. If $i \leq i'$ then $j^*(i) \leq j^*(i')$ and $m^*(i) \leq m^*(i')$.

Property (v) is a weakened form of tha
is sufficient for our purposes. Given i and
search in logarithmic serial time; thus, gi
use binary search to compute $j^*(i)$ in $O($
can compute the locally minimum triangl
globally minimum triangle will be found
in $O(\log(n))$ further steps. We shall see
$O(\log(n))$ parallel algorithm using wha
We define partial right inverses for th
unique i such that $j^*(i-1) < k \leq j^*(i)$ (
the same way. (The functions j^* and m^*
the range of j^* then $j^*(j^{-1}(k)) = k$; simila
(iv) shows that these quantities are w
interval I of length s containing $j^{-1}(k)$,
then a single processor can compute j^*
we shall compute approximate values
inverse functions j^{-1} and m^{-1}. We ca
information about j^{-1} and m^* and perfo
to contain the true value. The other appr
Repeating this process sufficiently ofte
all these functions exactly. A single ph
each k an interval $a \cdots b$ of length s,
we say that the interval $[a, b]$ *brackets*
(n overall), we can improve this estim
goes for calculating estimates of the ot
parallel steps.

For simplicity we assume that n is
maximum integer $\leq n$ of the form

where t is an integer (specifically, t
difficulties. This initial value of s lies

LEMMA 9. *Let s be a perfect square*
a power of 2). For $0 \leq i \leq n/\sqrt{s} - 1$ let s
the polygon P. Suppose that to each
$M^{-1}(k)$ are associated respectively br
where each interval is of the form $[s_i\sqrt{s},$
intervals are of length $s+1$ and their
processors all of these intervals can be
$O(\log(s))$ parallel steps.

PROOF. For $0 \leq i < n/\sqrt{s}$ we assig
parallel steps the correct subinterval
processor uses binary search on M^*

Property (v) is a weakened form of that derived by O'Rourke *et al.* (1986) but is sufficient for our purposes. Given i and j we can compute $m^*(i, j)$ using binary search in logarithmic serial time; thus, given i, the monotonicity property lets us use binary search to compute $j^*(i)$ in $O(\log^2(n))$ serial time. Thus n processors can compute the locally minimum triangles $T^*(i)$ for each i. By property (i), the globally minimum triangle will be found among these $T^*(i)$. This can be found in $O(\log(n))$ further steps. We shall see that this can be improved to yield an $O(\log(n))$ parallel algorithm using what we call "back-and-forth subdivision."

We define partial right inverses for the two functions j^* and m^*: $j^{-1}(k)$ is the unique i such that $j^*(i-1) < k \le j^*(i)$ (in clockwise order); $m^{-1}(k)$ is defined the same way. (The functions j^* and m^* are not usually bijective, but if k is in the range of j^* then $j^*(j^{-1}(k)) = k$; similarly for m^{-1}.) The monotonicity property (iv) shows that these quantities are well defined and, furthermore, given an interval I of length s containing $j^{-1}(k)$, if $j^*(i)$ is known for all i in that interval, then a single processor can compute $j^{-1}(k)$ in $O(\log(s))$ time. The idea is that we shall compute approximate values for functions j^* and m^* and for their inverse functions j^{-1} and m^{-1}. We can refine an estimate of $j^*(i)$ by using information about j^{-1} and m^* and performing binary search in the interval known to contain the true value. The other approximations can be refined simultaneously. Repeating this process sufficiently often, namely, log log(n) times, we compute all these functions exactly. A single phase of the process involves knowing for each k an interval $a \cdots b$ of length s, say, such that $a < j^*(k) \le b$. In this case we say that the interval $[a, b]$ *brackets* $j^*(k)$. Using sufficiently many processors (n overall), we can improve this estimate to an interval of length $\sqrt{s}$: the same goes for calculating estimates of the other functions. The phase takes $O(\log(s))$ parallel steps.

For simplicity we assume that n is a power of 2. We initially choose s as the maximum integer $\le n$ of the form

$$2^{2^t},$$

where t is an integer (specifically, $t = \lfloor \log_2(\log_2(n)) \rfloor$). This avoids rounding difficulties. This initial value of s lies between $\sqrt{n}$ and n.

LEMMA 9. *Let s be a perfect square perfectly dividing n (which is assumed to be a power of 2). For $0 \le i \le n/\sqrt{s} - 1$ let s_i be the $i\sqrt{s}$th edge in clockwise order around the polygon P. Suppose that to each k four intervals $J^*(k)$, $J^{-1}(k)$, $M^*(k)$, and $M^{-1}(k)$ are associated respectively bracketing $j^*(k)$, $j^{-1}(k)$, $m^*(k)$, and $m^{-1}(k)$, where each interval is of the form $[s_{i\sqrt{s}}, s_{(i+1)\sqrt{s}}]$. In other words, the given bracketing intervals are of length $s+1$ and their boundaries are multiples of s. Then with n processors all of these intervals can be reduced to intervals of the form $[s_i, s_{i+1}]$, in $O(\log(s))$ parallel steps.*

PROOF. For $0 \le i < n/\sqrt{s}$ we assign $\sqrt{s}$ processors to compute in $O(\log(s))$ parallel steps the correct subinterval $[s_t, s_{t+1}]$ of $J^*(s_i)$ bracketing $j^*(s_i)$. The tth processor uses binary search on $M^*(s_i)$ to determine whether $s_t \le j^*(s_i)$, i.e., to

determine whether s_t is low, critical, or high with respect to s_i. Thus the correct interval bracketing $j^*(s_i)$ is located. Reassigning the given processors to this interval, $j^*(s_i)$ is computed exactly. This exact information about $j^*(s_i)$ at these $n/\sqrt{s}$ sample sides s_i enables us to bracket $j^{-1}(k)$ for *all* sides k of P, since

$$s_i < j^{-1}(k) \leq s_{i+1} \qquad \text{if and only if} \quad j^*(s_i) < k \leq j^*(s_{i+1}),$$

so this can be checked in time $O(\log(s))$ with one processor reassigned to each edge k, using binary search on the subsequence of edges s_i contained in the interval $J^{-1}(k)$ ($j^*(s_t)$ is known exactly for all t).

Thus we have $j^*(s_i)$ computed for all i and $j^{-1}(k)$ correctly bracketed for all k. Next for each sample side s_k we assign $\sqrt{s}$ processors: one is assigned to each side i in the interval $J^{-1}(k)$ (which is now of the form $[s_t, s_{t+1}]$). The processor assigned to i queries $M^*(i)$ and determines whether k is low, critical, or high with respect to i. By the monotonicity property this enables $j^{-1}(s_k)$ to be determined exactly. When this computation is finished, $J^*(i)$ can be recomputed for *every* edge i just as $J^{-1}(i)$ was computed. The refinement of our estimates of $M^*(k)$ and $M^{-1}(k)$ is achieved by symmetric methods. □

The initial conditions for applying the lemma are slightly different since in general $n > s$. The aim then is to achieve initial bracketing intervals of length s; however, essentially the same ideas apply. Repeating the refinement phase $\log \log(n)$ times all quantities are computed exactly. Notice that since all the tasks assigned involve easily computed subsequences and subintervals of $0 \cdots n-1$, processor assignment presents no difficulty in this algorithm. It is clear that this method takes overall parallel time

$$\sum_k O(\log(n^{2^{-k}})) = \sum_k O\left(\frac{\log(n)}{2^k}\right) = O(\log(n)).$$

Additional Remarks. Boyce *et al.* (1985) have shown that some of the properties similar to those given for the minimum circumscribing triangle also hold for the maximum inscribed triangle. It is not clear if the back-and-forth technique can be extended here since the monotonicity property (iv) is not available. However, an $O(\log^2(n))$ solution is possible. For the dual problem of computing the maximum inscribed triangle, simple divide and conquer gives a solution using $O(\log^2(n))$ time and n processors. In fact, unlike the minimum circumscribed k-gons (for $k \geq 4$), such a divide-and-conquer technique yields an $O(k \log^2(n))$ time for computing a maximum inscribed k-gon using n processors. Finally, Toussaint (1983) has shown that the minimum circumscribing rectangle for a given convex n-gon can be found in $O(n)$ sequential time. Using the geometric characterization given by Freeman and Shapira (1975), it is easy to compute such a rectangle in $O(\log(n))$ time using only $n/\log(n)$ processors.

In fact, the diameter of a convex polygon can also be computed in $O(\log(n))$ time using $n/\log(n)$ processors, as the following brief outline suggests.[10] If we can compute for each i its antipodal vertex $\delta(i)$ (the endpoint of $\alpha(i)$) it is straightforward to compute the distance of $\delta(i)$ from the line through i for each i, and then compute the minimum of all these distances with the stated resources. We take $n/\log(n)$ processors and assign each processor to a group of $\log(n)$ successive sides of the polygon. Each processor first computes the antipodal vertex for the first side in its alotted group (this involves unimodular search in logarithmic serial time), and then begins to scan in a clockwise direction along its group of sides. Its "primary task" is to compute $\delta(i)$ for all sides within this group. Going from side i to $i+1$, the corresponding antipodal vertex may either remain the same, increase by 1, or increase by more than 1. In the latter case the gap between $\delta(i)$ and $\delta(i+1)$ could be considerably more than $\log(n)$, and the processor might not achieve its primary task. However, in this case, if k denotes the side beginning at $\delta(i)$, we know that $\delta(k)$ is the vertex ending the ith edge. Thus the processor can begin a "secondary task," namely computing $\delta(j)$ for $j=k, k+1, \cdots$ until it either reaches the end of the group containing k or $\delta(j)$ increases, in which case it can determine $\delta(i+1)$ and revert to its primary task. It is easy to show that, for every i, $\delta(i)$ is computed by some processor, either as a primary or secondary task.

7. Convex Hull in Three Dimensions and Related Structures. In this section we discuss the problem of computing the convex hull $H(S)$ of a set S of points in three-dimensional space. The method originally described in the FOCS paper ran in parallel time $O(\log^4(n))$, but the method to be outlined here is in $NC_3^+(n)$. Thus we now match the asymptotic efficiency of Chow's method (1980); since our method does not use optimal parallel sorting it should be expected to work better in practice. As with the two-dimensional Voronoi diagram problem the algorithm is based on a serial divide-and-conquer technique already known in the literature. We employ a new data-structure, which has independent interest, to solve queries of the following kind: given a line L and a polytope K, determine whether L intersects K and if not return a plane through L tangent to K. Such queries are called *line-queries*; they will be discussed in detail later. Let us state the main result of this section.

THEOREM 10. *There is an $NC_3^+(n)$ algorithm for constructing the convex hull $H(S)$ of a set S of n points in 3-space, which only uses sorting at a preprocessing stage and hence can use any sorting algorithm in $NC_3^+(n)$ or better. The algorithm produces as a by-product a data-structure suitable for answering line-queries about $H(S)$ in serial time $O(\log^2(n))$.*

For reasons of space we only provide an outline description of our algorithm, which, while of some interest in its own right, is very elaborate and achieves a

[10] As an anonymous referee pointed out, the same problem can be solved just as efficiently taking the set of edges as displacement vectors and merging this set with its mirror image (see Shamos [1977]).

runtime which has been bettered by much simpler methods (this is discussed in a concluding subsection). Much more detail is supplied in an existing technical report [Aggarwal *et al.* (1987)]. Recall the serial divide-and-conquer algorithm [Preparata and Hong (1977)] to compute the convex hull of S: the set of points S is evenly split into two sets by a horizontal plane R, which does not contain any point in S; let P and Q denote the points in S above and below R, respectively. Recursively compute $H(P)$ and $H(Q)$; compute the *sleeve* joining the two hulls, i.e., the chain of faces tangent to the two hulls and meeting three or more points of S; then merge $H(P)$ with $H(Q)$ along the sleeve to form the convex hull $H(S)$.

For the rest of this section R will continue to denote a horizontal plane, which we call the *separating plane*, partitioning S into upper and lower parts P and Q, respectively. Consider that part of the sleeve on and above the separating plane R. We might imagine that it was homeomorphic to a cylinder with boundary, but sleeve faces can meet one another above the separating plane, so it can have several holes. Each partial sleeve face has an edge meeting R, and two edges meeting this edge; all other edges on this partial face are called *seam edges* and are given a left-to-right (i.e., clockwise) orientation. These chains of seam edges join together to form what is called the *upper seam* of the $H(S)$. Thus the upper seam has the form of a directed Eulerian circuit, but vertices may be visited more than once and the same edge of $H(S)$ may occur with both orientations along the seam.[11] The *lower seam*, occurring below the separating plane R, is defined similarly.

Our algorithm is simple in principle though the development is lengthy: it depends on computing the seams efficiently at each stage. To compute the upper seam, say, assign one processor to each edge of $H(P)$, to perform what is called a "line-query," relative to $H(Q)$. Consider a fixed edge of $H(P)$; let L be the line containing it. If the edge belongs to the upper seam it is necessary (a) that L should not meet the interior of $H(Q)$, and (b) one (or conceivably both) of the two planes through L tangent to $H(Q)$ should not intersect the interior of $H(P)$. To determine whether a given plane passing through L is tangent to $H(P)$ can be accomplished in constant time by inspecting the two faces of $H(P)$ meeting L.

Once the edges in the seam have been calculated, their cyclic connection order can be ascertained in logarithmic time by a list-ranking process. The full structure of the sleeve can be deduced once both seams have been constructed by implementing what is essentially a merging process, easily accomplished in logarithmic time. Therefore we now consider these line-queries more closely.

Let K be a convex polytope and L an oriented line not meeting the interior of K. There are two half-planes through L tangent to K (they bound the convex hull of $L \cup K$: let us ignore the trivial case where K and L are contained in the same plane). It is helpful to distinguish these planes in a canonical way.

[11] The possibility that the seam can have self-intersections complicates both the sequential algorithm and ours. We are indebted to Herbert Edelsbrunner (private communication) for drawing our attention to this possibility.

DEFINITION 1. With K and L as above, consider the set of all half-planes bounded by L which intersect K. All such half-planes can be parametrized by a rotational angle θ about L. Assuming that L does not intersect (the interior of) K, these angles θ sweep out an interval $[\theta_0, \theta_1]$ of angles where we can distinguish these two bounding angles (since L is oriented) by requiring that $\theta_1 > \theta_0$ in the anticlockwise sense, and $\theta_1 - \theta_0 \leq \pi$. These bounding angles define the two planes through L tangent to K, and we call the one at angle θ_0 the *back tangent-plane* to K through L, and the other the *front* tangent-plane. The *line-query* for L and K is to determine whether L meets K transversally, and, if not, to compute the front and back tangent planes through L to K.

If we visualize L and K so that L is in the plane of the page, and oriented upward, and K is to its left, then the front and back planes will be in front of and behind K, respectively (see Figure 11). Such queries arise naturally in implementing the convex hull algorithm. To begin with, let us consider a much simpler query which we call a *point-query*:

> Given a plane T meeting a corner p of a convex polyhedron K, determine whether T is tangent to K, and, if not, give the two faces incident to p which T cuts.

This is solved by providing a *test polygon* at p, namely a polygon which forms the boundary of the region of intersection of a fixed plane M separating p from all the other corners of K. Clearly, the plane T cuts K if and only if it cuts this test polygon, and if so the edges where it cuts the polygon identify the faces incident to p which it cuts. The corners of the test polygon are unimodal with respect to height above (or below) the plane T, so it is then easy to solve the queries in logarithmic serial time by a variant of binary search. The problem, therefore, is to construct a test polygon in parallel, with one processor, say, assigned to each edge of K incident to p. Indeed, it is easy to construct such a

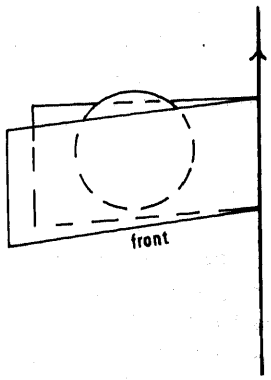

Fig. 11. Illustrating the front and back tangent planes.

polygon once at least one line N has been found which meets the interior of K at p, since then a suitable separating plane normal to this line can be found (take all the projections of the incident edges on N, find the one closest to p in logarithmic time; let r be its distance from p; then choose the plane passing perpendicularly through N at half this distance). To find a line entering K at p, simply choose three corners adjacent to p and let q be their average; then pq defines a suitable line. Summarizing,

LEMMA 11. *Given a convex polyhedron K with l corners, it is possible to construct test polygons for each corner in $O(\log(l))$ parallel time overall with l processors.*

The following lemma shows where point-queries could arise in constructing the convex hull. Its simple proof is omitted.

LEMMA 12. *Let T be a plane which meets the sleeve of $H(S)$ at some corner p but does not intersect any face of the sleeve. Then T can meet the interior of $H(S)$ if and only if it meets the interior at p, and in this case the polygonal region of intersection of T with $H(S)$ is contained in $H(P)$ if p is in the upper seam and in $H(Q)$ if p is in the lower seam.*

The rest of this section is organized as follows. We introduce the notion of "seam polytope," and of localizing a query to a subpolytope. We introduce the idea of a "near split" on a polytope which can be used to localize a line-query on the polytope. We then give only a rough description of the remaining components of our algorithm: for further details please refer to the technical report. We conclude with a discussion of some simpler methods (given by others).

7.1. Definition and Characteristics of a Seam Polytope. For each of the two seams of the sleeve as considered above we construct a "seam polytope," a bounded convex polytope which has a distinguished circuit of directed edges, called the seam, which matches the corresponding seam of the sleeve.

DEFINITION 2. A *seam polytope* is a convex three-dimensional polytope with two distinguished horizontal faces, the *base* and the *top*, and the other faces partitioned into two sets, called, for mnemonic purposes, the *blue* and *green* faces, respectively. The edges along the top and base are called top and base edges, respectively. No green face meets the base and no blue face meets the top. Each blue face meets the base in a proper edge and is either triangular or trapezoidal in shape. The *seam vertices and edges* are those vertices and edges bounding blue faces but not meeting the base. Edges connecting the seam to the

base are called *blue edges*, and edges meeting only green faces are called *green edges*, and vertices meeting only green faces are called *green vertices*.

The assumption about the blue faces is not essential but simplifies things: it is based on the assumption that the blue faces are formed by intersecting sleeve faces with the halfspace above the separating plane, and if S is in general position then all its faces are triangular. From our previous discussion we know that by orienting the blue faces we can describe an Eulerian circuit of directed edges which includes all seam vertices and edges but can include the same (undirected) edge twice (with both orientations): specifically, when two blue faces meet at a seam edge, that edge will be met twice in the directed circuit. Henceforth the *seam* will generally denote the *directed* circuit so formed. It cannot be assumed that every green face meets the top of the seam polytope; moreover, the green faces can have any number of bounding edges, and their connectivity pattern can be complex. The set of blue faces is connected, since each face is connected to the base; also, there is a connected set of green faces meeting the top, which is called the "mainland." However, there may be more "green islands" formed where the seam returns upon itself. See Figure 12.

Let us reconsider briefly the construction of the sleeve and the two seam polytopes that are built during this construction. Without loss of generality we only consider the upper seam. The base of the seam polytope is the intersection of $H(S)$ with the separating plane R. The top of the seam polytope may be defined by a horizontal (parallel to R) plane which passes just above the highest vertex on the upper seam (since the vertices in S can be assumed presorted with respect to the z-coordinate, this vertex can be obtained with little computational overhead). The planes supporting the faces on the sleeve yield the blue faces of the polytope; the planes supporting those faces of $H(S)$ which meet the blue faces above R yield the green faces of the seam polytope.

For the rest of this section, if K is any polytope and A any subset of K, suppose that we have determined for a given line L that if L penetrates K then it penetrates A, and if it does not penetrate K then the front (resp. back) plane through L to K must coincide with the front (resp. back) plane through L to A. Then we say that the given line-query (for front or back plane) has been *localized* to A. Since the front plane to K through L coincides with the back plane to K through L',

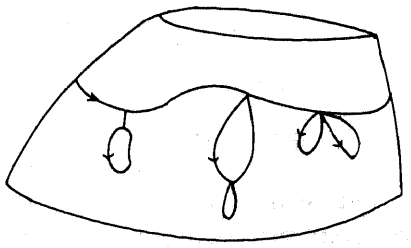

Fig. 12. A seam polytope: sea, mainland, and islands.

where L' is the same line as L with the opposite orientation, we can always restrict our attention to the front plane.

A hierarchy of seam polytopes can be constructed as $H(S)$ is being built; for any oriented line L, a line-query for $H(S)$ and L can be solved by a recursive sequence of line-queries for L and this hierarchy of seam polytopes, as follows. No proof is offered here; one is given in Aggarwal *et al.* (1987).

LEMMA 13. *Let $H(S)$ be as above and suppose that Π is the seam polytope for its upper seam. Given an oriented line L, suppose that we wish to determine whether L intersects the interior of $H(S)$ and if not to give the front plane through L tangent to $H(S)$. Then a solution to the corresponding question for Π either solves the query for $H(S)$ or localizes it to $H(P)$ or $H(Q)$.*

The following is an immediate corollary.

LEMMA 14. *Suppose that a hierarchy of seam polytopes have been constructed along with $H(S)$, and that S has size $O(n)$. If auxiliary search structures have been provided which enable the line-queries for the seam polytopes to be solved in serial time $O(\log(n))$ then line-queries for $H(S)$ can be solved in $O(\log^2(n))$ serial time overall.*

We shall next see an appropriate search structure for an individual seam polytope which achieves the time-bound in Lemma 13. It will be enough to build such a structure (and also the seam polytope itself, which only resembles $H(S)$ near the seam) in $O(\log^2(n))$ time; this construction will be given in outline.

7.2. Near Splits, Core, and Lobes of a Seam Polytope. A data-structure we describe in outline to help solve line-queries makes extensive usage of a partition method which we call *near-splitting*. A natural divide-and-conquer method to solve a line-query for a line L and a polytope K would be to divide K by a plane into two simpler parts: by solving the line-query relative to the face common to the two parts we could localize to one or other part. Since such a splitting face could have a large number of vertices, to localize the query would require logarithmic serial time. Therefore rather than splitting K into two parts meeting at a common face we decompose K into two convex polytopes which meet in a set with nonempty interior and whose union is K. This motivates the following definition.

DEFINITION 3. Let K be a (seam) polytope, and J a sequence of corners of K such that in cyclic order every successive pair of vertices defines a line-segment belonging to one of the faces of K and no three vertices of J belong to the same

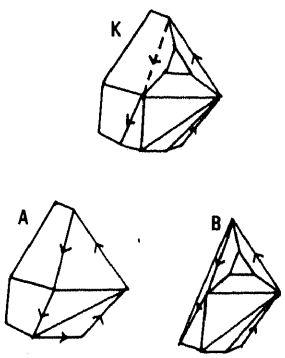

Fig. 13. Near-split of a polytope.

face of K. Connecting pairs of successive vertices in cyclic order around J we obtain a polygonal curve, a Jordan curve, around ∂K. Let A be the convex hull of those corners of K on and outside J; let B be the convex hull of those corners of K on and inside J. Then A and B form a *near-split* of the polytope K. See Figure 13.

LEMMA 15. *With A, B, J, and K as in the above definition, let L be an oriented line. Then if test polygons are provided for both A and B at each vertex of J, a solution to a line-query for L relative to $A \cap B$ either solves the line-query for K or can be used with $O(1)$ extra work (on a serial processor) to localize the query to A or B.*

No proof of the above lemma is offered here. Notice that the provision of test polygons enables a query to be localized rapidly (constant time) even at a high-degree vertex.

The remaining components of our algorithm are rather detailed and we only present them in outline here.

(1) First, introducing several near-splits, we can decompose the seam polytope into what are called in Aggarwal *et al.* (1987) the "core" and "lobes." The core has a rather simple structure in which line-queries can be solved by a form of binary search (based, however, on the idea of near-splits). This enables a line-query to be localized, in logarithmic time, to the correct lobe of the seam polytope.

(2) The green edges on each lobe of the seam polytope form what is essentially a forest of trees, where the leaves are (mostly) along the seam. (Although several green edges can meet the same point on the seam, they can be regarded, for accounting purposes, as defining different leaves.) By adding fictitious edges we can combine all the trees in the forest into one (free) tree. Using,

for instance, a trick of Tarjan and Vishkin's (1985) involving Eulerian circuits, we can orient the edges of this free tree in logarithmic time, giving it a root.

(3) This tree can be "covered" by a binary tree (i.e., a node with $k \geq 3$ children being replaced by a complete binary tree with k leaves). Finally, this binary tree, which we call the *binary structure* of the lobe, can be partitioned recursively using Goodrich's decomposition method as described in Section 4. This induces a recursive decomposition of the lobe by near-splits, yielding the structure we wanted.

(4) The question remains of how to build the seam polytope in the first place (the steps described above can be performed once the seam polytope has been given, together with a cyclic ordering of the seam edges). The polytope can be built by divide and conquer, roughly as follows. Partition the faces incident to the seam into two parts, each contributing half the seam. Recursively construct the polytopes defined by intersecting the halfspaces which the faces define, and intersect the resulting two polytopes to obtain the seam polytope. The concept of near-split is also useful here when forming the intersection, and it is possible to form the intersection (and, at deeper levels of recursion, auxiliary search-structures) in logarithmic time. This yields the seam polytope in $O(\log^2(n))$ parallel time, as claimed.

7.3. A Simpler Approach. This concludes our discussion of the convex hull problem. The construction given here, while elaborate, is slightly more efficient than Chow's. (The more detailed analysis given in Aggarwal *et al.* (1987) makes careful provision for the possibility that the seam can return upon itself, which introduces many complications.) The FOCS version was based on an adaptation of Kirkpatrick's method, and as mentioned above ran in time $O(\log^4(n))$. Richard Cole (private communication) has given a strikingly simple method to implement Kirkpatrick's method in parallel. Let us say only that Kirkpatrick's method was based on the observation that every planar graph has a substantial fraction of low-degree vertices (vertices of degree ≤ 11, say). To develop a suitable search-structure for the graph, compute a maximal independent set of low-degree vertices, discard them, and then reprocess the resulting graph. For our purposes, the "planar graph" is the graph dual to a given convex hull K (see Dobkin and Kirkpatrick (1983)). Richard Cole's observation is as follows: to compute a maximal independent set of low-degree vertices, let G' be the graph formed by the edges incident to the low-degree vertices. Construct a spanning forest of G', and let S be the subset of odd-depth vertices in this spanning forest (relative to an arbitrary choice of roots for the trees in the forest). Since the trees of the forest have bounded degree, the size of S is a fixed fraction of the size of G'. Let G'' be the graph spanned by S. Then every vertex in G'' has degree at most $k-2$ (i.e., 9) where k is the valence of $G' \ldots$ continuing this process for a few more stages we obtain a maximal independent set.

This method can be implemented to solve the convex hull problem in $\alpha(n) \times \log^2(n)$ parallel CREW time with n processors, where $\alpha(n)$ is the cost of construct-

ing a spanning forest. The cost is $\log(n)$ on a CRCW machine using the method of Shiloach and Vishkin (1982), so this implies a much more elegant CRCW solution to our problem.

Recently, Dadoun and Kirkpatrick (1987) showed that a Kirkpatrick decomposition could be built in $O(\log(n)\log^*(n))$ CREW time, thereby constructing the convex hull with an elegant $O(\log^2(n)\log^*(n))$ CREW algorithm.

8. Final Remarks. Although parallel algorithms in areas such as graph-theoretic algorithms and numerical problems have been well studied for several years, a similar development is only beginning in computational geometry. Consequently, even the most basic problems such as triangulation problems and those in geometric optimization have not been put in the class *NC* until the FOCS paper. Standard techniques in the subject such as contour-tracing, plane-sweeping, and gift-wrapping initially seem inherently sequential; one of our main contributions is to show that *NC*-analogues of these techniques actually exist. As hinted before, efficient solutions to the basic problems imply correspondingly efficient solutions to a multitude of other problems, and we have only shown some of the reductions.

It is interesting to note that in the sequential setting the problems of planar convex hulls, planar Voronoi diagrams, and three-dimensional convex hulls all have the same $\Theta(n \log n)$-time complexity. In the parallel case, these problems have been shown to be in

$$NC_1^+(n),\ NC_2^+(n),\ NC_3^+(n),$$

respectively. Our planar convex-hull algorithm, culminating several previous works, turns out to be optimal in a very strong sense and it remains open whether the parallel solutions for the other problems can be further improved.

Although all common problems in computational geometry are in *NC* [Kozen and Yap (1985)], we think that a main goal of parallel studies in computational geometry is to bring problems into the practical subclasses of *NC*. In this paper we focused on $NC_k^+(n)$ for small k. It is our belief that such algorithms (as opposed to $NC_k^+(n^2)$, say) will have practical implications. In particular, it is possible that they could be employed in VLSI and in graphics machines or robots.

Acknowledgments. Several people, including Richard Cole, David Kirkpatrick, Mi Lu, and Ivan Stojmenovic, noticed a false assumption in our Voronoi-diagram algorithm in the original extended abstract. Ivan also provided an alternative merge procedure for the Voronoi-diagram algorithm. Hubert Wagener pointed out the need for semiedges in our Voronoi-diagram algorithm. Herbert Edelsbrunner pointed out that the seams could be nonsimple in the three-dimensional convex-hull algorithm. Finally, Prasoon Tiwari drew our attention to Anita

Chow's valuable Ph.D. dissertation of 1980, which anticipated some of our results: for the three-dimensional convex hull her algorithm was actually better than the one presented at the 1985 FOCS conference.

References

A. Aggarwal, J. S. Chang, and C. K. Yap (1985). Minimum area circumscribing polygons. *Visual Comput.*, **1**, pp. 112–117.

A. Aggarwal, M. Klawe, S. Moran, P. Shor, and R. Wilber (1986). Geometric applications of a matrix searching algorithm. *Proc. 2nd ACM Symposium on Computational Geometry*, pp. 285–292.

A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. K. Yap (1987). Parallel computational geometry. Robotics Report No. 115, Courant Institute, New York University. (Reported at the 26th IEEE FOCS Symposium, Portland, Oregon, 1985.)

M. Ajtai, J. Komlós, and E. Szemerédi (1982). An $O(n \log(n))$ Sorting Network. *Proc. 15th ACM Symposium on Theory of Computing*, pp. 1–9. Also in *Combinatorica*, **3**(1) (1983), pp. 1–19.

S. Akl (1983). Parallel algorithm for convex hulls. Manuscript, Department of Computer Science, Queen's University, Kingston, Ontario.

M. J. Atallah and M. T. Goodrich (1985). Efficient parallel solutions to geometric problems. *Proc. 1985 IEEE Conference on Parallel Processing*, pp. 411–417.

M. J. Atallah and M. T. Goodrich (1986). Efficient plane sweeping in parallel. *Proc. 2nd Symposium on Computational Geometry*, pp. 216–225.

M. J. Atallah, R. Cole, and M. T. Goodrich (1987). Cascading divide-and-conquer: a technique for designing parallel algorithms. *Proc. 28th IEEE FOCS Symposium*, pp. 151–160.

M. Ben-Or (1983). Lower bounds for algebraic computational trees. *Proc. 15th ACM Symposium on Theory of Computing*, pp. 80–86.

M. Ben-Or, D. Kozen, and J. Reif (1984). The complexity of elementary algebra and geometry. *Proc. 16th ACM Symposium on Theory of Computing*, pp. 457–464.

J. L. Bentley (1977). Algorithms for Klee's rectangle problems. Unpublished manuscript, CMU.

J. L. Bentley and D. Wood (1980). An optimal worst-case algorithm for reporting intersections of rectangles. *IEEE Trans. Comput.*, **29**, pp. 571–577.

J. E. Boyce, D. P. Dobkin, R. L. Drysdale, and L. J. Guibas (1985). Finding extremal polygons. *SIAM J. Comput.*, **14**, pp. 134–147.

R. P. Brent (1974). The parallel evaluation of general arithmetic expressions. *J. Assoc. Comput. Mach.*, **21**(2), pp. 201–206.

K. Q. Brown (1979). Voronoi diagrams from convex hulls. *Inform. Process. Lett.*, **9**(5), pp. 223–228.

J. S. Chang (1986). Polygon optimization problems. Ph.D. Dissertation, Robotics Report No. 78, Department of Computer Science, Courant Institute, New York University.

B. Chazelle (1982). A theorem on polygon cutting with applications. *Proc. 23rd IEEE FOCS Symposium*, pp. 339–349.

B. M. Chazelle (1984). Computational geometry on a systolic chip. *IEEE Trans. Comput.*, **33**, pp. 774–785.

B. Chazelle (1986). Reporting and counting segment intersections. *J. Comput. System Sci.*, **32**(2), pp. 156–182.

B. Chazelle and J. Incerpi (1984). Triangulation and shape-complexity. *ACM Trans. Graphics*, **3**(2), pp. 135–152.

A. Chow (1980). Parallel algorithms for geometric problems. Ph.D. Dissertation, Computer Science Department, University of Illinois at Urbana-Champaign, 1980.

A. Chow (1981). A parallel algorithm for determining convex hulls of sets of points in two dimensions. *Proc. 19th Allerton Conference on Communication, Control and Computing*, pp. 214–233.

V. Chvátal (1975). A combinatorial theorem in plane geometry. *J. Combin. Theory Ser. B*, **18**, pp. 39–41.

R. Cole (1986). Parallel merge sort. *Proc. 27th IEEE FOCS Symposium*, pp. 511–516.

G. E. Collins (1975). Quantifier elimination for real closed fields by cylindrical algebraic decomposition. *2nd GI Conference on Automata Theory and Formal Languages*. Lecture Notes in Computer Science, Vol. 33, Springer-Verlag, Berlin, pp. 134–183.

S. Cook and C. Dwork (1982). Bounds on the time for parallel RAMS to compute simple functions. *Proc. 14th ACM Symposium on Theory of Computing*, pp. 231–233.

N. Dadoun and D. G. Kirkpatrick (1987). Parallel processing for efficient subdivision search. *Proc. 3rd Annual Symposium on Computational Geometry*, 1987, 205–214.

D. P. Dobkin and D. G. Kirkpatrick (1983). Fast detection of polyhedral intersections. *Theoret. Comput. Science*, **27**, pp. 241–253.

H. Freeman and R. Shapira (1975). Determining the minimum-area encasing rectangle for an arbitrary closed curve. *Comm. ACM*, **18**(7), 409–413.

L. J. Guibas and J. Stolfi (1983). Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *Proc. 15th ACM Symposium on Theory of Computing*, pp. 221–234. Also to appear in *ACM Trans. Graphics.*

V. Klee and M. C. Laskowski (1985). Finding the smallest triangles containing a given polygon. *J. Algorithms*, **6**, pp. 457–464.

D. Kozen and C. K. Yap (1985). Algebraic cell decomposition in NC. *Proc. 26th IEEE FOCS Symposium*, pp. 515–521.

T. Leighton (1984). Tight bounds on the complexity of parallel sorting. *Proc. 16th ACM Symposium on Theory of Computing*, pp. 71–80.

W. Lipski, Jr., and F. P. Preparata (1981). Segments, rectangles, contours. *J. Algorithms*, **2**, pp. 63–76.

D. Nath, S. N. Maheshwari, and P. C. P. Bhatt (1981). Parallel algorithms for the convex hull in two dimensions. *Proc. Conference on Analysis Problem Classes and Programming for Parallel Computing*, pp. 358–372.

J. O'Rourke, A. Aggarwal, S. Madilla, and M. Baldwin (1986). An optimal algorithm for finding minimal enclosing triangles. *J. Algorithms*, 7(2), 258–269.

M. H. Overmars and J. Van Leeuwen (1981). Maintenance of configurations in the plane. *J. Comput. Systems Sci.*, **23**, pp. 166–204.

F. Preparata and S. J. Hong (1977). Convex hulls of finite sets of points in two and three dimensions. *Comm. ACM*, **20**, pp. 87–93.

F. Preparata and M. I. Shamos (1985). *Computational Geometry: An Introduction.* Springer-Verlag, Berlin.

M. I. Shamos (1977). Computational geometry. Ph.D. Dissertation, Yale University.

M. I. Shamos and D. Hoey (1975). Closest point problems. *Proc. 16th IEEE Symposium on Foundations of Computing*, pp. 151–162.

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

R. E. Tarjan and U. Vishkin (1985). An efficient parallel biconnectivity algorithm. *SIAM J. Comput.*, **14**(4), pp. 862–874.

G. T. Toussaint (1983). Solving geometric problems using "rotating callipers." *Proc. IEEE Melecon '83.*

L. G. Valiant (1975). Parallelism in comparison problems. *SIAM J. Comput.*, **4**(3), pp. 348–355.

H. Wagener (1985). Parallel computational geometry using polygon ordering. Ph.D. Thesis, Technical University of Berlin.

H. Wagener (1987). Optimally parallel algorithms for convex hull determination (submitted).

C. K. Yap (1987). What can be parallelized in computational geometry? International Workshop on Parallel Algorithms and Architecture, Humboldt University, Berlin, DDR (invited talk). Proceedings to appear.