1989

Efficient Geometric Algorithms in the EREW-PRAM

Danny Z. Chen

Report Number:
89-928

EFFICIENT GEOMETRIC ALGORITHMS IN THE EREW-PRAM

Danny Z. Chen

CSD-TR-928
December 1988
(Revised October 1990)

Efficient Geometric Algorithms in the EREW-PRAM (Preliminary Version)

Danny Z. Chen*

Department of Computer Science
Purdue University
West Lafayette, IN 47907.

Abstract

We present a technique that can be used to obtain efficient parallel algorithms in the EREW-PRAM computational model. This technique enables us to optimally solve a number of geometric problems in $O(\log n)$ time using $O(n/\log n)$ EREW-PRAM processors, where n is the input size. These problems include: computing the convex hull of a sorted point set in the plane, computing the convex hull of a simple polygon, finding the kernel of a simple polygon, triangulating a sorted point set in the plane, triangulating monotone polygons and star-shaped polygons, computing the all dominating neighbors, etc. PRAM algorithms for these problems were previously known to be optimal (i.e., $O(\log n)$ time and $O(n/\log n)$ processors) only in the CREW-PRAM, which is a stronger model than the EREW-PRAM.

1 Introduction

We present a technique for obtaining efficient parallel algorithms in the EREW-PRAM computational model. This technique, when applied to a number of geometric problems on sorted point sets and simple polygons, enables us to get optimal solutions (in $O(\log n)$ time using $O(n/\log n)$ EREW-PRAM processors). The technique consists of a binary tree data structure (to be defined in Section 3), several kinds of parallel operations on the binary tree, and a combination of a two-way divide-and-conquer and a quarter-root divide-and-conquer strategies that are used in conjunction with the parallel tree operations. The parallel tree operations include parallel searching, parallel concatenation, and parallel split. These parallel tree operations are performed without read conflicts. This technique is likely to be useful for obtaining efficient parallel algorithms in the EREW-PRAM for problems other than those discussed in this paper.

*This research was partially supported by the Office of Naval Research under Grants N00014-84-K-0502 and N00014-86-K-0689, the National Science Foundation under Grant DCR-8451393, and the National Library of Medicine under Grant R01-LM05118.

The computational model we use is the EREW-PRAM (Exclusive Read Exclusive Write Parallel Random Access Machine); it is a synchronous parallel computational model in which all processors share a common memory and each processor can access any memory location in constant time. The EREW-PRAM does not allow more than one processor to simultaneously access the same memory address. We also refer to another version of the PRAM model, called the CREW (Concurrent Read Exclusive Write) PRAM. The CREW-PRAM allows simultaneous accesses to the same memory location by multiple processors *if* all such concurrent accesses are for reading data only. The CREW-PRAM is obviously a more powerful model than the EREW-PRAM, and the simulation of a CREW-PRAM algorithm on an EREW-PRAM, using the same number of processors, can cost an increase in the time complexity by a logarithmic factor.

We use the technique to optimally solve the following problems: computing the convex hull of n sorted points in the plane (and hence the dual problem of finding the common intersection of n half-planes given sorted by their slopes), computing the convex hull of an n-vertex simple polygon, finding the kernel of an n-vertex simple polygon, triangulating n sorted points in the plane, triangulating an n-vertex monotone polygon or star-shaped polygon, and computing the *all dominating neighbors* of n values. Our EREW-PRAM algorithms for these problems all take $O(\log n)$ time using $O(n/\log n)$ processors.

The problems of computing the convex hulls of point sets and polygons, computing the kernel of a simple polygon, and triangulating point sets and polygons, are of fundamental importance in computational geometry and have applications in many areas. A great deal of work, both in the sequential and parallel computational models, has been done on finding efficient solutions for these problems (see [22, 27] for the sequential algorithms for these problems). For the problem of computing the convex hull of n *arbitrary* points in the plane, optimal solutions (i.e., $O(\log n)$ time and $O(n)$ processors) have been given in the CREW-PRAM [1, 4, 5] and in the EREW-PRAM [26]. Optimal CREW-PRAM algorithms were also known for the problem of triangulating n *arbitrary* points in the plane [25, 32] and for the problem of triangulating polygons [13, 33].

The problems we consider in this paper all have an obvious lower bound of linear work, and sequential linear time algorithms for them have already been known. Some of the solutions can be found in [6, 11, 14, 15, 16, 18, 21, 23, 30, 31]. (Our algorithms are optimal in the EREW-PRAM since they all run in $O(\log n)$ time and their *time* $\times$ *processors* products match the lower bound of these problems.)

Efficient CREW-PRAM algorithms solving the problems that we consider in this paper have also been discovered. The convex hull problem for a sorted point set can be solved in $O(\log n)$ time using $O(n/\log n)$ CREW-PRAM processors [6, 12, 29], and such an algorithm implies (by duality) the same complexity bounds for computing the common intersection of n half-planes whose slopes are given sorted [12]. Note that in the case where the points are already given sorted, the EREW-PRAM algorithm of [26] still requires $O(n)$ processors, which is sub-optimal. Our algorithm for the convex hull of a sorted point set can be viewed as another optimal algorithm (i.e., $O(\log n)$ time and $O(n)$ processors) in the EREW-PRAM for the case of unsorted input, because we can first obtain a sorted point set with $O(n)$ processors [8] and then use $O(n/\log n)$ processors for the remaining computation. For the case where the input points are given as a list of vertices on a simple polygon, the convex hull problem can be solved optimally in $O(\log n)$ time using $O(n/\log n)$ CREW-PRAM processors [29].

The problem of computing the kernel of a simple polygon has been solved optimally in the CREW-PRAM by Cole and Goodrich [9]. Their algorithm is based on the interesting observations which characterize the "curvature" of the polygon boundary.

For sorted point sets and monotone polygons, the triangulation problems can be solved optimally in the CREW-PRAM [6, 13, 18, 30]. In fact, Goodrich [13] showed that if the trapezoidal decomposition of a polygon (possibly with holes) has been provided, then a triangulation for that polygon can be done in $O(\log n)$ time using $O(n/\log n)$ CREW-PRAM processors. A CREW-PRAM algorithm in $O(\log n)$ time with $O(n/\log n)$ processors for triangulating a star-shaped polygon is recently given in [24].

The problem of computing the *all dominating neighbors* of n values is defined as follows: Given values $w_1, w_2, \ldots, w_n$, find for each index i the largest (resp., smallest) index $j < i$ (resp., $k > i$) such that $w_j \geq w_i$ (resp., $w_k \geq w_i$). This problem was considered by [6, 18] to be fundamentally important for solving several other problems (not only in computational geometry) in the PRAM. Especially, this problem was used as the basic subproblem for triangulating point sets in the plane and monotone polygons [6, 18, 25, 30]. Optimal algorithms for this problem (in $O(\log n)$ time and $O(n/\log n)$ CREW-PRAM processors) have been given in [6, 18, 30].

Most of the constituent parts of our algorithms, namely, the geometric observations, the divide and conquer strategies, the binary tree data structure, and the parallel tree operations (except for the parallel split), have been used before (for example, see [3, 5, 9, 12, 13, 28]).

Our contribution is in putting these already available "parts" together in such a way that will enable us to avoid read conflicts that occurred in the previous known CREW-PRAM algorithms.

The rest of this paper is organized as follows. Section 2 gives the notation we use in the paper and outlines the general structure of the algorithms. Section 3 discusses the binary tree data structure and the parallel tree operations. Sections 4 to 6 show how to solve the problems we mentioned above.

2 Notation and Basic Algorithm Structure

Let S be a set of n points $p_1, p_2, \ldots, p_n$. S is sorted either by x-coordinate, or by polar angle with respect to a specified *polar point* $q \in S$. Let P be a simple polygon defined by the list of vertices $v_1, v_2, \ldots, v_n$, in the order of a clockwise travel along the polygon boundary.

WLOG (without loss of generality), we assume that in S (resp., P), no two points (resp., vertices) have the same x- or y-coordinate and no three points (resp., vertices) are collinear. The general situations can be taken care of by slightly modifying our algorithms.

We say two point sets S' and S'' are *separable* if there exists a vertical line such that S' and S'' are on the opposite sides of the line. Furthermore, for $k \geq 2$, we say k point sets are *separable* if there exist $k-1$ vertical lines such that for any two point sets, at least one of the $k-1$ vertical lines separates them.

The main procedure of our algorithms has a basic structure, which is the same as the one used in [3]. We outline it as follows.

Input: A set X of size m, which is either a sorted point set or the vertex set of a simple polygonal chain, and a positive integer d.

Output: The desired output $T(X)$ represented by a tree data structure.

Case 1. If $m \leq d$, then compute $T(X)$ with one processor in $O(m)$ time, using a sequential linear-time algorithm.

Case 2. If $d < m \leq d^2$, then divide X into two subsets X_1 and X_2 of equal size and recursively solve the two subproblems in parallel. Then compute $T(X)$ from $T(X_1)$ and $T(X_2)$, in $O(\log m)$ time using one processor.

Case 3. If $m > d^2$, then partition X into $g = (m/d)^{1/4}$ subsets $X_1, X_2, \ldots, X_g$ of size $m^{3/4}d^{1/4}$ each. Then, in parallel, recursively solve the g subproblems. Finally, compute $T(X)$ from $T(X_1), T(X_2), \ldots, T(X_g)$, in $O(\log m)$ time using $m/d = g^4$ processors.

end.

Observe that, if we could perform the various cases of the above outline within the claimed bounds, then the algorithm would run in $O(d+\log m)$ time with $O(m/d)$ processors since the recurrences of the time and processor complexities are (almost) the same as those in [3] (the only difference is that Case 2 in [3] runs in $O(\log^2 m)$ time while it takes only $O(\log m)$ time here). Choosing $d = \log m$, the above implies a time bound of $O(\log m)$ and a processor bound of $O(m/\log m)$. Therefore, a call to the algorithm with input $(X, \log n)$, $|X| = n$, will compute $T(X)$ in $O(\log n)$ time using $O(n/\log n)$ processors. The rest of the paper shows how to solve the problems by using algorithms like the one outlined above.

3 The Binary Tree Data Structure and Operations

The algorithms make use of a binary tree data structure. The definition of this tree structure is similar to the one for *hull tree*, used by Goodrich to store the information of the convex hull for a set of points [12] or the monotone funnel polygons [13]. We call such trees the *rank trees* because they support efficient operations based on the ranks of the leaves in the trees. A rank tree T is a binary search tree with a set of points stored at its leaves in some specified order (e.g., by increasing x-coordinate or polar angle). WLOG, we assume that the points are sorted by increasing x-coordinate. The leaves of T are doubly linked together. We denote the height of T (the length of the longest root-to-leaf path in T) by $h(T)$. Let T_v be the subtree of T rooted at node v of T. Each internal node v of T has four labels: the first stores the number of leaves in T_v, the second stores the point p at a leaf of T_v such that p has the smallest x-coordinate among the points stored at the leaves of T_v, and the other two respectively store the *predecessor* and the *successor* of p in the sorted point set stored at the leaves of T. In $O(h(T))$ time, one processor can search in T for an x-coordinate (and hence for a point) or search for the i-th ranked point stored in T (i.e., the i-th leaf of T in the left-to-right order) using the first or the second label stored in the internal nodes, respectively.

In the rest of this paper, all trees are assumed to be rank trees unless otherwise specified.

Our algorithms may need many processors to simultaneously search in such a tree. The next lemma, which is based on the parallel searching scheme of [28], enables us to avoid read conflicts during such parallel searching.

Lemma 1 ([3]) *Given a tree T, suppose each of k processors wants to perform a search in T. Two types of searches are allowed: the first type is a search for a particular point using its x-coordinate, and the second is of the type "find the t-th leaf of T starting from leaf l and moving to the right". Then the k processors can perform their searches in $O(\log k + h(T))$ time, without any read conflict.*

Proof. Same as that for Corollary 5.2 of [3] and hence omitted. □

The next two lemmas are for the parallel concatenation and split.

Lemma 2 (Goodrich [12, 13]) *Let $S_1, S_2, \ldots, S_k$ be subsets of a point set S' separated by $k-1$ vertical lines, and let the trees $T(S_1), T(S_2), \ldots, T(S_k)$ for the subsets be given. Then tree $T(S')$ for S' can be built in $O(\log k + h)$ time using k EREW-PRAM processors, where h is the maximum of the $h(T(S_i))$'s. Also, $h(T(S')) = O(h + \log k)$.*

Proof. Same as the proof of Lemma 4.3 in [13] and hence omitted. □

Lemma 3 *Given a tree T and a list $(x_1, x_2, \ldots, x_k)$ of x-coordinates, suppose k EREW-PRAM processors want to split T into $k+1$ trees $T_0, T_1, \ldots, T_k$ such that all points in T_i have their x-coordinates within interval $[x_i, x_{i+1}]$, for $i \in \{0, 1, \ldots, k\}$ ($x_0 = -\infty$ and $x_{k+1} = +\infty$). Then the parallel split can be done in $O(\log k + h(T))$ time.*

Proof. WLOG, we assume that the list $(x_1, x_2, \ldots, x_k)$ is given sorted (otherwise, we have enough processors to do the sorting in $O(\log k)$ time [8]). Let processor P_i have value x_i, $i = 1, 2, \ldots, k$. Each P_i does the split operation in the same way as shown in Lemma 3.2 of [12]. That is, it searches for x_i in a tree and, when going down the tree following a root-to-leaf path, it makes two copies of the root-to-leaf path, whose nodes have the appropriately modified data from the original root-to-leaf path (actually, the original path is replaced by one copy); after it reaches the leaf of the path, it retraces the two paths which it just created, to update the labels of the nodes on the paths (see Lemma 3.2 in [12] for more details).

To avoid read conflicts, we use a procedure consisting of $O(\log k)$ stages. Let $T_{1,k} = T$. Before a tree $T_{a,b}$ is split ($a \leq b$), there is a group of processors $P_a, P_{a+1}, \ldots, P_b$ associated

with it. The following is the general step. Suppose (the root of) $T_{a,b}$ is available in stage i but not in stage $i-1$. If $a = b$, then P_a splits $T_{a,b}$ at its root; otherwise, $P_{\lceil (a+b)/2 \rceil}$ splits $T_{a,b}$ at its root (by making two copies) and makes the roots of $T_{a,\lceil (a+b)/2 \rceil - 1}$ and $T_{\lceil (a+b)/2 \rceil + 1,b}$ available for stage $i+1$ (no split is done on $T_{\lceil (a+b)/2 \rceil + 1,b}$ if $\lceil (a+b)/2 \rceil + 1 > b$). The processors stop when they reach the leaves on the root-to-leaf paths they follow. After all processors stop at the leaves of the $k+1$ trees so obtained, we let each processor retrace the leaf-to-root paths in each tree which it just created and update the labels of the nodes on the paths just as was done in [12].

The correctness of this parallel split procedure is guaranteed by the facts that $(x_1, x_2, \ldots, x_k)$ is sorted and the split is based on searching the x_i's. No read conflict can occur in the procedure because although in the searching, different processors may follow the same root-to-leaf path in T, the processors, when doing the split, actually use different copies of the path, and such copies would have been created in the previous stages of the procedure. The time complexity of the procedure is clearly $O(\log k + h(T))$. □

4 Computing the Convex Hull of Sorted Points

This section discusses the algorithm for computing the convex hull of a point set in the plane sorted by increasing x-coordinate. Let $S = \{p_1, p_2, \ldots, p_n\}$ be a set of sorted points. We denote the convex hull of S by $CH(S)$. Points p_1 and p_n are both vertices of $CH(S)$ because p_1 and p_n, respectively, have the smallest and the largest x-coordinates among the points in S. Traveling along $CH(S)$ from p_1 to p_n clockwise, the portion of $CH(S)$ so visited is called the *upper hull* of S, denoted by $UH(S)$. Similarly, the portion of $CH(S)$ visited by traveling along $CH(S)$ from p_n to p_1 clockwise is called the *lower hull* of S, denoted by $LH(S)$. Due to the similarity in the computation of $UH(S)$ and $LH(S)$, we only discuss the algorithm for $UH(S)$. For two upper hulls $UH(S')$ and $UH(S'')$, where S' and S'' are separable point sets, the *upper common tangent* between $UH(S')$ and $UH(S'')$ is the common tangent of $UH(S')$ and $UH(S'')$ such that both $UH(S')$ and $UH(S'')$ are below it. The *lower common tangent* for two lower hulls is defined similarly. In the rest of this section, we just say the "common tangent" to mean the "upper common tangent".

The following two known results are useful.

Lemma 4 (Goodrich [12, 13]) *Let two upper hulls $UH(S_1)$ and $UH(S_2)$ be stored in trees T_1 and T_2, respectively, where S_1 and S_2 are two separable point sets. Then in $O(h(T_1)+h(T_2))$*

time, one processor can find the common tangent between $UH(S_1)$ and $UH(S_2)$.

Proof. See Lemma 3.1 in [12]. □

Lemma 5 (Atallah, Goodrich [5]) *Let S_1 and S_2 be two separable point sets with both $|S_1|$ and $|S_2|$ being $O(m)$, and let $UH(S_1)$ and $UH(S_2)$ be their upper hulls stored in two arrays, respectively. Then the common tangent between $UH(S_1)$ and $UH(S_2)$ can be computed in $O(c^2) = O(1)$ time using $m^{1/c}$ CREW-PRAM processors, where c is a positive constant.*

Proof. See Theorem 1 and **Algorithm A** in [5]. □

We immediately have the following corollary.

Corollary 1 *Let S_1 and S_2 be two separable point sets with both $|S_1|$ and $|S_2|$ being $O(m)$, and let $T(S_1)$ and $T(S_2)$ be two trees storing the upper hulls $UH(S_1)$ and $UH(S_2)$, respectively. Then the common tangent between $UH(S_1)$ and $UH(S_2)$ can be computed in $O(\log m + h)$ time using $m^{1/c}$ EREW-PRAM processors, where c is a positive constant and h is the maximum of $h(T(S_1))$ and $h(T(S_2))$.*

Proof. Recall that **Algorithm A** in [5] partitions the two arrays for the two upper hulls $UH(S_1)$ and $UH(S_2)$ into subarrays, then it finds in which subarrays the common tangent lies, and it then recursively solves the problem in the (two) subarrays so found. We simulate **Algorithm A** using $m^{1/c}$ EREW-PRAM processors. Note that **Algorithm A** is in the CREW-PRAM and it requires $O(1)$ time (given c a constant). Since now $UH(S_1)$ and $UH(S_2)$ are stored in trees (instead of arrays), each access to a leaf of a tree requires $O(h)$ time and one processor. Parallel searching for the points at the leaves of a tree (without read conflicts) can be done in $O(\log m + h)$ time using the available processors by Lemma 1. Each time an array (or a subarray) is partitioned in **Algorithm A**, we can achieve the same effect by partitioning the leaves of the relevant tree by using the ranks of the leaves. Such a partition can also be done in $O(\log m + h)$ time by doing parallel searching in the tree by Lemma 1. The other steps of **Algorithm A** can be easily simulated in $O(\log m)$ time in the EREW-PRAM. □

Now we show the algorithm for computing $UH(S)$. We refer to the cases of the outline in Section 2. In Case 1, we call the linear time algorithm in [14] to compute the upper hull. In Case 2, we use Lemma 4 to compute the common tangent between the two upper hulls returned from the two recursive calls, then we split the tree for each upper hull to remove the portion of that upper hull (if any) that is under the common tangent. The portions of

the two upper hulls that remain form the upper hull that we seek in this case (by doing a simple concatenation). Since we use only one processor in this case, no read conflict occurs. We perform Case 3 as follows. Given g subsets $S_1, S_2, \ldots, S_g$ of a point set S', separated by $g-1$ vertical lines, where $|S'| = m$ and $g = (m/d)^{1/4}$, and given the trees $T(S_1), T(S_2), \ldots, T(S_g)$ representing their upper hulls, respectively, we compute the common tangent C_{ij} for each pair of $UH(S_i)$ and $UH(S_j)$, $1 \leq i < j \leq g$. Recall that we have $g^4 = m/d$ processors to do so, and $|S_k| = m^{3/4}d^{1/4}$ for each k. Every C_{ij} is obtained in $O(\log m)$ time using $g^2 = (m/d)^{1/2}$ processors by Corollary 1 (it has been shown in [3] that $h(T(S_k))$ is $O(\log m)$ for every k). Note that, in this case, we do not use the procedure for Lemma 4 to compute the C_{ij}'s. This is because the procedure for Lemma 4 searches for the points, on two upper hulls, where the common tangent for the two upper hulls lies, and before the search starts, we do not know at all which points we are getting to. In Case 3, each $T(S_k)$ is involved in the computation of $O(g)$ common tangents. If we used the procedure for Lemma 4, we would have read conflicts from the $O(g)$ simultaneous searches in $T(S_k)$, since we could not prearrange the $O(g)$ processors doing the searches (as was done in the scheme of [28]) in order to avoid read conflicts. From the $O(g^2)$ common tangents (the C_{ij}'s), we can find the portions of the $UH(S_k)$'s that form $UH(S')$, by doing parallel prefix [19, 20] (see [12] for the details on how this is done). The tree $T(S')$ is then built using Lemma 2.

5 Triangulating a Trapezoidally Decomposed Polygon

This section deals with the algorithm for triangulating an n-vertex polygon P (possibly with holes) when given a trapezoidal decomposition of P. Goodrich [13] showed how to triangulate a polygon in $O(\log n)$ time using $O(n/\log n)$ CREW-PRAM processors, provided that the trapezoidal decomposition of the polygon has been given (the trapezoidal decomposition of P can be done in $O(\log n)$ time using $O(n)$ CREW-PRAM processors [2]). Here we assume that the same input as in [13] is given. We will basically perform the same computation as in [13] (hence the reader is referred to [13] for more details of the algorithm). We only show how to use a *quarter-root* divide and conquer strategy and the parallel tree operations to perform various operations of [13] in the required time and processor complexities without read conflicts.

There are three phases in [13]. The reader is referred to [13] for the definitions used here. There is no read conflict in **Phase One**, whose goal is to construct the set of *one-sided*

monotone polygons which decomposes P (by using parallel prefix [19, 20] and list ranking [10]). There is also no read conflict in **Phase Three**, whose goal is to triangulate the set of *monotone funnel polygons* resulting from **Phase Two** (by using parallel prefix and parallel merging [7, 17]). Hence we only need to concern ourselves with **Phase Two**, whose goal is to decompose every one-sided monotone polygon (from **Phase One**) into a set of monotone funnel polygons.

The difficult computation in **Phase Two** is to decompose every one-sided monotone polygon whose size is larger than $\log n$ into a set of monotone funnel polygons. Given a monotone chain C (from the one-sided monotone polygon), $|C| = m$, the procedure in **Phase Three** for this computation first partitions chain C into α subchains of equal size, and recursively solves the α subproblems in parallel. Then, from the results for the α subproblems, it computes the *bases* of the monotone funnel polygons that consist of the decomposition of the one-sided monotone polygon. Finally, it computes the left and right boundaries of the monotone funnel polygons and the lower hull of C. This procedure, although being quite complicated, essentially consists of the following operations: parallel prefix, sorting $O(\alpha)$ values, computing $O(\alpha^2)$ lower common tangents (among the lower hulls of the α subchains represented by α trees, as returned by the recursive calls), parallel splits on the α trees (into $O(\alpha)$ trees), and parallel concatenations of $O(\alpha)$ trees (to construct new trees representing the left and right boundaries of the monotone funnel polygons as well as the lower hull of chain C). Recall that our algorithm is based on the outline given in Section 2. Case 1 and Case 2 of this algorithm can be easily handled in the required complexity bounds. In Case 3, we have $g = (m/d)^{1/4}$ subproblems and g^4 processors. The parallel prefix and sorting [8] can be done using the available number of processors. The $O(g^2)$ lower common tangents are computed in a way similar to the convex hull algorithm of Section 4 (i.e., by Corollary 1). The parallel splits are done by Lemma 3 and the parallel concatenations are done by Lemma 2. None of these operations introduces read conflicts and all of them can be performed in $O(\log m)$ time using g^4 EREW-PRAM processors.

6 Other Geometric Algorithms

The algorithms described in Sections 4 and 5 enable us to obtain optimal EREW-PRAM algorithms for other geometric problems. All algorithms in this section take $O(\log n)$ time using $(n/\log n)$ EREW-PRAM processors.

If a point set in the plane is given sorted by polar angle with respect to a polar point q, the convex hull algorithm in Section 4 can be slightly modified to compute the convex hull of this point set. Using the convex hull algorithm in Section 4, we compute the convex hull of an n-vertex simple polygon P as follows: apply the visibility algorithm by Atallah, Chen, and Wagener [3] (which runs in $O(\log n)$ time using $O(n/\log n)$ EREW-PRAM processors) to P to obtain a point set sorted by x-coordinate (the vertices of P that are visible from the point $q = (0, +\infty)$), then apply the convex hull algorithm in Section 4 to the visible vertex set to find the upper hull of P. Using the geometric duality transformation [27], an immediate result from the convex hull algorithm in Section 4 is an optimal EREW-PRAM algorithm for the problem of computing the common intersection of n half-planes given sorted by their slopes. The kernel of a simple polygon can be computed optimally by using the convex hull algorithm in Section 4 as a subroutine in the algorithm of [9] (parallel prefix and parallel merging [7, 17] are also used in [9]).

The triangulation algorithm in Section 5 implies an optimal EREW-PRAM solution for triangulating a monotone polygon P, since a parallel merging will decompose P into a set of one-sided monotone polygons (triangulating one-sided monotone polygons is done in Section 5). Using the algorithm for computing the kernel of a simple polygon, we can check whether a simple polygon P is star-shaped or not. If it is, then the kernel of P is nonempty. Let r be the ray starting from vertex v_1 of P and going through a point q in the kernel of P. WLOG, we assume that r does not contain any edge of P. Let r intersect the boundary of P at a point $p \neq v_1$. If p is at some vertex of P, then we partition the boundary of P into two polygonal chains C' and C'' by r. Otherwise, let p be on edge e with endpoints v_i and v_{i+1}, and let C' be the polygonal chain consisting of vertices $v_1, v_2, \ldots, v_i$, and C'' the polygonal chain consisting of $v_{i+1}, v_{i+2}, \ldots, v_n, v_1$. Clearly, C' and C'' are both star-shaped (i.e., they are all visible from q). A triangulation for each of C' and C'' can be done in a way similar to the algorithm in Section 5 for triangulating a one-sided monotone polygon, since the vertices of P are sorted along the boundary of P in polar angle with respect to point q (in this case, q plays the role for C' and C'' same as the distinguished edge of a one-sided monotone polygon does for the polygon). The triangulation of C' and C'' also gives a monotone funnel polygon inside P (with base e, the right boundary from the triangulation of C', and the left boundary from the triangulation of C''). This monotone funnel polygon is the only portion of P that has not yet been triangulated. A triangulation of P can be completed by doing a parallel merging (see **Phase Three** in [13]). Hence we

optimally solve the problem of triangulating a star-shaped polygon.

Triangulating a point set in the plane sorted by x-coordinate can be reduced to that of triangulating a set of one-sided monotone polygons, as follows. We first construct a monotone chain with the sorted points being the vertices of the chain. Then we compute the convex hull of the monotone chain (by using the convex hull algorithm of Section 4). The convex hull and the monotone chain, together, partition the region bounded by the convex hull into a set of one-sided monotone polygons. Triangulating a point set in the plane sorted by polar angle with respect to a polar point q can be reduced to that of triangulating the interior and the exterior of a star-shaped polygon, which can be done optimally in a way similar to the triangulation algorithm for a star-shaped polygon.

Given n values $w_1, w_2, \ldots, w_n$, the *all dominating neighbors problem*, in fact, can be viewed as a rectilinear version of triangulating a monotone polygonal chain. That is, we reduce the n values into n points $(1, w_1), (2, w_2), \ldots, (n, w_n)$, and let a rectilinear monotone polygonal chain C have the n points as part of its vertices. The chain C consists of only vertical and horizontal line segments. All the "common tangents" we compute, and all the "diagonals" we add to the "triangulation", are horizontal line segments. Our algorithm in Section 5 can be modified to solve this problem.

Acknowledgement. The author is very grateful to Professor Mikhail Atallah for his great support and valuable suggestions to this work.

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlang and C. Yap. "Parallel computational geometry." *Algorithmica* 3 (3) (1988), 293–327.

[2] M. Atallah, R. Cole and M. Goodrich. "Cascading divide-and-conquer: a technique for designing parallel algorithm." *SIAM J. Comput.* 18 (3) (1989), 499–532.

[3] M. J. Atallah, D. Z. Chen and H. Wagener. "An optimal parallel algorithm for the visibility of a simple polygon from a point." TR 88-759, Dept. of Comput. Sci., Purdue Univ., April 1988. To appear in *Journal of the ACM*. A preliminary version appeared in *Proc. 5th Annual ACM Symp. on Computational Geometry*, 1989, pp. 114–123.

[4] M. J. Atallah and M. T. Goodrich. "Efficient parallel solutions to some geometric problems." *J. Parallel & Distributed Comput.* 3 (1986), 492–507.

[5] M. J. Atallah and M. T. Goodrich. "Parallel algorithms for some functions of two convex polygons." *Algorithmica* 3 (1988), 535–548.

[6] O. Berkman, D. Breslauer, Z. Galil, B. Schieber and U. Vishkin. "Highly parallelizable problems (Extended abstract)." *Proc. 21st Annual ACM Symp. on Theory of Computing*, 1989, pp. 309–319.

[7] G. Bilardi and A. Nicolau. "Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines." *SIAM J. Comput.* 18 (1989), 216–228.

[8] R. Cole. "Parallel merge sort." *SIAM J. Comput.* 17 (4) (1988), 770–785.

[9] R. Cole and M. T. Goodrich. "Optimal parallel algorithms for polygon and point-set problems." *Proc. 4th Annual ACM Symp. Computational Geometry*, 1988, pp. 211–220. (To appear in *Algorithmica*.)

[10] R. Cole and U. Vishkin. "Approximate parallel scheduling. Part I: the basic technique with applications to optimal parallel list ranking in logarithmic time." *SIAM J. Comput.* 17 (1) (1988), 128–142.

[11] A. Fournier and D. Y. Montuno. "Triangulating simple polygons and equivalent problems." *ACM Trans. Graphics* 3 (2) (1984), 153–174.

[12] M. T. Goodrich. "Finding the convex hull of a sorted point set in parallel." *Inform. Process. Lett.* 26 (1987/88), 173–179.

[13] M. T. Goodrich. "Triangulating a polygon in parallel." *J. of Algorithms* 10 (1989), 327–351.

[14] R. L. Graham. "An efficient algorithm for determining the convex hull of a finite planar set." *Inform. Process. Lett.* 1 (1972), 132–133.

[15] R. L. Graham and F. F. Yao. "Finding the convex hull of a simple polygon." *J. Algorithms* 4 (4) (1983), 324–331.

[16] M. R. Garey, D. S. Johnson, F. P. Preparata and R. E. Tarjan. "Triangulating a simple polygon." *Inform. Process. Lett.* 7 (4) (1978), 175–179.

[17] T. Hagerup, and C. Rub. "Optimal merging and sorting on the EREW PRAM." *Inform. Process. Lett.* 33 (1989), 181–185.

[18] D. G. Kirkpatrick and T. Przytycka. "An optimal parallel algorithm for all dominating neighbors problem and its applications." Manuscript.

[19] C. P. Kruskal, L. Rudolph and M. Snir. "The power of parallel prefix." *IEEE Trans. Comput.* C-34 (1985), 965–968.

[20] R. E. Ladner and M. J. Fischer. "Parallel prefix computation." *J. of the ACM* 27 (4) (1980), 831–838.

[21] D. T. Lee. "On finding the convex hull of a simple polygon." *Int'l J. Comput. and Inform. Sci.* 12 (2) (1983), 87–98.

[22] D. T. Lee and F. P. Preparata. "Computational geometry — A survey." *IEEE Trans. Comput.* C-33 (12) (1984), 1072–1101.

[23] D. T. Lee and F. P. Preparata. "An optimal algorithm for finding the kernel of a polygon." *Journal of the ACM* 26 (1979), 415–421.

[24] C. J. Lee and W. J. Hsu. "Triangulating a star-shaped polygon in parallel." To appear in *Proc. 28th Annual Allerton Conf. on Communication, Control, and Computing*, Monticello, Illinois, 1990.

[25] E. Merks. "An optimal parallel algorithm for triangulating a set of points in the plane." *International Journal of Parallel Programming* Vol. 15, No. 5 (1986), 399–411.

[26] R. Miller and Q. F. Stout. "Efficient parallel convex hull algorithms." *IEEE Trans. Comput.* C-37 (12) (1988), 1605–1618.

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

[28] W. Paul, U. Vishkin and H. Wagener. "Parallel dictionaries on 2-3 trees." *Proc. 10th Coll. on Autom., Lang., and Prog. (ICALP), LNCS 154*, Springer, Berlin, 1983, pp. 597–609.

[29] H. Wagener. "Parallel Computational Geometry Using Polygonal Order." Ph.D. thesis, Technical University of Berlin, FRG, 1985.

[30] H. Wagener. "Triangulating a monotone polygon in parallel." *Computational Geometry and Its Applications, Proc. International Workshop on Computational Geometry (CG'88)*, Wurzburg, FRG, 1988, pp. 136–147.

[31] T. C. Woo and S. Y. Shin. "A linear time algorithm for triangulating a point-visible polygon." *ACM Trans. Graphics* 4 (1) (1985), 60–70.

[32] C. A. Wang and Y. H. Tsin. "An $O(\log n)$ time parallel algorithm for triangulating a set of points in the plane." *Inform. Process. Lett.* 25 (1987), 55–60.

[33] C. K. Yap. "Parallel triangulation of a polygon in two calls to the trapezoidal map." *Algorithmica* 3 (1988), 279–288.