

Purdue University

Purdue e-Pubs

Department of Computer Science Technical
Reports

Department of Computer Science

1987

Triangulating A Polygon in Parallel

Michael T. Goodrich

Report Number:

87-679

Goodrich, Michael T., "Triangulating A Polygon in Parallel" (1987). *Department of Computer Science Technical Reports*. Paper 590.

<https://docs.lib.purdue.edu/cstech/590>

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

TRIANGULATING A POLYGON IN PARALLEL

Michael T. Goodrich

CSD-TR-679

May 1987

Triangulating a Polygon in Parallel

Michael T. Goodrich*

Department of Computer Sciences

Purdue University

West Lafayette, IN 47907

Abstract

In this paper we present an efficient parallel algorithm for polygon triangulation. The algorithm we present runs in $O(\log n)$ time using $O(n)$ processors, which is optimal if the polygon is allowed to contain holes. This improves the previous parallel complexity bounds for this problem by a $\log n$ factor. If we are also given a trapezoidal decomposition of the polygon as input, then we can triangulate the polygon in $O(\log n)$ time using only $O(n/\log n)$ processors. This immediately implies that we can triangulate a monotone polygon in $O(\log n)$ time using $O(n/\log n)$ processors, which is optimal. All of our results are for the CREW PRAM computational model. **Keywords.** Parallel algorithms, computational geometry, polygon triangulation, trapezoidal decomposition.

1 Introduction

The polygon triangulation problem is the following: we are given an n -vertex simple polygon P , which may contain holes, and wish to augment P with diagonal edges so that each interior face of the resulting subdivision is a triangle (see Figure 1). This problem arises in many applications, including computer graphics, image analysis, and robotics (see [22,25]). Guibas *et al.* [20] have shown that several problems, including shortest paths in a polygon, visibility in a polygon, hierarchical decomposition of a polygon, and the "convex ropes" of a polygon, are all linear-time reducible to polygon triangulation. The most efficient sequential algorithms for polygon triangulation are (i) an algorithm by Chazelle and Incerpi [9], which runs in $O(n \log s)$ time, where s measures the *sinuosity*

*This research was supported by the Office of Naval Research under Grants N00014-84-K-0502 and N00014-86-K-0689, and the National Science Foundation under Grant DCR-84-51393, with matching funds from AT&T.

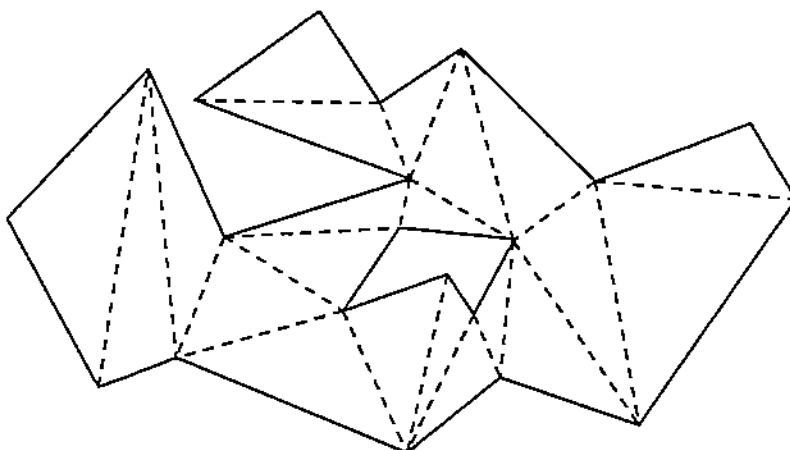


Figure 1: The polygon triangulation problem.

of the polygon, i.e., the number of times the polygon alternates between spirals of opposite orientation, and (ii) an algorithm by Tarjan and Van Wyk [28], which runs in $O(n \log \log n)$ time. In addition, it is well known, from the work of Fournier and Montuno [16], that polygon triangulation is linear-time reducible to trapezoidal decomposition. In the trapezoidal decomposition problem we wish to augment a polygon P with vertical line segments interior to P such that each segment passes through a vertex and the set of segments partitions the interior of P into trapezoids.

Since polygon triangulation has so many applications, it is natural that we wish to solve it as fast as possible. We seem to be at the limit of what can be achieved by sequential processing, however. Thus, we are interested in exploring what kinds of speed-ups can be achieved through parallel processing. More precisely, we are interested in minimizing the product TP , where T is the time and P is the number of processors used by the algorithm. Given that the product TP is as small as possible then our secondary goal is to minimize T . If the product TP matches the sequential lower bound for a problem, then we say that the algorithm is *optimal*, since a single processor can simulate the algorithm in $O(TP)$ time. The parallel model we choose for this work is the Concurrent-Read, Exclusive-Write Parallel RAM (or CREW PRAM). Recall that this is the synchronous parallel model in which processors share a common memory which allows for concurrent reads from any memory location, but no two processors may simultaneously write to the same location.

There are a number of algorithmic techniques which have proven useful for computational geometry problems in this model [1,2,5,6,7,10,14,15,18,19,23,30]. One technique, as presented in [1,2,6,7,19], is a variation on the divide-and-conquer paradigm. As opposed to simply dividing a

problem into a small number of subproblems, the idea here is to divide it into many subproblems, say into \sqrt{n} problems of size $O(\sqrt{n})$ each. One then solves each subproblem recursively in parallel, and merges all the subproblems quickly in parallel (say in $O(\log n)$ time). This many-way divide-and-conquer technique can often be used in conjunction with another fundamental parallel technique, which we call *sequential subsets*, in which we “stop” the recursion early (say when the subproblems are all of size $O(\log n)$) and solve all the subproblems sequentially, one processor per subproblem [8,12]. This often results in a savings in the processor bounds (usually by a factor of $\log n$ or $\log^2 n$).

Another technique which is fundamental to this area is the *parallel prefix* technique. This technique makes use of the “recursive doubling” paradigm [29] to solve the following problem: given a list (a_1, a_2, \dots, a_n) of integers compute all the prefix sums $c_k = \sum_{i=1}^k a_i$, for $k \in \{1, 2, \dots, n\}$. This can be done in $O(\log n)$ time using $O(n/\log n)$ processors. (See [21] for details.) There are a host of problems that can be reduced to the parallel prefix problem, or at least need a parallel prefix computation to be solved as a subroutine.

The previous parallel algorithm for polygon triangulation runs in $O(\log^2 n)$ time using $O(n)$ processors in the CREW PRAM model [1]. In this paper we present a parallel algorithm for polygon triangulation which runs in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM model. These bounds are optimal if the polygon is allowed to contain holes, since, as Asano, Asano and Pinter have shown [3], polygon triangulation has a sequential lower bound of $\Omega(n \log n)$ in this case. We divide our polygon triangulation procedure into three phases, each of which decomposes the polygon into subpolygons which have a “simpler” structure than the polygons in the previous phase. With the exception of the first step of the first phase, which is trapezoidal decomposition, our algorithm runs in $O(\log n)$ time using only $O(n/\log n)$ processors. This provides a parallel analog of the sequential linear-time reduction of triangulation to trapezoidal decomposition by Fournier and Montuno [16], since our reduction has a linear TP product. (We have recently discovered that Aggarwal *et al* improved their triangulation algorithm in the final version of their paper [2] so that it runs faster given the polygon’s trapezoidal decomposition. Their method runs in $O(\log n)$ time using $O(n)$ processors, which is still a $\log n$ factor from our TP product. We have also learned that Yap [30] has a parallel triangulation method which runs in these bounds and makes two calls to trapezoidal decomposition.)

We use the parallel prefix and sequential subsets techniques in each of the three phases. In fact, we use a generalized version of the sequential subsets technique, in which there can be a large number of differing sized “small” subproblems (possibly even $O(n)$ of them). In the most difficult

phase, Phase 2, we also make use of the many-way divide-and-conquer technique. We do not apply it in the standard way, however, for that would not result in an efficient processor bound. Instead, we “pipeline” the sequential subsets paradigm through every recursive call (not just the last one), and use a parallel data structure, which we call the *HQ-tree* [18,19], to aid us in quickly merging subproblem solutions.

We present some preliminary definitions and observations in Section 2, including the introduction of the HQ-tree data structure and an overview of our triangulation algorithm. In Sections 3, 4, and 5 we present phases 1, 2, and 3, respectively, of our triangulation algorithm.

2 Preliminaries

We make some definitions and observations we will be using throughout this paper. If p is a point in the plane, then we let $x(p)$ and $y(p)$ denote the x - and y -coordinate of p , respectively. Let $C = (v_1, v_2, \dots, v_n)$ be a simple polygonal chain. The *convex hull* of C is defined to be the smallest convex region containing C . We let $CH(C)$ denote the vertices of C which are on the boundary of the convex hull of C listed in clockwise order. The list $CH(C)$ can be decomposed into two sublists $LH(C)$ and $UH(C)$, where $LH(C)$ (resp. $UH(C)$) denotes the maximal subchain C' of $CH(C)$ such that all the vertices of C are either on or above (resp. below) C' , relative to some y -axis. We call $LH(C)$ the *lower hull* of C and $UH(C)$ the *upper hull* of C . If there is a line L such that every line perpendicular to L intersects C in at most one point, then we say that C is *monotone* with respect to L . The chain C is *convex* if C can be made into a convex polygon by adding the edge $v_n v_1$. We say that C is a *lower-hemispheric* chain (resp., an *upper-hemispheric* chain) if $C = LH(C)$ (resp., $C = UH(C)$). Note that if C is a hemispheric chain, then it must be convex and monotone with respect to the x -axis. Intuitively, C is lower-hemispheric if one always make “left turns” when traversing C from left to right, and upper-hemispheric if one always makes “right turns.”

Let B be a binary tree. We define the height of B , denoted $height(B)$, to be the length of the longest leaf-to-root path in B . Let π be a leaf-to-root path. We say that a node v belongs to the *left fringe* (resp. *right fringe*) of π if v is not on π and is the left child (resp. right child) of a node on π .

The *HQ-tree* is a data structure which can be used to efficiently manipulate hemispheric chains in parallel. Let $C = (v_1, v_2, \dots, v_n)$ be a convex chain monotone with respect to the x -axis. Without loss of generality, we assume that $x(v_i) < x(v_{i+1})$ and that C is an upper-hemispheric chain. We

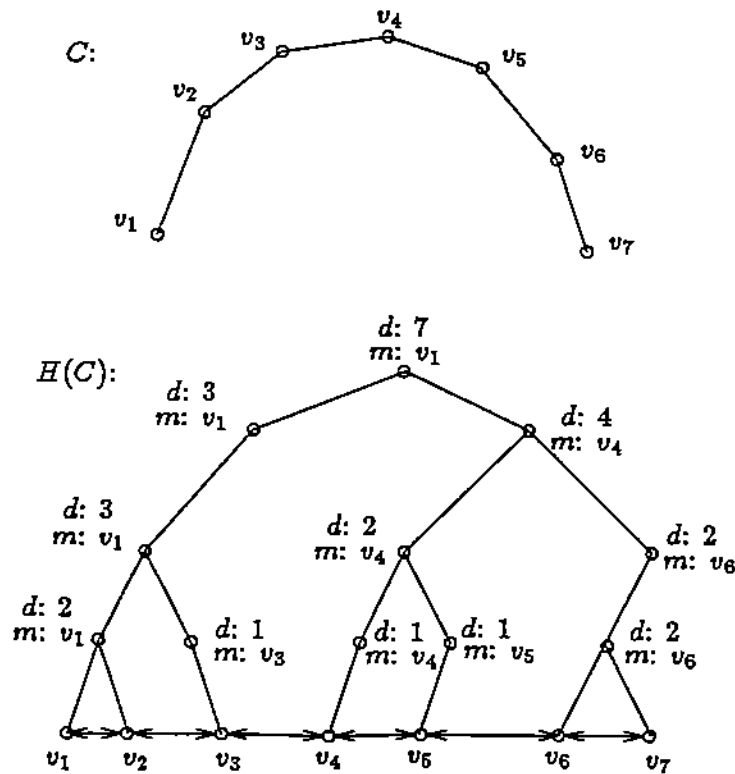


Figure 2: An example HQ-tree $H(C)$ for an upper-hemispheric chain C . The d and m labels are given for each internal node, and the $succ$ and $prev$ pointers are denoted by arrows at the leaves.

define the HQ-tree data structure $H(C)$ as follows. It is a binary search tree which stores the vertices of C in its leaf nodes, all of which are at the same level in the tree, sorted from left to right by increasing x -coordinates. For simplicity of expression, for each leaf node v we also let v denote the vertex in C associated with this node. With each leaf v we store two labels $prev(v)$ and $succ(v)$ which are, respectively, the predecessor and successor points of v in C (i.e., $prev(v_i) = v_{i-1}$ and $succ(v_i) = v_{i+1}$). If the predecessor (resp., successor) of v is undefined then we take $prev(v)$ (resp., $succ(v)$) to be nil. For each internal node v in T we let $Desc(v)$ denote the set of descendant leaves of v . With each internal node v in T we store two labels $d(v)$ and $m(v)$ which are, respectively, the number of vertices in $Desc(v)$ and a pointer to the vertex (leaf node) in $Desc(v)$ with minimum x -coordinate. (See Figure 2.) These pointers and labels allow for us to perform a variety of operations on hemispheric chains efficiently in parallel.

In the following lemmas we study some of the properties of HQ-trees. Given two upper-hemispheric chains C_1 and C_2 , recall that the *common upper tangent* of C_1 and C_2 is the tangent

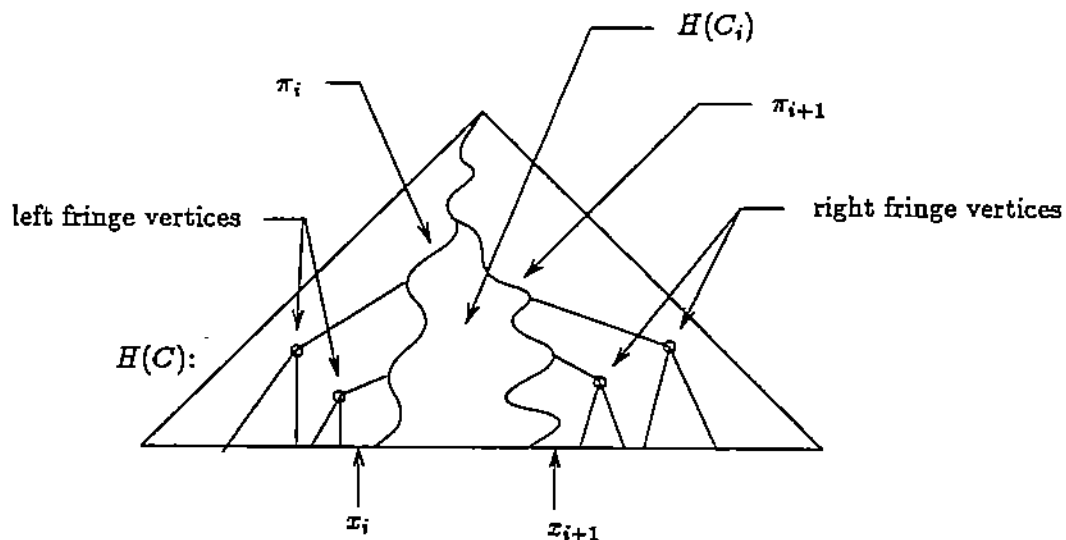


Figure 3: The k -way split operation.

line T such that none of the vertices of C_1 or C_2 are above T . The next lemma shows that HQ-trees can be used to efficiently find the common tangent of two upper-hemispheric chains.

Lemma 2.1: [18,19]: *Given HQ-trees $H(C_1)$ and $H(C_2)$, representing two upper-hemispheric chains C_1 and C_2 separable by a vertical line, we can find the common upper tangent of C_1 and C_2 in $O(h)$ time using a single processor, where $h = \text{height}(H(C_1)) + \text{height}(H(C_2))$.*

Proof sketch: The method is based on the binary search method of Overmars and Van Leeuwen [24] for finding the common upper tangent between two upper-hemispheric chains. The proof follows from the fact that the binary tree structure of HQ-trees and the labels *pred*, *succ*, *d*, and *m* can be used to exactly mimic their binary search method in $O(h)$ time. ■

Besides finding common tangents, we also need to be able to split hemispherical chains into smaller chains as well as being able to concatenate chains together. In the next lemma we show how to quickly perform a k -way split operation on an HQ-tree. That is, given a hemispheric chain C represented in some HQ-tree $H(C)$ and k vertical lines, we show how to construct HQ-tree representations of all the hemispheric chains which would be left if we "cut" C by the k lines.

Lemma 2.2: *Let $H(C)$ be an HQ-tree representing some hemispheric chain C . Given a sorted list (x_1, x_2, \dots, x_k) of real numbers, we can split $H(C)$ into $k + 1$ HQ-trees $H(C_0), H(C_1), \dots, H(C_k)$ such that all the vertices of each C_i have x -coordinates in the interval $[x_i, x_{i+1}]$ for $i \in \{0, 1, 2, \dots, k\}$*

(define $x_0 = -\infty$ and $x_{k+1} = +\infty$). Moreover, this construction can be done in $O(h)$ time using $O(k)$ processors, where $h = \text{height}(H(C))$.

Proof: The method is for each processor $i \in \{0, 1, \dots, k+1\}$ to traverse a root-to-leaf path π_i in $H(C)$ by searching for x_i , using the m labels of internal nodes and the x -coordinates of the vertices pointed to by the m labels to direct the search. As it traverses this path it copies every node v it visits into a new location in memory. It copies all the pointer information stored at v , as well, unless the pointer points to a child on the left fringe of π_i . That is, it copies v and then tests to see if the next node in π_i is $lchild(v)$ or $rchild(v)$. If the next node is $lchild(v)$, then processor i copies both $lchild(v)$ and $rchild(v)$ into the memory record for the new copy of v . If the next node is $rchild(v)$, then processor i only copies $rchild(v)$ into the new memory record for v and sets $lchild(v)$ of this record to nil. Once processor i completes this traversal, and reaches the leaf level of $H(C)$, it then repeats this root-to-leaf search procedure, this time traversing a path π_{i+1} by searching for x_{i+1} . In traversing this path it copies all nodes it visits into a new memory location, as it did while traversing π_i , except this time it doesn't copy pointers to any children on the right fringe. That is, with each node v it copies it also copies all the pointers stored at v , unless the pointer points to a child on the right fringe of π_{i+1} . (See Figure 3.) Once the processor completes these two traversals it updates the *pred* and *succ* pointers of the first and last elements in the resulting tree, so that the *pred* pointer for the first element and the *succ* pointer for the last element are both nil. Finally, processor i backtracks along each of the paths π_i and π_{i+1} updating the m and d labels of internal nodes along each of these paths, so they are based only on the elements left in the (copied) tree. This method clearly takes at most $O(\text{height}(H(C)))$ time using $O(k)$ processors. ■

The following lemma shows that we can perform an analogous k -way concatenate operation efficiently in parallel as well. That is, given a collection of HQ-trees representing hemispherical chains separated by vertical lines, and such that the concatenation of these chains is itself a hemispherical chain, then we can efficiently construct in parallel an HQ-tree representing the concatenation of these chains.

Lemma 2.3: Let $H(C_1), H(C_2), \dots, H(C_k)$ be a collection of HQ-trees such that the vertices in C_i all have x -coordinate less than the vertices in C_{i+1} , and the concatenation C of all the hemispheric chains C_1, \dots, C_k is itself a hemispheric chain. Then we can construct an HQ-tree $H(C)$ representing the concatenated chain C in $O(h + \log k)$ time using $O(k)$ processors in the CREW PRAM model. The resulting tree has height at most $h + \lceil \log k \rceil$, where h is the maximum height of any $H(C_i)$.

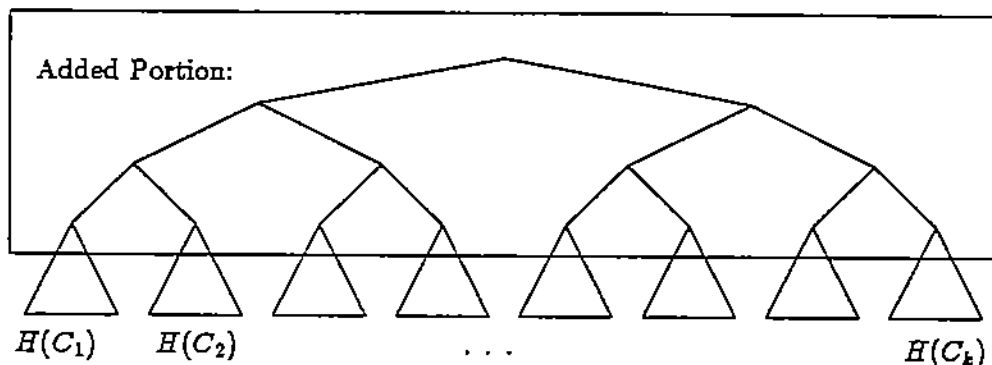


Figure 4: The k -way concatenation operations.

Proof: Let $C = C_1 \oplus C_2 \oplus \dots \oplus C_k$, where $A \oplus B$ denotes the concatenation of two lists. We construct an HQ-tree $H(C)$ by the following method. We compute the value of h , the maximum height of any $H(C_i)$, and augment each $H(C_i)$ by repeatedly adding a parent to the root of $H(C_i)$ until it has height h . We then build a complete binary tree "on top" of the $H(C_i)$'s (that is, each leaf of this tree is the root of an $H(C_i)$). (See Figure 4.) If we build this tree in parallel level-by-level starting with the leaves associated with each $H(C_i)$, then it is an easy matter to be assigning the m and d labels for the new internal nodes as we go. This new HQ-tree clearly has height at most $\lceil \log k \rceil + h$. The total time is clearly $O(h + \log k)$ since we have $O(k)$ processors at our disposal. ■

In the next section we give an overview of our triangulation algorithm and in the subsequent sections we show how to implement each of the three decompositions phases efficiently in parallel.

In the remainder of this section we present an overview of our triangulation algorithm. In the three sections which follow we present phases 1, 2, and 3 of our algorithm, respectively. In the first phase we decompose P into polygons which are one-sided and monotone with respect to the x -axis. We say that a polygon P is *one-sided* if there is a distinguished edge s on P such that the vertices of P are all above (or all below) s (except for the endpoints of the edge). (See Figure 5.a.) A polygon P is *monotone* if there is a line L such that each perpendicular of L intersects P in at most two points. This first phase runs in $O(\log n)$ time using $O(n/\log n)$ processors, if we are given the trapezoidal decomposition, and $O(n)$ processors, otherwise. In the second phase we decompose each of the one-sided monotone polygons into monotone funnel polygons in parallel. We say that a polygonal chain is a *funnel* if its boundary consists of a single edge followed by a convex chain followed by a single edge followed by another convex chain (see Figure 5.b). This is the most difficult of the three phases, and the method we use to implement this step utilizes the HQ-tree data structure as well as the many-way divide-and-conquer technique. This phase runs in $O(\log n)$

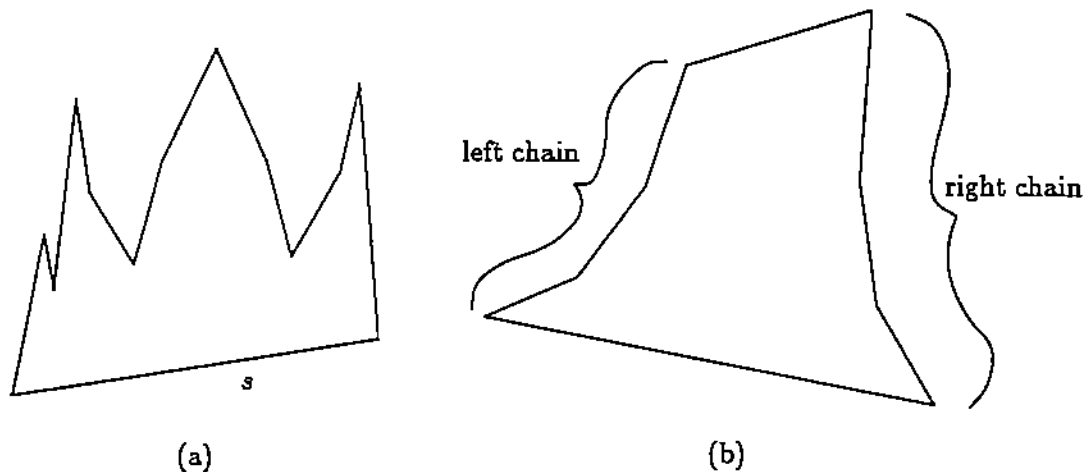


Figure 5: Simple-structure polygons. Figure (a) illustrates a one-sided monotone polygon and figure (b) illustrates a monotone funnel polygon.

time using $O(n/\log n)$ processors. Finally, in the third phase we triangulate each of the funnel polygons. We show that parallel merging can be used to implement this step in $O(\log n)$ time using $O(n/\log n)$ processors. Thus, the entire triangulation computation requires $O(\log n)$ time using $O(n/\log n)$ processors, if we are given the trapezoidal decomposition, and $O(n)$ processors, otherwise.

We show how to perform the first phase of our triangulation algorithm in the following section.

3 Decomposition into One-Sided Monotone Polygons

Let P be a simple polygon which may contain holes. (One way to represent P is as a list of vertices and a list of edge segments joining pairs of vertices.) We assume that for each edge segment s of P we are given which side of s is in the interior of P . As mentioned above, the first phase in our triangulation algorithm is to decompose P into subpolygons which are one-sided and monotone with respect to the x -axis. The algorithm PHASE-ONE which follows performs this first phase of our triangulation procedure. Before presenting the algorithm we make the following definitions. Given a vertex v , we say that the edge segment s is a *trapezoidal segment* of v if the vertical line segment from v to s is entirely interior to P (hence, does not cross any other segment of P). We call the point q on s such that $x(q) = x(v)$ the *vertical shadow* of q on s . Note that a vertex can have zero, one, or two vertical shadows. (See Figure 6. The figure illustrates the general trapezoidal decomposition problem, when the n line segments don't necessarily form a simple polygon.)

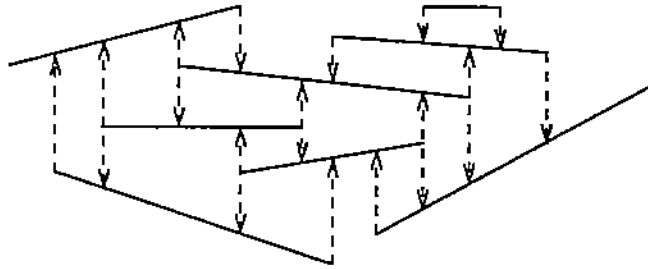


Figure 6: A trapezoidal decomposition.

Algorithm PHASE-ONE:

Input: A simple polygon P which may contain holes. For simplicity, we assume that the vertices in P have distinct x -coordinates. It is straightforward to generalize our results to the general case.

Output: A decomposition of P into one-sided monotone polygons.

Step 1. If we are given the trapezoidal decomposition of P , then skip to Step 2. Otherwise, construct a trapezoidal decomposition for P . After performing this construction we will have an adjacency list representing the decomposition. That is, we will have a graph $G = (V, E)$ such that each vertex and vertical shadow is in V and there is an edge between v and w in V if v and w are adjacent in the decomposition (i.e., there is a line segment in the decomposition which joins v and w and contains no other vertices in V). This step can be performed in $O(\log n)$ time using $O(n)$ processors [4,19]. Let us continue the discussion assuming this procedure is correct and runs in these bounds.

Step 2. For each edge segment s in P construct a sorted list V_s of the vertices of P which have a vertical shadow on s , sorted by increasing x -coordinates. Since the trapezoidal decomposition gives us the adjacencies in V_s , i.e., the vertical shadows on any segment s form a simple linked-list structure in the trapezoidal decomposition, this step can be implemented by a list-ranking procedure. More specifically, let G' be the subgraph of G which is formed by removing all the nodes in G which correspond to vertices of P . Then the graph G' is actually just a collection of linked lists (one for every edge segment of P which contains vertical shadows from some vertices). Thus, we can treat G' as a single linked list (with many of the pointers begin nil) and rank all the nodes in G' , computing for each node $v \in G'$ the distance from v to the nearest nil pointer. This ranking procedure can be performed in $O(\log n)$ time using $O(n/\log n)$ processors by an algorithm by Cole and Vishkin [13], since there are $O(n)$ vertical shadows in all (at most two per vertex). This will give us for each segment s on P and each vertical shadow v on s the number of vertical shadows which precede v on s . It is then an easy matter to construct each V_s in parallel from this

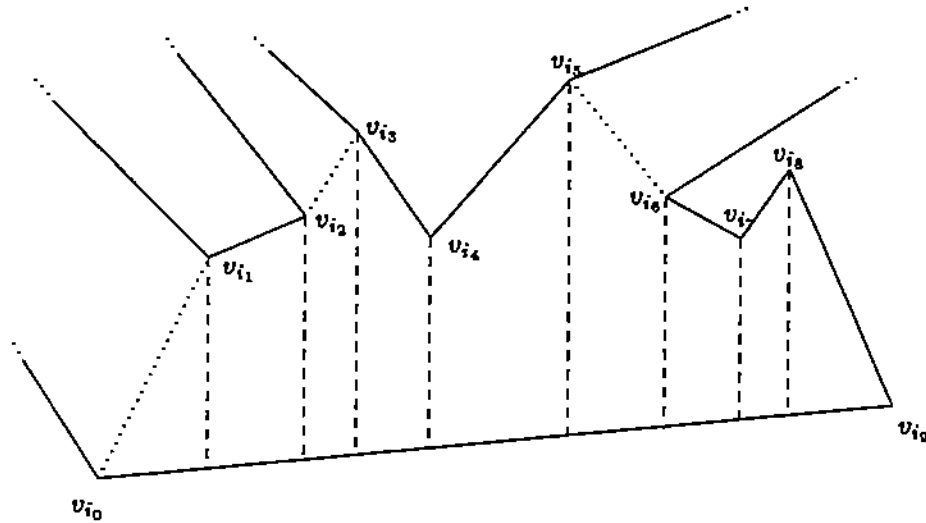


Figure 7: A one-sided monotone polygon formed by the decomposition. The figure shows a polygon P_s for $s = (v_{i_0}, v_{i_0})$ and $V_s = (v_{i_1}, \dots, v_{i_8})$. The edges in P_s but not in P are shown dotted. Note that the sequence $v_{i_0}, v_{i_1}, v_{i_2}, \dots, v_{i_8}$ is monotone in the x -direction.

information in $O(\log n)$ time using $O(n/\log n)$ processors (using the sequential subsets technique).

Step 3. Let $V_s = (v_{s,1}, v_{s,2}, \dots, v_{s,n_s})$ be the list of vertices constructed in Step 2 for the edge segment s . Augment P by adding an edge from $v_{s,i}$ to $v_{s,i+1}$ if it is not already an edge of P . (See Figure 7.) We show below that this decomposes P into a collection of one-sided monotone polygons P_s . This step can clearly be performed for all the V_s 's in parallel in $O(\log n)$ time using $O(n/\log n)$ processors, using the sequential subsets technique.

End of algorithm PHASE-ONE.

We analyze this algorithm in the following theorem:

Theorem 3.1: *Given a simple polygon P , which may contain holes, we can decompose P into one-sided monotone polygons in $O(\log n)$ time using $O(n/\log n)$ processors, if we are given the trapezoidal decomposition of P , and $O(n)$ processors, otherwise, in the CREW PRAM model.*

Proof: First, note that the algorithm PHASE-ONE constructs a decomposition. That is, an edge added to P while performing Step 3 for some edge segment s_1 may coincide with an edge added for some edge segment s_2 , but it will not cut across any other edge. This is because we only add an edge between two vertices v and w when v and w belong to the same trapezoid in the decomposition. Second, the vertices of V_s are all on the same side of s , because the vertical line segment from any point in V_s to the segment s must be interior to P , and the interior of P can only be on one side

of s . Thus, each P_s is one-sided. Finally, each P_s is monotone because we sorted the points in V_s by x -coordinate in Step 2.

The complexity bounds for PHASE-ONE follow from observations made above in the discussion.

■

After decomposing P into one-sided monotone polygons, we decompose P further into a collection of monotone funnel polygons. We describe the method for doing this efficiently in parallel in the following section.

4 Decomposition into Monotone Funnel Polygons

The second phase of our triangulation algorithm decomposes all the one-sided monotone polygons P_s into monotone funnel polygons in parallel. Since we only have $O(n/\log n)$ processors, we must first perform an application of the sequential subsets technique. We divide the collection of polygons P_s into two groups: (i) those polygons with less than $\log n$ vertices and (ii) those polygons with more than $\log n$ vertices. Those polygons in group (i) we triangulate sequentially in $O(\log n)$ time [16] and the ones in group (ii) we decompose into monotone funnel polygons using the method described later in this section. Before we describe the general method we must first explain how to solve the processor assignment problem for the polygons in group (i), since there may be $O(n)$ of them. We group all the P_s 's with $|P_s| \in [1, 2]$ into groups containing $\frac{1}{2} \log n$ polygons, all the P_s 's with $|P_s| \in [2, 4]$ into groups of size $\frac{1}{4} \log n$, all the P_s 's with $|P_s| \in [4, 8]$ into groups of size $\frac{1}{8} \log n$, and so on, so that each group contains $O(\log n)$ vertices. This grouping step can be performed in $O(\log n)$ using $O(n/\log n)$ processors using an algorithm by Reif [26] for sorting $O(n)$ integers in the range $[1, \log n]$. We can then assign a single processor to each group and triangulate all the polygons in the group sequentially in $O(\log n)$ time [17]. Since this completes the computation for these one-sided monotone polygons, for the remainder of this section we assume that each P_s has more than $\log n$ vertices.

Since the computation which follows is to be performed for each one-sided monotone polygon P_s in parallel, let us concentrate on the problem of decomposing a single one-sided monotone polygon into monotone funnel polygons. To simplify the notation, let N denote the number of vertices in the original polygon, and let $P = (v_1, v_2, \dots, v_n, v_1)$ be the one-sided monotone polygon which we wish to decompose, where $v_n v_1$ is the distinguished edge of P . Without loss of generality, let us assume that P is monotone in the x -direction and the vertices not on the distinguished edge $s = v_n v_1$ are all above s . We will show how to decompose P into monotone funnel polygons in

$O(\log N)$ time using $O(n/\log N)$ processors.

We define the *underside* of a monotone convex chain C to be the region R between C and $LH(C)$. Note that R need not be connected. We define the *topside* of C analogously. We decompose P into funnel polygons using the HQ-tree data structure and the many-way divide-and-conquer technique. This second phase is the most complicated of the three phases. In the recursive algorithm which follows we show how, given a polygonal chain C monotone with respect to the x -axis, we can decompose the underside of C (i.e., the regions of the plane between C and $LH(C)$) into monotone funnel polygons represented implicitly by HQ-trees. We can use HQ-trees in this case because a monotone funnel polygon is uniquely defined by two hemispheric chains (its left chain and its right chain). We call this procedure initially with $C = (v_1, v_2, \dots, v_n)$, i.e., the polygonal chain formed by removing the distinguished edge $s = v_n v_1$ from P . Each funnel polygon of the decomposition is represented by two HQ-trees—one for the left convex chain and one for the right convex chain which define the funnel polygon. We also construct the lower hull $LH(C)$ of C represented by an HQ-tree $H(LH(C))$. Since we call the procedure with $C = (v_1, v_2, \dots, v_n)$, hence $LH(C)$ is just the line $v_1 v_n$, one may ask why we need to output a representation of the lower hull of C . We do this because it may be the case that $LH(C)$ is a non-trivial lower hull in a recursive call. After the procedure returns we construct array representations of each funnel polygon from the HQ-tree representations in a post-processing step. The procedure also takes an integer parameter d , which we set to $\lceil \log N \rceil$, and never change. We will show later that the algorithm PHASE-TWO presented below runs in $O(\log n + d + \log d \log \log n)$ time using $O(n/d)$ processors in the CREW PRAM model. Thus, with $d = \lceil \log N \rceil$ we can implement PHASE-TWO in $O(\log N)$ time using $O(n/\log N)$ processors.

Algorithm PHASE-TWO(C, d):

Input: A polygonal chain $C = (v_1, v_2, \dots, v_n)$ which is monotone with respect to the x -axis, and integer $d > 0$.

Output: An HQ-tree $H(LH(C))$ representing the vertices belonging to the lower convex hull of C , sorted by increasing x -coordinate, and a decomposition of the underside of C (i.e., the region bounded from above by C and from below by $LH(C)$) into funnel polygons, each one represented by two HQ-trees (one for the left convex chain and one for the right convex chain defining the funnel).

Method: Since the method is rather involved, we first present a high-level description of the algorithm, and then show how to efficiently implement each of its constituent parts.

High-Level Description:

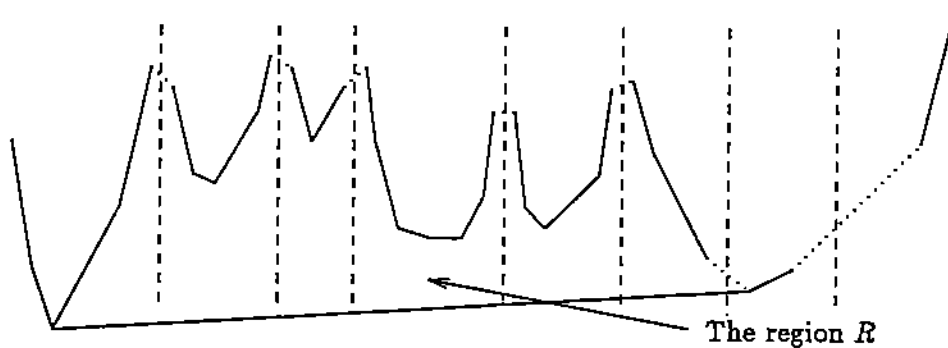


Figure 8: The untriangulated portion R .

Step 0. If $n \leq 4d$ then decompose the polygon into funnel polygons sequentially using a single processor in $O(d)$ time [17]. Also construct the lower hull $LH(C)$ of C and build an HQ-tree of height $\lceil \log n \rceil$ which represents it. Since this completes the algorithm for this case, we assume for the remainder of the algorithm that $n > 4d$.

Step 1. Divide C into $\sqrt{n/d}$ subchains $C_1, C_2, \dots, C_{\sqrt{n/d}}$ of size $O(\sqrt{nd})$ each, and call PHASE-TWO(C_i, d) for each C_i in parallel. When the parallel recursive call returns we will have an HQ-tree $H(LH(C_i))$ representing the lower hull of C_i for each C_i . We have yet to decompose the region between the lower hulls returned from the recursive call and the lower hull $LH(C)$ of C . Let R denote this region. (See Figure 8.)

Step 2. Build a complete binary tree B such that each leaf is associated with one of the C_i 's. For each internal node w in B find the common supporting tangent t_w between the hulls which are descendants of $lchild(w)$ and the hulls which are descendants of $rchild(w)$. (See Figure 9.) Let T denote the set of all tangent lines t_w .

Comment. We show below that the t_w 's decompose R into a collection of funnel polygons. That is, each t_w forms the base of a funnel polygon P_w , all of whose vertices are above t_w . The remainder of the algorithm is dedicated to constructing the HQ-trees representing these funnel polygons (two HQ-trees per funnel).

Step 3. For each $LH(C_i)$ construct the sorted list $X_i = (x_1, x_2, \dots, x_{k_i})$ of x -coordinates of all the intersections of $LH(C_i)$ with tangents in T . Perform a k_i -way split of $H(LH(C_i))$ using X_i , constructing HQ-trees $H_{i,0}, H_{i,2}, \dots, H_{i,k_i}$. The vertices in $H_{i,j}$ all have x -coordinates in the interval $[x_j, x_{j+1}]$.

Step 4. For each HQ-tree $H_{i,j}$ determine the funnel polygon P_w that the vertices of $H_{i,j}$ belong to. If the vertices of $H_{i,j}$ do not belong to any P_w (hence are in $LH(C)$), then we say

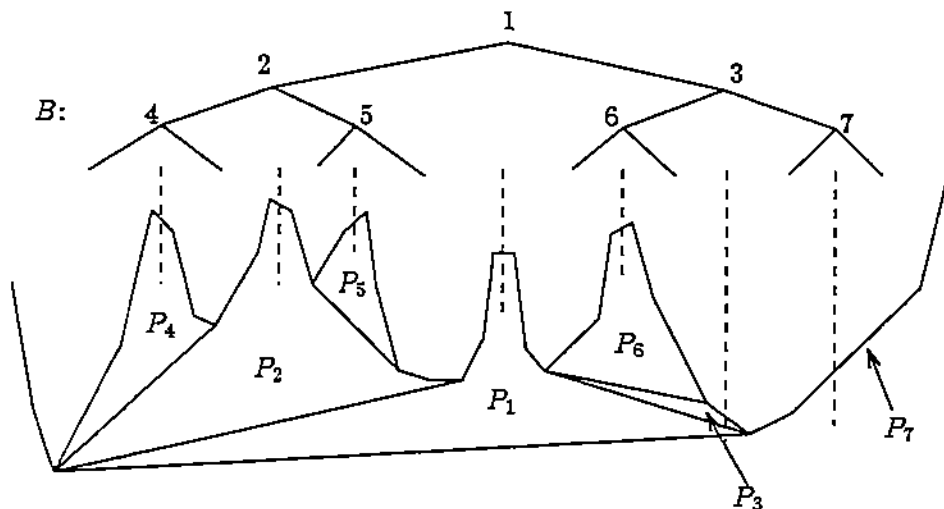


Figure 9: The monotone funnel polygons P_w formed by the decomposition. The figure illustrates the polygon P_w for each internal node $w \in B$. Note that the polygon P_3 is simply a triangle and the polygon P_7 is just a line segment.

that $H_{i,j}$ belongs to P . For each P_w in parallel sort the collection of HQ-trees defining P_w by the x -coordinates of the vertices they contain. Collect these HQ-trees into two groups: those belonging to the left convex chain defining P_w and those belonging to the right convex chain defining P_w . Finally, perform a k -way concatenation of the HQ-trees in each of these two groups, for each P_w in parallel.

End of High Level Description.

We show below that the algorithm PHASE-TWO can be implemented to run in $O(\log n + d + \log d \log \log n)$ time using $O(n/d)$ processors. We consider each of the four high-level steps in turn. The method for performing Steps 0 and 1 should be clear from the description given above, so we begin the discussion with the details for performing Step 2.

Details of Step 2. Recall that at the beginning of this step, we have already divided C into $\sqrt{n/d}$ subchains of size $O(\sqrt{nd})$ using vertical dividing lines and recursively called PHASE-TWO(C_i, d) on each subchain C_i in parallel. So at this stage in the algorithm we have an HQ-tree $H(LH(C_i))$ constructed for each subchain C_i , and this HQ-tree represents the lower hull of C_i . In this step we build a complete binary tree B such that each leaf i of B is associated with one of the HQ-trees $H(LH(C_i))$. Since there are $\sqrt{n/d}$ such trees this can clearly be done in $O(\log n)$ time using $O(\sqrt{n/d})$ processors. For each internal node w in B we let L_w denote the vertical line separating the polygonal chains which are descendants of $lchild(w)$ and the polygonal chains which are descendants of $rchild(w)$. The details for the remaining computations for Step 2 follow:

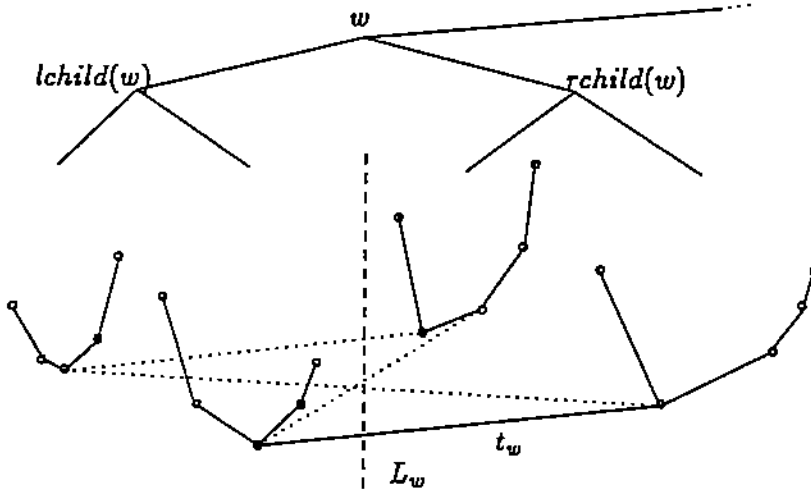


Figure 10: The tangent lines in T_w . The supporting tangent lines in T_w are each tangent to a hull in $lchild(w)$ and one in $rchild(w)$. The tangent t_w is shown as a solid line, and the others are shown dotted.

Step 2.1. For each pair (i, j) with $i, j \in \{1, 2, \dots, \sqrt{n/d}\}$ and $i < j$ **par**do

Compute the common supporting tangent $t_{i,j}$ of C_i and C_j
using the method of Lemma 2.1

Step 2.2. For each internal node w in B **par**do

Construct the set T_w of all tangents $t_{i,j}$ such
that $i \in Desc(lchild(w))$ and $j \in Desc(rchild(w))$;
Find the tangent t_w in T_w which has the lowest
intersection with L_w of all the tangents in T_w
(See Figure 10)

Note that t_w must be the common supporting tangent of the lower hull of the chains which are descendants of $lchild(w)$ and the lower hull of the chains which are descendants of $rchild(w)$. This is because t_w is chosen to be the “lowest” tangent between C_i and C_j with $i \in Desc(lchild(w))$ and $j \in Desc(rchild(w))$.

Analysis of Step 2. We have already noted that constructing the binary tree B can be done in $O(\log n)$ time using $O(\sqrt{n/d})$ processors. Lemma 2.1 implies that Step 2.1 runs in $O(h(\sqrt{n/d}))$ time using $O(n/d)$ processors, where $h(m)$ is the maximum height of any HQ-tree returned by PHASE-TWO when passed an m -vertex polygonal chain. Since the essential computation of Step 2.2 is computing a minimum of $|T_w|$ items for each w in parallel, and there are a total of $O(n/d)$ items in

all the T_w 's, we can clearly perform this step in $O(\log n)$ time using $O(n/d)$ processors. Thus, the entire Step 2 can be performed in $O(\log n + h(\sqrt{nd}))$ time using $O(n/d)$ processors in the CREW PRAM model.

Before we continue with the details of Steps 3 and 4 of the algorithm PHASE-TWO we show that the tangents t_w partition R , the region between $LH(C)$ and the $LH(C_i)$'s, into a collection of funnel polygons.

Lemma 4.1: *Let Γ be the planar subdivision determined by the region R and the tangents t_w . Then for any t_w the face of Γ immediately above t_w is a funnel polygon.*

Proof: Clearly, the claim is true for each node w in B with $height(w) = 1$, since t_w in this case is the common tangent between two lower-convex chains joined by a single edge. So, consider any node w with $height(w) > 1$. Clearly, the face above t_w is a monotone polygon, since C is a monotone chain. Let P_w be the polygon associated with this face. We can write P_w as $(v_{i_1}, \dots, v_{i_j}, v_{i_{j+1}}, \dots, v_{i_k}, v_{i_1})$, where $t_w = v_{i_k} v_{i_1}$ and $v_{i_j} v_{i_{j+1}}$ is the edge of C which crosses L_w . Note that the chain $(v_{i_1}, \dots, v_{i_j})$ must be convex, since if it were not convex then either one of the C_i 's is not convex or one of the common tangents t_z for some descendant node z is not actually a tangent. Similarly for the chain $(v_{i_{j+1}}, \dots, v_{i_k})$. Thus, P_w is a funnel polygon. ■

Having shown that each of the tangents t_w determines a funnel polygon P_w , we now show how to construct a representation of each P_w using HQ-trees. As mentioned above, we will use two HQ-trees to represent each funnel polygon, one for the left convex chain and one for the right convex chain determining the funnel.

Details of Step 3. Let T denote the set of all t_w 's computed in Step 2. Recall that in Step 3 we construct for each C_i the sorted list $X_i = (x_1, x_2, \dots, x_{k_i})$ of the x -coordinates of the intersections of $LH(C_i)$ with the tangents in T , and then perform a k_i -way split of the HQ-tree $H(LH(C_i))$ using this list. Let X be the set of all (i, x) pairs such that there is a t_w that intersects C_i at a vertex with x -coordinate equal to x . We construct the set X so that it is sorted lexicographically, and then construct each X_i by a simple parallel prefix computation. Using the method of Lemma 2.2 we split each $H(C_i)$ in parallel using the set X_i as the splitting set of x -coordinates. We let $H_{i,0}, H_{i,1}, \dots, H_{i,k_i}$ denote the resulting HQ-trees, where the vertices in $H_{i,j}$ all have x -coordinates in the interval $[x_j, x_{j+1}]$, where $x_0 = -\infty$ and $x_{k_i+1} = +\infty$. Note that if $x_j = x_{j+1}$ then the HQ-tree $H_{i,j}$ contains a single vertex (the vertex v in C_i with $x(v) = x_j$). (See Figure 11.)

Analysis of Step 3. We can determine the elements of the list X in $O(1)$ time using $O(\sqrt{n/d})$ processors, and then we can sort those elements lexicographically to construct X in $O(\log n)$ time

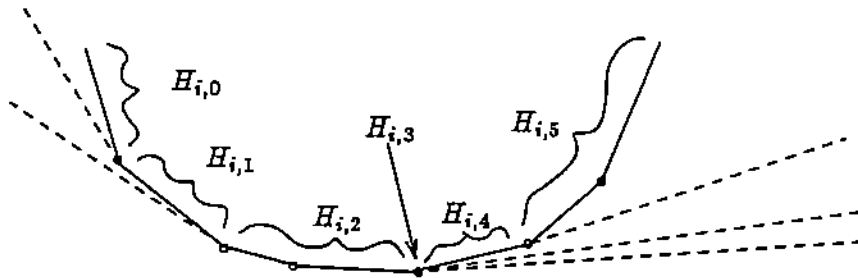


Figure 11: The splitting step. Each tangent t_w in T is denoted by a dashed line.

using $O(\sqrt{n/d})$ processors [11]. (We don't actually need as powerful a sorting routine as Cole's [11] in this case, since we can sort in $O(\log n)$ time using $O(n/d)$ processors by applying a simple "brute-force" sorting scheme.) Once we have constructed X , constructing each X_i list can be done in $O(\log n)$ time using $O(\sqrt{n/d})$ processors by a simple parallel prefix computation, since there are a total of $2\sqrt{n/d}$ elements in X . By Lemma 2.2 we can perform the k_i -way split in $O(h(\sqrt{nd}))$ time using $O(k_i)$ processors. Since there are $O(\sqrt{n/d})$ tangents in T , and two split operations performed for each one, there are a total of $O(\sqrt{n/d})$ split operations. Thus, we can perform all the splits of Step 3 in parallel using only $O(\sqrt{n/d})$ total processors, hence, the entire step can be performed in $O(\log n + h(\sqrt{nd}))$ time using $O(n/d)$ processors.

Details of Step 4. In Step 4 we construct an HQ-tree representation of each funnel polygon P_w . For each HQ-tree $H_{i,j}$ in parallel we search the tree B in a leaf-to-root fashion starting with the leaf corresponding to C_i . We perform this search to find the first internal node w on this path such that the tangent t_w completely spans the vertices in $H_{i,j}$. This is clearly the tangent which determines the funnel polygon containing the vertices in $H_{i,j}$. If there is no such tangent, then the vertices in $H_{i,j}$ must belong to $LH(C)$. Let A_w be the set of all HQ-trees $H_{i,j}$ such that t_w is the tangent determining the funnel polygon containing $H_{i,j}$. We can construct each A_w so that the member HQ-trees are listed by increasing x -coordinates, using a method similar to that used to construct the X_i lists in Step 3. Divide the HQ-trees in A_w into two lists: $A_{w,1}$, the ones with vertices to the left of L_w , and $A_{w,2}$, the ones with vertices to the right of L_w . We know that the concatenation of the vertices in HQ-trees in $A_{w,1}$ (resp., $A_{w,2}$) forms a convex chain, from Lemma 4.1. We complete the decomposition, then, by concatenating the HQ-trees in $A_{w,1}$ together, likewise with the HQ-trees in $A_{w,2}$, to form a representation of P_w . (See Figure 12.) Using a similar method we can collect all the $H_{i,j}$'s not spanned by any tangent line together and concatenate them to form an HQ-tree $H(LH(C))$ representing the lower hull $LH(C)$ of C .

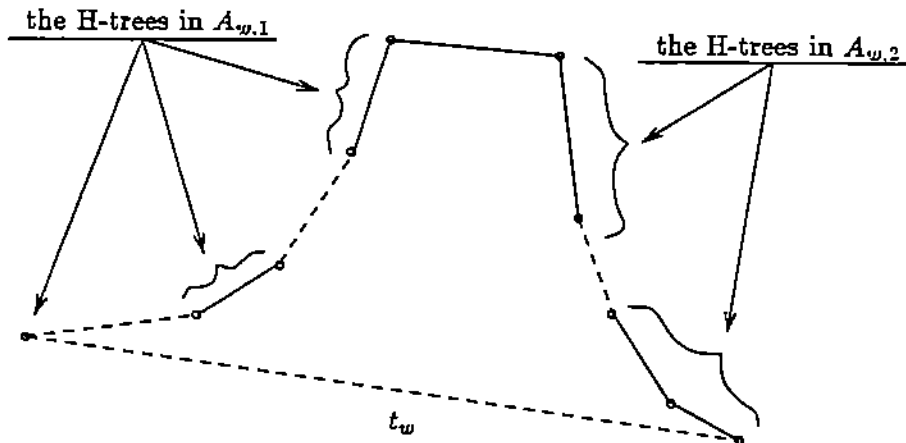


Figure 12: The concatenation step. Edges of P_w which are tangents in T are shown as dashed lines.

Analysis of Step 4. It should be clear that we can construct each of the sets $A_{w,1}$ and $A_{w,2}$ for each $w \in B$ in parallel in $O(\log n)$ time using a total of $O(\sqrt{n/d})$ processors [11], since there are $O(\sqrt{n/d})$ $H_{i,j}$'s (again, we could also use the simple "brute-force" sorting method). We can then concatenate each of the HQ-trees in the $A_{w,1}$'s and $A_{w,2}$'s in parallel in $O(\log n)$ time using $O(\sqrt{n/d})$ processors, by Lemma 2.3. Thus, the entire step can be performed in $O(\log n)$ time using $O(n/d)$ processors. This completes the detailed description of the PHASE-TWO algorithm.

We summarize the above discussion in the following lemma.

Lemma 4.2: *Given a polygonal chain $C = (v_1, v_2, \dots, v_n)$ which is monotone with respect to the x -axis and an integer $d > 0$, we can construct an HQ-tree $H(LH(C))$, representing the lower hull of C , and a decomposition of the underside of C into funnel polygons (each one represented by two HQ-trees) in $O(\log n + d + \log d \log \log n)$ time using $O(n/d)$ processors.*

Proof: The correctness of the PHASE-TWO method follows from the discussion made above. Let $h(n)$ denote the maximum height of any HQ-tree returned by the algorithm PHASE-TWO when passed an n -vertex polygonal chain. Also let $T(n)$ and $P(n)$ denote, respectively, the time and processor bounds of the procedure PHASE-TWO. We can bound the values of these three functions by the following recurrence relations:

$$\begin{aligned}
 h(n) &= \begin{cases} \lceil \log n \rceil & \text{if } n \leq 4d \\ h(\sqrt{nd}) + \lceil \log \sqrt{n/d} \rceil & \text{otherwise} \end{cases} \\
 T(n) &= \begin{cases} b_1 d & \text{if } n \leq 4d \\ T(\sqrt{nd}) + b_2(\log n + h(\sqrt{nd})) & \text{otherwise} \end{cases}
 \end{aligned}$$

$$P(n) = \begin{cases} 1 & \text{if } n \leq 4d \\ \max\{\lceil n/d \rceil, \sqrt{n/d}P(\sqrt{nd})\} & \text{otherwise} \end{cases}$$

where b_1 and b_2 are constants. This implies that $h(n) \leq 2 \log n$, that $T(n)$ is $O(d + \log n + \log d \log \log n)$, and that $P(n)$ is $O(\lceil n/d \rceil)$ [19]. This completes the proof. ■

Thus, by assigning $d = \lceil \log N \rceil$ we have that we can perform the PHASE-TWO procedure in $O(\log N)$ time using $O(n/\log N)$ processors in the CREW PRAM model. We complete the construction for phase two by constructing an array representation of each funnel polygon P_w from its HQ-tree representation in $O(\log N)$ additional time using $O(n/\log N)$ processors, using the method of the following lemma:

Lemma 4.3: *We can convert any HQ-tree H containing m vertices and having height $O(\log n)$ into a sorted array containing m vertices in $O(\log n)$ time using $O(m/\log n)$ processors, where $m \leq n$.*

Proof: The method is the following. For each processor $i \in \{0, 1, \dots, \lceil m/\log n \rceil\}$ we locate the leaf of H which has rank $i \lceil \log n \rceil$, using the d label stored at each node in the tree to direct the search. This takes $O(\log n)$ time. We can now for each processor i follow *succ* pointers from this point to find the next $\lceil \log n \rceil$ entries in the hemispherical chain (in parallel for each processor i). Thus, we can compute for each leaf of H how many vertices precede it. Thus we can convert the HQ-tree representation to an array representation by writing each vertex to its position in the array. This all can clearly be done in $O(\log n)$ time using $O(m/\log n)$ processors. ■

Since we perform this computation for each one-sided monotone polygon in parallel, we can perform this entire phase in $O(\log N)$ time using $O(N/\log N)$ processors. Now that we are done with the description of the second phase, let us go back to our convention of letting n denote the number of vertices in the original polygon. We summarize this section in the following theorem.

Theorem 4.4: *Given a collection of one-sided monotone polygons P_s , with a total of n vertices, we can decompose each P_s into a collection of monotone funnel polygons P_w in $O(\log n)$ time using $O(n/\log n)$ processors in the CREW PRAM model, where each monotone funnel polygon P_w is represented by two arrays, each listing the vertices of the convex chains defining P_w . ■*

The final phase of our algorithm is to triangulate each of the funnel polygons P_w . We present our method for performing the final phase of our algorithm in the following section.

5 Decomposition into Triangles

The final phase of our triangulation algorithm is to decompose all the monotone funnel polygons P_w into triangles in parallel. Since we only have $O(n/\log n)$ processors at our disposal, we must first perform an application of the sequential subsets technique. The details of this sequential subsets method are essentially the same as those of the method performed in the previous section. It allows us to triangulate all the polygons P_w with less than $\log n$ vertices in $O(\log n)$ time using $O(n/\log n)$ processors. So for the remainder of this section we assume that each P_w has more than $\log n$ vertices.

Let P_w be the monotone funnel polygon which we wish to triangulate, where t_w is the distinguished edge of P_w . Without loss of generality, let us assume that P is monotone in the x -direction and the vertices not on t_w are all above t_w . We will show how to decompose P_w into monotone funnel polygons in $O(\log n)$ time using $O(n_w/\log n)$ processors, where $n_w = |P_w|$.

Let A be an array listing the vertices in the left convex chain and let B be an array listing the vertices in the right convex chain. Merge the lists A and B using the method of Shiloach and Vishkin [27], basing comparisons on the distance of the points to the segment s . This can be done in $O(\log n)$ time using $O(n_w/\log n)$ processors. Augment P_w by adding an edge from each vertex in A (resp., B) to its predecessor in B (resp., A). This also can be done in $O(\log n)$ time using $O(n_w/\log n)$ processors. We show in the following lemma that this in fact triangulates P_w . (See Figure 13.)

Lemma 5.1: *Suppose we are given a funnel polygon P_w with base t_w , left chain A , and right chain B . If we add an edge from each vertex v in A (resp., B) to its predecessor in B (resp., A), where comparisons are based on the distance of vertices to the segment s , then we form a triangulation of P_w .*

Proof: Consider any edge e in P_w , other than s , with endpoints v and q , i.e. $e = vq$. Without loss of generality, v and q are in A . Let u (z) be the predecessor of v (q) in B . It is enough to show that the slice of P_w between vu and wz is triangulated correctly. Without loss of generality, s is parallel to the x -axis, $y(v) < y(w)$, and the edge e has positive slope. If $u = z$, then in adding the edges vu and wz we construct the triangle vuz (See Figure 14.a). On the other hand, if $u \neq z$, then there must be a chain of vertices ($u = v_1, v_2, \dots, v_m = z$) in P_w such that $y(v_1) < y(v) < y(v_2) < y(v_3) < \dots < y(v_m) < y(q)$ (see Figure 14.b). Thus, in Step 5.2 we will add an edge from each vertex v_2, \dots, v_m to v . Therefore, the portion of P_w between vu and qz consists of the triangle

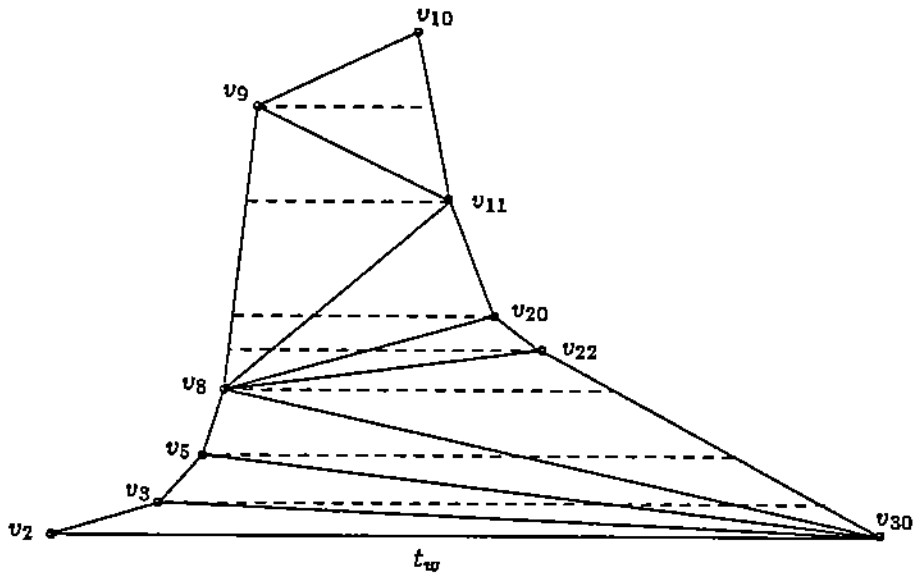


Figure 13: Triangulating a monotone funnel polygon.

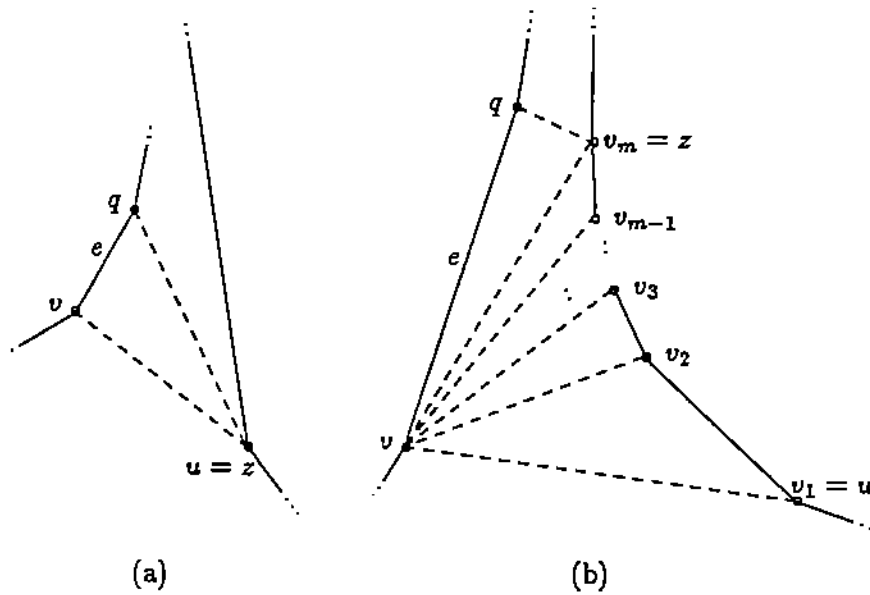


Figure 14: The triangulated portion defined by the edge vq . Figures (a) and (b) illustrate the two cases for proving that the portion of P_w between vu and qz is triangulated.

vqz and a series of triangles $vv_{i+1}v_i$, for $i \in \{1, \dots, m-1\}$ (See Figure 14.b). This completes the proof. ■

This completes the final phase of our triangulation algorithm. We summarize the previous three sections in the following theorem:

Theorem 5.2: *We can triangulate an n -vertex simple polygon P in $O(\log n)$ time using $O(n/\log n)$ processors in the CREW PRAM model, if we are given the trapezoidal decomposition of P , and this is optimal. If we are not given the trapezoidal decomposition of P we can triangulate P in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM model, and this is optimal if the polygon is allowed to contain holes.*

Proof: We have already established the correctness and complexity bounds of our triangulation procedure in the previous three sections. The first optimality claim follows immediately from the fact that the algorithm has a linear TP product. The second optimality claim follows from the proof by Asano, Asano, and Pinter [3] that the problem of triangulating a simple polygon which may contain holes has a lower bound of $\Omega(n \log n)$ (the TP product of our algorithm in this case), by a linear-time reduction from sorting. ■

Our algorithm also implies that a monotone polygon can be triangulated in $O(\log n)$ time using $O(n/\log n)$ processors, which is optimal. This is because one can form a trapezoidal decomposition of a monotone polygon P (monotone, say, with respect to the x -axis) by merging the vertices of the upper chain of P with the vertices of the lower chain of P , basing comparisons on x -coordinates. This can be done in $O(\log n)$ time using $O(n/\log n)$ processors using the algorithm by Shiloach and Vishkin [27].

6 Conclusion

We have given an efficient parallel algorithm which triangulates a simple polygon. This algorithm consists of three phases. In the first phase we decompose a simple polygon into a collection of one-sided polygons monotone with respect to the x -axis in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM model. If we given a trapezoidal decomposition of the polygon, then this phase runs in $O(\log n)$ time using only $O(n/\log n)$ processors. In the second phase we decompose each one-sided monotone polygon into a collection of funnel polygons in $O(\log n)$ time using $O(n)$ processors. Finally, in the third phase we triangulate each funnel polygon in $O(\log n)$ time using $O(n)$ processors. Thus, we have shown how to triangulate a simple polygon in $O(\log n + n \log n/p)$ time using p processors in the CREW PRAM model, which is optimal for $p \leq n$ if we allow for the

polygon to contain holes, since polygon triangulation has a sequential $\Omega(n \log n)$ lower bound [3]. If we are given a uniform trapezoidal decomposition of the polygon as input then we can triangulate the polygon in $O(\log n + n/p)$ time using p processors, which is optimal for $p \leq n/\log n$. Our algorithm also implies that a monotone polygon can be triangulated in $O(\log n + n/p)$ time using p processors, which is optimal for $p \leq n/\log n$.

Acknowledgement

We would like to thank Greg Shannon for a stimulating discussion involving topics related to this paper.

References

- [1] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap, "Parallel Computational Geometry," *Proc. 25th IEEE Symp. on Foundations of Computer Science*, 1985, pp. 468–477.
- [2] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap, "Parallel Computational Geometry," 1987, manuscript.
- [3] T. Asano, T. Asano, R. Pinter, "Polygon Triangulation: Efficiency and Minimality," *Journal of Algorithms*, Vol. 7, 1986, pp. 221–231.
- [4] M.J. Atallah, R. Cole, and M.T. Goodrich, "Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms," Tech. Report TR-665, Dept. of Computer Science, Purdue University, 1987.
- [5] M.J. Atallah and M.T. Goodrich, "Efficient Plane Sweeping in Parallel," *Proc. 2nd ACM Symp. on Computational Geometry*, 1986, pp. 216–225.
- [6] M.J. Atallah and M.T. Goodrich, "Efficient Parallel Solutions to Some Geometric Problems," *Journal of Parallel and Distributed Computing*, Vol. 3, 1986, pp. 492–507.
- [7] M.J. Atallah and M.T. Goodrich, "Parallel Algorithms for Some Functions of Two Convex Polygons," *Proc. 24th Allerton Conference on Communication, Control and Computing*, pp. 758–767, 1986.
- [8] R.P. Brent, "The Parallel Evaluation of General Arithmetic Expressions," *J. ACM*, Vol. 21, No. 2, 1974, pp. 201–206.

- [9] B. Chazelle and J. Incerpi, "Triangulating a Polygon by Divide-and-Conquer," *Proc. 21st Allerton Conference on Communication, Control, and Computing*, 1983.
- [10] A. Chow, "Parallel Algorithms for Geometric Problems," Ph.D. dissertation, Comp. Sci. Dept., Univ. of Illinois at Urbana-Champaign, 1980.
- [11] R. Cole, "Parallel Merge Sort," *Proc. 27th IEEE Symp. on Foundations of Computer Science*, 1986, pp. 511–516.
- [12] R. Cole and U. Vishkin, "Deterministic Coin Tossing and Accelerating Cascades: Micro and Macro Techniques for Designing Parallel Algorithms," *Proc. 18th ACM Symp. on Theory of Computation*, 1986, pp. 206–219.
- [13] R. Cole and U. Vishkin, "Approximate and Exact Parallel Scheduling with Applications to List, Tree, and Graph Problems," *Proc. 27th IEEE Symp. on Foundations of Computer Science*, 1986, pp. 478–491.
- [14] H. ElGindy, "A Parallel Algorithm for Triangulating Simplicial Point Sets in Space with Optimal Speed-up," *Proc. 24th Allerton Conference on Communication, Control, and Computing*, 1986.
- [15] H. ElGindy and M.T. Goodrich, "Parallel Algorithms for Shortest Path Problems in Polygons," Technical Report MS-CIS-87-20, Department of Computer and Information Science, University of Pennsylvania, 1987.
- [16] A. Fournier and D.Y. Montuno, "Triangulating Simple Polygons and Equivalent Problems," *ACM Trans. on Graphics*, Vol. 3, No. 2, April 1984, pp. 153–174.
- [17] M.R. Garey, D.S. Johnson, F.P. Preparata, and R.E. Tarjan, "Triangulating a Simple Polygon," *Information Processing Letters*, Vol. 7, No. 4, pp. 175–179.
- [18] M.T. Goodrich, "Finding the Convex Hull of a Sorted Point Set," to appear in *Information Processing Letters*; also available as Tech. Report TR-655, Computer Science Department, Purdue University, 1986.
- [19] M.T. Goodrich, "Efficient Parallel Techniques for Computational Geometry," Ph.D. thesis, Dept. of Computer Science, Purdue University, 1987.

- [20] L. Guibas, J. Hershberger, D. Leven, M. Sharir and R. Tarjan, "Linear Time Algorithms for Visibility and Shortest Path Problems Inside Simple Polygons," *Proceedings of the Second Symposium on Computational Geometry*, 1986, pp. 1-13.
- [21] C. Kruskal, L. Rudolph, and M. Snir, "The Power of Parallel Prefix," *Proc. 1985 IEEE Int. Conf. on Parallel Proc.*, pp. 180-185.
- [22] D.T. Lee and F.P. Preparata, "Computational Geometry—A Survey," *IEEE Trans. on Computers*, Vol. C-33, No. 12, December 1984, pp. 872-1101.
- [23] E. Merks, "An Optimal Parallel Algorithm for Triangulating a Set of Points in the Plane," Tech. Report TR 86-9, Simon Fraser University, Burnaby, British Columbia, Canada, 1986.
- [24] M.H. Overmars, and J. Van Leeuwen, "Maintenance of Configurations in the Plane," *Jour. of Comp. and Sys. Sci.*, **23**, 1981, pp. 166-204.
- [25] F.P. Preparata and M.I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, 1985.
- [26] J.H. Reif, "An Optimal Parallel Algorithm for Integer Sorting," *Proc. 26th IEEE Symp. on Foundations of Computer Science*, 1985, pp. 496-504.
- [27] Y. Shiloach and U. Vishkin, "Finding the Maximum, Merging, and Sorting in a Parallel Computation Model," *Journal of Algorithms*, Vol. 2, 1981, pp. 88-102.
- [28] R.E. Tarjan and C.J. Van Wyk, "An $O(n \log \log n)$ -Time Algorithm for Triangulating Simple Polygons," Technical Report CS-TR-052-86, Dept. of Computer Science, Princeton University.
- [29] J.C. Wyllie, "The Complexity of Parallel Computation," Ph.D. thesis, Tech. Report TR 79-387, Dept. of Computer Science, Cornell University, 1979.
- [30] C.K. Yap, "Parallel Triangulation of a Polygon in Two Calls to the Trapezoidal Map," manuscript, 1987.