

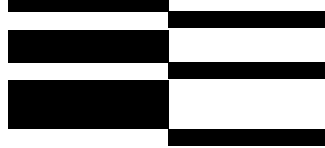
# Illustration of Reif Macros<sup>1</sup>

Mukund N. Thapa

October 10, 2000

<sup>1</sup>Designer: Morgan Kaufmann

# 3



## Parallel Randomized Algorithms for Selection, Sorting, and Convex Hulls

Sanguthevar Rajasekaran

*Department of Computer and Information Science*

*University of Pennsylvania*

*Philadelphia, PA 19104*

Sandeep Sen

*Department of Computer Science*

*Duke University*

*Durham, NC 27706*

## 3.1 Introduction

### 3.1.1 Randomized Algorithms

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

Even though the idea of randomization is at least as old as Hoare's quicksort algorithm, these previous approaches assume a distribution on the space of all possible inputs, which may not be a valid assumption at all. For example, Hoare's quicksort algorithm may run for a long period of time on certain inputs. However the number of such bad input permutations is only a small fraction of the possible inputs. If we assume (which indeed Hoare does) that each input permutation is equally likely to occur, then quicksort algorithm is quite practical because with very high probability the given input permutation will not be a bad one and hence the algorithm will terminate quickly. But, Hoare's assumption of a uniform distribution on the input space is questionable, since the input distribution may vary quite unpredictably. Rabin and Solovay & Strassen rectify this problem by introducing randomization into the algorithm itself.

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. For a given input, some of the algorithms in this family may run for indefinitely long periods of time. The objective in the design of a randomized algorithm is to ensure that the number of such bad algorithms in the family is only a small fraction of the total number of algorithms. If for *any* input we can show that at least  $(1 - \epsilon)$  ( $\epsilon$  being very close to 0) fraction of algorithms in the family will run quickly on that input, then clearly, a random algorithm in the family will run quickly on any input with probability  $\geq (1 - \epsilon)$ . In this case we say that this family of algorithms (or this randomized algorithm) runs quickly with probability at least  $(1 - \epsilon)$  where  $\epsilon$  is called the error probability. Observe that this probability is independent of the input distribution.

To give a flavor for the above notions, we now give an example of a randomized algorithm. Given a polynomial of  $n$  variables  $f(x_1, \dots, x_n)$  over

a field  $F$ , it is required to check if  $f$  is identically zero. We generate a random  $n$ -vector  $(r_1, \dots, r_n)$  ( $r_i \in F, i = 1, \dots, n$ ) and check if  $f(r_1, \dots, r_n) = 0$ . We repeat this for  $k$  independent random vectors. If there is even one vector for which  $f$  evaluates to a non zero value, then clearly  $f$  is nonzero. If  $f$  evaluated to zero on all the  $k$  vectors tried, we conclude  $f$  is zero. It can be shown that the probability of error in our conclusion will be very small if we choose a sufficiently large  $k$ . In comparison, the best known deterministic algorithm for this problem is much more complicated and has a much higher time complexity.

### **Advantages of Randomization**

Two of the most important advantages of using randomized algorithms are their simplicity and efficiency. A majority of the randomized algorithms found in the literature are simpler and easier to understand than the best deterministic algorithms for the same problems. The reader may have already got some feel for this from the above given example of testing if a polynomial is identically zero. Randomized algorithms have also been shown to yield better complexity bounds.

A skeptical reader at this point might ask: How dependable are randomized algorithms in practice, after all there is a non zero probability that they may fail? Most readers are also aware that there is a probability (however small it might be) that the hardware itself might fail. So, if we design a fast algorithm for a problem with an error probability  $< 2^{-k}$  for some integer  $k$  independent of the problem size, we can reduce the error probability far below the hardware error probability by making  $k$  large enough.

### **Different types of randomized algorithms**

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. 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.

### Randomization in Parallel Algorithms

The ever-decreasing low cost of hardware nowadays has prompted computer scientists to design parallel machines and algorithms to solve problems very efficiently. In an early paper Reif proposed using randomization in parallel computation. In this paper he also solved many algebraic and graph theoretic problems in parallel using randomization. Since then a new area of research in computer science has evolved that tries to exploit the special features offered by both randomization and parallelization.

#### 3.1.2 Model of Computation

A large number of parallel machine models have been proposed. Some of the more widely accepted models are: 1) fixed connection machines, 2) shared memory models, 3) the boolean circuit model, and 4) the parallel comparison trees. For all the algorithms in this chapter, we assume the shared memory model. In the randomized version of these models, each processor is equipped with a random number generator which can produce independent random bits in unit time. This is in addition to the computations allowed by the corresponding deterministic version. The *time complexity* of a parallel machine is a function of its input size. More precisely, time complexity is a function  $g(n)$  that is the maximum over all inputs of size  $n$  of the time elapsed when the first processor begins execution until the time the last processor stops execution.

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

*Notation* We say a randomized algorithm has a resource (time, space etc.) bound of  $\tilde{O}(g(n))$  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  $\geq 1 - 1/n^\alpha$ . We shall refer to these bounds as *high probability* bounds.

We say a parallel algorithm is *optimal* if its processor bound  $P_n$  and time bound  $T_n$  are such that  $P_n T_n = \tilde{O}(S)$  where  $S$  is the time bound of the best known sequential algorithm for that problem.

In shared memory models, a number (say  $P$ ) of processors work synchronously communicating with each other with the help of a common block

of memory accessible by all processors. Each processor is a random access machine. Each step of the algorithm is an arithmetic operation, a comparison, or a memory access. Several conventions are possible to resolve read or write conflicts that might arise while accessing the shared memory. *EREW PRAM* is the shared memory model where no simultaneous read or write is allowed on any cell of the shared memory. *CREW PRAM* is a variation which permits concurrent read but not concurrent write. And finally, *CRCW PRAM* model allows both concurrent read and concurrent write. Each processor has a unique identifier (or *id* for short) which is a  $\log P$  bit integer where  $P$  is the number of processors. We also assume that each processor has access to  $O(\log n)$  bit random numbers in unit time.

### 3.1.3 Problems of Interest

In this chapter we consider the following problems: Selection, Sorting, and Convex hulls. These are very important fundamental problems in computer science which is reflected by the vast amount of literature devoted to these problems.

The problem of selection is to find the  $i$ th smallest key in a given sequence of keys (for some specified  $i$ ). The problem of sorting is to rearrange the given sequence of keys in either descending order or ascending order.

#### DEFINITION

*A convex hull of a set  $S$  of points is the smallest convex set containing  $S$ . A planar convex hull is the convex hull of a set of points in two dimensional space.*

We shall describe a parallel algorithm for constructing planar convex hulls. The algorithm can actually be extended to three dimensions with some additional effort; however we have confined ourselves to two dimensions to make the presentation simpler. The problem of sorting and construction of convex hulls are closely related. Specifically, any lower bound for sorting is also a lower bound for constructing convex hulls. Moreover, in the sequential context the algorithms for the problems share an interesting relationship. The techniques for sorting like mergesort and insertion sort can be extended almost directly for convex hulls.

## 3.2 Random Sampling Lemmas for Sorting and Selection

### 3.2.1 Selection

Let  $X = \{k_1, k_2, \dots, k_n\}$  be a set of  $n$  distinct keys. Let  $<$  be a total ordering over  $X$ . The problem of selection is to find the  $i$ th smallest key in  $X$  (for some specified  $i \leq n$ ). Let the rank of any key in  $X$  be the number of keys less than this key plus one. A trivial sequential algorithm for selection computes the rank of each key and finds the one with rank  $i$ . Such an algorithm makes  $n^2$  comparisons. But there are both deterministic and randomized algorithms that make only  $O(n)$  comparisons. The constant in the time bound of the randomized algorithm is optimal, whereas no such deterministic algorithm is known. The power of randomization in algorithms design is also illustrated by the extremal selection algorithm given in the next section.

#### Chernoff Bounds

Frequently, in the design of randomized algorithms, many parameters of interest (like run 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. Before giving the Chernoff bounds, it is illustrative to consider the following simple example.

We toss an unbiased coin  $10 \log n$  times. What is the probability  $P$  that  $\leq \log n$  of the outcomes are tails? It is easy to see that

$$P = \sum_{i=0}^{\log n} \binom{10 \log n}{i} 2^{-10 \log n}.$$

i.e.,

$$P \leq \log n \binom{10 \log n}{\log n} 2^{-10 \log n}.$$

Using Stirling's approximation for factorials,

$$P \leq \log n 2^{(10 \log 10 - 9 \log 9) \log n} 2^{-10 \log n}.$$

Thus  $P$  is very very small.

It turns out that if the mean of a binomial distribution  $X$  is  $\Omega(\log n)$ , then, the probability that  $X$  is greater than  $O(\log n)$  is  $\leq n^{-\alpha}$ , for any  $\alpha > 1$  (the constants in  $\Omega()$  and  $O()$  will depend on  $\alpha$ ). 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

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

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

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

Chernoff and Angluin & Valiant have found ways of approximating the tail ends of a binomial distribution. In particular, their results can be summarized as

#### LEMMA 3.1 Chernoff Bounds

If  $X$  is binomial with parameters  $(n, p)$ , and  $m > np$  is an integer, then

$$\text{Probability}(X \geq m) \leq \left(\frac{np}{m}\right)^m e^{m-np}. \quad (3.1)$$

Also,

$$\text{Probability}(X \leq \lfloor (1-\epsilon)pn \rfloor) \leq \exp(-\epsilon^2 np/2) \quad (3.2)$$

and

$$\text{Probability}(X \geq \lceil (1+\epsilon)np \rceil) \leq \exp(-\epsilon^2 np/3) \quad (3.3)$$

for all  $0 < \epsilon < 1$ .

### 3.2.2 A Simple Algorithm for Extremal Selection

Consider the problem of selection when  $i = n$ , i.e., the problem of finding the largest (or smallest) key from among a given set of  $n$  keys. In this section we describe a randomized algorithm on an  $n$  processor *CRCW PRAM* that solves this problem in  $\tilde{O}(1)$  time. In contrast there are known lower bounds which establish that any deterministic algorithm for maximal selection requires  $\Omega(\log \log n)$  time on an  $n$  processor *CRCW PRAM*. Before we describe the algorithm we state and prove a lemma that will be used in the algorithm.

#### LEMMA 3.2

*Maximum of  $n$  keys can be found in  $O(1/\epsilon)$  time using  $n^{1+\epsilon}$  CRCW PRAM processors.*

#### PROOF

We prove this for the special case when  $\epsilon = 1/2$  and leave the general case as an exercise problem. Let  $k_1, k_2, \dots, k_n$  be the input keys. Group the  $n^{1+1/2}$  processors into  $\sqrt{n}$  groups  $G_1, G_2, \dots, G_{\sqrt{n}}$  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). ■

Now we present the maximal selection algorithm for  $n$  keys. In the beginning processor  $i$  is assigned the key  $k_i$ , and the set of ‘surviving keys’,  $Y$ , is the same as  $X$ .

#### Step 1.1

In this step roughly  $n^{1/2}$  random keys are sampled (i.e. chosen randomly) from  $Y$ . Each processor decides to include its key in the sample with probability  $\frac{1}{2n^{1-\frac{1}{2}}}$ .

**Step 1.2**

Keys that were sampled in step 1 are assigned to unique locations in a memory block of size  $n^{3/4}$ . Call these memory cells

$M(1), M(2), \dots, M(n^{3/4})$ . For each sampled key, a random cell is chosen. If that cell has not been assigned to any other key, assignment for the key is complete. If the chosen cell has been previously assigned to some other key, another random cell is chosen. This process is repeated until a unique assignment has been found for the key.

**Step 1.3**

$n$  processors in parallel find the maximum of  $M(1), M(2), \dots, M(n^{3/4})$  in  $O(1)$  time using the procedure described in the proof of Lemma 3.2. (Note: All the  $M(i)$ 's will contain  $-\infty$  at the beginning of step 2.) Let  $m$  be the maximum found.

**Step 1.4**

Each processor  $i, (1 \leq i \leq n)$  (with a key in  $Y$ ) in parallel compares  $k_i$  with  $m$ . If  $k_i$  is less than  $m$ , then  $k_i$  will be dropped from ( $Y$  and) future consideration as a potential candidate for maximum.

The surviving keys execute steps 2 and 3 once more and  $m$  found in step 3 is output as the *maximum*.

*Analysis* The number of keys that will be included in the sample in step 1 is a binomial variable with parameters  $(n, 1/(2n^{1/2}))$ . Thus using Chernoff bounds, this number is no more than  $n^{1/2}$  with high probability. The number of keys surviving after step 4 is no more than  $n^{1/2} \log n$  with high probability (Exercise problem 6).

In second step, assignment for sampled keys can be found in  $O(1)$  time with high probability (Exercise problem 6).

From these assertions, it is easy to see that the algorithm runs in  $\tilde{O}(1)$  time.

**3.2.3 A Selection Algorithm**

In this section we give a sequential randomized selection algorithm and show how this can be modified to get a sorting algorithm also. We only

perform expected time analysis, though the same time bounds also hold with high probability. Also, we can assume that the input keys are distinct. If they are not distinct, append  $i$  (as a  $\log n$  bit binary number) to the right of  $k_i$  (for  $1 \leq i \leq n$ ). This modification of the keys does not alter either the time or the processor bounds of the algorithms given in this chapter.

ALGORITHM Select( $i, X$ );

**begin**

**if**  $X = \{x\}$  **then return**  $x$ ;

Choose a random key  $k \in X$  (called the splitter);

Let  $X_1 = \{x \in X | x < k\}$  and  $X_2 = X - X_1$ ;

**if**  $|X_1| \geq i$  **then return** Select( $i, X_1$ )

**else return** Select( $i - |X_1|, X_2$ )

**end ;**

Let  $\bar{T}(i, n)$  be the expected (sequential) run time of Select. After selecting the splitter key, partitioning of  $X$  into  $X_1$  and  $X_2$  can be performed in  $n$  steps. Also, the randomly chosen splitter key can be any one of the  $n$  keys in  $X$  with equal probability. In particular, the splitter key will be the  $j$ th smallest key of  $X$  with probability  $1/n$  (for  $1 \leq j \leq n$ ). Thus,  $\bar{T}(i, n)$  satisfies the following recurrence.

$$\bar{T}(i, n) = n + \frac{1}{n} \left[ \sum_{j=1}^i \bar{T}(i-j, n-j) + \sum_{j=i+1}^n \bar{T}(i, j) \right].$$

By induction we can show  $\bar{T}(i, n) \leq 2n + \min(i, n-i) + o(n)$ . Notice that the expectation in the above equation is over the space of all outcomes of coin flips and not over the space of all possible inputs. It would be more desirable to show that the asymptotic time bound is the same with high probability, i.e., to show that the run time is no more than  $cn$  with probability  $> 1 - \frac{1}{n^\alpha}$  for some fixed constant  $c$ .

Markov's inequality asserts that if  $\bar{T}$  is the expected value of a random variable  $T$ , then the probability that  $T$  exceeds  $k\bar{T}$  is less than  $1/k$ . This implies for example that the run time of Select will not exceed  $2\bar{T}(i, n)$  with probability  $\geq 1/2$ . This probability is not good enough when compared with  $1 - \frac{1}{n^\alpha}$ .

We can modify Select slightly so as to make the same time bound hold with high probability. The modification is to choose a random sample  $S$  of

size  $s$  from  $X$ , to find the median of  $S$ , and to use this median as the splitter key. An exactly similar algorithm can be given for sorting also. The algorithm can be outlined as the following.

```

ALGORITHM Sort( $X$ );
  begin
    if  $|X| = 1$  then return  $X$ ;
    Choose a random subset  $S \subseteq X$  of size  $s$ ;
    Let  $k = \text{select}(\lfloor s/2 \rfloor, S)$ ;
    return Sort( $\{x \in X | x < k\}$ ) . ( $k$ ) .
    Sort( $\{x \in X | x > k\}$ );
  end;

```

The correctness of Sort is based on the following lemma:

**LEMMA 3.3**

Let  $s = n/\log n$ . Then,

$$\text{Prob.} \left[ \left| \text{rank}(k, X) - n/2 \right| > \sqrt{d\alpha n \log n} \right] < n^{-\alpha}$$

for some constant  $d$ .

The proof of lemma 3.3 is left as an exercise.

Let  $\bar{T}(n)$  be the expected number of comparisons made by Sort( $X$ ). Since selection on an input of  $n$  keys takes only  $O(n)$  comparisons, we have for  $s(n) = n/\log n$ ,

$$\bar{T}(n) \leq 2\bar{T}(n_1) + n^{-\alpha}\bar{T}(n) + O(n)$$

where

$$n_1 = n/2 + \sqrt{d\alpha n \log n}.$$

This solves to

$$\bar{T}(n) = O(n \log n),$$

which asymptotically approaches the optimal number of comparisons needed to sort  $n$  numbers.

It can be proven that the order of time bounds of both Sort and modified Select remains unaltered even with high probability.

### 3.3 General and Integer Sorting

Given a sequence of keys  $k_1, k_2, \dots, k_n$ , the problem of sorting is to rearrange this sequence either in ascending order or descending order. Sorting is of vital importance in computer science since almost every application program calls for a sorting sub routine. There are a large number of optimal sequential algorithms for sorting. Examples include merge-sort, quick-sort, heap-sort etc. The run times of all these algorithms are  $O(n \log n)$ . All these algorithms assume no prior information about the keys being sorted. On the other hand if we know some structure about the keys, sorting becomes easier. For example if the keys to be sorted are integers with  $O(\log n)$  binary bits each, sorting can be performed in  $O(n)$  time, using the radix sort algorithm.

#### DEFINITION

*Keys with no known structure will be called general keys, and keys that are integers with at the most  $O(\log n)$  bits will be called integer keys.*

Ajtai, Komlos and Szemerédi proposed the first non-trivial deterministic  $O(\log n)$  depth parallel sorting network for sorting general keys. Subsequently Leighton improved their algorithm so that it ran on a sorting network with  $O(n)$  processors in  $O(\log n)$  time (or depth). The constant in this time bound is quite large. Following this, Cole gave a PRAM algorithm that was optimal with a small constant in the time bound. Cole's algorithm is less complicated than the previous algorithm but uses non-trivial pointer updating mechanism. For this reason it is not suitable for interconnection network models. In this section we describe a simple (and optimal) randomized algorithm for general-sort. A noteworthy feature of this algorithm is that it can be extended (with some variations) to run in the interconnection networks with similar asymptotic behavior; however the description of this implementation falls outside the scope of this chapter. We shall also describe an optimal algorithm for sorting integer keys in the range  $[1, n]$ . The algorithm INTEGER\_SORT uses  $n/\log n$  processors and runs in  $\tilde{O}(\log n)$  time.

#### 3.3.1 Preliminary Results Prefix Circuits

#### DEFINITION

Let  $\Sigma$  be a domain and let  $\circ$  be an associative operation that takes  $O(1)$  sequential time over this domain. The prefix computation problem is defined as given input  $(X(1), X(2), \dots, X(n)) \in \Sigma^n$ , compute outputs  $(X(1), X(1) \circ X(2), \dots, X(1) \circ X(2) \circ \dots \circ X(n))$ . When the  $\circ$  operation is the ordinary summation, it is called the prefix sum problem.

There are many optimal algorithms to solve this problem on various models. In particular,

**LEMMA 3.4**

*Prefix computation can be performed using  $n/\log n$  processors and  $O(\log n)$  time on any PRAM.*

Prefix sum can be computed optimally in time less than  $\log n$  provided the elements are only integers of  $O(\log n)$  bits. Cole and Vishkin have proved the following

**LEMMA 3.5**

*Prefix sum computation of  $n$  integers ( $O(\log n)$  bits each) can be performed in  $O(\log n / \log \log n)$  time using  $n \log \log n / \log n$  CRCW PRAM processors.*

**DEFINITION**

*A sorting algorithm is said to be Stable if equal elements remain in the same relative order in the sorted sequence as they were in originally. In more precise terms, given input  $k_1, k_2, \dots, k_n$ , the algorithm outputs a sorting permutation  $\sigma$  of  $(1, 2, \dots, n)$  such that for all  $i, j \in [n]$ , if  $k_i = k_j$  and  $i < j$  then  $\sigma(i) < \sigma(j)$ . A sorting algorithm that is not guaranteed to output a stable sorted sequence is called Non-Stable.*

It is well known that Stable INTEGER\_SORT of  $n$  keys can be done in time  $O(n)$  by a deterministic sequential RAM.

*Notation* Throughout this chapter we let  $[m]$  stand for  $\{1, 2, \dots, m\}$ .

## An Assignment Problem

Let  $Q$  be a set  $\{1, 2, \dots, n\}$  of  $n$  indices where each index belongs to exactly one of  $m$  groups  $G_1, G_2, \dots, G_m$ . Let  $g_i$  denote the number of indices belonging to group  $G_i, i = 1, \dots, m$ . Given a sequence  $N(1), N(2), \dots, N(m)$  where  $\sum_{i=1}^m N(i) = O(n)$  and  $N(i)$  is an upper bound for  $g_i, i = 1, 2, \dots, m$ . The problem is to find a permutation of  $(1, 2, \dots, n)$  in which all the indices belonging to  $G_1$  appear first, all the indices belonging to  $G_2$  appear next, and so on. (Assume that given an index  $i$ , the group  $G_{i'}$  that  $i$  belongs to can be found in  $O(1)$  time).

For example, if  $n = 5, m = 2, G_1 = \{2, 5\}, G_2 = \{1, 3, 4\}$ , then  $(5, 2, 1, 3, 4)$  and  $(2, 5, 3, 1, 4)$  are (two of the) valid answers. It is a trivial task to design a linear time algorithm for the above problem.

**LEMMA 3.6** *Assignment Lemma*

*The above assignment problem can be solved in  $\tilde{O}(\log n)$  parallel time using  $n/\log n$  PRAM processors.*

PROOF

The following algorithm achieves the bound stated in the lemma. We use a shared memory of size  $2 \sum_{i=1}^m N(i)$  ( $= L$ , say). This memory is divided into  $m$  blocks  $B_1, B_2, \dots, B_m$  the size of  $B_i$  being  $2N(i)$ . A unique assignment for the indices belonging to  $G_i$  will be found in the block  $B_i$ , for  $i = 1, 2, \dots, m$ .

Each one of the  $P$  ( $= n/\log n$ ) processors is given  $\log n$  successive indices. More precisely, processor  $\pi$  is given the indices  $(\pi - 1)\log n + 1, (\pi - 1)\log n + 2, \dots, \pi \log n$ , for  $\pi = 1, 2, \dots, P$ . There are three phases of the algorithm. In the first phase, boundaries of the  $m$  blocks are computed. In the second phase every processor sequentially finds unique assignments for the  $\log n$  indices given to it in their **respective** blocks. In the third phase, a prefix sum computation is done to eliminate the unused cells and the position of each index in the output is computed. Details follow.

### Step 2.1

$P$  processors collectively do a prefix sum of  $(N(1), N(2), \dots, N(m))$  and hence compute the boundaries of blocks in the common memory.

**Step 2.2**

Each processor  $\pi$  is given a total time of  $d \log n$  ( $d$  being a constant to be fixed) to find assignments for all its indices sequentially.

$\pi$  starts with its first index (call it)  $l$ . If  $G_l$  is the group that  $l$  belongs to,  $\pi$  chooses a random cell in  $B_l$  and tries to write its id in it. If the chosen cell did not contain the id of any other processor and  $\pi$  succeeds in writing, then that cell is assigned to  $l$ . The probability of success in one trial is  $\geq 1/2$ . If  $\pi$  has failed in this trial then it tries as many times as it takes to find an assignment for  $l$  and then it takes up the next index.

After  $d \log n$  steps, even if there is a single processor that has not found assignments for all its keys, the algorithm is aborted and started anew.

**Step 2.3**

Each processor  $\pi$  writes a 1 in each of the cells that have been assigned to its indices. Unassigned cells in the common memory will have 0's.  $P$  processors perform a prefix sum computation on the contents of the memory cells  $(1, 2, \dots, L)$ . Finally, every processor reads out from the prefix sum the position of each one of its indices in the output.

*Analysis* Steps 1 and 3 can be completed in  $O(\log n)$  time in accordance with lemma 3.4. In step 2, the probability that a particular processor  $\pi$  successfully finds an assignment for one of its keys in a single trial is  $\geq 1/2$ . Let  $Y$  be the random variable equal to the number of successes of  $\pi$  in  $d \log n$  trials. We require  $Y$  to be  $\geq \log n$  for every processor. Clearly  $Y$  is lower bounded by a binomial variable with parameters  $(d \log n, 1/2)$ . It follows from the Chernoff bounds (equation 3) that the probability that there will be at least a single processor which has not found assignments for all of its indices after  $d \log n$  trials can be made  $\leq n^{-\alpha}$  for any  $\alpha \geq 1$ , if we choose a proper constant  $d$ .

Therefore the whole algorithm runs in time  $\tilde{O}(\log n)$ . This completes the proof of lemma 3.6 ■

### 3.3.2 Preparata's GENERAL\_SORT algorithm

Preparata's algorithm is nearly optimal and runs on *CREW PRAM*. It uses  $n \log n$  processors and runs in time  $O(\log n)$ . One of the subroutines used is an  $O(\log \log n)$  time algorithm of Valiant for merging two  $n$  element sorted sequences using  $n$  processors. The problem of merging two sorted sequences is to obtain a sorted sequence of elements of both the sequences. A trivial sequential algorithm can achieve this task in linear time. Details of Preparata's algorithm follows.

#### Step 3.1

If the problem is of constant size, solve it directly and quit.

#### Step 3.2

Partition the given  $n$  keys into  $\log n$  parts, with  $n/\log n$  keys in each part. Sort each part recursively and separately in parallel, assigning  $n$  processors to each part. Let  $S_1, S_2, \dots, S_{\log n}$  be the sorted sequences.

#### Step 3.3

Merge  $S_i$  with  $S_j$  for  $1 \leq i, j \leq \log n$  in parallel. This can be done by allocating  $n/\log n$  processors to each pair  $(i, j)$ . That is, using  $n \log n$  processors this step can be accomplished in  $O(\log \log n)$  time using Valiant's algorithm. As a by product of this merging step, we have computed the rank of each key in each one of the  $S_i$ 's ( $1 \leq i \leq \log n$ ).

#### Step 3.4

Allocate  $\log n$  processors to compute the rank of each key in the original input. This is done in parallel for all the keys by adding up the  $\log n$  ranks computed (for each key) in step 3. This can be done in  $O(\log \log n)$  time (lemma 3.4). Finally, the keys are written out in the order of their ranks.

*Analysis* Let  $T(n)$  be the run time of the above algorithm using  $n \log n$  processors. Clearly, step 1 takes  $T(n/\log n)$  time. Put together, steps 2 and 3 take  $O(\log \log n)$  time. Thus we have,

$$T(n) = T(n/\log n) + O(\log \log n),$$

which solves to  $T(n) = O(\log n)$ . Also, the number of processors used in each step is  $n \log n$ .

We summarize as follows:

**THEOREM 3.1**

*n general keys can be sorted in  $O(\log n)$  time using  $n \log n$  CREW PRAM processors.*

**COROLLARY 3.1**

*n general keys can be sorted in  $O(t \log n)$  time using  $n \log n / t$  CREW PRAM processors.*

### 3.3.3 Integer Sorting

In this section we present an algorithm `INTEGER_SORT` for sorting integer keys in the range  $[n]$ . This algorithm employs  $n / \log n$  processors and runs in time  $\tilde{O}(\log n)$ .

#### Summary of the Algorithm

The main idea behind our algorithm is radix sorting. As an example of radix sorting, consider the problem of sorting a sequence of two-bit decimal integers. One way of doing this is to sort the sequence with respect to the least significant bits (LSB) of the keys and then to sort the resultant sequence with respect to the most significant bits (MSB) of the keys. This will work, provided that in the second sort keys with equal MSBs will remain in the same relative order as they were in originally. In other words, the second sort should be stable.

Given keys  $k_1, k_2, \dots, k_n \in [n]$ , where each key is a  $\log n$ -bit integer, we first (non-stable) sort this sequence with respect to the  $(\log n - 3 \log \log n)$  LSBs of the keys. (Call this sort *Coarse\_Sort*). In the resultant sequence we apply a stable sort with respect to the  $3 \log \log n$  MSBs of the keys. (Call this sort *Fine\_Sort*).

Even though the sequential time complexity of stable sort is no different from that of non-stable sort, it seems that parallel stable sort is inherently more complex than parallel non-stable sort. This the reason why we partition the bits of the keys unevenly.

In `Coarse_Sort`, we (non-stable) sort a sequence of  $n$  keys, each key being in the range  $[1, n / \log^3 n]$  and, in `Fine_Sort` we (stable) sort  $n$  keys in

the range  $[1, \log^3 n]$ . In more formal terms, algorithm `INTEGER_SORT` can be summarized as follows.

Let  $D = n / \log^3 n$  and  $k'_i = \lfloor k_i / D \rfloor$  and  $k''_i = k_i - k'_i * D$  for all  $i \in [n]$ .

**Coarse\_Sort.** Sort  $k''_1, k''_2, \dots, k''_n \in [D]$ . Let  $\sigma$  be the resultant permutation.

**Fine\_Sort.** Stable-sort  $k'_{\sigma(1)}, k'_{\sigma(2)}, \dots, k'_{\sigma(n)} \in [\log^3 n]$ . Let  $\rho$  be the resultant permutation.

**Output.** The permutation  $\rho \cdot \sigma$ , the composition of  $\rho$  and  $\sigma$ .

In the next two sections, we describe `Fine_Sort` and `Coarse_Sort` respectively.

### Fine\_Sort

We give a deterministic algorithm for `Fine_Sort`. First we will show how to stable-sort  $n$  keys in the range  $[\log n]$  using  $n / \log n$  processors in time  $O(\log n)$  and then apply the idea of radix sorting to prove that we can stable-sort  $n$  keys in the range  $[(\log n)^{O(1)}]$  within the same resource bounds.

#### LEMMA 3.7

*$n$  keys  $k_1, k_2, \dots, k_n \in [\log n]$  can be stable-sorted in  $O(\log n)$  time using  $P = n / \log n$  processors.*

#### PROOF

In `Fine_Sort` algorithm, each processor  $\pi$  is given  $\log n$  successive keys. Each one of the  $P$  processors starts by sequentially stable-sorting the keys given to it. Then, collectively, the  $P$  processors group all the keys with equal values. (There are  $\log n$  groups in all). Finally, they output a rearrangement of the given sequence in which all the 1's (i.e., keys with a value 1) appear first, all the 2's appear next, and so on. Throughout the algorithm the relative order of equal keys is preserved. More details follow.

To each processor  $\pi \in [P]$  we assign the key indices  $J(\pi) = \{j | (\pi - 1) \log n < j \leq \min(n, \pi \log n)\}$ . There are three steps in the algorithm.

#### Step 4.1

Each processor  $\pi$  sequentially stable-sorts the keys  $\{k_j | j \in J(\pi)\}$  in time  $O(\log n)$ , and hence constructs  $\log n$  lists  $J_{\pi,k} = \{j \in J(\pi) | k_j = k\}$  for  $k \in [\log n]$ . Elements in  $J_{\pi,k}$  are ordered in the same relative order as in the input.

**Step 4.2**

The  $P$  processors collectively perform the prefix sum of

$$\begin{aligned} &(|J_{1,1}|, |J_{2,1}|, \dots, |J_{P,1}|, \\ &|J_{1,2}|, |J_{2,2}|, \dots, |J_{P,2}|, \\ &\dots \\ &|J_{1,q}|, |J_{2,q}|, \dots, |J_{P,q}|) \end{aligned}$$

where  $q = \log n$ . Call this sum

$$\begin{aligned} &(S_{1,1}, S_{2,1}, \dots, S_{P,1}, \\ &S_{1,2}, S_{2,2}, \dots, S_{P,2}, \\ &\dots \\ &S_{1,q}, S_{2,q}, \dots, S_{P,q}). \end{aligned}$$

**Step 4.3**

Each processor  $\pi$  sequentially computes the position of each one of its keys in the output using the prefix sum. The position of keys in the list  $J_{\pi,l}$  will be  $S_{\pi-1,l} + 1, S_{\pi-1,l} + 2, \dots, S_{\pi,l}$ .

*Analysis* It is easy to see that steps 1 and 3 can be performed within the stated resource bounds. Step 2 also can be completed within the stated resource bounds as stated in lemma 3.4

This concludes the proof of Lemma 3.7. ■

**LEMMA 3.8**

*If  $n$  keys in the range  $[R]$  (for any  $R = n^{O(1)}$ ) can be stable-sorted in  $O(\log n)$  time using  $P = n/\log n$  processors, then  $n$  keys  $k_1, k_2, \dots, k_n \in [R^2]$  can be stable-sorted in time  $O(\log n)$  using the same number of processors.*

PROOF

Let  $k'_i = \lfloor k_i/R \rfloor$  and  $k''_i = k_i - k'_i * R$  for every  $i \in [n]$ . First, stable-sort  $k''_1, k''_2, \dots, k''_n$  obtaining a permutation  $\sigma$ . Then stable-sort  $k'_{\sigma(1)}, k'_{\sigma(2)}, \dots, k'_{\sigma(n)}$  obtaining a permutation  $\rho$ . Output  $\rho.\sigma$ . Clearly both these sorts can be completed in time  $O(\log n)$  using  $P$  processors. ■

Lemmas 3.7 and 3.8 immediately imply the following

**LEMMA 3.9**

$n$  integer keys in the range  $[(\log n)^{O(1)}]$  can be stable-sorted in time  $O(\log n)$  using  $n/\log n$  processors.

**Coarse\_Sort**

In this sub-section we fix a key domain  $[D]$  where  $D = n/\log^3 n$ . We assume for the sake of convenience,  $\log^3 n$  divides  $n$ . Let the input keys be  $k_1, k_2, \dots, k_n \in [D]$ . Define the *index sequence* for each key  $k \in [D]$  to be  $I(k) = \{i | k_i = k\}$ . The randomized algorithm for Coarse\_Sort to be presented in this sub-section employs  $P = n/\log n$  processors and runs in time  $\tilde{O}(\log n)$ . The sorted sequence is non-stable.

The main idea is to calculate the cardinalities of the index sequences  $I(k), k \in [D]$  approximately, and then to use the assignment algorithm in the proof of lemma 3.6 to rearrange the given sequence in sorted order.

**LEMMA 3.10** *Estimation Lemma*

Given as input  $k_1, k_2, \dots, k_n \in [D]$  we can compute  $N(1), N(2), \dots, N(D)$  in  $\tilde{O}(\log n)$  time using  $P = n/\log n$  processors such that  $\sum_{k \in [D]} N(k) = O(n)$  and furthermore, with very high likelihood  $N(k) \geq |I(k)|$  for each  $k \in [D]$ .

PROOF

The following sampling algorithm serves as a proof.

**Step 5.1**

Each processor  $\pi \in [D \log n]$  in parallel chooses a random index  $s_\pi \in [n]$ . Let  $S$  be the sequence  $\{s_1, s_2, \dots, s_{D \log n}\}$ .

**Step 5.2**

The  $P$  processors collectively sort the keys with the chosen indices. That is, they sort  $k_{s_1}, k_{s_2}, \dots, k_{s_{D \log n}}$  and compute index sequences  $I_S(k) = \{i \in S | k_i = k\}$  (for each  $k \in [D]$ ).

**Step 5.3**

$D$  of the  $P$  processors in parallel set  $N(k) = d(\log^2 n) \max(|I_S(k)|, \log n)$

for  $k \in [D]$ ,  $d$  being a constant to be fixed in the analysis. Output  $N(1), N(2), \dots, N(D)$ .

*Analysis* Trivially, steps 1 and 3 can be performed in  $O(1)$  time. Step 2 can be performed using Preparata's GENERAL\_SORT algorithm in  $O(\log n)$  time. (Notice that we have to sort only  $n/\log^2 n$  keys in step 2). It remains to be shown that  $N(i)$ 's computed by the sampling algorithm satisfy the conditions in lemma 3.10.

If  $|I(k)| \leq d \log^3 n$ , then always  $N(k) \geq d \log^3 n \geq |I(k)|$ . So suppose  $|I(k)| > d \log^3 n$ . Then it is easy to see that  $|I_S(k)|$  is a binomial variable with parameters  $(\frac{n}{\log^2 n}, \frac{|I(k)|}{n})$ . The Chernoff bounds (see lemma 3.1, equation 2) imply that for all  $\alpha \geq 1$ , there exists a  $c$  such that

$$\text{Prob.}(|I_S(k)| \leq c\alpha |I(k)| / \log^2 n) \leq \frac{1}{n^\alpha}.$$

Therefore, if we choose  $d = (c\alpha)^{-1}$  then  $N(k) \geq |I(k)|$  (for every  $k \in [D]$ ) with probability  $\geq 1 - n^{-\alpha}$ . The Chernoff bounds (lemma 3.1, equation 3) also imply that for all  $\alpha \geq 1$  there exists a  $h$  such that  $N(k) \leq (h\alpha)|I(k)|$  (for every  $k \in [D]$ ) with probability  $\geq 1 - n^{-\alpha}$ .

The bound on  $\sum_{k \in [D]} N(k)$  clearly holds since

$$\begin{aligned} \sum_{k \in [D]} N(k) &\leq \sum_{k \in [D]} d \log^2 n [|I_S(k)| + \log n] \\ &= d \log^3 n D + d \log^2 n \sum_{k \in [D]} |I_S(k)| \\ &= dn + d \log^2 n D \log n = 2dn \end{aligned}$$

This concludes the proof of lemma 3.10 ■

Having obtained the approximate cardinalities of the index sets, we apply the assignment algorithm. The set  $Q$  is the set of key indices, namely  $\{1, 2, \dots, n\}$ . An index  $i$  belongs to group  $G_{i'}$  if the value of the key with index  $i$  is  $i'$ . Under this definition, group  $G_j$  is the same as index sequence  $I(j)$ ,  $j = 1, 2, \dots, D$ . Since we can find approximate cardinalities of these groups (lemma 3.10), we can use the assignment algorithm to rearrange the given sequence in sorted order. Thus we have the following

**LEMMA 3.11**

$n$  keys  $k_1, k_2, \dots, k_n \in [D]$  can be sorted in time  $\tilde{O}(\log n)$  time using  $n/\log n$  processors.

Lemmas 3.9 and 3.11 together with the algorithm summary in section 3 prove the following

**THEOREM 3.2**

$n$  integer keys in the range  $[n]$  can be sorted in  $\tilde{O}(\log n)$  time using  $n/\log n$  CRCW PRAM processors.

**3.3.4 Reischuk's Algorithm for GENERAL\_SORT**

The original algorithm of Reischuk for sorting general keys was recursive and hence the analysis was tedious. We give a non-recursive version of his algorithm. This algorithm will use  $n$  processors and run in time  $\tilde{O}(\log n)$ . The underlying constant is small. The basis for this algorithm is Preparata's sorting scheme.

Reischuk's algorithm runs in the same time bound as Preparata's (with high probability), but using only  $n$  processors. The idea is to randomly sample  $N = \frac{n}{\log^4 n}$  keys from the input and sort these using a non optimal algorithm like Preparata's. The sorted sample partitions the original problem into  $N$  independent subproblems of nearly equal size, and hence all these subproblems can be solved easily. These ideas are made concrete in the following algorithm.

**Step 6.1**

$N = n/(\log^4 n)$  processors randomly sample a key (each) from  $X = k_1, k_2, \dots, k_n$ , the given input sequence.

**Step 6.2**

Sort the  $N$  keys sampled in step 1 using Preparata's algorithm. Let  $l_1, l_2, \dots, l_N$  be the sorted sequence.

**Step 6.3**

Let  $X_1 = \{k \in X | k \leq l_1\}$ ;  $X_i = \{k \in X | l_{i-1} < k \leq l_i\}$ ,  $i = 2, 3, \dots, N-1$ ;  $X_N = \{k \in X | k > l_N\}$ . Partition the given input  $X$  into  $X_i$ 's as

defined. This is done by first finding the part each key belongs to (using binary search in parallel). Now partitioning the keys reduces to sorting the keys according to their part numbers.

#### Step 6.4

For  $1 \leq i \leq N$  in parallel do: sort  $X_i$  using Preparata's algorithm.

#### Step 6.5

Output  $\text{sorted}(X_1), \text{sorted}(X_2), \dots, \text{sorted}(X_N)$ .

*Analysis* Step 2 can be done using  $N \log N \leq N \log n$  processors in  $O(\log N) = O(\log n)$  time (theorem 3.1). In step 3, partitioning of  $X$  can be done using binary search and the INTEGER\_SORT algorithms. Thus step 3 can be performed in  $\tilde{O}(\log n)$  time, using  $\leq n$  processors. With high probability there will be no more than  $O(\log^5 n)$  keys in each of the  $X_i$ 's ( $1 \leq i \leq N$ ). Proof of this fact is left as an exercise. Within the same processor and time bounds, we can also count  $|X_i|$  for each  $i$ . In step 4, each  $X_i$  can be sorted in  $O(\log |X_i|)$  time using  $|X_i| \log |X_i|$  processors. Also  $X_i$  can be sorted in  $(\log |X_i|)^2$  time using  $|X_i|$  processors (see corollary 3.1). Thus step 4 can be completed in  $(\max_i \log |X_i|)^2$  time using  $n$  processors. If  $\max_i |X_i| = O(\log^5 n)$ , step 4 takes  $O((\log \log n)^2)$  time.

Thus we have proved the following

#### THEOREM 3.3

*We can sort  $n$  general keys using  $n$  CRCW PRAM processors in  $\tilde{O}(\log n)$  time.*

### 3.4

#### Recursive Parallel Divide and Conquer

Although we presented a non-recursive version of Reischuk's parallel sorting algorithm, it is instructive to analyze the recursive version of his algorithm. This will be used later for more general situations where the algorithms

are recursive by nature. In Reischuk's original algorithm,  $\lfloor \sqrt{n} \rfloor$  keys were chosen randomly such that each key was chosen with probability  $\frac{1}{\sqrt{n}}$ . These keys (called splitters) were then sorted using pairwise comparisons 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  $\lfloor \sqrt{n} \rfloor + 1$  buckets. For each key we can determine the appropriate bucket by a simple binary search using one processor for each key (using simultaneous reads). If we let  $n_i$  denote the size of the  $i$ -th bucket then we claim the following:

**LEMMA 3.12**

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

PROOF

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

$$\bar{T}(n) \leq \left(1 - \frac{1}{n^2}\right)\bar{T}(3\sqrt{n}\log n) + \frac{1}{n^2}\bar{T}(n - \lfloor \sqrt{n} \rfloor + 1)$$

By induction it can be shown that  $\bar{T}(n) \leq O(\log n)$  and we leave it as an exercise problem. ■

To derive high probability bounds, we shall actually derive a more general bound. This will be used repeatedly for analyzing algorithms in future. We shall also make use of the fact that Reischuk's algorithm would execute in the same asymptotic time bound even if the number of sampled keys is  $\lfloor n^\epsilon \rfloor$  for any fixed  $\epsilon$ ,  $0 < \epsilon \leq 1/2$ . Note that in general, the probabilistic bound of lemma 3.12 holds for partitions of sizes  $O(n^{1-\epsilon}\log n)$  (the lemma was for

the case  $\epsilon = 1/2$ ). For convenience, we shall use the bound  $O(n^{\epsilon_0})$  where  $\epsilon_0 > 1 - \epsilon$ . Thus at depth  $i$  from the root, the size of a sub-problem can be bounded from above by  $n^{\epsilon_0^i}$ .

It may be helpful to view the algorithm as a tree where a node represents a subproblem and its children represent the recursive calls made by this node. For example, the root represents the procedure  $\text{Sort}[1..n]$  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 Batchers's sort to sort in time  $O(\log \log n)$  thereby adding a factor of  $o(\log n)$ . Our objective is to show that all the leaf-level procedures are completed in  $O(\log n)$  time with high probability. For this it suffices to show that a particular leaf-level procedure is completed in  $O(\log n)$  time. This leaf-node defines a fixed path from the root to the leaf such that the problem sizes at successive nodes of this path are decreasing. For this let us denote the node at depth  $i$  from the root as  $N_i$ , the problem-size as  $n_i$  and the time taken at  $N_i$  by  $T_i$ . We claim the following:

**LEMMA 3.13**

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

where  $c$  and  $\alpha$  are integers and  $k$  is a constant.

PROOF

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

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

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

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

The above inequality implies that

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

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

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

**THEOREM 3.4**

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

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

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

PROOF

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

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

If  $T$  is the total time for this worst case chain of nested calls and  $m = 1/(1-\epsilon_0)$ , the probability that it takes more than  $mk\alpha \log n c_o + t$ , is less than the sum of the probability of events where  $\sum_i t_i = t + j$ ,  $0 \leq j \leq \mu$ . Here  $\mu = mk\alpha \log n c_o$ . We shall compute the probability that  $\sum_i t_i = t$  and multiply by  $\mu$ .

$$\prod \sum_{t_i=t} 2^{-t_i} \leq \sum 2^{-t/k} \text{ over } t^{O(\log \log n)} \text{ tuples.}$$

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

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

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

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

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

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

## 3.5 Higher dimensional Problems: Convex Hulls

We shall now try to apply the techniques developed in the previous section to a specific geometric problem on the plane. For this purpose we have chosen the problem of constructing two-dimensional convex hull of point sites. Although optimal  $O(\log n)$  time parallel algorithms have been known for some time, its close relation to sorting makes it a natural candidate for the methods developed in the earlier part of this chapter. Moreover, the additional methods that will be developed for this problem are applicable to more general situations and the reader will be referred to the relevant literature at the end of this chapter.

### 3.5.1 A straightforward extension

We shall actually look at its dual problem, namely, the intersection of  $n$  half-planes. For readers unfamiliar with this transform, a brief sketch is included under Notes and References at the end of the chapter. Without loss of generality we can assume that the origin lies in its interior (we can ensure that the origin lies in the interior of the primal problem from which this property will hold trivially). Following on the lines of the sorting algorithm, we choose a random subset of  $\lfloor n^\epsilon \rfloor$  half-planes where  $0 < \epsilon < 1$ . We can construct their intersection in  $O(\log n)$  time using a brute-force method like checking for each pairwise intersection, if it is a vertex of the convex hull. Let  $h_0, h_1, \dots$  be the vertices of the convex-hull in a cyclic order (there can be at most  $\lfloor n^\epsilon \rfloor$  of them). Consider the triangles (will also be referred to as sectors) of the form  $O, h_1, h_2$ . These will be intersected by a number of half-planes that were not chosen in the sample. The output convex hull is the union of the boundaries formed by the intersection of the half-planes inside each of the sectors. This gives a recursive procedure for constructing the convex hull. For each sector, we determine the half-planes intersecting the region and then call the algorithm recursively for each of the sectors.

To determine the sectors that a half-plane intersects we can use a very simple procedure due to Chazelle and Dobkin which uses *Fibonacci search* on a bimodal sequence to determine extremal points. The distances of the vertices of a convex  $n$ -gon from a line form a bimodal sequence and the closest vertex (including the intersecting points) can be determined using *Fibonacci search* which is similar to a binary search on the cyclic sequence of the vertices. The interested reader is encouraged to look up the reference mentioned

in the last section for further details of this search procedure. Using one processor for every half-plane, in  $O(\log n)$  time we can determine the edges of the convex hull that the half-plane intersects which gives us the information of the bounding sectors that this half-plane intersects. Because of convexity, it intersects all the sectors that lie in between. Note that we can very easily determine the number of sectors that it intersects in the same time (without explicitly listing the sectors).

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

**LEMMA 3.14**

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

PROOF

Consider all the  $O(n^2)$  pairwise intersections defined by the lines bounding the half-planes. Draw the segments joining  $O$  and each of these intersections and consider the ordered intersections of lines (representing the boundaries of half-planes) on this segment. For any given segment the probability that the number of intersections exceeds  $(c + 2)n^{1-\epsilon} \log n$  before the first (counting from  $O$ ) sampled half-plane is less than  $(1 - \frac{1}{n^{1-\epsilon}})^{(c+2)n^{1-\epsilon} \log n}$  which is less than  $n^{-(c+2)}$  for large  $n$ . Thus the probability that this event happens for any segment is less than  $n^{-c}$ . This implies the lemma since any half-plane intersecting the sector intersects at least one of the two bounding segments. ■

**LEMMA 3.15**

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

PROOF

From the proof of the previous lemma, it is clear that if the number of halfplanes intersecting a segment is greater than  $n^{1-\epsilon}$ , then the expected number of half-planes obeys a geometric distribution. The probability of success is  $1/n^{1-\epsilon}$  which is the probability of being selected in the sample. So the expected number of half-planes that we do not select in the ordered list of half-planes (starting from  $O$ ) before we select the first half-plane is  $n^{1-\epsilon}$ . Using the property that the expectation of the sum of random variables is the sum of the individual expectations, we arrive at the required bound. In our case we are interested in the sum of  $n^\epsilon$  random variables. Note that in the proof there is no conditioning on the set of half-planes chosen; rather it is on the number of such planes chosen. Technically the number of such planes chosen is a random variable whose expected value is  $n^\epsilon$ . The reader is encouraged to solve Exercise 14. ■

In the parallel setting if we use as an abstract representation of a recursive algorithm the tree as described in the previous section, the running time of the algorithm is proportional to the longest path (in time) from the root to a leaf. Thus at any given node we are interested in the recursive call that takes the the longest time. Given that we only have a bound for the *expected* time taken by the child-procedures, and there are  $n^\epsilon$  of them, we cannot derive any useful tail estimates. The reason that we could derive interesting bound for the sorting algorithm is because we were able to get tail estimates (of the problem size exceeding a certain size). However, for sequential algorithms one may obtain bounds on the *expected* running time by the using the linearity property of *expectations*. We shall come back to this issue at the end of the chapter.

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

#### LEMMA 3.16

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

- (i) The maximum number of half-planes intersecting any sector is less than  $2n^{1-\epsilon} \log n$*

(ii) The sum of half-planes taken over all the sectors of the number of half-planes intersecting a sector is less than  $k_{total}n$ .

#### DEFINITION

We shall call a random sample good if the above conditions are satisfied and bad otherwise.

#### PROOF

From Lemma 3.15 and Markov's inequality we can choose  $k_{total}$  such that the probability that (ii) fails is at most  $1/3$  (i.e.  $kn$  is thrice the mean). For sufficiently large  $n$ ,  $1/n + 1/3$  is less than  $1/2$ . Thus the probability that both (i) and (ii) are satisfied is at least  $1/2$ . ■

### 3.5.2 Resampling and Polling

As a consequence of the previous claim, if we repeat the sampling algorithm  $r \log n$  times, the probability that the conditions are not satisfied during all the tries is less than  $n^{-r}$ . That is if we choose independently  $p(n) = O(\log n)$  sets of samples, one of them is good with very high likelihood. However, to determine if a sample is 'good', we would have to carry out the search procedure  $O(\log n)$  times each of which requires  $O(\log n)$  time (such a method was described earlier). Instead, we try to estimate the number of half-planes intersecting a sector  $C_i$  using only a fraction of the input half-planes. For example, we can choose  $c_0 \cdot n/\log^d n$  for some fixed integer  $d > 2$  and a constant  $c_0$  (the actual value will be determined from the required success probability of the algorithm) of the input half-planes randomly for the  $j$ th sample,  $R_j$ . Let  $X_i^j$  be the number of half-planes intersecting sector  $C_i$  corresponding to sample  $R_j$ ,  $1 \leq j \leq b \log n$  where  $b$  is fixed integer greater than 0 which is determined from the success probability of the algorithm.  $A_i^j$  be the number of half-planes intersecting  $C_i$  out of the  $n/\log^d n$  randomly chosen input half-planes for the same sample. Clearly,  $A_i^j$  is a binomial random variable with parameters  $c_0 \cdot n/\log^d n$  (number of trials)  $X_i^j/n$  (probability of success). Assuming that  $X_i^j$  is greater than  $\bar{c} \cdot \log^{d+1} n$ , for some constant  $\bar{c}$ , we will apply Chernoff bounds to tightly bound the estimates within a constant multiplicative factor. Since we do it only for  $1/\log^d n$  of the input half-planes, the total number of operations for the  $O(\log n)$  random subsets

can be bounded by  $O(n \log n)$  (as we show in the next section). Note that  $X_i^j < \bar{c} \log^{d+1} n$ , is an easy case since  $n^\epsilon \cdot \bar{c} \log^{d+1} n = o(n)$ .

### 3.5.3 Probabilistic analysis of Polling

More formally, by invoking Chernoff bounds, for any  $\alpha > 0$  ( $\alpha$  is a function of  $c_0$ ), there exists a  $c_1$ , independent of  $n$ ,  $\text{Prob}(A_i^j \leq \alpha c_1 X_i^j / \log^d n) \leq 1/n^\alpha$  and  $\text{Prob}(A_i^j \geq c_2 \alpha c_0 \cdot X_i^j / \log^d n) < 1/n^{c_0 \alpha} < 1/n^\alpha$  (for  $c_0 > 1$ ).

From the last two inequalities,  $X_i^j$  is bounded by  $L^j = A_i^j \log^d n / c_0 c_2 \alpha$  from below, and by  $U^j = A_i^j \log^d n / c_1 \alpha$  from above. With appropriate changes in the constants, this condition holds with high likelihood (as defined in section 2.1) for all  $X_i^j$  simultaneously. We do the procedure (described in the next section) simultaneously for all the samples  $R_j$  and choose the sample  $R^{j_0}$  using the following simple test:

#### ALGORITHM 3.1

*Finding a good sample using Polling*

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

*Output:* A good sample  $R^{j_0}$ .

*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.

(clearly good)

**If**  $k_{total} n > U^j$  **then** accept sample  $R^j$  (since  $k_{total} n \geq U^j \geq T^j$ ),

(clearly bad)

**if**  $k_{total} n \leq L^j$  **then** the sample is ‘bad’ (since  $k_{total} n \leq L^j \leq T^j$ ),

(choose the best)

**if**  $L^j \leq k_{total} n \leq U^j$ , **then** accept the sample  $R^{j_0}$  for which  $N^{j_0}$  is minimum. Since both  $k_{total} n$  and  $T^{j_0}$  lie in this interval this guarantees that  $T^{j_0} \leq c_3 \cdot k_{total} n$  where  $c_3 = U^j / L^j$  which is a constant.

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

**LEMMA 3.17**    *Polling lemma*

*Using the previous procedure we can obtain a sample that is ‘good’ with high probability.*

The more careful reader would have noticed there is a technical inconsistency with the above claim. From our definition of a good sample, the sum of the half-planes should be less than  $k_{total} \cdot n$ , 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  $O(n)$  with high probability. The above procedure can be used in a more general situation where we need ‘good’ samples with very high likelihood from samples that only expect to be ‘good’. Moreover, according to our previous discussion, the extra amount of overhead does not affect the asymptotic work done by the algorithm, because it uses only a fraction of the input to test the samples.

### 3.5.4 Bounding the number of processors

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

For this we shall use geometric properties of the specific problem. After getting a ‘good’ sample we would do some further processing to bound the sum of sizes of the sub-problems by the exact size of the parent problem. This will prevent proliferation in the problem size over successive recursive calls. This *filtering* procedure is a kind of post-processing step after random-sampling. The input size is at most a constant factor times the input size (guaranteed

by the polling algorithm) of the problem while the output is no more than the input size.

In case of two-dimensional convex hulls, we make use of the following filtering scheme. For any sector, we identify the half-planes that intersect it. Some of them may be part of the output while some of them may not show up in the output (in that sector). Since the output size is bounded by the input size, our objective is to eliminate all the half-planes from a sector that do not show up in the output. There are the following cases to consider.

**Case 1:** If a half-plane is occluded completely by another half-plane within a sector (see Figure 3.1 a), then we can discard these half-planes by the following strategy. For every half-plane intersecting a sector, consider the points of intersection with the two boundaries of the sector. Sort these intersection points in increasing order starting from  $O$ . At the end of this step we have two sorted lists. For every half-plane consider a tuple of the form  $(x_i, y_i)$  where  $x_i$  and  $y_i$  are respectively the ranks of the sorted intersection on the boundaries. A half-plane  $H_i$  is completely occluded by another half-plane  $H_j$  if and only if  $x_i > x_j$  and  $y_i > y_j$ . In other words, if we compute the maximal elements of the tuples, then the elements that are not a part of this set can be left-out from further calls of recursion. This is the easy case.

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

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

**Case 4:** If a half-plane is visible in more than one sector, then in all but two sectors it will eliminate all other half-planes by case 1 and moreover we do not have to call the algorithm in these sectors (since we know the output). This half-plane can contribute to vertices of the output hull in at most two sectors. During subsequent recursive calls on these two sectors at most one copy will be retained in each.

From a global view-point, at any stage of the algorithm, we retain at most two copies of a half-plane. If we let an output vertex to be represented by the two

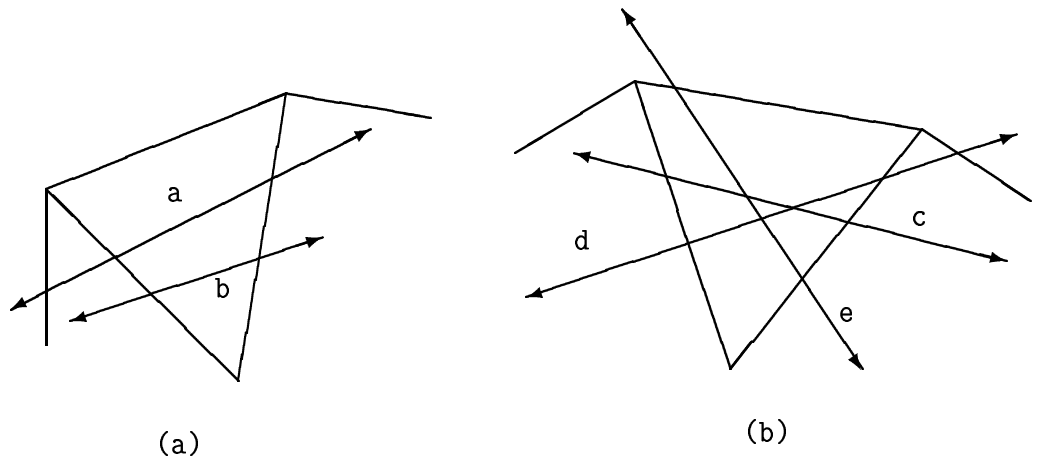


Figure 2: (a) illustrates case (a) - line a is completely occluded by b. (b) illustrates case (b). Line c is occluded by lines d and e but not by any one of them completely. Also it is clear that in all other cones line c will be eliminated by case (a).

intersecting lines, a half-plane can appear in at most two such tuples, which we can denote by the left-vertex ( $l_v$ ) and the right-vertex ( $r_v$ ). The *output size* is defined as twice the number of vertices. (Thus if a sector does not have a vertex the output size is 0 implying that an edge traverses the sector and hence we have already determined the convex hull in that sector). Assume that we have  $2n$  processors for the algorithm. Then our processor-allocation strategy is as follows:

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

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

To summarize, we have achieved the following:

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

We now have a parallel algorithm for constructing a convex hull in two dimensions which satisfies the properties of parallel sorting algorithm, namely lemma 3.12 (although we had to work much harder to get to it). Hence by invoking Theorem 3.1, we can claim that the above algorithm runs in  $\tilde{O}(\log n)$  using  $n$  processors in a *CREW PRAM* model.

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

- (1) Select  $O(\log n)$  subsets of random objects (in case of 2-D hulls these were half-planes) each of size  $\lfloor n^\epsilon \rfloor$  for some  $0 < \epsilon < 1$ .
- (2) Select a ‘good’ sample using *Polling*
- (3) Divide the original problem into smaller sub-problems (the maximum size can be bounded by  $O(n^{1-\epsilon} \log n)$ ) using the ‘good’ sample.
- (4) Use a *Filtering* scheme to bound the sum of the sub-problem

size by some fixed measure like the output size or input size. This step is problem dependent and uses the specific geometry properties of a problem. The purpose is to bound the number of processors.

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

The above general strategy has been used successfully to obtain efficient algorithms for a number of fundamental problems like triangulation and convex-hulls in three dimensions. However the implementation of some of these steps depend heavily on the specific problem. The probabilistic bounds used in step 3 have to be proved for the specific problem. In this regard, Clarkson presented bounds for very general situations which are applicable to a certain extent; however we have chosen to present alternate arguments which are simpler. Moreover, the procedure used for dividing the subproblems would naturally depend on the problem at hand. Perhaps the step that is most specific to a problem is the *Filtering* step where we have to use the geometry of the problem.

## Notes and References

Selection and Sorting were among the first problems that captured the attention of researchers in parallel algorithms. One of the earliest significant results was obtained by Valiant [V75]. He proved a lower bound of  $\Omega(\log \log n)$  for extremal selection which we overcame by use of randomization. Beame and Hastad [BH87] proved a lowerbound of  $\Omega(\log n / \log \log n)$  for general selection as long as one uses a polynomial number of *CRCW PRAM* processors. For some special cases (Exercise 16), the lower bound can be circumvented ([S90]). The first optimal  $O(\log n)$  time algorithm for selection was presented by Reischuk [R81] (Exercise 4) adopting the sequential algorithm of Floyd and Rivest [FR75]. These techniques were extended by Rajasekaran to obtain an optimal randomized selection algorithm for the hypercube [R90].

The first optimal  $O(\log n)$  time PRAM sorting algorithm was obtained by Reischuk [R81] using random-sampling. The basic methodology was adapted for the *Flashsort* algorithm by Reif and Valiant [RV87] around the same time

as the celebrated AKS network was proposed by Ajtai, Komlos and Szemerédi [AKS83]. Leighton [L84] reduced the processor complexity in AKS to obtain a truly optimal deterministic algorithm. However the AKS has suffered due to astronomical constants involved in the construction the underlying network which is an expander graph. More recently Cole [C86] designed an elegant  $O(\log n)$  time sorting algorithm which has virtually settled the problem of sorting on PRAM models; however Flashsort continues to remain the most practical algorithm for networks. The optimal sub-logarithmic algorithm for prefix-sum stated in lemma 3.5 was discovered by Cole and Vishkin [CV86]. The first optimal sub-logarithmic time algorithms for General sorting and integer sorting were provided by Rajasekaran and Reif [RR89].

The presentation of our general sorting algorithm uses ideas drawn from a lot of the earlier work and was given in Rajasekaran and Reif [RR87]. [RR87] also provides a survey of parallel selection and sorting algorithms. The first optimal algorithm for integer sorting in the range  $[n]$  was given in [RR89]. Integer sorting in range  $[n^2]$  continues to remain a challenging problem although significant progress has been achieved recently (by Bhatt et al. [B89], and by Rajasekaran and Sen [RS87]).

The algorithms for general-sorting and convex hulls presented in this chapter are not only optimal in  $PT$  bounds but are also optimal in the time bounds for the model ( $CREW$ ) used. However, for the stronger  $CRCW$  model one can actually improve some of the algorithms to obtain an optimal time bound of  $\Theta(\frac{\log n}{\log \log n})$ . This is discussed in [RR89].

Use of randomized methods for parallel computational geometry was introduced by Reif and Sen [RS87]. In Reif and Sen [RS89], they provide further applications of these methods to three-dimensional convex hulls and 2-D Voronoi diagrams. Random sampling in computational geometry has proved to be very useful especially in a sequential context. Clarkson [CL88] introduced the use of random sampling to computational geometry and derived similar bounds for a more general setting using more involved techniques. He had also given numerous applications of these very general probabilistic bounds. In his case, however he was dealing with the sequential algorithms and so he could derive bounds on the expected running time of the algorithms as the sum of the expected time for individual steps.

Another direction for research in randomized algorithms is to minimize the use of random bits in the algorithms. This is especially crucial in practical situations where it is often difficult to generate *truly* random bits (as opposed to *pseudo-random bits*). Using techniques of Chor and Goldreich [CG89],

Karloff and Raghavan [KR88] were able to show that Reischuk's algorithm can be made to run in the same asymptotic bounds using only  $O(\log n)$  *purely* random bits. Their methods were further extended by Reif and Sen [RS89] to show that some of the algorithms in computational geometry (including the convex-hull algorithm) can be implemented using  $O(\log^2 n)$  bits.

### The dual transform

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

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

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

where  $h^{\text{pos}}$  denotes the half-space containing the origin.

It can be verified that, 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.

## 3.6 Exercises

- 3.1 Prove Lemma 3.2 for any fixed  $\epsilon$ .
- 3.2 Prove Lemma 3.3.
- 3.3 Given a two-sided biased coin (i.e., the probability of a *head* is not  $1/2$ ). How will you use it to simulate an unbiased coin? Also, how will you simulate an  $n$ -sided coin using a 2-sided coin?
- 3.4 (Parallel Selection). Using the bounds of lemma 3.3, design a simple algorithm for choosing the  $k$ -th largest element for any  $1 \leq k \leq n$ . Prove that

your algorithm runs in  $\tilde{O}(\log n)$  time using  $n/\log n$  *EREW PRAM* processors, i.e. it is *PT* optimal.

*Hint:* In the first phase, select two elements such that the required element is smaller than one and larger than the other. Show that the total number of elements that lie between these two elements is  $O(n^\epsilon)$  for some  $\epsilon < 1$  with high probability.

- 3.5 Let  $A[1..n]$  be an array of  $n$  elements. An element  $x$  of  $A$  is said to be a *semi-majority element* if  $|\{i : A(i) = x\}| \geq \frac{n}{4}$ . Give an  $\tilde{O}(\log n)$  time  $n/\log n$  *EREW PRAM* processor algorithm to find all the semi-majority elements of  $A$ .
- 3.6 Prove the claims about the size bounds after steps 3 and 4 of the algorithm for maximal selection. Do the same for step 3 of the General sorting algorithm.
- 3.7 Given a coin for which the probability of getting a head is at least  $\alpha$ ,  $0 < \alpha < 1$ , prove that there is a constant  $c$  ( $c > 1$ ) such that with high probability, there are at least  $\log n$  heads if the coin is tossed  $\frac{c \log n}{\alpha}$  times.
- 3.8 You roll an  $N$  sided dice. If you get  $n$  ( $1 \leq n \leq N$ ), you roll an  $n$ -sided dice. What is the expected number of times you have to roll to get a 1? Use this result to derive an expected time bound for Quicksort.
- 3.9 Let  $S_1, S_2, \dots, S_k$  (where  $k = (\log n)^{O(1)}$ ) be sets of integers in the range  $[1, n(\log n)^{O(1)}]$ . Given also that  $\sum_{i=1}^k |S_i| = n$ . Present an  $\tilde{O}(\log n)$  time *CRCW PRAM* algorithm that sorts all the  $k$  sets using  $n/\log n$  processors.
- 3.10 Given  $n$  integers in the range  $[1, n^{O(1)}]$ . If the computer word length is  $n^\epsilon$ , for some fixed  $0 < \epsilon < 1$ , how will you sort these keys in  $O(\log n)$  time using  $n/\log n$  *CREW PRAM* processors?
- 3.11 If each one of  $n$  integers is picked randomly and uniformly from the range  $[1, n^{O(1)}]$ , how will you sort them using  $n/\log n$  *CRCW PRAM* processors such that for a large fraction of all possible inputs the algorithm terminates in  $\tilde{O}(\log n)$  time?  
*Hint* Make use of the integer sorting algorithm.
- 3.12 Given  $n$  keys (not necessarily integers) with many duplications such that the number of distinct keys is  $(\log n)^{O(1)}$ . Present an  $\tilde{O}(\log n)$  time,  $\frac{n \log \log n}{\log n}$  *CRCW PRAM* processor algorithm to sort this input.
- 3.13 Consider a probabilistic experiment where there are  $N$  events of interest each of which has a success probability of  $\frac{1}{2}$  independent of each other. We consider the experiment to be successful if all the  $N$  events are successful. What can you say about the success probability of the experiment if you

have  $O(\log N)$  independent runs of the experiment? Can you obtain any meaningful bound as we could do with *Polling*?

- 3.14 Let  $X_i$ , be a family of random variables (not necessarily independent) such that the mean of each  $X_i$  is less than  $\mu$ . Suppose  $Y = \sum_{i=1}^n X_i$  where  $n$  is a random variable (integral valued) with mean  $N$ . Show that the expectation of  $Y$  can be bound by  $N \cdot \mu$ .  $Y$  is called a *random sum*; it is the sum of a random number of random variables.  
*Hint* : Use the method of conditional expectation.
- 3.15 Present an  $\tilde{O}(n)$  algorithm to compute an *approximate rank* of a given element  $x$  in a set of  $n$  keys. An approximate rank of any element  $x$  is an integer in the range  $[r - \delta n, r + \delta n]$ , where  $r$  is the true rank of  $x$  and  $\delta$  is a fixed number ( $0 < \delta < 1$ ).
- 3.16 \* An *approximate median* of a given set of  $n$  elements is an element of the set whose rank is  $\gamma n$  for some fixed  $0 < \gamma < 1$  ( $\gamma$  is independent of  $n$ ). Describe a parallel algorithm to choose such an element that runs in  $O(1)$  time using  $n$  *CRCW* processors. Your algorithm should succeed with probability  $> 1/2$ , i.e., the output element of your algorithm should satisfy the property of an *approximate median* with this probability.

## Bibliography

- [AHU74] A. Aho, J. Hopcroft and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
- [AV79] D. Angluin and L.G. Valiant, *Fast Probabilistic Algorithms for Hamiltonian Paths and Matchings*, J. Comp. Syst. Sci., 18 (1979), pp. 155–193.
- [AKS83] M. Ajtai, J. Komlós and E. Szemerédi, *An  $O(n \log n)$  Sorting Network*, Proc. 15th ACM Symposium on Theory of Computing, 1983, pp. 1–9.
- [BH87] P. Beame and J. Hastad, *Optimal Bounds for Decision Problems on the CRCW PRAM*, 19th ACM Symposium on Theory Of Computing, 1987, pp. 83–93.
- [B89] P. Bhatt, K. Diks, T. Hagerup, V. Prasad, T. Radzik, and S. Saxena, *Improved Deterministic Parallel Integer Sorting*, Unpublished Manuscript, 1989.
- [C52] H. Chernoff, *A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the Sum of Observations*, Annals of Math. Statistics 23, 1952, pp. 493–507.
- [C86] R. Cole, *Parallel Merge Sort*, Proc. 27th IEEE Symposium on Foundations of Computer Science, 1986, pp. 511–516.

- [CD87] B. Chazelle and D. Dobkin, *Intersection of convex objects in two and three dimensions*, Journal of the ACM, Volume 34(1), 1987, pp. 1–27.
- [CG89] B. Chor and O. Goldreich, *On the power of two-point based sampling*, Journal of Complexity, Volume 5, 1989, pp. 96–106.
- [CL88] K. Clarkson, *Applications of random sampling in computational geometry II*, Proc. of the 4th Annual Symp. on Computational Geometry, 1988, pp. 1–11.
- [CV86] R. Cole and U. Vishkin, *Approximate and Exact Parallel Scheduling with Applications to List, Tree, and Graph Problems*, Proc. 27th IEEE Symposium on Foundations of Computer Science, 1986, pp. 478–491.
- [F50] W. Feller, *An Introduction to Probability Theory and Its Applications*, vol.1, Wiley, New York, 1950.
- [FR75] Floyd and Rivest, *Expected Time Bounds for Selection*, Communications of the ACM, vol. 18, no. 3, 1975, pp. 165–172.
- [H62] Hoare, *Quicksort*, Computer Journal 5, 1962, pp. 10–15.
- [H56] W. Hoeffding, *On the Distribution of the Number of Successes in Independent Trials*, Annals of Math. Stat. 27, 1956, pp. 713–721.
- [K73] D.E. Knuth, *The Art of Computer Programming, Vol.3: Sorting and Searching*, Addison-Wesley Publishing Company, Massachusetts, 1973.
- [KR88] H. Karloff and P. Raghavan, *Randomized algorithms and pseudorandom numbers*, Proc. 20th ACM Symposium on Theory of Computing, 1988, pp. 310–321.
- [L84] T. Leighton, *Tight Bounds on the Complexity of Parallel Sorting*, 16th ACM Symposium on Theory of Computing, Washington, D.C., 1984, pp. 71–80.
- [P78] F. Preparata, *New Parallel Sorting Schemes*, IEEE Transactions on Computers, vol. C27, no. 7, 1978, pp. 669–673.
- [R76] M. O. Rabin, *Probabilistic Algorithms*, in *Algorithms and Complexity, New Directions and Recent Results*, edited by J. TRAUB, Academic Press, 1976, pp. 21–36.
- [R90] S. Rajasekaran, *Randomized Parallel Selection*, 10th Annual Conference on Foundations of Software Technology and Theoretical Computer Science, 1990. Springer-Verlage Lecture Notes in Computer Science 472, pp. 215–224.
- [RR87] S. Rajasekaran, and J.H. Reif, *Derivation of Randomized Algorithms for Sorting and Selection*, Technical Report, Aiken Computing Lab., Harvard University, 1987.
- [RR89] S. Rajasekaran, and J.H. Reif, *Optimal and Sub-Logarithmic Time Randomized Parallel Sorting Algorithms*, SIAM Journal on Computing, vol. 18, no. 3, 1989, pp. 594–607.
- [RS87] S. Rajasekaran, and S. Sen, *On Parallel Integer Sorting*, Technical Report, Department of Computer Science, Duke University, 1987. To appear in ACTA INFORMATICA.

- [R84b] J.H. Reif, *On the Power of Probabilistic Choice in Synchronous Parallel Computations*, SIAM J. Computing 13(1), 1984b, pp. 46–56.
- [RS87] J. Reif and S. Sen, *Optimal randomized parallel algorithms for computational geometry*, Proc. 16th International Conference on Parallel Processing, 1987, *Revised version to appear in Algorithmica*.
- [RS89] J. Reif and S. Sen, *Polling: A new randomized sampling technique for computational geometry*, Proc. 21st ACM Symposium on Theory of Computing, 1989, pp. 394–404.
- [RV87] J.H. Reif and L.G. Valiant, *A Logarithmic Time Sort for Linear Size Networks*, Proc. 15th ACM Symposium on Theory of Computing, 1983, pp. 10-16. Also in JACM 34(1), 1987, pp. 60–76.
- [R81] R. Reischuk, *A Fast Probabilistic Sorting Algorithm*, Proc. 22nd IEEE Symposium on Foundations of Computer Science, 1981, pp.88–102.
- [S90] S. Sen, *Finding an approximate median with high probability in constant parallel time*, Information Processing Letters, 34, 1990, pp. 77–80.
- [SV81] Y. Shiloach and U. Vishkin, *Finding the Maximum, Merging, and Sorting in a Parallel Computation Model*, J. Algorithms 2, 1981, pp.212–219.
- [V75] Valiant, *Parallelism in Comparison Problems*, SIAM Journal of Computing, vol. 4, 1975, pp. 348–355.
- [W62] Wilks, *Mathematical Statistics*, John Wiley and sons, New York 1962.