Algorithmica (1993) 9: 128–141

Algorithmica
© 1993 Springer-Verlag New York Inc.

Constructing the Voronoi Diagram of a Set of Line Segments in Parallel[1]

Michael T. Goodrich,[2] Colm Ó'Dúnlaing,[3] and Chee K. Yap[4]

Abstract. In this paper we give a parallel algorithm for constructing the Voronoi diagram of a polygonal scene, i.e., a set of line segments in the plane such that no two segments intersect except possibly at their endpoints. Our algorithm runs in $O(\log^2 n)$ time using $O(n)$ processors in the CREW PRAM model.

Key Words. Voronoi diagram, PRAM, Parallel algorithm.

1. Introduction. Ever since the pioneering work of Shamos [18], the Voronoi diagram has been recognized as a highly versatile geometric structure in computational geometry, being the key to the efficient solution of a host of different problems. In [18] Shamos shows how to construct the Voronoi diagram of a set of n points in the plane in $O(n \log n)$ time, and how this structure can be used to solve a number of proximity problems in this same bound. Given a set S of points in the plane, the Voronoi diagram defines a region for each point p in S such that all the points in that region are closer to p than to any other point in S.

In this paper we address an important generalization of the Voronoi diagram, namely to the case when the underlying objects are either line segments or points. This structure has been called the *continuous skeleton* [10] of the segments and, for the case when the segments form a simple polygon, the *medial axis* [5], [13], [17]. One of the first efficient methods for constructing the Voronoi diagram of a set of line segments is an algorithm of Lee and Drysdale [14], which runs in $O(n \log^2 n)$ time. Subsequently, a number of researchers have shown that this can be improved to $\Theta(n \log n)$ (e.g., Fortune [8], Kirkpatrick [10], and Yap [21]).

We are interested in solving this problem in parallel. This is motivated by the fact that Voronoi diagram construction cannot be made faster by anything more than a constant factor over the previous methods without using more than a single processor, due to known lower bounds [18]. Specifically, the goal of this research is to find an algorithm that runs as fast as possible and is *efficient* in the following

[1] The research of M. T. Goodrich was supported by NSF under Grants CCR-8810568 and CCR-9003299 and by NSF/DARPA under Grant CCR-8908092. C. K. Yap's research was supported in part by NSF Grants DCR-8401898 and CCR-9002819.

[2] Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218, USA.
[3] School of Mathematics, University of Dublin, Dublin 2, Irish Republic.
[4] Courant Institute, New York University, New York, NY 10012, USA.

Received October 1, 1989; revised February 21, 1990. Communicated by Bernard Chazelle.

sense: in $P(n)$ is its processor complexity, $T(n)$ is its time complexity, and $Seq(n)$ is the time complexity of the best-known sequential algorithm, then $T(n) * P(n) = O(Seq(n) \log^k n)$ for some small constant k [9].

There is no previous parallel algorithm for constructing the Voronoi diagram of a collection of points and line segments, although there has been work done on the case when the Voronoi sites are only points. In particular, Aggarwal *et al.* [1] show how the Voronoi diagram of a set of points in the plane may be constructed in $O(\log^2 n)$ time using $O(n)$ processors in the CREW PRAM model. More recently, Cole *et al.* [7] improve this to $O(\log^2 n)$ time using $O(n/\log n)$ processors. Neither of these methods seem to generalize easily to construct the Voronoi diagram of a set of line segments, however.

The traditional approach to Voronoi diagram construction is to divide the set of objects into two equal-sized subsets S_1 and S_2 whose separate Voronoi diagrams can be constructed recursively, and then merge the two diagrams into one. The essential computation needed to merge the two recursively constructed Voronoi diagrams is the construction of the *contour* between S_1 and S_2, i.e., the locus of all points that are equidistant from S_1 and S_2. This divide-and-conquer approach is the one used by Kirkpatrick [10], Lee and Drysdale [14], and Yap [21], for example. These algorithms differ, however, in how they perform the division. The method of Lee and Drysdale, as well as that of Kirkpatrick, divide the segments so that S_1 and S_2 are disjoint and no segment is cut in two. Yap, on the other hand, divides S into S_1 and S_2 by a vertical line L that conceptually "cuts" each segment it crosses into two pieces. By avoiding unnecessary computations, Yap's algorithm avoids the $O(n^2)$-time behavior that this approach could exhibit. In all of these algorithms the method for constructing the contour is based on a search procedure to find "seed points" on the contour, followed by a method that traces out the contour starting with the seed points. Although quite elegant in the sequential setting, these contour-tracing methods seem of little worth in the parallel setting.

Another interesting approach to the problem of constructing the Voronoi diagram of a set of line segments is demonstrated by the algorithm by Fortune [8]. His algorithm uses the well-known "plane-sweeping" technique, where the plane is conceptually swept with a line L. As L moves across the plane, his algorithm constructs the Voronoi diagram a short distance behind it. Unfortunately, as with the traditional approaches, this approach is quite elegant in the sequential setting, but appears difficult to translate into an efficient parallel method.

In this paper we show how to construct the Voronoi diagram of a set of line segments in the plane in $O(\log^2 n)$ time using $O(n)$ processors in the CREW PRAM model. Recall that this is the synchronous shared-memory model where processors can simultaneously access the same memory cell only if they are all trying to read from it. Thus, our algorithm is efficient, and matches the complexity bounds of the point-set Voronoi diagram algorithm of Aggarwal, *et al.* [1].

Our method is based on the approaches used by Yap [21] and Aggarwal *et al.* [1]. Our procedure avoids the sequential bottlenecks of Yap's method, and is

actually simpler than that of Aggarwal *et al.* This is due primarily to our emphasis on "primitive regions" as being the basic objects in the subproblem merge procedure, rather than Voronoi edges, and our use of a method of Atallah *et al.* [2] to perform certain point-location queries. In addition, our method avoids the assumption that the only interest is in constructing the Voronoi diagram clipped to some "viewing rectangle," as was assumed by Kirkpatrick [10] and Aggarwal *et al.* [1].

In the next section we review some known geometric properties of Voronoi diagrams. In the subsequent section we give our algorithm for constructing this diagram, and we give some applications in Section 4.

2. Geometric Preliminaries. In this section we introduce the definitions, notations, and approaches we use in this paper. Let S be a set of line segments in the plane. As in [10] and [21] we consider any line segment in S to consist actually of three distinct objects: the segment's two endpoints and the segment minus its endpoints (i.e., an open segment). Thus, our *objects* are open segments and points. The *projection* of a point p onto an object s, denoted $proj(p, s)$, is defined to be a point q in the closure $\bar{s}$ of s such that the Euclidean distance $d(p, q)$ is minimized [14]. Note that the projection of a point onto an object is unique, since each of our primitive objects s is either an open line segments or a point (and if s is a point, then $proj(p, s) = s$). Thus, we can extend our distance metric by defining the *distance* from a point p to an object s, denoted $d(p, s)$, to be $d(p, proj(p, s))$. Given two objects s_1 and s_2, this also allows us to define the *bisector* of s_1 and s_2, denoted $B(s_1, s_2)$, to be the locus of all points that are equidistant from s_1 and s_2. Since our primitive objects are points and open segments, the bisector of two objects is made up of portions of line segments and parabolic arcs, which may be infinite or semi-infinite. By a mild abuse of notation, we further extend our distance metric to define the *distance* a point p to a set of objects S to be $\min_{s \in S}\{d(p, s)\}$, and use $d(p, S)$ to denote this quantity.

As in [21] we say that a set of objects S is *proper* if for every open segment s in S the endpoints of s are also in S and no two segments in S cross each other, i.e., the segments can intersect only at their endpoints. All sets of objects in this paper are proper, so suppose henceforth that S is such a proper set. There are a number of ways the *Voronoi diagram* of S can be defined [10], [14], [21] We define it to be the subdivision of the plane produced by the locus of all points p such that $d(p, S)$ is realized by at least two objects in S. The Voronoi diagram of S contains two-dimensional objects, called *Voronoi cells*, one-dimensional objects, called *Voronoi edges*, and zero-dimensional objects, called *Voronoi vertices*. Each Voronoi cell consists of all points p such that $d(p, S)$ is realized by exactly one object $s \in S$; each Voronoi edge consists of all points such that $d(p, S)$ is realized by exactly two objects; and each Voronoi vertex is a point such that $d(p, S)$ is realized by at least three objects. It is a trivial observation to note that each Voronoi edge is a portion of a bisector of two objects, and that each Voronoi vertex is the intersection of three or more Voronoi edges. We denote the Voronoi diagram of S by $Vor(S)$. Figure 1 illustrates the Voronoi diagram of a proper set

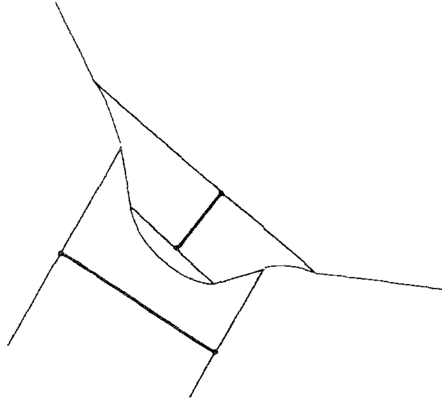

Fig. 1. The Voronoi diagram of two segments.

of two segments. In the following lemmas we review some of the well-known properties of *Vor*(*S*).

LEMMA 2.1 [14]. *Let* $V(s)$ *be the cell in* $Vor(S)$ *for an object* s *in* S. *Then, for any point* p *in the closure* $\overline{V(s)}$ *of* $V(s)$, *the closed line segment from* p *to* $proj(p, s)$ *is completely contained in* $\overline{V(s)}$. *In addition, for any two points* p_1 *and* p_2 *in* $\overline{V(s)}$, *the segment from* p_1 *to* $proj(p_1, s)$ *and the segment from* p_{21} *to* $proj(p_2, s)$ *do not cross.*

We refer to the previous lemma as defining the skeleton property of *Vor*(*S*). It generalizes the convexity property obtained should all the objects be points. The next lemma establishes another convexity property for *Vor*(*S*): namely, the relationship between *Vor*(*S*) and *CH*(*S*), the convex hull of S (the smallest convex set containing all the objects in S).

LEMMA 2.2 [14]. *An object* s *in* S *is on the boundary of* $CH(S)$ *of* S *if and only if the Voronoi cell* $V(s)$ *in* $Vor(S)$ *is unbounded.*

The next lemma gives us an important upper bound on the size of *Vor*(*S*).

LEMMA 2.3 [14]. *The number of Voronoi cells, Voronoi edges, and Voronoi vertices in* $Vor(S)$ *is* $O(n)$, *where* n *is the number of segments in* S.

Having reviewed some of the important properties of the Voronoi diagram of a set of line segments, let us present our method for its construction.

3. Constructing the Voronoi Diagram in Parallel.

Let a proper set S of n objects (points and line segments) in the plane be given. In this section we give the details of our method for constructing the Voronoi diagram of S.

We begin our construction by placing a vertical dividing line through each point

object. This divides the plane into at most n regions, which we call *slabs*. Let T be a complete binary tree whose leaves are associated, one per leaf, with these slabs, listed from left to right (in both the tree and the plane). With each internal node v of T we associate the slab $\mathcal{U}_v$ that is the union of the slabs associated with descendants v, where we regard the $\mathcal{U}_v$'s as closed subsets of the plane, i.e., two adjacent $\mathcal{U}_v$'s share a common boundary. We can perform the construction of T in $O(\log n)$ time using $O(n)$ processors by using the sorting algorithm of Cole [6] to order the segment endpoints and then using a simple recursive-doubling procedure [20] to construct T.

Intuitively, the tree T provides the schematic for the divide-and-conquer procedure, where the generic problem we wish to solve is the following. Let v be a note in T, and let S_v denote the set of all the objects of S clipped to $\mathcal{U}_v$, i.e., the set $\{s \cap \mathcal{U}_v | s \in S \text{ and } s \cap \mathcal{U}_v \neq \varnothing\}$. By an abuse of notation, we write $S \cap \mathcal{U}_v$. We say that a segment s *spans* $\mathcal{U}_v$ if the closure of s intersects both vertical boundaries of $\mathcal{U}_v$. Consider all the segments of S_v that span $\mathcal{U}_v$. They partition $\mathcal{U}_v$ into a collection of closed regions, called quads (since, for any slab bounded by two vertical lines, all but the topmost and bottommost regions are quadrilaterals). A quad $\mathcal{Q}$ in $\mathcal{U}_v$ is *active* if $\mathcal{Q}$ contains any segment of S_v that is not part of the boundary of $\mathcal{Q}$. The generic problem is to compute, for each active quad $\mathcal{Q}$ in $\mathcal{U}_v$, a collection of diagrams that contain the Voronoi diagram $Vor(S_v \cap \mathcal{Q})$. We let $VorSet(S_v)$ denote this collection. We ignore the inactive quads in $S_v \cap \mathcal{U}_v$, since their Voronoi diagrams are easily computed as needed in $O(1)$ time each. Note that $VorSet(S) = \{Vor(S_{root(R)})\}$, since there is only one active quad in $\mathcal{U}_{root((T)}$ and it is the entire plane.

Even though S_v can contain $O(n)$ segments, in order for this approach to result in an efficient parallel algorithm we must be able to construct $VorSet(S_v)$ quickly using only $O(n_v)$ processors, where n_v is the number of leaf descendants of v. This amounts to the parallel analogue of Yap's notion of only performing the "necessary" constructions.

We give an overview of our method for constructing $VorSet(S_v)$ below. The procedure is invoked by calling it to construct $VorSet(S_{root(T)})$.

THE $VorSet(S_v)$ CONSTRUCTION PROCEDURE (High-Level Description).

Step 0: Preprocessing. In this step we construct the tree T as described above, including, for each leaf v in T, the construction of a list $End(v)$, which is the list of all the segments of S whose closure has an endpoint in $\mathcal{U}_v$. We also construct a data structure D, which, given a point p in $\Re^2$ and a node v in T, allows a single processor to find the two segments bounding the quad in $\mathcal{U}_v$ that contains p in $O(\log n)$ time. We only perform this step the first time the procedure is invoked. This step can be implemented in $O(\log n)$ time using $O(n)$ processors by a method due to Atallah *et al.* [2].

Step 1: Recursive Call. If v is a leaf node in T, then we return immediately, for we have already computed $End(v)$, and $\mathcal{U}_v$ does not have any active quads; hence, $VorSet(S_v)$ is empty. Otherwise, we construct $VorSet(S_x)$, $End(x)$, $VorSet(S_y)$, and $End(y)$ recursively in parallel, where x is the left child of v and y is the right

child of v. Complexity: $T(n_v/2)$ time using $2P(n_v/2)$ processors, where T and P are functions characterizing the running time and the number of processors of algorithm, respectively.

Comment: The remaining steps of this procedure comprise the subproblem-merge part of our divide-and-conquer algorithm.

Step 2: Determining active quads in $\mathcal{U}_v$. In this step we use the data structure D to determine all the active quads in $\mathcal{U}_v$. The essential computation in this step is to perform a search in D for each endpoint in $\mathcal{U}_v$ of a segment in *End(v)*. *Complexity:* $O(\log n)$ using $O(n)$ processors.

Comment: For the remainder of this procedure we concentrate on the computations that need to be performed for a particular active quad $\mathcal{Q}$ in $\mathcal{U}_v$. These steps are to be performed for each such $\mathcal{Q}$ in parallel.

Step 3: The Vertical Merge. Note that an active quad $\mathcal{Q}$ of $\mathcal{U}$ is composed of a contiguous series of quads of $\mathcal{U}_x$ (none of which was necessarily active), together with a similar series of quads of $\mathcal{U}_y$. Let $\mathcal{Q}_L$ (resp. $\mathcal{Q}_R$) denote the union of the quads in $\mathcal{U}_x$ (resp. $\mathcal{U}_y$) contained in $\mathcal{Q}$. In this step we construct the Voronoi diagrams $V_L = Vor(S_v \cap \mathcal{Q}_L)$ and $V_R = Vor(S_v \cap \mathcal{Q}_R)$. Complexity: $O(\log n)$ time using $O(n_v)$ processors (total, for all $\mathcal{Q}$'s).

Step 4: The Horizontal Merge. In this step we merge V_L and V_R into the Voronoi diagram of the segments in $\mathcal{Q}$. This step is the most important step in the construction, and is implemented through the use of a number of subprocedures. In this spirit of [1], [10], and [21] we divide each cell $V(s)$ of V_L into *primitive regions*, or *prims* for short, by adding edges, called *spokes* [10], from s to the Voronoi edges of $V(s)$. We construct point-location data structures for V_L and V_R, and use these data structures to determine which prims of V_L and V_R intersect the contour between V_L and V_R. This then allows us to construct the contour from these prims. Once the contour is constructed, we merge the part of V_L left of the contour, the contour itself, and the part of V_R right of the contour to give us the Voronoi diagram of the segments in $\mathcal{Q}$. Complexity: $O(\log n)$ time using $O(n_v)$ processors (total, for all $\mathcal{Q}$'s).

End of High-Level Description.

Assuming that we can perform each of the above steps correctly in the stated time and processor bounds, then the total running time of the construction is characterized by the recurrence relation $T(n_v) = T(n_v/2) + c \log n$ for some constant c. This implies that $T(n)$ is $O(\log^2 n)$. Similarly, the number of processors is characterized by the relation $P(n) = \max\{2P(n/2), dn\}$, for some constant d, which has the solution $P(n) = O(n)$.

In the sections that follow we give the details of each of the steps above.

We begin with Step 2.

3.1. Step 2: Determining Active Quads in $\mathcal{U}_v$. Let us assume recursively that we have constructed $VorSet(S_x)$, $End(x)$, $VorSet(S_y)$, and $End(y)$, where x is the left child of v and y is the right child of v. In addition, we assume that for each segment s in $End(x)$ (resp. $End(y)$) we have a pointer from s to the name of the Voronoi

cell in $VorSet(S_x)$ (resp. $VorSet(S_y)$) that contains s. In this step we identify all the active quads in $\mathcal{U}_v$. We begin by constructing $End(v) = End(x) \cup End(y)$. For each endpoint p of the closure of a segment in $End(v)$, *if* p is inside the slab $\mathcal{U}_v$, then use the data structure D to determine the quad $\mathcal{Q}$ in $\mathcal{U}_v$ that contains p. This takes $O(\log n)$ time for each p in $End(v)$. Use the results of these queries to construct, for each p, the pair $(\mathcal{Q}, s)$, where $\mathcal{Q}$ is the quad containing p and s is the segment that has p as an endpoint. Each such $\mathcal{Q}$ is active by definition.

To complete the computation for this step, then, we must collect, for each active quad $\mathcal{Q}$ in $\mathcal{U}_v$, all the objects that intersect $\mathcal{Q}$. We do this by sorting all the $(\mathcal{Q}, s)$ pairs by their first coordinate, which takes $O(\log n_v)$ time using $O(n_v)$ processors [6], and follow that by a simple parallel prefix computation. For each such $\mathcal{Q}$, this gives us the names of all the segments in S_v that have an endpoint in $\mathcal{Q}$. In addition, using the pointer information stored for each endpoint, we have, for each active quad $\mathcal{Q}$, all the members of $VorSet(S_x)$ and $VorSet(S_y)$ that lie in $\mathcal{Q}$. This step takes a total of $O(\log n)$ time using $O(n_v)$ processors.

Recall that for the remainder of this algorithm description we concentrate on the computations needed for a particular active quad $\mathcal{Q}$ in S_v, the understanding being that we are performing these steps for each such active quad in parallel.

3.2. Step 3: The Vertical Merge. We have already noted that $\mathcal{Q}$ is composed of a contiguous series of quads of $\mathcal{U}_x$ (none of which is necessarily active in $\mathcal{U}_x$), together with a similar series of quads of $\mathcal{U}_y$. Let $\mathcal{Q}_L$ (resp. $\mathcal{Q}_R$) denote the union of the quads of $\mathcal{U}_x$ (resp. $\mathcal{U}_v$) in $\mathcal{Q}$. Constructing $\mathcal{Q}_L$ and $\mathcal{Q}_R$ can easily be done in $O(\log n_v)$ time using $O(|S_v \cap \mathcal{Q}|)$ processors. In this step we construct the Voronoi diagrams $V_L = Vor(S_x \cap \mathcal{Q})$ and $V_R = Vor(S_y \cap \mathcal{Q})$.

Let us concentrate on V_L, since the method for building V_R is similar. The method is quite simple. Compute the Voronoi diagram for each empty quad in $\mathcal{Q}_L$ (empty with respect to $\mathcal{U}_x$) in $O(1)$ time using a single processor by the sequential method of Yap [21]. The Voronoi diagram of each empty quad is simply the Voronoi diagram for the two segments that bound the quad (and is similar to that given in Figure 1). If the quad is the topmost or bottommost in $\mathcal{U}_x$, then there is only one bounding segment. In any case, the diagram consist of only $O(1)$ edges.

Then, to construct V_L, we simply "concatenate" all the Voronoi diagrams in $\mathcal{Q}_L$, as in [21]. Since the endpoints of the segments defining the active quads in $\mathcal{U}_x$ terminate on the boundaries of $\mathcal{U}_x$, there is no interference between adjacent diagrams in $\mathcal{Q}_L$. Thus, joining all the diagrams in $\mathcal{Q}_L$ can be accomplished simply by ordering them by their intersections along the boundaries of $\mathcal{U}_x$. This takes at most $O(\log n)$ time using $O(|S_v \cap \mathcal{Q}_L|)$ processors, for we can using the sorting method of Cole [6] to put all the diagrams in the correct order.

3.3. Step 4: The Horizontal Merge. In this step we merge the Voronoi diagrams V_L and V_R into the diagram $Vor(S_v \cap \mathcal{Q})$. Let us restrict our attention to V_L for the time being, with the understanding that for each computation we perform for V_L we perform the analogous computation for V_R. In the spirit of [1], [10], and [21] we divide each cell $V(s)$ of V_L into *primitive regions*, or *prims* for short, by adding edges from s to the Voronoi edges of $V(s)$. The added edges are called

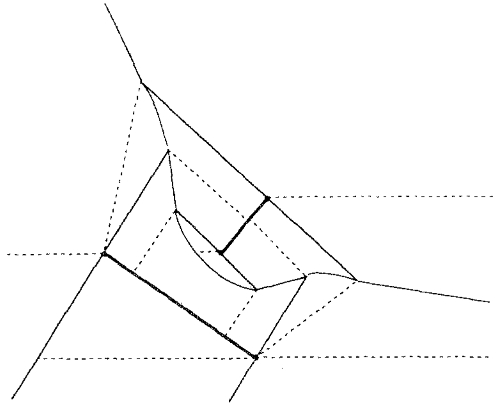

Fig. 2. The spokes and prims in a quad.

spokes [10]. Specifically, we add an edge from each Voronoi vertex of $V(s)$ to its projection on s. In addition, for each point object s in $S_v \cap \mathcal{Q}_L$ we determine the first point of V_L that is intersected by a horizontal ray emanating from s to the left and to the right, respectively. If the line from s to this point does not cross any other spokes, then we also call this edge a spoke. Note: if the horizontal ray emanating from s does not intersect any part of V_L nor any other spokes, then we still call this (semi-infinite) edge a spoke. We call each piece of a Voronoi edge between two consecutive spoke endpoints a *semiedge*. (See Figure 2 for an example decomposition.) Since we add at most two spokes for each Voronoi vertex, we multiply the total size of the augmented diagrams V_L and V_R by at most a constant factor.

Recall that the important computation we perform in this step is the construction of the contour between V_L and V_R. Before we describe our method for doing this, let us study some properties of the contour that are crucial to motivating our method and for proving it correct.

LEMMA 3.1. *Let α and β be two prims such that there are Voronoi cells $V(s_\alpha)$ and $V(s_\beta)$ in V_L and V_R, respectively, such that $\alpha \subset V(s_\alpha)$ and $\beta \subset V(s\beta)$. Let $b_{\alpha,\beta} = B(s_\alpha, s_\beta) \cap \alpha \cap \beta$, where $B(s_\alpha, s_\beta)$ is the bisector of s_α and s_β. If $b_{\alpha,\beta}$ is nonempty, then $b_{\alpha,\beta}$ defines a piece of the contour.*

PROOF. Let p be a point on $b_{\alpha,\beta}$. Since $b_{\alpha,\beta}$ is contained in $\alpha \cap \beta$, there are no objects in V_L (resp. V_R) closer to p than s_α (resp. s_β). Moreover, since $b_{\alpha,\beta}$ is contained in $B(s_\alpha, s_\beta)$, $d(p, s_\alpha) = d(p, s_\beta)$. Thus, $d(p, S_x) = d(p, S_y)$. Therefore, p is on the contour between V_L and V_R. □

This lemma immediately implies a method for constructing the contour in $O(\log n)$ time with a quadratic number of processors—simply compute $b_{\alpha,\beta}$ for each pair (α, β). We do not have this many processors at our disposal, however, Thus, we must be more clever in how we exploit this lemma. The following lemmas

establish additional properties of the contour that lead to a solution using only a linear number of processors.

LEMMA 3.2. *The contour is monotone with respect to the y-axis.*

PROOF. Since our objects are points and open line segments, the monotonicity of the contour follows immediately from a similar lemma by Yap [21] (he also included semicircles as objects). □

LEMMA 3.3. *The contour intersects each spoke at most once.*

PROOF. Suppose the contour intersects a spoke e more than once. Clearly, e is not a horizontal spoke, since if e were horizontal, this would contradict the monotonicity of the contour with respect to the y-axis as shown in Lemma 3.2. Let s be the object incident to e, and let $V(s)$ denote the Voronoi cell for s. The contour partitions the plane into "halves", and in so doing determines a new Voronoi cell for s (call it $V'(s)$), which is defined by all the points in $V(s)$ that are also in the "half" that contains s. Note that since e is not a horizontal spoke, for each point p on e, the projection of p and s is the endpoint of e on s, i.e., $proj(p, s) = e \cap s$ for each $p \in e$. Moreover, by the skeleton property (as defined in Lemma 2.1), the line segment from any such p to its projection $proj(p, s)$ is completely contained in $V(s)$. Since the contour intersects e more than once, the intersection of e with $V'(s)$ must be disconnected, but this contradicts the skeleton property of the Voronoi diagram. Thus, the contour could not have intersected e more than once. □

LEMMA 3.4. *The contour intersects each Voronoi semiedge at most once.*

PROOF. Suppose the contour intersects some Voronoi semiedge e more than once. Without loss of generality, e is in V_R. Let α_1 and α_2 be the two prims that are adjacent to e in V_R. We use s_1 and s_2 to denote the objects in S_v defining α_1 and α_2, respectively.

Case 1: α_1 and α_2 are bounded. Let a, b, c, and d denote the spokes of α_1 and α_2 so that a clockwise listing of the edges bounding $\alpha_1 \cup \alpha_2$ would be (s_1, a, b, s_2, d, c). (See Figure 3(a).) As mentioned before, the contour partitions the plane into "halves." In so doing, it determines two new Voronoi cells for s_1 and s_2, $V(s_1)$ and $V'(s_2)$, which are defined by taking the "half" that contains s_1 and s_2 and intersecting it with the old Voronoi cells for s_1 and s_2, respectively. By the previous lemma, the contour can intersect each of a, b, c, and d at most once. In addition, since s_1 and s_2 are both in the same "half" of the plane determined by the contour, if the contour enters $\alpha_1 \cup \alpha_2$ at a point on b (resp. c), then it must exit at a point on a (resp. d). We claim that the contour intersects e exactly once between its intersections with b and a (and similarly between its intersections with c and d). Otherwise, it must intersect e an odd number of times, say at points p_1, $p_2, \ldots, p_{2k+1}$, where $k \geq 1$. However, the portion of the contour from p_1 and p_2,

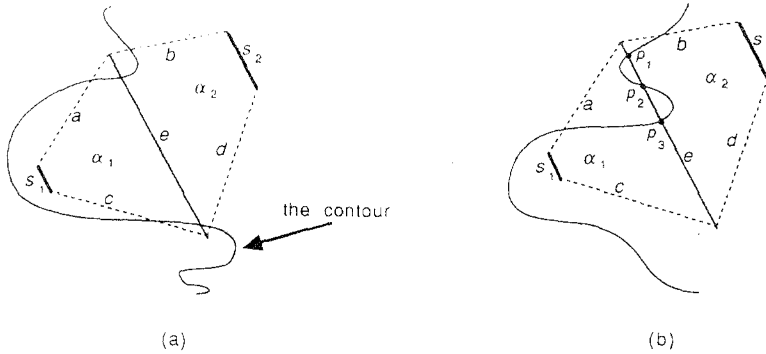

Fig. 3. Illustrating a possible contour (a) and a contour that is impossible (b).

together with the portion of e between p_1 and p_2, defines a disconnected piece of either $V'(s_1)$ or $V'(s_2)$, which contradicts the skeleton property (see Figure 3(b)).

Thus, if the contour intersects e more than once, it must intersect e exactly twice, once between its intersections with b and a, and again between its intersections with c and d (see Figure 3(a)). So the contour intersects the edges bounding α_1 and α_2 in the order (b, e, a, c, e, d) or (a, e, b, d, e, c). Suppose the order is (b, e, a, c, e, d). Then the contour enters α_1 at a point on e, exits at a point on a, enters α_1 again at a point on c, and exits α_1 for the final time at a point on e. Since the contour is monotone with respect to the y-axis (by Lemma 3.2), this implies that there is a horizontal line segment from an endpoint of s_1 to e that lies entirely inside α_1. But this contradicts our method for constructing prims, for such a horizontal segment would have cut α_1 into two prims. A similar argument proves that the sequence (a, e, b, d, e, c) is not possible. Therefore, the contour can intersect e at most once.

Case 2: α_1 or α_2 is unbounded. The proof for this case follows from an argument similar to that above, although it requires placing some segment endpoints at infinity. We leave the details to the reader. □

Given the two previous lemmas, it is an immediate corollary that the contour can intersect any prim in at most one continuous segment. This property is essential to our method for performing the horizontal merge, which we now describe.

Assume, recursively, that we have available $CH_L = CH(S_v \cap \mathcal{Q}_L)$ and $CH_R = CH(S_v \cap \mathcal{Q}_R)$. We begin by computing the convex hull of $CH_L \cup CH_R$, by computing the upper and lower common supporting tangents of CH_L and CH_R. This can be done in $O(1)$ time with $O(|CH_L| + |CH_R|)$ processors using the method of Atallah and Goodrich [3] or Wagener [19]. We know from Lemma 2.2 that the contour must begin with the bisector of the two upper tangent points, and end with the bisector of the two lower tangent points. Let p and q be the two upper tangent points, with p being an object in V_L and q being an object in V_R. We can compute the uppermost vertex of the contour by intersecting $B(p, q)$ with V_L and

V_R, and taking the intersection point that is highest. Using a similar method, we can determine the lowermost vertex of the contour. Let $\mathscr{H}$ be the region of the plane delimited by two horizontal lines l_1 and l_2, where l_1 is above the uppermost vertex of the contour, and l_2 is below the lowermost vertex of the contour. Construct $V'_L = V_L \cap \mathscr{H}$ and $V'_R = V_R \cap \mathscr{H}$, i.e., clip V_L and V_R to $\mathscr{H}$, but do not discard V_L and V_R. Intuitively, $\mathscr{H}$ is the *viewing strip* for the merge, since all the vertices of the contour must lie inside $\mathscr{H}$.

Construct planar point-location data structures D_L and D_R for V'_L and V'_R, respectively. The structure D_L (resp. D_R) allows us to determine for any point p the cell of V'_L (resp. V'_R) containing p. The construction of D_L and D_R can be done in $O(\log n)$ time with $O(n)$ processors using a slightly modified version of the method of Atallah *et al.* [2]. Given a planar subdivision W made up of straight line segments, their method constructs a data structure than can answer the following type of queries: given a point p, locate the segments of W that are immediately above and below p. By associating with each segment of W the names of the faces that lie on either side it, a planar point-location data structure is obtained, because one can simply read off the face containing a point p given the segments immediately above and below p. Even though their method was intended for subdivisions of straight line segments, we can extend their method to a subdivision W' made up of curve segments. The essential properties of segments that Atallah *et al.* exploited were

(1) that segments are monotone with respect to both the x- and y-axes and
(2) given a point p and segment s, one can easily test if p is above s or not.

V'_L and V'_R are made up of line segments and parabolic segments. So, to apply their method we may have to cut some parabolic segments in two at their maxima or minima points (with respect to either the x- or y-axis). We can easily test if a point p is above such a segment or not. Note: we only perform such segment cuts to implement the method of Atallah *et al.*; in all the other steps in our algorithm we consider each such segment to be completely intact. So, to sum up, after performing a simple preprocessing step, we can apply the method of Atallah *et al.* as if all the segments were straight line segments, giving the data structures D_L and D_R.

Construct each prim, clipped to $\mathscr{H}$. For each spoke edge e of V_R we determine if e is intersected by the contour by testing if the far endpoint p of e is closer to $S_v \cap \mathscr{Q}_L$ than it is to $S_v \cap \mathscr{Q}_R$. This can be done in $O(\log n)$ time with a single processor using the data structure D_L just constructed. Similarly, for each spoke edge e of V_L we determine if e is intersected by the contour.

For each prim α in V_L that is intersected by the contour, we determine the prims of V_L that the contour intersects immediately before and after it intersects α, assuming that the vertices of the contour are to be listed by decreasing y-coordinates. This is a well-defined relationship, by Lemmas 3.2–3.4. Moreover, it can easily be determined in $O(1)$ time. In particular, let α be a prim that is intersected by the contour. Note that α is bounded by three edges (not counting the object bounding α), and two of these edges are spokes. By the previous computation, we have determined which of α's spokes are intersected by the

contour. If only one spoke is intersected by the contour, then the contour must necessarily intersect the Voronoi semi-edge bounding α. In any case, it is easy to determine which other prims are before and after α as they intersect the contour given that we know which of the three edges bounding α intersect the contour. Thus, we can easily build a linked list that represents the ordering of the prims of V_L that intersect the contour. We use this linear ordering to construct an array A_L of all the prims in V_L that intersect the contour in the order they are intersected by it. This can be done in $O(\log n)$ time using $O(|V_L|)$ processors by a simple list-ranking technique [20]. By a similar construction, we can build an array A_R of all the prims in V_R that intersect the contour in the order they are intersected by it.

We use the arrays A_L and A_R to construct the contour. The method is similar to that of Aggarwal *et al.* [1], and is as follows. Let α be the median prim in A_L and let s_α be the object that defines α. For each prim β in A_R construct the bisector $B(s_\alpha, s_\beta)$ of s_α and s_β, where s_β is the object that defines β. Then, for each bisector $B(s_\alpha, s_\beta)$, compute the intersection $b_{\alpha,\beta} = F(s_\alpha, s_\beta) \cap \alpha \cap \beta$. This takes $O(1)$ time with $O(|A_L| + |A_R|)$ processors. From Lemma 3.1 we know that if $b_{\alpha,\beta}$ is nonempty, then it defines part of the contour. Moreover, from Lemmas 3.2 and 3.4, the collection of β's in A_R such that $b_{\alpha,\beta}$ is nonempty defines a contiguous interval I of prims in the list A_R. This is because the contour is y-monotone, it intersects each prim in A_R once, and it intersects α in one continuous piece; hence, this piece must lie in contiguous prims of A_R. In addition, this implies that the prims above I in A_R can only interact with the prims in A_L above α, and the prims below I in A_R can only interact wth the prims in A_L below α. Thus, we can recurse on each such pair of prim collections in parallel to construct the entire contour. This entire computation takes $O(\log n)$ time with $O(n)$ processors.

Given the contour C between V_L and V_R we can construct the Voronoi diagram $Vor(S_v \cap \mathcal{Q})$ by taking the portions of V_L to the left of C and the portions of V_R to the right of C and "sewing" them together along C. This is a simple operation to perform, since we know, for each prim α intersecting the contour, the position in C where α's intersection occurs. We also know, for each Voronoi edge e on the boundary of α, whether or not e intersects the contour and if so, where its intersection occurs in C. Thus, for each Voronoi edge e in V_L and V_R we know if e is in $Vor(S \cap \mathcal{Q})$ or not. We can then compress out all the e's that are not a part of $Vor(S_v \cap \mathcal{Q})$. Moreover, for each e that intersects C we can update the fields in e's data record to store correctly the Voronoi vertices in $Vor(S_v \cap \mathcal{Q})$ to which e is adjacent. This procedure requires $O(\log n)$ time using $O(|S_v \cap \mathcal{Q}_L| + |S_v \cap \mathcal{Q}_R|)$ processors. Thus, we have the following:

THEOREM 3.5. *The Voronoi diagram of a proper set of n planar line segments can be constructed in $O(\log^2 n)$ time with $O(n)$ processors in the CREW PRAM model.*

4. Applications. Constructing a Voronoi diagram is often a preprocessing step for solving various problems in robot motion planning, computer vision, and geometric operations research. In this section we review two such problems,

showing that, in each case, Voronoi diagram construction is the bottleneck computation for an efficient parallel solution.

The first problem is that of finding a robust path for a robot, modeled by a circular disk, traveling through a scene containing polygonal obstacles [16]. More precisely, suppose we are given a scene S made up of nonintersecting polygonal objects and wish to move a robot r modelled as a circular disk from some start position p_s to some target position p_t. Ó'Dúnlaing and Yap [16] show that a robust path can be found by tracing a path with r's center that starts from p_s, moves to an edge of the Voronoi cell containing p_s, moves along the edges of the Voronoi diagram of S to an edge of the Voronoi cell containing p_t, and then moves to p_t. Care must be taken however, that an edge e is never traversed such that the minimum distance from e to the segments it bisects is less than the radius of r. To implement this approach in parallel the Voronoi diagram of S can be constructed as given above, and then mark as "removed" each edge e such than the minimum distance from e to the segments it bisects is less than the radius of r. Then a spanning tree of the edges that remain can be formed and a path from an edge of the Voronoi cell containing p_s to the cell containing p_t can be constructed using the methods of Atallah and Vishkin [4], assuming such a path exists. If no such path exists, then this method will correctly find this out. This results in an algorithm that runs in $O(\log^2 n)$ time using $O(n)$ processors in the CREW PRAM model.

The second problem is finding the maximum-flow path of a liquid flowing through a polygonal pipe with a uniform capacity defined on its interior [15]. In [15] Mitchell studies the following problem: we are given a pipe P modeled as a simple polygon and two distinguished edges e_s and e_t on P, and wish to find a maximum flow from e_s to e_t through P, where the interior of P has a uniform capacity. Mitchell shows that, as in the traditional maximum-flow problem, this problem has a max-flow/min-cut property. In particular, he shows that if the removal of e_s and e_f is imagined as splitting the boundary of P into two chains, then the maximum-flow is equal to the length of the shortest segment inside P that joins the two chains. Moreover, this segment is bisected by some edge on the Voronoi diagram of P. Thus, to implement this procedure in parallel, one might begin by performing a parallel prefix computation [11], [12] on the boundary of P to determine the two chains resulting from the "removal" of e_s and e_t. Then the Voronoi diagram of P can be constructed as given above, and, for each Vornoi edge e that bisects objects on opposite chains, determine a shortest segment that joins these objects. Then by a simple min-finding procedure over all such e's which of these segments is the min-cut edge can be determined. Given this edge, a maximum flow can be constructed using a procdure similar to that used to solve the robot motion problem. This also results in a solution running in $O(\log^2 n)$ time using $O(n)$ processors in the CREW PRAM model.

References

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

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

[3] M. J. Atallah and M. T. Goodrich, Parallel Algorithms for Some Functions of Two Convex Polygons, *Algorithmica*, 4 (1988), 535–548.

[4] M. J. Atallah and U. Vishkin, Finding Euler Tours in Parallel, *J. Comput. System Sci.*, **29** (1985), 330–337.

[5] H. Blum, A Transformation for Extracting New Descriptors of Shape, *Proc. Symp. on Models for Perception of Speech and Visual Form* (W. Whaten-Dunn, ed.), M.I.T. Press, Cambridge, MA, 1967, pp. 362–380.

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

[7] R. Cole, M. T. Goodrich, and C. Ó'Dúnlaing, Merging Free Trees in Parallel for Efficient Voronoi Diagram Construction, *Proc. 17th Internat. Conf. on Automata, Languages, and Programming*, 1990.

[8] S. Fortune, A Sweepline Algorithm for Voronoi Diagrams, *Algorithmica*, **2** (1987), 153–174.

[9] R. M. Karp and V. Ramachandran, A Survey of Parallel Algorithms for Shared-Memory Machines, in *Handbook of Theoretical Computer Science*, North-Holland, Amsterdam, to appear.

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

[11] C. P. Kruskal, L. Rudolph, and M. Snir, The Power of Parallel Prefix, *Proc. 1985 IEEE Internat. Conf. on Parallel Processing*, 1985, pp. 180–185.

[12] R. E. Ladner and M. J. Fischer, Parallel Prefix Computation, *J. Assoc. Comput. Mach.*, **27** (1980), 831–838.

[13] D. T. Lee, Medial Axis Transformation of a Planar Shape, *IEEE Trans. Pattern Anal. Mach. Intell.*, **4**(4) (1982), 363–369.

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

[15] J. S. B. Mitchell, On Maximum Flows in Polyhedral Domains, *Proc. 4th ACM Symp. on Comput. Geometry*, 1988, pp. 341–351.

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

[17] F. P. Preparata, The Medial Axis of a Simple Polygon, *Proc. 6th Symp. on Mathematical Foundations of Computer Science*, 1977, pp. 443–450.

[18] M. I. Shamos, Geometric Complexity, *Proc. 7th ACM Symp. on Theory of Computing*, 1975, pp. 224–233.

[19] H. Wagener, Optimally Parallel Algorithms for Convex Hull Determination, unpublished manuscript, September 1985.

[20] J. C. Wyllie, The Complexity of Parallel Computation, Ph.D. thesis, Technical Report TR 79-387, Department of Computer Science, Cornell University, 1979.

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