December 1993

Algorithmic Motion Planning and Related Geometric Problems on Parallel Machines (Dissertation Proposal)

Suneeta Ramaswami
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-93-98.

Algorithmic Motion Planning and Related Geometric Problems on Parallel Machines (Dissertation Proposal)

Abstract

The problem of algorithmic motion planning is one that 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. Numerous approximate techniques (AI-based, heuristics-based, potential field methods, for example) for motion planning have long been in existence, and have resulted in the design of experimental systems that work reasonably well under various special conditions [7, 29, 30]. Our interest in this problem, however, is in the use of *algorithmic* techniques for motion planning, with provable worst case performance guarantees. The study of algorithmic motion planning has been spurred by recent research that has established the mathematical depth of motion planning. Classical geometry, algebra, algebraic geometry and combinatorics are some of the fields of mathematics that have been used to prove various results that have provided better insight into the issues involved in motion planning [49]. In particular, the design and analysis of geometric algorithms has proved to be very useful for numerous important special cases. In the remainder of this proposal we will substitute the more precise term of "algorithmic motion planning" by just "motion planning".

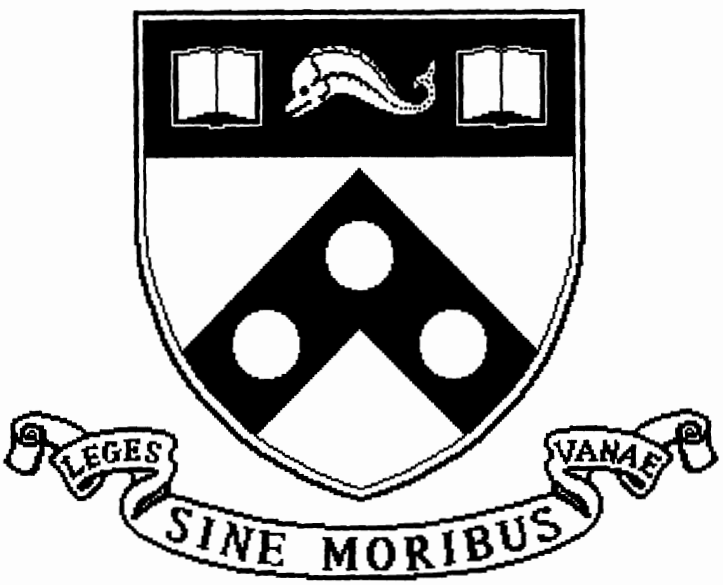
Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-93-98.

Algorithmic Motion Planning and Related Geometric Problems on Parallel Machines (Dissertation Proposal)

MS-CIS-93-98

Suneeta Ramaswami



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

December 1993

Algorithmic Motion Planning and Related Geometric Problems on Parallel Machines

A Dissertation Proposal

Suneeta Ramaswami

May 1992

Contents

List of Figures

1 Motivation

1.1 Introduction

The problem of algorithmic motion planning is one that 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. Numerous approximate techniques (AI-based, heuristics-based, potential field methods, for example) for motion planning have long been in existence, and have resulted in the design of experimental systems that work reasonably well under various special conditions [7, 29, 30]. Our interest in this problem, however, is in the use of *algorithmic* techniques for motion planning, with provable worst-case performance guarantees. The study of algorithmic motion planning has been spurred by recent research that has established the mathematical depth of motion planning. Classical geometry, algebra, algebraic geometry and combinatorics are some of the fields of mathematics that have been used to prove various results that have provided better insight into the issues involved in motion planning [49]. In particular, the design and analysis of geometric algorithms has proved to be very useful for numerous important special cases. In the remainder of this proposal we will substitute the more precise term of "algorithmic motion planning" by just "motion planning".

Let B be a programmable object (for example, a robot) with k degrees of freedom[1] (*dofs*) that is mobile in two- or three-dimensional space. In its most general form, the algorithmic motion planning problem can be stated in the following way [46]: Given an initial starting position P_I, a final destination position P_F and a set of 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. There are numerous variations of the motion planning problem; for example, the set of obstacles might themselves be moving, or B may have incomplete knowledge about its environment. In addition, it may be necessary to take the dynamics of the system into account i.e. a system may be restricted to move within certain velocity or acceleration bounds [49]. Our interest, however, is in planning the motion of B in static and known environments; the obstacles are stationary and B has complete knowledge about them and our interest is in the geometric nature of the problem. As we will see soon, even under this simplifying assumption, general cases of motion planning can be computationally intractable. The more general variations mentioned above are substantially more difficult, and we will not directly address them in our proposal.

We first give a summary of the results that provide lower bound results for certain general cases of motion planning. We will see in Subsection 1.2 that planning the motion of an object rapidly

[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.

becomes intractable as the number of degrees of freedom of the object increases. This result was reinforced by Reif in [38]. In that paper, he shows that planning the motion of a 3 dimensional system of linkages consisting of arbitrarily many links and moving through a system of narrow tunnels is $PSPACE$-hard[2]. Subsequently, Hopcroft *et al.* [19] showed that the general motion planning problem is $PSPACE$-hard even in 2 dimensions. They used a mechanical system of 2-dimensional linkages to establish the result. Another instance of a $PSPACE$-hard motion planning problem in 2 dimensions was demonstrated in [21]. NP-hardness or NP-completeness results have also been established for numerous simpler special cases (in [20], for example). As observed in [49], these lower bound results are strong since they hold for the decision problem corresponding to the motion planning problem. In other words, these lower bound results are for the problem of determining whether or not the object can be moved from P_I to P_F (a 'yes' or 'no' answer); the path itself is not constructed.

In what follows, we will give a brief summary of some general strategies that have been developed for motion planning. These strategies have led to efficient algorithms for some special cases in 2 and 3 dimensions.

1.2 General Strategies for Motion Planning

Despite these discouraging lower bounds, some general techniques have been developed for algorithmic motion planning. These techniques yield polynomial-time algorithms for useful special cases of motion planning for objects with a low number of *dofs*. Schwartz and Sharir [42, 43, 44, 45] did some of the earliest and most fundamental work in the design of exact *geometric* strategies for planning motion. We give a brief summary of the general strategy. Let n be the size of the obstacle set and let k be the number of *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. FP is the subset of k-dimensional space that contains all the free configurations of B. Construction of FP is the first step. 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. Each such connected component consists of *cells*. A *connectivity graph* is now constructed with a node for each cell. By doing a graph search on the connectivity graph, it is possible to plan a collision-free path for B, if it exists. Schwartz and Sharir showed that this strategy leads to algorithms whose worst-case run-times are polynomial in n, but doubly exponential in k (i.e. $O(n^{2^k})$) [43].

A dramatic breakthrough was made by Canny [8] who came up with a general algorithm with

[2]A problem is said to be $PSPACE$-hard if it is at least as hard (with respect to polynomial time reductions) as any problem that can be solved by using storage that is polynomial in the input size. It is highly unlikely that such problems can be solved by efficient polynomial-time algorithms.

a worst-case run-time that is polynomial in n, but *single* exponential in k. Instead of decomposing FP into cells, Canny constructs a one-dimensional skeleton (he calls this the "road map") that has a one-to-one correspondence with the curves in FP in the following sense: Every placement in FP can be moved continuously to a placement on the skeleton, and the intersection of the skeleton with every connected component of FP is non-empty. Thus, in order to plan the motion of B from P_I to P_F, we move the corresponding placements in k-dimensions to some points X_I and X_F on the skeleton. We then search for a path from X_I to X_F along the skeleton. If such a path exists, we know how to move B from P_I to P_F. If it does not, the property of the skeleton (stated above) ensures that it is not possible to move B from P_I to P_F.

A direct application of the above algorithms for special cases may not give us efficient algorithms. However, they provide us with very useful insights into possible geometric approaches that can be tailor-made to the specific cases that we are interested in. Note that even though the above algorithms are exponential in k, for objects with a small number of *dofs*, they have a worst-case run-time that is polynomial in n. Acceptably efficient algorithms have been designed for many special cases.

Two different methods have commonly been used in the design of such algorithms: the *projection method* and the *retraction method* [49, 41]. The projection method is a derivative of the strategy developed by Schwartz and Sharir, outlined earlier. The main goal of this method is to design efficient ways to come up with a cell decomposition of FP for the specific instance of motion planning that is being considered. In Subsection 1.2.1, we will briefly describe an algorithm by Lozano-Pérez and Wesley [30] for one such specific instance. The algorithm uses the ideas of the projection technique. In Section 6, we point out some other cases of motion planning that have been solved reasonably efficiently by this technique.

The retraction method is similar to the general technique developed by Canny. This method proceeds by "retracting" FP onto some lower dimensional space in an appropriate way. In Subsection 1.2.2, we describe a particular motion planning algorithm by Ó'Dúnlaing and Yap [34] that uses the ideas of retraction method. Other planning algorithms that also use this method will be noted in Section 6. We provide the summary of the following specific instances of motion planning because of our interest in developing efficient parallel algorithms for them.

1.2.1 Application of the projection method for an object with 2 *dofs*

One of the earliest geometric approaches to motion planning was given by Lozano-Pérez and Wesley in [30]. In that paper they give approximate solutions for planning the motion of a 2-d convex translational object moving amongst convex obstacles, a 2-d convex rotational (3 *dofs*) object moving amongst convex obstacles, and a translational convex polyhedron moving amongst convex polyhedral obstacles.

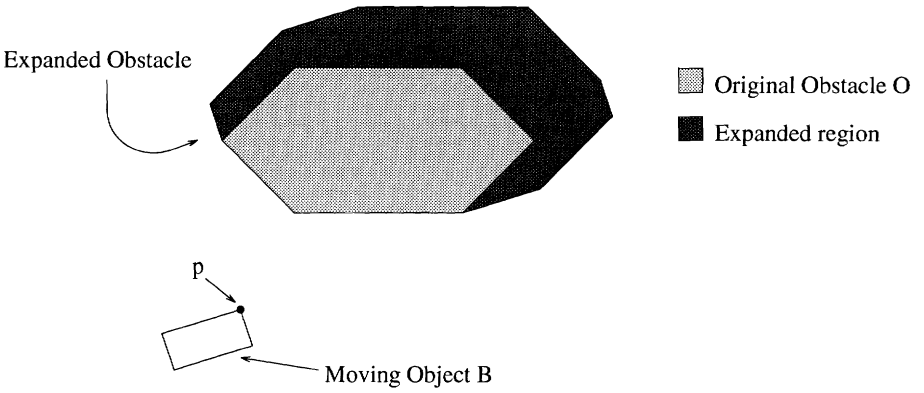


Figure 1: Obstacle O is expanded by object B (which has 2 *dofs*).

We are interested in the method used in [30] for planning the translational motion of a convex object B (B has 2 *dofs*) moving among convex obstacles in the plane. Their method is an approximation of the projection method when applied to this case (their technique actually precedes the development of the general projection method). Let b be some reference point on B, and without loss of generality, let us assume that b is at the origin. Let B_e be the number of edges in B. The strategy in [30] is to first "expand" each convex obstacle O by B. This can be done by computing the Minkowski set difference $O - B = \{x - y \mid x \in O, y \in B\}$ (x and y are vectors). It can be shown that the expansion of O has $B_e + O_e$ (O_e is the number of edges of O) edges and can be constructed in time proportional to its size. See Figure 1 for an example of an expanded obstacle. It is clear that B will not collide with O if and only if the reference point b of B lies outside of the expansion of O. Let A be the union of all the expanded obstacles, and let E be the set of edges[3] of A. 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.

The next step is to compute the *visibility graph* of the set of edges E. The visibility graph gives us information about which endpoints of the obstacle edges are visible to each other (assuming the obstacle edges are opaque). We will be considering the problem of visibility graph construction at more length and we leave its details, including its formal definition, to Section 2. 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.

Let n be the number of edges in the obstacle set and we assume that the size of B is a constant.

[3]Note that the expanded obstacles may now intersect with each other and thus we have to perform some computation before we can determine E. This step was, in fact, not given in [30], but is given by Sharir in [48]. We will not go into the details of the relevant results here.

The expansion of the obstacles and the possible intersections between the expanded obstacles can be computed in $O(n\log^2 n)$ time [48]. The computation of the visibility graph can be done in $O(n^2)$ time, and the shortest-path search takes time proportional to the size of the visibility graph. Thus the sequential run-time of the above algorithm is $O(n^2)$.

1.2.2 Application of the retraction method for an object with 2 *dofs*

In [34], Ó'Dúnlaing and Yap give a motion planning algorithm for the simple case of a disc moving in 2-dimensions among polygonal obstacles. The position of a disc is specified by the coordinates of its center. The algorithm given in [34] is an elegant application of the retraction method (the method of [34] actually precedes the general retraction approach developed by Canny [8]).

The retraction approach of [34] is based on the idea that the disc D should be moved in such a way that it should be far away from its nearest obstacles as it moves. In [34], D is moved so that during its entire motion, it is equally far away from its closest obstacles. This is achieved by the construction of the *Voronoi diagram* of the set of line segments that form the polygonal obstacles. The construction of this diagram is the most important step of this method, and we will be considering this problem in much more detail in the following sections. Intuitively speaking, any point on the Voronoi diagram is equally distant from two segments, and is closer to these segments than to any other segment of the input set. Ó'Dúnlaing *et al.* prove that the Voronoi diagram is in fact a one-dimensional retraction of the free space FP of D. In other words, D can move continuously between two points in FP if and only if it can move between a corresponding two points on the Voronoi diagram.

Once the Voronoi diagram is constructed, they discard those portions of the diagram where the closest obstacle edge is too close for D to fit. This will happen if the closest obstacle is closer than the radius of the disc. We can plan the motion for D by searching along the remaining Voronoi diagram; this can be done by any path-finding method for graphs. Note that this method has the useful property that the object always moves in such a way that it is as far away as possible from all the obstacles. In other words, it has the maximum clearance property. As observed in [34], the motion paths planned by this method may be much longer than the shortest paths. We would also like to observe that it is possible to extend this technique to plan the motion of any convex object with 2 *dofs* moving among polygonal obstacles [48].

Let n be the number of edge segments in the polygonal obstacle set. The Voronoi diagram of these segments can be constructed sequentially in $O(n \log n)$ time [25, 26, 57]. Also, the size of the Voronoi diagram is $O(n)$ [26]. Hence the removal of the appropriate edges of the diagram, and the search for a path can be done in $O(n)$ time. Therefore, the sequential run-time of the above motion planning method is $O(n \log n)$.

1.3 Motivation for Parallel Algorithms

It is clear that algorithmic motion planning relies on efficient solutions to a wide variety of geometric problems. The goal of this proposal is the study of such geometric problems in the parallel environment. The availability of parallel computers has motivated the development of parallel algorithms for solving a number of problems. Parallelism, aside from being an interesting problem-solving strategy in its own right, is of particular relevance in the arena of computationally expensive and intractable problems, such as those in algorithmic motion planning. For a number of special cases of motion planning, we know that their sequential algorithms are either optimal or close to optimal because of known lower bounds. Therefore, speed-ups obtained from a single sequential processor will be limited by a constant or small factor. Parallel algorithms, however, offer the possibility of significant speed-ups. Given that speed is an extremely important consideration for motion planning problems, we will benefit from the study of parallel algorithms for geometric problems related to motion planning. In addition, attempting to solve a given geometric problem in parallel might give us insights into new strategies for solving that problem.

The *P*arallel *R*andom *A*ccess *M*achine (PRAM) is the most general shared memory model of parallel computation. Communication between any two processors occurs through memory cells that are shared by all the processors. Depending on how the processors handle read and write conflicts, we have three different kinds of PRAMs: *E*xclusive *R*ead *E*xclusive *W*rite (EREW) PRAMs (no simultaneous reads or writes are allowed), *C*oncurrent *R*ead *E*xclusive *W*rite (CREW) PRAMs, and *C*oncurrent *R*ead *C*oncurrent *W*rite (CRCW) PRAMs (write conflicts are handled in some pre-determined fashion). Let A be a parallel algorithm for some problem whose input is of size n, and let B be the best known sequential algorithm for that problem. Let $p(n)$ be the number of processors used by A, $t(n)$ the run-time of A and $s(n)$ the run-time of B. A belongs to the parallel complexity class NC^k if $p(n)$ is polynomial in n and $t(n)$ is $O(\log^k n)$. The parallel complexity class NC is defined to be $\bigcup_k NC^k$. $p(n) * t(n)$ is called the PT-product (or PT-bound) of A, also known as the *work* done by A. A is said to be *optimal* if its PT-product is $s(n)$ (or even $\theta(s(n))$). The *speedup* of A is defined to be the ratio $s(n)/t(n)$.

The PRAM model is a powerful model of parallel computation and a good place to start when we want to design parallel algorithms. This model frees the designer from having to worry about inter-processor communication issues and memory organization considerations. She or he can thus think about the more fundamental issues that arise when designing a parallel algorithm for a particular problem; in essence, this model allows one to think in terms of abstract parallelism. It gives us enough flexibility so that we need not be constrained by practical limitations! We will rely on the PRAM model of computation in order to look into the design of parallel algorithms for various instances of motion planning for which parallel algorithms do not currently exist.

Even though designing parallel algorithms on the PRAM model provides a good starting point,

it is not sufficient to stop there. Powerful though it might be, the PRAM model is not practical. In real life, parallel machines are *fixed connection networks* in which inter-processor communication *cannot* be considered to be a constant time operation and there are limits on the amount of memory available to each processor. Hence, if we are interested in implementing algorithms on existing parallel machines, then it is important to look into the design of parallel algorithms on practical fixed-connection network architectures. Theoretically speaking, since it is possible to simulate a PRAM on fixed-connection networks, PRAM algorithms can immediately lead us to algorithms on actual parallel machines. However, algorithms designed explicitly for the particular parallel architecture in which we are interested, are often significantly better than those obtained by a direct simulation of PRAM algorithms. In this proposal, we will also consider the design of parallel algorithms for geometric problems on a particular fixed-connection architecture called the *mesh-connected computer* (also known as the *mesh*). The details of the mesh and its operations will be provided in Section 3.

1.4 Parallel Algorithms for Related Geometric Problems

As we have mentioned before, our interest is in geometric problems that are related to motion planning and parallel algorithms for them. Our research will be aided by the significant progress that has been made in the area of parallel algorithms for computational geometry in recent years ([2, 6, 15, 16, 22, 31, 40], for example). In the two specific instances of motion planning that we mentioned earlier, the important related geometric problems are the construction of the visibility graph and the Voronoi diagram of the set of line segments in the plane.

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. As we mentioned earlier, we can also find shortest paths in the plane from the visibility graphs. 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. We note that Goodrich *et al.* give efficient PRAM algorithms both for the visibility graph construction [6] and for the construction of the Voronoi diagram of a set of line segments [15]. In this proposal, we develop efficient parallel algorithms for these geometric problems on the mesh-connected-computer, and, as a result, for the corresponding motion planning problems.

1.5 Outline of the Proposal

In the next section we give the relevant definitions and establish some notation for visibility graphs and Voronoi diagrams of a set of line segments in the plane. We also summarize the main ideas

behind the existing sequential as well as PRAM algorithms for these problems. Following that, in Section 3 we describe important details of the mesh-connected-computer and some useful operations that are performed on it.

In sections 4 and 5, we describe our contributions to date. In Section 4 we provide an optimal algorithm for visibility graph construction on the mesh. We also summarize the resulting mesh-optimal implementation of the motion planning algorithm of [30] outlined earlier. In Section 5, we first give an optimal mesh algorithm for a special case of Multipoint location. The constant in the run-time of this algorithm is a significant improvement over the corresponding constant of the algorithm given in [22], leading to an improvement in the mesh algorithm for the general multilocation problem. In addition, this algorithm is used repeatedly for Voronoi diagram construction. The bulk of Section 5 consists of the description of the mesh algorithm for Voronoi diagram construction of a set of line segments in the plane. This algorithm is optimal for the mesh. As a result of the Voronoi diagram algorithm, we obtain an optimal mesh implementation of the motion planning algorithm of [34] (outlined in Subsection 1.2.2). We give this algorithm in the last part of Section 5.

Finally, in Section 6 we discuss the scope of our research and delineate the specific problems of interest.

2 Brief Overview of Sequential and PRAM Algorithms and their Complexity

2.1 Visibility Graphs

The efficient construction of the visibility graph is an interesting problem in its own right and, as we mentioned earlier, it is an important substep for certain motion planning algorithms. In this subsection we define the visibility graph and briefly outline the sequential algorithms for the planar case. Following that, we give a summary of the known PRAM algorithm [6] for this problem.

2.1.1 Definitions and Sequential Algorithms

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. (see Figure 2). Assuming that we want the output in sorted order about a specified point (by polar angle with respect to some fixed axis through that point), the visibility from a point problem (i.e.identifying those vertices that are visible from that point) has a lower bound of $\Omega(n \log n)$. This can be established by showing a straightforward reduction from sorting to this problem.

All the visibility algorithms mentioned here (and those that will be described in the coming sections) are described for a set of line segments. The construction of the visibility graph when the input is a set of disjoint polygons can be done with the same complexity (here n is the total number of polygon edges). The edges of the polygons are considered to be the set S, and those graph edges that lie in the interior of the polygons can be eliminated, without any increase in the time complexity.

Welzl [55] and Asano *et al.* [3] 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. Visibility from a point can be found in $O(n \log n)$ time by using a recursive algorithm [6] optimally. If we were to apply this algorithm to each of the endpoints of the segments in S in a straightforward way, we would get an $O(n^2 \log n)$ algorithm. The reason for the $O(n^2)$ time-bound of [3, 55] is the use of a procedure called *line arrangement construction*[4], which can be performed sequentially in $O(n^2)$ time [9, 12]. By exploiting the point-line duality [9, 12] and using the arrangement construction algorithm, we can find for every endpoint p, the sorted order of the other endpoints about p in $O(n^2)$ time. Once this sorted order is obtained, Welzl and Asano *et al.* use different methods to find the final visibility graph.

[4]The *line arrangement* problem is the construction of the planar graph determined by the pair-wise intersections of a set of lines in the plane.

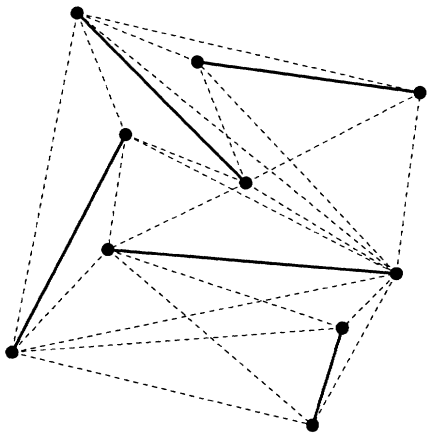


Figure 2: The visibility graph of a set of line segments. The dashed line segments are the edges of the graph.

Welzl computes the visibility graph by first establishing a topologically sorted[5] order on the set of directions determined by p_1 and p_2, where p_1, p_2 are endpoints of segments in S (this can be done by the topological sort of the line arrangement graph determined by the endpoints in the dual space). Then, by stepping through this sorted set and updating the visibility information (this visibility information has been initialized in an appropriate way) at every step, he finally ends up with the visibility graph. Since the number of such pairs is $O(n^2)$, and the topological sort can be done in $O(n^2)$ time, the run-time of this algorithm is also $O(n^2)$. Asano *et al.* use a very different technique for the construction of the visibility graph. Their approach is to solve the problem of visibility from a given point in $O(n)$ time, and they give two different methods for doing this. The first method uses triangulation (of the set of line segments), topological sort and set-union. The second method is a sweep-line technique. Notice that both these algorithms are worst-case optimal. However, it is still not known whether it is possible to compute the visibility graph in such a way that the run-time depends on the size of the visibility graph (e.g. $O(n \log n) + k$ where k is the number of edges in the visibility graph)[6]. As we mention below, neither of the two sequential techniques mentioned here lends itself to optimal parallelization.

2.1.2 Parallel Algorithm on the CREW PRAM

The procedures that are common to both the sequential techniques mentioned above are arrangement construction and topological sorting. Goodrich solves the problem of constructing the arrangement of a set of lines optimally in parallel by using some sophisticated algorithmic technqiues

[5] A *topological sort* of a directed acyclic graph $G(V, E)$ is a mapping $ord : V \rightarrow \{1, \ldots, n\}$ such that for all edges $(v, w) \in E$, $ord(v) < ord(w)$. The topological sort of G can be done in sequential time $O(|V| + |E|)$.

[6] We would like to note that problems involving visibility and shortest paths in *simple polygons* have been widely studied, both in the sequential as well as the parallel setting. In this case, the edges of the simple polygon form the set S. We will not discuss this case here, and refer the interested reader to [1, 5, 13, 14, 17, 18, 53].

[16]. However, topological sorting of a directed acyclic graph cannot be performed optimally by known techniques. Also, the sweep-line method, although a very useful sequential technique, does not seem to be useful in the parallel environment. Hence, the sequential algorithms mentioned above do not appear to lend themselves to parallelization.

The best known parallel algorithm for constructing the visibility graph is offered by Atallah *et al.* in [6], where they use a technique called *cascading divide-and-conquer*. The cascading divide-and-conquer method is a powerful technique for designing parallel algorithms on the CREW and EREW PRAM. It can be applied to problems that are solvable using the divide-and-conquer strategy. The general techniques developed in [6] consist of non-trivial generalizations of the "cascaded merging" method used in the optimal PRAM algorithm for merge sort by Cole [11]. In [6], the authors use cascading divide-and-conquer to come up with optimal parallel algorithms for a wide variety of fundamental problems in computational geometry. In particular, they solve the problem of computing visibility from a point by using a parallel recursive algorithm that runs in $O(\log n)$ time using n processors. By applying this method to every endpoint of the input set of segments, the visibility graph can be constructed in $O(\log n)$ time using n^2 processors. We use this divide-and-conquer method to derive an optimal algorithm for this problem on the mesh in Section 4.

2.2 Voronoi Diagram of a set of Line Segments in the Plane

In the previous section, we noted the relevance of the Voronoi diagram of a set of line segments as a tool for motion planning. We start off with the definition of this diagram, and establish some notation that will be used in the coming sections. Next, we give a summary of the sequential approaches used to construct the diagram; the approaches provide insight into the geometric issues involved in the construction of the diagram. Finally, we will give a brief outline of the known PRAM algorithm [15] for this problem.

2.2.1 Notation, Definitions and Sequential Algorithms

Let S be a set of nonintersecting closed line segments in the plane. Following the convention in [26, 57], 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 [15, 26], 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 $proj(q, s)$, is defined as follows: Let p be the intersection point of the straight line containing s (call this line $\overleftrightarrow{s}$), and the line going through q that is perpendicular to $\overleftrightarrow{s}$. If p belongs to s, then $proj(q, s) = p$. If not, then $proj(q, s) = a$ if $d(q, a) < d(q, b)$ and $proj(q, s) = b$, otherwise. The *distance* of a point q from a closed line segment s is nothing but $d(q, proj(q, s))$. By an abuse of notation, we denote this distance as $d(q, s)$. Let s_1 and s_2 be two objects in S. The

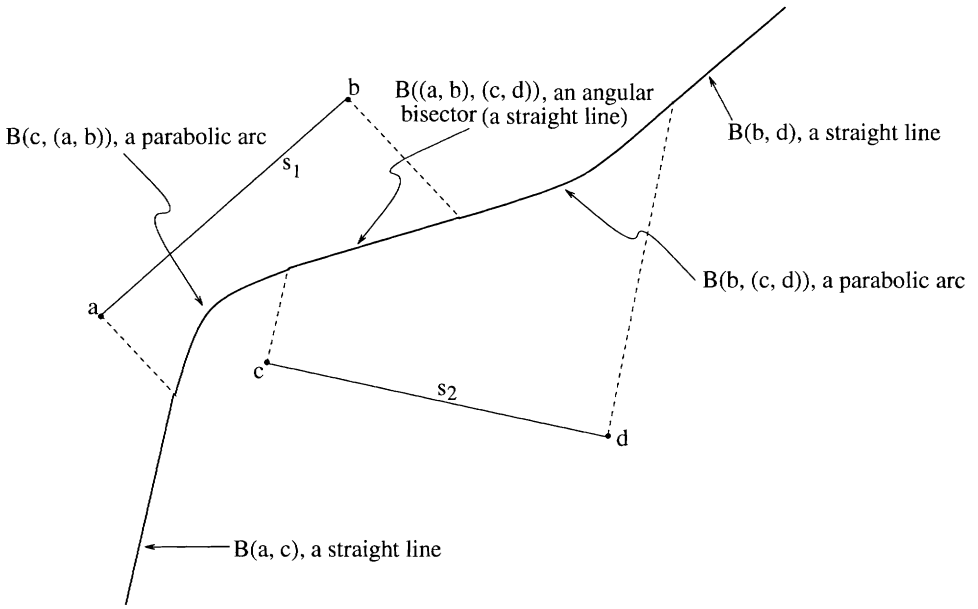


Figure 3: The bisector of two line segments s_1 and s_2.

bisector of s_1 and s_2, $B(s_1, s_2)$, is the locus of all points q that are equidistant from s_1 and s_2 i.e. $d(q, s_1) = d(q, s_2)$. Since the objects in S are either points or open line segments, the bisectors will either be parts of lines or parabolas. The bisector of two line segments is shown in Figure 3. As in [15], let $d(q, S)$ denote the distance between q and the object of S that is closest to q i.e. $d(q, S)$ is $\min_{s \in S} d(q, s)$.

Definition 2.1 *[26] The* Voronoi region, $Vor(e)$, *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. The* Voronoi diagram *of S, $Vor(S)$, is the union of the Voronoi regions $Vor(e)$, $e \in S$. The boundary edges of the Voronoi regions are called* Voronoi edges, *and the vertices of the diagram,* Voronoi vertices.

The Voronoi diagram of a set of segments is shown in Figure 4. Note that a Voronoi region consists of all points q such that $d(q, S)$ is realized by exactly one object s in S, a Voronoi edge consists of all points q such that $d(q, S)$ is realized by exactly two objects of S, and a Voronoi vertex consists of one point q such that $d(q, S)$ is realized by at least three objects of S [15]. The following is a very important property of $Vor(S)$.

Theorem 2.2 (Lee et al. [26]) *Given a set S of n nonintersecting closed line segments in the plane, the number of Voronoi regions, Voronoi edges, and Voronoi vertices of $Vor(S)$ are all $O(n)$. To be precise, for $n \geq 3$, $Vor(S)$ has at most n vertices and at most $3n - 5$ edges.*

There is an important relationship between $Vor(S)$ and the convex hull of S $CH(S)$, which is stated in the following theorem. The algorithms for the construction of $Vor(S)$ make crucial use

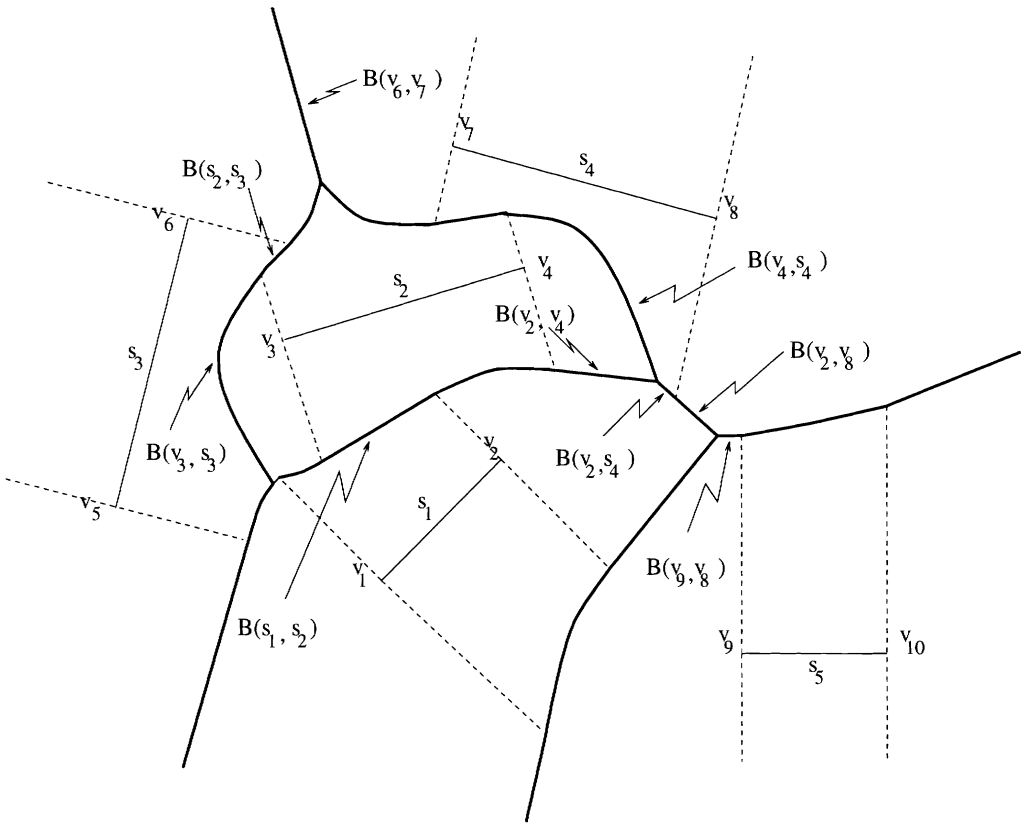


Figure 4: The Voronoi diagram of a set of closed line segments $S = \{s'_1, s'_2, s'_3, s'_4, s'_5\}$. s'_i consists of the two endpoints v_{2i-1} and v_{2i}, and the open line segment s_i. Some of the Voronoi edges have been marked.

of this property.

Theorem 2.3 (Lee et al. [26]) *An object e of S is on the convex hull $CH(S)$ of S if and only if the Voronoi region $Vor(e)$ is unbounded.*

The general idea behind the sequential algorithms for the construction of $Vor(S)$ is as follows: S is divided into sets of equal size, S_1 and S_2. $Vor(S_1)$ and $Vor(S_2)$ are then recursively computed. In order to merge these two Voronoi diagrams to form the final diagram $Vor(S)$, 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 $Vor(S_1)$ that lies to the right of the contour, and that part of the diagram of $Vor(S_2)$ 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 $Vor(S)$. This is the motivation behind the approaches used by [25, 26, 57]. The algorithms of [25, 57] run in $O(n \log n)$ time, which is optimal since a lower bound of $\Omega(n \log n)$ is known for this problem[47]. The run-time of the algorithm in [26] is $O(n\log^2 n)$.

For the case of the Voronoi diagram of a set of points we have to construct just a single contour chain during the merge step. This is because of the fact that we can divide a set of points into two disjoint subsets of equal size such that they are linearly separable[7]. This can be done by sorting the points according to their x-coordinate, say. In the case of a set of line segments, such a linear separability cannot be found in general. Thus, an arbitrary separation of the input set S into S_1 and S_2 could mean that the merge contour is not necessarily composed of a single piece - we could have several disconnected pieces. In [25], Kirkpatrick divides the input set S arbitrarily into two disjoint sets S_1 and S_2 of equal size. In [26], Lee and Drysdale first sort S according to the left endpoint of the segments; S_1 then consists of the first $n/2$ segments and S_2 consists of the last $n/2$. In both these methods, we could have several contours. The contours are constructed by establishing a start point for every contour and, subsequently tracing out the contours by starting at these start points. As observed in [15], both these approaches seem to be inherently sequential in nature. Since our goal is to develop parallel algorithms, our interest is directed more towards the method used by Yap in [57], which uses a different approach to subdivide S.

First, all the endpoints of the segments are sorted. Let m be the median of this sorted set. A vertical line is drawn through m, cutting segments of S into two, if necessary. S_1 then consists of the segments lying to the left of this vertical line and S_2 consists of those lying to the right. Note that this approach simulates linear separability. A naive implementation of this method could

[7]Two sets are said to be *linearly separable* if and only if there exists a hyperplane (in two dimensions, a straight line) that separates them [37].

lead to an $O(n^2)$ algorithm. However, since he restricts the computation in the merge step to the minimum necessary, Yap's algorithm runs in time $O(n \log n)$.

The PRAM algorithm of Goodrich, Ó'Dúnlaing and Yap [15] (which we will summarize briefly in the following subsection) is based on the sequential approach outlined in the preceding paragraph. Since our mesh algorithm, to be presented in Section 5, also relies on this approach, we will give a brief summary of the main ideas in Yap's technique. Let us assume that a vertical line has been drawn through each endpoint of the input set of segments S. The vertical strip of region between any two such (not necessarily adjacent) vertical lines is called a *slab.* Now consider a particular slab U, represented by a pair of vertical lines l_1 and l_2. A closed segment s is said to *span* U if it intersects l_1 and l_2. If we consider all segments that span U, there is a well-defined ordering (according to the point at which they intersect l_1, say) on these segments. The region of slab U that is bounded by any two consecutive spanning segments is called a *quad.* Note that the bottommost and topmost such regions are unbounded. A quad Q in the slab U is said to be an *active quad* if it contains an endpoint of a segment $s \in S$ in its interior (thus endpoints along the boundary edges of Q do not count).

Yap's algorithm does a slab-wise and quad-wise computation of the Voronoi diagram. During the merge step, two adjacent slabs U_l and U_r, whose Voronoi diagrams have been recursively computed, are merged to form a larger slab U. Now, the Voronoi diagram is computed *only for the active quads* of U by using the recursively computed Voronoi diagrams of the active quads of U_l and U_r. As a result, the amount of work done in each slab U is proportional to the number of segments with endpoints in that slab[8], and this is what was meant by performing only the necessary computations. Notice that at the topmost level of recursion, the entire plane is the slab, and there is just one active quad which contains all the segments in S; hence, $Vor(S)$ will be computed, which is the goal. The details of the merge procedure will become clearer in the sections that address the issue of solving this problem in parallel on the mesh-connected-computer.

2.2.2 Parallel Algorithm on the CREW PRAM

The best (and only) known parallel algorithm for the construction of the Voronoi diagram of a set of line segments in the plane is the CREW PRAM algorithm by Goodrich *et al.* [15]. Their algorithm runs in $O(\log^2 n)$ time using $O(n)$ processors, and is based on Yap's sequential algorithm [57] and on the parallel approach used by Aggarwal *et al.* for constructing the Voronoi diagram of a set of points in the plane [2]. This efficient parallelization of Yap's algorithm is because the authors manipulate objects called *primitive regions* to construct the contour in the merge step of their recursive algorithm. In addition, they use the techniques developed in [6] for solving certain

[8] If, instead, the amount of work done in each slab were proportional to the number of segments that span that slab, then we would have had an $O(n^2)$ algorithm.

planar point location problems. In Section 5, we develop a mesh-optimal parallel algorithm for constructing the Voronoi diagram of a set of line segments on the mesh. The salient features of Goodrich *et al.*'s algorithm will be mentioned in that section, since the mesh algorithm is based on their PRAM algorithm.

3 Preliminaries on the Mesh-Connected-Computer

In this section, we will talk about one particular fixed-connection network architecture, the *mesh-connected computer* (*mesh*). In Sections 4 and 5, we describe algorithms on the mesh for the visibility graph construction, the multilocation problem (to be described later), and the Voronoi diagram of sets of line segments in the plane. As mentioned in [31], several large mesh computers have already been constructed. They can be constructed more economically than hypercubes and other parallel architectures because of their simple nearest-neighbor wiring and the ease of scalability.

3.1 The Mesh-Connected Computer

A *mesh-connected computer* of size n is a fixed-connection network of n simple *processing elements (PEs)* that are arranged in a square two-dimensional array (see Figure 5). Following the convention in [31], we will assume that $n = 4^c$ for some constant c. There are $\sqrt{n}$ rows and $\sqrt{n}$ columns in the mesh. For $i, j \in \{0, 1, 2, \ldots, \sqrt{n}-1\}$, $P_{i,j}$ refers to the PE at row i and column j. Each $P_{i,j}$ has a communication link to each of its four neighboring PEs (processors along the sides of the mesh will have fewer), $P_{i\pm1,\, j\pm1}$. These communication links between pairs of processors are not allowed to vary with time (hence the term *fixed-connection network*). Each PE has a constant number of storage registers (each of size $\Omega(\log n)$ bits), and can perform standard arithmetic and boolean operations on the contents of the registers in unit time. The mesh is a SIMD (*S*ingle *I*nstruction stream *M*ultiple *D*ata stream) machine. During each time unit, a single instruction is executed by all the processors (that are specified by the instruction) in parallel. The communication links are unit-time bidirectional links; in other words, a PE can send or receive at most one word of data from each of its neighboring PEs in one unit of time, and this can be achieved through routing instructions. Concurrent data movement in the mesh is allowed, as long as it is all in the same direction.

It is important to note that an implicit lower bound of $\Omega(\sqrt{n})$ on run-time holds for most algorithms on the mesh. This is because of the following: The *distance* between a pair of processors is the smallest number of wires that have to be traversed in order to get from one processor to another, and the *diameter* d of a network is the maximum distance between any pair of processors [27]. The diameter of a network is often a lower bound on the run time of an algorithm on that network, since it is always possible to come up with data arranged in such a way that there needs to be an exchange of information between two processors that are separated by a distance d, the diameter. It takes at least d steps for one of these processors to communicate with the other. The diameter of a $\sqrt{n} \times \sqrt{n}$ mesh is $2\sqrt{n} - 2$, which is the distance between the two processors at the opposite corners of the mesh. Hence most algorithms on the mesh will have a lower bound of $\Omega(\sqrt{n})$.

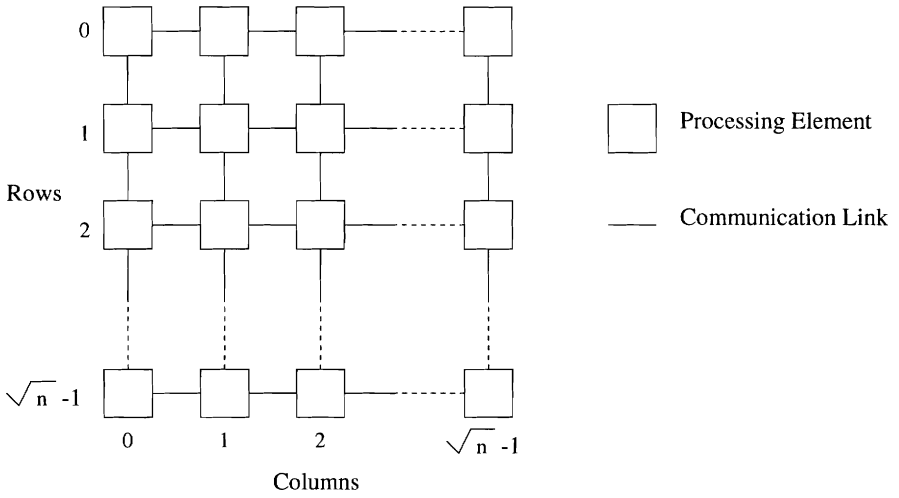


Figure 5: A mesh-connected computer of size n.

Each PE contains its row and column numbers, and also an identification register. The contents of the identification register depend on the particular *indexing scheme* that is being used for the mesh. The useful and commonly used ways of indexing processors are: *row-major indexing*, *shuffled row-major indexing*, *snake-like row-major indexing* and *proximity order indexing*, as illustrated in Figure 6 for a 4 $\times$ 4 mesh. The different indexing schemes have their own advantages and which particular one we choose to use will depend on the problem that we are trying to solve. For instance, as mentioned in [54], row-major indexing is a poor choice for merge sorting. Snake-like ordering has the useful property that PEs with consecutive numbers are physically adjacent in the mesh. Shuffled row-major and proximity order indexing schemes are used when divide-and-conquer approaches are being used. This is because of the fact that in a mesh of size n, the processors with the first $n/4$ of the numbers lie in the first quadrant of the mesh, the second fourth in the second quadrant etc. and this property recursively holds within each quadrant. Proximity order combines the advantages of both snake-like ordering as well as shuffled row-major ordering. Observe that it is possible to generate the number of a processor, in any of these indexing schemes, in $\theta(\sqrt{n})$ time. In row-major indexing, each processor can compute its number for the row and column indices (assuming it knows the size of the mesh, which can be found in $\theta(\sqrt{n})$ time). Snake-like indexing is obtained from row-major indexing by reversing the order in even rows, which can be done in the stated time bound. The number of a processor in the shuffled row-major indexing scheme is nothing but the shuffle[9] of the binary representation of that processor in the row-major indexing scheme. For example, the row-major index 10 in Figure 6(a) has binary representation 1010, the shuffle of which is 1100, and this is the shuffled row-major index of that same processor.

[9]The *shuffle* of "abcdefgh" is "aebfcgdh" and the *unshuffle* of "abcdefgh" is "acegbdfh".

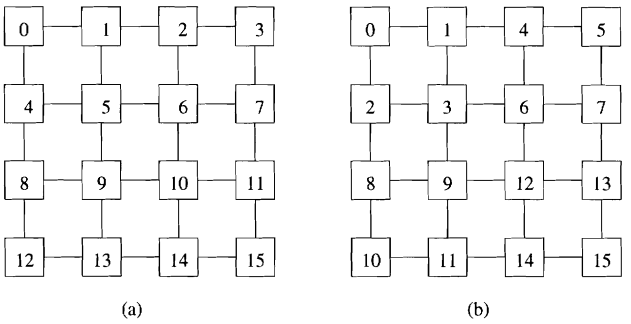


Figure 6: Indexing schemes on the mesh. (a) Row-major (b) Shuffled row-major (c) Snake-like row-major (d) Proximity

3.2 Useful Operations on the Mesh

We will now describe some of the basic operations on the mesh, concentrating on those that will be heavily used in the following subsections. The *perfect shuffle* and *perfect unshuffle* data-routing operations are commonly used in mesh algorithms. Obviously, when a shuffle and unshuffle operation are performed in sequence on a set of elements, we get back the elements in the original order (the order on the mesh will be determined by the particular indexing scheme that we use). Notice that on a linear array[10] of size k, the shuffle can be achieved by using the triangular interchange pattern, as shown in Figure 7 (the double headed arrows indicate an interchange, which takes two routing steps). The unshuffle can be done in a similar manner by using an inverted triangular interchange pattern [54]. Thus, the perfect shuffle as well as the perfect unshuffle operations can be done in $k/2 - 1$ interchanges, i.e. $k - 2$ routing steps on a linear array of size k. We can use this result to do the shuffle and unshuffle operations on the mesh efficiently. Consider the case of row-major indexing. To perform a shuffle, first do a shuffle along each row in parallel, then do a shuffle along each column in parallel (each row and each column is a linear array, so use the shuffle method described earlier). Now, for all $i \in \{0, 2, \ldots, \sqrt{n} - 2\}$, $j \in \{1, 3, \ldots, \sqrt{n} - 1\}$, the element in $P_{i,j}$ needs to be interchanged with the element in $P_{i+1,\, j-1}$, which can be done in 4 steps. Now the elements of the mesh are in shuffled order. See Figure 8 for an illustration. The unshuffle can be obtained by reversing the order of steps just described (and replacing the shuffles by unshuffles). Thus both these operations can be performed on a $\sqrt{n} \times \sqrt{n}$ mesh in $2\sqrt{n}$ steps. *Sorting* will be used often in our mesh algorithms, either as a preprocessing step or as a sub-step in a recursive merge procedure. In [54], the authors gave one of the first algorithms for sorting optimally on the mesh. Currently, the best known algorithm for sorting n elements distributed one per processor on a $\sqrt{n} \times \sqrt{n}$ mesh takes $3\sqrt{n} + o(\sqrt{n})$ steps [[27], Chapter 1, Section 1.6.3].

In addition to these basic operations on the mesh, there are a number of others that are com-

[10] A *linear array of size k* is nothing but k linearly connected processors, so that every processor (except the ones at either end) is connected to its left and right neighbors.

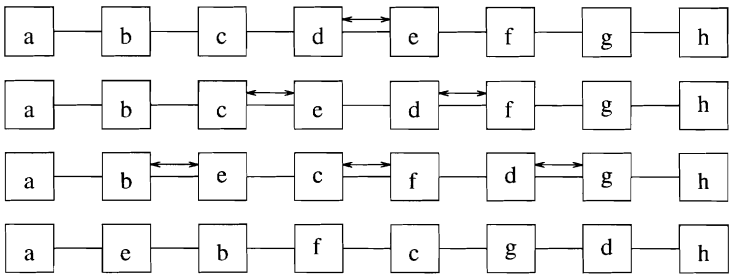


Figure 7: Shuffle on a linear array.

monly used in mesh algorithms. Some of these are *Selected Broadcasting*, *Segmented Prefix Scan*, *Random Access Read (RAR)* and *Random Access Write (RAW)*. The RAR and RAW operations allow the mesh to simulate the concurrent read and concurrent write capabilities of PRAMs. The implementations of these operations are described in detail in [31, 32]. We will just describe what these operations do. As given in [32], in the *RAR* operation, each PE P_i, $0 \leq i \leq n-1$, contains some index S_i, and wants to receive data from some register(s) of P_{S_i}. If P_i is not to receive data, then $S_i = \infty$. In *RAW*, each PE P_i contains some index W_i. Data from some register(s) of P_i is to be transmitted to P_{W_i}. If $W_i = \infty$, then no data from P_i is transmitted to any PE. *Selected broadcasting* is the following operation [22]: Let $\{a_1, a_2, \ldots, a_k\}$ be some subset of elements on the mesh (not necessarily in consecutive processors). Let the index of the processor in which a_i resides be $I(a_i)$. $P_{I(a_i)}$ contains, along with a_i, an index $S(a_i)$. Also, $I(a_i) < I(a_{i+1})$, $S(a_i) < S(a_{i+1})$, $1 \leq i \leq (k-1)$. Selected broadcasting sends each a_i to all the processors from $P_{S(a_i)}$ to $P_{S(a_{i+1})-1}$. The *prefix scan* operation is the following: Let $A = \{a_1, a_2, \ldots, a_n\}$ be a set of elements such that PE P_i has element a_i in it, and let $\otimes$ be some binary associative operation. The prefix scan of the elements of A is the set of elements $B = \{b_1, b_2, \ldots, b_n\}$ where $b_1 = a_1$ and $b_i = a_1 \otimes a_2 \otimes \ldots \otimes a_i$ for $2 \leq i \leq n$. At the end of the prefix scan operation on the mesh, element b_i will be in PE P_i. This can be done in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh. In the *segmented prefix scan* operation, we are interested in computing the prefix with the same associative operator $\otimes$, but on different sets of data. Let $A_1, A_2, \ldots, A_k$ be k sets of elements with $|A_i| = l_i$, $1 \leq i \leq k$, and such that $l_1 + l_2 + \ldots + l_k = n$. A_1 resides in the first l_1 PEs of the mesh, A_2 in the next l_2 PEs and so on. The segmented prefix scan operation will compute the prefix scan of the set A_i for each i, and place the resulting prefixes in the corresponding PEs that hold the elements of A_i. The set of consecutive PEs that hold A_i is referred to as the i-th *component* for the segmented prefix scan. RAR, RAW, selected broadcasting, and segmented prefix scan[11] can all be done in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh.

We explain briefly how the strategy of divide-and-conquer is applied on a $\sqrt{n} \times \sqrt{n}$ mesh [22]: The problem is divided into two halves, one in the top half of the mesh and one in the bottom half.

[11] The Connection Machine, manufactured by Thinking Machines, Inc., provides the segmented prefix scan as a primitive operation.

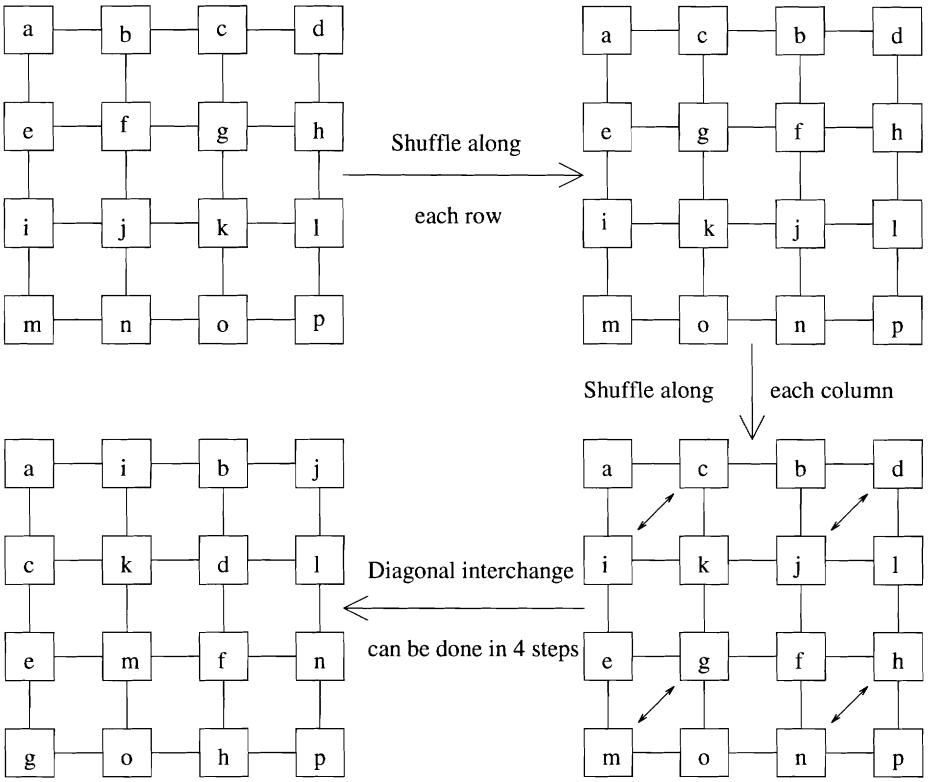


Figure 8: Shuffle on a mesh with row-major indexing.

Each of these halves is then recursively solved, and then merged to give the final solution. An easier solution to the recurrence relation for the run-time is obtained if we assume that the top half and bottom half are further divided into two. Each of the four quadrants is solved in parallel. The top (bottom) two quadrants are merged to give us the solution to the top (bottom) half; this is done in parallel for the top and bottom halves. These two halves are now merged to give us the final solution. Let the time for the first merge (merge1) be T_1 and that for the second merge (merge2) be T_2. The recurrence relation for the total run-time is $T(\sqrt{n}, \sqrt{n}) = T(\sqrt{n}/2, \sqrt{n}/2) + T_1 + T_2$. If $T_1 + T_2$ is $O(\sqrt{n})$ then the solution to this recurrence is $T(\sqrt{n}, \sqrt{n}) = O(\sqrt{n})$. Thus, our main concern when designing divide-and-conquer algorithms on the mesh will be to come up with merge steps that run in $O(\sqrt{n})$ time. The method used for the first and the second merge will be similar. Thus, for the sake of simplicity, in our algorithms we will discuss the merge step in general without any reference to merge1 or merge2.

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. We also noted in Section 2 that the best known parallel algorithm for this problem is on the CREW PRAM using the cascading divide-and-conquer approach [6]. To our knowledge, this problem has not been solved on the mesh. In this section, we design an optimal algorithm for this problem. 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 [6], 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 [6]. The lower envelope is the collection of those segment parts that can be seen from below.

In [6], 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 (see Figure 9). 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. Our input set consists of a set S of segments along with their endpoints (these total n). As mentioned earlier, segment s_i, $i \in \{0, 1, \ldots, N-1\}$ has endpoints v_{2i} and v_{2i+1}, with $x(v_{2i}) < x(v_{2i+1})$.

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.

1. *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.

2. *Basis:* **lowerseg** is set to the segment $s_{i/2}$ if i is even and to $\emptyset$ otherwise[12]. Let S_1 be the subset of segments of S in the left block, and S_2 be the subset in the right block.

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

4. *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 **lowerseg** fields of the two old slabs u_1 and u_2 that it is a part of. How do we find these two **lowerseg** fields? Let u be a slab

[12]Initially, the slabs are those defined by each individual segment, and hence the lowest segment in that slab is nothing but the segment itself.

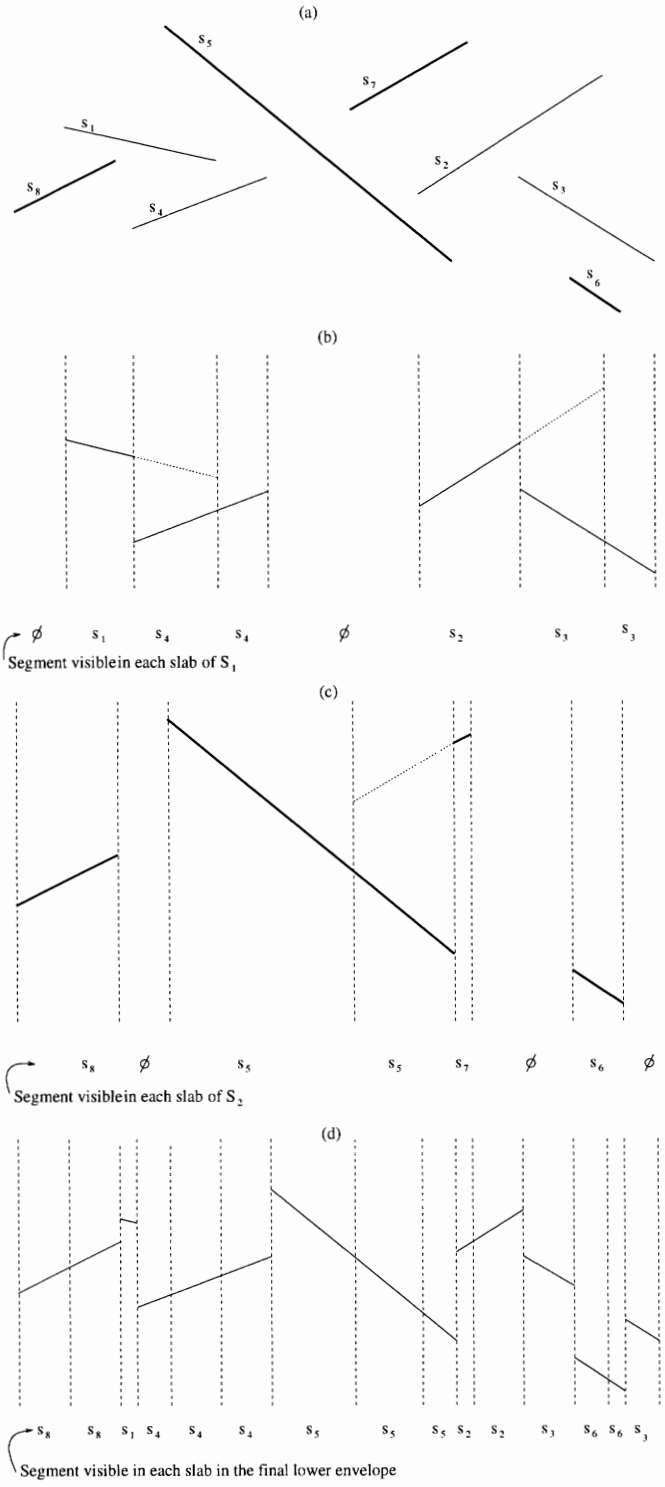


Figure 9: Merging of lower envelopes. (a) The set S with $S_1 = \{s_1, s_2, s_3, s_4\}$ (the light line segments) and $S_2 = \{s_5, s_6, s_7, s_8\}$ (the dark line segments). (b) The recursively computed lower envelope of S_1. (c) The recursively computed lower envelope of S_2. In (b) and (c), the hidden segment parts are dotted. (d) The final lower envelope.

in the merged set (it is represented by the **endpoint** field) and suppose it is in PE P_i. One of the two **lowerseg** fields is already in P_i. This is because **endpoint** in P_i represents either the left boundary of u_1 (the slab from S_1 that u is a part of) or the left boundary of u_2 (the slab from S_2 that u is a part of). Let us say it is u_2. The other **lowerseg** has to come from the PE containing u_1: call it P_j; note that j will be less than i. Note, of course, that the endpoint in P_j is the rightmost of all the endpoints of S_1 that lie to the left of the endpoint in P_i. In other words, there are no endpoints of S_1 between P_j and P_i. In fact, the **lowerseg** of P_j is to be broadcast to all the PEs that lie between P_j and the PE containing the right end point of u_1. We need to do this for every slab u_1 of S_1. A similar step needs to be done for every slab u_2 of S_2. This can be achieved through the *selected broadcasting* operation. Refer to Figure 10 for an illustration of these steps.

(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**s in sorted order and let I_{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 $P_{I(l_i)}$ to $P_{I(l_{i+1})-1}$. 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**.

Lemma 4.1 *Algorithm* VISFROMPOINT, *which computes the lower envelope of a set of segments S, runs in $O(\sqrt{n})$ time (with no queueing) on a $\sqrt{n} \times \sqrt{n}$ mesh, where $|S| = n/2$.*

Proof: Since there is no preprocessing, our timing analysis is just for the merge step. The first step of the merge procedure can obviously be done in constant time (this step can be done by looking at the appropriate bit in the binary representation of the PE's id.). Step (ii) can be done in $O(\sqrt{n})$ time using the standard shuffle and exchange technique[54]. Both step (iii) and step (iv) need $O(\sqrt{n})$ time for the selected broadcasting. Step (v) takes constant time. Therefore, the merge procedure takes $O(\sqrt{n})$, and this immediately implies that the overall time required is $O(\sqrt{n})$. □

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

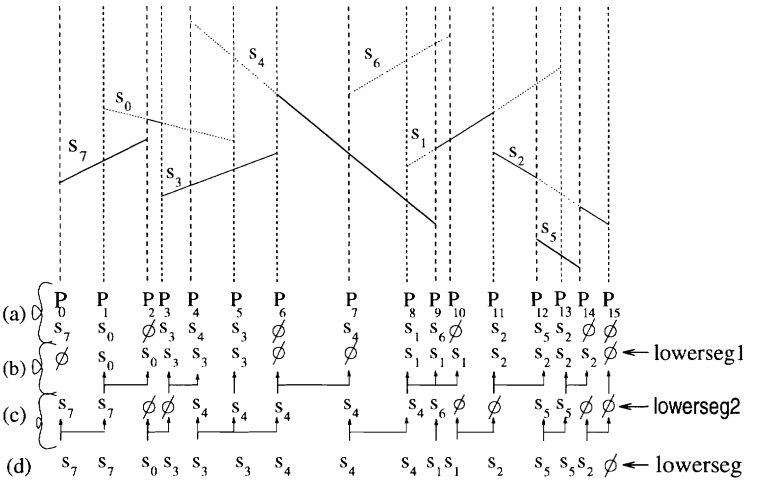


Figure 10: Computing **lowerseg** in the merge step. As before, the lighter segments form S_1 and the darker ones form S_2. (a) The value of **lowerseg** in each PE after step (ii) of the merge step. (b) The values of **lowerseg1** after step (iii) of the merge step. (c) The values of **lowerseg2** after step (iv) of the merge step. (d) The final **lowerseg** value in each PE, as found in step (v).

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 $\theta(\sqrt{n})$ 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

We summarized, in Section 1, the method of Lozano-Pérez and Wesley [30] for planning the motion of a convex object with two *dofs*, moving between convex obstacles. Assume the size of the obstacle set is n (i.e. the obstacle set has n endpoints) and it is stored in a $\sqrt{n} \times \sqrt{n}$ mesh. Let the mobile object be A; we will assume that the size of A is a constant. First we expand the obstacles according to the moving object A as described in Section 1 (Subsection 1.2.1): We relay the information about A to each of the PEs in $\sqrt{n}$ time. Since the expansion of each obstacle can be done in time proportional to its size (refer to Section 1), 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 $O(n)$ [48]. Thus the new obstacle edge set will also be $O(n)$ and there are efficient sequential algorithms to compute it [48]. 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 A 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[13]. 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[14], 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 [27], the all-pairs shortest path problem can be solved in $O(n)$ 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 $O(n)$ 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.

[13] Note 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.

[14] Consider 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 Mesh Algorithms for the Voronoi Diagram of a Set of Line Segments and the Related Motion Planning Problem

We start this section with the definition of a problem called the *Multipoint Location* problem, which is an important subroutine for the construction of the Voronoi diagram. We also give a brief description of the mesh algorithm for this problem, given by Jeong and Lee [22] and give a different approach to one of the sub-steps used in the algorithm in [22]. Following that, we 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 [34] is given in the last part of this section.

5.1 Multipoint Location and why it is important

In this section, we will discuss the problem of *Multipoint Location.* Multipoint location comes under the class of problems called *planar point location* problems. In this class of problems, we are given some planar subdivision[15] S consisting of n edges, and a query point p, and we are to find the region of S that p lies in. The main reason for discussing this problem here is that it has very important applications for the problem to be discussed in the next section, viz. the problem of the construction of the Voronoi diagram of a set of line segments in the plane. Before moving on, let us give the exact statement of the multipoint location problem.

Problem Statement: Given a set S of nonintersecting line segments, and a set P of points, where $|S| + |P| = n$, to find, for every point $p \in P$, the segments p^a, $p^b \in S$ that lie immediately above and below p, respectively. If there is no segment that lies immediately above (below) p, p^a (p^b) is set to $s_{+\infty}$ ($s_{-\infty}$), an imaginary segment lying at $+\infty$ ($-\infty$).

An $O(\sqrt{n})$ algorithm for solving this problem on a $\sqrt{n} \times \sqrt{n}$ mesh is given by Jeong and Lee [22]. The problem is solved for three cases of input: in the first case, the line segments have the same x-coordinates for the left and right endpoints, in the second case they have the same x-coordinate for the left endpoints, and the third is the general case. The first is used as a substep in the second, and the second is used as a substep for the general case. We describe a different approach for solving the first case. While the approach is not asymptotically faster than Jeong *et al.*'s algorithm, the constant factor of our algorithm is significantly smaller[16]. We mention this improvement primarily because the first step is used repeatedly as a substep for general multipoint

[15] A *planar graph* is one that can be embedded in the plane without crossings of edges. A *planar subdivision* is the subdivision of the plane induced by a connected planar graph. For our considerations, the vertices of the graph will be points in the plane, and the edges of the graph will be straight line segments. In the case of the Voronoi diagram of line segments, some of these edges might be parts of parabolas.

[16] The method used in [22] has a high constant because it uses a recursive method, and uses RARs in its merge procedure. RARs are expensive operations on the mesh.

location, and in the construction of the Voronoi diagram of a set of line segments in the plane. Hence, the improved algorithm is more suitable for possible practical implementations.

Let us consider the case when the nonintersecting segments of the input set S have the same x-coordinates for the left and right endpoints, say lx and rx respectively. Let the set of points P be such that they lie between the lines $x = lx$ and $x = rx$. We will assume that the mesh has shuffled row-major indexing, since the other two cases are solved by recursive algorithms. The elements are distributed so that there is either one segment or one point per PE. First, we will arrange the elements so that they are in row-major ordering, and then sort the elements so that the segments are in sorted order (by the decreasing y-coordinate of their left endpoint, say) and occupy the first $|S|$ PEs. Let s_i, $0 \leq i \leq |S| - 1$ be the i-th segment in this sorted order; s_i is in PE P_i. The point set P occupies the next $|P|$ processors. The following is the basic idea behind this method: Let us assume that $|S| > \sqrt{n}$, since the other case is straightforward[17]. For every point p, we want to perform a search for segments p^a and p^b on the mesh. Let the segments given by $s_{-\sqrt{n}}$, $s_{-\sqrt{n}+1}$, ..., s_{-1} be dummy segments (initialized to the segment at $+\infty$) for points lying above the topmost segment in each column. Let the segments given by $s_{|S|}$, $s_{|S|+1}$, ..., $s_{|S|+\sqrt{n}-1}$ be dummy segments (initialized to the segment at $-\infty$) for points lying below the lowermost segment in each column.

Suppose a point $p \in P$ is in some column j of the mesh. If we send p up along its column, then there are exactly two segments, $s_{i-\sqrt{n}}$ and s_i (for some $0 \leq i \leq |S| + \sqrt{n} - 1$), in that column between which it lies (since the segments are ordered). Thus, p has to examine just another $\sqrt{n}$ segments in order to determine p^a and p^b. This could be done by letting p sit in PE P_i, and by passing all these $\sqrt{n}$ segments down that row so that p can determine p^a and p^b. In order to do this efficiently we need to ensure that not too many points end up in the same row. We do this by counting how many points belong in a row. Then we make enough copies of that row so that when we send a point to a processor in the appropriate row, there is at most one point per processor. Note that since the total number of points is $O(n)$, the maximum number of new copied rows that we need to make is $O(\sqrt{n})$, and hence we can overlay the new copies of rows on the existing mesh with a constant factor increase in memory per processor. This can be achieved as follows (assume that p^a and p^b have been initialized to $s_{+\infty}$ and $s_{-\infty}$, respectively, for every p).

1. Pass each segment down its column so that every point p knows the segments of that column that lie immediately above and below it. Call these u_{temp} and l_{temp}, respectively.

2. Pass each point up along its column j so that, at the end of $\sqrt{n}$ steps, PE P_i in column j knows the number of points from that column that lie above s_i and below $s_{i-\sqrt{n}}$. Let

[17] If the number of segments is less than or equal to $\sqrt{n}$, then all we have to do is make copies of the segments in the first row in every row, and then pass each segment through its entire row so that p can determine p^a and p^b. This can obviously be done in $3\sqrt{n}$ time.

a_{k0}, a_{k1}, ..., $a_{k(\sqrt{n}-1)}$ be these numbers in row k.

3. Do a parallel prefix on a_{k0}, a_{k1}, ..., $a_{k(\sqrt{n}-1)}$ (from left to right) along each row k in order to determine the total number of points in each row. Let l_{k0}, l_{k1}, ..., $l_{k(\sqrt{n}-1)}$ be the result of this parallel prefix in row k. Obviously, the total number of copies of row k that we need is $\left\lceil \frac{l_{k(\sqrt{n}-1)}}{\sqrt{n}} \right\rceil$. Call this number $numcop_k$.

4. We now know how many copies of a particular row we need to make, but we don't know where to start making them, i.e. at which row of our $\sqrt{n} \times \sqrt{n}$ mesh. This can be determined by doing a parallel prefix on $numcop_k$ along the leftmost column. Call the result of this prefix computation $whichrow_k$, in row k. Propagate the $whichrow_k$ and $numcop_k$ fields down row k.

5. Now we need to figure out the row and column index of the processor to which we want to send each point. From this we can determine the id of the PE to which that point needs to be sent. This can be done by sending each segment in location (k, j) down its column j, along with the $whichrow_k$, $numcop_k$, a_{kj} and l_{kj} fields. Each PE with a point in it can use this information, along with the u_{temp} and l_{temp} computed in step 1, to determine the row and column index of the processor that it should be sent to.

6. We now make $numcop_k$ copies of each row k. Before doing this, each segment notes down the segment that lies in the processor immediately above it, and forms a segment pair. This is necessary in the final step to determine p^a and p^b. The $numcop_k$ copies of each row (i.e. the segment pair in each row) can be made in at most $\sqrt{n}$ steps by as many downward pulses of each row.

7. Send the points to the PE as determined in step 5. This can be done by a sort step.

8. Send the segment pair of each row down that row, once in either direction, and we can determine p^a and p^b for every point p.

9. The elements can be sent back to their original configuration by a sort step.

The time taken to perform the above steps is $O(\sqrt{n})$ with the constant factor being a significant (about 10-fold) improvement over that of the method given by Jeong and Lee [22]. Note that in the above technique, we are essentially doing a simultaneous search of all the points among the line segments. We will refer to this as Algorithm SIMULTSRCH.

After solving the above special case, Jeong and Lee [22] solve the case in which all the segments in the segment set have the same left endpoint. This is done recursively, and the above special case is used as a subroutine in the merge step. They call this Algorithm LB_MULTILOC, and it runs in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh. Finally, they solve the general case of multipoint location by using a recursive algorithm, Algorithm MULTILOC. MULTILOC uses LB_MULTILOC

as a subroutine in the merge step, and runs in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh. At the end of Algorithm MULTILOC, each point p in P will know segments p^a and p^b, along with the index of these segments.

As we mentioned earlier, an important application of multipoint location is the problem of planar point location, which we stated earlier. Given a planar subdivision, and a set of points P, we find for each point p the region of the subdivision that p lies in. In particular, we are interested in performing planar point location for a set of points, given the planar subdivision induced by the Voronoi diagram of a set of line segments. This can be done by almost a direct application of the MULTILOC algorithm of [22], with some minor modifications. We state the pertinent lemma from [22].

Lemma 5.1 (Jeong and Lee [22]) *Given a planar subdivision PS with a set S of edges and a set P of query points, the planar point location can be executed in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh, where $|S| + |P| \leq n$.*

In the case when PS is the Voronoi diagram of a set of n line segments, $|S|$ is $O(n)$ since the number of Voronoi edges is $O(n)$.

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

The Voronoi diagram is a very useful geometric structure, with applications to varied problems in computational geometry. In particular, as we discussed in Section 1, the Voronoi diagram of a set of line segments turns out to be a useful tool in motion planning [34, 33, 56]. We are interested in the parallel construction of the Voronoi diagram. In Section 2, we briefly described the PRAM algorithm of [15] for this problem. In this section, we will 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 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 [22], but none (to our knowledge) for n line segments.

Let us recapitulate some of the important issues in the Voronoi diagram construction. We will use the notation established in Section 2; $Vor(S)$ refers to the Voronoi diagram of a set of elements S, and $Vor(e)$, $e \in S$ refers to the Voronoi region associated with the element e. The usual method is to divide the input set S into two sets of equal size S_1 and S_2, recursively compute the Voronoi diagram of each half, and then merge the two resulting diagrams to form the final Voronoi diagram. The merge step involves the construction of the *contour*, which is the locus of all points that are equidistant from S_1 and S_2. The contour will give us information about (a) the new Voronoi edges that need to be added to the final diagram and (b) which of the edges of

the recursively computed diagrams need to be discarded. Thus, the construction of the contour is the single most important step. 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 [22], the authors exploit this property by first identifying those Voronoi edges of $Vor(S_1)$ and $Vor(S_2)$ 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.

The parallel construction of the Voronoi diagram of a set of line segments is much more involved. As observed in Section 2, the sequential algorithms of [25, 26] do not lend themselves well to parallelization. In that section, we also noted that the best (and only) known parallel algorithm for the construction of the Voronoi diagram of a set of n line segments is the CREW PRAM algorithm by Goodrich *et al.* [15], which runs in $O(\log^2 n)$ time and uses $O(n)$ processors. This algorithm is based on the approaches in the sequential algorithm by Yap [57], and on some of the techniques of the CREW PRAM algorithm for the Voronoi diagram of a set of points [2]. In the remainder of this section, we will show that Voronoi diagram of a set of N line segments in the plane can be constructed in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh, where $n = 2N$. Our method on the mesh is based on the approach used in [15].

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). As before, 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_j is some v_i and $n = 2N$). The mesh algorithm for constructing $Vor(S)$ will be a divide-and-conquer algorithm, and so we will assume shuffled row-major indexing on the mesh. Let U be a slab[18]. The subset of E in the interior of U will be referred to as E_U (thus, endpoints lying on the vertical boundaries of U do not count). The set of segments obtained by restricting S to the slab U will be called S_U i.e. $S_U = \{s \cap U \mid s \in S \text{ and } s \cap U \neq \emptyset\}$. Recall that Yap's algorithm [57] does a slab-wise and quad-wise computation of the Voronoi diagram. 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, $Vor(S_U \cap Q)$. Following [15], we let $VorSet(S_U)$ represent the set containing the Voronoi diagrams of all the active quads Q of U i.e. $VorSet(S_U) = \{Vor(S_U \cap Q) \mid Q \text{ is an active quad of } U\}$. At the topmost level of recursion, the

[18] We recall some definitions. 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. A quad is said to be an *active quad* if it contains an endpoint of S in its interior.

entire plane is the slab U, and the algorithm computes $Vor(S)$, since $VorSet(S_U)$ is nothing but $Vor(S)$. The recursion bottoms out when a slab has just one point in its interior, which happens when a slab is defined by the two vertical lines $x = p_i$ and $x = p_{i+2}$, for all even i between 0 and $n-3$. p_{i+1} is the point in the interior of the slab.

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, which will be the key, $s_{i/2}$ and v_{i+1}, if i is even. If i is odd, these will be v_i, $s_{(i-1)/2}$ and v_{i-1} respectively[19]. In either case, 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, 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. As mentioned in the description of Algorithm MULTILOC, ${p_i}^a$ 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 point in the interior of the slab. This point will be p_i, for odd i, $1 \leq i \leq n-1$. The slab that p_i lies in is defined by the vertical lines going through p_{i-1} and p_{i+1} (p_n is some dummy point that lies to the right of all points in E). The 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_l and U_r be two adjacent slabs, and let $|E_{U_l}| = |E_{U_r}| = k$ (i.e. each slab has k endpoints in its interior). Suppose that $VorSet(S_{U_l})$ and $VorSet(S_{U_r})$ 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

[19] When we say that a particular segment s_j is stored in PE P_i, we mean that the index j of that segment is stored. We will, however, continue to refer to this as "storing the segment s_j".

manner.

(1) *Active Quads of U_l :* The active quads in U_l have a sorted order defined on them in the natural way. Let A_l be the number of active quads in U_l ($A_l \leq k$); let these be Q_{l1}, Q_{l2}, ..., Q_{lA_l} in sorted order (from top to bottom, say). Let the number of endpoints in these active quads be k_{l1}, k_{l2}, ..., k_{lA_l}, respectively. Note that $k_{l1} + k_{l2} + \ldots + k_{lA_l} = k$. In M_l, the endpoints in Q_{l1} are in the first k_{l1} processors, the endpoints in Q_{l2} are in the next k_{l2} processors and so on. We will call this the *active-quad-wise ordering* of the endpoints of E_{U_l}. Each endpoint in Q_{li} will specify its quad by the upper and lower bounding segments of Q_{li}.

(2) *Voronoi Edges of $VorSet(S_{U_l})$:* As stated earlier, $VorSet(S_{U_l})$ is the collection of the Voronoi diagrams of all the active quads in U_l. Because of the quad-wise computation of the Voronoi diagram, the Voronoi edges of $VorSet(S_{U_l})$ are stored in a quad-wise manner. In other words, in M_l, we will first have the Voronoi edges of $Vor(S_{U_l} \cap Q_{l1})$, followed by the edges of $Vor(S_{U_l} \cap Q_{l2})$, and so on. Notice that since $VorSet(S_{U_l})$ consists of the Voronoi diagram of at most k line segments (since only the active quads are considered), it will have $O(k)$ 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 [[57], Lemma 5]: The number of Voronoi edges in the Voronoi diagram of an active quad Q_{li} of U_l is proportional to the number of segments in that quad. In other words, the number of Voronoi edges in $Vor(S_{U_l} \cap Q_{li})$ is $O(k_{li})$[20]. Therefore, the PEs of M_l that store active quad Q_{li} suffice to store the complete diagram $Vor(S_{U_l} \cap Q_{li})$, 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}, $1 \leq i \leq A_r$. The information about the active quads of U_r and the Voronoi edges of $VorSet(S_{U_r})$ are available in M_r in a similar and analogous way.

Following the PRAM technique of Goodrich *et al.* in [15], we first give a concise summary of the steps involved (and the mesh operations needed) in performing the sub-problem merge. In the final part of this section, we give the details of the implementation of each of these steps on the mesh.

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_l and U_r available in M_l and M_r, respectively.

[20]Intuitively 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 $Vor(S_{U'} \cap Q_1)$ will not be affected by the segments in $S_{U'} \cap Q_2$. Hence the assertion that the number of edges in $Vor(S_{U_l} \cap Q_{li})$ is $O(k_{li})$.

This can be done by an appropriate sort step, followed by a selected broadcasting step on the mesh $M_l \cup M_r$. This takes $O(\sqrt{k})$ time on the mesh $M_l \cup M_r$ (which has $2k+2$ PEs).

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 not be (see Figure 12 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, $Vor(S_U \cap Q)$, 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 quad Q of U, the Voronoi diagram of $S_{U_l} \cap Q_l$, called the Q_l-diagram and of $S_{U_r} \cap Q_r$, called the Q_r-diagram. Notice that the Voronoi diagram of the non-empty Q_l-quads (Q_r-quads) has already been recursively computed. The Voronoi diagram of the empty Q_l-quads (Q_r-quads) is easy to compute. On the mesh, we can find the empty Q_l-quads and their Voronoi diagrams by doing an appropriate sort step, followed by a segmented prefix scan operation. An analogous application of these steps give us the empty Q_r-quads and their Voronoi diagrams. The construction of the Q_l-diagram (Q_r-diagram) requires us to merge together the Voronoi diagrams of *all* the Q_l-quads (Q_r-quads), empty as well as non-empty. As will be seen in the detailed version of this step, this merging turns out to be very straightforward. This step takes $O(\sqrt{k})$ time on $M_l \cup M_r$.

(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_l- and the Q_r-diagram, which involves the construction of the contour. The basic objects of manipulation in this step are *primitive regions* or *prims* [15], which will be defined in the detailed version of this step. The horizontal merge is the most complicated part of this algorithm, and depends on certain crucial lemmas developed in [15]. Once the contour is constructed, the Q_l-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. The mesh implementation of this step requires the application of the planar point location algorithm (Lemma 5.1), and a sort step. The final determination of the contour requires the application of Algorithm SIMULTSRCH (with some minor modifications), a prefix scan operation, one routing step, one selected broadcasting step, and one RAR. This takes $O(\sqrt{k})$ time on $M_l \cup M_r$.

As mentioned earlier, 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. From the recurrence relation for divide-and-conquer algorithms on the mesh, given in Section 3.2, 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 lemma.

Lemma 5.2 *The Voronoi diagram of a set of n nonintersecting (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 queueing).*

We now provide details of the implementation of the merge step on the mesh. These details can be skipped without any loss of continuity; the summary of the motion planning algorithm on the mesh is given in section 5.3.

5.2.1 Details of the Merge Step on the Mesh

We will now give details of the method to compute $VorSet(S_U)$, using the recursively computed $VorSet(S_{U_l})$ and $VorSet(S_{U_r})$ (where U is the slab obtained by merging the adjacent slabs, U_l and U_r). In other words, we will describe the details of how each of the three substeps outlined above are executed on the mesh. Without loss of generality, let us assume that the PEs in the mesh $M_l \cup M_r$ are $P_1, P_2, \ldots, P_{2k+2}$.

Determination of the Active Quads of U : As before, let $Q_{l1}, Q_{l2}, \ldots, Q_{lA_l}$ be the active quads of U_l in sorted order. Each such quad Q_{li} is defined by an upper and lower bounding segment. Call these segments ${Q_{li}}^a$ and ${Q_{li}}^b$, respectively. If the upper bounding segment does not exist for Q_{l1}, ${Q_{l1}}^a$ is set to $s_{+\infty}$, an imaginary segment lying at $+\infty$. Similarly, ${Q_{lA_l}}^b$ is set to $s_{-\infty}$, if Q_{lA_l} does not have a lower bounding segment. In an analogous manner, we define ${Q_{ri}}^a$ and ${Q_{ri}}^b$ to be the upper and lower bounding segment of the i-th quad Q_{ri} $(1 \leq i \leq A_r)$ of U_r (see Figure 11 for an example of such bounding segments). The active quads of U_l and U_r are available to us from the recursive computation, and are arranged in M_l and M_r, respectively, in the manner described earlier. Thus all endpoints p belonging to a particular quad Q_{li} of U_l will be in consecutive PEs. Each endpoint p of U_l will indicate its quad Q_{li} by specifying ${Q_{li}}^a$ and ${Q_{li}}^b$; similarly for every endpoint p' of U_r. Let us use p^Q to denote the current active quad that the point p lies in.

Now we want to determine the active quads of U. In the active-quad-wise ordering of the endpoints of E_{U_l} (E_{U_r})[21], let el_i (er_i) be the endpoint in the i-th $(1 \leq i \leq k)$ processor of M_l (M_r). Let $\mathcal{E}_{U_l}$ $(\mathcal{E}_{U_r})$ be the ordered set containing these endpoints according to their active-quad-wise ordering in M_l (M_r), i.e. $\mathcal{E}_{U_l} = \{el_1, \ldots, el_{k_{l1}}, el_{k_{l1}+1}, \ldots, el_{k_{l1}+k_{l2}}, \ldots, el_k\}$ $(\mathcal{E}_{U_r} = \{er_1, \ldots, er_{k_{r1}}, er_{k_{r1}+1}, \ldots, er_{k_{r1}+k_{r2}}, \ldots, er_k\})$. Let $\mathcal{H}_{U_l}$ $(\mathcal{H}_{U_r})$ be the ordered set consisting of all the active quads of U_l (U_r) i.e. $\mathcal{H}_{U_l} = \{Q_{l1}, Q_{l2}, \ldots, Q_{lA_l}\}$ and $\mathcal{H}_{U_r} = \{Q_{r1}, Q_{r2}, \ldots, Q_{rA_r}\}$. Consider the set $\mathcal{H} = \mathcal{H}_{U_l} \cup \mathcal{H}_{U_r}$. The bounding segments of the quads in the set $\mathcal{H}$ have a unique ordering defined on them; this ordering is given by the intersection of these bounding segments with the common boundary of the slabs U_l and U_r. Consider an active quad Q of U, and let Q^a and Q^b be its bounding segments. Obviously, Q^a will be an upper bounding segment for some active

[21] Recall that E_{U_l} (E_{U_r}) is the ordered set of endpoints that lie in the interior of the slab U_l (U_r). In this set, the endpoints are ordered according to their x-coordinates. We described the notion of active-quad-wise ordering on page 37.

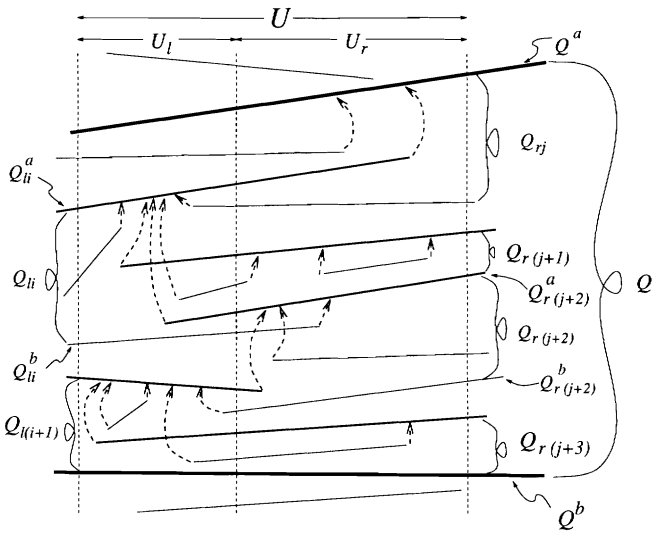


Figure 11: An active quad Q of U. Q_{li}, $Q_{l(i+1)}$ (Q_{rj}, $Q_{r(j+1)}$, $Q_{r(j+2)}$, $Q_{r(j+3)}$) are consecutive active quads of U_l (U_r). The dashed arrowed curves going out of each endpoint indicate the upper bounding segment of that points quad.

quad of U_l or U_r (possibly both). Similarly, Q^b will be a lower bounding segment for some active quad of U_l or U_r (note that the quad(s) in this case might be different from the quad(s) for Q^a). For example, in Figure 11, Q^a is an upper bounding segment for Q_{rj} and Q^b is a lower bounding segment for $Q_{l(i+1)}$ and for $Q_{r(j+3)}$. Hence, the bounding segments for all the active quads in U can be found from the bounding segments of the quads in the set $\mathcal{H}$.

We now want to arrange the endpoints in E_U in the desired active-quad-wise order. This can be achieved in the following way.

(a) First, we merge the ordered sets of endpoints, $\mathcal{E}_{U_l}$ and $\mathcal{E}_{U_r}$. This merging of endpoints el_i ($\in \mathcal{E}_{U_l}$) and er_i ($\in \mathcal{E}_{U_r}$) is done according to the ordering given by the upper bounding segments of the quads ${el_i}^Q \in \mathcal{H}_{U_l}$ and ${er_i}^Q \in \mathcal{H}_{U_r}$. In Figure 11, these upper bounding segments are indicated by the dashed arrowed curves from each endpoint. Let this merged (ordered) set of endpoints be $\mathcal{E}_U = \{e_1, e_2, \ldots, e_{2k+1}\}$ ($\mathcal{E}_U$ consists of the endpoints from $\mathcal{E}_{U_l}$, $\mathcal{E}_{U_r}$, and the endpoint on the common boundary of U_l and U_r)[22].

(b) For every e_i in PE P_i, either ${e_i}^Q \in \mathcal{H}_{U_l}$ or ${e_i}^Q \in \mathcal{H}_{U_r}$. Each P_i determines if the upper bounding segment of ${e_i}^Q$ spans the entire slab U. If so, then let us say that some field in P_i is set to 1, and if not, it is set to 0. Corresponding to every such spanning segment s of U, there will be a consecutive set of PEs with endpoints that have s as an upper bounding segment: all these PEs will be set to 1. Let D_1 and D_2 be two such successive sets of PEs. Now, we know that

[22] We would like to point out a small detail: Let p be the point on the boundary of U_l and U_r. If the segments p^a and p^b (the segments that lie immediately above and below p) span U, then the active quad of U that p lies in is the one defined by p^a and p^b. If p^a does not span U, then p's active quad is the same as the active quad that the endpoint(s) of p^a lies in.

the endpoints have already been sorted according to the upper bounding segments of their quads. Thus, all endpoints that lie in PEs between D_1 and D_2 (and which are hence set to 0) must lie in the same active quad of U. We can now update the information to indicate which active quad of U each e_i lies in: this can be done by an appropriate *selected broadcasting* operation, which will update, for every e_i, the upper and lower bounding segments of ${e_i}^Q$ to the new values. Obviously, all such active quads of U lie in sorted order on the mesh $M_l \cup M_r$. Hence, the determination of the active quads takes $O(\sqrt{k})$ time on $M_l \cup M_r$.

The Vertical Merge: Let Q' be a Q_l-quad (a similar argument will hold for a Q_r-quad.). If Q' is an active quad, then we already know its Voronoi diagram $Vor(S_{U_l} \cap Q')$, since it was computed recursively. If Q' is not active, then, since $S_{U_l} \cap Q'$ consists of just the two bounding segments ${Q'}^a$ and ${Q'}^b$, the Voronoi diagram of Q' can be computed trivially in constant time. There is an obvious upper bound on the total number of such empty quads that we can have in all such Q_l of U_l. Notice that an endpoint of at least one of ${Q'}^a$ and ${Q'}^b$ must lie in Q_r. Hence, the number of empty quads of U_l that we will need to consider can be at most $O(k)$, since that is the number of endpoints in U_r. Therefore, the computation of $Vor(S_{U_l} \cap Q')$ for all such empty Q' will take at most $O(k)$ time, sequentially. In the *vertical merge* step, we want to merge the Voronoi diagrams of all the Q_l-quads. The result of such a merge is called the Q_l-diagram, denoted by $Vor(S_{U_l} \cap Q_l)$. The analogous diagram in Q_r is called the Q_r-diagram, $Vor(S_{U_r} \cap Q_r)$. Since, as we just mentioned, the Voronoi diagram of each Q_l-quad Q' is readily available to us, this merge step involves just merging these $Vor(S_{U_l} \cap Q')$, which turns out to be fairly straightforward. This will become clear as we describe the computation of the Q_l-diagram on the mesh. The Q_r-diagram can be computed in a similar way. Keep in mind that we are talking about *one* active quad Q of U, and that such a computation needs to be performed for every active quad of U.

Let $\mathcal{L}_m$ represent the common boundary line between U_l and U_r, $\mathcal{L}_l$ the left boundary line of U_l, and $\mathcal{L}_r$ the right boundary line of U_r. We use $s(p_i)$ to represent the segment that p_i is an endpoint of. We will first give the description of the method to find the empty quads of Q_l and Q_r, since this information is not immediately available to us at the end of the previous step (the determination of the active quads of U). Consider all the Q_l-quads and the Q_r-quads. Clearly, there is a sorted order on the bounding segments of these quads according to the point at which they intersect $\mathcal{L}_m$. In this technique on the mesh, we will arrange these bounding segments in this sorted order on the mesh $M_l \cup M_r$. Suppose $s(p_i)$ and $s(p_j)$ are two successive spanning segments of Q_l. We want all points of Q_r (if any) that lie between these two segments to lie between the PEs that hold p_i and p_j. Similarly for Q_r. As we will see later, this arrangement will enable us to determine the empty Q_l-quads, and the empty Q_r-quads.

Recall that in the previous step the endpoints in $\mathcal{E}_{U_l}$ and $\mathcal{E}_{U_r}$ were merged according to the ordering given by the upper bounding segment of their quads. Thus the points in Q will still be grouped according to their old quads (from U_l or U_r). As an example, the points in quad Q of

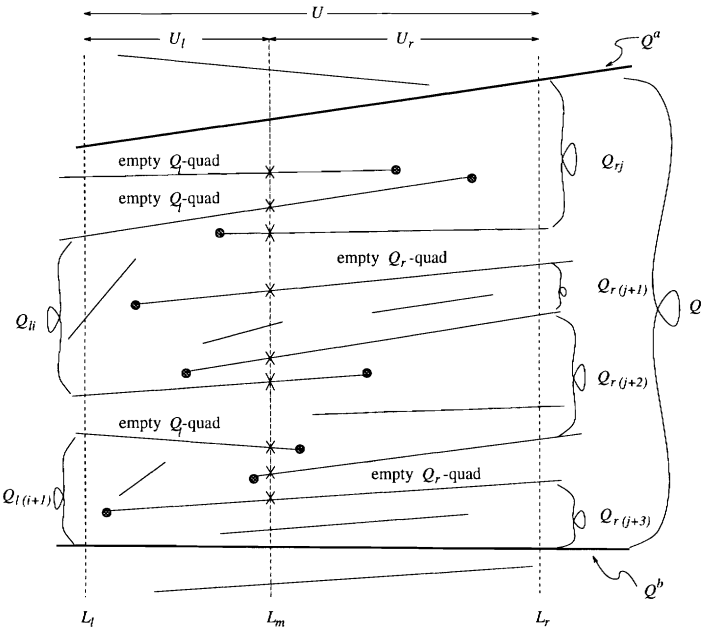


Figure 12: The Q_l-quads and the Q_r-quads of an active quad Q of U. We want the endpoints p, marked by the shaded circles, to be sorted according to the point at which $s(p)$ intersects $\mathcal{L}_m$ (this point is marked by a cross).

Figure 12 are arranged quad-wise in the order specified: points of Q_{rj}, Q_{li}, $Q_{r(j+1)}$, $Q_{r(j+2)}$, $Q_{l(i+1)}$ and finally $Q_{r(j+3)}$. But this ordering does not necessarily guarantee that the points are in the order that we want. To achieve this, we will need to a further sort of the points of Q. The gist of the method used to find this new order of the points of Q is this[23]: All points p of quad Q such that $s(p)$ spans either U_l or U_r are sorted according to the point at which $s(p)$ intersects $\mathcal{L}_m$. In Figure 12, such endpoints are marked by the small shaded circles. For those points p such that $s(p)$ does not span U_l or U_r (in Figure 12, these are the unmarked endpoints), we solve this problem by affiliating p with the upper bounding segment of its (recursively computed old) quad. This segment will certainly be a spanning segment of either U_l or U_r, depending on whether p lies in U_l or U_r, respectively. This can be given as follows. Let p_1 and p_2 be two points that belong to Q.

Case 1: $p_1 \in U_l$, $p_2 \in U_l$. The order between these two points was established during the recursive step: we maintain that order (i.e. the order given by the upper bounding segments of the active Q_l-quads that p_1 and p_2 belong to).

Case 2: $p_1 \in U_r$, $p_2 \in U_r$. Similar to above.

Case 3: $p_1 \in U_l$, $p_2 \in U_r$. Let Q_1 (Q_2) be the active Q_l-quad (Q_r-quad) that p_1 (p_2) lies in.

Case 3.1: $s(p_1)$ intersects $\mathcal{L}_r$. If p_2 lies above $s(p_1)$, then $p_2 < p_1$, else $p_1 < p_2$.

Case 3.2: $s(p_2)$ intersects $\mathcal{L}_l$. If p_1 lies above $s(p_2)$, then $p_1 < p_2$, else $p_2 < p_1$.

[23]Note that we could actually have done this sort during the previous merge step of determining the active quads. However, we choose to do it here since these details are not really necessary to determine the active quads of U.

Case 3.3: $s(p_1)$ does not intersect $\mathcal{L}_r$ and $s(p_2)$ does not intersect $\mathcal{L}_l$. In this case, we maintain the order as established by the upper bounding segment of each point's quad. In other words, if ${Q_1}^a$ intersects $\mathcal{L}_m$ at a higher point than or same point as ${Q_2}^a$, then $p_1 < p_2$. If not, $p_2 < p_1$.

Case 4: $p_1 \in U_r, p_2 \in U_l$. Similar to above.

$\mathcal{E}_U$ will now refer to the set of endpoints of U in this updated ordering. Now that we have the points of Q in the desired order, we will describe the method used to find the empty quads of Q_l. The idea is simple: If two consecutive spanning segments of U_l (U_r) in Q_l (Q_r) do not have any endpoints of Q_l (Q_r) between them, then they must define an empty Q_l-quad (Q_r-quad). On the mesh this works as follows. The endpoint p in each PE of $M_l \cup M_r$ determines if $s(p)$ spans U_l. If so, some field in P_i is set to 1, say, and to 0 otherwise. In the PEs that contain the endpoints of Q, consider two PEs P_j and P_k that are successive in the set of PEs set to 1. Let the endpoints in these be p_1 and p_2, respectively. Clearly, $s(p_1)$ and $s(p_2)$ are two consecutive spanning segments of U_l. If there are no endpoints of Q_l in the PEs between P_j and P_k, then $s(p_1)$ and $s(p_2)$ will define the upper and lower bounding segment of an empty quad. Let InUleft be a field in each PE P_i that is set to 1 if the endpoint in P_i belongs to U_l; otherwise, it is set to 0. The next step is to use, for each such P_j and P_k, the PEs $P_{j+1}, \ldots, P_k$ as one of the components for a segmented prefix scan. We perform the segmented scan operation on InUleft. If PE P_k has 0 as the result of the segmented prefix scan, then that means that there are no points from Q_l in the PEs between P_j and P_k. We have thus found an empty Q_l-quad. We compute the Voronoi diagram of this empty quad, which is nothing but $B(s(p_1) \cap Q_l,\ s(p_2) \cap Q_l)$ and this consists of at most 5 Voronoi edges. We store the diagram of this empty quad in PE P_k and mark the edges to indicate that they are bisectors of segments in U_l. In the next phase, we perform steps analogous to the above, and find the Voronoi diagram of the empty quads of Q_r.

Now that we have the Voronoi diagram of all the Q_l-quads and the Q_r-quads, we can compute the Q_l-diagram and the Q_r-diagram. This turns out to be very straightforward. We will describe how to construct the Q_l-diagram; the Q_r-diagram can be computed in a similar way. Let Q_1 and Q_2 be two adjacent quads in Q_l with Q_1 above Q_2. Also, let $Vor_1 = Vor(S_{U_l} \cap Q_1)$ and $Vor_2 = Vor(S_{U_l} \cap Q_2)$. We want to merge Vor_1 and Vor_2 in order to find the Voronoi diagram of $Q_1 \cup Q_2$. From a lemma by Yap [[57], Lemma 5] we know that the objects of Q_1 and the objects of Q_2 do not interact with each other. Thus the edges of Vor_1 (Vor_2) will not be modified in any way by the objects of Q_2 (Q_1). The only point to note is about the segment $s \in S_{U_l}$ that forms the boundary of Q_1 and Q_2: $Vor(s)$ now consists of the edges of $Vor(s)$ from Vor_1 *and* the edges of $Vor(s)$ from Vor_2.

Thus, none of the bisectors of Vor_1 and Vor_2 have to be modified in any way when we merge the two diagrams. The set of Voronoi edges in the merged diagram is the union of the sets of Voronoi edges of Vor_1 and Vor_2. In essence, we just have to "concatenate" the Voronoi diagrams

of the adjacent quads in the correct sorted order[15]. This fact makes the computation of the Q_l-diagram very straightforward. To merge the Voronoi diagrams of all the Q_l-quads Q', we just have to arrange the $Vor(S_{U_l} \cap Q')$ in the correct sorted order. The method that we just described to find the empty Q_l-quads ensures that all the Q_l-quads are in the correct sorted order on the mesh $M_l \cup M_r$. Hence, all their Voronoi diagrams are also in the correct sorted order. With a sort step we can ensure that within Q, we have all the Voronoi edges of the Q_l-diagram in consecutive PEs, followed by the Voronoi edges of the Q_r-diagram.

From the above, it can be seen that the vertical merge step takes $O(\sqrt{k})$ time on the k PEs of $M_l \cup M_r$.

The Horizontal Merge: This is the most important part of the merge process. In the *horizontal merge* step, we reach our final goal of constructing the Voronoi diagram $Vor(S_U \cap Q)$ for every active quad Q of U, and this is done by merging the Q_l and Q_r-diagrams. Recall that the merging of these two diagrams involves the construction of the *contour*, which is the locus of all points that are equidistant from the objects in $S_{U_l} \cap Q$ (call these the Q_l-objects) and the objects in $S_{U_r} \cap Q$ (Q_r-objects). As before, our discussion will be based on 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 [25, 57] and the parallel PRAM method of [15], we will manipulate objects known as *primitive regions*, to be defined shortly, for the construction of the contour. For the rest of this discussion, we will assume that the Q_l-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 $S_{U_l} \cap Q_l$, we add *spokes* [25] 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 13 for a Voronoi diagram augmented with spokes. In [15], the authors add some additional spokes. For all e that are point elements, we check if the horizontal leftward 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 rightward ray from e. If these leftward and rightward 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 13. 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) [15]. The piece of Voronoi edge that forms one of the boundary edges of each prim is called a *semi-edge* [15]. Notice that since $VorSet(S_{U_l})$ 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_l-diagram as Q_l-spokes, the prims of the Q_l-diagram as Q_l-prims, and the semi-edges of the Q_l-diagram as Q_l-semi-edges.

In the merge computation on the mesh so far, our technique has been to store a constant number

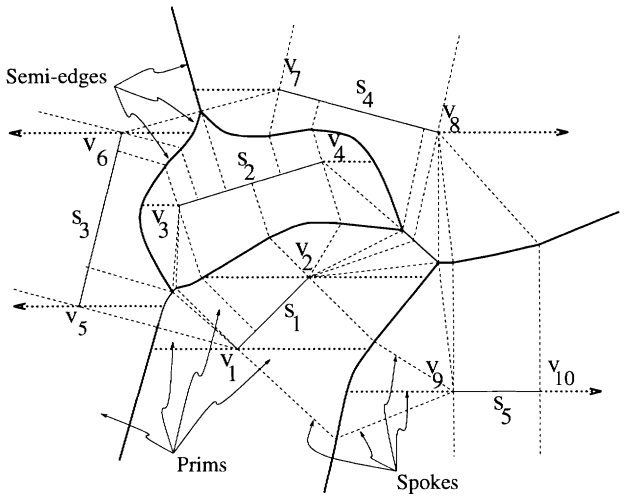


Figure 13: A Voronoi diagram augmented with spokes.

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 $Vor(e_1)$ and $Vor(e_2)$. 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 $Vor(e_1)$, where e_1 is either an endpoint or an open line segment corresponding to segment s_1 in $S_{U_l} \cap Q$, determines if it is intersected in the desired manner by the leftward and rightward 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_l-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 [15, 57] for the proofs.

Lemma 5.3 (Goodrich et al. [15]) *Let α and β be Q_l- and Q_r-prims, respectively. Let $s_\alpha \in S_{U_l}$ and $s_\beta \in S_{U_r}$ be such that $\alpha \subseteq Vor(s_\alpha)$ and $\beta \subseteq Vor(s_\beta)$. 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 5.4 (Goodrich et al. [15]) *The contour is monotone with respect to the y-axis.*

Lemma 5.5 (Goodrich et al. [15]) *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 [15].

The motivation behind the method to construct the contour (as developed in [15]) is as follows: Since the contour is the locus of points that are equidistant from the Q_l-objects and the Q_r-objects,

it will be made up of parts of bisectors of the form $B(e_l,\ e_r)$, where e_l is a Q_l-object and e_r is a Q_r-object. The construction of the contour involves finding all such pairs $(e_l,\ e_r)$. Obviously, every such Q_l-object e_l that contributes a bisector to the contour will have some of its Q_l-prims intersected by the contour. Thus the first step towards the construction of the contour is to find all the Q_l-prims that are intersected by the contour, and the order in which they are intersected. Similarly, we identify all the Q_r-prims that are intersected by the contour and put them in the right order. Once we have these two ordered sets of prims, we identify all $(e_l,\ e_r)$ pairs such that part of $B(e_l,\ e_r)$ is a piece of the contour.

Given below are the important details of 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 $M_l \cup M_r$, and within each such Q, we have the Q_l-diagram, followed by the Q_r-diagram.

(1) *Finding the intersected Q_l-prims in the correct order* The method described below can be applied in an obvious way to find the intersected Q_r-prims in the correct order.

(1.1) *Finding the Q_l-spokes that are intersected by the contour:* We wish to determine the Q_l-prims that are intersected by the contour. This is equivalent to identifying the Q_l-spokes that are intersected by the contour because, by Lemma 5.5, if the contour intersects a prim, it must intersect at least one of the spokes of that prim. For every Q_l-spoke l', one endpoint a of the spoke is adjacent on a Q_l-object e_l and one endpoint b is adjacent on a Q_l-semi-edge[24]. If b is closer to a Q_r-object than it is to e_l, then l' must be intersected by the contour. In order to determine if b is closer to a Q_r-object, we find the Voronoi region $Vor(e_r)$ of the Q_r-diagram that b lies in. If $d(b,\ e_r) > d(b,\ e_l)$, then the contour must intersect l'. Call a (b) the o-endpoint (s-endpoint) of l'.

We can find such a $Vor(e_r)$ for the s-endpoint of every Q_l-spoke by doing planar point location. From Lemma 5.1, we know that we can do this by running Algorithm MULTILOC with some slight modifications: The set S is the set of Q_r-semi-edges of all the active quads Q and the set P is the set of s-endpoints of the Q_l-spokes of all the active quads Q. Since $|S| + |P| = O(k)$, this step can be done on $M_l \cup M_r$ in $O(\sqrt{k})$ time. Once we find such a $Vor(e_r)$ for every Q_l-spoke, we can determine, in constant time, if it is intersected by the contour. Let us assume that all Q_l-spokes that are intersected by the contour are marked in an appropriate way.

(1.2) *Sorting the intersected Q_l-spokes:* In this step we sort the set of intersected Q_l-spokes according to the well-defined sorted order that is guaranteed by Lemma 5.4. If we assume that the contour is oriented from bottom to top, we know that the o-endpoint of every intersected Q_l-spoke must lie to the left of the contour and the s-endpoint of every

[24] what about unbounded spokes? we should be able to deal with it in a manner similar to that in [22].

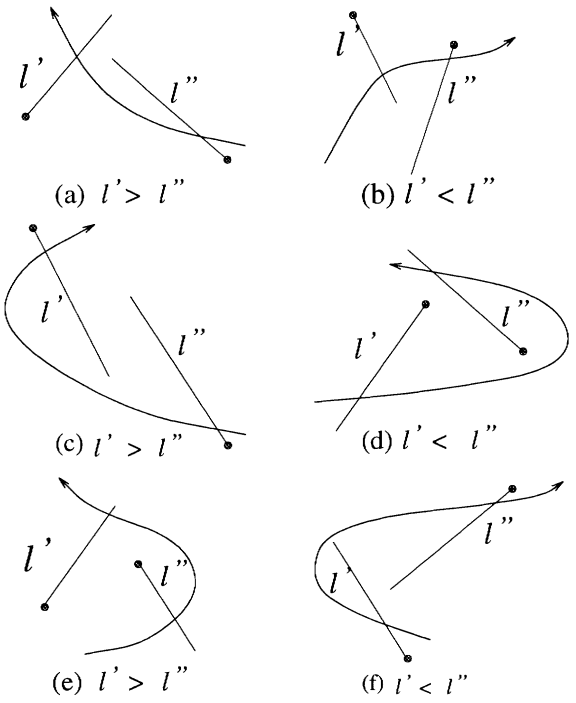


Figure 14: An illustration of Lemma 5.6 (Case 2). Each spoke's o-endpoint, which must lie to the left of the oriented contour, is indicated by the small shaded circle.

intersected Q_l-spoke must lie to the right of the contour. The spokes will be sorted from top to bottom. The following lemma gives us the method to find the ordering.

Lemma 5.6 *Let l' and l'' be two Q_l-spokes, and let y_1' (y_1'') be the y-coordinate of the o-endpoint of l' (l'') and y_2' (y_2'') be the y-coordinate of the s-endpoint of l' (l''). The order between l' and l'' is determined in the following way:*

Case 1 *One of l' or l'' lies entirely above the other i.e. $[y_1', y_2'] \cap [y_1'', y_2''] = \emptyset$.*

(a) $\min(y_1', y_2') > \max(y_1'', y_2'')$*: $l' > l''$.*

(b) $\min(y_1'', y_2'') > \max(y_1', y_2')$*: $l'' > l'$.*

Case 2 $[y_1', y_2'] \cap [y_1'', y_2''] \neq \emptyset$.

For the sake of brevity, we refer the reader to Figure 14 for an illustration of the different cases that are possible in this instance.

Proof: Case 1 is obviously true, by Lemma 5.4. The details of the proof for Case 2 are also fairly obvious: a figurative proof is offered in Figure 14. □

This step can be done in $O(\sqrt{k})$ time on $M_l \cup M_r$. Let this ordered set of intersected Q_l-spokes (Q_r-spokes) be called IS_l (IS_r). Note that the information about the ordered set of intersected Q_l-prims (Q_r-prims) is implicit in IS_l (IS_r): call this ordered set IP_l (IP_r). Within each quad Q in $M_l \cup M_r$, we have the the prims of IP_l in order, followed by the prims of IP_r.

(2) *Finding the contour between Q_l-diagram and the Q_r-diagram:* We use the two sorted sets IP_l and IP_r to construct the contour. The PRAM method of Goodrich *et al.* [15] does this in the following way: Let α be the median prim in the sorted set IP_l, and let s_α be its Q_l-object. For every prim β in IP_r (whose Q_r-object is s_β), we compute $b_{\alpha,\beta} = B(s_\alpha, s_\beta) \cap \alpha \cap \beta$. From Lemma 5.3, we know that if $b_{\alpha,\beta}$ is non-empty, then it is part of the contour. Also, all the β's such that $b_{\alpha,\beta}$ is non-empty form a continuous interval I_α of prims in IP_r [15]. Furthermore, all the prims of IP_l that lie above (below) α can interact only with the prims of IP_r that lie above (below) I_α. In[15], the authors recurse on the half above α and above I_α, and below α and below I_α, in parallel.

We will give a brief description of a non-recursive solution of this final step on the mesh. Consider a prim α from IP_l. We wish to find the interval I_α of prims from IP_r that interact with α. One way to do this is by identifying the topmost prim of the interval I_α (call this α^t) and the bottommost prim of the interval I_α (call this α^b). Sequentially, we can find α^t by doing a binary search in the following way (α^b can be found in a similar way): Let β be the median prim of IP_r. (a) *If $b_{\alpha,\beta}$ is non-empty*, then we know that α^t must lie in the top half of IP_r, so we recurse on that half to find α^t. (b) *If $b_{\alpha,\beta}$ is empty*, then we can determine which of α and β the contour intersects first. This can be done by comparing the order of α and β in a manner analogous to that given in Lemma 5.6. If $\alpha > \beta$ (i.e. the contour, oriented from bottom to top, intersects β before it intersects α), then the binary search for α^t has to recurse on the top half of IP_r. If $\alpha < \beta$, then it recurses on the bottom half of IP_r.

Now, in order to find α^t for every α in IP_l in parallel, we have to do a simultaneous search. The mesh implementation of such a step can be done by an application of Algorithm SIMULTSRCH, by using prims from IP_l as the equivalent of the point set P, and the prims from IP_r as the equivalent of the segment set S in Algorithm SIMULTSRCH. Similarly, α^b can be found by a similar application of Algorithm SIMULTSRCH. Let us assume that every α also knows the PE ids P_t and P_b of α^t and α^b, respectively. From this information, α can find the length $|I_\alpha|$ of the interval I_α. If we now make $|I_\alpha|$ copies of α, each of those copies can read the prim β from one of the PEs from P_t to P_b. We can thus determine the piece of the contour $b_{\alpha,\beta}$. Making $|I_\alpha|$ copies of every α in IP_l can be done by a prefix scan on $|I_\alpha|$, followed by a one-to-one routing, and finally by a selected broadcasting step. To determine each $b_{\alpha,\beta}$ that is part of the final contour, each of the copies of α reads the β from one of the PEs from P_t to P_b. This can be done with one RAR step.

Since the lengths of the lists IP_l 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 $M_l \cup M_r$.

5.3 Motion Planning Using Voronoi Diagrams

We summarized, in Section 1, the main ideas behind the method by Ó'Dúnlaing and Yap for planning the motion of an object (a disc) with two *dofs*, moving amongst obstacles [34]. 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 $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh, as we have just shown. Recall that 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 prespecified 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 5.1, 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 [4], Atallah and Hambrusch show that in a graph with edge set E, we can solve this problem in $\sqrt{|E|}$ on a mesh with $|E|$ PEs. In the graph defined by the Voronoi diagram, $|E|$ is $O(n)$. It follows, therefore, that we can implement the motion planning technique of [34] in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh.

6 Proposed Research

In recent years, there has been steady progress in the development of exact algorithms for various special cases of motion planning. These algorithms provide us with polynomial-time solutions that are often worst-case optimal. However, in many cases, the sequential complexity is high enough to merit investigation into possible speed-ups offered by parallelization. The goal of our research is two-fold. On the one hand, we are interested in the parallelization of existing sequential algorithms for various special cases of motion planning. On the other hand, there is a wide array of useful motion planning problems that have important applications, but whose solutions are computationally intractable in practice. The intractability of such problems is related to the number of degrees of freedom of the moving object. In such cases, it may be feasible to apply approximation techniques as well as randomization, which provide faster run-times. Randomization has been shown to be a particularly suitable strategy for "breaking" the inherently sequential nature of many problems. In what follows, we state problems that we propose to investigate for solutions on fixed-connection networks and/or on PRAMs.

In our investigation so far, we have derived efficient mesh algorithms for certain geometric techniques that are used in special instances of motion planning viz. planning the motion of an object in two dimensions with 2 *dofs* by using visibility graphs or Voronoi diagrams. These techniques are specialized applications of general motion planning strategies. We would like to look into the development of efficient parallel algorithms for other special cases of motion planning for which polynomial sequential algorithms exist. Efficient parallel algorithms for such cases, whether on the PRAM or on fixed-connection networks, offer the possibility of significant speed-ups, which will prove to be very useful. For example, consider an object with 3 *dofs* moving in 2-dimensional space among polygonal obstacles. Lozano-Pérez and Wesley provide an approximate technique (assuming the moving object is convex) for this case in [30]. Although this technique can be incorporated into our mesh algorithm, we are interested in parallel algorithms for the more exact methods of solving this problem.

(a) As we mentioned in Section 1, Sharir *et al.* [42, 45, 50] have developed polynomial-time sequential algorithms for a number of special cases by applying the projection method. By using discrete combinatorial representations of the free configuration space, they come up with algorithms for problems such as a rod (a line segment, sometimes also called a ladder) moving among convex obstacles, a rigid polygonal object (with 3 *dofs*) moving among convex obstacles etc. Even though these algorithms are polynomial-time algorithms, they are not very efficient. Refinements for some of these algorithms have been proposed. In particular, the exact algorithm for a rod moving among polygonal obstacles has been improved substantially to run in time $O(n^2 \log n)$ [28, 52]. We would like to look into how special applications of the projection method can benefit from parallelism. In particular, we are interested in designing

fast parallel algorithms for planning the motion of a rod or a polygonal object with 3 *dofs* moving among polygonal obstacles.

(b) The retraction method was mentioned in Section 1, and we noted there that this approach has the advantage of providing "maximum clearance" from the obstacles for the moving object. In Section 5, we developed efficient mesh algorithms for one such application of the retraction approach of [34], namely using Voronoi diagrams to plan the motion of an object with 2 *dofs* (either a disc or a convex object) moving among polygonal obstacles. The principle of retraction has led to efficient algorithms for planning the motion of a rod (3 *dofs*) moving among polygonal barriers [33] and of two independent discs in the plane [56].

For the former problem, Ó'Dúnlaing, Sharir and Yap [33] generalize the retraction approach by defining a a variant of the Voronoi diagram in the 3 dimensional configuration space of the rod [41]. They construct a one-dimensional diagram by performing two retractions (as opposed to just one in the case of [34]), followed by a graph search on this diagram in order to find a motion path. Via a complicated geometric analysis, they show that their algorithm runs in time $O(n^2 \log n \log^* n)$. We observed in Section 5 that even for the simpler case of an object with 2 *dofs*, the parallel construction of the Voronoi diagram is fairly involved. The retraction method for the rod promises to be even more involved! However, given that this retraction method is an exact technique that has the appealing "maximum clearance" property, we would like to investigate the adaptation of this strategy to the parallel environment.

(c) We have considered various geometric algorithms for the motion planning problem in 2 dimensions. As previously noted, the problem rapidly becomes harder in 3 dimensions. In general, the geometric characterizations of the various special cases of motion planning in 3 dimensions, even for small degrees of freedom, are much less understood than the corresponding problems in 2 dimensions. Consider the problem of a purely translational object moving in 3 dimensions among polyhedral obstacles. For the 2 dimensional counterpart of this problem, the method by Lozano-Pérez *et al.* [30] provides us with an exact algorithm that plans the shortest path for the object. We developed a mesh-optimal algorithm for this case in Section 4.

In 2 dimensions, we have the useful property that the shortest path from one point to another can be found by performing a shortest-path graph search on the visibility graph. Unfortunately, this useful relation between visibility graphs and shortest paths does not hold in 3 dimensions. The shortest paths between two points while avoiding polyhedral obstacles is still known to be piecewise linear. However, the vertices of the path can now lie on the edges of the polyhedra, and need not always coincide with the vertices of the polyhedra [41]. For general polyhedra in 3 dimensions, algorithms for finding such shortest paths have run-times that are no better than doubly-exponential [51]. An approximating pseudopolynomial technique for finding shortest paths among general polyhedral objects is given in [35]. Polynomial-time

algorithms are also offered for certain special cases; for example, when the obstacles are a small number of convex polyhedra [51]. We would like to study these problems in the parallel setting.

(d) As we have seen, the efficiency of solving various subroutines like the Voronoi diagram construction for line segments, multipoint location etc., directly affects the efficiency of many general motion planning strategies. There are several interesting open problems in this regard. For example, the best known parallel algorithm for the Voronoi diagram of a set of n line segments uses n processors and runs in $O(\log^2 n)$ time. This is not optimal in PT bounds, since the best known sequential algorithms for this problem are $O(n \log n)$. A similar result holds for the Voronoi diagram of a set of points in the plane, i.e. there are sequential algorithms which run in optimal $O(n \log n)$ time, but the best known deterministic parallel algorithm for this problem uses n processors and runs in $O(\log^2 n)$ time. There are a number of important geometric problems for which sub-optimal parallel algorithms were given in [[2], [10]]. For many of these problems, optimal parallel algorithms have since been derived, largely due to the work of Atallah, Cole and Goodrich [6], using the versatile technique of cascading divide-and-conquer to construct the algorithms. However, the important problems of the Voronoi diagram of a set of points and of a set of line segments have eluded optimal deterministic parallel solutions e.g. solutions that use n processors and run in $O(\log n)$ time. As observed in [40], it appears that very different techniques would have to be used in order to eliminate the extra $\log n$ factor in the PT-product.

One such technique could be *randomization.* A *randomized algorithm* makes some of its decisions on the basis of the outcomes of coin-flips, and we must be able to show that such a strategy will result in a fast run-time on *any input* with *high probability.* Theoretical development in the area of randomized algorithms has shown that randomization can be a very useful strategy for designing parallel algorithms for problems that seem to be inherently sequential. In addition, randomization has often resulted in parallel algorithms with better PT products than the best known deterministic parallel algorithms for numerous problems. For instance, Reif and Sen [40] show that the Voronoi diagram of a set of points in the plane can be constructed optimally on a CRCW PRAM with n processors in $O(\log n)$ time with high probability. Additional applications of randomization to problems in computational geometry can be found in [39].

We are interested in applications of randomization for different motion planning problems. Randomization might be a particularly well-suited strategy for running motion planning algorithms in simulated environments, as in computer graphics. In particular, we would like to investigate the possibility of developing an optimal randomized PRAM algorithm for the construction of the Voronoi diagram of a set of line segments in the plane.

(e) A very important class of problems in motion planning that we have not yet mentioned is

the algorithmic planning of movement for a *chain of links*, also known as a robot arm or a multilink. The problem of planning the motion of a chain of links is one that has very useful applications, especially in the arena of human figure animation in computer graphics. If the number of links, k, in the chain is our input size, then from the results stated in Section 1, the best algorithm for planning the motion of a multilink moving among obstacles takes time exponential in k. In fact, general motion planning for chains of links *even in the absence of obstacles* is intractable.

Consider a chain of links hinged together consecutively such that each link is allowed to rotate freely about its joint (the links may cross over one another). This is also known as the *carpenter's ruler*. In [20], Hopcroft *et al.* show that the problem of deciding whether such a ruler can be folded to within a specified length (thus every link makes an angle of 0 or π with the previous link) is NP-complete. Clearly then, if the chain consists of links that are only allowed to make an angle of 0 or π with the previous link, the reachability decision problem (i.e. does the free endpoint of the end effector reach a particular point or not) for this chain is NP-complete. Thus general motion planning for a chain of links appears to be intractable, even in the abscence of obstacles.

In addition, Hopcroft *et al.* [20] show that the reachability decision problem for a chain of k links moving among certain rectilinear barriers is NP-hard. The goal then is to look for classes of problems for which efficient algorithms (i.e. polynomial in k) can be found. For example, there are polynomial time algorithms for planning the motion of a carpenter's ruler in which every joint of the ruler is confined to move within a circle [20, 23]. When the motion is restricted by corners, the problem becomes more difficult and in fact, some lower bound results have been established. For example, Ke and O'Rourke [24] have established a lower bound of $\Omega(n^2)$ elementary submotions for moving a line segment (a single link) in a 2-dimensional polygonal space with n corners.

A link such that one of its endpoints is free is known as the *end effector*. For human figure animation in computer graphics, an important problem is to move a chain of links (such as an arm) so that the end effector (the hand, say) reaches a specified point, subject to additional constraints such as the angle limits for every joint. Approximate techniques have been developed for this problem. For instance, if we are given the position of the end effector, we can use inverse kinematics to find joint angles for the other links. This problem can be formulated as a non-linear programming problem. In the Computer Graphics Lab. at Penn, the approach to this problem has been to use approximate numerical techniques for non-linear programming with linear constraints [58]. For some simple cases of non-linear programming, closed form solutions are possible [36], but for more general problems only approximate numerical techniques[25] are possible.

[25] As opposed to closed form solutions, numerical techniques find only one solution to the non-linear programming

We are interested in looking into the usefulness of parallelism for specific cases of this important class of motion planning problems. The more exact polynomial-time techniques for simpler cases could benefit from valuable speed-ups offered by parallel algorithms. In addition, we would like to explore the possibility of applying other approximate techniques as well as randomization to cases that are more computationally expensive.

problem.

References

[1] P. K. Agarwal, A. Aggarwal, B. Aronov, S. R. Kosaraju, B. Scheiber, and S. Suri. Computing External-Farthest Neighbors in a Simple Polygon. *Discrete Applied Mathematics*, 31(2):97–111, 1991.

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

[3] T. Asano, T. Asano, L. Guibas, J. Hershberger, and H. Imai. Visibility of Disjoint Polygons. *Algorithmica*, 1986.

[4] M. Atallah and S. Hambrusch. Solving Tree Problems on a Mesh-connected Processor Array. *Information and Computation*, 69(1-3):168–187, 1986.

[5] M. J. Atallah and D. Z. Chen. Optimal Parallel Algorithm for Visibility of a Simple Polygon From a Point. In *Proc. 5th ACM Symp. on Comp. Geo.*, pages 114–123, 1989.

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

[7] R. A. Brooks. Solving the Find-Path Problem by Good Representation of Free Space. *IEEE Trans. on Systems, Man, and Cybernetics*, 13(3):190–197, March/April 1983.

[8] J. Canny. *Complexity of Robot Motion Planning.* PhD thesis, MIT, 1987.

[9] B. Chazelle, L. J. Guibas, and D. T. Lee. The Power of Geometric Duality. In *Proc. 24th IEEE Symp. on Foundations of Computer Science*, pages 217–225, 1983.

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

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

[12] H. Edelsbrunner, J. O'Rourke, and R. Seidel. Constructing Arrangements of Lines and Hyperplanes with Applications. In *Proc. 24th IEEE Symp. on Foundations of Computer Science*, pages 83–91, 1983.

[13] H. ElGindy and D. Avis. A Linear Algorithm for Computing the Visibility Polygon From a Point. *J. of Algorithms*, 2:186–197, 1981.

[14] H. ElGindy and M. T. Goodrich. Parallel Algorithms for Shortest Path Problems in Polygons. *The Visual Computer*, 3(6):371–378, 1988.

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

[16] M.T. Goodrich. Constructing Arrangements Optimally in Parallel. Technical report, Johns Hopkins University, 1991.

[17] L. J. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan. Linear Time Algorithms for Visibility and Shortest Path Problems Inside Simple Polygons. In *Proc. 2nd ACM Symp. on Comp. Geo.*, pages 1–13, 1986.

[18] J. Hershberger. Finding the Visibility Graph of a Simple Polygon in Time Proportional to its Size. *Algorithmica*, 4:141–155, 1989.

[19] J. E. Hopcroft, D. A. Joseph, and S. H. Whitesides. Movement Problems for Two-Dimensional Linkages. *SIAM J. Comput.*, 13:610–629, 1984.

[20] J. E. Hopcroft, D. A. Joseph, and S. H. Whitesides. On the Movement of Robot Arms in Two-Dimensional Bounded Regions. *SIAM J. Comput.*, 14(2):315–333, 1985.

[21] J. E. Hopcroft, J. T. Schwartz, and M. Sharir. On the Complexity of Motion Planning for Multiple Independent Objects; *PSPACE*-Hardness of the 'Warehouseman's Problem'. *The Int'l Journal of Robotics Research*, 3(4):76–88, Winter 1984.

[22] C. S. Jeong and D. T. Lee. Parallel Geometric Algorithms on a Mesh-Connected Computer. *Algorithmica*, 5(2):155–177, 1990.

[23] V. Kantabutra and S. R. Kosaraju. New Algorithms for Multilink Robot Arms. *Journal of Computer and System Sciences*, 32:136–153, 1986.

[24] Y. Ke and J. O'Rourke. Moving a Ladder in Three Dimensions: Upper and Lower Bounds. In *Proc. Third ACM Symp. on Computational Geometry*, pages 136–146, 1987.

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

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

[27] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, San Mateo, California, 1992.

[28] D. Leven and M. Sharir. An Efficient and Simple Motion Planning Algorithm for a Ladder Amidst Polygonal Barriers. *J. Algorithms*, 8:192–215, 1987.

[29] T. Lozano-Pérez. A Simple Motion-Planning Algorithm for General Robot Manipulators. *IEEE Journal of Robotics and Automation*, RA-3(3):224–238, 1987.

[30] T. Lozano-Pérez and M. A. Wesley. An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles. *Comm. ACM*, 22(10):560–570, 1979.

[31] R. Miller and Q. F. Stout. Mesh Computer Algorithms for Computational Geometry. *IEEE Transactions on Computers*, 38(3):321–340, March 1989.

[32] D. Nassimi and S. Sahni. Data Broadcasting in SIMD Computers. *IEEE Transactions on Computers*, 30:101–107, 1981.

[33] C. Ó'Dúnlaing, M. Sharir, and C. K. Yap. Retraction: A New Approach to Motion-Planning. In J. E. Hopcroft, J. T. Schwartz, and M. Sharir, editors, *Planning, Geometry and Complexity of Robot Motion*, chapter 7, pages 193–213. Ablex Pub. Co., Norwood, N.J., 1987.

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

[35] C. Papadimitriou. An Algorithm for Shortest Path Motion in Three Dimensions. *Info. Proc. Letters*, 20:259–263, 1985.

[36] R. P. Paul. *Robot Manipulators.* MIT Press, Cambridge, Mass., 1981.

[37] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag New York Inc., 1985.

[38] J. Reif. Complexity of the Movers' Problem and Generalizations. In *Proc. 20th IEEE Symp. on Foundations of Computer Science*, pages 421–427, 1979.

[39] J. H. Reif and S. Sen. Optimal Randomized Parallel Algorithms for Computational Geometry. In *Proc. 1987 IEEE International Conf. on Parallel Processing*, pages 270–277, 1987.

[40] J. H. Reif and S. Sen. Polling: A New Randomized Sampling Technique for Computational Geometry. In *Proc. 21st ACM Symp. on Theory of Computing*, pages 394–404, 1989.

[41] J. F. Schwartz and M. Sharir. Motion Planning and Related Geometric Algorithms in Robotics. *Proc. Int'l Congress of Mathematicians*, 2:1594–1611, August 1986.

[42] J. T. Schwartz and M. Sharir. On the Piano Movers' Problem: I. The Case of a Two-Dimensional Rigid Polygonal Body Moving Amidst Polygonal Barriers. *Comm. Pure and Applied Math.*, 36:345–398, 1983.

[43] J. T. Schwartz and M. Sharir. On the Piano Movers' Problem: II. General Techniques for Computing Topological Properties of Real Algebraic Manifolds. *Adv. in Appl. Math*, 4:298–351, 1983.

[44] J. T. Schwartz and M. Sharir. On the Piano Movers' Problem: III. Coordinating the Motion of Several Independent Bodies: The Special Case of Circular Bodies Moving Amidst Polygonal Barriers. *Robotics Res.*, 2:46–75, 1983.

[45] J. T. Schwartz and M. Sharir. On the Piano Movers' Problem: V. The Case of a Rod Moving in Three-Dimensional Space Amidst Polyhedral Obstacles. *Comm. Pure and Appl. Math*, 37:815–848, 1984.

[46] J. T. Schwartz and M. Sharir. A Survey of Motion Planning and Related Geometric Algorithms. *Artificial Intelligence*, 1988.

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

[48] M. Sharir. Efficient Algorithms for Planning Purely Translational Collision-Free Motion in Two and Three Dimensions. In *Proc. IEEE Symp. on Robotics and Automation*, pages 1326–1331, Los Alamitos, Calif., 1987. CS Press.

[49] M. Sharir. Algorithmic Motion Planning in Robotics. *IEEE Computer*, March 1989.

[50] M. Sharir and E. Ariel-Sheffi. On the Piano Movers' Problem: IV. Various Decomposable Two-Dimensional Motion-Planning Problems. *Comm. Pure and Appl. Math*, 37:479–493, 1984.

[51] M. Sharir and A. Schorr. On Shortest Paths in Polyhedral Spaces. *SIAM J. Comput.*, 15(1):193–215, 1986.

[52] S. Sifrony and M. Sharir. An Efficient Motion-Planning Algorithm for a Rod Moving in Two-Dimensional Polygonal Space. *Algorithmica*, 2:367–402, 1987.

[53] S. Suri. Computing Geodesic Furthest Neighbors in Simple Polygons. *Journal of Computer and System Sciences*, pages 220–235, 1989.

[54] C. D. Thompson and H. T. Kung. Sorting on a Mesh-Connected Parallel Computer. *Comm. ACM*, 20(4):263–271, April 1977.

[55] E. Welzl. Constructing the Visibility Graph for n Line Segments in $O(n^2)$ Time. *Info. Proc. Letters*, 20:167–171, 1985.

[56] C. K. Yap. Coordinating the Motion of Several Discs. Technical report, Courant Institute of Mathematical Sciences, 1983.

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

[58] J. Zhao. Real-time Inverse Kinematics with Joint Limits and Spatial Constraints. Technical Report MS-CIS-89-89, Univ. of Penn., 1989.