Order Number 9025050

Random sampling techniques for efficient parallel algorithms in computational geometry

Sen, Sandeep, Ph.D.

Duke University, 1989

U·M·I

300 N. Zeeb Rd.
Ann Arbor, MI 48106

RANDOM SAMPLING TECHNIQUES FOR EFFICIENT PARALLEL ALGORITHMS IN COMPUTATIONAL GEOMETRY

by

Sandeep Sen

Department of Computer Science
Duke University

Date: 12/12/89

Approved:

John Reif, Supervisor

Henry Greenside

Donald Loveland

Donald Rose

Matt Stallman

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University

1989

ABSTRACT

RANDOM SAMPLING TECHNIQUES FOR EFFICIENT PARALLEL ALGORITHMS IN COMPUTATIONAL GEOMETRY

by

Sandeep Sen

Computer Science Department
Duke University

Date: 12/12/89

Approved:

John H. Reif, Supervisor

Henry Greenside

Donald Loveland

Donald Rose

Matt Stallman

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of doctor of philosophy in the Department of Computer Science in the Graduate School of Duke University

1989

Abstract

In this dissertation we investigate the use of random sampling techniques for obtaining parallel algorithms in computational geometry. One of the major thrust of research in parallel algorithms has been to minimize parallel time complexity without using a disproportionately large number of processors. In an ideal case the objective is to keep the the total number of operations within a constant factor of the best known sequential time bounds for these problems. These algorithms are referred to as being 'optimal'. The model of computation used throughout is the PRAM model.

We have considered very fundamental problems including planar-point location, two variable linear programming, triangulation of simple polygon, convex hulls in two and three dimensions. In most cases we have been able to obtain $O(\log n)$ time algorithm using an optimal number of processors (except triangulation). Being very fundamental in nature, these problems imply equally efficient algorithms for various other problems including the 2-D Voronoi diagram of point sites. Presently no such deterministic algorithm is known for 3-D convex hulls or 2-D Voronoi diagrams. The randomized techniques often result in simple algorithms (compared to their deterministic counterparts) which is one of the main contributions of this thesis.

For developing the algorithms we have been able to make use of some known randomized techniques but for the parallel environment we require additional techniques. The more sophisticated algorithms rely on a new method called 'Polling' which seems to be a useful technique for a number of problems in computational geometry. Using this method one is able to obtain high-confidence bounds for algorithms as opposed to expected bounds. We also outline methods for limiting the number of purely random bits used by our algorithms while maintaining the claimed asymptotic bounds.

To my parents:
Krishna and Sisir Sen

Acknowledgements

The work in this dissertation was inspired by my advisor, John Reif who suggested the use of randomization for parallel algorithms in computational geometry. Moreover he has been a constant source of new ideas which sustained my motivation through some very despairing periods. Apart from the dissertational work, I have benefited tremendously from his breadth of knowledge, innovativeness and an uncanny intuition for the right answers. I am certain that these will have a lasting impact on my future research career.

I am very grateful to Ken Clarkson who pointed out a crucial mistake in one of the preliminary versions of this work; the rectification of which led to some of the more interesting developments in this thesis. I am deeply indebted to Sanguthevar Rajasekaran for teaching me some of the basics of randomized techniques and always be willing to listen to some of my half-baked ideas. I would also like to thank all my teachers in computer science and otherwise who have taught me all that I may claim to know.

I am grateful to my fellow graduate students of this department some of who deserve special mention. Salman Azhar was a great help in meticulous reading of a number of preliminary drafts. He has also been a constant source of entertainment with his comical sense of humour and some lively discussions on cricket. Ronnie Smith kept be abreast of all other kinds of sports and even gave me tips to sharpen my tennis skills. Jitendra Apte was always there to answer another stupid question about my constant struggle with the friendly computer system. Ronnie and Sujit also spent valuable hours in careful reading of an earlier draft.

I also take this opportunity to thank my committee members for some insightful comments regarding presentation.

I am solely responsible for whatever errors may remain.

Contents

Chapter 1

INTRODUCTION AND PRELIMINARIES

1.1 The nature of our work

The design and analysis of algorithms has been recognized as one of the foundations of computer science. The concept of an algorithm dates back to at least 300 B.C. when Euclid described a method for determining the greatest common divisor of two integers (which had been around even before it appeared in Euclid's *Elements*) However, it is only since the advent of the modern computer that this area observed a phenomenal growth. Not only has the precise meaning of an algorithm received much attention, the research has also matured tremendously.

From a historical perspective, it would not be inappropriate to say that the geometric constructions described by Greeks, namely Euclid were the earliest systematic efforts which satisfied the requirements of an algorithm. The constructions used very rudimentary primitives (operations that could be done using rulers and compasses) which can be thought of as a model of computation. The motivation for these geometric constructions was driven by such

fundamental needs as measuring and partitioning land. Arguably, geometry is one of the most intuitive branches of mathematics besides arithmetic. The introduction of coordinates by Descartes laid the foundations for its interaction with algebraic and other analytical techniques and also gave a new interpretation to geometric complexity in terms of algebraic manipulations.

Algorithms have been an integral part of constructive mathematics and one of the earliest significant results was showing that the theory of real closed fields is decidable. Tarski's proof was converted into a more explicit algorithm by Collins [28] which would certainly fall under the category of geometric algorithms. In spite of such early recognition of the need for geometric constructions, its inception into the realm of an algorithmic discipline had to wait until Shamos's [71] Ph.D. thesis. He described a formal framework for studying the complexity of some very fundamental problems and christened the area as computational geometry. The planarity testing algorithm of Hopcroft and Tarjan had a strong flavor of geometric reasoning and can be viewed as an interaction of ideas from topology and geometry. The last decade has seen a phenomenal growth in the research activity in geometric algorithms.

A primary direction for research in computer science has been to look for increasingly faster methods of solving problems. Inspired by the development of low cost hardware and to circumvent the limitations imposed by lower bounds (which limits the computing speed of a uniprocessor machine), a major thrust of the research has been directed towards study of parallel and multiprocessing computers. The obvious goal is that by employing more than one processor simultaneously we will be able to compute faster. Although the idea of parallel computation has been around for a few decades, the recent progress in Integrated Circuit technology (making such machines feasible) has acted as a

strong catalyst. Consequently, the present day algorithm designer not only looks for algorithms which are more efficient but also easier to speed up when more than one processor is available. With this perspective it has become important to study the effect of parallelism on some fundamental geometric problems.

A further development in the study of algorithms is the use of random bits in the algorithm. In a nutshell, a randomized algorithm chooses to proceed along a particular path depending on the value of some random bits. It has often been observed that this additional feature makes some algorithms simpler and more efficient in practice. The impact of using randomization has been more significant in the study of parallel algorithms. In section 3, we shall look at these kinds of algorithms and their characteristics more closely.

The underlying machine model plays a critical role in the approach to algorithm design. For the purpose of mathematical analysis, and to focus on the crucial issues, abstraction of real machines becomes necessary. These abstract machine models have to be 'realistic' enough to capture the constraints imposed by the real-machine and also be general enough to allow the algorithm designer to develop his methods without being bogged down with excessive details. These two seemingly contradictory goals have often resulted in a rift between theoreticians and practitioners. Surprisingly, for sequential algorithms, the Random Access Machine (RAM) has now been universally accepted as a standard model of computation. This model has been able to capture the salient features of computers with different kinds of architectures and its success can be gauged by the fact that a large number of algorithms designed for the RAM model have been implemented in practical machines. Unfortunately, the scenario is not as clear for the parallel environment. There is a great deal

of disagreement as to what the right model for algorithm designers is and how effective they are. Presently, the most commonly used model is called the Parallel RAM (PRAM) model which appears to be a natural extension of the sequential model. We shall defer a more detailed discussion to a later section.

In this thesis, we present parallel algorithms for some fundamental problems in computational geometry which use randomization. In doing so, we have exploited known techniques from all these areas namely, geometric algorithms, parallel techniques and random-sampling methods and developed some new methods. In the latter part of this chapter we have stated some of the well-known results in the theory of parallel algorithms and computational geometry and have referred to relevant literature in case the reader is interested to pursue the details. In contrast, we have provided more details of the randomization techniques in the main part of the thesis which require more specialized treatment.

In the rest of this chapter, we give a description of the main results of this thesis and how they relate to other work in this area. We give a brief description of well-known results in computational geometry and some commonly used parallel techniques. One of the main contributions of this work is the successful use of randomization for parallel algorithms and hence we felt it was appropriate to motivate the reader with a brief history of randomized algorithms and its impact on algorithm design in the next section.

1.2 Randomized algorithms

Randomization was formally introduced by Rabin [61] and independently by Solovay & Strassen [74] as a tool for improving the efficiency of certain algo-

rithms. In a nutshell, in a randomized algorithm, the processor has access to random bits to make decisions at different steps of the algorithm. Therefore a randomized algorithm is actually a family of algorithms where each member of this family corresponds to a fixed sequence of the value of the random bits. More formally, a randomized algorithm $\mathcal{A}$ has a probability space (Ω, P) associated with it, where Ω is the sample space and P is some probability measure. The algorithm $\mathcal{A}$ is a function of input I and $w \in \Omega$. Two of the most commonly used forms of randomization in literature are the *Las Vegas* algorithms and *Monte Carlo* algorithms. The former kind ensures that the output of the algorithm is always correct - however only a fraction (usually greater than $1/2$) of this family of algorithms halt within a certain time bound (as well as with respect to some other resources like space). In contrast, the Monte Carlo procedures always halt in a pre-determined time period; however the final output is correct with a certain probability (typically $> 1/2$). This lends itself very naturally to decision algorithms (Rabin's primality testing being a good example). For the purpose of this discussion we shall limit ourselves to the Las Vegas algorithms which have been more popular with the algorithm designers. For a general algorithm which produces more than just 'yes-no' output, the precise meaning of an incorrect output becomes subjective; for example we may need to know how close we are to the correct output in order to decide if the output is acceptable. Although this is one of the reasons for bias towards Las Vegas algorithms, the use of either kind of algorithm depends on the particular application.

1.2.1 Complexity measures of randomized algorithms

Before we discuss the applications of these algorithms in parallel computing, it is important to review some of the performance measures used by these algorithms. This will enable us to compare the relative merits of different randomized algorithms. In the beginning, we must emphasize the distinctions between a randomized algorithm and a probabilistic algorithm. By probabilistic algorithms, we imply those algorithms whose performance depend on the input distribution. For such algorithms, we are often interested in the average resources used over all inputs (assuming a fixed probability distribution of the input). A randomized algorithm does not necessarily depend on the input distribution. A randomized algorithm uses a certain amount of resources for the worst-case input with probability $1 - \epsilon, (0 < \epsilon < 1)$, i.e. the bound holds for any input (which is a stronger bound than the average bounds). This can be very well illustrated with the example of Hoare's *Quicksort* algorithm. In its original form, it is a probabilistic algorithm which performs very well on certain inputs and deteriorates sharply on some other inputs. By assuming that all inputs are equally likely (known as random-input assumption), the algorithm performs very well on the average. By introducing randomization in the algorithm itself, it has been shown to perform very well on **all** inputs with high probability. This is certainly a more desirable property since a *malicious oracle* who could control the performance of the original algorithm by giving it worst case inputs, can no longer affect it. Of course, the onus of a successful run of the algorithm is now shifted to the outcome of the coin-flips. This depends on certain randomness properties of the random-number generator, which is a topic by itself. Also note that this discussion does not preclude de-

signing randomized algorithms which are dependent on the input distribution, but these algorithms are no different from their deterministic counterparts.

Until now we have characterized the randomized algorithms with a success probability of $1 - \epsilon$, without specifying the possible forms of ϵ. It can be a fixed constant or a function (which takes values between (0,1)). It must be clear that ϵ should be minimized (compare this with deterministic algorithms where ϵ is 0). Intuitively, we can expect a trade-off between ϵ and the amount of resource used. In other words, the failure probability ϵ must decrease with increasing amount of resources. Let us consider a concrete example. Suppose $T_A(n)$ is the *expected* running time of the randomized algorithm A for input size n. What can we say about ϵ? If we don't have any bounds other than the expectation we can only use Markov's inequality. From Markov's inequality, the probability that running time exceeds k $T_A(n)$ is less than $1/k$. For example, if $k = 2$, $\epsilon = 1/2$. Compare this with an algorithm B for the same problem whose running time exceeds $k\alpha T_B(n)$ with probability less than $1/n^\alpha$, and suppose that for any given α, k is a constant independent of n. This implies that the probability of failure diminishes rapidly as n increases and vanishes asymptotically. We have characterized the failure probability ϵ as a decreasing function of the problem size, n, and resources used by the algorithm. The reader will recognize that the faster ϵ decreases with these parameters the better is the algorithm. This makes algorithm B superior to algorithm A if $T_A(n)$ and $T_B(n)$ represent the same function. The basic idea is that depending on the application, the user chooses a certain value of ϵ and accordingly chooses k (given the value of n) with the objective of minimizing k. There is no reason to be pedantic about the kind of function ϵ should be except that a failure probability of the second form (that of algorithm B) has been very widely used

in literature, and such algorithms have been termed as having *high probability* of success. This kind of failure probability function is quite robust with respect to a polynomial number of procedures. In other words, the union of a polynomial number of events, each with high probability of success, succeeds with high probability.

1.2.2 A notation and its significance

The term *very high likelihood (probability)* is used in this thesis to denote probability greater than $1 - n^{-\alpha}$ for some $\alpha > 1$ where n is the input size. Just like the big-O function serves to represent the complexity bounds of deterministic algorithms, we shall use $\tilde{O}$ to represent complexity bounds of the randomized algorithms. We say that a randomized algorithm has resource bound $\tilde{O}$ $(f(n))$ if there is a constant c such that the resource used by the algorithm is no more than $c\alpha f(n)$ with probability $\geq 1 - 1/n^{\alpha}$ for any $\alpha > 1$. (An equivalent definition will be bounding the resource by $\alpha \cdot f(n)$ with probability greater than 1 - $n^{-c\alpha}$, and in the rest of the thesis they will be used in an interchangeable manner). An algorithm whose *expected* resource bound is O(f(n)) does not have any better confidence interval beyond using Markov's inequality (i.e. the probability that it exceeds the resource bound by a factor k is less than 1/k). This implies that the failure probability does not diminish as rapidly as the high likelihood bounds. High-likelihood bounds are especially useful for parallel algorithms, where we need to bound the time complexity of all the processes.

Lemma 1.1 *The union of k events (k being any polynomial in n), each of which succeeds with high probability also succeeds with high probability.*

If the failure probability of event i is $< 1/n^{\alpha_i}$ the failure probability of the union of the events is less than $\sum_{i=1}^{k} n^{-\alpha_i} < k/n^{\alpha}$ where $\alpha = min(\alpha_1, .., \alpha_k)$. By choosing large enough α, this is less than $n^{-(\alpha-\delta)}$ for $k = n^{\delta}$.

This simple fact will be used repeatedly throughout the rest of the thesis. It may be a non-trivial task to transform an algorithm like A to an algorithm like B (which succeeds with high probability) without loss of efficiency. The reader must also appreciate that randomized algorithms like B, which have such high probability of success would be competitive with deterministic algorithms for the same problem. According to Adleman & Manders [2], a randomized algorithm with success probability more than $1 - 2^{-k}$ (for some large fixed k) has a lower probability of failure than the hardware itself. Randomization has proved to be extremely effective in parallel algorithm design. One of the earliest treatments of this topic can be found in Reif [66]. For a more recent and extensive survey the reader is encouraged to read the first two chapters of Rajasekaran [62].

1.3 Parallel Algorithms : models and goals

In the quest for faster methods of computation, coupled with the development of VLSI technology, there has been a tremendous growth in research in parallel computation. The objective is to reduce the time complexity of the sequential algorithm by using several processors that are working on the same problem simultaneously. If we let $Seq(n)$ denote the best known sequential time bound for a problem of size n, then clearly the parallel time complexity can be no better than $Seq(n)/P$ where P is the number of processors being used. Otherwise, we can improve on the best sequential algorithm by taking the parallel

algorithm and executing the parallel steps sequentially. Throughout this dissertation, unless otherwise stated, the parameter n will be used to represent the input size of the problem under consideration.

A major drawback in the area of parallel computation is the lack of consensus regarding an ideal model of parallel computation. Although some parallel machines like MPP [10], and Connection Machine are available commercially, they have vastly different architectures which play a crucial role in algorithm design. As mentioned earlier, the RAM model serves as a reasonable abstraction for almost all uniprocessor computers available today. A substantial amount of research has focussed on developing algorithms which are tailor-made for a specific architecture like the 2-D Mesh, Butterfly, or the Hypercube interconnection networks. These algorithms are specialized to an extent that they are not as efficient when implemented on a different architecture from the one that it was designed for. Notwithstanding the contributions of these algorithms, it should be clear that there is a need for more generality in parallel algorithm research and also from a theoretical perspective, a need to decrease the over-dependence of the algorithms on the architecture. It is with these objectives in mind that a major amount of the present research in the theory of parallel algorithms has been developed around the Parallel Random Access Memory (P-RAM) model. This model is a very natural extension of the RAM model (although not as realistic).

1.3.1 The PRAM model

In this model, there are P identical processors which execute instructions synchronously (with a global clock) in parallel on different data values. Each

processor is identified using a unique integral label which is also called the processor-id. The processors have access to a common shared memory. In one time step, a processor is capable of accessing a memory (shared or local) location, or executing a logical or an arithmetic operation on $O(\log n)$ bit words. If the number of processors is one, then this is the usual RAM model. There are variations within this basic model depending on resolution schemes for memory access conflicts. In the most restrictive model, the Exclusive-Read-Exclusive-Write (EREW) no two processors are allowed to read or write simultaneously in the same memory location. In the Concurrent-Read-Exclusive-Write model, no two processors can write to the same memory location whereas any number of processors are allowed to read from a memory location. Note that these constraints have to be guaranteed by the algorithm-designer. In the strongest model, Concurrent-Read-Concurrent-Write model (CRCW) any number of processors are allowed to read or write simultaneously to the same memory location. While reading is not a destructive operation, in concurrent-write models, we have to further specify the result in the memory location when more than one processor attempts to write to the same location. In the *arbitrary* CRCW model, we assume that one of the processors succeed in writing its value but we do not make any assumption regarding which processor succeeds. Clearly, if all the processors try writing the same value, it is simpler than the case where the processors are trying to write different values. It is the algorithm-designer's responsibility to ensure that the correctness of the algorithm does not depend on which value gets written. An example of this kind of algorithm is computing 'OR' of n bits using n processors, Each processor has one bit and if it is a '1' it tries writing a '1' into a specified register which has been initialized to zero. Clearly the algorithm correctly computes

the 'OR' since even when more than one processor tries to write a '1', the result is '1' irrespective of which processor succeeds.

A more powerful model is the *Priority* CRCW model where the processor with the largest label succeeds during concurrent writes. Model A is said to be more powerful than model B (denoted by $A \geq B$), if an algorithm for model B can be executed in the same time bound in model A using the same number of processors. Clearly $CRCW(priority) \geq CRCW(arbitrary) \geq CREW \geq EREW$. If there exists a problem for which model A exhibits lower asymptotic time-complexity than the lower bound in model B then model A is *strictly* more powerful than model B. The previous hierarchy is known to hold with strict inequality. In this context it is also important to investigate a measure of how much more powerful model A is in comparison to model B. It is often possible to be able to make formal statements like - 'A single step in model A can be executed in at most g(n) steps in model B for any problem of size n'. For example, it is known that a single step of CRCW PRAM model can be executed in $O(\log n)$ steps in an EREW model. For a more detailed discussion of relationship between various PRAM models the reader is encouraged to read [39].

In this thesis, we shall be mostly using the CREW model. In addition we shall assume that a single memory location can hold a real-number (of infinite precision) and a processor can perform an arithmetic operation involving two real-numbers in a single step. This is consistent with the model that is used for sequential computational geometry called the 'Real'-RAM. In this model a register (or a memory location) can store a real number and the arithmetic operations can be computed to arbitrary precision. Although it is somewhat unrealistic, this is unavoidable for solving problems in R^d Euclidean space.

Since we shall be using randomization, we also assume that processors have access to random bits (upto $O(\log n)$ bits) in constant time.

For a practical-minded person, the PRAM model may appear to be far from 'reality' (which even a theoretician also acknowledges to have serious drawbacks). Especially, the problems regarding unbounded fan-in (in CREW and CRCW models) and the near-impossibility of synchronization of a large number of processors in O(1) time are inconsistent with the physics of a real machine. These issues and others about 'what constitutes the right parallel model' have been a subject of continuing debate in the research community and some stimulating discussions can be found in the NSF-IBM workshop proceedings [38]. In defense of the present work, we shall only note that a significant amount of research has been devoted to emulating PRAM models on interconnection networks ([49, 65, 75]) and any improvement in the PRAM algorithms implies a corresponding improvement for the algorithms in the network model.

1.3.2 The class $\mathcal{NC}$ versus optimal parallel algorithms

Parallelism seems to be an intrinsic property of a problem; some problems admit a greater degree of parallelism than others. By degree of parallelism we imply the speed up (over its sequential counter-part) obtained by using P processors. The class $\mathcal{NC}$ refers to problems for which we can obtain a poly-logarithmic running time algorithm using a polynomial number of processors. The class $\mathcal{RNC}$ includes problems for which we can obtain an algorithm with the previous property allowing randomization. While certain problems like sorting can be easily shown to be in $\mathcal{NC}$, problems like Depth-First Search on general graphs are not known to be in $\mathcal{NC}$ (however it is known to be in $\mathcal{RNC}$ [3]). It is an important theoretical question if the class $\mathcal{P}$ is the same as $\mathcal{NC}$.

Clearly, using huge number of processors (i.e. high degree polynomial) for a problem may make it very wasteful in the amount of resources used. A very useful metric in this context is the 'processor-time' (PT) product. Ideally we would like $P_nT_n = Seq(n)$ where P_n and T_n represent the number of processors and time used for solving a problem of size n. In future we shall use PT without the subscript. For problems in $\mathcal{NC}$, $PT = O(n^k Seq(n))$. A major portion of research has been involved with deriving $\mathcal{NC}$ algorithms for various problems without concern for the speed-up (to which Richard Karp had once alluded to be a 'pernicious' influence of this class of algorithms). A commonly agreed-upon definition of 'efficient' algorithms are those for which $PT = O(Seq(n)\log^k n)$ i.e. the number of parallel operations is within a poly-log factor of the best known sequential algorithm. A parallel algorithm is called 'optimal' if $PT = O(Seq(n))$.

In this dissertation, our goal has been to derive 'optimal' algorithms rather than simply $\mathcal{NC}$ algorithms. The typical sequential complexity of the problems tackled is $\Theta(n \log n)$ and in most cases we have been able to obtain an $O(\log n)$ time optimal algorithm. Although even n processors appear too large in most situations, it implies that the problems have a linear speed-up (linear in the number of processors) when the number of processors is less than n. The following well known result justifies this observation:

Lemma 1.2 (slow-down lemma) *If there exists an algorithm using P_1 processors and running in T_1 time then it runs in $O(T_1P_1/P_2)$ time using P_2 processors for $P_2 \leq P_1$.*

The proof follows from the observation that each of the P_2 processors can emulate $\lceil P_1/P_2 \rceil$ processors of the first algorithm. This implies that the speed-

up obtained does not decrease by reducing the number of processors (it may in fact increase) and so if we can get a linear speed up with n processors, we can get a linear speed-up with $p < n$ processors. By studying the relationship between speed up and number of processors we can determine how profitable it is to employ more processors for a problem of a certain size.

1.4 Problems, results and previous work

Computational geometry, as it stands today, is concerned with the design and analysis of geometric algorithms. Although the earlier work in this area was more algorithmic, there is growing interaction combinatorial geometry, a related area. In these algorithms, the geometric properties of the problem restricts the solution space (from the naive brute-force approach) which make these algorithms more efficient. For an excellent introduction to the field the reader is referred to a text by Preparata and Shamos [60]. For a more combinatorial approach, a text book by Edelsbrunner [37] provides a very comprehensive treatment.

1.4.1 Problems considered

Convex hulls

Convex polygons are very important objects in computational geometry, and in a large number of cases give rise to very efficient algorithms because of their nice property, namely convexity. Convex hull of a set of points in E^d (the Euclidean d-dimensional space) is the smallest polygon containing these points. The input is given as a set of n d-tuples and the output consists of an 'ordered' set of points of the convex hulls. In two-dimensions, this 'ordering' is simply

the clockwise (or counter-clockwise) ordering of the vertices of convex hulls. In three-dimensions it can be the cyclic ordering of edges of the convex hull around each vertex. We restrict ourselves to the two and three dimensional case. It is known that sequentially it takes $\Omega(n \log n)$ operations to construct the convex hull in two dimensions and there exist algorithms with matching upper-bound.

A useful transform

The convex-hull problem has a very interesting dual problem, namely the intersection of half-spaces. This dual transformation $\mathcal{D}$ maps a point in E^d to a non-vertical hyperplane in E^d and vice-versa. Let $p = (\pi_1, \pi_2, ..., \pi_d)$ be a point in E^d. Then $\mathcal{D}(p)$ is the hyperplane $1 = \pi_1 x_1 + \pi_2 x_2 + ...\pi_d x_d$ and vice-versa such that a hyperplane h not containing the origin is mapped to a point p for which $\mathcal{D}(\mathcal{D}(p)) = h$. The following properties can be verified [37]

(i) *Incidence preservation* : point p belongs to hyperplane h if and only if $\mathcal{D}(h)$ belongs to $\mathcal{D}(p)$.
(ii) *Order preservation*: point p lies in half-space h^{pos} (h^{neg}) if and only if point $\mathcal{D}(h)$ lies in $\mathcal{D}(p)^{pos}$ ($\mathcal{D}(p)^{neg}$).

The transform $\mathcal{D}$ is extended to sets of points (hyperplanes) in a natural way. Let $\mathcal{P}$ be a convex polytope with non-empty interior $int\mathcal{P}$ and assume that the origin O is contained in $\mathcal{P}$. Then $\mathcal{D}(\mathcal{P})$ is an infinite set of hyperplanes that avoid some convex region around O. The dual of $\mathcal{P}$ is defined as

$$\bar{\mathcal{P}} = closure(\bigcap_{h \in \mathcal{D}(\mathcal{P})} h^{pos})$$

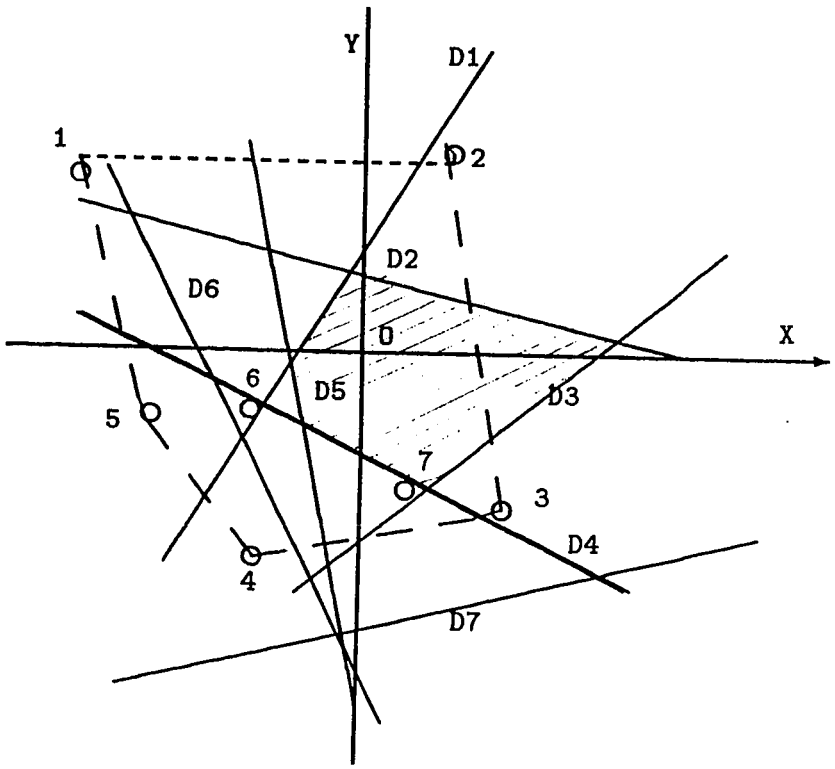

Figure 1.1: The dual transform on plane. For example, the point labeled 1 and the line D1 are duals of each other. The shaded portion represents the dual of the convex hull bounded by 1,2,3,4,5,6,7. Notice that the duals of the vertices of the convex hull are the facets of the dual hull.

The following claim follows from the previous observations:

Claim 1.1 *Point p belongs to the int$\mathcal{P}$, boundary$\mathcal{P}$ or complement$\mathcal{P}$ if and only if the hyperplane $\mathcal{D}(p)$ avoids $\bar{\mathcal{P}}$, avoids int$\bar{\mathcal{P}}$ but not $\bar{\mathcal{P}}$, or intersects int$\bar{\mathcal{P}}$ respectively.*

In other words the vertices of the convex hull are the dual transforms of the facets of the dual hull. See Figure 1 for an example.

This property has been exploited very often so that the same algorithm can

be used for both convex-hulls and intersection of half-spaces (if we know an interior point).

Voronoi Diagrams

A very general definition of Voronoi diagram given by Edelsbrunner [37] is as follows:

> Let S be a finite set of subsets of E^d and for each $s \in S$ let d_s be a mapping of E^d to positive real numbers; we call $d_s(p)$ the distance function of s. The set $\{p \in E^d\colon\ d_s(p) < d_t(p),\ t \in S - \{s\}\}$ is the Voronoi cell of s and the cell complex defined by the *Voronoi cells* of all subsets in S is called the *Voronoi diagram* of S.

In this thesis, we confine ourselves to the case where S is a set of points in E^2 and the distance function is the L_2 metric. In mathematical literature, Voronoi diagrams appeared as early as in 1850 (due to Dirichlet) and again in 1907 due to Voronoi. Problems about packing and covering of space by balls and other convex figures were among the first major applications of such diagrams. Shamos and Hoey [72] introduced Voronoi diagrams to computer science and since then a considerable amount of research has been devoted for deriving efficient sequential algorithms for the 2-D Voronoi diagram problem ([40, 69, 21]).

The Voronoi diagram is a very versatile tool for obtaining efficient solutions of some important proximity problems and is also a fundamental mathematical object in its own right. A large number of the problems can be solved in linear or $O(n \log n)$ time from the information contained in the Voronoi diagram that

includes *all-points nearest neighbor, Euclidean minimal spanning tree, diameter, smallest enclosing circle* among others.

Closest Pair: For a set of points in the plane, determine a pair of points such that the Euclidean distance between them is smaller than any other pair of points.

All Nearest Neighbor: For each of the given points in the plane determine the point that is closest to it.

Largest empty circle: Given a set of points, find the largest circle that contains no point in its interior and its center lies inside the convex hull of the points.

Euclidean minimal spanning tree Given a set of points, find a minimal spanning tree where the edge weights are proportional to the Euclidean distance between the points.

There are known reductions of Voronoi-diagram (in plane) to convex-hull in three dimensions due to Brown [14]. In the thesis we actually describe an algorithm for computing the 3-D convex hull.

Triangulation and Visibility

Given a simple polygon (i.e. one without self-intersecting edges) with n vertices, a triangulation involves adding $n - 3$ edges such that each face is a triangle. An equivalent problem is that of determining *visibility* where for each vertex of the polygon we have to determine the edges (0, 1 or 2) that are the first

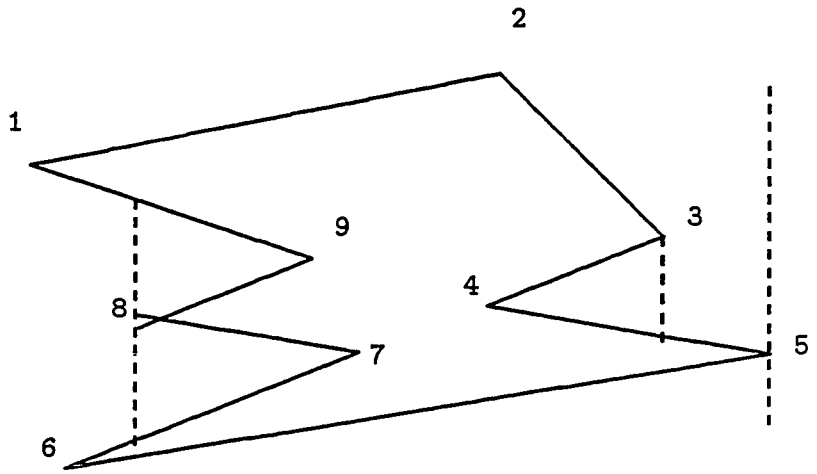

Figure 1.2: Vertex 8 has (1,9) and (6,7) as visible edges. Vertex 3 has (4,5) visible while vertex 5 does not have any visible edge

edges (both above and below the vertex) that intersect the vertical line drawn through this vertex. Figure 1.2 illustrates a typical case. The best known sequential algorithm for triangulation runs in $O(n \log\log n)$ time ([76]). Using randomization, this has been improved to $O(n \log^* n)$ by Clarkson, Tarjan and VanWyk [48].

A related problem is **trapezoidal decomposition** where given n non-intersecting line segments (not necessarily forming a simple polygon), we have to determine the visible edges for each query point. This problem has a lower bound of $\Omega(n \log n)$ because the *two-dimensional dominance* problem can be reduced to it (which is known to have a lower bound of $\Omega(n \log n)$).

Dominance problem in d-dimensions: Given a set of n points in E^d, a point p is a *maxima* of the set if there exists no other point q in the set such that all the coordinates of q are larger than the corresponding coordinates of

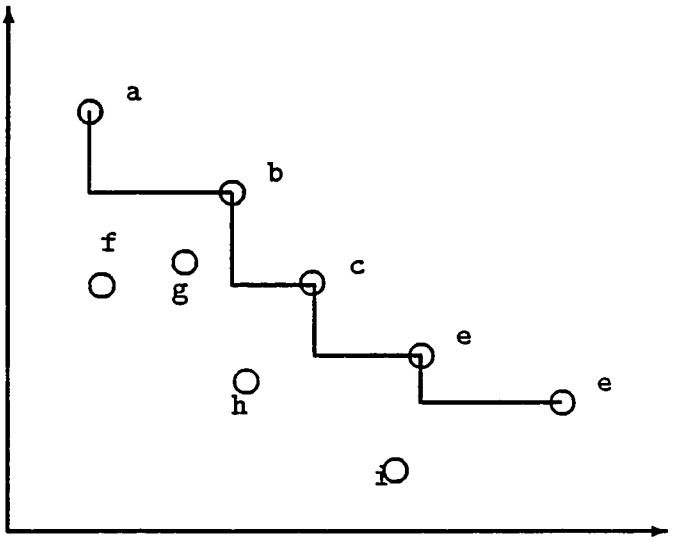

Figure 1.3: The points a, b, c, d and e are maximal points

p. The dominance problem involves identifying all the maximal elements of a given set. See Figure 1.3.

Of particular interest to us are the two and three dimensional cases. Trapezoidal decomposition is used to solve the triangulation and visibility problems. Note that this is not an optimal reduction since the sequential complexity of triangulation is better than that of trapezoidal decomposition.

Point location on plane

Given a planar sub-division, we are allowed to pre-process it such that given any query point, we have to determine quickly (typically in $O(\log n)$ sequential time) the subdivision that the point belongs to.

If the sub-division does not have holes, the preprocessing time is dominated by the time to triangulate the sub-division. One of most elegant solutions was given by Kirkpatrick, who decomposed the sub-divisions into a hierarchy of planar maps of decreasing sizes. This has recently been used for determining intersections of polyhedras in three dimensions.

Linear Programming

In its most general form we have to minimize a cost function

$$c \cdot x$$

subject to

$$Ax \geq b$$

where A is a $n \times d$ matrix and c is a d-dimensional vector. This is a classic problem which has far-reaching applications in various combinatorial optimization problems. We consider a restricted version of the problem, namely in two-dimensions. Meggido [55] and Dyer [34] had independently shown that for a fixed dimension, this problem can be solved in time linear in the number of constraints. We present an optimal $O(\log n)$ time parallel algorithm for this problem and in the process we also derive a constant time (randomized) parallel algorithm for finding an *approximate-median.*

The problem of finding an *approximate median* is defined as following: Given n elements from a totally ordered set we are required to choose an element whose rank is γn for some $0 < \gamma < 1$.

We feel that this algorithm may have other interesting applications.

1.4.2 Previous work in parallel computational geometry

Designing efficient parallel algorithms for various fundamental problems in computational geometry has received much attention in the last few years. After some early work by Chow [18] in her thesis, Aggarwal et al. [4] developed some general techniques for designing efficient parallel algorithms for fundamental geometric problems. Most of the problems tackled in that paper had $\Theta(n \log n)$ sequential complexity and the authors presented parallel algorithms which used a linear number of processors and ran in $O(\log^k n)$ time (k being typically 2,3 or 4) where n is the size of the input. These problems included 2-D convex hulls, 2-D Voronoi diagram, 3-D convex hulls and triangulation among others. Consequently, none of their algorithms except the one for 2-D convex hulls were optimal in $P \cdot T$ bounds. A number of the problems in the original list (in [4]) have now been successfully resolved as far as O(logn) time, n processors algorithms are concerned in a paper by Atallah, Cole and Goodrich [7]. They extended the techniques used by Cole [23] for his parallel mergesort algorithm and used a data-structure called *plane-sweep tree* (first proposed by Aggarwal et al. [4]) to arrive at the optimal algorithms. In an independent development, around the same time , Reif and Sen [67] obtained similar results using randomized techniques.

A more comprehensive discussion of the problems tackled in the paper by Atallah, Cole and Goodrich can be found in the doctoral thesis of Goodrich [43]. They gave $O(\log n)$ time optimal parallel algorithms for 3-D dominance, trapezoidal decomposition, range-counting, intersection detection of line segments, and an optimal $O(\log n)$ time algorithm for constructing a data-structure for

fractional cascading (a data-structure technique originally proposed by Chazelle and Guibas to speed-up searching the same key in different catalogues). Cole and Goodrich [25] provided further applications of their cascaded-merging technique to obtain optimal $O(\log n)$ time algorithm for the all-nearest neighbor problem of a point set, and also when these points form the vertices of a convex polygon.

In his thesis Chandran [15] was able to apply Cole's mergesort techniques to give $O(\log n)$ optimal algorithms for several rectangle problems like computing the area and perimeter of the union of n rectangles. He was also able to extend Cole's methods to derive optimal algorithms for constructing the lower-envelop of n line segments in the plane. In Chandran and Mount [58] an $O(\log \log n)$ time algorithm for finding a minimum-area circumscribing triangle has been presented. Convex hulls in three dimensions can also be constructed sequentially in $\Theta(n \log n)$ time where as the best known deterministic parallel algorithm due to Dadoun and Kirkpatrick [30] runs in $O(\log^2 n \log^* n)$ time using n processors.

Perhaps the two most important problems which have proved to be difficult to optimize (in parallel) are the 2-D Voronoi diagram problem and the convex hull of points in 3-space. These are very fundamental problems in computational geometry, and optimal algorithms for these problems would imply corresponding optimal solutions for a multitude of other problems. In this thesis we settle this question by presenting a randomized algorithm for these problems that run in O(logn) time and use n processors in a shared memory model of parallel computation. The reader should note that the lower-bound of $\Omega(n \log n)$ also applies to the randomized algorithms by a reduction of sorting to (1-dimensional) Voronoi-diagrams. Levcopoulos, Katajainen and Lin-

gas [53] presented an optimal expected time algorithm for Voronoi diagrams for a randomly chosen set of input points; in contrast our algorithm makes no assumption about the input distribution and is optimal for the worst-case input. Dyer [36] gave efficient sequential algorithms for convex hulls and voronoi diagrams for points when the distribution of input points is known.

1.4.3 Random-sampling in computational geometry

Randomization has been successfully used in a wide number of applications (for example see [42, 68, 78]) and has recently been used to obtain efficient algorithms in computational geometry. Clarkson [19, 20, 21], Haussler and Welzl [46], and Mulmuley [59] used random sampling techniques to derive better upper-bounds for a large number of problems including the post-office problem, higher-order Voronoi diagrams, segment intersections, linear programming, and higher-dimensional convex hulls. The general approach taken by these algorithms is as follows: a randomly chosen subset R of the input set S is used to partition the problem into smaller ones. Clarkson [21] proved that for a wide class of problems in computational geometry, the *expected* size of each subproblem is $O(|S|/|R|)$ and moreover the *expected* total size of the subproblems is $O(|S|)$. A random subset R which satisfies these conditions for fixed constant multiples is called a 'good' sample and is called 'bad' otherwise. Clarkson's results show that by using a straight-forward random sampling technique any randomly chosen subset is good with constant probability; implying that it can also be 'bad' with constant probability. Consequently, his methods yielded *expected* resource bounds but cannot be used to obtain high-likelihood bounds (i.e. bounds that hold with probability greater than 1 - $1/n^{\alpha}$ for any $\alpha > 0$). This makes it very difficult to extend his methods in the context of

parallel algorithms due to the recursive nature of the algorithms. In particular, the *expected* bounds at each recursive call are not strong enough to bound the resources used by the entire parallel algorithm due to the following reason. In a sequential algorithm, due to the linearity property of expectation (i.e. the expectation of the sum is the sum of expectations), it suffices to bound the expected time required by individual steps. The total expected time of the sequential algorithm is the sum of expected time of the individual steps. In contrast, consider the recursive parallel algorithm as a tree where a node corresponds to a procedure and the children of a node corresponds to the parallel recursive calls made by the procedure. The time required at each level of this tree is the *maximum* of the time required by any node of that level. There is no known method to bound the maximum of the expectations without using higher moments. The total time required by the parallel algorithm is the time when all the procedures corresponding to the leaf nodes are completed. Typically, in a parallel algorithm, the number of leaves in the corresponding process tree is at least n^{ϵ} $(0 < \epsilon < 1)$. Even if we succeed in bounding the expected time for completion of a leaf-node procedure, the expected bounds are too weak to bound the *maximum* of the time required by all such processes.

One of the main contributions of our work is a sampling method for obtaining 'good' sample with high probability. We introduce this technique called *polling* in Chapter 3 to obtain a 'good' sample with high probability with relatively small overhead. Polling appears to be a general tool for obtaining improved parallel randomized algorithms. A similar idea had been used previously by Dyer and Frieze [35] to improve the efficiency of a linear-programming algorithm.

1.5 Some fundamental results for PRAM algorithms

In this section we review some known results (without proof) for some fundamental problems and some commonly used techniques in parallel algorithms. These will be used very frequently in the main part of the thesis where we shall simply appeal to the results of this section.

1.5.1 Parallel Prefix computations

Given n elements a_i, $1 \leq i \leq n$ from a semigroup where $\oplus$ is the semigroup operation, we are required to compute all the partial sums of the form $S_k = \oplus_{i=1}^{k} a_i$ for $1 \leq k \leq n$. Since these partial sums can be computed in $O(n)$ time sequentially using a straight-forward approach, an optimal PT product is $O(n)$. Although an $O(\log n)$ time algorithm is not difficult to derive using a binary-tree, Cole and Vishkin [26] obtained the following result:

Lemma 1.3 *In a CRCW (arbitrary) PRAM model, the prefix sum of n elements can be computed optimally in* $O(\log n / \log\log n)$ *time.*

The significance of this result is that it matches a lower-bound for such a problem in the PRAM model due to Beame and Hastad [11]. They prove that in a CRCW PRAM model, using a polynomial number of processors, the parity of n bits takes $\Omega(\log n / \log\log n)$ time. Moreover, Cole and Vishkin's result has very elegant applications to *load-balancing* problems which we shall discuss in the latter part of this section.

1.5.2 List Ranking

A related problem to prefix-computation is that of computing the prefix sums when the input is given as a linked-list rather than an array. This linked-list of n elements is a linear chain such that every element has in-degree and out-degree identically 1 (except for the first and the last elements). A special case is that of determining for every element in the list its distance from the head (or tail) of the list. Once the position of an element in the list is known, the algorithm for prefix sum can be applied to the linked-list prefix sums. Although an $O(\log n)$ time algorithm due to Wyllie [80] was known for a while, it used n processors and was not optimal. The following result was obtained independently by Anderson and Miller [6] , Han [45] and Cole and Vishkin [23].

Lemma 1.4 *The list ranking problem can be solved in an EREW model optimally in $O(\log n)$ time.*

1.5.3 Merging, Selection and Sorting

Sorting is certainly the most frequently called-upon subroutine in most applications. It therefore doesn't come as a surprise that a great amount of research in parallel algorithms has been devoted to obtaining optimal algorithms for sorting. In the context of PRAM algorithms, the parallel merge-sort algorithm due to Cole [23] is perhaps the most elegant and has small constants. It has also been used successfully for other algorithms (like the Cascaded merging techniques in computational geometry referred to in the previous section). The drawback however is that it is not amenable for network models. There are other known $O(\log n)$ time optimal sorting algorithms, like the AKS-network [1], Reischuk [70] and Flashsort [68]. Both Reischuk's

algorithm and Flashsort use randomization.

Lemma 1.5 *: In an EREW PRAM model, n keys from a totally ordered set can be sorted in $O(\log n)$ time using n processors.*

In our algorithms, whenever we refer to sorting we can use any of the above mentioned sorting algorithms without changing the running time by more than a constant factor.

Valiant [77] had given one of the first parallel merging algorithms in the parallel context which was later adapted into the standard PRAM models due to work by Borodin and Hopcroft [12] and Shiloach and Vishkin [73].

Lemma 1.6 *In a CREW PRAM model, two sorted lists of n and m elements can be merged optimally in $O(\log(n+m))$ time. Moreover they can be merged in $O(\log\log(m+n))$ time using $n+m$ processors.*

The lower-bound result of [11] implies that given n elements we cannot select the kth element for all $1 \leq k \leq n$ in $o(\log n/\log\log n)$ time. However, we can do much better for selecting the minimum and the maximum element.

Lemma 1.7 *The maximum (minimum) element of a set of n elements can be identified in $O(\log\log n)$ time using $n/\log\log n$ processors.*

1.5.4 Load-balancing and processor allocation

For a number of problems, it has been observed that the processor complexity can be reduced by a careful scheduling of processors. For example, consider the problem of computing the sum of n elements. A naive algorithm would start with $n/2$ processors and let the processors add the numbers in a pairwise

fashion. By letting this pattern repeat over $O(\log n)$ stages, the computation is similar to that of a binary tree. However the processor time product is $O(n \log n)$ which exceeds the sequential complexity by a factor of $O(\log n)$. This can be rectified by a simple modification where we start with $n/\log n$ processors and let each of them compute the sum of $O(\log n)$ elements sequentially and then proceed with the previous algorithm. Although this adds a factor of $O(\log n)$ the PT product is asymptotically optimal. The following result due to Brent [13] makes this observation into a formal statement:

Lemma 1.8 (Brent) *Any synchronous parallel algorithm needing time T and a total of R elementary operations can be executed on p processors in time $\lfloor R/p \rfloor + T$.*

Proof: Let R_i denote the number of operations to be executed in the ith step of the algorithm (simultaneously), they can be computed in time $\lceil R_i/p \rceil$ with p processors. The total running time is therefore

$$\sum_{i}^{T} \lceil R_i/p \rceil \leq \sum_{i}^{T} (\lfloor R_i/p \rfloor + 1) \leq \lfloor R/p \rfloor + T$$

This observation is not as useful as it may seem at first sight because it doesn't take into account the time required to allocate processors to the tasks at every step i. By letting each processor *simulate* a **fixed** set of the processors of the previous algorithm we may obtain a more efficient algorithm. Notice that the **slow down lemma** in section 1.3 is a special case of this lemma. In our example of the summation of n elements, we can view it as the following: We let each processor *simulate* the operations carried out by $O(\log n)$ processors of the naive algorithm. This task scheduling is possible since the execution

profile can be determined in advance. By the execution profile we mean the task carried out by every processor at any time step.

In other cases, the scenario is much more unpredictable. Even though we can determine that the total number of operations is R, by simulating a fixed set of processors by one processor, we may not be distributing the load evenly among all processors. Brent's theorem assumes that we can distribute the tasks almost *evenly* among all processors to achieve the optimal running time.

We shall now consider a special case where the parallel algorithm has the following property: The algorithm has $O(\log n)$ stages such that the total number of operations is $O(n)$. Using n processors, the algorithm executes in $O(\log n)$ time and at stage i only $n/2^i$ processors are active (but we do not know in advance which processor is active in a particular stage). A processor that ceases to be active at any stage does not participate during the subsequent stages of the algorithm. We make the following claim:

Claim 1.2 *The algorithm can be made to run in $O(\log n)$ time using only $O(n/\log n)$ processors in a CRCW PRAM model.*

This scheme was proposed by Cole and Vishkin [23] based on their sub-logarithmic time algorithm for prefix-sum. After $O(\log \log n)$ stages only $(n/\log n)$ processors are required which are available. From the slow-down lemma, we know that for $p \leq (n \log \log n/\log n)$ we can compute the prefix sum in optimal time. In stage i of the algorithm ($i \leq \log \log n$), we let each new processor simulate $\log n/2^i$ active processors. To distribute the load uniformly at stage i+1, we first associate a tag '0' with a processor that is not going to be active in future and a tag '1' otherwise. We can find the k-th active processor $k \leq n/2^{i+1}$ by a prefix sum. Then we let a new processor with processor-id

j simulate a block of $\log n/2^{i+1}$ processors that are active in the $(i+1)$-th stage. The additional time needed for re-distribution of processors over all the $O(\log\log n)$ levels using this scheme is:

$$\sum_{i=0}^{\log\log n} O(\frac{\log n}{2^i}) = O(\log n)$$

The claim holds as long as the problem size decreases by a constant factor (not necessarily 2) and proof follows on same lines. Miller and Reif [56] also describe a scheme for load-balancing using random permutations which succeeds with high probability.

1.6 Organization of this thesis

In Chapter 2, we shall describe optimal $O(\log n)$ time algorithms for planar-point location and 2-D linear programming. These are obtained using known randomized techniques and the resulting algorithms themselves are extremely simple. In Chapter 3, we review some of the known methods used for sorting and show that a straight-forward extension to a related problem in computational geometry gives rise to additional complications. We discuss a method called 'Polling' to fix this problem, and outline a very general method for divide-and-conquer for problems in computational geometry. In Chapter 4, we apply the techniques developed in Chapter 3 to derive optimal algorithms for various problems including trapezoidal decomposition and 3-D convex hulls. In Chapter 5 we focus on the issue of minimizing the use of random bits and suggest use of alternate random number generators. We conclude with a discussion on directions for future research.

Chapter 2

OPTIMAL PARALLEL AlGORITHMS USING SIMPLE RANDOMIZED TECHNIQUES

In this chapter we illustrate the usefulness of random sampling methods by presenting algorithms for two classical problems in computational geometry, namely planar-point location and 2-D linear programming. These methods have been applied previously to combinatorial problems like Selection and Graph Connectivity. Not only do these techniques yield optimal speed-up, the algorithms themselves are extremely simple. Often it has been observed that parallel algorithms assume very complicated forms when one attempts to achieve optimal speed-up, i.e. proportional to the number of processors. The algorithms presented in this chapter are very similar in structure to their sequential counterparts.

The main tools used here are: Symmetry breaking using Random-mate and probabilistic inequalities to bound the tail-probabilities of Binomial distribution. Symmetry breaking was introduced by Reif [66], and was later used successfully by Luby [54] for a parallel maximal-independent set problem and

by Gazit [42] to obtain an optimal parallel algorithm for graph connectivity. It was also used to obtain an optimal algorithm for list-ranking. The bounds for Binomial distribution have been used in the following context; the probability of a binomial random variable deviating from its mean is very low. Very often this random variable is a measure of the running time of the algorithm, which is then used to make statements of the following nature: the probability that the running time exceeds a certain bound asymptotically goes to zero (as the problem size grows larger).

Below we present an algorithm for building a data-structure for planar-point location and an algorithm for linear programming on the plane. Both of these algorithms run in $\tilde{O}(\log n)$ time and achieve optimal processor bounds i.e. $O(n/\log n)$. Although there exist other algorithms for point-location ([7] and [27]) which achieve $O(\log n)$ parallel running time, our algorithm has an added advantage of building a data-structure (namely, Kirkpatrick's hierarchical representation) which has applications to intersection problems in three dimensions. This property is actually used in a later chapter when we design more sophisticated algorithms for convex hull in three-dimensions.

The algorithm for two-dimensional linear programming revolves around a procedure for selecting an approximate median in constant time. The median-find algorithm could be of independent interest in itself and it was used recently for a string matching algorithm (Vishkin [79]).

2.1 Point Location In Planar Subdivisions In Logarithmic Time With High Probability

A PSLG (planar straight line graph) is a connected graph which can be embedded on a plane using only straight line edges. Given a PSLG and a query point, the point location problem is to identify the subdivision in the plane (induced by the PSLG) which contains the query point. If a PSLG has n vertices, in the sequential (uniprocessor) case, the asymptotic optimal query time performance of $O(\log n)$ can be achieved by a variety of approaches (Kirkpatrick [52], Guibas et al. [44] among others). The performance is achieved by binary search on a well organized data structure constructed on the PSLG during the preprocessing. The preprocessing cost for the best known sequential algorithms for point location is $\Omega(n \log n)$ operations, and thus the best parallel time for this problem cannot be better than $O(\log n)$ time using n processors. We propose a randomized method by which the preprocessing can be done in $O(\log n)$ time with very high probability using $O(n)$ processors given a PSLG which has only convex subdivisions. A similar algorithm has also been independently proposed by Dadoun and Kirkpatrick [30].

2.1.1 The sequential algorithm

Kirkpatrick [52] proposed a triangulation refinement technique which builds a hierarchy of triangulated PSLG from the initial graph. Unfortunately, his method does not appear to be directly parallelizable. A review of his method will help the reader to understand the parallel version presented here. Starting with a triangulated PSLG (i.e. all faces including the exterior face are triangular), his algorithm removes an independent set of vertices, and retriangulates

the remaining graph. In addition, it keeps track of the set of the eliminated triangles that have non-empty intersection with each of the new triangles of the retriangulated graph. This procedure is repeated successively until we are left with a constant number of triangles. Actually we can stop when the problem size is reduced to $O(\log n)$. The search proceeds in a hierarchical manner from the highest level, where the problem can now be solved in constant time ($O(\log n)$ time if we stop at $O(\log n)$ size graph). Subsequently, at each level we relocate the point with respect to the triangles which intersect the triangle in the current level, thus refining the regions of location at each level. At the lowest level the point is located with respect to the regions of the original PSLG, and the search is complete. The search data structure is a directed acyclic graph (not a tree). Each of the nodes represent a triangular region, and is connected by directed arcs to all the triangular regions that it intersects in the previous level. At any node, the algorithm determines which of its children the point lies in. Notice that a node can have more than one parent and hence the graph (the underlying search structure) is not a tree. The efficiency of this approach depends on the number of levels of the triangulated PSLG and the number of intersections of each triangle with the triangles in the next lower level. By guaranteeing that a constant fraction of the vertices, and hence the triangles, (since it is a planar graph) are eliminated at each successive level a logarithmic level search is achieved. However in doing so, one must also be careful that a triangle intersects a maximum of a constant number of triangles in the next level (i.e. each node has a bounded constant degree so that a constant time is spent at each search level). Both of the above criteria can be satisfied by identifying an independent set of vertices each having a degree $\leq d$, where d is a fixed constant. The number edges in a planar triangulated graph (V, E) is

$3|V|$- 6 from Euler's formula (assuming that the exterior face is also a triangle). This adds up to a total vertex degree of $6|V|$-12, giving us a lower bound on the number of vertices with degree less than d. Thus the number of vertices with degree less than d is at least $6|V|/d$ - 2 in a triangulated graph for $d > 6$, (a typical value of d is 12). Since a constant fraction of these vertices can be eliminated at each stage by identifying a maximal independent set of vertices, the height of the search tree is at most $O(\log |V|)$. The preprocessing time is dominated by the identification of this independent set of degree $\leq d$, which costs linear in size of the PSLG at each level resulting in total time of $O(n)$. If the initial PSLG is not triangulated, the cost of triangulation dominates the pre-processing time which can be as high as $\Omega(n \log n)$ for subdivisions that have holes.

In the discussion below we present a randomized method to identify a large independent set in constant time for each level using n processors with a high probability. As a result, the search structure can be constructed in $O(\log n)$ time.

2.1.2 Choosing a large independent set with high probability

Associate a processor with each vertex and each edge of the PSLG (V, E). From Euler's formula we need only $O(|V|)$ processors for the PSLG. An independent set consisting of vertices of degrees less than or equal to d can be identified in the following manner. [1]

Algorithm Random-mate

[1] this is the random-mate technique mentioned earlier

Input: A PSLG in form of a doubly connected edge list (DCEL).

Output: A large independent set of vertices with degree $\leq d$.

(1) Identify the set of vertices with degree $\leq d$. This can be done in constant time using one processor for each vertex. There will be at least $6|V|/d$ (ignoring the small constant) such vertices from the previous discussion.

(2) This has two phases :

(2a) Randomly assign a tag 'male' or 'female' with equal probability to each selected vertex from step (1). The 'males' constitute the candidates for an independent set. For each edge between two 'males' pronounce both vertices 'dead' by marking their corresponding cells. This is easily done by using one processor for every edge with concurrent write capability. Note that it is very easy to get rid of the concurrent-read feature since a vertex has at most degree d and hence this step can be carried out in constant number of steps (instead of exactly one step).
(2b) The 'males' which are not 'dead' form an independent set.

(3) Output the set X of 'males' which are not 'dead'. The set X is an independent set of vertices having degree less than d.

It is obvious that the algorithm gives us an independent set which is possibly smaller than the maximal independent set (a null set in the worst case for a chain of males). However, we shall show that on the average this technique will produce an independent set proportional to a constant fraction of the total number of nodes (vertices of degree $\leq d$) with a very high probability.

Lemma 2.1 *Let $n = |V|$ and n_d denote the number of vertices with degree less than or equal to d ($n_d \geq 6n/d$ from [52]). There exist constants c, ν ($0 < \nu < 1$), such that $P(|X| < \nu n) \leq e^{-cn}$.*

Proof: Assuming equal probabilities for a vertex being assigned a 'male' and a 'female' tag, the probability of a vertex being in the independent set is greater than or equal to $\left(\frac{1}{2}\right)^{d+1}$ (since it has $\leq d+1$ neighbors). We consider an independent set $I_3(n_d)$ of vertices which are at a distance 3 (i.e. the shortest distance between any two vertex in this set is three edges). Since the graph has a bounded degree d, the selection of each vertex can prevent the selection of at most $d_{max} = (d-1)^3 + (d-1)^2 + (d-1)$ other vertices. Thus the cardinality of this set is at least $(1/d_{max})6n/d$ or $6n/D$ (D is a constant). The event of a vertex v $\in I_3(n_d)$ being in X (independent set) is *independent* of any other vertex in $I_3(n_d)$. Thus the distribution of the cardinality of the independent set produced by algorithm Random-mate can be bounded from below by a binomial distribution with $p = \left(\frac{1}{2}\right)^{d+1}$, and mean $\mu = |I_3(n_d)|\text{p} \geq 6np/D$.

Using Angluin-Valiant's bounds (see Chapter 5 for more details), for $0 < \epsilon < 1$,

$$Prob[|X| \leq (1 - \epsilon)\mu] = \exp(-\epsilon^2\mu/2)$$

Using $\nu = (1-\epsilon)6p/D$ and $c = \epsilon^2 p/(12D)$ we arrive at the required form. □

2.1.3 A parallel algorithm

We have shown that with very high probability, the size of the independent set will be greater than a constant fraction of n, which is the key to finding a logarithmic depth search structure. We present the complete **Point-Location-Tree** algorithm below

Procedure Point-Location-Tree

(0) Let N be the number of vertices in the current stage. If $N \leq k \log n$ (for some fixed constant k), then halt. (then we can locate the query point in the $O(\log n)$ triangular faces in $O(\log n)$ time using a single processor).

(1) Assume that the given PSLG is triangulated. (If not, then we can use the procedure for triangulation described in the latter sections to triangulate a PSLG in $O(\log n)$ time using $O(n)$ processors.)

(2) Choose an independent set of vertices each of which has degree ≤ 12 using the method described above with $d = 12$. Time $= O(1)$.

(3) Triangulate the remaining PSLG (after the removal of the independent set). The removal of a vertex of degree d ($d \leq 12$) necessitates triangulating a simple polygon of d vertices. This can be done in constant time using 1 processor for each polygon.

(4) Determine for each of the new triangles (created in step 3), which of the old triangles it intersects. This can also be done in constant time using one

processor for each of the new triangles using concurrent read.

(5) go to (0)

Theorem 2.1 *Algorithm* **Point-Location-Tree** *runs in* $\tilde{O}(\log n)$ *time using* $O(n)$ *processors and* $O(n \log\log n)$ *space in a CREW PRAM model. Furthermore, if the input PSLG is triangulated, we can reduce the processor bound to* $O(n/\log n)$ *without increasing the asymptotic running time of the algorithm.*

Proof : Each of the steps at any level of recursion of the algorithm can be done in constant time using $O(n)$ processors. From Lemma 2.1, there exist constants ν and c such that the probability of reducing the problem by a constant fraction (1 - ν) is very high. Let n_i denote the problem size (the number of vertices remaining in the graph) at stage i of recursion, where $n_0 = n$. We shall show that after $O(\log n)$ stages, the problem size is less than $O(\log n)$ with very high likelihood.

From lemma 1, $P[n_i \leq (1 - \nu)n_{i-1}] \geq 1 - n^{-cK}$ for $n_{i-1} > k \log n$ for some constant k. The probability that this fails to hold in any level i ($1 \leq i \leq \log_{1/(1-\nu)} n$) is less than $n^{-ck+\delta}$ for any δ. We abort the algorithm if the number of vertices is greater than $k \log n$ after the last stage and re-run the entire procedure. The probability that the entire algorithm fails is less than the probability that one of the stages fail (at least one of the stages has to fail in order that the algorithm fails).

We have yet to account for the cost of updating data structures. We assume that the input is given as a list of edges ordered clockwise around a vertex for each of the vertices. The basic operations that we perform during the course of the algorithm are:

- determining if a vertex has degree less than d
- inserting an edge
- deleting an edge

The first operation takes constant time by letting a processor count the number of edges incident on a vertex up to $d+1$. Because the graph is triangulated at each stage, two consecutive edges cannot be removed simultaneously, and hence insertion and deletion of edges would take only a constant number of pointer updates.

The argument for space is similar to the sequential case. For storing the data-structure (that is being constructed), we can set aside a constant amount of space per vertex. Each vertex that is being removed causes a constant number of old triangles to be eliminated, and a constant number of new triangles to be created. For each edge, we keep a pointers to the (two) triangles that it bounds. We need an extra $O(\log\log n)$ factor space for the triangulation procedure (described in a later chapter).

If the input PSLG is triangulated, the number of operations performed is linear in n which follows from the sequential algorithm. From Brent's slow-down procedure and using the load-balancing scheme of Cole and Viskin [26] or the randomized scheme of Miller and Reif [56], (as explained in section 1.5) the processor bounds can be reduced to $O(n/\log n)$ without affecting the asymptotic run-time. □

REMARK : Dadoun and Kirkpatrick [30] show that their algorithm runs in $O(\log n)$ *expected parallel time. However they do not extend their analysis for the high-probability bounds which is also* $O(\log n)$ *as shown in the previous*

theorem.

Corollary 2.1 *Given a planar subdivision S, consisting of $O(n)$ vertices, we can locate $O(n)$ query points in $\tilde{O}(\log n)$ time using $O(n)$ processors.*

Proof Follows immediately from the previous discussion. □

A direct application of this result leads to the improvement in performance of the two-dimensional Voronoi diagram algorithm due to Aggarwal et al. [4]

Corollary 2.2 *The Voronoi diagram of a set of $O(n)$ points on a plane can be constructed in $O(\log^2 n)$ expected running time using $O(n)$ processors in a CREW PRAM model.*

Proof Aggarwal et al. [4] uses a divide and conquer algorithm, where in each of the $O(\log n)$ stages of recursion we need to perform point location simultaneously for $O(n)$ vertices. Since their planar-point location needs $O(\log^2 n)$ time, the overall algorithm runs in $O(\log^3 n)$ time. Using our randomized planar-point location algorithm, the expected running time of algorithm is $O(\log^2 n)$. □

2.2 An Optimal algorithm for 2-D linear programming

2.2.1 Motivation and main idea

A commonly used strategy in sequential algorithms is to reduce the problem size by a constant fraction over successive stages until the problem size becomes trivial. This is often done by choosing a median element and using it to eliminate a constant fraction of the input elements. Selection, 2-D linear

programming and finding ham-sandwich cuts in two dimensions are examples of such algorithms. Since linear time sequential algorithms for selecting median are known an adaptation of this strategy in the parallel context would require a constant time median-find algorithm using a linear number of processors. Some recent lower-bound results for sorting in CRCW PRAMs (due to Beame and Hastad [11]) makes it unlikely that a similar strategy can be extended to parallel algorithms. Their lower bound of $\Omega(\log n/\log\log n)$ for sorting using a polynomial number of processors implies a similar bound for selection (one can sort by selecting all the elements in parallel). Consequently, it will be difficult to come up with a constant-time median-find algorithm. Note that such a bound does not apply to parallel comparison tree model and in fact constant time selection algorithms are known to exist.

There is evidence that the lower bound of Beame and Hastad would be difficult to overcome even by use of randomization (Ajtai and BenOr [5]). The complexity of most of these algorithms would be unaffected if one works with an *approximate median* rather than the exact median. We define an *approximate median* to be an element in the given set such that the rank of the element is γn for $0 < \gamma < 1$. In this note, we outline a method of finding an *approximate median* in constant time with high probability using randomization employing a linear number of processors. It should however be noted that this does not directly yield a method for mimicing the sequential algorithms since packing would still remain a bottle-neck.

The idea behind the algorithm is the following: From the given input we sample n^ϵ elements. Due to a result by Rajasekaran and Reif [63] and Reischuk [70], the median of the sampled set is an *approximate median* of the given input. Since we want find the median in constant time we cannot hope to ap-

ply the algorithm recursively. One of the problems is that to find the median we have to sum $O(n^\epsilon)$ elements which cannot be done in constant time using a polynomial number of processors. For the n^ϵ elements we can do all the pairwise comparisons in constant time using n processors (assuming $\epsilon < 1/2$). In a n^ϵ by n^ϵ array T we store a 0 in position $T_{i,j}$ $(1 < i,j < n^\epsilon)$ if the ith element is less than the jth element. Hence the median is an element whose corresponding row(column) would have equal number of 0's and 1's. We now sample approximately $\log n$ entries from a row and sum using table-lookup. We claim that the element for which the difference between number of 0's and 1's in the sample is the least gives us an *approximate median* of the n^ϵ sample and consequently of the given input. Below we outline a proof of the above claim. The sequential algorithm would execute in $O(n^\epsilon)$ time steps.

2.2.2 Proof of the key lemma

Let $y_1, y_2, ..y_s$ be s random samples in sorted order drawn independently from a set X of n elements. Let r_i denote the rank of y_k element of the sample. Intuitively we would expect the elements to be roughly n/s apart, so we can expect r_i to be $i \cdot n/s$. The following result from Rajasekaran and Reif [63] gives a bound on the accuracy of this approximation.

Theorem 2.2 *For every α, $\alpha > 0$, $Prob\left(|r_i - i\frac{n}{s+1}| > c\alpha\frac{n}{\sqrt{s}}\sqrt{\log n}\right) < n^{-\alpha}$ for some constant c.*

By choosing s to be n^ϵ for some $0 < \epsilon < 1$ and i to be $s/2$, y_i will be an *approximate-median* with very high probability. However as mentioned before, it appears very difficult to identify $y_{s/2}$[2] in constant time and so we will settle

[2]more appropriately $y_{\lceil \frac{s}{2} \rceil}$

for an *approximate median* of the sample. We shall then show that this element of the sample will be an *approximate median* of the input set. We adopt the following strategy to compute an *approximate median* of the sample: Assume that $\epsilon < 1/2$ (the actual value will be determined later). For every element y_j of the sample we compare it against every other element. We store the results in a table as follows $T_{i,j}$ is 0 if $y_i \leq y_j$ and is 1 otherwise. Next we sample $\beta \cdot \log n$ entries from each row of the table independently where β is a constant less than 1. Since there are only n^ϵ rows, with $n^\beta \cdot \beta \log n \cdot n^\epsilon$ processors we can compute the sum of the zeros' and ones' for the $\beta \log n$ sample in each row in constant time (by table look-up). Next we compute the difference between 0's and 1's for each row and compute the minimum over all such rows. The minimum can be computed in constant time using the following result from [41]:

Lemma 2.2 *The minimum of n elements, each of $O(\log n)$ bits can be computed in constant time using n processors in a CRCW PRAM model.*

We claim that the element s_k such that k is the row for which the minimum was computed is an *approximate median* of the sample with high probability. For this we shall use the following probabilistic bounds known in literature as Chernoff bounds (see Chapter 5 for a more detailed description).

We say a random variable X upper-bounds another random variable Y (equivalently Y lower bounds X) if for all x such that $0 \leq x \leq 1$, Prob(X $\leq x$) $\leq$ Prob($Y \leq x$).

A Bernoulli trial is an experiment with two possible outcomes namely, success and failure. The probability of success is p.

A binomial variable X with parameters (n, p) is the number of successes in n independent Bernoulli trials, the probability of success in each trial being p. The *cumulative density function* of X is given by

$$\text{Prob}(X \leq x) = \sum_{k=0}^{x} \binom{n}{k} p^k (1-p)^{n-k}$$

The tail end of the Binomial distribution can be bounded by *Chernoff* bounds (see Chapter 5 for more details). In particular, the following approximations due to Angluin and Valiant are frequently used:

$$\text{Prob}(X \leq (1-\delta)pn) \leq exp(-\delta^2 np/2) \quad (1)$$
$$\text{Prob}(X \geq (1+\delta)np) \leq exp(-\delta^2 np/3) \quad (2)$$

for all $0 < \delta < 1$.

For keeping the arguments simple, we shall use only the second equation since it is a little weaker. When we sample randomly $\beta \log n$ entries from a row, each sample is a Bernoulli trial and the probability of it being a 1 (0) equals the number of 1's (0's) divided by n^{ϵ}. It follows that for row $y_{s/2}$, the probability that the difference is more than $2\gamma \cdot \beta \log n$ is less than $n^{-\beta\gamma^2/6}$. Thus the row which is chosen as the median by our algorithm has a difference less than this amount. We shall show that such an element is not far away from the median of the sample. In the worst case, the true median and the chosen median are displaced in the 'opposite' directions (because of the sampling error). By 'opposite' directions we mean that if for the median element the number of 1's in the sample is less than the expected value then for the chosen element the

sample has more 1's than the expected value. Let the chosen element E_s be $|1/2 - \gamma_1|s$ away from the median where $\gamma_1 > 1/4$. Again for simplification of our analysis, we shall look at the following event A : $y_{s/4}$ and M_s (median element) cross over. Further define event B to be the case where M_s remains within the band [0, 1/8] i.e. the sampling error is less than $\frac{\beta \log n}{8}$. From elementary considerations $Prob[A] \leq Prob[A|B] + Prob[\neg B]$. Using Chernoff bounds the probability of each of these events can be bounded from above by $n^{-\beta/96}$. This follows from the displacement errors of 1/2 and 1/4 respectively. This gives us the following result: For E_s to be chosen median,

Lemma 2.3 *If the element E_s chosen as median is y_k, then $Prob[|k - s/2| > s/4]$ is less than $n^{\epsilon-(\beta/100)}$.*

Proof: Follows directly by using the above bound for a fixed element and then summing the probabilities over all elements. This is a very crude upperbound since the probabilities of elements which are worse than $y_{s/4}$ to be chosen as median is much less.

A possible choice of values for parameters can be $\epsilon < 1/200$, $\beta = 9/10$ so that this probability is less than 1- n^{-c} for some $c > 0$. Although the bound does not appear to be of any practical significance, it can be improved considerably by a minor modification. Instead of sampling only $\beta \cdot \log n$ entries from each row of the table, we can repeat this K times and add them (i.e. the number of 1's) so that we have an effective sample of size $K \cdot \beta \cdot \log n$. Thus the sample mean of the elements increases by a factor of K. This enables us to improve the bound to $n^{\epsilon-(K \cdot \beta/100)}$.

If we denote y_k to be the chosen sample median , we can state our main result as follows:

Theorem 2.3 *The probability that* $|r_k - n(1/2 - \gamma)| > c\alpha n^{1-\zeta}$ *is less than* n^{-c} *for some non-zero constants* ζ *and* c. *Here* $0 < \zeta$, $c > 0$ *and* $0 < \gamma < 1/2$. *Furthermore such an element can be chosen in constant time in a CRCW PRAM model using a linear number of processors.*

Proof The bound on r_k follows from our previous discussion. The only algorithmic detail left out was the formation of the matrix $T_{i,j}$ in constant time. For this, we partition the input array into n^ϵ non-overlapping groups and let each group choose one random element and write (using concurrent write) to a predetermined position in the table. We consider the algorithm to have failed if one of the two conditions in Theorem 1 or Lemma 2 do not hold. That is either the element chosen as the median of the sample is more than $s/4$ away from the median or its rank is more than $n/4 + \alpha n^{1-\epsilon}$ away from the true median for a fixed constant α. Summing the failure probabilities yields the result. □

Remark: The processor bound can actually be made sublinear with appropriately chosen parameters. This does not come as a surprise since the sequential algorithm is also sublinear. Moreover, we do not have to compute all the entries of table T. For each row, we compare the element with $\beta \cdot \log n$ randomly chosen elements of the table.

2.2.3 Sequential algorithm for linear programming

Meggido [55] gave a linear time (linear in the number of constraints) algorithm for linear programming when the number of variables is fixed. It works especially well for two and three dimensions. A brief description of the two dimensional case is given below.

Given a optimization problem of the form

minimize $ax + by$

subject to $a_i \cdot x + b_i \cdot y + c_i \leq 0$, $i = 1, 2, ...n$

it can be transformed into the form

minimize Y

subject to $\alpha_i X + \beta_i Y + c_i \leq 0$ $i = 1, 2, ...n$ where $Y = ax + by$ and

$X = x$.

Let I_0, I_+ and I_- denote the index sets of the inequalities with β_i zero, positive and negative respectively. Let

$$u_1 = max\{-c_i/\alpha_i : i \in I_0\}$$

$$u_2 = min\{-c_i/\alpha_i : i \in I_0\}$$

Moreover define

$$F_+(X) \equiv min(-\alpha_i/\beta_i(X) + -c_i/\beta_i), i \in I_+$$

and

$$F_-(X) \equiv max(-\alpha_i/\beta_i(X) + -c_i/\beta_i), i \in I_-$$

Now we can write the transformed linear programming problem as

minimize $F_-(X)$

subject to $F_-(X) \leq F_+(X)$

and $u_1 \leq X \leq u_2$

where $F_{-}(X)$ is a piecewise linear downward-convex function and $F_{+}(X)$ is a piecewise linear upward-concave function. Here the function $F_{-}(X)$ is defined as the maximum value of the constraints with the feasible region extending below and $F_{+}(X)$ is the minimum value of the constraints with with the feasible region lying above. This transformed problem has a nice property that given any X, we can determine in $O(n)$ time, whether the optimal point lies to the left or to the right of X. (See [55] for details or [60] pages 284-289 for an overview). This results in the following strategy.

We pair the constraints and compute the intersections for all pairs. This median value is now used as the test point. Thus for all pairs whose intersections lie on the other side of the median (relative to the optimum) we can discard one constraints from each pair. Because of our choice of the test point we can discard one-fourth of the constraints which results in an algorithm which has a linear running time.

2.2.4 Parallel algorithm for linear programming with two variables

Using the aforementioned constant-time algorithm for median find, one can easily derive an $O(\log n)$ time optimal algorithm for 2-D linear programming which has n constraints. This can be done by simulating Meggido's [55] linear-time algorithm for the first $O(\log \log n)$ stages and then solving the problem by constructing the convex hull (of the constraints) once the number of constraints is less than $O(n/\log n)$. Note that Meggido's sequential algorithm will have a linear running time even if we substitute the median by an approximate median. Since convex-hull can be constructed in $O(\log n)$ time using $n/\log n$ processors, the entire algorithm runs in $O(\log n)$ time using $n/\log n$

processors. During the first $O(\log \log n)$ stages, processor allocation can be done using the load-balancing schemes of Cole and Vishkin [26] or Miller and Reif [56]. A similar algorithm was discovered independently by Deng [31] who used a $O(\log n / \log \log n)$ algorithm due to Cole [24] for finding an *approximate median.* Notice that for this particular application an $O(\log n / \log \log n)$ algorithm for finding an *approximate median* suffices.

Chapter 3

RANDOMIZED TECHNIQUES FOR DIVIDE-AND-CONQUER

In this chapter, we shall develop techniques for doing divide-and-conquer efficiently for various problems in computational geometry. Divide-and-conquer is certainly the most commonly used technique for designing parallel algorithms. The idea is analogous to sequential algorithm design where the original problem is sub-divided into smaller subproblems and then the solutions of the sub-problems are combined to obtain a solution to the original problem. The smaller sub-problems are solved recursively until a sub-problem size becomes smaller than a predetermined threshold. At this stage a direct (usually brute-force) method is used to solve it. For analyzing this procedure it usually suffices to write down the recurrence equation for the time complexity as:

$$T(n) = \begin{cases} \max\limits_{i} T(n_i) + g(n) & \text{if } n_i > K \\ F(n) & \text{otherwise} \end{cases}$$

where K is a predetermined threshold, n_i is the size of the ith subproblem, F is the complexity of a direct algorithm and $g(n)$ is the cost of dividing the

problem and recombining the solutions.

For a number of problems, $\sum_i n_i = n$ and hence it is the size of the largest subproblem (which is of size less than n) that is crucial for determining the running time of the algorithm. The equations for the processor and the space bounds can be written similarly, and the processor complexity is the maximum number of processors used at any step of the algorithm. Since there is a trade-off between the number of processors used and the running time, sometimes it becomes necessary to write a recurrence using two variables namely, the problem size and the number of processors used.

A generalization of the above procedure is to allow for *expected* bounds where it is possible to write down the recurrence equation for expected bound with respect to a specific resource. In the above equation for time bound, we can associate a distribution with the size of the largest subproblem, and the solution would be the expected running time of the algorithm. However, such a procedure wouldn't directly yield high-probability bounds for the algorithm. We shall review some of the known techniques that have been used for designing efficient and optimal algorithms for Sorting. In the context of computational geometry, sorting can be looked upon as a one-dimensional problem, and it is instructive to carefully go through the analysis of some well-known sorting algorithms. This will enable us to extend some of the techniques to higher dimensional problems whenever it is possible to do so and also recognize the need for additional techniques because the higher dimensions present additional difficulties.

For the most part of this chapter, the techniques discussed are very general without referring to any particular problem. We shall present some interesting

applications in the next chapter to specific problems where algorithms will be presented more formally. In the context of using random sampling in computational geometry we shall also discuss some important contributions due to Clarkson [19, 20, 21]. These have wide applications in sequential algorithms but they have serious drawbacks if one tries to apply them in the parallel context. In particular, we shall try to bring out the difference between using expected bounds and high-probability bounds.

3.1 Review of Randomized Sorting Algorithms

The idea of using probabilistic analysis for sorting algorithms certainly goes back to an algorithm due to Hoare [47] known as Quicksort. Although the form in which the algorithm was presented would only be efficient for the case for which the input was random, one could always permute a given input randomly to meet this precondition. The algorithm was shown to have an expected running time of $O(n \log n)$ but a slightly modified analysis enables one to obtain high-probability bounds for the same running time. Since it is known that the information theoretic lower bound for any sorting algorithm is $\Omega(n \log n)$ even when the algorithm uses random input, Quicksort is an optimal algorithm.

3.1.1 The sequential algorithm

The algorithm is extremely simple to understand and implement. Given a random input we choose the first key to split the input into two subsets - keys with values greater than or equal to the splitter key and the ones with values smaller than the splitter key. (If the input is not random then a random key

can be chosen as the splitter). If we name these subsets S_2 and S_1 respectively, the sorted output is sorted output of S_1 followed by the sorted output of S_2. The algorithm is applied recursively to the subsets S_i until the size of a subset becomes 2 (when a single comparison suffices). To analyze the above procedure we can write the recurrence relation as:

$$T(n) = T(|S_1|) + T(n - |S_1|) + c \cdot n$$

Assuming that n is a power of 2 (if not, one can always pad some dummy elements increasing the problem size by at most a constant factor in the process), this gives us an optimal solution of $T(n) = O(n \log n)$. The term $c \cdot n$ includes the cost for choosing a random key and comparing all the keys against this. To keep track of the gradually shrinking sets, we can keep a record for each key defining the left and right boundaries of its current interval.

Clearly we cannot hope to split the input into equinumerous sets all the time by choosing a random key. On the contrary, if we choose the keys uniformly over the input, this will happen approximately with probability $1/n$. With a little more thought one can write down a recurrence relation for the expected running time of the algorithm as :

$$\bar{T}(n) = c \cdot n + \frac{1}{n} \sum_{i=1}^{n} [\bar{T}(i-1) + \bar{T}(n-i)]$$

which after some manipulation can be shown have a solution of the form $\bar{T}(n) \leq k \cdot n \log n$ for some constant k.

Although this analysis gives us a formula for an expected running time of the algorithm, it does not give us a good handle on distribution for the running time i.e. how often and by what factor can we deviate from the mean.

Using an alternate analysis and some rudimentary probabilistic inequalities we can show that the algorithm has the same asymptotic running time with high probability. Note that the algorithm is not modified. For the alternate analysis we shall show that a particular key is 'touched' no more than $c \log n$ (no relation to the earlier constant c) time with probability greater than $1 - \frac{1}{n^2}$ times. We say that a key is 'touched' at a particular step of the algorithm if its bounding interval is modified. This happens when the latest splitter is in the bounding interval of that key. We define a binary tree corresponding to the algorithm as follows. Each node represents a sub-interval of [1,n] where the root represents the interval [1,n]. An internal node [l,r] of size greater than 2, has children [l,$k-1$] and [k,r] where k is the rank of the latest key chosen from the sub-interval [l,r]. To determine the number of times a key is touched, we look at the path taken by this key in the binary tree. The path starts at the root and goes left or right depending on which sub-interval the key falls in, and hence the length of this path is an upper-bound to the number of times this key is touched.

Starting from the root, partition this path into disjoint contiguous sub-paths of length 8. We claim the following:

Claim 3.1 *In any subpath, the size of the sub-interval representing the beginning of the sub-path expects to be at least four times the size of the sub-interval represented by the last node.*

We can argue in the following manner. Let the length of the beginning interval be i and with respect to this interval we have chosen 8 random keys. As we go to the right of this key, we expect to to have chosen a key $\frac{i}{8}$ away from this key and similarly to its left. So the expected length of the interval

containing it is at most $\frac{i}{4}$. From Markov's inequality, the probability that the size of this interval exceeds twice the expected size is less than 1/2.

In other words, if we define a successful event as the one where the size of the interval in the end of the path is at most $\frac{i}{2}$ the probability of success is at least 1/2. Since the sub-paths are disjoint and the keys are chosen independently the successful events are independent. The length of the root interval is n and the length of a leaf interval is 2 and so the length of this path is the number of sub-paths which have a total of $\log n$ successful events. This is similar to the number of trials required to obtain $\log n$) heads when the coin is unbiased. The probability that we obtain more than $c \log n$ tails in $(c+1)\log n$ trials (c is an integer greater than 5) is equal to:

$$\sum_{i=c\log n}^{(c+1)\log n} \binom{\log n}{i} \frac{1}{2^{(c+1)\log n}}$$

Using $k! = \Theta(k/e)^k$ from Stirling's approximation, this is less than $\frac{1}{n^2}$. Thus the probability that any key is touched more than $O(\log n)$ times is less than $\frac{1}{n}$. This is in turn implies that the total number of operations over the whole algorithm is $O(n \log n)$ with high probability.

3.1.2 A parallel algorithm

To make this algorithm run in parallel in O(logn) time, we cannot afford to choose the keys one-after the other. A natural way to parallelize will be to choose a set of keys randomly and divide the input into a number of subsets and proceed with all the subsets in parallel by applying the same algorithm recursively. Reischuk [69] observed that by choosing $\lceil \sqrt{n} \rceil$ random keys, the expected parallel running time of the algorithm can by bounded by $O(\log n)$.

In fact the analysis would hold for $\lceil n^\epsilon \rceil$ keys for any $0 < \epsilon < 1$. We shall first outline his proof which gives expected bounds and then give an alternate proof due to [68] which yields high-probability bounds. We note here that while Reischuk's original algorithm was described for a PRAM model, the Flashsort algorithm due to Reif and Valiant [68] executes in a network model.

In Reischuk's algorithm, $\lfloor \sqrt{n} \rfloor$ keys were chosen randomly such that each key was chosen with probability $\frac{1}{\sqrt{n}}$. These keys (called splitters) were then sorted using pairwise comparisons and summing their ranks. The latter can be done in $O(\log n)$ time very easily and we can do the pairwise comparisons simultaneously in constant time by using one processor for each comparison. These splitters partition the n input keys into $\lfloor \sqrt{n} \rfloor + 1$ buckets. For each key we can determine the appropriate bucket by a simple binary search using one processor for each key (using simultaneous reads). If we let n_i denote the size of the i-th bucket then we claim the following:

Claim 3.2 *The probability that for any i, n_i is larger than $c\sqrt{n}\log n$ is less than $\frac{1}{n^{(c-1)}}$.*

Proof: We shall show that for any key (whether or not it is in the sample), the probability that it is more than $c\sqrt{n}\log n$ away (in rank) from the next sampled key on its right is less than $\frac{1}{n^{(c-1)}}$. This follows from the fact that each key was chosen with probability $\frac{1}{\sqrt{n}}$ and hence the above event can happen with probability less than $\left(1 - \frac{1}{\sqrt{n}}\right)^{c\sqrt{n}\log n}$. For large n this can be bounded by $\frac{1}{n^3}$. Hence the probability that it can happen for any element is less than $\frac{1}{n^2}$. Consequently the distance (in rank) between two sampled elements is less than $c\sqrt{n}\log n$ with probability at least $\frac{1}{n^{(c-1)}}$.

For $c = 3$, we can write the recurrence for the expected running time of the algorithm as:

$$\bar{T}(n) \leq (1 - \frac{1}{n^2})\bar{T}(3\sqrt{n}\log n) + \frac{1}{n^2}\bar{T}(n - \lceil\sqrt{n}\rceil + 1) + O(\log n)$$

By induction it can be shown that $\bar{T}(n) \leq O(\log n)$.

To derive high probability bounds, we borrow an idea from [68] and present it in a more general form. This will be used repeatedly for analyzing algorithms in future. We shall also make use of the fact that Reishchuk's algorithm would execute in the same asymptotic time bound even if the number of sampled keys is $\lfloor n^\epsilon \rfloor$ for any fixed ϵ, $0 < \epsilon \leq 1/2$. Note that in general, the probabilistic bound of Claim 3.1 holds for partitions of sizes $O(n^{1-\epsilon}\log n)$ (the claim was for the case $\epsilon = 1/2$). For convenience, we shall use the bound $O(n^{\epsilon_0})$ where $\epsilon_0 > 1 - \epsilon$. Thus at depth i from the root, the size of a sub-problem can be bounded from above by $n^{\epsilon_0^i}$.

It may be helpful to view the algorithm as a tree where a node represents a subproblem and its children represent the recursive calls made by this node. For example, the root represents the procedure Sort[1,n] which has $\lfloor n^\epsilon \rfloor + 1$ children procedure calls each of size at most $n^{1-\epsilon}\log n$. The leaves of this tree represent problems of size less than a pre-determined threshold, say $\log^r n$ for some fixed integer r. At this stage the problem size is so small that we can use a direct sorting procedure like Batcher's sort to sort in time $O(\log\log n)$, thereby adding a factor of $o(\log n)$. Our objective is to show that all the leaf-level procedures are completed in $O(\log n)$ time with high probability. For this it suffices to show that a particular leaf-level procedure is completed in $\tilde{O}(\log n)$ time. This leaf-node defines a fixed path from the root to the leaf such that the problem sizes at successive nodes of this path are decreasing. For this let

us denote the node at depth i from the root as N_i, the problem-size as n_i and the time taken at N_i by T_i. We claim the following:

Claim 3.3 *:*

$$Prob[T_i \geq k \cdot \epsilon_0^i c\alpha \log n] \leq 2^{-c\alpha\epsilon_0^i \log n}$$

where c and α are integers and k is a constant.

Proof: From the previous claim and the comment in the previous paragraph, $\text{Prob}[n_{(i+1)} > n_i^{\epsilon_0}] < \frac{1}{n_i^2}$ for an appropriately chosen constant $0 < \epsilon_0 < 1$. We can verify this in $\log(n_i)$ time (using prefix sum) and we repeat the sampling until $n_{(i+1)} \leq n_i^{\epsilon_0}$. If $k\log(n_i)$ is the time for each iteration (of the sampling algorithm) then we can immediately conclude the following:

$$Prob[T_{(i+1)} > kc\alpha \log n_i] < 2^{-c\alpha 2 \log n_i}$$

If $n_i = 2^{\epsilon_0^i \log n}$ then we arrive at the required inequality. From our resampling scheme we have guaranteed that $n_{(i+1)} \leq 2^{\epsilon_0^i \log n}$, so we have to prove that the claim is true when n_i is strictly less than $2^{\epsilon_0^i \log n}$. Let $n_i = 2^{(1/a)\epsilon_0^i \log n}$ where $a > 0$. Substituting this value of n_i in the previous inequality we get:

$$Prob[T_{(i+1)} > kc\alpha(1/a)\epsilon_0^i \log n] < 2^{-c\alpha 2(1/a)\epsilon_0^i \log n_i}$$

The above inequality implies that

$$Prob[T_{(i+1)} > kc\alpha(\lfloor a \rfloor/a)\epsilon_0^i \log n] < 2^{-c\alpha 2(\lfloor a \rfloor/a)\epsilon_0^i \log n_i}$$

Since $2\lfloor a \rfloor/a \geq 1$ for any $a > 1$, the claim follows.

We are now ready to prove the main result of this section:

Theorem 3.1 *Given a process-tree which has the property that a procedure at depth i from the root takes time T_i such that*

$$P[T_i \geq kc\alpha \log n(\epsilon_0)^i] \leq 2^{-(\epsilon_0)^i c\alpha \log n}$$

then, all the leaf-level procedures are completed in $\tilde{O}(\log n)$ time.

Proof: Setting $t_i = k(\epsilon_0)^i \log n\alpha(c - c_o)$, we obtain

$$Prob[T_i \geq kac \log n(\epsilon_0)^i + t_i] \leq 2^{-(\epsilon_0)^i c\alpha \log n} \leq 2^{-t_i/k}$$

If T is the total time for this worst case chain of nested calls and m = 1/(1-ϵ_0), the probability that it takes more than $mk\alpha \log nc_o + t$ time, is less than the sum of the probability of events where $\sum_i t_i = t + j$, $0 \leq j \leq \mu$. Here $\mu = mk\alpha \log nc_o$. We shall compute the probability that $\sum_i t_i$ = t and multiply by μ.

$\prod_{\sum t_i = t} 2^{-t_i} \leq \sum 2^{-t/k}$ over $(\log n)^{O(loglogn)}$ tuples (number of distinct ways of partitioning t into $O(\log \log n)$ parts).

Thus Prob$[T > km\alpha \log nc_o + t] < \mu 2^{-t/k + O(loglog^2 n)}$

Using $t = 2km\alpha(c - c_o) \log n$, for large values of n and $m > 1$, we can rewrite the above expression as

$$Prob[T > km\alpha c \log n] < \mu 2^{-\alpha m(c - c_o)\log n}$$

For $c > 4c_o$, i.e. $c - c_o > 3/4c$, we have the following required bound,

$$Prob[T > \alpha logn] \leq \mu 2^{-(3/4)c\alpha \log n} \leq n^{-c_1 \alpha}.$$

assuming that k, m and c are larger than 1.

3.2 Extension to Higher dimension

We shall now try to apply the techniques developed in the previous section to a specific geometric problem on the plane. For this purpose we have chosen the problem of constructing two-dimensional convex hull of point sites. Although optimal $O(\log n)$ time parallel algorithms have been known for some time, its close relation to sorting makes it an ideal testing ground for the methods developed in the earlier part of this chapter. Moreover, being a simple problem to describe and understand it is also a suitable candidate for exposition.

3.2.1 A straightforward extension

We shall actually look at its dual problem namely, the intersection of n half-planes. Without loss of generality we can assume that a point O in its interior is known (since such a point can be found easily in in $O(\log n)$ time in parallel using linear programming). Following on the lines of the sorting algorithm, we choose a random subset of $\lfloor n^{\epsilon} \rfloor$ half-planes where $0 < \epsilon < 1$. We can construct their intersection in $O(\log n)$ time using a brute-force method like checking for each pairwise intersection, if it is a vertex of the convex hull. Let $h_0, h_1, ..$ be the vertices of the convex-hull in a cyclic order (there can be at most $\lfloor n^{\epsilon} \rfloor$ of them). Consider the triangles (will also be referred to as sectors) of the form O,h_i, h_j. These will be intersected by a number of half-planes that were not chosen in the sample. The output convex hull is the union of the boundaries formed by the intersection of the half-planes inside each of the sectors. This gives a recursive procedure for constructing the convex hull. For each sector, we determine the half-planes intersecting the region and then call the algorithm recursively for each of the sectors.

To determine the sectors that a half-plane intersects, we can use a very simple procedure due to Chazelle and Dobkin [16]. Using a processor for every half-plane, in $O(\log n)$ time, we can determine the edges of the convex hull that the half-plane intersects which gives us the information of the bounding sectors that this half-plane intersects. Due to convexity, it intersects all the sectors that lie in between. We can very easily determine the number of sectors that it intersects in the same time (without explicitly listing the sectors).

In essence, the algorithm appears to be identical to that of parallel sorting. To prove any interesting results we have to determine how quickly the subproblem sizes are decreasing. However there is an obvious difference namely, the total size of the subproblems may not be bound by the parent's problem size. This happens because a half-plane can intersect more than one sector which results in fragmentation. This is crucial for the processor's bound and large fragmentation could ruin the possibility of an optimal algorithm (i.e. one in which the PT product is $O(n \log n)$). Below we prove some results on the problem size and obtain a bound on fragmentation. For simplicity of the arguments we shall assume that no three half-planes have a common intersection point.

Lemma 3.1 *The probability that the maximum number of half-planes intersecting any sector exceeds* $2(c+2)n^{1-\epsilon} \log n$ *is less than* n^{-c}.

Proof: Consider all the $O(n^2)$ pairwise intersections of the lines bounding the half-planes. Draw the segments joining O and each of these intersections and consider the ordered intersections of lines (representing the boundaries of half-planes) on this segment. For any given segment the probability that the number of intersections exceeds $(c+2)n^{1-\epsilon} \log n$ before the first (counting from

0) sampled half-plane is less than $\left(1 - \frac{1}{n^{1-\epsilon}}\right)^{(c+2)n^{1-\epsilon}\log n}$ which is less than $n^{-(c+2)}$ for large n. Thus the probability that this happens for any segment is less than n^{-c}. This implies the lemma since any half-plane intersecting the sector intersects at least one of the two bounding segments. □

Lemma 3.2 *The expected value of the sum taken over all the sectors of all the half-planes intersecting a sector is O(n).*

Proof From the proof of the previous lemma, it is clear that if the number of half-planes intersecting a segment is greater than $n^{1-\epsilon}$, then the expected number of half-planes can be upper-bounded by a geometric distribution. The probability of success is $1/n^{1-\epsilon}$ which is the probability of being selected in the sample. So the expected number of half-planes that we do not select in the ordered list of half-planes (starting from 0) before we select the first half-plane is $n^{1-\epsilon}$. Using the property that the expectation of the sum of random variables is the sum of the individual expectations, we arrive at the required bound. In our case we are interested in the sum of n^{ϵ} random variables.

Clarkson [21] had derived similar bounds for a more general setting using more complicated machinery. He had also given numerous applications of these very general probabilistic bounds. In his case, however he was dealing with the sequential algorithms and so he could derive bounds on the expected running time of the algorithms as the sum of the expected time for individual steps. His methods are not known to yield high-probability bounds without changing the asymptotic running time of the algorithms. In the parallel setting, if we use as an abstract representation of a recursive algorithm the tree as described in the previous section, the running time of the algorithm is proportional to

the longest path (in time) from the root to a leaf. Thus at any given node we are interested in the recursive call that takes the the longest time. Given that we only have a bound for the *expected* time taken by the child-procedures, and there are n^ϵ of them, we cannot derive any useful tail estimates. The reason that we could derive interesting bound for the sorting algorithm is because we were able to get tail estimates (of the problem size exceeding a certain size).

The bound on the total size of the subproblems is not known to hold with high-probability. However, we can claim the following:

Claim 3.4 *For some suitable constant k_{total} and large n, the following conditions hold with probability at least 1/2:*

> *(i) The maximum number of half-planes intersecting any sector is less than $2n^{1-\epsilon}\log n$*
>
> *(ii) The sum of half-planes taken over all the sectors of the number of half-planes intersecting a sector is less than $k_{total}n$.*

Proof From Lemma 3.2 and Markov's inequality we can choose k_{total} such that the probability that (ii) fails is at most 1/3 (i.e. kn is thrice the mean). For sufficiently large n, $1/n + 1/3$ is less than 1/2. Thus the probability that both (i) and (ii) are satisfied is at least 1/2. In future we shall call a random sample 'good' if the above conditions are satisfied and 'bad' otherwise.

3.2.2 Resampling and Polling

As a consequence of the previous claim, if we repeat the sampling algorithm $r\log n$ times, the probability that the conditions are not satisfied during all the tries is less than n^{-r}. That is if we choose independently $p(n) = O(\log n)$

sets of samples, one of them is good with very high likelihood. However, to determine if a sample is 'good', we would have to carry out the search procedure $O(\log n)$ times each of which requires $O(\log n)$ time (such a method was described earlier). Instead, we try to estimate the the number of half-planes intersecting a sector C_i using only a fraction of the input half-planes. For example, we can choose $c_0 \cdot n/\log^d n$ for some fixed integer $d \geq 1$ and a constant c_0[1] of the input half-planes randomly for the jth sample, R_j. Let X_i^j be the number of half-planes intersecting sector C_i corresponding to sample R_j, $1 \leq j \leq b \log n$ where b is fixed integer greater than 0 which is determined from the success probability of the algorithm. A_i^j be the number of half-planes intersecting C_i out of the $n/\log^d n$ randomly chosen input half-planes for the same sample. Clearly, A_i^j is a binomial random variable with parameters $c_0 \cdot n/\log^d n$ (number of trials) and X_i^j/n (success probability). Assuming that X_i^j is greater than $\bar{c} \cdot \log^{d+1} n$, for some constant $\bar{c}$, we will apply Chernoff bounds (see Chapter 5 for discussion on probabilistic inequalities) to tightly bound the estimates within a constant multiplicative factor. Since we do it only for $1/\log^d n$ of the input half-planes, the total number of operations for the $O(\log n)$ random subsets can be bounded by $O(n \log n)$ (as we show in the next section). Note that $X_i^j < \bar{c} \log^{d+1} n$, is an easy case since $n^{\epsilon} \cdot \bar{c} \log^{d+1} n = o(n)$.

3.2.3 Probabilistic analysis of Polling

More formally, by invoking Chernoff bounds), for any $\alpha > 0$ (α is a function of c_0), there exists a c_1, independent of n, Prob$(A_i^j \leq \alpha c_1 X_i^j/\log^d n) \leq 1/n^{\alpha}$

[1]the actual value will be determined from the required success probability of the algorithm

and $\text{Prob}(A_i^j \geq c_2\alpha c_0 \cdot X_i^j / \log^d n) < 1/n^{c_0\alpha} < 1/n^{\alpha}$ (for $c_0 > 1$). From the last two inequalities, X_i^j is bounded by $L^j = A_i^j \log^d n / c_0 c_2 \alpha$ from below, and by $U^j = A_i^j \log^d n / c_1 \alpha$ from above. With appropriate changes in the constants, this condition holds with high likelihood (as defined in section 2.1) for all X_i^j simultaneously. We do the searching procedure (described in the next section) simultaneously for all the samples R_j and choose the sample R^{j_o} using the following simple test:

Algorithm *Polling*

(Let $N^j = \sum A_i^j$ and the let actual number of intersections be denoted by T^j and the upper and lower bounds obtained from N^j by U^j and L^j respectively).

> If $k_{total}n > U^j$ then accept sample R^j (since $k_{total}n \geq U^j \geq T^j$), else if $k_{total}n \leq L^j$ then the sample is 'bad' (since $k_{total}n \leq L^j \leq T^j$), else if $L^j \leq k_{total}n \leq U^j$, then accept the sample R^{j_o} for which U^{j_o} is minimum. Since both $k_{total}n$ and T^{j_o} lie in this interval this guarantees that $T^{j_o} \leq c_3 \cdot k_{total}n$ where $c_3 = U^j/L^j$ which is a constant.

Recall, that from our earlier discussion at least one of the samples would satisfy conditions 1 or 3 with very high likelihood. We summarize as following :

Lemma 3.3 (Polling lemma) *If we can choose a set of random splitters that expects to be 'good' (i.e. satisfies certain properties), then by using the* polling *algorithm, as described earlier we obtain a sample that is 'good' with high probability.*

The above procedure can be used in a more general situation where we need 'good' samples with very high likelihood from samples that only expect to be 'good'. Moreover, according to our previous discussion, the extra amount of overhead does not affect the asymptotic work done by the algorithm, because it uses only a fraction of the input to test the samples. A nice feature of this technique is that it is inherently parallelizable.

3.2.4 Bounding the number of processors

Until now we had focussed on getting a 'good' sample with high probability which will ensure that the sum of the sizes of sub-problems is within a constant factor of size of the parent problem. But this only guarantees that over $O(\log \log n)$ levels of recursive calls, the sum of the sizes of the subproblems will be $O(n \log^b n)$ for some constant b. This implies that either we use $O(n \log^b n)$ processors or settle for a corresponding trade-off in the running time. Clearly, we need some stronger observations to prevent this proliferation over every level of recursive calls.

For this we shall use geometric properties of the specific problem. After getting a 'good' sample we shall do further processing to bound the sum of sizes of the sub-problems by the exact size of the parent problem. This will prevent proliferation in the problem size over successive recursive calls. This *filtering* procedure is a kind of post-processing step after random-sampling. The input size is at most a constant factor times the input size (guaranteed by the polling algorithm) of the problem while the output is no more than the input size.

In case of two-dimensional convex hulls, we make use of the following filter-

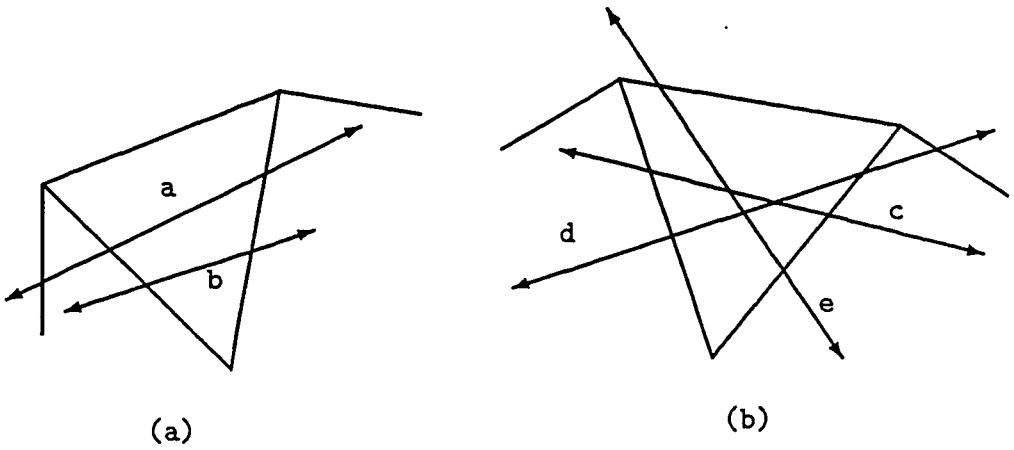

Figure 3.1: (a) illustrates case (a) - line a is completely occluded by b. (b) illustrates case (b). Line c is occluded by lines d and e but not by any one of them completely. Also it is clear that in all other cones line c will be eliminated by case (a).

ing scheme. For any sector, we identify the half-planes that intersect it. Some of them may be part of the output while some of them may not show up in the output (in that sector). Since the output size is bounded by the input size, our objective is to eliminate all the half-planes from a sector that do not show up in the output. It suffices to consider the following cases.

Case 1: If a half-plane is occluded completely by another half-plane within a sector (see Figure 3.1 a), then we can discard these half-planes by the following strategy. For every half-plane intersecting a sector, consider the points of intersection with the two boundaries of the sector. Sort these intersection points in increasing order starting from O. At the end of this step we have two sorted lists. For every half-plane, consider a tuple of the form (x_i, y_i) where x_i and y_i are respectively the ranks of the sorted intersection on the two boundaries.

A half-plane H_i is completely occluded by another half-plane H_j if and only if $x_i > x_j$ and $y_i > y_j$. In other words, if we compute the maximal elements of the tuples, then the elements that are not a part of this set can be left-out from further calls of recursion. This is the easy case.

Case 2: A half-plane may be occluded due to the combined interaction of two half-planes (see Figure 3.1b). In this case, notice that in all other sectors (that this half-plane intersects) it will be eliminated by the previous case.

Case 3: If a half-plane is visible in at most one sector, it will clearly be eliminated from all other sectors by case 1 (by convexity arguments).

Case 4: If a half-plane is visible in more than one sector, then in all but two sectors it will eliminate all other half-planes by case 1 and moreover we do not have to call the algorithm in these sectors (since we know the output). This half-plane can contribute to vertices of the output hull in at most two sectors. During subsequent recursive calls on these two sectors, at most one copy of the half-space will be retained in each sector. From a global view-point, at any stage of the algorithm, we retain at most two copies of a half-plane. If we let an output vertex to be represented by the two intersecting lines, a half-plane can appear in at most two such tuples, which we can denote by the left-vertex (l_v) and the right-vertex (r_v). The *output size* is defined as twice the number of vertices. (Thus if a sector does not have a vertex the output size is 0 implying that an edge traverses the sector and hence we have already determined the convex hull in that sector). Assume that we have $2n$ processors for the algorithm. Then our processor-allocation strategy is as follows:

For each half-plane that we know to be visible in more than one sector allocate one processor each to the bounding sectors on the left and right. These processors can be charged to l_v and r_v respectively. In subsequent recursive calls

at most one processor will be allocated for l_v and r_v.

For each half-plane that we have not been able to determine its visibility in the current recursive call, allocate two processors. From our previous observations, these half-planes can appear in at most one sector.

To summarize, we have achieved the following:

- The total number of processors is always bounded by $2n$.
- For any subproblem (at any recursive call), the number of processors is greater than or equal to the output-size of that sector. This ensures that we do not have to rebalance processors between these sub-problems as the algorithm proceeds recursively.

We now have a parallel algorithm for constructing a convex hull in two dimensions which satisfies the properties of parallel sorting algorithm, especially Claim 3.3 (although we had to work much harder to get to it). Hence from Theorem 3.1, it follows that the above algorithm runs in $\tilde{O}(\log n)$ using n processors in a CREW PRAM model. We had mentioned before that optimal deterministic $O(\log n)$ time algorithms for 2-D convex hulls have been reported in the literature previously. Nevertheless, we feel that our approach opens up a new set of techniques for parallel algorithms in computational geometry which are quite general and should have wider applications. In the next chapter we use the basic tools developed here to present algorithms for some very fundamental problems in computational geometry. Although the algorithms do not remain as simple as in the case of 2-D convex hulls, the underlying approach is very similar.

Chapter 4

APPLICATIONS OF RANDOMIZED DIVIDE-AND-CONQUER

We shall now present optimal algorithms for two fundamental problems namely, trapezoidal decomposition and 3-D convex hull using techniques developed in the previous chapter. While an $O(\log n)$ time deterministic optimal algorithm for trapezoidal decomposition was presented (by Atallah, Cole and Goodrich [7] around the same time this work was done, there is no known deterministic optimal $O(\log n)$ time algorithm for 3-D convex hulls. Both of these problems have numerous applications to other problems, the more important ones being triangulation and 2-D Voronoi diagram.

Let us recapitulate the main steps of the algorithm for 2-D convex hulls described in the last section in a more general context of a divide-and conquer algorithm

(1) Select $O(\log n)$ subsets of random objects (in case of 2-D hulls these were half-planes) each of size $\lfloor n^{\epsilon} \rfloor$ for some $0 < \epsilon < 1$.

(2) Select a 'good' sample using *Polling.*

(3) Divide the original problem into smaller sub-problems (the maximum size can be bounded by $O(n^{1-\epsilon}\log n)$) using the 'good' sample.

(4) Use a *Filtering* algorithm to bound the sum of the sub-problem sizes by some fixed measure like the output size or input size. This step is problem dependent and uses the specific geometry properties of a problem. The purpose is to bound the number of processors.

(5) If the size of a sub-problem is more than a threshold (usually it is chosen to be $O(\log^k n)$ for some constant k), then call the algorithm recursively else solve the problem using some direct method.

The algorithms presented in this chapter are based on this approach. However the implementation of some of these steps depend heavily on the specific problem. The probabilistic bounds used in step 3 have to be proved for the specific problem. In this regard, Clarkson [21] presented bounds for very general situations which are applicable to our problems. However we have chosen to present alternate arguments which are simpler. Moreover, the procedure used for dividing the subproblems would naturally depend on the problem at hand. Perhaps the step that is most specific to a problem is the *Filtering* step where we have to use some geometric properties of the problem.

4.1 An optimal $O(\log n)$ time algorithm for Trapezoidal decomposition

There are many problems in computational geometry whose optimal sequential algorithms use the plane-sweeping paradigm. Aggarwal et al. [4] had proposed

a data-structure, which they call **Plane-sweep-tree.** Using this, they were able to solve a number of problems efficiently (such as planar point location and triangulations among others) in parallel using a linear number of processors. This data-structure is similar to the **Segment-tree** (proposed by J. Bentley and is also discussed in [60]), the intervals of which are induced by the projection of the endpoints of the input set of segments on one of the axes. This was used to emulate the plane-sweeping paradigm. The idea of using the plane sweep tree was later further enhanced by Atallah and Goodrich [8]. We shall review some definitions and results from their papers as they relate to our work.

4.1.1 Review of plane-sweep tree

Let T be a complete binary tree with its leaves corresponding to the $2e + 1$ intervals formed by projecting the $2e$ endpoints of a given set of e line segments onto the x-axis. Associated with each node v in the tree is an interval $[a_v, b_v]$ on the x axis corresponding to the union of intervals of its descendants. Let Π_v represent the vertical strip $[a_v, b_v] \times (-\infty, \infty)$. A segment s_i *covers* a node v in T if its projection $\mathrm{X}(s_i)$ on the x-axis spans the interval $[a_v, b_v]$ (corresponding to v) but does not span the interval of v's parent node. It can be easily shown that no segment s_i covers more than 2 nodes at any level and hence no more than $2 \log n$ nodes of the tree.

For each node v of T, we keep track of the the following properties:

$\mathrm{H(v)} = \{s_i \mid s_i \text{ covers v}\}$,

$\mathrm{W(v)} = \{s_i \mid s_i \text{ has at least one endpoint in } \Pi_v\}$.

$\mathrm{L(v)} = \{s_i \mid s_i \in \mathrm{W(v)} \text{ and } s_i \text{ intersects the left boundary of } \Pi_v\}$.

$\mathrm{R(v)} = \{s_i \mid s_i \in \mathrm{W(v)} \text{ and it intersects the right boundary}\}$.

I(v) = $\{s_i$ | the left endpoint of s_i is in $\Pi_{leftchild(v)}$ and the right endpoint is in $\Pi_{rightchild(v)}$.

If the given set of segments are non-intersecting (as in the case of a polygon), then these sets are completely ordered on y coordinate. The above quantities are computed while constructing the plane-sweep tree, which are used to search for segments directly above or below a given query point.

Multilocation : Given a plane-sweep tree, this procedure locates for each input point p, the segment in H(v) which is directly above (or below) p for all vertices v in the tree such that p belongs to the interval Π_v.

Augmented Plane sweep tree : This is a data structure obtained from a plane sweep tree by adding pointers to every node such that given the position of an element in any node, its position in the parent node can be located in an additional constant time. (For readers familiar with fractional cascading, this data structure enables us to perform that.)

4.1.2 Development of the new data-structure

Fact 4.1 *(Atallah and Goodrich [8]): Multilocation of any query point can be done in $O(\log n)$ sequential time in an augmented plane-sweep tree.*

The preprocessing cost of building this data-structure was $O(\log^2 n)$ time using $O(n)$ processors and $O(n \log n)$ space. Consequently, the above-mentioned problems needed $\Omega(\log^2 n)$ time. Atallah and Goodrich [8] improved the preprocessing time for constructing the plane-sweep tree to $O(\log n \log\log n)$ using parallel merging techniques of Borodin and Hopcroft [12] while keeping the number of processors and the space requirement unchanged. The depen-

dence of their approach on parallel merging can be summed up in the following manner :

Fact 4.2 *The Plane-Sweep Tree of k segments can be constructed in $O(f(p,k)\log k)$ time using p processors ($p \geq k$), where $f(p,k)$ denotes the time complexity of merging two sorted lists of $O(k)$ elements using p processors.*

Since $f(k,k) = \Theta(\log\log k)$ (see Valiant [77] and Borodin and Hopcroft [12]), the bound $O(\log n \log\log n)$ of Atallah and Goodrich's **Build-Up** algorithm for plane-sweep tree follows. Moreover, it enables us to state a useful lemma:

Lemma 4.1 *The Plane-Sweep tree of n segments can be constructed in $O(\log n)$ time with $O(n^{1+\epsilon})$ processors using Atallah and Goodrich's [8] approach.*

The proof follows from the well known result (Valiant [77] and Borodin and Hopcroft [12]) that two sorted lists of O(n) elements can be merged in constant time using $n^{1+\epsilon}$ processors. Using $f(p,k) = O(1)$ in Fact 4.1 yields the required result.

An important common characteristic of Atallah and Goodrich's [8] algorithms is that the time complexity is dominated by the preprocessing cost. More specifically, given the precomputed data-structure, triangulation and planar-point location can be done in $O(\log n)$ time using n processors in a CREW model. Thus, it is worthwhile to try to improve the preprocessing time to $O(\log n)$ so that the overall running time of the algorithms can be reduced to $O(\log n)$. Our approach avoids merging two O(n) size lists by using random sampling to build up a new data-structure called the **Nested-Plane-Sweep tree** in $O(\log n)$ time with very high probability. The overall running time of our algorithms is $\tilde{O}(\log n)$ with $O(n)$ processors.

Procedure Nested-Sweep-Tree

Input: Set S of non-intersecting line-segments $l_1, l_2, ..., l_n$ in a plane.

(1) Choose a set of $\sqrt{n}$ segments randomly from the original set. Each segment is selected with equal probability of $\frac{1}{\sqrt{n}}$.

(2) Use the **Build-up** and **Augment** algorithms of Atallah and Goodrich [8] to construct the augmented plane sweep tree. Time = O(logn), Space = $O(\sqrt{n}\log n)$.

(3) Locate the $n - \sqrt{n}$ segments in the plane sweep tree i.e. identify the region (the $O(\sqrt{n})$ regions) into which the sample set divides the plane) where each of the segments lie. A segment may lie in more than one region in which case it is broken into a number of smaller segments each of which lies in only one region. The implementation of this step is described in section 4.1.4.

(4) If the number of segments in any region is more than a threshold, then apply procedure Nested-Sweep-Tree to each of these regions. (The threshold can be chosen to be $O(\log^r n)$ for some non-negative integer r).

4.1.3 Probabilistic bounds

Lemma 4.2 *The Plane-Sweep tree of the random sample of $\sqrt{n}$ segments divides the plane into less than or equal to $3\sqrt{n}$ trapezoidal regions.*

Proof : Assuming distinct endpoints there are $2\sqrt{n}$ vertical segments through the endpoints, each of which can be shared by 3 such trapezoidal regions (see Figure 4.1). Moreover each trapezoidal region is bounded by at most two vertical segments. Thus, there are at most $3\sqrt{n}$ such regions. □

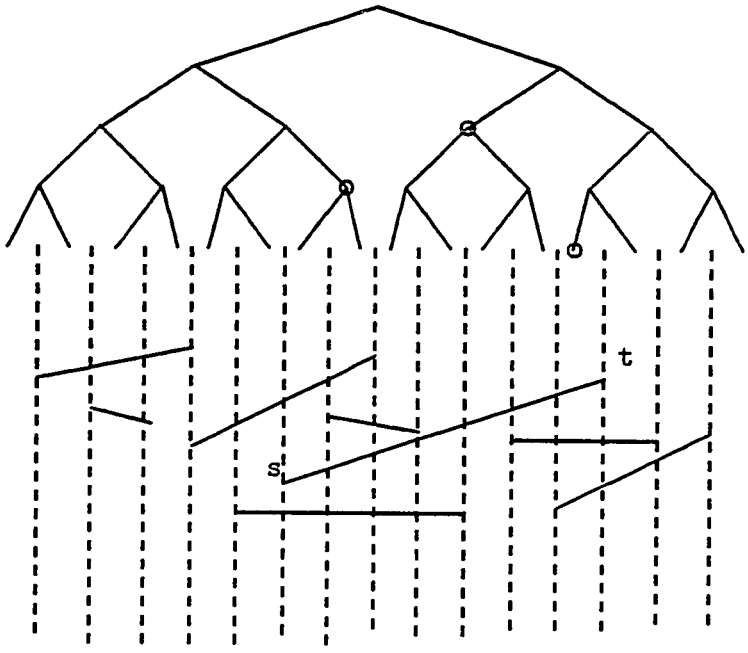

Figure 4.1: The skeleton of a plane-sweep tree. The circled nodes have portions of the segment s-t (corresponding to the intervals covered by the nodes).

To bound the number of segments lying in a given trapezoidal region, we classify the segments into two groups:

(a) Segments that have at most one of their endpoints inside the region.

(b) Segments that lie completely within the region.

(a) The segments that lie partially in the region intersect either the left boundary, or the right boundary or both boundaries of the region. Consider an ordered list of the segments $L_U(i))$ intersecting the vertical segment passing through an end-point in the upward direction. These segments can be completely ordered with respect to the ordinate (on y). The probability that this ordered list exceeds m is less than $(1 - \frac{1}{\sqrt{n}})^m$ because each segment has probability $\frac{1}{\sqrt{n}}$ of being chosen in the sample. Let us denote by E_i the event that $m < k\sqrt{n}\log n$ (for some $k > 1$) for a fixed endpoint i. The probability that the event occurs for any end-point (and there are $2n$ such end-points) can be upper-bounded by $\sum_{i=1}^{2n} e^{-k\log n}$. For $k > 2$, the event that the ordered list of line segments $L_U(i)$ is less than $k\sqrt{n}\log n$ for all i (which is the complement of the previous event) holds with very high likelihood. Thus the number of segments intersecting any boundary is less than $O(\sqrt{n}\log n)$ with high likelihood (this is only a subset of the previous event).

Moreover, the expected number of line segments that intersect a boundary can be upper-bounded by $\sqrt{n}$ (mean of geometric distribution). Note that the random variables representing the number of segments intersecting each of the regions are not independent. Since the expectation of the sum is the sum of expectation, the total number of such segments over all the regions is expected to be less than $12n$ (since a trapezoid has at most two vertical segments going up

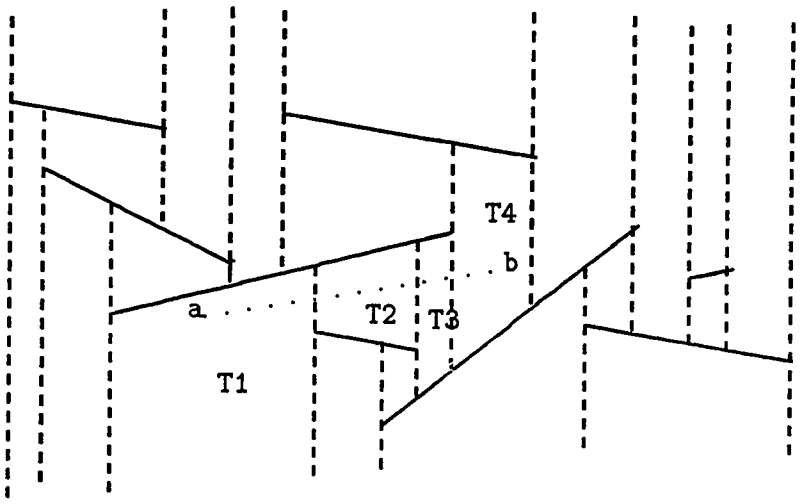

Figure 4.2: Segment a-b is multilocated in the trapezoidal regions labeled T1, T2, T3, T4.

and two going down and there are less than $3\sqrt{n}$ trapezoids). It follows that, the probability that the total number of lines intersecting all the trapezoids exceeds $k_{max}n$ ($k_{max} > 24$) is less than 1/2. This implies that by repeating the experiment of choosing $\sqrt{n}$ segments $c \log n$ times, the probability of failure (where a success is the event that the total number of intersections is less than k_{max}n) is less than $\frac{1}{n^c}$.

(b) The $O(\sqrt{n})$ sub-divisions are trapezoidal (see Figure 4.2) regions which can be specified by at most four segments. Thus, there are $O(n^4)$ potential trapezoids of which only $O(\sqrt{n})$ were chosen in the random sample (Lemma 3). We shall call a trapezoid 'candidate' if it has more than $O(\sqrt{n} \log n)$ segments within it. Note that this is defined on the input configuration. If a trapezoid selected in the random sample has more than $O(\sqrt{n} \log n)$ segments within it, it implies that this trapezoid is a 'candidate' trapezoid and that none of the segments lying in it were selected in the random sample. The probability that

a particular trapezoid contains more than m segments is less than $(1-\frac{1}{\sqrt{n}})^m$. If we choose m to be $5\sqrt{n}\log n$, the probability that such a events can happen for any 'candidate' trapezoid (certainly bound by $O(n^4)$) is less than $1/n$. The total number of segments lying within all trapezoids is obviously less than n (from the input size).

Using arguments analogous to the previous chapters, we can use **Polling** to chose a 'good' sample with high probability. We can summarize the above observations in the following lemma:

Lemma 4.3 *There exist constants α, $\bar{c}$ such that the probability of any one of the $O(\sqrt{n})$ trapezoidal regions containing more than $O(\bar{c}\alpha\sqrt{n}\log n)$ segments (or more appropriately broken segments) is less than $n^{-\alpha}$. Moreover, the total number of such segments over all trapezoidal regions is less than $k_{max}n$ with very high likelihood.*

The success of the recursive calls at any level can be argued in an identical manner except that we allow the probability of success at level i to diminish to $1 - n^{-1/2^{i-1}\alpha}$. For example, in level i, we do the resampling only $\log n/2^i$ times.

4.1.4 Partitioning segments into trapezoidal regions

We shall use a *locus-based* approach to solve this problem. This approach involves considering each query as a higher dimensional point and partitioning the underlying space into regions providing the same answer. Thus the query problem is reduced to a point location problem, given sufficient preprocessing time and space. The problem at hand involves preprocessing the trapezoidal subdivisions (induced by the sample) in such a manner that given the end-points of any segment we should be able to list the regions it intersects in

$O(\log n)$ time using $\lceil k/\log n \rceil$ processors where k is the number of regions that it intersects. We shall show that the preprocessing for n segments can be done in $O(\log n)$ time using $O(n^c)$ processors, where c is a fixed constant. Thus we can choose any sample of size less than $n^{1/c}$.

Since the input segments are non-intersecting, these can only extend between regions where there exists a 'clear-path' which does not intersect any other sample segment. It must also be clear that there can exist more than such 'clear-path' between two regions. Our objective is to partition the plane into regions (equivalent classes), such that given any fixed pair of such regions (where the end-points of a segment lies), the regions that the segment intersects can be pre-determined. The boundaries of these 'clear-paths' are straight-lines joining vertices of the trapezoidal subdivisions and the equivalent regions are intersections of the half-spaces defined by these boundaries. Since we need a fast preprocessing procedure, we shall settle for a finer partition of space (i.e. more than one pair of partitioned region may intersect exactly the same set of trapezoids). If n is the size of the trapezoidal regions, there are $O(n)$ vertices which gives rise to at most $O(n^2)$ boundary conditions which are straight lines joining every pair of vertices. Some of them may intersect sampled segments and hence are not boundaries of 'clear-paths' but it doesn't affect our procedure since they would only make the partition more fine (See Figure 4.3 for an illustration). These $O(n^2)$ lines can intersect in $O(n^4)$ vertices, giving rise to a $O(n^4)$ regions in the partition. This implies that there are $O(n^8)$ possible pairs of regions where end-points of a segment could lie and one can pre-compute for each such pair which trapezoids the corresponding segment intersects using $O(n^9)$ processors in $O(\log n)$ time.

For the point location problem, we use a pre-processing scheme similar to

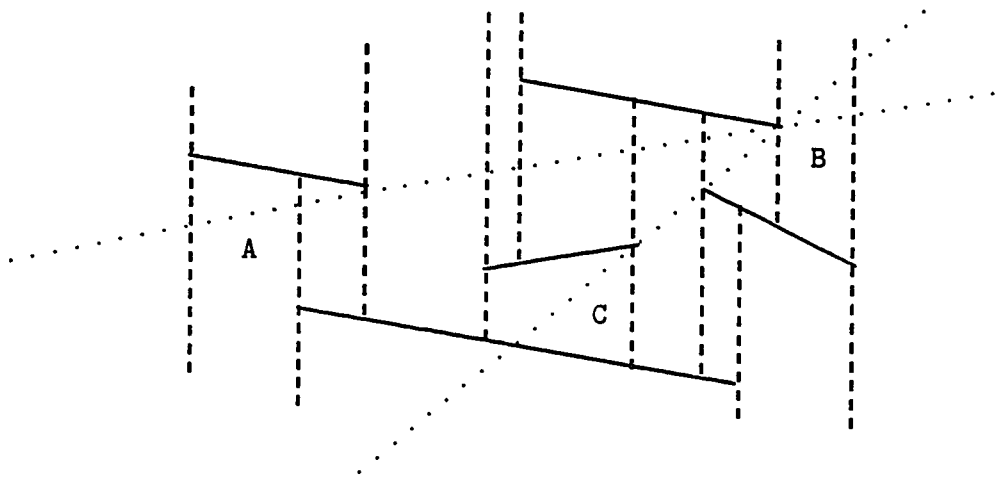

Figure 4.3: There is a clear path between regions A and B where as there cannot be a segment with end-points in regions C and B. This redundant partitioning does not affect the asymptotic complexity of the algorithm.

Dobkin and Lipton [33]. The following result is a straightforward consequence of their paper

Lemma 4.4 *Given a set of m line segments in a plane, it is possible to pre-process them in* $O(\log m)$ *time time using* $O(m^3)$ *processors such that point-location for any query point can be done in* $O(\log m)$ *time. The space needed is* $O(m^3)$.

Proof: We merely mimic the sequential algorithm. After computing all possible pairs of intersections in O(1) time using $O(m^2)$ processors, we project the intersection on the x-axis and for each interval we compute the total ordering of the lines in $O(\log m)$ time using m processors for each of the intervals. Thus the space required is $O(m^3)$. □

Due to this pre-processing there is an increase in the number of regions

from what we mentioned in the previous paragraph. Each region is now a trapezoid (can also be a triangle) and there are $O(n^6)$ regions. Accordingly we require $O(n^{13})$ processors to pre-compute the possible intersections. Since each region is a trapezoid, we choose a sample point in each trapezoid to do the pre-computing part. We make a table corresponding to each pair of regions containing the trapezoids the corresponding segment intersects. For any input segment, we first do the point location for its end-points and then perform another search on this ordered pair of regions. Clearly the whole procedure is a constant number of binary searches and takes $O(\log n)$ time. We can also store the number of intersecting trapezoids for each entry, so that we can allocate the required number of processors corresponding to each segment to list the intersecting trapezoids (using a prefix sum).

Thus we can state the main result of this section as follows :

Theorem 4.1 *Algorithm Nested-Plane-Sweep-tree runs in* $\tilde{O}(\log n)$ *time and* $O(n \log\log n)$ *space using* $O(n)$ *processors in a CREW PRAM model.*

Proof From the previous discussion and our earlier description of the algorithm as a process tree, we can choose appropriate constants to satisfy the premise of Theorem 3.1 i.e.

$$Prob[T_i \geq k\epsilon_0^i c\alpha \log n] \leq 2^{-c\alpha\epsilon_0^i \log n}.$$

The time-bound follows.

To satisfy the processor bounds, we make the following modification in the algorithm. After partitioning the segments into the trapezoidal regions (using the procedure in section 3.4), we group the segments into two categories:

(a) Segments (part-segments) that have at least one end-point in the region

(b) Segments that span the region (horizontally)

Notice that number of segments of type (a) is less than $2n$ and the segments of type (b) in a region i, S^i can be completely ordered (with respect to y-coordinate). While performing *multilocation*, a straight-forward binary search suffices for ordered segments in S_i. Consequently, we do not further pre-process the segments in S_i, i.e. we can leave them out from further recursive calls. Thus, the total size of the subproblems at any level of the recursive call is no more than $2n$. Processor allocation is achieved by simply setting processors in a region equal to the number of end-points lying in it. This shows that the algorithm does not need more than $O(n)$ processors. □

4.2 Applications Of Nested-Plane-Sweep Tree

In this section, we exploit the data structure that we constructed in the previous section to improve the running time of various algorithms like triangulation, intersection detection, 3-D maxima, 2-D dominance counting, and visibility from a point.

Lemma 4.5 *The nested-plane-sweep-tree constructed in the previous section can be used to multilocate any query point p in* $\tilde{O}(\log n)$ *sequential time.*

Proof: From Fact 4.1 multilocation in an $O(n_i)$ node plane-sweep tree can be performed in $O(\log n_i)$ serial time. Thus, the expected time for multilocation in the nested-plane sweep tree obeys the following recurrence relation:

$$\bar{T}(n) = O(\log n) + (1 - n^{-\beta})\bar{T}(\sqrt{n}\log n) + (n^{-\beta})\bar{T}(n)$$

giving $\bar{T}(n) = O(\log n)$ which is the multilocation time for each query point p using one processor. Equivalently, multilocation for $O(n)$ query points can be performed in $O(\log n)$ expected time using $O(n)$ processors. The worst case time bounds can be derived using techniques similar to the proof of Theorem 2. □

4.2.1 Trapezoidal decomposition and triangulation

Definition *Trapezoidal Decomposition* Let $P = \{v_1, v_2, ..., v_n\}$ be a simple polygon, where v_i's denote the vertices of P, and are listed so that the interior of P is to the left of the walk $v_1v_2..v_n$. For any vertex v_i of P a trapezoidal edge for v_i is an edge of P, which is directly above or below v_i, such that the vertical line segment from v_i to this edge is interior to P. A vertex can have 0,1 or 2 such edges. Trapezoidal decomposition of a polygon P is to find the trapezoidal edges for each vertex of P.

Lemma 4.6 *Given a simple polygon P, a trapezoidal decomposition of P can be constructed in $\tilde{O}(\log n)$ time using $O(n)$ processors and $O(n \log n)$ space in a CREW PRAM model.*

Proof : The trapezoidal decomposition can be done in two distinct stages (see Atallah and Goodrich [8] for details). In the preprocessing part, a nested plane-sweep tree is constructed on the edges of the simple polygon P. From theorem 2 this runs in $O(\log n)$ time with very high probability. Subsequently, all the vertices of the polygon are multilocated simultaneously using $O(n)$ processors which takes $O(\log n)$ time with very high likelihood (from Lemma 4.5). For

each point, it takes a constant time to determine if the vertical line intersecting the trapezoidal edge is within the polygon P using one processor per point. □

Fact 4.3 *(Atallah and Goodrich [8]): Given a one-sided monotone polygon with n vertices, P can be triangulated in $O(\log n)$ time and $O(n)$ space using $O(n)$ processors in a CREW PRAM model.*

Our next theorem is a direct consequence of this fact and Lemma 4.6.

Theorem 4.2 *Given a simple polygon with n vertices, P can be triangulated in $\tilde{O}(\log n)$ time and $O(n \log n)$ space using $O(n)$ processors in a CREW PRAM model.*

Proof: The algorithm consists of three phases:
(1) We first build the Nested Plane-Sweep-Tree on the edges of the polygon. With very high probability this can be done in $O(\log n)$ time and $O(n \log n)$ space (from theorem 2).
(2) The simple polygon is decomposed into one-sided monotone polygons by using trapezoidal decomposition (see Atallah and Goodrich [8] for details). Thus, from lemma 7 the expected running time for this phase is $O(\log n)$.
(3) Each of the one-sided monotone polygons can be triangulated in $O(\log n)$ time and $O(n)$ space from Fact 4.3.
The proof of the theorem follows directly from the above steps. □

4.2.2 Visibility from a point

Definition *Visibility from a point*: Given a set of line segments $S = \{s_1, s_2 ... s_n\}$, which do not intersect, except possibly at end-points, and a point p, determine the part of the plane which is visible from p when every segment s_i is opaque.

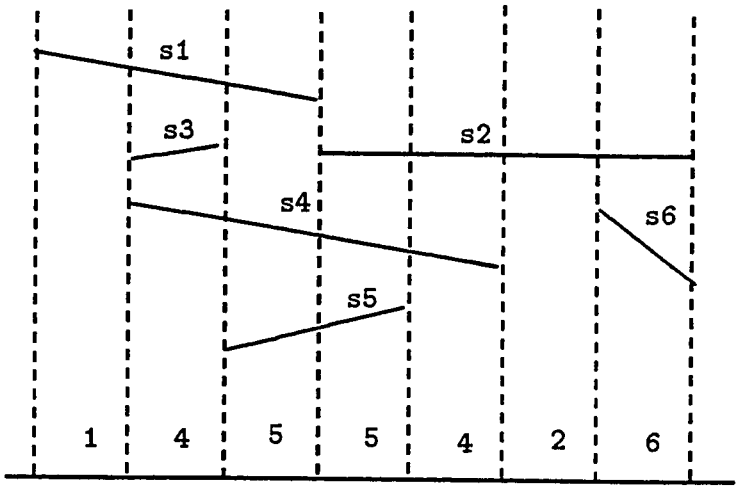

Figure 4.4: The intervals on the x-axis are labeled by the segment visible in that interval.

We shall assume the point p to be at negative infinity below all segments. The algorithm that we present below can appropriately modified for any general point. Notice that now (with the assumption) the problem is to determine the projections of segments on the x-axis, such that in case of overlap the segment closer to x axis has precedence (see Figure 4.4).

The solution to this problem consists of a sequence of points $\{p_1, p_2 ... p_{2n}\}$ such that two consecutive points p_i, p_{i+1} define a segment visible in the interval $[x(p_i), x(p_{i+1})]$. An important property of this problem is that the visibility is a constant function between the endpoints of the segments, i.e. for all points between two endpoints (projection of the endpoints on the x axis), the same segment will be visible. We can refer to the endpoints as *critical* points where the visibility function defined as $Vis(x)$ may change its value. This property enables us to solve the problem efficiently. Since $Vis(x)$ is constant between any two consecutive endpoints, we choose a point p which lies in this inter-

val below all the segments and draw a vertical line through p. The segment which intersects the vertical line first (from below) is the visible segment in the corresponding interval. Formally, the algorithm can be described as follows:

Algorithm *Visibility*

(1) Given a set S of non-intersecting segments, sort their endpoints on the x-coordinate. This can be done in $O(\log n)$ time using Cole's [23] parallel mergesort algorithm which does not have a large constant as does the AKS algorithm.

(2) For each of the $2n-1$ bounded intervals, choose a point within the interval. A possible choice is the midpoint of an interval. Let us denote them by $\{p_1, p_2, \cdots p_{2n-1}\}$.

(3) Construct a Nested Plane-Sweep tree on the segments of set S.

(4) Multilocate the $2n-1$ midpoints on this tree giving us ordered pairs of the form (p_i, s_j) which carries the required visibility information for the interval $[x_i, x_{i+1}]$.

The following theorem is a direct consequence of our previous discussion:

Theorem 4.3 *Given a set S of segments and a point p, the visibility of the plane induced by this set of segments from this point can be computed in $\tilde{O}(\log n)$ time using $O(n)$ processors in a CREW PRAM.*

4.3 An optimal $O(\log n)$ time algorithm for 3-D Convex hulls

In this section we shall describe an algorithm to construct the intersection of half-spaces in three dimensions. We assume that an internal point is known in the intersection and that the planes bounding the half-spaces are in general position. This implies that no four planes share a common intersection point. The former assumption is necessary for the algorithm to succeed and so this algorithm cannot be used to determine a feasible point in the intersection. However for constructing the convex hull of point sites in three dimensions, we transform the problem into the dual space with respect to an internal point. The latter assumption regarding non-degeneracy is used for keeping the arguments simple noting that there are perturbation techniques to ensure this condition.

4.3.1 Useful results

In the remainder of the section we shall assume that the half-spaces are described as inequalities of the form $ax + by + cz + d \geq 0$. The output is the list of vertices of the output hull defined by the 3-tuples of the three intersecting planes defining the vertex. The edges are those pairs of vertices which have two common planes in the tuples. A face is defined by all tuples which have one common plane. A tuple can be written in 6 ways (permutation of the 3 planes) and thus sorting them (all the 6 possible representation of the vertices) would transform the hull into one of these representations.

The following observation is useful for constructing the intersection of a random subset of half-spaces that is used to split up the problem evenly.

Lemma 4.7 *The intersection of a given set of n half-spaces can be computed in $O(\log n)$ time using n^4 processors in a CREW PRAM model.*

Proof: Assuming non-degeneracy (i.e. no 4 planes intersect at a common point), there are $O(n^3)$ candidate vertices for vertices of the convex hull (of the intersection). For each vertex, test whether it is a vertex of the convex hull by checking if it satisfies all the equations defining the half-spaces. This can be done trivially in $O(\log n)$ time using n processors for each candidate point. Only the vertices would survive. Determine the faces of the convex hull by identifying planes that contain 3 vertices of the intersection. □

Lemma 4.8 *Given a set of n half-spaces, it is possible to compute their intersection in $O(\log^3 n)$ time using n processors in a CREW PRAM model.*

Proof: Follows immediately from Aggarwal et al. [4].

This result is useful to stop the recursion at a level when the problem size is small (typically $O(\log^k n)$ for some integer k) and solve the problem directly. Any polylog-time algorithm using a linear number of processors would again suffice for our purpose. At this stage the problem size is so small that using a sub-optimal algorithm will not affect the asymptotic complexity of the algorithm.

4.3.2 A parallel algorithm

We shall assume for the time being that we know a point p^* in the intersection of the n half-spaces and later show how to determine such a point efficiently for the applications that we are interested in. Using a random subset of S, we split the original problem *evenly* into smaller sized problems and then apply

the algorithm recursively to each of the problems. By using a random subset of size n^ϵ, $(0 < \epsilon < 1)$ we split up the problem into sub-problems of expected size $n^{1-\epsilon}$. The generic divide-and-conquer algorithm assumes the following form in the context of intersecting half-spaces.

Algorithm

Input: A set S of n half-spaces H_1, $H_2, \ldots H_n$.

Output: The output convex hull C which is intersection of the n half-spaces.

(1) Choose a random subset R $\subset$ S of half-spaces such that $|R| = n^\epsilon$ (for some ϵ, $0 < \epsilon < 1$ that we shall determine during the course of analysis).

2) Find the intersection of the R half-spaces and again without loss of generality assume that there is no degeneracy i.e. each vertex is the intersection of exactly three planes. Take a fixed plane and cut up each face of the polyhedron with parallel translates of this plane passing through the vertices. Thus each face is a trapezoid. Further, partition each trapezoid with a diagonal so that each face is triangular. For a face F_i consisting of vertices x_i, y_i, z_i consider the cone C_i formed by p^* as the apex and F_i as the base. Let C_R denote the number of cones. Note that $C_R \leq |R|$.

3) For the $S - R$ remaining half-spaces (which are actually equations of planes) find the intersection of the planes with the cones. A plane may intersect more than one cone. The intersection of the S half spaces is the union of the intersection of the half-spaces intersecting a cone (over all cones). That is, C is $\cup_{i=1}^{C_R} I_i$ where I_i is the intersection of all half-spaces formed by $C_i \cap \{H^j\}$ for all j.

4) If the number of planes intersecting the cone is more than a pre-determined

threshold apply step 1-3 recursively to this set of set of half-spaces else solve the problem directly (using Lemma 2.4).

end

4.3.3 Probabilistic lemmas

A crucial part of the analysis rests on showing that a random subset R can be chosen efficiently in the first step of the algorithm that divides the the problem into almost equal sized sub-problems. In addition we have to show that the union of the sub-problems is almost equal to the complexity of the original problem at every stage of the recursive calls. The following result follows from Clarkson (Clarkson [21], Corollary 4.3) for any random $R \subset S$ with $|R| = r$. This can also be proved directly using arguments analogous to Claim 3..4

Lemma 4.9 *Let X_i denote the set of planes intersecting cone C_i (using the same terminology as in step 2 of the algorithm). Then the following conditions hold with with probability at least 1/2*

(i) $\sum_{i=1}^{C_R} |X_i| \leq k_{total}(n/r) \cdot E(C_R)$

and

(ii) $max_i|X_i — \leq k_{max}(n/r) \cdot \log r$

where k_{total} and k_{max} are constants and C_R is defined previously.

Any subset of the input that satisfies the above conditions for some fixed constants is defined to be 'good' and otherwise 'bad'. A direct consequence of the lemma is that we can divide up the problem into almost equal size sub-problems, such that the increase in the original problem size can be bounded

by at most a constant multiplicative factor of k_{max}. Using **Polling** we can select a sample that is 'good' with high probability.

4.3.4 Finding intersections quickly

We now focus on describing a procedure to find the intersection of planes with each of the cones, C_i. Notice that a plane may intersect more than one cone which rules out detecting the intersections sequentially. That is, if a plane intersects n^δ cones ($\delta > 0$), we cannot afford to detect them one after the other since we are looking for an $O(\log n)$ time procedure. In the sequential case, Clarkson and Shor's [22] randomized incremental constructions give optimal expected time bounds that cannot be applied in our case.

We shall use a *locus-based* approach to solve this problem. This approach involves considering each query as a higher-dimensional point and partitioning the underlying space into regions providing the same answer. Thus any query problem can be reduced to a point location problem given sufficient preprocessing time and space. In our case, we have to pre-process the convex hull of the sampled half-spaces in such a way that given any plane, we should be able to report the cones that it intersects in $O(\log n)$ time using at most k processors where k is the number of intersections. We shall show that the pre-processing for a convex hull of $O(n)$ size can be done in $O(\log n)$ parallel time using $O(n^c)$ processors, where c is a fixed constant. Thus we can choose any sample of size less than $n^{1/c}$ since we have n processors.

Given a convex polyhedron in 3-D of size $O(n)$ along with an internal point which is the apex of the cones, there can exist only a polynomial (in n) number of combinatorially distinct possibilities of the way any given plane

can intersect the cones. This can be seen from the following simple argument. Given any plane that intersects the polyhedron, we can perturb the plane without changing the cones it intersects so long as it remains within a fixed set of bounding vertices. Figure 1 illustrates the situation for a two-dimensional case. If we consider an equivalence relation where two lines are equivalent if and only if they intersect the same sets of cones then the equivalence classes correspond to the cells in the arrangement $\mathcal{A}(H)$ where H = $\{\mathcal{D}(p) : p$ is a vertex of the convex hull or internal point and $\mathcal{D}$ is a dual transform $\}$ (see [37] for more details). Given any query line l, the cones that it intersects is defined by the partition of $\mathcal{A}(H)$ that $\mathcal{D}(l)$ belongs to. This observation can be extended to hold for any dimension; in our case three. If we consider the partitions of the three-space induced by the intersections of the constraining half-spaces, these are equivalent classes with respect to the cones they intersect. Notice that even if this partitioning may not be minimal but it suffices for our purpose. All that remains to be done is pre-compute for each of these regions the cones that the corresponding planes would intersect so that for any query plane in the same equivalence class we can list off the intersecting planes by a table look-up.

For the point-location problem, we use a pre-processing scheme due to Dobkin and Lipton [33] because of the ease in parallelization. The following is a fairly straight-forward extension of their method

Lemma 4.10 *For any set of m planes in E^3, it is possible to pre-process them in $O(\log m)$ time using $O(m^7)$ processors, such that point-location for an arbitrary query point can be done in $O(\log m)$ time. The space required is $O(m^7)$.*

Proof: Find the pairwise intersections of the given set of planes (there are

$O(m^2)$ of them). Project the resulting lines on a plane which is not normal to any of the lines. Find the pairwise intersections of the straight lines and consider their projection on the x-axis. There are $O(m^4)$ intervals induced by these. For each of these intervals, order the straight-lines by sorting. This can be done in $O(\log m)$ time using $O(m^2)$ processors for each of the $O(m^4)$ intervals. There are now $O(m^6)$ trapezoidal regions. For each of these, order the planes for binary-search by sorting which are totally ordered in these subdivisions. This can be done in $O(\log m)$ time using $O(m^7)$ processors. The subdivisions induced by this pre-processing are homeomorphic to a 3-cube, so that given any query point it can be located in such a subdivision with 3 binary searches. □

For each of the subdivisions in 3-space, we can precompute the cones that the corresponding plane intersects using $O(n^8)$ processors. Note that these subdivisions are finer than the minimal equivalence classes, i.e. more than one subdivisions could have the same set of intersecting cones. We also store the number of intersecting cones for each of the subdivisions so that while listing the number of cones each query plane intersects we can do the processor allocation easily in $O(\log n)$ time using a prefix computation. By choosing less than $n^{1/8}$ samples, we can complete the entire preprocessing in the required time and processor bounds.

We summarize our conclusion in this section as follows

Lemma 4.11 *Step three of the algorithm uses $O(n\log n)$ space and terminates in $O(\log n)$ time using n processors in a CREW PRAM model.*

In the course of the entire algorithm, the concurrent reads are utilized only during the binary searches.

4.3.5 Controlling the size of subproblems and processor allocation

Even though we have shown that most of the algorithm works out as desired, there is more that needs to be covered to complete the analysis. From lemma 4, we know that the size of the problem can increase by a constant factor at each level and we wish to avoid this happen over $O(\log \log n)$ levels, which would increase the number of processors required by a polylog factor.

For this we need to quickly identify the redundant planes that do not contribute to the output complexity and eliminate them from further recursive calls. This enables us to get a global bound on the total size of the subproblems at any stage which we shall show to be linear in the input plus the output size. More specifically, we allocate the processors recursively to the cones such that the number of processors is proportional to the number of output vertices in that cone, thereby bounding the number of processors to be $O(n)$. The details of the procedure is described below.

After we have found the planes intersecting a particular cone, we categorize them as following:

(a) The planes that are completely occluded by another plane in the cone and hence these cannot be a part of the output in the cone

(b) Planes that are occluded because of more than one other plane in the cone i.e. there is no one plane that completely occludes them.

(c) Planes that contribute to an edge without an end-point i.e. the end-points lie in some other cones.

(d) Planes that do contribute to a vertex in the cone

To eliminate planes of type (a), we use a variant of the 3-D maxima algorithm.

The *3-D* maxima problem is defined as:

> Given a set S of n points in a three-dimensional space, determine all points p in S such that no other point of S has x,y and z coordinates that simultaneously exceed the corresponding coordinates of p.

Since cones have a triangular base there are 3 edges that join it to the apex p^*. We sort the intersections of the planes with an edge in increasing distances from the apex. We repeat this for all the three edges. Call these three edges X, Y, Z and denote the intersection of a plane h_i as X_i , Y_i , Z_i and the ranks in the sorted list as r(X_i), r(Y_i) and r(Z_i).

Observation 1: If a plane A is occluded completely by another plane B if and only if it is dominated on its ranks of intersection on all the three edges by plane B.

This gives us an effective strategy for eliminating planes of type (a) by identifying the complement of the set of the maximal elements, where we use the ranks of the intersection on the three edges as the order relation. Using the $O(\log n)$ time n processors algorithm of [7], we can do this in $O(\log n)$ time.

To identify planes of type (b) (c) and (d) we construct the intersection of the 3-D convex hull C with each of the three faces of the cone. These are intersections of the faces with C that are 2-D convex hulls. These will be referred to as *contours* for the following discussion. The *contours* can be computed in $O(\log n)$ time with n processors using any of the optimal 2-D convex hull algorithms. These convex *contours* on the three faces are a part of the output and any plane that appears on this contour is a part of the final output. Consequently, a plane of type (b) cannot be a part of this contour. Unfortunately, there can be planes that are part of the output but are not part of any *contour* (consider

a plane that chops off a cap of the hull within the cone). For the time being let us focus on only those planes that show up in the contours and consider the 3-D convex hull formed only by these planes within a cone. We shall refer to such a 3-D hull as a *skeletal-hull.* We now make following observation

Observation 2: Any plane that is is not a part of the contour on any face can intersect at most one *skeletal hull.*

This follows from convexity. Notice that such planes are not necessarily a part of the output but we are not aiming for an output sensitive algorithm. The previous observation guarantees that if a plane is not a part of $\mathcal{C}$ it will not survive in more that one cone when the algorithm is called recursively in the cones. The planes that do not intersect the *skeletal-hull* cannot be a part of $\mathcal{C}$ within the cone.

A plane that does not contribute to any vertex of the convex hull and is a part of the contour is incident on exactly two faces (intersection of the edge with the cone) and hence can be identified quickly using sorting.

The objective of the above procedure is to eliminate some of the planes that do not contribute to the output and ensure that going into any recursive call, the sum of the subproblems is less than the output size. We define the output size of the 3-D convex hull to be $3|V|$ where V is the number of vertices of the convex-hull. Since the surface of the convex-hull is a planar map, we can use Euler's equation to show that $|V| = 2|F| - 2$ where F is the set of faces in the hull. Since $|F| \leq n$, $|V| \leq 2n - 2$. This calculation is done using the assumption that each vertex is of degree 3 (non=degeneracy condition). Let the number of processors be $6|V|$. We distribute the processors among the subproblems depending on the output size. For a cone C_i, the output can be

bound by the following:

Claim 4.1 *The output size of a cone is bounded by* $3n_i + 2n_i^e - 2$ *where* n_i *is the number of planes in the contour contributing at least one vertex and* n_i^e *is the number of planes of type (b) and (d).*

Proof: Let e_1 denote the number of edges of $\mathcal{C}$ that intersect the contour and contribute a vertex within the cone (the vertices of the contour are these edges). Let e_2 be the number of edges which lie within the cone (including both end-points). Let v be the number of vertices of $\mathcal{C}$ within the cone (these have degree 3), then $e_1 + 2e_2 = 3v$ or

$$e_1 + e_2 = \frac{3v + n_i}{2}$$

as $e_1 = n_i$. Consider the planar map of the polyhedron formed by n_i and n_i^e planes and a 'base' face by 'flattening' the contour. The number of edges on the contour equals the number of vertices on the contour. Then by applying Euler's formula $|V| = n_i + 2n_f - 2$ where n_f is the number of faces that show up in the cone but are not a part of the contour. Since n_f can be bounded by n_i^e the claim is proved.

The processor allocation strategy is to simply allocate this number of processors to the sub-problem (in the cone). The total number of vertices over all the cones is bound by the output size and hence we have sufficient number of processors.

We shall now describe a procedure to construct the *skeletal-hull* within a cone and preprocess the *skeletal-hull* such that queries of the kind plane-polyhedra intersection detection can be answered quickly. The latter part can be done efficiently using a hierarchical polyhedra decomposition scheme due to

Dobkin and Kirkpatrick [32]. The construction of the hierarchical representation can be done in $\tilde{O}(\log n)$ time using an algorithm of described in Chapter 2 (also discovered independently by Dadoun and Kirkpatrick [30] but the analysis given in their paper is not sufficient for our purposes). Given this representation, the plane-polyhedra intersection detection query can be answered in $O(\log n)$ sequential time (Kirkpatrick [51]).

We shall now discuss how to construct the *skeletal-hulls* quickly. Although the *skeletal-hulls* are themselves 3-D convex hulls they have a much simpler structure. More specifically, they have the following property: all faces are unbounded (i.e. they are part of the *contours*. This implies that, if we construct them recursively using the same algorithm, we do not have to worry about case (b) since all planes that are part of the output will show up in the *contours* and this holds for any level of the recursive call. From the analysis given in the next sub-section the *skeletal-hulls* can be constructed in $\tilde{O}(\log n)$ time using a linear number of processors. The reader should convince himself that there is no circularity of arguments here. One way to look at the problem is the following : assuming that case (b) doesn't arise (i.e. all planes that are part of the output show up in the *contours*), the algorithm terminates in $\tilde{O}(\log n)$ time using a linear number of processors. So after having constructed the *skeletal-hull* for the cone, the redundant planes are quickly eliminated using the procedure outlined in the previous paragraph. Subsequently, the algorithm is called recursively on the cone - this time to build the actual hull (as opposed to the *skeletal-hull*).

4.3.6 Final analysis

Consider the algorithm as a tree where each node corresponds to a procedure and the children of a node representing processes corresponding to the recursive calls made by the procedure. Then the running time of the algorithm corresponds to a worst-case sequence of nested procedure calls along any path in this tree from the root to a leaf node. This process tree corresponding to the algorithm has the following property. A process at level i for $1< i < O(\log\log n)$, has size $O(n^{(10/9)^{-i}})$ and the process terminates in time $O(\log n^{(9/10)^i})$ $(= (9/10)^i O(\log n))$ with probability greater than 1 - $1/n^{(9/10)^i}$. From Theorem 3.1, any nested sequence of recursive calls exceeds time $c\gamma logn$ with probability less than $1/n^{\gamma}$ for any $\gamma > 1$. It follows that all the leaf processes and hence the algorithm are completed within the same time with high likelihood. The space used is $O(n)$ at step 3 of each recursive level giving a total bound of $O(n\log\log n)$ for all the $O(\log\log n)$ recursive levels of the algorithm. This proves the main result of the section. □

Corollary 4.1 *The following problems can be solved in $\tilde{O}(logn)$ time using n processors in a CREW PRAM*

(i) Convex hull of a set of points in 3-D

(ii) Voronoi diagram of point-sites in plane

(iii) All-points nearest neighbor

(iv) Euclidean minimal spanning tree

Proof: (i) follows immediately because of well-known reduction of convex hulls to intersection of half-spaces. To determine an internal point p^* in the

intersection, we can determine an internal point of the convex hull and use it as the origin for the duality transform. The origin is known to be contained in the intersection of the half-spaces.

For (ii), given a set of n points in the plane we apply an inversive transformation given by Brown [14] to the input points to transform the problem into finding the convex hull of n points in 3-space.

(iii) can be obtained in $O(\log n)$ time from the Voronoi-diagram.

(iv) can be obtained by running a minimal-spanning tree algorithm on the edges of Delaunay triangulation which is the dual graph of the Voronoi diagram. This algorithm uses the stronger Priority CRCW model as in Awerbach and Shiloach [9] □

Chapter 5

PROBABILISTIC INEQUALITIES AND MINIMIZING RANDOM BITS

In this chapter we shall look more carefully at the probabilistic inequalities that have been used to obtain bounds for the resources used by the parallel algorithms. In doing so, we shall focus on the issue of using random numbers (or random bits). It is known that obtaining pure random sequences is a formidable problem in practice and often one can only have access to sequences which are *pseudo-random.* In the following discussion we shall not concern ourselves with the generation of random sequence; instead we shall try to minimize the number of *purely* random bits used by the algorithm while maintaining the same asymptotic resource bounds.

The objective behind this approach is primarily two-fold: since purely random bits are expensive, we should try to minimize their use along with other resources like time and number of processors. Secondly, there is a growing effort towards de-randomization of randomized algorithms where the complexity of the de-randomized deterministic algorithm depend on the number of random bits used by the algorithm. We emphasize that de-randomization inevitably

leads to the loss in efficiency of the algorithms and is not very pertinent in our context where the objective is to design optimal algorithms. However, it can be an important aspect in the study of relationship between the classes $\mathcal{NC}$ and $\mathcal{RNC}$.

We shall show that analyzing algorithms with a slightly different approach may lead to considerable reduction in the number of random bits. In this respect we shall mainly compare using Chernoff bounds with Chebychev-like inequalities.

5.1 Chernoff bounds

Chernoff bounds which (for a discrete distribution) can be stated as

> Prob[A $\geq$ x] $\leq$ $z^{-x}G_A(z)$ where $G_A(z)$ is the probability generating function.

To minimize the bound we substitute z = z_o that minimizes the right side expression.

We say a random variable X upper-bounds another random variable Y (equivalently Y lower bounds X) if for all x such that $0 \leq x \leq 1$, Prob($X \leq x$) $\leq$ Prob($Y \leq x$).

A Bernoulli trial is an experiment with two possible outcomes namely, success and failure. The probability of success is p.

A binomial variable X with parameters (n, p) is the number of successes in n independent Bernoulli trials, the probability of success in each trial being p. The *probability mass function* of X can be easily seen to be

$$\text{Prob}(X \leq x) = \sum_{k=0}^{x} \binom{n}{k} p^k (1-p)^{n-k}$$

The tail end of the Binomial distribution can be bounded by *Chernoff* bounds. In particular the following approximations due to Angluin and Valiant are frequently used:

$$\text{Prob}(X \geq m) \leq \left(\frac{np}{m}\right)^m e^{m-np} \quad (1)$$

$$\text{Prob}(X \leq m) \leq \left(\frac{np}{m}\right)^m e^{-np+m} \quad (2)$$

$$\text{Prob}(X \leq (1-\epsilon)pn) \leq exp(-\epsilon^2 np/2) \quad (3)$$

$$\text{Prob}(X \geq (1+\epsilon)np) \leq exp(-\epsilon^2 np/3) \quad (4)$$

for all $0 < \epsilon < 1$.

The probability mass function of a modified geometric random variable X is specified by $p_X(i) = p(1-p)^i$ for i = 0,1,2 .. The probability generating function of a modified geometric random variable X is given by $G_Y(z) = \frac{p}{1-(1-p)z}$ where p is the probability of success in each individual trial. Recall that, Y is the number of trials before we have a success. Thus the generating function of sum of logn independent identical modified geometric random variable is given by

$$G_Y(z)\left(\frac{p}{1-(1-p)z}\right)^{\log n}$$

Using Chernoff bounds and minimizing the bounds by differentiating we obtain

$$\text{Prob}[Y \geq \text{clogn}] \leq n^{-c\log_2 \frac{2c}{c+1}} \text{ for p} = 1/2$$

For $c > 2$, we can write it as $\text{Prob}[Y \geq \text{clogn}] \leq n^{-\alpha c}$ where $\alpha > 0$. (5)

The above result can be directly applied to yield better constants for the running time of quicksort (described in section 2.1.2).

5.2 Chebychev's inequality

The commonly used form of Chebychev's inequality has the form:

$$Prob[(|X - \mu| \geq t)] \leq \frac{\sigma^2}{t^2}$$

.

This simple fact was exploited by Chor and Goldreich [17] for their 2-point sampling theorem where they consider the following scenario. To determine if a property P holds for x (for example P can be the primality property), we often use a function $f(x,r)$ which satisfies the condition that if P holds for x then $f(x,r)$ is 1 with probability at least $1/2$ whereas if P does not hold for x then $f(x,r)$ is 0. Here r is a witness which is a random number in a certain range. This implies that if $f(x,r)$ is 1, then repeating the experiment for t *independent* random witnesses decreases the failure probability (of incorrectly categorizing x) to 2^{-t}. Instead of choosing t independent witnesses, if we choose t witnesses which are pairwise independent then we can analyze the probability of error as follows. Assume that it t trials $f(x,r_i)$ is 0 for all the trials and P(x) is true. Let $Y = \sum_i f(x,r_i)$. Then $E[Y] \geq t/2$ and the variance $\sigma_Y \leq \sqrt{t}/2$. Then $Prob[Y = 0] \leq Prob[|Y - E[Y]| \geq t/2$ which is less than $1/t$ from Chebychev's inequality.

The t *pseudorandom* numbers can be generated from two purely random numbers by using the following scheme $r_i = a + bi$ where a and b are random numbers. Thus instead of the confidence bound of $1/4$ (for two numbers), we

can do much better i.e. $1/t$. Karloff and Raghavan [50] were able extend these techniques to show that Reischuk's sorting algorithm can be implemented in the same asymptotic bounds by using only $O(\log n)$ purely random bits (instead of the naive scheme requiring $O(\sqrt{n})$ random bits). Here we generalize their scheme to minimize the number of random bits used by the algorithms described in the previous chapter. For this we need to prove some preliminary results.

Definition: A family of random variables is called k-way independent if any subset of k variables are mutually independent.
Clearly a k-way independent family is l-way independent for any $l \leq k$.

Let p be a suitable prime number and choose k numbers a_i, $0 \leq i \leq k-1$ randomly from Z_p. Consider the numbers of the form $r_i = (\sum_{j=0}^{j=k-1} a_j \cdot i^j) \bmod p$ The following is a well known result for this class of *pseudo-random* number generators.

Fact 5.1 : *The numbers r_i's are uniformly distributed in Z_p and are also k-way independent.*

Fact 5.2 : *If X_i, $1 \leq i \leq n$ are n mutually independent random variables then*

$$E[\prod_{s=1}^{n} \phi_s(X_s)] = \prod_{s=1}^{n} E[\phi_s(X_s)]$$

We now describe a simple scheme to implement the algorithm for planar-point location using $\tilde{O}(\log n)$ purely random bits. Recall that the most crucial

step in the algorithm is to eliminate a large independent set of vertices in a bounded degree planar graph. Instead of assigning the tags male and female independently we can generate the tags using only d-way (d is the maximum degree of any vertex) independence. The probability that vertex gets eliminated is still the same (as in the mutually independent case) and hence the expected size of the set of vertices eliminated is larger than νn where $0 < \nu < 1$. The actual value of ν does not affect our analysis significantly. This implies that the expected number of vertices remaining is less than $(1 - \nu)n$. Let that be cn. From Markov's inequality the probability that the number of vertices exceed kcn is less than $1/k$ for some $k > 1$ and $kc < 1$. By choosing the tags independently in every phase it can be shown that the algorithm still terminates in $O(\log n)$ phases with very high probability. Since we are using a constant number of random bits in every phase the total number of random bits is $\tilde{O}(\log n)$.

Lemma 5.1 (Generalized Chebychev inequality)

$$Prob\{|X| \geq t\} \leq \frac{E(\phi(X))}{\phi(t)}$$

where ϕ is a non-negative monotonically increasing function.

Proof: Let $Prob[X = x_j] = f(x_j)$. Then

$$Prob\{|X| \geq t\} = \sum_{|x_j| \geq t} f(x_j)$$

$$\leq \sum \frac{\phi(|x_j|) \cdot f(x_j)}{\phi(t)}$$

$$\leq \frac{E(\phi(|X|))}{\phi(t)}$$

Lemma 5.2 *Let X be the sum of n 2k-way independent and identical Bernoulli random variables X_i, $1 \leq i \leq n$ each of which has a success probability p. Then for a fixed k (chosen independently of n), $Prob\{|X - \mu| \geq \mu\} \leq O(\frac{1}{\mu^k})$ where $\mu = np$ and $p \leq O(n^{-\beta})$ for some $0 < \beta < 1$.*

Proof: Consider the generalized Chebychev's inequality

$$Prob\{|X| \geq t\} \leq \frac{E(\phi(X)}{\phi(t)}$$

. Using $\phi(t) = t^{2k}$ and substituting $X - E[X]$ for X and setting t to be equal to μ, we get

$$Prob\{|X - E[X]| \geq E[X]\} \leq \frac{E[(X - E[X])^{2k}]}{E^{2k}[X]}$$

Let us focus on the numerator - We shall show that it is $O(\mu^k)$ and the Lemma follows. Since E[X] = $\sum_{i=1}^{n} E[X_i]$, we can write $(X - E[X])^{2k}$ as $(\sum_{i=1}^{n} X_i - E[X_i])^{2k}$

In the multinomial expansion, all the terms containing $X_i - E[X_i]$ (for any i) as a factor vanish because of the $2k$-way independence property and Fact 5.2. There are $\binom{n}{c} \cdot \binom{2k-1}{c-1}$ terms which have c distinct non-unit product terms of the form $(X_j - E[X_j])^i$ such that $i > 0$ and $\sum i = 2k$. Also note that $E[(X_j - E[X_j])^i] = (1-p)(-p)^i + p(1-p)^i$

We can factor out p^i so that we can write the coefficient of n^c as $p^{2k} \cdot f(p, c, k)$, where f is a function independent of n and can be absorbed in the big-O notation. From our observation about the first-order terms (which vanish), the maximum value of c is k. The numerator can be bound by the asymptotically

dominating term $O(n^k \cdot p^k) = O(\mu^k)$. Since the denominator is μ^{2k}, the lemma follows. □

Corollary 5.1

$$Prob\{|X - \mu| \geq a \cdot \mu\} \leq O(\frac{1}{a^k \cdot \mu^k})$$

where a is a constant between 0 and 1.

Proof: Set $t = a\mu$ in the previous lemma. □

5.2.1 Rederiving probabilistic bounds with fewer random bits

In order to limit the number of random bits, we shall rederive the some of the random-sampling bounds and the **Polling lemma** using the $2k$-way independent random variables. In particular we shall prove a slightly weaker bound than Lemma 3.1. We shall continue using the previous notations.

Lemma 5.3 *The probability that the maximum number of half-planes intersecting any sector exceeds $n^{1-\epsilon+\delta}$ is less than $n^{-k\delta+2}$ where $0 < \delta < \epsilon$.*

Proof: By randomly choosing n^ϵ half-planes, the expected number of half-planes chosen in the sample in a sector that has more than $n^{1-\epsilon+\delta}$ half-planes is greater than n^δ. Thus the probability that none of the half-planes were chosen in the sample is less than $n^{-k\delta}$ from Lemma 5.2. Summing over the n^2 sectors gives us the required result. □

Although this bound is somewhat weaker than Lemma 3.1, it still suffices to show that the process-tree (corresponding to the algorithm) has $O(\log \log n)$ depth (the constant is somewhat larger). Lemma 3.2 is not affected since it

uses only expectations. For probabilistic analysis of **Polling** we make use of the fact that if the expected number of half-planes in a sector is larger than n^β then Lemma 5.1 directly yields high probability bounds for some $0 < \beta < 1$ and choosing a sufficiently large value of k. If the mean is less than n^β, then for small β, $n^{\epsilon+\beta} = o(n)$.

An identical argument can be carried out for the other problems namely, trapezoidal decomposition and 3-D convex hulls. Notice that the Chernoff bounds that we were using earlier yielded much stronger bounds (of the order of 2^{-n^ϵ} instead of $1/n^c$) which were not required to prove Theorem 3.1. The ramification of using these bounds is that the constants associated with the running time for the same confidence bounds are much larger.

From here on we can directly use the scheme of Karloff and Raghavan. We need an extra $O(\log n)$ multiplicative factor of truly random bits for implementing Polling for which we need $O(\log n)$ independently chosen $2k$ random seeds of $O(\log n)$ bits each. We summarize as following:

Theorem 5.1 *The algorithms for 2-D convex hulls, 3-D convex hulls and trapezoidal decomposition runs in* $\tilde{O}\log n)$ *time using n processors and* $\tilde{O}(\log^2 n)$ *purely random bits.*

Note that the 3-D convex hull algorithm uses the point-location algorithm which requires an additional $O(\log n)$ bits.

Chapter 6

CONCLUSION

We have presented optimal parallel algorithms for a number of fundamental problems in computational geometry. The main theme was to demonstrate the usefulness of randomized techniques to derive these algorithms. Optimal deterministic algorithms for some of these problems were discovered around the same time. The cascaded-merging technique of Atallah, Cole and Goodrich has been used effectively for a number of problems. However, in a recent paper Cole and Goodrich [25] reiterated the difficulties in extending the cascaded-merge technique to parallel algorithms for Voronoi diagrams. Although it is not expedient to conjecture about the extent of applicability of their techniques to the 2-D Voronoi diagram problem, it is unlikely that such an algorithm will be simpler than the one presented.

A number of issues have been left open by this work. Perhaps the most important direction for further research is to investigate the extensions of our algorithms to interconnection networks. By well known general-purpose emulation schemes, all these algorithms can be implemented to run on butterfly (or a hypercube) network with a $O(\log n)$ multiplicative factor degradation in

time complexity. So the crucial question is if we can do better ? We conjecture that the algorithms for 2-D convex hull and trapezoidal decomposition can be implemented on the networks so as to execute in the same asymptotic time complexity as the PRAM model. Note that Reishchuk's algorithm was successfully extended to networks by Reif and Valiant to run in $O(\log n)$ time. In contrast Cole's mergesort algorithm (which is the crux of the cascaded-merge technique) seems prohibitively difficult to implement on the networks because of its liberal use of pointers. This is also a strong justification in favor of using randomized algorithms. Presently, the only known $O(\log n)$ time algorithm for the network model is a 2-D convex hull algorithm due to Miller and Stout [57].

A direction for research in PRAM models is to extend our algorithms to the less powerful EREW model. Although we have been able to avoid use of concurrent-reads in most of the algorithms (except 2-D linear programming) our algorithms use concurrent reads mainly for doing simultaneous searches on a common data-structure. One of the underlying problems is doing binary search optimally in an EREW model. Specifically, given a binary tree of depth $O(\log n)$ whose leaves represent certain intervals, and $O(n)$ keys with one processor per key, we would like to locate the key in the right interval in $O(\log n)$ time. This problem is trivial in a model allowing concurrent-reads.

A more theoretical issue is that of designing sub-logarithmic time algorithms. While $O(\log n)$ time is optimal for models without concurrent reads (this is a lower bound for computing OR of n bits in such a model [29]), there is a good likelihood that these algorithms can be improved to run in $o(\log n)$ time in a CRCW model. We believe that techniques similar to Rajasekaran and Reif [64] can be used to reduce the running time to $O(\frac{\log n}{\log \log n})$ time for most of the algorithms.

Finally, there is the issue of extending the methods developed in this thesis to other problems in computational geometry. Specifically, can these be used for all the problems tackled successfully by Clarkson. Note that we would like to do it without loss of efficiency i.e. $PT = O(Seq(n)$. Constructing arrangements of lines in two dimensions optimally in $O(\log n)$ time is an important problem in this category.

Bibliography

[1] J. Komlos A. Ajtai and E. Szemeredi. sorting in clogn parallel steps. *Combinatorica*, 3:1–19, 1983.

[2] L. Adleman and K. Manders. Reducibility, randomness and untractability. *Proc. 9th ACM STOC*, pages 151–163, 1977.

[3] A. Aggarwal and R. Anderson. A random nc algorithm for depth first search. *Proc of the 19th ACM STOC*, pages 325–334, 1987.

[4] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap. Parallel computational geometry. *Proc. of 25th Annual Symposium on Foundations of Computer Science*, pages 468 – 477, 1985. also appears in full version in Algorithmica, Vol. 3, No. 3, 1988, pp. 293-327.

[5] M. Ajtai and M. Ben-Or. A theorem on probabilistic constant depth computation. *Proc. of the 16th Annual STOC*, pages 471 – 474, 1984.

[6] R. Anderson and G. Miller. Deterministic parallel list ranking. *Manuscript*, 1988.

[7] M.J. Atallah, R. Cole, and M.T. Goodrich. Cascading divide-and-conquer: A technique for designing parallel algorithms. *Proc. of the 28th Annual Symposium on the Foundations of Computer Science*, pages 151 – 160, 1987.

[8] M.J. Atallah and M.T. Goodrich. Efficient plane sweeping in parallel. *Proc. of the 2nd ACM Symp on Comput. Geom*, pages 215 – 225, 1986.

[9] B. Awerbuch and Y. Shiloach. New connectivityh and msf algorithms for ultracomputer and pram. *Proc. of te Int'l Conf. on Parallel Processing*, pages 175–179, 1983.

[10] K. Batcher. Design of a massively parallel processor. *IEEE Transaction on Computers*, 29:836–844, 1980.

[11] P. Beame and J. Hastad. Optimal bounds for decision problems on crcw pram. *Proc. of the 19th Annual STOC*, pages 83 – 93, 1987.

[12] A. Borodin and J. Hopcroft. Routing, merging and sorting on parallel models of computation. *Journal of Computer and System Sciences*, 30(1):130 – 145, 1985.

[13] R. Brent. Parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21:201–206, 1974.

[14] K.Q. Brown. Voronoi diagram from convex hulls. *Informat. Process Lett.*, 9:223 – 228.

[15] S. Chandran. *Merging in Parallel Computational Geometry.* PhD thesis, University of Maryland, 1989.

[16] B. Chazelle and D. Dobkin. Intersection of convex objects in two and three dimensions. *J.A.C.M.*, 34(1):1–27, 1987.

[17] B. Chor and O. Goldreich. On the power of two-point sampling. *Journal of Complexity*, 5:96–106, 1989.

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

[19] K.L. Clarkson. A probabilistic algorithm for the post-office problem. *Proc of the 17th Annual SIGACT Symposium*, pages 174 – 184, 1985.

[20] K.L. Clarkson. New applications of random sampling in computational geometry. *Discrete and Computational Geometry*, pages 195 – 222, 1987.

[21] K.L. Clarkson. Applications of random sampling in computational geometry ii. *Proc of the 4th Annual ACM Symp on Computational Geometry*, pages 1 – 11, 1988.

[22] K.L. Clarkson and P. Shor. Algorithms for diametral pairs and convex hulls that are optimal, randomized and incremental. *Proc. of the 4th ACM Symp. on Computational Geometry*, 1988.

[23] R. Cole. Parallel merge sort. *Proc. of the 27th Annual IEEE Symp. on Foundations of Computer Science*, pages 511 – 516, 1986.

[24] R. Cole. An optimal efficient selection algorithm. *Information Processing Letters*, 26:285 – 299, 1987.

[25] R. Cole and M.T. Goodrich. Optimal parallel algorithms for polygon and point-set problems. *Proc. of the 4th ACM Symp. on Computational Geometry*, pages 201 – 210, 1988.

[26] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree and graph problems. *Proc. 27th IEEE Symp. on Foundations of computer Science*, pages 511 – 516, 1986.

[27] R. Cole and O. Zajicek. An optimal algorithm for building a data structure for planar point location. *to appear in Journal of parallel and distributed computing*, 1989.

[28] G. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. *Lecture Notes in Computer Science*, 33:134–183, 1975.

[29] S. Cook and C. Dwork. Bounds on the time for parallel ram's to compute simple functions. *Proc of the ACM Annual Symp. on Theory of Computing*, pages 231–233, 1982.

[30] N. Dadoun and D.G. Kirkpatrick. Parallel processing for efficient subdivision search. *Proc. of the 3rd Annual ACM Symp on Comput. Geom*, pages 205 – 214, 1987.

[31] X. Deng. An optimal parallel algorithm for linear programming. *manuscript*, 1989.

[32] D. Dobkin and D. Kirkpatrick. A linear time algorithm for determining the separation of covex polyhedra. *Journal of Algorithms*, 6(3):381 – 392, 1985.

[33] D. Dobkin and R.J. Lipton. Multidimensional searching problems. *SIAM J. on Computing*, 5:181 – 186, 1976.

[34] M. Dyer. Linear time algorithms for two and three variable linear programs. *SIAM Journal of Computing*, 13:31–45, 1984.

[35] M.E. Dyer and A.M. Frieze. A randomized algorithm for fixed-dimensional linear programming. *Unpublished manuscript*, 1987.

[36] R. Dyer. *Average-case analysis of algorithms for Convex hulls and Voronoi diagrams.* PhD thesis, Carnegie Mellon University, 1988.

[37] H. Edelsbrunner. *Algorithms in combinatorial geometry.* EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1987.

[38] J. Sanz edits. Opportunities and constraints of parallel computing. *Workshop organized by NSF and IBM Almaden*, 1988.

[39] D. Eppstein and Z. Galil. Parallel algorithmic techniques for combinatorial optimization. *in Annual Rev. of Comput. Sci.*, pages 233–283, 1988.

[40] S. Fortune. A sweepline algorthm for voronoi diagrams. *Proc of the 2nd ACM Symp. on Comput.*, pages 511 – 516, 1986.

[41] Z. Galil. Optimal parallel algorithms for string matching. *Proc. of the 16th STOC*, pages 240 – 248, 1984.

[42] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *Proc. of the 27th Annual IEEE Symp. on Foundations of Computer Science*, pages 492 – 501, 1986.

[43] M.T. Goodrich. *Efficient Parallel Techniques for Computational Geometry.* PhD thesis, Purdue University, 1987.

[44] L.J. Guibas H. Edelsbrunner and J. Stolfi. Optimal point location in monotone subdivision. *SIAM Journal on Computing*, 15(2), 1986.

[45] Y. Han. *Designing Fast and efficient Parallel Algorithms.* PhD thesis, Duke University, 1987.

[46] D. Haussler and E. Welzl. ϵ-nets and simplex range queries. *Discrete and Computational Geometry*, 2(2):127 – 152, 1987.

[47] C.A.R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[48] R. Tarjan K. Clarkson and C.J. Van Wyk. A fast las vegas algorithm for triangulating a simple polygon. *Proc. of the fourth annual ACM Symp on Comput. Geometry*, pages 18–21, 1988.

[49] A. Karlin and E. Upfal. Parallel hashing - an efficient implementation of shared memory. *Proc. of the 18th ACM STOC*, pages 160–168, 1986.

[50] H. Karloff and P. Raghavan. Randomized algorithms and pseudorandom numbers. *Proc. of the 20th Annual STOC*, 1988.

[51] D.G. Kirkpatrick. private communication.

[52] D.G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12:18 – 27, 1979.

[53] C. Levcopoulos, J. Katajainen, and A. Lingas. An optimal expected-time parallel algorithm for voronoi diagrams. *Scandenavian conference on theoretical computer science*, 1988.

[54] Michael Luby. A simple parallel algorithm for maximal independent set problem, 1986.

[55] N. Meggido. Linear time algorithm for linear programming in r^3 and related problems. *Proc. of the FOCS*, pages 329 – 338, 1982.

[56] G. Miller and J. Reif. Parallel tree contraction and its applications. *Proc. of the 26th Annual IEEE F.O.C.S.*, pages 478 – 489, 1985.

[57] R. Miller and Q. Stout. Efficient parallel convex hull algorithms. *IEEE Transaction on Computers*, 37(12):1605–1618, 1988.

[58] D. Mount and S. Chandran. A unified approach to finding enclosing and enclosed triangles. *Proc. of the Allerton Conf on Communications, Control and Comput*, pages 87–96, 1988.

[59] K. Mulmuley. A fast planar partition algorithm 1. *Proc. of the 29th IEEE FOCS*, pages 580 – 589, 1988.

[60] F.P. Preparata and I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.

[61] M.O. Rabin. Probabilistic algorithms. *in: J.F. Traub, ed., Algorithms and Complexity, Academic Press*, pages 21–36, 1976.

[62] S. Rajasekaran. *Randomized Parallel Computation.* PhD thesis, Aiken Computing Lab, Harvard University, 1988.

[63] S. Rajasekaran and J. Reif. Derivation of randomized algorithms. *Technical report, Harvard University*, 1984.

[64] S. Rajasekaran and J. Reif. Optimal and sublogarithmic time radomized parallel sorting algorithms. *Technical Rept, Aiken Computing lab, Harvard University*, 1986. To appear in SIAM Journal on Computing.

[65] A. Ranade. How to emulate shared memory. *Proc. of the 28th IEEE FOCS*, pages 185–194, 1987.

[66] J.H. Reif. On synchronous parallel computations with independent probabilistic choice. *SIAM J. Comput.*, 13(1):46–56, 1984.

[67] J.H. Reif and S. Sen. Optimal randomized parallel algorithms for computational geometry. *Proc. of the 16th International conference on Parallel Processing*, 1987. A revised version is available as Duke University technical report CS-88-01.

[68] J.H. Reif and L.G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM*, 34:60 – 76, 1987.

[69] R. Reischuk. A fast probabilistic parallel sorting algorithm. *Proc. of the 22nd IEEE FOCS*, pages 212 – 219, 1981.

[70] R. Reischuk. A fast probabilistic parallel sorting algorithm. *Proc. of the 22nd Annual FOCS*, pages 212 – 219, 1981.

[71] I. Shamos. *Computational Geometry.* PhD thesis, Yale University, 1978.

[72] M. Shamos and D. Hoey. Closest-point problems. *Proc. of the 7th ACM STOC*, pages 224 – 233.

[73] Y. Shiloach and U. Vishkin. Finding the maximum, merging and sorting in a parallel computation model. *Journal of Algorithms*, 2:88–102, 1981.

[74] R. Solovay and V. Strassen. A fast monte-carlo test for primality. *SIAM Journal of Computing*, pages 84–85, 1977.

[75] B. Maggs T. Leighton and S. Rao. Universal packet routing algorithms. *Proc. of the 29th IEEE FOCS*, pages 256–269, 1988.

[76] R. Tarjan and C.J. Van Wyk. An o(nloglogn)-time algorithm for triangulating a simple polygon. *SIAM Journal on Computing*, 17:143–178, 1988.

[77] L.G. Valiant. Parallelism in comparison problems. *SIAM Journal on Computing*, 4:348 – 355, 1975.

[78] L.G. Valiant. A scheme for fast parallel communication. *SIAM J. on Computing*, 11:350 – 361, 1982.

[79] U. Vishkin. Deterministic sampling methods for fast pattern matching. *Technical Report, Univ of Maryland*, 1989.

[80] J. Wyllie. *The Complexity of Parallel Computation.* PhD thesis, Cornell University, 1979.

BIOGRAPHY

Sandeep Sen was born in Calcutta, India on May 24, 1962. After graduating from Senior Secondary High School in Bhilai, Madhya Pradesh, in 1980, he attended the Indian Institute of Technology, Kharagpur where he received his Bachelors of Technology degree in Computer Science and Engineering in 1984. Mr. Sen continued his education in the University of California at Santa Barbara where he received the M.S. degree in Computer Engineering in 1986. For his M.S. thesis he worked on parallel sorting algorithms for mesh-connected computers. A part of this work appeared in a paper entitled 'Parallel Sorting in Two Dimensional VLSI models of Computation' in the IEEE Transactions on Computers, February 1989. During his first year of study in the University of California, he was supported by a University of California Regents Fellowship. Mr. Sen transferred to Duke University in the Fall of 1986 to pursue the Ph.D. in Computer Science. His primary research interests are in the areas of parallel algorithms, randomization techniques, computational geometry and combinatorics.