EFFICIENT COMPUTATION ON SPARSE INTERCONNECTION NETWORKS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Charles Gregory Plaxton
September 1989

© Copyright 1989 by Charles Gregory Plaxton
All Rights Reserved

Abstract

This thesis presents fast hypercube and shuffle-exchange algorithms for certain load balancing, selection and sorting problems. Non-trivial lower bounds are established for load balancing and selection. In addition, efficient network implementations of the parallel prefix operation and of the elementary Boolean matrix multiplication algorithm are described.

Acknowledgements

I would like to thank my principal advisor, Ernst Mayr, and the other members of my reading committee, Bob Floyd, Andrew Goldberg and Jeff Ullman, for their enormous help in shaping the contents of this thesis.

While hanging around the Computer Science department, I have managed to familiarize myself with a number of its remarkable inhabitants, in addition to the surrounding brickwork and foliage. It's been great, and I hope that we cross paths again!

This work has been primarily supported by a 1967 Science and Engineering Scholarship from the Natural Sciences and Engineering Research Council of Canada. Additional funding was provided by a grant from the AT&T Foundation, NSF grant DCR-8351757 and ONR grant N00014-88-K-0166.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A considerable amount of research effort in the field of parallel computation has concentrated on developing algorithms for idealized machine models. The primary example of this is the PRAM model of Fortune and Wyllie, which assumes the existence of a shared memory allowing simultaneous random access by an unbounded number of processors [FW78].

This thesis adds to the growing body of work that addresses the design and analysis of algorithms for more realistic models of parallel computation. Specifically, all of the algorithms to be described are designed to run on sparse interconnection networks such as the hypercube and shuffle-exchange. Algorithms for performing operations such as parallel prefix, matrix multiplication, load balancing, selection and sorting will be considered. The primary motivation for developing fast implementations of these basic operations is to provide useful primitives for writing higher-level parallel programs.

1.1 Notation and Terminology

A p processor fixed interconnection network may be viewed as an undirected graph, where vertices correspond to processors and edges correspond to bidirectional[1] communication channels. Each processor has an infinite local memory, and a unique integer ID. There is no global memory; processors communicate with one another by sending and receiving data over the channels provided by the network. In order to discuss the time complexity of an algorithm it is necessary to define exactly what operations can be performed in a single unit of time, or *time step*. For establishing asymptotic upper bounds, it is realistic to assume that:

[1] With respect to the shuffle-exchange, the ability to "unshuffle" data is assumed.

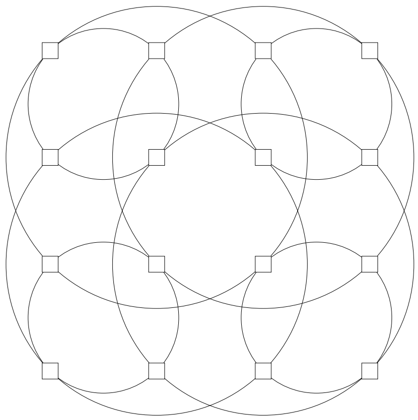

Figure 1.1: A hypercube of dimension 4 drawn with circular edges.

1. Memory is configured in $O(\log p)$ bit words.
2. In a single time step, a processor can send and/or receive a single word of data and perform $O(1)$ CPU operations on word-sized operands.

Most of the algorithms described in this thesis are designed to run on the hypercube and shuffle-exchange network families. A dimension d hypercube has 2^d processors with IDs ranging from 0 to $2^d - 1$. Processor i is adjacent to processor j if and only if the binary representations of i and j differ in a single bit position. A hypercube of dimension 4 is depicted in Figure 1.1.

The shuffle-exchange was introduced by Stone [Sto71]. Like the hypercube, a shuffle-exchange of dimension d has 2^d processors with IDs ranging from 0 to 2^{d-1}. Processor $i = (i_{d-1} \cdots i_0)_2$ is connected to processors *Exchange*(i), *Shuffle*(i) and *Unshuffle*(i), where

$$\begin{aligned} \mathit{Exchange}(i) &= (i_{d-1} \cdots i_1 (i_0 \oplus 1))_2, \\ \mathit{Shuffle}(i) &= (i_{d-2} \cdots i_0 i_{d-1})_2, \text{ and} \\ \mathit{Unshuffle}(i) &= (i_0 i_{d-1} \cdots i_1)_2, \end{aligned}$$

$0 \leq i < d$. A shuffle-exchange of dimension 4 is depicted in Figure 1.2.

Some important properties of the hypercube and shuffle-exchange network families are summarized in Table 1.1. Note that the degree of the shuffle-exchange is constant, while that of the hypercube is unbounded. Furthermore, the optimal VLSI layout area of the shuffle-exchange is somewhat smaller.

A more powerful model of the hypercube will also be considered, one which does not adhere to the 1-port restriction on communication imposed above. This is the *pipelined*

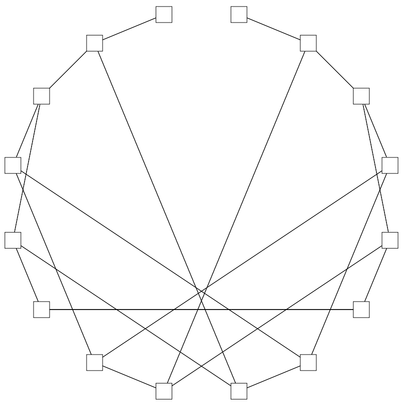

Figure 1.2: A shuffle-exchange of dimension 4 embedded on a circle.

Network	*Processors*	*Degree*	*Diameter*	*Layout Area*
hypercube	p	$\log p$	$\log p$	$\Theta(p^2)$
shuffle-exchange	p	3	$2 \log p$	$\Theta(p^2/\log^2 p)$

Table 1.1: Important properties of the hypercube and shuffle-exchange.

hypercube model of Varman and Doshi [VD88]. The pipelined hypercube remains a realistic model of computation by providing only a very restrictive form of d-port communication. Communication on the pipelined hypercube is via word-sized packets, routed according to the following simple scheme. Address bits are successively corrected in either ascending or descending (as determined by the sender) order of significance, with no collisions permitted. A collision occurs when two packets attempt to traverse the same edge in the same direction at the same time.

In routing a packet, one time step is expended for each bit in the smallest contiguous block of address bits that contains all of the bits to be corrected. For example, a packet sent from processor 10110101_2 to processor 10010011_2 must pass through dimensions 1, 2, and 5. Assuming that the sending processor elects to have address bits corrected in descending order of significance, the packet would be routed according to the schedule given by the following list of (time, processor) pairs: $(0, 10\mathbf{1}10101_2)$, $(1, 100\mathbf{1}0101_2)$, $(2, 1001\mathbf{0}101_2)$, $(3, 10010\mathbf{1}01_2)$, $(4, 100100\mathbf{0}1_2)$, $(5, 10010011_2)$. This packet is sent by processor 10110101_2, received and sent by processors 10010101_2 and 10010001, and received by 10110011_2. Let the first sender (10110101_2, in this example) be called the *originator* of the packet, and let the last receiver (10010011_2, in this example) be called the *acceptor* of the packet. The pipelined hypercube imposes the following pair of restrictions on communication:

1. Each processor is allowed to originate and/or accept at most one packet per time step.

2. Each edge can transmit at most one packet in each direction per time step.

The pipelined hypercube model is not realistic in a strict sense, since a single RAM cannot hope to examine $O(\log p)$ packets in $O(1)$ time. However, it may be a useful model in practice since the only additional hardware required at each hypercube processor is a ring of $O(\log p)$ trivial coprocessors to handle the packet routing scheme described above. Viewing this as an enhancement to the $O(\log p)$ I/O channel hardware already required by a hypercube processor, one would expect to suffer only a small constant factor increase in the VLSI area needed to implement a processor.

Several comments should be made with regard to mathematical notation. First, all logarithms are to be taken base 2, that is, $\log x$ denotes $\log_2 x$. Second, it will sometimes be convenient to make use of the function løg, defined as

$$\text{løg}\, x = \max\{\log x, 1\}.$$

Finally, $[a, b)$ will designate $\{i \mid a \leq i < b\}$.

1.2 Thesis Organization

The following is an overview of the main results contained in the thesis.

Chapter 2, which represents joint work with Ernst Mayr, provides pipelined parallel prefix algorithms for the complete binary tree, hypercube and shuffle-exchange. This primitive is used to develop a pipelined version of the multi-way merge sort of Nassimi and Sahni [NS82] that runs on the pipelined hypercube. Given p processors and $n < p \log p$ keys to be sorted, the running time of the pipelined hypercube sorting algorithm is $O(\log^2 p / \log((p \log p)/n))$, which improves (asymptotically) upon Batcher's bitonic sort by a $\log \log p$ factor in the important case $n = p$.

It has been shown that the product of two $n \times n$ Boolean matrices can be computed in $O(\log n)$ time on a hypercube or shuffle-exchange with $O(n^3)$ processors. Chapter 3 reduces this processor requirement to $O(n^3/(\log^2 n \log \log n))$ by making use of simulation techniques and a parallel version of the Four Russians' algorithm. This bound improves upon a result of Agerwala and Lint by a factor of $\log n$ [AL78].

Maintaining a balanced load is of fundamental importance on any parallel computer, since a strongly imbalanced load often leads to low processor utilization. Chapter 4 considers two load balancing problems. First, given n tokens arbitrarily distributed over a p processor network with no more than m tokens at any one processor, how fast can the tokens be redistributed so that each processor holds the same number? Second, given n tokens uniformly distributed over a p processor network, and arbitrarily partitioned into g groups, how fast can the tokens be redistributed so that each processor holds the same number from each group? Upper bounds on the worst case complexity of these two problems are obtained from the analysis of practical algorithms for the hypercube, pipelined hypercube and shuffle-exchange. Matching lower bounds are also provided for certain cases. Average performance is also considered.

Chapters 5 and 6 are concerned with the problem of selection, that is, determining the kth largest key out of a given set of n keys. Three different selection algorithms are given in Chapter 5, each of which represents the best known selection algorithm over some range of the ratio n/p (p is the number of processors) for one or more of the networks under consideration. One of the algorithms is based on fast sorting of small sets, one is based on load balancing, and one is based on a sequential tradeoff between preprocessing and search time in a partial order. For $n \geq p \log^2 p$, the latter algorithm runs in $O((n/p) \log\log p)$ time on the hypercube and shuffle exchange. Since the sequential complexity of selection is linear, one might hope that for n/p sufficiently large the $\log\log p$ factor in this running time could be eliminated. However, Chapter 6 proves that the $\log\log p$ factor cannot be eliminated. Specifically, a lower bound of $\Omega((n/p) \log\log p + \log p)$ is established for a large class of networks that includes the complete binary tree, multi-dimensional mesh, hypercube, butterfly and shuffle-exchange.

Chapters 7 and 8 deal with the problem of sorting a set of n keys with p processors. Chapter 7 makes use of the load balancing and selection results of Chapters 4 and 5 to derive fast, practical sorting algorithms for the hypercube, shuffle-exchange and pipelined hypercube. Chapter 8 considers two approaches to sorting when algorithms are confined to a class corresponding essentially to sorting circuits.[2] For sufficiently large values of the ratio n/p, all of the sorting algorithms described in these two chapters have a lower asymptotic complexity than Batcher's bitonic sort [Bat68].

[2]The term "sorting circuit" will be used in lieu of the more usual "sorting network" in order to avoid confusion with interconnection networks. For a thorough introduction to the design and analysis of sorting circuits, see Knuth [Knu73].

Finally, Chapter 9 offers some concluding remarks and open problems for further consideration.

Chapter 2

Pipelining

This chapter combines several previously known techniques to obtain fast implementations of the so-called parallel prefix operation. Algorithms are given for the complete binary tree as well as the hypercube and shuffle-exchange. Pipelined schemes for performing k prefix operations in $O(k + \log p)$ time on p processors are given for the same set of networks. Pipelined parallel prefix is then used to develop a simplified implementation of the optimal merging algorithm of Varman and Doshi, which runs on the pipelined hypercube [VD88]. Finally, a pipelined version of the multi-way merge sort of Nassimi and Sahni [NS82], running on the pipelined hypercube, is described. Given p processors and $n < p \log p$ keys to be sorted, the running time of the pipelined algorithm is $O(\log^2 p / \log((p \log p)/n))$. For the interesting case $n = p$ this yields a running time of $O(\frac{\log^2 p}{\log \log p})$, which is asymptotically faster than Batcher's bitonic sort [Bat68].

2.1 The Prefix Operation

The prefix operation was introduced independently by Schwartz [Sch80] and by Ladner and Fischer [LF80]. For other work on parallel prefix, the reader is referred to [Fic83] and [Rei84].

Let $\oplus$ denote a binary associative operator on some domain $\mathcal{X}$. Given $\{x_0, \ldots, x_{n-1}\} \subseteq \mathcal{X}$, the Prefix operation computes each of the partial sums $y_i = x_0 \oplus \cdots \oplus x_i$, $0 \leq i < n$. For example, assuming that $\oplus$ is addition, $n = 5$, $x_0 = 5$, $x_1 = 2$, $x_2 = 6$, $x_3 = 4$ and $x_4 = 9$, then the output of Prefix is $y_0 = 5$, $y_1 = 7$, $y_2 = 13$, $y_3 = 17$ and $y_4 = 26$.

Given an additional n Boolean values $a_0, \ldots, a_{n-1}$, the n given x_i values can be partitioned into contiguous intervals in the following manner: an interval begins at each i such that $a_i = \mathbf{true}$ and extends up to, but not including, the next highest integer j such that $a_j = \mathbf{true}$. The first interval begins at processor 0 regardless of the value of a_0, and the last interval ends at processor $n-1$. The segmented Prefix operation executes a prefix operation over each interval. Extending the example of the preceding paragraph, assume that a_2 and a_4 are **true** while a_0, a_1 and a_3 are **false**. Then the x_i values are partitioned into the intervals $\{x_0, x_1\}$, $\{x_2, x_3\}$ and $\{x_4\}$ and the output of the segmented Prefix operation is $y_0 = 5$, $y_1 = 7$, $y_2 = 6$, $y_3 = 10$ and $y_4 = 9$.

When implementations of the Prefix operation for various networks are given in Section 2.2, it will be convenient to assume that there is an identity element for $\oplus$ in $\mathcal{X}$, which will be denoted $\mathbf{0}_\oplus$. This assumption can be made without loss of generality because if no such element exists, the set $\mathcal{X}$ can be augmented with an identity element $\mathbf{0}_\oplus$ by defining $\mathbf{0}_\oplus \oplus x = x$ and $x \oplus \mathbf{0}_\oplus = x$ for all $x \in \mathcal{X} \cup \{\mathbf{0}_\oplus\}$. Note that associativity is preserved.

Definition 2.1.1 For all pairs of Boolean values a_0, a_1 and all $x_0, x_1 \in \mathcal{X}$, let $\oplus'$ denote the binary operator

$$(a_0, x_0) \oplus' (a_1, x_1) = (a_0 \ \mathbf{or} \ a_1, \ \mathbf{if} \ a_1 \ \mathbf{then} \ x_1 \ \mathbf{else} \ x_0 \oplus x_1).$$

The operator $\oplus'$ will be referred to as the *segmented $\oplus$ operator.*

Remark 1 The $\oplus'$ operator has identity $\mathbf{0}_{\oplus'} = (\mathbf{false}, \mathbf{0}_\oplus)$.

Remark 2 The $\oplus'$ operator is not commutative, assuming $|\mathcal{X}| > 1$.

Remark 3 The $\oplus'$ operator is associative.

Remark 4 For $k \geq 0$,

$$(a_0, x_0) \oplus' \cdots \oplus' (a_k, x_k) = (a_0 \ \mathbf{or} \ \cdots \ \mathbf{or} \ a_k, \ x_j \oplus \cdots \oplus x_k),$$

where j is the highest index less than or equal to k such that $a_j = \mathbf{true}$, or 0 if there is no such index.

Remark 1 is an immediate consequence of Definition 2.1.1. For Remark 2, let x_0, x_1 be distinct elements of $\mathcal{X}$ and note that $(\mathbf{true}, x_0) \oplus' (\mathbf{true}, x_1) = x_1$ while $(\mathbf{true}, x_1) \oplus'$

$(\mathbf{true}, x_0) = x_0$. Remark 3 follows from the observation that for all Boolean values a_0, a_1, a_2 and $x_0, x_1, x_2 \in \mathcal{X}$

$$\begin{aligned}
&((a_0, x_0) \oplus' (a_1, x_1)) \oplus' (a_2, x_2) \\
&\quad = (a_0 \textbf{ or } a_1,\ \textbf{if } a_1 \textbf{ then } x_1 \textbf{ else } x_0 \oplus x_1) \oplus' (a_2, x_2) \\
&\quad = (a_0 \textbf{ or } a_1 \textbf{ or } a_2,\ \textbf{if } a_2 \textbf{ then } x_2 \textbf{ else if } a_1 \textbf{ then } x_1 \oplus x_2 \textbf{ else } x_0 \oplus x_1 \oplus x_2) \\
&\quad = (a_0 \textbf{ or } (a_1 \textbf{ or } a_2),\ \textbf{if } (a_1 \textbf{ or } a_2) \textbf{ then } X \textbf{ else } x_0 \oplus X) \\
&\quad = (a_0, x_0) \oplus' (a_1 \textbf{ or } a_2,\ X) \\
&\quad = (a_0, x_0) \oplus' ((a_1, x_1) \oplus' (a_2, x_2)),
\end{aligned}$$

where X denotes the conditional expression: $\textbf{if } a_2 \textbf{ then } x_2 \textbf{ else } x_1 \oplus x_2$. Finally, Remark 4 may be easily established by induction on k.

Remarks 3 and 4 demonstrate that any segmented Prefix operation with operator $\oplus$ mapping $\mathcal{X} \times \mathcal{X}$ to $\mathcal{X}$ is equivalent to an ordinary Prefix operation with operator $\oplus'$ mapping $(\mathcal{B} \times \mathcal{X}) \times (\mathcal{B} \times \mathcal{X})$ to $\mathcal{B} \times \mathcal{X}$, where $\mathcal{B}$ denotes the set of Boolean values $\{\mathbf{true}, \mathbf{false}\}$. The second component of each output pair is the result of the desired segmented Prefix operation, and the first component indicates whether or not that processor belongs to an "undefined" interval; it is **false** at processor i if and only if $a_0, \ldots, a_i$ are all **false**. This reduces coding segmented prefix to coding ordinary prefix.

2.2 Network Implementations

This section presents efficient implementations of the Prefix operation for the complete binary tree, hypercube and shuffle-exchange families of networks. It will be assumed that the network consists of $p = n$ processors, and that processor i initially contains the value x_i, $0 \le i < p$. The computation is considered to be complete when the partial sum $y_i = x_0 \oplus \cdots \oplus x_i$ has been computed at processor i, $0 \le i < p$. The complexity of the algorithms will be stated in terms of *time steps*, as defined in Section 1.1. Unless otherwise stated, running times should be assumed to be accurate to within an additive constant. It will be assumed that the x_i's, as well as all partial sums of the x_i's, are word-sized quantities.

In the programs to follow, all interprocessor communication will be specified using the pair of routines Send and Receive. Send takes two arguments: the first specifies the word of data to be transmitted, and the second specifies the ID of the destination processor. Receive is a function with one argument which specifies the ID of the source processor. Once a packet

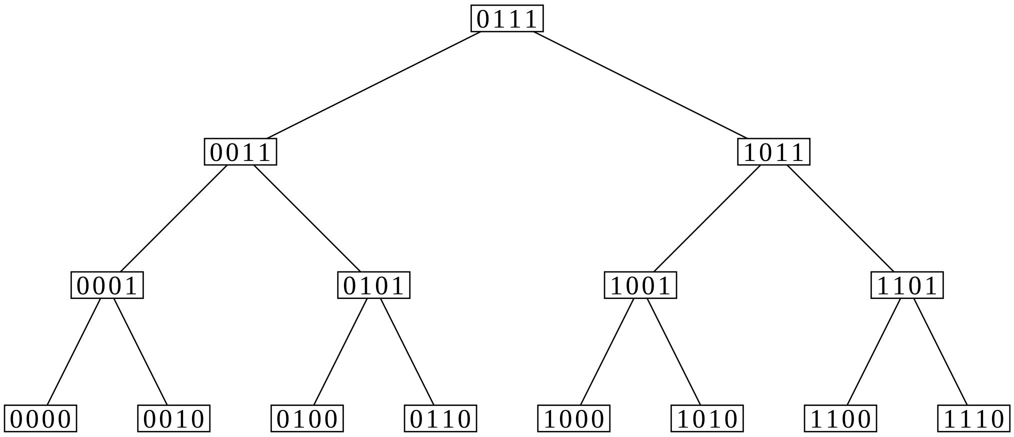

Figure 2.1: An inorder complete binary tree.

arrives from the source, the word of data contained in that packet is returned as the value of the function. In order to comprise a valid source/destination pair, two processors must be adjacent in the network.

2.2.1 Binary Tree

The first implementation of Prefix that will be considered is the standard two-pass algorithm for the inorder complete binary tree. Assume that a binary tree of size $p = 2^d - 1$ is given, with processors numbered inorder from 0 to $2^d - 2$. An example of such a network is shown in Figure 2.1, where the processor IDs have been written in binary, and $d = 4$. The code for this algorithm assumes that each processor has initialized the variables *Root*, *Leaf*, *LeftChild*, *RightChild* and *Parent* in the following manner. The Boolean variable *Root* (*Leaf*) is **true** if and only if the processor represents the root (a leaf) of the tree. The integer variables *LeftChild*, *RightChild* and *Parent* hold the IDs of the neighboring processors, and are undefined whenever such a neighbor does not exist.

begin Prefix($\oplus$, x)
$x_{\mathrm{L}} \longleftarrow$ **if** *Leaf* **then** $\mathbf{0}_{\oplus}$ **else** Receive(*LeftChild*);
$x_{\mathrm{R}} \longleftarrow$ **if** *Leaf* **then** $\mathbf{0}_{\oplus}$ **else** Receive(*RightChild*);
if not *Root* **then** Send($x_{\mathrm{L}} \oplus x \oplus x_{\mathrm{R}}$, *Parent*);
$y_{\mathrm{L}} \longleftarrow$ **if** *Root* **then** $\mathbf{0}_{\oplus}$ **else** Receive(*Parent*);
$y_{\mathrm{R}} \longleftarrow y_{\mathrm{L}} \oplus x_{\mathrm{L}} \oplus x$;
if not *Leaf* **then** Send(y_{L}, *LeftChild*);
if not *Leaf* **then** Send(y_{R}, *RightChild*);

```
        return(y_R);
    end Prefix
```

As mentioned above, the program makes two passes over the tree. The first pass is upward, from the leaves to the root, and the second pass is downward. For every processor p, let $T(p)$ denote the subtree rooted at processor p. Note that the IDs of the processors in $T(p)$ form a contiguous block of integers. During the upward pass, each processor receives the sum of its left and right subtrees (x_{L} and x_{R}), computes the sum over $T(p)$, and passes the result to its parent. During the downward pass, each processor receives from its parent the sum y_{L} over all processors with IDs less than those in $T(p)$, computes the sum over all processors with IDs less than those in its right subtree (y_{R}), and sends the appropriate values to its left and right children (y_{L} and y_{R}). The correctness of the program is easily established by induction on the depth of the tree, and it runs in $4 \log p$ time steps.

Note that in any given time step, only two of the levels of the tree are active, implying that the algorithm can be pipelined level by level. By initiating a new prefix computation every second time step, it is possible to perform k Prefix operations on the inorder complete binary tree in $2k + 4 \log p$ time steps.

2.2.2 Hypercube

For the hypercube, the following FFT-like computation executes Prefix in $\log p$ time steps:

```
    begin Prefix(⊕, x)
        y ⟵ x;
        for i ⟵ 0 to d − 1 do
            Send(y, i);
            if MyId_i = 0 then
                y ⟵ y ⊕ Receive(i);
            else
                temp ⟵ Receive(i);
                x ⟵ temp ⊕ x;
                y ⟵ temp ⊕ y;
            end if
        end for
        return(x);
```

end Prefix

The variable *MyId* holds the ID of the processor, and $MyId_i$ denotes the ith bit of the ID (the least significant bit is bit 0). The source and destination arguments of Send and Receive specify the bit position in which the two communicating processors differ.

The program runs in $\log p$ time steps, and functions in the following manner. In addition to the partial sums demanded by the Prefix operation, the total sum is computed at every processor. The local variables x and y accumulate the partial and total sums, respectively. For a hypercube consisting of a single processor, the computation is trivial. Given p processors with associated x_i values and where $p = 2^d$, $d \geq 1$, the program first recursively computes partial and total sums for the upper and lower halves of the values independently, and then exchanges the total sums between halves. This enables the revised partial sums for the upper half to be computed, as well as the new total sums.

Unfortunately, the above program does not lead to a pipelined implementation of the Prefix operation because it uses all of the processors at every time step. One way of achieving pipelined speedup is to make use of the dilation 2 inorder complete binary tree embedding [BCLR86]. Figure 2.2 gives this embedding for the case $p = 16$, where the "extra" processor (with ID $p - 1$) has been added as an extra level above the root. The edges depicted in Figure 2.2 are physical hypercube edges. The left child of a non-leaf processor is connected directly to its parent, while the right child is connected to its parent via the left child. It is easy to verify that the pipelined algorithm given for the inorder complete binary tree in Section 2.2.1 can be modified to run in the same time bound on the dilation 2 inorder complete binary tree embedding. In particular, note that processor $p - 1$ is in an appropriate location to receive the sum over all of the other processors. To summarize, k Prefix operations can be performed in $2k + 4 \log p$ time steps on the hypercube.

2.2.3 Shuffle-Exchange

The hypercube code given in the preceding section for performing a single Prefix operation can be easily adapted to the shuffle-exchange:

begin Prefix($\oplus$, x)
(1) $y \longleftarrow x$;
(2) **repeat** d **times**

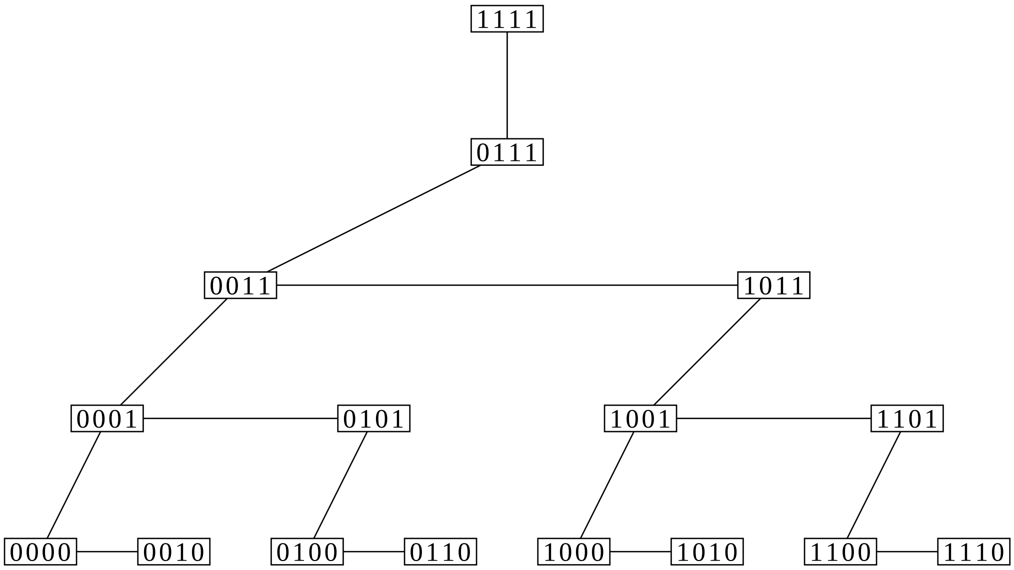

Figure 2.2: Embedding the inorder binary tree in the hypercube.

        Send(y, *Exchange*);
        if $MyId_0 = 0$ **then**
            $y \longleftarrow y \oplus$ Receive(*Exchange*);
        else
            temp $\longleftarrow$ Receive(*Exchange*);
            $x \longleftarrow temp \oplus x$;
            $y \longleftarrow temp \oplus y$;
        end if
        Send(x, *Unshuffle*);
        $x \longleftarrow$ Receive(*Shuffle*);
        Send(y, *Unshuffle*);
        $y \longleftarrow$ Receive(*Shuffle*);
    end repeat
    return(x);
end Prefix

The above program runs in $3 \log p$ time steps. As in the case of the hypercube, however, a different approach is needed in order to obtain a pipelined implementation of the Prefix operation. Unfortunately, it is not possible to embed the inorder complete binary tree in the shuffle-exchange with constant dilation. Instead, a pipelined implementation will be

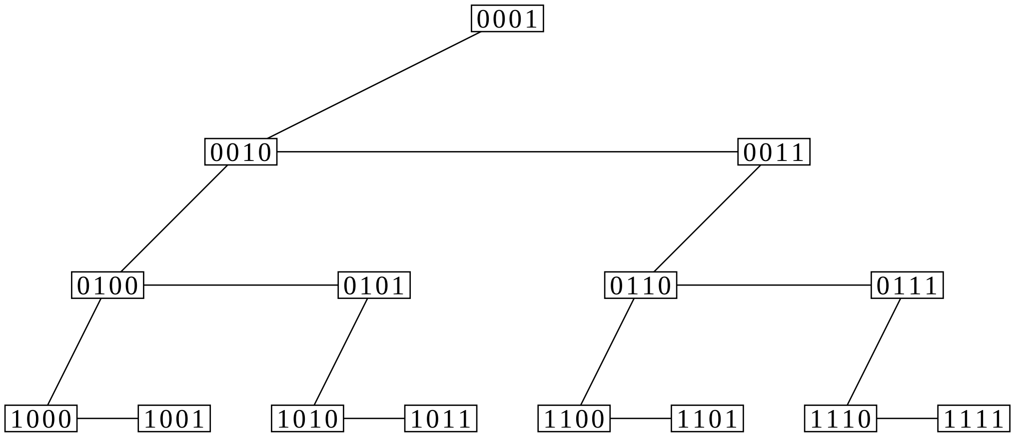

Figure 2.3: A shuffle-exchange embedding for the high-numbered processors.

obtained by making use of the dilation 2 complete binary tree embeddings depicted, for the case $p = 16$, in Figures 2.3 and 2.4. The leaves of the tree in Figure 2.3 are the high-numbered processors (those with IDs in the range $p/2$ to $p - 1$), numbered inorder. In this embedding, the ID of the left child of an internal processor is the shuffle of the ID of its parent, and siblings communicate via the exchange connection. The embedding of Figure 2.4 is defined in a similar fashion, and has the low-numbered processors (0 to $p/2-1$) at its leaves.

These embeddings can be used to obtain a pipelined implementation of k Prefix operations as follows. First, the embedding of Figure 2.3 is used to compute the k sets of partial sums over the high-numbered processors. This takes $2k + 4\log p$ time steps. Similarly, the embedding of Figure 2.4 can be used to perform k prefix sums over the low-numbered processors in $2k+4\log p$ time steps. At this point, all that remains to be done is to broadcast, in a pipelined fashion, the k total sums over the low-numbered processors to the $p/2$ high-numbered processors, and to add these values to the partial sums computed earlier. This last phase can be performed in $2k + 2\log p$ time steps using the embedding of Figure 2.4.[1] Hence, k Prefix operations can be executed in $6k + 10\log p$ time steps on the shuffle-exchange.

[1]Note that as a side-effect of the prefix sums performed over the low-numbered processors, the desired sums are already available at the root.

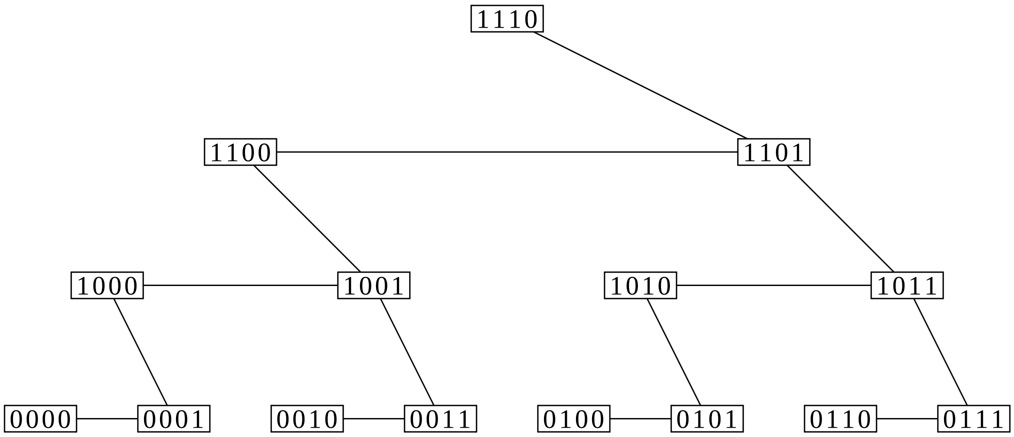

Figure 2.4: A shuffle-exchange embedding for the low-numbered processors.

2.2.4 A Useful Variation

Section 2.4 will make use of a variant of the Prefix operation, Prefix′, defined as follows. Rather than computing $x_0 \oplus \cdots \oplus x_i$ at processor i, $0 \leq i < p$, Prefix′ outputs $\mathbf{0}_\oplus$ at processor 0 and $x_0 \oplus \cdots \oplus x_{i-1}$ at processor i, $1 \leq i < p$. This is sometimes more convenient, particularly when the operator $\oplus$ is not invertible. Note that all of the implementations of Prefix given above may be trivially modified to implement Prefix′ within precisely the same time bounds. For example, in the complete binary tree program of Section 2.2.1, it suffices to change the return value from y_R to $y_\mathrm{L} \oplus x_\mathrm{L}$.

2.3 Data Distribution

Consider the binary associative operator $\oplus$ defined over $\mathcal{X}$ by $x \oplus y = x$, for all $x, y \in \mathcal{X}$. This is sometimes referred to as the Copy operator. Observe that the effect of applying Prefix with the Copy operator is to perform a broadcast of a single value from processor 0 to all other processors. Of course, there are simpler techniques for broadcasting a single value over the processors of any of the networks under consideration. However, combining this observation with the results of the previous section immediately implies that k *segmented* broadcasts can be executed in $2k + 4\log p$ time steps on the tree or hypercube, and in $6k + 10\log p$ time steps on the shuffle-exchange.

In order to fully illustrate the techniques discussed in Section 2.1, the implementation of segmented Prefix with the Copy operation will now be studied in greater detail. As stated in Section 2.1, processor i initially holds the Boolean value a_i and $x_i \in \mathcal{X}$, $0 \leq i < p$. Note that under the Copy operation the only relevant x_i's are those for which the corresponding a_i is **true** .

Clearly, there is no identity element for the Copy operation in $\mathcal{X}$. To remedy this situation, the domain of Copy is extended from $\mathcal{X}$ to $\mathcal{B} \times \mathcal{X}$ where every pair with first component **false**, say, is defined to be an identity element. In practice, this corresponds to prefixing a single bit b_i to each of the x_i's. Formally, the operator $\oplus = \mathsf{Copy}$ becomes

$$(b_0, x_0) \oplus (b_1, x_1) = (b_0 \textbf{ or } b_1,\ \textbf{if } b_0 \textbf{ then } x_0 \textbf{ else } x_1),$$

for all $b_0, b_1 \in \mathcal{B}$ and $x_0, x_1 \in \mathcal{X}$.

In order to reduce segmented Prefix with operator $\oplus = \mathsf{Copy}$ to ordinary Prefix with operator $\oplus' = \mathsf{Copy}'$, let $\oplus'$ be defined as follows:

$$(a_0, (b_0, x_0)) \oplus' (a_1, (b_1, x_1)) = (a_0 \textbf{ or } a_1,\ \textbf{if } a_1 \textbf{ then } (b_1, x_1) \textbf{ else } (b_0, x_0) \oplus (b_1, x_1)).$$

Dropping the inner parentheses and simplifying gives

$$\begin{aligned}(a_0, b_0, x_0) \oplus' (a_1, b_1, x_1) \quad = \quad & (a_0 \textbf{ or } a_1, \\ & \quad \textbf{if } a_1 \textbf{ then } b_1 \textbf{ else } b_0 \textbf{ or } b_1, \\ & \quad \textbf{if } a_1 \textbf{ or not } b_0 \textbf{ then } x_1 \textbf{ else } x_0).\end{aligned}$$

Note that the above formulation allows bit pipelining in the sense described by Blelloch [Ble87]. In other words, as each bit of the two operands is received, the next output bit can be computed. This property holds not only for the Copy operator, but also for any other *single-pass operator*, as defined by Blelloch [Ble87].

Finally, observe that the data distribution operation defined by Ullman [Ull84] is equivalent to a segmented Prefix operation with the Copy operator. Thus, the techniques outlined in Section 2.2 immediately lead to efficient pipelined implementations of this primitive for the complete inorder binary tree, hypercube and shuffle-exchange network families.

2.4 Sorting on the Pipelined Hypercube

In this section, a simplified implementation of the optimal merging algorithm of Varman and Doshi [VD88] will be described, and this will be used to develop a pipelined version of the sorting algorithm of Nassimi and Sahni [NS82] for the pipelined hypercube. A formal definition of the Sort operation, along with a discussion of previous sorting results for the hypercube and shuffle-exchange, may be found in Chapter 7.

The added power of the pipelined hypercube will only be used for performing pipelined inverse concentration routes. It is interesting to note that the pipelined hypercube is not needed in order to perform pipelined concentration routes, nor is it needed to perform the pipelined inverse concentration *with copy* operation of Varman and Doshi. Concentration and inverse concentration routes were defined by Nassimi and Sahni [NS82], and it is easy to show that k such operations can be performed in $k + \log p$ time steps on the pipelined hypercube. Furthermore, there is no hope of achieving this asymptotic time bound on the 1-port hypercube since there is a lower bound of $\Omega(k \log^{1/2} p)$ time steps in this case. To prove this lower bound, consider a set of k monotone routes for which the source processors are exactly those with strictly more 0's than 1's in their IDs and the destination processors are those with more 1's than 0's. In such a case, $\Omega(kp)$ packets must pass through the $O(p \log^{-1/2} p)$ processors with an equal number of 0's and 1's (or one more 0 than 1, say, if $\log p$ is odd), which implies a lower bound of $\Omega(k \log^{1/2} p)$ time steps for performing k monotone routes. Since every monotone route can be decomposed into a concentration route followed by an inverse concentration route, and these operations have equal complexity, this lower bound applies to the pipelined concentration and inverse concentration operations as well.

A pipelined algorithm for merging two sorted lists X and Y will now be described. Both X and Y are of length pk, and the algorithm runs on p processors. The algorithm is similar to that proposed by Varman and Doshi [VD88], but is somewhat simpler. The optimal merging algorithm of Anderson, Mayr and Warmuth for the EREW PRAM also takes a similar approach [AMW88]. For simplicity, it will be assumed that all of the $2pk$ input keys are distinct. For both X and Y, the keys with ranks (numbered from 0) in the range ik to $(i+1)k-1$ are initially stored at processor i, $0 \leq i < p$. The two ordered sets of k keys located at processor i will be referred to as X_i and Y_i, respectively. Let x_i denote the least element of X_i, and let y_i denote the greatest element of Y_i, $0 \leq i < p$. Let X' and Y' denote the set of all x_i's and y_i's, respectively. Let Z denote the sorted list of length $2pk$ that results from merging X and Y. Those elements of Z with ranks in the range $2ik$ to $2(i+1)k-1$, denoted Z_i, must be routed to processor i by the end of the computation, $0 \leq i < p$, and must be sorted locally.

The approach taken is first to merge X' and Y', and then to use the resulting list to guide the merging of X and Y. Let Z' denote the sorted list of length $2p$ that results from merging X' and Y'. Let z_j denote the key with rank j in Z', $0 \leq j < 2p$. Let Z'_j denote the set of k keys associated with z_j, that is, either $z_j = x_i$ for some $x_i \in X'$ and $Z'_j = X_i$, or $z_j = y_i$ for some $y_i \in Y'$ and $Z'_j = Y_i$. Note that if $z_j \in X'$ then the rank of z_j in Z is between jk and $(j+1)k - 1$, inclusive. The exact rank of z_j in Z can be determined by computing its rank in the set Y_i, where y_i is the least element of Y' exceeding z_j. Similarly, if $z_j \in Y'$ then the rank of z_j in Z is between jk and $(j+1)k - 1$, and the exact rank of z_j in Z depends upon the set X_i, where x_i is the largest element of X' that is less than z_j. Furthermore, it is easy to check that the set Z_j is contained in the union of Z'_{2j}, Z'_{2j+1}, the set X_i corresponding to the largest x_i that is less than z_{2j}, and the set Y_i corresponding to the smallest y_i that is greater than z_{2j+1}. These observations lead to the following pipelined merging algorithm.

Algorithm Merge

1. Reverse the list Y', that is, route y_i to processor $p - i - 1$, $0 \leq i < p$. This takes $\log p$ time steps.

2. Merge X' and Y' by simulating a bitonic merge over $2p$ processors. Record the data movements to facilitate the "unmerge" of Step 3. This takes $2 \log p$ time steps.

3. Route the rank of each key in Z' back to the processor which originally held that key. This can be done in $2 \log p$ time steps by following the paths recorded in Step 2 in the reverse direction.

4. Route each set X_i to the processor that held x_i after Step 2, $0 \leq i < p$. The ID of that processor can be computed from the rank received by processor i in Step 3. The routing can be performed in $2k + 2 \log p$ time steps using a pipelined inverse concentration. Route the Y_i's in a similar fashion, for a total cost of $4k + 4 \log p$ time steps.

5. Assuming the set X_i was routed to processor j_i in the previous step, broadcast X_i to all processors with IDs in the range $j_i + 1$ to j_{i+1}, $0 \leq i < p$. This can be done in $2k + 4 \log p$ time steps with a single application of the Prefix$'$ operation, as described in Section 2.2.

6. Assuming the set Y_i was routed to processor j_i in the previous step, broadcast Y_i to all processors with IDs in the range j_{i-1} to $j_i - 1$, $0 \le i < p$. This can be done with a single application of a "backwards" version of Prefix$'$, and takes $2k + 4\log p$ time steps.

7. At this point, processor j contains a copy of Z'_{2j}, Z'_{2j+1}, the largest X_i with $x_i < z_{2j}$ and the smallest Y_i with $y_i > z_{2j+1}$, $0 \le j < p$. As observed above, the union of these sets contains the desired set Z_j, and the keys to be discarded (i.e., those not belonging to Z_j) can be determined by computing the exact rank of either z_{2j} or z_{2j+1}. These sets can be merged, and the rank computation performed, with $O(k)$ local operations. The definition of a time step allows these local operations to be interleaved with the computations of Steps 5 and 6 at no extra cost.

Note that only Step 4 uses the power of the pipelined model. The total running time of Merge is $8k + 17\log p$ time steps. Now consider the case in which $2p$ processors are available to perform the merge, where it is assumed that X_i is initially stored at processor i, Y_i is initially stored at processor $2p - i - 1$, and Z_j is to be output at processor j, $0 \le i < p$, $0 \le j < 2p$. In this case, Step 1 is unnecessary, and the cost of each of Steps 2, 3 and 4 is halved, while the cost of the remaining steps is unchanged. Thus, the total cost of Merge with $2p$ processors is $6k + 12\log p$ time steps. Note that for $k = \Omega(\log p)$, this running time is within a constant factor of optimal. Furthermore, as observed by Varman and Doshi, this optimal merging routine immediately implies an optimal algorithm for sorting when the number of keys to be sorted, n, exceeds the number of processors, p, by a factor k that is $\Omega(\log p)$. The idea is to sort the set of k keys at each processor locally, and then to merge sorted subcubes repeatedly until the entire hypercube has been sorted. At each level, even subcubes are sorted in ascending order and odd subcubes are sorted in descending order. The running time of this algorithm, which will be referred to as MergeSort, is

$$\sum_{0 \le i < \log p} (6k + 12i) = 6k\log p + O(\log^2 p).$$

As mentioned above, this running time is optimal for $k = \Omega(\log p)$.

A pipelined version of the multi-way merging procedure of Nassimi and Sahni [NS82] that runs on the pipelined hypercube will now be described. The input consists of 2^l sorted lists of length $k2^m$, and the output is a single sorted list of length $k2^{l+m}$. The merging is performed in $O(k + \log p)$ time steps on a hypercube with $p = 2^{2l+m}$ processors. Let the ith input list be denoted X^i, $0 \leq i < 2^l$, and let the set of k elements of X^i with ranks between jk and $(j+1)k-1$ (inclusive) be denoted X^i_j, $0 \leq j < 2^m$. The set X^i_j is initially stored at processor $i2^m + j$. Let the output list be denoted X. At the end of the merging process, the elements of X with ranks between jk and $(j+1)k-1$ (inclusive) should be stored at processor j, $0 \leq j < 2^{l+m}$.

It is useful to view the processors of the given hypercube as forming a 2^l by 2^{l+m} array, where the processor in row i and column j has ID $i2^{l+m} + j$ (row-major order). Note that all of the X^i_j's are stored in row 0. In fact, each processor in row 0 contains exactly one set X^i_j.

The algorithm makes use of pipelined broadcast and sum operations over entire subcubes. Formally, a pipelined broadcast operation takes k keys stored at a single processor and broadcasts them over the entire subcube. For a pipelined sum operation, processor i initially holds k keys a_{ij}, $0 \leq i < p$, $0 \leq j < k$. The output is the k sums $\sum_{0 \leq i < p} a_{ij}$, $0 \leq j < k$, all of which are output at a single designated processor. Although such operations can be performed using Prefix, other implementations exist which are more efficient by a constant factor. For example, using the multiple spanning binomial tree (MSBT) embedding of Ho and Johnsson [HJ86] it is possible to perform k broadcasts in $k + \log p$ time steps. Similarly, k sums can be performed in $k + \log p$ time steps. Note that although these operations are pipelined, they run on the 1-port hypercube and thus do not require the additional power of the pipelined hypercube.

Algorithm MultiWayMerge

1. Broadcast X^i_j to all of the processors in column $i2^m + j$, $0 \leq i < 2^l$, $0 \leq j < 2^m$. Each of the columns is an independent subcube of dimension l. Thus, the broadcasts can be performed in $k + l$ time steps using an MSBT embedding within each column.

2. Replicate list X^i across the ith row, $0 \leq i < 2^l$. In other words, route a copy of X^i_j to each column of the ith row that is congruent to $j \bmod 2^m$. This amounts to performing pipelined broadcasts over subcubes of dimension l, which can be done in $k + l$ time steps using the MSBT embedding.

3. Merge the lists X^i and X^j using the jth block of 2^m processors of row i (i.e., columns $j2^m$ to $(j+1)2^m - 1$), $0 \leq i, j < 2^l$, $i \neq j$. This takes $8k + 17m$ time steps.

4. In the jth block of 2^m processors of row i, "unmerge" the rank of each element of X^i in X^j (this is the rank of that key in $X^i \cup X^j$ minus its rank in X_i), $0 \leq i, j < 2^l$, $i \neq j$. In other words, route the rank of each key back to the processor that contained the key before Step 3. This is a pipelined inverse concentration, and can be performed in $k + m$ time steps. Where $i = j$, simply label each key with its rank in X^i.

5. Compute the rank of every key in X. The processors of row i are used to perform this computation for the elements of the set X^i, $0 \leq i < 2^l$. For each set X^i_j, a pipelined sum is performed over a subcube of dimension l, adding the ranks computed in Step 4 and routing the results to the first block of 2^m processors in each row. This takes $k+l$ time steps using the MSBT embedding.

6. In row i, route the elements of X_i to the correct output column (given by the floor of the rank computed in Step 5 divided by k), $0 \leq i < 2^l$. This is a pipelined inverse concentration in a subcube of dimension $l + m$, and takes $k + l + m$ time steps.

7. Each column of the array now contains k keys. Route these keys to the top of the column (row 0). In terms of data paths, this is essentially an inverse pipelined broadcast operation over a subcube of dimension l, and it can be performed in $k + l$ time steps using the MSBT embedding.

Only Steps 3, 4 and 6 require the power of the pipelined hypercube. Summing all of the costs stated above, the total running time of MultiWayMerge is readily seen to be $14k + 5l + 19m$ time steps.

Repeated application of MultiWayMerge on successively larger subcubes leads to a fast sorting algorithm for the case $n < p \log p$. The running time of this algorithm, which will be referred to as MultiWayMergeSort, will be shown to be $O(\log^2 p / \log((p \log p)/n)))$, as opposed to $O(\log^2 p / \log(p/n))$ for the sorting algorithm of Nassimi and Sahni. For the interesting case $n = p$, the running time of MultiWayMergeSort is $O(\log^2 p / \log \log p)$, a slight asymptotic improvement over that of Batcher's bitonic sort. It must be emphasized, however, that MultiWayMergeSort only runs on the pipelined hypercube.

A more formal description of the MultiWayMergeSort algorithm will now be given, along with an analysis of its time complexity. The algorithm is designed to sort $n = k2^m$ keys on a hypercube with $p = 2^{l+m}$ processors. It is useful to view the processors as being arranged in a 2^l by 2^m array, where the processor in row i and column j has ID $i2^m + j$ (row-major order).

Algorithm MultiWayMergeSort

1. Each column of the array contains k keys. Route all of these to the top of the column (row 0). As in Step 7 of MultiWayMerge, this takes $k + l$ time steps.

2. At every processor in row 0, sort the set of k keys using an efficient sequential sorting routine. This takes $O(k \log k)$ time steps.

3. Repeatedly call MultiWayMerge. The length of the sorted lists increases by a factor of 2^l after each call. Thus, after $\lceil m/l \rceil$ iterations all of the keys have been sorted. The cost of the ith iteration is $14k + 5l + 19il$ time steps, for a total cost of approximately $(14k + 4l + 12m)m/l$ time steps.

4. The keys have been sorted, but they are not configured appropriately (i.e., all of the keys are in row 0). All of the keys can be routed to the correct output locations using k pipelined inverse concentration routes, which takes $k + \log p$ time steps.

Steps 3 and 4 make use of the power of the pipelined hypercube. The total running time of MultiWayMergeSort is minimized (to within a constant factor) by setting $k = \log p$, and for this choice of k the running time is dominated by the cost of Step 3. Observing that $l = \log(pk/n)$ and $m = \log p - l \leq \log p$, one finds that for $k = \log p$ the algorithm runs in $\frac{47}{2} \log^2 p / \log((p \log p)/n) + O(\log p)$ time steps. For the case $n = p$, k can be set to $\log p / \log\log p$ in order to reduce the dominant term in the running time to $\frac{19}{2} \log^2 p / \log\log p$.

2.5 Summary

This chapter has described simple and efficient pipelined implementations for the Prefix operation on the complete inorder binary tree, hypercube and shuffle-exchange families of networks. Since Ullman's data distribution primitive may be viewed as a special case of the Prefix operation, these results immediately yield a pipelined implementation for that primitive. A variant of the Prefix operation was used to obtain a simplified implementation of Varman and Doshi's optimal merging algorithm for the pipelined model of the hypercube.

In order to better assess the practical speed of the various algorithms presented in this paper, the coefficient on the leading term of the running time has been computed in each case. It is quite possible that one or more of the moderately large coefficients in Section 2.4 could be improved with only minor modifications to the code.

It should be mentioned that for permutation routing, an important special case of the sorting problem, there is a much simpler $O(\log^2 p / \log\log p)$ time algorithm for the case $n = p$ than MultiWayMergeSort [Pel]. The idea is to route packets in a greedy fashion over sets of $\log\log p$ dimensions at a time. Each set of routings produces a load balancing problem in which there may be as many as $\log p$ packets at any one processor, and the objective is to redistribute the packets so that there is exactly one at each processor. Section 4.1.1 demonstrates how this redistribution can be performed in $O(\log p)$ time on the pipelined hypercube by making use of the pipelined prefix, broadcast and concentration operations described in this chapter.

Chapter 3

Boolean Matrix Multiplication

This chapter considers processor-efficient, optimal time implementations of the elementary Boolean matrix multiplication algorithm for the hypercube and shuffle-exchange. The phrase "elementary matrix multiplication algorithm" refers to the standard $O(n^3)$ time sequential algorithm for computing the product of two $n \times n$ matrices, as opposed to asymptotically faster (but in most cases less practical) methods due to Coppersmith and Winograd [CW82][CW87], Schönhage [Sch82], and Strassen [Str69][Str86].

The problem of implementing general matrix multiplication on the hypercube and shuffle-exchange was studied extensively by Dekel, Nassimi and Sahni [DNS81]. For the special case of Boolean matrix multiplication, Agerwala and Lint have given a parallel implementation of the four Russians' algorithm which runs in $O(\log n)$ time using $O(n^3/(\log n \log\log n))$ processors [AL78]. This chapter provides $O(\log n)$ time hypercube and shuffle-exchange implementations of Boolean matrix multiplication that improve this processor bound to $O(n^3/(\log^2 n \log\log n))$.

3.1 The Basic Algorithm

Let $n \times n$ Boolean matrices $A = (a_{ij})$ and $B = (b_{ij})$ be given. Letting $C = (c_{ij})$ denote the matrix AB, the entries of C are given by the elementary formula

$$c_{ij} = \bigvee_{0 \le k < n} a_{ik} \wedge b_{kj}, \tag{3.1}$$

$0 \le i, j < n$. This relationship leads to a simple and well-known $O(\log n)$ time matrix multiplication algorithm running on a hypercube with n^3 processors. Some notation will be introduced before describing this algorithm.

Given a hypercube of dimension d, let every string α of length d over the alphabet $\{0, 1, *\}$ correspond to that set of processors for which the ID "matches" α in the natural sense. For example, in a hypercube of dimension 4, the string $*1*0$ corresponds to the 4 processors 0100, 0110, 1100 and 1110.[1] It is often convenient to specify such a d-bit string as a tuple of the form $(w_0 : \alpha_0,\ \ldots,\ w_{t-1} : \alpha_{t-1})$, where t and the w_i's are nonnegative integers, $\sum_{0 \le i < t} w_i = d$, and α_i is either $*$ or a w_i-bit integer. As one might suspect, such a tuple is intended to correspond to the string $\beta_0 \cdots \beta_{t-1}$, where β_i is the w_i-bit string corresponding to the binary representation of α_i if $\alpha_i \ne *$, and $*^{w_i}$ otherwise. For example, the tuple $(3:6,\ 4:*,\ 1:0)$ corresponds to the string $110****0$.

The basic $O(\log n)$ algorithm for Boolean matrix multiplication on a hypercube with n^3 processors can now be easily described. Let $x = \log n$, and note that each processor has a $3x$-bit ID. Assume that input bits a_{ij} and b_{ij} are initially stored in processor $(x:i,\ x:j,\ x:0)$.

After broadcasting over ID bits $[0, x)$ (the *rightmost* field), a copy of a_{ij} resides in each processor in the set $(x:i,\ x:j,\ *)$. Hence, it certainly resides in the particular processor $(x:i,\ x:j,\ x:j)$, and by broadcasting over bits $[x, 2x)$, a copy of a_{ij} can be sent to all processors of the form $(x:i,\ x:*,\ x:j)$. Similarly, b_{ij} can be routed to the set of processors $(x:*,\ x:j,\ x:i)$ in $O(x)$ time. At this point, note that processor $(x:i,\ x:j,\ x:k)$ contains a_{ik} and b_{kj}, $0 \le i, j, k < n$. Thus, the n^3 Boolean AND operations of Equation (3.1) can be performed in a single time step. The c_{ij}'s can now be computed by simply ORing over ID bits $[0, x)$. This takes $O(x)$ time and leaves c_{ij} in the desired output processor $(x:i,\ x:j,\ x:0)$. Thus, the entire algorithm runs in $O(\log n)$ time, as claimed. Furthermore, it can be easily adapted to run on the shuffle-exchange in the same asymptotic time bound.

3.2 A Simple Improvement

This section describes simple modifications to the preceding algorithm that reduce the processor requirement to $O(n^3/\log^2 n)$ without affecting the asymptotic time bound. Using

[1] Note that a string with s occurences of the symbol $*$ corresponds to a subcube of dimension s (hence, 2^s processors).

a more complicated scheme, Section 3.3.1 will reduce the processor requirement by a further factor of $\log\log n$.

Let $x = \log n$ and let $y = \log\log n$. For simplicity, assume that y is an integer. The algorithm of Section 3.1 will now be modified to run without (more than a constant factor) slowdown on a hypercube of dimension $3x-2y$. In the modified algorithm, physical processor p simulates the subcube $(3x-2y : p,\ 2y : *)$ in a hypercube of dimension $3x$ running the basic algorithm. Recall that the basic algorithm routes a copy of a_{ij} to the subcube $(x : i,\ x : *,\ x : j)$. In the modified algorithm, this computation is simulated as follows.

1. By broadcasting over dimensions $[0, x-y)$, a copy of the bit a_{ij} can be sent to each processor in the set $(x : i,\ x : j,\ x-2y : *)$. This takes $O(x)$ time.

2. From the previous step, processor $(x : i,\ x : j,\ x-2y : j \text{ div } x^2)$ certainly holds a copy of bit a_{ij}. By broadcasting over dimensions $[x, 2x-2y)$, this copy of bit a_{ij} can be sent to each processor in the set $(x : i,\ x-2y : *,\ 2y : j \bmod x^2,\ x-2y : j \text{ div } x^2)$. This takes $O(x)$ time.

3. At this point, every processor contains a single entry from the matrix A. This data may be viewed as a bit vector of length 1. By repeatedly concatenating the bit vector held at each processor with that of its neighbor over dimensions $[x-y, x)$, every processor in the set $(x : i,\ x-y : *,\ y : j \bmod x,\ x-2y : j \text{ div } x^2)$ ends up with a copy of the x entries of A, packed into a single (or, at least, some constant number of) $O(\log n)$-bit registers. This takes $O(y)$ time.

4. The length of the bit vector at each processor is now $x = \log n$, and as the bit vector held at each processor is repeatedly concatenated with that of its neighbor over dimensions $[x-2y, x-y)$, the number of memory words required to represent a bit vector doubles at each iteration. Thus, the amount of time required to complete each successive iteration also doubles, and the total time is proportional to the length in words of the bit vectors after the last iteration. Since there are y iterations, the bit vectors reach a length of x words, and the total time required to perform this set of concatenations is $O(x)$.

The preceding algorithm requires that each processor be capable of concatenating two $O(\log n)$ bit operands in constant time. This could be accomplished by performing an appropriate shift (or multiply) operation followed by an OR. The model of computation defined in Section 1.1 is not violated since the operands never exceed $O(\log n)$ bits in length.

The end result of applying the above procedure (which runs in $O(x)$ time) is that each processor in the subcube $S = (x : i,\ x : *,\ x - 2y : j \text{ div } x^2)$ holds the appropriate set of x^2 bits, namely, a_{ik} for those integers k given by the tuple $(x - 2y : j \text{ div } x^2,\ 2y : *)$. These bits are stored in x words of length x and, assuming that the concatentations have been performed in an appropriate manner, a copy of bit a_{ik} is stored in bit position r of word s at each processor in the set S, where k is the unique integer given by the tuple $(x - 2y : j \text{ div } x^2,\ y : r,\ y : s)$.

It will be convenient to specify certain sets of bit locations using tuple notation. In order to avoid confusion between sets of processors and sets of bit locations, square brackets will be used to denote bit locations. Let the bit location corresponding to position r of word s at processor p be denoted $[x - 2y : p,\ y : r,\ y : s]$. Using this notation, it should be apparent that the sequence of operations described above places a copy of a_{ij} in the set of "matrix A" bit locations given by $[x : i,\ x : *,\ x : j]$. A similar procedure can be used to route b_{ij} to the set of "matrix B" bit locations given by $[x : *,\ x : j,\ x : i]$. The n^3 AND operations of Equation (3.1) can now be performed in x time steps by ANDing together the x corresponding pairs of matrix A and matrix B words at each processor. In order to compute the c_{ij}'s efficiently, each processor first locally ORs together the $\log^2 p$ bits that it contains. This reduces the amount of relevant data to a single bit per processor, and takes $O(x)$ time. The remaining OR operations are performed over ID bits $[0, x - 2y]$ as in the basic algorithm.

Unlike the basic algorithm of Section 3.1, implementing the algorithm described in this section on the shuffle-exchange so that it runs in $O(\log n)$ time is not entirely straightforward. The problem is that once the data corresponding to array A, say, has been replicated to the point that every processor contains $O(\log^2 p)$ bits, the algorithm cannot afford to shuffle the data more than a constant number of bit positions. Hence, the data must be aligned[2] correctly just as the processors become "saturated". A second requirement is that the data corresponding to arrays A and B be aligned in the same way. Finally, the ORing will be too expensive unless the shuffle-exchange is aligned in such a way that the "k field" (the bit positions corresponding to k in Equation (3.1)) is a constant number of shuffles away from the exchange position. It is not hard to prove that these three requirements cannot

[2] The "alignment" of the data corresponds to the net number of shuffle operations that have been applied, modulo $\log p$.

be simultaneously satisfied if the i, j and k fields each consist of a contiguous set of ID bits. One solution to this dilemma is to *interleave* the embedding of the i, j and k fields among the $\log p$ ID bits. For example, bit positions $[0, 3x - 6y)$ can be alternately assigned to i, j and k (in ascending order of significance), and the bit positions $[3x - 6y, 3x - 2y)$ can be alternately assigned to the remaining bits of i and j. Note that it is actually not necessary to interleave the entire address space in order to allow the three alignment requirements to be simultaneously satisfied.

3.3 The Four Russians' Algorithm

The elementary sequential algorithm for multiplying two $n \times n$ Boolean matrices requires $O(n^3)$ bit operations. Arlazarov, Dinic, Kronrod and Faradzev gave a practical algorithm that reduces this number of bit operations to $O(n^3/\log n)$ [ADKF70]. A detailed description of the so-called Four Russians' algorithm may be found in Aho, Hopcroft and Ullman [AHU74]; the following section will assume that the reader is familiar with the Four Russians' algorithm. Note that under a uniform cost criterion, assuming $O(\log n)$-bit registers, the Four Russians' algorithm can be easily modified to run in $O(n^3/\log^2 n)$ time.

3.3.1 Parallel Four Russians'

The problem of parallelizing the Four Russians' Boolean matrix multiplication algorithm was considered previously by Agerwala and Lint [AL78], who exhibited an algorithm that runs in $O(\log n)$ time on a network with $O(n^3/(\log n \log\log n))$ processors. Note that the simple algorithm of Section 3.2 already yields an improvement over the result of Agerwala and Lint. The purpose of this section is to establish that the Four Russians' approach can be combined with the techniques of Section 3.2 in order to obtain $O(\log n)$ time algorithms for the hypercube and shuffle-exchange using only

$$O\left(\frac{n^3}{\log^2 n \log\log n}\right)$$

processors. The hypercube algorithm will now be presented in detail, followed by an indication of the modifications necessary to achieve the same asymptotic performance on the shuffle-exchange.

Let x, y and z denote $\log n$, $\log\log n$ and $\log\log\log n$, respectively. In order to simplify the exposition, y and z will be assumed to be integers and round-off errors will be ignored.

$$
\begin{aligned}
&[x : i,\ x : j,\ x - z : 0] \\
&\quad \longrightarrow [x : i,\ x : j,\ x - 2y - z : *,\ y - z : 0,\ z : *,\ y : 0] \\
&\quad \longrightarrow [x : i,\ x - z : j \text{ div } y,\ x - 2y : *,\ y - z : 0,\ z : j \bmod y,\ y : 0] \\
&\quad \longrightarrow [x : i,\ x - 2y : *,\ x - z : j \text{ div } y,\ y - z : 0,\ z : j \bmod y,\ y : 0] \\
&\quad \longrightarrow [x - y : i \text{ div } x,\ x - y : *,\ x - z : j \text{ div } y,\ y - z : 0,\ z : j \bmod y,\ y : i \bmod x]
\end{aligned}
$$

Figure 3.1: The path followed by the a_{ij}'s.

For example, it will be assumed that n/y is an integer. It is straightforward to verify that the algorithm can be modified to handle round-off errors.

The task at hand is to multiply two $n \times n$ Boolean matrices in $O(\log n)$ time on a hypercube with $O(n^3/(x^2 y))$ processors, that is, a hypercube of dimension $3x - 2y - z$. As in Section 3.1, it will be assumed that input bits a_{ij} and b_{ij} are initially stored at processor $(x : i,\ x : j,\ x - 2y - z : 0)$, and that output bit c_{ij} should appear at the same processor, $0 \le i, j < n$.

As in Section 3.2, each processor will store up to x^2 bits of data at some point during the computation. The bit location corresponding to position r of word s at processor p will be denoted $[x - 2y - z : p,\ y : r,\ y : s]$.

As in Sections 3.1 and 3.2, the first phase of the algorithm consists of permuting and replicating the elements of the two input arrays A and B. Figure 3.1 indicates the four stage path followed by the a_{ij}'s during the first phase. Note that the low order y bits remain 0 until the last stage, implying that there is only one word of relevant data at each processor during the first three stages. Thus, the first three stages take a total of $O(x)$ time. The final stage builds up x-word bit vectors at each processor, and also takes $O(x)$ time. Figure 3.2 gives the somewhat more complicated seven stage path followed by the b_{ij}'s during the first phase. Once again, one may verify that the total cost of all stages is $O(x)$.

It remains to show how to compute the c_{ij}'s in $O(x)$ time given that the arrays A and B are stored in the manner indicated by the last tuples of Figures 3.1 and 3.2, respectively. At the end of the first phase, the processor given by the tuple $(x - y : r,\ x - y : s,\ x - z : t)$ holds the $2xy$ array elements a_{ik} and b_{kj} for $rx \le i < (r+1)x$, $sx \le j < (s+1)x$, and $ty \le k < (t+1)y$. In the second phase of the algorithm, the task of this processor is to perform the set of AND and OR operations associated with this set of array elements. The Four Russians' technique is used to perform these operations in $O(x)$ time, as follows. Note

$$
\begin{aligned}
&[x : i,\ x : j,\ x-z : 0] \\
&\quad \longrightarrow [x : i,\ x : j,\ x-2y-z : *,\ y : *,\ y : 0] \\
&\quad \longrightarrow [x : i,\ x-y : j \operatorname{div} x,\ x-y-z : *,\ y : j \bmod x,\ y : 0] \\
&\quad \longrightarrow [x-y-z : i \operatorname{div} xy,\ y : *,\ z : i \bmod y,\ x-y : j \operatorname{div} x, \\
&\qquad\qquad\qquad x-2y-z : *,\ y : (i \bmod xy) \operatorname{div} y,\ y : j \bmod x,\ y : 0] \\
&\quad \longrightarrow [x-y-z : i \operatorname{div} xy,\ z : i \bmod y,\ y : *,\ x-y : j \operatorname{div} x, \\
&\qquad\qquad\qquad x-2y-z : *,\ y : (i \bmod xy) \operatorname{div} y,\ y : j \bmod x,\ y : 0] \\
&\quad \longrightarrow [x-y-z : i \operatorname{div} xy,\ z : i \bmod y,\ x-y : j \operatorname{div} x, \\
&\qquad\qquad\qquad x-y-z : *,\ y : (i \bmod xy) \operatorname{div} y,\ y : j \bmod x,\ y : 0] \\
&\quad \longrightarrow [x-y-z : *,\ z : i \bmod y,\ x-y : j \operatorname{div} x,\ x-z : i \operatorname{div} y,\ y : j \bmod x,\ y : 0] \\
&\quad \longrightarrow [x-y : *,\ x-y : j \operatorname{div} x,\ x-z : i \operatorname{div} y,\ y : j \bmod x,\ y-z : 0,\ z : i \bmod y]
\end{aligned}
$$

Figure 3.2: The path followed by the b_{ij}'s.

that the b_{kj}'s are stored in y words of x bits apiece. There are $2^y = x$ possible words that can be obtained by ORing together a subset of these y words. A table T of these x words is computed, where the lth entry in the table (denoted $T(l)$) corresponds to the subset given by the binary representation of l. For example, if $y = 4$ and $l = 7 = 0111_2$, then $T(7)$ is obtained by ORing together words 0, 1 and 2 (but not word 3). Note that the table T can be constructed in $O(x)$ time.

The motivation for computing T is that now the x Boolean values given by

$$\bigvee_{ty \le k < (t+1)y} a_{ik} \wedge b_{kj},\ rx \le i < (r+1)x,$$

can be computed in a single table lookup for any fixed value of j, $sx \le j < (s+1)x$. Namely, one may check that these bits are given by $T(u)$, where u is the y-bit integer $u_{y-1} \cdots u_0$ with $u_v = a_{i(ty+v)}$. Note that the first phase has already constructed the word u at the appropriate processor. Thus, $O(x)$ time suffices to perform all of the AND and OR operations local to any particular processor.

The third phase of the algorithm consists of performing the remaining OR operations and routing the c_{ij}'s to the appropriate output processors. At the beginning of the third phase, each processor holds x^2 relevant bits of information. Unlike the algorithm of Section 3.2, the bits cannot be ORed together locally in order to immediately reduce the amount of data at each procesor to a single bit. The reason is that the $2y$ dimensions being simulated within the processors correspond to the i and j fields, rather than to the k field. However, the amount of data per processor can still be reduced geometrically by ORing across appropriate physical dimensions in the k field. This takes $O(x)$ time, and once the data has been reduced to a single bit per processor, the remainder of the third phase can be easily completed in $O(x)$ time. Hence, the overall running time of the algorithm is $O(x)$, as claimed.

As in Section 3.2, it is possible to adapt this algorithm to run in $O(x)$ time on the shuffle-exchange by appropriately interleaving the embedding of the i, j and k fields among the $\log p$ ID bits. The details are left to the reader.

3.4 Summary

This chapter has presented $O(\log n)$ time, $O(n^3/(\log^2 n \log\log n))$ processor implementations of the elementary Boolean matrix multiplication algorithm. Interleaving fields of ID bits leads to efficient performance on the shuffle-exchange; this technique may be more generally useful. Note that the $O(n^3/\log^2 n)$ processor implementation of Section 3.2 pipelines the standard Boolean matrix multiplication algorithm to the maximum possible extent. To see this, note that there are $\Theta(n^3)$ bit operations to be performed, and that each processor can perform at most $O(\log n)$ bit operations per time step (the register size), and hence at most $O(\log^2 n)$ bit operations in $O(\log n)$ time.

While the sequential Four Russians' algorithm saves a factor of $\log n$ time, the parallel version described in this chapter (as well as that of Agerwala and Lint [AL78]) reduces the processor requirement by only a $\log\log n$ factor. The reason for this is that a parallel algorithm that runs in $O(\log n)$ time can only afford to build tables of length $O(\log n)$ at each processor. The tables constructed by the sequential Four Russians' algorithm are of length n, allowing $\log n$ bit computations to be performed by a single table lookup.

The algorithm of Section 3.2 was coded up on an NCUBE/ten parallel computer as a sample application program within a virtual processing environment [Pla87].

Chapter 4

Load Balancing

Maintaining a balanced load is of fundamental importance on any parallel computer, since a strongly imbalanced load often leads to low processor utilization. In this chapter, two load balancing operations will be considered: Balance and MultiBalance. The Balance operation corresponds to the token distribution problem considered by Peleg and Upfal [PU89] for certain expander networks. The MultiBalance operation balances several populations of distinct token types simultaneously.

These load balancing operations form the basis of the selection algorithm given in Section 5.3, and of the sorting algorithms presented in Chapter 7.

4.1 Problem Definition: Balance

The first load balancing problem to be considered, Balance, is defined as follows. Let n tokens be distributed over p processors, with no more than m tokens assigned to any single processor, $\lceil n/p \rceil \leq m \leq n$. It will be assumed that $n = O(p^c)$ for some constant c in order that calculations involving token counts can be performed with a constant number of CPU operations. The problem is to redistribute the tokens so that each processor has either $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$ tokens, that is, so that the load is distributed as evenly as possible. Peleg and Upfal have exhibited tight bounds for this operation on a certain class of expander networks [PU89]. In many applications, it is not necessary to balance the population of tokens exactly. If the difference between the maximum number of tokens at any processor and the minimum number of tokens at any processor is ξ, it will be said that the tokens have been *balanced with error* ξ. The corresponding operation will be referred to as Balance *with error* ξ.

4.1.1 Tight Bounds for the Pipelined Hypercube

This section describes an algorithm for the Balance operation with minimum error that runs in $O(m \log p)$ time on the hypercube or shuffle-exchange, and in $O(m + \log p)$ time on the pipelined hypercube. This algorithm is due to Tom Leighton [Lei]. Assuming that m exceeds $\lceil n/p \rceil$ by at least some constant factor, there is a trivial $\Omega(m + \log p)$ lower bound for the pipelined hypercube. The $\Omega(m)$ term in the lower bound holds since at least $m - \lceil n/p \rceil = \Omega(m)$ time steps are necessary for a processor initially holding m tokens to send away sufficiently many tokens to reach $\lceil n/p \rceil$, the maximum allowable number in any balanced configuration. The $\log p$ term in the lower bound holds because, as will be proven rigorously in Section 4.1.2, it is possible to configure the tokens in such a way that no token is placed within $\Omega(\log p)$ hops of a particular processor. Thus, Leighton's algorithm provides tight bounds for the pipelined hypercube when m exceeds $\lceil n/p \rceil$ by some constant factor. Note that if m does not exceed $\lceil n/p \rceil$ by a factor of 2 (say), then load balancing is probably not necessary anyway.

The 1-port version of Leighton's algorithm will now be described. The algorithm runs in m phases, and each phase takes care of one token from every processor for which the supply of tokens has not yet been exhausted. In a phase, the designated tokens are routed to a contiguous block (with respect to processor ID modulo p) of processors. Each token is routed in exactly one phase. The first block begins at processor 0 (say), and each subsequent block begins at the processor following the end of the previous block. In this manner, the population of tokens gets distributed as evenly as possible.

A single phase of Leighton's algorithm is implemented by performing a prefix sum over the designated tokens followed by a concentration route as defined by Nassimi and Sahni [NS82]. The prefix sum gives the offset of each token within the contiguous block of destination processors. The size of the block is broadcast so that all processors can compute the start of the next block. All of these operations can be performed in $O(\log p)$ time , so the over-all running time of Leighton's algorithm is $O(m \log p)$. Note that this time bound is valid for the shuffle-exchange as well as the hypercube.

As indicated above, Leighton's algorithm is best suited for the pipelined hypercube.

The prefix sum and broadcast operations can be pipelined on the hypercube but the concentration routes cannot (provably). On the other hand, the pipelined hypercube allows concentration routes to be pipelined [VD88]. Hence, the Balance operation can be implemented to run in $O(m + \log p)$ time on the pipelined hypercube.

4.1.2 A Lower Bound for the Hypercube

A lower bound for the running time of Balance on the hypercube can be obtained by concentrating all of the tokens in a small number of processors and then bounding the time required to eliminate the excess tokens from this set of processors. In the following, recall that $d = \log p$ denotes the dimension of the given hypercube.

Definition 4.1.1 Let $\Gamma^r(i)$ denote the set of $\binom{d}{r}$ processors at Hamming distance r from processor i, $0 \leq r \leq d$.

Definition 4.1.2 Let $B(i, r)$ denote the *complete Hamming ball* of radius r centered at processor i. More formally, this is the set of processors given by $B(i, r) = \cup_{0 \leq l \leq r} \Gamma^l(i)$.

Note that $|B(i, r)| = \sum_{0 \leq l \leq r} \binom{d}{l}$. It will also be necessary to consider "incomplete" Hamming balls, that is, Hamming balls with only a partially filled outer layer.

Definition 4.1.3 Given a positive integer x, let r and y be the unique pair of nonnegative integers such that $x = y + \sum_{0 \leq l \leq r-1} \binom{d}{l}$, $1 \leq y \leq \binom{d}{r}$. A set of processors B is a *Hamming ball of size x and radius r* if there is some processor i and some set of processors S such that $B = B(i, r-1) \cup S$, $S \subseteq \Gamma^r(i)$ and $|S| = y$. Let $\mathcal{B}(x)$ denote the set of all Hamming balls of size x.

Definition 4.1.4 Let a graph G with vertex set $V(G)$ and edge set $E(G)$ be given. For every $U \subseteq V(G)$, the *fringe of U with respect to G*, $\mathcal{F}(G, U)$, is defined as the set

$$\{u \in U \mid (u, v) \in E(G) \text{ for some } v \in V(G) \setminus U\}.$$

Finally, let the function $f(G, x)$ be defined as

$$f(G, x) = \min_{\substack{U \subseteq V(G) \\ |U| = x}} |\mathcal{F}(G, U)|,$$

where G is a graph and x is an integer, $0 \leq x \leq |V(G)|$.

Lemma 4.1.1 The Balance operation requires $\Omega((n - \lceil n/m \rceil \lceil n/p \rceil)/f(G, \lceil n/m \rceil))$ time.

Proof: The n tokens can be concentrated in a set S of $\lceil n/m \rceil$ processors. Under a balanced load, S contains at most $\lceil n/m \rceil \lceil n/p \rceil$ tokens. Hence, at least $n - \lceil n/m \rceil \lceil n/p \rceil$ tokens must leave the set S. Only processors located at the fringe of S, those in $\mathcal{F}(G, S)$, can send tokens out of S, and these can only transmit one token per time step. Therefore, the running time of Balance must be at least $(n - \lceil n/m \rceil \lceil n/p \rceil)/(f(G, \lceil n/m \rceil))$. □

Having established Lemma 4.1.1, the desired lower bound can now be obtained by computing $f(H_d, \lceil n/m \rceil)$, where H_d denotes the undirected graph corresponding to a hypercube of dimension d. Intuitively, one might expect that the value of $f(H_d, x)$ is determined by a Hamming ball configuration. The correctness of this intuition is borne out by the following theorem due to Harper [Har66]. Note that Frankl and Füredi have given a simpler proof of the same result [FF81].

Theorem 4.1.1 For every integer x, $0 \leq x \leq p$, there exists a ball $B \in \mathcal{B}(x)$ such that $f(H_d, x) = |\mathcal{F}(H_d, B)|$. □

The following estimate of the "volume-to-surface" ratio of a Hamming ball is proven in Appendix A.

Lemma 4.1.2 For positive integers d and $r = r(d)$, $0 \leq r \leq d/2$, let $S = \binom{d}{r}$ and let $V = \sum_{0 \leq l \leq r} \binom{d}{l}$. Then $V = 2^{d(1-1/k)}$ implies that $V/S = \Theta(k^{1/2})$, $1 \leq k \leq d$. □

Theorem 4.1.2 The Balance operation requires $\Omega(k^{1/2}m)$ time on the hypercube if $m = \Theta(p^{1/k}(n/p))$ and $m \geq 2n/p$.

Proof: Note that $k \geq 1$ since $m \leq n$, and $k \leq \log p$ since $m \geq 2n/p$. Theorem 4.1.1 and Lemma 4.1.2 together imply that

$$f(H_d, p^{1-1/k}) = \Theta(k^{-1/2}p^{1-1/k}),$$

$1 \leq k \leq \log p$. Now consider the lower bound given by substituting this equation into the statement of Lemma 4.1.1. The numerator, $n - \lceil n/m \rceil \lceil n/p \rceil$ is at least $n - \lceil p/2 \rceil \lceil n/p \rceil = \Omega(n)$. The denominator, $f(G, \lceil n/m \rceil)$, reduces to $f(H_d, p^{1-1/k}) = \Theta(k^{-1/2}p^{1-1/k})$. Hence, Balance requires $\Omega(k^{1/2}(n/p)p^{1/k}) = \Omega(k^{1/2}m)$ time on the hypercube. □

4.1.3 Upper Bounds for the Hypercube

Now consider the task of obtaining an efficient implementation of the Balance operation on the hypercube. Let a subcube of dimension 1 be given, with a tokens at processor 0 and b tokens at processor 1. Further assume that each processor knows the number of tokens that it is holding, that is, the value a (b) is stored in the local memory of processor 0 (1). In this case, it is easy to see that the Balance operation can be performed over the given subcube in $|a-b|/2+O(1)$ time. This observation motivates the following definition.

Definition 4.1.5 Let a set of n tokens be arbitrarily distributed over the processors of a p processor hypercube, $p \geq 2$. Let the Balance operation be applied to each of the $p/2$ subcubes of dimension 1 induced by the set of $p/2$ edges across dimension i. Then the hypercube has been *balanced across dimension i*.

Clearly the amount of time required to balance across dimension i is $\xi/2+O(1)$, where ξ is the maximum discrepancy between the number of tokens at a given processor and its neighbor in dimension i.

Lemma 4.1.3 Let the low and high subcubes with respect to dimension i of a given hypercube be balanced with error ξ. Then after balancing across dimension i, the entire hypercube is balanced with error $\xi+1$.

Proof: Initially, each processor in the low subcube contains a number of tokens in the range $[a, a+\xi]$ for some integer a. Similarly, each processor in the high subcube contains a number of tokens in the range $[b, b+\xi]$ for some integer b. Thus, after balancing across dimension i every processor contains a number of tokens in the range $\left[\left\lfloor\frac{a+b}{2}\right\rfloor, \left\lceil\frac{a+b+2\xi}{2}\right\rceil\right]$, and

$$\left\lceil\frac{a+b+2\xi}{2}\right\rceil = \left\lceil\frac{a+b}{2}\right\rceil + \xi \leq \left\lfloor\frac{a+b}{2}\right\rfloor + \xi + 1,$$

which completes the proof. □

By repeated application of Lemma 4.1.3, one finds that successively balancing across every dimension of the hypercube yields an implementation of Balance with error $\log p$. The running time of this algorithm is $O(m \log p)$, since no processor will ever contain more than m tokens and hence each balancing step requires at most $m/2+O(1)$ time. Furthermore, in the important case $m = O(n/p)$, it is possible to distribute the tokens so that this performance is achieved to within a constant factor. In other words, the worst case running time of such a balancing algorithm is $\Theta(m \log p)$ when $m = O(n/p)$.

The following algorithm improves on this time bound by making a more careful decomposition of the hypercube. One additional definition is needed in order to present the algorithm.

Definition 4.1.6 Let the *discrepancy across dimension* i, denoted δ_i, represent the absolute value of the difference between the total number of tokens in the high and low subcubes with respect to dimension i.

The efficiency of the following recursive procedure for Balance with error $\log p$ relies upon the fact (shown below) that there is always some dimension with a small associated discrepancy.

Algorithm Balance

1. Each processor counts how many tokens it has and stores the result. This takes $O(m)$ time.

2. Let l denote the dimension of the subcube being balanced ($l = d$ initially). If $l = 0$, return.

3. Compute δ_i, $0 \leq i < l$. This involves performing l independent sums over the entire subcube. These sum operations can be pipelined to run in a total of $O(l)$ time [HJ86].

4. Determine the dimension i^* with least associated discrepancy δ_{i^*}. This takes $O(l)$ time.

5. Recursively balance the high and low subcubes with respect to dimension i^*, using Steps 2 to 6.

6. Balance across dimension i^*, adjusting the token counts appropriately. The running time of this step is analyzed below.

In order to establish the correctness of the preceding algorithm, it is necessary to prove that the output hypercube is balanced with error $\log p$. This follows easily by induction using Lemma 4.1.3.

To analyze the time complexity, only the cost of Step 6 remains to be determined. When this step is executed, the $(l-1)$-dimensional high and low subcubes with respect to dimension i^* are each balanced with error $l-1$. Hence, there are integers a and b such that each processor in the high subcube contains a number of tokens that is in the range $[a, a+l-1]$, and each processor in the low subcube contains a number of tokens in the range $[b, b+l-1]$. Letting $\delta = \delta_{i^*}$ gives

$$2^{l-1}(b-(a+l-1)) \leq \delta,$$

and so $(b+l-1)-a \leq \delta 2^{1-l}+2l-2$. Similarly, $(a+l-1)-b$ can be bounded by the same quantity. Therefore, the cost of the balancing step is $O(\delta 2^{-l}+l)$. It remains to bound the minimum discrepancy δ. In the following sequence of lemmas, let Δ denote the sum of the discrepancies in the given hypercube of dimension d, $\sum_{0\leq i<d} \delta_i$.

Lemma 4.1.4 The value of Δ is maximized by packing all of the tokens into a Hamming ball B of size $\lceil n/m \rceil$.

Proof: Given an arbitrary distribution of the tokens, it will be shown that the corresponding value of Δ is at most that achieved by a particular Hamming ball configuration. First, transform the processor IDs of the given hypercube by toggling each bit that corresponds to a dimension for which there are more tokens in the low subcube than in the high subcube. Note that the transformed hypercube yields precisely the same value of Δ as the given hypercube. It has the additional property that for every dimension, the high subcube contains at least as many tokens as the low subcube. Let $w(i)$ denote the number of 1's in the d-bit processor ID i, and let $n(i)$ denote the number of tokens at processor i. Now observe that Δ may be expressed as

$$\begin{aligned}\Delta &= \sum_{0\leq i<p} w(i)n(i) - \sum_{0\leq i<p} (d-w(i))n(i) \\ &= 2\sum_{0\leq i<p} w(i)n(i) - nd,\end{aligned}$$

where $p = 2^d$. Thus, maximizing Δ is equivalent to maximizing $\sum_{0\leq i<p} w(i)n(i)$. This new sum is clearly maximized by distributing tokens according to the following algorithm: While there are tokens left to distribute, put m tokens (or the number of tokens remaining, if less than m) into an empty processor with largest $w(i)$ in the set of empty processors. The result follows since this algorithm fills a Hamming ball centered at processor 2^d-1. $\square$

Lemma 4.1.5 Let an instance of Balance be given for which the tokens are packed into a Hamming ball of size $\lceil n/m \rceil$. Then $\Delta = O(md^{1/2}p)$. Furthermore, if $m \geq 2n/p$ and $\lceil n/m \rceil = p^{1-1/k}$, then $\Delta = \Theta(mdk^{-1/2}p^{1-1/k})$.

Proof: Assume the tokens are packed into a Hamming ball B of radius r. The total discrepancy Δ is bounded by m times the number of edges between B and $H_d \setminus B$. The number of such edges is maximized when $r = \lfloor d/2 \rfloor$, so $\Delta = O(md\binom{d}{\lfloor d/2 \rfloor}) = O(md^{1/2}p)$. For the sharper bound, Lemma 4.1.2 tells us that the cardinality of the fringe of B is $f(H_d, p^{1-1/k}) = \Theta(k^{-1/2}p^{1-1/k})$. The number of edges between B and $H_d \setminus B$ is $d - r = \Theta(d)$ (r is at most $\lfloor d/2 \rfloor$ since $m \geq 2n/p$) times the size of the fringe of B. Hence, $\Delta = \Theta(mdk^{-1/2}p^{1-1/k})$, as claimed. □

This section will only make use of the $O(md^{1/2}p)$ bound of Lemma 4.1.5. The more detailed bound involving k will be needed in the next section in order to analyze a load balancing algorithm involving multiple token types. The following theorem is an immediate consequence of the preceding two lemmas.

Theorem 4.1.3 For any instance of Balance, the average discrepancy across a dimension, Δ/d, is $O(md^{-1/2}p)$. □

Recall that the cost of the balancing step in algorithm Balance was shown above to be $O(\delta 2^{-l} + l)$, where $\delta = \delta_{i^*}$ is the minimum discrepancy. Now the minimum discrepancy is certainly no larger than the average discrepancy, so δ must be $O(ml^{-1/2}2^l)$ by Theorem 4.1.3. Hence, the cost of the balancing step is $O(ml^{-1/2} + l)$, and the total running time of algorithm Balance is $O(\sum_{1 \leq l \leq d} ml^{-1/2} + l) = O(m \log^{1/2} p + \log^2 p)$.

Of course, this algorithm performs balancing with error $\log p$. This should be good enough for most applications, but if a minimum error (0 if $p|n$, 1 otherwise) balancing is desired, some post-processing is needed. Note that the post-processing task can be viewed as an instance of Balance with $m = \log p$, which can be solved in $O(\log^2 p)$ time using the 1-port version of Leighton's algorithm described in Section 4.1.1.

Theorem 4.1.4 The Balance operation, with minimum error, runs in $O(m \log^{1/2} p + \log^2 p)$ time on the hypercube.

Proof: First apply algorithm Balance described and analyzed earlier in this section to balance the load with error $\log p$. This takes $O(m \log^{1/2} p + \log^2 p)$ time. Compute the minimum number of tokens, a, at any processor and broadcast this value to all processors. This takes $O(\log p)$ time. Every processor then sets aside a tokens, and the remaining tokens (of which there are at most $\log p$ at any single processor) are balanced using Leighton's algorithm in $O(\log^2 p)$ time. □

4.1.4 Load Balancing on the Shuffle-Exchange

As noted in Section 4.1.1, the 1-port version of Leighton's algorithm runs on the shuffle-exchange. Hence, the Balance operation, with minimum error, runs in $O(m \log p)$ time on the shuffle-exchange. Now consider the following lower bound.

Theorem 4.1.5 Assuming $m \geq 2n/p$, the Balance operation requires $\Omega(m \operatorname{løg}(n/m))$ time on the shuffle-exchange.

Proof: Using techniques due to Leighton [Lei83], Cypher has proven that the p processors of a shuffle-exchange can be partitioned onto c chips in such a way that fewer than $p/2$ processors are assigned to any single chip, and the number of pins per chip is $O(p/(c \operatorname{løg}(p/c)))$ [Cyp93]. The pin count for a particular chip C is determined by the total number of edges joining a processor assigned to C to a processor assigned to some other chip. Letting SE_d denote the graph corresponding to the shuffle-exchange of dimension d, Cypher's bound implies (by an averaging argument) that for every integer x, $0 \leq x < p/2$, there exists an integer $g(x)$, $x \leq g(x) < p/2$, such that

$$f(SE_d, g(x)) = O(x/\operatorname{løg} x).$$

Now consider the lower bound for Balance obtained by packing the n tokens into a set S of $g(\lceil n/m \rceil)$ processors with $O(\lceil n/m \rceil / \operatorname{løg}\lceil n/m \rceil)$ neighbors. At least $n/2 = \Omega(n)$ tokens need to be moved to processors outside of the set S, and each edge leaving S can carry at most one token per time step. Hence, Balance requires $\Omega(m \operatorname{løg}(n/m))$ time on the shuffle-exchange if $m \geq 2n/p$. □

The upper and lower bounds are tight for $2n/p \leq m \leq n^{1-\epsilon}$, where ϵ denotes an arbitrarily small positive constant.

4.2 Problem Definition: MultiBalance

In this section, a slightly more complicated load balancing problem than Balance will be considered. Let n tokens be evenly distributed over p processors, that is, each processor contains either $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$ tokens. Each token has an associated *type*. There are $g \geq 2$ different types of tokens, and nothing is known about the distribution or proportion of the tokens of any particular type. The set of tokens of a particular type will be called a *group*, and it will be assumed that the g group types are given by the integers $\{0, \ldots, g-1\}$. The problem is to execute g Balance (with error ξ) operations, one over each group of tokens. This operation will be referred to as MultiBalance (with error ξ). The motivation for considering MultiBalance is that it turns out to be useful for sorting, as will be seen in Chapter 7.

4.2.1 Upper Bounds for the Hypercube

An implementation of MultiBalance that runs in $O((n/p)(\log g \log p)^{1/2}+g \log^2 p)$ time on the hypercube will now be presented. The following definitions, which build on Definitions 4.1.5 and 4.1.6, are required.

Definition 4.2.1 Given an instance of MultiBalance, the n tokens have been *multi-balanced across dimension* i if and only if each group j has been balanced across dimension i, $0 \leq j < g$.

Definition 4.2.2 Let δ_i^j denote the discrepancy across dimension i with respect to group j, $0 \leq j < g$. Define the *multi-discrepancy across dimension* i, denoted δ_i^M, as the sum $\sum_{0 \leq j < g} \delta_i^j$.

Algorithm MultiBalance

1. Each processor partitions its set of tokens into g subsets, one subset corresponding to each of the g token types. Each of the subsets is counted and the results are stored. This takes $O(n/p)$ time.

2. Let l denote the dimension of the subcube being multi-balanced ($l = d$ initially). If $l = 0$, return.

3. Compute δ_i^M, $0 \leq i < l$. This involves performing l independent sums over the entire subcube for each of the g groups. Each set of l sum operations can be pipelined to run in $O(l)$ time [HJ86], so the total running time is $O(gl)$.

4. Determine the dimension i^* with least associated multi-discrepancy $\delta_{i^*}^M$. This takes $O(l)$ time.

5. Recursively multi-balance the high and low subcubes with respect to dimension i^*, using Steps 2 to 6.

6. Multi-balance across dimension i^*, adjusting the token counts appropriately. The running time of this step is analyzed below.

The correctness of the preceding implementation of MultiBalance with error $\log p$ follows by induction using Lemma 4.1.3. If a minimum error multi-balancing is desired, $O(\log^2 p)$ post-processing per group can be performed as described in Section 4.1. The total cost of post-processing is thus $O(g \log^2 p)$ time.

In order to complete the analysis of the running time of algorithm MultiBalance, it is necessary to consider the cost of Step 6. Let Δ_j denote the sum of the discrepancies with respect to group j in the given hypercube of dimension d, $\sum_{0 \le i < d} \delta_i^j$. Let Δ^M denote the sum of the multi-discrepancies $\sum_{0 \le i < d} \delta_i^M = \sum_{0 \le j < g} \Delta_j$.

Lemma 4.2.1 For any instance of MultiBalance, $\Delta^M = O(dk^{-1/2}n)$, where k satisfies $g = p^{1/k}$.

Proof: Let the number of tokens in group j be denoted x_j, $0 \le j < g$, and consider the contribution of Δ_j to Δ^M. Since $\sum_{0 \le j < g} x_j = n$, there is at most one x_j that exceeds $n/2$. Suppose that $x_l \ge n/2$ for some group l. Then $\Delta_l = O((n/p)d^{1/2}p) = O(d^{1/2}n)$ by Theorem 4.1.3. Now $k \le \log p = d$ (since $g \ge 2$), so $d^{1/2} \le dk^{-1/2}$ and $\Delta_l = O(dk^{-1/2}n)$.

Hence, it may be assumed that $x_j \le n/2$, $0 \le j < g$. Let k_j satisfy $x_j/n = p^{-1/k_j}$, $0 \le j < g$. The tokens of group j can be packed into p^{1-1/k_j} processors, and by Lemmas 4.1.4 and 4.1.5, $\Delta_j = O((n/p)dk_j^{-1/2}p^{1-1/k_j}) = O(dx_jk_j^{-1/2}) = O(d^{1/2}x_j \log^{1/2}(n/x_j))$. Consider the function $f(x) = x \log^{1/2}(n/x)$, where x is a real value in the range $[1, n/2]$. One may easily verify that $f''(x) < 0$ in this range. In other words, $f(x)$ is a concave function. Therefore, the sum $\sum_{0 \le j < g} f(x_j)$, subject to the constraint $\sum_{0 \le j < g} x_j = n$, is maximized when all of the x_j's are equal, that is, when $x_j = n/g$. Forcing the x_j's to be integers can only decrease this sum, so $\Delta^M = O(d^{1/2}gf(n/g)) = O(d^{1/2}g(n/g)\log^{1/2} g) = O(dk^{-1/2}n)$, as required. □

Lemma 4.2.2 For any instance of MultiBalance, the average multi-discrepancy Δ^M/d across a dimension is $O(n(\log g/\log p)^{1/2})$.

Proof: Immediate from Lemma 4.2.1, since $g = p^{1/k}$ and $k = \log p/\log g$. □

Theorem 4.2.1 The MultiBalance operation runs in $O((n/p)(\log g \log p)^{1/2} + g \log^2 p)$ time on the hypercube.

Proof: By a simple extension of the argument given in Section 4.1, the time required to perform the multi-balancing step is $O(\delta_{i*}^M 2^{-l} + gl)$. Now δ_{i*}^M is certainly no more than Δ^M/d, and the number of tokens in a subcube of dimension l at depth $d-l$ of the recursion is $O(n2^{l-d} + g2^l(d-l))$, where the latter term bounds the cumulative effect of odd parity in the balancing operations. Thus, Lemma 4.2.2 implies that the total time expended in Step 6 is

$$O\left(\sum_{1\le l\le d} 2^{-l}(n2^{l-d} + g2^l(d-l))(\log g/l)^{1/2} + gl\right),$$

which reduces to $O((n/p)(\log g \log p)^{1/2} + g\log^2 p)$. This dominates the time required by all other steps of the algorithm, including the post-processing. □

4.2.2 A Lower Bound for the Hypercube

It is easy to see that $\frac{1}{2}(\Delta^M - dgp)/p$ is a lower bound on the running time of MultiBalance, since Δ^M can only decrease by $2p$ each time step and it is certainly less than dgp after the MultiBalance operation has been performed. Hence, exhibiting a particular input instance with a high value of Δ^M gives a good lower bound on the worst case running time of MultiBalance. Consider the input instance given by the following construction.

Assume for convenience that $g = 2^r$, $1 \le r \le d$, and let $q = \lfloor d/r \rfloor - 1$ or $\lfloor d/r \rfloor$, whichever is odd. Divide the first qr bits of each processor ID into r *fields* of q contiguous bits. The ith field determines the ith bit of an r-bit *condensed ID* according to the following rule, $0 \le i < r$. If the majority of the q bits in the ith field are 0, then the ith bit of the condensed ID is 0; otherwise, it is a 1. Note that since q is odd there will always be a strict majority of either 0's or 1's. Furthermore, symmetry implies that exactly 2^{d-r} processors share any particular condensed ID. In the following lemma, let U denote such a subset of 2^{d-r} processors.

Lemma 4.2.3 The number of hypercube edges from processors in U to processors outside of U is $\Theta(rq^{1/2}|U|)$.

Proof: By symmetry, it is sufficient to prove that the number of such edges associated with the first (say) field is $O(q^{1/2}|U|)$. Also, it may be assumed without loss of generality that the first bit of the condensed ID associated with U is a 0. Let U' denote the subset of the processors of U with $\lceil q/2 \rceil$ 0 bits and $\lfloor q/2 \rfloor$ 1 bits in the first field. Lemma A.1.2 implies that $|U'| = \Theta(q^{-1/2}|U|)$. It should be clear that only the processors in U' contribute to the desired edge count, and each of these contributes exactly $\lceil q/2 \rceil$. Thus, the number of edges leaving U that are associated with the first field is $O(q^{1/2}|U|)$, as required. $\Box$

Theorem 4.2.2 The MultiBalance operation requires $\Omega((n/p)(\log g \log p)^{1/2} - g \log p)$ time on the hypercube.

Proof: Consider the input configuration obtained by filling each of the processors having condensed ID i with n/p tokens from the ith group, $0 \leq i < 2^l$. Lemma 4.2.3 implies that each group contributes $\Theta((n/g)(\log g \log p)^{1/2})$ to Δ^M, so $\Delta^M = \Theta(n(\log g \log p)^{1/2})$. As argued above, this fact immediately implies the desired lower bound on the running time of the MultiBalance operation. $\Box$

4.2.3 Average Case Analysis

Given n distinct tokens and p processors, there are

$$\frac{n!}{[(n/p)!]^p}$$

different ways of assigning n/p tokens to each processor, assuming that n is an integer multiple of p. This section analyzes the average case running time of MultiBalance over all of these possible input configurations when there are g distinct groups of tokens. The upper bound to be derived will be interesting for sufficiently small values of g. In the following discussion, the phrase "with high probability" means with probability $1 - O(p^{-c})$ for an arbitrary positive constant c.

Let the ith group contain n_i tokens, and consider the expected contribution of group i to the total running time of this version of MultiBalance, $0 \leq i < g$. Letting $p = 2^d$, there are $\binom{d}{d'}$ distinct subcubes of dimension d'. Focus attention on one such subcube C, and let the random variable X denote the number of tokens from group i initially assigned to some processor in C. Let Y_j denote the random variable that is 1 if the jth token of group i contributes to X, and 0 otherwise, $0 \leq j < n_i$. Letting $\theta = 1/2^{d-d'}$, the expected value of Y_j is θ, and the expected value of X is $n_i\theta$. The variance of Y_j is bounded above by the variance of the binomial distribution with probability θ. Thus, the variance of X is at most $n_i\theta(1-\theta) \leq n_i\theta$. A standard Chernoff bound implies that with high probability, X is within $O((n_i\theta \log p)^{1/2})$ of its expected value. Since there are only $p = \sum_{0 \leq d' < d} \binom{d}{d'}$ choices for C, *every* subcube of dimension d' contains

$$\frac{n_i}{2^{d-d'}} \pm O\left(\sqrt{\frac{n_i \log p}{2^{d-d'}}}\right)$$

group i tokens with high probability, $0 \leq d' < d$. Thus, after balancing the group i tokens over subcubes of dimension d', *every* processor contains

$$\frac{n_i}{p} \pm O\left(\sqrt{\frac{n_i \log p}{p2^{d'}}} + d'\right)$$

group i tokens with high probability. The additive d' bounds the worst case error in the balancing, as given by Lemma 4.1.3. Note that this is a pessimistic estimate to apply to the average case behavior of MultiBalance, and could certainly be improved. Continuing the analysis, the preceding bound implies that the total cost of the jth multi-balancing operation performed by algorithm MultiBalance is

$$O\left(\sum_{0 \leq i < g} \sqrt{\frac{n_i \log p}{p2^j}} + j\right),$$

$0 \leq j < d$, with high probability. Summing over j and interchanging the order of summation, the cost of all of the multi-balancing operations is

$$O\left(\sum_{0 \leq i < g} \sqrt{(n_i/p) \log p} + g \log^2 p\right)$$

with high probability since the sum over j is dominated by the $j = 0$ term. The remaining sum is maximized by setting $n_i = n/g$, $0 \leq i < g$, which leads to a total multi-balancing cost of

$$O\left(\sqrt{(n/p) g \log p} + g \log^2 p\right)$$

with high probability. Note that the $g \log^2 p$ term matches the cost of post-processing and other computations performed by MultiBalance.

The preceding analysis can also be applied to the straightforward implementation of MultiBalance that multi-balances across each of the dimensions in ascending order. Furthermore, this version of MultiBalance can be made to run as efficiently on the shuffle-exchange as it does on the hypercube. For the shuffle-exchange version, shuffles are not performed by moving entire sets of n/p tokens, but rather by moving appropriate subsets of the tokens to make the composition of the set of tokens at each processor (that is, the number belonging to each group) the same as it would have been if a true shuffle had been performed. The total cost of simulating the shuffle operations in this manner is easily seen to be on the same order as that of the multi-balancing operations. This algorithm will be referred to as the shuffle-exchange version of algorithm MultiBalance. The following theorem summarizes the two main results of this section.

Theorem 4.2.3 The average running time of algorithm MultiBalance, as well as that of the shuffle-exchange version of algorithm MultiBalance, is

$$O\left(\sqrt{(n/p)g\log p} + g\log^2 p\right).$$

4.3 Summary

This chapter has described hypercube and shuffle-exchange algorithms for performing two load balancing operations: Balance and MultiBalance. For the Balance operation, lower bounds were derived by considering the particular input configuration obtained by packing the tokens into a smallest possible set of processors with low expansion. For the hypercube, an algorithm was given that matches the lower bound to within a multiplicative constant if $m \geq \max\{2n/p, \log^{3/2} p\}$ and $m = O(n/p)$. The lower bound for the shuffle-exchange is higher because the hypercube has better expansion properties than the shuffle-exchange. Tight upper and lower bounds were obtained for the shuffle-exchange for m in the range $2n/p \leq m \leq n^{1-\epsilon}$, where ϵ denotes an arbitrarily small positive constant.

Upper and lower bounds were given for the MultiBalance operation on the hypercube. These bounds are tight for $(n/p)(\log g \log p)^{1/2} = \Omega(g\log^2 p)$. A straightforward implementation of MultiBalance on the shuffle-exchange was also described. Finally, the average case complexity of the hypercube and shuffle-exchange implementations of MultiBalance was considered. Not surprisingly, these algorithms behave much better on average than they do in the worst case.

Chapter 5

Upper Bounds for Selection

This chapter describes three entirely different approaches to the problem of selection on the hypercube and shuffle-exchange. The first approach is based on the $O((\log\log n)^2)$ algorithm of Cole and Yap for the parallel comparison model [CY85]. The speed of that algorithm is based upon the fact that small sets of keys can be sorted very quickly. More formally, n keys can be sorted in constant time with n^2 processors on the parallel comparison model. There exists an analogous result for the hypercube and shuffle-exchange, namely, that n keys can be sorted in $O(\log n)$ time with n^2 processors.

The second approach is based on the straightforward EREW PRAM selection algorithm of Vishkin [Vis83]. The hypercube and shuffle-exchange implementations of this algorithm make use of the Balance operation described in Section 4.1. An optimal algorithm is obtained for the pipelined hypercube.

The third approach is not based on any previous parallel algorithm. The source of its efficiency is a sequential tradeoff between preprocessing and search time in a partial order due to Borodin et al. [BGLY81]. The lower bound proven in Chapter 6 establishes that, for a sufficiently large ratio of keys to processors, the running time of this algorithm is asymptotically optimal on a wide variety of networks.

5.1 Problem Definition: Select

The Select operation is defined as follows. Given n $O(\log p)$-bit keys and an integer k, $0 \leq k < n$, determine the kth largest key and broadcast it to all processors. It will be assumed that the set of n keys is initially balanced with minimum error, that is, each processor holds either $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$ keys. It will be assumed that $n = O(p^c)$ for some constant c, so that array indices and key counts can be manipulated with a constant number of CPU operations. Finally, it will be assumed that the n keys are *distinct*. The latter assumption is made without loss of generality, since ties can always be broken consistently by making use of the following convention.

1. If two keys originating at different processors are equal, the one originating from the processor with the higher ID is deemed to be larger.
2. If two keys originating at the same processor are equal, the one initially stored at the higher memory location is deemed to be larger.

This tie-breaking procedure appends $\lceil \log n \rceil = O(\log p)$ bits to each key, and thus produces at most a constant factor overhead.

Three selection algorithms will now be presented: SortSelect, BalanceSelect and Search-Select.

5.2 An Algorithm Based on Sorting

The first selection algorithm to be considered, SortSelect, is based on the existence of a fast sorting algorithm when the number of processors exceeds the number of keys by a polynomial factor. If $n > p$ then the algorithm will simulate $O(n)$ processors, incurring a slowdown penalty of $O(n/p)$. Hence, it suffices to provide a selection algorithm with running time $O(\log p \log\log p)$ for the case $n = p$ in order to establish the more general bound of $O((n/p) \log p \log\log p)$.

Algorithm SortSelect is an adaptation of the parallel selection algorithm of Cole and Yap [CY85]. That algorithm runs in $O(\log\log p)^2$ time on Valiant's parallel comparison model [Val75], and relies upon the fact that $p^{1/2}$ keys can be sorted in constant time on p processors to develop an $O(\log\log p)$ time approximation subroutine that achieves the following degree of accuracy. Given p processors and $n \leq p$ keys, it returns a key representing a "lower approximation" to the desired rank j key with rank j_l satisfying

$$j - \frac{n}{4(p/n)^{1/2}} \leq j_l \leq j, \tag{5.1}$$

so the approximation improves as p/n increases. Similarly, the approximation subroutine can be made to return an "upper approximation" with rank j_u satisfying

$$j \leq j_u \leq j + \frac{n}{4(p/n)^{1/2}}. \tag{5.2}$$

The ratio p/n increases dramatically with each successive application of the approximation subroutine because only $j_u - j_l$ keys survive to the next round. When p/n reaches $p^{1/2}$ or greater, the entire set of remaining candidates can be sorted, and the desired element found, in constant time.

It will now be shown that the approximation subroutine of Cole and Yap, which runs in $O(\log\log p)$ time in the parallel comparison model, can be implemented to run in $O(\log p)$ time on the hypercube or shuffle-exchange. The algorithm outlined below, NearSelect, computes a lower approximation satisfying Equation (5.1) in $O(\log p)$ time. The task of finding an upper approximation may be handled in an analogous manner. In the following, the p given "physical" processors are slowed down by a factor of 4096 in order to simulate $P = 4096p$ "virtual" processors. Physical processor i will simulate the 4096 virtual processors with IDs in the range $[4096i, 4096(i+1))$. Initially, each of the $n = p$ keys is "live", and the key located at physical processor i is considered to reside at virtual processor $4096i$.

Algorithm NearSelect

1. If $n \leq P^{1/2}$, sort the n keys in $O(\log p)$ time using MergeSort, return the jth key, and halt.

2. Note that P and n are both powers of 2, and $P \geq 4096n$. If P/n is an even power of 2, let r equal $(P/n)^{1/2}$. Otherwise, let r equal $(P/2n)^{1/2}$. Divide the n keys into s *short sets* of size r^2. Note that r and $s = n/r^2$ are both powers of 2. The kth key of the ith short set is located at virtual processor $ir^4 + kr^2$, $0 \leq k < r^2$. Sort each of the short sets independently in $O(\log r)$ time using MergeSort. The r^4 virtual processors used to sort the ith short set are those with IDs in the range $[ir^4, (i+1)r^4)$.

3. Kill off all but those n/r keys that have a rank in their short set that is an integer multiple of r. For short set i, the r surviving keys are located at virtual processors $ir^4 + kr^3$, $0 \leq k < r$. Let n equal n/r. Let j equal $\lfloor j/r \rfloor$. Go to Step 1.

It will now be proven that NearSelect has the required performance, that is, it terminates in $O(\log p)$ time and the rank j_l of the key ultimately returned by Step 1 is guaranteed to satisfy Equation (5.1). The following series of lemmas is very similar to that proven by Cole and Yap [CY85], though some minor changes have been made in order to accommodate the hypercube and shuffle-exchange implementations.

Lemma 5.2.1 NearSelect terminates in $O(\log p)$ time.

Proof: Let the values of n and r in the ith iteration of the loop be 2^{a_i} and 2^{b_i}, $i \geq 0$, respectively. Letting $p = 2^d$, note that $P = 2^{d+12}$, $a_0 = d$ and $b_0 = 6$. Furthermore, $a_{i+1} = a_i - b_i$ and the algorithm terminates when $a_i \leq (d+12)/2$. It is sufficient to prove that the sequence $\{b_i\}$ increases geometrically, since this would imply that the algorithm terminates within $O(\log d) = O(\log \log p)$ iterations. Now observe that

$$\begin{aligned} b_{i+1} &\geq \frac{(d+12) - (a_i - b_i) - 1}{2} \\ &\geq b_i + \frac{b_i - 1}{2} \\ &\geq \frac{17}{12} b_i, \end{aligned}$$

for all $b_i \geq 6$ and thus for all $i \geq 0$. To analyze the total running time of the loop, note that each iteration runs in $O(\log r)$ time so the cost of the last iteration will dominate. Hence, the total running time of the loop, and of NearSelect is $O(\log p)$. □

Observe that algorithm NearSelect has a recursive structure. Let S denote the set of live keys at the start of some iteration of the loop and let T denote the remaining set of $|S|/r$ live keys at the end of the iteration. To obtain a lower approximation to the jth key in S, NearSelect first checks $|S|$ to see whether or not it is small enough to be sorted in $O(\log p)$ time. If so, sorting is performed and the jth key is returned. If not, it returns a lower approximation to the $\lfloor j/r \rfloor$th key in the smaller set T, obtained recursively.

Lemma 5.2.2 Let a particular key belonging to $S \cap T$ have rank k in T and rank k' in S. Then $rk - rs + s \leq k' \leq rk$.

Proof: Associate with each key in T the $r-1$ immediately higher keys in its short set. A key with rank k in T can be greater than at most the k lowest keys in T plus their associated sets, in S. Hence, $k' \leq rk$. Similarly, a key with rank k in T must be greater than the k lowest keys in T and at least $k-s$ of their associated sets, in S. Hence, $k + (r-1)(k-s) \leq k'$.
□

Lemma 5.2.3 Let the lower approximation to the jth computed by NearSelect have actual rank j_l in S. Then $j - 8n/r \le j_l \le j$.

Proof: This result will be proven by induction on the number of iterations of the loop. If there is only one iteration, then the entire set S is sorted and $j_l = j$. If there is more than one iteration, then the induction hypothesis implies that the key returned as the approximation to the $\lfloor j/r \rfloor$th key in T has rank k in T satisfying

$$\lfloor j/r \rfloor - 8n'/r' \le k \le \lfloor j/r \rfloor,$$

where n' and r' are the values of n and r used in the next iteration, that is, $n' = n/r$ and r' is either $(P/n')^{1/2}$ or $(P/2n')^{1/2}$. Applying Lemma 5.2.2 one finds that

$$r\lfloor j/r \rfloor - rs + s - 8rn'/r' \le j_l \le r\lfloor j/r \rfloor.$$

The right inequality implies that $j_l \le j$. From the left inequality one may obtain

$$\begin{aligned} j_l &\ge r\lfloor j/r \rfloor - rs + s - \frac{8r(n/r)}{(P/(2n/r))^{1/2}} \\ &\ge (j - rs) - rs + s - 2^{7/2}(n/r)(nr/P)^{1/2} \\ &\ge j - 2rs - 2^{7/2}(n/r)(n/P)^{1/4} \\ &\ge j - 2n/r - 2^{1/2}(n/r) \\ &\ge j - 8n/r, \end{aligned}$$

as required. □

Corollary 5.2.3.1 The rank of the key returned by NearSelect, j_l, is guaranteed to satisfy Equation (5.1).

Proof: Use the preceding lemma and observe that

$$8n/r \le \frac{8n}{(P/2n)^{1/2}} \le \frac{n}{4(p/n)^{1/2}},$$

for $r \ge (P/2n)^{1/2}$ and $P = 4096p$. □

Algorithm SortSelect can now be stated. Note that the algorithm will work properly even if $n < p$, that is, if only a subset of the processors initially contain a key. Initially, each of the n keys is “live”.

Algorithm SortSelect

1. Count up the number of live keys, n, and broadcast n to all processors. This takes $O(\log p)$ time.

2. Use a prefix sum followed by a concentration route to move the live keys to the n lowest numbered processors, that is, to processors 0 through $n-1$. This takes $O(\log p)$ time.

3. Let n' be the least power of 2 that is greater than or equal to n. Put a dummy $+\infty$ key at each processor in the range n through $n'-1$. Let n equal n'. This takes constant time.

4. If $n \leq \sqrt{P}$, sort the n keys in $O(\log p)$ time using the MergeSort algorithm of Nassimi and Sahni [NS82], return the jth key, and halt.

5. Compute a lower approximation to the key of rank j by calling NearSelect. Compute an upper approximation in an analogous manner. Let these two keys have actual ranks j_l and j_u, respectively. This takes $O(\log p)$ time.

6. Broadcast the upper and lower approximations to all processors. This takes $O(\log p)$ time.

7. Kill off those keys that are less than the lower approximation or greater than the upper approximation. This takes constant time. Go to Step 1.

Each iteration of the main loop of algorithm SortSelect takes $O(\log p)$ time. A bound will now be established on the number of iterations.

Lemma 5.2.4 Algorithm SortSelect terminates within $O(\log\log p)$ iterations. Hence, SortSelect runs in $O(\log p \log\log p)$ time on the hypercube and shuffle-exchange.

Proof: Let $n = 2^{d-a_i}$ on the ith iteration of Step 4. Note that $a_0 = 0$ and $a_1 = 1$. Equations (5.1) and (5.2) imply that

$$\begin{aligned} d - a_{i+1} &\leq \left\lceil \frac{3}{2}(d-a_i) - \frac{1}{2}d - 1 \right\rceil, \\ &\leq \frac{3}{2}(d-a_i) - \frac{1}{2}d, \\ &\leq d - \frac{3}{2}a_i. \end{aligned}$$

Hence, $a_{i+1} \geq \left(\frac{3}{2}\right)^i$ for $i \geq 0$ and the algorithm will terminate within $O(\log d) = O(\log\log p)$ iterations. □

5.3 An Algorithm Based on Load Balancing

Algorithm SortSelect handled the case $n > p$ by simulating n processors, which incurs a multiplicative slowdown penalty of $O(n/p)$. Thus, algorithm SortSelect does not achieve optimal speedup for any value of the ratio n/p. On the other hand, certain parallel models of computation admit optimal speedup for selection when n/p is sufficiently large. For the EREW PRAM, Vishkin has exhibited a straightforward selection algorithm that achieves optimal speedup for $n = \Omega(p \log p \log\log p)$. This result has been improved by Cole, who obtained optimal speedup for $n = \Omega(p \log p \log^* p)$ [Col86].

Vishkin's algorithm is based on two ideas. First, if the set S is partitioned into p groups of size n/p, with a single processor assigned to each group, then the median of each group can be computed in $O(n/p)$ time sequentially, and the median of the resulting set of p medians (which can be obtained by using the fastest known selection algorithm for the case $n = p$) is guaranteed to be a constant fraction splitter for the set S. In other words, it is greater than a constant fraction of the keys in S, and also less than a constant fraction of the keys in S (the fraction is $\frac{1}{4}$). Hence, by computing the exact rank in S of this median of medians, a constant fraction of the set S can be discarded from further consideration. The second idea is that the keys which have not been discarded can be partitioned into p equal-sized groups in $O(n/p)$ time. Iterating this process of elimination and redistribution, one finds that the number of keys remaining decreases geometrically and the complexity of Vishkin's algorithm is $O(n/p + \log p \log(n/p))$, where Cole's parallel merge sort has been used to make the additive term small [Col88].

The selection algorithm to be presented in this section, BalanceSelect, represents an efficient implementation of Vishkin's algorithm for the hypercube and shuffle-exchange networks. At any given time, a key that has yet to be discarded by Vishkin's algorithm will be referred to as a *live key*. The routine NearSelect (defined in Section 5.2) will be used to compute a live key that is greater than, and also less than, some constant fraction of all of the live keys. This allows a constant fraction of the live keys to be discarded. However, there is no guarantee that any particular fraction of the live keys *within a particular processor* will be eliminated. In the second phase of each stage, Balance is used to redistribute the set of live keys uniformly over the p processors. A detailed description and analysis of algorithm BalanceSelect is given below. Initially, all keys are "live".

Algorithm BalanceSelect

1. Let S_i denote the set of live keys located at processor i, and let m_i denote the median of S_i. Let $S = \cup_{0 \le i < p} S_i$ and let $M = \{m_0, \ldots, m_{p-1}\}$. Let $s = \max_{0 \le i < p} |S_i|$. Since selection can be performed in linear time sequentially, each processor can compute m_i with an $O(|S_i|)$ time local computation. Hence, all of the medians can be computed in $O(s)$ time.

2. Run NearSelect over the set M with $j = \lfloor p/2 \rfloor$ in order to obtain an approximation to the median of S. Let m denote the key given by this approximation. Note that a constant fraction of the keys in S must rank lower (higher) than m. This takes $O(\log p)$ time.

3. Compute the rank j' of m in S and broadcast it to all processors. This operation takes $O(s + \log p)$ time.

4. If $j' = j$, return m and halt.

5. If $j' < j$, each processor i removes from S_i those keys which are less than m and j is set to $j - j' - 1$. If $j' > j$, those keys which are greater than m are removed and j is left unchanged. This takes $O(s)$ time.

6. Execute Balance over the remaining set of live keys. This takes $O(s \log p)$ time on the shuffle-exchange, $O(s \log^{1/2} p + \log^2 p)$ time on the hypercube and $O(s + \log p)$ time on the pipelined hypercube.

7. Determine whether or not any processor contains more than a single live key. This takes $O(\log p)$ time. If so, go to Step 1.

8. There are at most p live keys remaining, with 0 or 1 at each processor. Now use SortSelect to complete the selection. This takes $O(\log p \log\log p)$ time.

From the above analysis, the running time of each iteration of Steps 1 to 7 is dominated by the call to Balance in Step 6. Since s decreases geometrically from an initial value that is $O(n/p)$, the number of iterations required is $O(\text{løg}(n/p))$. The total running time of BalanceSelect is thus $O((n/p)\log p + \log p \log\log p)$ for the shuffle-exchange, $O((n/p)\log^{1/2} p + \log^2 p\,\text{løg}(n/p))$ for the hypercube, and $O(n/p + \log p \log\log p)$ for the pipelined hypercube. Note that the performance of BalanceSelect on the pipelined hypercube is optimal for $n \ge p \log p \log\log p$.

The asymptotic complexities of NearSelect and SortSelect are low, but hide rather large multiplicative constants. A more practical implementation of BalanceSelect would make use of BitonicSort to perform these selection operations. For sufficiently large values of the ratio n/p, this substitution has no effect on the asymptotic complexity of BalanceSelect.

5.4 An Algorithm Based on Search

The third and final selection algorithm to be considered, SearchSelect, obtains efficient performance by eliminating a constant fraction of the keys *at every processor* in each iteration. This is not accomplished by redistributing the keys as in algorithm BalanceSelect, but instead by searching for a more accurate approximation to the desired key. A detailed description and analysis of this algorithm will now be presented. Initially, all of the keys are "live".

Algorithm Select

1. Let S_i denote the set of live keys located at processor i, and let m_i denote the median of S_i. Let $S = \cup_{0 \le i < p} S_i$ and let $M = \{m_0, \ldots, m_{p-1}\}$. Let $s = \max_{0 \le i < p} |S_i|$. All of the medians can be obtained in $O(s)$ time using a linear time sequential method. Having found the medians, partition the set S_i into its upper and lower half. Continue this partitioning process to depth $\min\{\log(n/p), \log\log p\}$, that is, until S_i has either been fully sorted or has been split into $\log p$ subsets, each with approximately $s/\log p$ values. Build a binary tree of partition elements to facilitate searching. The total cost of this preprocessing is $O(s \min\{\log(n/p), \log\log p\})$. Given an arbitrary value, its rank in S_i can be determined in $O(\min\{\log(n/p), \log\log p\} + s/\log p)$ time by locating the correct subset and then looking at every key in that subset. This sequential tradeoff between preprocessing time and search time is well understood, see [BGLY81] and [KMR88].

2. Find that $m \in M$ with rank in S closest to j (if there is a tie, break it arbitrarily) and broadcast it to all processors. The key m can be computed in time $O(s + \log^2 p)$ as follows. First, let $m'_i = m_i$ at each processor i. Now sort the set $M' = \{m'_0, \ldots, m'_{p-1}\}$ so that $m'_i = m_k$, where m_k has rank i in M. This can be done using bitonic sort in $O(\log^2 p)$ time. Next perform a binary search over the set M' to determine m.

This requires $O(\log p)$ rank computations over S, each of which may be performed as follows.

(a) Let m' be that key in M' whose rank in S is currently required by the binary search. Broadcast m', along with the ID of the processor that it is stored in, to all processors. This takes $O(\log p)$ time.

(b) At each processor i, compute the rank r_i of m' in S_i. As observed above, the preprocessing performed in Step 1 allows this operation to be completed in $O(s/\log p + \min\{\log(n/p), \log\log p\})$ time.

(c) Sum the r_i values to obtain the rank of m' in S. This takes $O(\log p)$ time.

3. Let j' be the rank of m in S. If $j' = j$, return m and halt.

4. At each processor i, kill off those keys in S_i that cannot possibly have rank j in S. Let r_i be the rank of m in S_i as computed in Step 2b. Assume that $j' < j$; the case $j' > j$ is similar. All of the keys in S_i with ranks in S_i less than or equal to r_i can certainly be eliminated. If $m \geq m_i$, note that this has eliminated at least half of the keys in S_i. If $m < m_i$, then the keys in S_i with ranks greater than or equal to that of m_i can also be eliminated, since the rank of m_i in S must be greater than j in order to avoid contradicting the choice of m. Once again, and hence in all cases, the number of live keys in S_i is reduced by at least a factor of 2. Given the preprocessing performed in Step 1, this step can be performed in $O(s/\log p + \min\{\log(n/p), \log\log p\})$ time.

5. Set j to $j - \Delta$, where Δ is the total number of keys eliminated in Step 4 because their rank in S had to be less than j. Note that Δ can be computed and broadcast to all processors in $O(\log p)$ time. Go to Step 1.

The preceding analysis implies that each iteration of Steps 1 through 5 executes in $O(s \min\{\log(n/p), \log\log p\} + \log^2 p)$ time. Since $|S_i|$ is initially n/p and is cut at least in half every iteration, after $\text{løg}(n/p)$ iterations every S_i will contain at most one element, and the next m computed in Step 2 will have rank j in S. Since s decreases geometrically from an initial value of n/p, the total running time is $O((n/p)\min\{\log(n/p), \log\log p\} + \log^2 p \,\text{løg}(n/p)) = O((n/p)\log\log p + \log^2 p\,\text{løg}(n/p))$.

Note that the $(n/p)\log\log p$ term in the running time is entirely due to the cost of local preprocessing, as opposed to communication. The results of this section are summarized by the following theorem.

Algorithm	*Running Time*	*Transition Region*
MergeSort	$O(\log^2 p/\log(p/n))$	$n = p^{1-\Theta(1/\log\log p)}$
SortSelect	$O(\log p \log\log p)$	$n = \Theta(p)$
SortSelect	$O((n/p)\log p \log\log p)$	$n = \Theta(p\log p)$
SearchSelect	$O(\log^2 p \log\log p)$	$n = \Theta(p\log^2 p)$
SearchSelect	$O((n/p)\log\log p)$	

Table 5.1: Best known selection algorithms for the hypercube and shuffle-exchange.

Theorem 5.4.1 The SearchSelect algorithm runs on the hypercube and shuffle-exchange in $O((n/p)\log\log p + \log^2 p\,\text{løg}(n/p))$ time. If the values are given in locally sorted form, then SearchSelect runs in $O(\log^2 p\,\text{løg}(n/p))$ time. □

Note that the complexity of algorithm SearchSelect can be expressed in terms of the cost of the following primitive operations: sort of $n = p$ keys, broadcast and sum. Thus, it adapts easily to a variety of networks. To be precise, assume that a particular network is capable of sorting $n = p$ keys located one per processor in time T_1 and can perform broadcasting and summing operations in time T_2. Then the running time of algorithm SearchSelect may be written as

$$O((n/p)\log\log p + (T_1 + T_2\log p)\,\text{løg}(n/p)), \tag{5.3}$$

where the first term disappears if the keys are given in locally sorted form.

Consider the performance of algorithm SearchSelect on a number of common network families. For the butterfly, hypercube and shuffle-exchange, $T_1 = O(\log^2 p)$ and $T_2 = \Theta(\log p)$, so the second term in Equation (5.3) is $\log^2 p\,\text{løg}(n/p)$, and for $n = \Omega(p\log^2 p)$ the running time of SearchSelect is $O((n/p)\log\log p)$. For the d-dimensional mesh (d constant), $T_1 = T_2 = \Theta(p^{1/d})$, so the first term dominates for $n = \Omega(p^{1+1/d}\log^2 p/\log\log p)$. Let T_3 denote the time required for a given network to perform selection over $n = p$ keys located one at each processor. If $T_3 \log p < T_1$ (as in the case of the complete binary tree, for example) then the second term in Equation (5.3) can be reduced to $(T_2 + T_3)\log p\,\text{løg}(n/p)$ by implementing each of the $\log(n/p)$ binary searches over p medians with $\log p$ selection operations rather than a single sort.

5.5 Summary

Algorithm	*Running Time*	*Transition Region*
MergeSort	$O(\log^2 p/\log(p/n))$	$n = p^{1-\Theta(1/\log\log p)}$
SortSelect	$O(\log p \log\log p)$	$n = \Theta(p)$
BalanceSelect	$O(\log p \log\log p)$	$n = \Theta(p \log p \log\log p)$
BalanceSelect	$O(n/p)$	

Table 5.2: Best known selection algorithms for the pipelined hypercube.

This chapter has described and analyzed a number of selection algorithms for the hypercube and shuffle-exchange. Table 5.1 summarizes the running times of the best known selection algorithms over ascending ranges of the ratio n/p. For $n \leq p^{1-\Theta(1/\log\log p)}$, the fastest known selection method is Nassimi and Sahni's MergeSort algorithm. As the ratio n/p is increased beyond this point, SortSelect becomes the best known selection algorithm. Finally, algorithm SearchSelect overtakes SortSelect in the region $n = \Theta(p \log p)$. The ranges of n/p have been further subdivided at $n = \Theta(p)$ and $n = \Theta(p \log^2 p)$ in order to isolate the dominant term in the running time. When the keys are initially locally sorted, the running time of SearchSelect is reduced to $O(\log^2 p\, \mathrm{løg}(n/p))$, which represents an improvement for $n = \omega(p \log p)$.

Table 5.2 summarizes the running times of the best known selection algorithms for the pipelined hypercube.

Chapter 6

A Lower Bound for Selection

This chapter is concerned with deriving a lower bound on the complexity of the selection problem for a certain large class of networks. Given a set S of n keys and an integer k, $0 \leq k < n$, the selection problem is to determine a key with rank k in S. If all of the keys are distinct, there will be a unique key with rank k. The lower bound will apply to the special case of the selection problem in which all keys are distinct and the key being sought is the median of S, that is, $k = \lfloor n/2 \rfloor$. The only operations allowed on keys are copy and comparison.

Given that optimal speedup of selection is attainable on the EREW PRAM, for n/p sufficiently large, one is led to ask whether a similar result can be achieved under a more realistic model of computation such as the network model defined in Section 1.1. In fact, networks exist for which optimal speedup of selection is attainable. The sorting result of Leighton [Lei85] and the token distribution result of Peleg and Upfal [PU89] together imply that Vishkin's algorithm can be implemented to run in $O(n/p + \log p \log\log p)$ time on a certain class of bounded degree expander networks. Given a network corresponding to the graph $G = (V, E)$, the *expansion* of any subset U of the vertices (processors) is defined as $|\Gamma(U)|/|U|$, where $\Gamma(U)$ denotes the set of vertices in $V \setminus U$ that are adjacent (connected by a communication channel) to at least one vertex in U. The *expansion of a network* is the minimum over all $U \subseteq V$ such that $1 \leq |U| \leq |V|/2$ of the expansion of U. An *expander network* is a network with expansion $\Omega(1)$.

The main result of this chapter is an $\Omega((n/p)\log\log p + \log p)$ lower bound for selection on any network that satisfies a particular low expansion property defined in Section 6.3.

The class of networks satisfying this property includes all of the common network families such as the tree, multi-dimensional mesh, hypercube, butterfly and shuffle-exchange. The lower bound is proven in Sections 6.2 and 6.3. Note that this lower bound disproves a claim of Aggarwal and Huang stating that optimal speedup is possible for selection on the hypercube and shuffle-exchange [AH88]. When n/p is sufficiently large (for example, greater than $\log^2 p$ on the hypercube and shuffle-exchange), the lower bound is tight to within a multiplicative constant. The matching upper bound is provided by the algorithm SearchSelect presented in Chapter 5.

6.1 The Lower Bound Model

The lower bound for selection proven in this paper applies under a strictly more powerful model of network computation than the 1-port model defined in Section 1.1. In particular, the only restrictions enforced by this model are the following:

1. Each processor can send and/or receive at most one key per time step.
2. The only operations allowed on keys are copy and comparison, and each processor can perform at most one such operation per time step.

Note that an unlimited amount of computation and communication involving data other than keys can be performed in each time step. Under this model, it will be proven that any selection algorithm running on a network satisfying a particular low expansion property requires $\Omega((n/p)\log\log p + \log p)$ time steps in the worst case.

The model of network computation defined above will be referred to as the *lower bound model* throughout the remainder of this chapter.

6.2 A Restricted Lower Bound

This section provides an $\Omega((n/p)\log\log p - \log p/\log\log p)$ time lower bound for selection on a complete network with restricted capability. In order to simplify the exposition, it will be assumed that n and p are both powers of 2, $n \geq p$. Recall that S denotes the set of n keys, and that there are initially n/p keys located at each of the p processors. The lower bound established in this section applies under the model of computation defined below.

Definition 6.2.1 The *restricted lower bound model* is equivalent to the lower bound model defined in Section 6.1, with the following additional restriction. For every $R \subseteq S$, if R is initially assigned to a set of processors X, then at most $|X|$ comparisons per time step can involve keys belonging to R.

The motivation for the restricted lower bound model is that networks with poor expansion properties suffer from a similar, albeit less severe, inability to spread out a concentrated set of data in order to apply more processors to it. In particular, if the size of the neighborhood of a given set of processors X is $\alpha|X|$ where $\alpha = o(1/\log\log p)$, then in $O((n/p)\log\log p)$ time not even a constant fraction of the $|X|n/p$ keys initially located in the set X can have been moved or copied to processors outside of X. Of course, the restricted lower bound model is, by itself, quite unrealistic. Furthermore, it is unclear whether or not selection can be performed in $O((n/p)\log\log p)$ time on this model, even assuming the complete network. However, the preceding observation indicates that it may be possible to transfer a lower bound for the complete network operating under the restricted lower bound model to a realistic network operating under the lower bound model.

The remainder of Section 6.2 is devoted to proving an $\Omega((n/p)\log\log p - \log p/\log\log p)$ lower bound on the running time of any algorithm for computing the median on the complete network operating under the restricted lower bound model. The proof makes use of an adversary argument. At each time step, the algorithm indicates which set of at most p comparisons it would like to make, and the adversary resolves these comparisons sequentially. Of course, the algorithm must respect the rules of the restricted lower bound model, and the adversary must resolve comparisons in a manner that is consistent with at least one total ordering of the keys. Sometimes the adversary will give away the outcome of a comparison that has not been performed by the algorithm. This is done in order to simplify the lower bound argument. Note that giving away such additional information can only help the algorithm.

It is useful to keep in mind that the restricted lower bound model does not limit the amount of computation or communication involving non-key data that can be performed in each time step. Hence, it may be assumed that at all times, every processor is aware of all of the comparison information that has been gathered thus far. In other words, one may envision a global controller that receives the outcome of every comparison query made in a given time step, and then performs an unbounded amount of computation in order to determine the next set of comparison queries. This is essentially Valiant's parallel comparison model [Val75], except for the added restriction imposed by Definition 6.2.1.

The description and analysis of the adversary argument has been divided into a number of parts. Section 6.2.1 describes the information that the adversary gives away at the outset of the computation. Section 6.2.2 provides some useful definitions. Section 6.2.3 states, without proof, the invariants that will be satisfied by the adversary. Section 6.2.4 gives the procedure by which the adversary resolves comparison queries made by the algorithm. Section 6.2.5 completes the construction of the adversary by describing the additional information given away at certain points during the computation. Section 6.2.6 proves that the adversary resolves comparison queries in a consistent manner. Sections 6.2.4 to 6.2.6 assume that the invariants of Section 6.2.3 are satisfied. Section 6.2.7 proves that the construction of the adversary actually does ensure that these invariants are satisfied. Finally, Section 6.2.8 gives a precise statement of the lower bound established by the adversary argument.

6.2.1 The Initial Setup

Before the computation begins, certain information is given to the algorithm for free. Several definitions are needed in order to describe this information. Let a *block* of processors be defined as a set of processors B such that the $|B|n/p$ keys initially located in the set B have contiguous ranks in the set of all keys S. Let λ denote a positive integer to be defined later (it turns out that $\lambda = \Theta(\log p/\log\log p)$). It will be convenient to define the concept of a *block at level* i, $0 \le i < \lambda$, which is a block with the following additional properties.

1. All blocks at level i contain the same number of processors, s_i, and they are pairwise disjoint. Furthermore, $s_0 = p$.

2. A block at level i contains $b_i = 2^{\lceil \log(i+1) \rceil}$ pairwise disjoint blocks at level $i+1$, $0 \le i < \lambda - 1$.

3. The size of the blocks at level $i+1$ is such that the union of the blocks at level $i+1$ within a given block B at level i contains $\gamma|B|$ processors, $0 \le i < \lambda - 1$, where γ is a real constant between 0 and 1 (it turns out that $\frac{1}{8}$ is an appropriate choice for γ, see below and the proof of Lemma 6.2.4).

Thus, $s_{i+1} = \gamma s_i / b_i$ and

$$s_i = \frac{\gamma^i p}{b_0 \cdots b_{i-1}},$$

$0 \leq i < \lambda$. Setting γ to the reciprocal of some power of 2 will ensure that the s_i's are integer-valued as long as $p \geq c^i i!$ for some constant c. Taking logarithms, this requirement becomes $\log p \geq i \log i + O(i)$, which is asymptotically satisfied for certain values of λ in the range $\Theta(\log p / \log\log p)$. Note that every block has a unique level that can be determined from its size.

The following information is given away by the adversary before the computation begins. First, the input permutation of the keys is such that the set of all p processors forms a block at level 0. This implies the existence of a tree of blocks of depth λ. The algorithm is given both the IDs of the processors that make up each of the blocks in this tree as well as the ordering of the blocks within each level.

6.2.2 Useful Definitions

The adversary argument proceeds in *stages* consisting of a number of consecutive time steps. The number of stages is given by the positive integer λ defined in the preceding section. The ith stage begins at time $t_i = \max_{0 \leq j \leq i} \lfloor (n/p) d_j \rfloor - j$ and ends at time t_{i+1}, where $d_i = \frac{1}{2}\log(i+1)$, $0 \leq i < \lambda$. Note that $t_0 = 0$ and $t_{i+1} \geq t_i$, $0 \leq i < \lambda$. The following pair of technical lemmas will be useful for bounding the amount of work that can be performed in a single stage, and up to a particular stage.

Lemma 6.2.1 There are at most $\frac{1}{2\ln 2}\frac{n}{p(i+1)}$ time steps in the ith stage.

Proof: Observe that whenever $t_{i+1} - t_i$ is nonzero, it satisfies the inequality:

$$\begin{aligned} t_{i+1} - t_i &\leq \left(\lfloor \frac{1}{2}(n/p)\log(i+2) \rfloor - i - 1\right) - \left(\lfloor \frac{1}{2}(n/p)\log(i+1) \rfloor - i\right) \\ &\leq \frac{1}{2\ln 2}(n/p)\ln\left(1 + \frac{1}{i+1}\right) \\ &\leq \frac{1}{2\ln 2}\frac{n}{p(i+1)}. \end{aligned}$$

□

Lemma 6.2.2 The starting time of the ith stage, t_i, is at most $\frac{1}{2}(n/p)\log(i+1)$, $0 \leq i < \lambda$.

Proof: Immediate from the definition of t_i. □

Like blocks, processors and keys are assigned unique level numbers. The *level of a processor* is i if and only if the highest level block that it is contained in is at level i. The *level of a key* is given by the level of the processor that it is initially contained in. Note that a processor or key at level i is contained in exactly one block at level j, $0 \leq j \leq i$.

At any given time during the computation, some subset of the $n!$ possible total orderings of the keys remain consistent with all of the information that the algorithm has learned. Consider an arbitrary pair of distinct keys x and y. If $x < y$ in every one of the possible total orderings, then the outcome of the comparison between x and y is said to be *forced*.

Definition 6.2.2 Once the outcomes of all $n-1$ comparisons involving a particular key are forced, that key is said to be *dead*. Keys that are not dead are *live*.

Before the beginning of each stage, the adversary will give certain information away (described in Section 6.2.1 for stage 0, and in Section 6.2.5 for subsequent stages). After that information has been given away, and before the beginning of the ith stage, $0 \leq i < \lambda$, let D_i and $L_i = S \setminus D_i$ denote the sets of dead and live keys, respectively. Note that the ranks in S of the keys in D_i have all been determined.

Let U_i denote the set of all level i keys in L_i. Let $V_i = L_i \setminus U_i$. It will turn out that all keys of level less than i are dead by the beginning of stage i, so V_i denotes the set of all keys in L_i with level strictly greater than i.

6.2.3 Invariants

This section states, without proof, certain useful invariants that will be satisfied by the adversary. Section 6.2.7 proves that the construction of the adversary actually ensures that these invariants are satisfied.

As mentioned earlier, the adversary gives away certain information at the beginning of the ith stage, $0 \leq i < \lambda$. After this information has been given away, and before the ith stage begins, the construction of the adversary will guarantee that the following invariants hold:

1. There are integers $j < \lfloor n/2 \rfloor$ and $k > \lfloor n/2 \rfloor$ such that the set of ranks in S of the keys in D_i is exactly $\{0, \ldots, j-1\} \cup \{k, \ldots, n-1\}$. Thus, the set of ranks in S of the keys in L_i is $\{j, \ldots, k-1\}$.

2. The set L_i is a subset of a single block at level i. Thus, every key in L_i is of level i or higher.

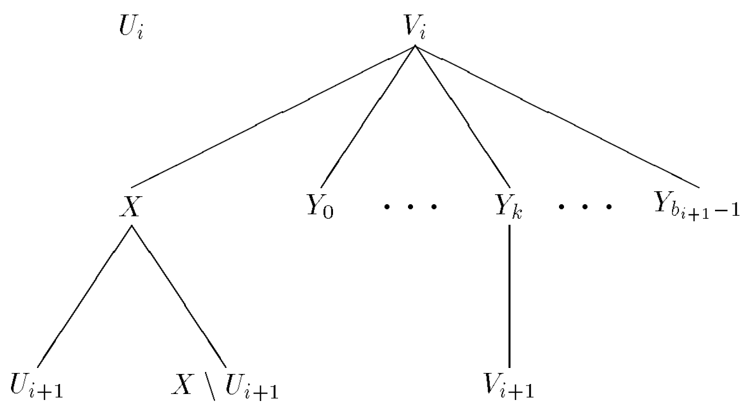

Figure 6.1: Extracting the sets U_{i+1} and V_{i+1} from U_i and V_i.

3. The key being sought, namely the median of S, is also the median of L_i.

4. All pairs of keys in U_i are incomparable. Furthermore, $|U_i| \geq (1-\gamma)(n/p)s_i/(i+1)$.

5. The set V_i is exactly the set of keys in a single block at level $i+1$.

6. Every key in U_i is incomparable to every key in V_i.

7. Two or more keys in L_i could still be the median of S.

In particular, note that Invariant 7 implies that the algorithm cannot yet have terminated successfully.

Figure 6.1 indicates how the sets U_{i+1} and V_{i+1} are related to U_i and V_i, $0 \leq i < \lambda - 1$. The set V_i is a block at level $i+1$, and hence may be partitioned into sets X and Y_j, $0 \leq j < b_{i+1}$, where X represents all of the level $i+1$ keys in V_i, and Y_j denotes the jth block of level $i+2$ in V_i. The adversary will be constructed in such a way that U_{i+1} is a subset of X and V_{i+1} is equal to Y_k for some k, $0 \leq k < b_{i+1}$.

6.2.4 Resolving Comparison Queries

This section gives the procedure by which the adversary resolves comparison queries made by the algorithm.

Consider a single comparison made by the algorithm, between keys x and y of levels j and k, respectively. Without loss of generality, assume that $j \leq k$. If either x or y is a dead key (this will be referred to as a type A comparison) then the outcome of the comparison is forced, and the adversary responds accordingly. Similarly, if y does not belong to the same block at level j as x (type B comparison), the response is forced. Otherwise, y belongs to the same block at level j as x, and the two cases $i < j$ and $i = j$ will be considered separately.

If $i = j$ (type C comparison) then x belongs to U_i and y belongs to either U_i or V_i. The adversary alternately resolves queries of this type by saying that x is smaller than (larger than), not only y, but the entire set of remaining live keys. The key x becomes a dead key.

In order to determine how to resolve a comparison query in the case $i < j$ (type D comparison), the adversary consults a *comparison tree* that it has been maintaining for the block at level j containing x. The adversary maintains such a comparison tree for every block. A comparison tree of the same sort was used by Borodin et al. [BGLY81] to obtain an easy (though not their strongest) sequential tradeoff between preprocessing time and search time in a partial order. A comparison tree is a binary tree with *tokens* placed at certain nodes. The comparison tree for a block B at level j contains $(1-\gamma)(n/p)s_j$ tokens corresponding to the keys of level j in B, and b_j tokens corresponding to the blocks of level $j+1$ in B. When it is important to distinguish between these two types of tokens, they will be referred to as *key tokens* and *block tokens*, respectively. At time 0, the key tokens are all placed at the root and the block tokens are placed, one per node, on the b_j nodes at depth $\log b_j = \lceil \log(j+1) \rceil$.

Note that every key corresponds to a key token in the comparison tree of exactly one block. Similarly, every block (except those at the highest level, $\lambda - 1$) corresponds to a block token in the comparison tree of exactly one block, namely that of its parent.

To resolve the comparison query between keys x and y in the case $i < j$ (type D comparison), the adversary locates the key token x' corresponding to x in the comparison tree T associated with the block at level j containing x. The adversary also locates the token y' corresponding to y in the same comparison tree. If $j = k$, this will be a key token; otherwise, it will be the block token corresponding to the unique block of level $j+1$ that contains y. Having located tokens x' and y' in the comparison tree T, the adversary resolves the comparison between keys x and y as follows. Let x' and y' reside in nodes u and v of the tree T, respectively, and let w be the least common ancestor of u and v. If w is not equal to either u or v, then the adversary does not move any tokens.

If w is equal to either u or v, the adversary must move at least one token downwards in the tree. If w is equal to both u and v, then x' is moved to the left child of w, and y' is moved to the right child of w. If w is equal to u but not v, then if v is located in the left subtree of w, x' is moved to the right child of w. The remaining cases are treated similarly.

The preceding algorithm describes how the adversary manipulates the tokens in each comparison tree, but does not indicate how comparison queries are resolved. To resolve queries, the adversary interprets each comparison tree as the partial order given by applying the following rules:

1. The keys corresponding to two tokens in the same comparison tree are incomparable if and only if they lie on a single downward path from the root.

2. If two tokens in the same comparison tree do not lie on a single downward path from the root, then the key corresponding to the token lying to the "left" (that is, the token residing in the left subtree of the least common ancestor) is deemed to be the smaller key.

Note that the preceding rules apply to block tokens as well, where the key corresponding to the block token is actually the entire set of keys in the corresponding block. This is appropriate since the set of keys in any block have contiguous ranks in S, and hence they all compare in the same way to any key outside of the block (that is, if B is a block and keys x, y and z are chosen such that $x, y \in B$ and $z \notin B$, then $x < z$ if and only if $y < z$).

At the beginning of the ith stage, consider the set of comparison trees corresponding to the unique block at level i given by Invariant 2 and all of the blocks that it contains. At any time during the ith stage, the inequalities between live keys to which the adversary has committed itself are exactly those encoded by this set of comparison trees. Note that the initial configuration of the tokens in the comparison trees encodes precisely the information given away by the adversary at the beginning of the computation, as described in Section 6.2.1. Furthermore, one may check that the token movements (if any) performed in processing a particular comparison query are always sufficient to resolve the query. In Section 6.2.6, it will be proven that this method of resolving queries can never lead to an inconsistent response by the adversary.

6.2.5 Additional Information

This section completes the construction of the adversary by describing the additional information given away before the beginning of each stage.

After stage i and before stage $i+1$, let T denote the comparison tree associated with the block V_i given by Invariant 5. The adversary counts the number of key tokens residing on each of the b_i paths of length $\lceil \log(i+1) \rceil$ descending from the root of T. Letting q denote the path containing the most key tokens, the adversary kills off certain keys in such a way that U_{i+1} becomes the set of key tokens on the path q, and V_{i+1} becomes the block of keys at level $i+2$ corresponding to the path q.

Before the beginning of stage $i+1$, the remaining live keys in $X = S \setminus L_{i+1}$ (recall that $L_{i+1} = U_{i+1} \cup V_{i+1}$) are killed off in such a way that the median of S is also the median of L_{i+1}. Every key that is killed off will either be said to be smaller than every key in L_{i+1}, or it will be said to be larger than every key in L_{i+1}. An arbitrary consistent order is maintained among the dead keys. Each key in X of level greater than or equal to $i+1$ corresponds to a key or block token in the tree T that does not reside on the path q. If the token resides to the left (right) of the path q, then the corresponding key is forced to be smaller (larger) than every key in L_{i+1}, in order to ensure consistency. On the other hand, the live level i keys of X remain incomparable to every key in L_{i+1}. Hence, in killing off each of these keys, the adversary has the freedom to decide whether to make it smaller or larger than every key in L_{i+1}. In Section 6.2.7, it will be proven that X contains sufficiently many live level i keys to force the median of S to be the median of L_{i+1}.

6.2.6 Consistency of the Adversary

This section proves that the adversary resolves comparison queries in a manner that is consistent with at least one total ordering of the keys.

Section 6.2.4 gave the adversary's procedure for resolving comparison queries, partitioning the queries into types A, B, C and D. For type A and B comparisons, consistency is immediate.

At any given time during the ith stage, let U_i^L denote the set of keys in U_i that are known to the algorithm to be less than every key in V_i, and let U_i^H denote the set of keys in U_i that are known to be greater than every key in V_i.

Now consider the type C comparisons. As argued in Section 6.2.4, the live keys in U_i remain incomparable to one another and to V_i throughout the ith stage. Therefore, at all times during the ith stage, any live key in U_i could be the minimum (maximum) key among all of the remaining live keys. Hence, the adversary can consistently kill off any key in U_i and assign it to either U_i^L or U_i^H. A similar argument can be used to prove that the comparison information given away by the adversary at the beginning of each stage (this information was described in Section 6.2.5) does not lead to an inconsistency.

The consistency of the procedure for resolving type D comparison queries follows immediately from the following lemma.

Lemma 6.2.3 Moving a token downward in a comparison tree can never result in an inconsistent response by the adversary.

Proof: Assume the lemma is false, and consider the first downward movement of a token that results in an inconsistent response by the adversary. Assume the token was moved downward in comparison tree T corresponding to block B at level i. First note that while a token movement in T can affect the partial order represented by T, it cannot affect the partial order represented by any other comparison tree. This "decoupling" of the partial orders represented by the various comparison trees follows from the use of block tokens to represent all of the keys in block B with level strictly greater than i. Since all of the keys within a block B' at level $i+1$ are treated as a single key in tree T, no comparison that is resolved within T can provide any ordering information regarding keys within block B'. By a similar argument, a token movement in tree T gives no information about keys at levels strictly less than i.

Thus, the inconsistency must arise within the partial order represented by comparison tree T alone. This is impossible since moving a token downward in a comparison tree can only augment the partial order that it represents, and a total order over the tokens in T that is consistent with this partial order is trivial to construct (by an inorder traversal of the tree, for example). □

The preceding discussion establishes that, up to any particular point in the computation, the behavior of the adversary is consistent with at least one total ordering of the keys.

6.2.7 Correctness of the Adversary

Sections 6.2.4 to 6.2.6 assume that the invariants of Section 6.2.3 are satisfied. This section proves that the construction of the adversary actually does ensure that these invariants are satisfied.

The proof is by induction on the number of stages. It is easy to check that all of the invariants are satisfied at the beginning of stage 0, with L_0 being S, U_0 being the set of $(1-\gamma)n$ level 0 keys, and $V_0 = S \setminus U_0$ being the set of keys in the lone block at level 1 (note that $b_0 = 2^{\lceil \log 1 \rceil} = 1$). The induction hypothesis is that the invariants hold at the beginning of the ith stage, $0 \le i < \lambda - 1$. It remains to be proven that the invariants hold at the beginning of stage $i+1$. Of these, only Invariants 3 and 4 do not follow immediately from the construction of the adversary. The task of establishing this remaining pair of invariants will now be addressed.

The set of keys U_i is initially contained in a set of at most $(1-\gamma)s_i$ processors (recall that all the keys in U_i are of level i), so Definition 6.2.1 and Lemma 6.2.1 imply that only $\frac{1}{2\ln 2}(n/p)(1-\gamma)s_i/(i+1)$ comparisons made during the ith stage can involve keys from U_i. Now each such comparison (even if it involves two keys from U_i) kills off only one key in U_i, and leaves the remaining live keys in U_i incomparable to one another and to V_i. By Invariant 4 (which holds at the beginning of stage i by the induction hypothesis), the preceding comparison bound implies that only a fraction $\frac{1}{2\ln 2}$ of the keys in U_i could have been killed off during the ith stage. Recall that the adversary kills off keys in U_i by alternately assigning them to U_i^L and U_i^H. Hence, at the end of stage i, the algorithm could at best have determined that $\frac{1}{4\ln 2}|U_i|$ of the keys in U_i belong to U_i^L, and that a similar number belong to U_i^H. At least $\left(1 - \frac{1}{2\ln 2}\right)|U_i|$ of the keys in U_i are still incomparable to one another and to V_i. Using the inequality of Invariant 4, and the fact that $|V_i| = (n/p)s_{i+1} = \gamma(n/p)s_i/b_i$, one finds that the ratio $|U_i|/|V_i|$ is at least $(1-\gamma)/\gamma$. Now consider the proof of the following lemma.

Lemma 6.2.4 For sufficiently small choices of γ, any of the keys in V_i could still be the median of L_i at the end of stage i.

Proof: The median of S is also the median of L_i by Invariant 3 (which holds at the beginning of stage i by the induction hypothesis). Hence, it is sufficient to prove that the size of the set of keys known to reside in U_i^L (U_i^H) plus the size of the set V_i is less than $|L_i|/2$. Using the inequalities $|U_i^L| \le \frac{1}{4\ln 2}|U_i|$ and $|U_i|/|V_i| \ge (1-\gamma)/\gamma$ proven above, this sum can be bounded by $\frac{1-\gamma}{4\ln 2}|L_i| + \gamma|L_i|$ at the end of the ith stage. Thus, the lemma holds for $\gamma < \frac{2\ln 2-1}{4\ln 2-1} \approx 0.218$. As mentioned earlier, it is convenient to set γ to the reciprocal of a power of 2; $\frac{1}{8}$ is an appropriate choice. □

Lemma 6.2.4 proves that the adversary can kill off keys at the beginning of stage $i+1$ in such a way that the median of S is also the median of L_i, as required in Section 6.2.5. Thus, Invariant 3 holds at the beginning of stage $i+1$.

It remains to prove Invariant 4. The construction of the adversary ensures that the keys of U_{i+1} are incomparable at the beginning of stage $i+1$, but the lower bound on $|U_{i+1}|$ requires proof. Let T denote the comparison tree from which the adversary extracts the set U_{i+1}. The number of key tokens in this tree is equal to $(1-\gamma)(n/p)s_{i+1}$, and these tokens reside initially in a set of $(1-\gamma)s_{i+1}$ processors. Let Λ denote the average depth of the key tokens in T. Observe that every comparison made by the algorithm increments the depth of at most two tokens. Hence, Definition 6.2.1 and Lemma 6.2.2 imply that $\Lambda \leq \frac{1}{2}\log(i+2)$ at time t_{i+1}. Let P denote the set of b_{i+1} paths in T from the root to the initial position of each of the b_{i+1} block tokens in T, that is, all paths of depth $\lceil \log(i+2) \rceil$. Let a_j denote the number of key tokens at depth j in T at time t_{i+1}. By a simple averaging argument, some path in P must contain at least

$$\sum_{0 \leq j \leq \lceil \log(i+2) \rceil} a_j 2^{-j}$$

key tokens. This sum is minimized by moving the tokens downward in a uniform fashion. Hence, the bound on Λ implies that some path q in P contains at least a fraction $2^{-\log(i+2)} = \frac{1}{i+2}$ of the key tokens. Therefore $|U_{i+1}| \geq (1-\gamma)(n/p)s_{i+1}/(i+2)$, and Invariant 4 holds.

Thus, the construction of the adversary ensures that Invariants 1 to 7 all hold at the beginning of stage $i+1$, $0 \leq i < \lambda - 1$, and the proof by induction is complete.

6.2.8 The Lower Bound

The following theorem summarizes the main result of Section 6.2.

Theorem 6.2.1 Any selection algorithm for the complete network running under the restricted lower bound model requires $\frac{1-o(1)}{2}(n/p)\log\log p - O(\log p/\log\log p)$ time steps.

Proof: This bound follows from Invariant 7 and the definition of t_i, with $i = \lambda - 1 = \Theta(\log p/\log\log p)$. □

The next section proves that the argument used to establish this lower bound can be adapted to a large class of realistic networks running under the lower bound model. Note that for $n/p = O(\log p/(\log\log p)^2)$, Theorem 6.2.1 does not provide any useful information; alternative lower bound arguments need to be applied. If every processor is required to receive a copy of the median, then a trivial $\Omega(\log p)$ lower bound holds, even for the complete network running under the lower bound model. If this requirement is not made, the task of proving an $\Omega(\log p)$ lower bound for such a powerful network may not be entirely trivial. For the complete network running under the restricted lower bound model, it is easy to prove an $\Omega(\log p)$ lower bound for computing the maximum (and hence for selection). However, this result is not particularly useful since the proof does not carry over to realistic networks running under the lower bound model. In any event, such considerations may be avoided in the special case of $\Omega(\log p)$ diameter networks, since a simple fooling argument implies that at least $\lceil d/2 \rceil$ time steps are necessary for any selection algorithm running (under the lower bound model) on a network with diameter d. Note that all bounded degree networks, and all of the networks considered in Section 6.3, have $\Omega(\log p)$ diameter.

6.3 The Network Lower Bound

The purpose of this section is to prove an $\Omega((n/p)\log\log p + \log p)$ time lower bound for selection on certain realistic networks. The lower bound will apply under the powerful lower bound model defined in Section 6.1, and will be obtained by making suitable modifications to the proof of Theorem 6.2.1. Consider the following definition.

Definition 6.3.1 Let $\mathcal{N}(\alpha, \beta)$ denote the class of all network families $\mathcal{F}$ for which, given any p processor network in $\mathcal{F}$, it is possible to construct all of the blocks (as defined in Section 6.2.1) at levels less than β in such a way that every block has expansion at most α, where α and β may depend on p.

A careful examination of the proof of Theorem 6.2.1 reveals that there are only two points at which the definition of the restricted lower bound model is invoked; the rest of the proof applies to the unrestricted lower bound model. The first use of Definition 6.2.1 is in the proof of Lemma 6.2.4, and the second use leads to the existence of a suitable set U_{i+1} in the induction step. In both cases, Definition 6.2.1 provides an upper bound of $(1-\gamma)s_i$ on the number of comparisons per time step involving the set of level i keys of a particular block at level i.

Theorem 6.3.1 Let $\mathcal{F}$ be a network family belonging to the class $\mathcal{N}(o(1/\log\log p), \beta)$. Then any selection algorithm for $\mathcal{F}$ has a running time of at least $\frac{1}{2}(n/p)\log\beta - \beta$ under the lower bound model.

Proof: In the following argument, let X denote a generic block at level i, $0 \leq i < \beta - 1$. Let Y denote the union of the blocks at level $i+1$ in X, and let $Z = X \setminus Y$. Let Y' and Z' denote the sets of keys initially residing in Y and Z, respectively. By the remarks preceding the statement of the theorem, it is sufficient to prove that the adversary construction of Section 6.2 can be revised in such a way that the two applications of Definition 6.2.1 can be avoided.

The construction of the adversary will be augmented in the following manner. Let T denote the comparison tree associated with block X. At any given time step, each of the $|Z'|$ key tokens in T is said to be either *bad* or *good*. A good key token is one for which no copy of the corresponding key has ever left the set of processors Z. Thus, all key tokens are initially good, and bad key tokens never become good again. How many good key tokens in tree T can become bad in a single time step? There are only two ways for a good key token to become bad. One way is for a copy of the corresponding key to be transmitted to a processor outside of X. Since block X has expansion $o(1/\log\log p)$, the number of good key tokens that can become bad in this manner is $o(|X|/\log\log p)$ per time step. The other way for a key token to become bad is for a copy of the corresponding key to be transmitted to a processor in Y. Since Y is the union of a number of disjoint sets with expansion $o(1/\log\log p)$, the number of these events is $o(|Y|/\log\log p)$ per time step. By construction, $|Z|$ is a constant fraction of $|X|$, so the total number of key tokens that can become bad in a single time step is $o(|Z|/\log\log p)$. Now the lower bound argument only runs for $O((n/p)\log\log p)$ time steps, during which time the number of bad tokens that can be generated is $o(|Z|n/p) = o(|Z'|)$. In other words, the number of good key tokens in T is $(1 - o(1))|Z'|$ at all times in the range of interest.

The final modification to the adversary construction of Section 6.2 is as follows. In the induction step, the adversary now extracts the set U_{i+1} from the set of good key tokens only. In a single time step, at most $|Z|$ comparisons can be made involving keys in the subset of Z' corresponding to the good key tokens in T, since processors outside of Z do not have copies of any of these keys. Thus, the argument of Section 6.2 goes through with the inequality of Invariant 4 weakened by a factor of $1 - o(1)$, the fraction of all key tokens that are guaranteed to be good. Now consider the proof of Lemma 6.2.4. At the beginning of the ith stage, all of the key tokens corresponding to U_i are good, and they reside in a set of $|Z|$ processors. Using an expansion argument as above, Lemma 6.2.1 implies that at most a $o(1)$ fraction of the key tokens corresponding to U_i can become bad during the ith stage. This minor effect is of no consequence since γ has already been set to a value that is bounded away from its maximum acceptable value. A similar comment applies to the effect of the weakened inequality in Invariant 4. □

Corollary 6.3.1.1 Let $\mathcal{F}$ be an $\Omega(\log p)$ diameter network family belonging to the class $\mathcal{N}(o(1/\log\log p), \Theta(\beta))$. Then any selection algorithm for $\mathcal{F}$ has a running time of at least $\frac{1}{2}(n/p)\log\beta + \Omega(\log p)$ under the lower bound model.

Proof: This bound follows immediately from Theorem 6.3.1 and the additional observation that any selection algorithm for a network of diameter d has a running time of at least $\lfloor d/2 \rfloor$ under the lower bound model. □

6.3.1 The Hypercube

Throughout this section, the quantity ϵ will be used to denote an arbitrarily small positive constant. A decomposition of the hypercube will now be defined that proves the hypercube network family belongs to $\mathcal{N}(o(1/\log\log p), \Theta(\log^{1/3-\epsilon} p))$. Given a hypercube with $p = 2^d$ processors, let $q = \lfloor d^\delta \rfloor$ or $\lfloor d^\delta \rfloor - 1$, whichever is odd. The exponent δ is a parameter between 0 and 1 to be determined later. Let $r = \lfloor d/q \rfloor$, and divide the first qr bits of each processor ID into r *fields* of q contiguous bits. The ith field determines the ith bit of an r-bit *condensed id* according to the following rule, $0 \le i < r$. If the majority of the q bits in the ith field are 0, then the ith bit of the condensed ID is 0; otherwise, it is a 1. Note that since q is odd there will always be a strict majority of either 0's or 1's. By symmetry, 2^{d-l} processors will belong to any *condensed subcube* of dimension $r - l$ obtained by fixing the values of l bits in the condensed ID, $0 \le l < r$.

Lemma 6.3.1 The expansion of a condensed subcube of dimension l is $O(lq^{-1/2})$.

Proof: Let U denote the set of processors belonging to a particular condensed subcube of dimension l. By symmetry, it will suffice to consider the condensed subcube corresponding to the condensed ID with first l bits fixed to 0, and with the remaining $r - l$ unspecified. Let V denote the set of processors in $\Gamma(U) \setminus U$ that are adjacent to some processor in V across some dimension in field 0. It is sufficient to prove that $|V|/|U| = O(q^{-1/2})$. But this ratio is readily seen to be precisely $\binom{q}{(q+1)/2}$ over 2^{q-1}, which is $O(q^{-1/2})$ by Stirling's approximation. □

Since $l \leq r$, $q = \Theta(d^{\delta})$ and $r = \Theta(d^{1-\delta})$, Lemma 6.3.1 implies that the expansion of every condensed subcube is $O(rq^{-1/2}) = O(d^{1-3\delta/2})$. If δ is chosen to be any constant strictly between 2/3 and 1, then the expansion of every condensed subcube will be $O(\log^{-\epsilon} p) = o(1/\log\log p)$. Furthermore, it should be clear that the condensed subcube structure can be used to construct the tree of blocks required by the lower bound argument of Section 6.2, at least to a certain depth, since the size of every block is a power of 2. All that remains is to determine the maximum possible value of β as a function of p. The relevant inequality is $c^{\beta}\beta! \leq 2^{r}$ for some constant c, which is satisfied for $\beta = \Theta(d^{1-\delta-\epsilon/2})$. Setting $\delta = 2/3 + \epsilon/2$ gives $\beta = d^{1/3-\epsilon}$, as claimed above. Slightly finer calculations allow this ϵ to be replaced by $o(1)$. Hence, Corollary 6.3.1.1 implies the following result.

Theorem 6.3.2 Any selection algorithm for the hypercube has a running time of at least $\frac{1-o(1)}{6}(n/p)\log\log p + \Omega(\log p)$ under the lower bound model.

6.3.2 Other Networks

The above decomposition also works for the shuffle-exchange, since it is easy to prove that Lemma 6.3.1 remains valid. Hence, the lower bound of Theorem 6.3.2 applies to the shuffle-exchange. Similar comments apply for the butterfly network.

Low flux networks such as the tree and multi-dimensional mesh can be easily decomposed into a large number of equal-sized components with very poor expansion. In such cases, β can be increased so that the lower bound of Theorem 6.3.2 applies with an improved multiplicative constant of $\frac{1-o(1)}{2}$.

6.4 Summary

The lower bounds for network selection discussed in this chapter significantly improve on previously known results when the number of keys at each processor, n/p, is sufficiently large [GK84]. In proving lower bounds, it was assumed that n and p are powers of 2, and that every processor begins with exactly n/p keys. The proofs can easily be extended to handle arbitrary values of n and p (losing at most a constant factor), and arbitrary initial distributions of the keys. Theorem 6.3.1 was proven for network families $\mathcal{F}$ belonging to

$\mathcal{N}(\alpha, \beta)$ with $\alpha = o(1/\log\log p)$; for $\alpha = \Omega(1/\log\log p)$ an obvious tradeoff occurs. It is likely that the multiplicative constants appearing in the lower bounds could be improved. Finally, it should be emphasized that this work deals with the worst case complexity of selection. Under an average case analysis, and for sufficiently high values of the ratio n/p, optimal speedup of selection is attainable on essentially any network.

Chapter 7

Adaptive Sorting Algorithms

This chapter deals with the problem of sorting n keys initially distributed uniformly over a hypercube with p processors, $n \geq p$. The well-known sequential lower bound for sorting implies an $\Omega((n \log n)/p)$ bound on the running time of any parallel sorting algorithm. For the case $n = p$, the best known sorting algorithm for the hypercube is Batcher's bitonic sort, which runs in $O(\log^2 p)$ time [Bat68]. For $n \neq p$, a number of other algorithms have been proposed. The running time and range of applicability of each of these algorithms is summarized in Table 7.1. Note that BitonicSort refers to the straightforward split-and-merge generalization of bitonic sort, due to Baudet and Stevenson [BS78]. Also, it should be emphasized that attention has been restricted to deterministic, worst case complexity algorithms running on the hypercube. For examples of results based on other assumptions, the reader is referred to [RV87], [VD88] and [Wag86].

One may verify that BitonicSort provides optimal speedup over sequential sorting only if $p = O(2^{\sqrt{\log n}})$. Two recent algorithms, which will be referred to as CubeSort (Cypher and Sanz, [CS92]) and ColumnSort (Aggarwal and Huang, [AH88]), have improved this result significantly. Both of these algorithms are optimal if n exceeds p by a polynomial factor, that is, if $n = p^{1+\epsilon}$ for any constant $\epsilon > 0$. ColumnSort is based on Leighton's technique for sorting n values by performing a constant number of smaller sorts [Lei85]. Note that Table 7.1 does not indicate the running time of ColumnSort when ϵ is allowed to vary. This algorithm is not competitive for $\epsilon = o(1)$ since the hidden constant in the running time is proportional to $(1+1/\epsilon)^{\alpha}$ with $\alpha = 2/(\log 3 - 1) \approx 3.419$, as opposed to $\alpha = 1$ for CubeSort. Of course, it may be possible to obtain an algorithm based on Leighton's column sorting technique that achieves a smaller value of α.

Algorithm	*Running Time*	*Range*
BitonicSort [Bat68][BS78]	$O((n/p)\log^2 p)$	$n = \Omega(p)$
MergeSort [NS82]	$O(\log^2 p/\log(p/n))$	$n = O(p)$
ColumnSort [AH88]	$O((n \log n)/p)$	$n = \Omega(p^{1+\epsilon})$, $\epsilon > 0$
CubeSort [CS92]	$O((n/p)\log^2 p/\log(n/p))$	$n = \Omega(p \log^{(k)} p)$

Table 7.1: Previous sorting algorithms for the hypercube and shuffle-exchange.

The main result of this chapter is a new sorting algorithm for the hypercube, SmoothSort, that runs asymptotically faster (in the worst case) than any previously known algorithm over a wide range of the ratio n/p. A simpler variant of this algorithm, which will be referred to as QuickSort, will also be presented. The running time of QuickSort is slightly greater than that of SmoothSort. The following example illustrates the nature of the results. For $n = p\log^2 p$, the sequential lower bound implies a lower bound of $\Omega(\log^3 p)$, the running time of BitonicSort is $O(\log^4 p)$, the running time of CubeSort is $O(\log^4 p/\log\log p)$, the running time of QuickSort is $O(\log^{7/2} p)$ and the running time of SmoothSort is $O(\log^{7/2} p(\log\log p)^{-1/2})$. ColumnSort is not competitive in this range, and has a running time of about $O(\log^{6.419} p)$.

Both QuickSort and SmoothSort make use of certain load balancing and selection algorithms given in Chapters 4 and 5, and these algorithms do not correspond to sorting circuits. In other words, they are not based solely on oblivious routing and compare-interchange operations. Such algorithms will be referred to as *adaptive* sorting algorithms. Chapter 8 describes two non-adaptive sorting algorithms that run on the hypercube and shuffle-exchange, including a slower version of SmoothSort.

7.1 Problem Definition: Sort

The Sort operation is defined as follows. Given n $O(\log p)$-bit keys distributed uniformly over p processors (that is, each processors holds at most $\lceil n/p \rceil$ keys), rearrange the n keys so that every key in processor i is less than or equal to every key in processor j whenever $0 \le i < j < p$. In addition, the n keys should remain uniformly distributed and the set of keys within any particular processor should be sorted. As for the Select operation defined in Chapter 5, it will be assumed that $n = O(p^c)$ for some constant c, and that the n keys are distinct.

7.2 Sorting on the Hypercube: QuickSort

The following algorithm is based on the well-known quicksorting paradigm of Hoare [Hoa62]. It makes use of algorithms Balance and SearchSelect from Sections 4.1.3 and 5.4, respectively.

Algorithm QuickSort

1. If the dimension of the hypercube being sorted is 0, locally sort the $O(n/p)$ keys located at each processor, and return. If performed, this operation takes $O((n/p)\log(n/p))$ time.

2. Let S denote the set of n keys. Call SearchSelect to find the value with rank $\lceil n/2 \rceil$ in S. Let this value be m. Using algorithm SearchSelect, this takes $O((n/p)\log\log p + \log^2 p\,\text{løg}(n/p))$ time.

3. Route all keys that are strictly less than m to the low subcube. Route all keys that are greater than or equal to m to the high subcube. To do this each processor splits the sorted list that it currently holds into two sorted sublists and sends the appropriate sublist to its neighbor in the highest dimension. This takes $O(n/p)$ time.

4. At this point, each processor contains between 0 and $2n/p$ keys. Call Balance to smooth out the load, that is, to redistribute the keys so that each processor holds a list of length $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$. This takes $O((n/p)\log^{1/2} p + \log^2 p)$ time.

5. Sort the low and high subcubes recursively.

The correctness of the preceding QuickSort algorithm should be obvious. The overall time complexity of QuickSort is readily seen to be

$$O((n/p)\log^{3/2} p + \log^3 p\,\text{løg}(n/p)).$$

7.2.1 QuickSort on the Pipelined Hypercube

As discussed in Section 2.4, the existence of optimal merging algorithms for the pipelined hypercube leads to an optimal bottom-up sorting algorithm for $n \geq p\log p$. The top-down quicksorting paradigm leads to an alternative optimal sorting algorithm for the pipelined hypercube for $n \geq p\log p\log\log p$. For a fast implementation of QuickSort running on the pipelined hypercube, the following pair of changes should be made to the algorithm stated above. First, the call to SearchSelect in Step 2 should be replaced by a call to the pipelined hypercube implementation of BalanceSelect(see Section 5.3). Second, Leighton's pipelined hypercube algorithm for Balance (see Section 4.1.1) should be used in Step 4. One may easily verify that the running time of this pipelined hypercube version of QuickSort is $O((n/p)\log p + \log^2 p \log\log p)$.

7.3 A Faster Hypercube Algorithm: SmoothSort

The SmoothSort sorting algorithm, which is also designed to run on the hypercube, will now be described. It makes use of the MultiBalance operation presented in Section 4.2.

Algorithm SmoothSort

1. Locally sort the $O(n/p)$ keys located at each processor. This takes $O((n/p)\log(n/p))$ time. If $p = 1$ then return.

2. Determine the 2^l keys with ranks $\left\lfloor \frac{in}{2^l} \right\rfloor$, $0 \leq i < 2^l$, and broadcast them to all processors. These will be called *splitter keys*. An appropriate choice for the parameter l will be specified later. For now, it will only be assumed that $2^l \leq n/p$. These selections can each be performed in $O(\log^2 p\,\mathrm{l{\o}g}(n/p))$ time as described in Section 5.4, since the keys have been sorted locally. Each broadcast takes $O(\log p)$ time. Thus, the total time required for this step is $O(2^l \log^2 p\,\mathrm{l{\o}g}(n/p))$. Each processor now contains a sorted list of n/p keys and a sorted list of 2^l splitter keys.

3. The 2^l splitter keys naturally partition the n keys into 2^l groups. The ith group consists of those keys with ranks between $\left\lfloor \frac{in}{2^l} \right\rfloor$ and $\left\lfloor \frac{(i+1)n}{2^l} \right\rfloor - 1$ inclusive, $0 \leq i < 2^l$. At each processor, label each of the n/p local keys with the appropriate l-bit group number. Since the list of keys and the list of splitter keys are sorted, this takes $O(2^l + n/p) = O(n/p)$ time.

4. Call MultiBalance to smooth out each of the 2^l groups of tokens. Now $g = 2^l$, so this takes $O((n/p)(l\log p)^{1/2} + 2^l \log^2 p)$ time.

5. Loop over the high order l dimensions, routing each group to the appropriate subcube. This takes $O(ln/p)$ time.

6. Call **Balance** to smooth out the load in each of the 2^l subcubes. The error in these subcubes is at most 2^l, so this takes $O(2^l \log^{1/2} p + \log^2 p)$ time. Note that if n/p is a power of 2 then there is no error, and this step can be omitted.

7. Sort the 2^l subcubes recursively.

It remains to determine the value of l that minimizes the total running time of **SmoothSort** subject to the constraints $l \geq 1$, $2^l \leq n/p$ and $l \leq \log p$. From the analysis accompanying the description of the algorithm, it may be seen that the cost of the top level of the recursion (that is, excluding the recursive calls) is dominated by an expression of the form

$$O\left(2^l \log^2 p \,\text{løg}(n/p) + (n/p)(l \log p)^{1/2}\right),$$

for all valid choices of l. The running time of **SmoothSort** is minimized (to within a constant factor) by increasing l to the point where the cost of performing the selections balances the cost of the **MultiBalance** operation. This leads to setting $l = \text{løg}(n/(pq))$, where $q = \log^{3/2} p \log\log p$. Substituting this choice of l into the above expression, one finds that the cost of a given level of the recursion is a function of p and n/p. The value of n/p does not vary with the depth of the recursion, while p is halved at each level. If $n \leq pq$, then l is forced to 1 and the running time of **SmoothSort** is the same as that of **QuickSort**. If $n \geq pq$, then $2^l = \Theta(n/(pq))$ and the depth of the recursion is $\Theta((\log p)/l)$. Furthermore, the cost of any level is at most that of the top level, so the total running time of **SmoothSort** is

$$O\left((n/p) \log p \sqrt{\frac{\log p}{\text{løg}(n/(pq))}} + \log^3 p \,\text{løg}(n/p)\right).$$

Note that the deviation from optimality (that is, from the time required by the sequential lower bound) for $n \geq pq$ is given by the square root factor. Previously, the best known algorithm for this range was **CubeSort**, which deviates from the lower bound by a factor of $\log p / \log(n/p)$.

The **MultiBalance** algorithm of Section 4.2.1 is inappropriate for the shuffle-exchange because it does not access the dimensions predominantly in ascending/descending order. The shuffle-exchange version of **MultiBalance** described at the end of Section 4.2.3 has a worst case running time of $O((n/p) \log p + g \log^2 p)$. This leads to a worst case running time of

$$O((n/p) \log^2 p / \,\text{løg}(n/p) + \log^3 p \,\text{løg}(n/p))$$

for the shuffle-exchange implementation of **SmoothSort**.

Algorithm	*Running Time*	*Transition Region*
MergeSort	$O(\log^2 p/\log(p/n))$	
BitonicSort	$O((n/p)\log^2 p)$	$n = \Theta(p)$
hybrid	$O((n/p)^{2/3}\log^2 p\,\text{løg}^{1/3}(n/p))$	$n = \Theta(p)$
SmoothSort	$O((n/p)\log^{3/2} p/\text{løg}^{1/2}(n/(pq))$	$n = \Theta(pq)$

Table 7.2: Running times of sorting algorithms for the hypercube.

7.3.1 Average Case Analysis

Theorem 4.2.3 shows that the hypercube and shuffle-exchange implementations of MultiBalance perform much better on average than in the worst case. When n/p exceeds a sufficiently large polylogarithmic factor, one may verify that the non-optimality of algorithm SmoothSort is entirely due to the cost of performing the MultiBalance operations. In fact, the following result holds.

Theorem 7.3.1 The average running time of both the hypercube and shuffle-exchange implementations of SmoothSort is

$$O((n/p)\log p + \log^3 p\,\text{løg}(n/p)),$$

which is optimal for $n \geq p\log^2 p\log\log p$.

The same theorem can be proven for QuickSort.

7.4 Summary

Table 7.2 summarizes the running times of the best known deterministic sorting algorithms for the hypercube over ascending ranges of the ratio n/p. MergeSort is listed first because it is the best known sorting method (in the sense of worst case asymptotic complexity) when $n \ll p$. The last column indicates that MergeSort remains the best known algorithm up to $n = \Theta(p)$, at which point BitonicSort has the same complexity. The "hybrid" entry refers to an algorithm to be defined and analyzed in Section 8.2.4. For $n = \Omega(pq)$, where $q = \log^{3/2} p\log\log p$, SmoothSort is the best known sorting algorithm and its complexity is given by the last entry in the table. Of course, when n exceeds p by a polynomial factor, CubeSort and ColumnSort also exhibit optimal complexity. Two more algorithms with this property will be described in Chapter 8.

The running times stated in Table 7.2 for the hybrid algorithm and SmoothSort do not apply to the shuffle-exchange. The running times of the fastest known sorting algorithms for the shuffle-exchange are summarized in Table 8.1 of Section 8.3.

In contrast with such non-adaptive sorting algorithms as BitonicSort and CubeSort, the average case complexity of SmoothSort is not equal to its worst case complexity. In fact, the average case complexity of SmoothSort is optimal for $n \geq p \log^2 p \log\log p$. This statement applies to the shuffle-exchange implementation of SmoothSort as well. By simultaneously guaranteeing good worst case performance, SmoothSort avoids the potential pitfalls of a simpler scheme such as HyperQuickSort [Wag86]. For solving the related problem of permutation routing, SmoothSort is even more practical because the cost of performing selections goes away. On the shuffle-exchange, SmoothSort performs permutation routing in $O((n/p)\log^2 p/\log(n/p))$ time. The constant hidden by the O-notation is small and, unlike CubeSort, this bound holds for all $n \geq p$.

Chapter 8

Non-Adaptive Sorting Algorithms

This chapter deals with *non-adaptive* sorting algorithms, that is, algorithms based solely on oblivious routing and compare-interchange operations.[1] There are several important reasons for considering this restricted class of algorithms.

1. Fast hardware can be used to implement the small number of operations required by non-adaptive algorithms.
2. Non-adaptive algorithms tend to perform very little local computation, and hence are likely to run quickly on computers for which the cost of communication is low relative to the cost of local computation.
3. Because non-adaptive algorithms are based on a small number of simple operations, they are more likely to run efficiently on a wide variety of parallel models.
4. In the case $n = p$, non-adaptive algorithms correspond to sorting circuits. It would be interesting to determine whether or not there exists a $o(\log^2 n)$ depth sorting circuit that can be simulated in $o(\log^2 n)$ time by a non-adaptive sorting algorithm running on the hypercube or shuffle-exchange.

With respect to the last point, it should be mentioned that Ajtai, Komlós and Szemerédi have developed an optimal $O(\log n)$ depth sorting circuit [AKS83]. Unfortunately, the O-notation hides an impractically large constant factor. Furthermore, no efficient simulation of the AKS sorting circuit has been found for the hypercube, shuffle-exchange or any other common network family.

[1] For $n > p$, compare-interchange is generalized to merge-and-split type operations.

This chapter describes a non-adaptive version of SmoothSort that runs on the shuffle-exchange and exhibits the same asymptotic performance as CubeSort for n/p sufficiently large. A sorting circuit based on recursive merging, called SquareSort, is also presented. SquareSort performs a large merging task by decomposing it into a number of smaller ones, and can be efficiently implemented on the hypercube and shuffle-exchange. The decomposition technique is similar to that considered by Van Voorhis, but obtains a more rapid decrease in the size of the subsorts [Van71]. Finally, three hybrid algorithms based on tradeoffs between SquareSort and other sorting algorithms are defined and analyzed.

8.1 A Non-Adaptive Version of SmoothSort

This section describes a non-adaptive implementation of SmoothSort that runs on the shuffle-exchange as well as the hypercube. It is interesting to note that this algorithm performs no explicit selections. The algorithm is described below in terms of the hypercube, but can easily be adapted to run in the same asymptotic time bound on the shuffle-exchange. Let d denote the dimension of the hypercube being sorted.

Algorithm SmoothSort

1. Locally sort the $O(n/p)$ keys located at each processor. This takes $O((n/p)\log(n/p))$ time. If $d = 0$ then return.

2. For $i = 0$ to $d-1$, merge pairs of lists across dimension i. Each of the resulting merged lists is of length $2n/p$. Partition each such list into two sublists of length n/p, one consisting of the even-ranked keys, and the other consisting of the odd-ranked keys. Send the sorted lists of even-ranked keys to the low subcube, and the odd-ranked keys to the high subcube. This set of merge-unshuffle-split operations takes $O((n/p)d)$ time.

3. For $i = 0$ to $d - 1$, merge pairs of lists across dimension i. Partition each of the resulting merged lists into two sublists of length n/p, one consisting of the lowest n/p keys, and the other consisting of the highest n/p keys. Send the low list to the low subcube, and the high list to the high subcube. This set of merge-and-split operations takes $O((n/p)d)$ time.

4. Let d' be as given by Equation (8.1) below. If $d' > 1$, then sort subcubes of dimension d' recursively using Steps 2 to 6.

5. Let L_i denote the sorted list of length $(n/p)2^{d'}$ located in the ith (low-order) subcube of dimension d', $0 \leq i < 2^{d-d'}$. Merge L_{2j} with L_{2j+1} by reversing L_{2j+1} and then performing a bitonic merge, $0 \leq j < 2^{d-d'-1}$. This takes $O((n/p)d')$ time.

6. Merge L_{2j+1} with L_{2j+2}, $0 \leq j < 2^{d-d'-1}-1$. This can be done as in the previous step, except that it is necessary to perform a monotone route first in order to move each pair of lists to be merged into a single subcube of dimension $2^{d'+1}$. The monotone route that sends the data at processor i to processor $i + 2^{d'} \bmod p$, $0 \leq i < p$, is appropriate. The inverse monotone route must be applied after the merging has been performed. This takes $O((n/p)d)$ time. Note that the time bound depends on d, and not d', due to the monotone routes. A useful trick described at the end of this section shows that the monotone route operations can be avoided, reducing the complexity of this step to $O((n/p)d')$ time.

As SmoothSort is based on compare-interchange operations, it is sufficient to consider its performance on inputs consisting entirely of 0's and 1's. This fact is known as the zero-one principle [Knu73]. Accordingly, assume that the input consists of k 0's and $n-k$ 1's for some arbitrary integer k, $0 \leq k \leq n$. Note that the effect of the ith merge-unshuffle-split operation of Step 2 is to balance the number of 0's (and 1's) between neighboring processors across dimension i. Hence, Lemma 4.1.3 implies that there exists a nonnegative integer a such that, after Step 2 has been completed, every processor contains a number of 0's in the range $[a, a+d]$. Furthermore, the inductive proof of Lemma 4.1.3 can easily be augmented to show that if some processor does contain d more 0's than another, then processor 0 is the unique processor with $a+d$ 0's, and processor 2^d-1 is the unique processor with a 0's.

It is useful to think of the n keys as being arranged in a $(n/p) \times p$ array, where the ith largest key in processor j resides in row i and column j, $0 \leq i < n/p$, $0 \leq j < p$. Intuitively, Step 2 is attempting to arrange the keys in row-major order. On the other hand, the goal of Step 3, and of the sort as a whole, is to arrange the keys in column-major order. Some additional notation is needed in order to measure the actual progress made by these steps. Let $R_0(i,j) = pi + j$ denote the *estimated rank* of the key in row i and column j just after Step 2, and let $R_1(i,j) = pj + i$ denote the estimated rank of the key in the same location just after Step 3. Let h_0 denote the maximum value of $R_0(i,j)$ over all 0 keys, and let l_0 denote the minimum value of $R_0(i,j)$ over all 1 keys. Let h_1 and l_1 be defined in a similar manner. Then the discussion of the preceding paragraph implies that $h_0 - l_0 \leq p(a+d) - [p(a+1)+p-1] = pd - 2p + 1$. Furthermore, it is straightforward to prove that $h_1 \le h_0$ and $l_1 \ge l_0$.

Hence, $h_1 - l_1 + 1 \le pd - 2p + 2$. This bound implies that after Step 3, every key is within $\lceil (pd - 2p + 2)/(n/p) \rceil$ columns (processors) of the correct output column. The sort can now be completed by recursively sorting each of the $2^{d-d'}$ (low-order) subcubes of dimension d', where

$$d' = \left\lceil \log \left\lceil \frac{p(pd - 2p + 2)}{n} \right\rceil \right\rceil , \tag{8.1}$$

and then merging even and odd pairs of subcubes of dimension d' as in Steps 5 and 6. The order of these two merging steps is interchangeable. In order to check that they actually complete the sort, it suffices to prove that odd-even transposition sort (see [Knu73]) terminates in two steps when the input is such that every key is at most one move away from its final position. This fact is easy to prove, and that it is sufficient follows from the split-and-merge technique of Baudet and Stevenson [BS78].

It follows from Equation (8.1) that the depth of the recursion is $O(\log p / \log(n/(p \log p)))$. Since the cost of every level (excluding recursive calls) of the recursion is bounded by that of the top level, the total running time of the non-adaptive version of SmoothSort on the hypercube or shuffle-exchange is

$$O\left(\frac{(n/p) \log^2 p}{\log(n/(p \log p))} \right).$$

This result matches the asymptotic performance of CubeSort for $n \ge p \log^{1+\epsilon} p$, where ϵ denotes an arbitrarily small positive constant. More importantly, the multiplicative constant hidden by the O-notation is very small, particularly for the hypercube implementation. For both the hypercube and shuffle-exchange, the constant associated with CubeSort is almost an order of magnitude higher.

There are a number of tricks that can be used to speed up the implementation of SmoothSort slightly. In Step 2, the communication cost can be reduced by performing an unshuffle-merge, that is, by sending every second key to the neighboring processor. This has the effect of increasing the balancing error from d to $2d$, but this adverse effect is insignificant if n/p is large. Step 3 may run faster if it is implemented as a transpose (no merging) followed by a local sort. Finally, the monotone routes in Step 6 can be eliminated by mapping columns (of the array defined above) to processors in a different manner. Specifically, the ith largest group of n/p keys should be sorted to the processor with ID equal to the ith binary Gray code, rather than to processor i.[2] After sorting to

[2]See any introductory text on switching theory for a definition of binary Gray codes. Gray codes are a

this configuration, the keys can be routed to the usual sorted configuration in $(n/p)\log p$ steps. The details of this Gray coded scheme are left to the reader. It should be mentioned that the same trick can be used to halve the depth of the shuffle-exchange implementation of the balanced sorting network of Dowd et al. [DPSR83].

8.2 The SquareSort Sorting Circuit

This section presents a sorting circuit based on recursive merging called SquareSort. Like BitonicSort, the depth of this sorting circuit is $\Theta(\log^2 n)$. On the other hand, SquareSort leads to improved tradeoffs for sorting on the hypercube and shuffle-exchange for $n > p$. As SquareSort is based on compare-interchange operations, the zero-one principle implies that it is sufficient to consider its performance on inputs consisting entirely of 0's and 1's. The SquareSort algorithm relies on the following merging technique, called SquareMerge. Consider a rectangular array A of 0's and 1's with 2^a rows and 2^b columns, where a and b are nonnegative integers, and in which the rows and columns have already been sorted in ascending order. Note that the boundary between the 0's and the 1's in array A forms a staircase. The elements of A may either be viewed as being organized in 2^a sorted lists of length 2^b, or in 2^b sorted lists of length 2^a. The goal is to produce a single sorted list of length 2^{a+b}. The depth of the SquareMerge sorting circuit that performs this merging task will be denoted $M(a,b)$. For convenience, the merging task itself will also be referred to as $M(a,b)$. Note that for all nonnegative integers a and b, $M(a,b) = M(b,a)$ and $M(a,0) = 0$. Furthermore, the problem $M(a,1)$ will be solved by a bitonic merge, so $M(a,1) = a+1$, $a \geq 1$. The most interesting case remains to be considered, namely, when a and b are both greater than 1. Assume without loss of generality that $a \geq b$. In this case, the construction of the SquareMerge circuit will satisfy

$$M(a,b) = M(\lfloor a/2 \rfloor, b) + M(\lceil a/2 \rceil, b) + 2M(\lceil a/2 \rceil + b, 1). \tag{8.2}$$

The following procedure for performing the merging problem $M(a,b)$ will establish the validity of Equation (8.2). First, partition the rows of array A into $2^{\lceil a/2 \rceil}$ groups, placing row i into group $i \bmod 2^{\lceil a/2 \rceil}$, $0 \leq i < 2^a$. All of the groups can be sorted in parallel in depth $M(\lfloor a/2 \rfloor, b)$. The resulting $2^{\lceil a/2 \rceil}$ sorted groups of size $2^{\lfloor a/2 \rfloor + b}$ must now be merged in depth $M(\lceil a/2 \rceil, b) + 2M(\lceil a/2 \rceil + b, 1)$. Consider the following lemma.

commonly used construct for obtaining efficient hypercube embeddings; see [Joh87], for example.

Lemma 8.2.1 Let integers i and j satisfy $0 \leq i < j < 2^{\lceil a/2 \rceil}$. Then group i contains fewer 1's than group j. Furthermore, group 0 contains at most 2^b fewer 1's than group $2^{\lceil a/2 \rceil} - 1$.

Proof: This follows easily from the existence of the staircase boundary between the 0's and the 1's in array A. □

Arrange the $2^{\lceil a/2 \rceil}$ sorted groups in an array A' with $2^{\lceil a/2 \rceil}$ rows and $2^{\lfloor a/2 \rfloor + b}$ columns. The ith row consists of group i, arranged in ascending order. The preceding lemma implies that the columns are also sorted in ascending order, so the remaining problem can be solved as an $M(\lceil a/2 \rceil, \lfloor a/2 \rfloor + b)$. However, there is additional structure to the remaining problem that permits it to be solved more rapidly. Namely, Lemma 8.2.1 implies that at most 2^b columns are dirty (a column is dirty if it contains both 0's and 1's), and that the dirty columns are contiguous. Thus, the groups can be merged as in Steps 4 to 7 of algorithm SquareMerge, stated below. The input to SquareMerge is a $2^a \times 2^b$ array A of 0's and 1's, where the rows and columns have already been sorted ascending and $a \geq b$. The code for $a < b$ is similar.

Algorithm SquareMerge

1. Partition the rows of array A into $2^{\lceil a/2 \rceil}$ groups, placing row i into group $i \bmod 2^{\lceil a/2 \rceil}$, $0 \leq i < 2^a$. Each group forms a subarray with $2^{\lfloor a/2 \rfloor}$ rows and 2^b columns.

2. Sort all of the groups in parallel. Each subproblem is an $M(\lfloor a/2 \rfloor, b)$.

3. Arrange the $2^{\lceil a/2 \rceil}$ sorted groups in an array A' with $2^{\lceil a/2 \rceil}$ rows and $2^{\lfloor a/2 \rfloor + b}$ columns. The ith row consists of group i, arranged in ascending order.

4. Partition the $2^{\lceil a/2 \rceil} \times 2^{\lfloor a/2 \rfloor + b}$ array A' into $2^{\lfloor a/2 \rfloor}$ subarrays A'_i, where the ith $2^{\lceil a/2 \rceil} \times 2^b$ subarray consists of the columns $i2^b$ through $(i+1)2^b - 1$ of A', $0 \leq i < 2^{\lfloor a/2 \rfloor}$.

5. Sort all of the $2^{\lfloor a/2 \rfloor}$ subarrays in parallel. Each subproblem is an $M(\lceil a/2 \rceil, b)$.

6. Merge A'_{2i} with A'_{2i+1}, $0 \leq i < 2^{\lfloor a/2 \rfloor - 1}$. Each of these subproblems is an $M(\lceil a/2 \rceil + b, 1)$.

7. Merge A'_{2i+1} with A'_{2i+2}, $0 \leq i < 2^{\lfloor a/2 \rfloor - 1} - 1$. Each of these subproblems is an $M(\lceil a/2 \rceil + b, 1)$.

The fact that the last two merge operations actually complete the sort follows by the same argument as was applied in Section 8.1. Thus, Equation (8.2) holds.

The recurrence of Equation (8.2) (with base cases as stated above) will now be used to obtain an upper bound on $M(a,b)$. Assuming without loss of generality that $a \geq b$, an easy induction proves that $M(a,b) \leq M(a,a)$. Two applications of Equation (8.2) then lead to

$$M(a,a) \leq 4M(\lceil a/2 \rceil, \lceil a/2 \rceil) + O(a).$$

Making use of the fact that $\lceil \lceil x/y \rceil / z \rceil = \lceil x/yz \rceil$ for all positive integers x, y and z, the recurrence can be unwound further to obtain

$$M(a,a) \leq 2^{2k} M(\lceil a/2^k \rceil, \lceil a/2^k \rceil) + O(2^k a), \tag{8.3}$$

for all nonnegative integers k. Setting $k = \log a$, one finds that $M(a,a) = O(a^2)$. Hence, $M(a,b) = O(a^2)$.

The SquareSort sorting circuit can now be defined in terms of SquareMerge. In the following algorithm, assume that SquareSort is sorting 2^a keys for some nonnegative integer a, and let $S(x)$ denote the depth of the SquareSort sorting circuit on 2^x keys.

Algorithm SquareSort

1. If $a \leq 3$ then sort the keys using BitonicSort and return.
2. Arrange the 2^a keys in a $2^{\lfloor a/2 \rfloor} \times 2^{\lceil a/2 \rceil}$ array A.
3. Sort each of the rows of A recursively in parallel. This uses depth $S(\lfloor a/2 \rfloor)$.
4. Sort each of the columns of A recursively in parallel. This uses depth $S(\lceil a/2 \rceil)$.
5. Apply SquareMerge to the array A. This uses depth $M(\lfloor a/2 \rfloor, \lceil a/2 \rceil)$, which is $O(a^2)$ by the preceding analysis.

Thus, an upper bound on the depth of the SquareSort sorting circuit is given by the solution to the recurrence

$$S(a) \leq S(\lfloor a/2 \rfloor) + S(\rceil a/2 \rceil) + O(a^2), \tag{8.4}$$

with $S(1)$ equal to a constant. Unwinding this recurrence, one finds that $S(a) = O(a^2)$, as promised. Of course, this result is not very interesting in view of the fact that BitonicSort achieves the same bound with a much simpler construction and a smaller multiplicative constant. The significance of SquareSort is that, like CubeSort, it gives a method for expressing a single large sort in terms of a number of smaller ones. Both techniques have the same (within a constant factor) efficiency in this regard, except that CubeSort places a lower bound on the size of the smaller sorts. Thus, in certain cases, SquareSort leads to a tradeoff while CubeSort does not. The main utility for expressing a large sort in terms of smaller ones is as follows. If some other sorting method can be used to speed up the small sorts, then SquareSort can be easily modified to reduce the running time of the large sort accordingly.

Formally, suppose that sets of keys of size 2^b, where $2^b \ll 2^a$, can be sorted in depth T by some circuit X. Using Equation (8.3) with $k = \log a - \log b + \Theta(1)$ gives

$$\begin{aligned} M(\lceil a/2 \rceil, \lfloor a/2 \rfloor) &= O\left(\frac{a^2 T}{b^2} + \frac{a^2}{b}\right) \\ &= O\left(\frac{a^2 T}{b^2}\right), \end{aligned}$$

since $T = \Omega(b)$. Thus, using circuit X to sort sets of size 2^b, the depth of the SquareSort sorting circuit satisfies the recurrence

$$S(a) \leq S(\lfloor a/2 \rfloor) + S(\lceil a/2 \rceil) + O\left(\frac{a^2 T}{b^2}\right), \tag{8.5}$$

with $S(1)$ equal to a constant. Three applications of this technique are given in Sections 8.2.2, 8.2.3 and 8.2.4. First, however, it must be shown that the SquareSort sorting circuit can be efficiently implemented on the hypercube and shuffle-exchange, that is, with a running time that is proportional to its depth. This is the subject of the following section.

8.2.1 Network Implementations of SquareSort

It is relatively easy to obtain an efficient implementation of SquareSort on the hypercube. Only the case $n = p$ (one key per processor) needs to be considered explicitly; Sections 8.2.2, 8.2.3 and 8.2.4 give tradeoffs with other sorting algorithms for $n \neq p$. In the case $n = p$, there are p keys in the array A defined by the top-level call to algorithm SquareSort.

Now consider the effect of embedding the array A in the hypercube in row-major order. Given this embedding, it is straightforward to prove that every array encountered during the execution of SquareSort satisfies the property that its row and column indices are encoded by two disjoint, contiguous sets of ID bits. Note that in algorithm SquareMerge, only Steps 6 and 7 involve any key comparisons and/or movement of data. The other steps consist of recursive calls and trivial computations to set up the recursive calls (to calculate which sets of ID bits define the row and column indices of the array to be sorted by the recursive call). Step 6 involves merging pairs of equal-length sorted lists. Each list resides in a subcube of dimension $2^{\lceil a/2 \rceil}$, and each pair of lists resides in a single subcube of the next higher dimension. Thus, the merging operation can be implemented by reversing one of the lists and then performing a bitonic merge, all of which can be done in $O(a)$ time.

Step 7 is slightly trickier to implement. Once again, the task is to merge pairs of equal-length sorted lists where each list resides in a subcube. The difficulty is that the pairs of lists to be merged are not necessarily located in adjacent subcubes. One solution to this problem is to perform a monotone route that shifts the location of each key in array A' by $2^{\lceil a/2 \rceil}$, the length of a sorted list. The pairs of lists can then be merged as in Step 6, and the merged lists routed back to the appropriate position by a second monotone route. The time required to perform each monotone route is proportional to the dimension of the subcube containing A'. Thus, the total time required to perform Step 7 is also $O(a)$.

The preceding discussion implies that the total running time of SquareMerge, excluding the cost of recursive calls, is $O(a)$. Hence, the analysis of Section 8.2 goes through unchanged and the upper bound of Equation (8.2) also applies to the running time of the hypercube implementation of SquareMerge. Given subroutine SquareMerge, algorithm SquareSort is straightforward to implement efficiently on the hypercube. Therefore, the circuit depth bounds of Equations (8.4) and (8.5) carry over to running time bounds for the hypercube implementation of SquareSort.

Figures 8.1 and 8.2 apply the SquareSort sorting technique to a random permutation of the 2^6 integers $[0, 64)$. The example is not entirely faithful to the SquareSort program stated in Section 8.2, since $a = 6$ and the array A was chosen to be 4×16 rather than 8×8. The bit sequences labelling the rows and columns of each of the 4×16 arrays in Figures 8.1 and 8.2 indicate how the array is mapped to the hypercube processors. For instance, the columns of the first array are labelled $b_3b_2b_1b_0$ and the rows are labelled b_5b_4, which means that the key in row $i = (i_1i_0)_2$ and column $j = (j_3j_2j_1j_0)_2$ is located at processor $(i_1i_0j_3j_2j_1j_0)_2$.

The asymptotic performance of SquareSort on the hypercube can be duplicated on the shuffle-exchange, but not simply by a naive translation of the hypercube implementation. There are two problems that must be avoided in order to ensure that the bitonic merge,

Figure 8.1: A sample run of SquareSort (continued in Figure 8.2).

Figure 8.2: A sample run of SquareSort (continued from Figure 8.1).

list reversal and monotone route operations, which are performed over subcubes, can be executed in time proportional to the dimension of the subcube (as opposed to the dimension of the entire shuffle-exchange, for instance). Recall that the subcubes of interest correspond to the arrays defined by SquareSort and SquareMerge, and that the row and column indices are given by two disjoint, contiguous sets of address bits. The first problem is that the row and column address bits can be far apart, forcing a pass over the corresponding dimensions to include a long sequences of shuffle or unshuffle operations. The second problem is that even if the row and column address bits form a single contiguous block, this block may be far from the exchange (bit 0) position.

Both of these problems may be solved by permuting the data before each recursive call to SquareMerge in order to bring the row and column bits together. Note that such a permutation is not always necessary. Specifically, the array associated with the recursive call in Step 5 of algorithm SquareMerge is already mapped to an appropriate subcube, while the one associated with Step 5 is not (for the case $a < b$, the situation is reversed). Where it is needed, the appropriate permutation can be performed efficiently using the self-routing Benes network of Nassimi and Sahni [NS81]. Of course, the inverse permutation must be applied once the recursive call to SquareMerge completes execution.

8.2.2 An Adaptive Tradeoff for $n \leq p$

Consider the problem of sorting n keys with p processors, where $n \leq p$. For this range, the MergeSort algorithm of Nassimi and Sahni runs in $O(\log^2 p/\log(p/n))$ time. MergeSort reduces to a particularly simple algorithm when $p \geq n^2$. The purpose of this section is to demonstrate that the same performance is achieved by a hybrid algorithm based on SquareSort and the simple version of MergeSort for $p \geq n^2$. The hybrid algorithm is SquareSort except that MergeSort is applied to perform sorts that are sufficiently small to allow a quadratic number of processors to be applied. This is an adaptive tradeoff, since MergeSort does not correspond to a sorting circuit. The running time of the hybrid algorithm on a hypercube or shuffle-exchange of dimension a is given by Equation (8.5) with $b = T = \text{løg}(p/n)$. Solving this recurrence, and setting $a = \log p$, leads to a running time of $O(\log^2 p/\,\text{løg}(p/n))$ for $n \leq p$, as claimed.

8.2.3 A Non-Adaptive Tradeoff for $n \geq p$

The non-adaptive CubeSort algorithm of Cypher and Sanz runs in $O((n/p)\log^2 p/\log(n/p))$ time on the hypercube or shuffle-exchange assuming that $n \geq p\log^{(k)} p$ for some constant k [CS92]. The constant hidden by the O-notation is exponential in k and is moderate even for $k = 1$. The non-adaptive version of SmoothSort presented in Section 8.1 has the same time complexity for $n \geq p\log^{1+\epsilon} p$. However, SmoothSort has a very small multiplicative constant (particularly on the hypercube), and appears to be a truly practical algorithm for certain realistic values of n and p.

The purpose of this section is to prove that a hybrid algorithm based on SquareSort and the non-adaptive version of SmoothSort (CubeSort could also be used here, at the expense of a constant factor) runs in $O((n/p)\log^2 p/\log(n/p))$ time on the hypercube or shuffle-exchange over the entire range $n \geq p$. The hybrid algorithm is SquareSort with the exception that SmoothSort is applied to sorts over subcubes consisting of fewer than n/p processors. The analysis of the hybrid algorithm is very similar to that of the previous section. Namely, the running time of the hybrid algorithm is given by the recurrence of Equation (8.5) with $a = \log p$ and $b = T = \text{løg}(n/p)$, which leads to $O(\log^2 p/\text{løg}(n/p))$ for $n \geq p$. Note that the hybrid algorithm is non-adaptive.

8.2.4 An Adaptive Tradeoff for $p \leq n \leq pq$

This section analyzes the performance of a hybrid sorting algorithm for the hypercube based on SquareSort and QuickSort. The range of applicability of the algorithm is $p \leq n \leq pq$, where $q = \log^{3/2} p \log\log p$. The hybrid algorithm is SquareSort, except that QuickSort is applied to every sorting subproblem involving n' keys in a subcube of p' processors where $n' = \Theta(p'\log^{3/2} p' \log\log p')$. Using the fact that $n' = (n/p)p'$, this condition implies $\log^{3/2} p' = \Theta((n/p)/\log(n/p))$. The cost of running QuickSort on a problem of this size is $O(\log^3 p' \log\log p') = O((n/p)^2/\log(n/p))$. Thus, the running time of the hybrid algorithm is given by the recurrence of Equation (8.5) with

$$
\begin{aligned}
a &= \log p, \\
b^{3/2} &= \Theta\left((n/p)/\log(n/p)\right), \text{ and} \\
T &= O\left((n/p)^2/\log(n/p)\right).
\end{aligned}
$$

Algorithm	*Running Time*	*Transition Region*
MergeSort	$O(\log^2 p/\log(p/n))$	
		$n = \Theta(p)$
BitonicSort	$O((n/p)\log^2 p)$	
		$n = \Theta(p)$
hybrid	$O((n/p)\log^2 p/\log(n/p))$	

Table 8.1: Running times of sorting algorithms for the shuffle-exchange.

Solving this recurrence, one finds that the running time of the hybrid algorithm is

$$O\left(\log^2 p(n/p)^{2/3}\log^{1/3}(n/p)\right),$$

as claimed in Table 7.2.

Two points concerning this hybrid algorithm should be emphasized. First, the algorithm does not run (with the stated complexity) on the shuffle-exchange, since it makes use of QuickSort. Second, replacing QuickSort with SmoothSort does not yield any improvement.

8.3 Summary

This chapter described two non-adaptive sorting methods and a number of hybrid algorithms. With the exception of the adaptive tradeoff considered in Section 8.2.4, all of the results discussed in this chapter apply to the shuffle-exchange as well as the hypercube. Table 8.1 summarizes the running times of the best known sorting algorithms for the shuffle-exchange over ascending ranges of the ratio n/p. For $n \leq p$, the bounds are the same as for the hypercube (see Table 7.2). For $n = \omega(p)$, the hybrid algorithm of Section 8.2.3 provides the best known bound. Note that for $n = \Omega(p\log^{(k)} p)$, where k is a fixed positive integer, the running time of the hybrid algorithm is matched (to within a constant factor) by CubeSort. Furthermore, it is matched by the non-adaptive version of SmoothSort for $n = \Omega(p\log^{1+\epsilon} p)$, and by ColumnSort for $n = \Omega(p^{1+\epsilon})$, where in each case ϵ denotes an arbitrarily small positive constant.

The non-adaptive version of SmoothSort does not perform any explicit selections and appears to be the most practical sorting method for sufficiently large values of n (say, 10^4 or more), and where n exceeds p by a significant polynomial factor (e.g., $n = p^2$).

Roughly speaking, the construction of the SquareSort sorting circuit is based on a technique for expressing a single large sort in terms of a number of smaller sorts. If the size of the smaller sorts is sufficiently large, CubeSort performs such a decomposition with the same asymptotic efficiency, although the multiplicative constant associated with CubeSort is almost an order of magnitude higher.

Chapter 9

Concluding Remarks

The preceding chapters have primarily considered algorithms for load balancing, selection and sorting on the hypercube and shuffle-exchange. While some progress has been made in these areas, many open problems remain. Several open problems which correspond to natural extensions of the work described in this thesis will now be considered.

Section 4.1 presented upper and lower bounds for the Balance operation running on the hypercube. The bounds are not tight in the case where the average number of tokens per processor is less than a constant fraction of the maximum number of tokens at any processor, and it seems likely that the upper bound could be improved in this case. One might also attempt to match the current performance of Balance with a hypercube algorithm that restricts all processors to communicate along the same dimension at any given time.

It would be interesting to try to prove a $\omega(\log n)$ lower bound for the problem of sorting n keys on a hypercube or shuffle-exchange with n processors, at least for some restricted class of algorithms. For example, one might consider non-adaptive algorithms or, being less restrictive, arbitrary algorithms with the sole restriction that keys cannot be duplicated. Note that even for the simpler problem of selection, there is no known $o(\log^2 n)$ hypercube algorithm which does not duplicate keys. With respect to proving a lower bound, the techniques of Chapter 6 may provide a useful starting point.

All of the sorting algorithms described in this thesis have the property that the progress achieved by the algorithm at any given time is relatively easy to characterize. For example, the partial progress of SmoothSort is given by the least integer k such that every key has been routed to the correct low-order subcube of dimension k. On the other hand, the partial progress of the $O(\log n)$ depth AKS sorting circuit is more complicated to characterize; in particular, it only guarantees progress with respect to *most* of the keys, as opposed to all of them (except at the end, of course). Thus, it may be worthwhile to investigate sorting algorithms which do not operate by partitioning the sorting task into disjoint subproblems, but instead perform successively refined approximate sorts over the entire set of keys.

Appendix A

Expansion Properties of the Hypercube

The calculations in this appendix analyze the volume-to-surface ratio of a Hamming ball of radius $r = r(d)$ lying in a hypercube of dimension d. Theorem A.1.1, which is used in Section 4.1.2, characterizes the asymptotic behavior of this ratio for r in the range 0 to $d/2$. The results could easily be extended to handle higher values of r (that is, $d/2 \leq r \leq d$) by taking advantage of symmetry.

A.1 Asymptotic Analysis

Definition A.1.1 Let $R_{d,r}$ denote $\sum_{0\leq l\leq r}\binom{d}{l}/\binom{d}{r}$.

Lemma A.1.1 Let d and r be positive integers, $1 \leq r \leq d$. Then $R_{d,r} > R_{d,r-1}$.

Proof: Observe that $R_{d,0} = 1$ and for $1 \leq r \leq d$

$$\begin{aligned}
R_{d,r}/R_{d,r-1} &= \frac{r}{d-r+1}\left(\sum_{0\leq l\leq r}\binom{d}{l}\Bigg/\sum_{0\leq l\leq r-1}\binom{d}{l}\right) \\
&> \frac{r}{d-r+1}\min_{1\leq l\leq r}\binom{d}{l}\Bigg/\binom{d}{l-1} \\
&= \frac{r}{d-r+1}\min_{1\leq l\leq r}\frac{d-l+1}{l} \\
&= 1.
\end{aligned}$$

□

Lemma A.1.2 For positive integers d and r, $r \leq d/2$,

$$R_{d,r} = \Theta\left(\sqrt{\frac{d}{z+1}}\right),$$

where $r = d/2 - \sqrt{dz}$.

Proof: The following three cases will be considered separately: $z = \Omega(d)$ and $z \leq d/4$; $z = o(d)$ and $z \geq 1$; $0 \leq z \leq 1$.

Case 1: $z = \Omega(d)$ and $z \leq d/4$. Exercise (9.42) of Graham, Knuth and Patashnik [GKP89] establishes that $R_{d,r} = \Theta(1)$ in this range.

Case 2: $z = o(d)$ and $z \geq 1$. It is sufficient to prove that $R_{d,r} = \Theta\left(\sqrt{d/z}\right)$ since $z = \Omega(1)$. For the lower bound, consider the inequality

$$\sum_{0 \leq l \leq r} \binom{d}{l} \geq \left\lfloor \sqrt{d/z} \right\rfloor \binom{d}{r - \left\lfloor \sqrt{d/z} \right\rfloor},$$

and observe that for sufficiently large values of d

$$\begin{aligned}
\binom{d}{r - \left\lfloor \sqrt{d/z} \right\rfloor} \Big/ \binom{d}{r} &\geq \left(\frac{r - \left\lfloor \sqrt{d/z} \right\rfloor}{d - r + \left\lfloor \sqrt{d/z} \right\rfloor}\right)^{\left\lfloor \sqrt{d/z} \right\rfloor} \\
&\geq \left(\frac{d/2 - \sqrt{dz} - \sqrt{d/z}}{d/2 + \sqrt{dz} + \sqrt{d/z}}\right)^{\sqrt{d/z}} \\
&\geq \left(\frac{d/2 - 2\sqrt{dz}}{d/2 + 2\sqrt{dz}}\right)^{\sqrt{d/z}} \\
&\geq \left(1 - \frac{8}{\sqrt{d/z}}\right)^{\sqrt{d/z}},
\end{aligned}$$

which converges to $e^{-8} = \Omega(1)$. Hence, $R_{d,r} = \Omega(\sqrt{d/z})$.

For the upper bound, note that

$$\binom{d}{l-1} \Big/ \binom{d}{l} \leq \frac{r}{d - r + 1},$$

for $1 \le l \le r$. Hence, the sum $\sum_{0\le l\le r}\binom{d}{l}$ is dominated by the infinite geometric progression with initial value $\binom{d}{r}$ and ratio $r/(d-r+1)$ between successive terms. Thus,

$$\begin{aligned} R_{d,r} &\le \frac{d-r+1}{d-2r+1} \\ &= O\left(\sqrt{d/z}\right). \end{aligned}$$

Case 3: $0 \le z \le 1$. It is sufficient to prove that $R_{d,r} = \Theta\left(\sqrt{d}\right)$ since $z = O(1)$. A lower bound on $R_{d,r}$ can be obtained as follows:

$$\begin{aligned} \sum_{0\le l\le r}\binom{d}{l} &\ge \sum_{r-\lfloor\sqrt{d}\rfloor\le l\le r}\binom{d}{l} \\ &\ge \left\lfloor\sqrt{d}\right\rfloor\binom{d}{r-\left\lfloor\sqrt{d}\right\rfloor} \end{aligned}$$

Furthermore, for sufficiently large values of d

$$\begin{aligned} \binom{d}{r-\left\lfloor\sqrt{d}\right\rfloor}\Big/\binom{d}{r} &\ge \left(\frac{r-\left\lfloor\sqrt{d}\right\rfloor}{d-r+\left\lfloor\sqrt{d}\right\rfloor}\right)^{\lfloor\sqrt{d}\rfloor} \\ &\ge \left(\frac{d/2-2\sqrt{d}}{d/2+2\sqrt{d}}\right)^{\sqrt{d}} \\ &\ge \left(1-\frac{8}{\sqrt{d}}\right)^{\sqrt{d}}, \end{aligned}$$

which converges to $e^{-8} = \Omega(1)$. Hence, $R_{d,r} = \Omega(\sqrt{d})$.

For the upper bound, Stirling's approximation can be used to show that $R_{d,\lfloor d/2\rfloor} = \Theta(\sqrt{d})$. It follows from Lemma A.1.1 that $R_{d,r} = O(\sqrt{d})$. □

Theorem A.1.1 Let d be a given integer and let $r = r(d)$ be an integer between 0 and $d/2$. If $\sum_{0\le l\le r}\binom{d}{l} = 2^{d(1-1/k)}$ then $R_{d,r} = \Theta(\sqrt{k})$.

Proof: The following four cases will be considered separately: $r = d/2 - \Omega(d)$ and $r \ge 0$; $r = d/2 - o(d)$ and $r \le d/2 - d^{2/3}$; $d/2 - d^{2/3} \le r \le d/2 - \sqrt{d}$; $d/2 - \sqrt{d} \le r \le d/2$.

Case 1: $r = d/2 - \Omega(d)$ and $r \ge 0$. In this range, $R_{d,r} = \Theta(1)$ by Lemma A.1.2, and $k = \Theta(1)$ by Exercise (9.42) of [GKP89]. Hence, $R_{d,r} = \Theta(\sqrt{k})$.

Case 2: $r = d/2 - o(d)$ and $r \le d/2 - d^{2/3}$. Let $r = d/2 - d^{1/2+\delta}$, hence $\delta = 1/2 - \omega(1/\log d)$ and $\delta \ge 1/6$. As in the preceding case, $R_{d,r} = \Theta(d^{1/2-\delta})$ by Lemma A.1.2.

The logarithmic form of Stirling's approximation implies that $\ln n! = n\ln n - n + O(\ln n)$. Hence,

$$\begin{aligned}
\ln\binom{d}{r} &= d\ln d - r\ln r - (d-r)\ln(d-r) + O(\ln d)\\
&= d\ln d - (d/2 - d^{\delta+1/2})\ln(d/2 - d^{\delta+1/2})\\
&\quad -(d/2 + d^{\delta+1/2})\ln(d/2 + d^{\delta+1/2}) + O(\ln d)\\
&= d\ln 2 - (d/2 - d^{\delta+1/2})\ln(1 - 2d^{\delta-1/2})\\
&\quad -(d/2 + d^{\delta+1/2})\ln(1 + 2d^{\delta-1/2}) + O(\ln d).
\end{aligned}$$

The following pair of inequalities may be easily derived from the Taylor's series expansion of $\ln(1+x)$:

$$x - x^2/2 \le \ln(1+x) \le x,\ \ x \ge 0,\ \text{and}$$
$$-x - x^2 \le \ln(1-x) \le -x - x^2/2,\ 0 \le x \le \frac{1}{2}.$$

These inequalities imply that for sufficiently large values of d,

$$\begin{aligned}
\ln\binom{d}{r} &\ge d\ln 2 - (d/2 - d^{\delta+1/2})(-2d^{\delta-1/2} - 2d^{2\delta-1})\\
&\quad -(d/2 + d^{\delta+1/2})(2d^{\delta-1/2}) + O(\ln d)\\
&= d\ln 2 - 3d^{2\delta} - 2d^{3\delta-1/2} + O(\ln d)\\
&\ge d\ln 2 - 5d^{2\delta} + O(\ln d),
\end{aligned}$$

and

$$\begin{aligned}
\ln\binom{d}{r} &\le d\ln 2 - (d/2 - d^{\delta+1/2})(-2d^{\delta-1/2} - 4d^{2\delta-1})\\
&\quad -(d/2 + d^{\delta+1/2})(2d^{\delta-1/2} - 2d^{2\delta-1}) + O(\ln d)\\
&= d\ln 2 - d^{2\delta} - 2d^{3\delta-1/2} + O(\ln d)\\
&\le d\ln 2 - d^{2\delta} + O(\ln d).
\end{aligned}$$

Thus, $\sum_{0\le l\le r}\binom{d}{l} = R_{d,r}\binom{d}{r} = 2^{d-\Theta(d^{2\delta})}$ where the $O(\ln d)$ term has been absorbed into the $\Theta(d^{2\delta})$ term (using the fact that $\delta \ge 1/6$). Hence, $k = \Theta(d^{1-2\delta})$ and $R_{d,r} = \Theta(\sqrt{k})$.

Case 3: $d/2 - d^{2/3} \le r \le d/2 - \sqrt{d}$. Let $r = d/2 - d^{1/2+\delta}$, $0 \le \delta \le 1/6$. In this case, $R_{d,r} = \Theta(d^{1/2-\delta})$ by Lemma A.1.2. Equation (9.98) of [GKP89] implies that

$$\binom{d}{r} = \Theta\left(\frac{2^d}{\sqrt{d}}e^{-2d^{2\delta}}\right),$$

so multiplying by $R_{d,r}$ gives

$$\sum_{0 \leq l \leq r} \binom{d}{l} = \Theta\left(\frac{2^d}{d^\delta} e^{-2d^{2\delta}}\right) = \Theta(2^{d(1-1/k)}),$$

for $k = d \ln 2/(\delta \ln d + 2d^{2\delta})$. Observe that $k = \Theta(d^{1-2\delta})$ since $d^{2\delta} = \Omega(\delta \ln d)$, $\delta \geq 0$. Hence $R_{d,r} = \Theta(\sqrt{k})$.

Case 4: $d/2 - \sqrt{d} \leq r \leq d/2$. In this case, $R_{d,r} = \Theta(\sqrt{d})$ by Lemma A.1.2. Together with Equation (9.98) of [GKP89], this implies that $\sum_{0 \leq l \leq r} \binom{d}{l} = \Theta(2^d)$. Hence, $k = \Theta(d)$ and $R_{d,r} = \Theta(\sqrt{k})$. $\square$

Bibliography

[ADKF70] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzev. On economical construction of the transitive closure of a directed graph. *Soviet Math. Dokl.*, 11:1209–1210, 1970.

[AH88] A. Aggarwal and M.-D. A. Huang. Network complexity of sorting and graph problems and simulating CRCW PRAMs by interconnection networks. In J. H. Reif, editor, *VLSI Algorithms and Architectures: Proceedings of the 3rd Aegean Workshop on Computing,* Lecture Notes in Computer Science, volume 319, pages 339–350. Springer-Verlag, 1988.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.

[AL78] T. Agerwala and B. Lint. Communication in parallel algorithms for Boolean matrix multiplication. In *Proceedings of the 1978 IEEE International Conference on Parallel Processing*, pages 146–153, 1978.

[AMW88] R. J. Anderson, E. W. Mayr, and M. K. Warmuth. Parallel approximation algorithms for bin packing. Technical Report STAN–CS–88–1200, Stanford University, Department of Computer Science, March 1988.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference,* vol. 32, pages 307–314, 1968.

[BCLR86] S. N. Bhatt, F. R. K. Chung, F. T. Leighton, and A. L. Rosenberg. Optimal simulations of tree machines. In *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science*, pages 274–282, October 1986.

[BGLY81] A. Borodin, L. J. Guibas, N. A. Lynch, and A. C. Yao. Efficient searching using partial ordering. *Information Processing Letters*, 12:71–75, 1981.

[Ble87] G. E. Blelloch. Scans as primitive parallel operations. In *Proceedings of the 1987 IEEE International Conference on Parallel Processing*, pages 355–362, 1987.

[BS78] G. Baudet and D. Stevenson. Optimal sorting algorithms for parallel computers. *IEEE Transactions on Computers*, C–27:84–87, 1978.

[Col86] R. Cole. An optimal selection algorithm. Technical Report #209, Ultracomputer Research Laboratory, March 1986.

[Col88] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17:770–785, 1988.

[CS92] R. E. Cypher and J. L. C. Sanz. Cubesort: A parallel algorithm for sorting N data items with S-sorters. *Journal of Algorithms*, 13:211–234, 1992.

[CW82] D. Coppersmith and S. Winograd. On the asymptotic complexity of matrix multiplication. *SIAM J. Comput.*, 11:472–492, 1982.

[CW87] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 1–6, May 1987.

[CY85] R. Cole and C. K. Yap. A parallel median algorithm. *Information Processing Letters*, 20:137–139, 1985.

[Cyp93] R. E. Cypher. Theoretical aspects of VLSI pin limitations. *SIAM J. Comput.*, 22:58–63, April 1993.

[DNS81] E. Dekel, D. Nassimi, and S. Sahni. Parallel matrix and graph algorithms. *SIAM J. Comput.*, 10:657–675, 1981.

[DPSR83] M. Dowd, Y. Perl, M. Saks, and L. Rudolph. The balanced sorting network. Technical Report DCS–TR–127, Department of Computer Science, Rutgers University, June 1983.

[FF81] P. Frankl and Z. Füredi. A short proof for a theorem of Harper about Hamming spheres. *Discrete Mathematics*, 34:311–313, 1981.

[Fic83] F. Fich. New bounds for parallel prefix circuits. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, pages 100–109, May 1983.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th ACM Symposium on Theory of Computing*, pages 114–118, May 1978.

[GK84] A. Gottlieb and C. P. Kruskal. Complexity results for permuting data and other computations on parallel processors. *Journal of the ACM*, 31:193–209, 1984.

[GKP89] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, Reading, MA, 1989.

[Har66] L. Harper. Optimal numberings and isoperimetric problems on graphs. *J. Combinatorial Theory*, 1:385–393, 1966.

[HJ86] C.-T. Ho and S. L. Johnsson. Distributed routing algorithms for broadcasting and personalized communication in hypercubes. In *Proceedings of the 1986 IEEE International Conference on Parallel Processing*, pages 640–648, 1986.

[Hoa62] C. A. R. Hoare. Quicksort. *Computer J.*, 5:10–15, 1962.

[Joh87] S. L. Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *J. Parallel and Distributed Computing*, 4:133–172, 1987.

[KMR88] R. M. Karp, R. Motwani, and P. Raghavan. Deferred data structuring. *SIAM J. Comput.*, 17:883–902, 1988.

[Knu73] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, MA, 1973.

[Lei] F. T. Leighton. Personal communication.

[Lei83] F. T. Leighton. *Complexity Issues in VLSI: Optimal Layouts for the Shuffle-Exchange Graph and Other Networks*. MIT Press, Cambridge, MA, 1983.

[Lei85] F. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C–34:344–354, 1985.

[LF80] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27:831–838, 1980.

[NS81] D. Nassimi and S. Sahni. A self routing Benes network and parallel permutation algorithms. *IEEE Transactions on Computers*, C–30:332–340, 1981.

[NS82] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *Journal of the ACM*, 29:642–667, 1982.

[Pel] D. Peleg. Personal communication.

[Pla87] C. G. Plaxton. Virtual processing on the hypercube. Ph.D. Programming Project, Stanford University, October 1987.

[PU89] D. Peleg and E. Upfal. The token distribution problem. *SIAM J. Comput.*, 18:229–243, 1989.

[Rei84] J. H. Reif. Probabilistic parallel prefix computation. In *Proceedings of the 1984 IEEE International Conference on Parallel Processing*, pages 291–298, 1984.

[RV87] J. H. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM*, 34:60–76, 1987.

[Sch80] J. T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2:484–521, 1980.

[Sch82] A. Schönhage. Partial and total matrix multiplication. *SIAM J. Comput.*, 10:434–456, 1982.

[Sto71] H. S. Stone. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers*, C–20:153–161, 1971.

[Str69] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[Str86] V. Strassen. The asymptotic spectrum of tensors and the exponent of matrix multiplication. In *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science*, pages 49–54, October 1986.

[Ull84] J. D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, Rockville, MD, 1984.

[Val75] L. G. Valiant. Parallelism in comparison problems. *SIAM J. Comput.*, 4:348–355, 1975.

[Van71] D. C. Van Voorhis. Large $[g, d]$ sorting networks. Technical Report STAN–CS–71–239, Stanford University, Department of Computer Science, August 1971.

[VD88] P. Varman and K. Doshi. Sorting with linear speedup on a pipelined hypercube. Technical Report TR–8802, Rice University, Department of Electrical and Computer Engineering, February 1988.

[Vis83] U. Vishkin. An optimal parallel algorithm for selection. Technical Report 106, Courant Institute of Mathematical Sciences, Department of Computer Science, December 1983.

[Wag86] B. Wagar. Hyperquicksort: A fast sorting algorithm for hypercubes. In *Proceedings of the Second Conference on Hypercube Multiprocessors*, pages 292–299, 1986.