Randomized Parallel Algorithms for Trapezoidal Diagrams

Kenneth L. Clarkson
AT&T Bell Labs

Richard Cole*
Courant Institute
New York University

Robert E. Tarjan†
Princeton University and
NEC Research Institute

Abstract

We describe randomized parallel algorithms for building trapezoidal diagrams of line segments in the plane. The algorithms are designed for a CRCW PRAM. For general segments, we give an algorithm requiring optimal $O(A + n\log n)$ expected work and optimal $O(\log n)$ time, where A is the number of intersecting pairs of segments. If the segments form a simple chain, we give an algorithm requiring optimal $O(n)$ expected work and $O(\log n \log\log n \log^* n)$ [1] expected time, and a simpler algorithm requiring $O(n \log^* n)$ expected work. The serial algorithm corresponding to the latter among the simplest known algorithms requiring $O(n\log^* n)$ expected operations. For a set of segments forming K chains, we give an algorithm requiring $O(A + n\log^* n + K\log n)$ expected work and $O(\log n \log\log n \log^* n)$ expected time. The parallel time bounds require the assumption that enough processors are available, with processor allocations every $\log n$ steps.

*Work was supported in part by NSF grants CCR-8902221 and CCR-8906949.

†Research at Princeton University partially supported by DIMACS (Center for Discrete Mathematics and Theoretical Computer Science), a National Science Foundation Science and Technology Center, Grant NSF-STC88-09648 and by the National Science Foundation, Grant No. CCR-8920505.

[1] $\log^* n$ is the least i such that $\log^{(i)} n \le 1$, where $\log^{(i)} n$ is the ith iterate of the logarithm: $\log^{(0)} n = n$, and $\log^{(i)} n = \log\log^{(i-1)} n$ for $i > 0$.

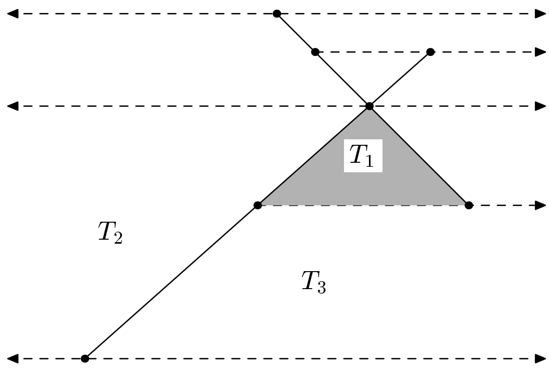

Figure 1: The trapezoidal diagram $\mathcal{T}(S)$ of two segments.

1 Introduction

1.1 Results and related work

We give several algorithms for building trapezoidal diagrams of line segments in the plane. These algorithms reach or approach optimality with respect to three parameters: the number n of segments, the number A of crossing pairs of segments, and the number K of *chains*. Here a chain is a sequence of segments such that a segment meets its successor in the sequence at an endpoint, and no endpoint is common to more than two segments. Specific values of these parameters correspond to problems in computational geometry that have been extensively studied. We consider these cases, as well as the general case, in the context of randomized parallel algorithms in the CRCW PRAM model.

Given a set S of n line segments, the trapezoidal diagram $\mathcal{T}(S)$ is a collection of simple regions defined as follows: for each endpoint of a segment in S, and each intersection point of segments in S, extend horizontal segments to the left and right as far as possible without crossing a segment of S. (Some of these segments will be rays, in fact; alternatively, construct the diagram inside a bounding rectangle.) These horizontal *visibility* edges, together with the edges of S, subdivide the plane into polygonal cells with four or fewer sides, that in general are trapezoids. (See Figure 1.1 for an example. The triangular cell T_1 is shaded, and the cells T_2 and T_3 are unbounded.)

Line arrangements. Computing a trapezoidal diagram can take $\Omega(n^2)$ work, since A can be $\Omega(n^2)$. Indeed, if S is a set of lines, which we can view as a special case, then $A = \binom{n}{2}$, and $\mathcal{T}(S)$ can be found in $O(n^2)$ work, both serially[EOS86] and in parallel. ([Goo91, HJW90];see also [AB90] for another deterministic algorithm.) The parallel algorithms require optimal $O(\log n)$ time, and while Goodrich's is deterministic, that of Hagerup *et al.* is randomized.

General line segments. If $K = \Omega(n)$, or the incidences between segment endpoints are unknown, we cannot hope to do better than $\Theta(n \log n)$, by reduction from sorting. If the input is not degenerate, so that no point meets more than two segments except at endpoints, the parameter A is also the output size, so $\Theta(A + n \log n)$ work is needed. This has been achieved in the serial case both deterministically [CE88] and with randomization [Mul88, CS89]. Recently Goodrich [Goo89a] has found a parallel algorithm requiring $O(A+n \log n) \log n$ operations and $O(\log n)$ time. If two sets of segments B and C are given, with the segments in each set non-intersecting, then an algorithm of Goodrich *et al.* [GSG89] can find the intersections between the two sets with $O(A + n \log n)$ work and $O(\log n)$ time, where n is the total number of segments and A is the number of intersections. Here we use these results to show that $O(A + n \log n)$ expected operations and $O(\log n)$ time suffice in general.

One simple chain. One special case of interest has $A = 0$ and $K = 1$: one simple chain. If this chain is a closed loop, it forms the boundary of a simple polygon, for which the problem of fast *triangulation* has attracted substantial effort. A triangulation of a simple polygon is a partition of its interior into triangles whose vertices are also vertices of the polygon. Most known algorithms for triangulation begin by computing a trapezoidal diagram of the polygon's boundary segments. Given the trapezoidal diagram, a triangulation can be found in $O(n)$ operations, both serially [CI84, FM84] and in $O(\log n)$ time in parallel [Goo89b].

Many problems involving a simple polygon can be solved in linear time if a triangulation of the polygon is known, so the complexity of triangulation is a particularly appealing question. Chazelle recently settled this question by finding a sophisticated algorithm requiring $O(n)$ time in the worst case [Cha90]. An earlier Las Vegas algorithm due to Clarkson, Tarjan, and Van Wyk requires $O(n \log^* n)$ expected time for any polygon [CTVW89]. Here we describe another Las Vegas algorithm requiring expected $O(n \log^* n)$ operations; the serial version is much simpler than the earlier Las Vegas algorithm, which is much simpler than Chazelle's. Seidel recently discovered a similar sequential algorithm [**?**]. A parallel version of our algorithm requires $O(\log n \log\log n \log^* n)$ expected time, with $O(n)$ operations if Chazelle's algorithm is invoked for subproblems, or $O(n \log^* n)$ operations if not.

Many non-simple chains. How fast can we compute the trapezoidal diagram of a *non-simple* chain of segments, that is, where $K = 1$ but A is not necessarily zero? (This problem was apparently first posed in [Tou88].) We generalize this question to allow a dependence on $K > 1$, and show that our techniques permit the construction of $\mathcal{T}(S)$ in $O(A + n \log^* n + K \log n)$ expected work and $O(\log n \log\log n \log^* n)$ expected time. This is optimal work with respect to A and K, and within $\log^* n$ of optimal with respect to n.

1.2 Outline of the paper

The remainder of the introduction has a subsection devoted to technical issues regarding the representation of the output, the nondegeneracy assumptions, and the model of parallel computation that we use. There follows a subsection giving an outline of a divide-and-conquer approach using random subsets, and lemmas for the analysis of algorithms using such an approach. Our serial algorithms are given in the next section; the parallel algorithms are given in §3.

1.3 Some technical issues

We compute representations of trapezoidal diagrams that include a data structure for navigating between adjacent cells; there is a technical issue here of what constitutes adjacency. We might say that two cells are adjacent when they share a visibility edge, and when no input segments cross, this definition is adequate. However, when input segments may cross, we must also be able to navigate between any two cells that share a common boundary. We will call a data structure recording adjacency under the former definition a *partial* adjacency representation, and the latter a *complete* adjacency representation. For example, in Figure 1.1, the cell T_1 is adjacent to only T_3 in the partial adjacency model, and to three cells including T_2 in the complete adjacency model. A complete adjacency representation might include for each cell an ordered list of cells incident to it.

The parameter A is the number of pairs of segments that cross: that is, whose nonempty intersection is not an endpoint of either segment. We assume in general that no point meets three segments except at endpoints; such a condition is easily simulated using a tie-breaking scheme.

We assume each chain is given as a list of consecutive line segments, in clockwise order, and that no two distinct endpoints have the same y coordinate; the latter nondegeneracy assumption is also easily avoided using a simple tie-breaking scheme: we compute the diagram using a coordinate system rotated clockwise infinitesimally, so that if points a and b are on a horizontal line with a to the left of b, we pretend that a is slightly higher than b.

Since each chain is readily transferred to an array with the edges in chain order, we assume henceforth that the chains are stored thus. (In the parallel case this transfer is done by a list-ranking algorithm [AM88]; it requires $O(m)$ operations and $O(\log m)$ time for m edges.)

We use the CRCW PRAM model for parallel computation, with random integers in the range 1 to n available at unit cost. We report the complexity as a pair (W, T), where W is the number of operations (or the work) done, and T is the parallel time achievable for that number of operations. This contrasts to the standard form of results, which reports the pair (P, T), where P is the number of processors used and T is the parallel time. Our description eases the analysis of algorithms where the number of processors used varies over the course of the algorithm. Of course, in an implementation, the number of processors used is fixed; phases that require more processors are handled by a round robin allocation of processors. It has to be verified that such an allocation of processors is feasible, but for our algorithms this presents no difficulty.

The number of processors (and operations) varies for two reasons: we do not know in advance the number A of intersections of the segments, and we don't know the results of some randomized divide-and-conquer steps. Our assumption is that every $\log n$ steps the algorithm is allowed to request an additional allocation of processors, which will then be provided in part or all by the system. As an example the usefulness of this assumption, in the randomized divide-and-conquer steps, we obtain a collection of subproblems with total expected size within appropriate bounds; we know the subproblem sizes, and allocate processors, before the subproblems are solved.

We assume further that the processors being used by the algorithm are always numbered consecutively. We make extensive use of the two algorithms mentioned earlier, due to Goodrich [Goo89a] and Goodrich, Shauck and Guha [GSG89]. While neither procedure is described in exactly this format it is not difficult to rewrite them for this model. Now we explain the meaning of our complexity results. The parallel time assumes that processor requests are met in full; the algorithm will be proportionately slower if fewer processors are provided; regardless of the number of processors provided the operation counts are as stated.

1.4 Probabilistic divide-and-conquer

Our algorithms use random subsets of S to divide and conquer, in outline as follows:

- Compute $\mathcal{T}(R)$ for random $R \subset S$ of size r, and possibly a point location data structure for it;
- insert the segments of S into $\mathcal{T}(R)$: that is, find the cells whose boundaries meet each segment, and so the set of segments S_T that meet each cell $T \in \mathcal{T}(R)$;
- for each cell $T \in \mathcal{T}(R)$, compute the subdiagram $T \cap \mathcal{T}(S_T)$;
- merge appropriate cells of the subdiagrams to build cells of $\mathcal{T}(S)$.

To show that this approach is effective, we use the following results.

Lemma 1 *Given a set S of n line segments with A intersecting pairs, let $R \subset S$ be a random subset of S of size r, with all subsets of size r equally likely. For cell $T \in \mathcal{T}(R)$, let n_T denote the number of line segments of S that meet the interior of T, and let N_T denote the number of cells adjacent to T (sharing a boundary point or edge). Then:*

(i) *The expected number of intersections of R is no more than Ar^2/n^2.*

(ii) *The expected value* $E\sum_{T\in\mathcal{T}(S)} n_T N_T = O(n + Ar/n)$.

(iii) *The expected value* $E\sum_{T\in\mathcal{T}(S)} n_T \log n_T = O((n + Ar/n)\log(n/r))$.

(iv) *With probability* $1 - 1/n$, $\max_{T\in\mathcal{T}(S)} n_T = O((\log n)n/r)$.

(v) *The expected value* $E\sum_{T\in\mathcal{T}(S)} n_T^2 = O(A + n^2/r)$.

Proof. The claim (i) is Lemma 4.1 of [CS89]. Claim (ii) can be proven analogously to Lemma 4.3 of that paper, and claim (iv) follows from Corollary 4.4 of [CS89]. Claim (iii) is a corollary of Theorem 3.6 of that paper, with $c = 2$ and $W(j) = \sqrt{j}\log j$ in the notation of the theorem, and claim (v) also follows from that theorem using $c = 2$ and $W(j) = j$. $\square$

2 Serial Algorithms

We start by describing an algorithm for segments that form a simple (non-self-intersecting) polygonal chain, then the case of a single polygonal chain that may have self-intersections, and finally the general case of several polygonal chains.

2.1 One simple chain

The algorithm for a simple polygonal chain is a modification of the randomized algorithm of [CTVW89]; this version also has a running time of $O(n\log^* n)$ time; its novelty is that it does not require Jordan sorting. (Given two simple curves in the plane, and their points of intersection in the order they appear on one curve, the problem of Jordan sorting is to determine the order in which they appear on the other curve. This operation can be performed in time linear in the number of intersection points, but known linear algorithms are complicated. [FNTW90, HMRT86])

Given a polygon P with set S of n edges, the algorithm builds $\mathcal{T}(S)$ in $\log^* n$ phases; in phase i, the diagram $\mathcal{T}(S^i)$ is found, where S^i is a random subset of S of size $r_i = \lfloor n/\lceil\log^{(i)} n\rceil\rfloor$. In fact we will use

$$S^1 \subset S^2 \subset \cdots \subset S^{\log^* n} = S,$$

with S^i a random subset of S^{i+1} of the given size; imagine the segments of S to be randomly permuted, and take S^i as the first r_i segments in the permuted order.

Each phase will require an expected $O(n)$ operations, as discussed below.

The diagram $\mathcal{T}(S^1)$ can be computed in $O(n)$ time, by various algorithms that require $O(m \log m)$ time for a set of m segments. [CE88, Mul88, CS89] This completes phase 1.

Each phase $i > 1$ uses the probabilistic divide-and-conquer scheme outlined in §1.4, where S^i takes on the role of S, and S^{i-1} the role of R. Recasting that outline, each phase $i > 1$ has three parts:

- insert the segments of S into $\mathcal{T}(S^{i-1})$; that is, find the cells that meet each segment, and so the set of segments S_T that meet each cell $T \in \mathcal{T}(S^{i-1})$.
- for each cell $T \in \mathcal{T}(S^{i-1})$, compute $T \cap \mathcal{T}(S^i_T)$, where $S^i_T = S_T \cap S^i$.
- merge appropriate cells of the diagrams $T \cap \mathcal{T}(S^i_T)$ to build cells of $\mathcal{T}(S^i)$.

It is in the first step of inserting the segments that we use the connectivity information: for this step, walk along the segments of P, and simultaneously through $\mathcal{T}(S^{i-1})$, finding for each segment in turn the cells of $\mathcal{T}(S^{i-1})$ that it meets. By the nondegeneracy assumption, no cell shares a bounding horizontal edge with more than four others, so $O(1)$ time is required for each cell that a segment meets.

Note that although we insert all of the segments of S, we only use the information for the segments in S^i: otherwise the computation of the subdiagrams $T \cap \mathcal{T}(S^i_T)$ would be too slow. To compute the subdiagrams, we again use an algorithm requiring $O(m \log m)$ time for a set of m segments.

The resulting visibility edges, over all the cells, together with the visibility edges for $\mathcal{T}(S^{i-1})$, form a superset of such edges for $\mathcal{T}(S^i)$. Some visibility edges of $\mathcal{T}(S^{i-1})$ are not in $\mathcal{T}(S^i)$, so the merging step of phase i is needed to obtain the cells of $\mathcal{T}(S^i)$. Note that a group of cells that we need to merge are in some collection of $T \in \mathcal{T}(S^{i-1})$ that are in order along a segment $e \in S^i$, and we know this order because it was obtained during the walk of the polygon and $\mathcal{T}(S^{i-1})$.

2.1.1 Analysis

As noted, each phase of the algorithm requires $O(n)$ expected time. For phase $i > 1$, the expected total time of the insertion step, which finds the sets S_T, is proportional to

$$E \sum_{T \in \mathcal{T}(S^{i-1})} |S_T| = O(n)$$

by Lemma 1(ii). Letting $n_T = |S^i_T|$ (so S^i plays the role of S in Lemma 1, and S^{i-1} the role of R), the expected time to compute $T \cap \mathcal{T}(S^i_T)$ for all $T \in \mathcal{T}(S^{i-1})$ is

$$E \sum_{T \in \mathcal{T}(S^{i-1})} n_T \log n_T = O(|S^i| \log \frac{|S^i|}{|S^{i-1}|})$$

$$
\begin{aligned}
&= O(\frac{n}{\log^{(i)} n} \log \frac{\log^{(i-1)} n}{\log^{(i)} n}) \\
&= O(n)
\end{aligned}
$$

by Lemma 1(iii). Finally, the merging step can be done in time proportional to the number of trapezoids in all the $\mathcal{T}(S_T^i)$, which is certainly an expected $O(n)$.

We conclude:

Theorem 2 *There is a randomized algorithm that triangulates a simple chain with n edges in expected $O(n \log^* n)$ time.*

2.2 One non-simple chain

The algorithm requires few changes when the chain has $A > 0$ pairs of intersecting segments, except to require the use of more sophisticated algorithms to compute $\mathcal{T}(S^1)$, the $\mathcal{T}(S_T^{i+1})$, and to do the walks. The "base" algorithm for trapezoidal diagrams must take $O(k + m \log m)$ expected time to compute the trapezoidal diagram of m segments with k intersecting pairs.

For the analysis, it suffices to bound the work due to intersecting pairs, since the remaining work is $O(n \log^* n)$ as in Theorem 2. By Lemma 1(i), the expected number of intersections among segments of S^1 is $O(A/\log^2 n)$, which bounds the expected work in computing $\mathcal{T}(S^1)$. Since the work in traversing a cell T adjacent to N_T other cells is proportional to N_T, the work in walking the chain to find the sets S_T is bounded, by Lemma 1(ii), by $O(A/\log n)$ (ignoring the term not dependent on A). In general, the work to compute the intersection points of S^i is expected $O(A/(\log^{(i)} n)^2)$, which is dominated by the time to traverse $\mathcal{T}(S^i)$ to find the sets S_T, which is $O(A/\log^{(i)} n)$. The expected work dependent on A is therefore

$$
O(A \cdot \sum_{1 \le i < \log^* n} \frac{1}{\log^{(i)} n}),
$$

which is $O(A)$.

2.3 Many chains

To complete this section, consider the general case: the input is a set of K chains comprising n segments, having A intersecting pairs. We modify the above algorithm only slightly: for each chain, choose one of its endpoints to be the *leader* of the chain. The algorithm will maintain the cells of $\mathcal{T}(S^i)$ containing these leaders, and use planar point location procedures to make this operation fast. With such leaders known, the walk through $\mathcal{T}(S^i)$ is as before, starting with the leader for each chain.

Recall that a planar subdivision defined by m non-intersecting straight edges can be preprocessed in $O(m \log m)$ time so that the region containing a given query point can be found in $O(\log m)$ time. (Several such algorithms are discussed in [Ede87].) We apply such a procedure to $\mathcal{T}(S^1)$, and find the cell containing each leader. To find the cell of $\mathcal{T}(S^{i+1})$ containing each leader, given that information for $\mathcal{T}(S^i)$, we preprocess each $\mathcal{T}(S_T^{i+1})$ for point location, locate each leader within the appropriate cell, and maintain that information when creating $\mathcal{T}(S^{i+1})$.

Theorem 3 *Given a set of S line segments forming K chains and with A intersecting pairs of segments, the trapezoidal diagram $\mathcal{T}(S)$ can be found in $O(A + n \log^* n + K \log n)$ expected time.*

Proof. Since the preprocessing for point location in the various trapezoidal diagrams takes no longer than the construction of those diagrams, it remains only to find the time needed for the queries.

Locating the K leaders within $\mathcal{T}(S^1)$ requires $O(K \log n)$. By Lemma 1(iv), for $i > 0$, the maximum size of a set S_T^{i+1}, over all cells $T \in \mathcal{T}(S^i)$, is

$$O(\log n \frac{|S^{i+1}|}{|S^i|}) = O(\log n \log^{(i)} n / \log^{(i+1)} n)$$

with probability $1 - \frac{1}{n}$. The query time for locating leaders within $\mathcal{T}(S^i)$ for $i > 1$ is thus $O(\log \log n)$. The cost of all queries after the first is therefore $O(K \log \log n \log^* n)$, which is dominated by the cost for locating within $\mathcal{T}(S^1)$. If some trapezoid $T \in \mathcal{T}(S^i)$ contains more than $\Theta(\log n \log^{(i)} n / \log^{(i+1)} n)$ edges, then the point location takes at most $O(\log n)$ time; but this happens with probability at most $\frac{1}{n}$, and so makes a contribution of $O(K \log n \log^* n/n)$ to the overall expected running time, a negligible term. □

3 Parallel Algorithms

We describe several parallel algorithms: first, one suitable for $A = \Omega(n^2)$, then for $K = \Omega(n)$ (or at least, any chain connectivity is unused). Following this is an algorithm for a single simple chain, then one for a single chain with intersections, and finally the general case. First we discuss the use of random subsets for divide-and-conquer, in the parallel setting.

3.1 The divide-and-conquer scheme

Each parallel algorithm described below generally has one or more divide-and-conquer phases, as outlined in §1.4.

For the first step of computing $\mathcal{T}(R)$, we generally apply some previous suboptimal algorithm. The insertion step is more problematic for parallel algorithms than in the serial case: we cannot simply walk along chains. In fact, single segments may meet $\Omega(r)$ trapezoids, even though an average segment meets no more than 6, and so walking through $\mathcal{T}(R)$ along some segments is too slow. This difficulty is the main problem for the rest of this paper.

In many variations of the insertion step of finding the segments S_T of S that meet each $T \in \mathcal{T}(R)$, we first obtain a collection of pairs, comprising segments of S and cells of $\mathcal{T}(R)$ that intersect. We do not have, for each cell $T \in \mathcal{T}(R)$, a list S_T of segments of S that meet it. To create such lists, the pairs are integer sorted, using their trapezoid as the key. Such sorting can be done with the algorithm of Rajasekaran and Reif [RR89] in $O(\log n)$ time and work linear in the number of sorted pairs.

As in the sequential algorithms, the merge step builds a trapezoid T of the final diagram from several trapezoids in the subdiagrams. This is done as follows. Implicitly, for each edge of S, the trapezoidal diagrams (for each cell $T \in \mathcal{T}(R)$) define a linked list of its intersections in order. By applying a list ranking algorithm to this list [AM88, CV86] we eliminate the intersections with visibility edges of $\mathcal{T}(R)$. This leaves the intersections among the edges of S together with the visibility edges of $\mathcal{T}(S)$; the computation of $\mathcal{T}(S)$ is completed by linking adjacent edges. The complete adjacency representation, which by Lemma 1(ii) is of size $O(n+A)$, is obtained by a further application of list ranking to the edges of S so as to determine for each trapezoid the number of its neighbors; a proportionate number of processors is then allocated to each trapezoid; these processors are used to create the adjacency lists, each processor being responsible for copying the name of one trapezoid to the adjacency list. Overall, this takes $O(n + A)$ operations and $O(\log n)$ time.

3.2 General line segments

This subsection gives an algorithm supporting the following theorem.

Theorem 4 *Given a set S of n line segments in the plane with A intersecting pairs, the trapezoidal diagram $\mathcal{T}(S)$ can be found in $O(A + n \log n)$ expected operations and in $O(\log n)$ worst-case time on a CRCW PRAM.*

The algorithm uses the divide-and-conquer approach described above: first, find an estimate $\hat{A}$ of A. Then take a random subset $R \subset S$ of size $r = n^2/(\hat{A}+n \log n)$. Compute $\mathcal{T}(R)$ and process it for point location using previous algorithms [Goo89a, ACG89]; then for each cell $T \in \mathcal{T}(R)$, find the sets S_T of segments that meet it, and compute visibility information among the segments of S_T using an adaptation of the algorithm of Hagerup *et al.* [HJW90]. Finally, merge the resulting cells to form $\mathcal{T}(S)$.

The idea is to balance between these known algorithms, building the relatively small $\mathcal{T}(R)$ to give subproblems with many intersecting pairs relative to the number of segments, so that an algorithm like that of [HJW90] is economical.

Here is the algorithm in more detail: the estimate $\hat{A}$ is found by taking n random pairs of segments and counting the number of intersecting pairs among them. Each pair (a_i, b_i) is chosen from among the $\binom{n}{2}$ pairs of segments of S, with all pairs equally likely. If this count is C, take $\hat{A} = (C+1)\binom{n}{2}/n$. (Other methods for estimating A might well be acceptable, but this is easy to analyze.) In particular, we note that

Lemma 5 *The expected value $E\hat{A} = A + \binom{n}{2}/n$, and $E[1/\hat{A}] \leq 1/A$.*

Proof. The quantity C is a binomial random variable with n trials and success rate per trial $p = A/\binom{n}{2}$. We have

$$E[1/(C+1)] = \sum_{j\geq 0} \frac{1}{j+1}\binom{n}{j} p^j (1-p)^{n-j},$$

and since $\binom{n}{j}(n+1)/(j+1) = \binom{n+1}{j+1}$, we have

$$E[1/(C+1)] \leq \frac{1}{p(n+1)} \sum_{j\geq 0} \binom{n+1}{j+1} p^{j+1} (1-p)^{n-j},$$

which is no more than $1/pn$, or $\binom{n}{2}/An$, and so $E[1/\hat{A}] \leq 1/A$. $\square$

Insertion. The insertion step of finding the sets S_T for cells $T \in \mathcal{T}(R)$ is done as follows: process $\mathcal{T}(R)$ for planar point location [ACG89], locate the endpoints of $S \setminus R$ within $\mathcal{T}(R)$, and then in parallel, walk along each segment through the diagram, finding cells that it meets. This is done until all but $n/\log n$ segments are completely traversed: do the walk $\log n$ steps, check the number of segments not yet completed, walk along $\log n$ steps, check again, and so on, until the condition holds.

There remain at most $n/\log n$ segments not completely inserted, that is, not completely traversed. To insert these, we split them, at random, into $\log n$ groups each of size at most $n/\log^2 n$. The algorithm of [Goo89a] is applied to each group to find all its intersection points, and to split the segments of the group up into segments that meet only at endpoints. For each group, we find the intersections of the resulting segments with the visibility edges of $\mathcal{T}(R)$ using the algorithm of [GSG89].

We now have the pairs of segments of S and trapezoids of $\mathcal{T}(R)$ that meet; as discussed in §3.1, we now use integer sorting to obtain the list S_T of segments that meet a trapezoid T, for every $T \in \mathcal{T}(R)$.

Subdiagrams. Next, for each cell $T \in \mathcal{T}(R)$, we compute the subdiagrams $T \cap \mathcal{T}(S_T)$ using an adaption of the algorithm of [HJW90]. Our adaption takes $O(n^2)$ expected operations to compute the trapezoidal diagram of a set of n line segments, and requires $O(\log n)$ time. While their algorithm finds triangulations of arrangements of lines, we are interested in trapezoidal diagrams of line segments. Nonetheless, their algorithm can be adapted to our purpose; Appendix H gives the details.

The merge step proceeds as discussed in §3.1.

Proof of Theorem 4. The proof of Theorem 4 is completed by analyzing the above algorithm. By Lemma 1(ii), the expected work in walking along the segments through $\mathcal{T}(R)$ is $O(n + An/(\hat{A} + n \log n))$, excluding busy waiting by processors handling segments whose traversal is complete. But $E[A/\hat{A}] \leq 1$, so the expected work is $O(n)$. This implies that after some $c \log n$ steps, for a large enough constant c, at most $n/\log n$ segments are not completely traversed.

The cost of inserting these remaining segments is bounded as follows. The algorithm of [Goo89b] requires an expected $O(n \log n + A)$ operations and $O(\log n)$ time (for by Lemma 1(i), there are an expected $O(A/\log n)$ intersections overall present in these randomly selected subsets), while the algorithm of [GSG89] requires an expected $O(n \log n + A)$ operations and $O(\log n)$ time (the expectation arises because there are an expected $O(n + A/\log n)$ edges present over all the subproblems).

The expected number of operations for all executions of the algorithm of Appendix H is $O(\sum_{T \in \mathcal{T}(R)} n_T^2)$; by Lemma 1(v), this has expected value $O(A + n^2/r)$ with respect to the choice of R, and the expected value of n^2/r with respect to the estimation of A is $O(A + n \log n)$, using Lemma 5. The running time of this step is $O(\log n)$.

The complexity of the final step (merging trapezoids) is bounded as follows. By Lemma 1(ii), there are an expected $O(n + Ar/n) = O(n)$ intersection points along the edges of S (including intersections with the visibility edges of $\mathcal{T}(R)$). So the complexity of the prefix lists algorithm is an expected $O(n)$ operations and $O(\log n)$ time.

On summing the complexities of each step, the theorem follows readily. □

3.3 One simple chain

This section gives our parallel algorithm for the case of a simple polygon (or just simple chain). The algorithm proceeds in the same $\log^* n$ phases as the sequential algorithm; each phase uses the randomized divide-and-conquer scheme, with $\mathcal{T}(S^{i-1})$ used to build $\mathcal{T}(S^i)$ in phase $i > 1$.

The diagram $\mathcal{T}(S^1)$ is built in phase 1 using the algorithm of Atallah, Cole and Goodrich [ACG89]; it does $O(n)$ operations in time $O(\log n)$. Also, we create a planar point location structure for $\mathcal{T}(S^1)$, again using an algorithm of Atallah *et al.*

First we consider phase 2, where we build $\mathcal{T}(S^2)$ in parallel using an expected $O(n)$ operations and $O(\log n \log \log n)$ time.

Insertion. The insertion of edges for phase 2 is done by traversal of $\mathcal{T}(S^1)$ and the edges of the chain, as in the sequential algorithm. However, we do many such traversals independently from several places.

Initially, the chain is partitioned into subchains of length $\log n$. This takes $O(n)$ operations and $O(\log n)$ time. There are then at most $2n/\log n$ subchains. We use $n/(\log n \log \log n)$ processors. We proceed in stages, continuing until at most $n/\log n$ edges have yet to be completely inserted. A stage lasts $\log n$

steps. Each step comprises the traversal of an edge to its next intersection point or to its far endpoint, whichever is nearer. At the end of a stage, if fewer than $\frac{1}{2}n/(\log n \log\log n)$ chains remain, the chains are evenly partitioned to obtain between $\frac{1}{2}n/(\log n \log\log n)$ and $n/(\log n \log\log n)$ chains anew (since the chains are contiguous sequences of edges of the polygon, storing these edges in clockwise order in an array makes the partitioning straightforward). The new endpoints are located using the point location data structure, at a cost of $O(\log n)$ operations per endpoint. So a stage does $O(n/\log\log n)$ operations. It either detects at least $\frac{1}{2}n/\log\log n$ intersection points and endpoints, or it halves the size of each remaining chain (or possibly both). By Lemma 1, there are an expected $O(n)$ intersection points, so after an expected $O(\log\log n)$ stages, each chain comprises at most one edge, so there are at most $n/\log n$ edges which have not been completely inserted; that is, the procedure terminates within an expected $O(\log\log n)$ stages.

To insert these remaining *bad* edges, we find the intersections between them and the $2n/\log n$ visibility edges. The intersections among these segments are computed using the algorithm of Goodrich *et al.* [GSG89]; for our application, it requires an expected $O(n)$ operations and $O(\log n)$ time (the expectation arises because the number of intersections is an expected $O(n)$).

As discussed in §3.1, we now use integer sorting to collect the sets S_T for each $T \in \mathcal{T}(S^1)$, using the algorithm of [RR89].

Subdiagrams. Next, for each $T \in \mathcal{T}(S^1)$, we find $T \cap \mathcal{T}(S^2_T)$, and also build a point location data structure for it, with the procedures of Atallah *et al.* [ACG89]. (The point location structure will be used for later processing.) We use $|S^2_T|$ processors for this task.

Merging. Finally, we need to merge cells from among those of the subdiagrams to find those of $\mathcal{T}(S^2)$. To carry out this process, for each edge in S^2, it is useful to have a linked list of its intersection points with the visibility edges of $\mathcal{T}(S^1)$. Additionally, in phase 3, it will be useful, for each edge in S, to have in sorted order its intersections with the visibility edges of $\mathcal{T}(S^1)$. So next we compute for each edge in S a list of its intersections with $\mathcal{T}(S^1)$, in sorted order along the edge.

It suffices to sort the intersection points for the $O(n/\log n)$ bad edges, since the other intersection points are already in order along the edges on which they lie. First, we randomly select $n/\log n$ of these intersection points. For each bad edge, we sort the selected points along the edge and split the edge into edge portions accordingly (this requires two sorts); sorting takes $O(n)$ operations and $O(\log n)$ time [Col88]. Next, we traverse the edge portions, seeking intersection points; each edge portion is traversed until either $\frac{2}{\alpha}\log n \log\log n$ intersection points are found, or the portion is completely traversed, whichever occurs sooner; α is a constant to be specified. The edge portions are evenly redistributed among the $n/(\log n \log\log n)$ processors every $\log n$ traversal steps, called a stage. For any stage, either at least $\frac{1}{2}n/\log\log n$ intersection points are traversed, or the processing of at least one half of the remaining edge portions is completed. Now we analyze this traversal: the expected number of edge portions containing more than $c \log n \log\log n$ intersection points is $O(n/(\log n)^{1+\Theta(c)}) = O(n/(\log n)^{1+\alpha c})$, say. It follows that the edge portions containing more than $\frac{2}{\alpha}\log n \log\log n$ intersection points between them contain an expected $O(n/\log n)$ intersection points. So the procedure lasts an expected $O(\log\log n)$ stages, which is $O(\log n \log\log n)$ time; an expected $O(n)$ operations are done. In addition, there now remain an expected $O(n/\log n)$ untraversed intersection points. These points have already been determined in the previous paragraph; they can now be sorted with respect to the bad edges on which they lie and then merged with the intersection points already on these edges. Overall, sorting the intersection points takes an expected $O(n)$ operations and expected $O(\log n \log\log n)$ time.

With this information, the merging step can be done as discussed in §3.1.

The algorithm for finding the diagrams in phases $i > 2$ is analogous. Before we describe this step, here is analysis of resource bounds for building $\mathcal{T}(S^2)$: as noted, the insertion step requires $O(\log n \log\log n)$ expected time and $O(n)$ expected operations, and building the lists S_T requires no more resources. The construction of the diagrams $T \cap \mathcal{T}(S^1_T)$ for all $T \in \mathcal{T}(S^1)$ requires expected

$$O(\sum_{T\in\mathcal{T}(S^1)} |S^2_T| \log |S^2_T|) = O(|S^2| \log \frac{|S^2|}{|S^1|}) = O(n)$$

operations, and $O(\log n)$ time. Sorting the intersection points, as already observed, requires an expected $O(n)$ operations and $O(\log n \log\log n)$ time. The list ranking algorithms require $O(m)$ operations and $O(\log m)$ time for a list of length m, so the expected number of operations to merge cells is only

$$O(\sum_{T\in\mathcal{T}(S^1)} |S^2_T|) = O(|S^2|) = O(n/\log\log n).$$

We turn to the procedure for building $\mathcal{T}(S^i)$ given $\mathcal{T}(S^{i-1})$, for $i > 2$. The first step is to insert the edges of the polygon into $\mathcal{T}(S^{i-1})$. The edges are partitioned at the points at which they cross visibility edges of $\mathcal{T}(S^{i-2})$; this increases the number of edges to an expected $O(n)$. The same edge insertion method is used as for the case $i = 2$. Following the partitioning of chains into subchains of length $\log n$ there are an expected $O(n/\log^{(i-2)} n)$ chains at hand. As before, we proceed in stages until at most $n/\log n$ edges have not been completely inserted. The only detail of the insertion that is novel is that for each endpoint of each edge, the cell $T \in \mathcal{T}(S^{i-2})$ containing the endpoint is known; thus a point location on an endpoint requires a search in the point location data structure for T. Note that only $O(n/\log n)$ point locations are done, as before. The insertion of the $n/\log n$ bad edges proceeds essentially as before, except that a separate subproblem is created for each cell of $\mathcal{T}(S^{i-2})$. However, the processor allocation and reallocation is done globally across all these subproblems; this allows for variations in the number of intersections found in the various subproblems and allows expectations to be with respect to the sum of the sizes of the subproblems. Then, for each cell $T \in \mathcal{T}(S^{i-1})$, we obtain a list of the segments S that intersect it, as before.

Next, for each cell $T \in \mathcal{T}(S^{i-1})$, we find $T \cap \mathcal{T}(S^i_T)$ and also build a point location data structure for it, using the procedure of Atallah *et al.* [ACG89]. We use $|S^i_T|$ processors for this task.

Finally, the cells are merged and for each edge the order of its intersection points with $\mathcal{T}(S^{i-1})$ is computed as before.

The analysis is similar to that for the case $i = 2$. Again, the insertion step requires $O(\log n \log\log n)$ expected time and $O(n)$ expected operations. The only difficult part is to analyze the cost of inserting the bad edges. Let n'_T denote the number of bad edges intersecting cell $T \in \mathcal{T}(S^{i-2})$. Then the operation count for inserting the bad edges, using the algorithm of Goodrich *et al.* [GSG89], is

$$\begin{aligned} &O(\sum_{T\in\mathcal{T}^{i-2}} (|S^{i-1}_T| + n'_T)\log(|S^{i-1}_T| + n'_T)) \\ &= \quad O(|S^{i-1}|\log\frac{|S^{i-1}|}{|S^{i-2}|} + \sum_{T\in\mathcal{T}^{i-2}} n'_T \log n'_T) \end{aligned}$$

This is an expected $O(n)$, since $\sum_{T\in\mathcal{T}(S^{i-2})} n'_T = O(n/\log n)$. Building the lists S_T requires an expected $O(n)$ operations and $O(\log n)$ time. The construction of the diagrams $\mathcal{T}(S^i_T)$ requires expected

$$O(\sum_{T\in\mathcal{T}(S^{i-1})} |S^i_T|\log|S^i_T|) = O(|S^i|)\log\frac{|S^i|}{|S^{i-1}|} = O(n)$$

operations and $O(\log n)$ time. Sorting the intersection points, as before, requires an expected $O(n)$ operations and $O(\log n \log\log n)$ time. The expected number of operations for merging cells is only

$$O(\sum_{T\in\mathcal{T}(S^{i-1})} |S^i_T|) = O(|S^i|) = O(n/\log^{(i)} n).$$

Summing over all phases, we see that the overall expected number of operations is $O(n\log^* n)$ and the expected time required is $O(\log n \log\log n \log^* n)$. This constructs the horizontal visibility structure. It remains to obtain the triangulation from the visibility structure, but this can be done in $O(\log n)$ time and $O(n)$ operations ([Goo89b]). We have shown

Theorem 6 *A simple polygon can be triangulated by a parallel randomized CRCW PRAM algorithm requiring expected $O(\log n \log\log n \log^* n)$ time and with expected $O(n\log^* n)$ operations.*

A more complex algorithm which does an expected $O(n)$ operations in expected $O(\log n \log\log n)$ time can be obtained; it uses Chazelle's linear-time triangulation algorithm as a subroutine.

Here, $\mathcal{T}(S^2)$ is constructed as before and the edges of the polygon are inserted into $\mathcal{T}(S^2)$. By Lemma 1(iv), with probability at least $1-\frac{1}{n}$, each cell $T \in \mathcal{T}(S^2)$ contains at most $\log n \log\log n$ edges. Using the method of Clarkson *et al.* [CTVW89] the edges crossing each visibility segment of $\mathcal{T}(S^2)$ are Jordan sorted by a sequential algorithm (this can be done using Chazelle's observation that Jordan sorting can be reduced in linear time to triangulation,[Cha90] and applying Chazelle's linear-time triangulation algorithm); this gives a set of visibility subproblems which are solved sequentially by applying Chazelle's linear-time algorithm. Since each subproblem is of size $O(\log n \log\log n)$ with overall probability $1-\frac{1}{n}$, this takes an expected $O(n)$ operations and expected $O(\log n \log\log n)$ time.

Theorem 7 *A simple polygon can be triangulated by a parallel randomized CRCW PRAM algorithm requiring expected $O(\log n \log\log n)$ time and with expected $O(n)$ operations.*

3.4 One non-simple chain

The algorithm for a single chain with $A > 0$ intersections is similar to that for a chain with no intersections.

We begin by computing an estimate $\hat{A}$ for A as in the algorithm of §3.2. If $\hat{A} \geq n\log n$, then we use the algorithm given in Theorem 4 to compute the trapezoidal diagram induced by the chain.

Otherwise, we define sets $S^1, S^2, \cdots, S^{\log^* n}$ as in the algorithm for a simple chain. We begin by determining $\mathcal{T}(S^1)$, but now we use the randomized algorithm given in §3.2. By Lemma 1(i), the expected number of intersections among segments of S^1 is $O(A/\log^2 n)$, so the randomized algorithm of §3.2 requires an expected $O(A/\log^2 n + n)$ operations and $O(\log n)$ time. Then we insert the edges of the chain in $\mathcal{T}(S^1)$ as in §3.3. The one change to the traversal, as in the sequential case, is that when and if the traversal crosses a segment of S^1 the work required is proportional to the number of cells adjacent to this edge of the cell. So to analyze the traversal, we count both the number of intersections found and the number of cells considered in crossing segments of S^1; by Lemma 1(ii), this totals an expected $O(n)$, and so after an expected $O(\log\log n)$ iterations, at most $n/\log n$ edges are not fully inserted. A new method is needed to find the intersection points involving these bad edges, for it is not guaranteed that the bad edges do not intersect. Instead of using the algorithm of Goodrich *et al.* [GSG89], the randomized parallel algorithm given in §3.2 is used. To apply it, as in §3.2, the $n/\log n$ bad segments are randomly partitioned into $\log n$ groups of $n/\log^2 n$ segments. By Lemma 1(i), each group of segments has an expected $O(A/\log^2 n)$ intersections. The algorithm of §3.2

is applied separately to each group of segments with the edges of $\mathcal{T}(S^1)$. Over all $\log n$ groups the algorithm requires an expected $O(n + A/\log n)$ operations and $O(\log n)$ time (for there are an expected $O(n+A/\log n)$ intersection points over all $\log n$ subproblems).

The sorting of intersection points is done as in §3.3.

For each cell $T \in \mathcal{T}(S^1)$ we obtain a list of the segments intersecting T as in §3.1. Overall this first phase requires an expected $O(n + A/\log n)$ operations and expected $O(\log n \log\log n)$ time.

The algorithm proceeds as in the simple chain case, as modified four paragraphs above, until $n/\log^{(j+1)} n = |S^{j+1}| > n^2/(n + \hat{A})$ (i.e. $n \log^{(j)} n \geq n + \hat{A} > n \log^{(j+1)} n$). So for each cell $T \in \mathcal{T}(S^j)$ the set S_T of segments intersecting T has been computed. For if we tried to proceed further, the insertion of the chain in $\mathcal{T}(S^{j+1})$ would be too expensive. Then the randomized algorithm of §3.2 is applied to each set S_T. The expected work for this final phase is $O(\sum_{T\in\mathcal{T}(S^j)} n_T \log n_T + A)$ which by Lemma 1(iii) is $O(n + |S^j|A/n)\log(n/|S^j|) = O(n\log^{(j+1)} n) = O(n+A)$; the running time is $O(\log n)$.

Phases 1 through j each have an expected operation count of $O(n+A/\log n)$ and an expected running time of $O(\log n \log\log n)$.

Overall, we obtain

Theorem 8 *There is a randomized CRCW PRAM algorithm for computing the trapezoidal diagram of a chain with A intersections which runs in expected $O(\log n \log\log n \beta(A, n))$ time and does an expected $O(A + n\beta(A, n))$ operations, where $\beta(A, n) = \min j$ such that $A \geq n \log^{(j)} n$, for $A \geq n$, and $\beta(A, n) = \log^* n$ for $A < n$.*

3.5 Many chains

As in the serial case, consider a set of n segments comprising K chains, having A intersecting pairs. For each chain, choose one of its endpoints to be the *leader* of the chain. The algorithm maintains the cells of $\mathcal{T}(S^i)$ containing these leaders, using planar point location procedures to make this operation fast. The walk through $\mathcal{T}(S^i)$ proceeds as before, starting at the leader of each chain. Since a planar point location data structure has already been computed for $\mathcal{T}(S^1)$, and for $\mathcal{T}(S^{i+1})$ with respect to each trapezoid of $\mathcal{T}(S^i)$, the additional expected work for doing the point locations on the leaders is just $O(K \log n)$, as in §2.3, and takes expected time $O(\log n)$. (As in §2.3, with probability at most $\frac{1}{n}$, a point location takes time $O(\log n)$, and otherwise takes time $O(\log\log n)$ for $i > 1$; overall, this is an expected $O(\log n)$.) So we obtain:

Theorem 9 *There is a randomized CRCW PRAM algorithm for computing the trapezoidal diagram of K chains with A intersections which runs in expected $O(\log n \log\log n \beta(A, n))$ time and does an expected $O(A + n\beta(A, n) + K \log n)$*

operations, where $\beta(A,n) = \min j$ *such that* $A \geq n \log^{(j)} n$*, for* $A \geq n$*, and* $\beta(A,n) = \log^* n$ *for* $A < n$*.*

References

[AB90] R. Anderson and E. Brisson. Parallel algorithms for arrangements. In *Proc. Second Symposium on Parallel Algorithms and Architectures*, pages 298–306, 1990.

[ACG89] M. Atallah, R. Cole, and M. Goodrich. Cascading divide and conquer: a technique for designing parallel algorithms. *SIAM Journal on Computing*, 3:499–532, 1989.

[AM88] R. Anderson and G. Miller. Deterministic parallel list ranking. In *Proceedings of the Third Aegean Workshop on Computing*, volume 319 of *Lecture Notes in Computer Science*, pages 81–90. Springer-Verlag, 1988.

[CE88] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. In *Proc. 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 590–600, 1988.

[Cha90] B. Chazelle. Triangulating a simple polygon in linear time. In *Proc. 31th Annual IEEE Symposium on Foundations of Computer Science*, pages 220–230, 1990.

[CI84] B. Chazelle and J. Incerpi. Triangulation and shape complexity. *ACM Transactions on Graphics*, pages 135–152, 1984.

[Col88] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 4:770–785, 1988.

[CS89] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete and Computational Geometry*, 4:387–421, 1989.

[CTVW89] K. L. Clarkson, R. E. Tarjan, and C. J. Van Wyk. A fast Las Vegas algorithm for triangulating a simple polygon. *Discrete and Computational Geometry*, 4:423–432, 1989.

[CV86] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal list ranking. *Information and Control*, 1:32–53, 1986.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, New York, 1987.

[EOS86] H. Edelsbrunner, J. O'Rourke, and R. Seidel. Constructing arrangements of lines and hyperplanes with applications. *SIAM Journal on Computing*, 15:341–363, 1986.

[FM84] A. Fournier and D. Y. Montuno. Triangulating simple polygons and equivalent problems. *ACM Transactions on Graphics*, pages 153–174, 1984.

[FNTW90] K. Y. Fung, T. M. Nicholl, R. E. Tarjan, and C. J. Van Wyk. Simplified linear-time jordan sorting and polygon clipping. *Information Processing Letters*, pages 85–92, 1990.

[Goo89a] M. T. Goodrich. Intersecting line segments in parallel with an output sensitive number of processors. In *Proc. First Symposium on Parallel Algorithms and Architectures*, pages 127–137, 1989.

[Goo89b] M. T. Goodrich. Triangulating a polygon in parallel. *Journal of Algorithms*, 10:327–351, 1989.

[Goo91] M. T. Goodrich. Constructing arrangements optimally in parallel. In *Proc. Third Symposium on Parallel Algorithms and Architectures*, pages 169–179, 1991.

[GSG89] M. T. Goodrich, S.B. Shauck, and S. Guha. Parallel methods for visibility and shortest path problems in simple polygons. Technical report, Computer Science Department, The Johns Hopkins University, 1989.

[HJW90] T. Hagerup, H. Jung, and E. Welzl. Efficient parallel computation of arrangements of hyperplanes in d dimensions. In *Proc. Second Symposium on Parallel Algorithms and Architectures*, pages 290–297, 1990.

[HMRT86] K. Hoffman, K. Mehlhorn, P. Rosenstiehl, and R. Tarjan. Sorting jordan sequences in linear time using level-linked search trees. *Information and Control*, pages 170–184, 1986.

[Mul88] K. Mulmuley. A fast planar point location algorithm: part I. In *Proc. 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 580–589, 1988.

[RR89] S. Rajasekaran and J.H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing*, 18(3):594–607, 1989.

[Tou88] G. Toussaint. *Computational Morphology.* North-Holland, 1988.

Appendix H

This appendix gives an algorithm for computing the trapezoidal diagram of a set S of n line segments in $O(n^2)$ expected time. The algorithm uses randomized divide-and-conquer: take random $R \subset S$ of size $r = n/\log n$, and compute $\mathcal{T}(R)$ using Goodrich's algorithm [Goo89a]. (Alternatively, use an algorithm analogous to the suboptimal one given in [HJW90].) Insert the remaining segments of S into $\mathcal{T}(R)$, and use subsampling of the segments meeting each $T \in \mathcal{T}(R)$ to yield subproblems all of size $O(\sqrt{\log n})$ with high probability. Now use an optimal serial algorithm for such subproblems, and Goodrich's algorithm for the remainder.

The insertion step, in more detail: note that the complete adjacency representation of $\mathcal{T}(R)$ divides the segments of R into $O(r^2)$ total pieces, with $O(r^2)$ on any segment of R (actually $O(r\alpha(r))$ trapezoids meet any given segment, but the larger bound will suffice). We use list ranking [AM88] to obtain, for each segment in R, a sorted array of the pieces induced by $\mathcal{T}(R)$, with pointers to the trapezoids meeting each piece. Now for each segment $a \in S$, use binary search on the array for every segment $b \in R$, to find the trapezoids containing the intersection point of a and b.

This does not give all the trapezoids meeting a; the remaining trapezoids are obtained as follows: consider the *convex diagram* of R, the subdivision induced using visibility edges from segment endpoints only, not intersection points. The trapezoids within each region of the convex diagram can be ordered top to bottom; such orderings can be obtained by list ranking, applied to adjacencies between trapezoids with common edges that are visibility segments from intersection points. The trapezoids in a convex cell that meet a segment $a \in S$ are an interval in that list, and the highest and lowest trapezoids contain either intersection points of a with R, or endpoints of a. Binary searches suffice to locate the endpoints of a in the list, if either are contained in trapezoids in the cell. Finally, the trapezoids meeting a, other than the highest and lowest, can be obtained after sufficient processors are allocated to do this. (Note that the number of trapezoids met is available.) We have, for every $a \in S$, the set of trapezoids of $\mathcal{T}(R)$ that it meets. By integer sorting we obtain a collection of subproblems as discussed in §3.1. This completes the first phase of processing.

Now for each $T \in \mathcal{T}(R)$, take a random subset of the segments that meet it, of size $K|n_T|\log|n_T|/\sqrt{\log n}$, where n_T is the number of segments meeting T, and K is an appropriate constant. Again apply Goodrich's algorithm to each such subset, and use the same insertion technique, to obtain a collection of subproblems. Now after appropriate processor allocations, apply an optimal serial algorithm to those subproblems with no more than $\sqrt{\log n}$ segments, and apply Goodrich's algorithm to the remainder.

Analysis. It is easy to verify that the algorithm consists of a constant number of stages each requiring $O(\log n)$ time in the worst case.

It is also easy to verify that $O(n^2)$ expected work is required, using Lemma 1.

For example, computing the trapezoidal diagrams of the random subsets in the second phase of processing requires expected work proportional to

$$\sum_{T \in \mathcal{T}(R)} [O(|n_T|) \log |n_T| / \sqrt{\log n}]^2 \log |n_T|;$$

since $|n_T| = O(\log^2 n)$ for all T with probability $1 - 1/n$, the above is no more than

$$\sum_{T \in \mathcal{T}(R)} O(|n_T|^2)(\log \log n)^2 / \log n,$$

which is $O(n^2 (\log \log n)^2 / \log n)$.