July 1992

Optimal Mesh Algorithms for the Voronoi Diagram of Line Segments, Visibility Graphs and Motion Planning in the Plane

Sanguthevar Rajasekaran
University of Pennsylvania

Suneeta Ramaswami
University of Pennsylvania

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

Recommended Citation
Optimal Mesh Algorithms for the Voronoi Diagram of Line Segments, Visibility Graphs and Motion Planning in the Plane

Abstract

The motion planning problem for an object with two degrees of freedom moving in the plane can be stated as follows: Given a set of polygonal obstacles in the plane, and a two-dimensional mobile object $B$ with two degrees of freedom, determine if it is possible to move $B$ from a start position to a final position while avoiding the obstacles. If so, plan a path for such a motion. Techniques from computational geometry have been used to develop exact algorithms for this fundamental case of motion planning. In this paper we obtain optimal mesh implementations of two different methods for planning motion in the plane. We do this by first presenting optimal mesh algorithms for some geometric problems that, in addition to being important substeps in motion planning, have numerous independent applications in computational geometry.

In particular, we first show that the Voronoi diagram of a set of $n$ nonintersecting (except possibly at endpoints) line segments in the plane can be constructed in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh, which is optimal for the mesh. Consequently, we obtain an optimal mesh implementation of the sequential motion planning algorithm described in [14]; in other words, given a disc $B$ and a polygonal obstacle set of size $n$, we can plan a path (if it exists) for the motion of $B$ from a start position to a final position in $O(\sqrt{n})$ time on a mesh of size $n$. Next we show that given a set of $n$ line segments and a point $p$, the set of segment endpoints that are visible from $p$ can be computed in $O(\sqrt{n})$ mesh-optimal time on a $\sqrt{n} \times \sqrt{n}$ mesh. As a result, the visibility graph of a set of $n$ line segments can be computed in $O(n)$ time on an $n \times n$ mesh. This result leads to an $O(n)$ algorithm on an $n \times n$ mesh for planning the shortest path motion between a start position and a final position for a convex object $B$ (of constant size) moving among convex polygonal obstacles of total size $n$.

Comments


This technical report is available at ScholarlyCommons: [http://repository.upenn.edu/cis_reports/518](http://repository.upenn.edu/cis_reports/518)
Optimal Mesh Algorithms For The Vironoi Diagram
Of Line
Segments, Visibility Graphs and Motion Planning In
The Plane

MS-CIS-92-57
GRASP LAB 324

Sanguthevar Rajasekaran
Suneeta Ramaswami

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

July 1992
Optimal Mesh Algorithms for the Voronoi Diagram of Line Segments, Visibility Graphs and Motion Planning in the Plane

Sanguthevar Rajasekaran         Suneeta Ramaswami

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

Abstract  The motion planning problem for an object with two degrees of freedom moving in the plane can be stated as follows: Given a set of polygonal obstacles in the plane, and a two-dimensional mobile object $B$ with two degrees of freedom, determine if it is possible to move $B$ from a start position to a final position while avoiding the obstacles. If so, plan a path for such a motion. Techniques from computational geometry have been used to develop exact algorithms for this fundamental case of motion planning. In this paper we obtain optimal mesh implementations of two different methods for planning motion in the plane. We do this by first presenting optimal mesh algorithms for some geometric problems that, in addition to being important substeps in motion planning, have numerous independent applications in computational geometry.

In particular, we first show that the Voronoi diagram of a set of $n$ nonintersecting (except possibly at endpoints) line segments in the plane can be constructed in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh, which is optimal for the mesh. Consequently, we obtain an optimal mesh implementation of the sequential motion planning algorithm described in [14]; in other words, given a disc $B$ and a polygonal obstacle set of size $n$, we can plan a path (if it exists) for the motion of $B$ from a start position to a final position in $O(\sqrt{n})$ time on a mesh of size $n$. Next we show that given a set of $n$ line segments and a point $p$, the set of segment endpoints that are visible from $p$ can be computed in $O(\sqrt{n})$ mesh-optimal time on a $\sqrt{n} \times \sqrt{n}$ mesh. As a result, the visibility graph of a set of $n$ line segments can be computed in $O(n)$ time on an $n \times n$ mesh. This result leads to an $O(n)$ algorithm on an $n \times n$ mesh for planning the shortest path motion between a start position and a final position for a convex object $B$ (of constant size) moving among convex polygonal obstacles of total size $n$.

1  Introduction

The problem of algorithmic motion planning has received considerable attention in recent years. The automatic planning of motion for a mobile object moving amongst obstacles is a fundamentally important problem with numerous applications in computer graphics and robotics. The study of algorithmic techniques for planning motion, with provable worst-case performance guarantees, has been spurred by recent research that has established the mathematical depth of this problem (see [16, 20, 21] for comprehensive surveys). In particular, the design and analysis of geometric algorithms has proved to be very useful, resulting in considerable interplay between computational geometry and algorithmic motion planning for numerous special cases.

We are interested in studying special cases of algorithmic motion planning and the related geometric problems using parallelism. For a number of special cases of motion planning, optimal or near optimal sequential algorithms have been discovered. Our research aims at obtaining optimal parallel algorithms for these problems and will be aided by the significant progress that has been made in the area of parallel algorithms for computational geometry in recent years ([1, 4, 6, 7, 12, 15], for example).

In this paper we develop efficient parallel mesh algorithms for two different techniques of planning motion for an object with two degrees of freedom moving in the plane among polygonal obstacles. One technique for this fundamental case of motion planning uses Voronoi diagram construction for a set of line segments as a subroutine, and the other technique uses planar visibility graph construction.

Visibility graph construction and Voronoi diagrams are geometric problems which, in addition to being tools for motion planning, have many useful applications. Given a set of line segments in the plane, the construction of the visibility graph can lead to information about that part of the plane that is hidden from a given point. This has useful applications in computer graphics. Visibility graphs of line segments also enable us to find the shortest path between two points in the plane while avoiding the line segments. The Voronoi diagram is an elegant and versatile geometric structure and has applications for a wide range of problems in computational geometry and in 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 from the Voronoi diagram. An efficient PRAM algorithm for computing visibility from a point is given by Atallah et. al [4] and Goodrich et al. give a CREW PRAM algorithm for constructing the Voronoi diagram of a set of line segments in the plane [6]. However, to our knowledge, these problems have not been solved on
fixed-connection networks. In this paper, we develop efficient parallel algorithms for these geometric problems on the mesh-connected-computer and as a result, for the corresponding motion planning problems.

The mesh-connected computer (mesh) of size \( n \) is a fixed-connection network of \( n \) simple processing elements (PEs) that are arranged in a \( \sqrt{n} \times \sqrt{n} \) two-dimensional grid. Each PE is connected to its (at most) four nearest neighbors. Attractive features such as simple near-neighbor wiring and ease of scalability have made the mesh-connected computer the focus of considerable attention in parallel algorithms research. The following mesh operations, which will be used in the remainder of this paper, can be implemented in \( \Theta(\sqrt{n}) \) time on a \( \sqrt{n} \times \sqrt{n} \) mesh [12, 22]: perfect shuffle, perfect unshuffle, sorting, selected broadcasting, segmented prefix scan, Random Access Read (RAR), and Random Access Write (RAW). The commonly used indexing schemes on the mesh are row-major, shuffled row-major, snake-like and proximity [12]. An implicit lower bound of \( \Omega(\sqrt{n}) \) holds for most algorithms on the mesh, because nontrivial data movement takes \( \Omega(\sqrt{n}) \) steps.

In the next section we give some important definitions and a brief introduction to relevant background. In Section 3, we develop an optimal mesh algorithm for the construction of the Voronoi diagram of a set of line segments in the plane. We summarize the resulting mesh algorithm for the related motion planning technique at the end of that section. In Section 4, we give an optimal mesh algorithm for determining visibility from a point, and therefore for visibility graph construction. The resulting mesh algorithm for planning motion is also given in that section.

2 Background and Definitions

The motion planning problem of interest to us can be stated in the following way [16]: Given an initial starting position \( P_I \), a final destination position \( P_F \) and a set of stationary obstacles whose geometry is known to \( B \), determine if there exists a continuous obstacle-avoiding motion for \( B \) from \( P_I \) to \( P_F \). If one exists, construct the path for such a motion. Let \( n \) be the size of the obstacle set and let \( k \) be the number of degrees of freedom\(^1\) (dofs) of the mobile object \( B \). Every position of \( B \) can be thought of as a point in \( k \)-dimensional parametric space. Let a free configuration be a placement of \( B \) in which it does not intersect with any of the obstacles. Define \( FP \) to be the subset of \( k \)-dimensional space that contains all the free configurations of \( B \). In general, \( FP \) will consist of many path-connected components. A collision-free path from \( P_I \) to \( P_F \) exists if and only if the corresponding \( k \)-dimensional configurations lie in the same connected component.

There are, in general, two kinds of strategies to solve the motion planning problem. The first general approach, which runs in time polynomial in \( n \) and doubly exponential in \( k \), was first demonstrated by Schwartz and Sharir in [18], and was applied to numerous special cases (see [17], for example; the runtimes for some of these cases have since been improved). The important step in this approach is to construct a connectivity graph that represents the connectivity information of the cells of \( FP \). Planning motion for \( B \) then reduces to performing a graph search on the connectivity graph. The second general approach is to find a one-dimensional representation of \( FP \) (called the "skeleton" or the "road-map") such that it is possible for \( B \) to move from \( P_I \) to \( P_F \) if it is possible to move between two corresponding points on the skeleton. This generalized approach was given by Canny [5] and runs in time polynomial in \( n \) and single exponential in \( k \). Techniques based on the ideas of the first approach will be called the projection methods, and those based on the second approach will be called the retraction methods. The planar motion planning algorithm given by Ö'Dúnlaing and Yap [14], which uses the Voronoi diagram of a set of line segments, employs the retraction method, and the method given by Lozano-Pérez and Wesley [11], which uses visibility graphs, is an approximate projection method (later in the paper, we develop parallel algorithms for the exact version of their method).

2.1 Notation and Important Definitions

2.1.1 Voronoi Diagram of a Set of Line Segments in the Plane

Let \( S \) be a set of nonintersecting closed line segments in the plane. Following the convention in [9, 25], 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. Following [6, 9], we now establish some basic definitions. 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 \( \text{proj}(q, s) \), is defined as follows: Let \( p \) be the intersection point of the straight line containing \( s \) (call this line \( \overline{s} \)), and the line going through \( q \) that is perpendicular to \( \overline{s} \). If \( p \) belongs to \( s \), then \( \text{proj}(q, s) = p \). If not, then \( \text{proj}(q, s) = a \) if \( d(q, a) < d(q, b) \) and \( \text{proj}(q, s) = b \), otherwise.

The distance of a point \( q \) from a closed line segment \( s \) is nothing but \( d(q, \text{proj}(q, s)) \). 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

\(^1\)The degrees of freedom of an object can be defined as the number of parameters that need to be specified in order to completely determine the position of the object.
two line segments is shown in Figure 1.

Definition 2.1 [9] The Voronoi region, \( \text{Vor}(e) \), associated with an object \( e \) in \( S \) is the locus of all points that are closer to \( e \) than to any other object in \( S \) i.e. \( \text{Vor}(e) = \{ p \mid d(p, e) \leq d(p, e') \text{ for all } e' \in S \} \). The Voronoi diagram of \( S \), \( \text{Vor}(S) \), is the union of the Voronoi regions \( \text{Vor}(e) \), \( e \in S \). The boundary edges of the Voronoi regions are called Voronoi edges, and the vertices of the diagram, Voronoi vertices.

The following is a very important property of \( \text{Vor}(S) \).

Theorem 2.2 (Lee et al. [9]) Given a set \( S \) of \( n \) non-intersecting closed line segments in the plane, 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.

Sequential algorithms for the construction of the Voronoi diagram of a set of line segments are given by Kirkpatrick [8], Lee and Drysdale [9], and Yap [25]. The algorithms in [8, 25] run in \( O(n \log n) \) time, which is optimal since a lower bound of \( \Omega(n \log n) \) is known for this problem[19]. The run-time of the algorithm in [9] is \( O(n \log^2 n) \). We will repeatedly refer to Yap’s algorithm in the coming sections, since it lends itself to efficient parallelization, whereas the other two techniques do not. Goodrich et al. [6] give a CREW PRAM algorithm for Voronoi diagram construction that uses \( n \) processors and runs in \( O(\log^2 n) \) time.

2.1.2 Visibility Graphs

Given a set \( S \) of \( n \) line segments in the plane, its visibility graph \( G_S \) is the undirected graph which has a node for every endpoint of the segments in \( S \), and in which there is an edge between two nodes if and only if they are visible to each other, assuming the line segments are opaque. Welzl [23] and Asano et al. [2] give sequential algorithms for constructing the visibility graph of a set \( S \) of line segments with \(|S| = n \) that run in \( O(n^2) \) time. Unfortunately, neither of these two sequential techniques lends itself to efficient parallelization.

The problem of computing visibility from a point, i.e. identifying those vertices of \( S \) that are visible from some specified point \( p \), has a lower bound of \( \Omega(n \log n) \). This lower bound can be established by showing a straightforward reduction from sorting, assuming that we want the output in sorted order about \( p \) (by polar angle with respect to some fixed axis through \( p \) ). Atallah et. al. [4] give an optimal CREW PRAM algorithm for computing visibility from a point that runs in \( O(\log n) \) time using \( n \) processors. The visibility graph can thus be constructed by repeating this algorithm for each of the endpoints of \( S \), which takes \( O(\log n) \) time with \( n^2 \) processors. All the visibility algorithms mentioned here (and those that will be described in the coming sections) are described for a set of line segments. When the input is a disjoint set of polygons, we use the polygon edges as the input set \( S \) to construct the visibility graph.

3 Mesh Algorithms for the Voronoi Diagram of a Set of Line Segments and the Related Motion Planning Problem

As we mentioned in Section 1, the Voronoi diagram turns out to be a useful tool in motion planning [14, 13, 24]. We now describe a mesh-optimal algorithm for the construction of the Voronoi diagram of a set of line segments in the plane. The resulting mesh implementation of the motion planning algorithm by Ó'Dúnlaing and Yap [14] is given in the last part of this section.

3.1 Voronoi Diagram of a Set of Line Segments in the Plane

In this section, we develop a parallel algorithm for constructing the Voronoi diagram of a set of \( N \) line segments in the plane on a \( \sqrt{n} \times \sqrt{n} \) mesh \((n = 2N)\) that runs in \( O(\sqrt{n}) \) time, which is optimal for the mesh. We would like to point out that there is an optimal \( O(\sqrt{n}) \) time parallel algorithm for the Voronoi diagram of a set of \( n \) points in the plane, on a mesh with as many PEs (Jeong and Lee [7]), but none, to our knowledge, for line segments.

The general idea behind the sequential algorithms for the construction of \( \text{Vor}(S) \) (\( S \) is the input set of line segments) is as follows: \( S \) is divided into sets of equal size, \( S_1 \) and \( S_2 \). \( \text{Vor}(S_1) \) and \( \text{Vor}(S_2) \) are then recursively computed. In order to merge these two Voronoi diagrams to form the final diagram \( \text{Vor}(S) \), we need to construct the contour between \( S_1 \) and \( S_2 \). The contour is the locus of all points.
in the plane that are equidistant from \( S_1 \) and \( S_2 \). Thus, assuming the correct orientation on the contour, all points lying to the left (right) of the contour are closer to \( S_1 \) (\( S_2 \)) than to \( S_2 \) (\( S_1 \)). Now, we discard that part of the diagram of \( \text{Vor}(S_1) \) that lies to the right of the contour, and that part of the diagram of \( \text{Vor}(S_2) \) that lies to the left of the contour. The remaining edges of the two diagrams, and the contour edges give us the final Voronoi diagram \( \text{Vor}(S) \).

This is the motivation behind the sequential approaches major indexing on the mesh. Suppose a vertical line is drawn through each point in \( S \). The vertical strip of region between any two such (not necessarily adjacent) vertical lines is called a slab. Consider the set of segments that span a slab \( U \). The region of \( U \) that is enclosed between two such consecutive spanning segments is called a quad of \( U \).

Thus, the construction of the contour is the single most important step in the merge phase of the divide-and-conquer algorithm for Voronoi diagram construction. For the case of a set of points in the plane, we have the nice property that there is exactly one contour to be constructed, and this contour is monotone with respect to the \( y \) axis. In [7], Jeong and Lee exploit this property by first identifying those Voronoi edges of \( \text{Vor}(S_1) \) and \( \text{Vor}(S_2) \) that are intersected by the contour. They then use the monotonicity property to explicitly sort these edges according to the order in which they are intersected. Once this is done, some additional computation gives us the contour. For the Voronoi diagram of line segments, however, it is much more complicated to ensure that this property of the contour holds. As mentioned before, Goodrich et al. [6] give a CREW PRAM algorithm for Voronoi diagram construction that runs in \( O(\log^2 n) \) time using \( n \) processors. Their algorithm makes uses of data structures that are of size \( O(n \log n) \). We cannot make use of such data structures if we assume constant storage per PE on a mesh-connected-computer of size \( n \). In addition, their method performs numerous pointer manipulations, which are very difficult to implement on the mesh.

We circumvent these difficulties by developing an algorithm that performs simpler data manipulation on the mesh. Before we proceed, we state two results that are of relevance to Voronoi diagram construction on the mesh.

**Lemma 3.1** Given a linearly ordered set of elements \( L \) and a set of elements \( E \) such that each \( e \in E \) lies between exactly two elements of \( L \) (call these \( e_a \) and \( e_b \)), and \(|L| + |E| = n\). The problem of finding \( e_a \) and \( e_b \) for every \( e \in E \) can be solved in \( O(\sqrt{n}) \) time on a \( \sqrt{n} \times \sqrt{n} \) mesh. Call this Algorithm SimultSrch.

**Proof:** Omitted. \( \square \)

**Lemma 3.2** (Jeong and Lee [7]) Given an arbitrary set of segments \( S \) in the plane and a set of points \( P \) such that \(|S| + |P| = n\). Let \( p^a \) (\( p^b \)) be the segment from \( S \) that lies immediately above (below) \( p \). The problem of finding \( p^a \) and \( p^b \) for every point \( p \in P \) (also known as the MultiLocation problem) can be solved in \( O(\sqrt{n}) \) time on a \( \sqrt{n} \times \sqrt{n} \) mesh. Call this Algorithm MultiLoc.

Let \( S = \{s_0, s_1, \ldots, s_{N-1}\} \) be the input set of line segments that do not intersect (except possibly at endpoints).

Let \( v_{2i} \) and \( v_{2i+1} \) be the two endpoints of segment \( s_i \), such that \( x(v_{2i}) < x(v_{2i+1}) \). Each segment \( s \) of \( S \) is actually represented as three elements: the two endpoints and the open line segment. Let \( E = \{p_0, p_1, \ldots, p_{n-1}\} \) be the ordered set consisting of these endpoints sorted according to their \( x \)-coordinates (each \( p_i \) is some \( v_j \) and \( n = 2N \)). The mesh algorithm for constructing \( \text{Vor}(S) \) will be a divide-and-conquer algorithm, and so we will assume shuffled row-major indexing on the mesh. Suppose a vertical line is drawn through each point in \( S \). The vertical strip of region between any two such (not necessarily adjacent) vertical lines is called a slab.

Let \( U \) be the slab obtained by merging the adjacent slabs \( U_1 \) and \( U_2 \). The merge step computes the Voronoi diagram in all the active quads of \( U \); this is done by using, with some additional computation, the recursively computed Voronoi diagrams of the active quads of \( U_1 \) and \( U_2 \) to construct the contour. Thus, the most important step in the merge procedure is to compute efficiently, for every active quad \( Q \) in \( U \), \( \text{Vor}(S_U \cap Q) \). Following [6], we let \( \text{VorSet}(S_U) \) represent the set containing the Voronoi diagrams of all the active quads \( Q \) of \( U \) i.e. \( \text{VorSet}(S_U) = \{\text{Vor}(S_U \cap Q) \mid Q \) is an active quad of \( U \} \).

Initially, each PE contains an endpoint \( v_i \) (i.e. the coordinates of \( v_i \)), the segment that \( v_i \) is an endpoint of, and the other endpoint of that segment. In other words, each PE \( P_i \), \( 0 \leq i \leq n - 1 \) has a packet that contains \( v_i \) and \( v_{i+(-1)} \). Initially \( v_i \) is used as the key for processor \( P_i \)'s information.

**Preprocessing:** In this step, (a) first we sort the packets according to the \( x \)-coordinate of the key. Notice that now the arrangement of the keys of the packets is as in the ordered set \( E \). (b) Next, we run Algorithm MultiLoc (refer Lemma 3.2), using \( S \) and \( E \) as the set of segments.
and points, respectively. At the end of this step, we will have for every endpoint \( p_i \) in PE \( P_i \), the segments that lie vertically above and below it. Call these \( p_i^a \) and \( p_i^b \), respectively. \( p_i^a \) will be represented by its two endpoints and its index; similarly for \( p_i^b \). \( p_i^a \) and \( p_i^b \) are now added on to the packet in PE \( P_i \). It will become clear later on that this preprocessing step is necessary in order to determine active quads. Clearly, (a) and (b) take \( O(\sqrt{n}) \) time on a \( \sqrt{n} \times \sqrt{n} \) mesh.

**Basis:** The base step is executed when there is exactly one active quad to which \( p_i \) belongs (obviously, it is the only active quad in said slab) is given by the spanning segments \( p_i^a \) and \( p_i^b \). Clearly, the Voronoi diagram of this quad can be computed in constant time. Hence the base step takes constant time.

**Merging:** Let \( U_1 \) and \( U_r \) be two adjacent slabs, and let \( |E_{U_1}| = |E_{U_r}| = k \) (i.e. each slab has \( k \) endpoints in its interior). Suppose that \( \text{VorSet}(S_{U_1}) \) and \( \text{VorSet}(S_{U_r}) \) have been recursively computed in two adjacent sub-blocks of the mesh, where each sub-block is of size \( \sqrt{k+1} \times \sqrt{k+1} \). Let the left sub-block be called \( M_l \) and the right sub-block \( M_r \). We will show that we can perform the merge in \( O(\sqrt{k}) \) time, using \( O(k) \) PEs.

The information that is necessary for the merge procedure is available in \( M_l \) in the following manner.

1. **Active Quads of \( U_1 \):** The active quads in \( U_1 \) have a sorted order defined on them in the natural way. Let \( A_i \) be the number of active quads in \( U_1 \) (\( A_i \leq k \)); let these be \( Q_{11}, Q_{12}, \ldots, Q_{1A_i} \) in sorted order (from top to bottom, say). See Figure 2 for an example. Let the number of endpoints in these active quads be \( k_{11}, k_{12}, \ldots, k_{1A_i} \), respectively. Note that \( k_{11} + k_{12} + \ldots + k_{1A_i} = k \).

2. **Voronoi Edges of \( \text{VorSet}(S_{U_1}) \):** As stated earlier, \( \text{VorSet}(S_{U_1}) \) is the collection of the Voronoi diagrams of all the active quads in \( U_1 \). Because of the quad-wise computation of the Voronoi diagram, the Voronoi edges of \( \text{VorSet}(S_{U_1}) \) are stored in a quad-wise manner. In other words, in \( M_l \), we will first have the Voronoi edges of \( \text{Vor}(S_{U_1} \cap Q_{11}) \), followed by the edges of \( \text{Vor}(S_{U_1} \cap Q_{12}) \), and so on. Notice that since \( \text{VorSet}(S_{U_1}) \) consists of the Voronoi diagram of at most \( O(k) \) line segments (since only the active quads are considered), it will have \( O(k) \) Voronoi edges; there will be a constant number of these Voronoi edges in each processor of \( M_l \). More importantly, the following observation holds, which follows directly from a lemma by Yap ([25], Lemma 5): The number of Voronoi edges in the Voronoi diagram of an active quad \( Q_{1i} \) of \( U_1 \) is proportional to the number of segments in that quad. In other words, the number of Voronoi edges in \( \text{Vor}(S_{U_1} \cap Q_{1i}) \) is \( O(k_{1i}) \). Therefore, the PEs of \( M_l \) that store active quad \( Q_{1i} \) suffice to store the complete diagram \( \text{Vor}(S_{U_1} \cap Q_{1i}) \), with just a constant number of Voronoi edges per PE.

Let \( A_r \) be the number of active quads of \( U_r \), and let \( k_{ri} \) be the number of points in the \( i \)-th (in the sorted order) active quad \( Q_{ri} \), \( 1 \leq i \leq A_r \) (see Figure 2). The information about the active quads of \( U_r \) and the Voronoi edges of \( \text{VorSet}(S_{U_r}) \) are available in \( M_r \) in a similar and analogous way.

For the sake of brevity, we will give a very general description of the merge step on the mesh without going into the details.

**Summary of the Merge Step on the Mesh** The merge part of this divide-and-conquer algorithm consists of three important substeps: the determination of the active quads of \( U \), the vertical merge, and the horizontal merge.

1. **Determination of the active quads of \( U \):** In this step we compute the active quads of \( U \) by using the information about the active quads of \( U_1 \) and \( U_r \) available in \( M_l \) and \( M_r \), respectively. This is done by merging the endpoints in \( M_l \) with the endpoints in \( M_r \) (recall that these endpoints are in active-quad-wise ordering) according to the upper bounding segment of the quad that they belong to (some \( Q_{1i} \) or \( Q_{ri} \)). This merge can be done by performing the standard shuffle-exchange step. This step ensures that all the points in \( E_{V} \) lie in \( M_l \cup M_r \) in the correct active-quad-wise ordering. An appropriate selected broadcasting step can now update, for every endpoint in \( E_V \), the upper and lower bounding segments of the active quad of \( U \) that it lies in. This step takes \( O(\sqrt{k}) \) time on the mesh \( M_l \cup M_r \) (which has \( 2k + 2 \) PEs).

   **Note:** Consider an active quad \( Q \) from the slab \( U \). Let \( Q_l \) (\( Q_r \)) represent the part of \( Q \) that lies in the left (right) slab \( U_l \) (\( U_r \)). In other words, \( Q_l = Q \cap U_l \) and \( Q_r = Q \cap U_r \). Observe that \( Q_l \) (\( Q_r \)) is the union of a contiguous set of quads of slab \( U_l \) (\( U_r \)). Some of these quads may be active and some or all of them may be (see Figure 2 for an example). We will call these quads (whether active or not) the \( Q_l \)-quads (\( Q_r \)-quads). In order to find the Voronoi diagram of \( Q \), \( \text{Vor}(S_U \cap Q) \), we need to “merge” the Voronoi diagrams of all the \( Q_l \)-quads and the \( Q_r \)-quads in the appropriate way. This merging is achieved by first doing a **vertical merge**, followed by a **horizontal merge**.

2. **The vertical merge:** In this step we find, for every active

---

3Intuitively speaking, the lemma states that for any two quads \( Q_1 \) and \( Q_2 \) in a slab \( U \), the objects in \( Q_1 \) and the objects in \( Q_2 \) do not interact with each other. In other words, the Voronoi edges of the diagram \( \text{Vor}(S_U \cap Q_1) \) will not be affected by the segments in \( S_U \cap Q_2 \). Hence the assertion that the number of edges in \( \text{Vor}(S_U \cap Q_i) \) is \( O(k_{1i}) \).
Figure 2: The $Q_I$-quads and the $Q_r$-quads of an active quad $Q$ of $U$.

tive quad $Q$ of $U$, the Voronoi diagram of $S_U \cap Q_I$, called the $Q_I$-diagram and of $S_U \cap Q_r$, called the $Q_r$-diagram. Notice that the Voronoi diagram of the non-empty $Q_I$-quads ($Q_r$-quads) has already been recursively computed. The Voronoi diagram of an empty $Q_I$-quad ($Q_r$-quad) can be computed in constant time. Thus, determining the empty quads is the important step.

Consider an empty $Q_I$-quad; call it $Q'$. On the mesh $M_I \cup M_r$, we arrange the upper and lower bounding segments of $Q'$ in such a way that there are no endpoints of $E_U$, between the two processors that hold these segments. In addition, we arrange all the $Q_I$-quads, whether empty or active, in the correct sorted order on the mesh (it is clear that such a sorted order on all $Q_I$-quads is well-defined). Similarly for the $Q_r$-quads. By defining an appropriate ordering on all the endpoints of $E_U$, we can sort them into the arrangement described above. We will not go into the details of this ordering for lack of space.

Once this is done, we can determine the empty $Q_I$-quads by performing a segmented prefix scan operation that will count the number of endpoints from $E_U$, between every two consecutive spanning segments of $U_I$. Let PEs $P_j$ and $P_k$ contain two such consecutive spanning segments of $U_I$. Each such set of PEs $P_j, P_{j+1}, \ldots, P_k$ forms a segment of the segmented prefix scan. If the result of the scan in $P_k$ is zero, then these two consecutive spanning segments define an empty $Q_I$-quad and we compute its Voronoi diagram.

This diagram clearly has just a constant number of Voronoi edges, and hence we can store these edges in $P_k$. An analogous application of these steps give us the empty $Q_r$-quads and their Voronoi diagrams. The construction of the $Q_I$-diagram ($Q_r$-diagram) requires us to merge together the Voronoi diagrams of all the $Q_I$-quads ($Q_r$-quads), empty as well as active. This just requires us to "concatenate" the diagrams of all the $Q_I$-quads ($Q_r$-quads) in the correct sorted order (as in [25]). The above computation ensures that these diagrams are, in fact, already in the right order. Hence, the horizontal merge takes $O(\sqrt{K})$ time on $M_I \cup M_r$.

(3) The horizontal merge: In this final stage of the merge step, we obtain the Voronoi diagram of each active quad $Q$. This is done by merging the $Q_I$- and the $Q_r$-diagram, which involves the construction of the contour. The horizontal merge is the most complicated part of this algorithm. Once the contour is constructed, the $Q_I$-diagram to the left of the contour, the contour itself, and the $Q_r$-diagram to the right of the contour give us the final Voronoi diagram $Vor(S_U \cap Q)$ for every active quad $Q$ of $U$. Our discussion will describe the computation performed for one active quad $Q$, with the assumption that the same steps are carried out for all the active quads of $U$.

As in the sequential methods of [8, 25] and the PRAM method of [6], we manipulate objects known as primitive regions for the construction of the contour. For the rest of this discussion, we will assume that the $Q_I$-diagram is augmented in the following way (the $Q_r$-diagram will be augmented in a similar way): For every element $e$ (either a point or an open line segment) in $S_U \cap Q_I$, we add spokes [8] to the Voronoi region $Vor(e)$ of $e$. If $v$ is a Voronoi vertex of $Vor(e)$, and if $v' = proj(v, e)$ (the projection of $v$ on $e$), then the line segment obtained by joining $v$ and $v'$ is a spoke of $Vor(e)$. See Figure 3 for a Voronoi diagram augmented with spokes. In [6], the authors add some additional spokes. For all $e$ that are point elements, we check if the horizontal left-ray from $e$ crosses any spokes before it intersects the boundary of $Vor(e)$. If not, then let $p$ be the point of intersection on the boundary.

The line segment from $e$ to $p$ is also added as a spoke. We do a similar step for the right-ray from $e$. If these left-ray and right-ray rays do not intersect any spokes or Voronoi edges, then these rays are also considered to be spokes. These additional spokes are indicated by bold dotted lines in Figure 3. All spokes define new sub-regions within $Vor(e)$. These sub-regions bounded by two spokes on two sides, part of $e$ on one side, and a piece of Voronoi edge on the other side are called primitive regions (prims for short) [6]. The piece of Voronoi edge that forms one of the boundary edges of each prim is called a semi-edge [6]. Notice that since $VorSet(S_U)$ consists of at most $O(k)$ Voronoi edges and vertices, the number of prims will also be $O(k)$. For the rest of this discussion, we will call the spokes of the $Q_I$-diagram as $Q_I$-spokes, the prims of the $Q_I$-diagram as $Q_I$-prims, and the semi-edges of the $Q_I$-diagram as $Q_I$-semi-edges (similarly for $Q_r$). The segment endpoints or open line segments that belong to $S_U \cap Q_I (S_U \cap Q_r)$ will be called $Q_I$-objects ($Q_r$-objects).

In the merge computation on the mesh so far, our technique has been to store a constant number of Voronoi edges
per PE. Notice that each Voronoi edge (part of \(B(e_1, e_2)\), say) actually defines two prims: one in each of the two Voronoi regions \(\text{Vor}(e_1)\) and \(\text{Vor}(e_2)\). So we will assume that both these prims are stored along with the Voronoi edge. It is also easy to determine the additional spokes (mentioned above) that need to be added. Every prim in \(\text{Vor}(e_1)\), where \(e_1\) is either an endpoint or an open line segment corresponding to segment \(s_1\) in \(\tilde{S}_1\) \(\cap Q\), determines if it is intersected in the desired manner by the left-ward and right-ward rays from both the endpoints of \(s_1\). This can be done in constant time for each prim, and in constant total time for all the prims since there are a constant number of prims per PE.

We now want to construct the contour between the \(Q_I\)-diagram and the \(Q_r\)-diagram. This construction depends crucially on certain properties of the contour. We state these properties as lemmas below, and refer the reader to [6, 25] for the proofs.

**Lemma 3.3** (Goodrich et al. [6]) Let \(\alpha\) and \(\beta\) be \(Q_I\)- and \(Q_r\)-prims, respectively. Let \(s_\alpha \in \tilde{S}_1\), and \(s_\beta \in \tilde{S}_r\), be such that \(\alpha \subseteq \text{Vor}(s_\alpha)\) and \(\beta \subseteq \text{Vor}(s_\beta)\). Let \(b_{\alpha,\beta} = B(s_\alpha, s_\beta) \cap \alpha \cap \beta\). If \(b_{\alpha,\beta}\) is non-empty, then \(b_{\alpha,\beta}\) defines a piece of the contour.

**Lemma 3.4** (Goodrich et al. [6]) The contour is monotone with respect to the y-axis.

**Lemma 3.5** (Goodrich et al. [6]) The contour intersects each spoke and each Voronoi semi-edge at most once.

From the above lemmas it is easy to see that the contour intersects each prim in at most one continuous piece [6].

Before we proceed, we state an important lemma.

**Lemma 3.6** Given a set \(P\) of points in the plane, and the Voronoi diagram of a set \(S\) of line segments, where \(|S| + |P| \leq n\). The problem of finding the Voronoi region that each point \(p \in P\) lies in, can be solved in \(O(\sqrt{n})\) time on a \(\sqrt{n} \times \sqrt{n}\) mesh. Call this Algorithm \text{VORREGIONLOC}.

Algorithm \text{VORREGIONLOC} can be implemented by using a technique similar to that given by Jeong and Lee [7] for Algorithm \text{MULTILOC}, with some minor modifications.

We now outline the important steps in the construction of the contour on the mesh. Notice that at this stage of the merge all the active quads of \(U\) are in sorted order in \(M_I \cup M_r\), and within each such \(Q\), we have the \(Q_I\)-diagram, followed by the \(Q_r\)-diagram.

The contour consists of edges that are of the form \(B(e_1, e_r)\), where \(e_1\) is a \(Q_I\)-object and \(e_r\) is a \(Q_r\)-object. Hence our goal is to identify all such pairs \((e_1, e_r)\). Obviously, if a \(Q_I\)-object \(e_1\) is part of such pairs, then some of its \(Q_r\)-prims will be intersected by the contour (similarly for \(Q_r\)-objects). Notice that determining if a prim is intersected by the contour is equivalent to determining if at least one of the spokes of that prim is intersected by the contour. This is because if the contour intersects a prim without intersecting either of its spokes, then it would have to intersect the semi-edge twice, contradicting Lemma 3.5.

Thus, in order to construct the contour we have to do the following:

(a) Identify the \(Q_I\)-spokes that are intersected by the contour and arrange them in the order that they are intersected by the contour. Such an order exists because of the monotonicity property (Lemma 3.4) of the contour. Call this sorted list \(IS_I\).

(b) Identify the \(Q_r\)-spokes that are intersected by the contour and arrange them in the order that they are intersected by the contour. Call this sorted list \(IS_r\).

(c) From the two sorted lists \(IS_I\) and \(IS_r\), determine the pairs \((e_1, e_r)\) such that \(B(e_1, e_r)\) forms part of the contour.

A summary of the implementation of steps (a), (b) and (c) is given below:

**Step (a):**

(i) Identifying the \(Q_I\)-spokes that are intersected by the contour: Every \(Q_I\)-spoke has one endpoint that is part of a \(Q_I\)-object. Obviously, this endpoint will always be closer to the \(Q_I\)-diagram than to the \(Q_r\)-diagram. However, the other endpoint (call this the "free" spoke-endpoint) of the \(Q_I\)-spoke may or may not be closer to the \(Q_I\)-diagram. If it is not, then the spoke will be intersected by the contour. Apply Algorithm \text{VORREGIONLOC}, using the \(Q_r\)-diagram as the Voronoi diagram of the input, and the "free" endpoints of the \(Q_I\)-spokes as the point set \(P\) of the input. Clearly this can be done in \(O(\sqrt{k})\) time on \(M_I \cup M_r\). Consider a \(Q_I\)-spoke \(l'; l'\) is part of \(\text{Vor}(e_1)\), say. Suppose the "free" endpoint \(p\) of \(l'\) lies in \(\text{Vor}(e_r)\), where \(e_r\) is a \(Q_r\)-object. If \(d(p, e_r) < d(p, e_1)\), then \(l'\) will be intersected by the contour. Since each PE has a constant number of Voronoi edges, we can now identify the intersected \(Q_I\)-spokes in constant time.
(ii) Sorting the intersected \(Q_1\)-spokes: We now arrange the intersected \(Q_1\)-spokes in the order in which they are intersected by the contour (from bottom to top, say). We find this order by explicitly sorting the spokes\(^4\). We will not go into the details of the ordering here for lack of space. Let this sorted list of spokes be called \(IS_1\). \(IS_1\) can be found in \(O(\sqrt{k})\) time on \(M_I \cup M_r\).

Step (b):
Analogous to steps (a)(i) and (a)(iii) above. Let the sorted list of intersected \(Q_r\)-spokes be called \(IS_r\).

Step (c):
Note that the sorted order of intersected \(Q_r\)-prims (\(Q_r\)-prims) is implicit in \(IS_1\) (\(IS_r\)): call this ordered set \(IP_1\) (\(IP_r\)). Consider some prim \(a\) form \(IP_1\). We say that \(a\) interacts with prim \(b \in IP_r\) if \(b_{a,b}\) (refer to Lemma 3.3) is non-empty. In general, \(a\) will interact with some subset of prims from \(IP_r\). This subset will be a continuous interval of prims from \(IP_r\) [6]. Call this interval of prims \(I_a\). Furthermore, all the prims that lie above \(a\) in \(IP_1\) can interact only with those prims of \(IP_r\) that lie above \(I_a\) [6].

We implement this step on the mesh in the following way. For every prim \(a \in IP_1\), we identify the topmost and bottommost prim of the interval \(I_a\). Sequentially, this can be done by using binary search for each \(a\). On the mesh, this step can be done by two applications of Algorithm SimultSRCH (refer Lemma 3.1), which takes \(O(\sqrt{E})\) time on \(M_I \cup M_r\). Let \(P_I\) be the PE that holds the topmost prim of \(I_a\) and \(P_b\) be the PE that holds the bottommost prim of \(I_a\). Each \(a\) can now find the length of the interval \(I_a\).

Next, we make \(|I_a|\) copies of \(a\), and each of those copies reads \(b \in IP_r\) from one of the PEs \(P_1, \ldots, P_k\). We thus determine the piece of the contour \(b_{a,b}\).

Making \(|I_a|\) copies of every \(a\) in \(IP_1\) can be done by a prefix scan on \(|I_a|\), followed by a one-to-one routing, and finally by a selected broadcasting step. To determine each \(b_{a,b}\) that is part of the final contour, each of the copies of \(a\) reads the \(b\) from one of the PEs from \(P_1\) to \(P_k\). This can be done with one Random Access Read step. Since the lengths of the lists \(IP_1\) and \(IP_r\) are each \(O(k)\) for all the active quads \(Q\) of \(U\), the above step can be done in \(O(\sqrt{k})\) time on \(M_I \cup M_r\).

The run-time of the preprocessing step is \(O(\sqrt{n})\). From the summary of the merge step described above, it is seen that the merge step takes \(O(\sqrt{n})\) time. It therefore follows that the Voronoi diagram of a set of \(n\) line segments in the plane can be computed in \(O(\sqrt{n})\) time on a \(\sqrt{n} \times \sqrt{n}\) mesh. We state this result as a theorem.

**Theorem 3.7** The Voronoi diagram of a set of \(n\) non-intersecting (except possibly at endpoints) line segments in the plane can be found on a \(\sqrt{n} \times \sqrt{n}\) mesh in \(O(\sqrt{n})\) time (with no queuing).

### 3.2 Motion Planning Using Voronoi Diagrams

In [14], Ó'Dúnlaing and Yap give a retraction method for planning the motion of an object (a disc) with two \(dofs\), moving amongst polygonal obstacles\(^5\). They use the Voronoi diagram of the line segments that make up the obstacles to plan the motion of the object. We give the mesh-optimal parallel implementation of this method of motion planning.

Let us assume that the object \(A\) has to be moved from point \(a\) to \(b\). First we construct the Voronoi diagram and this takes \(O(n)\) time on a \(\sqrt{n} \times \sqrt{n}\) mesh, as we have just shown. The next step is to remove all the Voronoi edges that do not satisfy the minimum clearance requirement. In other words, we want to delete all Voronoi semi-edges \(v' = B(e_1, e_2)\) such that the minimum distance of the points on \(v'\) from \(e_1\) and \(e_2\) is less than some pre-specified length \(r\) (the radius in the case of a moving disc). Clearly, assuming that we know \(r\), this deletion can be done in constant time on the mesh, since each PE has a constant number of Voronoi edges. The remaining Voronoi edges define a graph which may be disconnected.

The next step is to find the Voronoi cells \(Vor_a\) and \(Vor_b\) that contain the points \(a\) and \(b\), respectively. By Lemma 3.6, this can be done in \(O(\sqrt{n})\) time. The last step is to find a path from an (undeleted) edge of \(Vor_a\) to an (undeleted) edge of \(Vor_b\). One way to do this is by constructing the spanning tree and then finding this path, if one exists. In [3], Atallah and Hambrusch show that in a graph with edge set \(E\), we can solve this problem in \(O(\sqrt{|E|})\) time on a mesh with \(|E|\) PEs. In the graph defined by the Voronoi diagram, \(|E|\) is \(O(n)\). Hence, it follows that we can implement the motion planning technique of [14] in \(O(\sqrt{n})\) time on a \(\sqrt{n} \times \sqrt{n}\) mesh, as stated below.

**Theorem 3.8** Given a polygonal set of obstacles of size \(n\), and a disc \(B\), the motion of \(B\) from one position to another can be planned in \(O(\sqrt{n})\) time on a \(\sqrt{n} \times \sqrt{n}\) mesh.

### 4 Mesh Algorithms for Visibility Graphs and the Related Motion Planning Problem

#### 4.1 Visibility Graphs

We will now describe a mesh algorithm to compute the visibility graph of a given set of line segments in the plane. As noted in the earlier sections, the efficient construction of the
visibility graph is an important substep in motion planning. To our knowledge, this problem has not been solved on the mesh. We will show that, given an input set \( S \) \((|S| = N)\) of nonintersecting line segments in the plane, we can identify mesh-optimally all the segment vertices that are visible from a given point \( p \) in \( \theta(\sqrt{n}) \) (where \( n = 2N \)) time on a \( \sqrt{n} \times \sqrt{n} \) mesh. This will immediately give us an algorithm for constructing the visibility graph, \( G_S \).

Let \( S = \{s_0, s_1, \ldots, s_{N-1}\} \) be the input set of line segments that do not intersect (except possibly at endpoints), and let \( p \) be the point from which we want to determine visibility. Let \( v_{2i} \) and \( v_{2i+1} \) (we will assume \( x(v_{2i}) < x(v_{2i+1}) \)) be the two endpoints of segment \( s_i \). The visibility from a point problem is to determine that part of the plane that is visible from \( p \), assuming that every segment is opaque. Notice that this is equivalent to identifying those vertices \( v_i \) that are "seen" from \( p \). As in [4], we will assume, without loss of generality, that \( p \) is a point at \( -\infty \). This is only to make the description of the algorithm simpler. The case when \( p \) is not at infinity is a straightforward adaptation of this algorithm. Since \( p \) is at \( -\infty \), to compute the visibility from \( p \), we need to compute the lower envelope of the set of segments in \( S \) [4]. The lower envelope is the collection of those segment parts that can be seen from below.

In [4], the authors give a PRAM algorithm that uses the cascading divide-and-conquer technique for solving the visibility from a point problem. Along the same lines, we will describe a recursive algorithm for computing the lower envelope on the mesh. We will first describe the merge step and then give the details of the mesh algorithm. Let \( S_1 \) be the set consisting of half the elements of \( S \), and let \( S_2 \) contain the other half. Suppose that we have recursively computed the lower envelopes of \( S_1 \) and \( S_2 \). The lower envelope of the segments in \( S_i \) \((i = 1, 2)\) is available to us in the following manner: The endpoints of the segments in \( S_i \) have been sorted according to their \( x \)-coordinates (for the sake of simplicity, let us assume that no two endpoints have the same \( x \)-coordinate). In this sorted list (call it \( V_i \)), assume that a vertical line is placed through each endpoint. This divides the plane into vertical strips of region called slabs (call these the \( V_i \)-slabs). The recursive computation gives, for every \( V_i \)-slab, the segment of \( S_i \) that is visible from below (i.e. is part of the lower envelope) in that slab. Now, we want to merge these two envelopes to form the final lower envelope. First merge \( V_1 \) and \( V_2 \) to form \( V \). The set \( V \) defines a new set of slabs. Each \( V \)-slab (say \( u \)) lies within some unique \( V_1 \)-slab (say \( u_1 \)) and some unique \( V_2 \)-slab (say \( u_2 \)). Note that \( u \) could, in fact, be the same as either of \( u_1 \) or \( u_2 \). Let \( s_1 \) and \( s_2 \) be the recursively computed lower envelope segments in the slabs \( u_1 \) and \( u_2 \), respectively. Then, the segment of \( S \) that is visible from below in \( u \) is nothing but the lower of \( s_1 \) and \( s_2 \) (note that such an ordering is uniquely defined on the two segments).

The algorithm for computing the lower envelope (i.e. visibility from \( -\infty \)) is given below.

**Algorithm VisFromPoint;**

**Input:** The endpoints are distributed one per processor on a \( \sqrt{n} \times \sqrt{n} \) mesh with the shuffled row-major indexing scheme. The PE \( P_j \), \( j \in \{0, 1, \ldots, n-1\} \) has endpoint \( v_j \) and also the segment that \( v_j \) is an endpoint of.

**Output:** The endpoints will be in sorted order on the mesh. Thus each PE \( P_i \) is associated with a slab in the obvious way. \( P_i \) will also have the segment \( s \) that is part of the lower envelope (i.e. is visible) in that slab.

**Initialization:** Every PE \( P_i \) has the following fields as part of its record: endpoint initialized to \( v_i \); lowerseg, which contains, at any stage, the lowest segment (found up to that stage) for the slab defined by \( P_i \); whichblock, which indicates (for the merge step) whether an endpoint came from the left block or the right.

**Basis:** lowerseg is set to the segment \( s_{i/2} \) if \( i \) is even and to \( \emptyset \) otherwise.

**Recursive Step:** Solve recursively in parallel using \( S_1 \) for \( S \) in the left block and \( S_2 \) for \( S \) in the right block.

**Merge Step:**

(i) Set whichblock to 0 if \( P_i \) belongs to the left block and to 1 if it belongs to the right block.

(ii) Merge the two sets \( S_1 \) and \( S_2 \) according to the endpoint field.

**Note:** We now need to update the lowerseg field in each PE. As explained earlier, every new slab \( u \) of the merged set needs to compare the recursively computed lowerseg fields of the two old slabs \( u_1 \) and \( u_2 \) that it is a part of. This can be done by using the selected broadcasting operation as stated below.

(iii) The subset of elements that needs to be broadcast is the lowerseg field in every processor with whichblock = 0. Let \( \{l_1, l_2, \ldots, l_{n/2}\} \) (where \( n = 2|S| \)) be the set of these lowerseg in sorted order and let \( l_i \) be the index of the processor in which \( l_i \) resides. The selected broadcasting operation will send \( l_i \), \( 1 \leq i \leq n/2 \), to every PE from \( P_{l(i)} \) to \( P_{l(i+1)-1} \). Put \( l_i \) in a local register called lowerseg1.

(iv) Similar to step (iii), except that the broadcast elements are the lowerseg fields from processors with whichblock = 1. Here, the broadcast element is put in a local register called lowerseg2.

(v) Every PE updates the lowerseg field to the lower of lowerseg1 and lowerseg2.

It is clear that the merge step takes \( O(\sqrt{n}) \) time and thus we have the following theorem.

**Theorem 4.1** Algorithm VisFromPoint, which computes the lower envelope of a set of segments \( S \), runs in

\[ \text{\textit{TIME}} \leq \Theta(\sqrt{n}) \]
$O(\sqrt{n})$ time (with no queueing) on a $\sqrt{n} \times \sqrt{n}$ mesh, where $|S| = n/2$.

Notice that the computation of the lower envelope on the mesh immediately tells us which endpoints of $S$ are visible from $-\infty$. When the point $p$ is not at $-\infty$, the algorithm is the same as above, except that instead of merging the endpoints of the line segments according to their $x$-coordinate, we merge them according to the polar angle that they make with $p$ (measured with respect to some fixed axis). In order to construct the entire visibility graph, we can use the above algorithm in a straightforward way. When a vertex $v_i$ is used as $p$, we can obtain the set of vertices of $S$ that are visible from $v_i$. In other words, we know which nodes are adjacent to the node corresponding to $v_i$ in the visibility graph. If we repeat this for every endpoint $v_j$, in parallel, we can construct the visibility graph of a set $S$ of segments in $O(\sqrt{n})$ time using $n^2$ processors (i.e. $n$ of the $\sqrt{n} \times \sqrt{n}$ meshes). This is optimal since the visibility graph may have $O(n^2)$ edges in the worst case, and we will need $n^2$ processors to represent the graph (under the assumption that each processor has only a constant amount of storage).

4.2 Motion Planning Using Visibility Graphs

Lozano-Pérez and Wesley [11] give an approximate projection method for planning the motion of a convex object $B$ (of constant size) with two dofs, moving between convex obstacles (total size $n$). We summarize their sequential approach briefly: First “expand” each convex obstacle $O$ according to some reference point on $B$, which can be done in time proportional to the size of $O$. $B$ will not collide with $O$ if and only if the reference point of $B$ lies outside of the expansion of $O$. Let $A$ be the union of all the expanded obstacles. Since $B$ has 2 dofs, the configuration space of $B$ is 2-dimensional. In fact, the complement of $A$ in the plane is the set of free configurations, $FP$, for $B$. Let $E$ be the set of edges of $A$. The next step is to compute the visibility graph of the set of edges $E$. The visibility graph is precisely the connectivity graph (of the projection method) that we are looking for. In addition, we have the useful property that the shortest possible path for $B$ between two points in the plane while avoiding the obstacles is given by the shortest path between the corresponding two nodes in the visibility graph (where the edge weight is the Euclidean length of the edge). Thus we can find a shortest path for $B$ by performing a shortest-path graph search on the visibility graph. The sequential run-time of this motion planning method is $O(n^2)$.

Assume the obstacle set is stored in a $\sqrt{n} \times \sqrt{n}$ mesh. First we expand the obstacles according to the moving object $B$: We relay the information about $B$ to each of the PEs in $\sqrt{n}$ time. Since the expansion of each obstacle can be done in time proportional to its size, the expansion of all the obstacles can be done in at most $O(\sqrt{n})$ time. Note that these expanded obstacle edges might now intersect with each other. When the obstacles are convex, it can be shown that the number of such intersections can be at most $O(n)$ [20]. Thus the new obstacle edge set will also be $O(n)$ and there are efficient sequential algorithms to compute it [20]. We can also find the new obstacle edges by using a brute force technique which is very inefficient, but will not alter the run-time of this motion planning algorithm on the mesh.

We can simply compute the intersection of every edge of the expanded obstacle set with the other edges of that set. This will give us the new edge segments, and this can be computed in $O(n)$ time on a mesh with $n$ PEs.

We now have to make $n$ copies of the new obstacle edge set on $n$ sets of $\sqrt{n} \times \sqrt{n}$ meshes so that we may compute visibility from each of the $n$ endpoints. These copies can clearly be made in $O(n)$ time on a mesh with $n^2$ processors. We know, as mentioned above, that the visibility information from each endpoint can be computed in $O(\sqrt{n})$ time by using Algorithm VisFromPoint on each of these submeshes.

Suppose that the object $B$ has to be moved from point $a$ to point $b$. First we establish the visibility information from $a$ and $b$, which can be done in $O(\sqrt{n})$ time using Algorithm VisFromPoint. We can compute the shortest path from $a$ to $b$ by solving the all-pairs shortest path problem for the visibility graph, using the Euclidean length of the edges as the corresponding edge weights. In order to do this, we want to convert the information about the visibility graph into the form of an adjacency matrix on the mesh with $n^2$ PEs. This can be done easily with a sorting step, which will take $O(n)$ time. The all-pairs shortest path can be computed by a method that is very similar to the method used to compute the transitive closure of a matrix. As shown in [10], the all-pairs shortest path problem can be solved in $O(n^3)$ time by using a pipelining technique on a $n \times n$ mesh. Thus, planning the motion of a convex object of two dofs moving among convex obstacles can be done in $O(n^3)$ time on a $n \times n$ mesh. Even though this mesh algorithm is not very work-efficient when compared to the $O(n^2)$ sequential algorithm, note that this is the best we can do since we will need $n^2$ PEs to represent the adjacency matrix.

---

7Note that, for our purposes here, solving the single-source shortest path (from $a$) problem would have sufficed. However, there are no known optimal parallel algorithms for this problem.

8Consider the $\sqrt{n} \times \sqrt{n}$ submesh that computed visibility from a particular endpoint $v_i$. The PEs in this submesh have the endpoints in sorted order about $v_i$. Consider the PE $P'$ that holds vertex $v_j$. If $v_i$ can see $v_j$, then $P'$ will send a 1 to row $i$ and column $j$ of the adjacency matrix. If not, then $P'$ does nothing. This is a one-to-one routing step and can be accomplished through sorting.
5 Conclusions

We have given efficient parallel algorithms for some important geometric problems on the mesh-connected computer. As a consequence, we obtained efficient parallel motion planning algorithms for some fundamental special cases. Speed of execution is a very important consideration for motion planning problems. The development of parallel algorithms for the interesting and complex geometric problems that are of relevance can lead to significantly faster solutions. Moreover, different parallel techniques for such problems could lead to new insights into planning motion. For example, there are no known optimal PRAM algorithms for such problems. In particular, it is possible that randomization could lead to a better parallel algorithm for the construction of the Voronoi diagram of line segments. In addition, it would also be interesting to see if randomization can be a useful strategy for designing faster solutions to various special cases of motion planning.

References