Intersecting Line Segments in Parallel with an Output-Sensitive Number of Processors

Michael T. Goodrich*

Department of Computer Science
The Johns Hopkins University
Baltimore, MD 21218

Summary

We give an efficient parallel algorithm for constructing the arrangement of n line segments in the plane, i.e., the planar graph determined by the segment endpoints and intersections. Our algorithm is efficient relative to three efficiency measures—it is an NC algorithm, it has a small time-processor product, and it is output-size sensitive. In particular, it runs in $O(\log^2 n)$ time using $O(n + k/\log n)$ processors in the CREW PRAM model, where k is the size of the output (which is $\Omega(n^2)$ in the worst case). The algorithm does not receive the value of k as input, it determines it on-line.

1 Introduction

One of the major thrusts of computational geometry research has been to show that one can solve many geometric problems with a running time that is proportional to the input size plus the output size (times logarithmic factors in some cases); see, for example [4,8,9,17,19,20,23,29]. This is significant, because most of these problems have trivial $\Omega(n^2)$ lower bounds, which are based on constructing examples that have a large output size. Thus, these output-sensitive algorithms usually perform much better than the worst-case time on most inputs.

*Research supported by the National Science Foundation under Grant CCR-8810568.

One of the most studied of these problems is the problem of constructing the planar graph determined by the pair-wise intersections of a set of line segments in the plane, i.e., the *segment arrangement* problem [4,8,9]. This problem has several applications in computer graphics (e.g., [18,25,28]). One of the oldest algorithms solving this problem is an elegant method by Bentley and Ottmann [4] published in 1979 that uses the now-famous "plane-sweeping" paradigm [13,26]. The running time of their algorithm is sensitive to the size of the output, as it runs in $O((n + k) \log n)$ time for the general case, and in $\Theta(n \log n+k)$ time if the input segments are iso-oriented (i.e., parallel to the coordinate axes), where k is the size of the output. Since k is $\Omega(n^2)$ in the worst-case, it was not clear whether or not their general-case algorithm was optimal, however. Since then, there has been a considerable amount of research done to resolve this question (e.g., [9,10,16]). In fact, it wasn't until very recently that it was shown, by Chazelle and Edelsbrunner, that one can in fact solve this problem in $\Theta(n \log n + k)$ time [9].

In this paper we investigate how efficiently one can solve this problem in parallel. Our general goal is to design a parallel algorithm that simultaneously runs as fast as possible and has a time-processor product that is as small as possible. Thus, for the segment arrangement problem, we desire an algorithm that is output-sensitive.

There is no previous parallel algorithm for this problem other than the trivial brute-force method that is based on sorting [12] and runs in $O(\log n)$ time using $O(n^2)$ processors. There has been some related work done, however. In [2] Atallah, Cole, and Goodrich show to solve the decision problem, i.e., determining if *any* two segments intersect, in $O(\log n)$ time using $O(n)$ processors. It doesn't seem possible to extend their algorithm to the construction problem, however. In [11] Chow studies a restricted version of the problem, namely, she shows how to determine all the pair-wise intersections of n iso-oriented segments. Her algorithm runs in $O((1/\epsilon)\log n + k_{\max})$ time using $O(n^{1+\epsilon})$ processors [11], where $\epsilon > 0$ is a small constant and $k_{\max}$ is the maximum, taken over all input segments s, of the number of intersections on s. Note that this does not give an NC algorithm, since $k_{\max}$ is $\Omega(n)$ in the worst case, nor does is balance the computational burden for the case when only a few segments cause the majority of intersections.

The main result of this paper is the first output-sensitive parallel algorithm for solving the general segment arrangement problem. Our algorithm runs in $O(\log^2 n)$ time using $O(n+k/\log n)$ processors in the CREW PRAM model, where k is the size of the output. Note that our algorithm matches the time-processor product of the brute-force approach when the output size is large, i.e., $\Omega(n^2)$. We also give an algorithm for the case when the segments are iso-oriented that runs in $O(\log n)$ time using an optimal $O(n + k/\log n)$ number of processors in the CREW PRAM model.

The main obstacle to designing an output-sensitive parallel algorithm for this problem is that paradigms that led to the efficient sequential algorithms—such as plane-sweeping [13,26], topological sweeping [9,14], and incremental construction [13,26]—seem inherently sequential. Moreover, parallel techniques—such as the plane-sweep tree [1,2], cascading divide-and-conquer [2], and parallel sequence-evaluation [3]—that worked well for parallelizing fast sequential algorithms that use plane sweeping cannot be directly applied here, because it seems impossible to compute a priori all the places where a sweeping line would need to stop in a sequential algorithm. Our algorithm avoids the plane-sweeping approach all together. Instead, it is based on a number of new parallel techniques and a hierarchical geometric characterization of the types of intersections that can occur. The new parallel techniques include a "truncated" version of the zone lemma of [8,9,10,15,16] and a method for re-using processors created for enumerating intersections of one type to then discover intersections of another type. Our algorithm achieves its output-sensitivity by computing the size of the output while it is computing the answer, and dynamically allocates new processors accordingly.

In the next section we discuss some preliminaries, including a discussion of our computational model. In Section 3 we give an overview of our algorithm, and we give the details of our method in Sections 4 and 5. In Section 6 we outline our algorithm for the iso-oriented case. We conclude with Section 7.

2 Preliminaries

2.1 The Computational Model

The computational model we use in this paper is the CREW PRAM model. Recall that processors in this model act in a synchronous fashion and use a shared memory space where many processors may simultaneously read from the same location but no two processors may simultaneously write to the same location. Given an input of size n, the traditional way of utilizing this model is that one simply allocates, once and for all, a number of processors that depends on n (e.g., n^2, $n \log n$, etc.). Of course, a real parallel machine has a constant number, p, of processors, not a number that is a function of n. Thus, the p real processors must always simulate the "virtual" processors in the algorithm in order to implement it. Since we wish to solve a problem in an output-sensitive manner, in order to achieve the maximum speed-up possible we allow the set of virtual processors to grow dynamically. More specifically, we consider a version of the PRAM, as outlined by Reif and Sen in [27], where a new processor can be created by having some existing processor execute a *spawning* operation. Such an operation is issued by an existing processor specifying the task that a new processor is to perform, and in the next time step a new processor is created and begins executing that task. This is also similar to a model used by Bhatt and Cai [6], for example. We refer to this model as the *L-PRAM model*, since processors are created *locally*, instead of globally, as in the traditional PRAM model. It is beyond the scope of this paper to study general model-comparison results, but we do give the following lemma:

Lemma 2.1: *If an algorithm can be implemented in the L-PRAM model in t steps using p processors, then it can be implemented in $O(t \log p)$ steps using $O(p/\log p)$ processors in the analogous PRAM model.*

Proof sketch: Let p_i denote the number of processors used in the L-PRAM model in step i. The main idea of the proof is to simulate step i of the L-PRAM algorithm in $O(\log p_i)$ time using $\lceil p_i/\log p_i \rceil$ processors in the PRAM model. With each step of the L-PRAM algorithm, one first performs all the non-spawning computations, and then performs a parallel prefix [21,22] to determine the number, $p_{i+1} - p_i$, of new processors that are to be spawned in step i and to which tasks they are to be assigned. This takes at most $O(\log p_i)$ time using $\lceil p_i/\log p_i \rceil$ processors. One then requests $\lceil p_{i+1}/\log p_{i+1} \rceil - \lceil p_i/\log p_i \rceil$ new processors on the PRAM machine to help simulate the next step of the L-PRAM algorithm. Thus, the entire algorithm can be implemented in $O(t \log p_t)$ time using $O(p_t/\log p_t)$ processors in the PRAM model. ∎

Having discussed our computational model, let us now discuss some preliminaries for the segment arrangement problem.

2.2 Characterizing Intersections

In this section we review an observation by Chazelle [8] for characterizing intersections in terms of a segment tree data structure [5]. Let S be a set of n line segments in the plane, and let T be the complete binary tree whose at most $2n+1$ leaves, in left to right order, correspond to the vertical slabs determined by the endpoints of the segments in S. For each v in T we use Π_v to denote the union of all the slabs associated with the descendents of v (including v itself, if v is a leaf). A segment s_i *covers* a node $v \in T$ if it spans Π_v but not $\Pi_{parent(v)}$. Clearly, no segment covers more than 2 nodes on any level of T; hence, each segment covers at most $O(\log n)$ nodes of T. For each node $v \in T$ we define the following sets (see Figure 1):

$$Cover(v) = \{s \in S \mid s \text{ covers } v\},$$

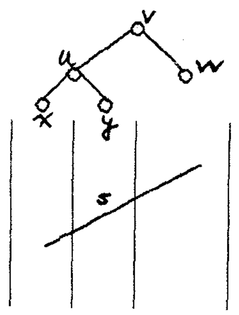

Figure 1: The segment s is in $Cover(y)$, $End(x)$, $End(u)$, $End(w)$, and $End(v)$.

$$End(v) = \{s \in S \mid s \text{ does not span } \Pi_v, \text{ but has an endpoint in } \Pi_v\}.$$

Observation 2.2 [8]: *Let S be a set of line segments in the plane, and let s_1 and s_2 be two segments in S that intersect at a point p. In addition, let T be a segment tree for S. Then there is a (unique) node $v \in T$ such that $p \in \Pi_v$ and one of the following is true:*

1. *$s_1, s_2 \in Cover(v)$,*
2. *$s_1 \in End(v)$ and $s_2 \in Cover(v)$,*
3. *$s_2 \in End(v)$ and $s_1 \in Cover(v)$.*

We call intersections of type 1 *CC-intersections* and intersections of types 2 and 3 *EC-intersections*. We present an overview of our algorithm in the next section.

3 An Overview

Suppose we are given a set S of n line segments in the plane. We define the *upper* (resp., *lower*) *vertical shadow* in S of a point p as the point on a segment in S that is intersected by the maximal vertical ray emanating upward (resp., downward) from p that does not intersect any other segment in S, if such a point exists. The *segment arrangement* of S is defined to be the planar graph determined by the pair-wise intersections in S as well as all the vertical shadows of the endpoints of segments in S (see Figure 2). For simplicity, we

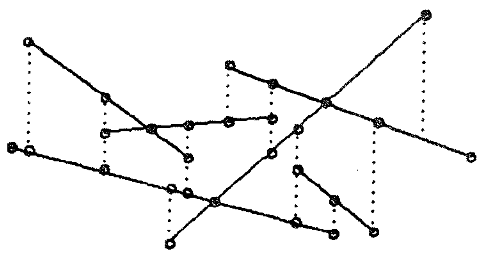

Figure 2: An example segment arrangement.

assume that at most two segments meet at any intersection point. One can easily modify our algorithm for the general case (using an appropriate definition of the "multiplicity" of an intersection point).

In what follows we describe the algorithm so as to run in $O(\log n)$ time using $O(n \log n + k)$ processors in the CREW L-PRAM model; we then use Lemma 2.1 to derive the bounds stated in the introduction.

Step 1. In this step we construct a segment tree T for the segments in S, including the lists $End(v)$ and $Cover(v)$ for each $v \in T$. This can be done in $O(\log n)$ time with $O(n \log n)$ processors using an algorithm by Aggarwal *et al.* [1]. Then, for each v in T in parallel, we sort the segments in $Cover(v)$, where comparisons are based on the y-coordinates of the intersections of the segments with the left boundary of Π_v. Since the total size of all the $Cover(v)$'s is $O(n \log n)$, this sorting step can also be performed in $O(\log n)$ time using $O(n \log n)$ processors [12].

Step 2. In this step we determine all the CC-intersections in S. Our method is based on the simple observation that if two segments in $Cover(v)$ intersect, then their relative order would be reversed if we were to base comparisons on the right boundary of Π_v (instead of the left boundary). We implement this step by constructing a data structure that can answer the related dominance query for a segment s in $O(\log n + \alpha_{s,v})$ time, where $\alpha_{s,v}$ is the number of answers for s in $Cover(v)$. We then use these lists to construct the arrangement of the segments in $Cover(v)$ (the so-called ***hammock*** [8,9]). This step requires $O(\log n)$ time using $O(n \log n + \alpha)$ processors, where α is the total number of CC-intersections in S.

Step 3. In this step we compute all the EC-intersections in S. This is the most involved step in our construction. The main idea is to construct two data structures from the hammock produced in Step 2, for each v in T in parallel. We use the first data structure to find all the EC-intersections with segments in $Cover(v)$ for each segment s in $End(v)$, so long as there are less than $c \log n$ of them (c is a constant parameter), or, alternatively, to determine if there are at least $c \log n$ such intersections. This requires $O(\log n)$ time and 1 processor per segment in $End(v)$, and is based on a "truncated" zone theorem. The second data structure allows us to find all the EC-intersections with segments in $Cover(v)$ for any segment s in $End(v)$ in $O(\log n)$ time with $O(\log n + \beta_{s,v})$ processors per s, where $\beta_{s,v}$ is the number of such intersections. But we only use this second data structure if the first one did not discover all the intersections for s; so $\beta_{s,v} > c \log n$ for all such segments. We conclude the construction by determining all the adjacencies between the intersection points and endpoints in the segment arrangement. This entire step requires $O(\log n)$ time using $O(n \log n + \alpha + \beta)$ processors, where β is the number of EC-intersections in S.

End of Outline.

So, assuming we can implement each of the above steps in the stated bounds, then we can enumerate all the pair-wise intersections in S in $O(\log n)$ time using $O(n \log n + k)$ processors, where $k = \alpha + \beta$ is the size of the output. Let us now give the details for performing each of the above steps. The details for Step 1 should already be apparent, so we begin our detailed description with Step 2.

4 Computing CC-Intersections

Let us concentrate on finding all the CC-intersections for a specific node v in T. Recall that in Step 1 we constructed all the $Cover(v)$ lists in T. For each segment s in $Cover(v)$, let $y_1(s)$ (resp., $y_2(s)$) denote the y-coordinate of the intersection of s with the left (resp., right) boundary of Π_v. The following observation characterizes all CC-intersections in terms of these labels.

Observation 4.1: *Two segments r and s in $Cover(v)$ have a CC-intersection intersection in Π_v if and only if one of the following is true:*

1. $y_1(r) < y_1(s)$ *but* $y_2(r) > y_2(s)$,
2. $y_1(r) > y_1(s)$ *but* $y_2(r) < y_2(s)$.

For each segment s in $Cover(v)$, if we define a point $p_s = (y_1(s), y_2(s))$, then we can interpret Observation 4.1 in terms of dominance relationships. Namely, a segment s' has a CC-intersection with s if and only if $p_{s'}$ is (i) above and to the left of p_s, or (ii) below and to the right of p_s. Thus, let us digress a bit to discuss how one can efficiently solve the dominance reporting problem in parallel.

4.1 Dominance Reporting

The generic problem is the following: we are given a set R of n points in the plane sorted by x-coordinates, and wish to construct a data structure D that allows one to efficiently report all the points in a query range $[x, \infty) \times [y, \infty)$. There are a number of ways one can solve this problem; we include one such method here to illustrate how one can use the spawning operation.

The data structure D is simply a binary tree that stores the points of R in its leaves (listed from left to right). With each internal node in D we associate two labels, $xmax(v)$ and $ymax(v)$, where $xmax(v)$ (resp., $ymax(v)$) is the descendent of v that has maximum x-coordinate (resp., y-coordinate). This construction can easily be done in $O(\log n)$ time with $O(n/\log n)$ processors [12,30], by a simple application of Brent's theorem [7].

One uses D to answer a dominance query $Q = [x, \infty) \times [y, \infty)$ as follows. We begin by assigning a processor p to search D, starting at the root, to locate the position of x among the leaves of D. Each time p visits a left child z, p tests if the subtree D_v rooted at z's sibling v contains any answers or not (by testing if $ymax(v) \geq y$). If D_v contains some answers, then p spawns a processor p' to enumerate all the answers in D_v. The processor p continues in this fashion until it reaches the leaf-level of D. The algorithm for p' is as follows. Let v be the node p' is currently at in the search, and let u and w be v's left and right child, respectively. If there are answers in both D_u and D_w, then p' spawns a new processor p'' to search D_u (using the same method) and continues its search in D_w. Otherwise, p' simply continues it search in the subtree that contains an answer. The spawned processors continue in this way until they reach the leaf-level in D. They complete the computation for Q by collecting all their answers into a single array. If the spawned processors have maintained themselves in a doubly-linked list (which is easy to do), then this can be done by a simple list-ranking computation [30]. Thus, we have the following lemma.

Lemma 4.2: *Given a set R of n points in the plane sorted by x-coordinates, one can construct a dominance-reporting data structure that can be used to answer dominance queries in $O(\log n)$ time using $O(1 + l)$ processors in the CREW L-PRAM model, where l is the number of answers. This construction requires $O(\log n)$ time using $O(n/\log n)$ processors.* ∎

We note in passing that we could have saved a $\log n$ factor in the number of spawned processors had we used a more powerful data structure (e.g., a priority search tree [24]), but the method outlined above will be sufficient for our purposes. Let us return, then, to the problem at hand.

4.2 Constructing the Hammock

From the $Cover(v)$ list we have the p_s's listed in sorted order by their first coordinates. Thus, we can construct the dominance query data structure in $O(\log n)$ time using $O(n_v/\log n)$ processors, by Lemma 4.2, where $n_v = |Cover(v)|$. The lemma also implies that the queries for a specific p_s can be answered in $O(\log n)$ time using $O(1 + \alpha_{s,v})$ processors, where $\alpha_{s,v}$ is the number of CC-intersections s has with other segments in $Cover(v)$. This of course implies that, given the segment tree constructed in Step 1, we can find

all the CC-intersections in S (for all v in T in parallel) in $O(\log n)$ time using $O(n+\alpha)$ processors, where α is the sum of all the $\alpha_{s,v}$'s (also recall that $\sum_{v\in T} n_v$ is $O(n\log n)$).

To complete Step 2 we have only to construct the adjacency information for the hammock. That is, for each intersection point p of segments r and s we must determine the other intersection points on r and s to which p is adjacent. We do this by sorting, for each s in parallel, the intersections along s by x-coordinates. Then for each intersection point p of a segment s with a segment r we locate the position of p in the list for r by a binary search. From this we then construct a representation of the planar graph induced by the adjacencies of the CC-intersection for $Cover(v)$. We finish the construction by augmenting the graph, as Chazelle does [8], by adding pointers for each edge e that point to the leftmost and rightmost vertex, respectively, of each face in the hammock to which e belongs. Since the bottleneck in this computation is the sorting of $O(n\log n+\alpha)$ elements, it takes $O(\log n)$ time using $O(n\log n+\alpha)$ processors [12].

Thus, we have shown how to efficiently find all the CC-intersections in S. In the next address the problem of finding the EC-intersections in S.

5 Computing EC-Intersections

To complete the algorithm we must find all the EC-intersections for each v in T in parallel. As mentioned earlier, this is the most involved step in the construction. It consists of two phases: one that finds the intersections along segments that have few EC-intersections, and the other that finds the intersections along segments that have many EC-intersections.

5.1 Segments with few intersections

Let us concentrate on the computations for a particular v in T. We begin by constructing a planar point location data structure for the hammock for each v using the methods of Atallah, Cole, and Goodrich [2], which takes $O(\log n)$ time using $O(|Cover(v)|+\alpha_v)$ processors per v, where α_v is the number of CC-intersection determined by the

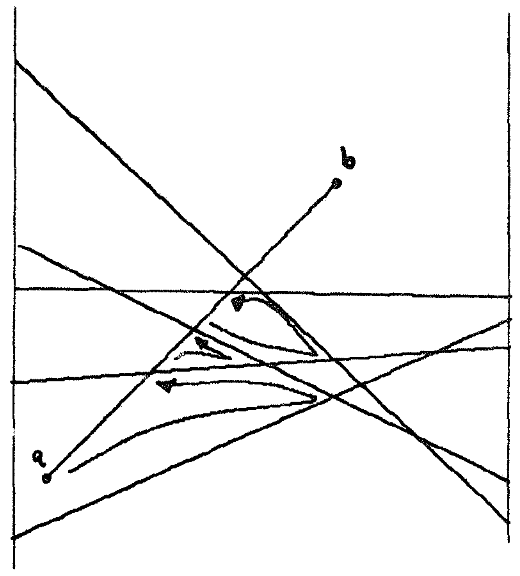

Figure 3: An example walk in the hammock.

segments in $Cover(v)$. This requires $O(n\log n+\alpha)$ processors total, and allows point locations to be performed in the hammock in $O(\log n)$ time using a single processor.

Suppose we are given a query segment s in $End(v)$. We begin by locating the two faces f_a and f_b that contain s's two endpoints a and b, respectively (with a being to the left of b). Our method is to then mimic the method of Chazelle [8] for walking through the hammock from f_a to f_b, except that we cut the walk short as soon as it traverses at least $4c\log n$ edges (where c is a constant parameter). We will show that if the walk is terminated early because of this restriction, then s must have at least $c\log n$ intersections with segments in $Cover(v)$.

So let us review the method of Chazelle [8]. If $f_a = f_b$, then we are done, so let us assume $f_a \neq f_b$. One begins by jumping to the rightmost vertex v_1 in $f_1 = f_a$ and then traversing the edges of f_1 until finding the edge e_1 of f_1 that intersects s. If v_1 is above the line supporting s, then this traversal is to be clockwise, and is to be counterclockwise, otherwise. Upon reaching e_1, one uses the adjacency information for e_1 to "hop" over e_1 into the next face, f_2, adjacent to s, and then jump to the rightmost vertex v_2 in f_2. (See Figure 3.) One continues in this way, going from face to face along s, provided that for each edge e traversed, the line supporting e intersects s. If one is about to traverse an edge whose supporting line does not intersect s, then one suspends the traversal from f_a at this point, and begins a symmetric traversal from f_b (using the rule that if v_i is above the line supporting s, then the traversal must be counter-clockwise, and must be clockwise, otherwise). One continues this traversal until all the intersections along s have been discovered or, as in our case, one traverses at least $4c \log n$ edges. Chazelle [8] shows if one uses this strategy, then one will eventually discover all the intersections along s and the total time spent will be proportional to the number of intersections. We need a stronger property than this, however:

Lemma 5.1: *Suppose one has traversed at least 4δ edges in performing the walk for a segment s. Then there are at least δ intersections along s in the hammock.*

Proof sketch: Since this is a slightly stronger version of a lemma proved by Chazelle [8], we use the proof technique of Chazelle, Guibas, and Lee [10] to prove it. Namely, we use an accounting scheme, where for each edge traversed, we charge one of the intersections along s for the cost of this traversal. Let f be a face traversed, and let s_i be the subsegment of s contained in f. The traversed edges of f can be divided into three groups: *left-hanging* edges, which intersect s left of s_i, *right-hanging* edges, which intersect s right of s_i, and *anchored* edges, which are adjacent to s_i. These groups suffice, because the line supporting each traversed edge intersects s. Note that for any face f all the edges we traverse in f will be either left-hanging or right-hanging, but not both. The accounting scheme is that each left-hanging edge e charges the intersection of s with the line supporting e's successor in a clockwise traversal around f, and each right-hanging edge e charges the intersection of s with the line supporting e's successor in a counter-clockwise traversal around f. Each anchored edge e simply charges its intersection with s. It is easy to see that each intersection point can be charged by at most one left-hanging edge, one right-hanging edge, and at most twice by its anchored edge. Thus, each intersection point can be charged at most 4 times. Therefore, if we have traversed at least 4δ edges, then we must have charged at least δ intersection points. ■

Thus, by this "truncated" zone lemma, if in traversing the hammock for a segment s we stopped by reaching the other endpoint of s, then we have discovered all the EC-intersections for s; and if we terminated the traversal early, then there are at least $c \log n$ intersections of s with segments in $Cover(v)$. Note, however, that these $c \log n$ intersection points need not be adjacent in the list of intersections along s.

Let E_v be the list of all segments in $End(v)$ that have at least $c \log n$ EC-intersections, and let S_v denote the set of segment "pieces" in the hammock for v, i.e., the segments resulting from cutting each s in $Cover(v)$ at its CC-intersections. We have yet to find all the EC-intersections for the segments in E_v.

5.2 Segments with many intersections

We begin by building a segment tree T_v for the segments in S_v. To avoid confusion, let us denote the sets and slabs for each node w in T_v using lower-case letters. Thus, for each w in T_v we define lists $cover(w)$ and $end(w)$ in terms of the slab π_w associated with w. (See Figure 4.) For each w in T_v we have $cover(w)$ stored in sorted order by the segment intersections with the left vertical boundary of π_w. Let us also define a list $left(w)$, which consists of all segments in $end(w)$ that intersection the left boundary of π_w, and let us also store the $left(w)$ lists sorted by the segment intersections with the left vertical boundary. Since the subsegments in S_v do not intersect, except at their endpoints, we can use the method of Atallah, Cole, and Goodrich [2] to build T_v. Note: the tree in the Atallah, Cole, Goodrich construction is built on every $\log n$-th x-coordinate; so that the $end(w)$ list stored in a leaf has $O(\log n)$ size rather than $O(1)$ size. This does not affect the running time of queries, however, as we will see later. Their method runs in $O(\log m)$ time using $O(m)$ processors, where m is the number of segments. In our case $m = |Cover(v)| + \alpha_v$. Thus, we use the processors created in Step 2 (to enumerate CC-intersections) to now allow us to construct T_v

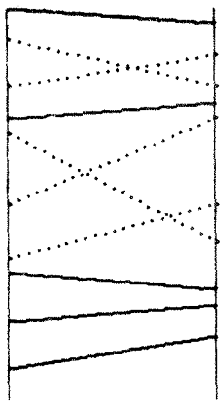

Figure 4: An example π_w. The segments in $end(w)$ are shown dotted and the segments in $cover(v)$ are shown solid.

for each v in T in parallel in $O(\log n)$ time. This requires a total of $O(n \log n + \alpha)$ processors.

For each w in T, we let $inter(w)$ denote the set of segments in $Cover(v)$ that have an intersection point in π_w. Recall that the subsegments in S_v are all pieces of segments in $Cover(v)$ that span Π_v. Thus, even though each subsegment in $left(w)$ does not span π_w, it is a piece of a segment that does span π_w. Thus, $left(w)$ contains a representative piece of each segment in $inter(w)$. With each segment s in $inter(w)$ we associate a pair $(y_1(s), y_2(s))$, where $y_1(s)$ (resp., $y_2(s)$) is the y-coordinate of the intersection of the left (resp., right) vertical boundary of π_w with s. Thus, by applying the techniques of Section 4.1, we can build a dominance reporting data structure $D(w)$ for the segments in $inter(w)$ in $O(\log n)$ time using $O(|left(w)|/\log n)$ processors (recall that $left(w)$ is given in sorted order). Since each CC-intersection can be contained in at most $O(\log n)$ different π_w's, the total size of all the $left(w)$'s in T_v is $O((|Cover(v)|+\alpha_v)\log n)$. Thus, the total number of processors needed to construct all the $D(w)$'s in T_v is $O(|Cover(v)| + \alpha_v)$. Therefore, the total number of processors needed to do this for all T_v's is $O(n \log n + \alpha)$.

We use the $cover(w)$'s and $D(w)$'s to allow us to find the EC-intersections for each segment s in E_v. In particular, we assign $O(\log n)$ processors to s to perform $O(\log n)$ queries, one for each node w in T_v such that s either covers w or has an endpoint in π_w. Note that all the segments in E_v have at least $c \log n$ EC-intersections with segments in $Cover(v)$. So the extra processors we are now allocating to s can be accounted for by "charging" the EC-intersections on s. There are three types of queries we perform:

Query 1. If s has an endpoint in π_w or s covers w, then we locate the two endpoints of the segment $s \cap \pi_w$ (i.e., s "clipped" to π_w) in $cover(w)$, by two binary searches. All the segments in $cover(w)$ between these two positions in the list must intersect s. The processor associated with this w then spawns enough other processors (in a doubling fashion) to enumerate these segments.

Query 2. If s covers w, and w is not a leaf, then we perform a dominance query for s using the data structure $D(w)$, since s necessarily spans the slab π_w. This query is exactly as that used in Section 4.

Query 3. If s covers w or s has an endpoint in π_w, and w is a leaf, then we simply search through all the segments s' with a piece $\hat{s}$ in $end(w)$ to see if any of the s' segments intersect s (there can be at most $O(\log n)$ such segments [2]).

The next lemma shows that when the processing of these types of queries is complete we have enumerated all the EC-intersections for s with segments in $Cover(v)$.

Lemma 5.2: *For any segment s in E_v all the EC-intersections of s with segments in $Cover(v)$ will be discovered by the above queries.*

Proof: Suppose p is an EC-intersection point of s and a segment s' in $Cover(v)$ that will be missed by the above queries. The set of nodes w such that p is contained in π_w forms a leaf-to-root path σ in T_v. Suppose there is a node u on σ that s covers. There are two cases.

Case 1: s' does not have a CC-intersection in π_u. In this case there must be a piece $\hat{s}$ of s' in S_v such that $\hat{s}$ spans π_u. This, of course, implies that $\hat{s}$ covers some node on the path from u to the root of T_v. But we will perform a type 1 query for s at each of these nodes. $(\rightarrow\leftarrow)$

Case 2: s' has a CC-intersection in π_u. In this case there must be a piece $\hat{s}$ of s' in S_v such that $\hat{s}$ is in $end(u)$. If u is not a leaf, then s' is in $inter(u)$; hence, s' will be included in $D(u)$. But

we will perform a type 2 query for s at such a u. So u must be a leaf. But we will perform a type 3 query for s at such a u. $(\rightarrow\leftarrow)$
Thus, there is no node on σ that s covers. That is, s does not span π_w for any node w on σ; hence, s has an endpoint in π_z, where z is the leaf such that $p \in \pi_z$. But we will perform a type 3 query at z. $(\rightarrow\leftarrow)$ Therefore, we will discover p. ∎

We complete our algorithm by constructing the segment arrangement, not counting the vertical shadows, from the intersection points and endpoints, using using essentially the same method we used to construct the hammocks. We then augment this structure with the vertical shadows by applying the trapezoidal decomposition of Atallah, Cole, and Goodrich [2] and the sorting algorithm of Cole [12]. This takes $O(\log n)$ time using $O(n+k)$ processors, where $k = \alpha + \beta$. We summarize:

Theorem 5.3: *Given a set S of n line segments in the plane, one can construct the segment arrangement for S in $O(\log n)$ time using $O(n \log n + k)$ processors in the CREW L-PRAM model, where k is the size of the output.* ∎

Corollary 5.4: *The segment arrangement for S can be constructed in $O(\log^2 n)$ time using $O(n + k/\log n)$ processors in the CREW PRAM model.*

6 Iso-Oriented Segments

In this section we outline how to construct the segment arrangement when all the segments are parallel to the x- and y-axes. We give the details in the full version of this paper. Our method runs in $O(\log n)$ time using $O(n + k/\log n)$ processors in the CREW PRAM model, which is optimal. Our method uses the ***array-of-trees*** parallel data structure of Atallah, Goodrich, and Kosaraju [3]. Given a sequence σ of **insert**(p) and **delete**(p) operations that operate on an initially empty set, the array-of-trees allows one to perform queries in the past (i.e., for some position i in σ) as if one had all the elements present in the set at "time" i stored in a complete binary tree, where each internal node stores $O(1)$ labels that are the values of associative operations applied to the descendents of v. This data structure can be constructed in $O(\log n)$ time using $O(n)$ processors [3]. We construct this data structure to represent a horizontal plane-sweep (e.g., [4]) and use it to perform a range query for every position i that corresponds to a vertical segment. This takes $O(\log n)$ time using a total of $O(n + k/\log n)$ processors in the CREW PRAM model. It also gives us all the vertical adjacencies in the segment arrangement. A similar method gives us the horizontal adjacencies.

Theorem 6.1: *Given a set S of n iso-oriented segments in the plane, one can construct the segment arrangement for S in $O(\log n)$ time using $O(n + k/\log n)$ processors in the CREW PRAM model, where k is the size of the output.* ∎

7 Open Problem

We have shown how construct the segment arrangement of a set of line segments in the plane in parallel so that total work performed is only a $\log n$ factor from the sequential lower bound (which is achievable [9]). If all the segments are extended to infinite lines, then our algorithm becomes equivalent to the brute-force method that runs in $O(\log n)$ time using $O(n^2)$ processors. Can the arrangement in this case be constructed in $O(\log n)$ time using only $O(n^2/\log n)$ processors (which would match the sequential running time for this problem [10,14,16])?

Acknowledgements

We thank Mikhail J. Atallah and S. Rao Kosaraju for helpful discussions, and Gregory Bachelis for pointing out an omission in an earlier version of the proof of Lemma 5.2.

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap, "Parallel Computational Geometry," *Algorithmica*, **3**(3), 1988, 293–328.

[2] M.J. Atallah, R. Cole, and M.T. Goodrich, "Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms," *28th FOCS*, 1987, 151–160.

[3] M.J. Atallah, M.T. Goodrich, and S.R. Kosaraju, "Parallel Algorithms for Evaluating Sequences of Set-Manipulation Operations," Aegean Workshop on Comp., 1988, 1–10.

[4] J.L. Bentley and T. Ottmann, "Algorithms for Reporting and Counting Geometric Intersections," *IEEE Trans. on Computers*, **C-28**, 1979, 643–647.

[5] J.L. Bentley and D. Wood, "An Optimal Worst Case Algorithm for Reporting Intersections of Rectangles," *IEEE Trans. on Computers*, **C-29**(7), 1980, 571–576.

[6] S. Bhatt and J.Y. Cai, "Take a Walk, Grow a Tree," *29th FOCS*, 1988, 469–478.

[7] R.P. Brent, "The Parallel Evalutation of General Arithmetic Expressions," *J. ACM*, Vol. 21, No. 2, 1974, pp. 201–206.

[8] B. Chazelle, "Intersecting is Easier Than Sorting," *16th ACM Symp. on Theory of Comp.* (STOC), 1984, pp. 125–134.

[9] B. Chazelle and H. Edelsbrunner, "An Optimal Algorithm for Intersecting Line Segments in the Plane," *29th FOCS*, 1988, 590–600.

[10] B. Chazelle, L.J. Guibas, and D.T. Lee, "The Power of Geometric Duality," *24th FOCS*, 1983, 217–225.

[11] A. Chow, "Parallel Algorithms for Geometric Problems," Ph.D. thesis, Comp. Sci. Dept., Univ. of Illinois, 1980.

[12] R. Cole, "Parallel Merge Sort," *SIAM J. Comput.*, **17**(4), 1988, 770–785.

[13] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, NY, 1987.

[14] H. Edesbrunner and L.J. Guibas, "Topologically Sweeping an Arrangement," *18th STOC*, 1986, 389–403.

[15] H. Edelsbrunner, L.J. Guibas, J. Pach, R. Pollack, R. Seidel, and M. Sharir, "Arrangements of Curves in the Plane – Topology, Combinatorics, and Algorithms," UIUCDCS-R-88-1477, Dept. of Comp. Sci., Univ. of Illinois, 1988.

[16] H. Edelsbrunner, J. O'Rourke, and R. Seidel, "Constructing Arrangements of Lines and Hyperplanes with Applications," *24th FOCS*, 1983, 83–91.

[17] S.K. Ghosh and D.M. Mount, "An Output Sensitive Algorithm for Computing Visibility Graphs," *28th FOCS*, 1987, 11–19.

[18] M.T. Goodrich, "A Polygonal Approach to Hidden-Line Elimination," *25th Allerton Conf.*, 1987, 849–858.

[19] R.H. Güting, "An Optimal Contour Algorithm for Iso-Oriented Rectangles," *J. Algorithms*, **5**, 1984, 303–326.

[20] J. Hershberger, "Finding the Visibility Graph of a Simple Polygon in Time Proportional to its Size," *3rd ACM Symp. on Comp. Geom.*, 1987, 11–20.

[21] Kruskal, C.P., Rudolph, L., and Snir, M., "The Power of Parallel Prefix," *1985 Int. Conf. on Parallel Processing*, 180–185.

[22] Ladner, R.E., and Fischer, M.J., "Parallel Prefix Computation," *J. ACM*, October 1980, 831–838.

[23] W. Lipski, Jr. and F.P. Preparata, "Finding the Contour of a Union of Iso-Oriented Rectangles," *J. Algorithms*, **1**, 1980, 235–246.

[24] E.M. McCreight, "Priority Search Trees," *SIAM J. on Comput.*, No. 14, 1985, 257–276.

[25] O. Nurmi, "A Fast Line-Sweep Algorithm For Hidden Line Elimination," *BIT*, Vol. 25, 1985, 466–472.

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

[27] J. Reif and S. Sen, "An Efficient Output-Sensitive Hidden-Surface Removal Algorithm and its Parallelization," *4th ACM Symp. on Comp. Geom.*, 1988, 193-200.

[28] A. Schmitt, "Time and Space Bounds for Hidden Line and Hidden Surface Algorithms," *EUROGRAPHICS '81*, 43–56.

[29] D. Wood, "The Contour Problem for Rectilinear Polygons," *Info. Proc. Let.*, Vol. 19, 1984, 229–236.

[30] J.C. Wyllie, "The Complexity of Parallel Computation," Ph.D. thesis, TR 79-387, Dept. of Comp. Sci., Cornell Univ., 1979.