Parallel Computational Geometry : An approach using randomization

John H. Reif* and Sandeep Sen†

March 20, 1999

Abstract

We describe very general methods for designing efficient parallel algorithms for problems in computational geometry. Although our main focus is the PRAM, we provide strong evidence that these techniques yield equally efficient algorithms in more concrete computing models like Butterfly networks. The algorithms exploit random sampling and randomized techniques for solving a wide class of fundamental problems from computational geometry like convex hulls, voronoi diagrams, triangulation, point-location and arrangements. In addition, the algorithms on the Butterfly network rely critically on an efficient randomized multisearching algorithm. Our description emphasizes the algorithmic techniques rather than a detailed treatment of the individual problems.

*Department of Computer Science, Duke University, Durham, N.C. 27706, U.S.A.

†Department of Computer Science and Engineering, Indian Institute of Technology, New Delhi 110016, India.

1 Introduction

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 chapter, 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, Connection Machine, Paragon, MasPar etc. are available commercially, they have vastly different architectures which play a crucial role in algorithm design. The RAM model serves as a reasonable abstraction for almost all uniprocessor computers available today. A substantial amount of research has focused 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 they were 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 Machine (PRAM) model. This model is a very natural extension of the RAM model (although not as ubiquitous).

1.1 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.

For a practitioner, the PRAM model may appear to be far from 'reality;' 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. We do not wish to confuse the reader any further by offering our own views. Instead, we claim that the methods described here are so general that they can be efficiently mapped into any reasonable parallel computation model. By that we imply that these algorithms will perform very well in any parallel environment, where the performance is measured with respect to a particular machine. For example, it is known that an algorithm running on a two-dimensional mesh cannot (theoretically) achieve the performance of a Butterfly machine with the same number of processors. So an exact (asymptotic) time bound makes sense only in the context of a fixed machine model. To illustrate our point, we give implementations of some of the algorithms on the Butterfly interconnection network that achieve optimal performance in that architecture.

As further evidence of generality, a number of these techniques have been exploited successfully for efficient algorithm design in the BSP (Bulk Synchronous Parallel) Model. This model was initially proposed by Valiant [87] to rectify some of the perceived shortcomings of the PRAM model. The BSP model is a coarse-grain parallel-computing model where all inter-processor communication is done in global phases and accounted for separately from computation phases. Algorithm design for BSP model is a relatively new area of research and the interested reader is referred to the Bibliography. Initial trends indicate that many PRAM algorithm design techniques carry over nicely to the BSP model (for example, see Goodrich [42]).

In this chapter, mostly we shall be using the CREW model and the performance is analyzed with respect to this model (without any significant loss of generality that follows from our discussion about other models). In addition, we shall assume that a single memory location can hold a real-number 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. Since we shall be using randomization, we also assume that processors have access to $O(\log n)$ random bits in constant time.

1.2 Performance measures of 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 polylogarithmic 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.

Clearly, using huge numbers 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 $P_n T_n = Seq(n)$ where P_n and T_n represent the number of processors and time used for solving a problem of size n. In the future we shall use PT without the subscript. For problems in $\mathcal{NC}$, $PT = O(n^k Seq(n))$. 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 polylog factor of the best known sequential algorithm. A parallel algorithm is called 'optimal' if $PT = O(Seq(n))$.

In this chapter, 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 1.1 (slow-down lemma) *If there exists an algorithm using P_1 processors and running in T_1 time then it runs in $O(T_1 P_1 / P_2)$ 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.

1.3 Randomized Algorithms

The technique of randomizing an algorithm to improve its efficiency was first introduced in 1976 independently by Rabin, and Solovay and Strassen. Since then, this idea has been used to solve a myriad of computational problems successfully. Today randomization has become a powerful tool in the design of both sequential and parallel algorithms.

Informally, a randomized algorithm is one which bases some of its decisions on the outcomes of coin flips. We can think of the algorithm with one possible sequence of outcomes for the coin flips to be different from the same algorithm with a different sequence of outcomes for the coin flips. Therefore, a randomized algorithm can be viewed as a family of algorithms. An objective in designing randomized algorithms is to ensure that the number of *bad* algorithms is a relatively small fraction of the total number of algorithms. If for *any* input we can show that at most ϵ (ϵ being very close to 0) fraction of algorithms in the family are bad, then this probability is independent of the input distribution.

Two types of randomized algorithms can be found in the literature: 1) algorithms that always produce the correct output but may run for an indefinite period (that is the running time is a random variable). These are called *Las Vegas* algorithms; and 2) those that run for a specified amount of time and whose output will be correct with a specified probability. These are called *Monte Carlo* algorithms. The primality testing algorithm of Rabin is of the second type.

The error of a randomized algorithm can either be 1-sided or 2-sided. Consider a randomized algorithm for recognizing a language. The output of the algorithm is either *yes* or *no*. There are algorithms which when saying *yes* are always correct, but when saying *no* may have produced a wrong answer. Such algorithms are said to have 1-sided error. Algorithms that have non-zero error probability on both possible outputs are said to have 2-sided error.

1.4 Performance measures of randomized algorithms

The algorithms that we describe in this chapter are *Las Vegas*, that is they always produce the correct results and only their running time is a stochastic process. So the predicted performance revolves around how well one is able to analyze the random variable that models the running time (or some other performance measures). It would be ideal if we could determine the actual probability distribution of this random variable but that is usually too complex. Instead, we try to bound the various characteristics of the actual probability generating function like *expectation*, *variance* or even higher moments. Sometimes, it is possible to bound the tails of the distribution function, which is of the type: What is the probability that the random variable exceeds a certain value ? Note that in the context of worst-case algorithm analysis, we are mainly concerned about the upper tail.

Just like the big-O function serves to represent the complexity bounds of deterministic algorithms, $\widetilde{O}$ serves to represent the complexity bounds of randomized algorithms.

Notation We say a randomized algorithm has a resource (time, space etc.) bound of $\widetilde{O}(g(n))$ if there exists a constant c such that the amount of resource used by the algorithm (on any input of size n) is no more than $c\alpha g(n)$ with probability $\geq 1 - 1/n^{\alpha}$ for any $\alpha > 0$. We shall refer to these bounds as *high probability* bounds.

This is also known as the *n-polynomial bounds*. Sometimes, it is possible to bound the tails with higher probability, namely, $1 - 2^{\Omega(n)}$. These are called *n-exponential* bounds.

1.5 Elementary tools for analyzing randomized algorithms

From our previous discussion, we would like to bound the probability that the algorithm does not terminate after certain time. (Although our discussion is with reference to running time, it applies to other parameters as well). We may think about it as the *failure probability*, that is, the failure of the algorithm to terminate in a certain time. Since the randomization is with respect to random numbers chosen by the algorithm, often this failure probability can be expressed as the probability of random numbers assuming certain sets of values. We can view them as *events* in some appropriate sample space so that the failure probability can be expressed as combinations (set operations) of certain events. If the events are *independent* (in the probabilistic sense), analysis becomes much simpler because the probability of the conjunction of independent events is the product of the probabilities of these events. However, often there is limited independence or no independence. In such cases we can rely only on trivial upper bounds as in

$$\Pr[\bigcup_i E_i] \leq \sum_i \Pr[E_i] \tag{1}$$

where for all i, E_i is an event in the sample space.

The significance of *high-probability* bounds is that the union of a polynomial number of events that succeed with high probability also succeeds with high probability. It follows from the definition of high-probability bounds.

The other frequently used tools are the probabilistic inequalities. They yield bounds on the tails of the probability distribution functions. The probability generating function $G_X(z)$ for a non-negative discrete random variable X is defined as $\sum_{i=0}^{\infty} \Pr[X = i]z^i$. Given $G_X(z)$, it is possible to determine the expectation, variance, and all the higher moments. The most celebrated is the

Chernoff bound, that bounds the tail of a random variable X given $G_X(z)$.
Generalized Chernoff bounds

$$\Pr[X \geq x] \leq z^{-x} \cdot G_X(z) \tag{2}$$

By minimizing with respect to z, one can obtain fairly sharp bounds. Since getting hold of $G_X(z)$ is usually quite difficult, the following inequality is quite useful when only the expectation of X, $E[X]$ is known.
Markov inequality

$$\Pr[X \geq k \cdot E[X]] \leq \frac{1}{k} \tag{3}$$

1.5.1 Chernoff Bounds for binomial random variables

Frequently, in the design of randomized algorithms, many parameters of interest (like running time, space used, etc.) are random variables with a binomial distribution. For example, let's assume that the run time of an algorithm is binomial with mean m. Can we say the run time of the algorithm is $O(m)$ with high probability? The answer is yes if m is sufficiently large. Intuitively, if m is large, the area under the tail ends of a binomial distribution is negligible. Chernoff bounds provide fairly tight estimates for computing the area under the tail ends of a binomial.

It turns out that if the mean $E[X]$ of a binomial random variable X is $\Omega(\log n)$, then the probability that X is greater than $O(E[X])$ is $\leq n^{-\alpha}$ for any $\alpha > 1$ (the constants in $\Omega()$ and $O()$ will depend on α). Very often it is easier to *bound* the random variable corresponding to the running time by a well-known distribution rather than analyze the exact distribution.

Definition 1.2: We say a random variable X upper bounds another random variable Y (equivalently, Y lower bounds X), if for all x such that $0 \leq x \leq 1$, Probability$(X \leq x) \leq$ Probability$(Y \leq x)$.

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 *distribution function* of X can easily be seen to be

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

Angluin and Valiant gave sharp bounds of approximating the tail ends of a binomial distribution. In particular, their results can be summarized as

Lemma 1.3 (Chernoff Bounds) *If X is binomial with parameters (n, p) and $m > np$ is an integer, then*

$$\Pr(X \geq m) \leq \left(\frac{np}{m}\right)^m e^{m-np}. \tag{4}$$

Also,

$$\Pr(X \leq \lfloor (1-\epsilon)pn \rfloor) \leq exp(-\epsilon^2 np/2) \tag{5}$$

and

$$\Pr(X \geq \lceil (1+\epsilon)np \rceil) \leq exp(-\epsilon^2 np/3) \tag{6}$$

for all $0 < \epsilon < 1$.

Remark 1.4: More general form of these inequalities are also known as Chernoff-Hoeffding bounds where the success probabilities of the n Bernoulli variables may be different. However, we do not need them in this chapter.

1.6 Problems considered

Computational geometry, as it stands today, is concerned with the design and analysis of geometric algorithms. This is quite vast in its scope - including problems from data-bases to mathematical programming. We have chosen some of the most fundamental problems that usually serve as an introduction to computational geometry. It is likely that these will serve as building blocks for solving more complex problems.

We will also rely on reductions to make the algorithms more versatile. An informal definition of parallel reduction can be given as:

> Problem A is $O(P(n), R(n))$ time parallel reducible to Problem B if any instance of A can be solved by using an algorithm for problem B as a subroutine in $O(R(n))$ parallel time using $P(n)$ processors. The sub-routine can be invoked at most a constant number of times and its running time is not included in $R(n)$.

In other words, a $T(n)$ time algorithm for problem B implies a $O(T(n) + R(n))$ time algorithm for problem A. Also a $\Omega(B(n))$ time lower bound for problem A implies a similar lower-bound for B.

1.6.1 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. A 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 clock-wise (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.

1.6.2 Voronoi Diagrams

A very general definition of Voronoi diagram given by Edelsbrunner [35] 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 $\{p \in E^d$: $d_s(p) < d_t(p), t \in S - \{s\}\}$ 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 chapter, 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. The subsets considered will be only the singleton points of S. 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 problems can be solved in linear or $O(n \log n)$ time from the information contained in the Voronoi diagram. A partial list includes

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.

Remark 1.5: Using a technique of Brown [18], construction of a Voronoi-diagram (in plane) is $(n, 1)$ reducible to convex-hull in three-dimensions. Later in the chapter, we describe an algorithm for computing the 3-D convex hull.

1.6.3 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 edges (both above and below the vertex) that intersect the vertical line drawn through this vertex. An optimal $O(n)$ time triangulation algorithm was a subject of much research before it was settled by Chazelle [19].

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 end-point in the vertical direction. 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 $\Omega(n \log n)$).
It is also called the *Vertical visibility* problem and in the context of sequential algorithms this problem and the triangulation problem are linear time reducible to each other.
Remark 1.6: Triangulation is $(n/\log n, \log n)$ reducible to Trapezoidal decomposition (Yap[90]). Note that this is not an optimal reduction since simple polygons can be triangulated faster than that of trapezoidal decomposition of arbitrary non-intersecting segments.

1.6.4 Point location

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

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 polyhedra in three dimensions.

1.7 Arrangements

The arrangement of n hyper-planes in d dimensions contains information about the (connected) regions into which the space is partitioned and the incidence relation between the regions of all dimensions. Arrangements are fundamental combinatorial structures that are useful for solving numerous problems including detecting degeneracy to motion planning.

2 Basic parallel Routines

In this section we will review parallel algorithms for problems like sorting, selection, and some basic parallel procedures that will be frequently used in the design of geometric algorithms. In particular, we will focus on randomized algorithms that are simpler and/or faster than their deterministic counterparts. Most of the material in this section can be skipped without loss of continuity of the remaining chapter. In fact, we recommend that if the reader has some familiarity with parallel algorithms, he can go directly to section 2.5 and refer back to the results of this section as necessary. Unlike other sections we have not attempted to develop all the algorithms from scratch. For more detailed treatment of the material of this section, we provide extensive bibliographic references at the end of this chapter. We have also omitted description of the more complicated algorithms, namely the sub-logarithmic time algorithms, as their understanding is not important for the development of the subsequent geometric algorithms. A detailed description of these methods is outside the scope of our discussion.

2.1 Parallel Prefix computations

Given n elements a_i $(1 \leq i \leq n)$ from a semi-group where $\oplus$ is the semi-group operation, we are required to compute all the partial sums of the form $S_k = \bigoplus_{i=1}^{k} a_i$ 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 [24] obtained the following result:

Lemma 2.1 *In a CRCW (arbitrary) PRAM model, the prefix sum of n elements can be computed optimally in* $O(\log n / \log\log n)$ *time.*

2.1.1 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 [88] was

known for a while, it used n processors and was not optimal. The following result was obtained independently by Anderson and Miller [10], Han [47], and Cole and Vishkin [22].

Lemma 2.2 *The list ranking problem can be solved in an EREW model optimally in $O(\log n)$ time.*

2.2 Parallel Sorting and Selection

Like sequential algorithms, sorting and selection play very important roles in the design of parallel algorithms. Various basic parallel routines like compaction and processor allocation can be accomplished directly by sorting. (We will see later that there are more direct and faster ways of implementing them.) Sorting also forms a preprocessing step for ordered searching where the searching is often performed in more abstract domains than an ordered set of points.

In the context of deterministic PRAM algorithms, the parallel mergesort algorithm due to Cole [22] is perhaps the most elegant and has small constants. 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 [9], Reischuk [79] and Flashsort [78]. Both Reischuk's algorithm and Flashsort use randomization that we will describe in more detail.

Lemma 2.3 *In an EREW PRAM model, n keys from a totally ordered set can be sorted in $O(\log n)$ time using n processors.*

In the 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. However, the development of quicksort is a natural exposition for the algorithms in the subsequent sections.

2.2.1 Parallel Quicksort

In Reischuk's parallel adaptation of quicksort algorithm, $\lfloor\sqrt{n}\rfloor$ keys are chosen randomly such that each key was chosen with probability $\frac{1}{\sqrt{n}}$. These keys (called splitters) are then sorted using pairwise comparisons from which their ranks can be computed easily. 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 $\lfloor\sqrt{n}\rfloor+1$ intervals in a natural fashion. For each key, we can determine the appropriate interval by a simple binary search using one processor for each key (using simultaneous reads). The algorithm is then called recursively in parallel for each interval. We have ignored here the task of putting the elements of each partition in contiguous locations for applying recursion properly; we shall discuss this later under the problem of *semisorting*. If n_i denotes the size of the $i-th$ bucket then we claim the following:

Lemma 2.4 *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 $\frac{1}{n^c}$. Hence the probability that it can happen for any element is less than $\frac{1}{n^{c-1}}$. Consequently the distance (in rank) between two sampled elements is less than $c\sqrt{n}\log n$ with probability at least $\frac{1}{n^{(c-1)}}$. □

Remark 2.5: The above lemma can be extended easily to the case when the number of sampled keys is r for any $r \leq n$. For any general r the proof of Lemma 2.4 extends easily to bound the interval sizes by $\tilde{O}(n \log n/r)$ (the lemma was proved for the case $r = n^{1/2}$).

In the following discussion, we shall use the bound that intervals are of size $O(n^{\epsilon_0})$ where $\epsilon_0 > 1 - \epsilon$ for $r = n^{1-\epsilon}$ (the extra $\log n$ factor is subsumed in the exponent). 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 $O(\log \log n)$ time, thereby adding a factor of $o(\log n)$. Our objective is to show that all the leaf-level procedures are completed in $\tilde{O}(\log n)$ time with high probability. For this it suffices to show that a particular leaf-level procedure is completed in $O(\log n)$ 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 by N_i, the problem-size by n_i, and the time taken at N_i by T_i. We claim the following:

Proposition 2.6

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

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

Proof: From the previous claim and the comment in the previous paragraph, $\Pr[n_{(i+1)} > n_i^{\epsilon_0}] < \frac{1}{n_i^2}$ for an appropriately chosen constant $0 < \epsilon_0 < 1$. We can verify this in $O(\log n_i)$ 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:

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

If $n_i = 2^{\epsilon_0^i \log n}$ then we arrive at the required inequality. In this 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 $a > 1$. Substituting this value of n_i in the previous inequality we obtain

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

The above inequality implies that

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

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

We shall now prove the following useful result:

Theorem 2.7 *Given a process-tree which has the property that a procedure at depth i from the root takes time T_i such that*

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

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

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

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

If T is the total time for this worst case chain of nested calls and $m = 1/(1\text{-}\epsilon_0)$, the probability that T exceeds $mk\alpha \log nc_o + t$ is less than the sum of the probability of events where $\sum_i t_i = t$, $t > 0$, and $\mu = mk\alpha \log nc_o$. We shall compute the probability that $\sum_i t_i = t$.
$\prod_{\sum t_i = t} 2^{-t_i} \leq \sum 2^{-t/k}$ over $t^{O(\log\log n)}$ tuples.
Thus $\Pr[T > km\alpha \log nc_o + t] < 2^{-t/k + O(\log t \log\log n)}$.
Using $t \geq km\alpha(c - c_o)\log n$, for large values of n and $m > 1$, we can rewrite the above expression as

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

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

$$\Pr[T > \alpha log n] \leq 2^{-(3/4)c\alpha \log n} \leq n^{-c_1\alpha},$$

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

2.2.2 Selection

We will focus primarily on extremal selection, that is, choosing the smallest or the largest element in a given set N where $|N| = n$. General selection, that is choosing the k-th ranked element for any $1 \leq k \leq n$, is an important problem in its own right but the geometric algorithms do not make use of that. It is known that extremal selection takes $\Omega(\log\log n)$ rounds using n processors in any reasonable parallel computing model of parallel computation. However, we will show that a fairly simple constant time randomized algorithm exists for selecting the smallest element.

The idea is to make use of Lemma 2.4 and focus on the interval that contains the smallest element. Choose a set R of $n^{2/3}$ splitters randomly, and let the smallest element among the randomly chosen elements be R_0. The candidate elements for minimum are all those elements of N that are smaller than R_0; denote this set by $\bar{N}$. From the Remark 2.2.1 (following Lemma 2.4), the size of each interval does not exceed $\tilde{O}(n^{1/3}\log n)$ and hence this bounds $|\bar{N}|$. If we ignore the time for processor allocation, then the entire algorithm works in time needed to compute the minimum of $n^{2/3}$ elements using n processors, or equivalently computing the minimum of n elements using $n^{3/2}$ processors. The latter can be done easily in $O(1)$ time as follows.

Let $k_1, k_2, \ldots, k_n$ be the input keys. Group the $n^{1+1/2}$ processors into $\sqrt{n}$ groups $G_1, G_2, \ldots, G_{\sqrt{n}}$ where each group has n processors. Each group G_i $(1 \leq i \leq \sqrt{n})$, in parallel, computes the maximum of $\sqrt{n}$ elements in constant time. This can be done by checking for each element k_i, if there is a larger element in the input than k_i. Using a processor for every pair and a special 'marker-cell' for every element this can be accomplished in $O(1)$ time. The processor comparing k_i and k_j writes (concurrently) into the 'marker-cell' i, if it finds that $k_i < k_j$. Only one 'marker-cell' will not be written into which corresponds to the maximum key. Subsequently we choose the maximum of the $\sqrt{n}$ maxima by another application of the previous strategy (there are $n^{3/2}$ processors and only $\sqrt{n}$ keys in this phase).

After determining R_0, we have to 'compress' the elements of $|\bar{N}|$ in contiguous locations for another round of the previous algorithm. In fact, even approximately compressing them (with some empty cells in between) also suffices for our purpose. We will discuss this in the next section.

2.3 Load-balancing and processor allocation

For a large 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 [17] makes this observation into a formal statement:

Lemma 2.8 (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** stated earlier is a special case of this lemma. In our example of the summation of n elements, we can view it as the following: 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 with 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 2.9 *The algorithm can be made to run in $O(\log n)$ time using only $O(n/\log n)$ processors in a CRCW PRAM model.*

The following scheme relies on the sub-logarithmic time algorithm for prefix-sum. After $O(\log \log n)$ stages, only $(n/\log n)$ processors are required. 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 the same lines.

2.4 Semisorting

A very important component of recursive parallel algorithms is the actual divide step. That is, after figuring out the recursive subproblems, we have to collate the elements of a subproblem together to restore the initial condition for recursive calls. When this division is uniform and predictable, like dividing the input array across the median (like the mergesort), then this step is trivial. On the other hand the elements of a subproblem can be scattered unpredictably so that these have to be brought into contiguous locations before we can proceed with the recursion. In the geometric algorithms that we will develop, we will often deal with the latter scenario, so we shall examine this in some detail. Also recall that in the parallel quicksort this is an important step.

A natural way to think about this problem is to associate integral labels (from a certain range) with each element of a subproblem - that is, all the elements of a subproblem have unique integral labels. In a problem of size n, it is unusual to have labels that exceed n, so we shall restrict ourselves to the case where the labels are in the range $[1 \ldots n]$. For the special case where there are only a constant number of labels, the counting techniques of the previous section can be used.

Definition 2.10: [Semisorting] Given n elements in range $1 \ldots m$, place these in an array of size $O(n)$ such that if there are n_i elements having label $1 \leq i \leq m$, these are placed in a subarray of size $O(n_i)$. Moreover, the subarrays corresponding to distinct labels should not overlap.

Clearly semisorting can be accomplished by sorting on the integer labels, namely by integer sorting. However, the definition of semisorting allows for more flexibility that will be exploited later for sub-logarithmic algorithms.

We will now describe a method for semisorting when labels are in the range $1 \ldots \frac{n}{\log^2 n}$ (somewhat smaller than n). Assume that we know how many elements of each label are present in the array; we shall return to this problem when we discuss *approximate counting*. Semisorting can be done by allocating a subarray of the required size for each label and then actually placing the elements in the appropriate subarray. The former can be done by using parallel prefix sums to compute the boundaries of the non-overlapping subarrays from the size requirements. Suppose there are at most n_i elements having label i (that is we have an overestimate of elements with label i), such that $\sum_i n_i$ is $O(n)$. Let the subarray $I(i)$ corresponding to label i have addresses from $S(i)$ to $F(i)$. Now each element having label i has to be placed in a location in $I(i)$. We call this the *placement problem*. When two (or more) processors try to place elements of label i into the same location (in $I(i)$) we call that *collision*. As there is no restriction on where a certain element having label i is placed in $I(i)$, it is difficult to avoid collision. However, if $|I(i)| > f \cdot n_i$ for some $f > 1$, then the

chances of collision are less. More formally, if elements are placed in a random location in $I(i)$, the probability of collision is no more than $\frac{1}{f}$. In the case of a collision, an attempt is made to place the element again. The expected number of tries for an element is $\frac{f}{f-1}$. The above observation can be refined into the following claim, which we state here without proof.

Lemma 2.11 (Placement Lemma) *Using $P \leq n/\log n$ processors, the placement problem of n elements can be solved in $\tilde{O}(n/P)$ time in an arbitrary CRCW model.*

2.4.1 Estimation and Approximate Counting

We now address the problem of estimating the number of elements having label i for all $i \in [D]$ without explicitly counting all the labels [1] More specifically, for each label i, we want to determine a number $D(i)$ which is an upper-bound on the number n_i of elements having label i and additionally $\sum_i D(i)$ is $O(n)$. We describe a method by which we can reduce this problem to explicit counting of only n/K elements for any $K \leq \frac{n}{D \log n}$.

We first sample every element with probability $\frac{1}{K}$; call this set N_S. Denote the number of elements in N_S having label i as $D_S(i)$. This is obtained by explicitly counting them (which is much smaller than n). Let $D(i) = dK\ max(|D_S(i)|, \log n)$ for $i \in [D]$, where d is a constant which will be determined in the analysis.

If $n_i \leq dK \log n$, then always $D(i) \geq dK \log n \geq n_i$. So suppose $n_i > dK \log n$. It is easy to see that $|D_S(i)|$ is a binomial variable with parameters $(n/K, \frac{n_i}{n})$. The Chernoff bounds (see Lemma 1.3, equation 5) imply that for all $\alpha \geq 1$, there exists a c such that

$$\Pr\left(|D_S(i)| \leq c\alpha n_i/K\right) \leq \tfrac{1}{n^\alpha}$$

Therefore, if we choose $d = (c\alpha)^{-1}$ then $D(i) \geq n_i$ (for every $i \in [D]$) with probability $\geq 1 - n^{-\alpha}$. The Chernoff bounds (equation 3) also imply that for all $\alpha \geq 1$ there exists an h such that $D(i) \leq (h\alpha)n_i$ (for every $i \in [D]$) with probability $\geq 1 - n^{-\alpha}$. Moreover,

$$\begin{aligned}
\sum_{i \in [D]} D(i) &\leq \sum_{i \in [D]} dK[|D_S(i)| + \log n] \\
&= dKD \log n + d \log^2 n \sum_{i \in [D]} |D_S(i)| \\
&= dn + dK \cdot n/K = 2dn
\end{aligned}$$

given that $K \leq \frac{n}{D \log n}$. We may summarize the above observation in the following manner that will be useful later.

Lemma 2.12 (Estimation Lemma) *For any $K \leq \frac{n}{D \log n}$, by examining only a $1/K$ fraction of a given input of n elements with labels $l_1, l_2, \ldots, l_n \in [D]$ we can compute $N(1), N(2), \ldots$ such that $\sum_{i \in [D]} N(i) = O(n)$ and with very high likelihood $N(i) \geq n_i$ for each $i \in [D]$ where n_i is the number of elements with label i.*

The immediate utility of the above lemma is in increasing processor efficiency of parallel algorithms. The problem of estimating the number of elements with a particular label has been reduced

[1] We are using $[D]$ to denote $\{1 \ldots D\}$.

to exact counting for a small fraction of elements, thereby reducing the number of processors. The exact counting for smaller number of elements can be done using less efficient algorithms. In particular, we can now use fast parallel sorting algorithms that are suboptimal. We refer the reader to the bibliography at the end of the chapter for references to such algorithms.

We state the following result for semisorting

Lemma 2.13 (Semisorting) *Semisorting of n elements with labels in the range $[1 \ldots n]$ can be done in $\tilde{O}(\log n)$ time using $n/\log n$ CRCW processors.*

2.5 Randomized symmetry breaking with application to point location

In a number of parallel algorithms a typical scenario is that a processor has to make a choice based on local information and there is (apparently) not much to distinguish between the choices. However, from a global perspective the local choices, if not properly coordinated, could be quite disastrous. Consider the following problem of identifying a *large independent* set in a linked list $\mathcal{L}$ of n elements. Recall that an independent set is a set of nodes that do not have edge (links) between them. By a *large* independent set, we imply a set that is larger than $\epsilon \cdot n$ for some constant $\epsilon > 0$. This is also called the *fractional independent set* (FIS). We know that the maximum sized independent set is of size at least $n/2$ (by choosing the elements with either the odd ranks or the even ranks). If the given linked list has not been ranked (the distances of each element from the end is not known), it is not easy for a processor to determine if a given element is of odd or even rank. So locally it is difficult to figure out simultaneously for each element whether or not to include it. However, inclusion (or exclusion) implies that its neighbor will be excluded (included). Sequentially, it is a simple procedure, but concurrent decision-making between symmetric situations causes complications in parallel context.

Consider the following method. Let us choose a tag $t_i \in \{0, 1\}$ with equal probability for each element $i \in \mathcal{L}$. Let N_0 and N_1 denote the sets of elements that have tags 0 and 1 respectively. Note that the expected size of N_0 (and N_1) is $n/2$. We choose an element x to be in FIS of the linked list N if and only if

1. $x \in N_1$
2. The (at most two) neighbors of x, $\mathcal{N}(x)$ have tags 0.

The first condition limits the FIS to be a subset of N_1 and the second condition ensures that two neighboring elements of the list are not chosen. The expected size of FIS is

$$\sum_{i \in \mathcal{L}} \Pr[i \in \text{FIS}]$$

$$= \sum_{i \in \mathcal{L}} \Pr[\{i \in N_1\} \cap \{\mathcal{N}(i) \subset N_0\}].$$

This is at least $n/8$ using the fact that an element is assigned tag 0 or 1 with probability 1/2. This procedure takes time $O(n/P)$ with P processors in an EREW PRAM and is the basis for a very simple optimal list ranking algorithm. The first geometric algorithm for planar point location makes very crucial use of the above technique which is often referred to in the literature as *random mate*.

2.5.1 Point location in triangulated planar maps

A Planar Straight Line Graph (PSLG) is an embedding of a planar graph on the 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. In addition we assume that the given PSLG is already triangulated. In a latter section we shall describe efficient parallel algorithms for triangulation so that this assumption is not a serious bottleneck.

For a PSLG with n vertices, the optimal query time of $O(\log n)$ with $O(n)$ space can be achieved by some very elegant methods [36, 57, 81]. The performance is achieved by binary search on a well organized data structure constructed on the PSLG during the preprocessing. We describe 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 triangulated PSLG.

Kirkpatrick had proposed a triangulation refinement technique which builds a hierarchy of triangulated PSLG from the initial graph. 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 triangles), 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 re-triangulated 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 top level, where the problem can now be solved in constant time or $O(\log n)$ time if we stop at an $O(\log n)$ size graph. Subsequently, at each level we relocate the point with respect to the triangles that 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 terminates. The search data structure is a directed acyclic graph. Each of the nodes represents 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 necessarily 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 of edges in a planar triangulated graph (V, E), where V and E are the vertices and edges respectively, 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 preprocessing time.

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.

We 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 by generalizing the random-mate technique described in the previous section.

Algorithm FIS

Input: A PSLG in the 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.

Remark 2.14: We have used 'male' and 'female' instead of the tags $\{0, 1\}$.
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 2.15 *Let $|V| = n$ and let n_d denote the number of vertices with degree less than or equal to d ($n_d \geq 6n/d$ as shown earlier). There exist constants c, ν $(0 < \nu < 1)$, such that*

$$\Pr[|X| < \nu n] \leq \exp(-cn)$$

.

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 had 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 $(1/d_{max}) \cdot 6n/d$ 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 expectation $\mu = |I_3(n_d)|$p $\geq 6np/D$.

Using Chernoff bounds (see Lemma 1.3 equation 2), for $0 < \epsilon < 1$,

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

Using $\nu = (1-\epsilon)6/D$ and c $= \epsilon^2/(12D)$ we obtain the required result. □

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 now present the complete algorithm below.

Procedure Point-Location-Tree

(1) Let N_i be the number of vertices in stage i where $N_0 = N$. If $N_i \leq k \log n$ (for some fixed constant k), then halt (we can then locate the query point in the $O(\log n)$ triangular faces in $O(\log n)$ time using a single processor).
(2) Choose an independent set of vertices that have degrees ≤ 12 using the method described above with $d = 12$. The time for this step is $O(1)$.
(3) Triangulate the remaining PSLG (after the removal of the independent set). Note that removal of a vertex of degree d necessitates triangulating a simple polygon of d vertices. This can be done in constant time using 1 processor for each polygon. In addition, the adjacency list of the vertices incident on the edges (removed or inserted) have to be updated. All this can be done in $O(1)$ time with one processor for every vertex.
(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) Increment i and go to Step (1).

Theorem 2.16 *Algorithm* **Point-Location-Tree** *runs in $\tilde{O}(\log n)$ time using $O(n)$ processors and $O(n)$ 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 2.15, 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, where $n_0 = n$. We show that after

$O(\log n)$ stages, the problem size is less than $O(\log n)$ with very high likelihood.
From Lemma 2.15, $\Pr[n_i \leq (1-\nu)n_{i-1}] \geq 1 - n^{-cK}$ for $n_{i-1} \geq K \log n$ 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 $\delta > 0$. 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 argument for space is similar to the sequential case. If the input PSLG is triangulated, the number of operations performed is linear in n (follows from the sequential algorithm). Using Slow-down technique, the processor bounds can be reduced to $O(n/\log n)$ without affecting the asymptotic run-time. □

2.6 Sublogarithmic routines

For a while, the designers of parallel algorithms were apprehensive about investigating parallel algorithms that achieved a running time of $o(\log n)$. This was mainly due to a lower bound of $\Omega(\frac{\log n}{\log \log n})$ running time for parity in the CRCW model even by using a polynomial number of processors. There were a few exceptions like the extremal selection that can be done in constant time (using n^2 processors). Moreover, for the weaker models like the EREW and the CREW models, $O(\log n)$ is a lower bound for computing a simple function like **OR** of n bits using a polynomial number of processors. So logarithmic time appeared like a natural barrier for most of the interesting problems, including sorting and prefix sums. Note that if one could sort elements into an array of size n, parity can be computed in constant time after that. Since sorting is considered such a fundamental problem, the idea of sublogarithmic time parallel algorithms were considered fairly impractical for most sorting-related problems. There were attempts to study parallelism in even stronger models like *parallel comparison tree* (PCT) where the only operations that are counted are comparisons. So, for example, counting or processor allocation is not accounted for. These results were much better than CRCW PRAM but the PCT model is considered too unrealistic for designing algorithms and primarily a tool for proving lower bounds.

The situation for CRCW models changed dramatically with a slightly modified notion of sorting called *padded-sorting*, first defined by MacKenzie and Stout [62]. Roughly speaking, the problem of padded-sorting involves ordering the input of size n into an output array of size $m \geq n$. When $m = n$, or when m is very close to n, the lower-bound of selection becomes applicable. A crucial factor in the performance of the padded-sorting algorithm is the size of the output array m, or more specifically, the ratio m/n. If $m = (1+\lambda)n$, then λ is called the *padding factor*. A surprising result (that we state below more formally) showed that it is possible to match the bounds of the PCT even for a padding factors significantly less than 1. There is actually a tradeoff between λ and running time; however we state a slightly simplified version that will be useful for our purpose

Theorem 2.17 *Given n elements from an ordered universe, these can be sorted with kn CRCW processors in $\tilde{O}(\log n/\log k)$ time with a padding-factor $\lambda \leq 1/\log n$. Moreover between any $\log n$ consecutive input keys, there is no more than one empty cell in the output array.*

Remark 2.18: The speed-up implied by this theorem is known to be optimal in the PCT model. A nice consequence of Theorem 2.17 is to ordered searching. The output of the padsort algorithm makes it almost directly applicable to search for the predecessor of a given key value. We simply probe the elements like a normal binary search except that when an empty cell is probed, we make an extra probe in the adjoining cell. As a consequence of Theorem 2.17 two adjacent cells cannot

be empty. Alternatively, we may simply fill up the empty cells with the contents of the preceding cell and do a usual binary search. The same holds true for any $k-ary$ search. In summary

Lemma 2.19 *The output of the padded-sorting algorithm can be used for performing k-ary search on an n-element ordered array in* $O(\log n/\log k)$ *steps.*

The algorithms that we develop in the following sections depend critically on sorting and searching and we will be able to use the above results to derive even sublogarithmic time algorithms in the same spirit as padded-sorting. Note that sorting can be used to accomplish some of the basic parallel routines like semisorting and compaction. However, there are more efficient and direct algorithms now available for these tasks.

Definition 2.20: [Approximate Compaction] Given n elements, of which only d are *active*, the problem of *approximate compaction* is to find the placement for the *active* elements in an array of size $O(d)$.

Definition 2.21: [Interval Allocation] For all $n \in N$, the *interval allocation* problem is the following: Given n non-negative integers $x_1, \ldots, x_n$, compute n nonnegative integers $y_1, \ldots, y_n$ (the starting addresses of intervals of size x_i) such that
(1) For all i, j, the block of size x_i starting at y_i does not overlap with the block of size x_j starting at y_j.
(2) $\max\{y_j : 1 \leq j \leq n\} = O(\sum_i x_i)$.

The interval allocation problem can be looked upon as a very general method for processor allocation. The integers x_i represent the processor requirement for task i. After interval allocation, we simply divide up the y array equally among the processors, thus ensuring load balancing. The constant hidden behind O of the output array captures the *slow-down* from the ideal load-balancing. The following result is stated without details of the algorithms.

Lemma 2.22 *There is a constant* $\epsilon > 0$ *such that for all given* $n, k \in N$, *problems of of interval allocation and approximate compaction of size n can be solved on a CRCW PRAM using* $O(k)$ *time,* $O(n \log^{(k)} n)$ *processors and* $O(n \log^{(k)} n)$ *space with probability at least* $1 - 2^{-n^{\epsilon}}$. *Alternatively, it can be done in* $\tilde{O}(t)$ *steps,* $t \geq \log^* n$ *using* n/t *processors.*

3 Random Sampling and Polling 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 subdivided into smaller subproblems and then the solutions of the subproblems are combined to obtain a solution to the original problem. The smaller subproblems 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:

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

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. The exact distribution is often hard to ascertain and in most cases only the expectation is known. For some special forms, one can solve these *probabilistic recurrence relations* satisfactorily (see Karp [53]). However, for most cases, such general solutions are hard to obtain. One example is the parallel quicksort described in the previous section where our analysis (proof of Theorem 2.7) depended heavily on the tail estimates of problem sizes.

In the context of computational geometry, sorting can be looked upon as a one-dimensional problem. The basic tools that we will develop in this section will enable us to extend some of the techniques to higher-dimensional problems. In some sense, this exercise can be viewed as (although readers are cautioned against over-simplified conclusions) expanding the basic paradigm of quicksort-like algorithms. For the most of this section, the techniques discussed are very general without reference to any particular problem. We shall present some interesting applications in the next section to specific problems where algorithms will be presented more formally.

3.1 Random Sampling

The divide-and-conquer mechanism that we will use is based on partitioning the problem using a random subset of the input. Recall that in quicksort, the input was partitioned by splitters that were randomly chosen elements of the input. For the higher-dimensional problems, it is not obvious how these splitters partition the input in the absence of a linear ordering.

Example 1 : For concreteness, consider the problem of constructing the *trapezoidal map* of a given set N of line segments. These segments partition the plane into regions which may have complicated shapes. By passing vertical lines through every end-point and intersection, these regions get partitioned into trapezoids or triangles (degenerate trapezoids). The vertical lines are not allowed to cross line segments. See Figure 1 for an illustration.

We will denote the trapezoidal map of a set S by $\mathcal{T}(S)$ and the set of trapezoids in $\mathcal{T}(S)$ as $H(S)$. By choosing a random subset of line segments $R \subset N$, it is not clear how to partition N as there is no natural ordering between segments (like points in a line). Since it is a two-dimensional problem, a natural solution is to consider the two-dimensional partitions induced by R. For example, consider $\mathcal{T}(R)$. For any trapezoid $\triangle \in H(R)$, consider $\mathcal{T}(N)$ restricted within $\triangle$. The union of $\triangle \cap \mathcal{T}(N)$ for all $\triangle \in \mathcal{T}(R)$ contains all the relevant information about $\mathcal{T}(N)$. Thus we can recursively compute construct $\mathcal{T}(N)$ within each $\triangle \in H(R)$. We will denote the line segments intersecting a trapezoid $\triangle \in H(R)$ by $L(\triangle)$ so that recursively we compute $\mathcal{T}(L(\triangle)$. From our earlier discussion, a bound on $\max\{L(\triangle\}$ will be crucial for the running time of the

algorithm. In addition, the quantity $\sum_{\triangle} L(\triangle)$ is also important as a segment $s \in N$ can intersect many $\triangle$s implying that this quantity could exceed n. So it represents the *blow-up* in the overall problem size in a recursive call and would affect the overall efficiency of any recursive algorithm.
Remark 3.1: In the case of quicksort, we did not have to worry about it since an element would belong to exactly one interval.

Example 2: Consider the problem of computing the intersection of a set N of half-spaces in three dimensions. If we adopt the previous approach, we choose a random sample R of half-spaces and construct the intersection $\cap R$. For convenience, we assume it is non-empty and contains the origin in its interior. Unlike the previous case, we do not have regions analogous to the trapezoids. With some thought, it is not difficult to come up with sub-divisions analogous to trapezoids. For example, the pyramids formed by joining the origin to every vertex of the intersection. For technical reasons that will become clear later, we will like to have these regions, which we shall call *ranges* defined by a constant number of input objects or equivalently having constant size. A pyramid defined as above can have a fairly large base if the corresponding face (of $\cap R$ is large). By further subdividing the bases using parallel translates of a fixed plane, we can restrict the pyramid bases to be trapezoids (or triangles), so that these have constant size. The intersection $\cap N$ can be constructed recursively within these pyramids. In this context, $L(\triangle)$ denotes the set of half-spaces whose defining half-planes intersect a pyramid $\triangle$.

The previous examples would have given the reader some intuition about how random sampling gives rise to a natural class of divide-and-conquer algorithms in computational geometry. It is not difficult to come up with a suitable definition of *range* in the context of a given problem. To prove any interesting results about these algorithms, we have to bound the quantities $\max\{L(\triangle\}$ and $\sum_{\triangle} L(\triangle)$. The former is crucial to bound the depth of recursion, which determines parallel running time and the latter will determine the efficiency of this approach. We shall prove some useful bounds for these quantities for fairly general situations. From here, we will use $\triangle$ to denote a range in an abstract setting where a range is a subset of the Euclidean space E^d defined appropriately in the context of the problem in E^d. We will require that a range be defined by at most a constant number of input objects, say b which bounds the possible number of ranges by a polynomial in n, namely $O(n^b)$. In the case of trapezoidal maps, the reader can verify that b is 4. We will use $l(\triangle)$ to denote $|L(\triangle)|$ which will be referred to as the *conflict size* of $\triangle$. Note that we are interested in those ranges $\triangle$ where $\triangle$ is a range in $\mathcal{T}(R)$. The following result gives a bound on $l(\triangle)$ for a random sample R.

Lemma 3.2 *For $|N| = n$ and a random sample R obtained by choosing every element of N independently with probability r/n, $\max_{\triangle}\{l(\triangle)\}$ can be bounded by $\tilde{O}(\frac{n}{r}\log n)$.*

Remark 3.3: The lemma uses sampling in a slightly different manner than what we had discussed previously. However, it can be seen easily that $|R|$ is $r + o(r)$ with high probability and the same bound would hold for a random sample of size r exactly. In the subsequent discussions, we will not distinguish between these sampling schemes.

Proof: To avoid complicated notations required for general situations, we will prove it for a special case, namely trapezoidal maps. Extensions to other problems like intersection of half-spaces and other problems considered in this chapter are fairly straightforward. The reader can refer to Clarkson [26] or Mulmuley [67] for more general proofs.

For any trapezoid $\triangle$, $L(\triangle)$ can be partitioned into two disjoint classes - those segments that intersect some (possibly more than one) boundary of $\triangle$ and the others that lie completely within the $\triangle$. We will bound each of these sets, denoted $L1(\triangle)$ and $L2(\triangle)$, by $\tilde{O}(\frac{n}{r}\log n)$. Let $\mathcal{L}$ be the union of vertical line through the end-points or the intersection points and the input segments - $|\mathcal{L}|$ can be bound by $O(n^2)$. For a fixed line in $\mathcal{L}$ look at the ordered intersections of N with this line and the intervals induced by the sampled segments in R. The maximum size of any interval is a bound on $L1(\triangle)$. This can be bounded by $\tilde{O}(\frac{n}{r}\log n)$ using arguments identical to Lemma 2.4 and Remarks 2.2.1.

For bounding $L2(\triangle)$, we define a trapezoid $\triangle$ to be 'bad' if it has more than $\alpha n \log n/r$ segments and none of them is chosen in R. The probability that a $\triangle$ is 'bad' is less than $(1-\frac{r}{n})^{\alpha n \log n/r} \leq \frac{1}{n^\alpha}$. The number of potential trapezoids $\triangle$ is $O(n^4)$ for n segments. So no trapezoid is 'bad' with high probability, for $\alpha \geq 5$. This implies that for $\triangle \in H(R)$, $L2(\triangle)$ is bounded by $\tilde{O}(\frac{n}{r}\log n)$ since the event $\triangle \in H(R)$ is 'bad' is a special case of the previous event. $\square$

Remark 3.4: The argument that we used for $L2(\triangle)$ would actually hold for $L1(\triangle)$ also. However, for a number of problems, the arguments are no more complicated than the one we used for $L1(\triangle)$. For the intersection of half spaces, a similar argument is applied to the segments joining the origin with a vertex of the arrangement of planes (bounding the input half-spaces). In this case, we bound the number of unsampled planes before the first sampled plane starting from the origin.
The bound can be improved in the following direction - with probability at least $1/2$, $\max_\triangle\{l(\triangle)\}$ is $O(\frac{n}{r}\log r)$ ($\log n$ is replaced by $\log r$). See [26] for details.

The next result bounds the expected value of $\sum_\triangle l(\triangle)$.

Lemma 3.5 *For $|N| = n$ and a random sample R obtained by choosing every element of N independently with probability r/n,*

$$E[\sum_{\triangle} l(\triangle)] = O(\frac{n}{r} \cdot E[|H(R)|])$$

where $E[|H(R)|]$ is the expected number of ranges in $\mathcal{T}(R)$.

Proof: Again we will prove it specifically for trapezoidal maps; extension to other problems dealt here is straightforward. Note that the segments in $L2(\triangle)$ present no complications since these segments lie exactly in one trapezoid so that $\sum_\triangle |L2(\triangle)| \leq n$. As $E[|H(R)|]$ is $\Omega(r)$, this is subsumed by the right hand side. So we focus only on $L1(\triangle)$ and in the remaining proof $l(\triangle)$ will denote the segments in $L1(\triangle)$ only unless otherwise stated. Again, let $\mathcal{L}$ be the union of vertical lines through the end-points or the intersection points and the input segments. For a fixed line in $\mathcal{L}$ look at the ordered intersections of N with this line. We associate a random variable X_i with each such intersection i that represents the number of unsampled segments of N before the first sampled segment in the upward direction. Let Y_i be corresponding the random variable for the downward direction. (The cardinality of the index set of i can be bounded by a polynomial in n and we do not care how they are numbered.) Since each segment is sampled independently with probability $\frac{r}{n}$, X_i has a geometric distribution with expected value $E[X_i] \leq \frac{n}{r}$. The inequality takes care of the case where there aren't $\frac{n}{r}$ intersections above i. The same holds for the Y_i.

The number of $\triangle \in H(R)$ is itself a random variable so that from the law of conditional expectation

$$E[\sum_{\triangle} l(\triangle)] = \sum \Pr[|H(R)| = k] \cdot k \cdot E[l(\triangle)]$$

$$= E[|H(R)|] \cdot E[l(\triangle)]$$

where $E[l(\triangle)]$ denotes the expected size of $l(\triangle)$ for $\triangle \in H(R)$. The expected number of trapezoids $E[|H(R)|]$ depends on the number of sampled segments and also on the number of intersections between segments of R. Although we don't know which $\triangle$ appear in the sample, in the above equation, we have only used the expected cardinality of the set of $\triangle$.

From Claim 3.6, it follows that for each trapezoid $\triangle \in H(R)$, $L1(\triangle)$ is the union of at most six X_i's. Although the X_i's are not independent, from the linearity of expectation

$$E[l(\triangle)] \leq 6 \cdot E[|X_i|] \leq O(\frac{n}{r})$$

□

Claim 3.6 *For any $\triangle \in H(R)$, $L1(\triangle)$ can be written as a union of at most six X_i.*

Proof: For every end-point or intersection point j, associate a trapezoid $\triangle_j$ for the $\triangle \in H(R)$ containing j (or the trapezoid on its right if it lies on the boundary). For distinct j_1 and j_2, $\triangle_{j_1}$ could be the same as $\triangle_{j_2}$. We shall show that for any j, $L1(\triangle_j)$ is the union of at most six X_i. Since every $\triangle \in H(R)$ is $\triangle_j$ for some j, the claim follows. Our arguments are valid for any arbitrary point in the plane and not necessarily an end-point.

For an arbitrary j, let D and L be the first sampled segment in the downward and upward directions respectively. We thus have the upper and lower bounding segments of $\triangle_j$. Let the vertical projections of j on U and D be u_j and d_j respectively. The set of segments intersecting $\triangle_j$ intersects L, D or one of the two vertical boundaries (which we haven't discovered yet). The segments intersecting U within $\triangle_j$ is the union of X_{u_j} and Y_{u_j}. Likewise the segments intersecting D is the union of X_{d_j} and Y_{d_j}. See Figure 2 for an illustration.

The segments intersecting the right vertical boundary of $\triangle_j$ can also be bounded by union of X_r and Y_r where r (an end-point of some segment) defines the right-boundary of $\triangle_j$. The argument for left-boundary is identical. This gives a bound of eight X_i's (two for each boundary of $\triangle_j$). It can be improved to six by walking around the boundary of $\triangle_j$ rather than looking at the four boundaries separately. In this claim we have not distinguished between X_i and Y_i which are identical for our purpose except for the geometric orientations.

In the proof we used a hypothetical *ordering* in the way the samples are selected (for example by coin-tossing) so as to 'discover' the relevant trapezoid. Since $H(R)$ is not dependent on the order in which the individual segments are selected, the bounds are not dependent on it either. □

The result of Lemma 3.5 can be extended to the case where we are interested in higher moments, namely $E[\sum_{\triangle} l^c(\triangle)$ for $c \leq 1$. Unfortunately the previous proof-technique is unable to deal with segments in $L2(\triangle)$. Our arguments only apply to situations where all the elements intersecting a range $\triangle$ also intersects a boundary of $\triangle$. The proof given below assumes that $L(\triangle) = L1(\triangle)$. This is true for the constructing intersection of half-spaces and arrangement of lines. For a more general proof, the reader can consult [30, 67] of the bibliographic references.

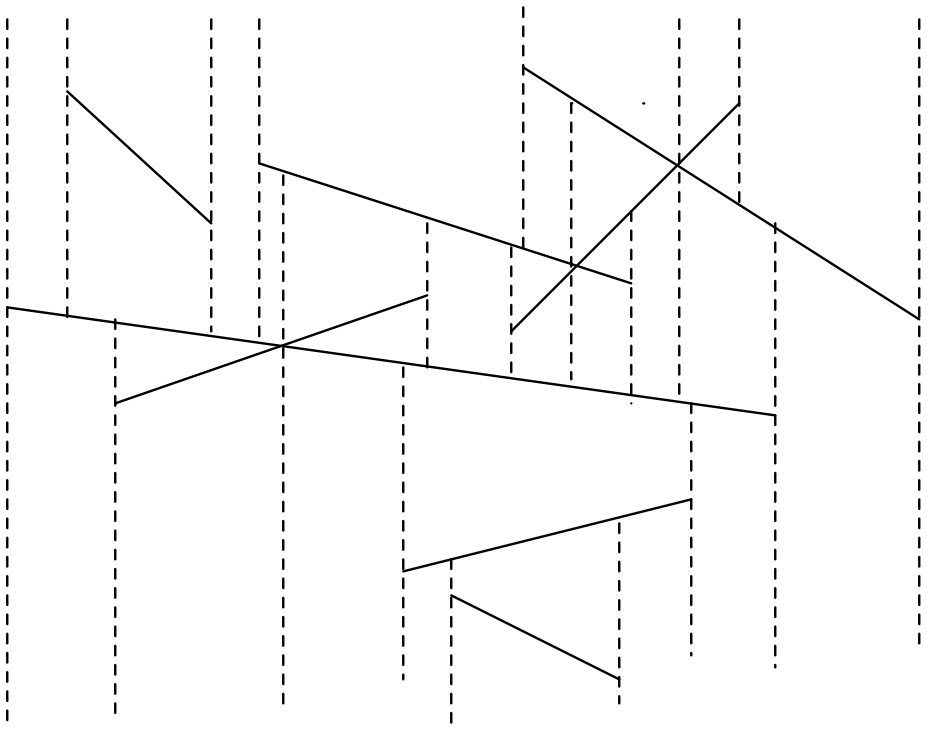

Figure 1: Trapezoidal map of a set of line segments

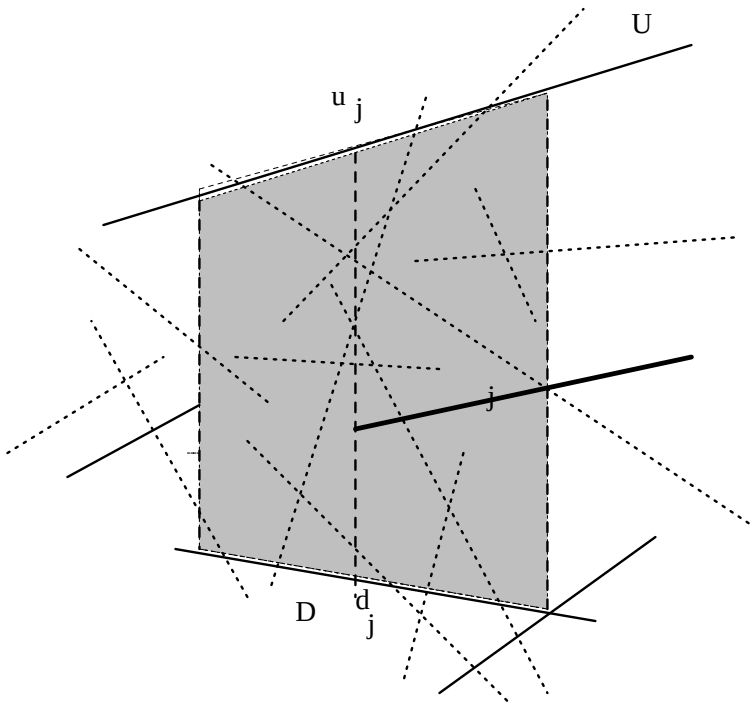

Figure 2: Segments intersecting $\triangle_j$

Lemma 3.7 *For $|N| = n$ and a random sample R obtained by choosing every element of N independently with probability r/n,*

$$E[\sum_{\triangle} l^c(\triangle)] = O(\left(\frac{n}{r}\right)^c \cdot E[|H(R)|])$$

where $E[|H(R)|]$ is the expected number of ranges in $\mathcal{T}(R)$ and c is a fixed positive integer (independent of n).

Proof: From the proof of Lemma 3.5, it follows that

$$E[\sum_{\triangle} l^c(\triangle)] = O(E[X_i^c]) \cdot E[|H(R)|]$$

where X_i has a geometric distribution. It can be shown that for a geometric random variable with success probability p, the c-th ordered moment is

$$\sum_{i=1}^{c} O(\left(\frac{1-p}{p}\right)^i)$$

where the constant hidden behind O is about i^i. For fixed c and $p = \frac{r}{n}$, the required bound follows. □

The above bound is useful in situations where the algorithm's performance can be expressed as $E[\sum_{\triangle} l^c(\triangle)]$ (cf. section 4.4).

3.2 Converting expected bounds to high-probability

As discussed in the beginning of this section, the probabilistic recurrence relation arising from a recursive randomized algorithm are often hard to solve without the knowledge of the tail distributions. The parallel algorithm's running time is the maximum of the resulting recursive calls unlike the sequential algorithm where it is a sum of the running times of the recursive procedures. The latter becomes easier to bound from the linearity of expectations. No such nice properties are known about the maximum of expected values and it does appear to depend on the tail estimates. This informal discussion is a motivation for the results obtained in this subsection; we do not claim that this is the only way to get around the problem of bounding maximum of expected values.

The naive random sampling gives us a high-probability bound on the maximum size of a subproblem (Lemma 3.2) and only an expected bound on the sum of subproblems (Lemma 3.7). We can restate these results in the following manner.

Lemma 3.8 *For suitable constants k_{total} and k_{max} the following conditions hold with probability at least 1/2 for a sample where each of the n input elements has been selected independently with probability r/n.*

(i) The maximum size of a subproblem is less than $k_{max}\frac{n}{r}\log n$
(ii) The sum of the subproblems is less than $k_{total}\frac{n}{r} \cdot E[|H(R)|]$.

Proof: From Lemma 3.5 and Markov's inequality we can choose k_{total} such that the probability that (ii) fails is at most 1/3 (i.e., sum of subproblems is thrice the expected value). Choose k_{max} such that failure probability in Lemma 3.2 is no more than $1/n$. 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. □

Remark 3.9: We can make additional conditions like bounding the higher moments hold with probability at least 1/2 by the above argument. A sample that satisfies a required set of conditions will be called a 'good' sample.

As a consequence of the previous claim, if we repeat the sampling $t \log n$ times, the probability that the conditions are not satisfied for all samples is less than n^{-t}. That is, if we choose independently $p(n) = O(\log n)$ sets of samples, one of them is good with very high likelihood. Let us assume that we have a verification procedure $\mathcal{V}$ that verifies if a sample is good using $v(n)$ operations for n input elements. Therefore, to determine if a sample is 'good', we have to run $\mathcal{V}$ $O(\log n)$ times for a total of $v(n) \log n$ operations. Depending on $v(n)$, this could make an algorithm inefficient. To make resampling more efficient, we can run $\mathcal{V}$ using only a fraction of the input for each of the samples. For example, we can estimate the size of a subproblem by looking at a fraction of the input. Recall that the method of Lemma 2.12, the Estimation Lemma, gives constant factor estimates from which we can estimate the sum of the subproblem-sizes. Below, we describe a method based on this observation where the main difference is that we have to bound the estimate from both sides.

For exposition, we describe it for the special case of trapezoidal maps - for other problems the same arguments apply with minimal modifications, by replacing trapezoids with appropriate ranges.

We choose $c_0 \cdot n/\log^d n$ input segments randomly from the n input segments for some fixed integer $d > 2$ and a constant c_0 (the actual value will be determined from the required success probability of the algorithm). Let X_i^j be the number of segments intersecting trapezoid $\triangle_i$ 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 segments intersecting $\triangle_i$ out of the $c_0 n \log^d n$ randomly chosen input segments 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) X_i^j/n (probability of success). Assuming that X_i^j is greater than $\bar{\mathrm{c}} \cdot \log^{d+1} n$, for some constant $\bar{\mathrm{c}}$, we will apply Chernoff bounds to tightly bound the estimates within a constant multiplicative factor. Since we do it only for $1/\log^d n$ of the input segments, the total number of operations for the $O(\log n)$ random subsets can be bounded by $O(v(n/\log^d n) \cdot \log n)$. We will see in the next section that for the problems at hand, this quantity is $o(v(n))$. Note that $X_i^j < \bar{\mathrm{c}} \log^{d+1} n$, is an easy case since $n^{\epsilon} \cdot \bar{\mathrm{c}} \log^{d+1} n = o(n)$ for sample size $= O(n^{\epsilon})$ for $\epsilon < 1$.

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, $\mathrm{Prob}(A_i^j \leq \alpha c_1 X_i^j/\log^d n) \leq 1/n^{\alpha}$ and $\mathrm{Prob}(A_i^j \geq c_2 \alpha c_0 \cdot X_i^j/\log^d n) < 1/n^{c_0 \alpha} < 1/n^{\alpha}$ (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 choice of the constants, this condition holds with high likelihood (as defined in section 2.1) for all X_i^j simultaneously. We do the following procedure simultaneously for all the samples R_j and choose the sample R^{j_o} using the following simple test:

Procedure Polling

Input: Samples $R_1 \ldots R_m$ where $m = O(\log n)$.

Output: A good sample R^{j_o}.

Notation: 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. We will use S to denote $k_{total}n/rE[|H(R)|]$.

(clearly good)
If $S > U^j$ **then** accept sample R^j (since $S \geq U^j \geq T^j$),

(clearly bad)
if $S \leq L^j$ then the sample is 'bad' (since $S \leq L^j \leq T^j$),

(choose the best)
if $L^j \leq S \leq U^j$, then accept the sample R^{j_o} for which N^{j_o} is minimum. Since both S and T^{j_o} lie in this interval this guarantees that $T^{j_o} \leq c_3 \cdot S$ 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 3.10 (Polling lemma) *Using procedure* **Polling**, *we can obtain a sample that is 'good' with high probability where a naive random sample is known to be 'good' with probability 1/2. Given a verification procedure $\mathcal{V}$ that runs in $O(v(n))$ operations for n elements, this procedure uses $O(v(n/polylog(n))\log n)$ operations.*

Remark 3.11: From our definition of a good sample, the sum of the segments should be less than S, where as the output of the Polling algorithm could be larger by a factor of c_3. Strictly speaking, one needs to modify the definition to accommodate an extra factor c_3. However, it should be clear that we have succeeded in our objective of choosing a sample for which the sum of subproblems is $O(\frac{n}{r} \cdot E[|H(R)|])$ with high probability.
The above procedure can be used in a more general situation where we need 'good' samples with a bound on higher moments whose expectations are known from Lemma 3.7.

4 Randomized divide-and-conquer

The tools and techniques developed in the previous two sections will be used to design a very general scheme for parallel divide-and-conquer. Random sampling will be used to achieve fairly even partitioning of the problem and the analysis will be done using various properties of random sampling proved earlier. We will illustrate the general methodology using the problem of computing trapezoidal map of line segments that can intersect only in the end-points. The parallel model that will be used is CREW unless otherwise mentioned.

4.1 Trapezoidal map construction

This problem is the same as defined in previous section except that the segments are non-intersecting except possibly in the end-points. For every end-point, we want to determine the segment lying immediately above and below. This problem is called the *vertical visibility map* and is particularly interesting because of its close connection to triangulation. The algorithm we will describe will construct the visibility map of a given set N of line segments. A very high-level description is as follows:

Algorithm Vertical Visibility

1. Select a 'good' sample R of size $O(n^\epsilon)$ ($\epsilon > 0$ is a constant that will be determined in the analysis). By 'good' sample, we imply that the conditions of Lemma 3.8 hold. We use the technique of polling to do this step efficiently.
2. Construct $\mathcal{T}(R)$ using a brute-force approach.
3. For each segment of N, determine the trapezoids of $\mathcal{T}(R)$ that it intersects by a procedure we will describe shortly. (The same procedure will be used as the verification algorithm for Polling in step 1).
4. For each trapezoid $\triangle \in \mathcal{T}(R)$, we apply a clean-up phase called *Filtering* to discard some of the segments in $L(\triangle)$. As we will show later, this phase is crucial for bounding the processor complexity.
5. If $l(\triangle) > C$, for some predefined threshold C, then call the algorithm recursively on $L(\triangle)$ **else** solve the problem directly (We assume that a suitable algorithm already exists - usually a brute-force method suffices).

We now look at the individual steps in some details. The procedures in steps 1-3 are actually quite related. For $\epsilon < 1/2$, the following brute-force method works. For every endpoint, we draw a vertical line and order the segments intersecting this line (sorting in the Y direction of the intersections suffices which can be done in $O(\log n)$ time using n processors for every end-point). From this information, we can determine the segment lying immediate above and below every end-point. From this, we can also compute easily for each segment all the end-points for which it is visible from above (and below). In fact we order the vertical projections of these end-points by sorting. Constructing individual trapezoids can be done by 'walking' around the vertices of a trapezoid using the successor information from the previous computation. So the entire computation can be done in $O(\log n)$ time using $n^\epsilon \cdot n^\epsilon \leq n$ processors.

4.1.1 Partitioning the problem

To determine the segments intersecting a trapezoid, we use a *locus* based approach. 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 $\mathrm{O}(n^c)$ processors, where c is a fixed constant. Thus any sample of size less than $n^{1/c}$ will suffice.

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. These produce 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 the procedure since they would only make the partition more fine (See Figure 3 for an illustration). These $\mathrm{O}(n^2)$ lines can intersect

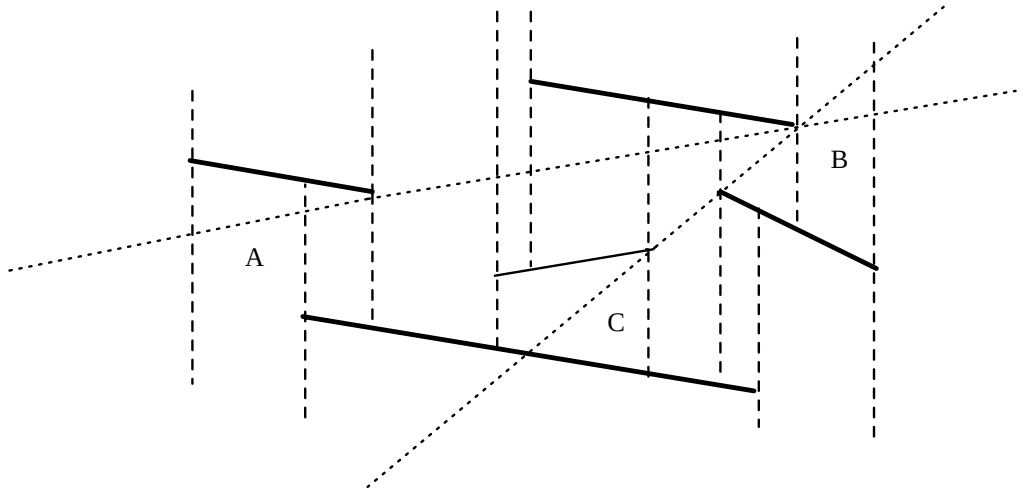

Figure 3: There is a clear path between regions A and B where as there cannot be a segment with end-points in regions C and B. This redundant partitioning does not affect the asymptotic complexity of the algorithm.

in $\mathrm{O}(n^4)$ vertices, giving rise to a $\mathrm{O}(n^4)$ regions in the partition. This implies that there are $\mathrm{O}(n^8)$ possible pairs of regions where end-points of a segment could lie and one can precompute for each such pair which trapezoids the corresponding segment intersects using $\mathrm{O}(n^9)$ processors in $O(\log n)$ time.

For the point location problem, we use a preprocessing scheme first proposed by Dobkin and Lipton. The following result is a straightforward consequence of their paper.

Lemma 4.1 *Given a set of m line segments in a plane, it is possible to preprocess 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 $\mathrm{O}(m^2)$ 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). For point search, we first locate the x interval using a binary search and then within an interval, we do a binary search on the lines that are totally ordered. □

Remark 4.2: In d dimensions, m hyper-planes can be preprocessed in m^{2^d-1} processors in $O(d \log m)$ steps that allows point searching in $d \log n$ steps. The idea is to take intersections of every pair of hyper-planes and project them in $d-1$ dimension. Within each region of the $d-1$ dimensional arrangement, the original hyper-planes can be totally ordered, so that one additional binary search suffices after inductively locating the query point in $d-1$ dimensions in $(d-1)\log m$ steps. The processor and the space complexity grow as claimed as they get squared for every level.

In this case, $m = n^2$, corresponding to the n^2 linear constraints and the regions induced by the preprocessing (trapezoidal regions within an interval), are finer partitioning of the of regions induced by the n^2 lines. Since there are O(n^6) 'finer' regions, we require O($n^6 \cdot n^6 \cdot n$) processors to precompute the possible intersections. This takes care of all possible placements of the two end-points of a segment. Since each region is a trapezoid, we choose a sample point in each trapezoid to do the precomputing 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 requires 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). All the segments belonging to a trapezoid can then be packed into contiguous locations by an application of semisorting (see section 2.4) with labels that identify the trapezoids.

4.1.2 Clean-up for processor bound

Recall that a good sample implies the total sub-problem size is within a constant factor of $n/r \cdot E[|H(R)|]$. In this case (for non-intersecting segments), $H(R)$ is $O(|R|)$, i.e., the total subproblem size will be $\leq kn$ for some constant k. So, after recursion depth i, the total subproblem size can be bounded only by k^i. This algorithm has a recursion depth $O(\log \log n)$ and hence the total sub-problem size could become $\Omega(n\text{polylog}(n)$. This quantity is related to the number of processors; so we wish to bound it by $O(n)$. This is achieved by the following clean-up phase that we call *Filtering*

After partitioning the segments into the trapezoidal regions $\triangle$, we group the segments $L(\triangle)$ into two categories:

(a) $L^1(\triangle)$:Segments (part-segments) that have at least one end-point in the region
(b)$L^2(\triangle)$: Segments that span the region (horizontally)

Notice that number of segments of type (a) is less than $2n$ and $L^2(\triangle)$ in trapezoid $\triangle$, can be completely ordered (with respect to y-coordinate) within $\triangle$. So for the end-points of type $L^1(\triangle)$, a straight-forward binary search suffices to find the nearest segments among $L^2(\triangle)$ in the vertical direction. Consequently, we do not further preprocess the segments in $L^2(\triangle)$, 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 allocating processors in a region equal to the number of end-points lying in it. This ensures that the algorithm does not require more than $O(n)$ processors for any step.

4.1.3 Analysis

The algorithm is recursive and we have shown that every step can be done in $\tilde{O}(\log n_i)$ steps for a subproblem of size n_i using n_i processors. The total processor requirement never exceeds n. The reader can verify that the time complexity T_i at depth i satisfies the preconditions of Theorem 2.7. Hence it follows that

Theorem 4.3 *Algorithm Vertical Visibility executes in $\tilde{O}(\log n)$ steps using n CREW processors.*

4.2 The general strategy

The algorithm developed for visibility maps in the previous section can be used for a number of other problems with some modifications required in the context of the problem. The following gives a bird's eye-view of this approach.

1. **Good-sampling** We use the method of Polling to choose a sample satisfying certain properties that holds with only constant probability for a naive sample.
2. **Partitioning** For computing the subproblems for a given sample. This step is also used in conjunction with the previous step for verifying if a sample is good. The most common partitioning method used is doing point-location in a parameterized space.
3. **Filtering** Keep the total problem size at any stage of the algorithm within the given processor bounds by pruning individual subproblems before recursively solving them. This step is the most problem-dependent and depends on the geometric properties of the problem.
4. Solve sub-problems recursively if the size exceeds a certain threshold.

We will illustrate this approach in the context of another problem, namely, construction of convex hull of n points in three dimensions.

4.3 3-D Convex hull

We shall actually solve the dual problem, namely the intersection of half-spaces in three dimensions. This dual transformation $\mathcal{D}$ maps a point in E^d to a non-vertical hyper-plane 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 hyper-plane $1 = \pi_1 x_1 + \pi_2 x_2 + ...\pi_d x_d$ and vice-versa such that a hyper-plane h not containing the origin is mapped to a point p for which $\mathcal{D}(p) = h$.

The transform $\mathcal{D}$ is extended to sets of points (hyper-planes) 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 hyper-planes that avoid some convex region around O. The dual of $\mathcal{P}$ is defined as

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

where h^{pos} denotes the half-space containing the origin. The following observation can be verified [35]:

Lemma 4.4 *Point p belongs to the $int\mathcal{P}$, $boundary\mathcal{P}$ or $complement\mathcal{P}$ if and only if the hyper-plane $\mathcal{D}(p)$ avoids $\bar{\mathcal{P}}$, avoids $int\bar{\mathcal{P}}$ but not $\bar{\mathcal{P}}$, or intersects $int\bar{\mathcal{P}}$ respectively.*

In other words, given a set of points S, the vertices of the convex hull are the dual transform of the facets of the intersection of the half-spaces $\mathcal{D}(S)$. 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). Here, we actually derive an algorithm for constructing the intersection of half-spaces. Moreover, the dual transform will have applications for the partitioning step.

In the remainder of the section we shall assume that the half-spaces are described as inequalities of the form $a_i x + b_i y + c_i z + d_i \geq 0$. We shall also use the terms 'half-space' and its bounding 'plane' interchangeably where it is clear from the context. The output is a list of vertices of the polyhedron $\mathcal{C}$ which is the intersection of the half-spaces. The vertices are defined by the 3-tuples of the three intersecting planes defining the vertex. We assume that the planes are in general position, that is every vertex is the intersection of exactly three planes. This assumption is for the convenience of analysis and not a real bottleneck for the algorithm. There are standard perturbation techniques which simulates the non-degeneracy condition. For example, changing the coefficients by a sufficiently small real value which is chosen randomly satisfies the property of non-degeneracy with probability approaching 1. The textbook [35] discusses symbolic perturbation techniques which are deterministic but more expensive. The edges of this polytope 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 enable us to obtain the faces and edges as the necessary adjacency structure of the polytope (which is a planar graph).

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 4.5 *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 of of the output convex polytope (of the intersection). For each vertex, test whether it is a vertex of the convex polyhedron by checking if it satisfies all the equations defining the half-spaces. This can be done trivially in $O(\log n)$ time using $O(n)$ processors for each candidate point. Only the vertices would survive. Determine the faces of the convex polyhedron by identifying planes that contain 3 vertices of the intersection. The necessary adjacency structure can be constructed by an application of sorting. □

The algorithm given below is derived from the general strategy in the context of the convex-hull problem.

Algorithm **3-D Hull**

Input: A set S of n half-spaces H_1, $H_2, \ldots H_n$, that contain point p^*
Output: The output convex polyhedron $\mathcal{C}$ 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 half-spaces in R using Lemma 4.5. Take a fixed plane and cut up each face of the polyhedron with (parallel) translates of this plane passing through the vertices (Example 2, section 3.1). 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

pyramid $\triangle_i$ formed by p^* as the apex and F_i as the base. Let C_R denote the set of pyramids. Note that $|C_R| = O(|R|)$.
(3) For the remaining $S - R$ half-spaces find the intersection of the planes (bounding these half-spaces) with the pyramids. Note that a plane may intersect more than one pyramid. The intersection of the S half spaces is the union of the intersection of the half-spaces intersecting a pyramid (over all pyramids). That is, $\mathcal{C}$ is $\cup_{i=1}^{C_R} I_i$ where I_i is $\cap_j \{H^j\}$ restricted to $\triangle_i$.
(4) If the number of planes intersecting a pyramid is more than a pre-determined threshold apply steps 1-3 recursively to this pyramid for the set of half-spaces (bounded by the planes) else solve the problem directly.

Recall from section 3.1 (Example 2), that for this problem, the ranges are the pyramids. The size bounds of Lemma 3.2 would hold with respect to the number of planes intersecting a pyramid. In the context of Lemma 3.5, $|H(R)| = O(r)$ because the skeleton of intersection of R half-spaces is a planar graph with at most $|R|$ faces. This implies that the sum of sub-problem sizes is $O(n)$ in a good sample. Polling is used in conjunction with steps 1-3 to achieve this.

4.3.1 Finding intersections quickly

We now focus on a procedure to find the intersection of planes with each of the pyramids, $\triangle_i$. We shall use a *locus-based* approach to solve this problem. In this case, we have to preprocess the convex polytope of the sampled half-spaces in such a way that given any plane, we should be able to report the list of pyramids 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 preprocessing for a convex polytope of size n 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. For this problem, a more precise value of c will emerge during the analysis. Given a convex polytope in three-dimensions of size $O(n)$ along with an internal point which is the apex of the pyramids, there are only a polynomial (in n) number of combinatorially distinct possibilities of the way any given plane can intersect the pyramids. This can be seen from the following simple argument. Given any plane that intersects the polyhedron, we can perturb the plane without changing the pyramids it intersects so long as it remains within a fixed set of bounding vertices. Figure 4 illustrates the situation for a two-dimensional case. If we consider an equivalence relation where two lines are equivalent iff they intersect the same sets of pyramids, 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 n-gon or internal point and $\mathcal{D}$ is a dual transform $\}$

Given any query line l, the pyramids 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 in any dimension; in this 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 pyramids they intersect. Notice that even if this partitioning may not be minimal it suffices for our purpose. All that remains to be done is precompute for each of these regions the pyramids 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.

Recall the method of Lemma 4.1 and its generalization in Remark 4.1.1, the entire preprocessing for point location for n planes in three dimensions can be done in n^7 space and processors in $O(\log n)$ time. For each of the cells in 3-space, we can precompute the pyramids that the corresponding

plane intersects using $O(n^8)$ processors (by choosing a representative point in each cell and testing it against all the pyramids). Note that these subdivisions are finer than the minimal equivalence classes i.e., more than one subdivisions could have the same set of intersecting pyramids. We also store the number of intersecting pyramids for each of the subdivisions so that while listing the number of pyramids each query plane intersects we can do the processor allocation easily in $O(\log n)$ time using a prefix computation. By choosing $|R|$ less than $n^{1/8}$, we can complete the entire preprocessing in the required time and processor bounds. So the entire procedure takes $O(\log n)$ time.

4.3.2 Filtering

Consider a pyramid $\triangle$ (we will not use the subscripts unless necessary). We will denote the set of half-spaces intersecting $L(\triangle)$. In this step, our goal is to identify half-spaces in $L(\triangle)$ that would not contribute to any vertices of $\mathcal{C}$ within $\triangle$. In the discussion below, we will often refer to the half-spaces as planes (bounding the half-spaces).

After we have found the planes intersecting $\triangle$, we categorize them as following:

(a) planes that are completely occluded by another plane in the $\triangle$ and hence these cannot be a part of the output in the $\triangle$,
(b) planes that are occluded because of more than one other plane in the $\triangle$ 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 pyramids,
(d) planes that do contribute to a vertex in $\triangle$.

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. In case there is such a point q, q is said to *dominate* p.

Since $\triangle$ has a triangular base there are 3 edges that join it to the apex p^*. We sort intersections of the planes in $L(\triangle)$ 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 $\mathrm{r}(X_i)$, $\mathrm{r}(Y_i)$ and $\mathrm{r}(Z_i)$.

Observation 4.6 *If a plane A is occluded completely by another plane B iff it is dominated on its ranks of intersection on all the three edges by plane B.*

This gives 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. We can do this in $O(\log n)$ time using a linear number of processors (using an algorithm of [6]).

Remark 4.7: Elimination of this step does not change the asymptotic bounds of our algorithm. However, it may eliminate some more planes going into the recursive call thereby improving the performance in practice.

To identify planes of type (b) (c) and (d) we construct the intersection of the convex polytope $\mathcal{C}$ with each of the three faces of $\triangle$. These are intersections of the faces with $\mathcal{C}$ that are 2-D convex polygonal chain. 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. Note that 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 $\mathcal{C}$. Denote the set of planes in $L(\triangle)$ that define the contours as $L^c(\triangle)$ and its complement by $L^{nc}(\triangle)$. Consequently, a plane of type (b) cannot be in $L^c(\triangle)$. 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 portion of the polytope within $\triangle$.

A plane in $L^c(\triangle)$ may not contribute to any vertex of the convex polytope $\mathcal{C}$, that is it only contributes a edge of the hull within $\triangle$. In this case the edge intersects $\triangle$ in exactly two faces and the these vertices can be labeled by the two intersecting planes (which contributes to the output edge). Thus these planes can be identified quickly using sorting on the labels of the intersecting planes.

We say that a plane $h \in L^{nc}(\triangle)$ and a contour *generates* the ray e that originates at the closest point p on that contour to h and is contained in the (at most two) bounding planes in $L^c(\triangle)$ that defined p. The point p can be identified by doing a binary search on the vertices of the contour. If the contour does not contain a vertex then choose any ray on the plane that defines the contour.

Observation 4.8 *If a ray generated from $h \in L^{nc}(\triangle)$ and a contour of $\triangle$ does not intersect h, then h does not contribute to a vertex of $\mathcal{C}$.*

We say that a plane $h \in L^{nc}(\triangle)$ is *pinned* to $\triangle$ iff it is intersected by all the three rays generated from h and the contours. From convexity, it follows that

Observation 4.9 *A plane $h \in L^{nc}(\triangle)$ can be pinned to at most one $\triangle$.*

From the previous observation, it follows that we need to call the algorithm recursively only on those $h \in L^{nc}(\triangle)$ that are pinned to $\triangle$.

The number of processors allocated is on the basis of the *potential* number of vertices in $\triangle$. This is calculated as follows. We shall use the term 'flattening' to imply that the vertices of the contour are projected along edges (intersection of two planes) they lie on, such that all of them become coplanar. Notice that there may be several such planes. We just choose one arbitrarily and these projected vertices defines a 'base' face. We wish to 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 polytope to be $3|V|$ where V is the number of vertices of the convex polytope. Since the surface of the convex polytope is a connected planar map, we can use Euler's equation to show that $|V| = 2|F| - 2$ where F is the set of faces in the polytope. Since $|F| \leq n$, $|V| \leq 2n - 2$. This calculation is done using the property that each vertex is of degree 3 (which follows from our assumption that the planes are in general position). Let the number of processors be $6n$. We distribute the processors among the subproblems depending on the output size. For a pyramid $\triangle$ that does not contain any *free-edge*, the output can be bound by the following:

Claim 4.10 *The output size of $\triangle$ is bounded by $3n_i + 6m_i - 6$, where n_i is the number of planes in the contour contributing at least one vertex and m_i 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 $\triangle$ (the vertices of the contour are these edges). Let e_2 be the number of edges which lie

within $\triangle$ (including both end-points). Let v be the number of vertices of $\mathcal{C}$ within $\triangle$(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 m_i planes and a 'base' face by 'flattening' the contour. Refer to the explanations of the terms 'base' face and 'flattening' in the previous paragraphs. 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 of of $\mathcal{C}$ that show up in $\triangle$ but are not a part of the contour. Since n_f (type d) can be bounded by m_i the claim is proved. □

Notice that if $\triangle$ contains d *free-edges*, then the above formula can be applied separately to each of the $d + 1$ partitions (induced by the free edges). The participant planes in each of these partitions are disjoint giving us the following:

Claim 4.11 *The output size of $\triangle$ is bounded by $3n_i + 6m_i - 6d$ where d is the number of free-edges.*

The processor allocation strategy is to simply allocate this number of processors to the subproblem (in the pyramid). The total number of vertices over all the pyramids is bound by the maximum output size and by our allocation strategy, we are allocating processors proportional to the maximum output-size in each pyramid. Note that the actual output size may be less but we shall never have fewer processors than required and this maximum size can be actually achieved. Another way to look at the processor allocation strategy is that two processors are allocated to each edge of the output hull and we allocate those processors to the pyramids which contains (or potentially contain) vertices associated with that edge. The *free-edges* are not allocated any processors. Hence we have sufficient number of processors.

4.3.3 Analysis of 3-D hull algorithm

In the previous subsections we showed that every step of the algorithm can be done in $\tilde{O}(\log n_i)$ steps for a subproblem of size n_i using n_i processors. The total processor requirement does not exceed $O(n)$ from our processor allocation strategy. From Theorem 2.7 it follows that

Theorem 4.12 *The convex hull of n points in three dimensions can be constructed in $\tilde{O}\log n)$ time using n CREW processors.*

4.4 Constructing arrangements

Given a set H of n lines, the arrangement $\mathcal{A}(H)$ contains information about the way the plane is partitioned into connected regions by these lines. It is known that in d dimensions, E^d is partitioned into at most n^d connected regions by n hyper-planes, where a connected region could have dimension 0 (points) to d (cells). The textbook by Edelsbrunner (see bibliography) is an excellent source for discussion of the combinatorial properties of arrangements. By construction of $\mathcal{A}(H)$, we will imply a representation of the incidence relation of the faces, where a face of dimension k (called a k-face) is contained in the intersection of at least $d - k$ hyper-planes. For $d = 2$, $\mathcal{A}(H)$ can be considered a planar graph where the vertices are the intersection points of lines, the edges are portions of line between consecutive intersections and the faces are the two-dimensional partitions of the plane. Constructing the arrangement is equivalent to constructing a representation of this graph. This

can be easily done by sorting the intersections on all the n lines which takes $O(n \cdot n \log n)$ steps in all. However there are sequential algorithms that can construct this graph in $\theta(n^2)$ steps and in general $O(n^d)$ in d dimensions. Here we will outline a parallel algorithm that matches the sequential work bound. Although the technique extends naturally to higher dimension our description will be for the case $d = 2$. For simplicity we will assume that no three lines have a common intersection.

We will differ slightly from our previous generic algorithm, namely that we will choose a much larger sample size so that the number of recursive levels will be constant. Consequently, we will do away with the Filtering phase. Note that for n lines, the arrangement can be constructed trivially in $O(\log n)$ time by sorting using n^2 processors. Each 2-face in the arrangement is convex and a *range* is a triangle obtained by triangulating a (convex) face by joining the bottommost vertex (of the face) with the other vertices. This is also known as the **bv**-triangulation. To deal with unbounded faces, we assume that all the intersections are contained within a (symbolic) triangle of appropriate size. Like our previous conventions, $L(\triangle)$ denotes the lines intersecting a triangle $\triangle$ and $l(\triangle)$ is the cardinality of $L(\triangle)$. The algorithm is as follows

Algorithm 2-D arrangement

1. Choose a good* sample R of $\frac{n}{\log^2 n}$ lines. By good*, we imply a sample that satisfies the properties of Lemma 3.8 and in addition $\sum_{\triangle} l^2(\triangle) = O(n^2)$.
2. Find the lines in $H - R$ intersecting each range (triangles corresponding to the bv-triangulation of $\mathcal{A}(R)$).
3. For each $\triangle$, such that $l(\triangle) \leq \sqrt{\log n}$, construct the arrangement using an optimal sequential algorithm.
4. For each triangle $\triangle$, such that $l(\triangle) \geq \sqrt{\log n}$, choose a good* sample $R(\triangle)$ of size $s(\triangle) = l(\triangle) \cdot \log(l(\triangle))/\sqrt{\log n}$. Here the random sample is good* with respect to $L(\triangle)$. Denote the set of triangles in the bv-triangulation of $\mathcal{A}(R(\triangle))$ by $T(\triangle)$.
5. For a $\triangle$, if $\max_{t \in T(\triangle)}\{l(t)\} \leq \alpha\sqrt{\log n}$ and $\sum_{t \in T(\triangle)} l^2(t) = O(l^2(\triangle))$, then for all triangles $t \in T(\triangle)$ obtained after step 4, construct the arrangement of the lines $L(\triangle) - R(\triangle)$ intersecting t using an optimal sequential algorithm.
6. For the remaining $\triangle$, construct the arrangement of the lines $L(\triangle)$ intersecting $\triangle$ directly using sorting.

For a random sample R of size r, $|H(R)| = O(r^2)$ since the size of $\mathcal{A}(R)$ is r^2. Thus from Lemma 3.5, $E[\sum_{\triangle} l(\triangle)] = O(nr)$ and from Lemma 3.7 $E[\sum_{\triangle} l^2(\triangle)] = O(n^2)$. Hence Polling can be used to select a good* sample. From Lemma 3.2, for each $\triangle$,

$$l(\triangle) = O(\log^3 n) \tag{7}$$

The actual partitioning step is carried out by sorting the intersections on the $\frac{n}{\log^2 n}$ lines. This can be done in $O(\log n)$ time using $n^2/\log^2 n$ processors. More generally, for a sample of size r, this can be done in $O(\log n)$ time using nr processors. At this point, we know the set of lines in $H - R$ that intersects each face of $\mathcal{A}(R)$. Denote the faces of $\mathcal{A}(R)$ by $\mathcal{F}(R)$. To determine $L(\triangle)$, (in the bv-triangulation of $\mathcal{A}(R)$) we can do a brute force method of checking with respect to each triangle in a face $f \in \mathcal{F}(R)$. The total work done can be bounded by $\sum_{h \in H-R} \sum_{h \cap f \neq \phi} |f|$ where $|f|$ denotes the size of a face

$$= \sum_{h \in H-R} O(r)$$

$$= n \cdot r$$

The first equality follows from Zone theorem that bounds the number of vertices of $\mathcal{A}(R)$ visible from any line by $O(|R|)$.

Lemma 4.13 *The partitioning of n lines into the faces induced by a subset of r lines can be done in $O(\log n)$ time using nr CREW processors.*

Step 3 can be done in $O(\log n)$ steps using one processor per triangle (there are no more than $O(r^2) = O(n^2/\log^4 n)$ triangles from Euler's formula).

In Step 4, the partitioning can be done in $O(\log(l(\triangle))$ steps using $l(\triangle) \cdot s(\triangle)$ processors from Lemma 4.13. Thus total number of processors required is $\sum_{\triangle} l(\triangle) \cdot s(\triangle)$

$$= \sum_{\triangle} l(\triangle) \cdot l(\triangle) \cdot \log(l(\triangle))/\sqrt{\log n}$$

$$= \sum_{\triangle} l^2(\triangle) \log(l(\triangle))/\sqrt{\log n}$$

$$\leq \log\log n/\sqrt{\log n} \sum_{\triangle} l^2(\triangle)$$

The last step follows from equation 7. From the property of good* sample this is less than $n^2 \log\log n/\sqrt{\log n}$. By slowing down from $O(\log\log n)$ to $O(\log n)$ time, the number of processors can be reduced to $n^2 \log\log^2 n/\log^{3/2} n$ which is less than $n^2/\log n$.

Step 5 takes no more than $O(\log n)$ steps for each $t \in T(\triangle)$ and the total work over all such t can be bounded by $\sum_{\triangle} \sum_{t \in T(\triangle)} l^2(t)$

$$= \sum_{\triangle} O(l^2(\triangle))$$

$$= O(n^2)$$

The two equations follow from the property of good* sample with respect to $L(\triangle)$ and n respectively. Thus it can be done in $O(\log n)$ time using $n^2/\log n$ processors after load-balancing.

From Lemma 3.2 applied to each $\triangle$, for some constant α,

$$\Pr[\max_{t \in T(\triangle)} \{l(t)\} \geq \alpha\sqrt{\log n}] \leq \frac{1}{l^4(\triangle)} \tag{8}$$

Also, by resampling and polling, the probability that $\sum_{t \in T(\triangle)} l^2(t)$ exceeds $O(l^2(\triangle)$ is less than $\frac{1}{l^4(\triangle)}$ from Lemma 3.10. For step 6, we bound the work as follows. Since $l(\triangle) \geq \sqrt{\log n}$, the probability that a sample is not good* (with respect to a $\triangle$) is less than $\frac{1}{\log^2 n}$. If $\max_{t \in T(\triangle)}\{l(t)\} \geq \alpha\sqrt{\log n}$ then a sample is chosen again (for $\triangle$). The probability that sampling has to be done is less than $\frac{1}{\log^2 n}$ from equation 8. Since this step is done independently over the $\triangle$, the probability that it has to be repeated for more than $\frac{2}{\log^2 n} \cdot n^2/\log^2 n$ triangles is less than $O(2^{-n/\log^4 n})$ from Chernoff bounds 1.3, equation 6. From equation 7, $l(\triangle) = O(\log^3 n)$. Thus by assigning $O(\log^3 n)$ processors to each $\triangle$, we can construct the arrangement directly (using sorting) in $O(\log\log n)$ steps. The total number of processors required is $n^2/\log n$ for this step.

Theorem 4.14 *The arrangement of n lines in the plane can be constructed in $\tilde{O}(\log n)$ steps using $\frac{n^2}{\log n}$ CREW processors.*

Remark 4.15: 1. We can dispense with Polling in this algorithm as there are enough processors in Steps 1 and Step 4 to do resampling with respect to the entire input in parallel.
2. Practically the same algorithm extends optimally for any fixed dimension d with carefully choosing a representation of the arrangements in higher dimensions.

5 Sublogarithmic algorithms

In the previous section, we described parallel algorithms for trapezoidal-maps and 3-D convex hulls that run in $O(\log n)$ time with high probability using n processors in a CREW model. This is both time and work optimal in the CREW model; even computing **OR** of n bits take $\Omega(\log n)$ time. However using concurrent writes, we can *padsort* in $\tilde{O}(\log n/\log k)$ steps using kn processors (Theorem 2.17). This suggests that we may be able to speed-up the previous algorithms further using faster sublogarithmic parallel routines described in section 3.

The basic idea is as follows. We assume that there are kn CRCW processors for some integer k. We sample roughly $(nk)^{1/c}$ input elements which we use to partition the problem. From our earlier discussion the maximum subproblem size is no more than $\frac{n}{(nk)^{1/c}}$ (actually we are ignoring a logarithmic factor which can be adjusted by choosing slightly larger sample) with high likelihood. The constant c is such, that given n^c processors, one can solve the problem in constant time. For example, in the vertical visibility problem, c is no more than 2 since one can determine using n processors per end-point which are the closest segments above and below. We shall show how to do each step of the general algorithm, namely the partitioning Polling and Filtering in $\tilde{O}(\log n/\log k)$ steps.

Consider the following recurrence relation whose solution will be the crux of our analysis of the sublogarithmic algorithms. Let $T(n,m)$ represent parallel running time for input size n with m processors.

$$T(n,nk) = T(\frac{n}{(nk)^{1/c}}, \frac{nk}{(nk)^{1/c}}) + a\log n/\log k$$

Here c and a are constants larger than 1. The recurrence arises from the following property of the algorithms : when $m = nk$, the maximum subproblem size is no more than $\frac{n}{(nk)^{1/c}}$ with the processor advantage $\geq k$.

It can be easily verified by induction that the solution of the above recurrence with appropriate stopping criterion is $O(\log n/\log k)$. Technically, we cannot use a deterministic solution of this recurrence directly for our purposes as our bounds are probabilistic. So we use a technique which is an extension of the solution described in the proof of Theorem 2.7. View the algorithm as a tree whose root represents the given problem (of size n) and an internal node as a subproblem. The children of a node represents the sub-problems obtained by partitioning the node (by random sampling) and the leaves represent problems which can be solved directly without resorting to recursive calls.

Denote the time taken at a node at depth i from the root by T_i.

Lemma 5.1 *Suppose* $\Pr[T_i \geq ac\alpha\epsilon^i \log n/\log k] \leq 2^{-\epsilon^i \log n c\alpha}$ *where a, c are constants and α a positive integer. Then all the leaf nodes of the tree representing the algorithm terminate within T steps, such that Prob[$T \geq \alpha \log n/\log k$] $\leq n^{-f\alpha}$. where f is a constant.*

The proof is along the lines of Theorem 2.7, so we have omitted the details here. Below we describe the modifications required in the generic algorithm to obtain faster running time and then look at the individual problems.

5.1 Speeding up in the sublogarithmic range

In the generic divide-and-conquer approach described in section 4.2, the Partitioning step is speeded up by using padded-sorting and the faster k-way searching (Lemma 2.19). In the locus-based approach of Lemma 4.1 we obtain the following generalization.

Lemma 5.2 *Given h hyper-planes in d dimensions, a data-structure for point location can be constructed in $\tilde{O}(d \cdot \log n / \log k)$ time using $k \cdot n^{2^d-1}$ processors. This data-structure can be used to do point location in $O(d \cdot \log n / \log m)$ steps using m processors for each point.*

The basic procedure for resampling and Polling is inherently parallel as the samples are tested concurrently. In fact we can dispense with Polling for $k = \Omega(\log n)$ since there are enough processors to test the entire input against a sample. Similarly Filtering becomes redundant once $k \geq \log^t n$ for some constant t from the observation that the algorithms have recursive depth $O(\log \log n)$ so a constant factor blow-up implies the above bound.

A more critical issue is that of processor allocation. Recall that a common scenario for the algorithms is the following. Suppose s is the number of subproblems ($s < n$) and each of the input elements for the subproblems has been tagged with an index in $1...s$. Then these can be sorted on their indices into an array of size $S(1+\lambda)$ from the previous theorem where S is the sum of the sizes of the subproblems. A processor indexed P is associated with the element in the cell numbered $\lceil P/S \rceil$ [2]. In most cases, $S = n$, so that if we have kn processors, then the number of processors allocated to a subproblem i of size s_i is at least $s_i \cdot k/(1+\lambda)$. The *processor advantage* which is defined to be the ratio of the number of processors to the subproblem size, is not as good as it was initially, namely it is $k/(1+\lambda)$ instead of k. However, for our purposes it will make little difference because of the property that the number of recursive levels in the algorithm will be bounded by $O(\log \log n)$. Hence the processor advantage at any depth of the recursion is no worse than $k/(1+\lambda)^{O(\log \log n)}$ which is still $\Omega(k)$. In our future discussions, we shall implicitly use this property for processor allocation.

5.2 Trapezoidal decomposition and triangulation

We modify Algorithm Vertical Visibility by substituting the above routines for partitioning and processor allocation. For the Filtering step, recall that it basically involves discarding segments that span the entire trapezoid. This is basically compaction that can be done in $\tilde{O}(\lceil \log^* n - \log^* k \rceil)$ time using Lemma 2.22 ($= o(\log n / \log k)$). It follows

Theorem 5.3 *The trapezoidal decomposition of n non-intersecting segments can be constructed in $\tilde{O}(\log n / \log k)$ steps using kn CRCW PRAM processors.*

[2] We will avoid using the ceiling and floor functions when it is clear from the context

5.3 3-D Convex hulls and 2-D Voronoi diagrams

An almost identical approach works for algorithm 3-D hull. The Filtering step is essentially locating the half-spaces that are *pinned* to a pyramid. To find the closest point in a contour to a half-space, we use k-ary search on the sorted vertices of each contour. This is clearly done in $O(\log n/\log k)$ steps. Again we use compaction to discard the redundant half-spaces within a pyramid. As mentioned previously, the 3-D maxima routine can be dispensed with without affecting the asymptotic bounds.

Theorem 5.4 *The convex-hull of n points in three dimensions can be constructed in $\tilde{O}(\log n/\log k)$ steps by kn CRCW PRAM processors for $k \geq 1$.*

Note that the output of this algorithm produces a list of vertices in an array of slightly larger size. To compute the adjacency information (the edges), we require an application of integer sorting. The sorting is done on the list of 3-tuples (the planes intersecting in that point) and we have all the six permutations corresponding to a tuple. If the adjacent tuple have two planes in common, they define an edge.

As a consequence of the 'lifting' transformation, we obtain a similar bound for 2-D Voronoi diagram. Here the output is the list of the Voronoi vertices with their adjacency information.

5.4 Lower bounds

To gauge the efficiency of the algorithms described in the previous section, we actually prove matching lower bounds for these in a related model of computation. The material in this subsection can be read independently of the rest of the chapter. The model of computation is the parallel analogue of the *Bounded-degree decision tree model*(BDD Tree). At each node of this tree, each of the p processors compares the value of a fixed degree polynomial with 0. Accordingly each processor gets a a *sign* $\in \{0, +, -\}$ depending on the result of the comparison being $\{=, >, <\}$ respectively. Subsequently the algorithm branches according to the *sign vector*, that is by considering the signs of all the processors. The algorithm terminates when we reach a leaf node containing the final answer. If the polynomials are restricted to be of the form $x_i - x_j \leq 0$, then it is the Parallel Comparison Tree (PCT) model. While there is no cost for branching (that includes processor allocation and read-write conflicts), it does not have the arithmetic instruction set of the PRAM model. So, strictly speaking it is incomparable with the PRAM model. Notably, the known geometric algorithms in the PRAM model do not exploit this extra power so lower bounds in the BDD Tree are often regarded as binding in the PRAM model also.

Note that comparison tree model is not a meaningful computing model for most problems in geometry like the convex hulls, that inherently involve polynomials of degree greater than one. For this, we will first prove a **worst case** bound along the lines of Ben-Or [15] and subsequently extend it to the average case. The number of leaves is related to the number of connected components in the solution space in R^n where n is the dimension of the solution space (which is often the input size). The arity of this tree is the number of distinct outcomes of computations performed by $p > 1$ processors. The additional complication present in a BDD tree model is that each leaf node may be associated with several connected components of the solution set W. Even if we know $|W|$ (the number of connected components of W), we still need a lower bound on the number of leaves. Ben-Or tackles this by bounding the number of connected components associated with a leaf using

results of Milnor and Thom. His result shows that even under these conditions, the *worst-case* sequential lower bound is $\Omega(\log |W|)$.

If the parallel BDD algorithm uses p processors then the signs of p polynomials can be computed simultaneously. Each test yields a sign and we branch according to the sign vector obtained from all the tests. We shall use the following result on the number of connected components induced by m fixed degree polynomial inequalities due to Pollack and Roy to bound the number of feasible sign-vectors

Lemma 5.5 *The number of connected components of all nonempty realizations of sign conditions of m polynomials in d variables, each of degree at most b is bounded by $(O(bm/d))^d$.*

This gives us a bound on the arity of the parallel BDD tree model as well as the number of connected components associated with a leaf node at depth h. The number of polynomials defining the space in a leaf-node at depth h is hp and hence the number of connected components associated with such a node is $((O(bhp/d))^d$. In our context, the number of processors and (hence the polynomial signs computed at each stage) is bound by kn and d is the dimension of the solution space which is approximately the size of the input. For example, in two dimensions, an n point input consists of $2n$ real numbers corresponding to all the x and y coordinates and it is considered to be a point in E^{2n}, the Euclidean $2n$ dimensional space.. This gives us the following theorem

Theorem 5.6 *Let $W \subset R^n$ be a set that has $|W|$ connected components. Then any parallel BDD tree algorithm that decides membership in W using kn ($k \geq 1$) processors has time complexity $\Omega(\log |W|/n \log k)$.*

Proof: If h is the length of the longest path in the tree then from Lemma 5.5

$$(ekn/n)^{hn} \cdot (ehkn/n)^n \geq |W|$$

where e is a constant that subsumes the degree of the polynomials. The first expression on the left hand side represents maximum number of leaves and the second expression is the maximum number of connected components associated with a leaf at depth h. By simple manipulations and using $hn > h \log n$ we arrive at the required result. □

The above theorem immediately yields as corollary $\Omega(\log n/\log k)$ worst-case bound for a number of problems for which $|W|$ at least $(n/2)^{(n/2)}$. This holds for a slightly modified version (used previously in [55, 85]) of the convex-hull identification problem where the objective is to determine if among a set of n points all the points belong to the convex-hull. Note that this version is constant time reducible to the standard version in a CRCW PRAM model with $p \geq n$ processors.

Corollary 5.7 *Any algorithm in the parallel algebraic decision tree model that constructs the convex hull of n points using kn processors requires $\Omega(\log n/\log k)$ rounds, $k \geq 1$.*

Clearly a similar bound also holds for sorting n real numbers which strengthens the lower bound for the PCT model.

Corollary 5.8 *Every parallel algebraic decision tree algorithm requires $\Omega(\log n/\log k)$ steps to sort n numbers using kn processors, $k \geq 1$.*

To extend the above result to the average case we require a technical lemma that generalizes the following property of balanced trees - the balanced tree on N leaves, achieves the minimum average height among all trees with N leaves. We extend this property of balanced trees to the case where a leaf node at depth l has a weight l^d associated with it where d is an integer significantly less than the arity of the tree. The *arity* of a tree is the maximum number of children of any node. The total weight of a tree is the sum of the weights of its leaf-nodes. The previous property can be viewed as a case where the leaves have unit weight.

Lemma 5.9 *The tree that attains the minimum average height among all trees with total weight w using the weight function defined above is a balanced tree.*

Proof: In the proof, we will use the following approach. We will start with a balanced tree T of height h for the given weight w. To avoid unnecessary complications, we assume w to be of the required form and we will justify later that it doesn't affect our arguments. Then we shall show that any unbalanced tree of the same weight must have a larger average height.

An unbalanced tree T of the same weight will have leaf nodes at depth less than h and greater than h, where depth of a node is its distance from the root. For a leaf-node A at depth i ($i < h$), the corresponding subtree at A in $\mathcal{T}$ has a^{h-i} leaf-nodes at depth h, where a is the arity of the tree. If T has to match the weight of $\mathcal{T}$, then this 'weight-loss' has to be compensated for by leaves at depth greater than h. This is the way we view it, i.e., the leaves in T at depths less than h have the corresponding sub-tree (in $\mathcal{T}$) missing whose weights are made-up for by leaves at depths greater than h. More precisely, for leaf A at depth i, we have to make up for a weight $a^{h-i} \cdot h^d - i^d$. So we can group with leaf A leaves at depth $h_1, h_2, .., h_l$ that will attain this weight. Notice that we can do this groupings for all leaves at depth less than h. However some of these groups could be overlapping because the same node N (at depth greater than h) may be compensating fractionally for more than one group. In the remainder of this proof, we shall show that the average height of each group exceeds h which will imply the lemma.

Let there be c_i leaves at height h_i ($> h$) compensating for A at height i. At most two c_i could be fractional - the first and the last. For weight compensation
$c_1 \cdot h_1^d + c_2 \cdot h_2^d + ...c_l \cdot h_l^d \geq a^{h-i} \cdot h^d - i^d$ where $h < h_1 < h_2$..
We want to show that $\sum_i c_i \cdot h_i + i / \sum_i c_i + 1 > h$
Or by rearranging, $\sum_i c_i \cdot (h_i - h) > h - i$
It suffices to show that $\max\{c_i\} > h - i$. Let c_m be the maximum of c_i. Clearly $c_m \geq \frac{a^{h-i} \cdot h^d - i^d}{h_1^d + h_2^d .. h_l^d}$.
If $l > 2h - i + 1$, then clearly the average height of this group exceeds h. So we will only consider $l \leq 2h - i + 1$.

$$c_m \geq \frac{a^{h-i} \cdot h^d - i^d}{(h - i + 1)(2h - i + 1)^d} \tag{9}$$

Since $i^d / a^{h-i} < i^d$, inequality 9 can be written as

$$c_m \geq \frac{a^{h-i} \cdot (h^d - i^d)}{(h - i + 1)(2h - i + 1)^d} \tag{10}$$

By minimizing $h^d - i^d$ and maximizing $(2h - i + 1)^d$ (use $i = h - 1$ and $i = 1$ respectively),

$$c_m \geq \frac{a^{h-i} \cdot d \cdot (h-1)^{d-1}}{(h - i + 1) \cdot (2h)^d} \geq \frac{a^{h-i}}{(h - i + 1) \cdot 3^d}$$

when $h < d$. For sufficiently large arity a, namely $a \approx (ek)^n$ where $e, k > 4$, $c_m > h - i$. □

In the context of the parallel algebraic decision trees, arity a is $(ek)^n$ and $d = n$. We can then bound the number of leaves of the BDD tree to be $\Omega(|W|/(enkL/n)^n)$ where L is the average height. This yields a bound similar to the previous theorem.

Theorem 5.10 *Let $W \subset R^n$ be a set that has $|W|$ connected components. Then any parallel BDD tree algorithm for deciding membership in W using kn ($k \geq 1$) processors has average time complexity $\Omega(\log |W|/n \log k)$. Consequently, any randomized algorithm has the same worst case time complexity.*

Since $|W|$ is at least $(n/2)^{(n/2)}$ for the the two dimensional convex hull problem, the average running time is at least $\Omega(\log l/n \log k)$ time. For sorting and the dominance problem, the same bounds hold. By a simple reduction of 2-D dominance to trapezoidal decomposition, we obtain a similar bound for the latter problem.

6 Geometry on fixed-connection machines

6.1 Motivation and overview

In the previous sections, we described fairly general methods for parallelizing algorithms in the PRAM environment. In the process, a number of basic problems were recognized and many sophisticated techniques have been developed which can be viewed as a 'tool-kit' for tackling increasingly complex problems. There is a general consensus that PRAM models are appropriate for the algorithm designer but these algorithms have to be implemented on fixed-connection networks to be of any practical significance. 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 the PRAM algorithms can be extended to fixed-connection networks without this logarithmic penalty in running time.

In a top-down approach to algorithm design, complicated algorithms are built on top of less complex procedures. The answer to the above question would depend on how far down in this hierarchy can one go without running into problems that cannot be mapped optimally on the fixed-connection network. Moreover, this would also depend on the nature of the algorithm itself. One of the most basic problem in this hierarchy is that of sorting. For example, Reischuk's [79] $O(\log n)$ time, n processors randomized PRAM sorting algorithm was successfully extended to networks by Reif and Valiant [78] to run in $O(\log n)$ time by using additional new sampling techniques for problem-size control. In contrast, Cole's deterministic $O(\log n)$ time parallel mergesort algorithm seems prohibitively difficult to implement (without a logarithmic slowdown) on the networks because of its liberal use of pointers. Consequently a number of algorithms that use this approach on PRAM models would be at least as difficult to be adapted to network models.

The eventual goal is to develop efficient algorithms on interconnection networks, and we would like the reader to view this in the more general context of mapping certain kinds of PRAM algorithms on fixed-connection networks and the difficulties associated therewith.

Remark 6.1: *In this section, the term fixed-connection network has been used to allude to networks which have $O(\log n)$ diameter for n-node networks. There already exists a large body of literature for geometric algorithms on grid-like networks where the diameter is a bottleneck for achieving the kind of time complexity we are aiming for.*

One of the underlying problems is doing binary search optimally in a model that does not allow concurrent reads. A common scenario is the following: we are given a tree whose leaves represent intervals and n keys for which we have to determine the interval that each key lies in. If the depth of this tree is d then it is trivial to do this sequentially in $O(nd)$ time. In case of PRAM models that allow concurrent reads, the problem is again quite simple. With n processors, we can simultaneously do the search for all keys in $O(d)$ parallel time, thus resulting in an optimal speed-up. The main difficulty associated with this problem stems from the possibility that the keys may be very unevenly distributed among the intervals. Recall that in the **Partitioning** step of the generic algorithm, we use point-location in arrangements in some parameterized space.

An additional problem is that of allocating of sub-problems to sub-networks for recursive calls. Unlike $PRAM$ models, the network topology imposes severe constraints on processor allocation - not only the number of processors should match with the sub-problem sizes but should also be inter-connected in a certain manner. Some of the ideas are similar to *Flashsort* where one does *splitter-directed routing* to route the keys to the appropriate sub-networks. However, unlike the *Flashsort* we may be confronted with situations where we have to dynamically allocate resources as the sub-problems could have varying sizes. In some cases, it is possible to have near-optimal solutions to the above problems that should have applications to a wide class of algorithms. These basic procedures serve as crucial link between the PRAM algorithms and inter-connection networks.

In the remaining section, we briefly sketch a method for implementing our generic divide-and-conquer strategy in the context of trapezoidal maps of non-intersecting line segments. We have omitted most details as these are outside the scope of our main focus. For more details of the network implementation, the reader is encouraged to consult the bibliographic references.

6.2 Model of computation

Throughout this section we will be using the butterfly inter-connection model where the processors operate in a synchronous fashion and have bounded buffer size. These assumptions are consistent with some of the existing machines like BBN Butterfly and the Connection Machine. During every step, each processor is allowed to perform a real-arithmetic operation consistent with standard models used for sequential geometric algorithms. Moreover, each processor has access to a random-number generator that returns in unit time a truly random number of $O(\log n)$ bits. One of the primary reasons for choosing the butterfly network is because of its nice 'recursive' nature. A butterfly network of size k (will be referred to as BF_k) has k levels of 2^k nodes each (i.e., it has $k2^k$ nodes). Each node has an address (w, t) where $w \in \{0,1\}^k$ and $0 \leq t < k$ (see Figure 5). The significance of this network is that there are numerous 'copies' of BB_l in BB_k for $l < k$. We shall make crucial use of the following fact.

Fact 6.2 *For any w_1, w_2 such that $|w_1| + |w_2| = k - l$, then the subgraph of BF_k spanned by the nodes $\{(w_1 w w_2, i) | w \in \{0,1\}^l\}$ and $|w_1| \leq i \leq |w_1| + l$ is isomorphic to BF_l.*

Moreover, we may have to use emulate a larger butterfly in a work-preserving fashion. The following simple result can be used.

Fact 6.3 *A BB_k can emulate a BB_{k+c} within $O(2^c + c)$ slowdown where c is a positive integer.*

This is similar to Brent's slowdown lemma except that we have to construct a mapping of the processors of BB_{k+c} to BB_k respecting the interconnection topology. In this case it is very straightforward. Assume that $c = 1$ and map the processors with addresses (xw, i) where $x \in \{0,1\}$ and

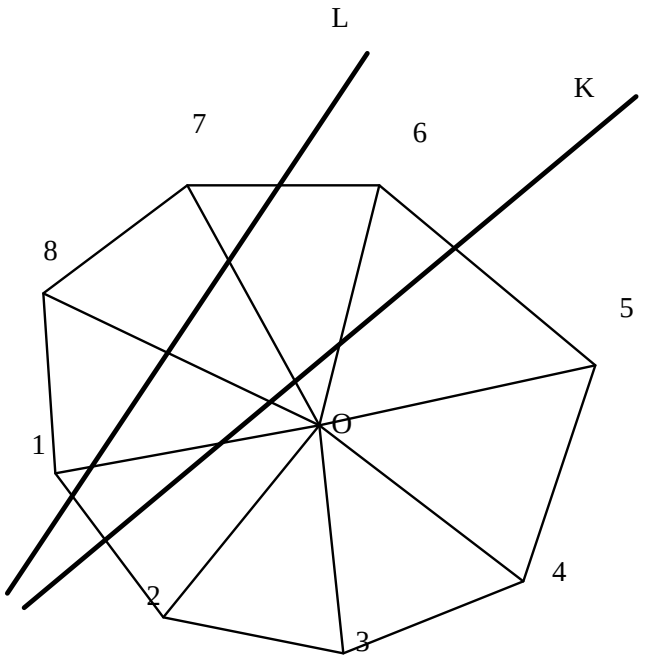

Figure 4: Lines L and K intersect a different set of sectors. In the dual plane, the duals of L and K lie in different faces - in this case separated by the dual of vertex 6.

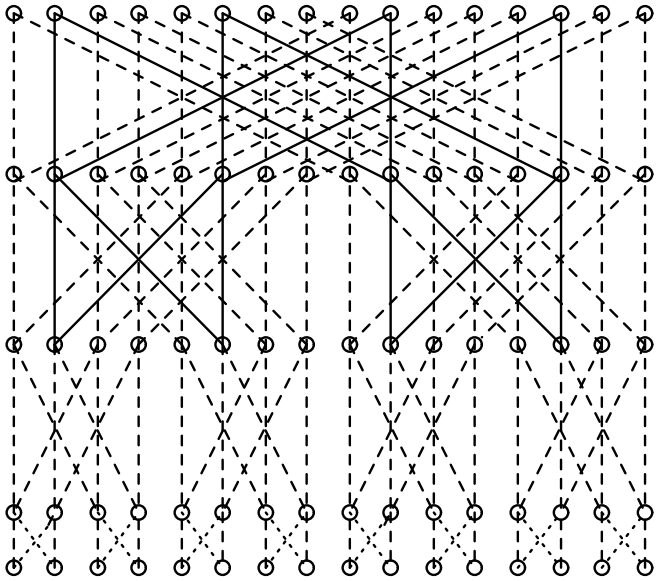

Figure 5: A butterfly network of size 4. The solid lines illustrates a sub-network isomorphic to BF_2

$|w| = k$ and $i \leq k$ to (w, i). One can verify that the processors of BF_{k+1} that were neighbors are neighbors in the smaller network. Each processor in the smaller network has to do at most twice the amount of work and require twice the amount of local memory. Only the processors of rank k have to do the extra work of emulating the processors of rank $k+1$. This scheme can be extended directly to yield the claimed bound. For a fixed c, this implies that there a constant factor slow-down. For these algorithms this scheme is often used with the value of c being 1 or 2.

6.3 Overview of sorting and routing on fixed-connection networks

The algorithms use sorting and routing extensively at various stages and a brief review of these routines will help us in understanding the latter algorithms that are built on them. The problem of *packet routing* involves routing a message from processor i to $\Pi(i)$ where Π is a permutation function. There has been a long and rich history of routing algorithms for fixed connection networks (see [86, 59, 74, 54]) and these can be summarized as following

Lemma 6.4 *There exists an algorithm for permutation routing on a n-node butterfly network that executes in $\tilde{O}(\log n)$ steps and uses only constant size queues to achieve this running time.*

A more general result has been proved by Maggs et al. [59] for *layered* networks. A *layered* network is one whose nodes can be assigned layer numbers and the edges connects a layer i node to a layer $i+1$ node (butterfly is an example of such a network). Let d denote the maximum distance traveled by any packet and c the largest number of packets that must traverse a single edge (c is also called the congestion of the network). These parameters are fixed for a given selection of paths by all the packets to be routed. Then there exists a scheme for scheduling the movements of the packets such that with high probability the routing can be completed in $O(c + d + \log n)$ steps where n is the size of the network and $O(n)$ packets are being routed.

Remark 6.5: Given the above result and also the fact that $d = O(\log n)$ for most path selection strategies (especially in a butterfly network), it remains to bound the value of c to get a bound on the routing time. For packets being routed to a random location, c can be bounded by $O(\log n)$ with high probability.

The first optimal $\tilde{O}(\log n)$ time sorting algorithm called *Flashsort* for the butterfly network was due to Reif and Valiant [78]. It was based on a PRAM sorting algorithm due to Reishchuk [79] but required several additional techniques because of the constraints imposed by the network connectivity. A slightly simplified version can be presented as the following.

1. Select n^{ϵ}, $\epsilon < 1/2$ size random subset from the given set of n keys.
2. Sort these, using a simple method like doing all the pairwise comparisons and ranking them.
3. Use these keys to set up a binary tree such that the leaves of the tree correspond to the intervals defined by a pair of consecutive splitter keys. Over-sampling techniques are used to ensure that these intervals partition the remaining keys into roughly equal sized subsets. This eliminates the need for dynamic load-balancing in the special case of sorting. The keys are assumed to be in random locations initially. For each subset a sub-network of appropriate size is set aside and the keys that belong to this subset are routed to this part of the network. This is done using a procedure called *Splitter Directed Routing* which will referred as SDR in future references. Since this is a very useful operation, we describe it in more details below.
4. These steps are applied recursively until the size of the subproblems is no more than $\log^2 n$.

Although the original analysis showed that $\tilde{O}(\log n)$ buffer size may be required the more recent results on routing enables one to do with a constant amount of storage in each buffer([59]).

6.3.1 Splitter Directed Routing

Let X be the set of cN keys that are totally ordered by the relation $<$. V is the set of nodes in the network. Suppose that for some l $(1 \leq l \leq n)$ we are given a set of *splitters* $\Sigma \subseteq X$ of size $|\Sigma| = 2^l - 1$. We index each splitter $\sigma[w] \in \Sigma$ by a distinct binary string $w \in \{0,1\}^L$ of length less than L. Let $\prec$ denote the ordering defined as follows: For $u, v, w \in \{0,1\}^L$,$w0u \prec w \prec w1v$. We require that for all $w_1, w_2 \in \{0,1\}^L, \sigma[w_1] < \sigma[w_2]$ if and only if $w_1 \prec w_2$. We assume that a copy of each splitter $\sigma[w]$ is available in each node $V[w]$. $V[w]$ is the set of nodes with rank $|w|$ with addresses prefixed by w (same as in Reif and Valiant [78].

Let $X[\lambda] = X$ where λ is the empty string. Initially we assume that the keys of $X[\lambda]$ are located in $V[\lambda]$, that is, the nodes of V having stage 0. The splitter directed routing tree is executed in l temporarily overlapping stages $i = 0, 1, ..l-1$. For each $w \in \{0,1\}^i$ the set of keys $X[w]$ that are eventually routed through $V[w]$ is defined recursively. The splitter $\sigma[w]$ partitions $X[w] - \sigma[w]$ into disjoint subsets

$$X[w0] = \{x \in X[w] | x < \sigma[w]\}$$

and

$$X[w0] = \{x \in X[w] | x > \sigma[w]\}$$

which are subsequently routed through $V[w0]$ and $V[w1]$ respectively.

In our case, we assume that after each recursive call, the sub-networks (of varying sizes corresponding to different subroutine calls) are relabeled as if these were isolated networks. The $V[w]$'s are then defined accordingly. The time analysis for this procedure is carried out using a *delay-sequence* argument [54] and it can be shown that this takes $\tilde{O}(\log n)$ time in a BF_n.

We would need a generalization of the result of Theorem 2.7 where the process tree is modified in the following manner. Instead of all the sub-routines from a node proceeding independently, all the subroutines for a fixed (constant) depth subtree are required to finish before proceeding to the next level (of subtrees). This can be reduced to the previous case if we contract a subtree of fixed depth (see Figure 6) into a single node of the tree. By appropriate adjustment of constants, we can prove the following result along the lines of Theorem 2.7.

Corollary 6.6 *All the leaf level procedures of this modified tree will terminate in $\tilde{O}(\log n)$ steps.*

6.4 Binary search without Concurrent Reads

One of the frequently encountered problems in case of sequential and parallel algorithms is that of doing a binary search on a tree structure. In particular, 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 interval that the key belongs to in $O(\log n)$ parallel time (for each key simultaneously). This problem is trivial in a model allowing concurrent-reads. However, the problem becomes more complicated when concurrent reads are not permitted in the model which is the case with inter-connection networks. The simple algorithm uses concurrent reads in an inherent fashion and for situations where the distribution of the keys is not known, this problem appears even more formidable - see Paul et al. [68]. We also note that in certain cases where the intervals and the

keys are chosen from the same total ordering the problem reduces to that of merging which can be done efficiently. However, we are concerned about cases where the intervals may induce an ordering different from the ordering among the keys (see Figure 7).

We shall look at a special case where the number of leaves is n^ϵ for $0 < \epsilon < 1/2$ and the search tree is roughly balanced, so that it has depth $O(\log n)$. The basic strategy is the following. We first try to get a reasonably accurate estimate of the number of keys associated with each of the leaf nodes. Following this, we allocate appropriate number of sub-networks based on the estimate. Next we route the keys to their destination sub-networks using a scheme similar to the *splitter directed routing* (SDR) used in **Flashsort**. We then solve the problem recursively within each sub-network. Omitting further details, we summarize as as follows.

Theorem 6.7 *Given a binary search tree with $O(k)$ leaves, we can do binary search for $O(k)$ keys in $\tilde{O}(\log k)$ time in a butterfly network using k processors.*

6.5 Applications to Computational Geometry

6.5.1 Searching in Arrangements

We first focus our attention the following problem. Given an arrangement of n^γ lines, $\gamma < 1/3$, we want to find out for n given points the region that it belongs to. In particular we would like to do this in $O(\log n)$ time on a butterfly network.

The above problem is useful for the locus-based partitioning. Using the result for binary searches on butterfly network, we can obtain the following result. We have omitted the details of load-balancing and processor allocation which are beyond the scope of our present discussion.

Theorem 6.8 *Given n points in the plane and an arrangement of m lines, $m \leq n^{1/3}$, for each of the n points, the face containing the point can be determined in $\tilde{O}(\log n)$ time in an n-node butterfly network.*

6.5.2 Trapezoidal maps

For trapezoidal decomposition, we sample line-segments and build its convex-map. We can build this using the following brute-force approach. For every segment end-point, we order the line-segments by their y coordinates. For every segment, we order the projection of the end-points those are visible (from the bottom and top). From this information, we can construct the trapezoids by simulating 'pointer-jumping' a fixed (at most 6) number of times.

Lemma 6.9 *The trapezoidal map of n^ϵ segments segments can be constructed in $\tilde{O}(\log n)$ time in a $n^{3\epsilon}$ processor butterfly network.*

The remaining segments (not part of the sample) are partitioned into subproblems defined by the trapezoidal map. The partitioning step in trapezoidal decomposition can also be reduced to the problem of searching in arrangements of linear constraints in two dimensions (see section 4.1.1). The faces of the arrangement are preprocessed so that after the point location, one has to do a table look-up to determine the answer. In this case it is a set of trapezoids (that a segment intersects). Since this can be of various length, it has to be done with a little care. First the number of trapezoids is determined (this is just a number) and then an appropriate number of processors is

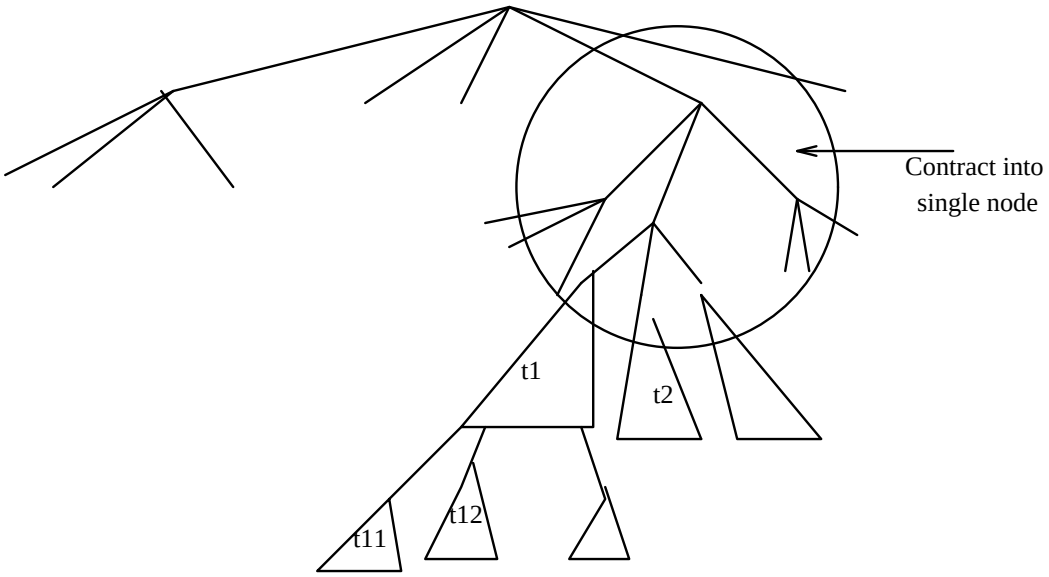

Figure 6: By contracting subtrees of a fixed depth, we get another tree satisfying the preconditions of the lemma

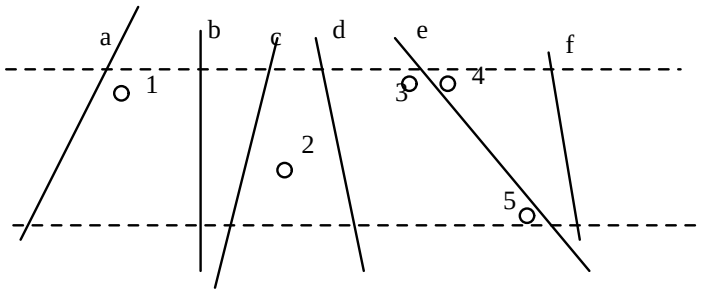

Figure 7: Points 4 and 5 are in different relative orderings with expect to each-other and the segments

delegated the responsibility of determining the actual trapezoids. This allocation is done with the help of a prefix computation. Subsequently, each of this processor do a table look up. The table look-up is done by emulating a single step of the CREW PRAM in $\tilde{O}(\log n)$ steps.

For the filtering step, we need to keep track of parts of segments that completely span a trapezoid. Such segments within a trapezoid have to be processed for binary search such that for end-points lying within the trapezoid, we can quickly determine its closest visible (upper and lower) segments. This can be done in $\tilde{O}(\log n)$ time using the algorithm of section 6.3. Moreover, only the segments that partially or completely lie within a trapezoid are needed for further recursive calls.

At each stage we keep track for each end-point, which are its closest (upper and lower) segments and hence at the end we have its trapezoidal edge(s). From this information, we can decompose a simple polygon into one-sided monotone polygons (see Figure 8). This can be accomplished by an application of sorting and prefix computation (Goodrich [38]). Using Yap's [90] technique, further calls to trapezoidal decomposition within these one-sided monotone polygons enables us to determine all the triangulation edges. The procedure is as follows.

Step 1 Construct the horizontal trapezoidal decomposition of the one-sided monotone polygon i.e., for every vertex v determine the edges of the polygon that a horizontal line through v while the line is contained within the polygon. Assume that the distinguished edge (the one with which the polygon is monotone) is horizontal.

Step 2 Denote the left visible edge by $l(v)$ and right visible edge by $r(v)$ and let $\lambda(v)$ and $\rho(v)$ denote vertices of $l(v)$ and $r(v)$ respectively which have the lower altitude.

Step 3 The set of edges $E = \{v\lambda(v)\} \cup \{v\rho(v)\}$ form a triangulation of the polygon.

Trapezoidal decomposition also enables us to solve the problem of determining visibility of a set of non-intersecting line segments when they are projected orthogonally. Sort the end points of the line segments projected on the x-direction and choose a point (say the mid-point) in every interval. For each of these points, determine its trapezoidal edge using the trapezoidal decomposition algorithm described above.

We can state the main result of this section as the following.

Theorem 6.10 *There exist algorithms for problems of Trapezoidal Decomposition, Triangulation of Simple Polygons and Visibility that execute in time $\tilde{O}(\log n)$ on an n-processor butterfly network where n is the input size of the problem.*

Using similar methods, we can also implement an algorithm for two-dimensional convex hulls along the lines of the 3-D hull algorithm described in section 4.3. In fact the Filtering step is much simpler.

Theorem 6.11 *The convex hull of n points in the plane can be constructed in $\tilde{O}(\log n)$ in an n node butterfly network with bounded buffer size.*

7 Bibliographic Notes

Random sampling was introduced into parallel computational geometry by Reif and Sen [75] (conference version) independently around the same time as the seminal papers of Clarkson [25, 26] and Haussler and Welzl [51]. All these papers primarily exploited the ε-net property, namely the (almost) even partitioning of the problem using a random subset. The subsequent papers of Clarkson [27] and Clarkson and Shor [30] refined the techniques considerably and developed very general and elegant techniques for designing geometric algorithms, namely, the randomized divide-and-conquer and randomized incremental construction. Mulmuley [65, 66], extended randomized incremental construction for dynamic settings and obtained impressive results for random updates. The textbook of Mulmuley [67] gives an excellent description of these general techniques.

A number of very general properties of random-sampling were proved in Clarkson and Shor [30] that simplified or/and improved existing algorithms. The discussion in section 3 on properties of random sampling follows the methods of Reif and Sen [75, 76]. The proofs of lemmas 3.2 and 3.5 are somewhat simpler and more direct compared to [26, 30] but less general.

The technique of Polling was first introduced by Reif and Sen [76] (conference version). It is an efficient method to find a sample with certain desirable properties that succeeds with high probability. The straight-forward sampling, as introduced initially, is guaranteed to succeed with only constant probability which does not suffice for bounding the running times of parallel algorithms (Section 3.2). In particular, the high-probability bounds are critical for divide-and-conquer based parallel algorithms. Even for many sequential algorithms based on Clarkson and Shor's paradigm, Polling can be used to bound the variance of the (otherwise only expected) running times without loss of efficiency.

The earliest known results in parallel computational geometry can be traced to the thesis of Anita Chow [21] who described polylog(n) algorithms for a number of basic problems. Aggarwal et al. [1] developed some of the earliest techniques towards a unified approach for solving problems like convex hulls, Voronoi diagrams and triangulation. Atallah, Cole and Goodrich [6] improved the efficiency of these problems by extending the techniques of Cole's parallel mergesort [23] which they called *Pipelined cascaded merging*. Goodrich's thesis [38] is a comprehensive source of some of the basic parallel techniques for solving geometric problems.

A majority of the research in this area was directed towards obtaining $O(\log n)$ time algorithms for the above-mentioned problems with optimal speed-up. Reif and Sen [75, 76] designed algorithms using randomization that achieved the above goals even for problems for which matching deterministic algorithms are not known, in particular the 3-D convex hull problem. The algorithm for trapezoidal decomposition in section 4 is a simpler version of the one in [75]. Clarkson, Cole and Tarjan [29] describe optimal speed-up algorithms for a more general version of this problem where the input is a union of K polygonal chains, possibly self-intersecting. For the case of a simple polygon Goodrich [40] designed an optimal deterministic $O(\log n)$ time algorithm based on Chazelle's [19] optimal sequential algorithm for triangulation and efficient construction of planar separators.

The optimal algorithm for three-dimensional convex hulls appears in [76] - the version presented here uses a simpler Filtering scheme used in Amato et al. [7]. The algorithm for two-dimensional arrangements is based on Hagerup et al.'s [49] optimal speed-up algorithms for constructing arrangements of hyperplanes in d dimensions. Ramaswamy and Rajasekaran [73] describe optimal speed-up algorithms for constructing Voronoi diagrams of line segments using an approach that

does not require Polling.

The sublogarithmic algorithms of section 5 appear in Sen [83]. The sublogarithmic algorithms play an important role in the fast ($O(\log H \log\log n)$ time for output size H) output-sensitive algorithms of Gupta and Sen [45, 46]. Efficient output-sensitive algorithms for convex-hulls were first described in Ghouse and Goodrich [37].

The details of the Butterfly implementation can be found in Reif and Sen [77]. The problem of binary-search without concurrent reads is also known as *multisearching* in literature.

For a more detailed treatment of the basic parallel routines of section 1.3, the reader is referred to the textbook of Ja'Ja' [52], and Rajasekaran and Sen [72]. The planar point-location algorithm was discovered independently by Dadoun and Kirkpatrick [32] and Reif and Sen [75] - the version presented here follows the latter.

The $\Omega(\log n/\log\log n)$ lower bound for selection due to Beame and Hastad [14] discouraged researchers from investigating sublogarithmic algorithms for a while. The first breakthrough was obtained by McKenzie and Stout [62] and subsequently Hagerup and Raman [50] obtained tight bounds for padded-sorting. The $O(\log^* n)$ result for approximate compaction was discovered simultaneously by Goodrich [39], Matias and Vishkin [61] and Hagerup [48]. The algorithms referred to in Lemma 2.22 can be found in Bast and Hagerup [13].

The issue of bounding the number of random bits has been dealt in Reif and Sen [76] who show that polylogarithmic number of random bits suffice for the algorithms described in this chapter. Bounding the number of random bits has important ramifications in derandomization. Recently there have been some improved results obtained by efficient derandomization of randomized parallel algorithms - Goodrich [41] and Amato et al. [7]. Among other impressive results they also obtain a deterministic work-optimal three-dimensional convex hull algorithm based on the algorithm of Reif and Sen.

An important problem that was not discussed in this chapter is the fixed-dimensional linear programming. Randomization has been very successfully used for obtaining some of the most elegant and efficient algorithms for this problem [28, 56, 60]. Alon and Meggido [3] describe an $O(1)$ expected time algorithm for this problem using n processors.

The algorithms described in this chapter for 3-D convex hull and multisearching have been further extended by Goodrich [43] to the BSP model.

References

[1] 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.

[2] N. Alon and Y. Azar. The average complexity of deterministic and randomized parallel comparison-sorting algorithms. *SIAM Journal on Computing*, 17:1178–1192, 1988.

[3] N. Alon and N. Meggido Parallel linear programming in fixed dimensions almost surely in constant time, *Journal of the ACM*, 1994, pp. 422–434.

[4] M.J. Atallah and M.T. Goodrich. Efficient parallel solutions to some geometric problems. *Journal of Parallel and Distributed Computing*, 3(4):492 – 507, 1986.

[5] M.J. Atallah and M.T. Goodrich. Parallel algorithm for some functions of two convex polygons. *Algorithmica*, 3(4):535 – 548, 1988.

[6] M.J. Atallah, R. Cole, and M.T. Goodrich. Cascading divide-and-conquer:, a technique for designing parallel algorithms. *SIAM Journal on Computing*, 18:499 – 532, 1989.

[7] N.M. Amato, M.T. Goodrich and E.A. Ramos. Parallel Algorithms for Higher-Dimensional Convex Hulls. *Proc. of the 35th Annual FOCS*, pages 683– 694, 1994.

[8] S. Akl. Optimal algorithms for computing convex hulls and sorting. *Computing*, 33:1, 1–11, 1984.

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

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

[11] N.M. Amato and F.P. Preparata. The Parallel 3D convex hull problem revisited. *Internat. Jl. Comput. Geom. Appl.*, 2(2):163-173, 1992.

[12] Y. Azar and U. Vishkin. Tight comparison bounds on the complexity of parallel sorting. *SIAM Journal on Computing*, 16:458–464, 1987.

[13] H. Bast and T. Hagerup. Fast parallel space allocation,estimation and integer sorting. *Technical Report, MPI-I-93-123*, June 1993.

[14] P. Beame and J. Hastad. Optimal bounds for decision problems on CRCW PRAMS. *Proc. of the Nineteenth ACM STOC*, 83–93, 1987.

[15] M. Ben-Or. Lower bounds for algebraic computation trees. *Proc. of the Fifteenth STOC*, 80–86, 1983.

[16] R.B. Boppana. The average-case parallel complexity of sorting. *Information Processing Letters*, 33:145–146, 1989.

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

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

[19] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete and Computational Geometry*, 6, pp. 485–524, 1991.

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

[21] A.L. Chow. Parallel Algorithms for Geometric Problems. PhD Thesis, Deptt. of Comp.Sc., Univ. of Illnois, Urbana, IL, 1980.

[22] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17:770 – 785, 1988.

[23] R. Cole. An optimal efficient selection algorithm. *Inform. Proc. Lett.*, 26:295–299, 1987/88.

[24] 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.

[25] K.L. Clarkson. A randomized algorithm for closest point queries. *SIAM Journal on Computing*, 17(4), August 1988, pp. 830–847.

[26] K.L. Clarkson. New Applications of random sampling in Computational Geometry. *Discrete and Computational Geometry*, (1987), pp. 195–222.

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

[28] K.L. Clarkson. Las Vegas algorithms for linear and integer programming when the dimension is small. *Proc. of the 29th IEEE Symposium on Foundations of Computer Science* 1988, pp. 452–456.

[29] K.L. Clarkson, R. Cole and R. Tarjan. Randomized parallel algorithms for trapezoidal diagrams. *Int'l journal of computational geometry and applications* 2, pp. 117–134, 1992.

[30] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry ii. *Discrete Comp. Geom.*, 4:387–421, 1989.

[31] X. Deng. An optimal parallel algorithm for linear programming in the plane. *Inform. Proc. Lett.*, 35:213–217, 1990.

[32] N. Dadoun and D.G. Kirkpatrick. Parallel Construction of subdivision hierarchies. *Jl. Comput. Syst. Sc.*, 39:153-165,1989.

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

[34] H. Bast, M. Dietzfelbinger and T. Hagerup. A perfect parallel dictionary. *Proc. of the 17th Intl. Symp on Math, Foundations of Computer Science, LNCS 629*, 133– 141, 1992.

[35] H. Edelsbrunner . *Algorithms in combinatorial geometry.* Springer-verlag, New York, 1987.

[36] H. Edelsbrunner, L. Guibas and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15, pp. 317–340, 1986.

[37] M.Ghouse and M.T. Goodrich. In-place techniques for parallel convex-hull algorithm. *Proc. 3rd ACM Sympos. Parallel Algo. Architect.*, 192-203,1991

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

[39] M. Goodrich. Using approximate algorithms to design parallel algorithms that may ignore processor allocation. *Proc. of the 32nd Annual FOCS*, 711–722, 1991.

[40] M. Goodrich Planar separators and parallel polygon triangulation. *to appear* in Journal of Computer and System Sciences. *A Preliminary version appeared in Proc. of the 24th ACM STOC* pp. 507–516, 1992.

[41] M. Goodrich. Geometric partitioning made easier, even in parallel. *Proc. of the 9th ACM Symp. on Computational Geometry*, 73–82, 1993.

[42] M. Goodrich. Communication efficient parallel sorting. *Proc. of the 28th ACM Symp. on Theory of Computing*, 1996.

[43] M. Goodrich. Randomized Fully-Scalable BSP Techniques for Multi-Searching and Convex Hull Construction.. *Proc. of the 8th Symp. on Discrete Algorithms*, 1997.

[44] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Proc. Lett.*, 1:132–133, 1972.

[45] N. Gupta and S. Sen. Optimal and Output Sensitive Algorithms for Constructing Planar Hulls in Parallel. to appear in *Computational Geometry: Theory and Applications.*

[46] N. Gupta and S. Sen. Faster output-sensitive parallel convex hulls for $d \leq 3$: optimal sublogarithmic algorithms for small outputs. *Proc. of the 12th ACM Symp. on Computational Geometry*, 176–185, 1996. *Proc. of the ACM Symposium on Computational Geometry* 1996.

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

[48] T. Hagerup. The log star revolution. *Proc. of the 9th Annual STACS, LNCS 577*, 259 – 278, 1992.

[49] T. Hagerup, H. Jung and E. Welzl. Efficient parallel computation of arrangements of hyperplanes in d dimensions. *Proc. of ACM Symp on Parallel Algorithms and Architectures*, 1990, pages 290–297.

[50] T. Hagerup and R. Raman. Waste makes haste: Tight bounds for loose, parallel sorting. *Proc. of the 33rd Annual FOCS*, pages 628– 637, 1992.

[51] D. Haussler and E. Welzl. ϵ-Nets and simplex range queries. *Discrete and Computational Geometry*, 2, 1987, pp. 127–152.

[52] Joseph JaJa: *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[53] R.M. Karp. Probabilistic Recurrence Relations *Proc. of the 23rd ACM STOC*, pp. 190–197, 1991.

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

[55] S. Kapoor and P. Ramanan. Lower bounds for maximal and convex layer problems. *Algorithmica*, pages 447–459, 1989.

[56] G. Kalai. A subexponential randomized simplex algorithm, *Proc. of the 24th ACM Symposium on Theory of Computing*, 1992.

[57] D.G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12, pp. 28–35, 1983.

[58] D G Kirkpatrick and R Seidel. The ultimate planar convex hull algorithm. *SIAM Journal of Computing*, 15(1):287–299, Feb. 1986.

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

[60] J. Matoušek, M. Sharir and E. Welzl. A subexponential bound for linear programming, *Proc. of the 8th ACM Symposium on Computational Geometry*, 1992, p p. 1–8.

[61] Y. Matias and U. Vishkin Converting high probability into nearly constant time - with applications to parallel hashing. *Proc 23rd ACM Symp. on Theory of Computing*, 307–316, 1991.

[62] P. MacKenzie and Q. Stout. Ultra-fast expected time parallel algorithms. *Proc. of the 2nd SODA*, 414–423, 1991.

[63] U. Manber and M. Tompa. The effect of number of hamiltonian paths on the complexity of a vertex colouring problem. *SIAM Journal on Computing*, 13:109–115, 1984.

[64] K. Mulmuley. A fast planar partition algorithm, I. *Proc. of the 29th IEEE Foundations of Computer Science* pp. 580–589, 1988.

[65] K. Mulmuley. Randomized multidimensional search trees: Dynamic sampling. *Proc. of the 7th ACMSymp. on Computational Geometry*, pp. 121–131, 1991.

[66] K. Mulmuley Randomized multidimensional search trees: lazy balancing and dynamic shuffling. *Proc. of the 32nd IEEE Foundations of Computer Science*, pp. 180–196, 1991.

[67] K. Mulmuley Computational Geometry: An introduction through randomized algorithms. Prentice Hall 1994.

[68] W. Paul, U. Vishkin and H. Wagener Parallel dictionaries on 2-3 trees. *Proc. of tenth ICALP*, Vol. 154, pp. 597–609, 1983.

[69] F. P. Preparata and S J Hong. Convex hulls of finite sets of points in two and three dimensions. *Comm. ACM*, 20:87–93, 1977.

[70] F. P. Preparata. An optimal real time algorithm for planar convex hulls. *Comm. ACM*, 22:402–405, 1979.

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

[72] S. Rajasekaran and S. Sen. *Random sampling Techniques and parallel algorithm design.* J.H. Reif editor. Morgan, Kaufman Publishers, 1993.

[73] S. Ramaswami and S. Rajasekaran Optimal parallel randomized algorithms for voronoi diagram of line segments in the plane and related problems *Proc. of the 10th ACM Symp. on Computational Geometry*, pp. 57–66, 1994.

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

[75] J.H. Reif and S. Sen. Optimal randomized parallel algorithms for computational geometry. *Algorithmica*, 7:91 – 117, 1992. *A Preliminary version appeared in Proc. of the 16th Int'l Conferen ce on Parallel Processing, Aug. 1987*

[76] J.H. Reif and S. Sen. Optimal Parallel Randomized Algorithms for three-dimensional convex hulls and related problems. *SIAM Journal on Computing*, 21(3),466-485, 1992. *A preliminary version appears as* Polling: A new random sampling technique for computational geometry, *in Proc. of the ACM STOC* May, 1989.

[77] J.H. Reif and S. Sen. *Randomized algorithms for binary search and load-balancing on fixed-connection networks with geometric applications SIAM Journal on Computing*, 23(3), pages 633–651.

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

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

[80] M.F. Roy and R. Pollack. On the number of cells defined by a set of polynomials. *Comptes Rendus*, 316:573–577, 1992.

[81] N. Saranak and R. Tarjan. Planar point locatin using persistent search trees. *Communications of the ACM*, 29, pp. 669–679, 1986.

[82] S. Sen. *Random Sampling Techniques for Efficient Parallel, Algorithms in Computational Geometry.* PhD thesis, Duke University, 1989.

[83] S. Sen. Lower bounds for algebraic decision trees, complexity of convex hulls and related problems, to appear in *Theoretical Computer Science.* A preliminary version appeared in *Proc. of the 14th FST&TCS, Madras, India* , 1994.

[84] M. I. Shamos. *Computational geometry.* PhD thesis, Yale Univ.,New Haven, 1978.

[85] J. Steele and A.C. Yao. Lower bounds for algebraic decision trees. *Journal of Algorithms*, 1–8, 1982.

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

[87] L.G. Valiant, A bridging model for parallel computation. *Communications of the ACM*, 33:103–111, 1990.

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

[89] A.C. Yao. A lower bound for finding convex hulls. *Journal of the A.C.M.*, 28:780–787, 1981.

[90] C.K. Yap. Parallel triangulation of a polygon in two calls to the, trapezoidal map. *Algorithmica*, 3:279 –288, 1988.

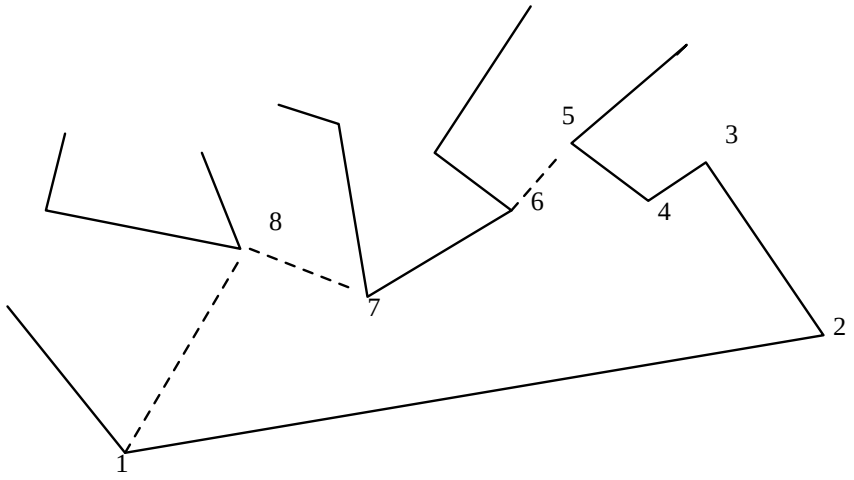

Figure 8: Decomposition into one sided monotone polygons. The dotted lines indicate some of the triangulation edges. The distinguished edge is 1,2.