1 PARALLEL RANDOMIZED TECHNIQUES FOR SOME FUNDAMENTAL GEOMETRIC PROBLEMS

Suneeta Ramaswami

Department of Computer Science
322, Business and Science Building
Rutgers University
Camden, NJ 08102

rsuneeta@crab.rutgers.edu

1.1 INTRODUCTION

Computational Geometry is the field of computer science that is concerned with algorithmic techniques for solving geometric problems. Geometric problems arise in innumerable applications, particularly in the fields of Computer Graphics, Computer-Aided Design and Manufacturing (CAD/CAM), Robotics and Geographic Information Systems (GIS). A typical example of a fundamental problem in computational geometry is the computation of the *convex hull* of a set of points in d-dimensional space. The convex hull of a set of points is the smallest convex set containing those points. (Informally stated, a *convex set* is such that for any two points in the set, the line connecting those two points is also contained in the set.)

Recent years have seen rapid advances in parallel algorithm design for problems in computational geometry. Some of the earliest work in this area was done by Chow [12] and Aggarwal *et. al.* [1]. In these papers, the authors gave parallel algorithms for various fundamental problems such as two-dimensional convex hulls, planar-point location, trapezoidal decomposition, Voronoi diagram of points, triangulation *etc.*, which are known to have sequential run-times of $O(n \log n)$. Most of their algorithms, though in NC, were not optimal in $P.T$

bounds and a number of them have since been improved.[1] Atallah, Cole and Goodrich [5] demonstrated optimal deterministic algorithms ($O(n)$ processors and $O(\log n)$ run-time) for many of these problems by applying the versatile technique of cascading divide-and-conquer and building on data structures developed in [1]. Cole and Goodrich give further applications of this technique in [15]. See [6] for a comprehensive survey on deterministic parallel algorithms for computational geometry. Reif and Sen [38] also obtained optimal randomized parallel algorithms for a number of these problems; these algorithms use n processors and run in $O(\log n)$ time with high probability. See [35] for a survey on the use of randomization in parallel algorithm design.

The important problems of constructing *the convex hull of points in three dimensions* and *the Voronoi diagram of points in two dimensions* , however, eluded optimal parallel solutions for a long time. Both these problems have sequential run-times of $O(n \log n)$ [17, 19, 24, 32, 43]. Aggarwal *et. al.* [1] gave $O(\log^2 n)$ and $O(\log^3 n)$ time algorithms (using $O(n)$ processors) for the Voronoi diagram and convex hull problems, respectively, and the technique of cascaded-merging could not be extended to these problems to improve their run-times [15]. Subsequently, Goodrich [20] has given an algorithm for 3-dimensional convex hulls that does optimal *work*, but has $O(\log^2 n)$ run-time, and Amato and Preparata [4] have described an algorithm that runs in $O(\log n)$ time but uses $n^{1+\epsilon}$ processors.

Randomization, however, proves to be very useful in obtaining optimal run-time as well as optimal $P.T$ bounds. (Sorting can be reduced to these problems, and hence the best possible run-time will be $\Theta(\log n)$ on n EREW and CREW PRAMs [16].[2]) Note that the lower bound of $\Omega(n \log n)$ for these problems also applies to randomized algorithms. In [37], Reif and Sen gave an optimal parallel randomized algorithm on the CREW PRAM for the construction of the convex hull of points in three dimensions. Since the problem of finding the Voronoi diagram of points in two dimensions can be reduced to the three-dimensional convex hull problem, they also obtained an optimal parallel method for the former. Their algorithm runs in $O(\log n)$ time using $O(n)$ processors, with high probability, and there are no known deterministic algorithms that match these bounds. We would like to point out that Levcopoulos, Katajainen and Lingas [26] gave an optimal expected time parallel algorithm for the Voronoi diagram of a randomly chosen set of points. However, the randomized algorithms surveyed here make no assumption about the distribution of the input set.

---

[1] NC is the parallel complexity class of problems that can be solved in poly-logarithmic time using a polynomial number of processors. The $P.T$ bound of a parallel algorithm is simply the product of the run-time with the number of processors used; this is also referred to as the *work* performed by the algorithm. This would be the run-time of a sequential algorithm that simulates a given parallel algorithm.

[2] The *P*arallel *R*andom *A*ccess *M*achine (PRAM) is the synchronous shared memory model of parallel computation in which all processors have access to a common memory. *C*oncurrent *R*ead *E*xclusive *W*rite (CREW) PRAMs allow two or more processors to read a memory location simultaneously, but do not allow simultaneous writes. EREW PRAMs allow no concurrent accesses.

Similarly, an optimal parallel solution to construct the *Voronoi diagram of line segments in the plane* also poses difficulties. Furthermore, the randomized technique presented by Reif and Sen [37] cannot be extended in a straightforward way to this problem. By designing a new randomized sampling technique to overcome some of the obstacles presented by the method in [37], Rajasekaran and Ramaswami [34] give an optimal $O(\log n)$-time randomized parallel algorithm for this problem using $O(n)$ processors. This algorithm is optimal in $P.T$ bounds and an $O(\log n)$ factor improvement in run-time over the previously best-known deterministic algorithm by Goodrich, Ó'Dúnlaing and Yap [21], providing further evidence of the usefulness of randomization in obtaining efficient algorithms. As in the previous case, there are no known deterministic algorithms that match these bounds.

Parallel randomized techniques have also led to efficient algorithms for *higher-dimensional convex hulls*, as shown by Amato, Goodrich and Ramos [3]. In particular, they demonstrate $O(\log n)$-time randomized parallel algorithms that perform optimal $O(n \log n + n^{\lfloor d/2 \rfloor})$ work in order to compute the intersection of a set of n half-planes in d dimensions (the dual of the problem of computing the convex hull of n points in d dimensions). Ramos [36] has also shown that randomized methods can be used to give near-optimal parallel algorithms for some one-dimensional lower envelope problems. The remainder of this chapter provides a brief survey of the randomized techniques used to design efficient parallel algorithms for some fundamental geometric problems. Section 1.2 gives all relevant definitions and establishes notation to be used in this chapter. Section 1.3 discusses the general randomization techniques that have yielded optimal parallel algorithms for the fundamental geometric problems of three-dimensional convex hulls, Voronoi diagrams of sets of line segments, and for higher-dimensional convex hulls. Finally, Section 1.4 describes the optimal parallel algorithms specifically for each of these three problems.

1.2 PRELIMINARIES AND DEFINITIONS

The use of *average-case* analysis of algorithms has been in existence for a long time, where the performance of the algorithm is measured by assuming a certain distribution on the input. In other words, resource bounds (such as run-time, space usage, *etc.*) are given for typical instances of the problem. For example, it can be shown that Hoare's quick-sort algorithm performs well on average. The difficulty of this approach is that it is not always possible to determine an accurate distribution for the input, and hence to carry out the average-case analysis. Making an assumption about the distribution of the input may not always be reasonable.

An alternative method is to introduce randomness into the algorithm itself. The idea of *randomizing* an algorithm by using a random number generator was first put forth independently by Rabin [33] and Solovay and Strassen [42]. It has proved to be an extremely powerful tool in the design of efficient algorithms for a wide variety of problems. A *randomized algorithm* is one in which some of the decisions depend on the result of coin flips. The objective of a

randomized algorithm is to ensure that *on any input*, the correct output will be produced with *high probability* within the stated time bounds. Randomized algorithms therefore do not make any assumptions about the distribution of the input. There has been significant interest in such algorithms because for many applications, randomized algorithms are considerably simpler and more efficient than their deterministic counterparts, in the sequential as well as parallel setting. These algorithms are also more implementable because of their simplicity. It can be shown that the high probability bounds hold even when a pseudo-random number generator is used, as is the case on real computers. Hence randomized algorithms fare well even in practice and for a number of applications, implementation results have borne out this fact.

A randomized algorithm is one which bases some of its decisions on the outcomes of coin-flips. The objective of a randomized algorithm is to ensure that *on any input*, the correct output will be produced with high probability within the stated time bounds. Hence there is a very small non-zero probability that the algorithm may fail (either to produce the correct answer or to stop within the specified run-time), and this is called the *error probability*. Let ϵ (ϵ being very close to 0) be the error probability of a randomized algorithm $\mathcal{A}$. Then for *any* input, $\mathcal{A}$ will succeed i.e. will run within the stated time bounds and produce the correct output with probability $\geq \ (1 - \epsilon)$. Notice that this probability holds for *any* input, and the success or failure of the algorithm does not depend on the input itself. In other words, the analysis of the run-times or space bounds of such algorithms does not make any assumptions about the distribution of the input.

There are two types of randomized algorithms, as given below:

1. In the first type of randomized algorithms, the output of the algorithm is always guaranteed to be correct, but the run-time of the algorithm might vary. In other words, the run-time of the algorithm is a random variable. These algorithms are known as **Las Vegas** algorithms.

2. In the second type of randomized algorithms, the run-time of the algorithm does not vary (i.e. is dependent just upon the input and not on the coin-flips), but the correctness of the output will hold with a certain high probability. In other words, the correctness of the output is a random variable. These algorithms are known as **Monte Carlo** algorithms.

The algorithms presented in this chapter are Las Vegas algorithms. Given below is a formal definition of resource bounds for a randomized parallel algorithm.

Definition 1 *A randomized parallel algorithm $\mathcal{A}$ is said to require resource bound $f(n)$ with* high probability *if, for any input of size n, the amount of resource used by $\mathcal{A}$ is at most $\bar{c}.\alpha.f(n)$ with probability $\geq \ 1 - 1/n^{\alpha}$ for positive constants $\bar{c}, \alpha$ ($\alpha > 1$). $\tilde{O}$ is used to represent the complexity bounds of randomized algorithms i.e. $\mathcal{A}$ is said to have resource bound $\tilde{O}(f(n))$.*

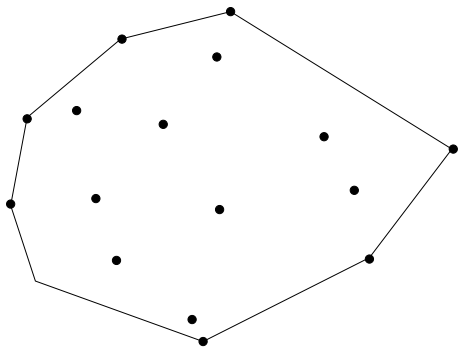

Figure 1.1 The convex hull of a set of points in two dimensions

The parallel complexity class RNC is defined in a manner similar to the class NC. It is the class of decision problems that can be solved by a *randomized* PRAM algorithm in polylogarithmic time using a polynomial number of processors.

1.2.1 Geometric Definitions and Notation

In this section, we give definitions and notation for the geometric problems addressed in this chapter. Since most of these problems are in two or three dimensions, we will restrict ourselves in this section to definitions of relevant geometric structures in two and three dimensions. This also allows us to introduce the material in a manner that is easier to follow. The reader's greater familiarity with the concepts in two and three dimensions will be useful when he or she reaches Section 1.4.3, which discusses the problem of higher-dimensional convex hulls. In order to keep the initial presentation simpler, the definitions relevant to that section will be given there. (All definitions and notation used here are standard; see e.g. [17, 21, 25, 32, 43]). We will also assume that the input set of objects is in general position, i.e., that no three are colinear and no four are cocircular.

Convex Hulls. A region R is said to be *convex* if for any two points p and q in R, the line segment from p to q is also contained entirely in R. The *convex hull*, $CH(S)$, of a set S of points in two or three dimensions is simply the *smallest* convex region that contains the points. Each point of S lies either on the boundary or in the interior of the convex hull. The points lying on the boundary of the convex hull are sometimes referred to as the *vertices* of the convex hull. See Figure 1.1 for the convex hull of a set of points in two dimensions. The computational problem for the convex hull is typically to specify the points lying on the boundary of the convex hull in some pre-specified manner (for example, the vertices listed in counter-clockwise order for the two-dimensional convex hull). The concept of convex hulls in two dimensions can be visualized

as follows: Consider the two-dimensional plane to be the floor and imagine the input set of points to be nails hammered into the floor at those points. Now suppose that a rubber-band is stretched all around the nails and then released. The shape that the rubber-band rests in is the boundary of the convex hull, with a nail at each vertex of the hull. Typically, $CH(S)$ is used interchangeably to refer to either the convex hull of S or the boundary of the convex hull, and the reference will be clear from the context. For a set of points in three dimensions, the convex hull is a convex polyhedron and each facet (face) of the polyhedron will be a convex polygon. Under the general position assumption that no four points are coplanar, each facet will be a triangle.

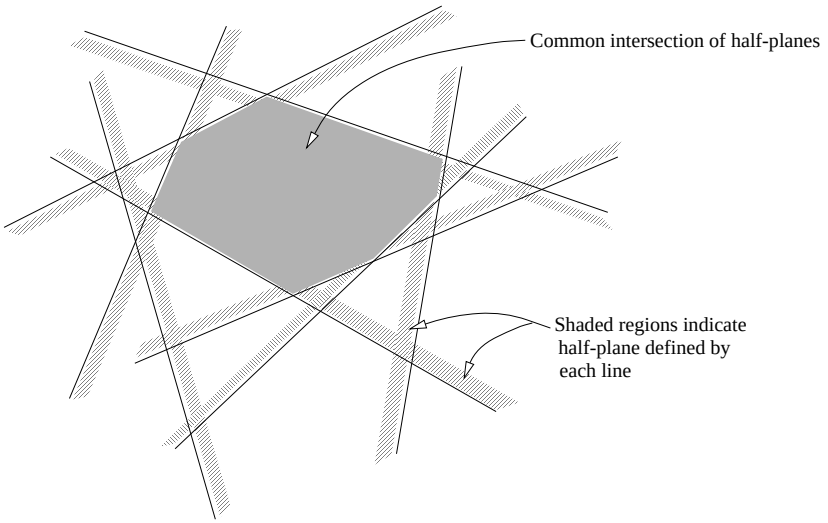

Figure 1.2 The intersection of a set of half-planes

Another fundamental problem in computational geometry is that of computing the intersection of n half-planes (or half-spaces, in three dimensions). A *half-plane* is the set of all points in the plane satisfying the linear inequality (or constraint) $ax + by + c \leq 0$. The common *intersection of n half-planes* is the region that satisfies n such linear inequalities simultaneously. See Figure 1.2 for an illustration of the problem in two dimensions. The problem in three dimensions is analogous, except that the inequalities are of the form $ax + by + cz + d \leq 0$.

The problems of computing the convex hull of a set of n points and that of computing the intersection of n half-planes are equivalent to one another due to the following *geometric duality transform*: A point (a, b) in the plane is mapped into a non-vertical line $ax + by + 1 = 0$, and any line $a'x + b'y + 1 = 0$ (a line not containing the origin) is mapped into the point (a', b'). Assume that the origin lies in the interior of the hull; it is easy to translate the points so that this is the case: simply take the origin to be the centroid of any three

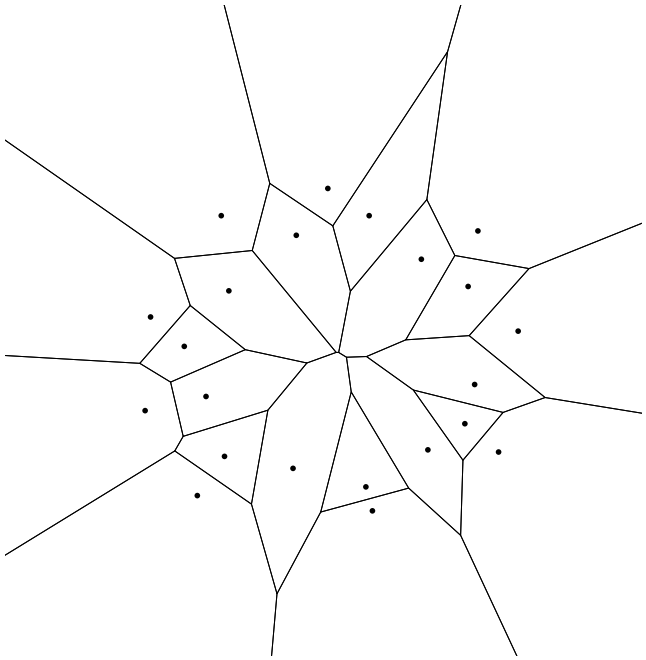

Figure 1.3 The nearest point Voronoi diagram of a set of points.

of the input points. Computing the convex hull of a set S of n points can be reduced to computing the intersection of n half-planes as follows: each point in S is mapped into a line by using the duality transform and the half-plane determined by this line is the one containing the origin. It can be shown that the half-planes that determine the boundary of the intersection are exactly the dual transforms of the vertices of $CH(S)$. Hence the same algorithm can be used to compute both the convex hull and the intersection of half-planes. The algorithms discussed in this chapter will typically be for the latter.

Voronoi Diagrams. *Voronoi diagrams* are elegant and versatile geometric structures with numerous applications. The Voronoi diagram or, more accurately, the *nearest point Voronoi diagram* of a set of objects S is defined as follows: The *Voronoi region* associated with an element from S is the set of all points in the plane that are closer to that element than to any other element in S. The nearest point Voronoi diagram is the union of all the Voronoi regions. Figure 1.3 shows the Voronoi diagram of a planar set of points. The Voronoi region associated with each point of S is a convex region given by the intersection of half-planes as follows: For each point p and any other point q in S , the region that is closer to p than to q is the half-plane containing p and bounded by the straight line bisector of p and q.

When S consists of a set of points, the Voronoi edges are all straight line segments, whereas if S consists of line segments, the Voronoi regions are bounded by parabolic arcs as well as straight line segments and might therefore be non-convex. A careful definition of the bisector of two line segments is given below, for which we need to clearly define the distance relation for line segments.

Let S be a set of nonintersecting closed line segments in the plane. Following the convention in [25, 43], we will consider each segment $s \in S$ to be composed of three distinct objects: the two endpoints of s and the open line segment bounded by those endpoints. The Euclidean distance between two points p and q is denoted by $d(p,\ q)$. The *projection* of a point q on to a closed line segment s with endpoints a and b, denoted $proj(q,\ s)$, is defined as follows: Let p be the intersection point of the straight line containing s (call this line $\overleftrightarrow{s}$), and the line going through q that is perpendicular to $\overleftrightarrow{s}$. If p belongs to s, then $proj(q,\ s) = p$. If not, then $proj(q,\ s) = a$ if $d(q,\ a) < d(q,\ b)$ and $proj(q,\ s) = b$, otherwise. The *distance* of a point q from a closed line segment s is nothing but $d(q,\ proj(q,\ s))$. By an abuse of notation, we denote this distance as $d(q,\ s)$. Let s_1 and s_2 be two objects in S. The *bisector* of s_1 and s_2, $B(s_1,\ s_2)$, is the locus of all points q that are equidistant from s_1 and s_2 i.e. $d(q,\ s_1) = d(q,\ s_2)$. Since the objects in S are either points or open line segments, the bisector will be part of either a line or a parabola. The bisector of two line segments is shown in Figure 1.4aa, and the Voronoi diagram of a set of line segments is shown in Figure 1.4bb. Clearly if S is a set of points, all the bisectors are parts of straight lines.

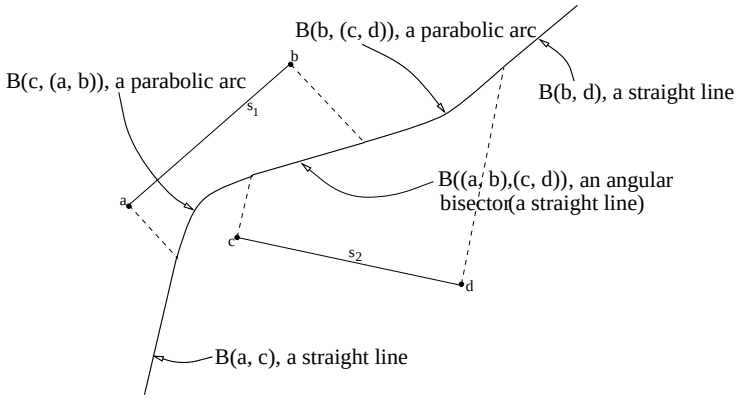

Figure 1.4a The bisector of two line segments

Definition 2 *The* Voronoi region, $V(s)$, *associated with an object s in S is the locus of all points that are closer to s than to any other object in S i.e. $V(s) = \{p \mid d(p,\ s) \leq d(p,\ s')$ for all $s' \in S\}$. The* Voronoi diagram *of S, $Vor(S)$, is the union of the Voronoi regions $V(s),\ s \in S$. The boundary*

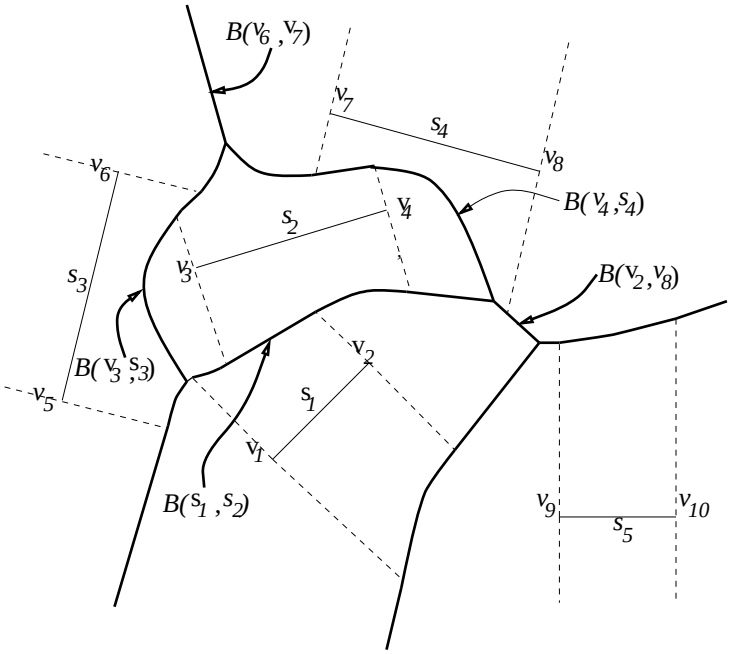

Figure 1.4b The Voronoi diagram of a set of line segments

edges of the Voronoi regions are called Voronoi edges*, and the vertices of the diagram,* Voronoi vertices*.*

The following is an important and useful property of $Vor(S)$.

Theorem 1 (Lee & Drysdale [25]) *Given a set S of n objects in the plane (here, these objects will either be nonintersecting closed line segments or points), 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.*

The problem of computing the Voronoi diagram of a set S of points in two dimensions can be reduced to the convex hull problem for a set of points in three dimensions by using the following reduction, as shown by Brown [8]: Assume that the planar set of points S lies on $z = 1$ and consider the paraboloid defined by $z = x^2 + y^2 + 1$. Each (x_i, y_i) in S is mapped onto the point $(x_i, y_i, {x_i}^2 + {y_i}^2 + 1)$ on the paraboloid. It can be shown then that the convex hull of the set of points on the paraboloid, when projected onto the plane $z = 1$, yields the Voronoi diagram of S. In fact, this relation between convex hulls and Voronoi diagrams extends to higher dimensions. In other words, the Voronoi diagram of a set of points in any dimension d can be obtained from the convex hull of an appropriately defined set of points in one higher dimension.

1.3 THE USE OF RANDOMIZATION IN COMPUTATIONAL GEOMETRY

The technique of randomization has been used to design sequential as well as parallel algorithms for a wide variety of problems. In particular, efficient randomized algorithms have been developed for a number of computational geometry problems. Recent work by Clarkson [13], Clarkson and Shor [14], Mulmuley [30], and Haussler and Welzl [23] has shown that random sampling can be used to obtain better upper bounds for various geometric problems such as higher-dimensional convex hulls, half-space range reporting, segment intersections, linear programming, *etc.*

Clarkson and Shor [14] used random sampling techniques to obtain tight bounds on the *expected* use of resources by algorithms for various geometric problems. The main idea behind their general technique is to use random sampling to divide the problem into smaller ones. The manner in which the random sample is used to divide the original input into subproblems depends on the particular geometric problem under consideration. They showed that for a variety of such problems:

Lemma 1 (Clarkson & Shor [14]) *Given a randomly chosen subset R of size r from a set of objects S of size n, the following two conditions hold with probability at least $1/2$:*

(a) the maximum size of a subproblem is $O((n/r)\log r)$, and

(b) the total size of all the subproblems is $O(n)$.

A sample that satisfies these conditions is said to be *good*, and *bad* otherwise. The above lemma implies that any randomly chosen sample is good with constant probability, and hence bad with constant probability as well.

The use of random sampling leads naturally to parallel recursive algorithms: Each subproblem defined by a random sample is solved *recursively* and in parallel. Typically, the number of processors required to solve each subproblem (recursively) will be proportional to the size of the subproblem itself. However, a number of issues relevant to the parallel environment need to be addressed in order for this approach to result in efficient parallel algorithms. These issues are discussed in the remainder of this section.

1.3.1 Randomized Techniques for Parallel Algorithm Design

Polling. Clarkson and Shor's results yield bounds on the *expected* use of resources, but do not give high probability results (i.e. bounds that hold with probability $\geq (1 - 1/n^{\alpha})$, where n is the input size, and $\alpha > 0$). Observe that this fact proves to be an impediment in the parallel environment due to the following reason [37]: As stated earlier, parallel algorithms for such problems are typically recursive. For sequential algorithms, since the expectation of the sum is the sum of expectations, it is enough to bound the expected run-time

of each recursive step. For recursive parallel algorithms, the run-time at any stage of the recursion will be the *maximum* of the run-times of the subproblems spawned at that stage. There is no way of determining the maximum of expected run-times without using higher moments. Moreover, even if we can bound the expected run-time at the lowest level of the recursion, this bound turns out to be too weak to bound the total run-time of the algorithm.

Reif and Sen [37, 38] give a novel technique called *polling* to tackle this problem. A parallel recursive algorithm can be thought of as a *process tree*, where a node corresponds to a procedure at a particular stage of the recursion, and the children of that node correspond to the subproblems created at that stage. The following theorem states the important result that if the time taken at a node which is at a distance i from the root is $O((\log n)/2^i)$ with high probability, then the run-time of the entire algorithm is $\tilde{O}(\log n)$. Note that the number of levels in the process tree will be $O(\log \log n)$.

Theorem 2 (Reif & Sen [38]) *Given a process tree that has the property that a procedure at depth i from the root takes time T_i such that*

$$Pr[T_i \geq k(\epsilon')^i \alpha \log n] \ \leq \ 2^{-(\epsilon')^i \alpha \log n},$$

then all the leaf-level procedures are completed in $\tilde{O}(\log n)$ time, where k and α are constants greater than zero, and $0 < \epsilon' < 1$.

The basic idea of the technique given in [37] is to find at every level of the process tree, a *good sample* of size $O(n^\epsilon)$ *with high probability* (where n can be thought of as the size of either the original input or the input to a subproblem). By doing this, they can show that the run-time of the processes at level i of the tree is $O(\log n/2^i)$ with high probability and hence the run-time of the entire algorithm is $O(\log n)$ with high probability.

By choosing a number of random samples (say $g(n)$ of them; typically $g(n) = O(\log n)$), we are guaranteed that one of them will be good with high likelihood. The procedure to determine if a sample is good or not will have to be repeated for each of the $g(n)$ samples. However, we would have to ensure that this does not cause the processor bound of $O(n)$ to be exceeded. This is achieved by *polling* i.e. using only a fraction of the input ($1/g(n)$, typically) to determine the "goodness" of a sample. The idea is that the assessment of a sample (good or bad) made by this smaller set is a very good estimate of the assessment that would be made by the entire set. Thus Reif and Sen give a method to find a good sample efficiently at every level of the process tree, and this idea is useful for converting expected value results into high probability results.

Two-Stage Sampling. It is important to consider the following side-effect that occurs in such recursive algorithms: When a random sample is used to divide the original problem into smaller ones, the total size of the subproblems can be bounded to only within a constant multiple of n. In a recursive algorithm, this results in an increase in the total problem size as the number of recursive levels increases. For a sample size of $O(n^\epsilon)$, the depth of the process tree for a

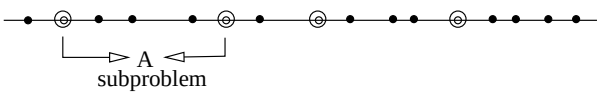

Figure 1.5 Subproblems in "one-dimensional" problems

parallel randomized algorithm would be $O(\log \log n)$, and even this could result in a polylogarithmic factor increase in the total problem size.

Observe that the issue of bounding the total size of the subproblems does not come up in "one-dimensional" problems like sorting because each element of the input set can lie in exactly one subproblem. This is not the case for problems such as convex hull or Voronoi diagram construction. This is illustrated in Figure 1.5, where the numbers that form the random sample are circled. The other numbers, shown as solid dots, fall in *exactly one subproblem* (each subproblem is defined by two consecutive numbers in the random sample), which will be solved recursively. In most geometric problems, however, each element of the input will typically fall in *several subproblems*. This means that the total subproblem size is not, in general, exactly equal to the original input size. Even with a good sample, we can only succeed in bounding the total to within a constant multiple of the original, which results in a problem size "blow-up" due to recursive calls.

In [14], Clarkson and Shor get around this problem by using only a constant number of levels of recursion in their algorithm. They are able to do this by combining the divide-and-conquer technique with incremental techniques (which are inherently sequential; see [14] for further details). Reif and Sen's [37] strategy to handle this problem is to eliminate redundancy at every level of the process tree. In other words, since it is known that the *final* output size is $O(n)$, it is possible to eliminate those input elements from a subproblem which do not contribute to the final output. By doing this, they bound the total problem size at *every* level of the process tree to be within $c'.n$ for some constant c'. This step is non-trivial and, in general, avoiding a growth in problem size in this manner can be quite complicated. Moreover, the strategy used to eliminate redundancy seems to be very problem-specific.

Rajasekaran and Ramaswami [34] describe a *two-stage random sampling* technique, which helps to overcome the problem of increase in total input size as the algorithm proceeds down the process tree. Their approach gets rid of the need to eliminate redundancy at each level of the process tree. In other words, it is not necessary to devise a method to control total problem size at every level of the process tree. By choosing much larger sized samples (of size $O(n/\log^q n)$ for an appropriate q) in the first stage of their algorithm, the polylog factor increase in processor bound still maintains the *total* processor bound as $O(n)$ at this stage. If this larger sized sample is good, it will again divide the original input into smaller problems of roughly equal size and the total subproblem size will be $O(n)$. Since the sample size is larger, the subproblem size will be

relatively small and can be solved by using non-optimal techniques. As before, to ensure high probability bounds, $O(\log n)$ such samples of larger size are chosen. Consequently, the two-stage sampling approach eliminates the problem posed by the polylog-factor increase in problem size. (The idea of two-stage sampling in a somewhat different form was independently discovered by Amato, Goodrich and Ramos [3], which they called *biased sampling*.)

In the following section, we elaborate upon these techniques. Even though these parallel randomized techniques are general, and apply to a wide variety of problems, they will be discussed with reference to particular problems in computational geometry. This allows for a clearer exposition on the main ideas behind the techniques, and we will see how they lead to efficient parallel algorithms for these fundamental geometric problems.

1.4 APPLICATIONS TO FUNDAMENTAL GEOMETRIC PROBLEMS

1.4.1 Convex Hull of Points in Three Dimensions

The technique of *polling* is used to give an optimal parallel randomized algorithm on the CREW PRAM for constructing the convex hull of a set of n points in three dimensions [37]. More accurately, an algorithm for the dual problem of computing the intersection of a set S of n half-spaces is given. The recursive parallel algorithm may be informally described as follows: (a) Compute the intersection of a random sample R (of size n^ϵ, say) of half-spaces and consider a point p inside it. (b) The intersection of R is a polyhedron composed of facets. Each of these facets, along with the point p, forms a "wedge". Observe that the final result will be a subset of this polyhedron, and only those half-spaces that intersect the interior of the polyhedron will contribute to the final result. In particular, each wedge will be intersected by a subset of planes which are possible contributors to that part of the final polyhedron that lies in that wedge. Therefore, each wedge defines a subproblem and the set of planes that intersect each wedge defines the input to the subproblem defined by that wedge. See Figure 1.6 for an illustration in two dimensions. As seen, each plane (line, in two dimensions) can intersect a number of wedges and hence may lie in several subproblems.

As discussed in Section 1.3.1, it is necessary to obtain a good sample R with high probability at each level of the recursion. This is done by choosing independently $O(\log n)$ samples and observing that there is a high probability that at least one of them will be good, since there is a constant probability that a sample will be bad; refer to Lemma 1. Recall that to determine if a sample is good, we need to

1. Find the maximum size of the subproblems (this must be $O(n^{1-\epsilon}\log n)$ for a good sample)

2. Find the total size of all the subproblems (this must be $O(n)$ for a good sample)

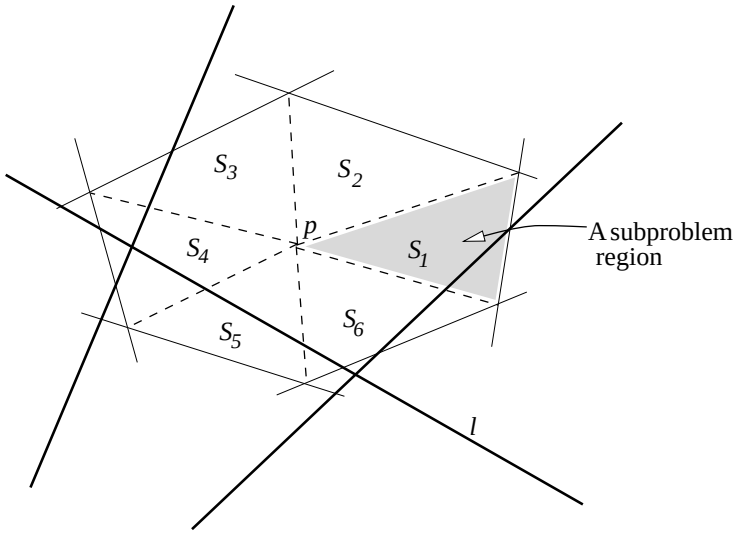

Figure 1.6 Intersection of half-planes: l lies in subproblems S_4, S_5 and S_6 (the thin lines form the random sample)

One way to do this is to use a search procedure to determine for each half-plane in S the subproblems in which it lies, i.e. the "wedges" that it intersects. (We will not go into the details of the search procedure here and refer the interested reader to [10, 37, 35]). Such a procedure would use $O(\log n)$ time with one processor per half-plane. Since there are $O(\log n)$ random samples, this search procedure would exceed the available resource bounds (the goal being an $O(\log n)$ time algorithm for the intersection of n half-planes using $O(n)$ processors). To overcome this obstacle, Reif and Sen [37] use the technique of *polling*, which can be described as follows: Instead of using the entire input S to determine the goodness of a sample, determine goodness from only *a fraction* of the input. (This is analogous to opinion polls that estimate the opinion of the population from polls conducted on a sample of the population; hence the term polling.) We summarize the probabilistic analysis for polling in the following paragraph, which refers to some basic facts from probability theory and can be found in any standard text on the subject (see [2, 18, 22], for example).

For each sample $R_j (1 \leq j \leq b \log n$, for some constant $b)$, choose $c_0 \cdot n/\log^d n$ half-planes randomly from the input (where c_0 and $d > 2$ are appropriately chosen constants) to determine the goodness of R_j. Suppose A_i^j is the number of these half-planes that lie in the i-th subproblem given by R_j. Suppose X_i^j is the actual number of half-planes from S that lie in the i-th subproblem given by R_j. Then, A_i^j is a binomial random variable with parameters $c_0 n/\log^d n$ (which is the total number of trials) and X_i^j/n (the probability of success of a trial).

One can then use Chernoff bounds [11, 22] to tightly bound the estimates for X_i^j. In other words, it is possible to show that

$$A_i^j \log^d n / c_0 c_2 \alpha \leq X_i^j \leq A_i^j \log^d n / c_1 \alpha$$

with high probability (i.e. probability $> 1 - 1/n^\alpha$) for appropriately chosen constants c_1, c_2 and α, independent of n. Therefore, one can use $\sum_i A_i^j$ to obtain high-probability estimates on $\sum_i X_i^j$, which is used to determine if a sample is good or not. Since only $c_0 n / \log^d n$ planes are used to determine the goodness of each sample (which takes $O(\log n)$ time per plane), and there are $O(\log n)$ samples, this part of the algorithm takes $\tilde{O}(\log n)$ time with $O(n)$ processors. We thus have the following lemma.

Lemma 2 (Reif & Sen [37]) *Given a method of choosing random samples that expect to be good, the* polling *algorithm gives a method to efficiently obtain a random sample that has a high probability of being good.*

The above method does not yet yield an optimal solution to the convex hull problem: it is also necessary to control the total size of all subproblems at each level of recursion. As discussed in Section 1.3.1, since the total subproblem size is bound to only within a constant multiple of n, this could result in a polylog factor problem size "blow-up", causing an increase in processor bound. In [37], the authors address this issue by exploiting the geometric properties of the specific problem. They carry out an exhaustive case analysis in order to remove from each subproblem all half-planes that are redundant. In other words, they remove from each subproblem those half-planes that cannot possibly form part of the output. By doing this, they ensure that the total problem size at *each* level of recursion is at most $c'n$ for some constant c'. Finally, by using a careful processor allocation strategy, they obtain an optimal algorithm for constructing the convex hull of a set of n points in three dimensions on the CREW PRAM. An alternative approach, called *pruning*, to control total problem size at each level of recursion in this algorithm is given by Amato, Goodrich and Ramos in [3]. This approach is used to obtain an optimal randomized parallel algorithm for this problem with the same bounds and on the EREW PRAM.

Theorem 3 (Reif & Sen [37], Amato, Goodrich & Ramos [3]) *The intersection of n half-spaces in three dimensions can be computed in $O(\log n)$ time with high probability using $O(n)$ processors on the CREW PRAM [37], or on the EREW PRAM [3].*

The technique in [37] of controlling total problem size at each level of recursion depends very much on the particular problem at hand, and such an approach is not always fruitful for more complex geometric problems. Therefore, we have deliberately avoided going into the details of this part of the algorithm for the convex hull problem, because in the following section we discuss a general strategy for dealing with the problem of increase in total subproblem size. This technique, which uses sampling at two stages of the algorithm, has been

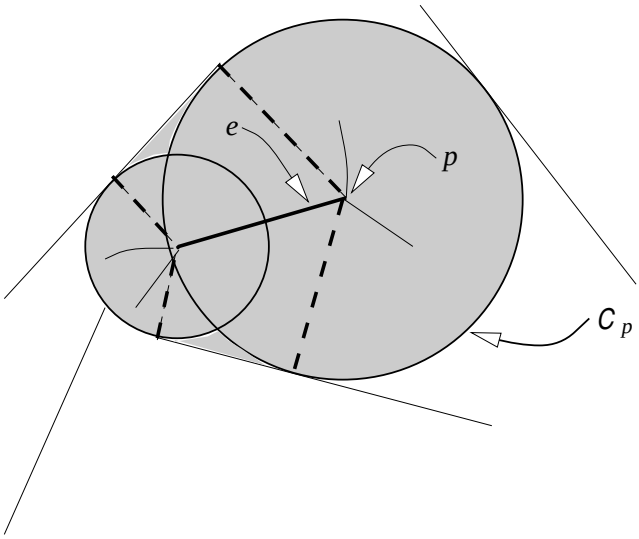

Figure 1.7 Subproblem regions for the Voronoi diagram of line segments

used to obtain an optimal parallel randomized algorithm for the Voronoi diagram of line segments [34]. It has also been used, along with other techniques, for designing efficient algorithms for higher-dimensional convex hulls [3].

1.4.2 Voronoi Diagrams

The optimal randomized parallel algorithm for three-dimensional convex hulls immediately gives an optimal algorithm for constructing the Voronoi diagram of a set of n points in the plane. As mentioned at the end of Section 1.2.1, this is due to the reduction from higher-dimensional convex hulls to Voronoi diagrams in one lower dimension. However, no such reduction is at hand for the problem of computing the Voronoi diagram of a set of *line segments*. This section describes the main ideas behind an optimal randomized parallel solution for this problem, which is the only known optimal solution. This technique can also be applied for Voronoi diagram construction of planar sets of points, giving an alternative optimal solution for the problem.

The algorithm for Voronoi diagram construction also uses the familiar random sampling approach. Let $S = \{s_1, s_2, \ldots, s_n\}$ be the input set of line segments in the plane and let R be a random sample from S. Let $|R| = n^\epsilon$ for some $0 < \epsilon < 1$. The sample R will be used to divide the original input S into smaller subproblems so that each of these can be solved in parallel. The subproblems are defined as follows: Each Voronoi edge of $Vor(R)$ defines a subproblem region. Rather than giving a rigorous definition of how these regions are defined, we refer to Figure 1.7 and appeal to intuition. Consider any point p on a Voronoi edge e of R. This point p defines a circle $\mathcal{C}_p$ such that

1. $\mathcal{C}_p$ has an empty interior (i.e., no objects of R intersect its interior) and
2. either two (when p lies in the interior of the Voronoi edge) or three (when p is a vertex of the Voronoi edge) objects of R are incident on the boundary of $\mathcal{C}_p$.

The subproblem region defined by each Voronoi edge e is simply the collection of such circles (and their interiors) given by the points on the edge. This is the shaded region in Figure 1.7. Any line segment that intersects this region will belong to the subproblem defined by e. Observe that these are the line segments whose final Voronoi regions might intersect e. It follows from Clarkson and Shor's [14] random sampling lemma that there is a fixed probability that such a sample will be good. In other words, the subproblem regions defined in the above manner will give subproblems of maximum size $O(n^{1-\epsilon}\log n)$ and total size $O(n)$.

The overall approach can then be outlined as follows:

- Construct the Voronoi diagram of R using some brute force technique (any approach that uses $O(\log n)$ time and a polynomial number of processors will do). Call this diagram $Vor(R)$. We use $Vor(R)$ to divide the original problem into smaller problems which will be solved in parallel.
- Process $Vor(R)$ appropriately in order to efficiently find the input set of line segments for each of these subproblems.
- Recursively compute (in parallel) the Voronoi diagram of each subproblem.
- Obtain the final Voronoi diagram from the recursively computed Voronoi diagrams.

By choosing an appropriate ϵ, we can ensure that the first step is done in $O(\log n)$ time using n processors. Randomized search techniques can be used to efficiently find the subproblems defined by a chosen sample; in particular, in $O(\log n)$ time with high probability using $O(n)$ processors. Keep in mind that, just as in the convex hull problem of the previous section, it is necessary to find a good sample with high probability at each level of recursion, which is done using polling. Parallel merge techniques can be used to compute the Voronoi diagram from recursively computed Voronoi diagrams and this can be done $O(\log n)$ time using $O(n)$ processors. Thus, the recurrence relation for the run-time is $T(n) = T(n^{1-\epsilon}) + \tilde{O}(\log n)$, which solves to $\tilde{O}(\log n)$.

However, the description of the algorithm that we have given here is incomplete. In particular, we need to tackle the problem of total problem size "blow-up" during recursive calls, and a technique called *two-stage sampling* is used to achieve this. This technique uses random sampling at two stages of the algorithm and, in essence, eliminates the need to control total problem size at each level of recursion. The remainder of this section summarizes this technique. (The other details of the algorithm for Voronoi diagram construction,

in particular the search and merge steps, will not be discussed here and the interested reader is referred to [34].)

Suppose that after $O(\log \log n)$ levels of recursion, the total size of all the subproblems at the leaf level of the process tree is at most $O(n.\log^c n)$, for some constant c. Then we have the following lemma:

Lemma 3 (Rajasekaran & Ramaswami [34]) *The Voronoi diagram of a set of n line segments can be constructed in $O(\log n)$ time with high probability using $n\log^c n$ processors, where $c > 0$ is a constant.*

The above lemma suggests that we might choose samples of size much larger than $O(n^\epsilon)$. In particular, such a sample S' could be of size $O(n/\log^q n)$, q being a constant integer $>$ c. If S' is a good sample, then it too will divide the original input into smaller problems of roughly equal size. Since this sample size is larger, the subproblems defined by this sample can be solved using any non-optimal technique (that uses a linear number of processors and polylogarithmic time). It is still necessary, however, to find a good sample S' with high probability. As before, this is done by choosing $O(\log n)$ such samples, at least one of which will be good with high probability.

Let $N = n/log^q n$. Let $S_1,\ S_2,\ \ldots,\ S_{d \log n}$ be the $O(\log n)$ samples of size N each, where d is a positive integer chosen according to the desired success probability at this stage of the algorithm. Since the size of S_i is large, we cannot afford to construct $Vor(S_i)$ using a brute force technique (as we can do with samples of size $O(n^\epsilon)$). Instead, we will run the randomized parallel algorithm on each S_i. Notice that we would only need $O(n)$ processors in order to do this. The outline of the algorithm is given as follows.

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

The function PICK_THE_RIGHT_SAMPLE picks a good sample from the S_i. This is done by constructing the Voronoi diagram for each S_i (using a randomized parallel algorithm) and computing the total subproblem size for each S_i in order to test the sample for goodness. Note that the testing of the samples S_i is done with respect to a restricted input set (polling). Let $R^i_1,\ R^i_2,\ \ldots,\ R^i_{a \log N}$

be the $O(\log N)$ random samples, each of size N^ϵ, chosen from S_i. In the following, it can be shown that every one of $Vor(S_i)$ will be constructed in $O(\log n)$ time with high probability. The function is given as follows:

Function PICK_THE_RIGHT_SAMPLE($S_1, S_2, \ldots, S_{d \log n}, I$);

- Do the following in parallel for each S_i ($1 \leq i \leq d \log n$).
  1. (a) Choose $a \log n$ random samples $R_1^i, R_2^i, \ldots, R_{a \log n}^i$ each of size N^ϵ from the set S_i.
     (b) Construct the Voronoi diagram of each R_j^i ($1 \leq j \leq a \log N$) using a brute force technique (that runs in logarithmic time with a polynomial number of processors)
     (c) Determine which of these R_j^i is a good sample for S_i (Note: Polling will not be necessary here due to the smaller input size $|S_i|$). Suppose $R_{j'}^i$ is one such good sample; with high probability, there will be such a j'.
     (d) Use $R_{j'}^i$ to divide S_i into smaller subproblems.
     (e) Recursively compute (in parallel) the Voronoi diagram of each subproblem.
     (f) Obtain the final Voronoi diagram $Vor(S_i)$ from these recursively computed Voronoi diagrams.
  2. Compute the total subproblems size when restricted to I (this is polling).
- Return the best S_i; with high probability there will be such an S_i.

By developing efficient search strategies to determine subproblems, and to merge the recursively computed Voronoi diagrams, an optimal parallel randomized algorithm for the Voronoi diagram of line segments in the plane is obtained on the CRCW PRAM[3] and thus we have the following.

Theorem 4 (Rajasekaran & Ramaswami [34]) *The Voronoi diagram of a set of n non-intersecting line segments in the plane can be computed in $O(\log n)$ time with high probability using $O(n)$ processors on the CRCW PRAM.*

Note that in order to maintain a processor bound of $O(n)$, the larger sample sizes used in the first stage of the algorithm necessitate fast methods to determine subproblems. In other words, we cannot afford to have a parallel algorithm that uses a polynomial number of processors. Whereas in [37], since the sample size is always $O(n^\epsilon)$, an appropriate ϵ can be chosen such that the processor bound of $O(n)$ is maintained, we do not have this flexibility. This is

[3]In this PRAM model, concurrent reads and concurrent writes are both allowed. There are many protocols for resolving write conflicts in an algorithm. In this case, they are resolved arbitrarily i.e., an arbitrary processor is allowed to succeed.

because of the large sample size during the first stage of sampling. The interested reader is referred to [34] for the details of the search and merge steps. The two-stage sampling approach is general enough to apply to other problems as well. For instance, Reif and Sen's algorithm [37] for three-dimensional convex hulls can be simplified considerably by applying the idea of two-stage sampling. It also applies to the Voronoi diagram of points in the plane, thus giving an alternative optimal randomized parallel algorithm for this problem.

1.4.3 Higher-Dimensional Convex Hulls

The higher-dimensional convex hull problem refers to the problem of computing the convex hull of a set of n points in an arbitrary d-dimensional space (denoted by E^d). Assume for the remainder of the section that the point set contains the origin. The dual relationship between convex hulls of point sets and the intersection of half-planes holds in any dimension. Therefore, the higher-dimensional convex hull problem is equivalent to computing the intersection of n half-spaces (all containing the origin). A *hyperplane* is the set of all d-dimensional points $(x_1, x_2, \ldots, x_{d-1}, x_d)$ that satisfy the equality $a_d x_d + a_{d-1} x_{d-1} + \ldots + a_2 x_2 + a_1 x_1 + a_0 = 0$ and a half-space in d dimensions is the set of all points that satisfy the inequality $a_d x_d + a_{d-1} x_{d-1} + \ldots + a_2 x_2 + a_1 x_1 + a_0 \leq 0$. Computing the intersection of half-spaces in high dimensions is an important and fundamental problem in its own right. Furthermore, as mentioned earlier, their relationship to Voronoi diagrams in one lower dimensions imply that efficient algorithms for the former will be immediately applicable to the latter. The convex hull of n points in E^d has size $\Theta(n^{\lfloor d/2 \rfloor})$ in the worst case and constructing it takes time $\Omega(n \log n + n^{\lfloor d/2 \rfloor})$ [17]. One way of measuring the performance of a higher-dimensional convex hull algorithm is in terms of the worst-case size of the output (i.e. the hull); the results summarized here use this measurement.

The first sequential algorithms were given by Seidel [39, 40]. An algorithm optimal in even dimensions was given in [39] and ran in $O(n \log n + n^{\lceil d/2 \rceil})$ time and a $O(n^{\lfloor d/2 \rfloor} \log n)$ algorithm was given in [40]. Optimal randomized solutions were given by Clarkson and Shor [14] and by Seidel [41]. More recently, optimal deterministic solutions have been given by Chazelle [9] and Brönnimann, Chazelle and Matoušek [7]. Research on parallel algorithms for higher-dimensional convex hulls has begun more recently, and some of the recent results are summarized below. We will discuss only the randomized algorithm for higher-dimensional convex hulls in [3], which is also the best known result to date.

Amato, Goodrich and Ramos [3] give $O(\log n)$ time randomized parallel algorithms on the EREW PRAM, using optimal $O(n \log n + n^{\lfloor d/2 \rfloor})$ work with high probability, for the dual problem of constructing the intersection of n half-spaces in d-dimensional space. As in the algorithms discussed in Sections 1.4.1 and 1.4.2, this algorithm is also based on parallel divide-and-conquer techniques, where the d-dimensional space is divided into cells and the half-spaces that intersect the cells define the subproblems. However, the issue of bounding the total subproblem size to only within a constant factor of the original

problem size comes up here as well, causing an unacceptable increase in processor bound. A technique called *biased sampling*, which is similar to two-stage sampling (and was discovered independently around the same time), is used to avoid the total problem size increase at each level of the recursion. By combining this with other sophisticated geometric techniques (in particular, by using a parallel analog by Goodrich [20] of Matoušek's shallow-cutting lemma [27, 28]), they obtain the stated result. We give below a high-level description of this method and outline the main ideas. Several details will not be discussed and the reader is encouraged to look up [3, 20].

First we describe how the subproblems are defined: Given a set S of n hyperplanes in E^d, a 0-shallow $1/r$-cutting for S is a partition of E^d into simplices (informally, a tetrahedron in three dimensions is analogous to a triangle in two dimensions and a *simplex* is the generalization of a tetrahedron to arbitrary dimensions) such that each simplex is intersected by at most n/r hyperplanes from S, and the collection of simplices contains the intersection of S. Given a 0-shallow $1/r$-cutting for S of size $O(r^{\lfloor d/2 \rfloor})$ (where $r = n^{\epsilon}$, where ϵ is an appropriately defined constant), the next step will be analogous to the three-dimensional case: Each simplex defines a subproblem whose input is the set of hyperplanes that intersect that simplex. Each subproblem is then solved recursively. The recursion bottoms out when the problem size is a constant, at which point the problem can be solved using some obvious brute-force technique. As before, the number of levels of recursion will be $O(\log \log n)$. It is shown in [3] that finding a 0-shallow $1/r$-cutting takes $O(\log n)$ time on the EREW PRAM using $O(nr^{\lfloor d/2 \rfloor + c'})$ work, for some constant $c' > 1$. It follows therefore that

Lemma 4 (Amato, Goodrich & Ramos [3]) *The intersection of a set of n half-spaces in E^d can be computed in $O(\log n)$ time $O(n^{\lfloor d/2 \rfloor} \log^c n)$ work on the EREW PRAM, where $c > 0$ is some constant.*

An optimal algorithm for the intersection of half-spaces in high dimensions is obtained by running the above non-optimal algorithm, which uses too many processors, on very large samples (in a manner similar to the two-stage sampling technique used for the Voronoi diagram). In particular, take a random sample R of size $r = n/\log^{c_0} n$ for some constant $c_0 > 0$. From [3, 14], it follows that with a constant probability, R is a good sample. This means that the the simplices determined by the intersection of the half-spaces in R form a 0-shallow, $1/r$-cutting for S and the cutting has size $O(r^{\lfloor d/2 \rfloor})$. The work-inefficient algorithm described above is now run on R (this is referred to as the *bias* in the sampling in [3]) and the intersection of the half-spaces in R is obtained. By choosing c_0 large enough, this step of the algorithm takes $O(\log n)$ time using $O(n^{\lfloor d/2 \rfloor}/\log n)$ work.

In order to obtain high-probability bounds, $O(\log n)$ such random samples are chosen and the above procedure is carried out on each of them. This takes $O(\log n)$ time using $O(n^{\lfloor d/2 \rfloor})$ work with high probability. If a random sample R is indeed a good sample, it is a $1/r$-cutting of size $O(r^{\lfloor d/2 \rfloor})$ and so each simplex in the intersection of R intersects a set T of at most n/r hyperplanes

from S. For each such T, one can then use a non-optimal algorithm that runs in poly-logarithmic time but does optimal work with high-probability. Refer to [3, 9] for the specifics of this step. It follows that

Theorem 5 (Amato, Goodrich & Ramos [3]) *The intersection of n half-spaces in E^d can be computed in $O(\log n)$ time using $O(n^{\lfloor d/2 \rfloor})$ work, for $d \geq 4$, with high probability on the EREW PRAM.*

1.5 SUMMARY

For some problems, randomization offers a simpler and more elegant alternative to a deterministic solution. But more importantly, randomization proves to be a powerful tool in the design of efficient parallel algorithms for some fundamental problems in computational geometry, whereas there are no known deterministic counterparts that match these bounds. In particular, this chapter covers the techniques of polling and two-stage sampling, and their use in the design of optimal parallel solutions for the convex hull of points in three dimensions and the Voronoi diagram of line segments in the plane. Furthermore, the use of randomization to obtain an optimal parallel algorithm for the important problem of higher-dimensional convex hulls is also discussed. Efficient solutions to numerous geometric problems can be obtained from the parallel algorithms for these fundamental geometric problems. For instance, the three-dimensional convex hull algorithm immediately leads to optimal solutions for the Voronoi diagram of points in the plane, the all-points nearest neighbor problem and the Euclidean minimum spanning tree problem. Similarly, the algorithm for the Voronoi diagram of line segments gives optimal parallel solutions for the minimum weight spanning tree, nearest neighbor, largest empty circle and the all-pairs nearest neighbor for a set of line segments. In addition, the Voronoi diagram of line segments is used to plan the motion of an object (a disc, for example) from one point in the plane to another while avoiding polygonal obstacles (see [31] for details) and is also used to find the maximum-flow path of a liquid flowing through a polygonal pipe with a uniform capacity defined on its interior [29].

The selection of the above results is meant to provide a flavor of parallel randomized techniques for some fundamental geometric problems. The list is certainly not exhaustive, and several pertinent results have not been discussed since they lie outside the scope of this brief survey. It is hoped, however, that the selected results demonstrate the effectiveness of randomization in parallel algorithm design for problems in computational geometry.

References

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

[2] N. Alon, J. H. Spencer, and P. Erdös. *The Probabilistic Method*. Wiley-Interscience, New York, 1992.

[3] N. Amato, M. Goodrich, and E. Ramos. Parallel Algorithms for Higher-Dimensional Convex Hulls. In *Proc. of the 35th Annual IEEE Symp. on Foundations of Computer Science*, pages 683–694, October 1994.

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

[5] 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.

[6] M. J. Atallah and M. T. Goodrich. Deterministic parallel computational geometry. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, pages 497–536. Morgan Kaufmann Publishers Inc., 1993.

[7] H. Brönnimann, B. Chazelle, and J. Matoušek. Product range spaces, sensitive sampling and derandomization. In *Proc. 34th Annu. IEEE Symp. on Foundations of Computer Science*, pages 400–409, 1993.

[8] K. Q. Brown. *Geometric transforms for fast geometric algorithms.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, 1980.

[9] B. Chazelle. An optimal convex hull algorithm in any fixed dimension. *Discrete Comput. Geom.*, 10:377–409, 1993.

[10] B. Chazelle and D. Dobkin. Intersection of convex objects in two and three dimensions. *Journal of the ACM*, 34(1):1–27, 1987.

[11] H. Chernoff. A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the Sum of Observations. *Annals of Math. Stat.*, 2:493–509, 1952.

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

[13] K. Clarkson. New applications of random sampling in computational geometry. *Discrete Comput. Geom.*, 2:195–222, 1987.

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

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

[16] S. A. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM J. Comput.*, 15:87–97, 1986.

[17] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, NY, 1987.

[18] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. John Wiley, New York, NY, 1968.

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

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

[21] 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.

[22] T. Hagerup and C. Rüb. A guided tour of chernoff bounds. *Info. Proc. Lett.*, 33(10):305–308, 1990.

[23] D. Haussler and E. Welzl. ϵ-nets and Simplex Range Queries. *Discrete Comput. Geom.*, 2:127–152, 1987.

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

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

[26] C. Levcopoulos, J. Katajainen, and A. Lingas. An Optimal Expected-time Parallel Algorithm for Voronoi Diagrams. In *Proc. of the First Scandinavian Workshop on Algorithm Theory*, volume 318 of *Lecture Notes in Computer Science*, pages 190–198. Springer-Verlag, 1988.

[27] J. Matoušek. Cutting hyperplane arrangements. *Discrete Comput. Geom.*, 6:385–406, 1991.

[28] J. Matoušek. Reporting points in halfspaces. *Comput. Geom Theory Appl.*, 2(3):169–186, 1992.

[29] J. S. B Mitchell. On Maximum Flows in Polyhedral Domains. In *Proceedings of the 4th Annual ACM Symposium on Computational Geometry*, pages 341–351, 1988.

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

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

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

[33] M. O. Rabin. Probabilistic Algorithms. In J. Traub, editor, *Algorithms and Complexity, New Directions and Recent Results*, pages 21–36. Academic Press, 1976.

[34] S. Rajasekaran and S. Ramaswami. Optimal parallel randomized algorithms for the voronoi diagram of line segments in the plane and related problems. In *Proc. of the 10th Annual ACM Symp. on Computational Geometry*, pages 57–66, Stony Brook, New York, June 1994. Full paper submitted to *Algorithmica*.

[35] S. Rajasekaran and S. Sen. Random Sampling Techniques and Parallel Algorithm Design. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, pages 411–451. Morgan Kaufmann Publishers, Inc., 1993.

[36] E. Ramos. Construction of 1-d lower envelopes and applications. In *Proc. of the 13th Annual ACM Symp. on Computational Geometry*, pages 57–66, Nice, France, June 1997.

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

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

[39] R. Seidel. A convex hull algorithm optimal for point sets in even dimensions. Master's thesis, Dept. Computer Sci., Univ. British Columbia, Vancouver, BC, 1981.

[40] R. Seidel. Constructing higher-dimensional convex hulls at logarithmic cost per face. In *Proc. 18th Annu. ACM Symp. on the Theory of Computing*, pages 404–413, 1986.

[41] R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete Comput. Geom.*, 6:423–434, 1991.

[42] R. Solovay and V. Strassen. A Fast Monte-Carlo Test for Primality. *SIAM J. Computing*, 6(1):84–85, 1977.

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

Suneeta Ramaswami is an assistant professor at the department of Computer Sciences in Rutgers University, Camden, which she joined in September 1997. Prior to that, she was a post-doctoral fellow with the Computational Geometry Research group in the School of Computer Science at McGill University in Montreal. She received her Ph.D. from the Computer and Information Science department at the University of Pennsylvania in December 1994 and her B.A. in Mathematics and Computer Science from Wellesley College in May 1988.

Dr. Ramaswami's research interests are in the areas of Computational Geometry and the design, analysis and implementation of algorithms for geometric problems in visualization, CAD/CAM (Computer-Aided Design and Manufacturing), computer graphics, robotics and Geographic Information Systems (GIS). Her broader research interests include randomized and parallel algorithms, and computational biology. Her current work is on geometric techniques for problems in the modeling and visualization of data, mesh-generation, computational tomography, Geographic Information Systems (geometric approaches and external-memory problems), facility location problems and non-degeneracy assumptions in computational geometry.