Sorting Networks

B.Tech. Project Report

By

Parikshit Gopalan
Roll No : 96005025

Under the guidance of

Dr. Sundar Vishwanathan

Department of Computer Science and Engineering
Indian Institute of Technology
Mumbai
April 17, 2000
Acknowledgment

I would like to thank Prof. Sundar Vishwanathan for the opportunity to work on this problem, and for all the help that he has given.

Parikshit Gopalan
April 17, 2000
Abstract

The problem of efficient sorting in parallel, using a Sorting Network is one of the celebrated problems in parallel computing. The only known network of depth $O(\log N)$ is the AKS sorting network. However, the constants involved are so large as to make the network impractical. Batcher’s network is a simple network that works in time $O(\log^2 N)$ and is used in practice.

Our aim is to find a sorting network of small depth.
Contents

1 Introduction to Sorting Networks ................................................. 1
   1.1 Sorting Networks ..................................................... 1
   1.2 Lower Bounds on Sorting in Parallel ................................ 2

2 Problems Related to Sorting .................................................... 3
   2.1 A Useful Theorem .................................................... 3
   2.2 Maximum and Minimum .............................................. 4
   2.3 Merging Sorted Sequences .......................................... 5
   2.4 Median and Majority ................................................. 5
   2.5 Halving a Given Sequence .......................................... 6

3 Butterfly Tournaments .......................................................... 8
   3.1 Introduction .......................................................... 8
   3.2 Properties of the Butterfly Tournament .......................... 8
   3.3 An Inference Mechanism ............................................ 10
   3.4 Analysis of the Inference Mechanism .............................. 11

4 Sorting on Multi-dimensional Arrays ....................................... 15
   4.1 Multidimensional Arrays ........................................... 15
      4.1.1 Rank of an Input .............................................. 16
      4.1.2 Performance Analysis ...................................... 17
      4.1.3 Comparison with the Butterfly Network .................. 20
   4.2 Further Sorting ..................................................... 20
      4.2.1 Diagonal Sorting in Two Dimensions ....................... 20
      4.2.2 Performance Analysis ...................................... 22
      4.2.3 Generalization to Higher Dimensions ..................... 23
4.3 A Generalization of Shear Sort .......................... 24
  4.3.1 Shear Sort on 2-Dimensional Arrays ............... 24
  4.3.2 Shear Sort on d-Dimensional arrays ............... 26

5 Conclusion .............................................. 28

References .............................................. 29
## List of Figures

1.1 A Comparator .................................................. 1

2.1 Finding Max and Min for $N = 8$ .................................. 5

3.1 Butterfly Tournament of Size 8 ................................. 9

3.2 The Inference Tree for the Winner .............................. 12

3.3 Players who Lose Once ......................................... 13

3.4 Worst Case for LWW ........................................... 14

4.1 Sorting in a 2d array ........................................... 16

4.2 Before Sorting Along the Diagonal Lines ..................... 21

4.3 The Win-loss Record for $(x,y)$ ............................... 23

4.4 Sheared order on a 2-dimensional array ..................... 25
Chapter 1

Introduction to Sorting Networks

1.1 Sorting Networks

A sorting network accepts a sequence of \( N \) numbers in \( N \) registers numbered 0 to \( N - 1 \) from top to bottom as input. At the output, the it places the numbers in registers \( 0 \cdots N - 1 \) in sorted order. The sorting network performs a series of matchings of these registers. At each stage, the contents of matched pairs are compared and the lesser value goes into the lower numbered register. The sequence of matchings is predetermined, it is independent of the input given. Each such matching is called a level of the network. The number of levels is called the depth of the sorting network.

Each matched pair of registers is compared using a comparator. A comparator receives its inputs \( x \) and \( y \) from registers \( R_1 \) and \( R_2 \). The output to the upper register is \( \min(x, y) \), and to the lower register is \( \max(x, y) \).

This model can only execute algorithms where the sequence of comparisons in terms of registers is pre-determined, like Bubble-sort. It cannot execute algo-

![Comparator Diagram]

Figure 1.1: A Comparator
gorithms where the registers to be compared in some step are determined by the result of the last comparison. An example of such an algorithm is merging of two sorted sequences in Merge-sort.

1.2 Lower Bounds on Sorting in Parallel

We now present some standard results regarding sorting and lower bounds on parallel algorithms, which can be found in [2] and [3].

**Theorem 1.2.1** If the complexity of a given problem is $\Omega(f(N))$ sequentially, then the complexity is $\Omega(f(N)/p)$ using $p$ processors.

**Theorem 1.2.2** The complexity of sorting $N$ numbers on a single processor is $\Omega(N \log N)$.

**Consequence 1.2.3** The complexity of sorting $N$ numbers using $N$ processors in parallel is $\Omega(\log N)$.

Can this lower bound be achieved in practice? The AKS network, discovered in 1983 by Ajtai, Komlós and Szemerédi [1] is an $O(\log N)$ depth sorting network. But the constants involved in it are very large and so the network is not of practical use. Batcher’s network, which is described in [2] is a simple network that sorts numbers by repeated merging in depth $O(\log^2 N)$.

**Problem Statement:**
To find a sorting network of small depth.
Chapter 2

Problems Related to Sorting

The following problems are closely related to sorting.

1. Finding Maximum and Minimum of a sequence of numbers.
2. Merging two sorted sequences.
3. Finding median of a sequence.
4. Finding majority in a zero-one sequence.
5. Halving a sequence.

In this chapter, we prove lower bounds on the depth of a comparator based network for each of these problems. We also look at some simple networks that solve some of these problems.

2.1 A Useful Theorem

Theorem 2.1.1 In a comparator network, a number $N$ starting at a particular register $R$ can go to one of at most $2^k$ distinct registers after $k$ steps.

Proof: The proof is by induction on $k$.

$k = 1$. In the first step, register $R$ is compared with one other register, so that $N$ can be in any of two registers.

Assume that this hypothesis holds for $k - 1$. So $N$ can be in at most $2^{k-1}$ registers. At step $k$, each of these can be compared with one other register. Together, all these registers will be compared to at most $2^{k-1}$ other registers. If all
these registers are distinct, after \( k \) steps, \( N \) can be in any of \( 2^k \) registers. From the definition of a binomial tree as given in [2], we see that if there are \( 2^k \) distinct positions, they form a binomial tree of depth \( k \) rooted at \( R \). \( \square \)

**Lemma 2.1.2** In a comparator network, the contents of register \( R \) after \( k \) steps originated from one of at most \( 2^k \) distinct registers.

**Proof:** It follows by reversing the directions of the arrows in the binomial tree, which is equivalent to tracing back the origin of the contents of a register starting at step \( k \). \( \square \)

## 2.2 Maximum and Minimum

The complexity of sequentially finding the maximum or minimum of \( N \) numbers is \( \Omega(N) \). So the speedup lower bound on a parallel algorithm is \( \Omega(1) \). But we can prove a better lower bound for the problem in a comparator based network.

**Theorem 2.2.1** Finding the maximum and minimum in a sequence of \( N \) numbers in an \( N \) register comparator based network is \( \Omega \log(N) \).

**Proof:** Consider the problem of finding the maximum element of a sequence of size \( N \). The maximum element needs to reach the bottom-most register. But it can begin at any of \( N \) positions. So, by 2.1.1 we need at least \( k \) steps where \( 2^k = N \Rightarrow k = \log_2(N) \). Similar argument holds for finding the minimum. \( \square \)

**Algorithm:** Given an input sequence of \( N \) elements, compare adjacent elements to find the larger elements. Repeat this recursively on the set of \( N/2 \) larger elements to find the maximum element.

The algorithm for finding the minimum element is similar. As can be seen from figure 2.1, the graphs for finding both the maximum and the minimum of \( N \) number are subgraphs of the \( N \) input butterfly network.
2.3 Merging Sorted Sequences

Given two sorted sequences of size $N$ each, we wish to merge them to form a single sorted sequence of size $2N$. The complexity of the sequential algorithm for this is $\Theta(N)$, giving a speedup lower bound of $\Omega(1)$.

**Theorem 2.3.1** The depth of a comparator based network for merging two sorted sequences of size $N$ is $\Omega(\log N)$ in a comparator network.

**Proof:** Consider merging two sequences $S_1$ and $S_2$ of size $N$. The $k^{th}$ element of $S_1$ has at least $k - 1$ smaller elements and at least $N - k$ larger elements in the merged sequence, which come from $S_1$ itself. So its position in the merged sequence lies in any of $N$ positions between $k$ and $N + k$. Hence by 2.1.1, this requires at least $\log_2 N$ steps. \hfill $\square$

This lower bound is achieved by the bitonic merging network that is used as a part of Batcher’s Network, which is described in [2].

2.4 Median and Majority

**Finding the Median:** The sequential complexity of finding the median of a sequence of numbers is $\Omega(N)$. There is a simple randomized algorithm that runs
in expected time $O(\log N)$. There is a deterministic algorithm of complexity $O(N)$ time, while $O(N \log N)$ sorting algorithms like merge sort can be used to find median in $O(N \log N)$ time. In any sorting network, the median needs to reach a particular register from any of $N$ registers, which will take time of at least $\log N$. Batcher’s algorithm finds the median in $\log^2 N$ steps. Besides the AKS network, there is no known network that achieves the bound of $O(\log N)$.

**Definition 1 Majority:** Given a sequence $X = x_1 \cdots x_N$ of $N$ binary variables, let $Y = y_1 \cdots y_N$ be the sequence in sorted order.

If $N = 2k$, $\text{maj}(X) = y_{k+1}$

If $N = 2k + 1$, $\text{maj}(X) = y_k$

The problem of finding majority is equivalent to finding the median of a binary sequence. Valiant has shown, that there exist monotone formulae of depth $O(\log N)$ for finding majority [4]. A monotone formula is a formula that can be implemented by a logic circuit that consists only of two input And gates and Or gates. A sorting network can be converted into such a circuit as follows. If the inputs to a comparator are $x_1$ and $x_2$, then the Max output is replaced by $x_1$ Or $x_2$ and the Min output by $x_1$ And $x_2$.

## 2.5 Halving a Given Sequence

Given a sequence of $2N$ numbers in registers $1 \cdots 2N$, the problem of halving is to place the smaller $N$ numbers in registers $1 \cdots N$ and the larger $N$ in registers $N + 1 \cdots 2N$. No restrictions are placed on the ordering within the upper and lower $N$ registers.

Sequentially, a simple way to halve a sequence is to find the median and then compare each element with the median. Since median can be found in time $O(N)$, halving can also be done in time $O(N)$.

**Theorem 2.5.1** A comparator based network for halving a sequence has depth $\Omega(\log N)$.

**Proof:** Consider an input sequence in which the registers $1 \cdots N - 1$ contain the smallest $N - 1$ numbers in sorted order. In a comparator based network, these elements will not be moved. So, the $N^{th}$ smallest number will be put in register
$N$ by a halving circuit. The $N^{th}$ number can lie in any of the registers $N \cdots 2N$. For the contents of any of these registers to be able to enter the $N^{th}$ register, the depth of the network needs to be at least $\log N$ by theorem 2.1.2. So, the problem of halving a given sequence is $\Omega(\log N)$.

Again, for this problem, beside the AKS network, there is no network known that can halve in time $O(\log N)$. Batcher’s algorithm takes time $\Theta(\log^2 N)$ to halve a sequence.

A circuit that can halve a sequence in time $O(\log N)$ can also be used to find the median in $O(\log N)$. We find the Maximum of the lower half and the Minimum of the upper half using the $O(\log N)$ depth circuit described earlier. Such a circuit also gives the majority of a binary sequence. While it may not result in a circuit that sorts in less than $O(\log^2 N)$, it will achieve the lower bounds for the problems of finding median and majority.
Chapter 3

Butterfly Tournaments

In this chapter we analyze the performance of a sorting network called the butterfly tournament.

3.1 Introduction

A butterfly tournament is a tournament played between $N = 2^d$ players, who have distinct rankings from $1 \cdots 2^d$. Whenever two players meet, the player with the higher ranking wins. The butterfly tournament on $2^d$ players is defined as follows:

- The $2^d$ players are paired up, each pair plays a match.
- A $2^{d-1}$ butterfly tournament is played among the winners and another among the losers.

To solve the question of how to pair up players at each stage, such a tournament can be got by replacing each switch in a butterfly by a comparator. Which players meet at each stage is decided by the starting positions. Note that only playing tournaments among winners at each stage results in a knockout tournament. This is the network for finding maximum in depth $\log N$.

3.2 Properties of the Butterfly Tournament

Claim 1 Players who meet in any round have the same win loss record.
\textbf{Proof:} Consider a player whose starting position is represented by the bit-string $a_1 \cdots a_d$. In the $i^{th}$ round, if he wins, his $i^{th}$ bit is set to 0, else it is set to 1. We shall also use the representation $W$ for a win, $L$ for a loss. After $i$ rounds, if the player’s win loss record is represented by the string $R = r_1 \cdots r_i$ of length $i$, then his position is $r_1 \cdots r_i a_{i+1} \cdots a_d$. After $d$ rounds, the position of each player is determined uniquely by the win-loss record $R$. The positions of two players who meet in round $i$ in the butterfly differ only in the $i^{th}$ bit. So, they must have identical records in the previous $i - 1$ rounds. 

\textbf{Claim 2} Players who meet in any round cannot have both met some other player.

\textbf{Proof:} Let us assume that players $X$ and $Y$ who meet in round $i$ have each met player $Z$ previously in rounds $j$ and $k$, $i > j, k$. Assume without loss of generality that $j < k$. In round $j$, $X$ and $Z$ meet. They then go into two separate tournaments. Since $j, k > i$, $Y$ is yet to meet either of them. But since players can meet only if they belong to the same tournament, $Y$ can meet at most one of $X$ and $Z$. Since $Y$ meets $X$, he cannot meet $Z$. 

Note that the condition that they meet in some round is actually stricter than necessary. We can claim that if $X$ and $Y$ are in the same tournament at some round, then set of players they have met so far is disjoint.
**Claim 3** Suppose players $X$ and $Y$ meet in some round in the tournament. If $P(X)$ and $P(Y)$ are the sets of players they each play against, $P(X) \cap P(Y) = \emptyset$.

**Proof:** If $X$ and $Y$ meet in some round, the sets of opponents they have faced so far are disjoint. Also, claim 3 implies that they cannot meet a common player in the future.

In this case, the condition that they actually meet is necessary. Suppose $X$ and $Y$ are in the same tournament in round $i$. They do not meet each other. Instead $X$ loses to $A$, $Y$ defeats $B$. In the next round, it is possible that $X$ meets $B$, and even $Y$ meets $A$.

### 3.3 An Inference Mechanism

Consider player $X$’s first win which happens against player $Y_1$. We now draw a directed edge $(X, Y_1)$. If $X$ wins against a second opponent $Y_2$, then we know that $Y_2$ has the same win loss record as $X$. Hence, there exists an edge $(Y_2, Z)$. Now, adding the edge $(X, Y_2)$ gives us a directed path from $X$ to $Z$.

Formally, we define this mechanism as follows. Initially, we have an isolated vertex $X$. Each time $X$ wins against an opponent $Y$, we add an edge from $X$ to $Y$ in $G$. This creates a directed path from $X$ to every vertex reachable from $Y$. Repeat this process for each win of $X$. If $X$ has $w$ wins, this gives us a binomial tree $B_w$ rooted at $X$, by the definition of binomial trees [2]. $B_w$ has $2^w$ nodes. Since $X$ is the root, it is greater than $2^w - 1$ other nodes.

We refer to the two binomial trees rooted at $X$ after $k$ rounds as the win tree $W_k(X)$ and loss tree $L_k(X)$ of $X$. So the above operation can be described as follows. If $X$ and $Y$ meet at round $k$ and $X$ wins

$$W_{k+1}(X) = W_k(X) \cup W_k(Y)$$

$$W_{k+1}(Y) = W_k(Y)$$

$$L_{k+1}(X) = L_k(X)$$

$$L_{k+1}(Y) = L_k(X) \cup L_k(Y)$$

**Claim 4** The binomial trees rooted at $X$ and $Y$ are node disjoint.
Proof: We prove the claim by showing inductively that at each round, the win trees and loss trees are disjoint for every player in the same tournament.

The claim holds trivially after round one, since every player has met a distinct player. Suppose that the claim is true after \( k \) rounds. Consider two vertices \( X \) and \( Y \) that win in round \( k + 1 \), defeating \( A \) and \( B \) respectively.

\[
W_{k+1}(X) = W_k(X) \cup W_k(A) \\
W_{k+1}(Y) = W_k(Y) \cup W_k(B)
\]

By the inductive hypothesis, the trees \( W_k(X), W_k(Y), W_k(A), W_k(B) \) are disjoint. So \( W_{k+1}(X), W_{k+1}(Y) \) are also disjoint. Also, it follows that each win tree does not contain repetition of vertices. \( \square \)

Claim 5 If player \( P \) has \( w \) wins and \( l \) losses, he is ranked higher than \( 2^w - 1 \) others and lower than \( 2^l - 1 \) others.

Proof: \( P \)'s win tree has \( 2^w \) nodes, and is rooted at \( P \). So, we can infer that \( P \) is ranked higher than \( 2^w - 1 \) nodes. Similarly for the loss tree. \( \square \)

3.4 Analysis of the Inference Mechanism

Maximum and Minimum: The player who is ranked 1 wins all his matches. His record is \( d \) wins. So he is ranked higher than \( 2^d - 1 \) others. Similarly for the lowest ranked players.

Bounds on Wins and Losses: If a player suffers \( l \) losses, there are at least \( 2^l - 1 \) players ranked higher than him. So, if his rank is \( i \), \( 2^l - 1 \leq i - 1 \rightarrow l \leq \log_2 i \) So, upper bound on the number of losses that player with rank \( i \) can suffer is \([\log_2 i]\). Similarly, there are \( N - i \) elements ranked lower than him. So, if \( w \) is the number of wins, \( 2^w - 1 \leq N - i \rightarrow w \geq \log_2 N - i + 1 \).

Number of Edges Known: We can consider the process of sorting \( N \) numbers to be equivalent to giving a direction to the edges of \( K_N \). If \( i > j \), the edge is directed from \( i \) to \( j \). If \( w \) is the number of wins, \( l = d - w \), so the direction
is known for $2^w + 2^{d-w}$ edges. The worst case occurs when $w = d/2$. Here the number of edges known is roughly $2^{d/2}$ which is $O(\sqrt{N})$. For elements that do not have extremely good or extremely bad records, the network doesn’t perform very well.

**Performance with regard to Halving:** Another problem where the network doesn’t perform very well is with regard to the problem of halving. If the player ranked 2 meets the player ranked 1 in the very first round, then his win-loss record will a single loss followed by all wins, which will put him at node $N/2 + 1$, in the wrong half of the network.

**Non-symmetric Inference:** At the end of the tournament, the player with the highest rank knows that he is ranked higher than everyone else. But everyone else does not necessarily know that he is higher than them. Consider for example, the records of players who suffer a single loss. Only the player with the record $WWL$ knows for sure that $WWW$ is above him. The rest have either lost to him, or lost to someone, who later loses to $WWW$. Hence they do not necessarily infer that he is ranked above them.

This problem arises from the fact that if player $X$ and $Y$ meet, and $X$ defeats $Y$, $X$ only infers that he is greater than those people that $Y$ has already defeated. $Y$ may later on defeat some player $Z$. $X$ is greater than $Z$, but has no way of inferring it. In sum, a player infers from only the past records of people he meets, as and when he meets them, not from their entire records.

One interesting consequence of this is that it is not just the exact number of
Figure 3.3: Players who Lose Once
wins and losses but even the order in which they occur is of importance. This is seen from the worst case scenario for $LWW$, who may be less than 4 others. In the best case, he may have lost to $WWW$ alone. $WWL$ on the other hand can only have lost to $WWW$. In general, if a player suffers his losses later, he is more likely to have lost to a better player.
Chapter 4

Sorting on Multi-dimensional Arrays

Introduction

In this chapter, we look at the problem of sorting $N$ numbers using $d$-dimensional arrays of size $k^d$. We shall see that this generalizes the butterfly tournaments of the last chapter, while at the same time, it gives us a better inference mechanism and a method for further comparisons.

We shall often be using $k$-sorters as the basic units for sorting. A $k$-sorter accepts $k$ inputs, and outputs them in sorted order. We can use Batcher’s network to build a $k$-sorter whose depth is $\log^2 k$.

4.1 Multidimensional Arrays

Suppose the input size is $N = k^d$. We place the inputs at the vertices of a $d$-dimensional array of side $k$. In order to sort the numbers, we need to define an ordering on the vertices. Consider a vertex $r = (r_1 \cdot r_d)$. We treat this as a $d$-digit number to the base $k + 1$. We order the vertices according to these numbers. This ordering is equivalent to lexical ordering along the dimensions.

We then sort the numbers along each of the $d$ dimensions, one dimension at a time, using $k$-sorters. Consider an input placed initially at $x = (x_1 \cdot x_d)$. While sorting along dimension $i$, it gets a rank $r_i$ between 1 and $k$. After sorting along all $d$ dimensions, it will be at vertex $r = (r_1 \cdot r_d)$. At each intermediate
After row-wise sort  
After column-wise sort

Figure 4.1: Row and column sort in a 2d Array

dimension, each set of \( k \) inputs that is sorted consists of inputs that have the same ranks from the comparisons done so far.

Let us suppose that the \( k \)-sorters used are of depth \( \log^2 k \). The depth of the network described above is \( d \log^2 k = \log N \log k \) since \( \log N = d \log k \). Note that the time taken to sort these \( N \) inputs using Batcher’s network is \( \log^2 N \).

- In the trivial case when \( k = N, d = 1 \), the above network is Batcher’s network.
- If \( k = 2, d = \log_2 N \), then the array is a \( d \)-dimensional hypercube and the comparator network obtained is the butterfly network described in the last chapter. Its depth is \( \log N \).
- For any intermediate value of \( k \) between 2 and \( N \), the depth of the network is between \( \log N \) and \( \log^2 N \).

4.1.1 Rank of an Input

The problem of sorting \( N \) numbers is equivalent to determining the rank \( n \) of each input, where \( 1 \leq n \leq N \). Consider an input that reaches the vertex \( r = (r_1 \cdots r_d) \). We will denote this input by \( I(r) \). We want to know what this tells us about the overall rank \( n \) of the input.
Lemma 4.1.1 Consider \( k^2 \) numbers placed at the vertices of a 2-dimensional \( k \times k \) array. If the inputs are first sorted along row-wise and then column-wise, numbers in the resulting arrangement are in sorted order along rows and columns.

Proof: We need to prove that after sorting column-wise, the inputs remain sorted row-wise. We prove it using the zero-one lemma. After sorting row-wise, the number of 1s in each column increases as column number increases. If there is a 1 in column \( j \) at \((i, j)\), there will be a 1 in all columns from \((j + 1), \cdots, k\) at positions \((i, j + 1)\cdots(i, k)\). After sorting along columns, the 1s in each column are placed consecutively. So, we get a series of columns of increasing height. So, as in figure 4.1 after sorting by column, the inputs are in sorted order along rows as well. \( \square \)

Lemma 4.1.2 Consider \( k^d \) numbers placed at the vertices of a \( d \)-dimensional array. If the inputs are sorted along each of the \( d \)-dimensions successively, then the final arrangement is in sorted order along each of the \( d \)-dimensions.

Proof: Suppose we are sorting along dimension \( i \). To prove that inputs remain in sorted order along any dimension \( j \) in the previous \((i - 1)\) dimensions, consider the 2 dimensional array formed in the \((i, j)\) plane, and apply lemma 4.1.1. \( \square \)

We can give a physical interpretation to the sorting done so far. Suppose there is a force, say gravity, acting on the numbers in the \( d \)-dimensional array, that is directed from \((0 \cdots 0)\) to \((k \cdots k)\), so that the larger numbers go towards \((k \cdots k)\). There are several such arrangements possible, the arrangement that results above is of them. We might, for instance get a different arrangement depending on the order in which we choose dimensions to sort along.

4.1.2 Performance Analysis

We shall now use the results that were proved in the last section to analyze the performance of the network.

Theorem 4.1.3 An input \( I(r) \) of rank \( r = (r_1 \cdots r_k) \) is less than all inputs \( I(s) \) of rank \( s \) where \( \forall i, s_i \geq r_i \). It is greater than all inputs of rank \( s \) where \( \forall i, s_i \leq r_i \).
Proof: Note that each input has a unique rank. Hence, given two inputs, their rank differs in at least one place. So, \( \forall i, s_i \geq r_i \rightarrow \exists j \) such that \( s_j > r_j \). Since the inputs are sorted along every dimension, we have \( I(s_1 \cdots s_d) \geq I(r_1, s_2 \cdots s_d) \geq I(r_1 \cdots r_j, s_{j+1} \cdots s_d) \geq I(r_1 \cdots r_d) \), with strict inequality holding in at least one step where \( s_j > r_j \).

**Consequence 4.1.4** The network finds the minimum and maximum of the numbers.

Proof: \( \forall r, I(1 \cdots 1) \leq I(r) \leq I(k \cdots k) \). The input \( I(1 \cdots 1) \) is the minimum. The input \( I(k \cdots k) \) is the maximum. When \( k = 2 \), the network finds the minimum and maximum in optimal \( \log N \) time.

**Consequence 4.1.5** An input \( I(r), r = (r_1 \cdots r_k) \) is less than \( (\prod_{i=1}^{d} r_i - 1) \) other inputs, and greater than \( (\prod_{i=1}^{d} (k+1-r_i) - 1) \) other inputs.

Proof: This result follows from theorem 4.1.3. Given \( r = (r_1 \cdots r_k) \), there are exactly \( (\prod_i r_i - 1) \) inputs \( I(s) \) that satisfy the condition \( \forall i, s_i \leq r_i, s \neq r \) and \( (\prod_i (k-r_i) - 1) \) inputs \( I(s) \) that satisfy the condition \( \forall i, s_i \geq r_i, s \neq r \).

We have seen that the depth of the network is \( \log k \log N \). As \( k \) increases, the depth of the network increases. Since we are performing more comparisons, we would expect to have more information about each input. Given an input at vertex \( I(r), r = (r_1 \cdots r_d) \), we know that the input is greater than or equal to \( w_r = \prod_{i=1}^{d} r_i \) inputs. The expectation value of \( w \) is

\[
E(w) = \frac{\sum_{r \in \mathcal{M}} w_r}{k^d} = \frac{\sum_{r \in \mathcal{M}} (\prod_{i=1}^{d} r_i)}{k^d} = \frac{\sum_{r \in \mathcal{M}^{-1}} (\prod_{i=1}^{d-1} r_i) (\sum_{r_d} r_d)}{k^d} = \frac{k(k-1) \sum_{r \in \mathcal{M}^{-1}} (\prod_{i=1}^{d-1} r_i)}{2k^d}
\]

18
\[
\frac{(k - 1)^d}{2^d} 
\approx \frac{N}{2^d}
\]

In the case where the inputs are fully sorted which corresponds to \(d = 1\), \(E(w) = N/2\). The expected number of losses is also the same by symmetric considerations. We see that as \(d\) increases, the average amount of information we have about the rank of an input decreases.

Using the terminology that we did for the butterfly network, we have seen that for an input whose rank is \(r = (r_1 \cdots r_d)\), the number of wins is \(w = (\prod_{i=1}^{d} ((k-r_i))\) and the number of losses is \(l = (\prod_{i=1}^{d} r_i)\). We would like to know for which vertices of the array this scheme performs well and for which we know very little about the rank of the input. This means inputs for which \(w + l\) is minimum. In the case of a two dimensional array, we see that the worst case occurs for vertices \((1,k)\) and \((k,1)\), where \(w = l = k\). Similarly for the general \(d\)-dimensional case, consider input \(I(r)\) which is placed at a vertex \(r = (r_1 \cdots r_d)\), such that \(1 \leq j \leq d/2, r_j = k, d/2 < j \leq d, r_j = 1.\) \(w + l = r^{d/2}\). So the total amount of information that we have about input \(I(r)\) is \(w + l = 2r^{d/2}\). To see that this is indeed the worst case, observe that

\[
w \times l = \prod_{i=1}^{d} r_i \prod_{i=1}^{d} (k - r_i) \\
= \prod_{i=1}^{d} r_i (k - r_i) \\
\geq \prod_{i=1}^{d} k \\
= k^d
\]

Since \(w \times l \geq k^d, w + l \geq 2k^{d/2}\). This is achieved for any input \(I(r)\), where half the \(r_i\)’s are \(k\) and half are \(1\). The network gives less information about ranks of inputs that perform very well along some dimensions and very badly along others.
4.1.3 Comparison with the Butterfly Network

We shall now consider in depth the case of the butterfly network, where $k = 2$.

Vertices of the butterfly network are of the form $I(r)$ where $r$ is some binary string. $I(r)$ is greater than all those inputs $I(s)$ where $\forall i, s_i \geq r_i$. These vertices can be obtained by replacing some subsets of the 0’s in the bit string $r$ by 1’s. If the input has $w$ wins, there are $w0^w$ and hence $2^w$ distinct strings that can be obtained by replacing 0’s in $r$ by 1’s. Similarly, replacing the 1’s by 0’s gives us the loss tree whose size is $2^l$. In the earlier case, we saw that the player with all losses was not included in everyone’s win tree. This does not happen in this case.

In the inference mechanism described earlier for the butterfly network, we did not know exactly which inputs a given input was greater than or less than. The results above tell us exactly which these inputs are. The reason for this is that in the butterfly network, we tried to merge the win and loss trees after each comparison. However, in this section, we try to sort along successively more dimensions, while ensuring that the previous sorted order is also maintained.

4.2 Further Sorting

4.2.1 Diagonal Sorting in Two Dimensions

The sorting done so far corresponds to application of a gravity vector directed from $(1, 1)$ to $(k, k)$. There are a family of hyper-planes perpendicular to this vector. Consider the lines of the form

$$x + y = c, 2 \leq c \leq 2k$$

Each of these lines is perpendicular to the vector from $(1, 1)$ to $(k, k)$. The overall ordering on the vertices defines an ordering on vertices in each of these lines. We hereafter refer to these lines as diagonal lines, and especially to the line $x + y = k + 1$ which contains the two vertices $(k, 1)$ and $(1, k)$ as the middle diagonal. If $x + y \leq k + 1$, we refer to these lines as the lines above the middle diagonal and if $x + y \geq k + 1$, we refer to them as lines below the middle diagonal.

We number these diagonal lines from 1 to $2k - 1$. Let us denote the number of 1s in diagonal line $i$ by $d_i$ and the number of 0s by $d_i'$. Note that these numbers remain unchanged in the course of diagonal sorting.
Figure 4.2: Before Sorting Along the Diagonal Lines: $d_x > d_y, d'_y > d'_z$

The following claim states that, above the middle diagonal, the number of 1s in any diagonal line, once it is non-zero is strictly increasing. For the lines below the middle diagonal, the number of 0s is strictly decreasing till it becomes zero.

**Claim 6** For $i \leq k - 1$, if $d_i \neq 0, d_i < d_{i+1}$. For $k \leq i < 2k - 1$, if $d'_{i+1} \neq 0, d'_i > d'_{i+1}$.

**Proof:** Since the numbers are initially sorted along both rows and columns, note that every 1 has a 1 to the left of it and below it. In case the 1 is in the $k^{th}$ row there is no vertex to the left of it. If it is in the $k^{th}$ column, there is no vertex below it. Each diagonal line $d_i$ consists of one or many consecutive sequences of 0s and 1s. It is easy to see that, corresponding to each such sequence, $d_{i+1}$ will have a sequence of 1’s of length at least one greater. Hence, when $1 \leq i \leq k - 1, d_i > d_{i+1}$.

Similarly, every 0 has a 0 above and to the right. So, for every sequence of 0s in diagonal line $i + 1$, there is a corresponding sequence of 0s of greater length in diagonal line $i$. \hfill \square

These two results can be put together to prove that the height of columns is monotonically increasing.

**Claim 7** Inputs remain sorted along rows after sorting along the diagonal lines.
Proof: Note that it is sufficient to prove that the numbers of 1s in each column is a monotonically increasing sequence. This is implied by the fact that, if \( d_i \neq 0 \) and \( i \leq k - 1, d_i < d_{i+1} \) and when \( i \geq k, d_i > d_{i+1} \). More formally, when we sort along diagonal \( i \), the \( d_i \)s and the \( d_{i+1} \)s are placed consecutively. Now suppose column \( i \) has \( c_1 \) 1s. Suppose that the vertex \((i, c_1)\), occurs in diagonal line \( j \). If \( j < k - 1 \), then the number of 1s in diagonal line \( j + 1 \) is greater. Also, the number of vertices along diagonal line \( j + 1 \) is greater than the number of vertices along line \( j \) by 1. This implies that \( c_{i+1} \geq c_i \). Similarly, if \((i, c_i)\) lies on diagonal line \( j \), where \( j \geq k \), then, since the sequence \( d_i \) is strictly decreasing, the height of successive columns must be decreasing. \( \square \)

Claim 8 Inputs remain sorted along columns after sorting along the diagonal lines.

Proof: The 1s in each row are placed consecutively towards the left of each row. This follows from the fact that the columns are in increasing in height. If we can prove that the length of the series of 1s in each row is monotonically increasing, this will imply that the numbers are in sorted order along columns. Suppose that the leftmost 1 of row \( i \) is in column \( j \). Since the inputs have been sorted along diagonals, there must be 1s in column \( j - 1 \) in row \( i + 1 \), and so on. Hence, the number of ones in each row is strictly increasing. \( \square \)

Hence, we have proved that even after sorting along diagonals, the input is sorted both along row and column.

4.2.2 Performance Analysis

The numbers are now sorted along row, column and the diagonal lines. To see how many wins and losses each element has suffered, consider an input at vertex \((x, y)\). We consider three different cases depending on the value of \( x + y \).

\[
\begin{align*}
  x + y < k & \quad W = xy + y^2/2 & \quad L = (k - y)k - x^2/2 \\
  x + y = k & \quad W = xy + y^2/2 & \quad L = xy + x^2/2 \\
  x + y > k & \quad W = ky - (k - x)^2/2 & \quad L = (k - x)(k - y) + (k - y)^2/2
\end{align*}
\]
We can see from the figure that this especially improves the performance for those vertices where we knew very little before sorting diagonally. For instance, consider the vertex \((k, 1)\). While we previously were only able to infer that \(W = k, L = k\), now we know that \(W = k^2/2, L = k\). Similarly, for vertex \((1, k)\), while previously \(W = k, L = k\), now \(W = k, L = k^2/2\).

Once we have sorted along diagonals, we are differentiating between various dimensions. Now the record for \((k, 1)\) is not the same as that for \((1, k)\). If for instance, we wanted the numbers sorted in column major order rather than row major order, then we could have sorted along the diagonals, but in the opposite direction.

We now analyze the performance of the network for the problem of halving. Note that, for inputs where \(W > k^2/2\), or \(L > k^2/2\), we know which half they lie in. Also, for every input for which \(W > k^2/2\), there must be an input for which \(L > k^2/2\). Hence, we can reduce the halving problem to a smaller subproblem.

### 4.2.3 Generalization to Higher Dimensions

We have seen that diagonal sorting greatly improves the performance in a 2-dimensional array. We would like to extend this to higher dimensions if possible. We present a brief review of how this could be done and what the problems
encountered might be.

As mentioned earlier, the sorting done so far corresponds to application of a gravity vector directed from $(1 \cdots 1)$ to $(k \cdots k)$. There are a family of hyperplanes perpendicular to this vector. The equation of each such hyper-plane is of the form

$$\sum i = i^d r_i = c, \; d \leq c \leq dk$$

The overall ordering on the vertices defines an ordering on vertices in each of these hyper-planes. No vertex in such a hyper-plane knows about its rank in relation to any other vertex in that hyper-plane. Also, they have win loss records which are similar. For example, in the case of the butterfly network, we will be comparing all inputs that have the same number of wins and losses.

However, for $N = k^d$, we will get $d - 1$ dimensional hyper-planes. So, we may not be able to sort entirely along these hyper-planes, in time less than $\log^2 N$. So, we might have to solve the problem recursively in $d - 1$ dimensions. We do have an ordering of the vertices in each of these hyper-planes. But, each of these hyper-planes may not contain vertices in the form of a square array. Consider for instance the case when $d = 3$. There is just one vertex $(1,1,1)$ where $x_1 + x_2 + x_3 = 3$. For $3 < c \leq k + 2$, the vertices $x_1 + x_2 + x_3 = c$ form a triangle, for $2k + 1 > c > k + 2$, they form a square, $3k > c \geq 2k + 1$, they again form a triangle, and $(k,k,k)$ is the unique vertex for which $x_1 + x_2 + x_3 = 3k$.

### 4.3 A Generalization of Shear Sort

In this section we present Shear Sort [3], an algorithm for sorting using a 2-dimensional array, and then generalize it to arbitrary dimensions.

#### 4.3.1 Shear Sort on 2-Dimensional Arrays

Shear Sort works on a $\sqrt{N} \times \sqrt{N}$ array. First we define a sheared order on the vertices of the array. The sheared order is as shown in figure 4.4. The algorithm is as follows.

For $i = 1$ to $\log \sqrt{N}$ {
    Sort along rows in sheared order.
    Sort along columns.
}
Figure 4.4: Sheared order on a 2-dimensional array

Sort along rows in sheared order.

Claim 9 Shear Sort sorts $N$ numbers in a $\sqrt{N} \times \sqrt{N}$ array in time $\sqrt{N} \log N$.

Proof: We prove the claim using the zero-one lemma. Suppose that the numbers have been sorted along rows in sheared order. One way to proceed with sorting along columns is to merge successive rows. We see that two dirty rows will combine to give at least one clean row. Rows that are already clean will stay clean. So the number of dirty rows reduces by half, for each iteration of the for loop. Since initially there are at most $\sqrt{N}$ dirty rows, after $\log \sqrt{N}$ repetitions, there is a most one single dirty row, which is sorted by the last row sort. The numbers are sorted along the sheared order. The time taken to sort $N$ numbers is

$$T(N) = \log(\sqrt{N}) T'(\sqrt{N})$$

(4.1)

where $T'(\sqrt{N})$ is the time taken to sort $\sqrt{N}$ numbers along a $\sqrt{N}$ length linear array. So, substituting in equation 4.1

$$T'(\sqrt{N}) = \sqrt{N} \rightarrow T(N) = O(\sqrt{N} \log N).$$

Now, returning to the problem of sorting using m-sorters, we see that we can use shear sort to give us a sorting network using m-sorters to sort along the rows and columns. Here $m = \sqrt{N}$. Substituting in equation 4.1

$$T'(\sqrt{N}) = \log^2(\sqrt{N}) = \log^2 N/4 \rightarrow T(N) = O(\log^3 N)$$

25
4.3.2 Shear Sort on d-Dimensional arrays

We now look at a generalization of shear sort to a d-dimensional array. Assume array size to be \(k^d\). We first define a sheared order for this array and then show that a similar method can be used to sort numbers.

**Sheared order for a d-dimensional array of size \(k^d\):** For \(d = 1\), the sheared order is a linear ordering of numbers from 1 to \(k\). Given a sheared order for \(k^{d-1}\), we use it to define a sheared order for \(k^d\) inductively. The reverse sheared order reverses the order given to the vertices by the sheared order. If an vertex is ranked \(r\) in the \(k^{d-1}\) array by the sheared order, it is ranked \(k^d + 1 - r\) in the reverse sheared order. Let \(B_j\) denote the vertices of \(B_j\) in sheared order and \(B_j^r\) denote the vertices in reverse sheared order. An array \(A\) of size \(k^d\) is composed of \(k\) smaller arrays \(B_1 \cdots B_k\), each of size \(k^{d-1}\). To get a sheared order on \(A\), we order vertices in \(B_j\) in sheared order if \(j\) is odd and in reverse sheared order if \(j\) is even. The overall sheared order on \(A\) is given by \(B_1 B_2^r B_3 B_4^r \cdots\).

**The Algorithm:** To sort array \(A\), we sort each sub-array \(B_j\) according to the order given to it by the sheared order on \(A\). We then sort along dimension \(d\). We repeat the above two steps \(\log k\) times. Finally, we again sort along each of the sub-arrays.

**Proof of correctness:** To prove that the algorithm works, consider two sub-arrays \(B_j, B_{j+1}\). Assume that \(B_j\) is in sheared order, \(B_{j+1}\) is in reverse sheared order. If they are both dirty, then merging them results in at least one clean sub-array. So, at each step, the number of dirty sub-arrays of size \(k^{d-1}\) is halved. Initially, there can be at most \(k\) of them. So, in \(\log k\) steps, only one dirty sub-array remains, which is sorted in the last step.

**Time Analysis:** Let \(T(k^d)\) be the time taken to sort array \(A\). \(T(k) = k\).

\[
T(k^d) = (T(k^{d-1}) + T(k)) \log k
\]

\[
T(k^d) = T(k) \log^{d-1} k = k \log^{d-1} k.
\]

If we sort each of the \(k\) sequences using Batcher’s Algorithm, this gives a sorting
network of depth $\log^{d+1} k$.

Note that there is a simple algorithm called Quadrant sort which works in time $O(\sqrt{N})$ on an $\sqrt{N} \times \sqrt{N}$ array. However, quadrant sort seems significantly more difficult to generalize to higher dimensions.
Chapter 5

Conclusion

The AKS network is of not of much practical use because of the huge constants involved. We have attempted to find a simple network of depth $o(\log^2 N)$. We have investigated the properties of butterfly networks and multidimensional arrays, two of the most commonly used simple architectures in parallel algorithms.
Bibliography


