DISTRIBUTED ALGORITHMS FOR MULTI-CHANNEL BROADCAST NETWORKS

John Michael Marberg

December 1986
CSD-860034

UNIVERSITY OF CALIFORNIA

Los Angeles

Distributed Algorithms for Multi-Channel Broadcast Networks

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Computer Science

by

John Michael Marberg

1986

The dissertation of John Michael Marberg is approved.

S a Greibach

Sheila A. Greibach

[illegible]

Leonard Kleinrock

[illegible]

Bruce L. Rothschild

[illegible]

Izhak Rubin

[illegible]

Eli Gafni, Committee Chair

University of California, Los Angeles

1986

In memory of my father

Dr. Kurt Marberg

TABLE OF CONTENTS

page

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Eli Gafni, may advisor, for his guidance and collaboration during the development of this dissertation. His enthusiasm for creative thinking, and attention and patience during long periods of hard work have been a continual source of motivation and inspiration.

I am also indebted to the other members of my doctoral committee, Sheila Greibach, Leonard Kleinrock, Bruce Rothschild and Izhak Rubin, for their support. In particular, I wish to thank Sheila Greibach, whose weekly seminar provided a stage for the presentation of new ideas and a setting for many heated discussions.

Funding for this research was provided in part by the National Science Foundation under grant DCR-84-51396 and by IBM under grant D840622.

My friends and colleagues Yehuda Afek, Valmir Barbosa, Joseph Betser, Rina Dechter, Hanoch Levy and Uri Zernik have shard with be the joys and frustrations of student life. Special mention is due to Rina Dechter, who in 1984 asked me to review one of her papers, thereby introducing me to the subject of my dissertation.

My family in Israel has been an important source of encouragement during my years away from home. In particular, I wish to convey my deepest appreciation to my mother, Hilde Marberg, whose confidence and generosity have always helped me realize my goals.

Last but not least, Debbie Smith provided me with invaluable companionship, loyalty and love without which it would have been impossible to go through this experience.

VITA

May 10, 1953	Born, Tel Aviv, Israel
1979	B.Sc. in Computer Science, Technion - Israel Institute of Technology
1979-1982	Teaching Associate, University of California, Los Angeles
1981	M.S. in Computer Science, University of California, Los Angeles
1981-1984	Research Assistant, IBM Los Angeles Scientific Center
1984-1986	Postgraduate Research Engineer, University of California, Los Angeles

PUBLICATIONS

R.C. Summers, C. Wood, J.M. Marberg, M. Ebrahimi, K.J. Perry and U. Zernik, "RM: A Resource Sharing System for Personal Computers," in *Proceedings 1983 ACM Conference on Personal and Small Computers.*

J.M. Marberg and E.M. Gafni, "An $O(N^3)$ Distributed Max-Flow Algorithm," in *Proceedings 1984 Conference on Information Sciences and Systems.*

R.C. Summers, M. Ebrahimi, J.M. Marberg and U. Zernik, "Design and Implementation of a Resource Sharing System as an Extension to a Personal Computer Operating System," in *proceedings 1985 ACM SIGSMALL Symposium on Small Systems.*

J.M. Marberg and E. Gafni, "Sorting and Selection in Multi-Channel Broadcast Networks," in *Proceedings 1985 International Conference on Parallel Processing.*

J.M. Marberg and E. Gafni, "An Optimal Shout-Echo Algorithm for Selection in Distributed Sets," in *Proceeding 23rd Annual Allerton Conference on Communication, Control and Computing,* 1985.

E. Gafni and J.M. Marberg, "Distributed Sorting Algorithms for Multi-Channel Broadcast Networks," Technical Report CSD-860055, Computer Science Department, University of California, Los Angeles, February 1986.

J.M. Marberg and E. Gafni, "Sorting in Constant Number of Row and Column Phases on a Mesh," in *Proceeding 24th Annual Allerton Conference on Communication, Control and Computing,* 1986.

ABSTRACT OF THE DISSERTATION

Distributed Algorithms for
Multi-Channel Broadcast Networks

by

John Michael Marberg
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1986
Professor Eli Gafni, Chair

This dissertation studies the use of broadcast communication in the design of distributed algorithms. Broadcast communication networks provide an attractive alternative to point-to-point networks, in that direct connection between any two or more processors is facilitated without the need for a large number of links. Moreover, the topology is modular, and multiple parallel channels, if available, exhibit redundancy and hence increased reliability.

Central to our study is the definition of a new computation model, called *Multi-Channel Broadcast network (MCB)*. It consists of p independent processors which communicate over a set of k broadcast channels, $k \leq p$. Computation proceeds in synchronous cycles, during each of which the processors first write and read the channels, then perform local computation. The underlying assumption is that at most one processor attempts to write on each channel in any given cycle (collision-free access). Performance is measured in terms of the number of cycles required by the computation. Several versions of the model are characterized, differing in aspects such as the number of channels a processor can access, the rate of transmission, and the dependence of the communication schedule on the data. A comparative analysis of the computation power of the different versions is given.

We use the MCB model as a framework for the investigation of three paradigmatic problems: sorting, selection and permutation. For each problem, we design efficient broadcast algorithms and establish tight bounds on the complexity. The main results are: an algorithm to sort n elements in $O(\frac{n}{k})$ cycles; an algorithm to sort p bit strings of length m in $O(m+p\log p)$ cycles, assuming the transmission of each bit requires a separate cycle; an algorithm to select by rank among n elements in $O(\frac{p}{k}\log\frac{nk}{p})$ cycles; and an algorithm to permute p elements in $O(\log p)$ cycles. Matching lower bounds are shown in each case. Several communication protocols that are developed as part of the solutions prove to be powerful design tools in the MCB model, illustrating its viability.

CHAPTER 1
INTRODUCTION

Rapid advances in microelectronics and communications technology have made the development of large multi-processor systems feasible. As a result, recent years have witnessed an increasing interest in all aspects of distributed and parallel computing. The design of distributed algorithms using different interprocess communication schemes and the study of the complexity of such algorithms play an important role in the effort to understand the capabilities of multi-processor systems.

In this work we focus on the use of broadcast communication in the design of distributed algorithms. Broadcast communication networks provide an attractive alternative to point-to-point networks, in that direct connection between any two or more processors is facilitated without the need for a large number of links. Moreover, the topology is modular, and multiple channels, if available, exhibit redundancy and hence increased reliability.

Our study defines a new computation model called *Multi-Channel Broadcast Network (MCB)*. The model is used as a framework for the characterization of distributed computations that use broadcast communication. Efficient algorithms and tight lower bounds for several problems are developed.

In this Chapter we discuss the motivation for our work, give an overview of our results, and survey related research.

1.1. Background and Motivation

Communication among processors in a multi-processor system can be established in several different ways. Three common approaches are: point-to-point communication; shared memory; and broadcast communication.

In the *point-to-point* approach, processors communicate by sending messages over a set of communication links, each of which connects two processors. The set of links induces a graph structure. Processors that are not directly connected by a link communicate via a path in the graph that connects them. A system of this type is called a *point-to-point network*. Numerous computation models based on this approach have been studied. The models vary mainly in the topology of the communication graph. For example: complete network [Afek85], mesh [Pete84], ring [Dole82], etc. The ARPANET [McQu77] and the Cosmic Cube [Seit85] are examples of existing networks with point-to-point communication.

In the *shared memory* approach, processors communicate by accessing a central shared memory facility. In contrast to point-to-point communication, there is no direct transfer of information between processors. Instead, a processor makes local information public by writing it into the shared memory, which can then be read by any processor. A system based on this approach is called *Parallel Random-Access Machine (PRAM)* [Wyll79]. Three different PRAM models can be distinguished by varying the assumptions on concurrent access to the shared memory: Exclusive-Read/Exclusive-Write (EREW); Concurrent-Read/Exclusive-Write (CREW); and Concurrent-read/Concurrent-Write (CRCW) (see [Snir85]). The NYU Ultracomputer [Gott83] is an example of an existing architecture based on the shared memory approach.

In the *broadcasting* approach, communication is established via a set of shared channels (called broadcast channels) that are accessible to all processors. Information is transferred by writing and reading the channels.

Broadcast communication can be viewed as an intermediate approach, between shared memory on the one hand and point-to-point communication on the other [Reis86]. Transmitting data over a broadcast channel makes it public, which resembles writing into shared memory. Unlike memory, however, broadcast channels are "memoryless," i.e., they do not retain data indefinitely. Only those processors that listen to the channel during the time of transmission receive the data. In this respect, broadcasting has the same effect as sending messages over point-to-point links to the processors that listen to the channel.

We call a system that uses broadcasting as the means of communication a *broadcast network*. The Ethernet [Metc76] is an example of an existing broadcast network with one channel.

Local area network architectures that use multiple broadcast channels have recently been proposed [Chou83, Marh85, Mars82a, Mars82b] as an alternative to single-channel Ethernet-like networks [Metc76]. In environments where messages are generated in real time, splitting a single channel into multiple channels of narrower bandwidth results in reduction of channel contention among processors at the expense of longer transmission time. It has been shown in [Mars83] that for high communication rates the reduced contention dominates the increased transmission time, and the overall message delay is decreased.

Broadcast communication is an attractive alternative to point-to-point communication. In the latter approach, two processors are able to communicate with each other directly only if they are connected by a link. If the communication graph is sparse, messages between two processors may have to traverse a path of multiple links. Broadcast channels, in contrast, provide for direct connection between any two or more processors. Moreover, sufficient connectivity can be achieved with relatively few channels compared to point-to-point networks.

Broadcast networks have several other attractive qualities. The interconnection structure is modular, which provides for gradual system growth dependent on user needs. Multiple parallel channels, if available, provides redundancy, and hence grater reliability and fault tolerance in case of communication failure. Also, failure of an individual processor does not disrupt the communication among other processors.

In view of these advantages, broadcasting seems a viable method of communication for distributed computation. It thus becomes of interest to characterize broadcast computation in the framework of a formal model. In this work, we provide such a characterization.

Central to our research is the definition of a new network model based on multiple broadcast channels. The model serves as a vehicle for the design of distributed algorithms using broadcasting and the analysis of their complexity. We apply the model in the study of three paradigmatic problems: sorting, selection and permutation. For each problem, we develop efficient broadcast algorithms and establish tight bound on the complexity. Several communication protocols that are developed as part of the solutions prove to be powerful design tools in the MCB model, illustrating its viability.

1.2. Overview of the Dissertation

Chapter 2 defines the computation model, called Multi-Channel Broadcast network (MCB). The model consists of a collection of independent processors, communicating over multiple parallel broadcast channels. A configuration with p processors and k channels is denoted MCB(p, k). We assume $k \leq p$. Computation proceeds in synchronous cycles. During each cycle, a processor writes one channel and reads one other channel, then performs local computation. The underlying assumption is that at most one processor attempts to write on each channel in any given cycle (collision-free access). We distinguish two variants of the model: the general MCB — where each processor is capable of writing and reading any of the channels; and the restricted MCB — where each processor is restricted to write only on one specific channel. Performance is measured in terms of the number of cycles and the number of messages used in the computation. Two different scales of cost are used: uniform — where the transmission of each atomic datum is assumed to take a single cycle; and logarithmic — where each bit to be transmitted requires a separate cycle.

Chapter 2 also investigates the complexity of computations in the MCB model. Among others, a comparative analysis of the computational power of the general and the restricted versions of the model is performed, and lower bounds for various classes of problems are proved. Also, the MCB model is related to the CREW shared memory model.

Chapters 3 and 4 discuss the problem of sorting a collection of elements distributed in an MCB(p, k). Chapter 3 assumes uniform cost, whereas Chapter 4 uses logarithmic cost. A different sorting approach is presented in each case.

The sorting algorithms in Chapter 3 are based on a method called *COLUMNSORT* [Leig85]. In this method, the input is organized in a matrix, and sorting is accomplished by iteratively sorting each column separately, then performing a transformation on the matrix. *COLUMNSORT* is practical for our purposes because each column can be sorted locally at a different processor, and the matrix transformations can be implemented efficiently using the broadcast channels. Let $n \geq k^3$ elements be distributed in the network, with at most $n_{\max}$ elements in any given processor. We develop a sorting algorithm that runs in $O(n)$ messages[1] and $O(\max\{\frac{n}{k}, n_{\max}\})$ cycles on the restricted version of the model. By showing matching lower bounds, we prove that the algorithm is optimal. We also develop a recursive version of *COLUMNSORT*, in which each column is sorted by recursive application of the basic algorithm. This version admits a wider range of inputs, namely $n \geq k^{1+\varepsilon}$, where $0 < \varepsilon \ll 1$ is a constant.

Chapter 4 focuses on sorting algorithms with logarithmic communication cost. Our approach is the following. The input elements are considered bit strings of uniform length m. In order to avoid repeated transmission of long elements

[1] Given two functions $g(n)$ and $f(n)$, where $n>0$, the notation "$g(n)$ is $O(f(n))$" (also written $g(n)=O(f(n))$) is interpreted as follows: there exist a constant $c>0$ and some n_0 such that $g(n) \leq cf(n)$ for all $n \geq n_0$. Similarly, "$g(n)$ is $\Omega(f(n))$" means that there exist a constant $c>0$ and some n_0 such that $g(n) \geq cf(n)$ for all $n \geq n_0$. Finally, we use "$g(n)$ is $\Theta(f(n))$" if $g(n)$ is both $O(f(n))$ and $\Omega(f(n))$. In other words, $O(\cdot)$, $\Omega(\cdot)$ and $\Theta(\cdot)$ denote, respectively, an upper bound, lower bound, and simultaneous upper and lower bound on $g(n)$ in terms of order of magnitude. The formulation can be generalized to functions with more than one parameter. For details, see [Aho83].

throughout the algorithm, each element is encoded into a shorter representation called *signature*, such that the signatures have the same relative order as the original elements. Sorting then proceeds efficiently using the signatures. Finally, the elements are rearranged according to the order of the signatures. We present a sequence of three algorithms based on this approach, each improving upon the previous one. The most efficient algorithm runs in $O(m+p\log p)$ cycles on a restricted MCB(p, p), where each processor contains one input element. By showing a lower bound of $\Omega(m)$ cycles, we prove that our approach is optimal for sufficiently large m. We also discuss generalizations for network configurations with fewer than p channels and more than one element in each processor.

Chapter 5 considers the problem of selecting the d'th largest among n elements, for some given rank d. Our approach is to iteratively reduce the number of elements that are candidates for selection by applying a filtering mechanism. We develop an algorithm that runs in $O(\frac{p}{k}\log\frac{kn}{p})$ cycles and $O(p\log\frac{kn}{p})$ messages on a restricted MCB(p, k). We also establish matching lower bounds for a wide range of cases, thereby proving that the algorithm is optimal.

Chapter 6 is concerned with the permutation problem: each processor is to deliver a message to a distinct destination processor according to a given permutation of the processors. Although the problem is trivial in the general MCB, there is an inherent difficulty in the restricted MCB, namely that a message cannot be sent to an arbitrary destination processor in one step unless the latter knows the identity of the sender. Our goal is to develop a permutation algorithm that is simpler than the obvious solution that resorts to sorting. The approach is the following. A permutation induces one or more "rings" in the network, such that a processor is followed by its destination processor in some ring. The permutation problem reduces to identifying for each processor its predecessor in the ring. We develop an efficient mechanism to traverse the ring from a processor all the way around to its predecessor. The algorithm runs in $O(\log p)$ cycles and $O(p \log p)$ messages on a restricted MCB(p, p) with uniform communication cost. A lower bound is also presented.

Chapter 7 suggests directions for future research in the area of distributed algorithms with broadcast communication. Specifically, alternative network models for broadcast communication are considered, and additional application domains are discussed.

1.3. Related Research

In this section we review related work of other researchers in the area of broadcast network models and their applications.

Landau, Yung and Galil [Land85] consider a model that consists of a fully-connected synchronous point-to-point network, where each processor is capable of broadcasting one bit on all incident links and reading one incoming bit during each cycle. This is equivalent to a restricted MCB(p, p) with logarithmic communication cost.

The Landau-Yung-Galil model is applied in solving the *multiple identification* problem. In this problem, each of p processors contains a string of m bits, and has to identify all the processors which have the same string as itself. The approach in [Land85] is to sort the strings, then use the sorted order to form groups of processors with equal strings. Sorting is performed by emulating the AKS sorting network [Ajta83], which takes $O(m \log p)$ cycles. When m is sufficiently large, this is the dominating factor in the complexity of the solution. By replacing the AKS emulation with the sorting method developed in Chapter 4 of this dissertation, we are able to improve the complexity of the multiple identification problem by a factor of $\log p$.

Dechter and Kleinrock [Dech86] investigate a broadcast model called *IPABM (Ideal Parallel Broadcast Model)*. This model differs from the MCB in two aspects. First, it has only one broadcast channel, and second, it uses concurrent-write access to the channel, assuming a global, cost-free ("ideal") collision resolution mechanism. In the MCB model, in contrast, exclusive-write access is used, thus avoiding altogether the issue of collision resolution.

The IPABM is applied in the design of algorithms for extrema finding, merging and sorting. In our model, these problems are solved without the use of concurrent-write access (Chapter 3). In terms of communication complexity, our solutions are comparable to those in [Dech86]. It should be noted, however, that the IPABM algorithms are designed to optimize local processing costs (in terms of the number of comparisons) as well as communication costs, whereas we consider only communication costs.

Levitan [Levi82] discusses a model called *BPM (Broadcast Protocol Multiprocessor)*, which has essentially the same properties as the IPABM. Algorithms for extrema finding and sorting similar to those in [Dech86] are given, as well as an algorithm for finding minimum spanning trees in graphs.

Santoro and Sidney [Sant82] consider a broadcast model called *Shout-Echo*, in which a communication cycle consists of a broadcast message from a single processor ("shout") followed by replies from all other processors ("echo"). In the MCB model, in contrast, each transmission is a broadcast message. Moreover, multiple disjoint broadcasts may proceed simultaneously in the same cycle, using separate channels.

The Shout-Echo model is used in the design of algorithms for selection [Rote83, Sant83a, Sant83b]. By using the selection method developed in Chapter 5 of this dissertation, we are able to improve the upper bound for selection in the Shout-Echo model by a factor of $\log p$ [Marb85a].

Several researchers use a hybrid approach, in which a point-to-point communication network is augmented by a broadcast bus. This structure exploits the advantages of both communication modes. Stout [Stou83] and Bokhari [Bokh84] apply the concept to mesh-connected networks, demonstrating considerable speedup for problems such as median and extrema finding, and semigroup computations. Andreatos [Andr85] extends these results to other strongly regular networks, such as triangular and hexagonal arrays. Kumar and Raghavendra [Kuma85], and Lin and Moldovan [Lin86] consider a mesh with multiple busses — one for each row or column.

The work of Chandra, Furst and Lipton [Chan83] characterizes in abstract fashion the class of distributed protocols (also called *Multi-Party Protocols*) that solve 0-1 predicates over a set of values distributed among the processors. The work examines the inherent communication complexity of such protocols in terms of the amount of information that needs to be known globally in the system. The model of communication being used is similar to a restricted MCB(p, 1) where processors broadcast one bit at a time in round-robin fashion. Tight upper and lower bounds are given for some specific predicates, however the results are mainly of theoretical value, since they depend on Ramsey-like counting arguments.

CHAPTER 2

THE MULTI-CHANNEL BROADCAST NETWORK MODEL

In this chapter we introduce a network model for distributed computation using broadcast communication. We first define the model itself, then give a comparative analysis of different classes of computations in the model.

2.1. Description of the Model

2.1.1. General Organization

The *Multi-Channel Broadcast (MCB)* network model consists of a collection of independent processors, which communicate by sending broadcast messages over a set of parallel broadcast channels. The topology is illustrated in Figure 2.1. A configuration with p processors and k channels is denoted MCB(p, k). It is assumed that $k \leq p$ and k divides p. Each processor and each channel have a unique identifier known to all processors. We denote the processors as $P_1, P_2, \ldots, P_p$, and the channels as $C_1, C_2, \ldots, C_k$.

Computation proceeds in synchronous *cycles*. We assume the existence of a global mechanism to synchronize the beginning of each cycle. A cycle consists of the following two phases at each processor.

1. COMMUNICATION: Write one channel and read one other channel.
2. PROCESSING: Perform local computation.

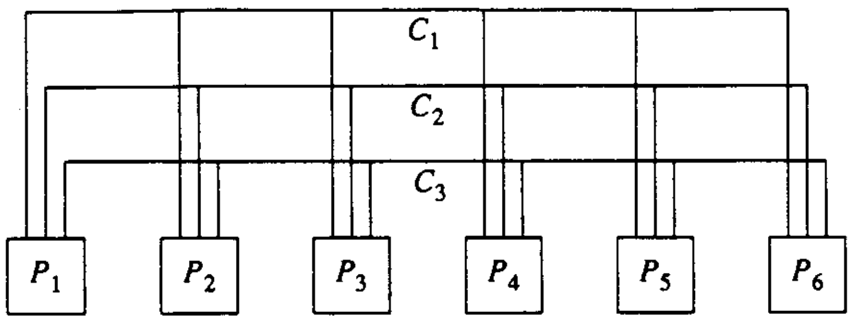


Figure 2.1. The MCB Network

The information written on a channel during a given cycle constitutes a *message* sent by the processor writing the channel. The message is received only by the processors reading the channel in that cycle. Processors reading a channel can detect that the channel is empty, i.e., that no processor has written on the channel during that cycle.

The capability of a processor to read and write two different channels simultaneously in the same cycle is assumed for convenience in algorithm design. It can be shown that limiting each processor to access a single channel per cycle does not decrease the power of the model.

The topology of the MCB generalizes several other broadcast network models. The configuration MCB(p, 1) resembles the IPABM model of Dechter and Kleinrock [Dech86] and Levitan's BPM [Levi82]. At the other extreme, the MCB(p, p) is similar to the model of Landau, Yung and Galil [Land85]. However, we use different assumptions on channel access than in those models (see Section 2.1.2).

As discussed in Chapter 1, broadcast channels and shared memory are related, in that both provide "public access." Yet, in the MCB model the number of broadcast channels does not exceed the number of processors, whereas in shared memory models such as the PRAM [Wyll79] there is usually an unbounded amount of shared memory. This difference reflects the fact that channels are "memoryless," i.e., they do not retain data from cycle to cycle. As such, they do not serve as a storage medium, but rather as a communication medium.

2.1.2. Channel Access Considerations

We distinguish two different versions of the MCB model by varying the channel access capabilities of the processors.

1. *General MCB* — each processor is capable of writing and reading any channel.
2. *Restricted MCB* — each processor is allowed to write only on one specific channel, but can read any channel.

In the restricted MCB, the allocation of processors to channels is fixed and known to all processors; it is henceforth assumed that processor P_i writes on channel $C_{\lceil \frac{ik}{p} \rceil}$.

One of the advantage of the restricted MCB is that fewer transmitters are connected to each channel. Particularly, when $p=k$, each processor has a "dedicated" channel on which only it can write. For many applications, the restricted model is sufficient. On the other hand, for some problems, the ability to write on any channel can help improve the performance. In particular, it can be shown that the general MCB is strictly more powerful than the restricted MCB (Section 2.2.1).

Among the major considerations involved in the design of broadcast networks is concurrent writing on the channels. If more than one processor attempts to write on the same channel in the same cycle, a collision occurs. To avoid the issue of collisions and collision resolution [Gree82, Will84], the underlying assumption in the MCB model is that computations are *collision-free*. In other words, it is assumed that at most one processor attempts to write on each channel during each cycle. Concurrent reading of the same channel by more than one processor is permitted (this reflects the notion of broadcasting). Only algorithms which adhere to these requirements are considered valid in the MCB model. This is similar to the approach of the Concurrent-Read/Exclusive-Write (CREW) shared memory model [Snir85].

2.1.3. Performance Measures

In analyzing the complexity of algorithms in the MCB model, we consider only the costs of communication, assuming that local processing costs are negligible by comparison. This is similar to the assumption in point-to-point networks. The performance measures of interest are the number of *cycles* and the number of *messages* that are used in the computation. All the complexity bounds discussed in this research are worst-case bounds.

We use two different scales of cost with respect to the amount of information that can be transmitted in each cycle.

1. *Uniform cost* — the transmission of an atomic datum of the computation is assumed to take one cycle.
2. *Logarithmic cost* — the transmission of each bit requires a separate cycle.

2.2. Bounds on MCB Computations

In this section we investigate the computational power of the MCB model. The analysis shows that the general MCB is strictly more powerful than the restricted MCB. Also, a class of computations called *oblivious* is defined and characterized. Finally, the relation between MCB and CREW is put in perspective.

2.2.1. Separating between General and Restricted MCB

It is obvious that the general MCB has at least the same computation power as the restricted MCB. We now show that there is a gap (or separation) in power between the two models, namely the general MCB is strictly more powerful.

Let I be an instance of a problem to be solved on the MCB. We denote the input of processor P_i as $I(P_i)$, and the corresponding output of that processor as $O_I(P_i)$.

A problem is *r-sensitive* if for every subset of processors S, $|S|=r-1$, there exist two instances I_1 and I_2 such that $I_1(P_i)=I_2(P_i)$ for all $P_i \in S$, and for some $P_j \in S$, $O_{I_1}(P_j) \neq O_{I_2}(P_j)$. Intuitively, in order to determine its output, P_j must have some "knowledge" about the input of at least r of the processors, regardless of the method of solution.

As an example, consider the following problem, called the *identification Problem*. The input of each processor is a single bit; all bits are 0, except for one processor whose bit is 1. The task is to identify the processor whose bit is 1.

We now show that the identification problem is $(p-1)$-sensitive. Let S be a subset containing $p-2$ processors, and let P_{i_1} and P_{i_2} be the two processors not in S.

Define I_1 as the instance where the bit of P_{i_1} is 1, and define I_2 as the instance where the bit of P_{i_2} is 1. Clearly, the output of any processor in S is different with I_1 than it is with I_2. This satisfies the definition of $(p-1)$-sensitivity. Notice that the problem is not p-sensitive, since knowing $p-1$ bits suffices to determine the output.

Theorem 2.1. Solving an r-sensitive problem on a restricted MCB (p, k) requires at least $\dfrac{\log r}{\log(\frac{p}{k}+1)}$ cycles.[1]

Corollary 2.1. Solving the identification problem on a restricted MCB(p, k) requires at least $\dfrac{\log(p-1)}{\log(\frac{p}{k}+1)}$ cycles. ■

Proof of Theorem 2.1. Let us number the cycles of the computation sequentially. The set of processors that *affect* processor P_i in cycle t, denoted $A_i(t)$, is defined recursively as follows.

1. $A_i(0) = P_i$.
2. $A_i(t+1) = A_i(t) \cup (\bigcup_{\lceil \frac{lk}{p} \rceil = j} A_l(t))$, where C_j is the channel being read by P_i in cycle $t+1$.

Intuitively, $A_i(t)$ consists of all the processors whose local "knowledge" (or parts thereof) could possibly have been conveyed to P_i by the end of cycle t. Notice that all the processors that have write-access to the channel being read by P_i are added to $A_i(t)$, regardless of which processor actually writes. This reflects the idea that implicit information can be gained about a given processor just from the fact

[1] Throughout this work, we use "log" to denote logarithm of base 2.

that it chose not to write the channel in a given cycle. We will see later that this information can actually be put to use.

Let t^* denote the last cycle of the computation. It follows from the recursive formulation that $|A_i(t^*)| \leq (\frac{p}{k}+1)^{t^*}$. On the other hand, by definition of r-sensitivity, there exists at least one P_i such that $|A_i(t^*)| \geq r$. It follows that $t^* \geq \frac{\log r}{\log(\frac{p}{k}+1)}$. ■

It is easy to see that in the general MCB the identification problem can be solved in one cycle. The processor whose bit is 1 writes its id on channel C_1, and all other processors read that channel. On the other hand, Corollary 2.1 shows a non-trivial lower bound for the problem in the restricted MCB. This establishes a separation between the two models.

2.2.2. Oblivious Computations

A computation in the MCB model is *oblivious* if the processors that write and read each given channel in each given cycle are known in advance, independent of the particular instance of the input. In other words, in an oblivious algorithm a processor can determine which channel to read or write in any given cycle simply as a function of the number of cycles that have elapsed form the beginning of the algorithm (and perhaps the general parameters of the problem). A similar definition of oblivious computation has been used in the context of shared memory models [Cook86].

Theorem 2.2. Solving an r-sensitive problem on an MCB(p, k) by means of an oblivious algorithm requires at least $\max\{\log r, \frac{r}{k}\}$ cycles.

Corollary 2.2. An oblivious algorithm for the identification problem requires at least $\max\{\log(p-1), \frac{p-1}{k}\}$ cycles. ∎

Proof of Theorem 2.2. As in Theorem 2.1, let $A_i(t)$ denote the set of processors that *affect* processor P_i in cycle t. It is defined recursively as follows.

1. $A_i(0) = P_i$.
2. $A_i(t+1) = A_i(t) \cup A_j(t)$, where P_j is the processor that writes on the channel that P_i reads in cycle $t+1$.

The definition reflects the fact that the processor writing a given channel in a given cycle is fixed. Hence, contrary to the argument in Theorem 2.1, in an oblivious computation no knowledge is gained about a processor unless that processor actually writes the channel.

Let t^* be the last cycle of the computation. From the recursive formulation, $|A_i(*)| \le 2^{t^*}$. On the other hand, by definition of r-sensitivity, there exists at least one P_i such that $|A_i(t^*)| \ge r$. It follows that $t^* \ge \log r$.

To complete the proof, it remains to show that $\frac{r}{k}$ cycles is also a lower bound. To this end, r processors have to "reveal" local information, which entails r transmissions. Since there are only k channels, at least $\frac{r}{k}$ cycles are needed. ∎

The single-cycle identification algorithm for the general MCB shown in the previous section is not oblivious. This is because the id of the processor that writes the channel is dependent on the particular instance of the input. On the other hand, by Corollary 2.2, there is a nontrivial lower bound on any oblivious algorithm for the problem. This proves that in the general MCB, oblivious computation is less powerful than arbitrary (non-oblivious) computation.

To prove a similar separation in the restricted MCB, we now present a non-oblivious identification algorithm for the restricted model that runs in $\left\lceil \frac{\log k}{\log(\frac{p}{k}+1)} \right\rceil + 1$ cycles.

let us denote the group of $\frac{p}{k}$ processors that have write-access to channel C_i as *group i*. Also, we refer to the processor with the j'th largest id in group i as the j'th processor in the group.

Consider a tree with branching factor $\frac{p}{k}+1$ and k leaves, such that all leaves are located in at most two adjacent levels (a full tree). Figure 2.2 shows a tree for $p=28$, $k=14$. The leaves are numbered sequentially from left to right, beginning with leaf 1. Each parent node has the same number as its rightmost child. The levels of the tree are numbered sequentially from the leaves upward, beginning with level 0. Thus, the root is in level $\left\lceil \frac{\log k}{\log(\frac{p}{k}+1)} \right\rceil$.

We associate the nodes labeled i with processor group i, and the link between each such node and its parent with channel C_i. The idea of the identification algo-

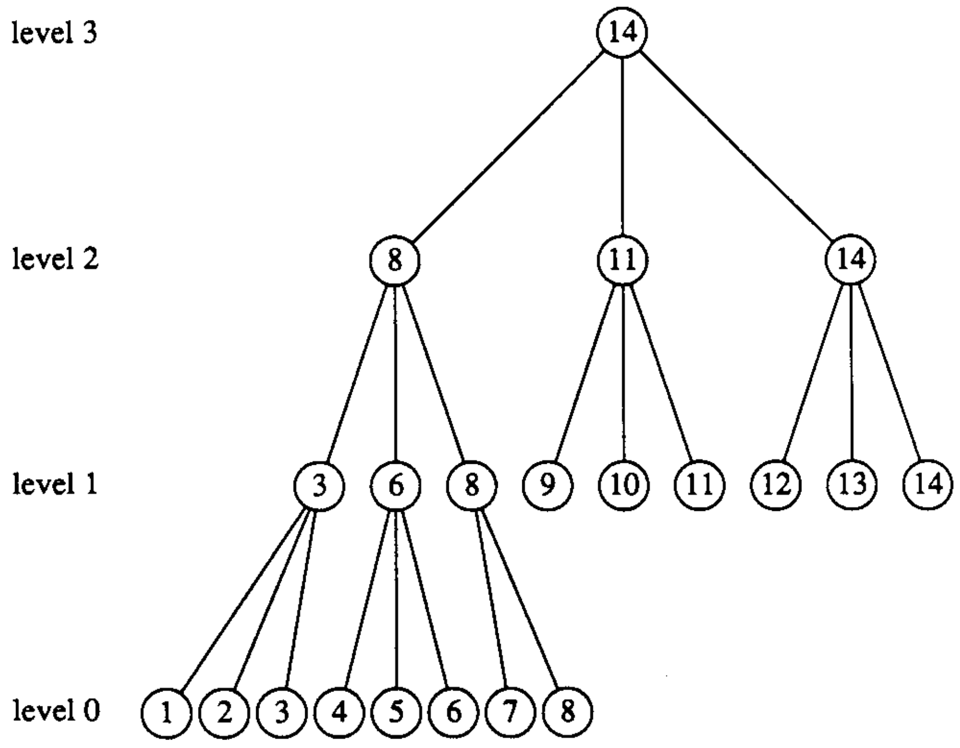


Figure 2.2. Tree for the Identification Problem

rithm is to accumulate the information of the processors bottom-up along the tree. Initially, each processor knows only whether or not its own bit is 1, and this information is in the leaves.

In the first cycle, the information is transferred form the leaf groups in level 0 to the parents in level 1. This is done as follows. The unique processor in level 0 (if any) having the bit 1 writes its id on the corresponding channel. In each parent group in level 1, the i'th processor, $1 \le i \le \frac{p}{k}$, reads the channel of the i'th leftmost child. Notice that the rightmost child and the parent are implemented by the same group, so there is no need to read the rightmost channel. Since only one processor reads each channel, there is a unique processor in all the groups in level 1 that knows the processor whose bit is 1.

In the next cycle, the same scheme is used to transfer the information to level 2. The unique processor in level 1 that knows the id of the processor whose bit is 1 writes that id on the channel. Again, it can be seen that there is a unique processor in level 2 that obtains that id. The scheme is repeated $\left\lceil \frac{\log k}{\log(\frac{p}{k}+1)} \right\rceil$ times until the computation reaches the root of the tree. At that point there is a unique processor in group k (the root group) that knows the solution, i.e., the id of the processor that has the bit 1. To complete the algorithm, the solution is broadcast to all processors over channel C_k.

The total number of cycles used in the algorithm is $\left\lceil \frac{\log k}{\log(\frac{p}{k}+1)} \right\rceil + 1$. For $p \geq 5k$, this is less than $\max\{\log(p-1), \frac{p-1}{k}\}$, thus establishing the separation between oblivious and non-oblivious computation on the restricted MCB. Notice that $\frac{\log k}{\log(\frac{p}{k}+1)} \leq \frac{\log(p-1)}{\log(\frac{p}{k}+1)} + 2$. Following Corollary 2.1, the algorithm takes only 3 cycles more than the lower bound.

The lower bound of Theorem 2.2 is valid both in the general and in the restricted MCB. This gives rise to the question whether the two models are equally powerful for oblivious computation. Intuitively, it seems that deciding "on the fly" which channel to write or read helps only if the decision is based on the input. We therefore conjecture that the answer is affirmative, i.e., that the models are equivalent under oblivious computation.

Clearly, no mechanism can be devised that will convert any arbitrary oblivious algorithm from the general MCB to the restricted MCB without increase in the number of cycles. This is because for some "bad" algorithms there is no way to partition the processors among the channels without contention for write-access. Yet, this in itself does not imply that there exists no algorithm for the problem at hand which runs on the restricted MCB without added cost. Proving (or disproving) the conjecture thus seems to be a difficult task, which may require new proof techniques.

2.2.3. The Relation Between MCB and CREW

The assumptions on concurrent access to the channels in the MCB network are similar to the assumptions on shared memory access in the CREW model [Snir85]. Despite this similarity, we now show that there is a gap in computation power between the two models. The gap stems from the inherent difference between channels and memory we observed earlier, namely that channels are "memoryless."

Recall that a computation is oblivious if the processors that write and read each given channel in each given cycle are known in advance, independent of the input. Now, suppose that the processor scheduled to write on a given channel in a given cycle is allowed to decide, dependent on the input, whether to actually go ahead and write, or just keep silent. A computation that provides this capability but is otherwise oblivious is called *semi-oblivious*. A similar definition of semi-oblivious computation has been used in the CREW model [Cook86].

It is easy to see that the lower bound of Theorem 2.2 is valid for semi-oblivious algorithms. Intuitively, the argument (specifically the definition of $A_i(t)$) does not depend on whether any data is actually written on the channel. Thus, a semi-oblivious MCB algorithm for an r-sensitive problem requires at least $\log r$ cycles.

Consider the problem of computing the logical "or" of p bits distributed among p processors. This is clearly a p-sensitive problem. Hence, a semi-oblivious MCB algorithm for this problem requires $\log p$ cycles. The straightforward solution is to use bottom up processing along a binary tree. This takes $\lceil \log p \rceil$ cycles, which is optimal.

A semi-oblivious algorithm designed for MCB(p, k) can be emulated cycle by cycle on a CREW with p processors and k memory cells. The only delicate issue in the emulation is how does a processor convey its decision to keep silent. This can be done by writing some fixed dummy symbol into the corresponding memory cell.

Consequently, one would expect the tight bound of $\log p$ cycles for the logical "or" problem to carry over to the CREW model. Yet, it has been shown in [Cook86] that the problem can be solved in $\log_{2.618} p + O(1)$ cycles on a CREW comprised of p processors and p shared memory cells, using a semi-oblivious algorithm (the reader is reminded that $\log p$ denotes $\log_2 p$). This establishes that the CREW model is more powerful than an MCB of the same size, when considering semi-oblivious computation.

The logical "or" algorithm on the CREW proceeds as follows. Let f_n denote the n'th Fibonacci number, where $f_0 = 0$, $f_1 = 1$, and $f_{j+2} = f_j + f_{j+1}$. Also, let m_i be the i'th shared memory cell, and let y_i be a local variable of P_i. Initially, $m_i = b_i$, the i'th input bit, and $y_i = 0$. In cycle t, $t \geq 0$, processor P_i executes the following.

1. READ: if $i + f_{2t} \leq p$ then read $m_{i+f_{2t}}$.
2. COMPUTE: $y_i := y_i \vee m_{i+f_{2t}}$.
3. WRITE: if ($y_i = 1$ and $i > f_{2t+1}$) then $m_{i-f_{2t+1}} := 1$.

It is shown in [Cook86] that after cycle t, $m_i = b_i \vee b_{i+1} \vee \cdots \vee b_{i+f_{2t+1}-1}$. Let $f_{2n-1} < p \leq f_{2n+1}$. Then, after n cycles, m_1 contains the solution. It can be shown that $n \leq \log_{2.618} p + O(1)$.

To understand how the "obvious" bound of $\log p$ cycles is beaten, observe that the key feature of the algorithm is the conditional writing in phase 3 of each cycle. A shared memory cell is overwritten only by the value 1, and never by a 0. The effect is that once a 1 has been written into the cell, it is preserved to the end of the computation.

The ability to preserve the current information in a given memory cell by not writing into it is what gives the CREW more computational power than the MCB. In other words, in the CREW model, the decision of a scheduled processor not to write may in itself provide arbitrary information. In the MCB model, on the other hand, due to the memorylessness of the channels, this decision reveals nothing but the fact that no writing took place.

CHAPTER 3
SORTING ALGORITHMS

In this chapter and in Chapter 4 we investigate the problem of sorting in the MCB model. In this chapter we use uniform communication cost, whereas in Chapter 4 we use logarithmic communication cost. In both cases we present efficient sorting algorithms and establish tight lower bounds on the complexity. [1]

3.1. Introduction

Let n elements from a totally ordered domain be distributed in a network of p processors, such that processor P_i contains n_i elements. *Sorting* is the task of reorganizing the elements in the network so that each of the n_i elements in P_i will be greater or equal to each of the elements in P_{i+1}.

We assume $n \geq p$ and $n_i > 0$. If $n_i = \frac{n}{p}$ for all i, we say that the distribution of the input is *even*. Otherwise, the distribution is *uneven*.

The sorting algorithms described in this chapter are based on a method called *COLUMNSORT* [Leig85], which is a generalization of odd-even sorting [Knut73]. In this method, the input is organized in a matrix, and sorting is accomplished by iteratively sorting each column separately, then performing a transformation on the

[1] The results in this chapter have been previously published in the *Proceedings of the 1985 International Conference on Parallel Processing* [Marb85b].
© 1985 IEEE. Reproduced with permission.

matrix. *COLUMNSORT* is practical for our purposes because each column can be implemented in a different processor and sorted locally. Moreover, the matrix transformations can be performed efficiently using the broadcast channels.

Following is a summary of our results. Let $n \geq k^3$ elements be distributed evenly among the processors of an MCB(p, k). We develop an oblivious sorting algorithm that runs in $O(\frac{n}{k})$ cycles and $O(n)$ messages on the restricted version of the model. Generalizing the algorithm to uneven distributions, we achieve a complexity of $O(\max\{\frac{n}{k}, n_{\max}\})$ cycles and $O(n)$ messages, where $n_{\max}$ is the maximum number of elements in any processor. The latter algorithm is non-oblivious and runs on the general MCB. We also develop a recursive version of the algorithm for even distributions, in which each column is sorted by recursive application of the basic algorithm. This version admits a wider range of inputs, namely $n \geq k^{1+\varepsilon}$, where $0<\varepsilon \ll 1$, and achieves the same complexity as the nonrecursive version. Finally, we prove that our algorithms are optimal by establishing matching lower bounds on the complexity.

The discussion proceeds as follows. In Section 3.2 we describe *COLUMNSORT.* In Section 3.3 we give the implementation for even distributions of the input. In Section 3.4 we present the recursive algorithm. Uneven distributions are discussed in Section 3.5. Finally, Section 3.6 shows the lower bounds.

3.2. Algorithm *COLUMNSORT*

Let $m=\frac{n}{k}$. Consider an input of n elements organized in a matrix of size $m \times k$. Traversing the positions of the matrix in lexicographic order by (column, row) is called *column–major* order. Similarly, if the traversal is lexicographic by (row, column), it is called *row–major* order. The output of *COLUMNSORT* is the matrix sorted in column-major order.

COLUMNSORT uses four transformations on the matrix, which are described informally below, and illustrated by examples in Figure 3.1.

Transpose	Take the elements of the matrix in column-major order and store them in row-major order.
Un-Transpose	This is the reverse of Transpose. Take the elements in row-major order and store them in column-major order.
Up-Shift	Shift each element $\lfloor \frac{m}{2} \rfloor$ positions in the ascending direction of the column-major order. The last $\lfloor \frac{m}{2} \rfloor$ elements are shifted circularly to the beginning of the matrix.
Down-Shift	This is similar to up-shift, except that the direction of shift is reversed.

Following is a description of *COLUMNSORT*. The algorithm consists of 9 phases. Upon termination, the elements are stored in descending order of magnitude in column-major order.

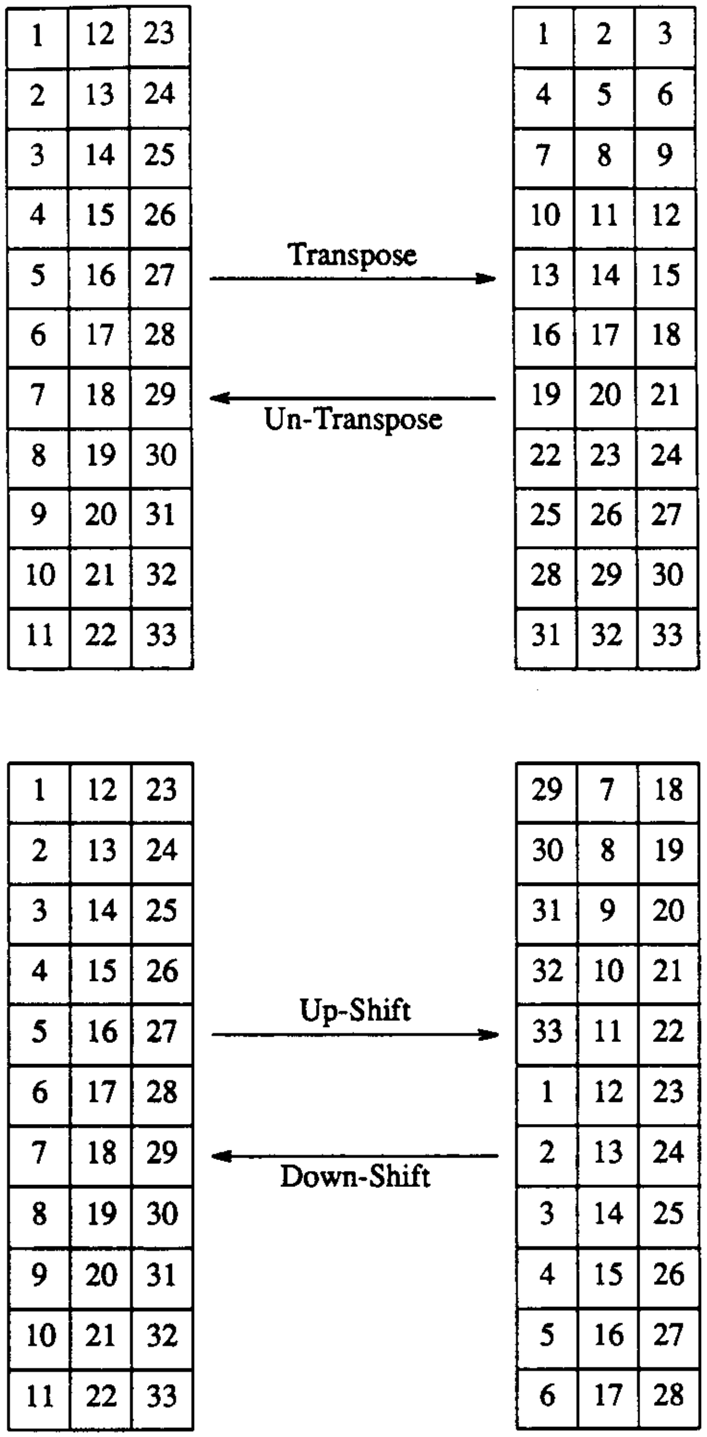


Figure 3.1. Matrix Transformations in *COLUMNSORT*

ALGORITHM *COLUMNSORT* [Leig85]

1. Sort each column.
2. Transpose the matrix.
3. Sort each column.
4. Un-Transpose the matrix.
5. Sort each column.
6. Up-shift the matrix.
7. Sort each column except column 1.
8. Down-shift the matrix.
9. Sort each column.

The version of *COLUMNSORT* presented above is essentially the same as the original version in [Leig85], except that we have added phase 9, which simplifies the distributed implementation, as will be discussed in Section 3.3.

The proof of correctness given in [Leig85] is based on an analysis of the distance of each element from its final position after each phase. Here we present an alternative proof, based on the 0-1 principle [Knut73]. The principle states that if a sorting algorithm works correctly for an arbitrary input of 0's and 1's, then it works correctly for any input.

Given a matrix of 0's and 1's, we say that a given region in the matrix (e.g, row, column, sequence of rows) is *dirty* if it contains both 0's and 1's; otherwise, it is *clean*. The correctness argument proceeds by showing which rows and columns become dirty or clean at different points in the algorithm. The goal is to show that upon termination:

(a) there exists at most one dirty column;

(b) the dirty column is sorted;

(c) the dirty column separates the clean columns of 0's from the clean columns of 1's.

Theorem 3.1.

1. After phase 3 of *COLUMNSORT* there are at most k dirty rows.
2. Given that $m \geq k^2$, after phase 4 there are at most two dirty columns.
3. After phase 8 there is at most one dirty column.
4. After phase 9 the input is sorted.

Proof.

1. Phase 2 has the effect of "dealing" the elements of each column in round-robin fashion among all the columns. Since the columns were previously sorted in phase 1, the difference between the number of 0's that any two columns receive from any given column is at most one. Thus, the total difference in the number of 0's between any two columns after phase 2 is at most k. Sorting the columns again in phase 3 yields at most k dirty rows. Moreover, all the resulting dirty rows are in one contiguous region that separates the clean rows of 0's from the clean rows of 1's. The state of the matrix at the end of phase 3 is illustrated in Figure 3.2.

2. Since $m \geq k^2$, the contents of any k consecutive rows is distributed by the un-transpose operation among at most two adjacent columns. Given the state of the matrix after phase 3, the un-transpose in phase 4 creates at most two adjacent

dirty columns. These separate the clean columns of 0's from the clean columns of 1's.

3. Let x denote the number of 1's in the first dirty column after phase 4, and let y denote the number of 0's in the second dirty column. In order to clean at least one of the two columns, $\min\{x, y\}$ "misplaced" 1's need to be moved from the first dirty column to the second, and an equal number of 0's need to be moved in the opposite direction. Since there are at most k dirty rows at the end of phase 3, it must be that $x+y \leq k^2$, and hence $\min\{x, y\} \leq \lfloor \frac{k^2}{2} \rfloor$.

   The exchange of elements between the two dirty columns is achieved as follows. Sorting the columns in phase 5 separates the 0's from the 1's in each column. The up-shift of $\lfloor \frac{m}{2} \rfloor \geq \lfloor \frac{k^2}{2} \rfloor$ positions in phase 6 has the effect of moving the "misplaced" 1's from the first column to the second. This is illustrated in Figure 3.3. Notice that the 0's that need to be moved in the opposite direction remain in the second column despite the up-shift. Phases 7 and 8 have the symmetric effect in the opposite direction. To prevent exchange of elements between column k and column 1 due to the wrap-around of the circular shift, column 1 is not sorted in phase 7.

   The dirty column that remains at the end of phase 8 separates the clean columns of 0's from the clean columns of 1's. Hence termination conditions (a) and (b) are satisfied after phase 8.

4. Sorting the columns in phase 9 satisfies termination condition (c). This completes the proof of correctness of the algorithm. ■

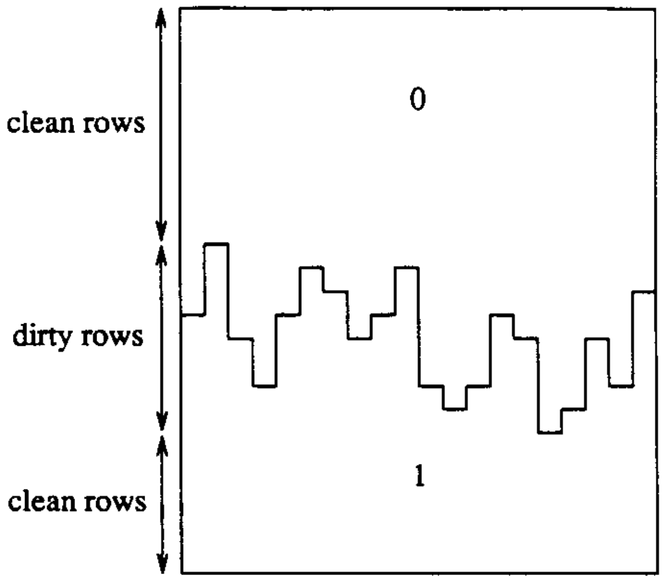


Figure 3.2. Clean and Dirty Rows after Phase 3

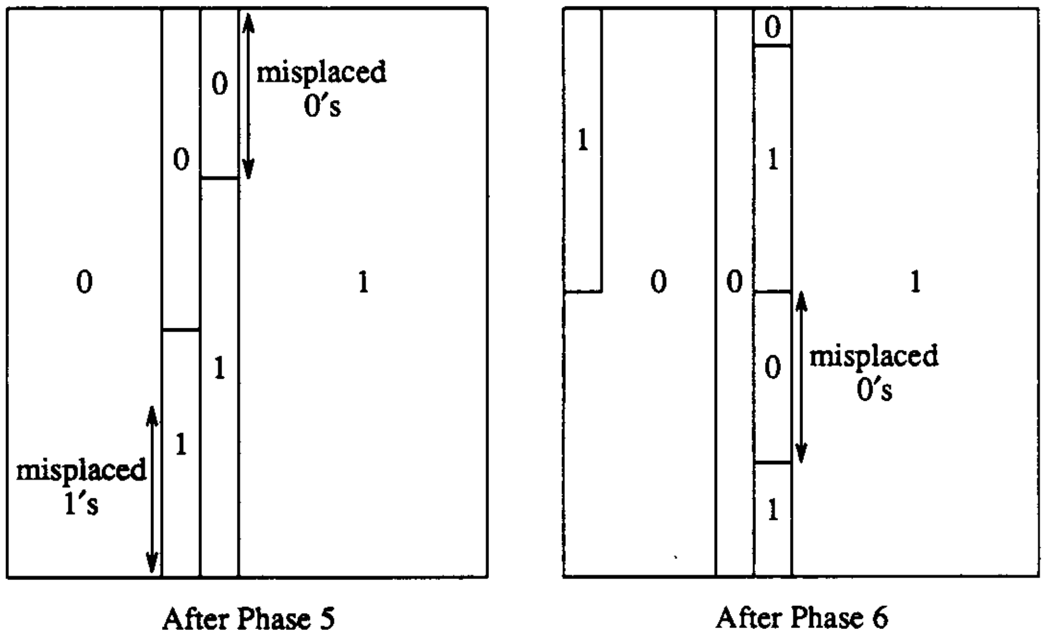


Figure 3.3. Element Exchange between the Two Dirty Columns

It can be seen from Theorem 3.1 that *COLUMNSORT* works correctly only if the dimensions of the matrix satisfy $m \geq k^2$. In other words, in order to use k columns during *COLUMNSORT*, the total number of elements, n, must be at least k^3.

3.3. Implementation for Even Distributions

In this section we show how to implement *COLUMNSORT* on an MCB(p, k) when the distribution of the input is even. First, we handle the simple case where $p=k$ and $n \geq k^3$; then, we generalize to arbitrary $p>k$; finally we discuss the case $n<k^3$. Unless otherwise mentioned, we use the restricted MCB.

3.3.1. The Case $p=k$

There are $k(=p)$ columns, each containing $m=\frac{n}{k}$ elements. Column i is implemented at processor P_i. Phases 1, 3, 5, 7 and 9 are executed locally at each processor, using some efficient sequential sorting algorithm (see [Knut73] for an extensive survey of such algorithms). The remaining phases consist only of traffic over the channels. There are k channels, so all processors can broadcast their columns simultaneously, processor P_i using channel C_i. To move an element from column i to column j, processor P_j must read channel C_i in the appropriate cycle, according to a fixed (predetermined) broadcast schedule. It remains to devise a broadcast schedule for each transformation phase.

We give a schedule for phase 2. Similar schedules can be devised for phases 4, 6 and 8. To maximize the concurrency, k elements with distinct target columns are moved in each cycle. Let us denote by $E_i[j, l]$ the l'th element of column i whose target is column j, where $0 \leq l \leq \lceil \frac{m}{k} \rceil - 1$. Also, let us number the cycles of phase 2 sequentially, beginning with cycle 0. In cycle t, each processor P_i broadcasts element $E_i\left[(i+\left\lfloor \frac{t}{\lceil \frac{m}{k} \rceil} \right\rfloor) \bmod k+1,\ t \bmod \lceil \frac{m}{k} \rceil\right]$ over channel C_i, and the corresponding target processor reads the channel. It can be verified that the schedule implements the transpose operation in $\lceil \frac{m}{k} \rceil(k-1) \leq m$ cycles. Notice that the schedule is oblivious. As an example, a schedule for $m=11$, $k=3$ is given in Figure 3.4. Each element $E_i[j, l]$ is shown with the cycle t in which it is moved.

The order in which elements are stored in each target column during the transformation phases is immaterial, because the columns are sorted in the next phase anyway. The only exception is column 1, which is not sorted in phase 7. The effect of not sorting column 1 is that the same elements that are shifted from column k to column 1 in phase 6 are shifted back to column k in phase 8. Thus, these elements need not be stored in any specific order at P_1 either. Alternatively, these elements need not be shifted at all.

During each of phases 2, 4, 6, and 8, each processor broadcasts at most m elements. The schedules are such that all processors broadcast simultaneously. Thus, the number of cycles in each phase is $O(m)=O(\frac{n}{k})$, and the number of messages is $O(mk)=O(n)$. The sorting phases are implemented locally and incur no communication cost. The total complexity of the algorithm is $O(n)$ messages and $O(\frac{n}{k})$ cycles. The algorithm is oblivious.

	$E_2[3,0]$ $t=0$	$E_3[2,0]$ $t=3$
$E_1[2,0]$ $t=0$	$E_2[1,0]$ $t=4$	
$E_1[3,0]$ $t=4$		$E_3[1,0]$ $t=0$
	$E_2[3,1]$ $t=1$	$E_3[2,1]$ $t=4$
$E_1[2,1]$ $t=1$	$E_2[1,1]$ $t=5$	
$E_1[3,1]$ $t=5$		$E_3[1,1]$ $t=1$
	$E_2[3,2]$ $t=2$	$E_3[2,2]$ $t=5$
$E_1[2,2]$ $t=2$	$E_2[1,2]$ $t=6$	
$E_1[3,2]$ $t=6$		$E_3[1,2]$ $t=2$
	$E_2[3,3]$ $t=3$	$E_3[2,3]$ $t=6$
$E_1[2,3]$ $t=3$	$E_2[1,3]$ $t=7$	

Figure 3.4. Broadcast Schedule for Phase 2

3.3.2. The Case $p>k$

We now show how to generalize the algorithm for arbitrary MCB(p, k), $p>k$. Our approach is to reduce the problem to the previous case. We augment the algorithm with a preprocessing phase and a postprocessing phase, numbered, respectively, 0 and 10. In phase 0, all elements are collected into k processors. Phases 1-9 then proceed as before, as if the network were an MCB(k, k). In phase 10, the sorted elements are redistributed to all the processors.

Phase 0 is implemented as follows. The processors are divided into k *groups* of equal size $\frac{p}{k}$. Group j consists of all the processors P_i such that $\lceil \frac{ik}{p} \rceil = j$ (i.e., the processors that have write-access to channel C_j). Processor $P_{\frac{jp}{k}}$ is the "representative" of the group. One processor after another, in ascending order of processors within the group, all elements of the group are sent to the representative. All groups proceed concurrently, group j using channel C_j. Phase 10 is the inverse of phase 0. The representatives broadcast the columns, and each processor collects its respective elements.

The number of elements in each group is $\frac{n}{p} \cdot \frac{p}{k} = \frac{n}{k}$. Thus, phases 0 and 10 take $O(\frac{n}{k})$ cycles and $O(n)$ messages. Phases 1-9 have the same cost as in the previous case. The total complexity of the algorithm is $O(n)$ messages and $O(\frac{n}{k})$ cycles.

3.3.3. Improvements in Memory Utilization

The implementation described in the previous section requires $\Omega(\frac{n}{k})$ auxiliary memory at each representative processor. The memory requirements can be improved by modifying the implementation, as follows.

We distribute the role of group representative, considering each group to be a single virtual processor with a single virtual column. The elements of the group are not collected, thus phases 0 and 10 are eliminated altogether. Virtual columns are sorted as if each group were a separate MCB($\frac{p}{k}$, 1). Below, we describe a single-channel sorting algorithm called *RANKSORT*, that runs in linear number of messages and cycles and requires only $O(\frac{n}{p})$ auxiliary memory at each processor. Using this algorithm to sort the virtual columns, the complexity of each sorting phase is $O(\frac{n}{k})$ cycles and $O(n)$ messages. In the transformation phases, the work of a given virtual processor in a given cycle is carried out by the processor containing the element to be broadcast in that cycle. The element received during the cycle can be stored over the element just sent, thus requiring no extra memory. The total cycle and message complexities of *COLUMNSORT* remain the same as before.

Algorithm *RANKSORT* proceeds as follows. Without loss of generality (w.l.o.g.) assume that all the elements are distinct. If not, replace each element ξ in P_i with the triple $\langle \xi, i, j_\xi \rangle$, where j_ξ is a unique index within P_i, and use lexicographical order among the triples. Each processor maintains a rank counter for each of its elements. Initially, all counters are set to 1. The algorithm consists of two phases. In the first phase, elements are broadcast in some arbitrary order (e.g., column by column). After each broadcast, the counters of those elements that are smaller than the element just broadcast are incremented by 1. Thus, at the end of the first phase every processor knows the ranks of all its elements. Then, in the second phase, the elements are broadcast in rank order and moved to the appropriate target processors. It is easy to see that the algorithm runs in a linear number of cycles and messages, using only $O(\frac{n}{p})$ auxiliary memory at each processor. Notice, however, that the algorithm is not oblivious.

We may further reduce the memory requirements by replacing *RANKSORT* with the algorithm described below, called *MERGESORT*. The entire *COLUMNSORT* implementation then requires only $O(1)$ auxiliary memory at each processor.

MERGESORT proceeds as follows. W.l.o.g. all elements are distinct. First, each processor sorts its input list. Then, in repetitive phases, the next-largest element in the network is chosen among the top elements of all lists and moved to the appropriate target processor. In order to keep track of the top elements efficiently, the processors maintain a distributed linked list of the top elements, sorted in descending order. Each processor knows its own top element and the next-smaller top element, the latter playing the role of a "pointer." In addition, each processor P_i knows its rank in the distributed list, denoted R_i. The list can be initialized by applying *RANKSORT* to the top elements.

Let P_a be the processor at the head of the list in the current phase (i.e., R_a=1). P_a sends its top element to the appropriate target processor, and all the processors decrement their rank by one, thus effectively removing the head of the linked list. To insert the new top element of P_a into the list, P_a broadcasts the element, and all processors compare its value with that of their own top element. Let P_b be the unique processor, if any, which has both a top element that is larger than the new element and a pointer that is smaller or null. The new element is inserted after that of P_b. This is effectively accomplished as follows. All processors with a smaller top element than that of P_a increment their rank by one. P_b sends R_b and its current pointer to P_a, then changes its pointer to the new element. Finally, P_a changes its pointer to that sent by P_b, and sets R_a to R_b+1. In case the new element is larger than all the other top elements, which can be detected by silence on the channel during the cycles where the rank and pointer for P_a are supposed to be transmitted, P_a resets R_a to 1, thereby remaining at the head of the list.

To achieve $O(1)$ auxiliary memory utilization, when an element is moved to its target processor, the target processor sends its smallest remaining input element to the processor at the head of the linked list, which then inserts this element in the proper position in its own input list. With this scheme, no extra memory is needed for output lists, thus each processor uses only $O(1)$ auxiliary memory. It can be easily verified that *MERGESORT* runs in linear number of cycles and messages.

In terms of communication costs, *RANKSORT* and *MERGESORT* are comparable to the single-channel sorting algorithm in [Dech86], even though the latter algorithm uses concurrent-write access to the channel. It should be noted, however, that the algorithm in [Dech86] is designed to optimize local processing costs (in terms of the number of comparisons) as well as communication costs.

3.3.4. The Case $n<k^3$

We have seen that inputs of size $n<k^3$ cannot be sorted using k columns. To sort inputs of such size, we need to use fewer columns. Clearly, $\lfloor n^{1/3} \rfloor$ columns will work. The processors are divided into groups of size $\lceil \frac{p}{\lfloor n^{1/3} \rfloor} \rceil$, with the last group possibly "padded" with dummy processors containing dummy elements. The algorithm then proceeds as before. The complexity is $O(n)$ messages and $O(n^{2/3})$ cycles. This is obviously suboptimal in the number of cycles, because not all channels are utilized. Also, since not all processors within a group have write-access to the same channel, the algorithm requires the use of the general MCB.

For the subcase $n \le k^{3/2}$, we can provide a different implementation, which results in better cycle complexity than $O(n^{2/3})$. The idea is the following. We collect the elements into k processors, then view the input as a matrix of size $k \times \frac{n}{k}$. In other words, each processor now contains a row, not a column. The dimensions of the "inverted" matrix satisfy the requirements of *COLUMNSORT*. We apply the algorithm, thereby obtaining column-major sorted order, which is then converted to row-major order by transposing the matrix.

Unlike the previous implementations of *COLUMNSORT*, in this configuration the sorting phases cannot be performed locally. Yet, since each of the k elements of a column is located in a different processor, sorting can be done by emulating some sorting network, such as AKS [Ajta83] or the bitonic network [Batc68]. Using AKS, the $\frac{n}{k}$ columns can be sorted in $O(\frac{n}{k} \log k)$ cycles and $O(n \log k)$ messages.

The total complexity of the implementation is $O(\frac{n}{k}\log k)$ cycles, which is better than the previous complexity of $O(n^{2/3})$ cycles. This, however, comes at the price of increasing the message complexity from $O(n)$ to $O(n\log k)$.

Notice that despite its asymptotic optimality, the AKS network is considered impractical due to the large constants involved. Substituting AKS with the bitonic network (which is considerably more practical), the complexity of our algorithm increases by a factor of $\log k$ in both messages and cycles.

3.4. A Recursive Version of *COLUMNSORT*

We have seen that in the range $n<k^3$, the channel utilization of *COLUMNSORT* becomes suboptimal. We now show a recursive implementation of the algorithm which effectively reduces the range of input sizes that exhibit suboptimal performance.

The idea is the following. We implement the sorting phases by applying the algorithm recursively on each column. The column length decreases from one recursive level to the next. The recursion is continued until the columns are sufficiently short to be sorted directly in a small number of cycles. By using the appropriate number of columns in each level we achieve maximum channel utilization, thereby improving the performance.

We now formalize the approach. Let $s\geq 1$ be an integer such that $k\geq 4^s$ and $n\geq k^{\frac{3s+6}{3s+2}}$. W.l.o.g. assume that n, p and k are powers of 4^s (i.e., $n=4^{sr}$, $p=4^{sq}$ and $k=4^{sl}$ for some integers $r\geq q\geq l\geq 1$). Finally, let $\bar{k}=(\frac{k^3}{n})^{\frac{1}{2s}}$ (since we are only interested in the case $n<k^3$, clearly $\bar{k}\geq 2$).

We apply the recursion to a depth of $s+1$ levels. A recursive call in any level except the last uses $\bar{k}$ virtual columns. Thus in level j, $1\leq j\leq s$, each group consists of $\frac{p}{\bar{k}^j}$ processors, and each virtual column consists of $\frac{n}{\bar{k}^j}$ elements. In the last level, level $s+1$, each call uses $\frac{k}{\bar{k}^s}=(\frac{n}{k})^{½}$ columns, and hence each column is of length $\frac{n}{k}$. Notice that the elements of a column are not collected into one processor.

It can be easily verified using the assumptions on n, p and k given above, that $\frac{n}{\bar{k}^j}\geq\bar{k}^2$ for all $1\leq j\leq s$. In other words, the requirement that the column length be at least $\bar{k}^2$ is satisfied in each level of the recursion.

Since $\bar{k}^s\leq k$, there are enough channels for all the recursive calls in the same phase of the same level to proceed in parallel. In level j, each recursive call uses a separate set of $\frac{k}{\bar{k}^{j-1}}$ channels. During the transformation phases, each virtual column is divided into $\frac{k}{\bar{k}^j}$ segments of $\frac{n}{k}$ elements, and all segments are broadcast simultaneously — each on a separate channel. Thus, the number of cycles in each transformation phase in each level is $O(\frac{n}{k})$. In level $s+1$, the length of the columns is $\frac{n}{k}$, so the number of cycles in each sorting phase of that level is also $O(\frac{n}{k})$.

The total number of cycles in the algorithm is $O(s\frac{n}{k})$. The total number of messages is $O(sn)$. We now show that in a wide range of cases s can be assumed constant. Consequently, the complexity of the recursive algorithm is $O(\frac{n}{k})$ cycles and $O(n)$ messages. Let $0<\varepsilon\leq 2$ be a constant such that $k\geq 4^{\lceil\frac{4-2\varepsilon}{3\varepsilon}\rceil}$ and $n\geq k^{1+\varepsilon}$. Now, choose $s=\lceil\frac{4-2\varepsilon}{3\varepsilon}\rceil$. It can be verified that all the restrictions on the magnitude of n and k imposed by the recursive implementation are met. Thus, given ε, s is constant.

3.5. Implementation for Uneven Distributions

In this section we generalize the *COLUMNSORT* approach to uneven distributions. First, however, we digress to discuss a simple algorithm to compute partial sums. We will use this algorithm as a subroutine in the sorting algorithm.

3.5.1. A Partial-Sums Algorithm

Let $\oplus$ denote an associative arithmetic operator, such as "+", "max", etc. Let $\{a_1, a_2, \ldots, a_p\}$ be a set of values distributed in the network, such that a_i is at P_i. We denote by $a_i^{\oplus}$ the partial sum $a_1 \oplus a_2 \oplus \cdots \oplus a_i$. The largest partial sum, $a_p^{\oplus}$, is called the total sum. Also, for convenience, we use $a_0^{\oplus}=\omega$, where ω denotes the identity value of $\oplus$. In the following, we describe an algorithm to compute at each P_i the partial sum $a_i^{\oplus}$. The algorithm is adapted from the F&* tree-machine [Vish84], which is used for similar purposes in a different setting.

Consider a network in the shape of a full binary tree with p leaves (i.e., all leaves are located in at most two adjacent levels). Each node is a processor, and communication is along the edges. a_i is initially at the i'th leftmost leaf. The following computation produces the partial sums $a_i^{\oplus}$ at the leaves.

Assume the local variables L, R, and F at each node contain the last value received from the left son, the right son, and the father, respectively. The computation consists of a bottom-up phase followed by a top-down phase. The bottom-up phase starts with the leaves sending a_i to their father. Upon receiving values from both sons, an internal node sends $L \oplus R$ to its father. When the computation reaches the root, the top-down phase is started with the root sending ω to its left son and L to its right son. An internal node, upon receiving a value from the father, sends F to the left son and $F \oplus L$ to the right son. When the computation reaches the leaves, each leaf sets $a_i^{\oplus} = F \oplus a_i$. It is easy to verify that this scheme correctly computes the partial sums.

The implementation in the MCB model is straightforward. Consider first the case $p=k$. The tree computation is emulated level by level, first bottom-up, then top-down. A father node is implemented in the same processor that implement its right son, thus only the messages between father and left son need actually be sent. Since there are $\lceil \log p \rceil$ levels, the complexity is $O(p)$ messages and $O(\log p)$ cycles.

Now consider an arbitrary MCB(p, k). We divide the processors into k equal groups in the same fashion as in Section 3.3.2. The values of each group are collected to the representative. A tree computation is then performed as if the network were an MCB(k, k) consisting only of the representatives, where the initial value of a representative is the sum (using $\oplus$) of the values in its group. Upon completion of the tree computation, each representative knows (via the variable F of the corresponding leaf) the sum of all the values in the groups to its left. Using this information, the representative computes all the partial sums for its group and broadcasts them. Since there are $\frac{p}{k}$ processors in each group, the total complexity of the algorithm is $O(\frac{p}{k}+\log k)$ cycles and $O(p)$ messages.

Notice that at the root of the tree $L\oplus R=a_p^{\oplus}$. Thus, if only the total sum is of interest, the bottom-up phase followed by a single broadcast message from P_p (the root processor) suffices.

3.5.2. The Sorting Algorithm

The implementation of *COLUMNSORT* in Section 3.3 makes explicit use of the fact that all processors have the same number of elements. We now discuss the problems that arise in generalizing the implementation to uneven distributions, and how these problems can be solved.

With an arbitrary number of elements in each processor, it is impossible to partition the processors into groups that comprise columns of uniform length. Instead, we form groups that have approximately the same number of elements. After the elements of each group are collected into one processor, dummy elements are added where necessary, to make all columns exactly equal.

Now consider the issue of collecting the elements. In the restricted MCB, since the groups are nonuniform, we cannot allocate a separate channel to each group. Consequently, there might be a situation where a subset of processors which have write-access to the same channel contain a total of $\Omega(n)$ elements, even though each group in itself has strictly less than $O(n)$ elements. This creates a communication bottleneck, resulting in a cycle complexity of $\Theta(n)$, which is clearly inefficient. In fact, this complexity can be achieved with a single channel (e.g., using *RANK-SORT*, as described in Section 3.3.3). We therefore use the general MCB. This enables us to allocate a separate channel to each group, as in the even case.

As for synchronization among the processors during element transfer — with even distribution it is easy to implement an exclusive-write strategy: processor P_i waits $\frac{n}{p}((i-1) \bmod \frac{p}{k})$ cycles before sending its elements. In the uneven case, on the other hand, since the groups are nonuniform, the number of cycles each processor has to wait must be determined explicitly during the algorithm.

It is obvious from the above observations that the computation, in particular the task of element collection, is dependent on the specific distribution of the input. This renders the algorithm non-oblivious, in contrast to the algorithm for even distributions.

We now describe the implementation. Let n_i denote the number of elements in processor P_i. Also, let $n_{\max}=\max\{n_i \mid 1 \le i \le n\}$ and $m=\max\{\frac{n}{k}, n_{\max}\}$. We assume $n \ge k^3$.

Using the partial-sums algorithm, the processors compute the values $n_{\max}$ and m, and the partial sums $n_i^{\dagger}$. Group j is formed by all processors P_i such that $\lceil \frac{n_i^{\dagger}}{m} \rceil = j$. It can be seen that there are at most k groups, each group containing at most $m+n_{\max}-1 \le 2m$ elements. Similar to the even case, the representative of group j is processor $P_{\frac{jp}{k}}$ (which may or may not be in the group), and the channel allocated to the group is C_j.

As for element collection, the number of cycles a processor has to wait to send its elements is equal to the total number of elements in lower-indexed processors in the same group. Let P_{f_j} denote the first (i.e., least-indexed) processor in group j. The number of cycles processor P_i in group j has to wait can be expressed as $(n_i^\dagger - n_i) - (n_{f_j}^\dagger - n_{f_j})$. All that needs to be done to compute this number is for P_{f_j} to broadcast $n_{f_j}^\dagger - n_{f_j}$. To this end, notice that P_i is the first processor of group j if and only if $\lceil \frac{n_i^\dagger}{m} \rceil = j$ and $\lceil \frac{n_i^\dagger - n_i}{m} \rceil = j-1$. Thus, P_{f_j} can easily identify itself.

After the elements have been collected, the columns are padded with dummy elements up to length $m+n_{\max}-1$. Since element collection may take a different number of cycles in each group, a global synchronization point for the beginning of phase 1 needs to be set. This can be at $m+n_{\max}$ cycles after the beginning of element collection.

The remainder of the algorithm then proceeds as in the even case. Notice, however, that due to the padding of columns, it might be that not all the elements that go to a given processor in phase 10 are in the same column at the end of phase 9. Nevertheless, since there are at least $n_{\max}$ elements in each column, the elements that go to each processor are guaranteed to be in at most two adjacent columns. Group representatives broadcast each element twice, thus enabling processors to collect all their elements without missing any messages.

There are at most $m+n_{\max}-1<2m$ elements in each group, so the cost of collecting the elements and the cost of each transformation phase is $O(m)=O(\max\{\frac{n}{k}, n_{\max}\})$ cycles and $O(n)$ messages (notice that dummy elements need not be broadcast). The applications of the partial-sums algorithm in phase 0 incur a cost of $O(\frac{p}{k}+\log k)$ cycles and $O(p)$ messages. The total complexity of the algorithm is thus $O(n)$ messages and $O(\max\{\frac{n}{k}, n_{\max}\})$ cycles.

3.6. Lower Bounds

We now establish lower bounds on the complexity of sorting under the assumption of uniform communication cost.

Clearly, if n elements are to be rearranged in the network, $\Omega(n)$ messages and $\Omega(\frac{n}{k})$ cycles are lower bounds. We can, however, prove a stronger result. We show that these bounds hold even if all we require is that processors obtain a list of "pointers" to the elements in the sorted order, without actually transferring the elements to their destination processors. For example, the index pair (i, j) could serve as a pointer to the j'th element in processor P_i.

We use the following notation. N is the list of all the elements in the network, $|N|=n$. N_i denotes the list of elements at processor P_i, $|N_i|=n_i$. Also, $N[j]$ is the j'th largest element in N, and $N[j_1, j_2]$, $j_1 \le j_2$, is the list $[N[j_1], N[j_1+1], \ldots, N[j_2]]$. We define $N_i[j]$ and $N_i[j_1, j_2]$ similarly. Finally, $n_{\max}$ and $n_{\max2}$ denote, respectively, the largest and the second-largest among the n_i.

Theorems 3.2 and 3.3 below use a comparison argument. The results therefore apply only to comparison-based sorting algorithms (i.e., algorithms where the elements are used only in comparisons with each other).

Theorem 3.2. Sorting n elements requires $\Omega(n-n_{\max}+n_{\max 2})$ messages.

Clearly, a lower bound on messages for MCB(p, k) immediately implies a lower bound, smaller by a factor of k, on cycles.

Corollary 3.1. Sorting n elements requires $\Omega(\frac{n-n_{\max}+n_{\max 2}}{k})$ cycles. ■

Proof of Theorem 3.2. Given the input N and the cardinalities n_i, we will devise lists N_i such that $\Omega(n-n_{\max}+n_{\max 2})$ messages are required.

Regardless of the sorting method being used, each element must be directly compared with its immediate predecessor and successor in the sorted order. The lists N_i are constructed so that for sufficiently many disjoint comparisons, the two elements to be compared are in different processors, thus requiring at least one message per comparison.

Let $n_{i_1} \geq n_{i_2} \geq \cdots \geq n_{i_p}$ be a non-increasing order among the n_i. Let q_m denote the number of processors for which $n_i \geq m$. Consider the distribution where $N_{i_j}[l]=N[j+\sum_{m=1}^{l-1} q_m]$. In other words, the elements are distributed by going in round-robin fashion from processor to processor, and placing one element at a time in the sorted order in each processor. It can be easily seen that no two immediate neighbors in $N[1, n-n_{\max}+n_{\max 2}]$ are in the same processor. Thus, at least $\lfloor \frac{n-n_{\max}+n_{\max 2}}{2} \rfloor = \Omega(n-n_{\max}+n_{\max 2})$ messages are required to complete the sorting. ■

The following theorem gives a different lower bound on the number of cycles.

Theorem 3.3. Sorting n elements requires $\Omega(\min\{n_{\max}, n-n_{\max}\})$ cycles.

Proof. Let $P_{i_{\max}}$ denote a processor such that $n_{i_{\max}}=n_{\max}$. We use an argument similar to Theorem 3.2, except that here we focus only on comparisons involving elements of $P_{i_{\max}}$.

Assume first that $n_{\max}\le\frac{n}{2}$ (i.e., $\min\{n_{\max}, n-n_{\max}\}=n_{\max}$). Consider a distribution where for all $1\le j\le n_{\max}$, $N[2j]$ is at $P_{i_{\max}}$ whereas $N[2j-1]$ is at some other processor. In this distribution, at least $n_{\max}$ messages are required in order to compare all pairs of immediate neighbors in $N[1, 2n_{\max}]$. Since $P_{i_{\max}}$ is involved in each such message (either as sender or as receiver), the number of cycles is also at least $n_{\max}$. A similar argument shows that when $n_{\max}>\frac{n}{2}$, at least $n-n_{\max}$ cycles are required, and the result follows. ■

The following corollary shows that the lower bounds are tight, and that the algorithms we have presented are optimal in a wide range of cases.

Corollary 3.2. Given a constant $0<\alpha<1$ such that $n_{\max}\le\alpha n$ and $n\ge k^3$, the complexity of sorting n elements on an MCB(p, k) is $\Theta(n)$ messages and $\Theta(\max\{\frac{n}{k}, n_{\max}\})$ cycles.

Proof. $n_{\max}\le\alpha n$ implies $n-n_{\max}\ge(1-\alpha)n\ge(1-\alpha)n_{\max}$. Thus, the lower bounds of Theorems 3.2 and 3.3 and Corollary 3.1 reduce to $\Omega(n)$, $\Omega(n_{\max})$, and $\Omega(\frac{n}{k})$, respectively. The matching upper bounds are provided by the algorithms presented in previous sections. ■

For even distribution of the input, Corollary 3.2 applies both in the general and in the restricted MCB. For uneven distribution, the upper bound is valid only in the general MCB. Moreover, in the even case, due to the recursive version of *COLUMNSORT*, the result holds in the wider range $n \geq k^{1+\varepsilon}$, $0<\varepsilon \ll 1$. It should be noted that the tight bounds on messages and cycles are simultaneous.

It is easy to see that sorting is a p-sensitive problem. This is because a change in the value of one input element is likely to affect the position of any number of elements in the sorted order. Following Theorem 2.1, a sorting algorithm on the restricted MCB(p, k) requires $\dfrac{\log p}{\log(\frac{p}{k}+1)}$ cycles. Consequently, the AKS-based algorithm presented in Section 3.3.4, which runs in $O(\frac{n}{k}\log k)$ cycles, is optimal for $n=O(p)=O(k)$.

CHAPTER 4

SORTING ALGORITHMS WITH BIT COMMUNICATION

4.1. Introduction

This chapter continues the investigation of sorting algorithms. Here we assume logarithmic communication cost, i.e., each bit to be transmitted requires a separate cycle. Although one could apply the algorithms of Chapter 3 under logarithmic cost, we use a completely different approach, tailored specifically for bit communication.

The configuration being considered is an MCB(p, p) with one element per processor. For our purposes, each element is a bit string of uniform length m. An MCB(p, p) can easily emulate a sorting network for p elements, such as AKS [Ajta83] or the bitonic network [Batc68]. However, under logarithmic communication cost this becomes inefficient when the elements are long, since $\Theta(m)$ cycles are needed to emulate each stage of the sorting network. AKS emulation, for example, requires a total of $\Theta(m \log p)$ cycles.

Our goal is to develop a sorting method which does not entail repeated transmission of long elements. To accomplish this, we separate the task of computing the position of each element in the sorted order from the actual rearrangement of the elements in the network.

The idea is to compute for each element a short encoding called *signature*, such that the signatures have the same relative order as the original elements. The sorted order can then be found efficiently using the signatures.

We present a sequence of three algorithms, A, B and C, based on this approach. Each algorithm improves upon the previous one, using a more efficient technique to sort the signatures. Algorithm A uses bottom-up processing on a tree, running in a total of $O(m+p\log^2 p)$ cycles. Algorithm B combines the tree with a bitonic network, and runs in $O(m+p\log p \log\log p)$ cycles. Algorithm C employs divide and conquer, and has a complexity of $O(m+p\log p)$ cycles. The first two algorithms are oblivious. By showing a lower bound of $\Omega(m)$ cycles, we prove that the algorithms are optimal for sufficiently large m. We also discuss generalizations of algorithm A for network configurations with fewer than p channels and more than one element in each processor.

The discussion proceeds as follows. Sections 4.2 through 4.4 present the sorting algorithms. The lower bounds are shown in Section 4.5. Generalizations are discused in Section 4.6. Concluding remarks are given in Section 4.7.

4.2. Algorithm A

Before we describe the algorithm, we need the following definitions. Let $[a_1, a_2, \ldots, a_l]$ denote a list of l elements. Given two lists of equal length l, $X=[x_1, x_2, \ldots, x_l]$ and $Y=[y_1, y_2, \ldots, y_l]$, we use (X, Y) to denote the list of pairs $[(x_1, y_1), (x_2, y_2), \ldots, (x_l, y_l)]$.

Let $Z=[z_1, z_2, \ldots, z_l]$ be a list of l not necessarily distinct elements from a totally ordered domain, and let $z_{i_1} \geq z_{i_2} \geq \cdots \geq z_{i_l}$ be a nonincreasing order among the elements. The list $[i_1, i_2, \ldots, i_l]$ is called the *sorting permutation* of Z.

The *rank* of z_i in Z, denoted $R(z_i, Z)$, is defined as the number of elements in Z that are strictly larger than z_i. We use $R(Z)$ to denote the list $[R(z_1, Z), R(z_2, Z), \ldots, R(z_l, Z)]$, called the *rank list* of Z. Computing $R(Z)$ is called *ranking*.

Following our approach, the algorithm consists of three phases: (1) signature computation; (2) signature sorting; and (3) element transfer.

4.2.1. Signature Computation

W.l.o.g. assume p divides m. Let e_i denote the element initially at processor P_i. Let $e_{i,j}$, $1 \le j \le p$, be a substring of e_i of length $\frac{m}{p}$, starting at position $(j-1)\frac{m}{p}+1$. We call $e_{i,j}$ the j'th *component* of e_i. We can view the input as a square matrix of components $\{e_{i,j} \mid 1 \le i,j \le p\}$, with the i'th row located at processor P_i.

The signatures are obtained in the following manner. Let B_j denote the j'th column of the component matrix. We compute for each component $e_{i,j}$ the rank $r_{i,j}=R(e_{i,j}, B_j)$. All ranks are in the range 0 to $p-1$, so we view them as strings of $\lceil \log p \rceil$ bits. The signature of element e_i is the concatenation of the ranks of its components, i.e., the string $r_{i,1} r_{i,2} \cdots r_{ip}$. Thus, each signature consists of $p \lceil \log p \rceil$ bits.

It can be verified that the signatures have the same relative order as the original elements. This follows from the fact that ranking preserves order. Notice that when $m < p \log p$ the signature is actually longer than the element itself. This has no bearing on the performance of the algorithm.

To implement phase 1, we first transpose the component matrix, thereby moving each column B_j to processor P_j. Then, the ranks in each column are computed locally, yielding a transposed matrix of ranks. As will be seen in the description of phase 2, there is no need to "un-transpose" the ranks and explicitly form the signatures.

The transpose operation is implemented using the following oblivious communication protocol. There are $p-1$ steps. In step t, $0 \le t \le p-2$, processor P_i sends component $e_{i,(i+t) \bmod p+1}$ to processor $P_{(i+t) \bmod p+1}$ over channel C_i. Each step takes $\frac{m}{p}$ cycles, for a total of $\frac{m}{p}(p-1)=O(m)$ cycles.

4.2.2. Signature Sorting

The second phase computes the sorting permutation using the signatures. This could be done in $O(p \log^2 p)$ cycles by straightforward emulation of the AKS sorting network. However, due to the large constants involved, this is impractical. Instead, we use the following method, which achieves the same complexity but is considerably more practical.

We begin by ranking the signatures. Let Z denote the corresponding rank list. Since ranking preserves order, the ranks in Z have the same relative order as the original elements. We thus obtain the sorting permutation from Z.

We now describe how to compute Z. Let s_i denote the signature of element e_i. Breaking each signature into two parts of equal length, let s_i^+ denote the left part (the most significant bits) and let s_i^- denote the right part (the least significant bits). Let S, S^+, and S^- denote the lists comprising all the s_i, s_i^+, and s_i^-, respectively.

The reader may verify the correctness of the formula $Z = R(S) = R((R(S^+), R(S^-)))$, where the order among rank pairs is determined lexicographically. This suggests the following bottom-up tree computation of Z. We divide each signature into p equal substrings, or components. Now, consider a full binary tree with p leaves (i.e., the leaves are in at most two adjacent levels), where the j'th leftmost leaf contains a list comprising the j'th component of every signature. Moving bottom-up in the tree, we combine at each parent the lists of its two children, as follows. Let G^+ and G^- denote the lists of the left and right child, respectively. The parent is assigned the list $R((G^+, G^-))$. It is easy to see that the root contains the rank list Z.

Notice that the j'th leaf list is actually the j'th column in the rank matrix of phase 1. That column is located in processor P_j. We implement each parent node in the same processor that implements its left child. The root is thus in processor P_1. To evaluate a parent node, G^- is sent from the processor implementing the right child to the processor implementing the parent. The latter then computes $R((G^+, G^-))$ locally. All nodes in the same level in the tree are processed in parallel. There are $\lfloor \log p \rfloor$ levels, each entailing $p \lceil \log p \rceil$ cycles to transmit the lists G^-. The total cost of the tree is therefore $O(p \log^2 p)$ cycles.

It remains to obtain the sorting permutation from Z. Since Z is in P_1, this can be done locally. Finally, P_1 broadcasts the permutation to all processors. The total cost of phase 2 is $O(p \log^2 p)$ cycles.

4.2.3. Element Transfer

In the third and final phase of the algorithm, the elements are transferred to their destination processors according to the sorting permutation. Let P_{d_i} be the destination of element e_i. Since the permutation has been broadcast, P_{d_i} knows the index i. e_i can therefore be sent directly from P_i to P_{d_i} over channel C_i. All the processors proceed in parallel. The total cost of the transfer is $O(m)$ cycles.

The reader may observe that the transfer protocol just described is not oblivious, since the id of the channel to be read by each processor depends on the sorting permutation. On the other hand, phases 1 and 2 are oblivious. We now give an oblivious transfer protocol which is as efficient as the non-oblivious one.

The protocol consists of three steps. First, the elements are divided into p equal components and transposed, similar to phase 1. In fact, if storage availability permits the processors to save the component lists B_j in phase 1, this step is redundant. Second, each P_j locally rearranges B_j according to the sorting permutation. That is, if the destination of element e_i is processor P_{d_i}, then $e_{i,j}$ is placed in position d_i in the list. Finally, the rearranged components are transposed a second time. It can be verified that this effectively accomplishes the transfer. The cost is $O(m)$ cycles.

4.2.4. Complexity

Summing up the costs of all three phases, the complexity of algorithm A is $O(m+p\log^2 p)$ cycles. If m is sufficiently large, it dominates the complexity. In Section 4.5 we show that in this case the algorithm is optimal.

Notice that the algorithm beats AKS emulation when $m \geq p \log p$. This illustrates the difference between logarithmic and uniform communication.

4.3. Algorithm B

In phase 2 of algorithm A, as the computation gets closer to the root of the tree, more and more processors become idle. The idea of algorithm B is to modify the tree computation in order to increase processor utilization, thereby improving the performance. We now describe how this is accomplished.

The ranks in each level of the tree can be viewed as components of new signatures whose length is half the length of the signatures in the previous level. Based on this observation, we make the following change. Instead of evaluating the entire tree, we stop at a level where the new signatures are sufficiently short, then switch to emulation of the bitonic sorting network [Batc68] on these signatures. Appending the signature of element e_i with the index i, the effect of the sorting is that each processor P_j knows the j'th index in the sorting permutation. To make the entire permutation public, one processor after another broadcasts the permutation index. Notice that prior to the bitonic sort it is necessary to transpose the current ranks, so that each processor will contain its signature.

Let the tree computation be discontinued after r levels. The length of the signatures is then $O(\frac{p \log p}{2^r})$ bits. There are $O(\log^2 p)$ phases in the bitonic network, so the emulation takes $O(\frac{p \log p}{2^r} \log^2 p)$ cycles. The cost of r tree levels is $O(rp \log p)$ cycles. By choosing $r = 2 \log\log p$, the total cost of phase 2 becomes $O(p \log p \log\log p)$ cycles.

Phases 1 and 3 are the same as in algorithm A. The total complexity of algorithm B is therefore $O(m+p\log p \log\log p)$ cycles. Notice that in contrast to the AKS network, the simple structure of the bitonic network results in a very practical algorithm.

4.4. Algorithm C

The main idea in phase 2 of algorithms A and B is to iteratively reduce the length of the signatures by half. Yet, using the tree mechanism, each reduction step takes the same number of cycles, $p\lceil\log p\rceil$, regardless of the current length of the signatures. If each reduction could be performed at a cost which is linear in the current length of the signatures, the complexity of phase 2 would improve to $O(p\log p)$ cycles. This is basically what is achieved in algorithm C.

Phases 1 and 3 of algorithm C are the same as in the previous algorithms, and will not be discussed. In phase 2, instead of a tree, we use a divide and conquer approach resembling radix-exchange sort [Knut73]. The idea is the following. Initially, all p processors comprise one group. We divide the processors into several subgroups, each comprising at most $\lceil\frac{p}{2}\rceil$ processors. The division is such that the input elements in each subgroup occupy successive positions in the sorted order, starting at a given (known) position. From now on, it remains only to determine the order within each subgroup. Moreover, all subgroups can proceed in parallel, independent of each other. Each subgroup computes a new set of signatures, using the same method as in phase 1 but starting from the current signatures. It can be seen that the length of the new signatures is at most half the previous length. The division is then repeated recursively in each subgroup until all subgroups become singletons, at which point each processor knows the position of its input element in the sorted order. To obtain the sorting permutation, one processor after another broadcasts its position.

We now show how to implement each recursive level in linear number of cycles in the current length of the signatures. Consider a group R in a given level of the recursion, consisting of $r=|R|$ processors, $P_{i_1}, P_{i_2}, \ldots, P_{i_r}$. The length of the signatures in R is $r\lceil \log p \rceil$ bits. Let the signatures be organized in a transposed matrix of $r \times r$ components (see algorithm A).

Processor P_{i_1}, which contains the first (most significant) column of signature components, divides the processors into subgroups, such that each subgroup comprises all and only those processors which have the same first component. Clearly, the elements in each subgroup belong in successive positions in the sorted order. Moreover, since each signature component is actually a rank, the component corresponding to a given subgroup is a "pointer" to the position in the sorted order (of the elements of R) where the largest element of the subgroup belongs.

Obviously, there exists at most one subgroup with more than $\lceil \frac{r}{2} \rceil$ processors. Let us call this the "bad" subgroup. P_{i_1} informs group R about the division by broadcasting the processor ids and the corresponding pointer of every subgroup, except the bad subgroup. The processors and pointer of the latter can then be determined by elimination.

It now remains to further divide the bad subgroup. This is done by processor P_{i_2}, using the second most-significant signature component, in a similar way as before. The scheme continues component after component until either the size of the bad group is reduced below $\lceil \frac{r}{2} \rceil$, or all components are exhausted. In the latter case, the bad subgroup can be excluded from the remainder of the recursion since all its input elements are identical.

It can be seen that each processor id is broadcast at most once during the above protocol. Thus, the cost of dividing group R is $O(r \log p)$ cycles. The new signatures in each subgroup are then computed form the current signatures using the method of phase 1, which also costs $O(r \log p)$ cycles.

Since the size of the groups is reduced by at least half in each recursive level, the recursion terminates after $\lceil \log p \rceil$ levels. In each level the groups proceed in parallel, so the total cost of phase 2 is $O(\sum_{l=0}^{\log p} (\frac{p}{2^l} \log p))=O(p \log p)$ cycles. Notice that it is necessary to set a global synchronization point at the end of phase 2, since groups of different sizes proceed through the recursion at a different pace.

The total complexity of algorithm C is $O(m+p \log p)$ cycles. It can be seen that the algorithm is not oblivious, in contrast to algorithms A and B. This is because the division into groups is dependent on the input. It is interesting whether the upper bound established by algorithm B can be improved by means of an oblivious algorithm.

4.5. Lower Bounds

We now establish a lower bound on the complexity of sorting under logarithmic communication cost.

Clearly, if elements of length m are to be rearranged in the network, $\Omega(m)$ cycles is a lower bound. Yet, similar to the uniform case, we can prove a stronger result by showing that this bound holds even if all we require is that destination processors obtain "pointers" to the elements in the sorted order, without actually transferring the elements. For example, processor P_j could use as pointer the j'th index in the sorting permutation.

Theorem 4.1. Sorting strings of length m requires $\Omega(m)$ cycles.

Proof. Consider the following problem. Let e_1 be a bit string known only to P_1, and e_2 a bit string known only to P_2. Given that $e_1 \neq e_2$, we want to determine whether or not $e_1 > e_2$.

Yao's lower bound on two-party protocols [Yao79] shows that a solution involving only P_1 and P_2 requires $\Omega(m)$ cycles. Essentially, the argument in [Yao79] is the following. Consider a $2^m \times 2^m$ matrix representing all the possible combinations of values of e_1 and e_2. Deciding on the outcome of the comparison is equivalent to identifying a rectangle within the matrix such that two properties hold: (1) the given input combination is in the rectangle; and (2) the outcome for all the combinations in the rectangle is the same. Since there are $\Omega(2^m)$ disjoint rectangles satisfying the second property, $\Omega(m)$ bits need to be communicated to isolate a specific rectangle.

The lower bound holds even if more than two processors are involved in the computation. This is because the contribution of processors other than P_1 and P_2 is warranted only by the information they obtain from P_1 and P_2, and such information may as well be communicated directly between the two processors.

On the other hand, the comparison problem can be solved by sorting, using dummy elements $e_i=0$ in all processors except P_1 and P_2. To this end, $e_1>e_2$ if and only if the first index in the sorting permutation is 1. Consequently, any lower bound on the comparison problem is also a lower bound on sorting. Hence, sorting requires $\Omega(m)$ cycles. ■

Corollary 4.1. When m is sufficiently large, algorithms A, B, and C are optimal. Specifically, algorithm C is optimal for $m \geq p \log p$. ■

Another implication of Theorem 4.1 is that the divide and conquer method used in algorithm C is optimal, in the sense that signatures of length $p \log p$ cannot be sorted in less than $\Omega(p \log p)$ cycles. Thus, algorithm C is the best possible implementation of the signature approach.

4.6. Generalizations of Algorithm A

We now generalize algorithm A to networks with fewer than p channels and more than one element in each processor. First, we consider the case of an MCB(p, p) with a total of n elements, each processor containing $\frac{n}{p}$ elements. We then show how to extend the approach to arbitrary $k \leq p$ channels. Finally, we discuss uneven input distributions.

4.6.1. The Case $n>p=k$

Dividing each element into p components, we can view the input as a set of $\frac{n}{p}$ component matrices, each matrix comprising one element from every processor. We transpose each matrix separately, then form at each processor a single component list by merging the corresponding columns of all matrices. The remainder of phase 1 and phase 2 then proceed as before. In phase 3, we rearrange the component matrices according to the sorting permutation, then use $\frac{n}{p}$ transpose operations to accomplish the transfer.

The cost of the transpose operations in phases 1 and 3 is $O(\frac{n}{p}m)$ cycles. The cost of phase 2 is $O(n\log n \log p)$ cycles (notice that each rank list now consists of $n\log n$ bits). The total complexity of the algorithm is therefore $O(\frac{n}{p}m + n\log n \log p)$ cycles.

4.6.2. The Case $n>p>k$

We now consider an MCB(p, k) with arbitrary $k \leq p$ channels. The approach is similar to the implementation of *COLUMNSORT* in Chapter 3. We divide the processors into k equal groups, collecting the elements of each group to one "representative" processor. We then proceed as if the network were an MCB(k, k) with $\frac{n}{k}$ elements in each processor. Finally, we redistribute the elements in the sorted order within each group.

The cost of collecting and redistributing the elements is $O(\frac{n}{k}m)$ cycles. Adding the cost of sorting (which is similar to the previous case), the total complexity is $O(\frac{n}{k}m + n\log n \log k)$ cycles. This is optimal for $m \geq k\log n \log k$, since moving n strings of m bits using k channels requires $\Omega(\frac{n}{k}m)$ cycles.

Provided $n \geq k^3$, we can sort optimally even if $m < k \log n \log k$. This can be done by adapting *COLUMNSORT* to bit communication. Under uniform communication cost, the algorithm runs in $O(\frac{n}{k})$ cycles. Under logarithmic cost, each cycle expands to m cycles, so the complexity is $O(\frac{n}{k}m)$ cycles, which is optimal.

4.6.3. Uneven Distributions

The difficulties in generalizing algorithm A to uneven distributions are similar to those encountered with *COLUMNSORT*. We therefore use the same approach, i.e., we form processor groups with approximately the same number of elements, collect the elements of each group to one processor, then proceed as in the even case. Similar to Chapter 3, the algorithm is non-oblivious and runs on the general MCB.

The implementation is considerably simpler than under uniform cost. Let n_i denote the number of elements in P_i, and let $n_{\max}$ be the largest n_i. The cardinalities n_i are broadcast one after another using a total of $O(p \log n)$ cycles. The calculations involved in group formation are then performed by each processor separately, making it unnecessary to use the partial-sums algorithm. Since each group contains at most $\max\{\frac{n}{k}, n_{\max}\}$ elements, the cost of element collection and redistribution is $O((\max\{\frac{n}{k}, n_{\max}\})m)$ cycles. A similar cost is incurred in phases 1 and 3 of the sorting. Phase 2 is identical to the even case, and costs $O(n \log n \log k)$ cycles. The total complexity for uneven distributions is therefore $O((\max\{\frac{n}{k}, n_{\max}\})m + n \log n \log k)$ cycles.

4.7. Concluding Remarks

We have investigated the complexity of sorting under logarithmic communication cost. Algorithm C is optimal for $m \geq p \log p$. On the other hand, when $m \leq p$, AKS emulation achieves a better performance of $O(m \log p)$ cycles. An open problem is to bridge the gap between the upper bound $O(p \log p)$ and the lower bound $\Omega(m)$ in the range $p < m < p \log p$. It also remains open whether $O(m \log p)$ is optimal for $m \leq p$.

Landau, Yung and Galil [Land85] use a model which is equivalent to ours to solve the *Multiple Identification* problem. In this problem, each of p processors contains a string of m bits, and needs to identify all the processors which have the same string as itself. The approach in [Land85] is to sort the strings, then use the sorted order to form groups of processors with identical strings. Sorting is performed by AKS emulation, which takes $O(m \log p)$ cycles. The total complexity of the solution is $O(m \log p + p)$ cycles. By replacing the AKS emulation with algorithm C, we are able to improve the upper bound for the multiple identification problem by a factor of $\log p$. Following Theorem 4.1, this is optimal.

CHAPTER 5
SELECTION ALGORITHMS

In this chapter we consider the problem of selecting the d'th largest element among a collection of elements distributed in an MCB(p, k). We present an efficient algorithm for the problem and establish matching lower bounds. [1]

5.1. Introduction

Let n elements from a totally ordered domain be distributed in the network. *Selection* is the task of identifying the d'th largest element, for a given rank d. W.l.o.g. we may assume that $1 \le d \le \lceil \frac{n}{2} \rceil$ (if not, reverse the order and select the element of rank $n-d+1$). Of specific interset is the rank $d=\lceil \frac{n}{2} \rceil$, which is called the *median*.

A naive approach to selection is to sort the elements, then retrieve the selected element directly by rank. This is inefficient because the extra information provided by sorting comes at a cost and is not really needed. A more promising approach is the following. Reduce the number of candidates for selection by repetitively applying an efficient filtering mechanism. When the number of remaining candidates gets below a specified threshold value, sort the remaining candidates and

[1] The results in this chapter have been previously published in the *Proceedings of the 1985 International Conference on Parallel Processing* [Marb85b].
© 1985 IEEE. Reproduced with permission.

retrieve the selected element by rank.

In this chapter we present a selection algorithm which follows this approach. The complexity of the algorithm is $O(\frac{p}{k}\log\frac{kn}{p})$ cycles and $O(p\log\frac{kn}{p})$ messages on a restricted MCB(p, k), using uniform cost. We prove that this is optimal in a wide range of cases by showing a matching lower bound.

The discussion proceeds as follows. In Section 5.2 we describe the algorithm. In Section 5.3 we prove its correctness. The complexity analysis is shown in Section 5.4. Concluding remarks are given in Section 5.5.

5.2. The Selection Algorithm

W.l.o.g. assume that all the elements in the network are distinct. If not, replace each element ξ in P_i with the triple $\langle \xi, i, j_\xi \rangle$, where j_ξ is a unique index within P_i, and use lexicographical order among the triples.

We use the following notation. The number of elements in processor P_i is n_i. The number of remaining candidates for selection at each stage of the algorithm is m. The number of remaining candidates in processor P_i is m_i. Initially, $m=n$ and $m_i=n_i$. The threshold value for the number of remaining candidates is denoted m^*. The rank of the element to be selected is d.

The algorithm consists of two phases: a filtering phase, followed by a termination phase. In the filtering phase, the number of candidates for selection is iteratively reduced until it gets below m^*. In the termination phase, the remaining candidates are collected to one processor; that processor completes the selection locally and broadcasts the result.

5.2.1. The Filtering Phase

A typical iteration of the filtering phase proceeds as follows. Using some efficient sequential selection algorithm (e.g., [Blum73]), each processor P_i locally computes the median of its m_i remaining candidates. Let us denote this value as med_i. If there are no remaining candidates in P_i, med_i is given a dummy value. Using *COLUMNSORT* (Chapter 3), the pairs $\langle med_i , m_i \rangle$ are sorted in descending order of the first coordinate. We denote the pair located at processor P_i after the sorting as $\langle med_i' , m_i' \rangle$, to distinguish it from the original pair of that processor.

Using the partial-sums algorithm (Chapter 3), the processors compute m and the partial sums $m_i'^{+}$. Let $m_l'^{+}$ be the smallest partial sum such that $m_l'^{+} \geq \frac{m}{2}$. We denote the corresponding median, med_l', as $med_{1/2}$. Intuitively, $med_{1/2}$ is chosen so that sufficiently many candidates are larger than it, and sufficiently many are smaller. P_l can identify $med_{1/2}$ by comparing $m_l'^{+}$ with the next-smaller partial sum. It then broadcasts $med_{1/2}$ to the other processors, which, using the partial-sums algorithm, calculate the total number of candidates that are greater or equal to $med_{1/2}$. Denote this number $m_{1/2}$. There are three cases.

Case 1. $m_{1/2}=d$

The selected element is $med_{1/2}$; the algorithm terminates.

Case 2. $m_{1/2}>d$

The candidates smaller or equal to $med_{1/2}$ are purged from each processor, and m is set to $m_{1/2}-1$. If $m>m^*$, the next filtering iteration is started; otherwise, the termination phase is executed.

Case 3. $m_{½} < d$

The candidates greater or equal to $med_{½}$ are purged from each processor, m is set to $m - m_{½}$, and d is set to $d - m_{½}$. Similar to case 2, if $m > m^*$ the next filtering iteration is started; otherwise, the termination phase is executed.

5.2.2. The Termination Phase

In the termination phase, all the remaining candidates are collected to processor P_1, which then selects the element of rank d and broadcasts it to the other processors.

The difficulty is that not every processor necessarily has remaining candidates. In order to implement the element collection efficiently, P_p must avoid polling processors which do not have candidates.

Let us call the processors which have remaining candidates *active* processors. The idea is to link all the active processors in a linked list. The elements are then sent to P_p in order of the linked list, as follows. When the current active processor has finished sending its elements, it notifies P_p of the identity of the next active processor on the list. All active processors listen to all the transmissions, thereby synchronizing with each other when to start sending elements.

The linked list of active processors is formed in the following way. Let $a_i = i$ if P_i is active, and otherwise let $a_i = 0$. A partial-sums computation is performed on the values a_i, using the operation $\oplus$ = "max". It can be seen that if P_i is active, then $a_{i-1}^{\oplus}$ points to the nearest active processor to P_i's left (with respect to the leaves in the tree). Also, $a_p^{\oplus}$ points to the rightmost active processor, which is the first in the list.

5.3. Correctness Proof

We now show that the selection algorithm works correctly. Assume inductively that at the beginning of the current filtering iteration the element to be selected has not been purged, and that d is the correct rank of this element among the remaining candidates. This is clearly true at the beginning of the algorithm.

In case 1, the number of candidates greater or equal to $med_{1/2}$ is d. Since w.l.o.g. we assume that all elements are distinct, the decision to select $med_{1/2}$ is correct. In case 2, since $m_{1/2}>d$, the element we are looking for is greater than $med_{1/2}$. Thus, all candidates smaller or equal to $med_{1/2}$ can be purged. In case 3, a similar argument applies, except that since the $m_{1/2}$ candidates being purged are greater than the selected element, the rank d needs to be lowered by the same amount. Since at least one candidate is purged in each instance of case 2 or 3, m becomes smaller and smaller, and eventually the algorithm either terminates in case 1 or reaches the termination phase, where the correct element is selected locally at P_p.

5.4. Complexity Analysis

In analyzing the complexity of the algorithm, we need to determine the cost of a filtering iteration and the termination phase, and calculate the number of filtering iterations.

Each filtering iteration involves the following: (1) sorting the pairs $\langle med_i', m_i' \rangle$; (2) computing $med_{1/2}$; and (3) computing $m_{1/2}$. Using the costs of *COLUMNSORT* (assuming $p \geq k^{1+\epsilon}$) and the partial-sums algorithm, the complexity of each iteration is $O(\frac{p}{k}+\log k)=O(\frac{p}{k})$ cycles and $O(p)$ messages.

The analysis of the number of filtering iterations is illustrated in Figure 5.1, which captures the situation at the beginning of a typical iteration. Let M_i denote the list of remaining candidates in P_i. The lists are shown in the order $M_1', M_2', \ldots, M_p'$ from left to right, where M_i' is the list that corresponds to the pair $\langle med_i', m_i' \rangle$. The elements in each list are shown in descending order from top to bottom. Since the lists are given in descending order of the medians, it can be seen that for any list M_i', half the candidates in M_i' and at least half the candidates in every list to the right of M_i' are smaller or equal to med_i'. Similarly, half the list M_i' and at least half of every list to the left of M_i' are greater or equal to med_i'. This is shown in Figure 5.1 by the encircled areas.

In particular, since $med_{\frac{1}{2}}(=med_l')$ was chosen such that $m_l'^{+}$ is the smallest partial sum of candidates which is greater or equal to $\frac{m}{2}$, it can be seen that at least $\frac{m}{4}$ candidates are smaller or equal to $med_{\frac{1}{2}}$, and at least $\frac{m}{4}$ candidates are greater or equal to $med_{\frac{1}{2}}$. Consequently, in each instance of case 2 or 3 of the algorithm, at least one fourth of the remaining candidates are purged. Thus, $O(\log \frac{n}{m^*})$ iterations suffice in order to reduce the number of candidates below m^*.

The termination phase involves application of the partial-sums algorithm, and the transfer of m^* elements. This amounts to $O(\frac{p}{k}+m^*)$ cycles and $O(p+m^*)$ messages.

The total complexity of the selection algorithm is $O(m^*+\frac{p}{k}\log \frac{n}{m^*})$ cycles and $O(m^*+p \log \frac{n}{m^*})$ messages. Choosing $m^*=\frac{p}{k}$, the complexity becomes

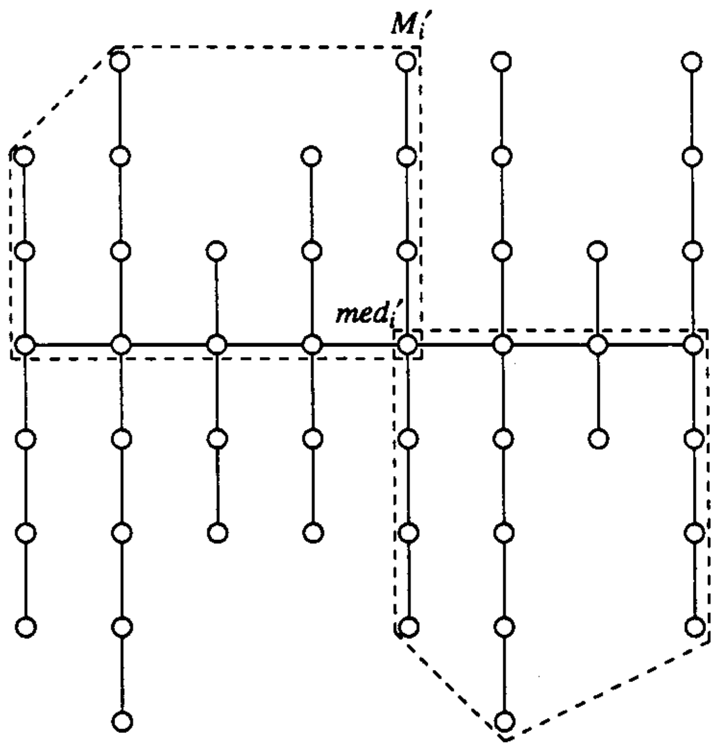


Figure 5.1. An Iteration of the Filtering Phase

$O(\frac{p}{k}\log\frac{kn}{p})$ cycles and $O(p\log\frac{kn}{p})$ messages.

We now show a simple modification that improves the performance of the algorithm for small ranks $d<\frac{n}{p}$. Clearly, only the d largest elements in each processor have the potential of becoming the selected element. Therefore, at the beginning of the algorithm each processor can eliminate from candidacy all but d of its elements. In other words, the initial number of candidates at P_i is $m_i=\min\{n_i, d\}$. The total number of candidates is at most pd, which is less than n. The improved complexity of the algorithm is $O(\frac{p}{k}\log dk)$ cycles and $O(p\log dk)$ messages.

Notice that because of the use of *COLUMNSORT* in the filtering phase, we have assumed in the analysis that $p \geq k^{1+\varepsilon}$. When this is not the case, we can replace *COLUMNSORT* with emulation of the AKS sorting network [Ajta83], thereby increasing the complexity by a factor of $\log p$ in both messages and cycles.

5.5. Lower Bounds

We now establish lower bounds on the complexity of selection under uniform communication cost. We first discuss finding the median, then generalize to arbitrary ranks. The bounds are based on an adversary argument adapted from [Fred83], and apply only to comparison-based algorithms.

Theorem 5.1. Selecting the median of n elements requires $\Omega(\sum_{i=1}^{p} \log 2n_i - \log 2n_{\max})$ messages.

Corollary 5.1. Selecting the median of n elements requires $\Omega(\frac{1}{k} \sum_{i=1}^{p} \log 2n_i - \frac{\log 2n_{\max}}{k})$ cycles. ■

Proof of Theorem 5.1. We devise an adversary that, given the cardinalities n_i and a selection algorithm, generates an input distribution such that the algorithm requires $\Omega(\sum_{i=1}^{p} \log 2n_i - \log 2n_{\max})$ messages when executed on that input.

The adversary is free to make each element arbitrary large or small, as long as the relative order in each processor is maintained consistently. Initially, none of the elements has a fixed magnitude. The adversary follows the execution of the algorithm, fixing the magnitude of elements as the algorithm proceeds. Elements not yet fixed are candidates for the median. Fixed elements are made either "very small" or "very large," in the sense of being smaller or larger than all the remaining candidates in the network. By keeping an equal number of very small and very large elements at all times, the adversary excludes such elements from being selected. Clearly, the algorithm cannot terminate before the number of candidates is reduced to one. Total order is maintained among the fixed elements of all processors by making each new very small element (very large element, respectively) larger (smaller) than all the existing very small (very large) elements.

Let $n_{i_1} \geq n_{i_2} \geq \cdots \geq n_{i_p}$ be a nonincreasing order among the n_i's, and assume w.l.o.g. that p is even. The adversary divides the processors into disjoint pairs $\langle P_{i_1}, P_{i_2} \rangle, \langle P_{i_3}, P_{i_4} \rangle, \ldots, \langle P_{i_{p-1}}, P_{i_p} \rangle$. Denote a typical pair $\langle P_a, P_b \rangle$, and assume w.l.o.g. that $n_a - n_b$ is even. The elements are initialized as follows. All the elements in P_b are made candidates. The $\frac{n_a - n_b}{2}$ smallest elements in P_a are made very small, and the $\frac{n_a - n_b}{2}$ largest elements in P_a are made be very large. The remaining n_b elements in P_a are made candidates. Thus, both processors of each pair have the same number of candidates.

Whenever a message is sent which contains a candidate of P_a that is larger or equal (smaller, respectively) than the median of the candidates in P_a, the adversary fixes that candidate and all those larger (smaller) than it in P_a to be very large (very small), and an equal number of candidates in P_b to be very small (very large). From that point on, these elements are no longer candidates. When the message contains a candidate of P_b, the same action with the roles of P_a and P_b reversed is taken. No action is taken by the adversary if the message does not contain a candidate. Concurrent messages (in the same cycle) are handled in arbitrary order.

Let $2q$ be the number of candidates in a given pair of processors immediately before a message containing a candidate of that pair is sent. It can be seen that at most $q+1$ candidates, all of them in the given pair, are fixed by the adversary as a result of that message. Thus, at least $\sum_{j=1}^{\lfloor \frac{p}{2} \rfloor} \log 2n_{i_{2j}} \geq \frac{1}{2}\sum_{j=2}^{p} \log 2n_{i_j} = \Omega(\sum_{i=1}^{p} \log 2n_i - \log 2n_{\max})$ messages are needed to reduce the number of candidates in the network to one. ■

The lower bounds of Theorem 5.1 and Corollary 5.1 can be generalized for an arbitrary rank d, as follows.

Theorem 5.2. Let d be an integer, $p \leq d \leq \lceil \frac{n}{2} \rceil$. Let s be the number of processors for which $n_i \geq \frac{d}{p}$. Let $n_{i_1} \geq n_{i_2} \geq \cdots \geq n_{i_p}$ be a nonincreasing order among the n_i's. Selecting the d'th largest element requires $\Omega((s-1)\log \frac{2d}{p} + \sum_{j=s+1}^{p} \log 2n_{i_j})$ messages.

Corollary 5.2. Selecting the d'th largest element, $p \leq d \leq \lceil \frac{n}{2} \rceil$, requires $\Omega(\frac{s-1}{k}\log \frac{2d}{p} + \frac{1}{k}\sum_{j=s+1}^{p} \log 2n_{i_j})$ cycles. ■

Proof of Theorem 5.2. We use the same adversary argument as in Theorem 5.1. The only difference is in the initialization of candidates and fixed elements. In pairs of processors $\langle P_a, P_b \rangle$ where $b \geq s+1$, all the n_b elements in P_b and an equal number of elements in P_a are made candidates. In the remaining pairs, candidates are chosen so that: (1) each processor will have at least $\frac{d}{p}$ candidates; (2) the total number of candidates in the network will not exceed $2d$; and (3) both processors of a pair will have the same number of candidates. Let $m=2q$ denote the initial number of candidates in the network. Among the remaining $n-2q$ elements, $d-q$ elements are made very large, and the rest very small. With this setup, the selection problem reduces to finding the median of the $2q$ candidates. Let m_i denote the initial number of candidates in P_i. Similar to Theorem 5.1, the number of messages required is at least $\sum_{j=1}^{\lfloor \frac{p}{2} \rfloor} \log 2m_{i_{2j}} \geq \frac{1}{2} \sum_{j=2}^{p} \log 2m_{i_j} = \Omega((s-1)\log \frac{2d}{p} + \sum_{j=s+1}^{p} \log 2n_{i_j})$. ■

The following corollary shows that the lower bounds are tight, and that the selection algorithm is optimal in a wide range of cases.

Corollary 5.3. Let $0<\varepsilon\leq 1$ be a constant such that $p \geq k^{1+\varepsilon}$, $n \geq \frac{pk}{\varepsilon^2}$, and $\frac{\varepsilon n}{2} \leq d \leq \lceil \frac{n}{2} \rceil$. Also, let the distribution of the input be such that for at least $\frac{\varepsilon p}{2}+1$ processors $n_i \geq \frac{d}{p}$. The complexity of selecting the d'th largest element in an MCB(p, k) is $\Theta(\frac{p}{k}\log \frac{kn}{p})$ cycles and $\Theta(p \log \frac{kn}{p})$ messages.

Proof. $n \geq \frac{pk}{\varepsilon^2}$ implies $\log \frac{2d}{p} \geq \log \frac{\varepsilon n}{p} \geq \frac{1}{2} \log \frac{kn}{p}$. Using $s = \lceil \frac{\varepsilon p}{2} \rceil + 1$, the lower bounds of Theorem 5.2 and Corollary 5.2 reduce to $\Omega(p \log \frac{kn}{p})$ and $\Omega(\frac{p}{k}\log \frac{kn}{p})$, respectively. The matching upper bounds are achieved by the algorithm in Section 5.4. ■

It should be noted that Corollary 5.3 applies both in the general and in the restricted MCB. Also notice that the tight bounds on cycles and on messages are simultaneous.

5.6. Concluding Remarks

Rotem, Santoro and Sidney [Rote83] study the selection problem in the framework of the Shout-Echo model. In this model, a communication cycle consists of a broadcast message from a single processor, followed by replies from all other processors. The selection algorithm given in [Rote83] has a complexity of $O(\log p \log(\min\{\frac{n}{p}, d\}))$ cycles, where d is the rank of the element to be selected. By implementing our selection method in the Shout-Echo model, we are able to improve this upper bound by a factor of $\log p$, which can be shown optimal (see [Marb85a]).

CHAPTER 6

PERMUTATION ALGORITHMS

In this chapter we address the problem of performing permutations among the processors in the MCB. We present a simple and efficient algorithm for the restricted MCB, and establish a matching lower bound.

6.1. Introduction

Consider the following problem. Each processor P_i needs to send a message to a destination processor P_{d_i}. It is given that no two destinations are the same; in other words, the list of destinations indexes $[d_1 , d_2 , \ldots , d_p]$ is a permutation of the list $[1, 2, \ldots , p]$. The task is to deliver the messages to their destinations efficiently.

In the general MCB(p, p) it is straightforward to permute the messages in a single cycle (assuming w.l.o.g. that each message is short enough to be transmitted in one cycle): P_i writes its message on channel C_{d_i}, and P_{d_i} reads the channel. In other words, channel C_j serves as the "mailbox" of processor P_j. In the case of a general MCB(p, k) where $k<p$, a similar approach accomplishes the task in $\frac{p}{k}$ cycles, with each channel serving as mailbox for $\frac{p}{k}$ processors.

Yet, the above method fails on the restricted MCB. Since each processor can write only on one predetermined channel, a message cannot be delivered to an arbitrary destination processor in one step unless the latter knows the identity of the sender, so that it can read the appropriate channel. In other words, in the restricted MCB, unlike the general MCB, there is no notion of mailboxes.

The obvious way to permute the messages in the restricted MCB is by sorting, namely the messages are tagged with their destination, then sorted. However, performing a permutation seems an "easier" task than sorting, since the destination of each item is already determined in advance. Moreover, it has been shown in Chapter 3 that when $p \leq k^{1+\varepsilon}$, sorting entails the use of AKS emulation, which is impractical due to the large constants involved. It is our goal in this chapter to develop an efficient method to perform permutations without relying on sorting.

As mentioned, the underlying difficulty is that a destination processor is initially unaware of the corresponding source processor (i.e., the sender). Our approach is to have each processor identify its source. Once this has been accomplished, messages can be delivered directly to their destination over the channel associated with the source. We call the task of identifying the sources the *permutation problem*. In a way, the problem is a generalization of the (single-bit) identification problem discussed in Chapter 2.

A formal definition of the permutation problem is the following. Each processor P_i has an index d_i, called the *destination index*, such that $d_i \in \{1, 2, \ldots, p\}$, and $d_i \neq d_j$ if and only if $i \neq j$. The task of processor P_j is to compute an index s_j, called the *source index*, such that $d_{s_j} = j$.

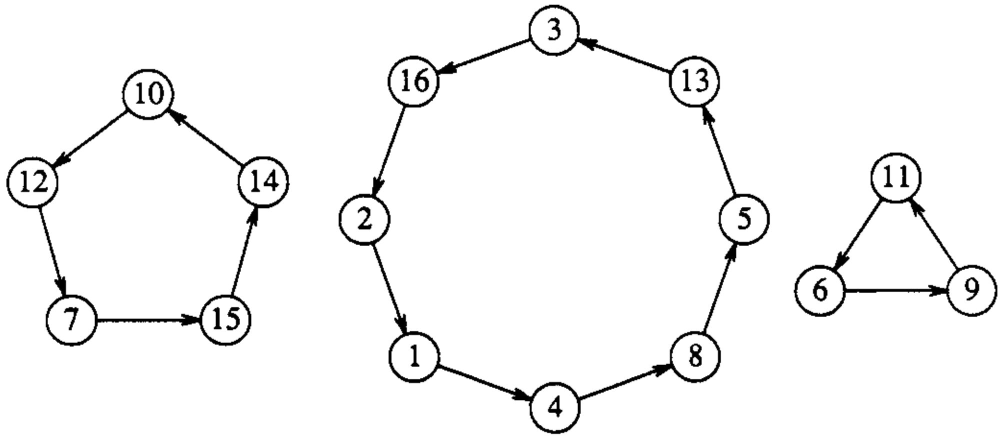
We present a non-oblivious algorithm for the permutation problem that runs in $O(\log p)$ cycles and $O(p \log p)$ messages on a restricted MCB(p, p) with uniform communication. By showing that the problem is ($p-1$)-sensitive, we prove that the algorithm is optimal in the number of cycles. The algorithm can be generalized to arbitrary MCB(p, k), $k \le p$, achieving a complexity of $O(\frac{p}{k}\log p)$ cycles and $O(p\log p)$ messages. When $p \le k^{1+\varepsilon}$, $0<\varepsilon \ll 1$, this complexity matches sorting, yet our algorithm is considerably simpler. We then prove that $\Omega(\frac{p}{k})$ cycles and $\Omega(p)$ messages are lower bounds for permutation. This improves the lower bound derived from the $(p-1)$-sensitivity of the problem when p is sufficiently larger than k. We conclude that for $p \ge k^{1+\varepsilon}$, the permutation problem can be solved optimally via sorting, using the *COLUMNSORT* algorithm of Chapter 3.

The discussion proceeds as follows. In Section 6.2 we present the algorithm. Section 6.3 shows the lower bounds. Concluding remarks are given in Section 6.4.

6.2. The Permutation Algorithm

Consider the following directed graph, induced by an instance $I=[d_1, d_2, \ldots, d_p]$ of the permutation problem. There are p nodes, labeled 1 through p. Each node i is connected by an outgoing edge to node d_i. It is immediate to see that each component of the graph is a directed ring (see Figure 6.1 for an example). Associating processor P_i with node i, computing s_i is equivalent to identifying the predecessor of P_i in the ring. In the remainder of the discussion we will use the terms node (in the context of the rings) and processor interchangeably.

Since initially each processor knows only its successor in the ring, the ring can be "traversed" only in the outgogoing direction of the edges. In order to identify its predecessor, a processor needs to go around the entire ring.



I=[4, 1, 16, 8, 13, 9, 15, 5, 11, 12, 6, 7, 3, 10, 14, 2]

Figure 6.1. Ring Graph Induced by a Permutation

A straightforward but slow method of traversing the ring is to go from node to node along the edges. This is implemented as follows. Each processor repeatedly broadcasts its destination index, which serves as a "pointer" to the next node in the ring. A processor first reads the channel of its successor, then follows the pointer and reads the next channel, etc., until it encounters the pointer back to itself. Depending on the size of the ring, this method could take up to $O(p)$ cycles and $O(p^2)$ messages.

A faster way to cover the distance is to use what is called the *doubling technique* [Wyll79]. The idea is to double the distance traversed in each step. Suppose a distance of r steps along the ring has already been covered. Each processor now broadcasts the pointer to the node at distance r from itself. In the next cycle, the processor reading that pointer can proceed directly to the node at distance $2r$. The algorithm presented in this section is based on this approach.

There is a problem with the approach, namely that the ring is of arbitrary size, i.e., the predecessor of a node can be at any distance, whereas doubling "visits" only the nodes at distance 2^t, $t=0, 1, 2, \cdots$. We will show later that the doubling technique can be refined to cover exactly the distance to the predecessor. First, however, we need to determine that distance. This can be accomplished by counting the number of nodes in the ring. Yet, the latter task is not trivial. The difficulty lies in the inherent symmetry of the ring, which does not provide a fixed point from which to start the counting. Such a point could, for example, be the node of minimum id in the ring. Fortunately, the standard doubling technique can be used to find the minimum node.

Based on these observations, our solution to the permutation problem consists of three phases: (1) finding the minimum node in each ring; (2) finding the size of each ring; and (3) identifying the predecessors. We describe the algorithm for MCB(p, p), then show how it can be generalizeed to arbitrary $k \le p$.

6.2.1. Finding the Minimum Node in the Ring

Although the exact size of each ring is not known, p is obviously an upper bound. The doubling technique is therefore guaranteed to cover the entire ring (perhaps more than once) in $\lceil \log p \rceil$ steps.

Let $M_i(t)$ denote the minimum id among the nodes in the ring which are at distance 2^t-1 or less from node i (in the outgoing direction of the links). Clearly, $M_i(\lceil \log p \rceil)$ is the minimum id in the ring. Initially, however, each node i knows only the value $M_i(0)=i$.

The values $M_i(t)$, $t \geq 1$ are computed recursively in $\lceil \log p \rceil$ steps using the doubling technique. Let $N_i(t)$ be the node at distance 2^t from node i. Initially, $N_i(0)=d_i$, where d_i is the destination index of processor P_i. In step t, $1 \leq t \leq \lceil \log p \rceil$, each P_i broadcasts the values $N_i(t-1)$ and $M_i(t-1)$, and reads channel $C_{N_i(t-1)}$. It then sets $N_i(t):=N_{N_i(t-1)}(t-1)$ and $M_i(t):=\min\{M_i(t-1), M_{N_i(t-1)}(t-1)\}$.

6.2.2. Finding the Size of the Ring

Once the minimum node in the ring has been identified, the ring can be "cut" between the minimum node and its predecessor, to form a linearly linked list. The goal is then to determine the length of the linked list.

We use the doubling technique in the following manner. Let i_f and i_l denote the first and last node, respectively, in a given list. Notice that both P_{i_f} and P_{i_l} can identify themselves after the first phase.

Each processor P_i has a counter L_i. The value of the counter after step t of the doubling is denoted $L_i(t)$. Initially, $L_{i_l}(0)=0$, and for all $i \neq i_l$ in the given list, $L_i(0)=1$. Let the pointers $N_i(t)$ be defined as in the first phase, except that $N_{i_l}(t)=i_l$ for all t (i.e., the last processor in the linked list points to itself at all times).

The computation proceeds as follows. In step t, $1 \leq t \leq \lceil \log p \rceil$, each processor P_i broadcasts $N_i(t-1)$ and $L_i(t-1)$, and reads channel $C_{N_i(t-1)}$. It then sets $N_i(t):=N_{N_i(t-1)}(t-1)$ and $L_i(t):=L_i(t-1)+L_{N_i(t-1)}(t-1)$.

It can be seen that $L_i(t)$ equals the number of links covered in t steps of doubling beginning at node i. Since P_{i_l} points to itself, the doubling becomes stagnant when the end of the list is reached. It can also be verified that $L_{i_l}(t)=0$ for all t.

Thus, $L_i(\lceil \log p \rceil)$ gives the distance from node i to the end of the list. The length of the list, in number of links, is $L=L_{i_f}(\lceil \log p \rceil)$. This is also the distance from each node to its predecessor. P_{i_f} broadcasts L to all the processors in the ring.

6.2.3. Identifying the Predecessors

As noted earlier, the standard doubling technique cannot be used to visit the predecessor unless the distance is a power of 2. We now show how to refine the doubling technique to cover an arbitrary distance L.

We assume w.l.o.g. that P_i remembers the values $N_i(1), N_i(2), \ldots, N_i(\lceil \log p \rceil)$ from the first phase. Let $b_{\lfloor \log L \rfloor} b_{\lfloor \log L \rfloor -1} \cdots b_1 b_0$ be the binary representation of L, where b_0 is the least significant bit. We define $N'_i(t)=N_i(t)$ if $b_t=1$, and otherwise $N'_i(t)=i$.

The id of the node at distance $b_t b_{t-1} \cdots b_0$ from node i, denoted $D_i(t)$, can be computed using the following recursive formula.

1. $D_i(0)=N'_i(0)$.
2. $D_i(t)=N'_{D_i(t-1)}(t)$.

We are interested in $D_i(\lfloor \log L \rfloor)$, which is the id of the predecessor of P_i. Intuitively, the distance to the predecessor is covered in "hops" of size $b_t 2^t$, $t=0, 1, \ldots, \lfloor \log L \rfloor$. This gives the correct result since $L=\sum_{t=0}^{\lfloor \log L \rfloor} b_t 2^t$.

The computation is implemented as follows. In step $1 \le t \le \lfloor \log L \rfloor$, processor P_i broadcasts $N'_i(t)$ and reads channel $D_i(t-1)$. It then sets $D_i(t)=N'_{D_i(t-1)}(t)$.

6.2.4. Complexity

Each of the three phases uses the doubling technique to traverse a ring (or list) of size at most p. Hence, the complexity of the algorithm is $O(\log p)$ cycles and $O(p)$ messages. In Section 6.3 we show that this is optimal.

The algorithm can be generalized to arbitrary MCB(p, k), $k \leq p$, by emulating each cycle of the original algorithm using $\frac{p}{k}$ cycles. This results in a complexity of $O(\frac{p}{k}\log p)$ cycles and $O(p\log p)$ messages.

when $p \leq k^{1+\varepsilon}$, $0<\varepsilon \ll 1$, our algorithm matches the complexity of sorting (see Chapter 3). Yet, our solution is considerably simpler and more practical, since the sorting algorithm uses AKS emulation. This supports the observation made earlier, namely that the permutation problem seems easier than sorting.

6.3. Lower Bounds

The permutation problem is $(p-1)$-sensitive. To see this, let S be a subset of $p-2$ processors, let P_j be a processor in S, and let P_{i_1} and P_{i_2} be the two processors not in S. Consider an input instance I_1 in which $d_{i_1}=j$, and an instance I_2 that is derived from I_1 by switching d_{i_1} and d_{i_2}. That is, in I_2, $d_{i_2}=j$. Clearly, S, I_1 and I_2 satisfy the definition of $(p-1)$-sensitivity (see Chapter 2).

Following Theorem 2.1, solving the permutation problem on the restricted MCB requires $\frac{\log(p-1)}{\log(\frac{p}{k}+1)}$ cycles. In particular, when $p=k$, at least $\log(p-1)$ cycles are required.

Corollary 6.1. The permutation algorithm for MCB(p, p) described in Section 6.2 is optimal in the number of cycles. ■

We now prove that $\lfloor \frac{p}{4} \rfloor$ messages and $\lfloor \frac{p}{4k} \rfloor$ cycles are lower bounds for permutation. This improves the lower bound derived from the $(p-1)$-sensitivity of the problem when p is sufficiently larger than k.

Theorem 6.1. The permutation problem requires at least $\lfloor \frac{p}{4} \rfloor$ messages.

Corollary 6.2. The permutation problem requires at least $\lfloor \frac{p}{4k} \rfloor$ cycles. ■

Proof of Theorem 6.1. We have seen in Chapter 2 that information can sometimes be gained implicitly from the fact that a certain processor does not "talk" (i.e. does not write on the channel) in a given cycle. The idea here is to show an instance of the permutation problem which forces $\lfloor \frac{p}{4} \rfloor$ processors to talk at least once.

Let I be the circular shift permutation, i.e., $d_i = (i \bmod p)+1$. Let I' be the permutation obtained from I by switching each pair d_{2j-1} and d_{2j}, $1 \le j \le \lfloor \frac{p}{2} \rfloor$. Thus, $I=[2, 3, 4, 5, \ldots, p, 1]$, and $I'=[3, 2, 5, 4, \cdots]$.

Suppose that neither P_1 nor P_2 talks during the execution of the algorithm on input I. Assuming the algorithm is correct, P_3 evaluates its source index as $s_3=2$. Now, consider the execution of the same algorithm on input I'. The claim is that either P_1 or P_2 must talk. To see this, assume that neither talks. Then, processor P_3 cannot distinguish instance I' from I, so it evaluates $s_3=2$, which is incorrect (the correct value is $s_3=1$).

In other words, regardless of the algorithm, given the inputs I and I' defined above, either P_1 or P_2 must talk during the computation for at least one of the two inputs. It can be seen that a similar argument holds for each of the $\lfloor \frac{p}{2} \rfloor$ disjoint processor pairs $\langle P_1, P_2 \rangle, \langle P_3, P_4 \rangle, \cdots$. Hence, the combined number of messages required for the two instances is at least $\lfloor \frac{p}{2} \rfloor$, which means that one of the instances entails $\left\lceil \frac{\lfloor \frac{p}{2} \rfloor}{2} \right\rceil \geq \lfloor \frac{p}{4} \rfloor$ messages. ■

Corollary 6.3. When $p \geq k^{1+\varepsilon}$, $0<\varepsilon \ll 1$, the complexity of the permutation problem on an MCB(p, k) is $\Theta(p)$ messages and $\Theta(\frac{p}{k})$ cycles.

Proof. The lower bounds are given by Theorem 6.1 and Corollary 6.2. The matching upper bounds can be achieved via sorting, using the *COLUMNSORT* algorithm of Chapter 3. ■

Corollary 6.3 applies both in the general and in the restricted MCB. In the general model, the upper bound is also achieved by the straightforward method discussed in Section 6.1.

6.4. Concluding Remarks

We have developed a permutation algorithm that runs in $O(\frac{p}{k}\log p)$ cycles. This is optimal when $p=k$, and matches the complexity of the obvious solution using sorting in the range $p \leq k^{1+\varepsilon}$. On the other hand, in the range $p > k^{1+\varepsilon}$, sorting achieves optimal complexity of $O(\frac{p}{k})$ cycles, whereas our algorithm is inferior. It remains open whether it is possible to perform permutations optimally in the range

$p > k^{1+\varepsilon}$ without relying on sorting.

There are two generalizations of the permutation problem which seem more difficult. The first is the *partial permutation problem.* It is similar to the (full) permutation problem, except that some processors do not have a destination index. The algorithm we have presented cannot be applied here, since partial permutations do not necessarily induce a ring graph.

The second generalization is the *emulation problem.* Every processor P_i has a *reading index* r_i, $1 \le r_i \le k$. Also, k processors P_j each have a *writing index* w_j, $1 \le w_j \le k$, such that no two writing indexes are the same. The task is to deliver a message from P_j to P_i if and only if $w_j = r_i$. An algorithm solving this problem on the restricted MCB would constitute a mechanism to emulate, cycle by cycle, arbitrary computations of the general MCB. It is interesting to investigate the complexity of such emulation.

CHAPTER 7
DIRECTIONS FOR FUTURE RESEARCH

This chapter suggests directions for future research in the area of distributed algorithms with broadcast communication. The discussion covers two main topics: network models for broadcast communication; and design and analysis of broadcast algorithms.

7.1. Network Models for Broadcast Communication

7.1.1. Variants of the MCB Model

We have distinguished different variants of the MCB model by changing some of the assumptions. With respect to channel access, there is the general MCB, where each processor can read and write any channel, and the restricted MCB, where each processor can write only on one predetermined channel. With respect to communication cost, there is uniform cost, where the transmission of an atomic datum is assumed to take one cycle, and logarithmic cost, where each bit to be transmitted requires a separate cycle. Also, we have defined oblivious computations, where the processors that access each channel during each cycle are predetermined, independent of the input.

Chapter 2 has presented several separation results regarding the computational power of the different variants of the model. An interesting open problem is whether the general and the restricted MCB are equally powerful for oblivious computation. As discussed in Chapter 2, we conjecture that the models are indeed equivalent. Proving (or disproving) the conjecture seems to be a difficult task, which may require new proof techniques.

Also of interest is to investigate other enhancements to the MCB model. For example, concurrent write-access to the channels, similar to the CRCW shared memory model [Fich84] or the single-channel broadcast models in [Dech86, Levi82]. Another possible enhancement is to allow a processor to read several channels simultaneously in each cycle.

7.1.2. Alternative Interconnection Structures

In the MCB model, every processor is connected to every channel. While this provides increased flexibility in communication, it is also expensive in terms of the power requirements of the communication devices. Specifically, the more receivers are attached to each channel, the more powerful the transmitters that are required. An alterative interconnection structure, where fewer processors have read-access to each channel, is desirable. Such an interconnection should, nevertheless, exhibit the following two properties of the MCB structure: (1) any two processors should be able to communicate with each other in a constant number of "hops" over the channels; and (2) the number of channels needed to achieve the first requirement should be small.

One possible interconnection is illustrated in Figure 7.1. The processors are organized in a square grid. Each row and each column of processors is connected by a broadcast channel. Given p processors, $2p^{\frac{1}{2}}$ channels are needed, and the number of processors connected to each channel is $p^{\frac{1}{2}}$. Moreover, each processor is con-

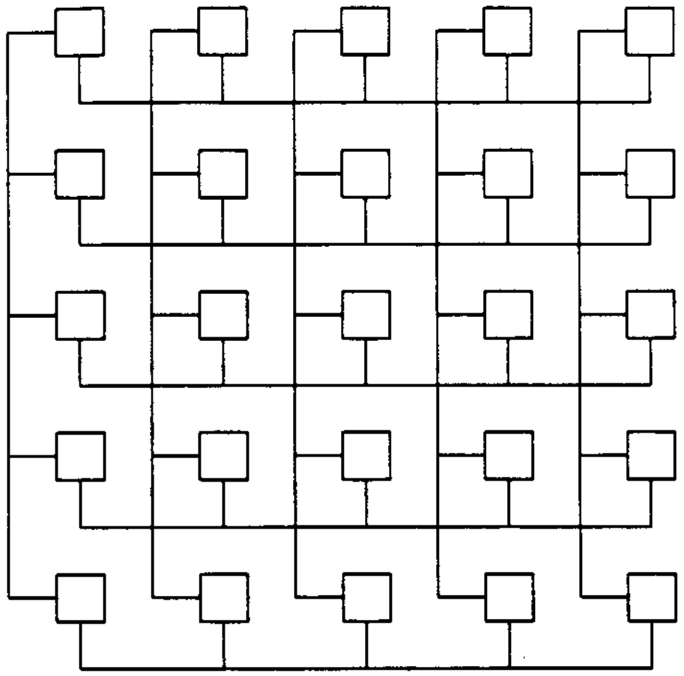


Figure 7.1. Grid Broadcast Interconnection

nected to only two channels, which makes the total number of connections $2p$, compared with pk in the MCB model. Also, it can be seen that any pair of processors can communicate in two hops. The approach can be generalized to three dimensions in the natural way.

A variation on this structure, which accommodates "private" channels, similar to a restricted $MCB(p, p)$, is the following. The processors are organized in a square grid, as before. There are p channels, each of which is dedicated to a different processor for writing. The processors in the row and column that intersect at a given processor have read-access to the channel written by that processor. The number of processors connected to each channel is thus $2p^{1/2}$. Again, each pair of processors can communicate in two hops.

Another possible organization, illustrated in Figure 7.2, is called *Selective Broadcast Interconnection (SBI)* [Marh85]. This is a parametrized family of interconnections, with the following parameters. Each processor can read k_r channels and write k_w channels. The total number of channels is $k_r \cdot k_w$. Let us number the channels using pairs (i, j), where $1 \le i \le k_r$ and $1 \le j \le k_w$. The processors are divided into k_r equal *writing groups*, and k_w equal *reading groups*. The processors in writing group i have write-access to the channels numbered $(i, *)$, and the processors in reading group j have read-access to the channels numbered $(*, j)$. In Figure 7.2, the case $p=12$, $k_r=3$, $k_w=2$ is illustrated; each processor is shown twice — once as a writer and once as a reader. It can be seen that a processor in writing group i can communicate with a processor in reading group j via channel (i, j), i.e., any two processors can communicate in one hop. The number of processors that are connected to each channel varies from channel to channel, but never exceeds $\frac{p}{k}(k_w+k_r)$. For $k_w=k_r$, this number is $\frac{2p}{k^{1/2}}$.

It is interesting to compare the computational power of these network models and others of similar type with the MCB.

7.2. Design and Analysis of Broadcast Algorithms

7.2.1. Open Questions in Sorting, Selection and Permutation

We have designed algorithms for sorting, selection and permutation. The optimality of these algorithms has been shown in a wide range of cases, but there remain unresolved questions with regard to tight bounds in the remaining cases. In

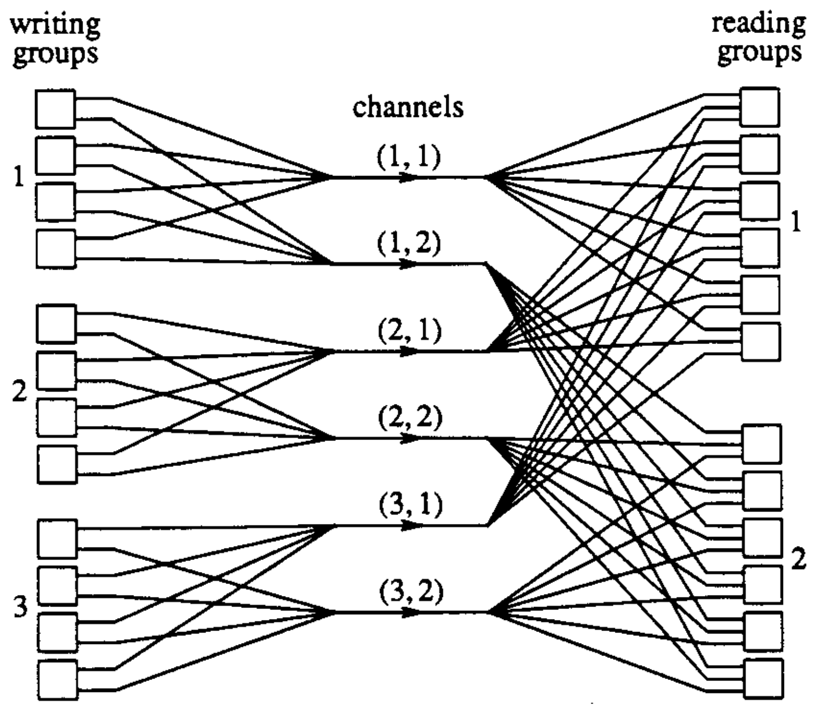


Figure 7.2. Selective Broadcast Interconnection

the following, we review some of these questions.

Algorithm *COLUMNSORT* in Chapter 3 is optimal for $n \geq k^{1+\varepsilon}$ when the distribution is even, and for $n \geq k^3$ when the distribution is arbitrary (uneven). It remains open to devise tight upper and lower bounds in the remaining ranges.

Sorting algorithm C in Chapter 4 is optimal for $m \geq p \log p$. On the other hand, when $m \leq p$, AKS emulation achieves a better performance of $O(m \log p)$ cycles. An open problem is to bridge the gap between the upper bound $O(p \log p)$ and the lower bound $\Omega(m)$ in the range $p < m < p \log p$. It also remains open whether $O(m \log p)$ is optimal for $m \leq p$.

The selection algorithm in Chapter 5 uses *COLUMNSORT* in the filtering phase, and we therefore assume that $p \geq k^{1+\varepsilon}$. When $p < k^{1+\varepsilon}$, we can replace *COLUMNSORT* with AKS emulation, thereby increasing the upper bound on selection by a factor of $\log p$ in both messages and cycles. It is open whether this complexity is optimal in the given range.

The permutation algorithm in Chapter 6 is optimal for $p=k$, and matches the complexity of the obvious solution via sorting in the range $p \leq k^{1+\varepsilon}$. On the other hand, in the range $p > k^{1+\varepsilon}$, sorting achieves optimal complexity of $O(\frac{p}{k})$ cycles, whereas the permutation algorithm is inferior. It remains open whether it is possible to perform permutations optimally in the range $p > k^{1+\varepsilon}$ without relying on sorting.

7.2.2. Additional Applications

The application algorithms developed in this dissertation demonstrate the practicality of the MCB model as a framework for distributed algorithm design, providing motivation to investigate additional problems in other domains.

One interesting domain is numeric algorithms. Examples of typical applications in this area which are of distributive nature include matrix operations (multiplication, determinant, etc.), dynamic programming, and recurrence equations. Such problems have been investigated in the context of other distributed computation models [Bert82, Boro82, Carl84].

Another important application domain is graph algorithms. It is not yet clear how to exploit broadcasting efficiently for this type of problems, the reason being that unlike point-to-point networks, the MCB model does not have an inherent graph structure. Several researchers, however, have successfully used a hybrid approach, in which a point-to-point network is augmented with a broadcast bus [Andr85, Bokh84, Stou83].

7.2.3. Characterization of Broadcast Protocols

Chandra, Furst and Lipton [Chan83] give a theoretical characterization of the class of multi-processor protocols that solve 0-1 predicates over a set of values distributed among the processors. The work examines the inherent communication complexity of such protocols in terms of the amount of information that needs to be known globally in the system. The model of communication being used is similar to a single-channel MCB where processors broadcast one bit at a time in a round-robin fashion. Yao [Yao79], and Papadimitriou and Sipster [Papa84] give a similar characterization of two-processor protocols. Generalizing these protocols to multiple broadcast channels is a difficult open problem. An MCB where each processor is capable of reading all the channels simultaneously could be used as framework for the analysis.

REFERENCES

[Afek85] Afek, Y. and E. Gafni, "Time and Message Bounds for Election in Synchronous and Asynchronous Complete Networks," in *Proceedings 4th ACM Symp. on Principles of Distributed Computing*, 1985, pp. 186-195.

[Aho83] Aho, A.V., J.E. Hopcroft, and J.D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.

[Ajta83] Ajtai, M., J. Komlos, and E. Szemeredi, "An $O(N \log N)$ Sorting Network," in *Proceedings 15th ACM Symp. on Theory of Computing*, 1983, pp. 1-9.

[Andr85] Andreatos, A., "Parallel Algorithms for the Strongly Regular Interconnection Networks," M.S. Thesis, Dept. of Electrical and Computer Engineering, Univ. of Massachusetts at Amherst, May 1985.

[Batc68] Batcher, K.E., "Sorting Networks and their Applications," in *Proceedings AFIPS Spring Joint Computer Conf.*, Vol. **32**, April 1968, pp. 307-314.

[Bert82] Bertsekas, D.P., "Distributed Dynamic Programming," *IEEE Trans. Automatic Control* **AC-27**, 3 (June 1982), pp. 610-616.

[Blum73] Blum, M., R.W. Floyd, V. Pratt, R.L. Rivest, and R.E. Tarjan, "Time Bounds for Selection," *JCSS* **7**, 4 (Aug. 1973), pp. 448-461.

[Bokh84] Bokhari, S.H., "Finding Maximum on an Array Processor with a Global Bus," *IEEE Trans. Computers* **C-33**, 2 (Feb. 1984), pp. 133-139.

[Boro82] Borodin, A., J. Von Zur Gathen, and J. Hopcroft, "Fast Parallel Matrix and GCD Computations," in *Proceedings 23rd IEEE Symp. on Foundations of Computer Science*, 1982, pp. 65-71.

[Carl84] Carlson, D.A. and B. Sugla, "Time and Processor Efficient Parallel Algorithms for Recurrence Equations and Related Problems," in *Proceedings 1984 Int. Conf. on Parallel Processing*, pp. 310-314.

[Chan83] Chandra, A.K., M.L. Furst, and R.J. Lipton, "Multi-Party Protocols," in *Proceedings 15th ACM Symp. on Theory of Computing*, 1983, pp. 94-99.

[Chou83] Choudhury, G.L. and S.S. Rappaport, "Diversity ALOHA - A Random Access Scheme for Satellite Communications," *IEEE Trans. Communications* **COM-31**, 3 (March 1983), pp. 450-457.

[Cook86] Cook, S., C. Dwork, and R. Reischuk, "Upper and Lower Bounds for Parallel Random Access Machines Without Simultaneous Writes," *SIAM J. Comput.* **15**, 1 (Feb. 1986), pp. 87-97.

[Dech86] Dechter, R. and L. Kleinrock, "Broadcast Communications and Distributed Algorithms," *IEEE Trans. Computers* **C-36**, 3 (March 1986), pp. 210-219.

[Dole82] Dolev, D., M. Klawe, and M. Rodeh, "An $O(n \log n)$ Unidirectional Distributed Algorithm for Extrema Finding in a Circle," *J. Algorithms* **3**, 3 (Sept. 1982), pp. 245-260.

[Fich84] Fich, F.E., P.L. Ragde, and A. Wigderson, "Relations between Concurrent-Write Models of Parallel Computation," in *Proc 3rd ACM Conf. on Principles of Distributed Computing*, 1984, pp. 179-189.

[Fred83] Frederickson, G.N., "Tradeoffs for Selection in Distributed Systems," in *Proceedings 2nd ACM Symp. on Principles of Distributed Computing*, 1983, pp. 154-160.

[Gott83] Gottlieb, A., R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. Computers* **C-32**, 2 (Feb. 1983), pp. 175-189.

[Gree82] Greenberg, A.G., "On the Time Complexity of Broadcast Communication Schemes," in *Proceedings 15th ACM Symp. on Theory of Computing*, 1982, pp. 354-364.

[Knut73] Knuth, D.E., *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison Wesley, Reading, MA, 1973.

[Kuma85] Kumar, V.K. and C.S. Raghavendra, "Array Processor with Multiple Broadcasting," in *Proceedings 12th Ann. Int. Symp. on Computer Architecture*, 1985, pp. 2-10.

[Land85] Landau, G.M., M.M. Yung, and Z. Galil, "Distributed Algorithms in Synchronous Broadcasting Networks," in *Proceedings 12th Int. Conf. on Automata, Languages and Programming*, 1985, pp. 363-372. To appear in *TCS*.

[Leig85] Leighton, T., "Tight Bounds on the Complexity of Parallel Sorting," *IEEE Trans. Computers* **C-34**, 4 (April 1985), pp. 344-354.

[Levi82] Levitan, S.P., "Algorithms for a Broadcast Protocol Multiprocessor," in *Proceedings 3rd Int. Conf. on Distributed Computing Systems*, 1982, pp. 666-671.

[Lin86] Lin, T.C. and D.I. Moldovan, "M^2 Mesh: An Augmented Mesh Architecture," in *Proceedings 1986 Int. Conf. on Parallel Processing*, pp. 308-315.

[Marb85a] Marberg, J.M. and E. Gafni, "An Optimal Shout-Echo Algorithm for Selection in Distributed Sets," in *Proceedings 23rd Ann. Allerton Conf. on Communication, Control, and Computing*, Univ. of Illinois at Urbana Champaign, 1985, pp. 283-291.

[Marb85b] Marberg, J.M. and E. Gafni, "Sorting and Selection in Multi-Channel Broadcast Networks," in *Proceedings 1985 Int. Conf. on Parallel Processing*, pp. 846-850.

[Marh85] Marhic, M.E., Y. Birk, and F.A. Tobagi, "Selective Broadcast Interconnection: a Novel Scheme for Fiber-Optic Local-Area Networks," *Optics Letters* **10**, 12 (Dec. 1985), pp. 629-631.

[Mars82a] Marsan, M.A., "Multichannel Local Area Networks," in *Proceedings IEEE Fall COMPCON*, 1982, pp. 493-502.

[Mars82b] Marsan, M.A., D. Roffinella, and A. Murru, "ALOHA and CSMA Protocols for Multichannel Broadcast Networks," in *Proceedings Canadian Communications and Energy Conf.*, 1982, pp. 375-378.

[Mars83] Marsan, M.A., P. Camarda, and D. Roffinella, "Throughput and Delay Characteristics of Multichannel CSMA-CD Protocols," in *Proceedings IEEE GLOBECOM*, 1983, pp. 1147-1151.

[McQu77] McQuillan, J.M. and D.C. Walden, "The ARPA Network Design Decisions," *Computer Networks* **1**, 5 (Aug. 1977), pp. 243-289.

[Metc76] Metcalfe, R.M. and D.R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *CACM* **19**, 7 (July 1976), pp. 395-403.

[Papa84] Papadimitriou, C.H. and M. Sipster, "Communication Complexity," *JCSS* **28**, 2 (April 1984), pp. 260-269.

[Pete84] Peterson, G.L., "Efficient Algorithms for Election in Meshes and Complete Networks," Tech. Rep. TR 140, Dept. of Computer Science, Univ. of Rochester, Rochester, NY, Aug. 1984.

[Reis86] Reischuk, K.R., "Parallel Machines and their Communication Theoretical Limits," in *Proceedings 3rd Annual Symp. on Theoretical Aspects of Computer Science*, 1986, pp. 359-368.

[Rote83] Rotem, D., N. Santoro, and J.B. Sidney, "A Shout-Echo Algorithm for Finding the Median of a Distributed Set," in *Proceedings 14th S.E. Conf. on Combinatorics, Graph Theory and Computing*, Boca Raton, FL, 1983, pp. 311-318.

[Sant82] Santoro, N. and J.B. Sidney, "Order Statistics on Distributed Sets," in *Proceedings 20th Ann. Allerton Conf. on Communication, Control and Computing*, Univ. of Illinois at Urbana Champaign, 1982, pp. 251-256.

[Sant83a] Santoro, N. and J.B. Sidney, "Communication Bounds for Selection in Distributed Sets," Working Paper 83-39, Faculty of Administration, Univ. of Ottawa, Ottawa, Canada, 1983.

[Sant83b] Santoro, N. and J.B. Sidney, "A Reduction Technique for Distributed Selection: I," Tech. Rep. SCS-TR-23, School of Computer Science, Carleton Univ., Ottawa, Canada, April 1983.

[Seit85] Seitz, C.L., "The Cosmic Cube," *CACM* **28**, 1 (Jan. 1985), pp. 22-33.

[Snir85] Snir, M., "On Parallel Searching," *SIAM J. Comput.* **14**, 3 (Aug. 1985), pp. 688-708.

[Stou83] Stout, Q.F., "Mesh-Connected Computers with Broadcasting," *IEEE Trans. Computers* **C-32**, 9 (Sept. 1983), pp. 826-830.

[Vish84] Vishkin, U., "A Parallel-Design Distributed-Implementation (PDDI) General-Purpose Computer," *TCS* **32**, 1 (July 1984), pp. 157-172.

[Will84] Willard, D., "Log-Logarithmic Protocol for Resolving Ethernet and Semaphore Conflicts," in *Proceedings 16th ACM Symp. on Theory of Computing*, 1984, pp. 512-521.

[Wyll79] Wyllie, J.C., "The Complexity of Parallel Computation," Tech. Rep. TR 79-387, Dept. of Computer Science, Cornell Univ., Ithaca, NY, Aug. 1979.

[Yao79] Yao, A.C., "Some Complexity Questions Related to Distributive Computing," in *Proceedings 11th ACM Symp. on Theory of Computing*, 1979, pp. 209-213.