Parallel Algorithms for Arrangements

Richard Anderson* Paul Beame*
Erik Brisson*
Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, Washington 98195

Abstract

We give the first efficient parallel algorithms for solving the arrangement problem. We give a deterministic algorithm for the CREW PRAM which runs in nearly optimal bounds of $O(\log n \log^* n)$ time and $n^2/\log n$ processors. We generalize this to obtain an $O(\log n \log^* n)$ time algorithm using $n^d/\log n$ processors for solving the problem in d dimensions. We also give a randomized algorithm for the EREW PRAM that constructs an arrangement of n lines on-line, in which each insertion is done in optimal $O(\log n)$ time using $n/\log n$ processors. Our algorithms develop new parallel data structures and new methods for traversing an arrangement.

1 Introduction

The problem of determining the geometric structure of the intersections of curves and surfaces has a long history in mathematics ([3], [5],[14]). For the purposes of computational geometry, a very important special case is that of determining this structure when the curves and surfaces being intersected are lines in $\mathbb{R}^2$ or, more generally, hyperplanes in $\mathbb{R}^d$ for $d \geq 2$. In this context the problem is known as the *arrangement* problem.

A simple and elegant sequential algorithm for computing arrangements in $\mathbb{R}^2$ was found by [7] and [10]; the latter also shows how the algorithm can be generalized to $\mathbb{R}^d$. In $\mathbb{R}^2$ this algorithm has worst case running time $O(n^2)$, which is obtained by inserting the lines one after another into the arrangement produced so far. In $\mathbb{R}^d$ the algorithm runs in $O(n^d)$ time. Since the problem generally requires that $\Omega(n^d)$ values be produced, the output requirements alone show that this is optimal.

Computing arrangements is an important building block in several computational geometry algorithms. In two dimensions, arrangements are used during a pre-processing step in algorithms for computing visibility graphs. They are also used by algorithms for finding shortest paths that avoid polygonal obstacles. Furthermore, the worst-case optimal hidden surface removal algorithm of McKenna [12] first projects the 3-dimensional problem (involving planes) onto a two-dimensional image plane, then computes the 2-dimensional arrangement produced in the image plane, and finally simplifies it to produce the viewed image.

There is a substantial body of work on the subject of parallel algorithms for computational geometry (e.g. [1], [4], [11], [13]). Included in this work are parallel algorithms for some problems related to finding arrangements, such as computing visibility from a point in 2 dimensions [4] and hidden surface removal in restricted 3-dimensional scenes [13]. However, finding an optimal parallel algorithm for computing arrangements has remained an open problem ([2], [11]).

A fairly straightforward parallel algorithm for computing arrangements can be constructed using $n^2/\log n$ processors, requiring $\Theta(\log^2 n)$ time. Goodrich in [11], which was the starting point for this research, gives an output-sensitive algorithm for computing the intersections of line segments; however when used to find arrangements of lines, its running time is no better than that of the straightforward algorithm.

*This work was supported by the National Science Foundation, under grants CCR-8657562 and CCR-8858799, NSF/DARPA under grant CCR-8907960, and Digital Equipment Corporation.

We present two algorithms for the arrangement problem. The first is a deterministic algorithm for the CREW PRAM which runs in near-optimal $O(\log n \log^* n)$ time using $O(n^2/\log n)$ processors for computing arrangements in $\mathbb{R}^2$. We also show how this generalizes to an $O(\log n \log^* n)$ time algorithm using $O(n^d/\log n)$ processors in $\mathbb{R}^d$. The second solves the on-line version of the arrangement problem, in which lines are only available as input one after another. It is a randomized algorithm for the EREW PRAM that constructs an arrangement of n lines on-line, so that each insertion is done in optimal $O(\log n)$ time using $n/\log n$ processors. Both of our algorithms develop new methods for traversing an arrangement efficiently in parallel.

Perhaps because of their perceived sequential nature, very little study has been made of parallel algorithms for on-line problems. However, efficient on-line parallel algorithms can be useful in a context where extremely fast response times are required in a dynamic environment. On-line problems place unique demands on parallel algorithms because, unlike static problems, they can require efficient maintenance of data structures significantly larger than the number of processors available. In our on-line algorithm for computing arrangements we encountered a problem requiring sophisticated data structures developed for sequential computation.

2 Background

2.1 Problem Statement

Given a set H of n hyperplanes in $\mathbb{R}^d$, where $d \geq 2$, their arrangement $A(H)$ is the subdivision of $\mathbb{R}^d$ they create. That is, if $H = \{h_1, \ldots, h_n\}$, and h_i^- and h_i^+ are the open half-spaces defined by h_i, then the *faces* of $A(H)$ are $\{\cap_{i=1}^n \tilde{h}_i \ : \ \tilde{h} = h_i^-, h_i, \text{ or } h_i^+\}$. A description of an arrangement must include an enumeration of the faces, along with their topological relationships, for instance an incidence graph. If the input hyperplanes are in general position, so that the intersection of any k hyperplanes is a $(d-k)$-dimensional face, then $A(H)$ is *simple*. The number of k-faces in a general arrangement is $O(n^d)$, and the number of k-faces in a simple arrangement is $\Theta(n^d)$. In the 2-dimensional case, the points, edges and regions of $A(H)$ will be denoted by $P(H)$, $E(H)$ and $R(H)$, respectively.

We will assume that the input set of hyperplanes forms a simple arrangement, and in the 2-dimensional case contains no horizontal or vertical lines. The latter assumption may be eliminated by making a small rotation of coordinates if the input includes horizontal or vertical lines.

For output we need to give a description of the arrangement. In the 2-dimensional case we will produce, for each line in H, a sorted list of its intersections with the other lines of H. The incidence graph may be produced within the same processor and time bounds. In higher dimensions, the incidence graph will be produced as output.

2.2 The Sequential Algorithm

In $\mathbb{R}^2$ the arrangement problem can be solved by brute force in $O(n^2 \log n)$ time by computing all the intersections along each line and then sorting these n lists independently. The optimal sequential algorithm for the arrangement problem in $\mathbb{R}^2$ given in [7] and [10] removes the $\log n$ factor, using an on-line algorithm which inserts each line ℓ into the existing arrangement of up to n lines in time $O(n)$, to achieve its running time.

For the purposes of illustration, view the line ℓ to be inserted as being horizontal. The leftmost intersection of ℓ with the arrangement is found and ℓ is inserted in the list of the line that it intersects. Then a left-to-right traversal of the arrangement is made along ℓ which discovers and adds each intersection point involving ℓ. Given any intersection point p on ℓ, let R be the region which ℓ intersects immediately to the right of p. The next intersection is found by traversing the portion of the boundary of R lying above ℓ by following the chain of edges incident to the boundary in clockwise order (this ordering is extended to infinite faces in the obvious way). Figure 1 gives an illustration of the traversal. Although it is not immediately obvious, it can be shown that such a traversal never encounters more than $3n$ segments along the way and thus the time for the insertion is $O(n)$.

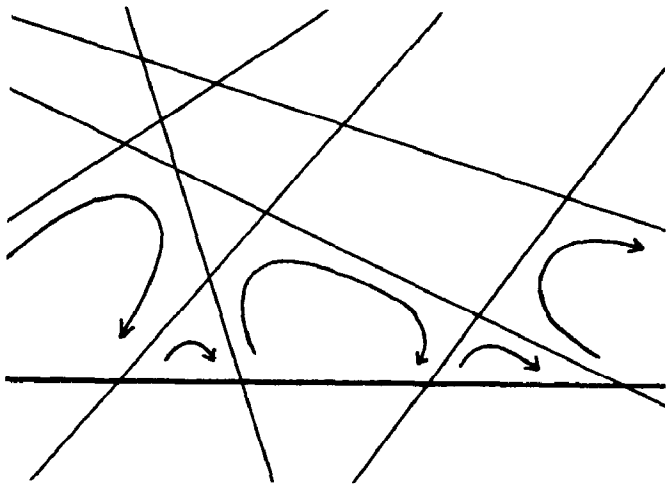

Figure 1: Traversal during sequential line insertion

3 Deterministic Algorithm for Arrangements

3.1 Overview

There are two key elements to the deterministic parallel algorithm. The first is the fast insertion of a single line into an arrangement, and the second is the fast merging of arrangements. The insert is a parallelization of the sequential insert, which requires doing the inserts in a particular order and maintaining two extra data structures, to allow an even distribution of processors and to speed up the traversal of region boundaries. The merge of a set of arrangements is done by simultaneously inserting every line into every arrangement other than its own, and then merging the results of these separate inserts independently for each line. Somewhat surprisingly, to obtain the most efficient algorithm, we must slow down the rate at which geometric information is produced, as the bottleneck in our algorithm is the standard merging of sorted lists.

The set of input lines will be denoted by H_{in}. Let $H \subseteq H_{\text{in}}$. To **insert** a line ℓ into H will mean to create a sorted list of the intersection points between ℓ and the lines in H (excluding ℓ, if $\ell \in H$). If every line in H has been inserted into H, then H taken with its lines' sorted lists will be called a **sub-arrangement** of $A(H_{\text{in}})$.

The algorithm is a divide-and-conquer algorithm which first performs a 'setup step' followed by $\log^* n$ 'phases'. (Figure 2 gives a visual presentation of the algorithm.) The setup step orders the input lines by their slopes, and then organizes them into $\log n$ groups of $n/\log n$ consecutive lines (taken in this slope ordering). Each line is then inserted into the group which contains it. Thus the setup step provides $\log n$ disjoint sub-arrangements, each of size $n/\log n$.

Each phase takes as input a partition of H_{in} into k disjoint sub-arrangements of size n/k (in this section, it will always be the case that $k \leq \log n$). A phase runs in three steps. The first step divides the input into groups of $k/\log k$ consecutive sub-arrangements (in the ordering of the lines they contain). It also computes two auxiliary data structures, 'splitters' and 'levels' (which will be defined later) for each of the input sub-arrangements.

Each line appears within exactly one sub-arrangement, and so appears in exactly one group, which we will call the line's group. The second step inserts every line into each of the sub-arrangements within the line's group. Thus $k/\log k$ sorted lists are created for every line.

In the third step the sorted lists for each line are merged into one sorted list. By creating this merged list, the line has now been inserted into its group. Thus the output is a partition of H_{in} into $\log k$ disjoint sub-arrangements of size $n/\log k$.

We will prove, in four lemmas, that the setup step and each of a merge phase's steps can be performed in time $O(\log n)$ using $n^2/\log n$ processors on a CREW PRAM, thus proving the main theorem of this section:

Theorem 3.1 *Given a set H of n lines in the plane, the deterministic algorithm outlined above constructs $A(H)$ in time $O(\log n \log^* n)$ using $n^2/\log n$ processors.*

3.2 Levels and Slope Ordering

The key to the insertion of a line into a sub-arrangement is the parallelization of the sequential traversal described earlier. This is accomplished by distributing the available processors evenly along the line being inserted, in particular by assigning a processor to every $\log n$'th intersection point, which can be done after making an observation about the levels of an arrangement: given a set of lines of 'consecutive' slope, the level structure of their arrangement is the same for all lines whose slope lies 'outside' of their set of slopes. The first step of a phase builds 'vertical' and 'horizontal' levels. Then

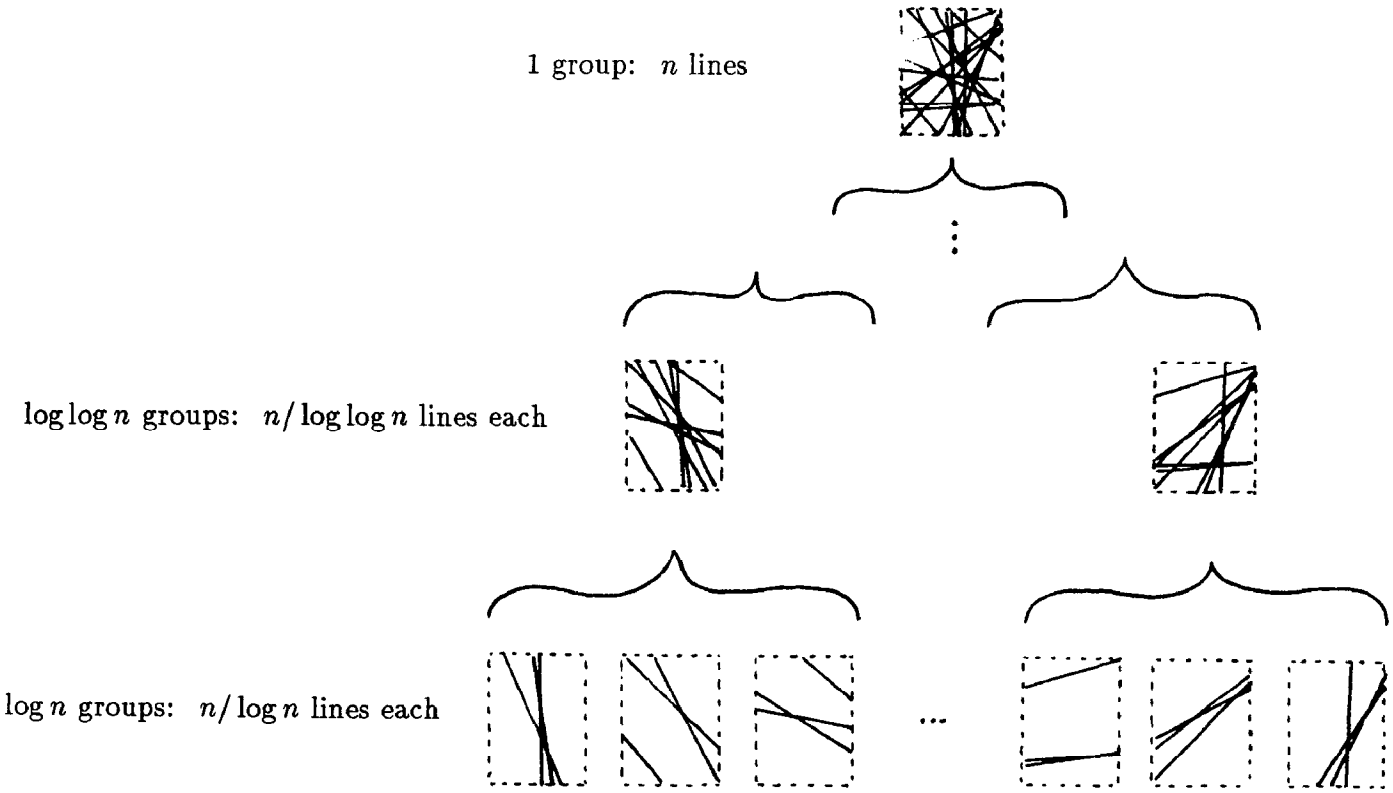

Figure 2: Overview of deterministic algorithm, in two dimensions

the intersection of a line with the $\log n$'th level, in one of these two directions, can be computed, which gives the $\log n$'th intersection point. This will be described in further detail below.

If e is an edge in $E(H)$, choose a vertical line through e which does not contain any points of $P(H)$. Define the **vertical level of e in $A(H)$** to be to be the number of edges of $E(H)$ this line intersects below e. It is easy to check that this is well-defined (in particular, is independent of the choice of vertical line), given the fact that there are no vertical lines in the input. The set of all edges in $E(H)$ whose level is k will be called the **k-level** of $A(H)$. Given any line ℓ, define the **intersection of ℓ with level k** to be the edge in level k which intersects ℓ. Horizontal levels are defined similarly. (See figure 3.)

Each input line ℓ of H_{in} has a slope $-\infty < m_\ell < \infty$. If H is a sub-arrangement, define $m_H^- = \min_{\ell \in H} m_\ell$ and $m_H^+ = \max_{\ell \in H} m_\ell$. Let ℓ be a line of H_{in} such that $m_\ell \notin [m_H^-, m_H^+]$. If $|m_\ell| > |m_H^-|$ we will say that ℓ is **vertically insertable** into H, and if $|m_\ell| < |m_H^+|$ we will say that ℓ is **horizontally insertable** into H.

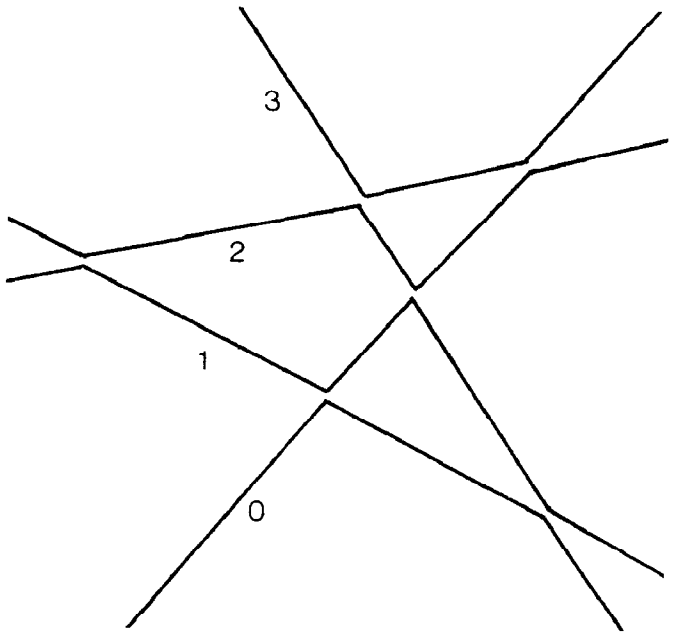

Figure 3: Example of vertical levels

Observation 3.2 *If ℓ is vertically insertable into H then, with increasing y, it intersects the vertical levels in strictly increasing order.*

Proof For any region of $A(H)$, classify its edges as **bottom** edges if they lie below it, and **top** edges if they lie above it. Observe that all bottom edges of a region are of the same level k, for some k, and all top edges of that region have the same level $k+1$. By the definition of insertability, if ℓ intersects a region, then it intersects the bottom of the region and the top of the region exactly once each. ∎

Observation 3.3 *Given an arrangement $A(H)$ of n lines, a data structure can be built in $O(\log n)$ time by $n^2/\log n$ processors, which allows finding the intersection of a vertically insertable line with any level of $A(H)$ in time $O(\log n)$ by a single processor.*

Proof An ordering is defined on all edges in the arrangement, first by level, then within levels from left to right. A binary tree is built, with the edges as leaves, in time $O(\log n)$ time using $n^2/\log n$ processors. ∎

The similar observations holds for horizontally insertable lines for increasing x and horizontal levels.

3.3 The Algorithm

3.3.1 Setup Step

Ordering the n input lines by the their slopes is done as a sort of n scalars in time $O(\log n)$ by n processors. Breaking the lines into groups of $n/\log n$ lines can be done in constant time by n processors. Each line must now be inserted into a group of $n/\log n$ lines. Assign $n/\log n$ processors to each line. For a specific line ℓ, each of its processors finds the intersection of ℓ with a different line in ℓ's group. To sort these intersection points is done as a sort of $n/\log n$ scalars by $n/\log n$ processors in time $O(\log n)$. This gives the following lemma:

Lemma 3.4 *The setup step can be done in time $O(\log n)$ using $n^2/\log n$ processors.*

3.3.2 Auxiliary Data Structures

We now define splitters, which will facilitate the fast traversal of large regions (those with many edges). The use of splitters first appears in [6]; it was also used in [11]. If e is an edge of $A(H)$, let R be the region below e. The **splitter for e in $A(H)$** is the rightmost edge among the bottom edges of R. (See figure 4.) If R has no bottom edges, then the splitter is undefined for the edges of R. This occurs only for the 'bottom-most' region, which will not be traversed.

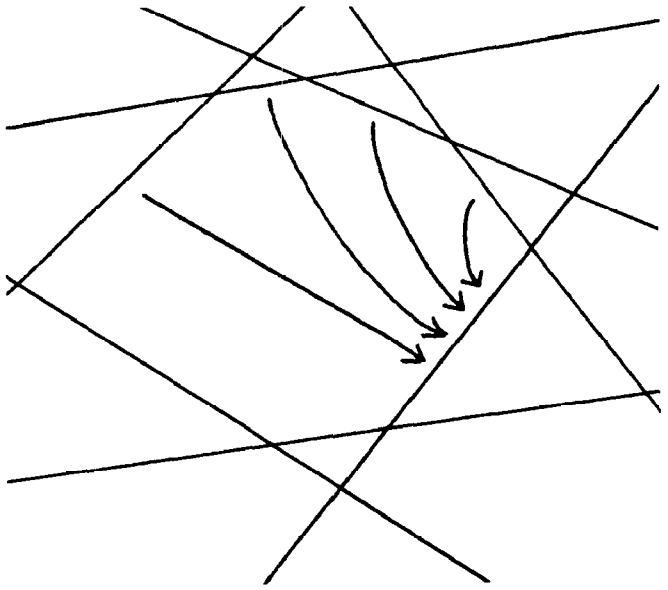

Figure 4: Pointers to (downward) vertical splitter

We will describe how to attach a pointer from every edge to its splitter, in a sub-arrangement of n/k lines, in time $O(\log n)$ using $n^2/(k\log n)$ processors. Note that such a sub-arrangement has $(n/k)^2$ edges, so there are $\log n/k$ edges per processor. To begin, every edge (except one) sets a pointer to its clockwise neighbor in the region below itself. The exception is the right-hand infinite edge of the bottom-most region, which points to itself. Thus there is at most one tree for every region, whose root is that region's splitter (except in the case of the bottom-most region). The goal is to have every edge point to the root of its tree. By using a variation of list ranking, we can set each edge's pointer to the root of its tree within the desired time bounds. Note that the edges of the bottom-most region will all be pointing at that region's right-hand infinite edge.

To calculate levels, observe that the level of any edge is one greater than that of its splitter, except in the case of edges of the bottom-most region, which all have level 0. The right-hand infinite edge of the bottom-most edge will be the **root edge**

for the purposes of this step. The starting configuration for making levels is just the result of the construction of splitters. These pointers are now labeled with 1, unless they point at the root edge, in which case they are labeled with 0. This gives a tree whose root is the root edge. Again using a variation of list-ranking, we can compute the cost of the path from each edge along these pointers to the root edge in the desired time bounds, which is exactly the level of the edge.

Lemma 3.5 *Given a sub-arrangement of n/k lines, where $k \leq \log n$, its splitters and levels can be produced in $O(\log n)$ time using $n^2/(k \log n)$ processors.*

3.3.3 Inserting a Line into a Sub-arrangement

As a result of the ordering done in the setup step and by merging of consecutive sub-arrangements, whenever we do an insert the line will be either vertically or horizontally insertable. A **vertical insert** or a **horizontal insert** will be done in each case, respectively. We will describe the vertical insert of a line ℓ into a sub-arrangement $A(H)$ of n/k lines, using $n/(k \log n)$ processors; the horizontal insert is similar.

A vertical insert is done in two passes, called **traversals**, the first downward and the second upward; we will describe the downward pass. A sub-arrangement of n/k lines has n/k levels. Assign a processor to every $\log n$'th level, and also to level n/k. Subscript the processors by successive positive integers in order of the levels to which they are assigned. Each processor P_i first finds the intersection e_i of ℓ with its level, and computes the intersection p_i of ℓ and the line containing e_i. Let R be the region below e_i. The processor now begins a clockwise traversal of the boundary of R. This cannot actually be done edge-by-edge, as it would take too long for large regions. Instead, the processor immediately jumps to its splitter, and then the clockwise search proceeds as it would in the sequential case. If this traversal reaches an edge e' which intersects ℓ, the process is started over, and so on. The processor stops when it reaches p_{i-1} or encounters an edge whose containing line intersects ℓ below p_{i-1}. (See figure 5.)

The upward pass is now done, traversing boundaries counterclockwise, using the appropriate re-

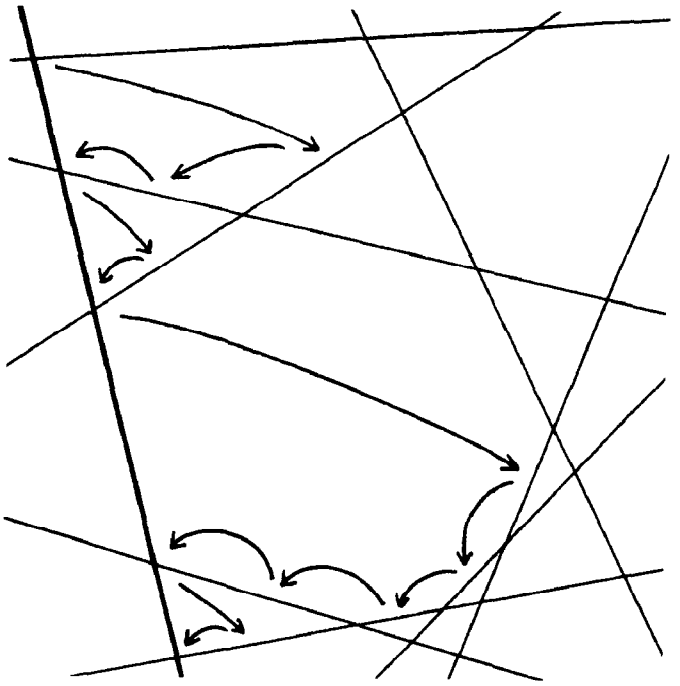

Figure 5: Example of downward traversal

definition of splitters, etc. What needs to be proven is that this takes time $O(\log n)$, and that each intersection point of ℓ with the lines of H is found by either the downward or upward pass. This is enough to give the sorted order of the intersection points.

Lemma 3.6 *The above algorithm inserts a line into a sub-arrangement of size n/k, in time $O(\log n)$ using $n/(k \log n)$ processors.*

Proof The argument mimics that of Goodrich [11]. ∎

3.3.4 Merging Sorted Lists

In the third step of each phase, every line must merge $k/\log k$ sorted lists using the processors assigned to it:

Lemma 3.7 *$k/\log k$ sorted lists of length n/k can be merged in time $O(\log n)$ using $n/\log n$ processors.*

Proof A balanced binary tree is formed with the lists at the leaves. The lists are merged in rounds, so that each round reduces the depth of the tree by one. Thus there are $\log k$ rounds, each requiring $n/\log k$ work, or total linear work. ∎

3.4 Generalization to Higher Dimensions

Given a set H of n hyperplanes in $\mathbb{R}^d$, the d-dimensional arrangement $A(H)$ is the subdivision of $\mathbb{R}^d$ generated by H. An optimal worst case sequential algorithm for constructing $A(H)$ is given in [10], which runs in $O(n^d)$ time. Our parallel algorithm for constructing arrangements in the plane may be used to give a near optimal algorithm for d-dimensional arrangements; it runs in time $O(\log n \log^* n)$, using $n^d/\log n$ processors (note that the constant in the time bound, in both the existing sequential algorithm and our parallel algorithm, depends on d).

We will only give a brief sketch of the algorithm here. Assign $n^{d-1}/\log n$ processors to each input hyperplane. Processing is done independently for each hyperplane h. First the intersection of h is taken with every other input plane. This gives a set H_h of n–1 (d–2)-dimensional hyperplanes within h. 'By induction', the set of processors assigned to h can produce the arrangement $A(H_h)$ in time $O(\log n \log^* n)$. The resulting arrangements can be merged in constant time per vertex.

Theorem 3.8 *Given a set H of n hyperplanes in $\mathbb{R}^d$, the algorithm sketched above constructs $A(H)$ deterministically in time $O(\log n \log^* n)$ using $n^d/\log n$ processors.*

Proof Proof by induction on dimension d. ∎

4 The Randomized On-line Algorithm

In this section we sketch an optimal randomized solution to the 2-dimensional on-line version of the problem. The on-line problem is to construct the arrangement by inserting n lines, one at a time, in the order that they are given. We give a randomized EREW PRAM algorithm which inserts each line in $O(\log n)$ time using $n/\log n$ processors. The algorithm always succeeds, and succeeds within the given time bound with high probability. Our result shows that it is possible to achieve full speed-up for inserting a line into an arrangement.

One reason we were interested in studying the problem is that it is on-line. On-line problems have not received much attention from people studying parallel algorithms. This is surprising, since on-line problems provide a domain where there can be tight time constraints which motivates using a high degree of parallelism.

An interesting aspect of our solution is that data structures play a very important role. In the majority of parallel algorithms that have been developed, it is not necessary to use sophisticated data structures; arrays and lists have usually been sufficient. However, for this problem we require complicated data structures. The study of on-line parallel algorithms has the potential to raise numerous interesting data structure problems.

In this paper we give only a high level sketch of the algorithm. It is a complicated algorithm and the proof that it achieves the desired run-time is involved. The algorithm attempts to mimic the sequential algorithm by performing a traversal of the line that is being inserted. As in the previous algorithm, we attempt to find a favorable distribution of starting points along the line being added, allowing independent sub-traversals. The hardest part of the algorithm is the load balancing that is done to make sure the traversals are all of roughly the same length.

4.1 Load Balancing

Suppose we with to insert the line ℓ into an arrangement $A(H)$ of n lines. We begin by selecting a random subarrangement of $A(H)$ that consists of $n/\log n$ lines. We can insert ℓ into the subarrangement in $O(\log n)$ time by a brute force method using sorting, giving a set of intersections along the line ℓ. We would like to perform a search starting from each of these intersections. There will be some variance in the lengths of these searches, arising both from the stochastic variation from picking the lines at random, and from the size of the faces that must be traversed. We look at the problem of doing these traversals as having a set of tasks $t_1, \ldots, t_m$, with $\sum_i t_i \leq cn$ for some constant c. A substantial number of these tasks may have run-time much greater than $\log n$. Our idea is to assign p processors to each task, and view each of these groups of p processors as a single 'pseudo-processor'. By showing that a task can be sped up by a factor of p by assigning p processors to it, we show that the run-time of each task can be reduced by a factor of p at the cost of reducing the number

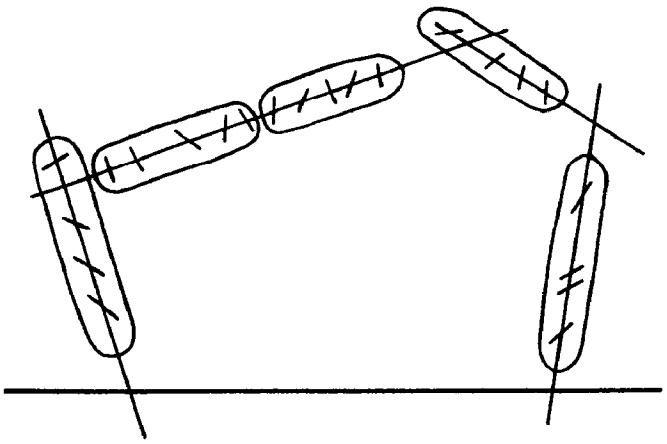

Figure 6: Example traversal in on-line algorithm

of available processors by a factor of p. Thus we have a set of smaller jobs to 'pack into a smaller number of bins' of the same size. We are able to perform some load balancing so as to achieve a better packing. Using the Cole-Vishkin [8] algorithm we are able to execute all of the tasks with run-time (on original processors) at most $p \log n$.

We now turn our attention to the problem of performing a task by a set of p processors. Each task consists of traversing a series of segments to detect the next intersection of ℓ with the arrangement. This process appears very sequential in nature, since at each intersection (point of $P(H)$ encountered in a traversal) we move to a different line. We modify the traversal to get one that we can speed up with a collection of processors. Instead of changing lines at each intersection, the set of processors looks at a collection of intersections along a line. One of these intersections is chosen, and the traversal proceeds along the corresponding line. If p processors look at intersections in collections of size p^2, it is easy to speed the search up by a factor of p. Figure 6 illustrates the type of traversal that the algorithm performs. An important fact that must be established is that the total length of the new traversal (summed over all tasks) is $O(n)$; this requires a complicated technical argument. We do not have the space in this paper to give this argument, or to go into the details of the traversal.

The modified traversal does not find all of the intersections with ℓ. However, we can detect where each line intersects the traversal. It follows from the definition of the traversal that with high probability the line $\hat{\ell}$ intersects the traversal very close to where $\hat{\ell}$ intersects ℓ. This allows us to quickly detect where these intersections occur. There are a number of other details that we are not addressing in this discussion. In particular, we cannot guarantee that we can finish all of the tasks, just most of them. It is then necessary to have a clean-up phase where we find the rest of the intersections.

4.2 Data Structures

One of the interesting aspects of this problem is that the data structures that arise are non-trivial. Data structures have not played a major role in the development of parallel algorithms; a review of parallel algorithms shows that in most cases lists and arrays have been sufficient. An explanation of why the data structures are more complicated in this problem is that it is an on-line problem, hence the number of processors $(n/\log n)$ is much smaller than the number of data items that must be kept track of $(\Omega(n^2))$.

The key to our data structure is to maintain for each line a *sorted* list of intersections. (Our data structure also maintains some geometric information, but this is not a source of difficulty.) For every insertion of a line, we must add one intersection to each list in $O(\log n)$ time using $n/\log n$ processors. If we only had to worry about the insertion, this would not be a difficult, since the on-line algorithm supplies us with an adjacent intersection to each intersection that we add. The difficulty is that we must be able to perform binary searches on these lists of intersections. The natural solution is to use some form of balanced tree to represent the list; however, that leads to $\Omega(\log n)$ worst case time for an insert. In addition to being able to perform binary search, we have additional requirements on the data structure: binary search between elements separated by distance k in the list needs to be done in time $O(\log k)$; and it must be possible to access j consecutive items in the list in time $O(j)$ for $j \approx \log^{1/2} n$.

The type of data structure that we need is a tree-like structure that supports constant *worst case* time insertion. The persistent data structures of

Driscoll et al. [9] can be modified to achieve the desired bounds. We have also developed a simpler data structure (since we do not require the full power of persistence, and since we can make some simplifying assumptions about the operations) that achieves the same performance. The data structure is a balanced binary tree except that its three bottom levels have degree $O(\log n)$ instead of two. These levels with higher degree allow us to restrict the amount of balancing that is performed during the insertion of each line, so as to stay within our resource bounds.

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. K. Yap. Parallel computational geometry. *Algorithmica*, 3:293–326, 1988.

[2] A. Aggarwal and J. Wein. Computational geometry: Lecture notes for 18.409, spring 1988. Technical report, MIT Laboratory for Computer Science, 1988. Technical Report MIT/LCS/RSS 3.

[3] D. Arnon, G. Collins, and S. McCallum. Cylindrical algebraic decomposition i and ii. *SIAM Journal on Computing*, 13(4):865–889, 1984.

[4] M. J. Atallah, R. Cole, and M. T. Goodrich. Cascading divide-and-conquer: A technique for designing parallel algorithms. In *28th Symposium on Foundations of Computer Science*, pages 151–160, 1987.

[5] J. Canny. A new algebraic method for robot motion planning and real geometry. In *28th Symposium on Foundations of Computer Science*, pages 29–38, 1987.

[6] B. Chazelle. Intersecting is easier than sorting. In *Proceedings of the 16th ACM Symposium on Theory of Computation*, pages 125–134, 1984.

[7] B. Chazelle, L. J. Guibas, and D. T. Lee. The power of geometric duality. In *24th Symposium on Foundations of Computer Science*, pages 217–225, 1983.

[8] R. Cole and U. Vishkin. Approximate scheduling, exact scheduling, and applications to parallel algorithms. In *27th Symposium on Foundations of Computer Science*, pages 478–491, 1986. Part I to appear in SIAM Journal of Computing.

[9] J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, February 1989.

[10] H. Edelsbrunner, J. O'Rourke, and R. Seidel. Constructing arrangements of lines and hyperplanes with applications. *SIAM Journal on Computing*, 15(2):341–363, 1986.

[11] M. T. Goodrich. Intersecting line segments in parallel with an output-sensitive number of processors. In *30th Symposium on Foundations of Computer Science*, pages 127–136, 1989.

[12] M. McKenna. Worst-case optimal hidden-surface removal. *ACM Transactions on Graphics*, 6(1):19–28, 1987.

[13] J. H. Reif and S. Sen. Polling: A new randomized sampling technique for computational geometry. In *Proceedings of the 21st ACM Symposium on Theory of Computation*, pages 394–404, 1989.

[14] A. Tarski. *A decision method for elementary algebra and geometry.* University of California Press, Berkeley, 1951.