SIAM J. COMPUT.
Vol. 20, No. 4 , pp. 737–755, August 1991

010

INTERSECTING LINE SEGMENTS IN PARALLEL WITH AN OUTPUT-SENSITIVE NUMBER OF PROCESSORS*

MICHAEL T. GOODRICH†

Abstract. An efficient parallel algorithm is given for constructing the arrangement of n line segments in the plane, i.e., the planar graph determined by the segment endpoints and intersections. This 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 $O(\log n)$ time using $O(n \log n + k)$ processors, 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. A method for solving an important special case of the segment arrangement problem is also shown, namely, when each input segment is parallel to one of the coordinate axes (i.e., iso-oriented). The algorithm for this problem runs in $O(\log n)$ time using an optimal $O(n + k/\log n)$ processors. The model of computation is the CREW PRAM model, where processor allocation must be explicit and global.

Key words. computational geometry, line-segment intersection, parallel algorithms, parallel data structures, PRAM model

AMS(MOS) subject classifications. 68E05, 68C05, 68C25

1. Introduction.

One of the major thrusts of computational geometry research has been to show that we can solve many geometric construction 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, [6], [11], [12], [20], [25], [27], [31], [39]. 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. These worst-case examples seldom arise in practice, however. Thus, an algorithm whose running time is essentially linear in the size of the output will perform much better than the worst-case time on most inputs.

1.1. The problem.

One of the most studied of these problems is the problem of constructing the planar graph determined by the pairwise intersections of a set of line segments in the plane, i.e., the *segment arrangement* problem (see [6], [11], [12], [16], [30], [34]). This problem has several applications in computer graphics, for example, [21], [33], [37]. One of the oldest algorithms solving the segment arrangement problem is an elegant method by Bentley and Ottmann [6] published in 1979 that uses the now-famous "plane-sweeping" paradigm [16], [30], [34]. 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., if each segment is parallel to one of the coordinate axes), where k is the size of the output. Since k is $\Omega(n^2)$ in the worst case, the existence of an optimal algorithm, running in $O(n \log n + k)$ time, became an open problem. This gave rise to a considerable amount of research done to resolve this question (e.g., [12], [13], [19]), and Chazelle and Edelsbrunner showed in 1988 that we can in fact solve this problem in $\Theta(n \log n + k)$ time.

* Received by the editors May 17, 1989; accepted for publication (in revised form) September 12, 1990. This research was announced in preliminary form in the Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures, Association for Computing Machinery, New York, pp. 127–137. This research was supported by National Science Foundation grants CCR-8810568 and CCR-9003299, and National Science Foundation and Defense Advanced Research Projects Agency under grant CCR-8908092.

† Department of Computer Science, The Johns Hopkins University, Baltimore, Maryland 21218 .

In this paper we investigate how efficiently we can solve this problem in parallel. Our primary goal is to design a parallel algorithm that runs as fast as possible. Given that, our secondary goal is to design an algorithm that has a time-processor product that is as small as possible. Our motivation for this is that we desire an algorithm that can be simulated on a real machine, with a fixed constant number of processors, so as to maximize the speedup over the best-known sequential algorithms. The product of the time bound and processor bound characterizes the *work* that such a simulation would perform, and provides a simple measure of the algorithm's efficiency relative to the best known sequential algorithms. Thus, for the segment arrangement problem, we desire an algorithm that runs in $O(\log n)$ time and has an output-sensitive work bound.

1.2. Previous work. Prior to a preliminary announcement of this research [22], we knew of no previous work for solving this problem in parallel, other than the trivial brute-force method based on sorting that runs in $O(\log n)$ time using $O(n^2)$ processors (e.g., using Cole's sorting method [15]). The only known results were for solving special cases of the segment arrangement problem in parallel. For example, Atallah, Cole, and Goodrich [3] addressed the decision version of this problem, i.e., determining if *any* two segments intersect, deriving a method running in $O(\log n)$ time using $O(n)$ processors. In [14] Chow studied a restricted version of the problem: namely, she showed how to determine all the pairwise intersections of n iso-oriented segments. Her algorithm runs in $O((1/\epsilon)\log n + k_{\max})$ time using $O(n^{1+\epsilon})$ processors [14], 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 it balance the computational burden for the case when only a few segments cause the majority of intersections. Neither of these approaches seem to extend to the general segment arrangement problem.

Following the preliminary announcement of this research, however, there have been a number of results that apply to this problem. In particular, Anderson, Beame, and Brisson [2] and Hagerup, Jung, and Welzl [26] have studied the related problem of constructing the arrangement of n lines in the plane (which, of course, always has $\Theta(n^2)$ size), a problem that can be solved sequentially in $O(n^2)$ time [13], [17], [19]. The method of Anderson, Beame, and Brisson builds upon the methods presented in [22] to derive a parallel algorithm running in $O(\log n \log^* n)$ time using $O(n^2/\log n)$ processors in the CREW PRAM model. The method of Hagerup, Jung, and Welzl is a randomized method running in $O(\log n)$ expected time using $O(n^2/\log n)$ processors in the CRCW PRAM model. Subsequentially, Goodrich has improved upon these methods to derive a deterministic method running in $O(\log n)$ time using an optimal $O(n^2/\log n)$ processors in the CREW PRAM model [23], solving an open problem posed in the preliminary version of this paper [22]. Of course, if we apply these methods to the segment arrangement problem, then these methods are efficient only if k, the number of intersections, is large.

In addition to these algorithms for the line arrangement problem, Rüb (see [36]) has independently shown that one can solve the segment arrangement problem in $O(\log n \log\log n)$ time using $O(n + k)$ processors in the CREW PRAM model. Her method improves upon the line arrangement algorithms, then, for instances when k is not too large (e.g., $k << n^2/\log n \log\log n$).

1.3. Our results. The main result of this paper is an output-sensitive parallel algorithm for solving the segment arrangement problem. Our algorithm runs in $O(\log n)$ time using $O(n \log n + k)$ processors, where k is the size of the output. Note

that the work performed by our algorithm matches the time-processor product of the brute-force approach when the output size is large, i.e., when k is $\Omega(n^2)$, and is smaller than the method of Rüb for $k >> n \log n / \log \log n$. 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. Our model of computation is the CREW PRAM model, where processor allocation must be explicit and global.

The main obstacle to designing an output-sensitive parallel algorithm for the general segment arrangement problem is that paradigms that led to efficient sequential algorithms, such as plane-sweeping [16], [34], topological sweeping [12], [17], and incremental construction [16], [34], seem inherently sequential. Moreover, parallel techniques that worked well for parallelizing fast plane-sweeping algorithms, such as the plane-sweep tree [1], [3], cascading divide-and-conquer [3], and parallel sequence-evaluation [4], cannot be directly applied, for they require one to know a priori all the places where a sweeping line would need to stop. Such a requirement "begs the question" in the case of constructing a segment arrangement, for a sweep-line would need to stop at each intersection point.

Our algorithm, instead, is based on a number of new parallel algorithmic techniques, as well as a new geometric characterization of the types of intersections that can occur. The new parallel techniques include a "truncated" version of the zone lemma of [11]–[13], [18], and [19] and a method for reusing processors created for enumerating intersections of one type to then discover intersections of another type. The new geometric characterization is a "hierarchical" extension of a characterization due to Chazelle [11]. 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. Our algorithm for the special case when the input segments are iso-oriented also uses this dynamic-allocation paradigm, in addition to the use of a "compressed" version of the array-of-trees parallel data structure of Atallah, Goodrich, and Kosaraju [4].

In the next section we discuss dynamic processor allocation in more detail, and show how to solve an important dominance reporting problem using this paradigm in §3. This problem arises as a natural subproblem in our segment arrangement algorithm, which we describe at a high level in §4. We give the details of our method in §§5 and 6. In §7 we present our algorithm for the iso-oriented case, and we conclude in §8.

2. A word about the computational model. The computational model we use in this paper is the Parallel Random Access Machine, or PRAM. Processors in this model act in a synchronous fashion and use a shared memory space. This model is divided into three types based on how memory can be accessed: the Exclusive-Read, Exclusive-Write (or EREW) model, the Concurrent-Read, Exclusive-Write (or CREW) model, and the Concurrent-Read, Concurrent-Write (or CRCW) model. All of our algorithms are for the CREW PRAM model.

Given an input of size n, the traditional way of utilizing this model is that we simply allocate, 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 of processors, c, not a number that is a function of n. Thus, the c real processors must 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 speedup possible we allow the set of virtual processors to grow dynamically.

There are essentially two different ways to allow for a dynamically growing pool of

virtual processors. One approach, as outlined by Reif and Sen [35], is that of allowing a new virtual processor to be created by having some existing virtual processor execute a *spawning* operation. Such an operation is issued by an existing processor specifying the task that a new processor is to perform. Then, 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 [8]. This model does not specify how to implement the processor assignment should a number of different virtual processors simultaneously perform spawning operations, however.

The model we use in this paper does not allow for the spawning operation. Instead, we insist that for r new virtual processors to be allocated in time t we must have already constructed an r-element array that stores pointers to the r tasks these processors are to begin performing in step $t+1$. We refer to this as a *global allocation* scheme. This is essentially the same as the traditional PRAM model, in that every PRAM algorithm does such an allocation as its first step, usually to allocate a number of virtual processors that is a function of the input size.

It is beyond the scope of this paper to address all the relative strengths of the various dynamic processor allocation schemes. Nevertheless, we would like to mention that, in spite of its apparent weakness, the global allocation CREW PRAM model can simulate any algorithm designed for the spawning CREW PRAM model in a work-optimal fashion.

LEMMA 2.1. *If an algorithm A runs in t steps using p processors in the* CREW PRAM *model with local spawning of processors allowed, then A can be implemented in $O(t \log p)$ steps using $O(p/\log p)$ processors in the* CREW PRAM *model with global processor allocation.*

Proof. Let p_i denote the number of processors used in the spawning PRAM model in step i, and let T_i denote the list of tasks to be performed in step i, with $T_i[j]$ being the task to be performed by processor j. The main idea of the proof is to simulate step i of the spawning PRAM algorithm in $O(\log p_i)$ time using $\lceil p_i/\log p_i \rceil$ processors in the global-allocation PRAM model. We begin by performing all the nonspawning operations of step i. This can easily be done in $O(\log p_i)$ time using $\lceil p_i/\log p_i \rceil$ processors, by a simple application of Brent's theorem [10]. We then perform a parallel prefix computation[1] to determine $p_{i+1} - p_i$, the number of new processors that are to be spawned in step i, and to which tasks they are to be assigned. (Recall that a parallel prefix computation is one in which we reduce a problem to the problem of computing all prefix sums $s_k = \sum_{i=1}^{k} a_i$ of a list of numbers $(a_1, a_2, \cdots, a_n)$.) This gives us T_{i+1} and takes $O(\log p_i)$ time using $\lceil p_i/\log p_i \rceil$ processors [28], [29]. We complete the processing for step i by requesting $\lceil p_{i+1}/\log p_{i+1} \rceil - \lceil p_i/\log p_i \rceil$ new processors, bringing the total to $\lceil p_{i+1}/\log p_{i+1} \rceil$. This prepares us to simulate the next step in A. Thus, the entire algorithm can be implemented in $O(t \log p)$ time using $O(p/\log p)$ processors in the global-allocation PRAM model, where $p = p_t$. □

In the next section we illustrate the power of dynamic processor allocation by describing a simple, efficient parallel method for solving an important dominance reporting problem, which arises naturally in our segment arrangement algorithm.

[1] Recall that a parallel prefix computation is a reduction to the problem of computing all prefix sums $s_k = \sum_{i=1}^{k} a_i$ for n numbers $(a_1, a_2, \cdots, a_n)$, where $+$ is any associative operation. Also recall that this problem can be solved in $O(\log n)$ time using $O(n/\log n)$ processors [28], [29].

3. Dominance reporting. Suppose we are given two point sets A and B, consisting of n and m points, respectively. Moreover, suppose the points in A and B are sorted by increasing x-coordinates. We wish to construct, for each point p in B, a list that contains each point q in A such that $x(q) < x(p)$ and $y(q) < y(p)$, i.e., each point in A that p *dominates*. We let $\mathrm{Dom}(A, p)$ denote this set, and refer to this problem as the *two-set dominance reporting* problem.

We do not know of any previous work for this problem. Atallah, Cole, and Goodrich [3] address the counting version of this problem (where one is simply interested in determining the value of $|\mathrm{Dom}(A, p)|$, the number of points of A that p dominates), deriving an algorithm that runs in $O(\log N)$ time using $O(N)$ processors, where $N = \max\{n, m\}$. In this section we show how to construct $\mathrm{Dom}(A, p)$ for each p in B in $O(\log N)$ time using $O(N/\log N + l)$ processors, where l is the total number of answers ($l = \sum_{p \in B} |\mathrm{Dom}(A, p)|$). Our method uses a different approach than that taken by Atallah, Cole, and Goodrich.

3.1. A simple data structuring approach. We first describe a solution based on the use of a simple data structure and global processor allocation. This method runs in $O(\log N)$ using $O(N + l)$ processors. We then show how to reduce the number of processors to $O(N/\log N + l)$ by some processing steps.

The approach is to build a data structure for the points of A and then query this structure for each point in B in parallel. In particular, the data structure, D, consists of a complete binary tree T with the points of A stored in its leaves in left-to-right order. Let v be an internal node of T; and let z, u, and w be, respectively, the parent, left child, and right child of v. For each such v we store a list $A(v)$, which contains all the points stored in descendants of v sorted by their y-coordinates. In addition, we augment each element p of $A(v)$ with pointers to p's predecessor in $A(z)$, $A(u)$, and $A(w)$ (recall that p's predecessor in a list $A(*)$ is the largest element in $A(*)$ smaller than p), using y-coordinates as comparison keys. Such a structure is easily constructed by Cole's parallel mergesort method [15] in $O(\log N)$ time using $O(N)$ processors.

We then perform two queries for each point p in B. The first query is to determine the size of $\mathrm{Dom}(A, p)$, and the second query is to construct $\mathrm{Dom}(A, p)$. The first query is answered by searching for the leaf position of x in T, starting at the root, while simultaneously locating the position of y in each $A(v)$ list such that v is on the left fringe of the search path (i.e., v is the left child of a node on the search path but is, itself, not on the search path). The elements less than y in each such $A(v)$ constitute the set of answers for p. Thus, we can perform this counting query for any p in $O(\log N)$ time, using a single processor, simply by adding up the ranks of the predecessor of p in each of these lists. Given the sizes of all the $\mathrm{Dom}(A, p)$ lists we can then perform a parallel prefix computation to determine the total number of answers and allocate the space for a global array Dom that will store all the $\mathrm{Dom}(A, p)$ lists as subarrays. We can then perform a global allocation of l processors that then collectively enumerate the answers in $O(\log N)$ time, filling in all the "slots" in the Dom array. Thus, the total procedure can be implemented in $O(\log N)$ time using $O(N + l)$ processors.

3.2. Improving the processor bounds. The method described above suffers from two inefficiencies: (1) it builds the data structure D using every point in A, including points that will not be included in any $\mathrm{Dom}(A, p)$ list, and (2) it performs a dominance reporting query for each point p in B, even if $\mathrm{Dom}(A, p)$ may turn out

to be empty. We can easily remove both of these inefficiencies by performing the following preprocessing steps, however:

(1) In this step we remove from B each point p such that $\mathrm{Dom}(A,p)$ is empty. We can determine, for each p in B, whether or not $\mathrm{Dom}(A,p)$ is empty by performing a parallel prefix computation on A to determine, for each q in A, the value $\mathrm{Min}Y(q) = \min_{q' \in A}\{y(q') : x(q') \leq x(q)\}$, and then performing a merge of A and B by increasing x-coordinates. Both of these operations, of course, take advantage of A and B being presorted. For any point p in B, it is easy to see that $\mathrm{Dom}(A,p) \neq \emptyset$ if and only if $\mathrm{Min}Y(q) < y(p)$, where q is the immediate predecessor of p in A. Given the merging of A and B, we can easily test this condition for each p in B in $O(1)$ time, and then perform a parallel prefix data compression procedure to remove any p's from B such that $\mathrm{Dom}(A,p) = \emptyset$. This step can be easily implemented in $O(\log N)$ time using $O(N/\log N)$ processors [9], [28], [29], [38].

(2) In this step we remove from A each point q that is not contained in any $\mathrm{Dom}(A,p)$ list. We do this using a method very similar to that used in Step 1. The time and processor bounds are as in Step 1.

Clearly, the total number of remaining points in A and B is dominated by l, the number of answers. Thus, by following this preprocessing step by the dominance reporting procedure described in the previous subsection, we derive the following lemma.

LEMMA 3.1. *Given two point sets A and B, with n and m points, respectively, sorted by increasing x-coordinates, we can construct* $\mathrm{Dom}(A,p)$ *for each p in B in* $O(\log N)$ *time using* $O(N/\log N + l)$ *processors in the* CREW PRAM *model, where* $N = n + m$ *and* $l = \sum_{p \in B} |\mathrm{Dom}(A,p)|$.

We make considerable use of this lemma in our method for constructing the arrangement of a collection of line segments. In fact, we usually need to solve a collection of 2-set dominance reporting problems in parallel. This presents no real problems, however, as we show in the following lemma.

LEMMA 3.2. *Given h instances of the 2-set dominance reporting problem, specified by h pairs of points sets* (A_1, B_1), (A_2, B_2), $\cdots$, (A_h, B_h), *we can construct* $\mathrm{Dom}(A_i, p)$ *for each point* $p \in B_i$, $i = 1, \cdots, h$, *in* $O(\log N)$ *time using* $O(N/\log N + l)$ *processors in the* CREW PRAM *model, where* $N = \sum_{i=1}^{h} |A_i| + |B_i|$ *and* $l = \sum_{i=1}^{h} \sum_{p \in B_i} |\mathrm{Dom}(A_i, p)|$.

Proof. The proof follows from a straightforward implementation of the method used to prove Lemma 3.1. The only modification necessary is that each place in the algorithm where a parallel prefix computation is performed (as a precursor to an allocation of new virtual processors), we must now coordinate h simultaneous parallel prefix computations. This is due to the requirement that dynamic processor allocation be global. As it does not raise any real difficulties, we leave the details of this implementation to the reader. □

Having discussed our computational model, and how it can be used for dominance reporting, we now give an overview of our method for constructing the arrangement of a collection of line segments.

4. An overview of our algorithm. Suppose we are given a set S of n line segments in the plane. We define the *upper* (respectively, *lower*) *vertical shadow* in S of a point p to be the point on the first segment in S that is intersected by the vertical ray emanating upward (respectively, downward) from p, if such a point exists. The *segment arrangement* of S is defined to be the planar graph determined by the pairwise

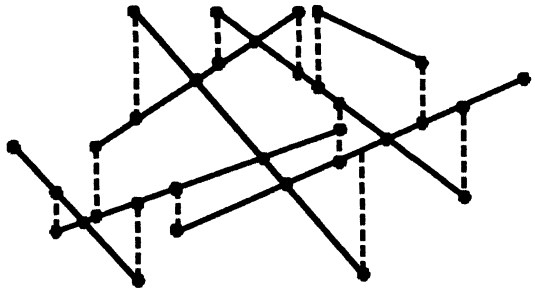

FIG. 1. *An example segment arrangement.*

intersections in S as well as all the vertical shadows of the endpoints of segments in S (see Fig. 1). The edges in this graph are determined by adjacent intersections (along some segment s) and by segment endpoints and their vertical shadows. For simplicity, we assume that at most two segments meet at any intersection point. We can easily modify our method to allow for multiple segments intersecting in the same point (using an appropriate definition of the "multiplicity" of an intersection point).

4.1. Characterizing intersections. Before we give our algorithm overview, we review an observation by Chazelle [11] for characterizing segment intersections in terms of a segment tree data structure [7]. Let T be the complete binary tree whose at most $2n+1$ leaves, in left-to-right order, correspond to the regions, called *slabs*, determined by placing a vertical line through each endpoint of each segment 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 *spans* a slab Π_v if s intersects both the left and right boundary of Π_v. A segment s_i *covers* a node $v \in T$ if it spans Π_v but not $\Pi_{\text{parent}(v)}$. Clearly, no segment covers more than two nodes on any level of T; hence, each segment covers at most $O(\log n)$ nodes of T. A segment s_i *ends in* Π_v if s_i does not span Π_v, but has an endpoint in Π_v. For each node $v \in T$ we define the following sets (see Fig. 2):

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

We can characterize the intersections in S as follows.

OBSERVATION 4.1 ([11]). 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) $s_1, s_2 \in \text{Cover}(v)$,
(2) $s_1 \in \text{End}(v)$ and $s_2 \in \text{Cover}(v)$,
(3) $s_2 \in \text{End}(v)$ and $s_1 \in \text{Cover}(v)$.

We call intersections of type 1 CC-*intersections* and intersections of types 2 and 3 EC-*intersections.*

4.2. The method. Our method, which we describe below, is based on finding all the CC-intersections first, and then using those intersections to help determine all

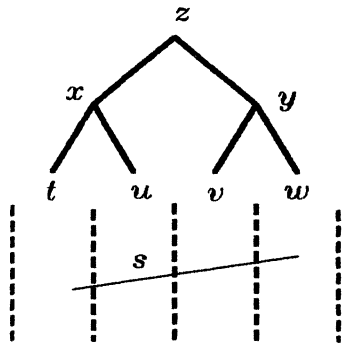

FIG. 2. *The segment s is in* Cover(u) *and* Cover(v), *as well as* End(t), End(w), End(x), End(y), *and* End(z).

the EC-intersections.

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 $v \in T$. In addition, for each v in T, 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. This step can be easily implemented in $O(\log n)$ time using $O(n \log n)$ processors in the CREW PRAM model, by using the method of Aggarwal et al. [1] to construct T and the method of Cole [15] to sort each Cover(v) list (since the total size of all the Cover(v)'s is $O(n \log n)$).

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 segment intersections along Π_v's right boundary rather than basing comparisons on segments intersections along Π_v's left boundary. We implement this step via a reduction to the dominance reporting problem, constructing, for each v in parallel, and for each segment s in Cover(v), a list of the other segments in Cover(v) that intersect s. We use these lists to construct the arrangement of the segments in Cover(v), which, following the convention of [11] and [12], we call the *hammock*. 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 our most involved step we compute all the EC-intersections in S. We implement it in two phases. In the first phase we find, for each $s \in$ End(v), all the EC-intersections of s with segments in Cover(v), so long as there are fewer than $c \log n$ such intersections (c is a constant parameter), or, alternatively, we determine if there are at least $c \log n$ such intersections. This requires $O(\log n)$ time using a processor per segment in End(v), for all v in T, and is based on a "truncated" version of the zone lemma of [11]–[13], [18], and [19]. In the second phase, then, we find, for each $s \in$ End(v), all the EC-intersections of s with segments in Cover(v), provided s has at least $c \log n$ such intersections. We restrict this phase to such segments, because our second phase requires at least $O(\log n)$ processors for each segment involved, and we wish to "charge" the cost of these processors to the intersections found. Our method runs in $O(\log n)$ time and takes advantage of a characterization similar to that of Observation 4.1. 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.

So, assuming we can implement each of the above steps in the stated bounds, then we can enumerate all the pairwise 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.

5. Computing CC-intersections. In Step 2 we compute all the CC-intersections in S. Let us concentrate on the problem of finding all the CC-intersections for a specific node v in T; we perform this computation for each v in parallel. Recall that in Step 1 we constructed all the Cover(v) lists for the nodes in T. For each segment s in Cover(v), let $y_1(s)$ (respectively, $y_2(s)$) denote the y-coordinate of the intersection of s with the left (respectively, right) boundary of Π_v. The following observation characterizes all CC-intersections in terms of these labels.

OBSERVATION 5.1. 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 5.1 in terms of dominance relationships. Namely, a segment r has a CC-intersection with s if and only if p_r is (i) above and to the left of p_s, or (ii) below and to the right of p_s. Thus, determining all CC-intersections in some slab Π_v can be reduced to two instances of the 2-set dominance reporting problem, where the set Cover(v) plays the roles of both sets A and B of Lemma 3.2. Of course, we must also re-orient the x- and y-axes so that the dominance relation of interest is downward and to the left. Note that the condition of Lemma 3.1 requiring that the points in A and B be presorted by their first coordinates is immediately satisfied, since the segments in each Cover(v) list are sorted by the y-coordinates of their intersections with the left boundary of Π_v. Of course, since we must implement this step for all nodes v in parallel, we must apply Lemma 3.2. Therefore, since the total size of all the Cover(v) lists is $O(n \log n)$, this entire computation can be implemented in $O(\log n)$ time using $O(n + \alpha)$ processors, where α is the total number of CC-intersections.

5.1. Constructing the hammock. 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, respectively, to which p is adjacent. We do this by sorting, for each s in parallel, the intersections along s (which were just computed) 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-intersections for Cover(v) (e.g., [5], [24], [32], [34]). We finish the construction by augmenting the graph, as Chazelle does [11], by adding two 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 this computation requires the sorting of $O(n \log n + \alpha)$ elements, it takes $O(\log n)$ time using $O(n \log n + \alpha)$ processors [15], which dominates the complexity of Step 2.

Thus, we have shown how to efficiently find all the CC-intersections in S and construct the hammock for each Cover(v) list. In the next section we address the problem of finding the EC-intersections in S.

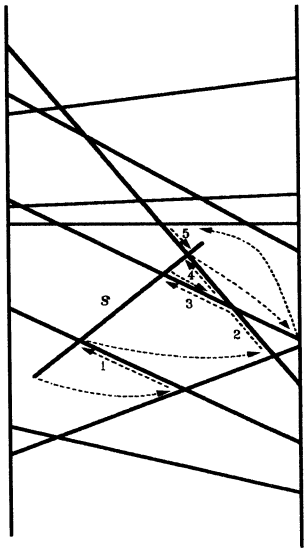

FIG. 3. *An example walk in the hammock. The traversed edges are numbered in the order they would appear in the walk in the hammock for s.*

6. Computing EC-intersections. To complete the algorithm we must implement Step 3, the finding of all the EC-intersections for each v in T. 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.

6.1. 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 v, e.g., using the method of Atallah, Cole, and Goodrich [3], which takes $O(\log n)$ time using $O(|\text{Cover}(v)| + \alpha_v)$ processors, where α_v is the number of CC-intersection determined by the segments in Cover(v). This requires $O(n \log n + \alpha)$ processors for all $v \in T$, and allows point locations to be performed in the hammock for a particular Cover(v) in $O(\log n)$ time using a single processor.

Suppose we are given a query segment s in End(v). We wish to find all the EC-intersections between s and segments in Cover(v), so long as there are fewer than $c \log n$ such intersections (where $c \geq 1$ is a constant parameter). We use the point location structure for Cover(v) to locate 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). We then mimic the method of Chazelle [11] for walking through the hammock from f_a to f_b, except that we cut the walk short as soon as it traverses $4c \log n$ edges. We show below 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 [11]. If $f_a = f_b$, then we are done, so let us assume $f_a \neq f_b$. We begin by jumping to the rightmost vertex v_1 in $f_1 = f_a$. We then traverse the edges of f_1 until we find the edge e_1 of f_1 that intersects s. If v_1 is above the line supporting s, then this traversal is clockwise, and is counterclockwise, otherwise. Upon reaching e_1, we use the adjacency information for e_1 to "hop" over

e_1 into the next face, f_2, which is adjacent to s. We then use the extra pointer for e_1 to jump to the rightmost vertex v_2 in f_2, going from face to face along s, provided that for each edge e traversed, the line supporting e intersects s. (See Fig. 3.) If we are about to traverse an edge whose supporting line does not intersect s, then we suspend the traversal from f_a at this point, and begin a symmetric traversal from f_b (using the rule that if v_i is above the line supporting s, then the traversal must be counterclockwise, and must be clockwise, otherwise). We continue this traversal until all the intersections along s have been discovered or, as in our case, we traverse at least $4c \log n$ edges. Chazelle [11] proves an important "zone" lemma for his scheme, establishing that if one uses his search strategy (without our extra stopping criterion, of course), then one will eventually discover all the intersections along s and the total time spent will be proportional to the number of intersections. The next lemma establishes a "truncated" version of this zone property.

LEMMA 6.1. *Suppose we have 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. Since this is a slightly stronger version of a lemma proved by Chazelle [11], we use the proof technique of Chazelle, Guibas, and Lee [13] 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 and each f is convex. Hence, for any face f, all the nonanchored 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 counterclockwise 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. So each intersection point can be charged at most four 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 must be at least $c \log n$ intersections of s with segments in Cover(v). Note, however, that the $c \log n$ intersection points need not be consecutive 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. Note that $\sum_{v \in T} |E_v|$ is at most $O(n \log n)$ and $\sum_{v \in T} |S_v|$ is at most $O(n \log n + \alpha)$.

6.2. Segments with many intersections. We have yet to find all the EC-intersections for the segments in E_v. Our method resembles a "recursive" application of the first two steps in our algorithm. Let us, then, concentrate on the computation for a specific node v in T, with the understanding that we perform this computation for all v in T in parallel.

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 lowercase letters. Thus,

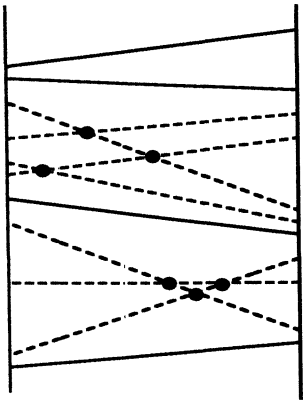

FIG. 4. *An example* π_w. *The segments in* end(w) *are shown dotted and the segments in* cover(v) *are shown solid.*

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 Fig. 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 intersect 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 [3] to build T_v. We use this method because it gives us the left(w) lists in sorted order without our having to explicitly perform a sorting operation. Note: the tree in the Atallah, Cole, and Goodrich construction is built on every $\lceil \log n \rceil$th x-coordinate; so that the end(w) list stored in a leaf has $O(\log n)$ size rather than $O(1)$ size. This will not affect the running time of our implementation by more than a constant factor, however. Their method runs in $O(\log m)$ time using $O(m)$ processors, where m is the number of segments. In our case $m = |\text{Cover}(v)| + \alpha_v$. Thus, we can use the processors created in Step 2 (to enumerate CC-intersections) to now help construct T_v for each v in T in parallel. This requires $O(\log n)$ time using 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 segments in S_v are all pieces of segments in Cover(v) that span Π_v. We exploit this property to characterize EC-intersections in the following lemma, in a manner analogous to that of Observation 4.1.

LEMMA 6.2. *Given a node* v *in* T, *let* s *be a segment in* E_v *and* t *be a segment in* Cover(v), *and suppose* s *and* t *intersect at a point* p. *In addition, let* T_v *and* S_v *be as above, and let* $\hat{t}$ *be the portion of* t *in* S_v *that contains* p. *Then there is a (unique) node* $w \in T_v$ *such that* $p \in \pi_w$ *and one of the following is true:*

(1) $\hat{t} \in$ cover(w) (*a "type 1" intersection*),
(2) $t \in$ inter(w) *and* s *covers* w (*a "type 2" intersection*),
(3) $t \in$ inter(w) *and* s *ends in* π_w, *where* w *is a leaf* (*a "type 3" intersection*).

Proof. Let z be the leaf in T_v that contains p. There are two cases:

(1) s ends in π_z. If $\hat{t}$ does not span z, then $t \in$ inter(z); hence, p is a type 3 intersection. If $\hat{t}$ spans z, then there must be an ancestor w of z that $\hat{t}$ covers; hence, p is a type 1 intersection.
(2) s spans z. Let w be the ancestor of z that s covers. If $\hat{t}$ also covers w, then p is again a type 1 intersection. Otherwise, if $\hat{t}$ has an endpoint in π_w, then $t \in$ inter(w); hence, p is a type 2 intersection. □

We implement Step 3, then, by searching for each type of intersection.

Type 1 *intersections.* For each segment s in E_v, we allocate $O(\log n)$ processors to s and perform the following query at each node w in T_v such that s has an endpoint in π_w or s covers w:

> 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. Note that this is possible, because the segments in cover(w) do not intersect, hence, are linearly ordered by the "above" relationship. All the segments in cover(w) between these two positions in the list must intersect s.

After performing this query, each processor assigned to s has determined some number of type 1 intersections for s, and, in fact, has an implicit representation of a list of these intersections. By performing a parallel prefix computation, then, we can allocate enough processors to enumerate all these type 1 intersections. This can easily be done in $O(\log n)$ time using $O(\sum_{v \in T} |E_v| \log n + \beta_1)$ processors (for all v in T), where β_1 is the total number of type 1 intersections.

Type 2 *intersections.* Our method is based on the observation that a type 2 intersection between $s \in E_v$ and $t \in \text{Cover}(v)$ is determined by a node w in T_v such that both s and t span π_w. Therefore, we can determine all such type 2 intersections by a reduction to the 2-set dominance reporting problem. In particular, we determine, for each node w in T_v, the set $ec(w)$ containing each segment $s \in E_v$ such that s covers w. We also sort each $ec(w)$ list by the y-coordinates of the points formed by the intersections of the segments in $ec(w)$ and the left boundary of π_w. This takes $O(\log n)$ time using $O(|E_v| \log n)$ processors. Note that the list left(w) stores a piece of each segment in inter(w), and these pieces are sorted by the y-coordinates of the points formed by the intersections of the segments in inter(w) and the left boundary of π_w. We associate a pair $(y_1(s), y_2(s))$ with each segment s in $ec(w)$ (respectively, left(w)), where $y_1(s)$ (respectively, $y_2(s)$) is the y-coordinate of the intersection of the left (respectively, right) vertical boundary of π_w with s. Thus, if interpreted as points, the elements of $ec(w)$ and left(w) are sorted by their first coordinates, satisfying the ordering precondition of Lemma 3.1. Just as in our method of §5, a solution to two instances of the 2-set dominance reporting problem gives us all the intersections between the "points" in $ec(w)$ and left(w) for each w in T_v. By Lemma 3.2 this takes $O(\log n)$ time using $O(n \log n + \sum_{v \in T} |E_v| + \alpha + \beta_2)$ processors (for all v in T), where β_2 is the number of type 2 intersections, since the total size of all $ec(w)$ lists is at most $O(\sum_{v \in T} |E_v| \log n)$ and the total size of all the left(w) lists is at most $O(n \log^2 n + \alpha \log n)$. Thus, the total number of processors needed for this step is $O(n \log n + \sum_{v \in T} |E_v| \log n + \alpha + \beta_2)$.

Type 3 *intersections.* Each type 3 intersection is determined by a leaf node w in T_v. Since $|\text{inter}(w)|$ in this case is $O(\log n)$, we can find all type 3 intersections by assigning a processor to each segment s in E_v and visiting each node w in T such that s ends in π_w. This processor simply tests each segment t with a piece $\hat{t}$ in end(w) to see if t intersects s. This clearly takes $O(\log n)$ time using $O(|E_v|)$ processors.

Having determined all three types of intersections completes the computation of the EC-intersections, giving us all the pairwise intersections of the segments in S. The total time needed is clearly $O(\log n)$. The total number of processors needed is $O(n \log n + \sum_{v \in T} |E_v| \log n + \alpha + \beta)$, where β is the number of EC-intersections. By construction, however, each segment s in E_v determines at least $c \log n$ EC-intersections; hence, $\sum_{v \in T} |E_v| \log n$ is $O(\beta)$. Therefore, the total number of processors needed is $O(n \log n + \alpha + \beta)$.

We complete our algorithm by constructing the segment arrangement, without vertical shadows, from the intersection points and endpoints, using essentially the same method we used to construct the hammocks (i.e., by sorting the intersections along each segment). We then augment this structure with the vertical shadows by applying the trapezoidal decomposition algorithm of Atallah, Cole, and Goodrich [3] and the sorting algorithm of Cole [15]. This takes $O(\log n)$ time using $O(n + k)$ processors, where $k = \alpha + \beta$. We summarize as follows.

THEOREM 6.3. *Given a set S of n line segments in the plane, we can construct the segment arrangement for S in $O(\log n)$ time using $O(n \log n + k)$ processors in the* CREW PRAM *model, where k is the size of the output.*

Thus, one can construct a segment arrangement efficiently in parallel in an output-sensitive manner. In the next section we show how to perform this construction optimally for the important special case when the segments are iso-oriented.

7. Iso-oriented segments. In this section we show how to construct the segment arrangement when all the segments are parallel to the x- or y-axes. Our method runs in $O(\log n)$ time using $O(n + k/\log n)$ processors in the CREW PRAM model, which is optimal. Since our algorithm is based on a "compressed" version of the *array-of-trees* parallel data structure of Atallah, Goodrich, and Kosaraju [4], we begin by reviewing this structure.

7.1. The array-of-trees. Suppose we are given a sequence $\sigma = (\sigma_1, \sigma_2, \cdots, \sigma_n)$, of **insert**$(a)$ and **delete**(a) operations. Let a_t denote the argument of the operation σ_t, and let A be the list of all distinct a_t values stored in sorted order. Also let A_t denote the set of items from A that would be present at "time" t if the operations $(\sigma_1, \cdots, \sigma_t)$ were evaluated sequentially, assuming that the initial set is $\emptyset$. A *tree query* is any query operation that can be performed on a complete binary tree T with $O(n)$ nodes in $O(\log n)$ time assuming that elements are stored in the leaves of T and each internal node v of T can store the values of $O(1)$ functions applied to values stored in v's children. Examples of such tree queries include the computation of the maximum y-coordinate of v's descendents or the computation of the number of v's descendents. The array-of-trees data structure allows us to perform any tree query on any given A_t in $O(\log n)$ time, assuming all the elements of A_t were stored in the leaves of a complete binary tree T. In fact, this structure can be viewed as an array of trees $(T_1, T_2, \cdots, T_n)$, where T_t is a complete binary tree whose leaves correspond to the elements of A, one element per leaf, such that the leaves associated with elements of A_t are *active* while all others are *in-active* (i.e, they store the **nil** value).

The "skeleton" of the array-of-trees is a complete binary tree T whose leaves are associated with the elements in A, one per leaf. For each a in A we construct $\sigma(a)$, the subsequence of σ consisting of all operations that have a as their argument. Note: with each operation in $\sigma(a)$ we store its position in σ; in fact, each time we refer to a σ_t, t denotes its index in σ. Using parallel sorting [15], it is easy to construct A, T, and all the $\sigma(a)$'s in $O(\log n)$ time using $O(n)$ processors.

For each v in T we construct a list $B(v)$ of records $(R_1, R_2, \cdots, R_{l_v})$ such that each R_i has the following fields:

(1) time, the index (time) when R_i becomes active.
(2) left, a pointer to the left child of R_i.
(3) right, a pointer to the right child of R_i.
(4) val, the value stored at R_i.
(5) Labels, a list of $O(1)$ labels, each of which is the result of an associative function applied to the values stored at the children of R_i.

Intuitively, each R_i represents a node in a complete binary tree rooted at v whose leaves (which are the same as those of the subtree rooted at v) and are either active or **nil**. The record R_i is active at time R_i.time and remains active until there is a change in one of the descendent nodes of R_i, namely, at time R_{i+1}.time.

More formally, suppose v is a leaf node, which, say, is associated with the element $a \in A$. Also suppose $\sigma(a) = (\sigma_{t_1}, \sigma_{t_2}, \cdots, \sigma_{t_{l_v}})$. Then $B(v) = (R_0, R_1, \cdots, R_{l_v})$, where the record R_i is associated with the operation σ_{t_i}, for $i = 1, 2, \cdots, l_v$. Specifically, given σ_{t_i} in $\sigma(a)$, we define the record R_i so that R_i.time $= t_i$, and R_i.left $= R_i$.right $=$ **nil**. If $\sigma_t =$ **insert**(a), then R_i.val $= a$, and if $\sigma_t =$ **delete**(a), then R_i.val $=$ **nil**. Each label in the Labels list is initialized based on R_i.val and the semantics of the function that defines that label. For example, if the label is "number of active descendents," then this label is "1" if R_i.val $= a$, and this label is 0 if R_i.val $=$ **nil**. The record R_0 represents the initial condition, i.e., $R_0 = (0, \textbf{nil}, \textbf{nil}, \textbf{nil}, L_0)$. Intuitively, each record in $B(v)$ is the node in a one-node binary tree that stores a "snapshot" of the history of a with respect to an evaluation of σ.

Now suppose v is an internal node with left child u and right child w. In this case there is a record in $B(v)$ for each record in $B(u) \cup B(w)$. More formally, let $B(u) = (U_0, U_1, \cdots, U_{l_u})$, and $B(w) = (W_0, W_1, \cdots, W_{l_w})$. Also let $(t_0, t_1, t_2, \cdots, t_{l_v})$ be the sorted list of *time* fields from the records in $B(u) \cup B(w)$, where $l_v = l_u + l_w - 1$ (we only store one copy of $t_0 = 0$). We define $B(v) = (R_0, R_1, \cdots, R_{l_v})$, where the record R_i is defined so that R_i.time $= t_i$, R_i.left points to the record U_j with largest index j such that U_j.time $\leq t_i$, and R_i.right points to the record W_j with largest index j such that U_j.time $\leq t_i$. In addition, R_i.val $=$ **nil**, and each label in R_i.Labels is defined by applying the appropriate function to the corresponding labels stored at the records that R_i.left and R_i.right point to. For example, if the label is "number of active descendents," then we simply need to add the corresponding labels from the records R_i.left and R_i.right. Intuitively, each record in $B(v)$ is the root of a binary tree that stores a "snapshot" of the history of the elements associated with the descendents of v with respect to an evaluation of σ.

Atallah, Goodrich, and Kosaraju [4] show that we can exploit the recursive structure of the $B(v)$ definitions to construct $B(v)$ for each v in T in $O(\log n)$ time with $O(n)$ processors in the CREW PRAM, using the cascading divide-and-conquer technique of Atallah, Cole, and Goodrich [3].

7.2. The compressed array-of-trees. In our algorithm we use a compressed version of the array-of-trees data structure. The compressed array-of-trees consists of T as above, with a list $B'(v)$ of records stored at each node v in T. The main idea of the compressed array-of-trees is to force each "tree" in $B(\text{root}(T))$ to (1) only store pointers leading to active elements, and (2) not have any internal nodes that have only one child. (See Fig. 5.)

Our method for enforcing this property is as follows. If v is a leaf of T, then the fields of each record in $B'(v)$ are defined as above, i.e., $B'(v) = B(v)$ in this case. If, on the other hand, v is an internal node in T (with left child u and right child w), then we define the structure of the records in $B'(v)$ to be slightly different from the structure of records in $B(v)$. For each record R_i in $B(v)$ there is a record R'_i in $B'(v)$, with R'_i.time $= R_i$.time and R'_i.Labels $= R_i$.Labels. The other fields in R'_i differ from their corresponding fields in R_i, however. In particular, let U denote R_i.left and W denote R_i.right, and let U' and W' denote the records corresponding to U and W in $B'(u)$ and $B'(w)$, respectively. Also let Desc(R) denote the set of all nonnull *val* fields in records reachable from R (by following left and right pointers). We define

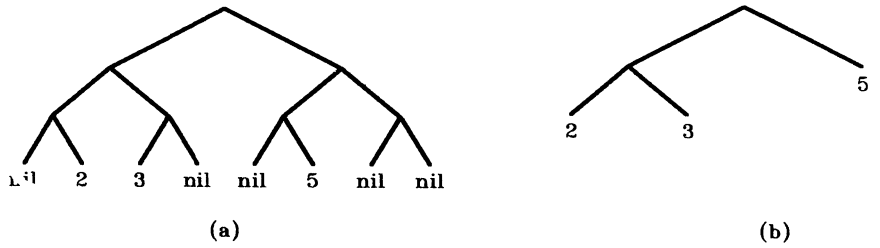

FIG. 5. *An example A_t represented* (a) *as a complete binary tree, and* (b) *as a compressed binary tree.*

the remaining fields of R'_i as follows:

- (1) if $\text{Desc}(R_i) = \emptyset$, then $R'_i.\text{left} = R'_i.\text{right} = \textbf{nil}$ and $R'_i.\text{val} = \textbf{nil}$.
- (2) If $\text{Desc}(R_i) = \{a\}$, then $R'_i.\text{left} = R'_i.\text{right} = \textbf{nil}$ and $R'_i.\text{val} = a$.
- (3) If $\text{Desc}(R_i) \neq \emptyset$ but $\text{Desc}(U) = \emptyset$, then $R'_i.\text{left} = W'.\text{left}$, $R'_i.\text{right} = W'.\text{right}$, and $R'_i.\text{val} = W'.\text{val}$.
- (4) If $\text{Desc}(R_i) \neq \emptyset$ but $\text{Desc}(W) = \emptyset$, then $R'_i.\text{left} = U'.\text{left}$, $R'_i.\text{right} = U'.\text{right}$, and $R'_i.\text{val} = U'.\text{val}$.

Note that to construct an R'_i we only need $\text{Desc}(R_i)$ if it contains a single element; otherwise, we need to know only the size of $\text{Desc}(R_i)$. This is itself an associative function. Thus, we can still use the method of Atallah, Goodrich, and Kosaraju [4] to construct $B'(v)$ for each v in T in $O(\log n)$ time with $O(n)$ processors in the CREW PRAM.

7.3. Determining iso-oriented intersections. Having described the compressed array-of-trees, let us return to the problem at hand, namely, the iso-oriented segment arrangement problem. Suppose we are given a set S of n iso-oriented line segments in the plane. We construct the compressed array-of-trees data structure to represent a horizontal plane-sweep (e.g., that of Bentley and Ottmann [6]) and use it to perform a range query for every position i that corresponds to a vertical segment. In particular, we use this data structure by sorting the endpoints of the horizontal segments in S in increasing order by x-coordinates; let Events denote this list. For each point $q_t = (x_t, y_t)$ in Events that is the left endpoint of a segment, we let $\sigma_t = \textbf{insert}(y_t)$, and for each $q_t = (x_t, y_t)$ in Events that is the right endpoint of a segment, we let $\sigma_t = \textbf{delete}(y_t)$. The labels we store in the Labels field for each record R in the compressed array-of-trees are $y$max, the maximum y-coordinate in the descendents of R, and desc, the number of active descendents of R. To perform the query for a vertical segment $s = \langle (x, y_1), (x, y_2) \rangle$ we first locate the point $q_t = (x_t, y_t)$ in Events such that t is the largest index satisfying $x_t \leq x$ (by a simple binary search). This immediately gives us σ_t, the operation associated with q_t. Intuitively, σ_t is the insertion or deletion event that would be encountered just before the query event for s in a sequential implementation of the plane-sweep. Given σ_t, we locate the record R in $B(\text{root}(T))$ with $R.\text{time} = t$. We then perform a search in the tree rooted at R to determine the number k_s of horizontal segments that have a y-value between y_1 and y_2 (using the $y$max and desc labels). This is easily done in $O(\log n)$ time using a single processor. We then assign $\lceil k_s / \log n \rceil$ processors to the task of enumerating these elements and placing them in a single array H_s. The ith processor in this collection is assigned to the task of enumerating the elements in the tree rooted at R

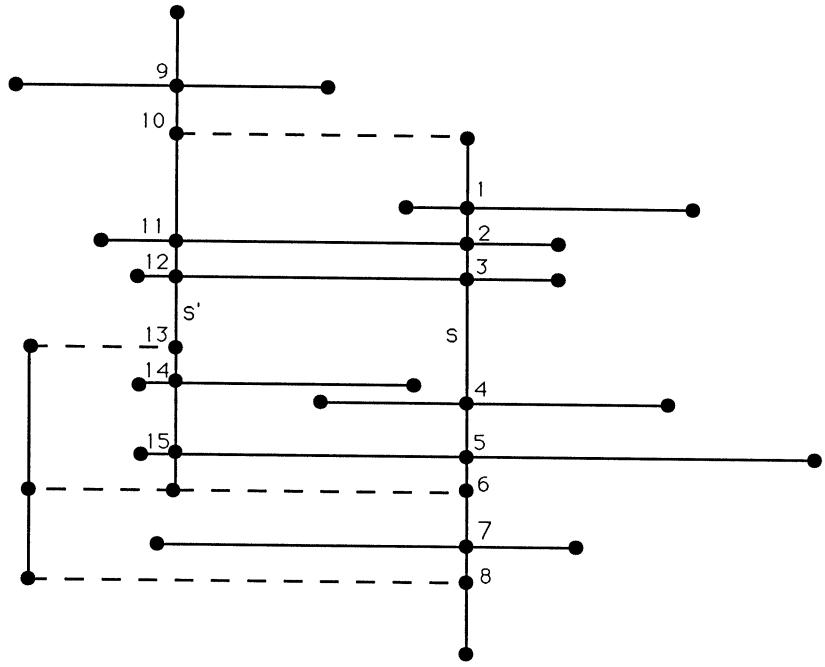

FIG. 6. *Using list merging to complete the construction of the iso-oriented segment arrangement. In this case,* $LV_s = (6,8)$, $RV_{s'} = (10)$, $H_s = (1,2,3,4,5,7)$, $H_{s'} = (9,11,12,14,15)$, $HL_{s',s} = (1,2,3,4,5)$, *and* $HR_{s,s'} = (11,12,14,15)$.

that are in the interval $[y_1, y_2]$ and of rank $i\lceil \log n \rceil$, $i\lceil \log n \rceil + 1, \cdots, (i+1)\lceil \log n \rceil - 1$. Since the tree rooted at R is compressed and the elements in its "leaves" are sorted by y-coordinate, the ith processor can use the $y$max and desc labels to locate all its elements in $O(\log n)$ time. Moreover, this also gives us all the vertical adjacencies in the segment arrangement for these intersection points. Thus, we have yet only to combine all the H_s lists to construct the segment arrangement.

We begin this combining procedure by using the cascading divide-and-conquer technique of Atallah, Cole, and Goodrich [3] to determine the horizontal shadows of each vertical segment s in S, i.e., the point on the first vertical segment intersected by a horizontal ray emanating out of the endpoints of s. This takes $O(\log n)$ time using $O(n)$ processors [3], and gives us $O(n)$ pairs of segments (s, s') such that s is horizontally "visible" from s'. Then, using parallel sorting [15], in $O(\log n)$ time we can construct, for each vertical segment s, two additional lists: LV_s, which is the sorted list all the horizontal shadows hitting s from the left, and RV_s, which is the sorted list of all the horizontal shadows hitting s from the right. These lists give us all the maximal pieces of s that are visible from another vertical segment in S from either the left or the right.

The remainder of the computation, which we illustrate in Fig. 6, consists of a number of list merging steps, where all lists are assumed to be sorted by y-coordinates. For each s in parallel we merge LV_s with H_s, the list of horizontal intersections along

s. We similarly merge RV_s with H_s. This can be implemented in $O(\log n)$ time using $O(n + k/\log n)$ processors using the merging methods of [9], and [38]. Let $HL_{s,s'}$ be the list of horizontal intersections in H_s that fall on the piece of s that is horizontally visible from s', where s' is to the left of s. Similarly, define $HR_{s,s'}$. Note that we can easily determine each $HL_{s,s'}$ and $HR_{s,s'}$ given the merges we have just performed (even if some of these lists are empty, since we have at least $O(n)$ processors). In parallel, for each pair of horizontally visible segments (s, s') such that s is to left of s', we merge $HL_{s',s}$ with $HR_{s,s'}$. Performing all these parallel merges gives us the horizontal adjacencies for each intersection point in H_s; hence, completes the construction. Since all these merges can also be performed in $O(\log n)$ time using $O(n + k/\log n)$ processors [9], [38], we have the following theorem.

THEOREM 7.1. *Given a set S of n iso-oriented segments in the plane, we 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.*

8. Conclusion. We have derived a parallel method for constructing the segment arrangement of a set of line segments in the plane in $O(\log n)$ time so that total work performed is only a $\log n$ factor from the sequential lower bound (which is achievable [12]). Moreover, we have shown how to solve the important iso-oriented special case of this problem with an optimal work bound. Thus, the obvious open problem that remains is to construct the segment arrangement in $O(\log n)$ time using only $O(n \log n + k)$ work.

Acknowledgments. We thank Mikhail J. Atallah, Richard Cole, Gregory Bachelis, and S. Rao Kosaraju for helpful discussions regarding the topics of this paper. We also thank an anonymous referee for several helpful comments, which significantly improved the presentation of §6.

REFERENCES

[1] A. AGGARWAL, B. CHAZELLE, L. GUIBAS, C. Ó'DÚNLAING, AND C. YAP, *Parallel computational geometry*, Algorithmica, 3 (1988), pp. 293–328.

[2] R. ANDERSON, P. BEAME, AND E. BRISSON, *Parallel algorithms for arrangements*, in Proc. 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, Island of Crete, Greece, 1990, pp. 298–306.

[3] M.J. ATALLAH, R. COLE, AND M.T. GOODRICH, *Cascading divide-and-conquer: A technique for designing parallel algorithms*, SIAM J. Comput., 18 (1989), pp. 499–532.

[4] M.J. ATALLAH, M.T. GOODRICH, AND S.R. KOSARAJU, *Parallel algorithms for evaluating sequences of set-manipulation operations*, in Proc. 3rd Aegean Workshop on Computing, AWOC 88, Lecture Notes in Computer Science 319, Springer-Verlag, Berlin, New York, 1988, pp. 1–10.

[5] B.G. BAUMGART, *A polyhedron representation for computer vision*, Proc. 1975 AFIPS National Computer Conference, 44, AFIPS Press, 1975, pp. 589–596.

[6] J.L. BENTLEY AND T. OTTMANN, *Algorithms for reporting and counting geometric intersections*, IEEE Trans. Comput., 28 (1979), pp. 643–647.

[7] J.L. BENTLEY AND D. WOOD, *An optimal worst case algorithm for reporting intersections of rectangles*, IEEE Trans. Comput., 29 (1980), pp. 571–576.

[8] S. BHATT AND J.Y. CAI, *Take a Walk, Grow a Tree*, Proc. 29th Annual IEEE Symposium on Foundations of Computer Science, White Plains, NY, IEEE Computer Society, Washington, DC, 1988, pp. 469–478.

[9] G. BILARDI AND A. NICOLAU, *Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines*, SIAM J. Comput., 18 (1989), pp. 216–228.

[10] R.P. BRENT, *The parallel evalutation of general arithmetic expressions*, J. Assoc. Comput. Mach., 21 (1974), pp. 201–206.

[11] B. CHAZELLE, *Reporting and counting segment intersections*, J. Comput. Systems Sci., 32 (1986), pp. 156–182.

[12] B. CHAZELLE AND H. EDELSBRUNNER, *An optimal algorithm for intersecting line segments in the plane,* Proc. 29th Annual IEEE Symposium on Foundations of Computer Science, White Plains, New York, IEEE Computer Society, Washington, DC, 1988, pp. 590–600.

[13] B. CHAZELLE, L.J. GUIBAS, AND D.T. LEE, *The power of geometric duality,* BIT, 25 (1985), pp. 76–90.

[14] A. CHOW, *Parallel algorithms for geometric problems,* Ph.D. thesis, Computer Science Department, University of Illinois, Urbana, IL, 1980.

[15] R. COLE, *Parallel merge sort,* SIAM J. Comput., 17 (1988), pp. 770–785.

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

[17] H. EDELSBRUNNER AND L.J. GUIBAS, *Topologically sweeping an arrangement,* in Proc. 18th Annual ACM Symposium on Theory of Computing, Berkeley, CA, Association for Computing Machinery, New York, 1986, pp. 389–403.

[18] 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, Department of Computer Science, University of Illinois, Urbana, IL, 1988.

[19] H. EDELSBRUNNER, J. O'ROURKE, AND R. SEIDEL, *Constructing arrangements of lines and hyperplanes with applications,* in Proc. 24th Annual IEEE Symposium on Foundations of Computer Science, Tucson, AZ, IEEE Computer Society, Washington, DC, 1983, pp. 83–91.

[20] S.K. GHOSH AND D.M. MOUNT, *An output sensitive algorithm for computing visibility graphs,* in Proc. 28th Annual IEEE Symposium on Foundations of Computer Science, Los Angeles, CA, IEEE Computer Society, Washington, DC, 1987, pp. 11–19.

[21] M.T. GOODRICH, *A polygonal approach to hidden-line elimination,* in Proc. 25th Annual Allerton Conference on Communication, Control, and Computing, Allterton, IL, 1987, pp. 849–858.

[22] M.T. GOODRICH, *Intersecting line segments in parallel with an output-sensitive number of processors,* in Proc. 1989 Annual ACM Symposium on Parallel Algorithms and Architectures, Santa Fe, NM, Association for Computing Machinery, New York, pp. 127–137.

[23] ———, *Constructing arrangements optimally in parallel,* Tech. Report 90/06, Department of Computer Science, The Johns Hopkins University, Baltimore, MD, 1990.

[24] L.J. GUIBAS AND J. STOLFI, *Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams,* ACM Trans. Graphics, 4 (1985), pp. 75–123.

[25] R.H. GÜTING, *An optimal contour algorithm for iso-oriented rectangles,* J. Algorithms, 5 (1984), pp. 303–326.

[26] T. HAGERUP, H. JUNG, AND E. WELZL, *Efficient parallel computation of arrangements of hyperplanes in d dimensions,* in Proc. 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, Island of Crete, Greece, Association for Computing Machinery, New York, 1990, pp. 290–297.

[27] J. HERSHBERGER, *Finding the visibility graph of a simple polygon in time proportional to its size,* in 3rd ACM Symposium on Computational Geometry, Waterloo, Ontario, Canada, Association for Computing Machinery, New York, 1987, pp. 11–20.

[28] C.P. KRUSKAL, L. RUDOLPH, AND M. SNIR, *The power of parallel prefix,* in Proc. 1985 Internat. Conference on Parallel Processing, St. Charles, IL, pp. 180–185.

[29] R.E. LADNER, AND M.J. FISCHER, *Parallel Prefix Computation,* J. Assoc. Comput. Mach., 27 (1980), pp. 831–838.

[30] D.T. LEE AND F.P. PREPARATA, *Computational geometry—a survey,* IEEE Trans. Comput., 33 (1984), pp. 872–1101.

[31] W. LIPSKI, JR. AND F.P. PREPARATA, *Finding the contour of a union of iso-oriented rectangles,* J. Algorithms, 1 (1980), pp. 235–246.

[32] D.E. MULLER AND F.P. PREPARATA, *Finding the intersection of two convex polyhedra,* Theoret. Comput. Sci., 7 (1978), pp. 217–236.

[33] O. NURMI, *A fast line-sweep algorithm for hidden line elimination,* BIT, 25 (1985), pp. 466–472.

[34] F.P. PREPARATA AND M.I. SHAMOS, *Computational Geometry: An Introduction,* Springer-Verlag, New York, 1985.

[35] J. REIF AND S. SEN, *An efficient output-sensitive hidden-surface removal algorithm and its parallelization,* in Proc. 4th ACM Symposium on Computational Geometry, Urbana-Champaign, IL, Association for Computing Machinery, New York, 1988, pp. 193–200.

[36] C. RÜB, *Parallele Algorithmen zum Berechnen der Schnittpunkte von Liniensegmenten,* Ph.D. dissertation, Universität des Saarlandes, Saarbrucken, Germany, 1990.

[37] A. SCHMITT, *Time and space bounds for hidden line and hidden surface algorithms,* in Proc. EUROGRAPHICS '81, North-Holland, Amsterdam, 1981, pp. 43–56.

[38] Y. SHILOACH AND U. VISHKIN, *Finding the maximum, merging, and sorting in a parallel computation model,* J. Algorithms, 2 (1981), pp. 88–102.

[39] D. WOOD, *The contour problem for rectilinear polygons,* Inform. Process. Lett. 19 (1984), pp. 229–236.