Algorithmica (1992) 7: 3–23

Algorithmica

Optimal Parallel Algorithms for Point-Set and Polygon Problems[1]

Richard Cole[2] and Michael T. Goodrich[3]

Abstract. In this paper we give parallel algorithms for a number of problems defined on point sets and polygons. All our algorithms have optimal $T(n) * P(n)$ products, where $T(n)$ is the time complexity and $P(n)$ is the number of processors used, and are for the EREW PRAM or CREW PRAM models. Our algorithms provide parallel analogues to well-known phenomena from sequential computational geometry, such as the fact that problems for polygons can oftentimes be solved more efficiently than point-set problems, and that nearest-neighbor problems can be solved without explicitly constructing a Voronoi diagram.

Key Words. Computational geometry, Parallel algorithms, Polygon, All nearest-neighbor problem, Kernel problem, Convex hull.

1. Introduction. We present a number of new algorithms for parallel computational geometry [1]–[4], [7], [11], [12]. the goal of this research is to find algorithms that run as fast as possible and are efficient in the following sense: if $P(n)$ is the processor complexity, $T(n)$ is the parallel time complexity, and $Seq(n)$ is the time complexity of the best-known sequential algorithm for the problem under consideration, then $T(n) * P(n) = O(Seq(n))$. If the product $T(n) * P(n)$ in fact achieves the sequential lower bound for the problem, then we say the algorithm is *optimal*. All our algorithms are optimal in this sense and are for the EREW or CREW PRAM models. The weaker of these two is the EREW PRAM model, the synchronous shared memory model in which simultaneous reads or writes are not allowed. The CREW PRAM allows for simultaneous reads. Specifically, our results are the following:

1. Constructing the convex hull of a set of n points in the plane in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM model.
2. Computing all nearest-neighbors for a set of n points in the plane in $O(\log n)$ time using $O(n)$ processors in the EREW PRAM model.
3. Computing all nearest-neighbors for the vertices of an n-vertex convex polygon in $O(\log n)$ time using $O(n/\log n)$ processors in the EREW PRAM model.

[1] The research of R. Cole was supported in part by NSF Grants CCR-8702271, CCR-8902221, and CCR-8906949, by ONR Grant N00014-85-K-0046, and by a John Simon Guggenheim Memorial Foundation fellowship. M. T. Goodrich's research was supported by the National Science Foundation under Grant CCR-8810568 and by the National Science Foundation and DARPA under Grant CCR-8908092.

[2] Courant Institute, New York University, New York, NY 10012, USA.

[3] Department of Computer Science, The Johns Hopkins University, Baltimore, MD 21218, USA.

Received April 10, 1988; revised May 2, 1989. Communicated by Alok Aggarwal.

4. Constructing the kernel of an n-vertex simple polygon in $O(\log n)$ time using $O(n/\log n)$ processors in the CREW PRAM model.

Before we give the methods for solving each of the above problems, let us review what was previously known about each of them and outline our contributions. Problem 1, the planar convex-hull problem, is perhaps the most-studied problem in computational geometry. There has been considerable previous work done on methods for solving this problem sequentially (see [9], [19], and [21]). It is well known, for example, that the sequential time complexity of this problem is $\Theta(n \log n)$. One of the most elegant of the optimal methods for constructing convex hulls is a divide-and-conquer algorithm based on Toussaint's idea of rotating a pair of "calipers" around two hulls to find their common supporting tangents [26]. Our algorithm is based on a generalization of the cascading divide-and-conquer technique [2] that provides a nontrivial parallel analogue to Toussaint's rotating-calipers paradigm. There are a number of other parallel algorithms for convex-hull construction that have the same bounds as our algorithm [1], [3], [4], [27]. They all assume, however, that the square-root function can be computed in $O(1)$ time. Our algorithm does not make this assumption.

Problem 2, the all nearest-neighbor problem, is also well known [9], [19], [21]. it can also be solved (optimally) in $O(n \log n)$ time, by a simple reduction to Voronoi diagram construction [23]. This reduction is not necessary, however, for we can still achieve $O(n \log n)$ time without reducing it to Voronoi diagram construction [5]. Our parallel algorithm for this problem shows that, just as in the sequential case, we can optimally solve the all-nearest neighbor problem without explicitly constructing a Voronoi diagram (for which the best-known parallel algorithm runs in nonoptimal $O(\log^2 n)$ time using $O(n)$ processors [1], [2]). As with our convex-hull algorithm, our method is based on a novel use of the cascading divide-and-conquer technique [2].

Finally, Problems 3 and 4 are significant, for they represent problems that have $\Omega(n \log n)$-time lower bounds in the general case, yet can be solved in $O(n)$ time if the input points come from the vertices of a polygon [10], [17], [18], [28]. Our algorithms for these problems provide parallel analogues to this phenomenon. Our algorithm for Problem 3 (all-nearest neighbors in a convex polygon) is based on a composite of parallel merging, parallel prefix, and broadcasting techniques. Our algorithm for Problem 4 (kernel of a simple polygon) is based on the discovery of a new way of characterizing the kernel of a simple polygon P in terms of the amount that the boundary of P "turns." Incidentally, this idea also leads to a new $O(n)$-time sequential algorithm.

In the remainder of this paper we present our algorithms, one per section, in the above order, and conclude with some final remarks and open problems in Section 6.

2. A Cascading Algorithm for Convex-Hull Construction. In this section we describe our algorithm for constructing the convex hull of a set of points in the plane. Our algorithm runs in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM model. There has been considerable previous work on this problem in the parallel setting, resulting in a number of algorithms running in $O(\log n)$ time using

$O(n)$ processors [1], [3], [4], [12], [27]. The algorithm we present in this section has the same complexity as these algorithms, but differs from them in that it is based on the elegant "rotating calipers" (or "merging slopes") technique of Toussaint [26], and in fact provides the first nontrivial parallel analogue to that technique. (By substituting known parallel merging methods for the sequential one used in the convex-hull algorithm by Toussaint, we can trivially get an algorithm running in $O(\log^2 n)$ time using $O(n/\log n)$ processors [6], [24].) Moreover, our algorithm makes no use of the square-root function, unlike the previous optimal parallel algorithms. Since our method for this problem (as well as that for the problem of the next section) depends on a generalization of the cascading divide-and-conquer technique, we begin our discussion by reviewing this technique.

2.1. A Review of Cascading Divide-and-Conquer. Since the cascading divide-and-conquer method was originally used to sort [8], let us review this technique in the context of the sorting problem: we are given a set S of n elements taken from some total order, and wish to list the elements of S in nondecreasing order.

Let T be a complete binary tree with the elements of S stored at its leaves, one element per leaf. Intuitively, T represents the divide-and-conquer structure of the merge sort algorithm. The method proceeds in stages. At the end of each stage t, each node v in T stores a sorted list $A_t(v)$, whose elements form a subset of $Desc(v)$, the elements that are stored at descendents of v. In stage $t+1$, for each node v whose $A_t(v)$ list does not contain all the elements in $Desc(v)$, we construct a new list $A_{t+1}(v)$, which contains roughly double the number of elements in $A_t(v)$. If $A_t(v)$ contains all the elements in $Desc(v)$, then v is said to be *full*. Specifically, $A_{t+1}(v)$ is defined as follows:

$$A_{t+1}(v) = SAMP(A_t(u)) \cup SAMP(A_t(w)),$$

where u and w are the children of v, and $SAMP(A)$ denotes the *sample* of A. If u was not full at the end of stage $t-1$, then $SAMP(A_t(u))$ consists of every fourth element from $A_t(u)$; if u just became full at the end of stage $t-1$, then $SAMP(A_t(u))$ consists of every other element from $A_t(u)$; and if u was full at the end of stage $t-2$, then $SAMP(A_t(v)) = A_t(V)$. Similarly, for $SAMP(A_t(w))$. Thus, once the nodes on a certain level of T become full, the nodes on the next level up become full three stages later. Therefore, the entire algorithm requires $3\lceil \log n \rceil$ stages.

Cole [8] shows that a *rank* label for each element e of $A_{t-1}(v)$ can be maintained that gives the rank of e's predecessor in $A_t(v)$, as well as similar labels from $A_t(v)$ to $SAMP(A_{t-1}(u))$ and $SAMP(A_{t-1}(w))$. Moreover, he shows that these labels can be used to perform the merge at node v for stage $t+1$ in $O(1)$ time using $O(|A_{t+1}(v)|)$ processors in the CREW PRAM model, assuming that $A_{t-1}(v)$ is a "good approximation" of $A_t(v)$. More precisely, $A_{t-1}(v)$ must be a *c-cover* of $A_t(v)$; that is, for each consecutive pair of elements e and f in $A_{t-1}(v) \cup \{-\infty, \infty\}$ there can be at most c elements of $A_t(v)$ that fall in the interval $[e, f)$. Since there are $O(\log n)$ stages, and we need only perform this computation for nonfull nodes, this implies that we can sort in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM model. Cole also shows that this approach can be modified so as to run

on the EREW PRAM model (we refer the interested reader to [8] for the details of this modification).

Since T is complete, the processor allocation is quite simple. In [2] Atallah *et al.* generalize Cole's method by showing that the processor allocation for cascade merging can be performed even when the tree T is not complete. Moreover, they show that there are a number of other computations that can be performed at v during a stage—such as applying a monotone function to the elements of $A_t(v)$ before "sending" then to v's parent, or computing other labeling functions for the elements in $A_{t+1}(v)$—all without sacrificing the $O(1)$ time performance for the stage. They also give a number of examples of problems that can be solved using these generalizations. One generalization that their techniques seems not to be able to handle, however, is that of inserting and deleting elements from $A_t(v)$ before sending $SAMP(A_t(v))$ to v's parent. In the next subsection we show that this difficulty for the insertions and deletions that arise from applying the cascade merging technique to convex-hull construction can be overcome by "rotating calipers" [26].

2.2. Our Convex-Hull Algorithm. Given a set of n points in the plane, the convex-hull problem is to construct a representation of the smallest convex set (a polygon) that contains all these points. This problem can be divided in two by considering the boundary of the convex hull to consist of two pieces, an upper hull and a lower hull, the upper hull being that piece visible from above (i.e., from the point $(0, +\infty)$) and the lower hull being that piece visible from below.

Consider the problem of constructing the upper hull of a set of n points (the problem of constructing the lower hull is similar). For simplicity we assume that the points have distinct x-coordinates, that no three are collinear, and that the number of points is a power of two (it is straightforward to modify our algorithm for the general case). Our algorithm is based on the following divide-and-conquer approach [26]: Suppose we have two disjoint upper hulls separated by a vertical line, and the edges of each hull are given sorted by slope. In order to find their common tangent we simply merge the two lists of edges by slope. Suppose edges e and g come from the left hull and edge f from the right, in the order $e > f > g$ (by slopes). If the line containing f is below the common vertex of e and g, then f cannot be on the hull. In fact, this rule eliminates exactly those edges that do not belong on the hull, and leaves two contiguous lists of edges on each side of the dividing line. The common upper tangent is determined by creating an edge that joins the rightmost vertex of the "surviving" hull on the left to the leftmost vertex of the surviving hull on the right.

To do this by cascade merging we begin by sorting the input points by x-coordinates; let $S = (q_1, q_2, \ldots, q_n)$ denote this list. We construct a complete binary tree T such that each leaf node v_i contains the list

$$H_0(v_i) = (\langle (x(q_{2i-1}), -\infty), q_{2i-1}\rangle, \langle q_{2i-1}, q_{2i}\rangle, \langle q_{2i}, (x(q_{2i}), -\infty)\rangle),$$

for $i = 1, 2, \ldots, n/2$, where $\langle p, q\rangle$ denotes the line segment from p to q. In other words, $H_0(v_i)$ is the upper hull determined by the edge $\langle q_{2i-1}, q_{2i}\rangle$. Our algorithm proceeds in stages, where in each stage $t + 1$, we construct $H_{t+1}(v)$, a sorted list

of edges, for each node v, where $H_{t+1}(v)$ is defined as follows:

$$H_{t+1}(v) = SAMP(H_t(u)) \cup SAMP(H_t(w)),$$

where u is v's left child and w is v's right child, and $SAMP$ is defined as in Section 2.1. Thus, the entire method requires $3\lceil \log n \rceil$ stages.

Ultimately, when a node v becomes full, we would like $H_{t+1}(v)$ to contain the edges of the upper hull of the edges stored in $Desc(v)$ sorted by slopes. Our current definition of $H_{t+1}(v)$ does not provide this, however, for it does nothing more than merge all the edges stored in the children of v. We modify the definition slightly, then, so that at the moment when a node v becomes full, we perform some extra computations to make $H_{t+1}(v)$ be the upper hull of all the edges stored in $Desc(v)$. In particular, suppose v has become full at the end of stage $t+1$ (i.e., we have just constructed $H_{t+1}(v) = H_t(u) \cup H_t(w)$). Inductively, also suppose $H_t(u)$ (resp. $H_t(w)$) is the upper hull for u (resp. w) stored by sorted slopes. By merging these two lists we can determine which elements of $H_t(u)$ and $H_t(w)$, respectively, belong to the hull of $H_t(u) \cup H_t(w)$, using the rule given above. Note that the elements of $H_t(u)$ that are on the hull of $H_t(u) \cup H_t(w)$ form a prefix of $H_t(u)$, and the elements of $H_t(w)$ that are on the hull of $H_t(u) \cup H_t(w)$ form a suffix of $H_t(w)$. In addition, the common upper tangent of $H_t(u)$ and $H_t(w)$ (call it L) must join the last surviving member of $H_t(u)$ (call it f) to the first surviving member of $H_t(w)$ (call it g). Thus, to construct a representation of the upper hull of $H_t(u) \cup H_t(w)$ we must concatenate the surviving prefix of $H_t(u)$ with L and the surviving suffix of $H_t(w)$. This is the list we use for $H_{t+2}(v)$ (when the $SAMP$ function for v's parent takes every other element).

In order to show that we can incorporate this computation into the cascading divide-and-conquer method, we must show that for each element e in $H_{t+1}(v)$ we can compute the predecessor of e in $H_{t+2}(v)$ in $O(1)$ time using a processor for each e, and that our definition of $H_{t+2}(v)$ does not violate the c-cover property (i.e., that $H_t(z)$ is a c-cover of $H_{t+1}(z)$, for all $z \in T$). For the predecessor computation, note that since $H_t(u)$ and $H_t(w)$ both contain an edge with slope $+\infty$ and an edge with slope $-\infty$, the surviving prefix of $H_t(u)$ and the surviving suffix of $H_t(w)$ are nonnull. Thus, f and g exist. Let $H'_t(u)$ denote the list $H_t(u)$ with the successor of e (which also must exist) replaced by L and every edge after this edge removed. Similarly, let $H'_t(w)$ denote the list $H_t(w)$ with every edge before g removed. We can easily construct $H'_t(u)$ and $H'_t(w)$ in constant time using $O(|H_t(u)| + |H_t(w)|)$ processors. Moreover, given an element e in $H_{t+1}(v)$, since we know e's predecessor in $H_t(u)$ and $H_t(w)$, this immediately gives us e's predecessor in $H'_t(u)$ and $H'_t(w)$. By adding the ranks of these two elements we get the rank of e in $H_{t+2}(v)$ (assuming e is a survivor). Thus, we have yet only to show that we maintain the c-cover property.

Since v is full after stage $t+1$, we know that $H_{t+1}(v)$ would be a 1-cover of $H_{t+2}(v)$ (in fact, $H_{t+1}(v)$ would equal $H_{t+2}(v)$) were it not the case that we are deleting elements from $H_t(u)$ and $H_t(w)$, and then adding an edge L to $H_t(u)$, in order to create $H'_t(u)$ and $H'_t(w)$, and, subsequently, $H_{t+2}(v)$. To see that we still have a c-cover property, however, first note that $H_t(u)$ is a 2-cover for $H'_t(u)$, since

in creating $H'_t(u)$ from $H_t(u)$ we deleted a contiguous block of elements from $H_t(u)$ and then added one additional edge. Also note that $H_t(w)$ is trivially a 1-cover for $H'_t(w)$, since $H'_t(w) \subset H_t(w)$. Thus, by transitivity, $H_{t+1}(v)$ is a 2-cover for $H_{t+2}(v)$. Since the c used in [2] is greater than 2, this is sufficient to maintain the c-cover property of the cascade merging method. Therefore, we have the following theorem:

THEOREM 2.1. *Given a set S of n points in the plane we can construct the convex hull of S in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM model without using the square-root function.*

3. All Nearest-Neighbors for a Point Set. Given a set S of n points in the plane, the problem is to find for each point q in S the point $q' \neq q$ in S that is closest to q. Our algorithm runs in $O(\log n)$ time using $O(n)$ processors. We first describe how to implement our algorithm in $O(n \log n)$ space in the EREW PRAM model, and then how it can be implemented in $O(n)$ space in the CREW PRAM model.

Our algorithm is based on two nontrivial applications of the cascading divide-and-conquer technique [2], each constituting a phase in our algorithm. In Phase 1 we determine, for each $q \in S$, an approximation to $N(q)$, the nearest-neighbor ball around q, and in Phase 2 we use these approximations to compute all the true $N(q)$'s in parallel. Specifically, in Phase 1 we determine a ball around each q, which we call the *neighborhood ball* about q, whose radius is the distance between q and the closest point q has "encountered" during the cascading merge procedure. During the second phase we construct for each point q a list $C(q)$ that contains points of S that may have q as their nearest neighbor. We call $C(q)$ the *candidate list* for q. It is easy to show that for any point q there can be at most six other points q' such that q is the nearest neighbor of q' (using an argument similar to that used by Bentley and Shamos in [5]). Thus, $C(q)$ need never contain more than six points. Our algorithm constructs a $C(q)$ list for each q in S and then performs a postprocessing step to eliminate any pairs that are not nearest-neighbor pairs. The details follow.

In Phase 1 we construct, for each q in S, the neighborhood ball centered at q, denoted $B(q)$. For simplicity let us assume that the points have distinct x- and y-coordinates, respectively; we can easily modify our algorithm for the general case. We begin by sorting the points in S into increasing order by x-coordinates; let $S = (q_1, q_2, \ldots, q_n)$ denote this list. This can be done in $O(\log n)$ time using $O(n)$ processors in the EREW PRAM model [8]. We then build a complete binary tree T that has the points $q_1, q_2, \ldots, q_n$ stored in its leaves (listed from left to right), one per leaf. For each node v in T let $Y(v)$ denote the points of $Desc(v)$ (the points stored in descendents of v) sorted by y-coordinates, and let $depth(v)$ denote the depth of v (with the root being at depth 0). With each point q in $Y(v)$ we store a label $b(q)$. At the end of Phase 1 the label $b(q)$ will store the name of the point that is closest to q of all the points that q has "encountered" during the phase.

We can perform a cascading method to construct $Y(v)$ for each v in T by defining

$Y_{t+1}(v)$ for stage $t+1$ as

$$Y_{t+1}(v) = SAMP(Y_t(u)) \cup SAMP(Y_t(w)),$$

where u is v's left child, w is v's right child, and the sample function $SAMP$ is defined as in Section 2.1. Thus, all the $Y(v)$ lists can be constructed in $O(\log n)$ time and $O(n)$ space using $O(n)$ processors in the EREW PRAM model [8].

For each q in each $Y(v)$ we use the ranking information computed in the construction of $Y(v)$ from $Y(u)$ and $Y(w)$ to build the set $\{pred(q, Y(u)), succ(q, Y(u)), pred(q, Y(w)), succ(q, Y(w))\}$, where $pred(p, A)$ (resp. $succ(p, A)$) denotes the predecessor (resp. successor) of p in A using the order on A. This defines $O(\log n)$ points for each q. These are the points that q "encountered" during the cascading merge. For each q in parallel we can then compute the closest of these $O(\log n)$ points to q in $O(\log n)$ serial time. This is $b(q)$.

Alternatively, we can compute the $b(q)$ labels while performing the cascading construction of $Y(v)$. For each leaf node v, which, say, stores the point q_i, we initialize $b(q_i)$ to be the point in $\{q_{i-1}, q_{i+1}\}$ that is closer to q_i. When a node v becomes full, i.e., when $Y(v)$ contains all the elements in $Desc(v)$, then we can update $b(q)$ for each q in $Y(v)$ by taking $b(q)$ for each $q \in Y(v)$ to be the closer of the old value of $b(q)$ and the point in $\{pred(q, Y(u)), succ(q, Y(u)), pred(q, Y(w)), succ(q, Y(w))\}$ closest to q. After updating the value of $b(q)$, we can then de-allocate the space for $Y(v)$, since we no longer need it, thus giving us $O(n)$ space for the entire computation (in the CREW PRAM model). When the cascading completes we will have computed $b(q)$ for each point q in S. So, after Phase 1 completes, we have the following:

LEMMA 3.1 *Given a list of points $S = (q_1, q_2, \ldots, q_n)$ the label $b(q_i)$ can be computed for each point $q_i \in S$ in $O(\log n)$ time and $O(n)$ space using $O(n)$ processors in the CREW PRAM model, or in $O(\log n)$ time and $O(n \log n)$ space using $O(n)$ processors in the EREW PRAM model.*

We define $B(q)$, the *neighborhood* ball centered at q, to be the region in $\Re^2$ consisting of all points q' such that $d(q, q') \leq d(q, b(q))$. In Phase 2 we refine each $B(q)$ into $N(q)$, the nearest-neighbor ball centered at q. Since the points in S all have distinct x-coordinates, we can partition the leaves of T by placing a vertical dividing line between q_i and q_{i+1} for $i = 1, 2, \ldots, n-1$. With each node v in T we associate a slab Π_v, which is the region bounded by the two vertical dividing lines that separate the points in $Desc(v)$ from the rest of the points in S. For each node v in T let $left(v)$ (resp., $right(v)$) denote the left (resp. right) vertical boundary of the slab Π_v. We define the following lists for each node $v \in T$:

$$L(v) = \{q \in Y(v): B(q) \cap left(v) \neq \varnothing\},$$

$$R(v) = \{q \in Y(v): B(q) \cap right(v) \neq \varnothing\}.$$

That is, $L(v)$ (resp. $R(v)$) consists of the points whose neighborhood ball intersects

the left (resp. right) boundary of the slab Π_v. Our method for refining the $B(q)$'s into $N(q)$'s (i.e., Phase 2) involves a second application of the cascading divide-and-conquer method. In this second cascade-merging procedure we not only compute $Y(v)$ for each node v but also $L(v)$ and $R(v)$, all sorted by increasing y-coordinates. Note, however, that $L(v)$ and $R(v)$ may be proper subsets of $Y(v)$, so the cascade will have to involve deletions. We first describe how to do this using $O(n \log n)$ space, and then show how this can be reduced to $O(n)$ space.

For each v we construct $Y(v)$ as in Phase 1, but do not de-allocate the space for $Y(v)$. From each $Y(v)$ construct $L(v)$ and $R(v)$, and for each element q in $Y(z)$, where z is the sibling of v, compute the predecessor of q in $L(v)$ and $R(v)$. All of these computations can be done in $O(\log n)$ time using $O(|Y(v)|/\log n)$ processors by simple parallel prefix and parallel broadcast computations [15], [16]. (Recall that the parallel prefix technique is to reduce a problem to the problem of computing all the prefix sums $c_k = \sum_{i=1}^{k} a_i$ of a sequence $(a_1, a_2, \ldots, a_m)$, where the $+$ operation is associative.) Let $SW(q)$ denote the region of $\mathfrak{R}^2$ consisting of all points q' such that $x(q') < x(q)$ and $y(q') < y(q)$, i.e., all points southwest of q. Define $SE(q)$, $NW(q)$, and $NE(q)$ similarly. For each point q in $Y(v)$ we define four pointers (labels):

$$sw(q) = \text{point w/max. } y\text{-coor. in } SW(q) \cap Y(v),$$

$$se(q) = \text{point w/max. } y\text{-coor. in } SE(q) \cap Y(v),$$

$$nw(q) = \text{point w/min. } y\text{-coor. in } NW(q) \cap Y(v),$$

$$ne(q) = \text{point w/min. } y\text{-coor. in } NE(q) \cap Y(v).$$

These labels can be constructed while we are building the $Y(v)$ lists. For example, we can compute $sw(q)$ for $q \in Y(v)$ from the labels for the elements in $Y(u)$ and $Y(w)$ (u being v's left child and w being v's right child) by the following simple rule: if q comes from $Y(u)$, then do nothing, for $sw(q)$ is the same in $Y(v)$ as it is in $Y(u)$; if, on the other hand, q comes from $Y(w)$, then take $sw(q)$ in $Y(v)$ to be the point with maximum y-coordinate between the $sw(q)$ point for q from $Y(w)$ and $pred(q, Y(u))$. The other labels can be maintained similarly. We use these labels and the L and R lists to compute the candidate list $C(q)$ for each point q. Specifically, the construction of $C(q)$ is performed in a bottom-up fashion in T using the observations from the following lemma:

LEMMA 3.2. *Let v be a node in T with left child u and right child w and let q be a point in $Y(v)$. If $q \in Y(u)$, then the only points in $Y(w)$ such that q could possibly be their nearest-neighbor are $pred(q, L(w))$, $sw(pred(q, L(w)))$, $succ(q, L(w))$, and $nw(succ(q, L(w)))$. (See Figure 1.) If $q \in Y(w)$, then the only points in $Y(w)$ such that q could possibly be their nearest-neighbor are $pred(q, R(u))$, $se(pred(q, R(u)))$, $succ(q, R(u))$, and $ne(succ(q, R(u)))$.*

PROOF. Without loss of generality we prove that the only p's in $Y(w)$ with $y(p) \geq y(q)$ such that q could be p's nearest neighbor are $p = succ(q, L(w))$ and $p = nw(succ(q, L(w)))$. Let l be the vertical line separating $Y(u)$ and $Y(w)$ and let

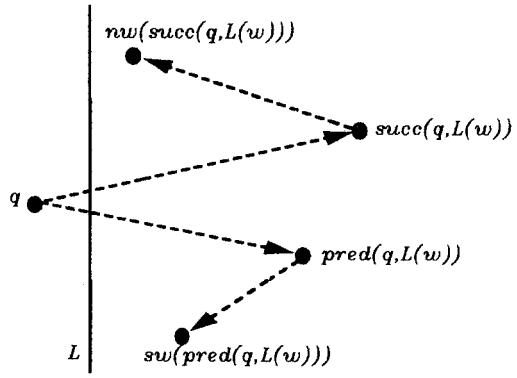

Fig. 1. The only points in $Y(w)$ that can have $q \in Y(u)$ as their nearest neighbor.

the origin, denoted o, be placed at the intersection of l and the horizontal line containing $succ(q, L(w))$. Furthermore, let $p_0 = (x_0, y_0) = (x_0, 0) = succ(q, L(w))$ and $p_1 = (x_1, y_1) = nw(succ(q, L(w)))$. Since p_0 is in $L(w)$ by definition, $B(p_0)$ contains the origin. In addition, the radius of $B(p_0)$ is at most $d(p_0, p_1)$, since p_1 must have been one of the points encountered by p_0 during Phase 1. Thus, $x_1 \leq y_1$. Suppose, for the sake of contradiction, that there is a point $p_2 = (x_2, y_2)$ in $Y(w)$, with $y_2 > y_0$, such that the circle C centered at p_2 with radius $\min\{d(p_2, p_1), d(p_2, p_0)\}$ contains the origin o. Note that these conditions are necessary for the nearest-neighbor ball for p_2 (with respect to $Y(w)$) to contain q. Since C contains the origin and does not contain $p_0 = (x_0, 0)$ $x_2 < x_0$ (recall that p_2 is above p_0 by assumption). Then $y_2 > y_1$, since p_1 is the point with smallest y-coordinate in $Y(w)$ of all points northwest of p_0. Thus, $x_1 < y_2$. Therefore,

$$\begin{aligned} d(p_2, p_1) &= \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \\ &= \sqrt{(x_1^2 + y_1^2 - 2y_1y_2) + (x_2^2 + y_2^2 - 2x_1x_2)} < \sqrt{x_2^2 + y_2^2} \\ &= d(p_2, o). \end{aligned}$$

But this contradicts the definition of C, for it states that if C is a circle centered at $p_2 = (x_2, y_2)$, with $x_2 > 0$ and $y_2 > 0$, such that C does not contain a point p_1 in the triangle $\langle (0, 0), (0, y_2), (y_2, y_2) \rangle$, then C cannot contain the origin. (See Figure 2.) □

Thus, for each point q in $Y(v)$, if q comes from $Y(w)$, we can compute the new list $C(q)$ at v given the old list $C(q)$ at w and at most four points in $Y(u)$. (The definition is similar if q comes from $Y(u)$.) Since the old $C(q)$ list can contain at most six points, there can be at most a total of ten points in this collection, from which we must determine which ones can possibly have q as their nearest neighbor. These points can be determined by solving the all-nearest neighbor problem for this collection of at most ten points with a single processor in $O(1)$ steps. Thus, we have the following:

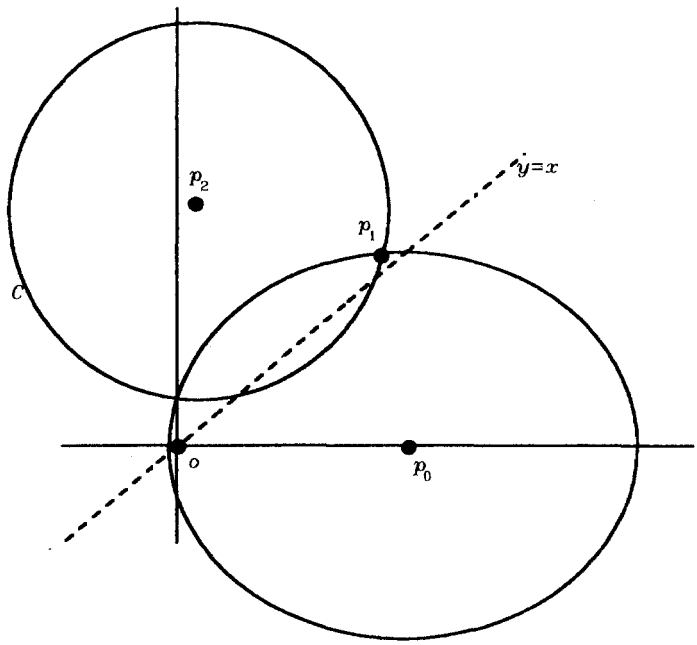

Fig. 2. C cannot contain o while excluding p_1.

LEMMA 3.3. *Given a set $S = \{q_1, q_2, \ldots, q_n\}$ of points in the plane, we can compute $C(q_i)$ for each q_i in $O(\log n)$ time and $O(n \log n)$ space using $O(n)$ processors in the EREW PRAM model.*

We complete our all nearest-neighbors algorithm with a simple postprocessing step. Let N be the set of all pairs (q, q') such that $q \in C(q')$. Since $|N| \leq 6n$ we can sort the pairs in N by first coordinate in $O(\log n)$ time and $O(n)$ space using $O(n)$ processors in the EREW PRAM model. We complete the algorithm by performing a simple bottom-up minimum-finding computation for each q to compute from all pairs (q, q') in N the point q' that is closest to q. Thus, we have the following:

LEMMA 3.4. *Given a set S of n points in the plane we can compute the nearest-neighbor in S of each point in S in $O(\log n)$ time and $O(n \log n)$ space using $O(n)$ processors in the EREW PRAM model.*

If we wish to perform this computation in $O(n)$ space we have to be more clever in how we compute the $L(v)$ and $R(v)$ lists. The main idea of our method is to construct the $L(v)$ and $R(v)$ lists in a cascading fashion (along with $Y(v)$). The following observation is central to our method:

OBSERVATION 3.5. *Let a point q in S be given, and let $(v_d, v_{d-1}, \ldots, v_0)$ be the leaf-to-root path from the leaf that contains q, i.e., $q \in Y(v_i)$ for $d \geq i \geq 0$. Then there is a threshold index l such that q is in $L(v_i)$ for $d \geq i \geq l$ but q is not in $L(v_i)$ for*

$l > i \geq 0$. Similarly, there is a threshold index r such that q is in $R(v_i)$ for $d \geq i \geq r$ but q is not in $R(v_i)$ for $r > i \geq 0$.

PROOF. The proof follows from the fact that the vertical slabs $\Pi_{v_d}, \Pi_{v_{d-1}}, \ldots, \Pi_{v_0}$ are nested one inside the next. □

Let us concentrate on how to compute the $L(v)$'s in a cascading fashion; the method for the $R(v)$'s is similar. We begin by computing for each q the threshold value l where $q \in L(v_l)$ but $q \notin L(v_{l-1})$. For each point q we can compute its threshold value by performing a search up the tree from the leaf q is stored in, traversing from a node to its parent until $B(q)$ no longer intersects the left boundary of the slab for that node. Let $D(v)$ be the list of all points in $Y(v)$ whose threshold l is equal to the depth of v. Note that we can compute *a priori* the elements belonging to each $D(w)$. This gives us the following recursive definition of $L(v)$:

$$L(v) = L(u) \cup (L(w) - D(w)),$$

where u and w are respectively the left and right children of v. Intuitively, the points in $D(v)$ "die" at the node v and do not cascade any higher in T. We perform the deletions from $L(w)$ similarly to how we performed the deletions in the "convex-hull" section (Section 2). Specifically, we construct a new tree T'_1 from T by taking each right child w and replacing w with a new node w' that has w as its right child. From the left child of w' we hang a complete balanced binary tree having the elements of $D(w)$ as its children. (See Figure 3.) We color each new node w' "blue" and all other nodes in T'_1 are colored "red." Intuitively, w' will correspond to the $(L(w) - D(w))$ term. Given the tree T'_1 we define $L'(v)$ for each v in T'_1 as

$$L'(v) = L'(u) \cup L'(w),$$

where u and w are respectively the left and right children of v. When a node v becomes full, i.e., it contains all the elements in $Desc(v)$, the way we proceed depends on the color of v. If v is blue, then v corresponds to a deletion, so we delete from

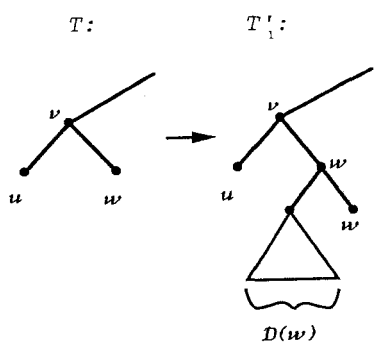

Fig. 3. Constructing T'_1 from T.

$L'(v)$ every element that also appears in $L'(u)$ before we pass the elements in $L'(v)$ to v's parent. Since v is full, this amounts to a simple compression computation: namely, given a processor for each element q of $L'(v)$, if q is in $L'(u)$, then we do nothing; otherwise, we compute q's new rank in $L'(v)$ by taking q's old rank in $L'(v)$ and subtracting two times its rank in $L'(u)$. Note that we must subtract two times q's rank in $L'(u)$ when q is not in $L'(u)$, since each element of $L'(u)$ appears twice in $L'(v)$. Also note that these deletions do not affect our ability to perform each stage in $O(1)$ time, since the set $L'(v)$ for any stage t will still be a 1-cover of $L'(v)$ for stage $t+1$ (since we are not inserting new elements, only deleting some elements). If v is red, then v corresponds to one of the original nodes in T. In this case, since v is now full, we have just constructed $L(v)$ for that node. Thus, we can implement each stage in $O(1)$ time; hence, construct all the $L(v)$'s in $O(\log n)$ time using $O(n)$ processors. Let us explain, then, how we coordinate the construction of $Y(v)$ and $R(v)$ with this cascading procedure.

We perform the cascading merge constructions of $Y(v)$, $L(v)$, and $R(v)$ simultaneously. The construction of $R(v)$ is entirely symmetric to that of $L(v)$, taking place in a tree T'_2. The computation of $Y(v)$ is tightly coupled with that of $L(v)$ and $R(v)$ in the sense that during each stage t the processors assigned to a node v in T for constructing $Y(v)$ are also the processors assigned to the corresponding nodes in T'_1 and T'_2. Since each level of T corresponds to two levels in T'_1 and T'_2, we must take care that the cascading merge in T runs at half the speed as those in T'_1 and T'_2. In addition, for each q in $L(v)$ (resp. $R(v)$), we maintain the rank of q in $Y(v)$, and for each q in $Y(v)$ we maintain the rank of q in $L(v)$ and $R(v)$. Using the merging methods of Cole [8], these ranks can be maintained during the cascading merge while still performing each stage in $O(1)$ time.

When a node v in T becomes full (along with its corresponding red nodes in T'_1 and T'_2), we update the *sw*, *se*, *nw*, and *ne* labels for each q in $Y(v)$, compute the new candidates for $C(q)$ using these labels (as described in Lemma 3.2), and determine which (at most six) elements may remain in $C(q)$, as described above. At this point we no longer need the space allocated to v in T, nor the space for v's corresponding nodes in T'_1 and T'_2. When the cascading computation completes we will have $C(q)$ for each q in S just as in the solution that required $O(n \log n)$ space. Thus, we can apply the postprocessing step just as before to complete the computation. We summarize this section in the following theorem:

THEOREM 3.6. *Given a set S of n points in the plane we can compute the nearest-neighbor in S of each point in S in $O(\log n)$ time and $O(n)$ space using $O(n)$ processors in the CREW PRAM model, or, alternately, in $O(n \log n)$ space using $O(n)$ processors in the EREW PRAM model.*

In the next section we show how to solve this same problem for the vertices of a convex polygon.

4. All-Nearest-Neighbor Problem for a Convex Polygon. In this section we show how to find the nearest-neighbor vertex of each vertex on a convex polygon in $O(\log n)$ time using $O(n/\log n)$ processors in the EREW PRAM model.

Let $P = (v_1, v_2, \ldots, v_n)$ be the clockwise listing of the vertices of a convex polygon. A polygonal chain C has the *semicircle* property if when v_i and v_j are a farthest pair of vertices in C, then all the vertices of C are contained in a circle with diameter $d(v_i, v_j)$. Let v_a and v_c be a farthest pair of vertices of P, and let v_b (resp. v_d) be a vertex that is farthest to the left (resp. right) of the line (v_a, v_c). Lee and Preparata [17] show that the vertices v_a, v_b, v_c, and v_d partition P into four polygonal chains $C_1 = (v_a, \ldots, v_b)$, $C_2 = (v_b, \ldots, v_c)$, $C_3 = (v_c, \ldots, v_d)$, and $C_4 = (v_d, \ldots, v_a)$, such that each chain has the semicircle property. They also show that the nearest-neighbor vertex in C_i of any vertex v_j in C_i is either v_{j-1} or v_{j+1}.

We can determine v_a and v_c by using parallel merging [6], [24] to implement the algorithm of Shamos [22] in $O(\log n)$ time using $O(n/\log n)$ processors [11]. Specifically, we can think of the edges of P as vectors and translate them to the origin. Then the region between two vectors corresponds to a vertex of P, and two regions that are cut by the same line correspond to antipodal vertices on P (see Figure 4). All such pairs can then be enumerated by rotating all the vectors below the x-axis by π radians and merging this list with the vectors above the x-axis. It is then an easy matter to find a farthest pair among this list of pairs.

Given v_a and v_c, it is also easy to find the vertices v_b and v_d in $O(\log n)$ time using $O(n/\log n)$ processors by a simple maximum-finding algorithm. We can then solve the all-nearest-neighbor problem for each of the polygonal chains in $O(\log n)$ time using $O(n/\log n)$ processors, since the nearest-neighbor vertex in the chain C_i of each vertex v_j in C_i is either v_{j-1} or v_{j+1} [17]. Incidentally, there are other choices we could have made for v_a, v_b, v_c, and v_d [10], [28], which are even simpler to compute and would still satisfy the semicircle property. We choose these vertices as above, for it does not affect the efficiency of our method and computing the diameter of a convex polygon may be of independent interest.

The rest of the computation is as follows: we first "merge" the subproblem solutions to C_1 and C_2 (resp. C_3 and C_4), and then merge the two subproblem solutions separated by the line (v_a, v_c).

Let us concentrate on the generic merge step. We are given two sets of points S_1 and S_2 separated by a line L such that we have solved the all-nearest-neighbor

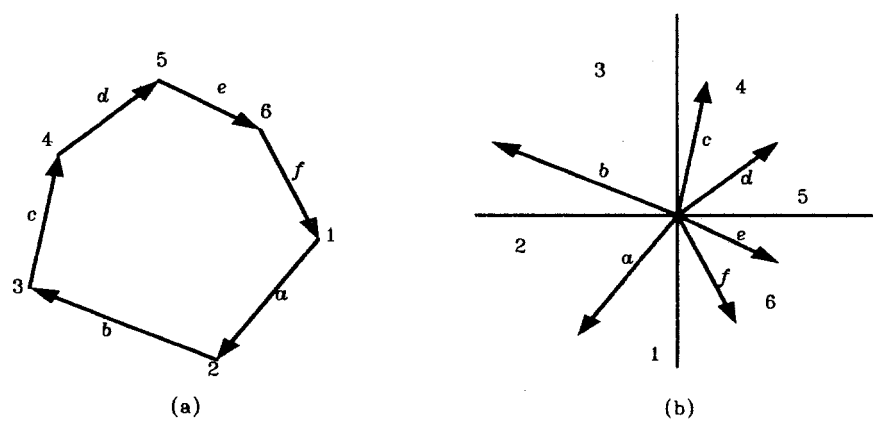

Fig. 4. Translating edges to the origin-like vectors.

problem for each set. In addition, we are given S_1 and S_2 listed in sorted order by the elements' projections along L. Without loss of generality we assume that L is a vertical line and the points in S_1 and S_2 are listed by nondecreasing y-coordinates. For simplicity we also assume the y-coordinates are distinct; our results are easily modified for the general case.

Let $d_i(p)$ denote the distance from a point p to its nearest-neighbor in S_i, and let $N_i(p)$ denote the $d_i(p)$-ball centered at p. It is known [5] that each point on L can intersect at most four $N_i(p)$'s for any $i \in \{1, 2\}$. Since we assumed that the all-nearest-neighbor problem has already been solved for S_1 and S_2, we can construct, for $i \in \{1, 2\}$, the sorted list S_i' that consists of all the points in S_i whose $d_i(p)$-ball intersects L by compressing out all the points whose $d_i(p)$-ball does not intersect L. This can be done in $O(\log n)$ time using $O(n/\log n)$ processors in the EREW PRAM model by a parallel prefix computation [15], [16].

Let us concentrate on the problem of combining S_1 with S_2'; the method for combining S_1' with S_2 is similar. We need to find for each point q in S_2' a point p in S_1 such that p is the closest of all points contained in $N_2(q)$, if it exists. We begin by merging the list S_1 with the list S_2'. For each p in S_1, this gives us the predecessor of p in S_2', which we denote by $pred(p, S_2')$. Since any point on L intersects at most four $N_2(q)$'s, any point p in S_1 intersects at most four $N_2(q)$'s as well. Moreover, the only q's in S_2' whose $d_2(q)$-ball could possibly contain p must be within four positions of $pred(p, S_2')$ in S_2'. If we had $O(n)$ processors at our disposal and we were working in the CREW (concurrent-read) PRAM model, it would be a simple matter to complete this merging procedure. But using only $O(n/\log n)$ processors in the EREW PRAM model it must be a little more involved, because for any point q in S_2' there may be many p's in S_1 with which we wish to compare q.

Recall that for each point p we wish to examine up to eight points in S_2'. Our computation consists of eight rounds, where in each round we examine one of the eight points in S_2' for each p in S_1. For each $p \in S_1$ we examine the points in S_2' associated with p in order by increasing y-coordinates. We also maintain a label $closest(q)$ for each point $q \in S_2'$, which identifies the point p in S_1 that is closest to q from all points in S_1 compared with q so far. Initially, $closest(q)$ is ∞ for each q in S_2'.

Let us concentrate on the computation for a single round. Let S_q denote the set of all points p in S_1 such that q is the point in S_2' we wish to examine for p in this round. Since we examine the points in S_2' associated with each p in S_1 by increasing y-coordinates, the points in S_q comprise a contiguous subarray of S_1. Thus, we can use a parallel prefix computation to determine the subarray S_q in S_1 for each q in S_2 (some S_q's may be empty) in $O(\log n)$ time using $O(n/\log n)$ processors. We can then perform a broadcast and find-minimum operation to find a point in S_q that is closest to q. We then let $closest(q)$ be the closer of this point and the previous $closest(q)$ value. This broadcast and minimum-finding step can also be performed in $O(\log n)$ time using $O(n/\log n)$ processors, and completes the computation for this round. When the eight rounds have completed, we will have solved the all-nearest-neighbor problem for each q in S_2, since we will have compared q with all points p in S_1 that are contained in $N_2(q)$ (recall that if

$q \in S_2 - S_2'$, then this is true vacuously). We then repeat this procedure to solve the all-nearest-neighbor problem for each p in S_1, by merging S_1' with S_2. Thus, we have established the following:

THEOREM 4.1. *Given a convex polygon P the nearest-neighbor vertex of each vertex on P can be determined in $O(\log n)$ time using $O(n/\log n)$ processors in the EREW PRAM model, which is optimal.*

The final problem we address is also a polygon problem.

5. Kernel of a Simple Polygon. Let $P = (e_0, e_1, \ldots, e_{n-1})$ be a listing of the edges of a simple polygon P (with e_0 and e_{n-1} sharing a common endpoint). We consider each edge of P to be an oriented line segment such that the interior of P is on its left. We let $H(e_i)$ denote the half-plane to the left of the line containing the edge e_i. Given any list Q of oriented edges $e_0, \ldots, e_{m-1}$, we define the *kernel of* Q, denoted $K(Q)$, to be the intersection of all the half-planes determined by the edges in Q, i.e., $K(Q) = \bigcap_{i=0}^{m-1} H(e_i)$. Our problem is the following: given an oriented simple polygon P, construct $K(P)$. This can be solved sequentially in $O(n)$ time [18].

Wagener [27] has shown that the convex hull of a simple polygon can be constructed in $O(\log n)$ time using $O(n/\log n)$ processors in the CREW PRAM model. Since the common intersection of n half-planes can be computed by dualization to the convex-hull problem [14], [20], it may be thought that the convex-hull problem and the kernel problem have a primal-dual relationship. This is not the case, however, because the dualization methods, even when extended to polygons [14], do not map simple polygons into simple polygons. It is not surprising, then, that our algorithm for the kernel problem is quite different from the convex-hull algorithm of Wagener.

We begin our discussion with a few definitions. Let $P[e_i, e_j]$ denote the subchain of P from e_i to e_j, inclusive (edge subscripts are modulo n). Note that since each edge has an orientation, $P[e_i, e_j]$ is well defined and is different from $P[e_j, e_i]$. Given two adjacent edges e_i to e_j, define the *angle between* e_i *and* e_{i+1}, denoted $\alpha_{i,i+1}$, to be the signed angle e_i makes with e_{i+1} when they are translated (as vectors) so as to share a common start vertex. The angle is positive if we turn in a counterclockwise angle in going from e_i to e_{i+1} (again, all subscripts are modulo n) and negative otherwise. We generalize this definition as follows: given a subchain $P[e_i, e_j]$ we define the *turn angle* of $P[e_i, e_j]$, denoted $\alpha_{i,j}$, to be the sum of all the edge angles from e_i to e_j. This can be expressed symbolically as

$$\alpha_{i,j} = \sum_{k=i}^{j-1} \alpha_{k,k+1}.$$

For completeness, we define $\alpha_{i,i} = 0$ for all $i \in \{0, 1, \ldots, n-1\}$.

If there are two edges e_i and e_j on P such that $\alpha_{i,j} \geq 3\pi$, then we say that P is a *spiral* polygon. The next lemma establishes an important property of spiral polygons.

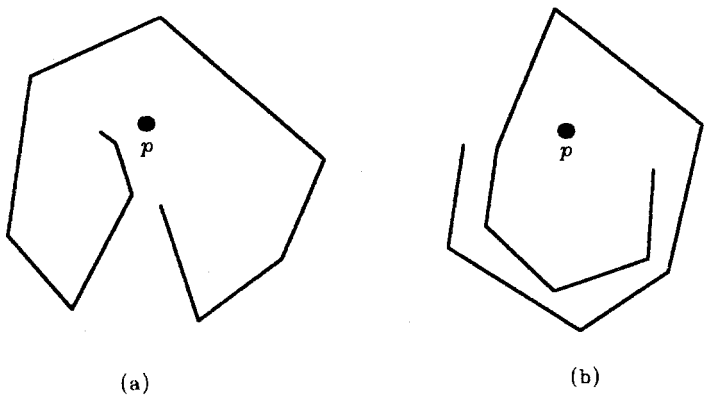

Fig. 5. A spiral polygon has an empty kernel.

LEMMA 5.1. *If P is a spiral polygon, then $K(P)$ is empty.*

PROOF. Suppose P is a spiral polygon, yet $K(P)$ is nonempty. Since $K(P)$ is nonempty, then P is star-shaped. That is, for each point $p \in K(P)$ the boundary of P is completely visible from p and the vertices of P, as listed around the boundary of P, are sorted radially around p. By hypothesis, there is some part of the boundary of P, say $P[e_i, e_j]$, with a turn angle of at least 3π. This contradicts one of the following: (a) that $P[e_i, e_j]$ is sorted radially around p or (b) that all of $P[e_i, e_j]$ is visible from p. (See Figure 5.) □

We can trivially test if P is a spiral polygon in $O(\log n)$ time using $O(n^2)$ processors (by computing all the $\alpha_{i,j}$ values). However, since we only have $O(n/\log n)$ processors at our disposal, our method for determining if P is a spiral polygon needs to be a little more involved. We begin by computing $\alpha_{0,i}$ and $\alpha_{i,0}$ for all $i \in \{0, 1, \ldots, n-1\}$. This can easily be done in $O(\log n)$ time using $O(n/\log n)$ processors by two simple parallel prefix computations [15], [16]. We also compute the following quantities:

$$f_i = \max_{0 \le j \le i-1} \alpha_{0,j},$$

$$b_i = \max_{i \le j \le n} \alpha_{j,0},$$

$$low = \min_{0 \le j \le n-1} \alpha_{0,j}.$$

Again, all subscripts are modulo n. Note that, since $\alpha_{0,0} = 0$, f_i and b_i are nonnegative, and *low* is nonpositive. Intuitively, having fixed the edge e_0, f_i measures the maximum turn angle formed by walking from e_0 to e_i. Similarly, b_i measures the maximum turn angle formed by walking from e_i to e_0 (which can be alternatively thought of as walking backward from e_0 to e_i). The quantity *low* measures the most one would turn in the negative direction in walking around P

starting from e_0. As with the $\alpha_{0,i}$'s and $\alpha_{i,0}$'s, these quantities can be easily computed in $O(\log n)$ time using $O(n/\log n)$ processors by parallel prefix computations. The next lemma establishes an important property of the above quantities.

LEMMA 5.2. *Suppose we are given n numbers $a_0, a_1, \ldots, a_{n-1}$ such that $\sum_{k=0}^{n-1} a_k \geq 0$. Let $\alpha_{i,j} = \sum_{k=i}^{j} a_k$ (where indices are modulo n), and let f_i, b_i, and low be defined as above (in terms of the $\alpha_{i,j}$'s). Then* $\max_{i,j}\{\alpha_{i,j}\} = \max\{f_{n-1} - low, \max_i\{b_i + f_i\}\}$.

PROOF. ($\geq$) We first show that $\max_{i,j}\{\alpha_{i,j}\} \geq \max\{f_{n-1} - low, \max_i\{b_i + f_i\}\}$. Consider $f_{n-1} - low$. Let c and d be indices such that $low = \alpha_{0,c}$ and $f_{n-1} = \alpha_{0,d}$. If $c < d$, then $f_{n-1} - low = \alpha_{0,d} - \alpha_{0,c} = \alpha_{c,d}$. If $d < c$, then $f_{n-1} - low = \alpha_{0,d} - \alpha_{0,c} = -\alpha_{d,c} \leq \alpha_{c,d}$. Thus, $f_{n-1} - low \leq \max_{i,j}\{\alpha_{i,j}\}$. Now consider $\max_i\{b_i + f_i\}$. Let c be the index such that $\max_i\{b_i + f_i\} = b_c + f_c$, and let $e \in [0, c-1]$ and $d \in [c, n]$ be indices such that $f_c = \alpha_{0,e}$ and $b_c = \alpha_{d,0}$. Since $e < d$, $f_c + b_c = \alpha_{0,e} + \alpha_{d,0} = \alpha_{d,e}$. Thus, $\max_{i,j}\{\alpha_{i,j}\} \geq \max\{f_{n-1} - low, \max_i\{b_i + f_i\}\}$.

($\leq$) Suppose, for the sake of contradiction, that $\max_{i,j}\{\alpha_{i,j}\} > \max\{f_{n-1} - low, \max_i\{b_i + f_i\}\}$. Let c and d be indices such that $\alpha_{c,d} = \max_{i,j}\{\alpha_{i,j}\}$.

Case 1: $c < d$. In this case $\alpha_{c,d} = \alpha_{0,d} - \alpha_{0,c}$, but $\alpha_{0,d} - \alpha_{0,c} \leq f_{n-1} - low$.

Case 2: $d < c$. In this case $\alpha_{c,d} = \alpha_{0,d} + \alpha_{c,0}$. Thus, $\alpha_{c,d} \leq f_{d+1} + b_c$, but $f_{d+1} + b_c \leq f_i + b_i$ for $i \in [d+1, c]$.

Thus, $\alpha_{c,d} \leq max\{f_{n-1} - low, \max_i\{b_i + f_i\}\}$. This completes the lemma. □

COROLLARY 5.3. *P is a spiral polygon if and only if* $\max\{f_{n-1} - low, \max_i\{b_i + f_i\}\} \geq 3\pi$.

Thus, we have a simple way to characterize spiral polygons that is easily tested on a parallel machine. Let Q_1 be the lexicographically first maximal increasing subsequence of $(e_0, \ldots, e_{n-1})$, using the f_i's as weights, and let Q_2 be the lexicographically first maximal increasing subsequence of $(e_0, e_{n-1}, \ldots, e_1)$, using the b_i's as weights. Recall that a lexicographically first maximal increasing subsequence is defined by placing the first item in the list in the set, then scanning through the list adding an item to the set each time its label is bigger than the biggest label encountered thus far. (Note that we could have just as easily defined Q_1 and Q_2 using $\alpha_{0,i}$ and $\alpha_{i,0}$ as weights, respectively.) The following lemma establishes an even stronger relationship between $K(P)$ and the turn-angle properties of P.

LEMMA 5.4. *If P is not a spiral polygon, then $K(P) = K(Q_1) \cap K(Q_2)$.*

PROOF. Since Q_1 and Q_2 are subsets of P, $K(P) \subseteq K(Q_1) \cap K(Q_2)$. So, we have yet to show that $K(Q_1) \cap K(Q_2) \subseteq K(P)$. Clearly, if $K(Q_1) \cap K(Q_2) = \varnothing$, then we are done; so suppose $K(Q_1) \cap K(Q_2) \neq \varnothing$. The proof is by contradiction. Suppose $K(P)$ is properly contained in $K(Q_1) \cap K(Q_2)$. Then there is an edge e_j of P with $e_j \notin Q_1 \cup Q_2$ and such that $H(e_j) \cap K(Q_1) \cap K(Q_2)$ is a proper subset of $K(Q_1) \cap K(Q_2)$. Let e_i be the edge closest to e_j in P such that e_i is in Q_1 and $i < j$.

CLAIM. *e_j does not intersect $H(e_i)$.*

PROOF OF THE CLAIM. We show that if e_j intersects $H(e_i)$, then P is a spiral polygon (which would be a contradiction). So suppose e_j intersects $H(e_i)$. Consider the polygonal chain $P[e_i, e_j]$. Let e_l be the first edge (other than e_i) in $P[e_i, e_j]$ that intersects $H(e_i)$. Note that e_l must intersect $H(e_i)$ behind e_i (as defined by e_i's orientation). If this was not so, then $P[e_i, e_l]$ would have a positive turn angle; hence, e_l would be in Q_1, which contradicts the definition of e_i. Thus, since $P[e_i, e_l]$ is a finite chain beginning and ending in $H(e_i)$, the turn angle from e_i to e_l must be less than $-\pi$. But this implies that the turn angle from e_l to e_i, in $P[e_l, e_i]$, is greater than 3π, since P is a simple polygon. Therefore, P must be a spiral polygon, which is a contradiction. (See Figure 6.) □

Let e_k be the edge closest to e_j in P such that e_k is in Q_2 and $j < k$. By an argument similar to the proof of the above claim we have that e_j is not contained in $H(e_k)$. These two facts imply that the edge e_j is not contained in $H(e_i) \cap H(e_k)$. But this implies that $H(e_i) \cap H(e_j) \cap H(e_k) = H(e_i) \cap H(e_k)$. In other words, $H(e_j) \cap K(Q_1) \cap K(Q_2)$ is not a proper subset of $K(Q_1) \cap K(Q_2)$, which of course is a contradiction. Therefore, $K(P) = K(Q_1) \cap K(Q_2)$. □

The above lemmas immediately give us the outline of our algorithm for constructing $K(P)$: test if P is a spiral polygon, and, if it is not a spiral polygon, construct $K(Q_1)$ and $K(Q_2)$ and their intersection.

We have already described how to test if P is a spiral polygon or not. So suppose P is not a spiral polygon. We begin by constructing Q_1 and Q_2. This can be done by yet another parallel prefix computation in $O(\log n)$ time using $O(n/\log n)$

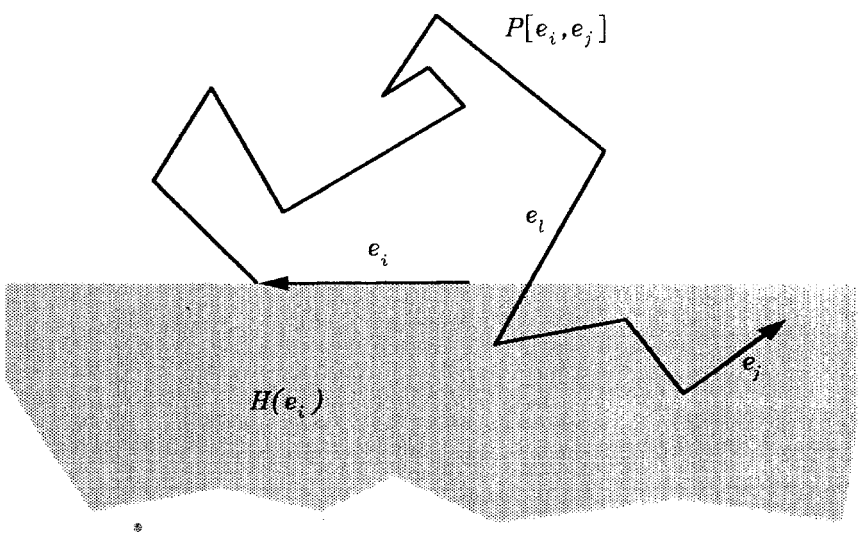

Fig. 6. If e_j intersects $H(e_i)$, then P is a spiral polygon.

processors (by computing, for each edge e_j in the list in question, the maximum prefix label of the edges proceding e_j). Note that the lists Q_1 and Q_2 are sorted by slopes. In addition, the list Q_1 (resp. Q_2) can be easily partitioned into $O(1)$ lists such that the range of label values in each list is at most π (this takes at most $O(\log n)$ time using $O(n/\log n)$ processors). By appropriately translating the origin for the edges in each of these lists we can guarantee that the origin is contained in their the common intersection of the half-planes they define (the details of this translation are left to the reader). We can then use the dualization methods of [14] and [20] to dualize each half-plane intersection problem to the problem of constructing the convex hull of a sorted point set, a problem that can be solved in $O(\log n)$ time using $O(n/\log n)$ processors in the CREW PRAM model [12], [27]. We can combine all the solutions to these convex-hull problems by converting their solutions back to the primal space, yielding a collection of convex polygons, each formed by a set of intersecting half-planes. We can then use parallel merging [6], [24] to implement the sequential algorithm of Shamos [22] for constructing the intersection of two convex polygons to compute the common intersection of these polygons, giving us $K(Q_1)$ and $K(Q_2)$. Each of these intersection computations runs in $O(\log n)$ time using $O(n/\log n)$ processors [6], [22], [24]. By making one additional call to the parallel version of the Shamos algorithm, we can construct $K(Q_1) \cap K(Q_2)$, which completes our algorithm. We summarize with the following theorem.

THEOREM 5.5. *Given an n-edge simple polygon P we can construct the kernel of P in $O(\log n)$ time using $O(n/\log n)$ processors in the CREW PRAM model.*

6. Final Remarks and Open Problems. In this paper we presented parallel analogues to some famous phenomena from sequential computational geometry. Namely, that convex hulls can be constructed by performing a cascading divide-and-conquer version of "rotating calipers," that the all-nearest-neighbor problem can be solved without constructing a Voronoi diagram, and that problems for polygons can oftentimes be solved more efficiently than point-set problems. Another interesting observation is that in developing an optimal parallel algorithm for the kernel problem we discovered some geometric relationships that result in a new optimal sequential algorithm (which is simpler than the previous best algorithm [18]). We leave two open problems:

1. Can the Voronoi diagram of n planar points be deterministically constructed in $O(\log n)$ time using $O(n)$ processors? The current best algorithm runs in $O(\log^2 n)$ time using $O(n)$ processors [1], [2].
2. Can a simple polygon (without holes) be deterministically triangulated in $O(\log n)$ time using $O(n \log\log n)$ total work? This is another problem that can be solved more efficiently for polygons than for arbitrary point sets, as it can be solved in $O(n \log\log n)$ time [25]. The best-known parallel algorithms run in $O(\log n)$ time using $O(n)$ processors (but allow the polygon to contain holes) [11], [13], [29].

Acknowledgments. We wish to thank S. Rao Kosaraju and the referees for a number of suggestions that helped improve the presentation of this paper.

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap, Parallel Computational Geometry, *Algorithmica*, Vol. 3, No. 3, 1988, pp. 293–328.

[2] M. J. Atallah, R. Cole, and M. T. Goodrich, Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms, *SIAM J. Comput.*, Vol. 18, No. 3, 1989, pp. 499–532 (appeared in preliminary form in *Proc. 28th IEEE Symp. on Foundations of Computer Science*, 1987, pp. 151–160).

[3] M. J. Atallah and M. T. Goodrich, Efficient Parallel Solutions to Some Geometric Problems, *J. Parallel Distrib. Comput.* Vol. 3, 1986, pp. 492–507.

[4] M. J. Atallah and M. T. Goodrich, Parallel Algorithms for Some Functions of Two Convex Polygons, *Algorithmica*, Vol. 3, No. 4, 1988, pp. 535–548.

[5] J. L. Bentley and M. I. Shamos, Divide-and-Conquer in Multidimensional Space, *Proc. 8th ACM Symp. on Theory of Computing*, 1976, pp. 220–230.

[6] G. Bilardi and A. Nicolau, Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared Memory Machines, TR 86-769, Dept. of Computer Science, Cornell University, August 1986.

[7] A. Chow, Parallel Algorithms for Geometric Problems, Ph.D. thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, 1980.

[8] R. Cole, Parallel Merge Sort, *SIAM J. Comput.*, Vol. 17, No. 4, August 1988. pp. 770–785.

[9] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, New York, 1987.

[10] A. Fournier and Z. Kedem, Comments on the All-Nearest-Neighbor Problem for Convex Polygons, *Inform. Process. Lett.*, Vol. 9, No. 3, 1979, pp. 105–107.

[11] M. T. Goodrich, Efficient Parallel Techniques for Computational Geometry, Ph.D. thesis, Dept. of Computer Science, Purdue University, August 1987.

[12] M. T. Goodrich, Finding the Convex Hull of a Sorted Point Set in Parallel, *Inform. Process. Lett.*, Vol. 26, December 1987, pp. 173–179.

[13] M. T. Goodrich, Triangulating a Polygon in Parallel, *J. Algorithms*, Vol. 10, 1989, pp. 327–351.

[14] L. Guibas, L. Ramshaw, and J. Stolfi, A Kinetic Framework for Computational Geometry, *Proc. 24th IEEE Symp. on Foundations of Computer Science*, 1983, pp. 100–111.

[15] C. P. Kruskal, L. Rudolph, and M. Snir, The Power of Parallel Prefix, *Proc. 1985 IEEE Internat. Conf. on Parallel Processing*, pp. 180–185.

[16] R. E. Ladner and M. J. Fischer, Parallel Prefix Computation, *J. Assoc. Comput. Mach.*, October 1980, pp. 831–838.

[17] D. T. Lee and F. P. Preparata, The All-Nearest-Neighbor Problem for Convex Polygons, *Inform. Process. Lett.*, Vol. 7, No. 4, June 1978, pp. 189–192.

[18] D. T. Lee and F. P. Preparata, An Optimal Algorithm for Finding the Kernel of a Polygon, *J. Assoc. Comput. Mach.*, Vol. 26, No. 3, July 1979, pp. 414–421.

[19] D. T. Lee and F. P. Preparata, Computational Geometry—A Survey, *IEEE Trans. Comput.*, Vol. 33, No. 12, December 1984, pp. 872–1101.

[20] F. P. Preparata and D. E. Muller, Finding the Intersection of n Half-Spaces in Time $O(n \log n)$, *Theoret. Comput. Sci.*, Vol. 8, 1979, pp. 45–55.

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

[22] M. I. Shamos, Geometric Complexity, *Proc. 7th ACM Symp. on Theory of Computing*, 1975, pp. 224–233.

[23] M. I. Shamos and D. Hoey, Closest-Point Problems, *Proc. 15th IEEE Symp. on Foundations of Computer Science*, 1975, pp. 151–162.

[24] Y. Shiloach and U. Vishkin, Finding the Maximum, Merging, and Sorting in a Parallel Computation Model, *J. Algorithms*, Vol. 2, 1981, pp. 88–102.

[25] R. E. Tarjan and C. J. Van Wyk, An $O(n \log \log n)$-Time Algorithm for Triangulating a Simple Polygon, *SIAM J. Comput.*, Vol. 17, 1988, pp. 143–178.

[26] G. T. Toussaint, Solving Geometric Problems with Rotating Calipers, *Proc. IEEE MELECON '83*, Athens, May 1983.
[27] H. Wagener, Optimally Parallel Algorithms for Convex Hull Determination, Manuscript, 1985.
[28] C. C. Yang and D. T. Lee, A Note on the All-Nearest-Neighbor Problem for Convex Polygons, *Inform. Process. Lett.*, Vol. 8, No. 4, 1979, pp. 193–194.
[29] C.-K. Yap, Parallel Triangulation of a Polygon in Two Calls to the Trapezoidal Map, *Algorithmica*, Vol. 3, No. 2, 1988, pp. 279–288.