University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS) Department of Computer & Information Science

December 1993

Optimal Parallel Randomized Algorithms for the Voronoi Diagram of Line Segments in the Plane and Related Problems

Sanguthevar Rajasekaran
University of Pennsylvania

Suneeta Ramaswami
University of Pennsylvania

Follow this and additional works at: https://repository.upenn.edu/cis_reports

Recommended Citation

Sanguthevar Rajasekaran and Suneeta Ramaswami, "Optimal Parallel Randomized Algorithms for the Voronoi Diagram of Line Segments in the Plane and Related Problems", . December 1993.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-93-99.

This paper is posted at ScholarlyCommons. https://repository.upenn.edu/cis_reports/265
For more information, please contact repository@pobox.upenn.edu.

Optimal Parallel Randomized Algorithms for the Voronoi Diagram of Line Segments in the Plane and Related Problems

Abstract

In this paper, we present an optimal parallel randomized algorithm for the Voronoi diagram of a set of n non-intersecting (except possibly at endpoints) line segments in the plane. Our algorithm runs in $O(\log n)$ time with very high probability and uses $O(n)$ processors on a CRCW PRAM. This algorithm is optimal in terms of $P.T$ bounds since the sequential time bound for this problem is $\Omega(n \log n)$. Our algorithm improves by an $O(\log n)$ factor the previously best known deterministic parallel algorithm which runs in $O(\log^2 n)$ time using $O(n)$ processors [13]. We obtain this result by using random sampling at "two stages" of our algorithm and using efficient randomized search techniques. This technique gives a direct optimal algorithm for the Voronoi diagram of points as well (all other optimal parallel algorithms for this problem use reduction from the 3-d convex hull construction).

Comments

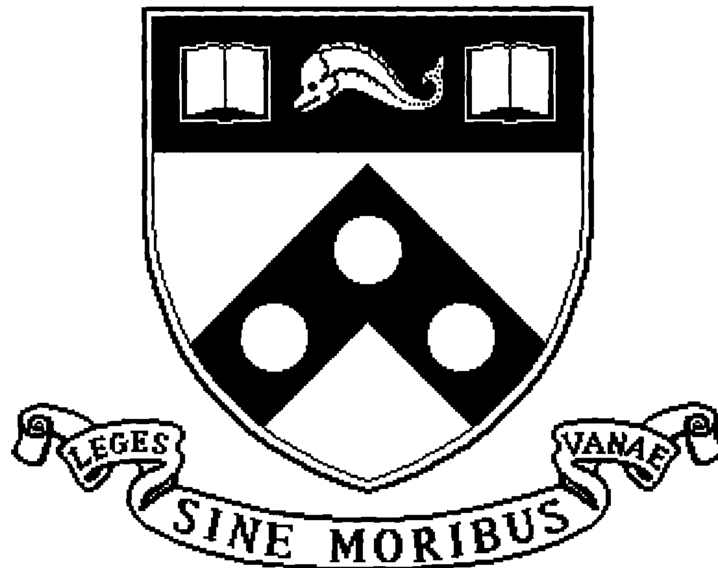
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-93-99.

This technical report is available at ScholarlyCommons: https://repository.upenn.edu/cis_reports/265

Optimal Parallel Randomized Algorithms for the Voronoi Diagram of Line Segments in the Plane and Related Problems

MS-CIS-93-99

Sanguthevar Rajasekaran
Suneeta Ramaswami



University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

December 1993

Optimal Parallel Randomized Algorithms for the Voronoi Diagram of Line Segments in the Plane and Related Problems*

Sanguthevar Rajasekaran Suneeta Ramaswami[†]

Department of Computer and Information Science,
University of Pennsylvania, Philadelphia, PA 19104

Abstract

In this paper, we present an optimal parallel randomized algorithm for the Voronoi diagram of a set of n non-intersecting (except possibly at endpoints) line segments in the plane. Our algorithm runs in $O(\log n)$ time with very high probability and uses $O(n)$ processors on a CRCW PRAM. This algorithm is optimal in terms of $P.T$ bounds since the sequential time bound for this problem is $\Omega(n \log n)$. Our algorithm improves by an $O(\log n)$ factor the previously best known deterministic parallel algorithm which runs in $O(\log^2 n)$ time using $O(n)$ processors [13]. We obtain this result by using random sampling at "two stages" of our algorithm and using efficient randomized search techniques. This technique gives a direct optimal algorithm for the Voronoi diagram of points as well (all other optimal parallel algorithms for this problem use reduction from the 3-d convex hull construction).

1 Introduction

Voronoi diagrams are elegant and versatile geometric structures which have applications for a wide range of problems in computational geometry and other areas. For example, computing the minimum weight spanning tree, or the all-nearest neighbor problem for a set of line segments can be solved immediately and efficiently from the Voronoi diagram of line segments. As we learnt from [18], these diagrams can also be used to compute a country's territorial waters! Certain two-dimensional motion planning problems can be solved quickly when the Voronoi diagram is available [20]. Thus, exploiting parallelism to obtain faster solutions is a desirable goal. In this paper, we are interested in developing an optimal parallel randomized algorithm on a PRAM (Parallel Random Access Machine) for the construction of the Voronoi diagram of a set of non-intersecting (except

*This paper has been submitted for review to the Symposium on Computational Geometry to be held in Stony Brook, June 1994.

[†]This research is partially supported by ARO Grant DAAL03-89-C-0031 including participation by the U.S. Army Human Engineering Laboratory.

possibly at endpoints) line segments in the plane. The first sequential algorithm for this problem was given by Lee and Drysdale [18], which ran in $O(n\log^2 n)$ time. This run-time was later improved to $O(n \log n)$ in numerous papers (Kirkpatrick [16], Yap [24] and Fortune [10]), which is optimal since sorting can be reduced to this problem. The best known parallel deterministic algorithm was given by Goodrich, Ó'Dúnlaing and Yap [13] and runs in $O(\log^2 n)$ time using $O(n)$ processors. It seems unlikely that existing deterministic techniques can be used to improve this run-time.

2 Background

Recent years have seen significant advances in parallel algorithm design for computational geometry problems. Some of the earliest work in this area was done by Chow [5] and Aggarwal *et. al.* [1]. In these papers, the authors gave parallel algorithms for various fundamental problems such as two-dimensional convex hulls, planar-point location, trapezoidal decomposition, Voronoi diagram of points, triangulation *etc.*, which are known to have sequential run-times of $O(n \log n)$. Most of their algorithms, though in NC, were not optimal in $P.T$ bounds and a number of them have since been improved. Atallah, Cole and Goodrich [3] demonstrated optimal deterministic algorithms ($O(n)$ processors and $O(\log n)$ run-time) for many of these problems by applying the versatile technique of cascading divide-and-conquer and building on data structures developed in [1]. Cole and Goodrich give further applications of this technique in [8]. Reif and Sen [23] also obtained optimal randomized parallel algorithms for a number of these problems; these algorithms use n processors and run in $O(\log n)$ time with very high probability.

The important problems of constructing Voronoi diagrams of points in two dimensions and convex hulls of points in three dimensions, however, eluded optimal parallel solutions for a long time. Both these problems have sequential run-times of $O(n \log n)$. Aggarwal *et. al.* [1] gave $O(\log^2 n)$ and $O(\log^3 n)$ time algorithms (using n processors) for the Voronoi diagram and convex hull problems, respectively, and the technique of cascaded-merging could not be extended to these problems to improve their run-times [8]. Very recently, Goodrich [12] has given an algorithm for 3-d convex hulls that does optimal *work*, but has $O(\log^2 n)$ run-time, and Amato and Preparata [2] have described an algorithm that runs in $O(\log n)$ time but uses $n^{1+\epsilon}$ processors. Randomization, however, proves to be very useful in obtaining optimal run-time as well as optimal $P.T$ bounds[1]. In [22], Reif and Sen gave an optimal randomized algorithm for the construction of the convex hull of points in three dimensions. Since the problem of finding the Voronoi diagram of points in two dimensions can be reduced to the three-dimensional convex hull problem, they also obtained an optimal parallel method for the former. Their algorithm runs in $O(\log n)$ time, using n processors, with very high probability, and there are no known deterministic algorithms that match these bounds.

[1] Sorting can be reduced to these problems, and hence the best possible run-time will be $O(\log n)$ on EREW and CREW PRAMs.

3 The Use of Randomization

The technique of randomization has been used to design sequential as well as parallel algorithms for a wide range of problems. In particular, efficient randomized algorithms have been developed for a number of computational geometry problems. Recent work by Clarkson and Shor [6], Mulmuley [19], and Haussler and Welzl [14] has shown that random sampling can be used to obtain better upper bounds for various geometric problems such as higher-dimensional convex hulls, halfspace range reporting, segment intersections, linear programming *etc.*

Clarkson and Shor [6] used random sampling techniques to obtain tight bounds on the *expected* use of resources by algorithms for various geometric problems. The main idea behind their general technique is to use random sampling to divide the problem into smaller ones. The manner in which the random sample is used to divide the original input into subproblems depends on the particular geometric problem under consideration. They showed that for a variety of such problems, given a randomly chosen subset R of the input set S, the *expected* size of each subproblem is $O(|S|/|R|)$ and the *expected* total size is $O(|S|)$. A sample that satisfies these conditions is said to be *good*, and *bad* otherwise. They showed that any randomly chosen sample is good with constant probability, and hence bad with constant probability as well. This allows us to obtain bounds on the *expected* use of resources, but does not give high probability results (i.e. bounds that hold with probability $\geq (1 - 1/n^{\alpha})$, where n is the input size, and $\alpha > 0$). As pointed out by Reif and Sen [22], this fact proves to be an impediment in the parallel environment due to the following reason: Parallel algorithms for such problems are, typically, recursive. For sequential algorithms, since the expectation of the sum is the sum of expectations, it is enough to bound the expected run-time of each step. For recursive parallel algorithms, the run-time at any stage of the recursion will be the *maximum* of the run-times of the subproblems spawned at that stage. There is no way of determining the maximum of expected run-times without using higher moments. Moreover, even if we can bound the expected run-time at the lowest level of the recursion, this bound turns out to be too weak to bound the total run-time of the algorithm.

In [22], Reif and Sen give a novel technique to overcome this problem. A parallel recursive algorithm can be thought of as a *process tree*, where a node corresponds to a procedure at a particular stage of the recursion, and the children of that node correspond to the subproblems created at that stage. The basic idea of the technique given in [22] is to find, at every level of the process tree, a good sample with high probability. By doing this, they can show that the run-time of the processes at level i of the tree is $O(\log n/2^i)$ with very high probability, and hence the run-time of the entire algorithm is $O(\log n)$ with very high probability. By choosing a number of random samples (say $g(n)$ of them; typically $g(n) = O(\log n)$), we are guaranteed that one of them will be good with high likelihood. The procedure to determine if a sample is good or not will have to be repeated for each of the $g(n)$ samples. However, we would have to ensure that this would not cause the processor bound of $O(n)$ to be exceeded and this is done by *polling* i.e. using only a fraction of the input ($1/g(n)$, typically) to determine the "goodness" of a sample. The idea is that the assessment of a sample (good or bad) made from this smaller set is a very good reflection of the assessment that would be made from the entire set. Thus they have a method to find a good sample efficiently at every level of the process tree, and is useful for converting expected value

results into high probability results.

Note that the total size of the subproblems can be bound to only within a constant multiple of the original problem size. Thus in a process tree of $O(\log\log n)$ levels, this could lead to a polylogarithmic factor increase in processor bound. In [6], Clarkson and Shor get around this problem by using only a constant number of levels of recursion in their algorithm. They are able to do this by combining the divide-and-conquer technique with incremental techniques (which are inherently sequential; see [6] for further details). As mentioned earlier, Reif and Sen's [22] strategy to handle this problem is to eliminate redundancy at every level of the process tree. This step is non-trivial and quite problem-specific. Our approach in this paper gets rid of this need altogether. In other words, we do not need to devise a method to control total problem size at every level of the process tree. The basic idea is to use random sampling at two stages of the algorithm. We show that the polylog factor increase in processor bound actually gives us a method to choose much larger samples. By choosing samples in this manner, we are essentially able to ignore the problem of increase in processor bound. By developing efficient search strategies to determine subproblems, we are able to obtain an optimal algorithm for the Voronoi diagram of line segments in the plane. We think that our approach is general enough to apply to other problems as well.

We would like to point out that our strategy applies to the Voronoi diagram of points in the plane as well, thus giving a direct algorithm for this problem (instead of using the reduction to 3-d convex hulls). During the course of the paper, we will point out the analogous steps for the case of points in the plane.

4 Preliminaries, Definitions and Notation

The parallel model of computation that will be used in this paper is the *C*oncurrent *R*ead *C*oncurrent *W*rite (CRCW) PRAM. This is the sychronous shared memory model of parallel computation in which processors may read from or write into a memory cell simultaneously. Write conflicts are resolved arbitrarily; in other words, the protocol used for resolving such conflicts does not affect our algorithm. Each processor performs standard real-arithmetic operations in constant time. In addition, each processor has access to a random number generator that returns a random number of $O(\log n)$ bits in constant time. A randomized parallel algorithm $\mathcal{A}$ is said to require resource bound $f(n)$ with *very high probability* if, for any input of size n, the amount of resource used by $\mathcal{A}$ is at most $\bar{c}.\alpha.f(n)$ with probability $\geq 1 - 1/n^{\alpha}$ for positive constants $\bar{c}, \alpha$ $(\alpha > 1)$. $\tilde{O}$ is used to represent the complexity bounds of randomized algorithms i.e. $\mathcal{A}$ is said to have resource bound $\tilde{O}(f(n))$. The following is an important theorem.

Theorem 4.1 (Reif and Sen, [23]) *Given a process tree that has the property that a procedure at depth i from the root takes time T_i such that $Pr[T_i \geq k(\epsilon')^i \alpha \log n] \leq 2^{-(\epsilon')^i \alpha \log n}$, then all the leaf-level procedures are completed in $\tilde{O}(\log n)$ time, where k and α are constants greater than zero, and $0 < \epsilon' < 1$.*

Note that the number of levels in the process tree will be $O(\log\log n)$. Intuitively, the above theorem says that if the time taken at a node which is at a distance i from the root is $O((\log n)/2^i)$

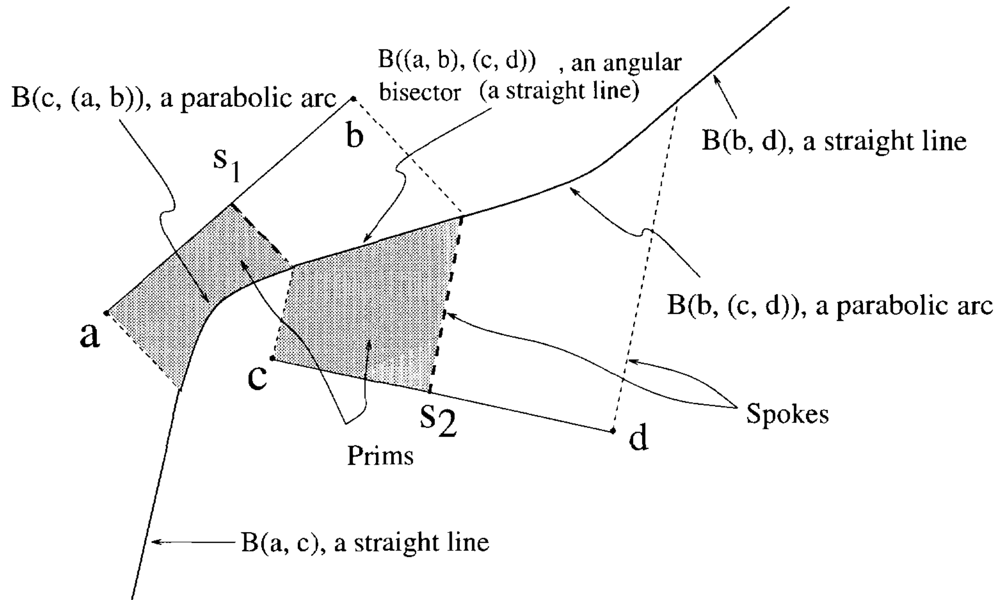


Figure 1: The bisector of two line segments s_1 and s_2.

with high probability, then the run-time of the entire algorithm is $\tilde{O}(\log n)$.

4.1 Voronoi Diagrams

We now give definitions and establish some notation for the geometric objects of interest in this paper i.e. Voronoi diagrams (these definitions are standard; see e.g. [13, 18]). Let S be a set of nonintersecting closed line segments in the plane. For simplicity of description, we will assume that the elements of S are in general position i.e. no three of them are cocircular. Following the convention in [18, 24], we will consider each segment $s \in S$ to be composed of three distinct objects: the two endpoints of s and the open line segment bounded by those endpoints. The Euclidean distance between two points p and q is denoted by $d(p, q)$. The *projection* of a point q on to a closed line segment s with endpoints a and b, denoted $proj(q, s)$, is defined as follows: Let p be the intersection point of the straight line containing s (call this line $\overleftrightarrow{s}$), and the line going through q that is perpendicular to $\overleftrightarrow{s}$. If p belongs to s, then $proj(q, s) = p$. If not, then $proj(q, s) = a$ if $d(q, a) < d(q, b)$ and $proj(q, s) = b$, otherwise. The *distance* of a point q from a closed line segment s is nothing but $d(q, proj(q, s))$. By an abuse of notation, we denote this distance as $d(q, s)$. Let s_1 and s_2 be two objects in S. The *bisector* of s_1 and s_2, $B(s_1, s_2)$, is the locus of all points q that are equidistant from s_1 and s_2 i.e. $d(q, s_1) = d(q, s_2)$. Since the objects in S are either points or open line segments, the bisectors will either be parts of lines or parabolas. The bisector of two line segments is shown in Figure 1. Note that if S is a set of points, the distance relation is the obvious one, and all the bisectors are parts of straight lines. The formal definition is stated below, followed by an important theorem.

Definition 4.2 *The* Voronoi region, *$V(s)$, associated with an object s in S is the locus of all points that are closer to s than to any other object in S i.e. $V(s) = \{p \mid d(p, s) \leq d(p, s')$ for all*

$s' \in S\}$. *The* Voronoi diagram *of* S, $\text{Vor}(S)$, *is the union of the Voronoi regions* $V(s)$, $s \in S$. *The boundary edges of the Voronoi regions are called* Voronoi edges, *and the vertices of the diagram,* Voronoi vertices.

Theorem 4.3 (Lee et al. [18]) *Given a set* S *of* n *objects in the plane (in this paper, these objects will either be nonintersecting closed line segments or points), the number of Voronoi regions, Voronoi edges, and Voronoi vertices of* $\text{Vor}(S)$ *are all* $O(n)$. *To be precise, for* $n \geq 3$, $\text{Vor}(S)$ *has at most* n *vertices and at most* $3n - 5$ *edges.*

It is convenient to divide Voronoi regions into smaller regions by augmenting the diagram in the following way (such an augmentation was originally proposed by Kirkpatrick [16]): If v is a Voronoi vertex of $V(s)$, and if $v' = proj(v, s)$, then the line segment obtained by joining v and v' is a *spoke* of $V(s)$. Note that a spoke of $V(s)$ must lie entirely in $V(s)$ since $V(s)$ is generalized-star-shaped with nucleus s [18]. The spokes define new sub-regions within $V(s)$. These sub-regions bounded by two spokes, part of e and a Voronoi edge are called *primitive regions* (*prims* for short) [13]. For each line segment s, there are spokes that are perpendicular to s at its endpoints. These come from (degenerate) vertices that demarcate the region of $V(s)$ that is closer to the open line segment from the region that is closer to the endpoints. Some spokes and prims are shown in Figure 1. Hereafter, all references to the Voronoi diagram will assume that it is augmented as above.

In the case of the Voronoi diagram of points, all bisectors are straight lines and hence two bisectors can intersect at most once. However for line segments, the bisectors are composed of straight line segments and parabolic arcs. Consequently, we give the following lemma, which will be of use to us later on in the paper.

Lemma 4.4 *Given line segments* s, s_1 *and* s_2 *in the plane, the two bisectors* $B(s, s_1)$ *and* $B(s, s_2)$ *can intersect at most twice.*

We give below some results that will be useful for the brute force construction of the Voronoi diagram of a sample of the appropriate size, and will be used in the course of our algorithm. We state the results for line segments as well as points.

Lemma 4.5 *The Voronoi diagram of a set of* n *line segments in the plane can be constructed on a CREW PRAM in* $O(\log n)$ *time using* n^3 *processors.*

Proof: Omitted. □

Lemma 4.6 *The Voronoi diagram of a set of* n *points in the plane can be constructed on a CREW PRAM in* $O(\log n)$ *using* n^2 *processors.*

The above lemma follows easily from the fact that intersections of n half-planes can be found in $O(\log n)$ time using n processors [1, 4]. (Amato and Preparata's [2] method, though a more

complicated approach, would give us the same result.) The following are well-known results that give us non-optimal parallel algorithms for the Voronoi diagram problems. These will be used when the input to a subproblem is of an appropriately small size. Any polylogarithmic time parallel algorithm that uses n processors will be enough for our requirements.

Lemma 4.7 (Goodrich *et. al.*, **[13])** *The Voronoi diagram of a set of n line segments in the plane can be constructed on a CREW PRAM in $O(\log^2 n)$ time using n processors.*

Lemma 4.8 (Aggarwal *et. al.*, **[1])** *The Voronoi diagram of a set of n points in the plane can be constructed on a CREW PRAM in $O(\log^2 n)$ time using n processors.*

5 Randomized Algorithms for Voronoi Diagrams

In this section, we develop an optimal parallel randomized algorithm for the construction of the Voronoi diagram of a set of line segments in the plane. As we mentioned in Section 3, Reif and Sen [22] use the novel technique of *polling* to give an optimal randomized algorithm for the three-dimensional convex hull problem. This immediately gives an optimal randomized method for the Voronoi diagram construction of a set of points in the plane because this problem is $O(\log n)$ (parallel) time reducible to the 3-d convex hull problem. However, no analogous reduction is at hand for the case of line segments. Our optimal parallel randomized algorithm tackles the Voronoi diagram problem directly which, to our knowledge, has not been done before. The technique that we present in the forthcoming sections is general enough that it can be applied to the case of line segments as well as points in the plane in order to obtain optimal randomized parallel algorithms. As a result, we also obtain a new and simpler randomized parallel method for the Voronoi diagram of points. We would like to point out here that even though, for the sake of simplicity, we restrict our attention to line segments in the plane, this technique would be applicable even when the input consists of circular arcs (see Yap's [24] sequential algorithm for some detail on including circular arcs).

5.1 Outline of the Method

We give here a broad outline of our parallel algorithm here. Let $S = \{s_1, s_2, \ldots, s_n\}$ be the input set of line segments in the plane and let R be a random sample from S. Let $|R| = n^\epsilon$ for some $0 < \epsilon < 1$. The sample R will be used to divide the original input S into smaller subproblems so that each of these can be solved in parallel. The technique of using a random sample to divide the original problem into subproblems will be useful only if we can show that the subproblems are of roughly equal size ($O(n^{1-\epsilon})$) and that the total size of all the subproblems is almost equal to the size of the original problem ($O(n)$). The expected value resource bounds for computational geometry problems obtained by Clarkson and Shor [6] will be applied here. As mentioned in Section 3, due to the nature of randomized parallel algorithms it is necessary to efficiently find, at each level of the process tree, such a sample with high probability. The technique of polling developed by Reif and Sen [22] will be utilised towards this end. So a crude outline of the algorithm could be as follows.

1. Construct the Voronoi diagram of R using the brute force technique described in Lemma 4.5 of Section 4 (Lemma 4.6 for the Voronoi diagram of points in the plane). Call this diagram $Vor(R)$. We use $Vor(R)$ to divide the original problem into smaller problems which will be solved in parallel.
2. Process $Vor(R)$ appropriately in order to efficiently find the input set of line segments to each of these subproblems.
3. Recursively compute (in parallel) the Voronoi diagram of each subproblem.
4. Obtain the final Voronoi diagram from the recursively computed Voronoi diagrams.

By choosing an appropriate ϵ, we can ensure that the first step can be done in $O(\log n)$ time using n processors. In Section 5.4, we describe the use of a randomized search technique in order to efficiently find the subproblems defined by a chosen sample. We show that step 2 can be carried out in $O(\log n)$ time with high probability using n processors. In Section 5.5, we describe the merge technique to compute the Voronoi diagram from recursively computed Voronoi diagrams (this step is non-trivial) and show that the final merge step can be done in $O(\log n)$ time using n processors. Thus, the recurrence relation for the run-time is $T(n) = T(n^{1-\epsilon}) + \tilde{O}(\log n)$, which solves to $\tilde{O}(\log n)$ (this follows from Theorem 4.1). However, the description of the algorithm that we have given here is incomplete.

As mentioned earlier, there is an important side-effect in such recursive algorithms that we have to consider. When we use a random sample to divide the original problem into smaller ones, we can succeed in bounding the total size of the subproblems to only within a constant multiple of n. In a recursive algorithm, this results in an increase in the total problem size as the number of recursive levels increases. For a sample size of $O(n^{\epsilon})$, the depth of the process tree for a parallel randomized algorithm would be $O(\log\log n)$, and even this could result in a polylogarithmic factor increase in the total problem size. In [22], Reif and Sen get around this problem by eliminating redundancy in the input to the subproblems at every level of the recursion. In other words, since it is known that the *final* output size is $O(n)$, it is possible to eliminate those input elements from a subproblem which do not contribute to the final output. By doing this, they bound the total problem size at *every* level of the process tree to be within $c.n$ for some constant c. This step is non-trivial and, in general, avoiding a growth in problem size in this manner can be quite complicated. Moreover, the strategy that can be used to eliminate redundancy seems to be very problem-specific. We describe a method to use sampling at two stages of the algorithm, which will help us to overcome the problem of the increase in total input size as our algorithm proceeds down the process tree. Moreover, it appears that our strategy is general enough to be applicable to other kinds of problems as well. This technique is described in further detail in Section 5.3.

Observe that the issue of bounding the total size of the subproblems does not come up in "one-dimensional" problems like sorting because each element of the input set can lie in exactly one subproblem. This is not the case for problems like Voronoi diagram construction. This property makes it necessary to device efficient search strategies in order to determine all the subproblems that an element lies in. In our algorithm, this need is particularly important because of the fact that one of the stages of sampling entails larger sample sizes (this will become clearer in Section 5.3). We give such an efficient search strategy in Section 5.4.

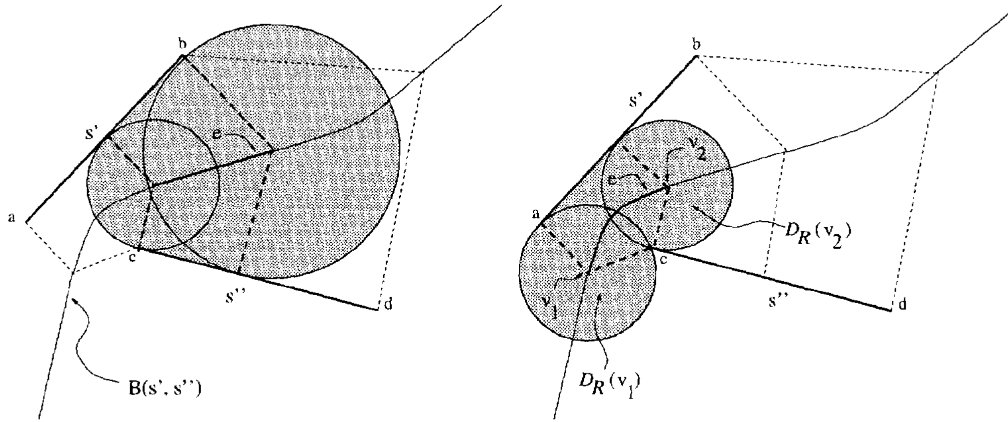


Figure 2: The shaded area is the region $\mathcal{C}_R(e)$ associated with the edge e.

5.2 Defining the Subproblems

We now give a precise description of how the Voronoi diagram of a random sample R is used to divide the original input S into smaller problems. Let $|R| = r$. Consider an edge e of $Vor(R)$. e has two primitive regions (prims were defined in Section 4) on either side of it; call these $\mathcal{P}_R^1(e)$ and $\mathcal{P}_R^2(e)$. Every vertex v of $Vor(R)$ is equidistant from exactly three elements of R, and v is closer to these three elements than to any other element of R. Thus every vertex defines a circle such that exactly three elements of R are incident on it and the interior of the circle is empty. Let $\mathcal{D}_R(v)$ denote the circle (and its interior) defined by Voronoi vertex v of $Vor(R)$. Let v_1 and v_2 be the two vertices of e.

Consider now the region obtained by the union of $\mathcal{P}_R^1(e)$, $\mathcal{P}_R^2(e)$, and the two circles $\mathcal{D}_R(v_1)$ and $\mathcal{D}_R(v_2)$ (see Figure 2); in the case of unbounded edges, we have two prims and one circle. Call this region $\mathcal{C}_R(e)$. Observe that for the Voronoi diagram of points, it is enough to consider $\mathcal{C}_R(e)$ to be the union of just the two circles defined by the two vertices of e because the prims $\mathcal{P}_R^1(e)$ and $\mathcal{P}_R^2(e)$ always lie entirely within this union. This is not always the case for line segments, as can be seen in Figure 2.

Remark: We would like the subproblem associated with every edge of $Vor(R)$ to satisfy the following condition: if the final Voronoi region $V(s_i)$ of element $s_i \in S$ intersects $\mathcal{P}_R^1(e)$ or $\mathcal{P}_R^2(e)$, then we would like at least all such s_i to be part of the input associated with the edge e.

For every element s_i of S, consider the set of points that are closer to s_i than to any other element of R. In other words, consider the Voronoi region of s_i in the Voronoi diagram of the set $R \cup \{s_i\}$. Denote this by $V^R(s_i)$. Clearly, the final Voronoi region of s_i will be a region smaller than (or equal to) $V^R(s_i)$. An element s_i will belong to the subproblem associated with edge e if and only if $V^R(s_i)$ has a non-empty intersection with $\mathcal{P}_R^1(e)$ and $\mathcal{P}_R^2(e)$. Observe that determining input for each subproblem in this manner satisfies the desired condition that we stated in the remark above. Note also that there might be s_i that are part of this input whose final Voronoi region $V(s_i)$ does not intersect $\mathcal{P}_R^1(e)$ or $\mathcal{P}_R^2(e)$. $V^R(s_i)$ intersects $\mathcal{P}_R^1(e)$ and $\mathcal{P}_R^2(e)$ if and only if *the element s_i has a non-empty intersection with $\mathcal{C}_R(e)$.*

Let $X(\mathcal{C}_R(e))$ denote the set of all elements $s_i \in S$ such that s_i has a non-empty intersection with $\mathcal{C}_R(e)$. Thus, the input set for the subproblem associated with e will be nothing but $X(\mathcal{C}_R(e))$. Each of the $Vor(X(\mathcal{C}_R(e)))$ will be computed recursively. The number of edges in $Vor(R)$ will be at most $3r - 5$ (Theorem 4.3), and, hence, so will the number of regions $\mathcal{C}_R(e)$. Let $\mathcal{C}_R = \{\mathcal{C}_R(e) \mid e$ is a Voronoi edge from $Vor(R)\}$. We can now use Clarkson and Shor's result (Corollary 3.8, [6]) to obtain the following. Note that it takes four elements of R to define each region $\mathcal{C}_R(e)$: the two elements that e bisects, and two other elements that determine the two vertices of e (referring now to the notation used in [6], clearly $b = 4$ and the "ranges" identified by the function δ are precisely the regions $\mathcal{C}_R(e)$ defined here).

Lemma 5.1 *Given a random sample R of size r from a set of objects S of size n and using the notation established above, both of the following conditions hold with probability at least* $1/2$*:*

$$(a) \quad \max_{\mathcal{C}_R(e) \in \mathcal{C}_R} \{ \, |X(\mathcal{C}_R(e))| \, \} \leq k_{max} \, (n/r) \, \log r$$

$$(b) \quad \sum_{\mathcal{C}_R(e) \in \mathcal{C}_R} |X(\mathcal{C}_R(e))| \leq k_{tot} \, (n/r) \, E(\, |\mathcal{C}_R| \,)$$

where k_{max} and k_{tot} are constants (obtained from Corollary 3.8 *in [6]).*

Since $E(\, |\mathcal{C}_R| \,)$ is $O(r)$, the above conditions guarantee that with probability at least $1/2$, the total size of the subproblems is almost equal to the size of the original problem and the subproblems are of roughly equal size. As in [6, 22], if a sample R of S satisfies these conditions then it is called a *good* sample and *bad* otherwise. In order to solve the subproblems in parallel and obtain an optimal run-time, R has to be a good sample with *high probability.*

5.3 Two Stages of Sampling

As mentioned in Section 3, Reif and Sen [22] describe the novel technique of polling in order to obtain a good sample with high probability from samples that are only expected to be good. If $O(\log n)$ samples are chosen, instead of just one, then it follows from Lemma 5.1 that with high probability one of these samples will be good. Let $R_1, R_2, \ldots, R_{a \log n}$ be these samples (where the value of a is fixed according to the desired success probability of the algorithm) of size $O(n^\epsilon)$ each. Let us assume for now that we have an efficient (n processors and $O(\log n)$ time) procedure to compute the total subproblem size $\sum_{\mathcal{C}_{R_i}(e) \in \mathcal{C}_{R_i}} | \, X(\mathcal{C}_{R_i}(e)) \, |$ for each $Vor(R_i)$. However, in order to determine which of these R_i is good, this procedure will have to be repeated for each of the $O(\log n)$ samples. This will mean an increase in the processor bound of $O(n)$ which we want to avoid. Reif and Sen [22] introduce the idea of polling in order to overcome this problem. They determine the goodness of sample R_i by using only $O(n/\log^b n)$ randomly chosen elements from the input set, where $b > 2$ is some constant. The procedure to determine the goodness of the $a \log n$ samples can be run in parallel for all the R_i.

The randomized parallel algorithm is then run on the good sample found in the above manner. However the technique outlined so far still *does not* guarantee that we stay within the desired

processor bound of $O(n)$. The best bound that Lemma 5.1 can give us is that the total size of the subproblems at level $(i+1)$ of the process tree is k_{tot} times the total size of the subproblems at level i. This implies that after $O(\log\log n)$ levels, the total size could increase by a polylogarithmic factor. Suppose the total size at the leaf level of the process tree is at most $n.\log^c n$, for some constant c. If the input to the divide-and-conquer algorithm were to be of size $n/\log^c n$, then the total problem size at the leaf level of the process tree would be $O(n)$. This observation (along with the results developed in the following sections) yields the following.

Theorem 5.2 *The Voronoi diagram of a set of n line segments can be constructed in $O(\log n)$ time with high probability using $n\log^c n$ processors.*

The above theorem actually enables us to choose samples of size much larger than $O(n^\epsilon)$. In particular, such a sample S' could be of size $O(n/\log^q n)$, q being a constant $> c$. If S' is a good sample, then it would again divide the original input into smaller problems of roughly equal size (i.e. $X(\mathcal{C}_{S'}(e))$ would be of size roughly $O(\log^q n)$ for all Voronoi edges e in *Vor*(S')) and $\sum |X(\mathcal{C}_{S'}(e))|$ would be $O(n)$. In order to compute the Voronoi diagram of these subproblems, we can then use any non-optimal algorithm like the one stated in Lemma 4.7 (Lemma 4.8 for points in the plane). We would, however, like to be able to find such a good sample S' with high probability. As we know from Lemma 5.1, there is a constant probability that S' is a good sample. As before, if we choose $O(\log n)$ such samples, we know that at least one of them will be good with very high probability.

Let $N = n/log^q n$. Let $S_1,\ S_2,\ \ldots,\ S_{d\log n}$ be the $O(\log n)$ samples of size N each. Since the size of S_i is large, we obviously cannot afford to construct *Vor*(S_i) using a brute force technique (as we can do with samples of size $O(n^\epsilon)$). We will have to run the randomized parallel algorithm using each of these S_i as input. Let $R^i_1,\ R^i_2,\ \ldots,\ R^i_{a\log N}$ be the $O(\log N)$ random samples, each of size N^ϵ, chosen from S_i. Thus the skeleton of our algorithm will now be as follows. Note that the testing of the samples S_i is done with respect to a restricted input set (polling).

Algorithm VORONOI_DIAGRAM;
- $N := n/\log^q n$.
- Pick $d\log n$ random samples $S_1, S_2, \ldots, S_{d\log n}$ of size N each.
- Let I be a random subset of the input set S such that $|I| = n/\log^{\bar{q}} n$, $\bar{q}$ being a constant $< q$.
- $S' :=$ PICK_THE_RIGHT_SAMPLE$(S_1, S_2, \ldots, S_{d\log n}, I)$.
- Partition the entire input S according to the good sample S'; such a method is given in Section 5.4.
- Solve each subproblem using a non-optimal technique (Lemma 4.7 or Lemma 4.8).
- Merge the results (see Section 5.5).

Function PICK_THE_RIGHT_SAMPLE$(S_1, S_2, \ldots, S_{d\log n}, I)$.
- Do the following in parallel for each S_i $(1 \le i \le d\log n)$.

1. (a) Choose $a\log n$ random samples $R^i_1, R^i_2, \ldots, R^i_{a\log n}$ each of size N^ϵ from the set S_i.

   (b) Construct the Voronoi diagram of each R^i_j $(1 \le j \le a\log N)$ using the brute force

technique described in Lemma 4.5 of Section 4 (Lemma 4.6 for the Voronoi diagram of points in the plane).

(c) Determine which of these R^i_j is a good sample for S_i.Hence the inputs to the method in Section 5.4 will be R^i_j and S_i. Suppose $R^i_{j'}$ is one such good sample; with high probability, there will be such a j'.

(d) Use $R^i_{j'}$ to divide S_i into smaller subproblems.

(e) Recursively compute (in parallel) the Voronoi diagram of each subproblem.

(f) Obtain the final Voronoi diagram $Vor(S_i)$ from these recursively computed Voronoi diagrams.

2. Compute the total subproblems size when restricted to I (this is polling). Hence the inputs to the method in Section 5.4 will be S_i and I.

- Return the best S_i; with high probability there will be such an S_i.

Observe that in the above function, it will *not* be necessary to use polling in step 1(c) because of the smaller size of the S_i. The whole point of polling is to ensure that the processor bound of $O(n)$ is not exceeded. However, we can afford to use, for all $1 \leq i \leq d\log n$, the whole set S_i to determine the goodness of R^i_j $(1 \leq j \leq a\log n)$ because $(d\log n).(a\log n).N$ is $o(n)$ as long as $q \geq 3$.

We know that each of the $Vor(S_i)$ can be constructed in $O(\log n)$ time with very high probability (more accurately, we will know this for sure after the next two sections). But we want to be sure that *every one of the* $Vor(S_i)$ will be constructed in $O(\log n)$ time with high probability. This follows immediately from the fact that for events $A_1, A_2, \ldots$ (not necessarily disjoint), $Pr[\cup_i A_i] \leq \sum_i Pr[A_i]$ Suppose the probability that the construction of $Vor(S_i)$ takes more than β log$n$ steps is $\leq n^{-\alpha}$ for some constants α and β. Then it follows from the stated inequality that the probability that the construction of one or more of the $Vor(S_i)$ takes more than β log$n$ time is $\leq (n^{-\alpha}).(d\log n)$. Consequently, the probability that all the Voronoi diagrams $Vor(S_1)$, $Vor(S_2)$, $\ldots$, $Vor(S_{d\log n})$ are constructed in $O(\log n)$ time is $\geq (1-(n^{-\alpha}).(d\log n))$, which is very high.

It will be necessary to process the Voronoi diagram of the random sample appropriately so that we have a fast method to find the subproblems defined by it. Note that in our scheme it is imperative that we have an efficient algorithm to perform this processing. In other words, we cannot afford to have a parallel algorithm that uses a polynomial number of processors. Whereas in [22], since the sample size is always $O(n^\epsilon)$, they can choose an appropriate ϵ such that the processor bound of $O(n)$ is maintained, we do not have this flexibility. This is because of the large sample size during the first stage of sampling. In the following section, we give an efficient method that satisfies our requirement.

5.4 Finding $X(\mathcal{C}_R(e))$ for each $\mathcal{C}_R(e)$

Let R be a random sample chosen from a set $\mathcal{S}$, and let $|R| = r$. (Hence R could either be one of the S_i chosen from the original input S ($r = N$ and $\mathcal{S} = S$) or one of the R^i_j chosen from S_i ($r = N^\epsilon$

and $\mathcal{S} = S_i$).) Let us assume $Vor(R)$ is available to us; $Vor(R)$ would have been constructed either through a brute-force technique or the divide-and-conquer method outlined in the previous section. We now describe a method to process $Vor(R)$ appropriately so that we can find $X(\mathcal{C}_R(e))$ ($\subset \mathcal{S}$) for all Voronoi edges e of $Vor(R)$ in $O(\log n)$ time with high probability using $O(n)$ processors. Note that we obviously cannot sequentially determine all the subproblems that an element of $\mathcal{S}$ lies in because this could be as large as $O(r)$.

5.4.1 Processing $Vor(R)$ to Form the Search Data Structure $\mathsf{VorDS}(R)$

Let U be a triangulated subdivision with u vertices. A *subdivision hierarchy* is a sequence $U_1, U_2, \ldots, U_{h(u)}$ of triangulated subdivisions satisfying the following conditions: (1) $U_1 = U$, (2) $|U_{h(u)}| = 3$, and (3) each region of U_{i+1} intersects at most τ regions of U_i for some constant τ. The idea of using fractional independent sets[2] to build a subdivision hierarchy was first proposed by Kirkpatrick [17]. Subdivision hierarchies can be used to construct efficient search data structures. Randomized parallel algorithms to solve this problem efficiently were given independently by Dadoun and Kirkpatrick [9] and by Reif and Sen [23]. We can clearly use the same idea of fractional independent sets to build a hierarchy of Voronoi diagrams that will be very useful for our purposes.

The dual $\mathcal{DV}(R)$ of the Voronoi diagram $Vor(R)$ of a set of elements R is the graph which has a node for every element in R and an edge between two nodes if their corresponding Voronoi regions share a Voronoi edge (for the case when R is a set of points, the dual is the well-known Delaunay triangulation of the set of points). $\mathcal{DV}(R)$ is planar, and can be used to build a hierarchy of Voronoi diagrams. By a slight abuse of notation, for each segment $s \in R$, we will use s to refer to its corresponding node in the graph $\mathcal{DV}(R)$ as well (hence R will refer to the set of nodes when we talk about $\mathcal{DV}(R)$).

Observe that when R is a set of line segments in the plane, $\mathcal{DV}(R)$ could be a multigraph (i.e. there could be more than one edge between two nodes): this is because a bisector can be split into two or more pieces in $Vor(R)$. It can be shown that every interior face of $\mathcal{DV}(R)$ must be triangular. In a triangulated subdivision, the exterior face is assumed to be a triangle as well. If $\mathcal{DV}(R)$ does not have a triangular exterior face, we can assume that R has been augmented appropriately so that we have this property (this can be done in $O(\log r)$ time using r processors). Thus $\mathcal{DV}(R)$ is a planar subdivision in which the interior faces as well as the exterior face is triangular and we can use fractional independent sets to construct a hierarchy of Voronoi diagrams in the following manner.

In the remainder of this paper, we use $V_R(s)$ to denote the Voronoi region of $s \in R$ in the Voronoi diagram $Vor(R)$. Also, as in [17], we will assume that every node of a fractional independent set is an internal node $\mathcal{DV}(R)$, and the degree of each such node is less than or equal to τ, where τ is a fixed constant (for our purposes, it suffices to assume that $\tau = 6$). Let $R_1 = R$. Consider a fractional independent set W of R_1 in the graph $\mathcal{DV}(R_1)$, and let $R_2 = R_1 - W$. Let s be a node in W. Let $\Gamma^{R_1}(s)$ represent the set of neighbors of s in $\mathcal{DV}(R_1)$ (obviously $|\Gamma^{R_1}(s)| \leq \tau$). Since

[2]If $G = (V, E)$ is a planar graph, then a set $X \subseteq V$ is a *fractional independent set* of G if (1) the degree of each vertex $v \in X$ is less than or equal to some constant d (2) no two vertices of X are connected by an edge (i.e. the set X is an independent set) and (3) $|X| \geq c\,|V|$ for some fraction c.

W is an independent set, we know that $\Gamma^{R_1}(s) \subset R_2$. Then we have the following.

Lemma 5.3 *For an element $s' \in R_2$, $V_{R_2}(s')$ contains a part of the region $V_{R_1}(s)$ (i.e. $V_{R_2}(s') \cap V_{R_1}(s) \neq \emptyset$) if and only if $s' \in \Gamma^{R_1}(s)$.*

Proof: Obvious. □

It is now easy to construct *Vor*(R_2) from *Vor*(R_1). First, we know from [9, 23] that with very high probability, we can find such an independent set W in constant time using r processors (the idea is to use *randomized symmetry breaking*; see [15] for a clear exposition). Now, to determine the part of $V_{R_1}(s)$ that belongs to $V_{R_2}(s')$ for each $s' \in \Gamma^{R_1}(s)$, all we have to do is compute the intersection of the regions defined by the bisectors $B(s', s'')$ for all $s'' \in \Gamma^{R_1}(s)$. This can clearly be done in constant time. If we do this for all $s \in W$, it follows from Lemma 5.3 that the resulting diagram will be *Vor*(R_2). Thus we see that $\mathcal{DV}(R_2)$ can be obtained from $\mathcal{DV}(R_1)$ in constant time using r processors: For all $s \in W$, (a) delete the node s and the edges between s and members of $\Gamma^{R_1}(s)$ and (b) add edges between pairs of elements from $\Gamma^{R_1}(s)$ if they share a Voronoi edge in *Vor*(R_2) (there will be at most 3 such new edges for each s). Note that for each new prim of *Vor*(R_2), we can determine in constant time the prims of *Vor*(R_1) that it intersects.

We can repeat on R_2 the steps described above and obtain a new set R_3 and the graph $\mathcal{DV}(R_3)$, and we can continue the process until we have a set of size 3. In other words, we can build a hierarchy (analogous to the subdivision hierarchy of Kirkpatrick [17]) of Voronoi diagrams *Vor*(R_1), *Vor*(R_2), ..., *Vor*(R_h) where $h = O(\log r)$. This will take $O(\log r)$ time and r processors with high probability. We build the search data structure as we construct this hierarchy of Voronoi diagrams. The data structure is a directed acyclic graph $\mathsf{VorDS}(R)$ that contains a node for each primitive region in the diagrams *Vor*(R_i) ($1 \leq i \leq h$). Each prim of *Vor*(R_{i+1}) is a node in the $(i+1)$-th level of $\mathsf{VorDS}(R)$ and has a directed edge to those nodes at level i with which it has a non-empty intersection. Obviously the degree of each node in the data structure is less than or equal to r. Thus we have the following.

Lemma 5.4 *The search data structure $\mathsf{VorDS}(R)$ can be built in $O(\log r)$ time with very high probability using r processors, where $r = |R|$.*

5.4.2 Searching $\mathsf{VorDS}(R)$ to find $X(\mathcal{C}_R(e))$ for all $\mathcal{C}_R(e)$

Once $\mathsf{VorDS}(R)$ has been constructed, we want to search it efficiently in order to find the input set to each subproblem determined by R i.e. $X(\mathcal{C}_R(e))$ for $\mathcal{C}_R(e) \in \mathcal{C}_R$. Essentially, each element $s \in \mathcal{S}$ searches through $\mathsf{VorDS}(R)$ to determine all the subproblems that it lies in, and this is done in parallel for all such s. Note that this is also the step that allows us to determine if R is a good sample or not. (Hence in our algorithm, $\mathcal{S}$ may sometimes be a polling set that is a subset of the original input S.) Let $| \mathcal{S} | = \mathcal{N}$. We first state the basic idea of how the search proceeds and omitting some important details; a sketch of the details follows.

For each element $s \in \mathcal{S}$, we start off the search by determining the prims at level h that $V^{R_h}(s)$

intersects[3]. Obviously $V^{R_h}(s)$ is of constant size and hence this step can be carried out in constant time using $O(\mathcal{N})$ processors. Our algorithm works in phases. We describe the steps during one phase: Suppose that for $s \in \mathcal{S}$, a processor Π has reached the node corresponding to prim $\mathcal{P}$ at level $(i+1)$ of $\mathsf{VorDS}(R)$. As we know from the previous section, $\mathcal{P}$ intersects a constant number of prims at level i. Let these be called $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_\rho$ (note that ρ will be less than or equal to τ, the maximum degree of a node in $\mathsf{VorDS}(R)$). Without loss of generality, let $s_1, s_2, \ldots, s_\rho$ be the elements of R_i such that for $1 \leq j \leq \rho$ $\mathcal{P}_j$ belongs to $V_{R_i}(s_j)$, respectively. If $B(s, s_j)$ intersects prim $\mathcal{P}_j$, then Π generates a request for a new processor in the following manner: it randomly generates a processor label and writes into that processor to indicate that it is being requested. There could be write conflicts during this step and an arbitrary processor will succeed (in the case of a success, the new processor will need to know $\mathcal{P}_j$ and s so as to continue the search from the node corresponding to $\mathcal{P}_j$). Obviously Π can carry out this step in constant time (ρ steps). In order to be sure that this scheme works, we need to know that for each $s \in \mathcal{S}$, every prim that should be reached at level i is reachable from level $(i+1)$ in the above described manner. This follows from the lemma given below (we omit the proof, which is quite straightforward).

Lemma 5.5 *If $V^{R_i}(s)$ intersects a prim $\mathcal{P}$ at level i, then $V^{R_{i+1}}(s)$ must intersect at least one of the prims that $\mathcal{P}$ intersects at level $(i+1)$ of $\mathsf{VorDS}(R)$.*

Proof: Omitted. □

We described one phase of the algorithm above. During the next phase, another set of requests are generated in a similar manner (including the ones that weren't satisfied in the previous phase), and so on. It follows from the above lemma that by searching through $\mathsf{VorDS}(R)$ in the described manner, we can find $V^R(s)$ and hence all the subproblems that s lies in.

There is an important fact that we have ignored so far. In order to carry out this search step efficiently, we want the total number of requests to be $O(\mathcal{N}\log\mathcal{N})$, and the method we have just described does not guarantee that because of the following. For $s \in \mathcal{S}$, let $Y^{R_i}(s)$ $(1 \leq i \leq h)$ be the set of prims in *Vor*(R_i) that $V^{R_i}(s)$ intersects. The prims in $Y^{R_1}(s)$ $(= Y^R(s))$ define all the subproblems that s lies in. Now, in order to carry out the search efficiently, we need to make sure that for good samples the amount of work done for s (i.e. the number of requests made for s) at each level of $\mathsf{VorDS}(R)$ is bounded above by $\sigma|Y^R(s)|$ for some positive constant σ. If we can show this, then we know that the total number of requests over all h levels of the data structure will be $O(\mathcal{N}\log\mathcal{N})$ for a good sample. We device a technique to achieve this. We will not go into the details, for lack of space and appeal to intuition instead: The problem with sending s to every prim of $Y^{R_i}(s)$ at every level i is that at some levels $|Y^{R_i}(s)|$ might be larger than the final $|Y^R(s)|$. In other words, there may be levels i such that the number of prims in $Y^{R_i}(s)$ could be less than the number of prims in $Y^{R_{i+1}}(s)$. Note that in this case, we would like to have a way to ignore these "extra prims" in $Y^{R_{i+1}}(s)$, since we know they are not going to contribute to the final $Y^R(s)$. We device a technique so that for any $s \in \mathcal{S}$, the number of prims that s looks at increases (or rather, does not decrease) from level $(i+1)$ to level i. Let $\bar{Y}_{R_i}(s)$ be the set of prims at level i that

[3] Recall that we use $V^R(s)$ to denote the region that is closer to s than to any other element in R i.e. the Voronoi region of s in the Voronoi diagram of $R \cup \{s\}$.

s actually looks at, $\bar{Y}_{R_i}(s) \subset Y^{R_i}(s)$. Then $|\bar{Y}_{R_i}(s)|$ monotonically increases as s is percolated from $i = h$ to $i = 1$. We do this by doing a form of look-ahead in $\mathsf{VorDS}(R)$. Thus we can show that for a good sample, the total number of requests over all levels of $\mathsf{VorDS}(R)$ will be $O(\mathcal{N}\log\mathcal{N})$.

The search through this data structure is carried out by the dynamic allocation of processors to each segment $s \in \mathcal{S}$. The idea is to use a number of processors proportional to $|Y^{R_1}(s)|$ for s. We use randomized techniques for dynamic processor allocation, as given by Reif and Sen [21]. We mentioned above that write conflicts (when requesting a processor) are resolved arbitrarily. Assume that the probability of success in such a case is 1/2. Also, since the total subproblems size over all levels of $\mathsf{VorDS}(R)$ is $O(\mathcal{N}\log\mathcal{N})$ (for good samples), this will be the total number of requests for processors. It is shown in [21] that for such a randomized allocation technique, all the requests will be satisfied in $O(\log\mathcal{N})$ phases with very high likelihood using $O(\mathcal{N})$ processors. If the search through $\mathsf{VorDS}(R)$ is incomplete after these $O(\log\mathcal{N})$ phases (i.e. there are requests that haven't been satisified), then we can reject R with high confidence.

Note that since the data structure that we are searching through is a directed acyclic graph, it is possible that an $s \in \mathcal{S}$ may arrive at the same node through two different paths. In such instances, we want to avoid repeating searches that have already commenced. We would like to use $O(\mathcal{N}\log\mathcal{N})$ space in order to solve this problem; we can use randomized hashing techniques to achieve this goal. This will use $O(\mathcal{N})$ space for each level of $\mathsf{VorDS}(R)$. From Gil, Matias and Vishkin [11], we know that we can certainly do this in $O(\log^*\mathcal{N})$ time (nearly constant) with high probability by doing an optimal amount of work at each phase. But we show that over all the levels of the data structure, the total processing time is $O(\log\mathcal{N})$ with high probability (omitted for lack of space). Thus we have the following.

Lemma 5.6 *We can find $X(\mathcal{C}_R(e))$ for all $\mathcal{C}_R(e) \in \mathcal{C}_R$ in $\tilde{O}(\log\mathcal{N})$ time using $O(\mathcal{N})$ processors and $\tilde{O}(\mathcal{N}\log\mathcal{N})$ space, where R is a random sample of S and $|S| = \mathcal{N}$.*

5.5 Merging Recursively Computed Voronoi Diagrams

In the previous section, we described the method to find the subproblems determined by a random sample from the input set $\mathcal{S}$. So assume now that for a *good* sample R, $X(\mathcal{C}_R(e))$ has been found for all $\mathcal{C}_R(e) \in \mathcal{C}_R$. The diagrams *Vor*$(X(\mathcal{C}_R(e)))$ will be computed recursively, in parallel for each $\mathcal{C}_R(e)$. We want to merge these Voronoi diagrams to form *Vor*$(\mathcal{S})$. In the remainder of the section, we describe the method to perform this merge step efficiently.

The final Voronoi diagram of $\mathcal{S}$ can be constructed by computing the final Voronoi region $V(s)$ for each element $s \in \mathcal{S}$. $V(s)$ can be determined by finding the *intersection* of all the Voronoi regions of s that have been computed recursively in the subproblems. As before, let $|R| = r$ and let $|\mathcal{S}| = \mathcal{N}$. Since R is a good sample of $\mathcal{S}$, we know that the total size of the subproblems is less than or equal to $k_{tot}\,\mathcal{N}$. It follows therefore that the *total* number of Voronoi edges in *all* the recursively computed Voronoi diagrams must be less than $3.k_{tot}\,\mathcal{N}$ (since the number of Voronoi edges in *Vor*$(X(\mathcal{C}_R(e)))$ is less than $3.|X(\mathcal{C}_R(e))|$).

Consider the set of subproblems in which $s \in \mathcal{S}$ lies. Let $\Upsilon_R(s)$ be the set of recursively com-

puted Voronoi regions of s in these subproblems. That is[4], $\Upsilon_R(s) = \{V_{X(\mathcal{C}_R(e))}(s) \mid X(\mathcal{C}_R(e) \in \mathcal{C}_R\}$ The Voronoi regions will be represented as the collection of their Voronoi edges. Note that the total size of all the $\Upsilon_R(s)$ for all $s \in \mathcal{S}$ will also be $O(\mathcal{N})$ since every Voronoi edge is counted exactly two times over the regions $\Upsilon_R(s)$ for all $s \in \mathcal{S}$ (once for each of the two Voronoi regions that it borders). Every Voronoi edge in $\Upsilon_R(s)$ is part of some bisector $B(s, s')$ ($s' \in \mathcal{S}$). In the case of line segments, computing the intersection of these bisectors to find $V(s)$ is not an efficient strategy because of the fact that two such bisectors can intersect twice (see Lemma 4.4). The following lemma provides us with a solution.

Lemma 5.7 *Every Voronoi vertex of the final Voronoi region $V(s)$ appears as a vertex in at least one of the Voronoi regions in the set $\Upsilon_R(s)$.*

Proof: Let v be a Voronoi vertex that is part of the final region $V(s)$. v must lie in some primitive region $\mathcal{P}$ of *Vor*(R). v is equidistant from three elements of $\mathcal{S}$, one of them being s; let the other two be s' and s''. Moreover, v is closer to these three elements than it is to any other element in $\mathcal{S}$. Obviously then, v must lie in $V^R(s)$, $V^R(s')$ and $V^R(s'')$ and hence all three of these regions must intersect prim $\mathcal{P}$. Let e' be the Voronoi edge of *Vor*(R) that borders prim $\mathcal{P}$. Then all three of s, s' and s'' must belong to $X(\mathcal{C}_R(e'))$. Clearly then, v will be a part of the diagram *Vor*$(X(\mathcal{C}_R(e')))$ since v is closer to s, s' and s'' than to any other element of $\mathcal{S}$. In particular, v will be part of the Voronoi region $V_{X(\mathcal{C}_R(e'))}(s)$, which proves our claim. □

Thus we just have to search the set of vertices that appear in the Voronoi regions in $\Upsilon_R(s)$ and eliminate those vertices that are not part of $V(s)$. We do this as follows. The vertices of any Voronoi region of an element s have well-defined sorted order defined by the projection of the vertices on s (this is given as the function *proj* in Section 4). Assume that the vertices appear in clockwise order (with respect to some axis that is fixed on s). We merge in parallel the Voronoi vertices of the regions in $\Upsilon_R(s)$ according to this ordering by using the cascading divide-and-conquer technique that sorts optimally [Cole [7],Atallah, Cole and Goodrich [3]], until finally we are left with the vertices of $V(s)$. Each leaf of the merge tree will contain a Voronoi vertex from the regions in $\Upsilon_R(s)$ along with the Voronoi edge to its right (for some unbounded edges, we will need to create a "dummy" vertex at ∞). Let y_s be the total number of Voronoi *edges* in $\Upsilon_R(s)$.

We say that a Voronoi vertex v is *hidden* by a Voronoi edge e if the segment from v to $proj(v, s)$ intersects e and v does not lie on e. In this case, this point of intersection on e will be denoted by $int(v, e)$. We give a crude description of the steps and omit details. Let $L(\kappa)$ be the list of vertices at a node κ of the merge tree at phase i of the cascaded merging. The idea is to maintain at κ a list of vertices $VV(\kappa)$ that consists of vertices that are not hidden by any of the Voronoi edges encountered at κ until phase i. Every time a merge step takes place at κ, $VV(\kappa)$ is updated. If a vertex v from $L(\kappa)$ is hidden by an edge e that just arrived at κ, we replace v by $int(v, e)$ in $VV(\kappa)$ and mark this vertex as spurious. Because cross-ranks[5] between sibling nodes are known,

[4] Recall that we use $V_A(a)$ to denote the Voronoi region of $a \in A$ in the diagram *Vor*(A).

[5] Let $A = \{a_1, a_2, \ldots, a_n\}$ and $B = \{b_1, b_2, \ldots, b_m\}$ be two sorted lists. The *rank* of $a_i \in A$ in B, denoted *rank*$(a_i : B)$, is the number of elements of B that are less than or equal to a_i. The rank of A in B *rank*$(A : B)$ is the array of ranks $(r_1, r_2, \ldots, r_n)$ such that $r_i =$ *rank*$(a_i : B)$. *rank*$(B : A)$ is similarly defined. A and B are said to be *cross-ranked* if we know *rank*$(A : B)$ and *rank*$(B : A)$.

the above computation takes constant time at each phase i. Hence the run-time of the algorithm will $O(\log\ y_s)$ using a total of $O(y_s)$ processors. We can now do a prefix scan operation on the list VV at the root to eliminate those vertices that were marked as spurious. Observe that from Lemma 5.7 we know we do not have to worry about intersections between Voronoi edges. In other words, even if such an intersection is a Voronoi vertex v in the final region $V(s)$, we can ignore the intersection because Lemma 5.7 guarantees that we will find v as a vertex in one of the regions of $\Upsilon_R(s)$. We carry out this step in parallel for all $s \in \mathcal{S}$ and since $\sum_{s \in \mathcal{S}} y_s$ is $O(\mathcal{N})$, we have the following.

Lemma 5.8 *The Voronoi diagram* $\mathrm{Vor}(\mathcal{S})$ *can be constructed from the recursively computed Voronoi diagrams* $\mathrm{Vor}(X(\mathcal{C}_R(e)))$ *(*$\mathcal{C}_R(e) \in \mathcal{C}_R$*) in* $O(\log \mathcal{N})$ *time using* $O(\mathcal{N})$ *processors.*

6 Conclusions and Applications

It follows from the previous section that we have an optimal algorithm for the Voronoi diagram of a set of line segments in the plane. We state this below.

Theorem 6.1 *The Voronoi diagram of a set of* n *non-intersecting line segments in the plane can be computed in* $\tilde{O}(\log n)$ *time using* $O(n)$ *processors and* $\tilde{O}(n \log n)$ *space.*

The above theorem immediately gives us optimal parallel randomized algorithms for computing the minimum weight spanning tree and the all-nearest neighbor for the set of segments. We also obtain an optimal parallel algorithm to plan the motion of an object from one point to another while avoiding polygonal obstacles (see [20] for details).

Clarkson and Shor's [6] results established the effectiveness of random sampling in deriving better expected bounds for computational geometry problems and Reif and Sen [22] demonstrate how to obtain high probability results from expected bounds, which is crucial for parallel algorithms. Our result offers more evidence of the usefulness of randomization in obtaining more efficient algorithms. It will be interesting to see if our technique can be applied to other problems to obtain more efficient results and perhaps less complicated algorithms since we believe that our approach is simpler and more general. For instance, we think that the 3-d convex hull algorithm given by Reif and Sen [22] could be simplified, especially since the surface of a convex polyhedron is topologically equivalent to a bounded planar subdivision.

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. K. Yap. Parallel Computational Geometry. *Algorithmica*, 3:293–327, 1988.

[2] N. M. Amato and F. P. Preparata. An NC^1 Parallel 3D Convex Hull Algorithm. In *Proc. 9th ACM Symp. on Computational Geometry*, 1993.

[3] M. J. Atallah, R. Cole, and M. T. Goodrich. Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms. *SIAM J. Comput.*, 18(3):499–532, June 1989.

[4] M. J. Atallah and M. T. Goodrich. Efficient Parallel Solutions to Geometric Problems. In *Proc. 1985 IEEE Conf. on Parallel Processing*, pages 411–417, 1985.

[5] A. Chow. *Parallel Algorithms for Geometric Problems.* PhD thesis, University of Illinois at Urbana-Champaign, 1980.

[6] K. L. Clarkson and P. W. Shor. Applications of Random Sampling in Computational Geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.

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

[8] R. Cole and M. T. Goodrich. Optimal Parallel Algorithms for Polygon and Point-set Problems. *Algorithmica*, 7:3–23, 1992.

[9] N. Dadoun and D. Kirkpatrick. Parallel Processing for Efficient Subdivision Search. In *Proc. Third ACM Symp. on Computational Geometry*, pages 205–214, 1987.

[10] S. Fortune. A Sweepline Algorithm for Voronoi Diagrams. In *Proc. 2nd ACM Symp. on Computational Geometry*, pages 313–322, 1986.

[11] J. Gil, Y. Matias, and U. Vishkin. Towards a Theory of Nearly Constant Time Parallel Algorithms. In *Proc. of Symp. on FOCS*, 1992.

[12] M. T. Goodrich. Geometric Partitioning Made Easier, Even in Parallel. In *Proc. 9th ACM Symp. on Computational Geometry*, 1993.

[13] M. T. Goodrich, C. Ó'Dúnlaing, and C. K. Yap. Constructing the Voronoi Diagram of a Set of Line Segments in Parallel. In *Lecture Notes in Computer Science: 382, Algorithms and Data Structures, WADS*, pages 12–23. Springer-Verlag, 1989.

[14] D. Haussler and E. Welzl. ϵ-nets and Simplex Range Queries. *Discrete and Computational Geometry*, 2:127–152, 1987.

[15] J. JáJá. *An Introduction to Parallel Algorithms*, chapter 9, pages 441–450. Addison-Wesley Publishing Company, 1992.

[16] D. G. Kirkpatrick. Efficient Computation of Continuous Skeletons. In *Proc. 20th IEEE Symp. on Foundations of Computer Science*, pages 18–27, 1979.

[17] D. G. Kirkpatrick. Optimal Search in Planar Subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.

[18] D. T. Lee and R. L. Drysdale. Generalization of Voronoi Diagrams in the Plane. *SIAM J. Comput.*, 10(1):73–87, February 1981.

[19] K. Mulmuley. A Fast Planar Partition Algorithm. In *Proc. 20th IEEE Symp. on the Foundations of Computer Science*, pages 580–589, 1988.

[20] C. Ó'Dúnlaing and C. K. Yap. A 'Retraction' Method for Planning the Motion of a Disc. *J. Algorithms*, 6:104–111, 1985.

[21] J. H. Reif and S. Sen. Polling: A New Randomized Sampling Technique for Computational Geometry. Manuscript.

[22] J. H. Reif and S. Sen. Optimal Parallel Randomized Algorithms for Three Dimensional Convex Hulls and Related Problems. *SIAM J. Comput.*, 21(3):466–485, 1992.

[23] J. H. Reif and S. Sen. Optimal Randomized Parallel Algorithms for Computational Geometry. *Algorithmica*, 7:91–117, 1992.

[24] C. K. Yap. An $O(n \log n)$ Algorithm for the Voronoi Diagram of a Set of Simple Curve Segments. *Discrete and Computational Geometry*, 2:365–393, 1987.