

AD-A227 144

UMIACS-TR-90-37
CS-TR -2426

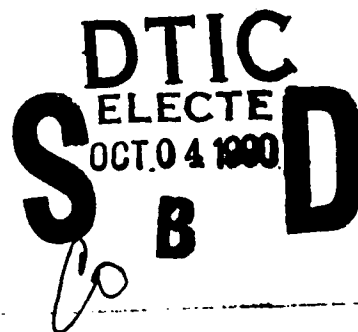
March 1990

**Efficient Parallel Algorithms on the
Network Model**

Kwan Woo Ryu

Department of Computer Science
University of Maryland
College Park, MD 20742

**COMPUTER SCIENCE
TECHNICAL REPORT SERIES**



**UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742**

UMIACS-TR-90-37
CS-TR -2426

March 1990

Efficient Parallel Algorithms on the Network Model

Kwan Woo Ryu

Department of Computer Science
University of Maryland
College Park, MD 20742

DTIC
ELECTE
OCT 04 1990
S B D

EFFICIENT PARALLEL ALGORITHMS ON THE NETWORK MODEL

by

Kwan Woo Ryu

Dissertation submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1990

Advisory Committee:

Professor Joseph F. JáJá
Associate Professor Carl H. Smith
Associate Professor Clyde P. Kruskal
Assistant Professor Howard Elman
Professor Uzi Vishkin

ABSTRACT

Title of Dissertation: Efficient Parallel Algorithms on The Network Model
 Kwan Woo Ryu, Doctor of Philosophy, 1990
 Dissertation directed by: Joseph F. Jájá, Professor, Electrical Engineering

We develop efficient parallel algorithms for several fundamental problems on the hypercube, the shuffle-exchange, the cube-connected cycles and the butterfly. Those problems are related to load balancing, packet routing, list ranking, graph theory and VLSI routing.

Load balancing, sorting and packet routing problems have been studied heavily on various parallel models. There are some optimal algorithms for these problems on few networks. We introduce a new simple and efficient algorithm for load balancing on our networks, and show that load balancing requires more time on our bounded-degree networks than on the weak hypercube. We also show that sorting n integers, each of which bounded by $p^{O(1)}$, can be done in $O(\frac{n}{p})$ time on the pipelined hypercube, whenever $n = \Omega(p^{1+\epsilon})$, for some fixed $\epsilon > 0$. Using these results, we provide an efficient algorithm for packet routing on several networks.

An algorithm will be called almost uniformly optimal if it is provably optimal whenever $p \leq \frac{n}{\log^k n}$, for some fixed constant k . We present almost uniformly optimal algorithms to solve several problems such as the all nearest smaller values (ANSV) problem and the line packing problem on our networks.

List ranking is a basic problem whose efficient solution can be used in many graph algorithms. We describe an algorithm to solve the list ranking problem on the pipelined hypercube in time $O(\frac{n}{p})$ when $n = \Omega(p^{1+\epsilon})$, and in time $O(\frac{n \log n}{p} + \log^3 p)$ otherwise. This clearly attains a linear speed-up when $n = \Omega(p^{1+\epsilon})$. We use this algorithm to obtain efficient algorithms for many basic graph problems such as tree expression evaluation, connected and biconnected components, ear decomposition and st-numbering on the networks.

Finally, we develop parallel algorithms for several one-layer routing problems. It is shown that the detailed routing and the routability testing problems within a rectangle can each be solved in time $O(\frac{n}{p})$ on the pipelined hypercube when $n = \Omega(p^{1+\epsilon})$. These problems are also addressed in the other network models.

For	
1	☒
Unannounced Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
12	

©Copyright by

Kwan Woo Ryu

1990

To my wife, Jung-Yi

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my Lord Jesus Christ, for guiding me and encouraging me at every step of this thesis. He introduced me to Professor Joseph JáJá whose guidance and encouragement knew no bounds. Professor Joseph JáJá has been more than an ideal dissertation advisor to me; he was my mentor. I couldn't have found a better teacher and a greater scholar as my advisor. I would also like to thank the other members of my committee, Professors Clyde Kruskal, Uzi Vishkin, Carl Smith, and Howard Elman. I am especially grateful to Clyde Kruskal for his kindness during my research and helpful advice in writing the outline and the conclusion of this thesis. I am also grateful to Shing-Chong Chang for his ideas on the solutions of the one-layer routing problems in Chapter 5.

There are many other people who directly and indirectly encouraged me during my stay in America. I would like to thank Rev. Dong Won Lee, the pastor of The First Korean Baptist Church, whose prayerful sermons firmly guided and kept me to my Lord Jesus Christ. I am also grateful to the ordained deacons of the church for their help and love; in particular to Shin-Il Koh (and Joon-Bok Koh), Youn-Yull Bae, and Seung-Kyu Cho. I am grateful to the members of the student bible study group of the church for their prayer and encouragement; in particular to Gun-Sik Park, Se-Kyu Chang, Jae-Hung Ahn, Joon-Ho Park, Dong-Hun Lee, Ssang-Ho Park, Lee-Kyu Park, Jong-Pil Yoon, Chan-Kyu Kang, Kyu-Seok Shim, Myung-Seok Kim, and their wives, Shi-Bok Ryu, Won-Kyung Park, Sung-Soo Chun, Kil-Ok Kim, etc. Without them my life in America would not have been so happy. I would like to thank all my friends in this university. Special thanks to Omer Berkman, Ramakrishna Thuriinnella, Chaitali Chakrabarti, Sridhar Krishnamurthy, Jagannathan Narasimhan, Ravi Kolagotla, Ying-Min Huang, Shu-Sun Yu, Bok-Kyu Joo, Eun-Sup Sim, Chong-Suk Rim, Chung-Hwan Kim, Young-Mi Choi, Hyun-Chul Kang, Sam-Hyuk Noh, Sungzoon Cho, Soo-Mook Moon, Jae-Heon Yang and Seongsoo Hong. I am also grateful to my neighbors, Hyun-Keun Yoon, Yong-Ook Cho, Hyung-Joo Lee, Jong-Hyuk Yoon, Chul-Jin Ahn, and their wives, for their warm and loving friendship.

I would like to express my gratitude to Korean Ministry of Education and Kyungpook National University for allowing and partially supporting me to study in America. I would also like to thank the Department of Computer Science, the Electrical Engineering Department, the Systems Research Center, and the Institute for Advanced Computer Studies (UMIACS) for their financial support.

Finally, I would especially like to thank my wife, Jung-Yi, for her prayer, patience, encouragement, and sincere love. She is God's greatest gift to me. I

would like to thank my children, Taejin, Hyungjin, and Gichun, for their cheer and patience during my research. I would also like to thank other family members, especially my mother, in Korea for their trust, patience, encouragement, and sincere love.

Contents

1	Introduction	1
1.1	Outline	1
1.2	The PRAM Model	5
1.2.1	NC and P-completeness	7
1.3	The Network Model	7
1.3.1	The Complete Network	9
1.3.2	The d-Dimensional Mesh	9
1.3.3	The Binary Tree Network	10
1.3.4	The Hypercube Network	10
1.3.5	The Butterfly Network	13
1.3.6	The Cube-Connected Cycles	14
1.3.7	The Shuffle-Exchange Network	15
1.4	Hypercube Algorithms	16
1.4.1	Normal Algorithms	16
1.4.2	Butterfly Communication Graphs	19
1.4.3	Conflict-Free Routing	21
2	Load Balancing, Sorting and Routing	23
2.1	Introduction	23
2.2	Basic Communication Schemes	24
2.3	Load Balancing	31
2.4	Sorting	35
2.4.1	Mergesort	35
2.4.2	Columnsort	36
2.5	Routing	41
2.6	Relationship With The CRCW Model	43
3	Almost Uniformly Optimal Algorithms	45
3.1	Introduction	45
3.2	ANSV and Related Problems	45
3.2.1	The Parentheses Matching Problem	50
3.2.2	Triangulating a Monotone Polygon	51
3.2.3	The All Nearest Neighbor Problem	51

3.3	VLSI Routing	51
3.4	Lower Bounds	54
4	List Ranking and Graph Algorithms	59
4.1	Introduction	59
4.2	List Ranking	59
4.3	Graph Problems	63
5	One-Layer Routing	71
5.1	Introduction	71
5.2	The Separation Problem	72
5.3	Routing In a Simple Polygon	74
5.4	Routability Testing	81
6	Conclusion	85
6.1	Summary	85
6.2	Future Research	86

List of Tables

1.1	Comparisons among the fixed interconnection networks	17
4.1	Performance on the pipelined hypercube (ϵ : constant)	69
4.2	Performance on the weak hypercube (ϵ : constant)	70

List of Figures

1.1	The PRAM model	6
1.2	The network model	8
1.3	Complete network of size eight	10
1.4	2-dimensional mesh of size 16	11
1.5	Binary tree network of size 15 and its layout	11
1.6	4-dimensional hypercube (a) and switches in each processor (b)	12
1.7	Butterfly network for $q = 3$ and $p = 4 \cdot 2^3$	14
1.8	Cube-connected cycles for $p = 3 \cdot 2^3$	15
1.9	Shuffle-exchange network of size eight	16
1.10	The F and R graphs for $p = 2^3$	19
1.11	The H_F and H_R corresponding to F and R of Figure 1.10	21
2.1	A Benes permutation network (a) and a butterfly permutation network (b)	28
2.2	Consecutive and cyclic storage of a matrix	30
2.3	Step by step implementation of BALANCE	33
2.4	A 6×3 matrix before and after sorting	37
2.5	The transpose and its inverse	37
2.6	The shift and its inverse	38
2.7	The step by step implementation of the column sort	39
2.8	The hypercube implementation of step 2 of the column sort. Steps [s3]-[s5] are omitted	40
3.1	Subsequences corresponding to (a) $i' = i + 1$ and (b) $i' > i + 1$	48
3.2	(a) A channel routing instance, (b) corresponding sorted list and prefix sums, (c) chains of intervals	53
5.1	Basic river routing problem	73
5.2	Basic river routing around a rectangle boundary	75
5.3	The step-by-step illustration of the overall strategy of routing representative nets	77
5.4	The union of the bounding perimeters. B_i is the bounding perimeter of N_i	78
5.5	Routability testing	84

Chapter 1

Introduction

In recent years, we have seen a tremendous surge in the availability of very fast and inexpensive hardware. This has been made possible partly by the use of faster circuit technologies and smaller feature sizes; partly by novel architectural features such as pipelining, vector processing, cache memories, and systolic arrays; and partly by using novel interconnections between processors and memories, such as hypercube, omega network, cube-connected cycles, and others. Such hardware technologies made it possible to design parallel computers – computers consisting of a number of processors dedicated to solving a single problem at a time – by putting thousands of processors together. In fact, parallel computers, such as the Connection Machine from Thinking Machines Inc. and the iPSC series from Intel Corp., with thousands of processors, are already available commercially. However, the tremendous computing power that has become available can be realized only if we design efficient parallel algorithms which run on these parallel computers. In this thesis, we develop parallel algorithms which can be efficiently implemented on models that are abstractions of such parallel computers.

The rest of the chapter is organized as follows. The outline, the main contributions, and the summary of results of the thesis are described in Section 1.1. The computational models – the PRAM model and the network model – are reviewed briefly in Sections 1.2 and 1.3, respectively. The last section reviews several basic hypercube algorithms and routing schemes.

1.1 Outline

Let $T_1(n)$ be the running time of the optimal sequential algorithm¹ for solving a problem Π , where n is the length of the input of Π . Let $T_p(n)$ be the running

¹An optimal sequential algorithm does not necessarily exist. See [43] for a discussion of the issue. All of the problems discussed in this thesis have optimal sequential algorithms, so this difficulty does not arise.

time of a parallel algorithm for solving Π with p processors. Then the *speed-up*, $S_p(n)$, of the parallel algorithm for solving Π is $\frac{T_1(n)}{T_p(n)}$. It measures how many times faster a parallel algorithm is than a sequential one. Clearly, $1 \leq S_p(n) \leq p$. The *efficiency*, $E_p(n)$, of the parallel algorithm is $\frac{S_p(n)}{p}$. It measures how effective each processor in a parallel algorithm is relative to a sequential algorithm. It normalizes the speed-up, so $0 < E_p(n) \leq 1$. A parallel algorithm that runs in time $T_p(n)$ is said to be *efficient* if $E_p(n) = \Theta(1)$, and is said to be *almost efficient* if $E_p(n) = \Omega(\frac{1}{\log n})$. A primary goal in parallel computation is to design efficient or almost efficient algorithms that also run as fast as possible [43].

The goal of this thesis can be described as follows: Given a network $\mathcal{N}$ with p processors and a problem Π , find an efficient algorithm to solve Π on $\mathcal{N}$ for all problem sizes $n \geq p$. Since this goal is difficult to achieve, we will often be satisfied with finding an efficient algorithm when $n \geq p(\log p)^{\Theta(1)}$, or even when $n = \Omega(p^{1+\epsilon})$ for some fixed $\epsilon > 0$. In this thesis, we show several results related to load balancing, sorting, packet routing, list ranking, graph theory, and VLSI routing on the pipelined hypercube, the weak hypercube, the shuffle-exchange, the cube-connected cycles, and the butterfly. These results include the following.

- Development of provably efficient algorithms on the network models.
- Establishment of lower bounds on weak hypercube and bounded-degree networks. All the problems considered are shown to require $\Omega(\frac{n \log p}{p})$ time on bounded-degree networks.

These results shed some light on the relative powers of the pipelined hypercube, the weak hypercube, and the bounded-degree networks.

We now outline the thesis, and consider the main contributions one by one. In Chapter 2, we show several results related to load balancing, sorting, and relate them to the general packet routing problem.

Balancing load among processors is very important since poor balance of load generally causes poor processor utilization. The *load balancing* problem is defined as follows. Let n items be distributed over the p processors of a network, with no more than M items assigned to any single processor ($\lceil n/p \rceil \leq M \leq n$). The problem is to redistribute the items so that the number of items in any two processors may differ by at most one.

Kruskal et al. studied load balancing (to solve the list ranking problem) on the complete network [42]. Peleg and Upfal developed an algorithm for this problem whose time complexity is $O(M + \log p \cdot \min(\log \frac{n}{p}, \log \log p))$ on the bounded-degree network based on expander graphs [58]. Plaxton developed a weak hypercube algorithm whose time complexity is $O(M\sqrt{\log p} + \log^2 p)$ [61].

We present an algorithm for load balancing whose time complexity is $O(M + \log p)$ on the pipelined hypercube and $O(M \log p)$ on the shuffle-exchange, cube-connected cycles and butterfly. This algorithm is optimal on the pipelined hypercube. We also provide a lower bound for our bounded-degree networks, and

show that load balancing requires more time on the shuffle-exchange, the cube-connected-cycles, or the butterfly than on the weak hypercube.

Sorting is a fundamental computational problem that has been investigated for several decades. This problem can be solved in $\Theta(n \log n)$ time sequentially. More recently, efficient parallel sorting algorithms on the PRAM model [12,24] and on the network model [1,18,39,47,55,61,64,78,82] have been developed. Cole developed an optimal $O(\frac{n \log n}{p})$ algorithm for the EREW PRAM [12]. On the network model, Leighton developed an $O(\frac{n \log n}{p})$ algorithm for his bounded-degree network based on the AKS sorting network [47]; Cypher and Sanz developed an $O(k \frac{n \log n}{p})$ algorithm for shuffle-exchange when $n = p^{1+\frac{1}{k}}$ for some $k \leq 2$ [18]; and Varman and Doshi developed an $O(\frac{n \log n}{p} + \log^2 p)$ algorithm for the pipelined hypercube [82].

Many of our algorithms in this thesis need to sort integers from a small range efficiently. This can be done in linear time sequentially when the range is polynomial in the number of integers. For parallel algorithm, Hagerup developed an $O(\frac{n \log \log n}{p})$ algorithm for the CRCW PRAM, for $1 \leq p \leq \frac{n \log \log n}{\log n}$ [24]. Clearly, this algorithm is not efficient.

On the network model, Aggarwal and Huang developed an $O(\frac{n \log n}{p})$ algorithm for the cube-connected cycles [1], and Han developed an $O(\frac{n}{p})$ algorithm for the complete network [27], whenever $n = \Omega(p^{1+\epsilon})$. We present an $O(\frac{n}{p})$ algorithm for the pipelined hypercube, whenever $n = \Omega(p^{1+\epsilon})$. Integer sorting requires $\Omega(\frac{n \log p}{p})$ time on the weak hypercube and on any bounded-degree network. Thus, Aggarwal and Huang, and our algorithms are optimal (for $n = \Omega(p^{1+\epsilon})$).

The load balancing algorithm and the integer sorting algorithm are used to find an efficient solution for the *general packet routing* problem. The (n, k_1, k_2) routing problem is a set of n packets, each of which is specified by a source and a destination, such that no processor appears as a source (respectively destination) in more than k_1 (respectively k_2) packets. The problem is to route these requests simultaneously. When $k_1 = k_2 = \frac{n}{p}$, this problem reduces to a permutation problem. Gottlieb and Kruskal showed that this permutation problem requires $\Omega(\frac{n \log p}{p})$ time on any bounded degree network [23]. Clearly, this permutation problem can be solved by using the integer sorting algorithms.

Peleg and Upfal developed an algorithm for this routing problem whose time complexity $\Theta(k_1 + k_2 + \frac{n \log n}{p})$ on their bounded-degree network based on expander graphs [58].

We develop an algorithm whose time complexity is $O(k_1 + k_2 + \frac{n}{p})$ on the pipelined hypercube, and $O((k_1 + k_2) \log p + \frac{n \log p}{p})$ on the weak hypercube and our bounded-degree networks, whenever $n = \Omega(p^{1+\epsilon})$. The problem requires $\Omega(\frac{n \log p}{p})$ time on the weak hypercube and on any bounded-degree networks. Thus the the upper bounds are tight for these networks.

A parallel algorithm is *almost uniformly optimal* if its running time is provably the best possible (up to a constant factor) for all $p \leq n/\log^k n$, for some fixed constant k . In Chapter 3, we present almost uniformly optimal algorithms to solve several problems such as the all nearest smaller values problem (ANSV), triangulating a monotone polygon, and line packing.

The ANSV problem is a fundamental problem since it can be used to solve several important problems such as triangulating a monotone polygon, reconstructing a binary tree, parenthesis matching [7], and line packing [9]. There is a simple linear time sequential algorithm using a stack for this problem. For parallel algorithm, Berkman, Schieber, and Vishkin developed an optimal $O(\frac{n}{p} + \log \log n)$ for the CRCW PRAM [7].

We present an algorithm for the ANSV problem whose time complexity is $O(\frac{n}{p} + \log^4 p)$ on the pipelined hypercube and $O(\frac{n \log p}{p} + \log^4 p)$ on all the remaining networks. This network algorithm is used to find algorithms for triangulating a monotone polygon and line packing. We also prove that the problems require $\Omega(\frac{n \sqrt{\log p}}{p})$ time on the weak hypercube and $\Omega(\frac{n \log p}{p})$ time on our bounded-degree networks. Thus, these algorithms are also almost uniformly optimal on our bounded-degree networks (despite being only almost efficient).

In Chapter 4, we present an algorithm to solve the list ranking problem on the networks. This is also a fundamental problem and there are many known results for this problem [2,14,15,16,17,26,27,42]. This problem has a simple linear time sequential algorithm.

Wyllie developed the first parallel algorithm for this problem [87]. This algorithm uses the doubling technique and can be implemented in $O(\frac{n \log n}{p})$ time on the EREW PRAM. Cole and Vishkin developed $O(\frac{n}{p} + \log n)$ time algorithms for the CRCW PRAM [16], and for the EREW PRAM [17]. Anderson and Miller developed a simplified $O(\frac{n}{p} + \log n)$ time algorithm for the EREW PRAM [2]. For the network model, Kruskal et al. developed an $O(\frac{n}{p} + p^2)$ algorithm on the complete network [42]. Han improved this result to $O(\frac{n}{p} + p \log p)$ [27].

We present a list ranking algorithm that runs on the pipelined hypercube in time $O(\frac{n}{p})$ when $n = \Omega(p^{1+\epsilon})$, and in time $O(\frac{n \log n}{p} + \log^3 p)$ otherwise. We use these techniques to obtain fast algorithms for several basic graph problems such as tree expression evaluation, connected and biconnected components, ear decomposition, and st-numbering. These problems are also addressed for the other network models. We also prove that list ranking requires $\Omega(\frac{n \log p}{p})$ time on the weak hypercube and any bounded-degree network. Thus, our algorithm is optimal.

In Chapter 5, we present fast network algorithms for several one-layer routing problems. Actually, many of the optimization problems arising in VLSI routing are NP-complete [41,46,67,76]. One notable exception is the class of *one-layer routing* problems associated with a hierarchical layout strategy such as Bristle-Blocks [36]. See [13,20,48,49,51,53,59,72,79] for more examples. Efficient serial

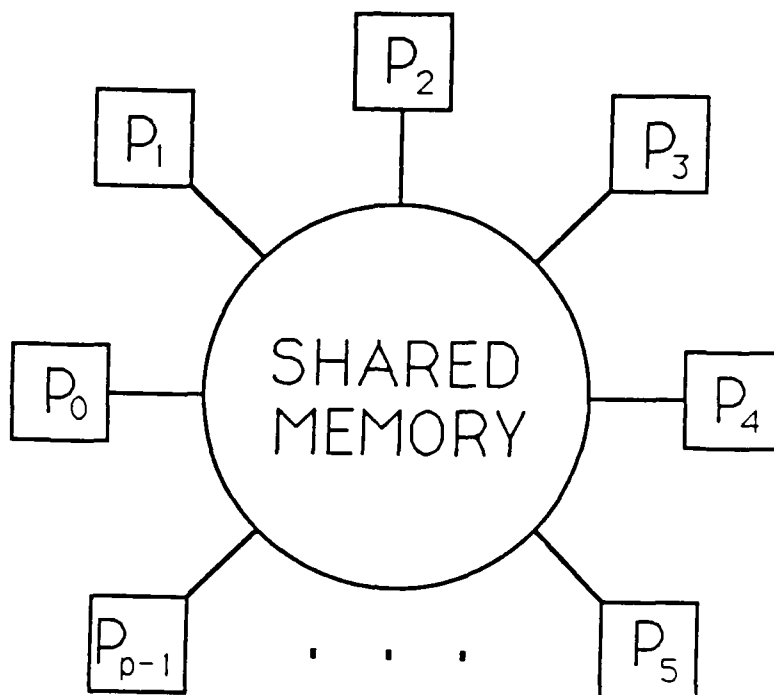


Figure 1.1: The PRAM model

solutions have been developed for most of these problems. For parallel solutions, algorithms that could run in $O(\frac{n}{p} + \log n)$ time on the CREW PRAM, and in $O(\frac{n}{p} + \frac{\log n}{\log \log n})$ time on the CRCW PRAM were developed for several one-layer routing problems [10].

We present fast algorithms for the detailed routing and the routability testing problems within a rectangle whose time complexities are $O(\frac{n}{p})$ on the pipelined hypercube, and $O(\frac{n \log p}{p})$ on all the remaining networks, when $n = \Omega(p^{1+\epsilon})$. Fast algorithms are also developed for several subproblems that are interesting on their own. One such subproblem is to determine the contours of the union of sets of contours within a rectangle.

In Chapter 6, we summarize the results obtained in this thesis and describe directions for future research.

1.2 The PRAM Model

The *PRAM* (*Parallel Random Access Machine*) consists of p synchronous processors, $P_0, P_1, \dots, P_{p-1}$, all having access to and interchanging data through a large shared memory (Figure 1.1). In a single cycle, each processor may read or write a data from or into a shared memory cell, or else perform in local mem-

ory one of a prescribed set of operations (various tests, arithmetic operations, Boolean operations, etc.). Each processor P_i , $0 \leq i \leq p-1$, is uniquely identified by an index i which can be referred to in the program.

There are several variations of the above general model based on the assumptions regarding the handling of the simultaneous access of several processors to a single location of the common memory. An *EREW* (*Exclusive-Read Exclusive-Write*) PRAM does not allow simultaneous access by more than one processor to the same memory location. A *CREW* (*Concurrent-Read Exclusive-Write*) PRAM allows simultaneous access only for read instructions. A *CRCW* (*Concurrent-Read Concurrent-Write*) PRAM allows simultaneous access for both read and write instructions. On the *Common CRCW* PRAM, it is assumed that if several processors attempt to write simultaneously at the same memory location, then all of them are trying to write the same value. In the *Arbitrary CRCW* PRAM, it is assumed that one of the processors attempting to write simultaneously at the same memory location succeeds, but we do not know in advance which one. On the *Priority CRCW* PRAM, it is assumed that the processor with minimum index among the processors attempting to write simultaneously into the same memory location succeeds. It turns out that all these machines do not differ substantially in their computing power, and that their computing power increases in a strict fashion in the order they were introduced.

The PRAM model of parallel computation was first studied in the late 70's by a number of researchers, and has become widely used henceforth. This research work presents some justifications to the selection of this model as an abstract model of parallel computation. The reader is referred to [21,38,54,83] for surveys of results concerning the PRAM.

The efficiency of a parallel algorithm is measured by its running time and the number of processors it uses. These two measures are strongly related. The following theorem due to Brent [8] implies that we can always slow down a parallel algorithm by reducing the number of its processors with the same processor-time product. This is the reason why we often measure the efficiency of a parallel algorithm by its minimal running time and the number of processors required to achieve this running time.

Theorem 1.1 *A PRAM algorithm requiring t parallel steps and a total of x operations can be implemented by a p -processor PRAM within $\lceil \frac{x}{p} \rceil + t$ parallel steps.*

Proof: Let x_i be the number of operations performed in step i , $1 \leq i \leq t$. The p -processor PRAM can perform the x_i operations in $\lceil \frac{x_i}{p} \rceil$ steps. Hence the total number of steps on the p -processor PRAM is

$$\sum_{i=1}^t \lceil \frac{x_i}{p} \rceil \leq \sum_{i=1}^t (\lceil \frac{x_i}{p} \rceil + 1) \leq \lceil \frac{x}{p} \rceil + t. \quad \square$$

1.2.1 NC and P-completeness

The study of parallel complexity within the PRAM model has led to some important negative results: there are some problems that are not likely to have fast parallel algorithms. Let P be the set of decision problems solvable by deterministic Turing machines in polynomial time, and let NC be the set of decision problems solvable in polylog time, *i.e.*, in time $O(\log^{O(1)} n)$, where n is the length of the input, using polynomial number of processors by deterministic algorithms [60]. Clearly, $NC \subseteq P$. A fundamental open question is whether $P \subseteq NC$. If it were so, it would mean, roughly speaking, that every problem that has a good solution in a sequential model of computation can be solved very fast in parallel, using a polynomial number of processors.

We adopt the usual convention of representing a decision problem as a subset of $\{0, 1\}^*$. Decision problem Π_1 is said to be *logspace reducible* to decision problem Π_2 if there is a function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ such that f is computable by any PRAM in polylog time using polynomial number of processors and, for all $x \in \{0, 1\}^*$, $x \in \Pi_1$ if and only if $f(x) \in \Pi_2$. A decision problem in P is called *P-complete* if every problem in P is logspace-reducible to it. If Π_1 is logspace-reducible to Π_2 and Π_2 is in NC , then Π_1 is in NC . This implies that if Π is a P -complete problem, then $P = NC$ if and only if $\Pi \in NC$. Thus the P -complete problems can be viewed as the problems in P most resistant to parallelization.

The usual method of showing that a problem is P -complete is to show that it lies in P and that some standard P -complete problem is logspace-reducible to it. P -complete problems include the circuit value problem, the greedy independent set problem, and the maxflow problem. See [38] for more details and a list of these problems.

1.3 The Network Model

The PRAM model can play a useful role as a theoretical yardstick for measuring the limits of parallel computation. Since the communication between its processors can be done trivially through its shared memory, this model can be also used to detect the intrinsic parallelism of a given problem. Moreover, the techniques and paradigms provided by the PRAM algorithms can be used for designing algorithms on a more realistic model. However, the PRAM model is not easy to realize physically because of physical fan-in limitations. In a physically realizable assemblage, we can only expect any computing element to have a small number of external connections. We must therefore consider parallel assemblages in which a large number of communicating processors, each with its own memory, are connected together, but where each processor communicates with a small number of other processors.

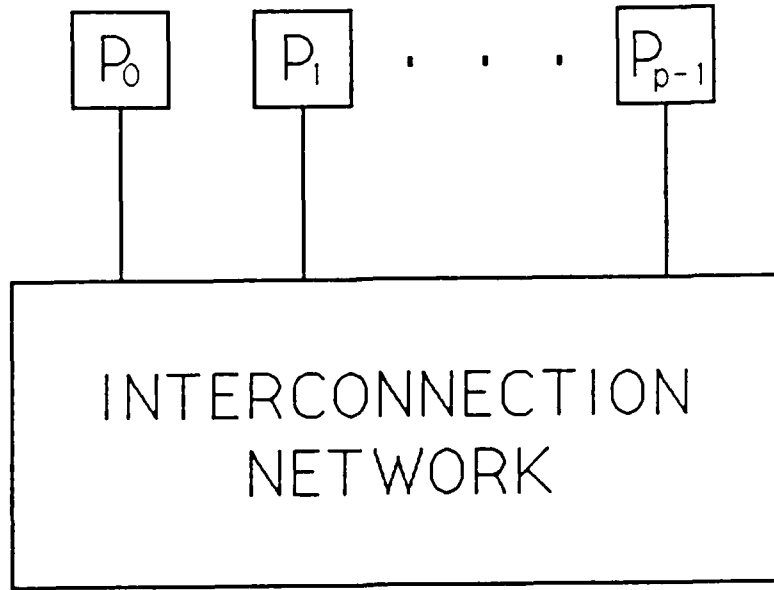


Figure 1.2: The network model

A p -processor *fixed interconnection network* may be viewed as an undirected graph, where vertices correspond to processors and edges correspond to communication links. Each processor P_i , $0 \leq i \leq p-1$, has a large local memory. There is no shared memory. We assume that the processors operate synchronously and they communicate with one another by sending and receiving data packets over the communication links provided by the network (Figure 1.2). Each processor can set up a single packet of bounded length in a unit of time.

The *distance* between two processors can be defined in the standard graph-theoretic way, *i.e.*, the distance from P_i to P_k is the minimum d for which there exists a sequence $P_i = P_{i_0}, P_{i_1}, \dots, P_{i_d} = P_k$, where P_{i_j} is directly connected to (or neighbor of) $P_{i_{j+1}}$, $0 \leq j < d$. The *diameter* of the network is the maximum distance between any two processors of the network. The *degree* of a processor is the number of its neighbors. The *degree of the network* is the maximum degree of any processor of the network. Any degree k network has diameter at least $\log_k p - 1$ [23]. A network is of *bounded-degree* if its degree is bounded. Hence, any bounded-degree network with p processors must have diameter $\Omega(\log p)$.

Another important property of the network is related to the graph-theoretic notion of separators. Formally, we say that a graph G has an $f(n)$ -separator ($f(n)$ -edge separator), or is $f(n)$ -separable ($f(n)$ -edge separable), if either it has only one vertex, or the following two statements are true.

- (1) Let n_0 be the number of vertices of G . Then there exist constants $\alpha < 1$ and $\beta > 0$ such that the removal of a set of at most $\beta f(n)$ vertices (edges) disconnects G into two graphs G_1 and G_2 , of n_1 and n_2 vertices each, such that $n_1 \leq \alpha n_0$ and $n_2 \leq (1 - \alpha)n_0$.

(2) Both G_1 and G_2 are $f(n)$ -separable ($f(n)$ -edge separable).

Note that $f(n)$ -edge separable graphs are $f(n)$ -separable, but the converse is not true in general. We say a family of graphs is $f(n)$ -separable ($f(n)$ -edge separable) if every member of the family is $f(n)$ -separable ($f(n)$ -edge separable). Separators which achieve exact partitioning, i.e., $\alpha = \frac{1}{2}$, are called *strong separators*. Strong edge separators are also known as *bisectors*. For example, planar graphs are $\sqrt{n}$ -edge separable for $\alpha = \frac{2}{3}$ and $\beta = 2\sqrt{2}$, and trees are 1-separable for $\alpha = \frac{2}{3}$ and $\beta = 1$ [81].

The separability property of a graph is important since it can provide some information on the layout area of the graph and a lower bound for routing on the graph. Note that graphs with small separators tend to have small layout areas since there are only small number of edges to connect their two separated subgraphs, and that they require much time for routing since it is difficult to send data from one separated subgraph to the other because of the lack in communicating links between them.

We now introduce several important network topologies, the complete network, the d -dimensional mesh, the binary tree network, the hypercube, the butterfly, the cube-connected cycles and the shuffle-exchange network, and compare them with respect to the properties mentioned above.

1.3.1 The Complete Network

The most general fixed interconnection scheme is the *complete network* in which every processor is directly connected to every other processor (Figure 1.3). Since its diameter is only one, it can perform any permutation in one cycle. However, it is physically unrealistic for several reasons. An arbitrarily large number of communication links can not enter a processor because of physical fan-in limitations, so only very small machines would be constructible. Moreover, since its degree is $p - 1$ and the number of its communication links is $\frac{p(p-1)}{2}$, the space it would occupy and the length of the longest communication link increases very rapidly as p increases. The complete network is interesting as a theoretical model since algorithmic lower bounds for this model are automatically lower bounds for all fixed interconnection networks.

1.3.2 The d -Dimensional Mesh

In a d -dimensional $\sqrt[p]{p} \times \sqrt[p]{p} \times \dots \times \sqrt[p]{p}$ mesh, the p processors may be thought of as logically arranged in a d -dimensional $\sqrt[p]{p} \times \sqrt[p]{p} \times \dots \times \sqrt[p]{p}$ array. The processor at location $(i_{d-1}, i_{d-2}, \dots, i_0)$ of the array is connected to the processors at locations $(i_{d-1}, \dots, i_j \pm 1, \dots, i_0)$, $0 \leq j \leq d - 1$. This network has degree $2d$, diameter $d(\sqrt[p]{p} - 1)$ and a $p^{1-\frac{1}{d}}$ -bisector. It can perform permutations in $\Theta(d\sqrt[p]{p})$ cycles [55,56,78].

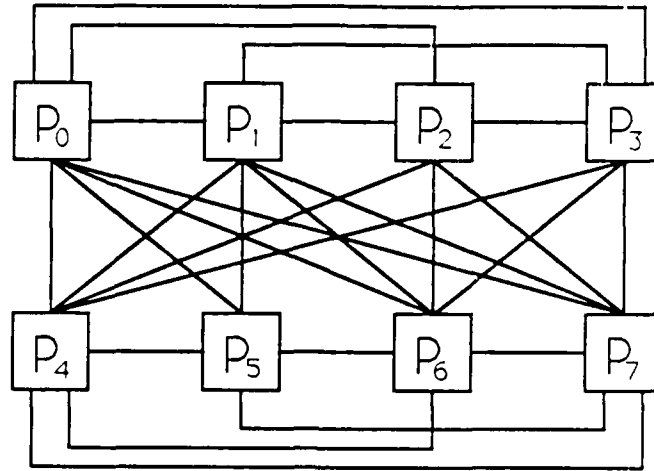


Figure 1.3: Complete network of size eight

One of the most natural interconnection schemes is the 2-dimensional $\sqrt{p} \times \sqrt{p}$ mesh (Figure 1.4). Its physical layout is straightforward in the 2-dimensional space. It has degree four, diameter $2(\sqrt{p} - 1)$ and a $\sqrt{p}$ -bisector. It performs permutations in $\Theta(\sqrt{p})$ cycles. The diameter can be halved by including end-around connections as in the ILLIAC IV [5].

1.3.3 The Binary Tree Network

In a *binary tree network*, the $p = 2^d - 1$ processors are connected into a complete binary tree with depth $d - 1$ (Figure 1.5). Each non-root internal processor P_i , $2 \leq i \leq 2^{d-1} - 1$, is connected to three processors, $P_{L(i)}$, $P_{R(i)}$ and $P_{F(i)}$, where $L(i) = 2i$, $R(i) = 2i + 1$ and $F(i) = \lfloor \frac{i}{2} \rfloor$. The root processor P_1 is connected to P_2 and P_3 as its left and right child respectively. The leaf processors P_i are connected to only their fathers $P_{F(i)}$, $2^{d-1} \leq i \leq 2^d - 1$.

This network has a 1-bisector and a $2 \log_2 \frac{p+1}{2}$ -diameter – the distance from a leaf up to the root and back down to another leaf. It also has a simple layout as shown in the above figure. Unfortunately, tree networks require linear time to perform permutations. For example, assume it is wished to move each item from the root's left subtree to the right subtree, and vice versa. The root is then a bottleneck since it is the only bridge between the two subtrees.

1.3.4 The Hypercube Network

In a *hypercube network*, the $p = 2^d$ processors are connected into a d -dimensional Boolean cube. Let the binary representation of i be $i_{d-1}i_{d-2} \dots i_0$, $0 \leq i \leq p - 1$.

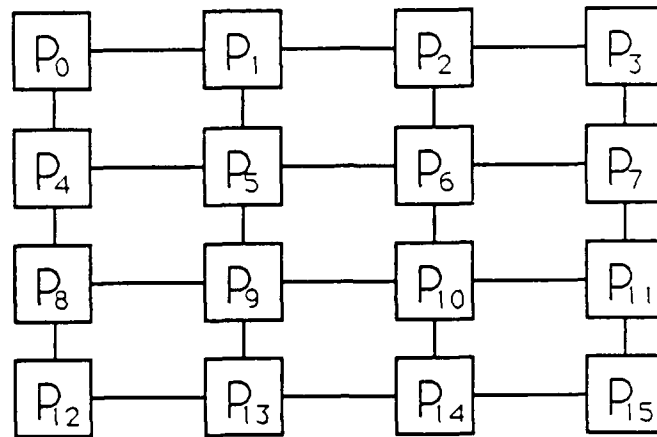


Figure 1.4: 2-dimensional mesh of size 16

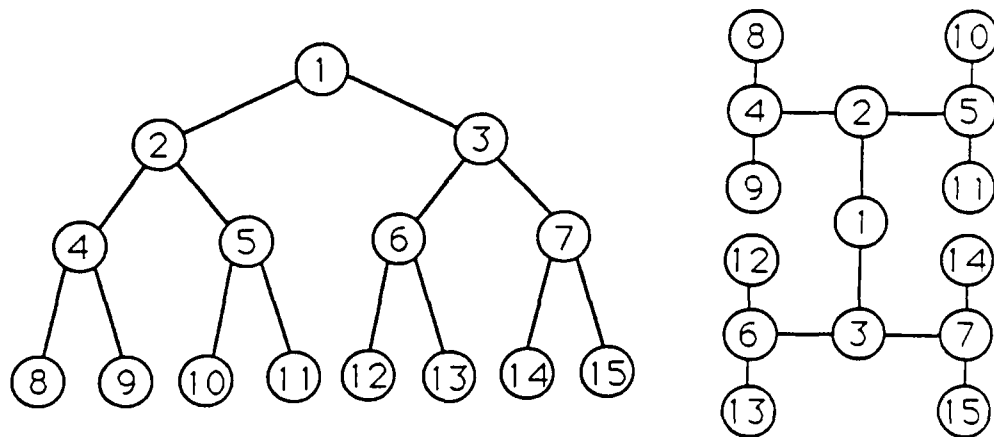
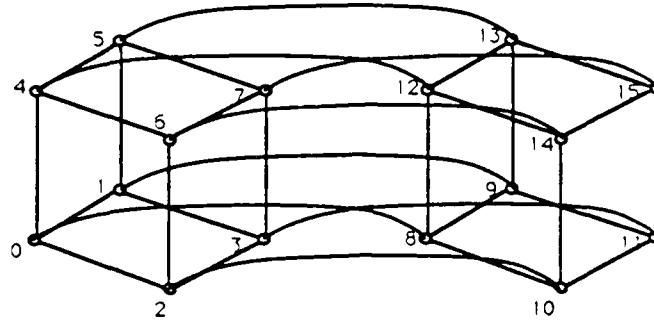
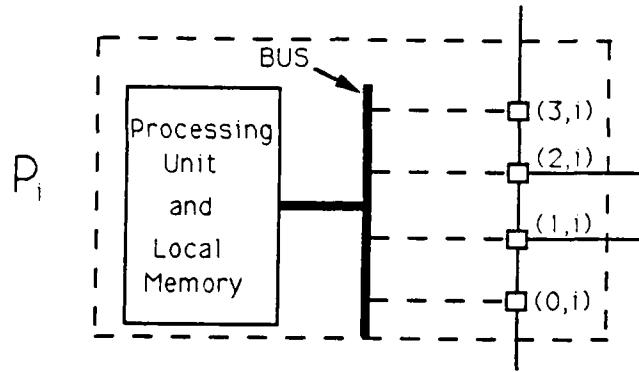


Figure 1.5: Binary tree network of size 15 and its layout



(a)



(b)

Figure 1.6: 4-dimensional hypercube (a) and switches in each processor (b)

Then processor P_i is connected to processors P_j , where $i^j = i_{d-1} \dots \bar{i}_j \dots i_0$ and $\bar{i}_j = 1 - i_j$, $0 \leq j \leq d-1$. The hypercube has a recursive structure: a d -dimensional cube can be extended to a $(d+1)$ -dimensional cube by connecting corresponding processors of two d -dimensional cubes. One has the highest-order address bit 0 and the other has the highest-order address bit 1 (Figure 1.6(a)).

This network has diameter $d = \log p$, for example, the distance between P_0 and P_{p-1} , and a strong $\frac{p}{\sqrt{\log p}}$ -separator. Since its degree is $\log p$ and the total number of its communication links is $d \cdot 2^{d-1}$, its layout area which is $\Theta(p^2)$ would grow more rapidly than similar networks such as the shuffle-exchange and the cube-connected cycles. It can perform an arbitrary permutation in $\Theta(\log p)$ cycles [35,86].

The hypercube architecture has many interesting topological and graph-theoretic properties that make it a very good candidate for parallel processing. Actually several hypercube networks have been available commercially for some time.

The hypercube network that will be used in the rest of this thesis consists of $p = 2^d$ synchronous processors. Two different hypercube models, the pipelined hypercube model and the weak hypercube model, will be used.

In the pipelined hypercube [82], there are d switches, $(0, i), (1, i), \dots, (d - 1, i)$, in each processor P_i , $0 \leq i \leq p - 1$. The switches are connected by a shared bus to the processing unit of the processor. Each switch (j, i) , $0 \leq j \leq d - 1$, is connected by a bidirectional *intra-processor* link to switch $((j + 1) \bmod d, i)$, and by a bidirectional *inter-processor* link to switch (j, i') (Figure 1.6 (b)). A cycle of a switch consists of an *odd* phase and an *even* phase. The odd phase consists of data transfer between switches in the same processor along the intra-processor link. In the even phase, data is transferred between different processors using the inter-processor link.

The switches form a synchronous, pipelined packet-switched network that is used to transfer blocks of data between the processors. A packet consists of a constant number of data elements. Three types of communication traffic, *forward routing*, *reverse routing* and *cube routing*, that arise in all the algorithms in this thesis must be supported by the network.

In forward routing, communication during the odd phase is from switch (j, i) to $((j + 1) \bmod d, i)$, while for reverse routing it is from (j, i) to $((j - 1) \bmod d, i)$. On receiving a packet, switch (j, i) decodes the destination address associated with the packet, and buffers it for transmission on either the intra-processor link or the inter-processor link to (j, i') as appropriate. If the packet is buffered on the intra-processor link, the packet will be transferred to the switch $((j + 1) \bmod d, i)$ or $((j - 1) \bmod d, i)$ in the odd phase of the next cycle. Otherwise, it will be transferred to (j, i') in the even phase of the current cycle. Cube routing is employed to emulate the point-to-point connections of the hypercube. We require at most one switch of a processor to send or receive a packet to or from the processing unit of the processor in the same cycle. Thus a shared bus between the processing unit and the switches in each processor represents an adequate connection.

In the weak hypercube [61], each processor is allowed to send or receive at most one packet and perform a constant number of local computations in a single time step. We assume that the instruction format does not restrict all packets to cross the same dimension in a given time step. Clearly, this model is weaker in communication than the pipelined hypercube model.

1.3.5 The Butterfly Network

The *butterfly network* is an interconnection system most frequently associated with Fast Fourier Transform. In general, it consists of $p = (q + 1)2^q$ processors, organized as $q + 1$ ranks of 2^q processors each (Figure 1.7). Optionally, we shall

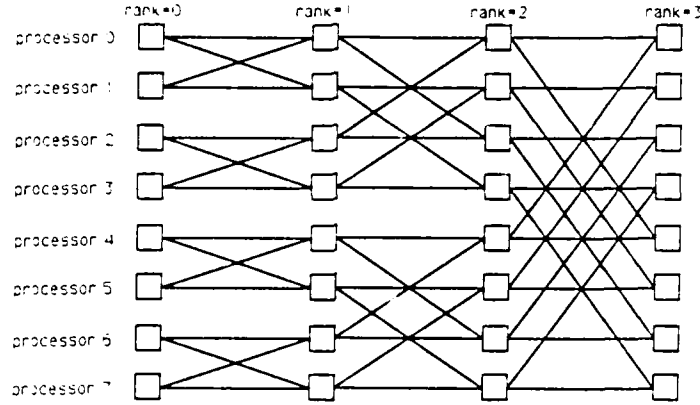


Figure 1.7: Butterfly network for $q = 3$ and $p = 4 \cdot 2^3$

identify the rightmost and the leftmost ranks, so there is no rank q , and the processors on ranks 0 and $q - 1$ are connected directly.

Let us denote the processor i on the rank r by $P_{i,r}$, $0 \leq i < 2^q$, $0 \leq r \leq q$. Then processor $P_{i,r+1}$ is connected to the two processors $P_{i,r}$ and $P_{i^r,r}$, and processor $P_{i^r,r+1}$ is connected to the two processors $P_{i,r}$ and $P_{i,r}$. Recall that $i^r = i_{q-1} \dots i_r \dots i_0$. These four connections form a “butterfly” pattern, from which the name of the network is derived.

The hypercube is actually the butterfly with the rows collapsed. The communication link in the hypercube between processors P_i and P_{i^r} is identified with the communication links in the butterfly between $P_{i,r+1}$ and $P_{i^r,r}$, and between $P_{i^r,r+1}$ and $P_{i,r}$.

1.3.6 The Cube-Connected Cycles

The *cube-connected cycles* is a network of $p = 2^d$ identical processors, where $d = l + 2^l$. When d is arbitrary, l is the smallest integer for which $l + 2^l \geq d$, and the resulting modifications are straightforward. Each processor has a d -bit address m , which in turn is expressed as a pair (i, r) of integers represented with $(d - l)$ and l bits, respectively, such that $i2^l + r = m$ [62]. This network consists of $\frac{p}{2^l}$ cycles of length 2^l and those cycles are connected as a 2^l -dimensional Boolean cube. Let $F(i, r) = (i, (r + 1) \bmod 2^l)$, $B(i, r) = (i, (r - 1) \bmod 2^l)$ and $L(i, r) = (i^r, r)$. Then, each processor $P_{(i,r)}$ of the cube-connected cycles has three neighbors $P_{F(i,r)}$, $P_{B(i,r)}$ and $P_{L(i,r)}$ (Figure 1.8). Processor $P_{(i,r)}$ is connected to processors $P_{F(i,r)}$ and $P_{B(i,r)}$ around the cycle, and to processor $P_{L(i,r)}$ across the cube.

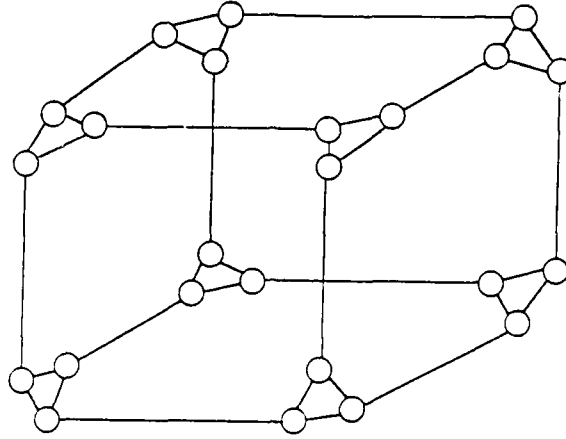


Figure 1.8: Cube-connected cycles for $p = 3 \cdot 2^3$

The p -processor cube-connected cycles can be considered as the p -processor butterfly with processors in rank 0 identified with processors in rank 2^l . The cycles of the cube-connected cycles are exactly the cycles of the butterfly. The only detail necessary to relate the butterfly to the cube-connected cycles is that in the cube-connected cycles, processor $P_{(i,r)}$ is connected to $P_{(i^r,r)}$, while in the butterfly processor $P_{i,r}$ is connected to $P_{i^r,r+1}$. However, by following a pair of links in the cube-connected cycles, we can get to $P_{(i^r,r+1)}$ from $P_{(i,r)}$; we go across the cube to $P_{(i^r,r)}$ and then around the cycle to $P_{(i^r,r+1)}$.

The cube-connected cycles has degree three, diameter $\Theta(\log p)$ and a $\frac{p}{\log p}$ -bisector, and its layout has area $\Theta(\frac{p^2}{\log^2 p})$. It can perform an arbitrary permutation in $\Theta(\log p)$ cycles [62].

1.3.7 The Shuffle-Exchange Network

The *shuffle-exchange network* is based on the *perfect shuffle* and the exchange interconnections [75]. Define $PS(i)$ and $EX(i)$, $0 \leq i < p = 2^d$, as follows:

$$PS(i) = \begin{cases} 2i & \text{if } i < \frac{p}{2}, \\ 2i - p + 1 & \text{otherwise;} \end{cases}$$

$$EX(i) = \begin{cases} i + 1 & \text{if } i \text{ is even,} \\ i - 1 & \text{otherwise.} \end{cases}$$

Then PS^{-1} can be described as

$$PS^{-1}(i) = \begin{cases} \frac{i}{2} & \text{if } i \text{ is even,} \\ \frac{i-1}{2} + \frac{p}{2} & \text{otherwise.} \end{cases}$$

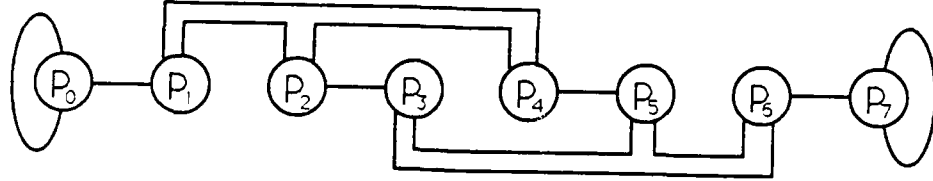


Figure 1.9: Shuffle-exchange network of size eight

If $i_{d-1}i_{d-2}\dots i_0$ denotes the binary representation of i , then $PS(i)$ and $PS^{-1}(i)$ correspond to the left rotation and right rotation of i one position respectively as follows;

$$PS(i_{d-1}i_{d-2}\dots i_0) = i_{d-2}\dots i_0i_{d-1}$$

and

$$PS^{-1}(i_{d-1}i_{d-2}\dots i_0) = i_0i_{d-1}\dots i_1.$$

In the shuffle-exchange network, each P_i has three neighbors, $P_{EX(i)}$, $P_{PS(i)}$ and $P_{PS^{-1}(i)}$ (Figure 1.9). It has diameter approximately $2 \log_2 p$ and a $\frac{p}{\log p}$ -bisector, and its layout has area $\Theta(\frac{p^2}{\log^2 p})$. It can perform an arbitrary permutation in $\Theta(\log p)$ cycles [71].

Table 1.1 shows the the asymptotic formulas for the various quantities associated with the fixed interconnection networks introduced.

1.4 Hypercube Algorithms

In this section, we introduce several fundamental hypercube algorithms that will be used in the rest of the thesis.

1.4.1 Normal Algorithms

There are three classes of algorithms for the weak hypercube: *leveled* algorithms, which use communication links in only one dimension at a time, but in arbitrary order; *normal* algorithms, which are leveled algorithms subject to the additional

	degree	number of links	diameter	separator	cycles (permutations)	layout area
complete network	$p - 1$	$\frac{p^2}{2}$	1	p^2	1	$\Theta(p^4)$
2-D mesh	4	$2p$	$2\sqrt{p}$	$\sqrt{p}$	$\Theta(p)$	$\Theta(p)$
binary tree	3	p	$2 \log p$	1	$\Theta(p)$	$\Theta(p)$
hypercube	$\log p$	$\frac{p \log p}{2}$	$\log p$	$\frac{p}{\sqrt{\log p}}$	$\Theta(\log p)$	$\Theta(p^2)$
cube-co. cycles	3	$1.5p$	$2 \log p$	$\frac{p}{\log p}$	$\Theta(\log p)$	$\Theta(\frac{p^2}{\log^2 p})$
shuffle exchange	3	$1.5p$	$2 \log p$	$\frac{p}{\log p}$	$\Theta(\log p)$	$\Theta(\frac{p^2}{\log^2 p})$

Table 1.1: Comparisons among the fixed interconnection networks

condition that consecutive dimensions are used at consecutive time steps; and *fully normal* algorithms, which are normal algorithms subject to the additional condition that all d dimensions of the hypercube are used in sequence.

Theorem 1.2 [62,70,81] *Normal algorithms can be simulated on the shuffle-exchange, the cube-connected cycles, or the butterfly with only a constant slowdown. $\square$*

Example 1.1: Let $p = 2^d$ elements $\{a_0, a_1, \dots, a_{p-1}\}$ and a binary associative operator $*$ be given. Suppose we have a p -processor weak hypercube such that a_i is stored in processor P_i , $0 \leq i \leq p-1$. Then *prefix sums* computation consists of evaluating the p partial sums $s_j = a_0 * a_1 * \dots * a_j$, $0 \leq j \leq p-1$. There is a fully normal algorithm to solve the problem.

```

 $t_i \leftarrow a_i;$ 
for  $j \leftarrow 0$  to  $d-1$  do in parallel
  if  $i > i \oplus 2^j$  { $\oplus$ : Exclusive or}
    then  $s_i \leftarrow t_{i \oplus 2^j} * s_i$ ;  $t_i \leftarrow t_{i \oplus 2^j} * t_i$ 
  else  $t_i \leftarrow t_i * t_{i \oplus 2^j}$ ;  $\square$ 

```

Example 1.2: Assume that there is an array $A = (a_0, a_1, \dots, a_{p-1})$ such that $a_0 \leq \dots \leq a_{p/2-1}$ and $a_{p/2} \geq \dots \geq a_{p-1}$. Suppose that a_i stored in processor

$P_i, 0 \leq i \leq p-1$. Then there is a fully normal algorithm to merge the array A . The algorithm is referred to as *bitonic merge* algorithm [6,39].

for $i \leftarrow d-1$ to 0 do in parallel
 if $(j < j \oplus 2^i \text{ and } a_j > a_{j \oplus 2^i})$ or $(j > j \oplus 2^i \text{ and } a_j < a_{j \oplus 2^i})$
 then $a_j \leftarrow a_{j \oplus 2^i}$.

Thus, an arbitrary array $A = (a_0, a_1, \dots, a_{p-1})$, a_i stored in processor P_i , can be sorted in $\frac{d(d-1)}{2}$ steps on the weak hypercube by applying the merge algorithm to each subcube of size 2^j , increasing j from 1 to d (*bitonic sort*). Note that the sorting algorithm is normal. $\square$

Since many powerful techniques have been developed for designing efficient parallel algorithms on the PRAM model, it is important to develop an efficient step by step simulation of a PRAM algorithm on the fixed interconnection networks. Using the above sorting algorithm, we can show that any PRAM algorithm can be simulated with $O(\log^2 p)$ delay on the weak hypercube under certain conditions stated in the next theorem.

Theorem 1.3 *Let a PRAM algorithm require t steps and m memory locations on any p -processor CRCW PRAM. Then this algorithm can be implemented to run in time $O(t \cdot \log^2 p)$ on the p -processor weak hypercube whenever $m = O(p)$.*

Proof: There are two CRCW operations that need to be simulated on the hypercube: the concurrent read and the concurrent write. The concurrent read operation can be simulated as follows.

- (1) Sort the read requests according to their destination addresses.
- (2) Choose only one read request for each destination address.
- (3) Distribute the picked requests to their destinations.
- (4) Read the data.
- (5) Return the data to the positions of the requesting packets in step (2).
- (6) Broadcast the data to the requests with the same destination addresses.
- (7) Return the data items to their original processors.

All the above steps can be performed on the hypercube in $O(\log^2 p)$ steps by the sorting algorithm and the prefix sums algorithm. The concurrent write operation can be simulated in a similar way. Among the write requests with the same destination, only one request is chosen according to the assumption of the concurrent writing. $\square$

Since normal algorithms can be simulated on the shuffle-exchange, the cube-connected cycles, or the butterfly with only a constant slowdown, the following corollary follows.

Corollary 1.1 *Let a PRAM algorithm require t steps and m memory locations on any p -processor CRCW PRAM. Then this algorithm can be implemented to run in time $O(t \cdot \log^2 p)$ on the p -processor shuffle-exchange, cube-connected cycles or butterfly whenever $m = O(p)$. $\square$*

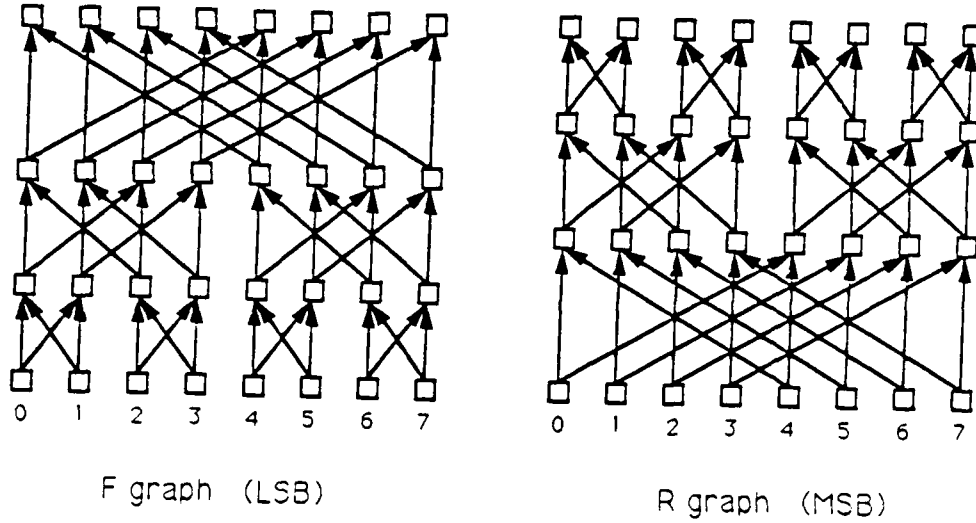


Figure 1.10: The F and R graphs for $p = 2^3$

1.4.2 Butterfly Communication Graphs

A *butterfly communication graph* is a directed graph whose vertices represent switches and whose edges represent unidirectional communication links between the switches. Vertices with no incoming (outgoing) edges will be called *sources* (*sinks*). We define two butterfly communication graphs, F and R , on which the required traffic patterns are proved to be conflict-free. We then show that a conflict-free set of routes in either F and R corresponds to conflict-free routing on the pipelined hypercube.

Both F and R have $p(d+1)$ vertices arranged in $d+1$ levels, with $p = 2^d$ vertices at each level. A vertex is denoted by (l, i) , where l is the *level* number, $0 \leq l \leq d$, and i is the *index* of the vertex within the level, $0 \leq i \leq p-1$. In F , a vertex (l, i) at level l , $0 \leq l \leq d-1$, is connected to the two vertices $(l+1, i)$ and $(l+1, i')$ by edges directed from the former into the latter. In R , the vertex (l, i) is connected to two vertices $(l+1, i)$ and $(l+1, i^{d-l-1})$. F will be referred to as the *forward* network and R will be referred to as the *reverse* network (Figure 1.10).

A switch at level l , $0 \leq l \leq d$, examines a bit of the address associated with a packet, and passes it at the next cycle to a switch at level $l+1$. We describe two routing operations that the switches support, namely *least significant bit (LSB)*

routing and *most significant bit (MSB)* routing. The switches in F employ LSB routing while those in R employ MSB routing.

In LSB routing, vertex (l, i) of F , $0 \leq l \leq d-1$, routes a packet to either vertex $(l+1, i)$ or vertex $(l+1, i')$ depending on whether the l -th bit of the address field A_F of the packet matches the l -th bit of i or not, respectively. In MSB routing, vertex (l, i) of R , $0 \leq l \leq d-1$, routes a packet to either vertex $(l+1, i)$ or to vertex $(l+1, i^{d-l-1})$ depending on whether the $(d-l-1)$ -th bit of the address field A_R of the packet matches the $(d-l-1)$ -th bit of i or not, respectively.

The switches in R also support a variant of MSB routing referred to as *MSB routing with copy*. This is used to implement a *broadcast* facility, in which a data packet can be sent simultaneously from a vertex $(0, i)$ to all the consecutively indexed destination vertices, (d, a_i) , $(d, a_i + 1)$, $\dots$, $(d, b_i - 1)$, (d, b_i) . The address field A_R now consists of the pair in integers (a_i, b_i) , $a_i \leq b_i$, which define the limits within which the packet must be sent. Each switch vertex (l, j) , $0 \leq l \leq d-1$, performs the following actions on receiving a packet of this form. If the $(d-l-1)$ -th bits of a_i and b_i are the same, the vertex implements the usual MSB routing to route the packet to the vertex indicated by the address a_i . If the two bits are different, then the packet is forwarded to both the vertices $(l+1, j)$ and $(l+1, j^{d-l-1})$. However, the addresses a_i and b_i that are forwarded to the two vertices are updated as follows. The copy forwarded to the vertex with the smaller index will have b_i set to $2^d - 1$, and that forwarded to the vertex with the larger index will have a_i set to zero.

The *route* in F (R) from vertex $(0, i)$ to vertex (d, j) is the ordered sequence of vertices in F (R), $((0, i), (1, i_1), \dots, (d, i_d = j))$, that a packet with address $A_F = j$ ($A_R = j$) passes through. The sequence of edges between vertices in the route is the *path* of the route. A route in F is referred to as *forward route*, while a route in R is referred to as *reverse route*. Two routes are said to be *conflict-free* if they are vertex disjoint. A set of routes are conflict-free if they are pairwise vertex disjoint.

We now relate F and R to the pipelined hypercube, and show how conflict-free routes in F or R imply link-disjoint routes in the pipelined hypercube.

In the following, let H_F (H_R) refer to the graph obtained from F (R) by replacing the directed edge $\langle (l, u), (l+1, u^l) \rangle$ ($\langle (l, u), (l+1, u^{d-l-1}) \rangle$) with the directed edge $\langle (l+1, u), (l+1, u^l) \rangle$ ($\langle (l+1, u), (l+1, u^{d-l-1}) \rangle$) (Figure 1.11). Notice that H_F and H_R maps directly onto the switches and links of the hypercube used for forward routing and reverse routing, respectively.

A route in H_F is obtained from a route in F by replacing every edge $\langle (l, u), (l+1, u^l) \rangle$ by the two directed edges $\langle (l, u), (l+1, u) \rangle$ and $\langle (l+1, u), (l+1, u^l) \rangle$, $0 \leq l \leq d-1$. Similarly, A route in H_R is obtained from a

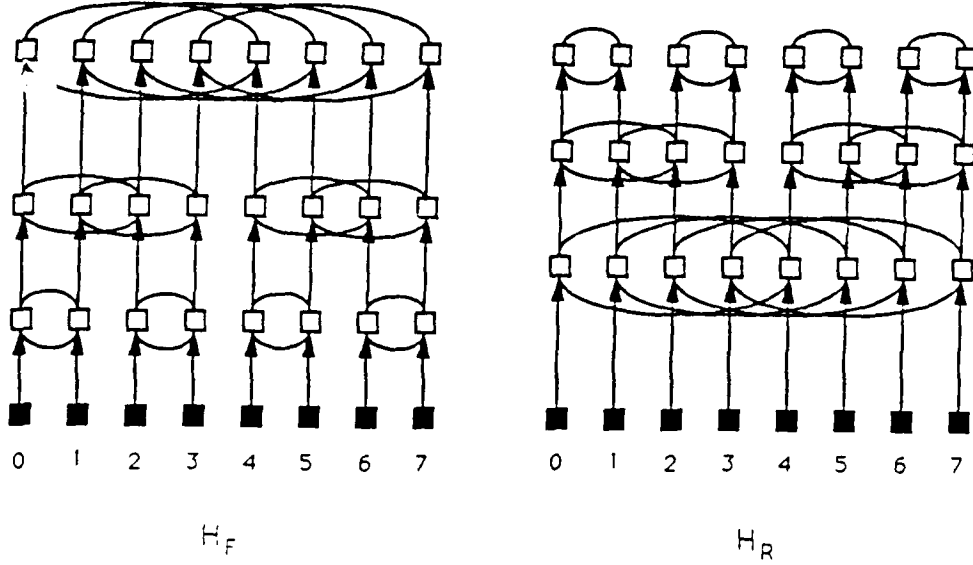


Figure 1.11: The H_F and H_R corresponding to F and R of Figure 1.10

route in R by replacing every edge $\langle (l, u), (l+1, u^{d-l-1}) \rangle$ by the two directed edges $\langle (l, u), (l+1, u) \rangle$ and $\langle (l+1, u), (l+1, u^{d-l-1}) \rangle$, $0 \leq l \leq d-1$.

Theorem 1.4 [82] *Let R_1 and R_2 be the paths of two vertex disjoint routes in F (R) and R'_1 and R'_2 be the corresponding paths in H_F (H_R). Then R'_1 and R'_2 are edge disjoint.*

Proof: Assume by way of contradiction that an edge e that is common to R'_1 and R'_2 exists. If $e = \langle (l, u), (l+1, u) \rangle$, then vertex (l, u) is common to both R_1 and R_2 . If $e = \langle (l, u), (l, u') \rangle$, where $u' = u^{l-1}$ or u^{d-l} according to whether R_1 and R_2 are from F or R , then vertex $(l-1, u)$ is common to both R_1 and R_2 . $\square$

Corollary 1.2 *Let S_1 be a set of conflict-free routes in F or R and S_2 be corresponding routes in the hypercube. Then, routes in S_2 are pair-wise link disjoint in the hypercube.* $\square$

1.4.3 Conflict-Free Routing

We now describe several routing patterns that arise throughout this thesis, and show that they are conflict-free in F or R . As a consequence of Theorem 1.3

and Corollary 1.2, these routings can be performed without link conflict in the hypercube network using *LSB* or *MSB* routing.

Lemma 1.1 [82] *Let (l, u) be a node on the route from $(0, i)$ to (d, j) in $F(R)$. Then the binary representation of u is $i_{d-1} \dots i_l j_{l-1} \dots j_0$ ($j_{d-1} \dots j_{d-l} i_{d-l-1} \dots i_0$).*

Proof: Direct consequence of *LSB* (*MSB*) routing. $\square$

Lemma 1.2 [82] *Let $((0, s_i), (d, t_i))$, $0 \leq i \leq r-1$, be a collection of r pairs such that, $0 \leq s_0 < s_1 < \dots < s_{r-1} \leq 2^d - 1$, $0 \leq t_0 < t_1 < \dots < t_{r-1} \leq 2^d - 1$, and $s_{i+1} - s_i \geq t_{i+1} - t_i$, for all i , $0 \leq i \leq r-2$. Then the set of routes in F from vertex $(0, s_i)$ to the vertex (d, t_i) is conflict-free.*

Proof: Let i and j be such that $0 \leq j < i \leq r-1$. Let $u = s_i$, $v = s_j$, $x = t_i$ and $y = t_j$. Assume by way of contradiction that (l, w) is a vertex that is common to the two routes (u, x) and (v, y) . From Lemma 1.1, $w = u_{d-1} \dots u_l x_{l-1} \dots x_0$ $v_{d-1} \dots v_l y_{l-1} \dots y_0$. Thus, $u - v < 2^l$ and $x - y \geq 2^l$, which contradicts the fact that $u - v \geq x - y$. Since i and j were arbitrary, the set of routes is conflict-free. $\square$

The special case of this lemma, where $t_i = i$, is known as *concentrate* routing. A similar lemma holds for the routes in R .

Lemma 1.3 [82] *Let $((0, s_i), (d, t_i))$, $0 \leq i \leq r-1$, be a collection of r pairs such that, $0 \leq s_0 < s_1 < \dots < s_{r-1} \leq 2^d - 1$, $0 \leq t_0 < t_1 < \dots < t_{r-1} \leq 2^d - 1$, and $s_{i+1} - s_i \leq t_{i+1} - t_i$, for all i , $0 \leq i \leq r-2$. Then the set of routes in R from vertex $(0, s_i)$ to the vertex (d, t_i) is conflict-free.* $\square$

The special case of this lemma, where $s_i = i$, is known as *spread* routing.

Broadcast routing is defined as follows. Let $\{(0, i) | 0 \leq i \leq r-1\}$ be a set of sources in R . Associated with each source $(0, i)$, is a pair of integers, a_i and b_i such that $a_i \leq b_i$. Let $a_{i+1} > b_i$, for all i , $0 \leq i \leq r-2$, and $b_{r-1} \leq p-1$. Then Broadcast routing is to route data from $(0, i)$ to all (d, u) , $a_i \leq u \leq b_i$. Broadcast routing can be performed using *MSB* with copy in R .

Lemma 1.4 [82] *Let $(0, i)$ and $(0, j)$ be two sources involved in a broadcast routing. Then the routes from $(0, i)$ to (d, u) and from $(0, j)$ to (d, v) , for any u and v , $a_i \leq u \leq b_i$ and $a_j \leq v \leq b_j$ are conflict-free.* $\square$

Corollary 1.3 *Broadcast routing is conflict-free.* $\square$

Note that all the above conflict-free routings can be performed by using fully normal algorithms. We later prove that this kind of routings can be performed optimally on the pipelined hypercube since their paths are conflict-free and so all the links of the hypercube can be used simultaneously.

Chapter 2

Load Balancing, Sorting and Routing

2.1 Introduction

Consider any of our networks in the case when the input size n is larger than the number of processors p . Compared with the PRAM model, there are two main drawbacks: (1) no two processors can access the same memory module simultaneously and hence memory conflicts should be avoided, and (2) there is an $O(\log p)$ cost for two arbitrary processors to communicate. An efficient algorithm should maintain a balance between local computation and communication, and should arrange the data dynamically in such a way that memory conflicts are avoided. Such algorithms have been developed for several basic data broadcasting and communication problems [31,32,34,35,37,65,66], numerical computing problems [31,32,37], and sorting [18,61,82].

We address in this chapter several problems related to load balancing, sorting and routing on the hypercube, the shuffle-exchange, the cube-connected cycles and the butterfly. These problems are important on their own and are fundamental to fast implementation of parallel algorithms on these networks. Our contribution is two-fold. First, we provide new algorithms to handle these problems. In most of the cases, our algorithms are efficient under certain conditions. For example, our algorithm for routing n packets on the p -processor hypercube is efficient whenever $n = \Omega(p^{1+\epsilon})$, for some positive constant ϵ . Second, we shed some insight into the relationship between these different networks. For example, we establish that load balancing can provably be solved faster on the weak hypercube than on the shuffle-exchange, the cube-connected cycles or the butterfly.

The rest of the chapter is organized as follows. Some results on basic communication schemes needed for the rest of this thesis are given in the next section. Load balancing and sorting are considered in sections 2.3 and 2.4 respectively.

while the algorithms for the general packet routing problem are presented in section 2.5. The last section is devoted to the relationship between our networks and the CRCW PRAM.

2.2 Basic Communication Schemes

Recall that a *fully-normal* algorithm on the hypercube with $p = 2^d$ processors consists of d stages such that the stage i involves the communication links in the dimension i or in the dimension $(d - i - 1)$ of the hypercube, $0 \leq i \leq d - 1$, and proceeds from the least significant bit to the most significant bit (LSB) or vice-versa (MSB), respectively (section 1.3). It turns out that many computational and routing problems, such as Fast Fourier Transform, prefix sums, odd-even and bitonic merge, matrix transpose, conflict-free routing, and any fixed permutation, can be solved optimally by such an algorithm. The existence of such simple optimal algorithms has stimulated initial interest in the hypercube model. In this section, we will review several important cases of the routing problem on the hypercube that can be solved optimally by a fully normal algorithm and develop the necessary background needed for the rest of the thesis. We are interested in the case when the number n of data items could be much larger than the number p of processors.

A simple routing problem consists of a set of n packets, each of which has a source, a destination and a data item to be moved from the source processor to the destination processor. Let $\langle i, t_i, x_i \rangle$ denote an arbitrary packet, where i is the source, t_i is the destination, and x_i is a data item. Suppose we know how to solve a special instance of this routing problem optimally on an n -processor hypercube. We are interested in mapping the algorithm into a p -processor hypercube. We define the corresponding routing problem on the p -processor hypercube as follows. Let $i = (\frac{n}{p}q_i + r$, where $0 \leq r \leq \frac{n}{p} - 1$. Similarly define q_{t_i} . Then replace packet $\langle i, t_i, x_i \rangle$ by $\langle q_i, q_{t_i}, x_i \rangle$. Each processor is now the source of $\frac{n}{p}$ packets. The following simple observation will have important implications.

Lemma 2.1 *Suppose that a simple routing problem with n packets can be solved on an n -processor hypercube by using a fully-normal algorithm. Then the corresponding problem can be solved in time $O(\frac{n}{p} + \log p)$ on a p -processor pipelined hypercube, where $p \leq n$.*

Proof: Let $n = 2^{d_1}$ and $p = 2^{d_2}$. Without loss of generality, assume that the given routing problem can be solved by using the LSB routing on the n -processor hypercube H . We will emulate this strategy on the p -processor hypercube H' . The first $d_1 - d_2$ stages of the algorithm involve data movements within the local memories of H' . The $(d_1 - d_2 + 1)$ -th stage involve moving possibly n packets

along the $(d_1 - d_2 + 1)$ -th dimension of H . H' will pipeline these requests starting with the first dimension. Processor P_i sends its initial packet to switch $(0, i)$, $0 \leq i \leq p - 1$, in the odd phase of the $(d_1 - d_2 + 1)$ -th stage. On receiving the packet, switch $(0, i)$ decodes the destination address associated with the packet, and buffers it for transmission on either the intra-processor link or the inter-processor link to switch $(0, i^0)$. If the packet in switch $(0, i)$ is buffered on the inter-processor link, it will be transferred to switch $(0, i^0)$ and buffered on the intra-processor link to switch $(1, i^0)$ in the even phase. In the odd phase of the $(d_1 - d_2 + 2)$ -th stage, the initial packets will be transferred to switches $(1, i)$, $0 \leq i \leq d_2 - 1$, while the next p packets will be sent to switches $(0, i)$, $0 \leq i \leq d_2 - 1$. At the d_1 -th stage of H , all the links in H' are busy handling the pipelined packets. Since we are allowing pipelining, the routing defined on H' is legal. Therefore the lemma follows. $\square$

The assumption of the pipelined communication indicated in the above lemma is crucial as we will show now. We will describe a routing problem which can be solved optimally using a fully-normal algorithm, and yet cannot be handled within the time bound stated in the above lemma on the weak hypercube.

Let $p = 2^{2d}$, for some positive integer d . Let $E(i) = (i + 0101 \dots 01_2) \bmod p$, $0 \leq i \leq p - 1$. For example, if $p = 2^2$, then $E(00) = 01$, $E(01) = 10$, $E(10) = 11$, and $E(11) = 00$.

Remark 2.1 *The Hamming distance between i and $E(i)$ is no less than d , for any i , $0 \leq i \leq p - 1$.*

Proof: The claim is obvious if $d = 1$. Assume that $d > 1$. One can easily check that i and $E(i)$ differ in at least one bit position of the two most significant bits. The claim follows by induction. $\square$

Lemma 2.2 *Consider the routing problem on a p -processor hypercube, where processor P_i has to send $\frac{n}{p}$ data items to processor $P_{E(i)}$, for $0 \leq i \leq (1010 \dots 10_2)$. Then this problem can be solved in time $O(\frac{n}{p} + \log p)$ on the pipelined hypercube by using a fully-normal algorithm. However, on the weak hypercube, it requires $\Omega(\frac{n \log p}{p})$ time.*

Proof: The paths that send items from P_i to $P_{E(i)}$, $0 \leq i \leq (1010 \dots 10_2)$, are conflict-free by Lemma 1.2, and the problem can be solved by a fully-normal algorithm. Thus, by Lemma 2.1, it can be solved in time $O(\frac{n}{p} + \log p)$ on the pipelined hypercube. However, the total number of data movements is $\Omega(n \log p)$ since, by the above remark, each item must pass $\Omega(\log p)$ communication links to get to its proper destination. Thus, $\Omega(\frac{n \log p}{p})$ steps are necessary on the weak hypercube since only one link in each processor can be used at each time step. $\square$

A routing problem of n packets on a p -processor hypercube will be viewed as a routing problem on an n -processor hypercube. A fully-normal algorithm will be found, and then using Lemma 2.1, a solution on the p -processor hypercube will be obtained.

The four important special routing problems, concentrate, broadcast, spread and collect, can be restated as follows. Assume that each processor P_{s_i} has a block B_i , $0 \leq i \leq r-1$, where $s_0 < s_1 < \dots < s_{r-1}$ and $|B_i| = t$.

The *concentrate* routing consists of sending the block B_i in P_{s_i} to P_{t_i} , $0 \leq i \leq r-1$. By Lemma 1.2, the concentrate routing can be performed by using a fully-normal algorithm on a n -processor hypercube. Therefore by Lemma 2.1, the concentrate routing can be solved in $O(t + \log p)$ time on the p -processor pipelined hypercube.

The *broadcast* routing can be redefined as follows. Each processor P_{a_i} , $0 \leq i \leq r-1$, has to broadcast its block to all processors P_{b_j} , for $a_i \leq j \leq b_i$, where $a_i \leq b_i < a_{i+1} \leq b_{i+1}$ and $b_{r-1} \leq p-1$. Again using Lemma 1.4 and Lemma 2.1, we conclude that the broadcast routing can be performed in time $O(t + \log p)$ on the pipelined hypercube.

The *spread* routing can be similarly redefined as follows. Each processor P_{a_i} , $0 \leq i \leq r-1$, has to spread its block among processors P_{b_j} , for $a_i \leq j \leq b_i$, with P_{b_j} receiving t_j data items after spreading, where $\sum_{k=a_i}^{b_i} t_k = t$, $a_i \leq b_i < a_{i+1} \leq b_{i+1}$ and $b_{r-1} \leq p-1$. This is similar to the broadcast routing and can be solved in $O(t + \log p)$ time on the pipelined hypercube.

The *collect* routing is the inverse of the spread routing. Each processor P_{b_i} , $0 \leq i \leq r-1$, has to collect all the data items from the processors P_{a_j} , for $a_i \leq j \leq b_i$, with P_{a_j} having t_j data items before collecting, where $\sum_{k=a_i}^{b_i} t_k = t$, $a_i \leq b_i < a_{i+1} \leq b_{i+1}$ and $b_{r-1} \leq p-1$. This can be solved in $O(t + \log p)$ time on the pipelined hypercube.

Finally, a routing problem that can be also solved optimally on the pipelined hypercube can be defined by the set $\langle s_i, t_i \rangle$, $0 \leq i \leq r-1$, where the block B_i in P_{s_i} has to be moved to processor P_{t_i} , and where $\{s_i\}$ and $\{t_i\}$ are strictly increasing sequences. Clearly, this routing can be solved by a combination of concentrate and broadcast. Notice that the routing problem introduced in Lemma 2.2 is of this type.

The class of *block permutation* can be defined as follows. Let π be a permutation of $\{0, 1, \dots, p-1\}$. The goal is to move block B_i of processor P_i to processor $P_{\pi(i)}$, $0 \leq i < p$. We will provide a solution to this problem based on *permutation networks*. We will briefly review some of the basic facts needed.

A Benes permutation network of Figure 2.1(a) can be used to realize any permutation on the input. For any given permutation π , the switches of the Benes permutation network realizing π can be set in $O(p \log p)$ sequential time

[86], or in $O(\log^4 p)$ time on a p -processor hypercube, shuffle-exchange, cube-connected cycles or the butterfly [50,57,71].

Any Benes permutation network can be emulated on the weak hypercube, the shuffle-exchange and the cube-connected cycles. However, emulating the permutation network on the pipelined hypercube is not appropriate, since it would use communication links in different dimensions in each stage and this makes pipelining impossible. So we need another permutation network which can be naturally related to the hypercube.

A *butterfly* permutation network, defined recursively in Figure 2.1(b), can be also used to realize any permutation on the input. For any given permutation π , the switches of the network realizing π can be similarly set in $O(p \log p)$ sequential time, or in $O(\log^4 p)$ time on a p -processor hypercube, shuffle-exchange, cube-connected cycles or butterfly. There is an obvious connection between the butterfly permutation network and the hypercube. Moreover pipelining data on the butterfly permutation network can be emulated efficiently on the pipelined hypercube since links used at any stage correspond to communication links in only one dimension.

Lemma 2.3 *An arbitrary block permutation on n elements can be performed in time $O(\frac{n}{p} + \log^4 p)$ on the pipelined hypercube.*

Proof: We can find the paths needed for the given permutation π in time $O(\log^4 p)$. Then an MSB routing followed by an LSB routing that will fully pipeline the elements of all the blocks will be used. Notice that no conflicts will arise because communication links in different dimensions correspond to different stages of the butterfly algorithm. $\square$

Using the above facts, we will show the following result which will be used heavily in obtaining the upper bounds on the pipelined hypercube model.

Theorem 2.1 *Given a p -processor hypercube such that each processor P_i holds a block of data B_i of size t , $0 \leq i \leq p-1$. Let $\alpha : \{0, 1, \dots, p-1\} \rightarrow \{0, 1, \dots, p-1\}$ be a partial function. Suppose it is desired to move block $B_{\alpha(i)}$ to processor P_i , whenever $\alpha(i)$ is defined. Then this can be done in $O(t + \log^4 p)$ time on the pipelined hypercube model.*

Proof: Each processor P_i creates a record $\langle i, \alpha(i) \rangle$, if $\alpha(i)$ is defined, and a record $\langle i, \infty \rangle$ otherwise. All the p records are sorted by their second components. Each set of records with the same second component will be in consecutive processors after sorting. For each such set, we mark a record residing in the lowest indexed processor as the representative record of the set. Note that the representative records construct a one-to-one partial function. Assume that processors $P_{j_1}, P_{j_2}, \dots, P_{j_k}$ have the representative records $\langle i_1, \alpha(i_1) \rangle$

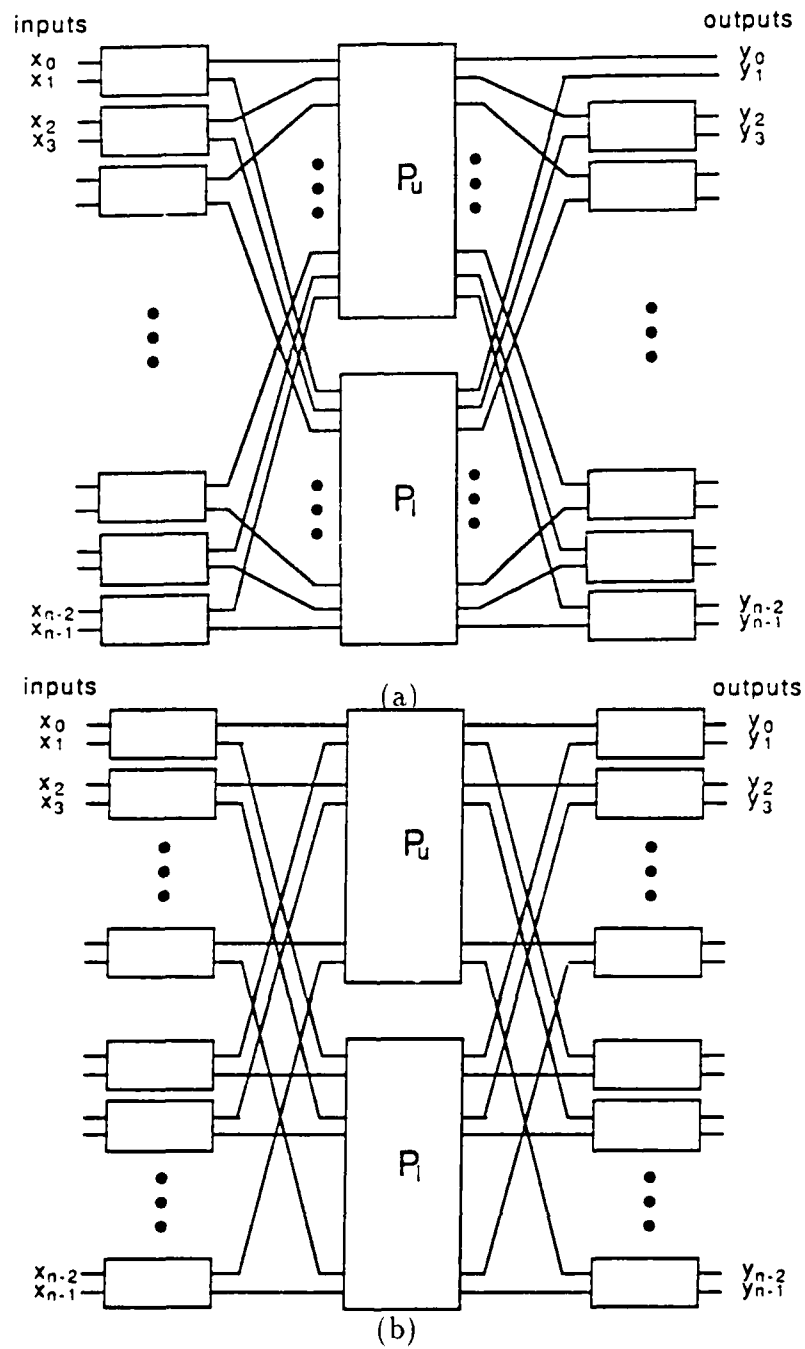


Figure 2.1: A Benes permutation network (a) and a butterfly permutation network (b)

, $\langle i_2, \alpha(i_2) \rangle, \dots, \langle i_k, \alpha(i_k) \rangle$ respectively. Then we construct a permutation π such that $\pi(\alpha(i_l)) = j_l$, $1 \leq l \leq k$, and send $B_{\alpha(i_l)}$ to P_{j_l} by the block permutation operation. We now broadcast $B_{\alpha(i_l)}$ to the consecutive processors with records $\langle *, \alpha(i_l) \rangle$. Note that any processor with the record $\langle i, \alpha(i) \rangle$ has the block $B_{\alpha(i)}$, and that sending $B_{\alpha(i)}$ to P_i can be done similarly. $\square$

We now consider the case of *fixed* permutation, for example, the transpose of a matrix, the perfect shuffle, etc. In this case, the routing paths can be predetermined, and in particular, we have the following lemma.

Lemma 2.4 *Given m fixed permutations on p elements, these permutations can be realized in $O(m + \log p)$ time on the pipelined hypercube. $\square$*

Using Lemma 2.1 and Lemma 2.4, many of the routing problems considered in [31,32,37] can be solved optimally. As a matter of fact, we have developed much simpler algorithms based on the above method than those reported in [31,32,37]. We will illustrate this with an example.

The *all-to-all personalized* communication can be defined as follows [32]. Processor P_i has p blocks $B_{i,0}, B_{i,1}, \dots, B_{i,p-1}$ each of the same size t , and $B_{i,j}$ is supposed to be moved to P_j , $0 \leq i, j \leq p-1$.

Lemma 2.5 *The all-to-all personalized communication problem can be solved in time $O(tp + \log p)$ on the pipelined hypercube.*

Proof: The procedure can be divided into $2p-1$ steps. In the i th step, $0 \leq i \leq p-1$, P_j sends block $B_{j,i+j}$ to P_{i+j} , $0 \leq j < p-i$. For $p \leq i \leq 2p-2$, processor P_j sends $B_{j,j-(2p-1-i)}$ to $P_{j-(2p-1-i)}$, $2p-1-i \leq j < p$. The various steps can be fully pipelined, and the proof of the lemma follows. $\square$

Finally, we introduce the following permutation problem which will be used in the next section. Consider a square matrix A of size $n \times n$ stored in a hypercube of dimension $2d$, $n \geq 2^d$. Generalization to non-square matrices is straightforward. There are several ways of distributing the matrix elements among the different processors of the hypercube. We mention here two schemes of interest. In *consecutive* storage, A is decomposed into subarrays of equal sizes and each subarray is stored in a processor. This means that all elements $(i, j) \in \{0, 1, \dots, n-1\} \times \{0, 1, \dots, n-1\}$ of the $n \times n$ array A that satisfy the relations $r = \lfloor \frac{i}{\lceil \frac{n}{2^d} \rceil} \rfloor$, $s = \lfloor \frac{j}{\lceil \frac{n}{2^d} \rceil} \rfloor$ are identified with element $(r, s) \in \{0, 1, \dots, 2^d-1\} \times \{0, 1, \dots, 2^d-1\}$ of $2^d \times 2^d$ array A' which can be embedded in a $2d$ -dimensional hypercube. In *cyclic* storage, all elements (i, j) of A that satisfy the relations $r = i \bmod 2^d$, $s = j \bmod 2^d$ are identified with element (r, s) of the array A' . The consecutive and cyclic storage schemes are illustrated in Figure 2.2. Using the observations above, it is clear that the conversion between consecutive storage to cyclic storage (and vice versa) can be performed in time $O(\frac{n^2}{p} + \log p)$ on the pipelined hypercube.

00	01	02	03	00	01	02	03	00	01	02	03	00	01	02	03
10	11	12	13	10	11	12	13	10	11	12	13	10	11	12	13
20	21	22	23	20	21	22	23	20	21	22	23	20	21	22	23
30	31	32	33	30	31	32	33	30	31	32	33	30	31	32	33
00	01	02	03	00	01	02	03	00	01	02	03	00	01	02	03
10	11	12	13	10	11	12	13	10	11	12	13	10	11	12	13
20	21	22	23	20	21	22	23	20	21	22	23	20	21	22	23
30	31	32	33	30	31	32	33	30	31	32	33	30	31	32	33
00	01	02	03	00	01	02	03	00	01	02	03	00	01	02	03
10	11	12	13	10	11	12	13	10	11	12	13	10	11	12	13
20	21	22	23	20	21	22	23	20	21	22	23	20	21	22	23
30	31	32	33	30	31	32	33	30	31	32	33	30	31	32	33
00	01	02	03	00	01	02	03	00	01	02	03	00	01	02	03
10	11	12	13	10	11	12	13	10	11	12	13	10	11	12	13
20	21	22	23	20	21	22	23	20	21	22	23	20	21	22	23
30	31	32	33	30	31	32	33	30	31	32	33	30	31	32	33

00	00	00	00	01	01	01	01	02	02	02	02	03	03	03	03
00	00	00	00	01	01	01	01	02	02	02	02	03	03	03	03
00	00	00	00	01	01	01	01	02	02	02	02	03	03	03	03
00	00	00	00	01	01	01	01	02	02	02	02	03	03	03	03
10	10	10	10	11	11	11	11	12	12	12	12	13	13	13	13
10	10	10	10	11	11	11	11	12	12	12	12	13	13	13	13
10	10	10	10	11	11	11	11	12	12	12	12	13	13	13	13
10	10	10	10	11	11	11	11	12	12	12	12	13	13	13	13
20	20	20	20	21	21	21	21	22	22	22	22	23	23	23	23
20	20	20	20	21	21	21	21	22	22	22	22	23	23	23	23
20	20	20	20	21	21	21	21	22	22	22	22	23	23	23	23
20	20	20	20	21	21	21	21	22	22	22	22	23	23	23	23
30	30	30	30	31	31	31	31	32	32	32	32	33	33	33	33
30	30	30	30	31	31	31	31	32	32	32	32	33	33	33	33
30	30	30	30	31	31	31	31	32	32	32	32	33	33	33	33
30	30	30	30	31	31	31	31	32	32	32	32	33	33	33	33

Figure 2.2: Consecutive and cyclic storage of a matrix

2.3 Load Balancing

Balancing load among processors is very important since poor balance of load generally causes poor processor utilization. The load balancing problem is a fundamental problem in the sense that the fast solutions of basic problems such as sorting, selection, list ranking, graph problems, and routing require fast load balancing [34,35,58,61,64]. In this section, some lower bounds and tight upper bounds for load balancing on the hypercube, the shuffle-exchange, the cube-connected cycles and the butterfly are shown.

The load balancing problem is defined as follows. Let n items be distributed over the p processors of a network, with no more than M items assigned to any single processor, $\lceil n/p \rceil \leq M \leq n$. The problem is to redistribute the items so that the number of items in any two processors may differ by at most one. It is irrelevant where a data item is routed to. This problem can also be solved in $O(M + \log p \cdot \min(\log \frac{n}{p}, \log \log p))$ time on a bounded-degree network based on expander graphs [58], and in $O(M\sqrt{\log p} + \log^2 p)$ time on the weak hypercube [61]. We start by addressing the case of the pipelined hypercube.

Assume that processors $P_0, P_1, \dots, P_{p-1}$ of the pipelined hypercube have $n_0, n_1, \dots, n_{p-1}$ data items respectively, and that $n_i \leq M, 0 \leq i \leq p-1$. Without loss of generality, we can assume that $\sum_p n_i = a$ is an integer. The basic idea of our algorithm is to make each processor P_i decide where to move its data items based on $\sum_{j=0}^{i-1} n_j$ and on $\sum_{j=0}^i n_j$ with the goal of balancing as many processors as possible starting from the lowest indexed processor. In other words, each processor P_i computes l_i and r_i such that $l_0 \leq r_0 \leq l_1 \leq r_1 \leq \dots \leq l_{p-1} \leq r_{p-1}$, and sends its data elements to processors $P_{l_i}, P_{l_i+1}, \dots, P_{r_i}$. More precisely, define l_i and r_i to be integers such that $l_i \cdot a \leq \sum_{j=0}^{i-1} n_j < (l_i + 1)a$ and $r_i \cdot a < \sum_{j=0}^i n_j \leq (r_i + 1)a$, respectively. Notice that l_i and r_i can be 0 and that $l_i \leq r_i$. Then P_i distributes its data items over $P_{l_i}, P_{l_i+1}, \dots, P_{r_i}$, if $n_i > 0$. If $l_i < r_i$, then P_{l_i} and P_{r_i} will receive $(l_i + 1)a - \sum_{j=0}^{i-1} n_j$ and $\sum_{j=0}^i n_j - r_i \cdot a$ data items from P_i , respectively. If $r_i > l_i + 1$, $P_{l_i+1}, \dots, P_{r_i-1}$ will each receive a data items. P_i will send its $n_i > 0$ data items to P_{l_i} in the case when $l_i = r_i$.

procedure BALANCE:

- [B1] For each $i, 0 \leq i \leq p-1$, compute a, l_i, r_i and the destination address of each data item.
- [B2] Let $P_{i_0}, P_{i_1}, \dots, P_{i_k}$ be all the processors such that $l_{i_j} < r_{i_j}, 0 \leq j \leq k$. Then P_{i_j} distributes the appropriate data items over $P_{l_{i_j}}, \dots, P_{r_{i_j}-1}$. This step will be executed in two substeps. In the first substep, P_{i_j} sends the appropriate data items to $P_{l_{i_j}}$ by using the concentrate operation. In the second substep, $P_{l_{i_j}}$ distributes the received elements to $P_{l_{i_j}}, \dots, P_{r_{i_j}-1}$ by using the spread operation.

[B3] After step [B2], each processor can only send its data items to a single processor. But each processor can receive data items from more than one processor. Assume that $P_{i_0}, P_{i_1}, \dots, P_{i_k}$ are all the processors that will send their data items to P_i and that P_{i_j} has to send m_{i_j} data items, $0 \leq j \leq k$. Clearly, $\sum_{j=0}^k m_{i_j} \leq a \leq M$. Also assume that P_i is the x_i -th processor among all the processors that will receive data items. Then in the first substep, $P_{i_0}, P_{i_1}, \dots, P_{i_k}$ send their appropriate data items to processor P_{x_i} one at a time by using the collect routing. In the second substep, P_{x_i} will send the collected data items to P_i by using the broadcast operation.

The step by step implementation of **BALANCE** is illustrated in Figure 2.3. Figure 2.3(a) shows that $l_0 = 0, r_0 = 2, l_1 = r_1 = 2, l_2 = r_2 = 2, l_3 = r_3 = 2, l_4 = 2, r_4 = 4, l_5 = 4, r_5 = 6, l_6 = r_6 = 6, l_7 = 6$ and $r_7 = 7$. In step [B2], P_0 distributes its appropriate data items over P_0 and P_1 , P_4 distributes its appropriate data items over P_2 and P_3 , P_5 distributes its appropriate data items over P_4 and P_5 , and P_7 sends its appropriate data items to P_6 . This step is illustrated in Figure 2.3(b). In step [B3], P_2 receives the appropriate data items from P_0, P_1, P_2 and P_3 , P_4 receives data items from P_4 , P_6 receives data items from P_5 and P_6 , and P_7 receives data items from P_7 . Notice that $x_2 = 0, x_4 = 1, x_6 = 2, x_7 = 3$. This step is illustrated in Figure 2.3(c).

Using the facts shown in the previous section, it is easy to show the following theorem.

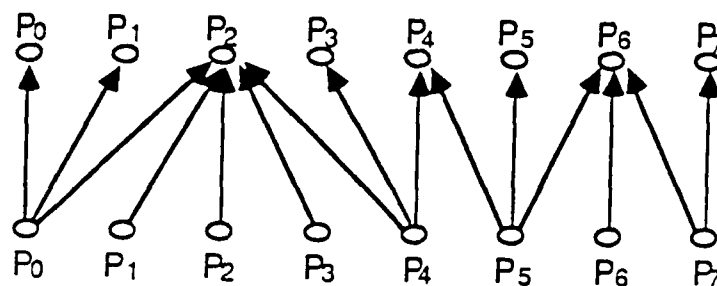
Theorem 2.2 *If each processor has a maximum of M data items, then load balancing can be achieved in time $O(M + \log p)$ on the pipeline hypercube. $\square$*

Notice that this algorithm is optimal, and faster than the considerably much more involved algorithm of [58]. However our model is not of bounded degree. We now consider the problem on the weak hypercube, the shuffle-exchange, cube-connected cycles and the butterfly. The following two lemmas are from [61].

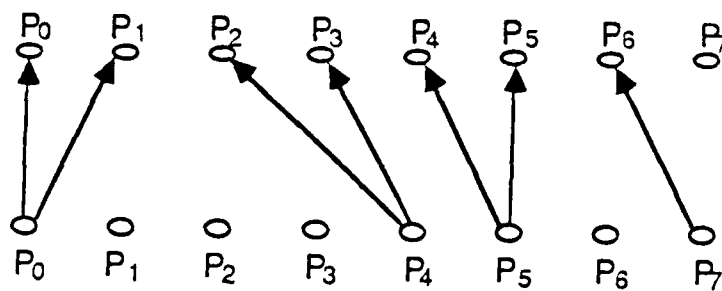
Lemma 2.6 *The load balancing problem requires $\Omega(k^{1/2}M)$ time on the weak hypercube when $M = \Theta(\frac{n}{p}p^{1/k})$ and $M \geq \frac{2n}{p}$. $\square$*

Lemma 2.7 *The load balancing problem can be solved in time $O(M \log^{1/2} p + \log^2 p)$ on the weak hypercube. $\square$*

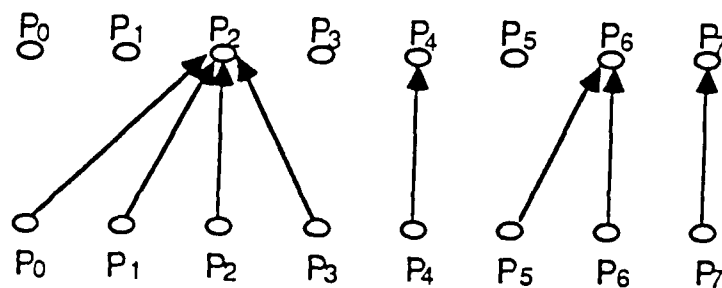
Notice that the lower bound of Lemma 2.6 can be rewritten in the form $\Omega(\frac{M\sqrt{\log p}}{\sqrt{\log \frac{2M}{n}}})$. If $M = O(\frac{n}{p})$, then this bound reduces to $\Omega(M\sqrt{\log p})$, and hence the upper bound of Lemma 2.7 is tight. We now establish similar results for the shuffle-exchange and the cube-connected cycles.



(a)



(b)



(c)

Figure 2.3: Step by step implementation of **BALANCE**

Theorem 2.3 *The load balancing problem requires $\Omega(\frac{n \log p}{p} + M \frac{\sqrt{\log p}}{\sqrt{\log(pM/n)}})$ time on a cube-connected cycles, a shuffle-exchange or a butterfly with p processors, when $M \geq \frac{2n}{p}$.*

Proof: We establish the proof for the shuffle-exchange network. The proof for the cube-connected cycles and the butterfly is similar. The main idea of the proof is to pack the items within a suitably chosen subset of the processors such that the number of links connecting this subset to the rest of the processors is small. The two terms in the lower bound correspond to two different choices of the subset. Let $p = 2^d$.

To get the first term, let $M < \frac{n \log p}{p}$. Using the characterization given in [33] for bisecting the shuffle-exchange graph, it is easy to see that we can partition the vertex set V into two subsets V_1 and V_2 such that $|V_1| = \lceil \frac{n}{M} \rceil$ and only $O(p/\log p)$ edges (shuffle) connect V_1 and V_2 . Pack all the n elements in V_1 . At least half of these elements have to cross the connecting edges to V_2 . Hence the first term of the lower bound follows.

To obtain the second term, let V_1 be the set of all vertices whose binary representations contain at most r ones, r a positive integer $\leq \frac{d}{2}$. Clearly $|V_1| = \sum_{i=0}^r C_i^d$. It is clear that the number of edges connecting V_1 to the rest of the nodes is bounded by C_r^d . Pack all the elements in V_1 . Thus $\Omega(\frac{M \sum_{i=0}^r C_i^d}{C_r^d})$ steps are required to send at least half of the elements in V_1 from V_1 to V_2 . Using an approximation shown in [61], we obtain the second term of the lower bound. $\square$

Notice that the first term in the above lower bound is dominant whenever $M = o(\frac{n}{p} \sqrt{\log p \log \log p})$. In particular, if $M = O(\frac{n}{p})$, we obtain that load balancing requires $\Omega(M \log p)$ time on the shuffle-exchange network, the cube-connected cycles or the butterfly. In view of Lemma 2.7, we conclude that the weak hypercube is strictly more powerful than the bounded-degree networks in load balancing. Notice that these bounded-degree networks can simulate normal hypercube algorithms with only constant slowdown, but the algorithm of [61] is not even a leveled algorithm.

The following theorem can be obtained by implementing algorithm BALANCE on the corresponding networks.

Theorem 2.4 *The load balancing problem can be solved on a p -processor shuffle-exchange, cube-connected cycles or butterfly in time $O(M \log p + \log^2 p)$. $\square$*

Notice that the bound of the above theorem is tight in the sense that it is optimal whenever $M = O(\frac{n}{p})$ and $p \leq \frac{n}{\log n}$.

2.4 Sorting

Sorting is a fundamental computational problem that has been investigated for several decades. Several efficient parallel sorting algorithms on the PRAM model [12,24] and on the network model [1,18,39,47,55,61,64,78,82] have been developed. In this section, we introduce two sorting algorithms, *mergesort* and *columnsort*, that are efficient on the hypercube under some conditions described below.

2.4.1 Mergesort

Let $W[0 \dots n-1]$ be an array of n data items such that subarray $W_i = W[\frac{ni}{p} \dots \frac{n(i+1)}{p} - 1]$ is stored in processor P_i , $0 \leq i < p$, of a hypercube. Then the mergesort algorithm can be described as follows.

procedure MERGESORT:

- [S1] Each processor independently sorts the subarray stored in it.
- [S2] For $j \leftarrow 1$ to $\log p$
 merge in each subcube of size 2^j .

We now describe the merge operation of step [S2] that utilizes a technique similar to that of the BALANCE algorithm. This algorithm is essentially from [82]. Let $A[0 \dots \frac{n}{2} - 1]$ and $B[0 \dots \frac{n}{2} - 1]$ be two sorted arrays of elements to be merged. The following simple algorithm merges the two arrays in time $O(\frac{n}{p} + \log p)$ on the pipelined hypercube. We assume that subarray $A_i = A[\frac{ni}{p} \dots \frac{n(i+1)}{p} - 1]$ is stored in processor P_i and subarray $B_j = B[\frac{nj}{p} \dots \frac{n(j+1)}{p} - 1]$ is stored in processor $P_{\frac{p}{2}+j}$, $0 \leq i, j < \frac{p}{2}$.

procedure MERGE:

- [M1] Let a_i and b_j be the minimum elements of subarrays A_i and B_j respectively, $0 \leq i, j < \frac{p}{2}$. Merge the a_i 's and b_j 's using odd-even or bitonic merge algorithm.
- [M2] If a_i is in P_k after merge, then move subarray A_i from P_i to P_k . Similarly, if b_j is in P_k after merge, then move subarray B_j from $P_{\frac{p}{2}+j}$ to P_k .
- [M3] Find every i such that P_i has a subarray from A and P_{i+1} has a subarray from B , or vice versa.
- [M4] Let i be such that P_i has subarray A_s , and $P_{i+1}, \dots, P_{i+l}$ have subarrays $B_{t+1}, \dots, B_{t+l}$ respectively, where l is the maximal such number. All the other cases can be treated similarly and simultaneously. Then broadcast A_s from P_i to P_{i+1} through P_{i+l} .

- [M5] In P_{i+j} , remove all the elements of A_s that are not greater than b_{t+j} or greater than b_{t+j+1} , $1 \leq j < l$. In P_{i+l} , remove all the elements of A_s that are not greater than b_{t+l} . And in P_i , remove all the elements of A_s that are greater than b_{t+1} . Let n_{ij} be the number of elements of A_s remained in P_{i+j} , $0 \leq j \leq l$. Clearly, $\sum_{j=0}^l n_{ij} = \frac{n}{p}$.
- [M6] Merge in each processor.
- [M7] Processor P_{i+j} sends its least $\sum_{k=j}^l n_{ik}$ elements to P_{i+j-1} , $1 \leq j \leq l$.

Clearly, step [M1] can be performed in time $O(\log p)$. Step [M3] can be also performed by using Concentrate routing. Steps [M2] and [M4] can be performed in time $O(\frac{n}{p} + \log p)$ by using Spread and Broadcast routing, respectively. Steps [M5] and [M6] are local operations, and can be done in time $O(\frac{n}{p})$. Step [M7] can be done in time $O(\frac{n}{p} + \log p)$ by using the strategy of steps [B2] and [B3] of algorithm **BALANCE**.

Lemma 2.8 [82] *Given two sorted arrays, each with $\frac{n}{2}$ elements, procedure **MERGE** merges the two arrays in time $O(\frac{n}{p} + \log p)$ on the pipelined hypercube.*
□

Corollary 2.1 *Given two sorted arrays, each with $\frac{n}{2}$ elements, procedure **MERGE** merges the two arrays in time $O(\frac{n \log p}{p})$ on the weak hypercube, the shuffle-exchange the cube-connected cycles and the butterfly.* □

Step [S1] of algorithm **MERGESORT** can be performed in time $O(\frac{n}{p} \log \frac{n}{p})$ and step [S2] can be performed in time $O(\frac{n \log p}{p} + \log^2 p)$. Thus, the total time for algorithm **MERGESORT** is $O(\frac{n \log n}{p} + \log^2 p)$ on the pipelined hypercube.

Theorem 2.5 [82] *If n elements are stored in p processors evenly, the elements can be sorted in time $O(\frac{n \log n}{p} + \log^2 p)$ on the pipelined hypercube.* □

Corollary 2.2 *If n elements are stored in p processors evenly, the elements can be sorted in time $O(\frac{n \log n \log p}{p})$ on the weak hypercube, the shuffle-exchange, the cube-connected cycles and the butterfly.* □

2.4.2 Columnsort

In many applications, we need an algorithm to sort small numbers faster. In this subsection, we will show how to sort $n = p^{1+\epsilon}$ numbers, each between 0 and $p^{O(1)}$, in time $O(\frac{n}{p})$, for any positive constant ϵ on the pipelined hypercube by using the *columnsort* algorithm of Leighton [47]. Clearly the algorithm can be implemented in time $O(\frac{n}{p} \log p)$ on the weak hypercube. Han used the same

$$\begin{pmatrix} 15 & 12 & 6 \\ 4 & 7 & 14 \\ 1 & 13 & 10 \\ 16 & 9 & 3 \\ 8 & 2 & 17 \\ 11 & 0 & 5 \end{pmatrix} \qquad \begin{pmatrix} 0 & 6 & 12 \\ 1 & 7 & 13 \\ 2 & 8 & 14 \\ 3 & 9 & 15 \\ 4 & 10 & 16 \\ 5 & 11 & 17 \end{pmatrix}$$

Figure 2.4: A 6×3 matrix before and after sorting

$$\begin{pmatrix} a_1 & a_7 & a_{13} \\ a_2 & a_8 & a_{14} \\ a_3 & a_9 & a_{15} \\ a_4 & a_{10} & a_{16} \\ a_5 & a_{11} & a_{17} \\ a_6 & a_{12} & a_{18} \end{pmatrix} \xRightarrow{\text{step 2}} \begin{pmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \\ a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} \\ a_{16} & a_{17} & a_{18} \end{pmatrix} \xleftarrow{\text{step 4}}$$

Figure 2.5: The transpose and its inverse

technique to implement the columnsort algorithm on the complete network [27]. We now describe the columnsort algorithm.

The columnsort algorithm is a generalization of odd-even merge sort and will be described as a series of elementary matrix operations. Let A be an $r \times s$ matrix of numbers where $rs = n$, s is divisible by r , and $r \geq 2(s-1)^2$. Initially, each entry of the matrix is one of the n numbers to be sorted. After the completion of the algorithm, A will be sorted in column major order form. For example, Figure 2.4 illustrates a typical matrix before and after sorting. Notice that this matrix does not satisfy $r \geq 2(s-1)^2$.

The columnsort algorithm has eight steps. In steps 1, 3, 5 and 7, the numbers within each column are sorted. We use the radix sort algorithm here. In steps 2, 4, 6 and 8, the entries of the matrix are permuted. The permutation in

$$\begin{array}{ccc}
\begin{pmatrix} a_1 & a_7 & a_{13} \\ a_2 & a_8 & a_{14} \\ a_3 & a_9 & a_{15} \\ a_4 & a_{10} & a_{16} \\ a_5 & a_{11} & a_{17} \\ a_6 & a_{12} & a_{18} \end{pmatrix} & \begin{array}{c} \text{step 6} \\ \Rightarrow \\ \\ \text{step 8} \\ \Leftarrow \end{array} & \begin{pmatrix} -\infty & a_4 & a_{10} & a_{16} \\ -\infty & a_5 & a_{11} & a_{17} \\ -\infty & a_6 & a_{12} & a_{18} \\ a_1 & a_7 & a_{13} & \infty \\ a_2 & a_8 & a_{14} & \infty \\ a_3 & a_9 & a_{15} & \infty \end{pmatrix}
\end{array}$$

Figure 2.6: The shift and its inverse

step 2 (shown for a 6×3 matrix in Figure 2.5) corresponds to a “transpose” of the matrix. The permutation in step 4 is the inverse of that in step 2. The permutation in step 6 corresponds to an $\lfloor \frac{7}{2} \rfloor$ shift of the entries, and is shown for a 6×3 matrix in Figure 2.6. The permutation in step 8 is the inverse of that in step 6. The step by step implementation of the columnsort algorithm is shown in Figure 2.7.

Before describing **COLUMNSORT** algorithm, which is the hypercube implementation of the columnsort algorithm, we show how to efficiently execute step 2 on the hypercube. We assume that each processor has p^ϵ numbers between 0 and $p^{O(1)}$ for some constant $\epsilon > 0$ and that $p^\epsilon = 2^j$ for some integer j . Let a be such that $p = 2^{3a+b}$, $0 \leq b \leq 2$. We can view the p processors as 2^b cubes each of size $2^a \times 2^a \times 2^a$. Each processor in cube l , ($0 \leq l \leq 3$), can be indexed as $PE(k_1, l, k_2, k_3)$, where $0 \leq k_1, k_2, k_3 \leq 2^a - 1$. The implementation of step 2 can be done as follows and is illustrated in Figure 2.8.

- [s1] For each k_1 , l and k_2 , let $PE(k_1, l, *, *)$ be a k_1 -block and $PE(*, l, k_2, *)$ be an k_2 -block of $2^a \times 2^a$ processors. We can consider each block as a matrix of size $2^a \times 2^a$. Transpose each k_2 -block matrix.
- [s2] Transpose each k_1 -block matrix.
- [s3] If there are more than 1 cube, shuffle the k_2 -blocks of cube 0 with those of cube 1. If there are more than 2 cubes, shuffle the k_2 -blocks of cube 2 with those of cube 3.
- [s4] If there are more than 2 cubes, then we consider 2 consecutive k_2 -blocks as an k_2 -block and shuffle k_2 -blocks of cube 0 and 1 with k_2 -blocks of cube 2 and 3.

$$\begin{array}{ccc}
\begin{pmatrix} 15 & 12 & 6 \\ 4 & 7 & 14 \\ 1 & 13 & 10 \\ 16 & 9 & 3 \\ 8 & 2 & 17 \\ 11 & 0 & 5 \end{pmatrix} & \Rightarrow & \begin{pmatrix} 1 & 0 & 3 \\ 4 & 2 & 5 \\ 8 & 7 & 6 \\ 11 & 9 & 10 \\ 15 & 12 & 14 \\ 16 & 13 & 17 \end{pmatrix} & \Rightarrow & \begin{pmatrix} 1 & 4 & 8 \\ 11 & 15 & 16 \\ 0 & 2 & 7 \\ 9 & 12 & 13 \\ 3 & 5 & 6 \\ 10 & 14 & 17 \end{pmatrix} & \Rightarrow & \\
\text{(input)} & & \text{step 1} & & \text{step 2} & & \\
\\
\begin{pmatrix} 0 & 2 & 6 \\ 1 & 4 & 7 \\ 3 & 5 & 8 \\ 9 & 12 & 13 \\ 10 & 14 & 16 \\ 11 & 15 & 17 \end{pmatrix} & \Rightarrow & \begin{pmatrix} 0 & 3 & 10 \\ 2 & 5 & 14 \\ 6 & 8 & 16 \\ 1 & 9 & 11 \\ 4 & 12 & 15 \\ 7 & 13 & 17 \end{pmatrix} & \Rightarrow & \begin{pmatrix} 0 & 3 & 10 \\ 1 & 5 & 11 \\ 2 & 8 & 14 \\ 4 & 9 & 15 \\ 6 & 12 & 16 \\ 7 & 13 & 17 \end{pmatrix} & \Rightarrow & \\
\text{step 3} & & \text{step 4} & & \text{step 5} & & \\
\\
\begin{pmatrix} -\infty & 4 & 9 & 15 \\ -\infty & 6 & 12 & 16 \\ -\infty & 7 & 13 & 17 \\ 0 & 3 & 10 & \infty \\ 1 & 5 & 11 & \infty \\ 2 & 8 & 14 & \infty \end{pmatrix} & \Rightarrow & \begin{pmatrix} -\infty & 3 & 9 & 15 \\ -\infty & 4 & 10 & 16 \\ -\infty & 5 & 11 & 17 \\ 0 & 6 & 12 & \infty \\ 1 & 7 & 13 & \infty \\ 2 & 8 & 14 & \infty \end{pmatrix} & \Rightarrow & \begin{pmatrix} 0 & 6 & 12 \\ 1 & 7 & 13 \\ 2 & 8 & 14 \\ 3 & 9 & 15 \\ 4 & 10 & 16 \\ 5 & 11 & 17 \end{pmatrix} & \Rightarrow & \\
\text{step 6} & & \text{step 7} & & \text{step 8 (output)} & &
\end{array}$$

Figure 2.7: The step by step implementation of the column sort

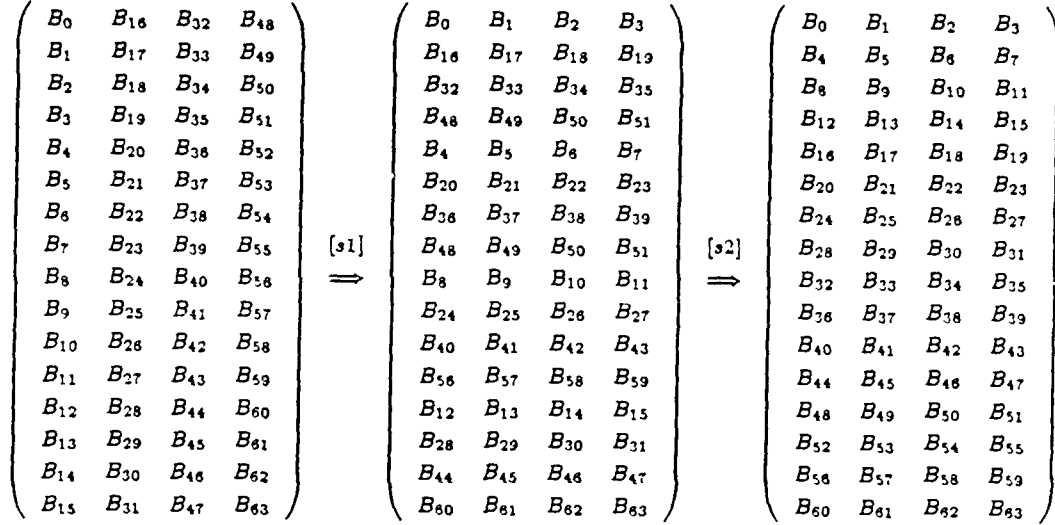


Figure 2.8: The hypercube implementation of step 2 of the columnsort. Steps [s3]-[s5] are omitted

[s5] For each l , k_2 and k_3 , perform a consecutive-to-cyclic conversion of elements in the 2^a processors of $PE(*, l, k_2, k_3)$.

Using the facts outlined in the previous section, it is clear that the time complexity of the above algorithm is $O(p^\epsilon)$ on the pipelined hypercube.

If $\epsilon \geq 2$, sorting each "column" can be done directly in each processor. Otherwise, sorting in step 1, 3, 5 and 7 can be done by a recursive application of the **COLUMNSORT** algorithm on $p^{\frac{2}{3}}$ processors. The recursive application ends when the number of remaining processors is less than or equal to $p^{\frac{1}{2}}$ and the recursion depth is at most $\lceil \frac{\log \epsilon - 1}{\log \frac{2}{3}} \rceil$. Step 4 is the inverse of step 2 and has the same time complexity. The permutations in step 6 and 8 can easily be implemented and their time complexities are $O(p^\epsilon)$. Thus, the total time $T(p, n)$ for **COLUMNSORT** is given by

$$T(p, p^{1+\epsilon}) = 4T(p^{\frac{2}{3}}, p^{\frac{2}{3}+\epsilon}) + O(p^\epsilon).$$

Theorem 2.6 *The algorithm columnsort can be implemented on the pipelined hypercube to sort $n = \Omega(p^{1+\epsilon})$ numbers between 0 and $p^{O(1)}$ in time $O(\frac{n}{p})$ for any positive constant ϵ . $\square$*

Corollary 2.3 *The algorithm columnsort can be implemented on the weak hypercube, the shuffle-exchange, the cube-connected cycles and the butterfly to sort*

$n = \Omega(p^{1+\epsilon})$ numbers between 0 and $p^{O(1)}$ in time $O(\frac{n \log p}{p})$ for any positive constant ϵ . $\square$

Notice that best known algorithm of integer sorting on the CRCW PRAM runs in time $O(\frac{n \log \log n}{p})$, for $1 \leq p \leq \frac{n \log \log n}{\log n}$ [24]. This PRAM algorithm is not efficient even on CRCW PRAM while our algorithm is efficient on the pipelined hypercube.

2.5 Routing

The most general version of the routing problem can be phrased as follows [58]. The (n, k_1, k_2) routing problem is a set of n packets, each of which is specified by a source and a destination, such that no processor appears as a source (respectively destination) in more than k_1 (respectively k_2) packets. We are assuming that these packets reside on the p processors, where $p \leq n$. The problem is to route these requests simultaneously. The best known solution to this problem is a $\Theta(k_1 + k_2 + \frac{n \log n}{p})$ algorithm on bounded-degree fixed connection network. The construction of such networks is highly nontrivial and depends on networks enabling sorting in $O(\log n)$ steps and on expander graphs. In this section, we will present a solution that matches this bound on the pipelined hypercube for all $p \leq n$. For the case when $n = p^{1+\epsilon}$, for any positive constant ϵ , our algorithm will run in time $O(k_1 + k_2 + \frac{n}{p})$. Optimal results on the weak hypercube, the shuffle-exchange, and the cube-connected cycles are also presented ($k_1 = O(\frac{n}{p}) = k_2$).

The algorithm presented in [58] depends crucially on the solutions of load balancing and sorting. Our algorithms for the general routing problem are similar to that of [58] but we provide new solutions to the above two subproblems on our networks. The load balancing and the sorting problem were considered in sections 2.3 and 2.4, respectively.

The (n, k_1, k_2) routing problem can be handled by a fully-normal algorithm combined with load balancing to obtain the following lemma.

Lemma 2.9 *The (n, k_1, k_2) routing problem can be solved in time $O(k_1 + k_2 \log p + \log^2 p)$ on the pipelined hypercube by using a fully-normal algorithm combined with the load balancing algorithm.*

Proof: The algorithm consists of $\log p$ stages. During the stage i , $i = \log p - 1, \dots, 0$, the packets in P_j , with the bit i of their destination labels different from that of j , are moved along the dimension i . We then apply the load balancing algorithm to the subcubes determined by the dimensions $0, 1, \dots, i - 1$. Notice that no processor will ever have more than $2k_2$ packets. $\square$

Corollary 2.4 *The (n, k_1, k_2) routing problem can be solved in time $O(k_1 + k_2 \log p + \log^2 p)$ on the pipelined hypercube. $\square$*

Actually the merge sort algorithm of the previous section whose time complexity is $O(\frac{n \log n}{p} + \log^2 p)$ can be used to obtain a better solution whenever k_2 is large.

Theorem 2.7 *The (n, k_1, k_2) routing problem can be solved on the pipelined hypercube in time $O(k_1 + k_2 + \frac{n \log n}{p} + \log^2 p)$.*

Proof: The overall strategy is similar to that of [58]. It consists of (1) load balancing, (2) sorting by destination labels, (3) counting the number of packets that have to be sent to each processor, (4) and then executing steps similar to [B2] and [B3] of the BALANCE algorithm. Using the time bounds of the load balancing and the sorting algorithms, the proof of the theorem follows. $\square$

We can do even better in the case when n is much larger than p by using algorithm **COLUMNSORT** of the previous section or the *cubesort* algorithm of Cypher and Sanz [18].

The cubesort algorithm sorts $n = p^{1+\frac{1}{l}}$ elements in time $O(l \frac{n \log n}{p})$ on the shuffle-exchange and in time $O(l^2 \frac{n \log n}{p})$ on the weak hypercube, the cube-connected cycles or the butterfly, where l is an arbitrary integer greater than 2. It consists of $O(l^2)$ stages, and each stage sorts groups containing $\frac{n}{p}$ elements and performs $O(l)$ shuffles or unshuffles. This algorithm can be easily modified to sort integers between 0 and $p^{O(1)}$ on the pipelined hypercube. Since each group is contained in a processor, it can be sorted in time $O(l \frac{n}{p})$ by using the radix sort algorithm. And by Lemma 2.4, the shuffle or unshuffle operation can be implemented in time $O(\frac{n}{p} + \log p)$. Thus we have proved the following lemma.

Lemma 2.10 *The cubesort algorithm can be implemented on the pipelined hypercube to sort $n = p^{1+\frac{1}{l}}$ integers between 0 and $p^{O(1)}$ in time $O(l^3 \frac{n}{p})$, where l is any positive integer greater than 2. $\square$*

Notice that when l is a constant, the cubesort algorithm can be implemented in time $O(\frac{n}{p})$, and when $l^3 = o(\log n)$, the algorithm is faster than the $O(\frac{n \log n}{p} + \log^2 p)$ merge sort algorithm of the previous section.

Corollary 2.5 *The (n, k_1, k_2) routing problem can be solved on the pipelined hypercube in time $O(k_1 + k_2 + \frac{n}{p} l^3)$, when $n = p^{1+\frac{1}{l}}$ for some integer $l > 2$. $\square$*

We now turn our attention to the case of the weak hypercube. When $n = p^{1+\frac{1}{l}}$, for a fixed positive constant l , the integer sorting and the (n, k_1, k_2) routing

problem can be solved in time $O(\frac{n}{p} \log p)$ and $O((k_1 + k_2) \log p)$ respectively on the weak hypercube, since each step on the pipelined hypercube can be simulated with $\log p$ steps on the weak hypercube. When l is not fixed, the routing problem can be solved in time $O((k_1 + k_2) \log p + l^2 \frac{n \log n}{p})$ by using the cubesort algorithm and a straightforward load balancing algorithm. However, the following lemma established by Plaxton [61] for sorting on the weak hypercube and the load balancing algorithm of Lemma 2.7 can be used to obtain a faster algorithm.

Lemma 2.11 *Let $q = \log^{3/2} p \log \log p$. Then, if $n \leq pq$, the n elements can be sorted in time*

$$T(n, p) = O(\frac{n}{p} \log(n/p) + \frac{n}{p} \log^{1.5} p + \log^3 p \log(n/p)),$$

and if $n > pq$, they can be sorted in time

$$T(n, p) = O(\frac{n}{p} \log p (\frac{\log p}{\log(n/(pq))})^{0.5} + \log^3 p \log(n/p)),$$

on the weak hypercube. $\square$

Corollary 2.6 *The (n, k_1, k_2) routing problem can be solved on the weak hypercube in time $O((k_1 + k_2) \log^{1/2} p + T(n, p))$, where $T(n, p)$ is as defined in Lemma 2.11. $\square$*

The facts that sorting can be done in time $O(l \frac{n \log n}{p})$ (respectively $O(l^2 \frac{n \log n}{p})$) on the shuffle-exchange (the cube-connected cycles or the butterfly) when $n = p^{1+\frac{1}{l}}$, for some $l > 2$, can be used to show the following.

Corollary 2.7 *The (n, k_1, k_2) routing problem can be solved on the shuffle-exchange or the cube-connected cycles in time $O((k_1 + k_2) \log p + l \frac{n \log n}{p})$ or $O((k_1 + k_2) \log p + l^2 \frac{n \log n}{p})$ when $n = p^{1+\frac{1}{l}}$, for some $l > 2$. $\square$*

Notice that the above upper bounds for the shuffle-exchange, the cube-connected cycles and the butterfly are optimal whenever $k_1 = O(\frac{n}{p})$, $k_2 = O(\frac{n}{p})$ and l is a fixed constant.

2.6 Relationship With The CRCW Model

Several powerful techniques have been developed for designing efficient parallel algorithms for the PRAM model, and it seems that this model is ideal for discovering inherent parallelism and for writing parallel algorithms. Therefore it is important to develop an efficient step by step simulation of a PRAM algorithm on our networks. Several simulations from the PRAM model onto these

networks are known. However there is always some loss of efficiency incurred by these simulations. The best known result emulates each step of a PRAM algorithm using p processors in time $O((\log p \log M)/\log \log p)$, where M is the size of the memory used by the PRAM algorithm [29]. This bound can be reduced to $O(\log p \log \log p)$ if $M = p(\log p)^{O(1)}$ [28]. The bounded-degree networks used are based on expander graphs that can sort optimally. Below we derive a simple simulation result for the pipelined hypercube model.

Lemma 2.12 *Let $\mathcal{A}$ be a PRAM algorithm whose running time is $T(n)$ using a total of $W(n)$ operations and a memory of size $M(n) = O(n)$, where n is the input size. Then $\mathcal{A}$ can be simulated on a p -processor pipelined hypercube in $O(\frac{W(n) \log n}{p} + \frac{n}{p}T(n) + T(n) \log^2 p)$ time.*

Proof: Decompose the memory into p equal size blocks B_i , $0 \leq i \leq p-1$. The memory M_i of processor P_i of the hypercube will hold the data in B_i . Let $W_j(n)$ be the number of operations used in step j , $1 \leq j \leq T(n)$, of the PRAM algorithm. Without loss of generality, assume that this step involves a read operation. The other cases are similar. Each processor P_i of the hypercube will handle $\frac{W_j(n)}{p}$ of these operations. This can be handled by a routing algorithm where each processor sets up $O(\frac{W_j(n)}{p})$ packets with the destinations assigned as determined by the initial memory map. Hence each processor is the source of $O(\frac{W_j(n)}{p})$ packets but could be the destination of $O(\frac{M(n)}{p}) = O(\frac{n}{p})$ packets. This can be done in $O(\frac{W_j(n) \log W_j(n)}{p} + \frac{n}{p} + \log^2 p)$ time. Therefore the overall simulation requires $O(\sum_{j=1}^{T(n)} (\frac{W_j(n) \log n}{p} + \frac{n}{p} + \log^2 p)) = O(\frac{W(n) \log n}{p} + \frac{n}{p}T(n) + T(n) \log^2 p)$. $\square$

For example, for algorithms with $W(n) = M(n) = O(n)$ and $T(n) = O(\log n)$, the simulation bound reduces to $O(\frac{n \log n}{p} + \log n \log^2 p)$. Note that if $M(n) \neq O(n)$, $\mathcal{A}$ can be simulated in time $O(\frac{W(n) \log n}{p} + \frac{M(n)}{p}T(n) + T(n) \log^2 p)$ on the pipelined hypercube. Note also that the above simulation result can be slightly improved if $n = \Omega(p^{1+\epsilon})$, for some fixed $\epsilon > 0$. Using a routing algorithm from the previous section, the bound of the above lemma can be improved to $O(\frac{W(n)}{p} + \frac{n}{p}T(n))$. By the same reasoning, the PRAM model can be simulated on the weak hypercube, the shuffle-exchange, the cube-connected cycles and the butterfly as follows.

Corollary 2.8 *Let $\mathcal{A}$ be a PRAM algorithm whose running time is $T(n)$ using a total of $W(n)$ operations and a memory of size $M(n) = O(n)$, where n is the input size. Then $\mathcal{A}$ can be simulated in time $O(\frac{W(n) \log n \log p}{p} + \frac{n \log p}{p}T(n))$ on a p -processor weak hypercube, the shuffle-exchange, the cube-connected cycles and the butterfly. $\square$*

Chapter 3

Almost Uniformly Optimal Algorithms

3.1 Introduction

Suppose we are given a network $\mathcal{N}$ with p processors. An algorithm to solve a given problem of size $n \geq p$ will be called *almost uniformly optimal* if the running time of the algorithm is provably the best possible for all $p \leq n / \log^k n$, for some fixed constant k . Such algorithms have been developed for many problems on the PRAM model. However except for very few cases, no such algorithms are known on the network model.

We address in this chapter several problems that can be solved by almost uniformly optimal algorithms on our networks. These problems are the all nearest smaller values (ANSV) problem, and some problems in computational geometry and in VLSI routing. All these problems can be solved efficiently in $O(\log n)$ time on the CREW PRAM (and even faster on the CRCW PRAM). The PRAM algorithms can be directly simulated in time $O(\frac{n \log n}{p} + \log n \log^2 p)$ on a p -processor pipelined hypercube. We provide faster algorithms to handle these problems. We also provide lower bound proofs of the problems on the weak hypercube, the shuffle-exchange, the cube-connected cycles and the butterfly.

The rest of the chapter is organized as follows. The ANSV and related problems are considered in section 3.2. The algorithms for a couple of basic problems in VLSI routing are presented in section 3.3. The last section is devoted to the lower bound proofs.

3.2 ANSV and Related Problems

The *all nearest smaller values* (ANSV) problem can be defined as follows. The input consists of an array $A = (a_0, a_1, \dots, a_{n-1})$, where the a_i 's come from a

totally ordered set. The output is an array B such that $B(i) = (a_{l(i)}, a_{r(i)})$, where $a_{l(i)}$ and $a_{r(i)}$ are respectively the nearest elements to the left and to the right of a_i that are less than a_i , if they exist. If one or both of them do not exist, this can be indicated with a special symbol. We call $a_{l(i)}$, whenever it exists, the *left match* of a_i . We similarly call $a_{r(i)}$, whenever it exists, the *right match* of a_i .

The ANSV problem was introduced in [7]. It turns out that merging is a special case of ANSV. This problem can be solved sequentially in linear time by using a stack. An optimal CRCW PRAM algorithm with running time $O(\log \log n)$ was shown in [7]. This algorithm was then used to solve the monotone polygon triangulation, the binary tree reconstruction, and parenthesis matching within the same bounds.

We will present an almost uniformly optimal algorithm to handle the ANSV problem on the pipelined hypercube model. We will later see that its implementation on the shuffle-exchange, the cube-connected cycles or the butterfly is also almost uniformly optimal.

Assume from now on that all elements in A are distinct and that n and p are both powers of 2. The other cases are treated in a straightforward way. We start with a simple divide-and-conquer algorithm which is fast in the case when $\frac{n}{\log^3 n} < p \leq n$.

procedure Simple ANSV

Input: An array $A = (a_0, \dots, a_{n-1})$ stored consecutively on a p -processor hypercube.

Output: An array $B = (b_0, \dots, b_{n-1})$ such that $b_i = (a_{l(i)}, a_{r(i)})$, where $a_{l(i)}$ and $a_{r(i)}$ are respectively the left and the right matches of a_i .

1. If $p = 1$, then use an optimal serial algorithm to solve the ANSV problem.
2. Recursively, solve separately each of the the ANSV problems corresponding to the two subarrays $A_0 = (a_0, a_1, \dots, a_{n/2-1})$ and $A_1 = (a_{n/2}, a_{n/2+1}, \dots, a_{n-1})$ stored in the low and high subcubes.
3. Let $A'_0 = (a_{i_1}, \dots, a_{i_s})$ be all the elements of A_0 that do not have their right matches in A_0 , and let $A'_1 = (a_{j_1}, \dots, a_{j_t})$ be all the elements of A_1 that do not have their left matches in A_1 . Then $a_{i_1} \leq a_{i_2} \leq \dots \leq a_{i_s}$ and $a_{j_1} \geq a_{j_2} \geq \dots \geq a_{j_t}$. Find the right matches of the elements of A'_0 and the left matches of the elements of A'_1 by merging the corresponding two sequences.

Lemma 3.1 *The right match of each element in A'_0 is in A'_1 , if it exists. Similarly, the left match of each element of A'_1 is in A'_0 , if it exists.*

Proof: Let a be an element of A_0 whose right match b is in A_1 . Then clearly a is in A'_0 . Suppose that b is not in A'_1 . This implies that b has a left match c in A_1 . Hence $c < b$ and c appears before b in the array A . But this means that the right match of a cannot be b . Therefore b does not have a left match in A_1 . $\square$.

Theorem 3.1 *Algorithm Simple ANSV correctly solves the ANSV problem in $O(\frac{n \log p}{p} + \log^2 p)$ on a p -processor pipelined hypercube.*

Proof: The correctness proof is simple and can be established by induction on p . As for the running time, we note that the merging required at step 3 can be done in $O(\frac{n}{p} + \log p)$ on the pipelined hypercube model (subsection 2.4.1). Thus the time bound follows. $\square$

We will develop a faster algorithm in the rest of this section. We start by stating a simple fact (also stated in [7]) that will be needed to justify the algorithm.

Lemma 3.2 [7] *Let $1 \leq j \leq n$ be an arbitrary index and let $I[j] = \{j + 1, \dots, r(j) - 1\}$ whenever $r(j)$ exists. Then the following statements hold.*

(1) *If $k \in I[j]$, then $r(k), l(k) \in I[j] \cup \{j, r(j)\}$.*

(2) *If $k \notin I[j]$, then $r(k), l(k) \notin I[j]$.*

Similar statements hold for $I'[j] = \{l(j) + 1, \dots, j - 1\}$. $\square$

Partition A consecutively into p blocks, say $A_0, A_1, \dots, A_{p-1}$, i.e., $A_i = (a_{in/p}, a_{(i+1)n/p}, \dots, a_{(i+1)n/p-1})$, $0 \leq i \leq p-1$. Let $m(i)$ be the index of the minimum element in A_i . Define the reduced array A' to be $A' = (a_{m(0)}, a_{m(1)}, \dots, a_{m(p-1)})$. For each block A_i , all the elements in A_i appearing before $a_{m(i)}$ have their right matches in A_i , and all elements appearing after $a_{m(i)}$ have their left matches in A_i . We now state the following fact from [7].

Lemma 3.3 *Let A_i be an arbitrary block such that the right match of $a_{m(i)}$ belongs to block $A_{i'}$, $i' \neq i + 1$. Then there exists a unique k , $i < k < i'$, such that the left match of $a_{m(k)}$ is in A_i and the right match of $a_{m(k)}$ is in $A_{i'}$.* $\square$

Let A_i be an arbitrary block such that the right match of $a_{m(i)}$ is in block $A_{i'}$. With each A_i we associate two subsequences $(S_{i,0}, S_{i,1})$, where $S_{i,0}$ is a segment of A_i and $S_{i,1}$ is a segment of $A_{i'}$ defined as follows:

1) If $i' = i + 1$, then $S_{i,0} = (a_{m(i)}, \dots, a_{(i+1)p})$ and $S_{i,1} = (a_{(i+1)p+1}, \dots, a_{r(m(i))})$. See Figure 3.1(a).

2) If $i' > i + 1$, then let k be the index whose existence is mentioned in the above lemma. Then $S_{i,0} = (a_{m(i)}, \dots, a_{l(m(k))})$ and $S_{i,1} = (a_{r(m(k))}, \dots, a_{r(m(i))})$. See Figure 3.2(b).

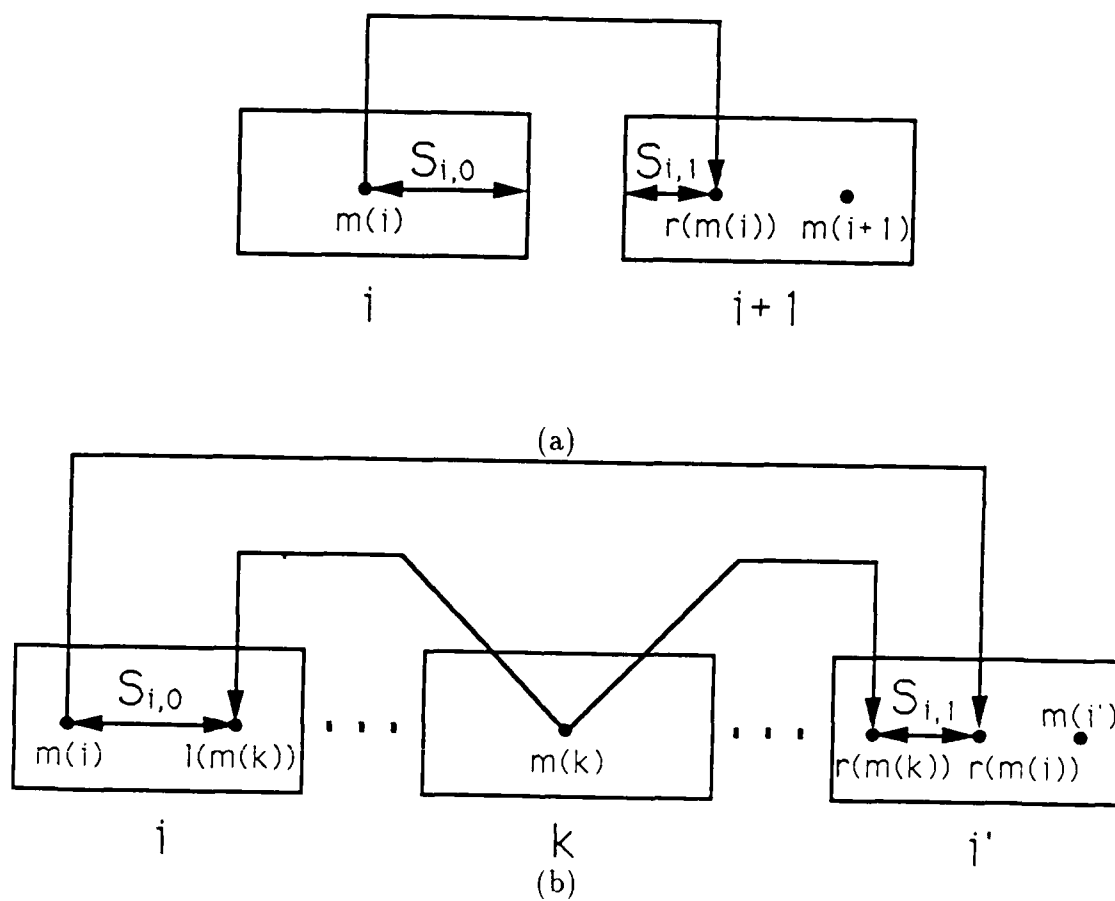


Figure 3.1: Subsequences corresponding to (a) $i' = i + 1$ and (b) $i' > i + 1$

Let $S'_{i,0}$ ($S'_{i,1}$) be the set of all the elements in $S_{i,0}$ ($S_{i,1}$) that do not have their right (left) matches in A_i (A'_i). By Lemma 3, it is clear that the right matches of $S'_{i,0}$ are in $S'_{i,1}$ and the left matches of $S'_{i,1}$ are in $S'_{i,0}$, if they exist. Moreover the elements of $S'_{i,0}$ are in increasing order while the elements of $S'_{i,1}$ are in decreasing order. Hence we can find the right matches of $S'_{i,0}$ and the left matches of $S'_{i,1}$ by merging them.

We can similarly consider each block A_i such that the left match of $a_{m(i)}$ exists. We can then introduce an index similar to k and the corresponding subsequences. Again the problem comes down to a set of disjoint merging problems.

It turns out that the left and the right matches of all the elements can be obtained by determining the left and the right matches within each block, and by merging the pairs of subsequences arising by considering the right matches of $a_{m(i)}$'s and then merging the subsequences arising from the left matches of the $a_{m(i)}$'s.

We are ready to describe our algorithm.

procedure ANSV

Input: An array $A = (a_0, a_1, \dots, a_{n-1})$ of n distinct elements.

Output: The array $B = (b_0, b_1, \dots, b_{n-1})$ such that $b_i = (a_{l(i)}, a_{r(i)})$, where $a_{l(i)}$ and $a_{r(i)}$ are respectively the left and the right matches of a_i .

1. Let $A_i = (a_{ni/p}, \dots, a_{n(i+1)/p-1})$ be the array stored in P_i , $0 \leq i \leq p-1$. Solve the ANSV problem corresponding to each subarray.
2. Find the minimum element $a_{m(i)}$ in each subarray A_i .
3. Solve the ANSV problem corresponding to the reduced array A' on p processors.
4. For each subarray A_i , if the right match of $a_{m(i)}$ is in block A_{i+1} , then move subarray A_{i+1} to P_i . Merge the corresponding subsequences within each processor. Move the left matches found for the subsequence in A_{i+1} to P_{i+1} .
5. For each subarray A_i such that the right match of $a_{m(i)}$ is in block $A_{i'}$ with $i' > i+1$, determine the index k described in Lemma 4. Move $a_{m(k)}$ and $A_{i'}$ to processor P_i . Merge the corresponding subsequences within each processor. Move the left matches found for the subsequence in $A_{i'}$ to $P_{i'}$.
6. Repeat steps 4 and 5 for the left pairs of subsequences.

Theorem 3.2 *Algorithm ANSV correctly finds all the right and the left matches of the array A of n elements. It can be implemented on a p -processor pipelined hypercube to run in time $O(\frac{n}{p} + \log^4 p)$.*

Proof: The correctness proof follows essentially from [7]. We will now establish the stated time bound.

Steps 1 and 2 can be obviously done in $O(\frac{n}{p})$. Step 3 can be done in $O(\log^2 p)$ time by using Simple ANSV. Step 4 can be implemented as follows. Each processor P_i determines whether or not the right match of $a_{m(i)}$ found in step 3 is the element $a_{m(i+1)}$. If this is the case, it sets up a request for the subarray A_{i+1} . Since each A_i is of size $\frac{n}{p}$, the corresponding data movement can be accomplished in $O(\frac{n}{p} + \log p)$ time.

As for step 5, each processor say P_k sets up a request for the right match of $a_{m(i)}$, where i is the index of the subarray containing the left match of $a_{m(k)}$. These requests can be satisfied in $O(\log^2 p)$ time. Then P_k checks to see if the right match of $a_{m(i)}$ is in the same subarray as that of $a_{m(k)}$. If this is the case, $a_{m(k)}$ is sent to P_i in the next step. Let $\alpha : \{0, 1, \dots, p-1\} \rightarrow \{0, 1, \dots, p-1\}$ be the partial function $\alpha(i) = i'$, where i' is the index of the subarray containing the right match of $a_{m(i)}$. We can now apply Theorem 2.1 to perform the data movement required for step 5. Merging within each processor can be easily done in $O(\frac{n}{p})$ time. The time required for the data movement is $O(\frac{n}{p} + \log^4 p)$. The only thing left is to send back the left matches of corresponding elements in A_i from P_i to $P_{i'}$. This can also be done by similar steps as in Theorem 1 and the collect operation.

The analysis of the time required to execute step 6 is similar and hence the theorem follows. $\square$

Note that the above performance cannot be improved for $p \leq \frac{n}{\log^4 n}$ even on the CRCW PRAM model.

Corollary 3.1 *The ANSV problem can be solved in $O(\frac{n \log p}{p} + \log^4 p)$ time on a p -processor weak hypercube with a normal algorithm. Hence it can be solved within the same time bound on a p -processor butterfly, shuffle-exchange, and cube-connected-cycles. $\square$*

3.2.1 The Parentheses Matching Problem

Let $A = (a_0, a_1, \dots, a_{n-1})$ be a legal sequence of parentheses, where each $a_i = '('$ or $)'$. The parentheses matching problem is to determine for each a_i the index j such that a_j is the match of a_i . It is well-known that this problem can be solved as follows. Compute the nesting levels of the parentheses using prefix sums, and then apply ANSV to find matching parentheses which will have the same level of nesting but the level of nesting of any parenthesis between them is higher. Using our ANSV algorithm, we obtain the following corollary.

Corollary 3.2 *Given a legal sequence of left and right parentheses, the parentheses matching problem can be solved within the same time bound as that of ANSV on any of the parallel models considered in this thesis. $\square$*

3.2.2 Triangulating a Monotone Polygon

A polygonal chain $Q = (q_0, \dots, q_{n-1})$ is said to be *monotone* if the vertices $q_0, \dots, q_{n-1}$ are in increasing (or decreasing) order by their x -coordinates. A *monotone polygon* consists of an upper monotone polygonal chain and a lower monotone polygonal chain. The goal is to determine a set of edges, each edge connecting a pair of vertices, which will triangulate the input polygon.

A known strategy to solve the problem of triangulating a monotone polygon is the following. Let a *one-sided* monotone polygon be a monotone polygon whose upper or lower chain is a straight line. A solution strategy consists of (i) decomposing the input polygon into one-sided monotone polygons by merging the vertices of the lower and the upper chains and then (ii) use ANSV to determine additional edges needed to triangulate the input polygon. We thus have the following corollary.

Corollary 3.3 *A monotone polygon can be triangulated within the same time bound as that of solving the ANSV problem on any of the parallel models considered in this thesis. $\square$*

3.2.3 The All Nearest Neighbor Problem

Let $Q = (q_0, \dots, q_{n-1})$ be a convex polygon, where (q_i, q_{i+1}) is an edge of Q , $0 \leq i \leq n-2$. The all nearest neighbor problem for Q is to determine, for each vertex q_i , a vertex q_j , $i \neq j$, such that the Euclidean distance between q_i and q_j is minimal. This problem can be solved optimally by using essentially merging [69]. Hence the corresponding algorithm runs in time $O(\frac{n}{p} + \log p)$ on the p -processor pipelined hypercube and in time $O(\frac{n \log p}{p} + \log^2 p)$ time on the p -processor weak hypercube, butterfly, shuffle-exchange, or cube-connected-cycles.

3.3 VLSI Routing

In this section we introduce two basic problems whose solutions can be used to solve several VLSI routing problems. We start with the first problem which is useful for handling river (one-layer) routing problems [10]. The input consists of two arrays $B = (b_0, b_1, \dots, b_{n-1})$ and $T = (t_0, t_1, \dots, t_{n-1})$, where $b_0 < b_1 < \dots < b_{n-1}$ and $t_0 < t_1 < \dots < t_{n-1}$ such that $b_j < b_{j+1} \leq t_j$, for all $0 \leq j \leq n-2$. The output is the array $S = (t_{j(0)}, t_{j(1)}, \dots, t_{j(n-1)})$, where $j(i) = \min_j \{j \leq i | t_j + i - j - 1 \geq b_i\}$. If we view B and T as representing the bottom and the top terminals of an arbitrary instance of river routing, then all the bendpoints of the detailed routing can be deduced from S . Moreover, the minimum separation is given by $\max_i \{i - j(i) + 1\} + 1$.

A simple algorithm to handle this problem can be obtained as follows. Let $t'_j = t_j - j - 1$ and $b'_i = b_i - i$. Then $j(i)$ is given by $j(i) = \min_j \{j \leq i | t'_j \geq b'_i\}$.

This can be done by merging the $(b'_0, b'_1, \dots, b'_{n-1})$ and $(t'_0, t'_1, \dots, t'_{n-1})$, and then determining for each b'_i the nearest t'_j to the right. Hence this problem can be solved in $O(\frac{n}{p} + \log p)$ time on a p -processor pipelined hypercube, and in time $O(\frac{n \log p}{p} + \log^2 p)$ on a p -processor butterfly, shuffle-exchange, cube-connected-cycles, or weak hypercube. See section 5.2 for more details. We will later derive the corresponding lower bound.

The second basic problem, called *line packing*, consists of packing a set of n intervals $I_0, I_1, \dots, I_{n-1}$ using the minimum possible number of tracks [19]. More precisely, our input is given as an array $A = (a_0, a_1, \dots, a_{2n-1})$, where $a_j = (x_j, id(j), mark(j))$ such that x_j is the x -coordinate of a terminal (end-point of an interval), $id(j)$ is the serial number of the corresponding interval, and $mark(j)$ indicates whether a_j corresponds to the left endpoint or the right endpoint of $I_{id(j)}$. Moreover we are assuming that the a_i 's are sorted by their first components. The desired output is an array $B = (b_0, b_1, \dots, b_{2n-1})$ such that $b_j = a_{s(j)}$, where $a_{s(j)}$ corresponds to a left terminal that follows in the same track the right terminal of a_j . If a_j corresponds to a left terminal or no interval comes after $I_{id(j)}$ in the same track, then b_j is not defined. The number of tracks should be minimized. It turns out that the minimum number of tracks is equal to the density $d = \max_x \{d_x\}$, where d_x is the number of intervals containing x .

The solution of the line packing problem can be used to solve the channel routing problem in the two-layer model, where each column contains at most one terminal. It can also be used to perform optimal routing in the knock-knee model [9]. The algorithm is given below.

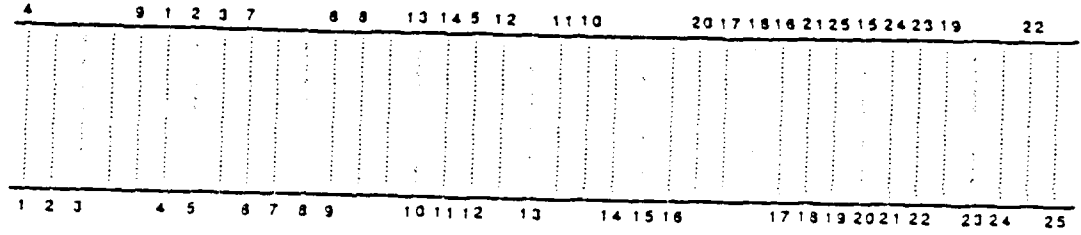
procedure Line Packing

Input: A sorted array $A = (a_0, a_1, \dots, a_{2n-1})$ representing the endpoints of n intervals. When two endpoints have an equal x -coordinate, the right endpoint precedes the left endpoint in A .

Output: The array $B = (b_0, b_1, \dots, b_{2n-1})$ as defined above.

1. Assign +1 to each left terminal and -1 to each right terminal, and compute the prefix sums of all the terminals.
2. For each right terminal a_j whose prefix sum value is v , find the nearest left terminal $a_{s(j)}$ to the right of a_j whose prefix sum value is greater than v . Set $b_j = a_{s(j)}$ if such $s(j)$ exist, and *nil* otherwise.

As an example, a channel routing instance, the corresponding sorted list and prefix sums, and chains of intervals to be put in the same tracks, are shown in Figure 3.2. This example is from [9]. The correctness proof of the algorithm follows from [19]. Now, we have the following corollary.



(a)

l_1	l_4	l_2	l_3	l_9	r_1	r_4	r_2	l_5	r_3	l_6	l_7	r_7	l_8	r_6	r_9	r_8
1	2	3	4	5	4	3	2	3	2	3	4	3	4	3	2	1
l_{10}	l_{13}	l_{11}	l_{14}	r_5	l_{12}	r_{12}	r_{13}	r_{11}	r_{10}	r_{14}	l_{15}	l_{16}	l_{20}	l_{17}	l_{18}	r_{16}
2	3	4	5	4	5	4	3	2	1	0	1	2	3	4	5	4
r_{17}	r_{18}	l_{21}	l_{19}	l_{25}	r_{15}	r_{20}	r_{21}	l_{24}	l_{22}	l_{23}	r_{19}	r_{23}	r_{24}	r_{22}	r_{25}	
3	2	3	4	5	4	3	2	3	4	5	4	3	2	1	0	

(b)

$$\begin{aligned}
 I_1 &\rightarrow I_{14} \rightarrow I_{15} \rightarrow I_{23} \\
 I_4 &\rightarrow I_7 \rightarrow I_8 \rightarrow I_{10} \rightarrow I_{16} \rightarrow I_{25} \\
 I_2 &\rightarrow I_5 \rightarrow I_{12} \rightarrow I_{18} \rightarrow I_{21} \rightarrow I_{24} \\
 I_3 &\rightarrow I_6 \rightarrow I_{11} \rightarrow I_{20} \rightarrow I_{22} \\
 I_9 &\rightarrow I_{13} \rightarrow I_{17} \rightarrow I_{19}
 \end{aligned}$$

(c)

Figure 3.2: (a) A channel routing instance, (b) corresponding sorted list and prefix sums, (c) chains of intervals

Corollary 3.4 *Given a set of intervals whose terminals are sorted as described above, the line packing problem can be solved within the same time bound as that of solving the ANSV problem on any of the parallel models considered in this thesis. $\square$*

We will derive the corresponding lower bound in the next section.

3.4 Lower Bounds

The performance of all the algorithms presented has degraded by a factor of $\log p$ in the transition from the pipelined hypercube model to the weak hypercube model or to any of the related bounded-degree networks. We will show in this section that these upper bounds cannot be improved on the bounded-degree networks in general (i.e. as long as $\frac{n}{p} \geq \log^3 p$). Before presenting the proofs, a couple of comments concerning the general network model are in order.

For all the problems considered in this chapter, each of the input and the output can be efficiently represented by an array of data items. Let $A = (a_0, a_1, \dots, a_{n-1})$ be such an input. The *input memory map* $\pi_{in} : \{0, 1, \dots, n-1\} \rightarrow \{0, 1, \dots, p-1\}$ specifies the index mapping of the elements of A into the local memories, say $\{M_0, M_1, \dots, M_{p-1}\}$, of the p -processor network. For all the algorithms presented, π_{in} corresponds to the *consecutive* memory mapping, i.e., $\pi_{in}(j) = \lfloor \frac{j}{n} \rfloor$ (assuming as usual that p divides n evenly). We can similarly define the *output memory map* $\pi_{out} : \{0, 1, \dots, n-1\} \rightarrow \{0, 1, \dots, p-1\}$, where $\pi_{out}(j)$ is the index of the local memory containing the j th data item of the output array. Again all our algorithms generate an output stored in consecutive order.

If we make the assumption that π_{in} and π_{out} correspond to consecutive storage, then the lower bound of $\Omega(\frac{n \log p}{p})$ can be established for the weak hypercube model and the bounded-degree networks by using the following simple technique. For each of our problems, there exist instances which will require the exchange of the data in the local memories of $\Omega(p)$ pairs of processors, each pair with a Hamming distance of $\Omega(\log p)$. Since only $O(p)$ data items can be communicated during each unit of time, we obtain that $\Omega(\frac{n \log p}{p})$ time is needed to handle the communication. However it is conceivable that a problem could become significantly simpler if the input memory map somehow exploits the topology of the network and match it properly with the problem. Therefore we will establish our lower bounds under the assumptions that π_{in} and π_{out} are arbitrary, data-independent mappings such that, for each j , $0 \leq j \leq p-1$, $|\{i | \pi_{out}(i) = j\}| = \frac{n}{p}$ (i.e. the output array is evenly distributed among the local memories of the different processors). Under these conditions we can also assume that the input array is evenly distributed among the local memories of the processors, for

otherwise balancing the data alone will require $\Omega(\frac{n \log p}{p})$ on the bounded-degree networks [35].

The basic technique we use to establish all of our lower bounds is simple and well-known. Our bounded-degree networks all have $\Theta(\frac{p}{\log p})$ strong separators. Since the separator of a graph is a communication bottleneck, if we can show that $\Omega(n)$ data items have to be exchanged between the two partitions of the processors induced by a separator, then the lower bound of $\Omega(\frac{n \log p}{p})$ will immediately follow. We will show that this is indeed the case for all our problems.

We begin by providing the lower bound for the merging problem. Let $A = (a_0, a_1, \dots, a_{\frac{n}{2}-1}, a_{\frac{n}{2}}, \dots, a_{n-1})$ be an array such that subarrays $A_0 = (a_0, \dots, a_{\frac{n}{2}-1})$ and $A_1 = (a_{\frac{n}{2}}, \dots, a_{n-1})$ are sorted in nondecreasing order. Assume for simplicity that p divides n . Processor P_i , $0 \leq i \leq p-1$, has $\frac{n}{p}$ elements of A before and after merging.

Lemma 3.4 *Let PT_0 and PT_1 be an arbitrary partition of the p processors such that $|PT_0| = |PT_1| = \frac{p}{2}$. Then the merging problem requires $\Omega(n)$ data exchanges between PT_0 and PT_1 .*

Proof: Without loss of generality, assume that at least $\frac{n}{4}$ elements of A_0 (the lower half of A) are in the local memories of PT_1 . Since π_{out} is data-independent, the items generated in PT_0 are of fixed ranks, say $1 \leq r_0 < r_1 < \dots < r_{\frac{n}{2}-1} \leq n$. Note that the rank of each element a_i of A_0 is equal to $i+1$ (its rank in A_0) plus its rank in A_1 . It is clear that A can be chosen so that the rank of each a_i , $0 \leq i \leq \frac{n}{2}-1$, is exactly r_i . But in this case all the elements of A_0 have to appear in PT_0 . By our assumption, at least $\frac{n}{4}$ of these elements are in PT_1 . Therefore $\Omega(n)$ data items have to be exchanged between PT_0 and PT_1 . $\square$

Corollary 3.5 *The problem of merging two sorted sequences each of length n requires $\Omega(\frac{n \log p}{p})$ time steps on a p -processor butterfly, shuffle-exchange, or cube-connected-cycles, and $\Omega(\frac{n \sqrt{\log p}}{p})$ time steps on the weak hypercube model with p processors. $\square$*

We now introduce the following Restricted ANSV (RANSV) which will be used to establish the lower bounds for ANSV and for the problem of triangulating a monotone polygon. The input consists of two arrays $A = (a_0, a_1, \dots, a_{n-1})$ and $B = (b_0, b_1, \dots, b_{n-1})$, each sorted in non-decreasing order. The output consist of two arrays $A' = (a'_0, a'_1, \dots, a'_{n-1})$ and $B' = (b'_0, b'_1, \dots, b'_{n-1})$, where b'_i (respectively a'_i) is the largest element in B (respectively A) such that $b'_i \leq a_i$ (respectively $a'_i \leq b_i$). Note that RANSV is obviously equivalent to merging in the PRAM model. We next establish a lower bound for this problem.

Lemma 3.5 *Let PT_0 and PT_1 be an arbitrary partition of p processors such that $|PT_0| = |PT_1| = \frac{p}{2}$. The problem of solving RANSV requires $\Omega(n)$ data exchanges between PT_0 and PT_1 .*

Proof: Let PT_0 and PT_1 be an arbitrary partition of the p processors such that $|PT_0| = |PT_1| = \frac{p}{2}$. Let n_0^A and n_1^A be the numbers of elements of A distributed in PT_0 and PT_1 respectively. We can define n_0^B and n_1^B similarly. Note that $n_0^A = n_1^B$ and $n_1^A = n_0^B$. Without loss of generality we assume that $n_0^A \geq n_1^A$. Let $\{a_{j(1)}, \dots, a_{j(n_0^A)}\}$ be elements from A that are in PT_0 and $\{b_{k(1)}, \dots, b_{k(n_1^B)}\}$ be elements from B that are in PT_1 , where $j(1) < \dots < j(n_0^A)$ and $k(1) < \dots < k(n_1^B)$. Let A be such that $b_{k(i)}$ is the largest element that is $\leq a_{j(i)}$, $1 \leq i \leq n_0^A$. Clearly, such arrays A and B exist and at least $\frac{n}{2}$ elements must pass between PT_0 and PT_1 to find the matches since $n_0^A \geq \frac{n}{2}$. $\square$

Corollary 3.6 *The ANSV problem requires $\Omega(\frac{n \log p}{p})$ time steps on a p -processor butterfly, shuffle-exchange, or cube-connected-cycles, and $\Omega(\frac{n \sqrt{\log p}}{p})$ time steps on the weak hypercube model. $\square$*

Proof: Let $\mathcal{A}$ be an algorithm of running time T for solving ANSV with the input memory map π_{in} and the output memory map π_{out} . We show how to solve RANSV in time T . Let A and B be the input arrays to RANSV. Assume without loss of generality that all the elements of A and B are distinct. The input to ANSV will be the array $C = (A, B')$, where B' is the array B given in non-increasing order. Use π_{in} to store C as required by $\mathcal{A}$. Run algorithm $\mathcal{A}$ on the corresponding input. The output map $\pi_{out}(j)$ determines the local memory containing the left and right matches of the j th input. Let $c_j = a_i$ for some i . The right match of a_i is precisely the largest element of B that is less than or equal to a_i . Similarly for the case when $c_j = b_i$. Hence we can solve RANSV in time T using algorithm $\mathcal{A}$. $\square$

Corollary 3.7 *Given a monotone polygon P with its n sides sorted, triangulating P requires $\Omega(\frac{n \sqrt{\log p}}{p})$ time steps on the weak hypercube and $\Omega(\frac{n \log p}{p})$ time steps on the butterfly, the shuffle-exchange or the cube-connected cycles.*

Proof: Let $A = (a_0, a_1, \dots, a_{n-1})$ and $B = (b_0, b_1, \dots, b_{n-1})$ be two arrays sorted in nondecreasing order. Without loss of generality, we assume that all the elements are distinct, and that $a_0 < b_0$, $a_{n-1} > b_{n-1}$ and $a_0 > 0$. Let $l = a_{n-1}$. We define the following monotone polygon P . The upper chain of the polygon is

$$((a_0, 0), (\frac{a_0 + a_1}{2}, 1), (a_1, 0), \dots, (\frac{a_{n-2} + a_{n-1}}{2}, 1), (a_{n-1}, 0))$$

and the lower chain is

$$((a_0, 0), (\frac{a_0 + b_0}{2}, -1), (b_0, \frac{\sqrt{b_0}}{l^2}), (\frac{b_0 + b_1}{2}, -1), (b_1, \frac{\sqrt{b_1}}{l^2}), \dots,$$

$$(b_{n-1}, \frac{\sqrt{b_{n-1}}}{l^2}), (\frac{b_{n-1} + a_{n-1}}{2}, -1), (a_{n-1}, 0)).$$

Then by triangulating the monotone polygon, we can find for each a_i (b_i) the largest element in B (A) that is $\leq a_i$ ($\leq b_i$) [7]. Therefore the edges $(a_i, b_{j(i)})$ and $(b_i, a_{k(i)})$ have to be generated. Using the same argument as in Lemma 6, we conclude that there are some instances of this problem requiring the specified amount of time on the networks. $\square$

We now consider the lower bound of the other computational geometry problem, the ANN problem. The problem RANSV1 which is similar to RANSV and useful in proving the lower bound of the ANN problem can be defined as follows: The input consists of two arrays $A = (a_0, a_1, \dots, a_{n-1})$ and $B = (b_0, b_1, \dots, b_{n-1})$, each sorted in non-decreasing order. The output consists of two arrays $A' = (b'_0, b'_1, \dots, b'_{n-1})$ and $B' = (a'_0, a'_1, \dots, a'_{n-1})$, where b'_i (respectively a'_i) is the element in B (respectively A) such that $b'_i = a_i$ (respectively $a'_i = b_i$), if it exists. We next establish a lower bound for this problem. The proof of the following lemma is similar to that for RANSV and is omitted.

Lemma 3.6 *Let PT_0 and PT_1 be an arbitrary partition of p processors such that $|PT_0| + |PT_1| = \frac{p}{2}$. The problem of solving RANSV1 requires $\Omega(n)$ data exchanges between PT_0 and PT_1 . $\square$*

Corollary 3.8 *Given a convex polygon P with its n sides sorted, solving the ANN problem requires $\Omega(\frac{n\sqrt{\log p}}{p})$ time steps on the weak hypercube and $\Omega(\frac{n \log p}{p})$ time steps on the butterfly, the shuffle-exchange or the cube-connected cycles.*

Proof: We now reduce problem RANSV1 to this problem. Let $A = (a_0, a_1, \dots, a_{n-1})$ and $B = (b_0, b_1, \dots, b_{n-1})$ be two arrays sorted in nondecreasing order. Without loss of generality, we assume that $a_{i+1} - a_i > 1$ and $b_{i+1} - b_i > 1$, $0 \leq i \leq n-2$, and that $a_0 < b_0$, $a_{n-1} > b_{n-1}$ and $a_0 > 0$. We define the following convex polygon P . The upper chain of the polygon is

$$((a_0, 0.5), (a_1, 0.5), \dots, (a_{n-1}, 0.5))$$

and the lower chain is

$$((a_0, 0.5), (b_0, 0), (b_1, 0), \dots, (b_{n-1}, 0), (a_{n-1}, 0.5)).$$

If the nearest point of $(a_i, 0.5)$ is $(b_j, 0)$, for some $0 \leq i, j \leq n-1$, and $a_i \leq b_j$, then a_i and b_j are the matching elements in the arrays A and B . Thus a solution to the ANN problem provides a solution to the RANSV1 problem and the corollary follows. $\square$

We now address the VLSI routing problems considered in the previous section. We start with the river routing problem. The solution to the following

problem was used as the main building block to determine the minimum separation. Let $B = (b_0, b_1, \dots, b_{n-1})$ and $T = (t_0, t_1, \dots, t_{n-1})$ be the input arrays and $S = (t_{j(0)}, t_{j(1)}, \dots, t_{j(n-1)})$ be the output array as before.

Recall that the minimum separation is given by $\max_i \{i - j(i) + 1\} + 1$. Let $t'_j = t_j - j - 1$ and $b'_i = b_i - i$. Then $j(i)$ can be defined as $j(i) = \min_j \{j \leq i \mid t'_j \geq b'_i\}$. This problem is of the same flavor as RANSV. We can use similar techniques to show the following.

Lemma 3.7 *Solving the above problem related to river routing requires $\Omega(\frac{n \log p}{p})$ time on a p -processor butterfly, shuffle-exchange, or cube-connected-cycles, and $\Omega(\frac{n \sqrt{\log p}}{p})$ time on the weak hypercube model. $\square$*

We finally discuss the lower bound proof of the line packing problem. Let n be the number of intervals. As before, we assume that the input and the output are given in arrays $A = (a_0, a_1, \dots, a_{2n-1})$, where a_j is a triple $(x_j, id(j), mark(j))$, and $B = (b_0, b_1, \dots, b_{2n-1})$, respectively. Recall that $x_0 < x_1 < \dots < x_{2n-1}$.

Lemma 3.8 *The line packing problem requires $\Omega(\frac{n \log p}{p})$ time on a p -processor butterfly, shuffle-exchange, or cube-connected-cycles, and $\Omega(\frac{n \sqrt{\log p}}{p})$ on the weak hypercube model.*

Proof: As before let PT_0 and PT_1 be a partition of the processors. There are n left terminals and n right terminals in A . Let $A_0 = (a_0, \dots, a_{n-1})$ and $A_1 = (a_n, \dots, a_{2n-1})$. Let n_0^0 and n_1^0 be the numbers of elements of A_0 that are stored in PT_0 and PT_1 respectively. n_0^1 and n_1^1 can be defined similarly. Note that $n_0^0 = n_1^1$ and $n_0^1 = n_1^0$. Without loss of generality, we assume $n_0^0 \geq n_1^0$ and n_0^0 is an even number. Then, we can construct an instance that will require $\Omega(n)$ data exchanges. Let $A_0^0 = (a_{j(1)}, \dots, a_{j(n_0^0)})$ be the elements of A_0 in PT_0 such that $j(1) < \dots < j(n_0^0)$. A_0^1 , A_1^0 and A_1^1 can be defined similarly. We construct $\frac{n_0^0}{2}$ intervals for A_0^0 as follows: $j(i)$ and $j(\frac{n_0^0}{2} + i)$ define the left and right ends of an interval, for $1 \leq i \leq \frac{n_0^0}{2}$. We construct $\frac{n_1^1}{2}$ intervals for A_1^1 in the same way. We construct n_1^0 intervals by letting the terminals in A_1^1 be left terminals and the terminals in A_0^1 be right terminals, and by connecting the terminals one by one. Clearly this is a valid instance for the problem. Then all the right terminals in A_0^0 have their successor left terminals in PT_1 and the lemma follows. $\square$

Chapter 4

List Ranking and Graph Algorithms

4.1 Introduction

The main goal of this chapter is to develop optimal network algorithms for non-numeric problems such as list processing and graph-theoretic problems; list ranking, tree expression evaluation, connected and biconnected components, ear decomposition and st-numbering. Given a linked list, the list ranking problem is to find the distance from each node to the end of the list. We present an $O(\frac{n}{p})$ time optimal algorithm for the list ranking problem on the pipelined hypercube, when $n = \Omega(p^{1+\epsilon})$ for any positive constant ϵ . This algorithm is used to develop optimal algorithms for all the other problems on our networks. We also proved some lower bounds of the problems on the weak hypercube, the shuffle exchange and the cube-connected cycles. All the algorithms utilize the basic results in the previous chapters.

The rest of this chapter is organized as follows. Section 4.2 deals with the list ranking problem. Several basic graph-theoretic problem are considered in section 4.3.

4.2 List Ranking

Given a linked list $w_0, w_1, \dots, w_{n-1}$ of n items with w_i following w_{i-1} in the list and a binary associative operation $*$, the *parallel prefix* problem is to compute all n initial prefixes $w_0, w_0 * w_1, \dots, w_0 * w_1 * \dots * w_{n-1}$ in parallel. An important special case is the *list ranking* problem in which the distance from each node to the end of the list is to be determined. In this section, an optimal hypercube algorithm for the list ranking problem is presented. This is a fundamental list processing problem which can be used to solve many graph-theoretic problems. The parallel graph algorithms presented in the next section will make a nontrivial

use of the list ranking algorithm presented here. Notice also that an efficient solution to the list ranking problem provides an efficient solution to the parallel prefix problem.

A known approach for solving the list ranking problem in parallel is to perform linked list contraction [2,11,15,16,26,27,42]. This approach relies on an efficient parallel scheme for obtaining a large *independent set* of the linked list. Here, an independent set of a linked list is a set of links such that no two links are incident on the same node. When an independent set is found, the two nodes of every link in the independent set can be combined. In order to obtain an efficient parallel algorithm for the list ranking problem, we need an efficient scheme to obtain a large independent set. There is a basic technique to obtain an independent set in the shared memory model [15,26]. This technique will yield an independent set algorithm whose time complexity is $O(\frac{n}{p} \log^* n)$ on the pipelined hypercube, whenever $n = \Omega(p^{1/c})$ for some positive constant c , by using the columnsort or the cubesort algorithm of section 2.4.

We now describe a simple algorithm to find an independent set with no less than $\frac{n}{4}$ links in time $O(\frac{n}{p})$, when $n = \Omega(p^{1/c})$ for some positive constant c . This algorithm can be implemented on the EREW PRAM in $O(\log n)$ time with $\frac{n}{\log n}$ processors, and can be used to find a list ranking algorithm of $O(\log n \log \log n)$ time with the same number of processors. Let the address (processor id and location within the corresponding memory module) of an arbitrary node be denoted by a . The address of the corresponding successor will be denoted by $\text{succ}(a)$. Define a function f to be $f(a, \text{succ}(a)) = 2k + a_k$, where k is the least significant bit position in which a and $\text{succ}(a)$ differ and a_k is the k -th bit of a . Let $f(a) = f(a, \text{succ}(a))$. Clearly $0 \leq f(a) \leq 2 \log n + 1$. For an integer $h \geq 1$, define $f^h(a) = f(f^{h-1}(a), f^{h-1}(\text{succ}(a)))$. Then $0 \leq f^h(a) \leq c \log^{(h)} n$ and any two links with the same f^h value are not adjacent since any two adjacent links have different values in their k -th bits. Below is our procedure to find a large independent set.

procedure INDEPENDENT SET:

- [M1] Compute f^h for sufficiently large integer constant h . Let the label of element a , $l(a)$, be equal to $f^h(a)$. Then $0 \leq l(a) \leq c \log^{(h)} n$, for some positive constant c .
- [M2] Construct the directed graph $G = (V, E)$ such that $V = \{0, 1, \dots, c \log^{(h)} n\}$ and $(u, v) \in E$, for all $u, v \in V$. A vertex $v \in V$ represents the set of all elements in the original list whose labels are equal to v , and a directed arc $(u, v) \in E$ represents the set of all the links (x, y) in the original list such that $l(x) = u$ and $l(y) = v$. The cost $c(u, v)$ of an arc $(u, v) \in E$ is the number of all the links (x, y) in the original list such that $l(x) = u$ and $l(y) = v$.

- [M3] Find a maximum partition of G . A maximum partition of G is a partition V_1 and V_2 of V such that the sum of the costs of the edges whose end points are in different sets is no less than that of any other partition.
- [M4] Let V_1, V_2 be a maximum partition of V . In the original list, mark all the links (x, y) to be in the independent set if $l(x) \in V_1$ and $l(y) \in V_2$ (or *vice versa*).

We are ready to establish a couple of facts about the above algorithm.

Lemma 4.1 *In step [M2], $c(i, i) = 0$ and $\sum_{i,j} c(i, j) = n - 1$, for each $i, j \in V$.*
□

In the graph G defined by step [M2], let $V' \subset V$ and let $CUT(V', V - V') = \sum_{i \in V', j \in V - V'} (c(i, j) + c(j, i))$. A partition V_1 and V_2 of V is maximal if for any $i \in V_1$, $CUT(V_1, V_2) \geq CUT(V_1 - \{i\}, V_2 \cup \{i\})$ and for any $j \in V_2$, $CUT(V_1, V_2) \geq CUT(V_1 \cup \{j\}, V_2 - \{j\})$.

Lemma 4.2 *Let V_1 and V_2 be a maximal partition of V . Then $CUT(V_1, V_2) \geq \frac{n}{2}$. Thus, the independent set in [M4] has no less than $\frac{n}{4}$ links.*

Proof: For any $i \in V_1$, we have

$\sum_{j \in V_2} (c(i, j) + c(j, i)) \geq \sum_{j \in V_1} (c(i, j) + c(j, i))$, because the partition is maximal. We have a similar inequality for each $i \in V_2$. Thus,

$$\begin{aligned} CUT(V_1, V_2) &= \sum_{i \in V_1, j \in V_2} (c(i, j) + c(j, i)) \\ &\geq \sum_{i, j \in V_1} c(i, j) + \sum_{i, j \in V_2} c(i, j) \\ &= n - 1 - CUT(V_1, V_2), \end{aligned}$$

and $CUT(V_1, V_2) \geq \frac{n-1}{2}$. □

We can easily check that the time needed to execute steps [M1], [M2] and [M4] is $O(\frac{n}{p})$ when $n = \Omega(p^{1+\epsilon})$. Step [M3] can be done by a straightforward exponential algorithm since $|V| = O(\log^{(h)} n)$ and hence $2^{|V|} = O(\log^{(h-1)} p)$. Therefore we have the following.

Lemma 4.3 *When $n = \Omega(p^{1+\epsilon})$, algorithm **INDEPENDENT SET** finds an independent set of no less than $\frac{n}{4}$ links in time $O(\frac{n}{p})$ on the pipelined hypercube.*
□

Corollary 4.1 *When $n = \Omega(p^{1+\epsilon})$, algorithm **INDEPENDENT SET** finds an independent set of no less than $\frac{n}{4}$ links in time $O(\frac{n}{p} \log p)$ on the weak hypercube, the shuffle-exchange and the cube-connected cycles.* □

Corollary 4.2 For any $n \geq p$, algorithm **INDEPENDENT SET** finds an independent set of no less than $\frac{n}{4}$ links in time $O(\frac{n \log n}{p} + \log^2 p)$ on the pipelined hypercube by using the mergesort algorithm in section 2.4. $\square$

We are ready to describe the list ranking algorithm. The well known overall strategy is to identify a large independent set, contract the links in this set, and repeat this process until the length of the list is small enough. For the remaining short list, we can use Wyllie's algorithm [87] which consists of contracting the list $O(\log n)$ times by using the path doubling technique at each iteration. The following procedure describes the overall strategy.

procedure LIST RANKING:

- [P1] Execute steps [P2] - [P4] until the number of remaining nodes is no more than $\frac{n}{\log n}$. $O(\log \log n)$ executions are necessary.
- [P2] Find an independent set with no less than $\frac{n}{4}$ links by using algorithm **INDEPENDENT SET**.
- [P3] Contract all the links in the independent set.
- [P4] The size of the collapsed list is no more than $\frac{3n}{4}$. The remaining links are distributed evenly by using the **BALANCE** algorithm of section 2.3. Successor field of each link is modified such that the redistributed links constitute a linked list.
- [P5] Apply Wyllie's algorithm to find the list ranking value of the remaining list.
- [P6] We restore the linked list and compute the list ranking value of the original list. This step is similar to step [P4].

When $n = \Omega(p^{1+\epsilon})$, single execution of steps [P2], [P3], [P4] and [P6] can be done in time $O(\frac{n}{p})$ on the pipelined hypercube. Thus, the total communication and computation time to execute these steps satisfies the recurrence

$$T_p(n) = T_p\left(\frac{3n}{4}\right) + O\left(\frac{n}{p}\right)$$

and hence $T_p(n) = O(\frac{n}{p})$. Step [P5] can be performed in time $O((\frac{n \log n}{p}) \log n) = O(\frac{n}{p})$. Therefore the overall time complexity is $O(\frac{n}{p})$.

Theorem 4.1 When $n = \Omega(p^{1+\epsilon})$, the list ranking problem can be solved on the pipelined hypercube in time $O(\frac{n}{p})$. $\square$

Corollary 4.3 When $n = \Omega(p^{1+\epsilon})$, the list ranking problem can be solved in time $O(\frac{n}{p} \log p)$ on the weak hypercube, the shuffle-exchange and the cube-connected cycles. $\square$

Corollary 4.4 For any $n \geq p$, the list ranking problem can be solved on the pipelined hypercube in time $O(\frac{n \log n}{p} + \log^3 p)$ by using the mergesort algorithm in section 2.4. $\square$

A natural question is whether the weak hypercube algorithm can be improved. We show next that this is not possible.

Lemma 4.4 The list ranking problem of n links on the weak hypercube requires $\Omega(\frac{n}{p} \log p)$ time, when the links are initially evenly distributed over the p processors.

Proof: Consider a routing problem similar to that of section 2.2 that moves $\frac{n}{p}$ data items from P_i to $P_{E(i)}$, $0 \leq i \leq p-1$, where $E(i) = (i + 0101 \dots 01_2) \bmod p$. This routing problem requires $\Omega(\frac{n \log p}{p})$ time on the weak hypercube. Now we reduce this routing problem to the parallel prefix algorithm. For each data item in P_i , we create a linked list with only one link, beginning at P_i , with the node value being the data value to be moved, and ending in $P_{E(i)}$ with the node value being 0 (the identity for operator $*$). Clearly a solution to this parallel prefix problem will solve the given routing problem and hence the lemma follows. $\square$

We can also prove the same lower bound of the problem on the shuffle-exchange and the cube-connected cycles in a similar way.

4.3 Graph Problems

In this section, we describe parallel algorithms for several well-known graph problems. The problems include tree expression evaluation, Euler tour on trees, finding lowest common ancestors, connectivity, biconnectivity, strong orientation, ear decomposition, Euler tour on graphs, graph coloring and finding maximal independent sets (see Tables 4.1 and 4.2). These algorithms utilize the algorithms for load balancing, integer sorting and list ranking of sections 2.3, and 2.4 and 4.2, respectively. We start by discussing those problems for which efficient algorithms are developed. These include computing various tree functions and planar graph problems. A common strategy that works well for all these problems consists of reducing the size of the problem by using an efficient hypercube algorithm and then applying a fast shared memory algorithm on the reduced size problem.

Theorem 4.2 *The following problems can be solved in time $O(\frac{n}{p})$ on the pipelined hypercube, when $n = \Omega(p^{1+\epsilon})$.*

1. *Tree expression evaluation.*
2. *Euler tours on trees, and hence all the basic tree functions.*
3. *Maximal independent set, connected components, biconnected components, strong orientation, ear decomposition, st-numbering, and Euler tour of planar graphs.*

Proof Sketch: All the algorithms used have been reported in the literature. See Tables 4.1 and 4.2 for appropriate references. However we use our routing and list ranking algorithms to achieve the time bound stated in the theorem. $\square$

Corollary 4.5 *The problems in the above theorem can be solved in time $O(\frac{n}{p} \log p)$ on the weak hypercube, the shuffle-exchange and the cube-connected cycles, when $n = \Omega(p^{1+\epsilon})$.*

Corollary 4.6 *The problems in the above theorem can be solved in time $O(\frac{n \log n}{p} + \log^3 p)$ on the pipelined hypercube, for any $n \geq p$.*

We now consider the important problem of finding the connected components of arbitrary graphs. There are two well-known shared memory algorithms: An $O(\frac{n^2}{p} + \log^2 n)$ algorithm when the input is given as an adjacency matrix [11,84], and an $O(\frac{m+n}{p} \log n)$ algorithm when the input is given as an edge list [74] with n vertices and m edges. We call the latter one the "SV-algorithm".

It can be easily verified that the above two algorithms can be implemented directly on the pipelined hypercube in time $O(\frac{n^2}{p})$ and $O(\frac{n+m}{p} \log n)$ when $n^2 \geq p^{1+\epsilon}$ and $n + m \geq p^{1+\epsilon}$, respectively, by using our previous algorithms.

However, if the adjacency matrix of a graph is given as input, there is an efficient algorithm even for the weak hypercube, the shuffle-exchange and the cube-connected cycles [1].

Lemma 4.5 *Given the adjacency matrix of a graph, the connected components can be found in time $O(\frac{n^2}{p})$ on the weak hypercube, the shuffle-exchange and the cube-connected cycles, when $p \leq \frac{n^2}{\log^2 n}$. $\square$*

We now consider the case of sparse graphs. A faster algorithm is known and is efficient for all graphs except those which are extremely sparse [17]. Using the strategy of [17], we derive a simple algorithm for finding the connected components of a graph in time $O(\frac{n+m}{p} \log \log n)$ on the pipelined hypercube, where the input is given as an edge list and $n + m \geq p^{1+\epsilon}$. Since we will be using the SV-algorithm in our description, we present a brief outline of the main strategy.

The output of the algorithm is a vector $D[1 \dots n]$ such that $D(u) = D(v)$ if and only if vertex u and v are in the same connected component. Initially, $D(v) = v$, for each $v = 1, \dots, n$. Notice that the vector D forms a forest during the execution of the algorithm. The algorithm performs the following steps $O(\log n)$ iterations:

1. Shortcutting. Set $D(i) = D(D(i))$, $1 \leq i \leq n$.
2. Hooking trees onto smaller vertices of other trees. If $D(i)$ did not change in step 1, then if there exists j such that (i, j) is an edge of G and $D(j) < D(i)$ then set $D(D(i)) = D(j)$.
3. If there is a tree that did not change in steps 1 and 2, then hook such a tree onto another tree if possible.

We now begin the description of our algorithm. At each step of the algorithm, a set of vertices with the same D value is referred to as a *supervertex*. Each edge (u, v) in the input graph induces an edge connecting the supervertex containing u with the supervertex containing v . The graph whose vertices are the supervertices and the edges are these induced edges is called the *supervertex graph*. Given a supervertex graph, an edge in G is *redundant* (with respect to the supervertex graph) if both of its endpoints lie in the same supervertex. An edge is an *outedge* if it is not redundant. If several outedges connect the same pair of supervertices, one of these outedges is chosen to be the *actual* outedge; the other outedges are called *duplicate* outedges. The rule for choosing the actual edge is arbitrary. The *degree* of a supervertex v , $degree(v)$, is defined to be the number of actual outedges incident on v . Initially, the input graph G is the supervertex graph in which V is the set of supervertices and E is the set of actual outedges.

procedure CONNECTIVITY:

- [C1] Execute steps [C2] - [C5] until the number of remaining vertices and edges are no more than $\frac{n}{\log n}$ and $\frac{m}{\log n}$, respectively. For the i -th iteration, let n_i be the number of vertices in the non-isolated supervertex containing the fewest number of vertices. Then we set $d = \sqrt{n_i}$.
- [C2] For each supervertex v , select exactly d actual outedges of v if $degree(v) \geq d$ or select all its actual outedges, otherwise.
- [C3] Run the SV algorithm $\lceil \log_{\frac{3}{2}} \frac{d}{2} \rceil + 1$ iterations on the graph induced by the edges selected in step [C2].
- [C4] The output of step [C3] is a rooted forest of its supervertices. Contract this rooted forest into rooted stars (rooted trees of height 1).
- [C5] Construct the new supervertex graph for the next iteration. To do this, modify vector D and delete all redundant edges, duplicate outedges and isolated supervertices.

[C6] For the supervertex graph with vertices and edges no more than $\frac{n}{\log n}$ and $\frac{m}{\log n}$, respectively, run the SV algorithm to find its connected components. Finally, modify vector D to represent the connectivity of the input graph G .

Lemma 4.6 *For each iteration, step [C3] can be executed in time $O(\frac{m+n}{p})$ on the pipelined hypercube when $m + n = \Omega(p^{1+\epsilon})$.*

Proof: The number of edges of the i -th iteration is $O(\frac{n}{n_i} \cdot d) = O(\frac{n}{d})$. Since the number of edges for the SV algorithm is $O(\frac{n+m}{\log d+1})$, the time for step [C3] is $O(\frac{n+m}{\log d} \cdot (\log_{\frac{3}{2}} \frac{d}{2} + 1) \cdot \frac{1}{p}) = O(\frac{n+m}{p})$. $\square$

Lemma 4.7 *After $O(\log \log n)$ iterations of steps [C2] - [C5], there is no non-isolated supervertex.*

Proof: In the graph comprising the supervertices and the edges selected in step [C2], there are two kinds of components:

1. Components with more than d supervertices.
2. Components with no more than d supervertices.

Supervertices of the components of type 2 belong to the same rooted tree after step [C3] and removed as an isolated supervertex in step [C5]. Supervertices of the components of type 1 are partitioned into one or more rooted forest each containing at least d supervertices after step [C3]. Thus, $n_{i+1} \geq n_i \cdot d = n_i^{\frac{3}{2}}$. Since $n_i \geq n_2^{(\frac{3}{2})^{i-2}} \geq 2^{(\frac{3}{2})^{i-2}}$, after $\frac{\log \log n}{\log \frac{3}{2}} + 2$ iterations, there can be no non-isolated supervertex. $\square$

Theorem 4.3 *When $m + n = \Omega(p^{1+\epsilon})$, the connected components of a graph can be found in time $O(\frac{n+m}{p} \log \log n)$ on the pipelined hypercube.*

Proof: Steps [C2] and [C5] can be done in time $O(\frac{n+m}{p})$ by using the routing algorithm of section 2.5. Step [C4] can be done within the same time bound with the Euler tour technique and list ranking. Step [C6] can be done in time $O(\frac{n+m}{p \log n} \cdot \log n) = O(\frac{n+m}{p})$. Thus the total time for this algorithm is $O(\frac{n+m}{p} \log \log n)$. $\square$

Corollary 4.7 *When $m + n = \Omega(p^{1+\epsilon})$, the connected components of a graph can be found in time $O(\frac{(n+m) \log p}{p} \log \log n)$ on the weak hypercube, the shuffle-exchange and the cube-connected cycles.*

Corollary 4.8 *For any $m, n \geq p$, the connected components of a graph can be found in time $O((\frac{(n+m) \log(n+m)}{p} + \log^3 p) \log \log n)$ on the pipelined hypercube.*

One can easily show that the following graph problems can be solved within the same time bounds on the hypercube as the connected components problem (for both forms of the input): biconnected components, strong orientation, ear decomposition, and st-numbering.

We now prove a lower bound result for the weak hypercube, the shuffle-exchange and the cube-connected cycles. The lower bound shown for all the above graph problems is $\Omega(\frac{n}{p} \log p)$. We also present the result that shows that it is possible to find the connected components faster than $O(\frac{m+n}{p} \log p)$ in some cases on the weak hypercube.

Lemma 4.8 *Solving any graph problem mentioned in this section requires $\Omega(\frac{n}{p} \log p)$ time on the weak hypercube, the shuffle-exchange and the cube-connected cycles.*

Proof: We prove this theorem only for the graph connectivity problem. The proofs for the other problems are similar. Assume with out loss of generality that $p = 2^d = 2^{3k}$. Let $E_1(i) = (i + 001001 \dots 001_2) \bmod p$, $E_2(i) = (i + 010010 \dots 010_2) \bmod p$ and $E_3(i) = (i + 011011 \dots 011_2) \bmod p$, $0 \leq i \leq p-1$. Then we can prove by induction that the Hamming distance of any two of $\{i, E_1(i), E_2(i), E_3(i)\}$ is $\Omega(\log p)$. Consider the following instance: $n = m = 4l$ for some integer l and each connected component is a cycle $(v_{j_0}, v_{j_1}, v_{j_2}, v_{j_3})$ of length 4, $0 \leq j \leq l-1$. If we assume that the two edges incident on v_{j_0} are in P_i and those incident on v_{j_1} in $P_{E_1(i)}$ and so on, then the time it takes to label the vertices of each connected component with the same label is $\Omega(\log p)$ and the total time is therefore $\Omega(\frac{n \log p}{p})$. The lower bound for the shuffle-exchange and the cube-connected cycles can be proved similarly. $\square$

Lemma 4.9 *The connected components of a graph $G = (V, E)$, where $|V| = n$ and $|E| = m$, can be obtained in $O((k\frac{m}{p} + (k+1)\frac{n}{\log^k p}) \log \log n \log \log p)$, for any positive integer constant k , when $m = \Omega(p \log p)$, on the weak hypercube, the shuffle-exchange and the cube-connected cycles. When $n \leq \frac{m \log^k p}{p}$, the time complexity is $O(k\frac{m}{p} \log \log n \log \log p)$.*

Proof: The connected components of a graph G can be obtained in $O(\frac{m}{p} \log \log n \log \log p)$ on the weak hypercube, the shuffle-exchange and the cube-connected cycles if $m \geq p^{1+\epsilon}$. With this fact, the following algorithm finds the connected components within the time bound claimed in the statement of the lemma.

- (1) For each subcube of size $\log^k p$, find the connected components. Then each processor has to hold only $O(\frac{n}{\log^k p})$ edges. This step takes $O(k\frac{m}{p} \log \log n \log \log p)$ time.

- (2) For each subcube of size $\log^{k+1} p$, find the corresponding connected components. Then each processor has to hold $O(\frac{n}{\log^{k+1} p})$ edges. The execution time of this step is $O((k+1)\frac{n}{\log^k p} \log \log n \log \log p)$.
- (3) Find the connected components on all processors. This step takes $O(\frac{n}{\log^k p} \log \log n)$ time. $\square$

For example, if $p = \sqrt{n}$ and $m \geq \frac{n\sqrt{n}}{\log n}$, then the running time of the above algorithm is $O(\frac{m}{p}(\log \log n)^2)$.

PROBLEMS	TIME	# of PRS's	REFER
radix sort range $\{1, \dots, p^{O(1)}\}$	$O(\frac{n}{p})$	$n \geq p^{1+\epsilon}$	[64]
list ranking	$O(\frac{n}{p})$	$n \geq p^{1+\epsilon}$	[64]
tree expression evaluation	$O(\frac{n}{p})$	$n \geq p^{1+\epsilon}$	[40]
Euler tour on trees	$O(\frac{n}{p})$	$n \geq p^{1+\epsilon}$	[77]
finding lowest common ancestors			
preprocessing	$O(\frac{n}{p})$	$n \geq p^{1+\epsilon}$	[68]
processing k queries	$O(\frac{n+k}{p})$	$n \geq p^{1+\epsilon}$	[68]
connectivity, (minimum) spanning tree	$O(\frac{n+m}{p} \log \log n)$ $O(\frac{n^2}{p})$	$n + m \geq p^{1+\epsilon}$ $p \leq \frac{n^2}{\log^2 n}$	[17,64] [1]
planar graph	$O(\frac{n}{p})$	$n \geq p^{1+\epsilon}$	[25]
biconnectivity	$O(\frac{n+m}{p} \log \log n)$ $O(\frac{n^2}{p} + \frac{m}{p} \log p)$ $O(\frac{n^2}{p})$	$n + m \geq p^{1+\epsilon}$ $p \leq \frac{n^2}{\log^2 n}$ $n^2 \geq p^{1+\epsilon}$	[64,77] [1,77] [77,84]
planar graph	$O(\frac{n}{p})$	$n \geq p^{1+\epsilon}$	[25,77]
strong orientation	$O(\frac{n+m}{p} \log \log n)$ $O(\frac{n^2}{p} + \frac{m}{p} \log p)$ $O(\frac{n^2}{p})$	$n + m \geq p^{1+\epsilon}$ $p \leq \frac{n^2}{\log^2 n}$ $n^2 \geq p^{1+\epsilon}$	[64,85] [1,85] [84,85]
planar graph	$O(\frac{n}{p})$	$n \geq p^{1+\epsilon}$	[25,85]
ear decomposition, st-numbering	$O(\frac{n+m}{p} \log \log n)$ $O(\frac{n^2}{p} + \frac{m}{p} \log p)$ $O(\frac{n^2}{p})$	$n + m \geq p^{1+\epsilon}$ $p \leq \frac{n^2}{\log^2 n}$ $n^2 \geq p^{1+\epsilon}$	[52,64] [1,52] [52,84]
planar graph	$O(\frac{n}{p})$	$n \geq p^{1+\epsilon}$	[25,52]
Euler tour on graphs	$O(\frac{n+m}{p} \log \log n)$	$n + m \geq p^{1+\epsilon}$	[4,64]
planar graph	$O(\frac{n}{p})$	$n \geq p^{1+\epsilon}$	[4,25]
2 (3) coloring (pseudo) forest	$O(\frac{n}{p})$	$n \geq p^{1+\epsilon}$	[22]
7 coloring planar graphs	$O(\frac{n}{p})$	$n \geq p^{1+\epsilon}$	[22]
finding maximal independent sets on planar graphs	$O(\frac{n}{p})$	$n \geq p^{1+\epsilon}$	[22]

Table 4.1: Performance on the pipelined hypercube (ϵ : constant).

PROBLEMS	TIME	# of PRS's	REFER
radix sort range $\{1, \dots, p^{O(1)}\}$	$O(\frac{n}{p} \log p)$	$n \geq p^{1+\epsilon}$	[64]
list ranking	$O(\frac{n}{p} \log p)$	$n \geq p^{1+\epsilon}$	[64]
tree expression evaluation	$O(\frac{n}{p} \log p)$	$n \geq p^{1+\epsilon}$	[40]
Euler tour on trees	$O(\frac{n}{p} \log p)$	$n \geq p^{1+\epsilon}$	[77]
finding lowest common ancestors			
preprocessing	$O(\frac{n}{p} \log p)$	$n \geq p^{1+\epsilon}$	[68]
processing k queries	$O(\frac{n+k}{p} \log p)$	$n \geq p^{1+\epsilon}$	[68]
connectivity, (minimum)	$O(\frac{n+m}{p} \log \log n \log p)$	$n + m \geq p^{1+\epsilon}$	[17,64]
spanning tree	$O(\frac{n^2}{p})$	$p \leq \frac{n^2}{\log^2 n}$	[1]
planar graph	$O(\frac{n}{p} \log p)$	$n \geq p^{1+\epsilon}$	[25]
biconnectivity	$O(\frac{n+m}{p} \log \log n \log p)$ $O(\frac{n^2}{p} + \frac{m}{p} \log^2 p)$	$n + m \geq p^{1+\epsilon}$ $p \leq \frac{n^2}{\log^2 n}$	[64,77] [1,77]
planar graph	$O(\frac{n^2}{p} + \frac{m}{p} \log p)$ $O(\frac{n}{p} \log p)$	$n^2 \geq p^{1+\epsilon}$ $n \geq p^{1+\epsilon}$	[77,84] [25,77]
strong orientation	$O(\frac{n+m}{p} \log \log n \log p)$ $O(\frac{n^2}{p} + \frac{m}{p} \log^2 p)$	$n + m \geq p^{1+\epsilon}$ $p \leq \frac{n^2}{\log^2 n}$	[64,85] [1,85]
planar graph	$O(\frac{n^2}{p} + \frac{m}{p} \log p)$ $O(\frac{n}{p} \log p)$	$n^2 \geq p^{1+\epsilon}$ $n \geq p^{1+\epsilon}$	[84,85] [25,85]
ear decomposition, st-numbering	$O(\frac{n+m}{p} \log \log n \log p)$ $O(\frac{n^2}{p} + \frac{m}{p} \log^2 p)$	$n + m \geq p^{1+\epsilon}$ $p \leq \frac{n^2}{\log^2 n}$	[52,64] [1,52]
planar graph	$O(\frac{n^2}{p} + \frac{m}{p} \log p)$ $O(\frac{n}{p} \log p)$	$n^2 \geq p^{1+\epsilon}$ $n \geq p^{1+\epsilon}$	[52,84] [25,52]
Euler tour on graphs	$O(\frac{n+m}{p} \log \log n \log p)$	$n + m \geq p^{1+\epsilon}$	[4,64]
planar graph	$O(\frac{n}{p} \log p)$	$n \geq p^{1+\epsilon}$	[4,25]
2 (3) coloring (pseudo) forest	$O(\frac{n}{p} \log p)$	$n \geq p^{1+\epsilon}$	[22]
7 coloring planar graphs	$O(\frac{n}{p} \log p)$	$n \geq p^{1+\epsilon}$	[22]
finding maximal independent sets on planar graphs	$O(\frac{n}{p} \log p)$	$n \geq p^{1+\epsilon}$	[22]

Table 4.2: Performance on the weak hypercube (ϵ : constant).

Chapter 5

One-Layer Routing

5.1 Introduction

It is well-known that many of the optimization problems arising in VLSI routing are NP-complete [41,46,67,76]. One notable exception is the class of *one-layer routing* problems associated with a hierarchical layout strategy such as Bristle-Blocks [36]. See [13,20,48,49,51,53,59,72,79] for more examples. Efficient serial solutions have already appeared in the literature for most of these problems. For parallel solutions, efficient algorithms that run in time $O(\log n)$ on CREW PRAM and in time $O(\frac{\log n}{\log \log n})$ on Common CRCW PRAM were developed for several one-layer routing problems [10]. In this chapter, fast parallel algorithms for the one-layer routing problems on the hypercube, the shuffle-exchange, the cube-connected cycles and the butterfly are presented.

The class of general one-layer routing problems involves routing between ordered sequences of terminals such that the final layout is planar. One such problem (river routing) is the wiring of two ordered sets of terminals $\{b_1, b_2, \dots, b_n\}$ and $\{t_1, t_2, \dots, t_n\}$ across a channel between the parallel boundaries of two rectangles. The width of the channel is the vertical distance between the two lines forming the channel. The *separation problem* is to find the minimum width of the channel necessary to wire all nets such that any two wires are separated by a unit distance. We will restrict ourselves to the case where the wires are rectilinear, i.e., there is a grid structure such that each wire consists of a connected set of grid line segments. This problem can be solved in time $O(\frac{n}{p} + \log p)$ on the pipelined hypercube.

A more general version of the river routing problem that is known to have an efficient serial algorithm is to perform planar routing where the ports lie on the boundary of a simple rectilinear polygon [59]. In this case, we are interested in whether the routing is possible or not and, in the affirmative, we have to provide the detailed routing. Several interesting subproblems such as finding the contour of the union of a set of rectilinear polygons or determining whether a set of nets

can be wired within a set of "passages" are also tackled. All these problem can be solved in time $O(\frac{n}{p}l^3)$ on the pipelined hypercube, when $n = p^{1+\frac{1}{l}}$, for any $l > 0$. These algorithms can be also implemented in time $O(\frac{n \log n}{p} + \log^2 p)$ on the pipelined hypercube.

The rest of this chapter is organized as follows. Section 5.2 deals with the separation problem. Sections 5.3 and 5.4 deal with the routing problem and the routability testing problem within a rectilinear polygon.

5.2 The Separation Problem

Let $\{N_i = \langle b_i, t_i \rangle \mid 1 \leq i \leq n\}$ be an instance of the channel separation problem. Notice that b_i and t_i will be also used to denote the horizontal coordinates of the terminals relative to an arbitrary origin. We make the reasonable assumption that all terminals lie in an interval $[0, N]$, where $N = O(n)$.

A net N_i is a *right* net if $b_i < t_i$, it is a *left* net if $b_i > t_i$, and it is a *vertical* net otherwise. We can partition the nets into *right* blocks, *left* blocks and *vertical* blocks. A set of right nets $N_i, N_{i+1}, \dots, N_k$ is a right block if it is maximal with the property $b_j < b_{j+1} \leq t_j$, for $i \leq j < k$. We can similarly define left blocks and vertical blocks.

The wiring problem is reduced to wiring each block separately. We will concentrate on the wiring of right blocks. Obvious changes can be made to deduce the corresponding algorithm for left blocks. An efficient strategy for right blocks consists of wiring the nets from left to right such that for each net we move from the bottom terminal upward and try to stay as close to the upper row as possible [20,59,72]. The wiring of a net can be specified by the coordinates of its bend points. For example, net N_1 of Figure 5.1 has the bend points A_{11}, B_{11} . For each net N_i , we have $2k$ bend points, $A_{i1}, A_{i2}, \dots, A_{ik}$ and $B_{i1}, B_{i2}, \dots, B_{ik}$, for some k . Not all of these bend points are needed to determine the overall wiring. We call A_{i1} and B_{i1} (bend points closest to the bottom row) the *characteristic* bend points of N_i . Notice that the characteristic bend points uniquely define the overall wiring since once we have the wiring of N_{i-1} and the characteristic bend points A_{i1} and B_{i1} of N_i , we can easily determine all the other bend points of N_i . Figure 5.1 shows an example of a river routing problem and a wiring achieving the minimum separation. The algorithm to find the minimum separation is based on the following lemma.

Lemma 5.1 *Let N_i be a net in a right block and let $\hat{j}$ be the minimum $j \leq i$ such that $t_j + (i - j - 1) \geq b_i$. Then the coordinates of the characteristic bend points of N_i are $A_{i1} = (b_i, i - \hat{j} + 1)$ and $B_{i1} = (t_j + i - \hat{j}, i - \hat{j} + 1)$.*

Proof. Since $\hat{j}$ is the minimum $j \leq i$ such that $t_j + (i - j - 1) \geq b_i$, there is no terminal point at $(t_j - 1, 0)$. Hence there is a bend point at $(t_j, 1)$ for net N_j .

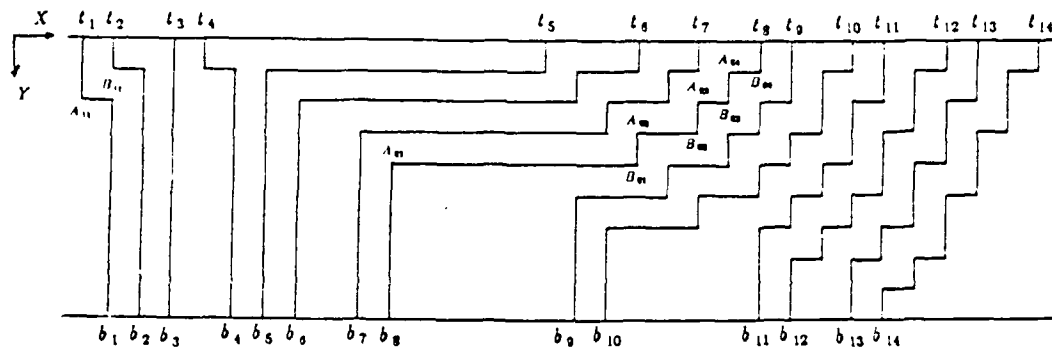


Figure 5.1: Basic river routing problem

The number of vertical grid lines between t_j and b_i is $b_i - t_j$ and hence smaller than the number of nets between N_j and N_i , i.e., $i - j + 1$ horizontal tracks are needed to route net N_i . A simple argument will show that the coordinates of the characteristic bend points have the values stated in the lemma. $\square$

The following procedure computes such an index $\hat{j}(i)$ for each net N_i of a right block $\{N_i | 1 \leq i \leq n\}$.

procedure Index

1. Compute $b'_i = b_i - i$ and $t'_k = t_k - k - 1$ for each i and k . Notice that $b'_1 \leq b'_2 \leq \dots \leq b'_n$ and $t'_1 \leq t'_2 \leq \dots \leq t'_n$.
2. Merge the two sequences. If $b'_i = t'_k$ then put b'_i before t'_k in the merged sequence.
3. For each b'_i , find the nearest t'_k to the right in the merged sequence. Then $\hat{j}(i) = k$.

The correctness proof of the above algorithm is straightforward. We can use the merge algorithm in subsection 2.4.1 for step 2. Step 3 can be done by performing the prefix sums operation. Thus the above algorithm can be implemented in time $O(\frac{n}{p} + \log p)$ on the pipelined hypercube. We now give

the algorithm to find the minimum separation as well as the characteristic bend points of all the nets.

procedure Separation

1. Partition the nets into blocks.
2. Apply algorithm **Index** to get the index $\hat{j}(i)$ for each net N_i . Use Lemma 5.1 to obtain all the characteristic bend points.
3. Let the characteristic bend points be $B_{i1} = (x_{i1}, y_{i1})$, $1 \leq i \leq n$. Then the minimum separation is $\max\{y_{11}, \dots, y_{n1}\} + 1$.

Theorem 5.1 *Algorithm Separation finds the characteristic bend points of the n input nets and the minimum channel separation in time $O(\frac{n}{p} + \log p)$ on the pipelined hypercube. $\square$*

Corollary 5.1 *Algorithm Separation finds the characteristic bend points of the n input nets and the minimum channel separation in time $O(\frac{n \log p}{p})$ on the weak hypercube, the shuffle-exchange, the cube-connected cycles and the butterfly. $\square$*

5.3 Routing In a Simple Polygon

The routing problem of nets within a simple rectilinear polygon introduced in [59] is a generalization of the standard river routing problem. In this case we are supposed to connect a set of terminals $\{a_1, a_2, \dots, a_n\}$ on the boundary of a simple rectilinear polygon to another set of terminals $\{b_1, b_2, \dots, b_n\}$ on the boundary of the same polygon such that all the wires lie within the polygon and no two wires intersect. *Routability testing* is to determine whether or not a one layer routing is possible, and *detailed routing* is to specify the actual wiring of the n nets, if they are routable. We will start by discussing a version of the detailed routing problem whose solution will be used in the routability testing algorithm. Routability testing will be discussed in the next section. We will restrict ourselves to the rectangle case. However all the algorithms can be generalized to any rectilinear polygon. We assume that the x and y coordinates of the terminals are integers which lie in an interval $[0, N]$, where $N = O(n)$.

We will begin with a few definitions. Let $\{N_i = \langle a_i, b_i \rangle \mid 1 \leq i \leq n\}$ be the set of n input nets whose terminals lie on the boundary of a rectangle R . Let the lower left corner of R be $(0,0)$, the origin of an (x, y) coordinate system. The four corners of R have coordinates $(0,0)$, $(l,0)$, (l,h) and $(0,h)$, where l and h are respectively the length and the height of R . If we cut R at $(0,0)$ and straighten the boundary counterclockwise into a line, the corresponding linear

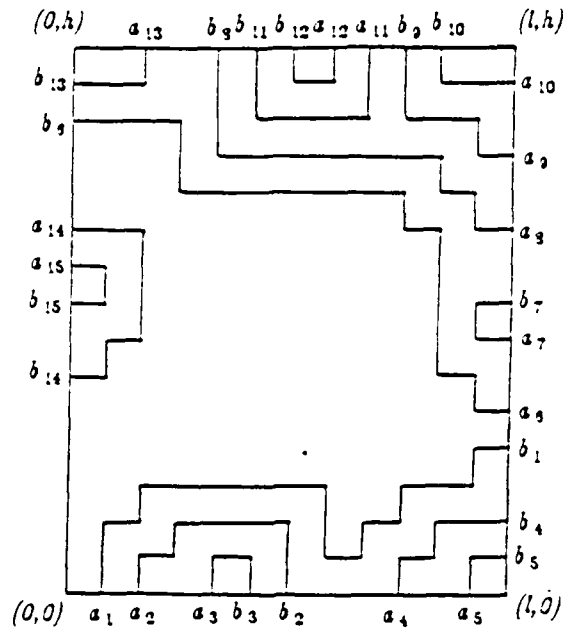


Figure 5.2: Basic river routing around a rectangle boundary

coordinate of a point w on the boundary will be denoted by $d(w)$. It is trivial to compute $d(w)$ from the two-dimensional coordinate of w .

Let $N_i = \langle a_i, b_i \rangle$ be an arbitrary net. The terminals a_i and b_i divide the boundary of R into two parts. The part of length $\leq h + l$ will be called the *internal boundary* of N_i . The other part will be called the *external boundary*. We assume without loss of generality that the internal boundary of N_i begins with a_i and ends with b_i counterclockwise. We call a_i and b_i the left terminal and the right terminal of N_i , respectively. Without loss of generality, we can also assume that $d(a_1) < d(a_2) < \dots < d(a_n)$.

A net N_i covers another net N_j if the internal boundary of N_i properly contains that of N_j . A *representative net* is a net that is not covered by any other net. Figure 5.2 shows an example of a detailed routing problem such that N_1 , N_6 and N_{14} are the representative nets. We can partition the nets into *groups* such that each group consists of a representative net and all the nets covered by it. Notice that the nets in each group appear consecutively in a circular fashion in R . The groups in Figure 5.2 are $\{N_1, N_2, N_3, N_4, N_5\}$, $\{N_6, N_7, N_8, N_9, N_{10}, N_{11}, N_{12}, N_{13}\}$, and $\{N_{14}, N_{15}\}$.

Clearly, if a given instance of the above problem is routable, then its routing can be performed by routing each group of nets separately. Thus, the general strategy for specifying the routing will be the following: (i) identify the representative nets, (ii) partition the nets into groups of nets, and (iii) specify the routing of the representative net of each group. The following algorithm handles

(i) and (ii).

procedure Representative Nets

1. Mark all the nets whose internal boundaries contain the corner (0,0). If no such net exists, go to step 3.
2. Let $N_{i_1}, N_{i_2}, \dots, N_{i_k}$ be all the nets that are marked and $d(a_{i_1}) < d(a_{i_2}) < \dots < d(a_{i_k})$. Then, clearly N_{i_1} is a representative net and it covers all the other $k - 1$ nets. Find all the nets covered by N_{i_1} and remove them with N_{i_1} from the input.
3. Sort all the remaining terminals according to their d-values. This step can be done by using the cubesort or the mergesort algorithm in chapter 2.
4. Assign +1 to the left terminal and -1 to the right terminal of each net and compute prefix sums of all the terminals. Clearly, N_i is a representative net if and only if the prefix sum value of a_i is 1.
5. For each representative net, find all the nets that it covers. Notice that if N_i and N_j are two adjacent representative nets such that $d(a_i) < d(a_j)$, then nets $N_{i+1}, \dots, N_{j-1}$ are covered by N_i .

Lemma 5.2 *Let n be the number of input nets. The representative nets and the corresponding groups can be found in time $O(\frac{n}{p}l^3)$ on the pipelined hypercube, when $n = p^{1+\frac{1}{l}}$, for any $l > 0$. $\square$*

We now turn to the problem that routes each group separately. Our goal here is to identify the bend points of each representative net. Note that in general the total number of bend points of all the nets could be $\Omega(n^2)$. However the total number of bend points of the representative nets is always $O(n)$.

Lemma 5.3 *Let $N_{r_1}, N_{r_2}, \dots, N_{r_k}$ be all the representative nets and let $I(N_{r_i})$ be the number of nets in the internal boundary of N_{r_i} . Then $\sum_{i=1}^k (I(N_{r_i}) + 1) = n$. Moreover, there exists a wiring strategy such that N_{r_i} has at most $4(I(N_{r_i}) + 1)$ bend points. Thus, the total number of bend points of all the representative nets is $O(n)$. $\square$*

Without loss of generality, we can assume that there is no net whose internal boundary contains the corner (0,0). If there is such a net, then we can consider the group consisting of such nets separately. The overall strategy for specifying the routing of the representative nets is as follows: (i) Unfold R into the line L of length $2l + 2h$ by cutting at (0,0), (ii) specify the routing of the representative nets on L , (iii) restore R by cutting and folding L at the corners, and (iv) remove

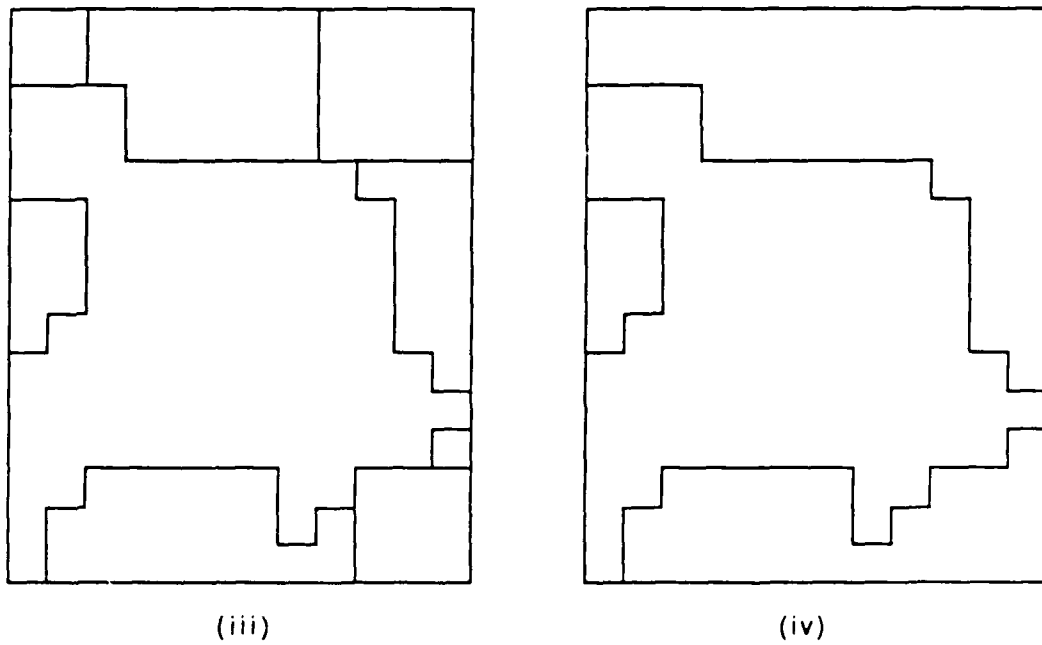
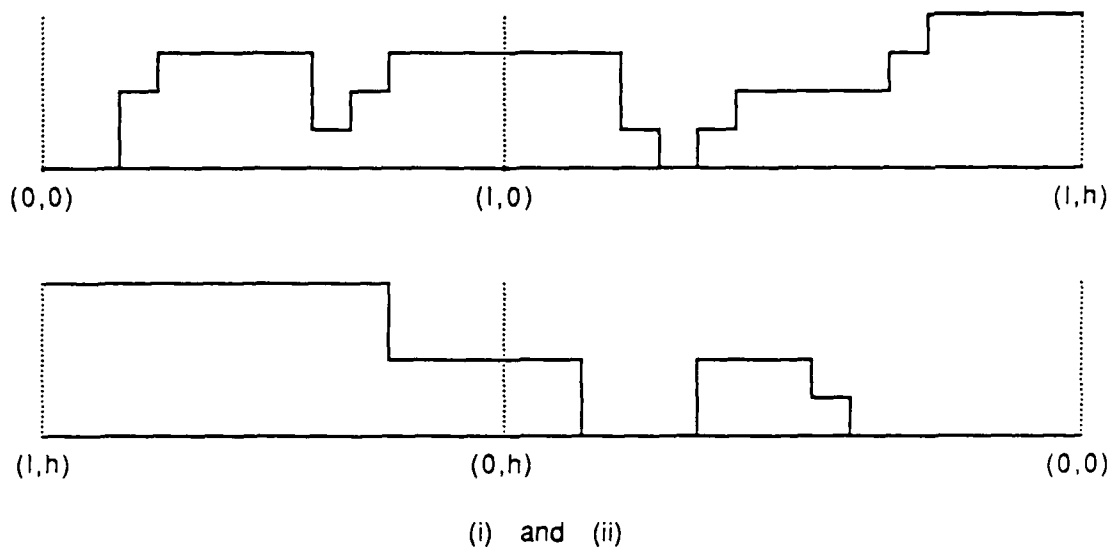


Figure 5.3: The step-by-step illustration of the overall strategy of routing representative nets

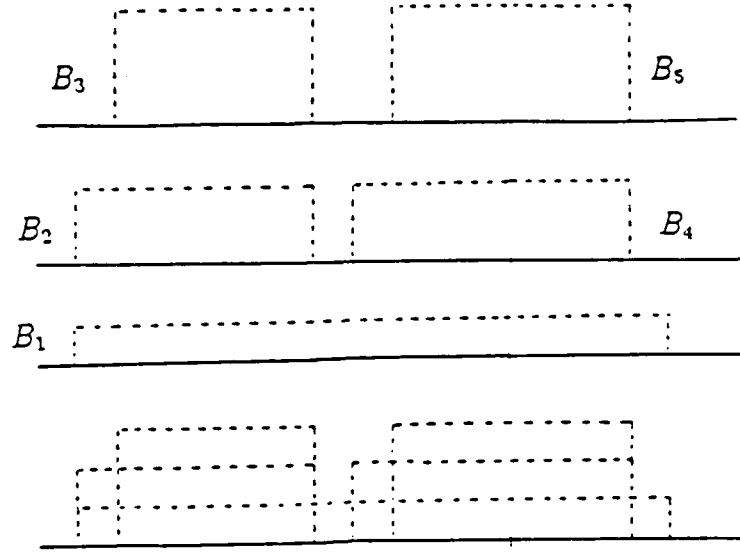


Figure 5.4: The union of the bounding perimeters. B_i is the bounding perimeter of N_i

unnecessary line segments. The step-by-step illustration of this strategy applied to the instance of Figure 5.2 is given in Figure 5.3.

For each net $N_i = \langle a_i, b_i \rangle$ on R , let $N'_i = \langle d(a_i), d(b_i) \rangle = \langle a'_i, b'_i \rangle$ be the corresponding net on L . Let the rank of net N'_i , $\text{rank}(N'_i)$, be the number of nets that cover it. Then the *bounding perimeter* of N'_i is the region that starts at $a'_i - k$ and ends at $b'_i + k$, and whose height is $k + 1$, where $k = \text{rank}(N'_i)$. Notice that the wiring of the representative net of N'_i can not intersect the inside of the bounding perimeter. Figure 5.4 shows the contour of nets N'_1, N'_2, N'_3, N'_4 and N'_5 of Figure 5.2. We claim the following.

Lemma 5.4 *The union of all the bounding perimeters of all the nets within a group determines the contour of the group and hence determines the wiring of the representative net.*

Proof: Notice that no portion of the wiring of the representative net can lie inside any of the bounding perimeters. Therefore it has to lie either on the contour of the union of the bounding perimeters or it will have some portions outside the union. We will show that the region determined by the union of the bounding perimeters is large enough for routing all the nets in the group.

The proof is by induction on the number n of the nets. The case of $n = 1$ is trivial. Suppose a group G has $n + 1$ nets. If we remove the representative net N'_r from G , we may obtain several groups $\{G_i\}$. By the induction hypothesis, the

union of the bounding perimeters of each group determines the corresponding contour. Since the rank of each net will increase by 1 if we put N'_r back, the union of bounding perimeters in G will stretch by a distance of 1 which is just enough to wire N'_r . $\square$

It is clear that the rank of each net can be determined by a similar strategy as in steps 3 and 4 of procedure **Representative Nets**. Moreover it is easy to specify the bounding perimeter of each net once its rank is determined. We now discuss the problem of determining the contours of groups of nets from the corresponding bounding perimeters. Let two nets $N'_i = \langle a'_i, b'_i \rangle$ and $N'_j = \langle a'_j, b'_j \rangle$ be such that $\text{rank}(N'_i) = \text{rank}(N'_j) = k$ and $b'_i < a'_j$. Then the bounding perimeters of N'_i and N'_j overlap if and only if $a'_j - b'_i \leq 2k$. Our initial task is to combine overlapping bounding perimeters of the same height.

Lemma 5.5 *Let N'_j be the nearest net to the right of N'_i such that $\text{rank}(N'_i) = \text{rank}(N'_j) = k$. Then,*

1. *the rank of all the nets whose terminals are between b'_i and a'_j are less than k ,*
2. *there are an even number of terminals between them, with half of them being left terminals and half of them being right terminals, and*
3. *if the bounding perimeters of N'_i and N'_j overlap, then the bounding perimeters of corresponding pairs of nets whose terminals lie between b'_i and a'_j overlap. $\square$*

procedure Net Modification

1. Sort the terminals. We use the rank information of each terminal.
2. For each net N'_i , find the nearest net N'_j to the right such that $\text{rank}(N'_i) = \text{rank}(N'_j)$ if it exists.
3. If such a net exists and $a'_j - b'_i \leq 2k$, then remove terminals b'_i and a'_j from the sorted list.
4. For each net N'_i whose b'_i is removed and whose a'_i is not removed, find the nearest N'_j to the right such that a'_j is removed and b'_j is not removed, and $\text{rank}(N'_i) = \text{rank}(N'_j)$. Make a net consisting of the two terminals a'_i and b'_j .

Clearly, the contours of the modified nets is the same as that of the original nets. Notice that steps 2 and 4 can be implemented by the integer sorting algorithms of chapter 2 or the **ANSV** algorithm of chapter 3.

Lemma 5.6 *The input nets can be modified as described above in time $O(\frac{n}{p}l^3)$ on the pipelined hypercube, when $n = p^{1+\frac{1}{l}}$, for any $l > 0$. $\square$*

Now we are ready to give the procedure to determine the contours of the different groups.

procedure Contours on L

1. Modify input nets by using procedure **Net Modification**.
2. For each modified net $N'_i = \langle a'_i, b'_i \rangle$ of rank k , produce two points $(a'_i - k, k + 1)$ and $(b'_i - k, k + 1)$.
3. Notice that several points from left (or right) terminals may have the same x -value. However, all such points come from consecutive nets and they can be identified easily. Remove all such points except the highest point.
4. Construct the contours with the resulting set of points.

Lemma 5.7 *Given n input nets, the contours of the nets can be found in time $O(\frac{n}{p}l^3)$ on the pipelined hypercube, when $n = p^{1+\frac{1}{l}}$, for any $l > 0$. $\square$*

We now consider the problem of determining the contours in R from the contours on L by cutting and folding L at the corners, and removing the unnecessary line segments. All the contours on L that do not cover any corner can easily be modified into the corresponding contours on R . Notice that there is at most one contour that cover each corner. We consider here the contour C that covers the lower right corner. All the other contours that cover the other corners can be treated similarly. Let the bottom side contour of C be $C_b = \{b_1, b_2, \dots, b_p\}$, where the b_i 's are line segments, ordered from left to right, and let the right side contour of C be $C_r = \{r_1, r_2, \dots, r_q\}$ ordered from top to bottom. We assume that C_r does not cover the top right corner, and the resulting contour on R is different from C_b or C_r alone, since these cases can be treated similarly.

Lemma 5.8 *Let (b_i, r_j) be an intersecting pair of line segments from C_b and C_r respectively such that i is smallest. Then no segment b_k , $k > i$, can intersect r_s , $s \leq j$.*

Proof: Notice that a horizontal segment u of height y of C came from a bounding perimeter of rank $y - 1$. Hence its length is at least $2y - 1$ or there is a higher horizontal segment in C that extends u . Let (b_i, r_j) be an intersecting pair such that i is smallest, and $(l - x, y)$ be the intersecting point. Assume that b_i is horizontal and r_j is vertical. The other case is symmetric. Let b_k , $k \geq i$, be a horizontal segment of the highest y -coordinate, say z , to the right of b_i . If

$z \leq y$, then we are done. If $z > y$, then $x > y$, since $(l - x, y)$ can not be on any line segment if $x \leq y$. Clearly $z \leq x$. Since $x > y$, b_k is to the right of r_j , and the lemma follows. $\square$

It follows from the above lemma that the desired contour can be obtained by finding the smallest i such that b_i of C_b intersects a segment r_j of C_r . We can find such an intersection as follows. We first find $h(a)$, the height of C_b at a , $a = 1, \dots, l$. If b_i is horizontal and r_j is vertical, then the intersecting point $(l - x, y)$ can be easily found by checking the height of C_b for each vertical segment of C_r . Clearly the steps can be done by using the packet routing algorithm of chapter 2.

Assume that b_i is vertical and r_j is horizontal. Let r_k be a vertical segment with smallest x -coordinate, say $\hat{x}$, among all the vertical segments of C_r below C_b . Then we can prove that the sequence of horizontal segments of C_b from $\hat{x}$ to $l - x$ is nonincreasing in the y -coordinate. Thus, we can find the intersecting point in the same way as the previous case. After finding the intersecting point, we can easily remove the unnecessary line segments. We now state the main result of this section.

Theorem 5.2 *Detailed routing of the representative nets of n nets within a rectangle can be done in time $O(\frac{n}{p}l^3)$ on the pipelined hypercube, when $n = p^{1+\frac{1}{l}}$, for any $l > 0$. $\square$*

Corollary 5.2 *Detailed routing of the representative nets of n nets within a rectangle can be done in time $O(\frac{n \log n}{p}l)$ on the shuffle-exchange or in time $O(\frac{n \log n}{p}l^2)$ on the weak hypercube, the cube-connected cycle and the butterfly, when $n = p^{1+\frac{1}{l}}$, for any $l > 0$. $\square$*

5.4 Routability Testing

In this section, we address the problem of testing whether or not it is possible to route a set of nets in a given rectangle R . Notice that the detailed routing algorithm of the previous section assumes that the nets are routable, and notice also that it does not generate enough information for the routability testing since it only produces the bend points of the representative nets. However this algorithm is important for the routability testing as will be shown in this section. The routability testing algorithm will have the same time performance as the detailed routing algorithm.

Given a set of nets $\{N_i = \langle a_i, b_i \rangle \mid 1 \leq i \leq n\}$ in a rectangle R , these nets may not be routable because of one of the following reasons: (i) the graph determined by the nets in the rectangle is not planar, or (ii) the wiring of all the nets requires more area. The first case can be settled easily by sorting and

computing prefix sums as in steps 3 and 4 of procedure **Representative Nets**. Actually, the graph is not planar if and only if there is a net $N_i = \langle a_i, b_i \rangle$ such that the prefix sum value of a_i is not equal to that of $b_i + 1$.

Before we tackle the question of whether the rectangle is large enough to realize all the nets, we need several definitions. A *side net* is a net whose terminals lie on the same side of the rectangle. A *side group* is a group whose representative net is a side net. A *corner net (group)* and a *cross net (group)* can be defined similarly. In Figure 5.2, the side group is $\{N_{14}, N_{15}\}$, the corner group is $\{N_1, N_2, N_3, N_4, N_5\}$, and the cross group is $\{N_6, N_7, N_8, N_9, N_{10}, N_{11}, N_{12}, N_{13}\}$. The overall strategy of the routability testing is described next.

procedure Routability Testing

1. Partition the input nets into groups.
2. Ignoring corner and cross groups, determine the contours of the side groups and test if the combined side groups are routable. If they are not routable then report "not routable" and stop.
3. For each corner group, test whether the nets in the group are routable within the rectangle ignoring the remaining groups. If they are, find the contour of the group, else report "not routable" and stop. Notice that there are at most four corner groups.
4. For each cross group, test whether the nets in the group are routable within the rectangle ignoring the remaining groups. If they are, find the contour of the group, else report "not routable" and stop. Notice that there are at most two cross groups.
5. Test whether any of the contours generated at steps 2, 3 and 4 intersect. The problem is routable if and only if no two contours intersect.

Notice that if the rank of each side net is less than $\min(l, h) - 1$, the side groups on a side of R are clearly routable altogether. Thus step 2 of this algorithm can be done by finding the contours of all the side groups with procedure **Contour on L**, and then testing in R whether any two contours intersect using a strategy as in finding an intersecting point of contours described in section 5.3 (the paragraph following the proof of Lemma 5.7). Step 5 can also be done similarly.

We now describe the routability testing of the corner groups. The routability of the cross groups can be tested similarly. We consider the corner group that covers the corner $(l, 0)$. The other corner groups can be treated similarly. If we remove all the corner nets from the corner group, the remaining side nets can be partitioned into a set of *side subgroups*. Let sb_i (sr_j) be a subgroup on

the bottom (right) side of R , where $1 \leq i \leq k_b$ ($1 \leq j \leq k_r$) from right to left (bottom to top). Then we define the *density* between sb_i and sr_j to be the number of corner nets that have to pass between the contours corresponding to the two subgroups and the *capacity* to be the number of corner nets that can pass within this passage. Let $cn(sb_i)$ ($cn(sr_j)$) be the number of corner nets whose terminals lie between the corner $(l, 0)$ and the leftmost terminal of the subgroup sb_i (sr_j). We can find such numbers by using the prefix sums algorithm on each side. Clearly the density between sb_i and sr_j is $|cn(sb_i) - cn(sr_j)|$. The following algorithm finds the capacities and tests the routability of the corner group.

procedure Corner Group

1. Remove all corner nets from the group and test the routability of the remaining side subgroups with a strategy as in step 2 of procedure **Routability Testing**. If they are routable then find the contours, else report "not routable" and stop.
2. We assume that all the line segments of the contours are sorted on each side. Let b_i (r_j) be the i -th (j -th) segment on bottom (right) side from right to left (bottom to top). Let D be the diagonal which is 45 degree line in R starting from the corner $(l, 0)$. Find the projection, $p(r_j)$, of r_j onto D and find $p'(r_j) = p(r_j) - \cup_{k=1}^{j-1} p(r_k)$. In Figure 5.5, $p'(CD) = C'D'$ and $p'(EF) = D'F'$.
3. For each corner point of the bottom contours, find the closest point x on D and the line segment r_j of the right contours such that $p'(r_j)$ contains x . In Figure 5.5, the line segments for corner points A and B are CD and EF respectively.
4. For each corner point of the right contours, find such a line segment of the bottom contours in the same way.
5. If there is a corner point such that the distance between the point and the corresponding line segment found in steps 4 and 5 is less than the density of the two corresponding subgroups then report "not routable", else the corner group is routable.

Lemma 5.9 *The procedure **Corner Group** tests the routability of the corner group in time $O(\frac{n}{p}l^3)$ on the pipelined hypercube, when $n = p^{1+\frac{1}{l}}$ for any $l > 0$. $\square$*

Theorem 5.3 *Testing the routability of n nets within a rectangle can be done in time $O(\frac{n}{p}l^3)$ on the pipelined hypercube, when $n = p^{1+\frac{1}{l}}$ for any $l > 0$. $\square$*

Chapter 6

Conclusion

6.1 Summary

In this thesis, we showed several results related to load balancing, sorting, packet routing, list ranking, graph theory, and VLSI routing on the pipelined hypercube, the weak hypercube, the shuffle-exchange, the cube-connected cycles, and the butterfly. These results included the following.

- Development of provably efficient algorithms on the above models.
- Establishment of lower bounds on the weak hypercube and bounded-degree networks. All the problems considered were shown to require $\Omega(\frac{n \log p}{p})$ time on bounded-degree networks.

These results shed some light on the relative powers of the pipelined hypercube, the weak hypercube, and the bounded-degree networks.

In Chapter 2, we showed several results concerning load balancing and sorting, and related them to the general packet routing problem. We presented an algorithm for load balancing whose time complexity is $O(M + \log p)$ on the pipelined hypercube and $O(M \log p)$ on the shuffle-exchange, cube-connected cycles, and butterfly. We also provided a lower bound for our bounded-degree networks, and showed that load balancing required more time on the shuffle-exchange, the cube-connected-cycles, or the butterfly than on the weak hypercube.

Many of our algorithms needed to sort integers from a small range efficiently. We presented an $O(\frac{n}{p})$ algorithm for sorting integers from a range polynomial in the number of processors for the pipelined hypercube, whenever $n = \Omega(p^{1+\epsilon})$. Integer sorting was shown to require $\Omega(\frac{n \log p}{p})$ time on the weak hypercube and on any bounded-degree network.

We also presented an algorithm for the general packet routing problem whose time complexity is $O(k_1 + k_2 + \frac{n}{p})$ on the pipelined hypercube, and

$O((k_1 + k_2) \log p + \frac{n \log p}{p})$ on the weak hypercube and our bounded-degree networks, whenever $n = \Omega(p^{1+\epsilon})$. The problem requires $\Omega(\frac{n \log p}{p})$ time on the weak hypercube and on any bounded-degree networks. Thus the upper bounds were tight for these networks.

In Chapter 3, we presented almost uniformly optimal algorithms to solve several problems such as the all nearest smaller values problem (ANSV), triangulating a monotone polygon, and line packing. We presented an algorithm for the ANSV problem whose time complexity was $O(\frac{n}{p} + \log^4 p)$ on the pipelined hypercube and $O(\frac{n \log p}{p} + \log^4 p)$ on all the remaining networks. This network algorithm was used to obtain algorithms for triangulating a monotone polygon and link packing. We also proved that these problems require $\Omega(\frac{n \sqrt{\log p}}{p})$ time on the weak hypercube and $\Omega(\frac{n \log p}{p})$ time on our bounded-degree networks. Thus, these algorithms were also almost uniformly optimal on our bounded-degree networks (despite being only almost efficient).

In Chapter 4, we presented a list ranking algorithm that could be executed on the pipelined hypercube in time $O(\frac{n}{p})$ when $n = \Omega(p^{1+\epsilon})$, and in time $O(\frac{n \log n}{p} + \log^3 p)$ otherwise. We used these techniques to obtain fast algorithms for several basic graph problems such as tree expression evaluation, connected and biconnected components, ear decomposition, and st-numbering. These problems were also addressed for the other network models. We also proved that list ranking requires $\Omega(\frac{n \log p}{p})$ time on the weak hypercube and any bounded-degree network. Thus, our algorithm was optimal.

In Chapter 5, we presented fast algorithms for the detailed routing and the routability testing problems within a rectangle whose time complexities were $O(\frac{n}{p})$ on the pipelined hypercube, and $O(\frac{n \log p}{p})$ on all the remaining networks, when $n = \Omega(p^{1+\epsilon})$. Fast algorithms were also developed for several subproblems that were interesting in their own right. One such subproblem was to determine the contours of the union of sets of contours within a rectangle.

6.2 Future Research

ANSV is a basic problem, and its faster solution can be used to obtain faster solutions for several important problems such as triangulating a monotone polygon, parenthesis matching, and VLSI routing problems. Our solution for ANSV depended on the block permutation algorithm that could be performed faster if we could find a faster algorithm to set up the data paths for an arbitrary input permutation on a butterfly permutation network. Thus, finding a faster algorithm for setting up the paths on our network models is an important open problem. Finding a faster solution for ANSV with or without the block permutation scheme on the network models is also an important open problem.

Load balancing is a fundamental problem for the network model since better balancing over the processors results in better processor utilization. Our load balancing algorithm is optimal on the pipelined hypercube. But this is not optimal on Ho and Johnsson's more powerful hypercube model in which each processor can set up, and send or receive $\log p$ packets in a given time step [31]. Actually, when $M = \Omega(n)$, load balancing can be done in $O(\frac{M}{\log p} + \log p)$ on this more powerful model. The load balancing algorithm is also not optimal on our bounded-degree networks when $M \gg \frac{n}{p}$. Thus, finding a general optimal solution for load balancing problem on these models when $\frac{n}{p} \leq M \leq n$ is an interesting open problem.

Sorting integers is also a fundamental problem and we provided an algorithm that was optimal when $n = \Omega(p^{1+\epsilon})$. That algorithm was used to solve the packet routing problem, and thus was used to solve several other problems such as list ranking, VLSI routing, and graph-theoretic problems. Actually, our algorithm could be performed in $O(\frac{n}{p}k^3)$ time, when $n = p^{1+\frac{1}{k}}$. Note that k can be as big as $\log p$, and the algorithm is not optimal when $n = o(p^{1+\epsilon})$. Thus, finding an integer sorting algorithm that has a better performance, when $n = o(p^{1+\epsilon})$, is an important open problem.

Finally, the algorithm for finding connected components of a graph is not efficient on the pipelined hypercube even when $n = \Omega(p^{1+\epsilon})$. Since a faster solution for the problem can be directly used to find faster solutions for other graph-theoretic problems, finding a faster connected component algorithm is also an important open problem.

Bibliography

- [1] A. Aggarwal and M. A. Huang. *Network complexity of sorting and graph problems and simulating CRCW PRAM's by interconnection networks*. Proc. 3rd Aegean Workshop on Computing, AWOC 88, Corfu, Greece, June/July 1988, pp. 339-350.
- [2] R.J. Anderson and G.L. Miller. *Deterministic parallel list ranking*. Proc. 3rd Aegean Workshop on Computing, AWOC 88, Corfu, Greece, June/July 1988, pp. 81-90.
- [3] M.J. Atallah and S.E. Hambrusch. *Solving tree problems on a mesh-connected processor array*. Proc. IEEE FOCS, 1985, pp. 222-231.
- [4] M.J. Atallah and U. Vishkin. *Finding euler tours in parallel*. Jour. of Comp. and Sys. Sci., Vol. 29, Dec. 1984, pp. 330-337.
- [5] G.H. Barnes, R.M. Brown, M. Kato, D.J. Kuck and D.L. Slotnick. *The ILLIAC IV computer*. IEEE Trans. Comp., Vol. C-17, 1968, pp. 746-757.
- [6] K.E. Batcher. *Sorting networks and their applications*. Spring Joint Computer Conference 32, AFIPS Press, 1968, pp.307-314.
- [7] O. Berkman, B. Schieber and U. Vishkin, *Some doubly logarithmic optimal parallel algorithms based on finding all nearest smaller values*. UMIACS-TR-88-79, Univ. of Maryland, 1988.
- [8] R.P. Brent. *The parallel evalution of general arithmetic expressions*. JACM, Vol. 21, 1974, pp. 201-206.
- [9] S.-C. Chang and J. JáJá. *Parallel algorithms for channel routing in the knock-knee model*. Submitted to SIAM Jour. on Computing. Also available in Proc. 1988 International Conf. on Parallel Processing.
- [10] S.-C. Chang, J. JáJá and K.W. Ryu. *Optimal parallel algorithms for one-layer routing*. Submitted to Jour. of Algorithms. Also available as Technical Report UMIACS-TR-89-46, CS-TR-2239, Univ. of Maryland, April 1989.

- [11] F.Y. Chin, J. Lam, and I-Ngo Chen. *Efficient parallel algorithms for some graph problems*. CACM, Vol. 25, 1982, pp. 659-665.
- [12] R. Cole. *Parallel merge sort*. SIAM J. COMPUT., Vol. 17, No. 4, Aug. 1988, pp. 770-785.
- [13] R. Cole and A. Siegel. *River routing every which way, but loose*. Proc. IEEE FOCS, 1984, pp. 65-73.
- [14] R. Cole and U. Vishkin. *Deterministic coin tossing with application to optimal parallel list ranking*. Information and Control, 70, 1986, pp. 32-53.
- [15] R. Cole and U. Vishkin. *Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms*. Proc. 18th ACM Symp. on Theory of Computing, 1986, pp. 206-219.
- [16] R. Cole and U. Vishkin. *Faster optimal parallel prefix sums and list ranking*. TR-56/86, The Moise and Frida Eskenasy Inst. of Comp. Sci., Tel Aviv University.
- [17] R. Cole and U. Vishkin. *Approximate parallel scheduling. Part 2: Applications to logarithmic-time optimal parallel graph algorithms*. To be published.
- [18] R. Cypher and J.L.C. Sanz. *Cubesort: An optimal sorting algorithm for feasible parallel computers*. Proc. 1988 international Conf. on Parallel Processing, 1988, pp. 308-311.
- [19] E. Dekel and S. Sahni. *Parallel scheduling algorithms*. Operations Research. Vol. 31, No. 1, Jan.-Feb. 1983.
- [20] D. Dolev, K. Karplus, A. Seigel, A. Strong and J. Ullman. *Optimal wiring between rectangles*. Proc. 13th ACM Symp. on Theory of Computing, 1981, pp. 312-317.
- [21] D. Eppstein and Z. Galil. *Parallel algorithmic techniques for combinatorial computation*. Feb. 1988.
- [22] A.V. Goldberg, S.A. Plotkin and G.E. Shannon. *Parallel symmetry-breaking in sparse graphs*. Proc. 19th ACM Symp. on Theory of Computing, 1987, pp. 315-324.
- [23] A. Gottlieb and C.P. Kruskal. *Complexity results for permuting data and other computations on parallel processors*. JACM, Vol. 31, 1984, pp.193-209.
- [24] T. Hagerup. *Towards optimal parallel bucket sorting*. Information and Computation 75, 1987, pp. 39-51.

- [25] T. Hagerup. *Optimal parallel algorithms on planar graphs*. Proc. 3rd Aegean Workshop on Computing, AWOC 88, Corfu, Greece, June/July 1988, pp. 24-32.
- [26] Y. Han. *Designing fast and efficient parallel algorithms*. Ph.D. Dissertation. Dept. Computer Sci., Duke Univ., 1987.
- [27] Y. Han. *An optimal parallel algorithm for computing linked list prefix*. TR-100-87, Dept. of Computer Sci., Univ. of Kentucky, Lexington, 1987.
- [28] K.T. Herley. *Efficient simulations of small shared memories*. Proc. IEEE FOCS, 1989, pp. 390-395.
- [29] K.T. Herley and G. Bilardi. *Deterministic simulations of PRAMs*. Proc. 26th Allerton Conf. on Communication, Control and Computing, 1988, pp. 1084-1093.
- [30] D.S. Hirschberg and A.K. Chandra. *Computing connected components on parallel computers*. CACM, Vol. 22, No. 8, Aug. 1979, pp. 461-464.
- [31] C.T. Ho and S.L. Johnson. *Distributed routing algorithms for broadcasting and personalized communication in hypercubes*. Proc. 1986 International Conf. on Parallel Processing, 1986, pp. 640-648.
- [32] C.T. Ho and S.L. Johnson. *Algorithms for matrix transposition on Boolean n-cube configured ensemble architectures*. Proc. 1987 International Conf. on Parallel Processing, 1987, pp. 621-629.
- [33] D. Hoey and C.E. Leiserson. *A layout for the shuffle-exchange network*. Proc. 1980 International Conf. on Parallel Processing, 1980, pp. 329-336.
- [34] J. JáJá and K.W. Ryu. *Efficient techniques for routing and for solving graph problems on the hypercube*. UMIACS-TR-89-33, CS-TR-2216, Univ. of Maryland, March 1989.
- [35] J. JáJá and K.W. Ryu. *Load balancing and routing on the hypercube and related networks*. Submitted to Jour. of Parallel and Distributed Computing. Also available as Technical Report UMIACS-TR-89-61, CS-TR-2264, Univ. of Maryland, June, 1989.
- [36] D. Johannsen. *Bristle blocks: a silicon compiler*. Proc. 16th Design Automation Conference, 1979, pp. 310-313.
- [37] S.L. Johnson. *Communication efficient basic linear algebra computations on hypercube architecture*. Jour. of Parallel and Distributed Computing, Vol. 4, 1987, pp. 133-172.

- [38] R.M. Karp and V. Ramachandran. *A survey of parallel algorithms for shared-memory machines*. Report No. UCB/CSD 88/408, Comp. Sci. Div., Univ. of California, Berkeley, Mar. 1988.
- [39] D.E. Knuth. *The art of computer programming, Vol. III: Sorting and searching*. Addison-Wesley, 1973.
- [40] S.R. Kosaraju and A.L. Delcher. *Optimal parallel evaluation of tree-structured computations by raking*. Proc. 3rd Aegean Workshop on Computing, AWOC 88, Corfu, Greece, June/July 1988, pp. 101-110.
- [41] M. Kramer and J. van Leeuwen. *Wire routing is NP-complete*. TR. University of Utrecht, the Netherlands, Feb. 1982.
- [42] C.P. Kruskal, T. Madej and L. Rudolph. *Parallel prefix on fully connected direct connection machines*. Proc. 1986 International Conf. on Parallel Processing, pp. 278-284.
- [43] C.P. Kruskal, L. Rudolph and M. Snir. *A complexity theory of efficient parallel algorithms*. To appear in Theoretical Computer Science.
- [44] C.P. Kruskal, L. Rudolph and M. Snir. *The power of parallel prefix*. IEEE. Trans. Comp., Vol. C-34, No. 10, Oct. 1985, pp. 965-968.
- [45] H.T. Kung. *The structure of parallel algorithm*. Advances in Computers, Vol. 19, Academic Press, Inc., 1980, pp. 65-112.
- [46] A. LaPaugh. *Algorithms for integrated circuit layout: an analytic approach*. Ph.D. dissertation, MIT, Cambridge, MA, Nov. 1980.
- [47] T. Leighton. *Tight bounds on the complexity of parallel sorting*. IEEE. Trans. Comp., Vol. C-34, No. 4, April 1985, pp. 344-354.
- [48] C.E. Leiserson and F.M. Maley. *Algorithms for routing and testing routability of planar VLSI layouts*. Proc. 17th ACM Symp. on Theory of Computing, 1985, pp. 69-78.
- [49] C. Leiserson and R. Pinter. *Optimal placement for river routing*. SIAM J. COMPUT. Vol. 12, No. 3, Aug. 1983, pp. 447-462.
- [50] G.F. Lev, N. Pippenger and L.G. Valiant. *A fast parallel algorithm for routing in permutation networks*. IEEE Trans. Comp., Vol. C-30, No. 2, Feb. 1981, pp. 93-100.
- [51] F.M. Maley. *Toward a mathematical theory of single-layer wire routing*. 5th MIT Conference on Advanced Research in VLSI, March 1988, pp. 277-296.

- [52] Y. Maon, B. Schieber and U. Vishkin. *Parallel ear decomposition search (EDS) and st-numbering in graphs*. Theoretical Comp. Sci. 47, 1986, pp. 277-298.
- [53] A. Mirzaian. *Channel routing in VLSI*. Proc. 16th ACM Symp. on Theory of Computing, 1984, pp. 101-107.
- [54] A. Moitra and S.S. Iyengar. *Parallel algorithms for some computational problems*. Advances on Computers, Vol. 26, Academic Press, Inc., 1987, pp. 93-153.
- [55] D. Nassimi and S. Sahni. *Bitonic sort on a mesh-connected parallel computer*. IEEE Trans. on Comp., Vol. C-27, No. 1, Jan. 1979, pp. 2-7.
- [56] D. Nassimi and S. Sahni. *Data broadcasting in SIMD computers*. IEEE Trans. on Comp., Vol. C-30, No. 2, Feb. 1981, pp. 101-107.
- [57] D. Nassimi and S. Sahni. *Parallel algorithms to set up the benes permutation network*. IEEE Trans. on Comp., Vol. C-31, No. 2, Feb. 1982, pp. 148-154.
- [58] D. Peleg and E. Upfal. *The generalized packet routing problem*. Theoretical Comp. Sci. 53, 1987, pp. 281-293.
- [59] R. Pinter. *River routing: methodology and analysis*. Proc. 3rd CALTECH Conference on Very Large Scale Integration, 1983, pp. 141-163.
- [60] N.J. Pippenger. *On simultaneous resource bounds*. Proc. IEEE FOCS, 1979, pp. 307-311.
- [61] C.G. Plaxton. *Load balancing, selection and sorting on the hypercube*. Proc. 1989 ACM Symp. on Parallel Algorithms and Architectures, pp. 64-73.
- [62] F.P. Preparata and J. Vuillemin. *The cube-connected cycle: A versatile network for parallel computation*. CACM 24, 1981, pp. 300-309.
- [63] K.W. Ryu and J. JáJá. *List ranking on the hypercube*. Proc. 1989 International Conf. on Parallel Processing, Vol. III, pp. 20-23.
- [64] K.W. Ryu and J. JáJá. *Efficient algorithms for list ranking and for solving graph problems on the hypercube*. IEEE Trans. Parallel and Distributed Systems, Vol. 1, No. 1, Jan. 1990, pp. 83-90.
- [65] Y. Saad and M.H. Schultz. *Data communication in hypercubes*. TR YALEU/DCS/RR-428, Yale University, Oct. 1985.
- [66] Y. Saad and M.H. Schultz. *Topological properties of hypercubes*. IEEE Trans. Comp., Vol. C-37, No. 7, July 1988, pp. 867-872.

- [67] S. Sahni and A. Bhatt. *Complexity of the Design Automation Problem*. Proc. 17th Design Automation Conference, June 1980, pp. 402-411.
- [68] B. Schieber and U. Vishkin. *On finding lowest common ancestors: simplification and parallelization*. SIAM J. COMPUT., Vol. 17, No. 6, Dec. 1988, pp. 1253-1262.
- [69] B. Schieber and U. Vishkin. *Finding all nearest neighbors for convex polygons in parallel: A new lower bound technique and a matching algorithm*. UMIACS-tr-88-82, CS-TR-2138, Univ. of Maryland, Nov. 1988.
- [70] E. Schwabe. *Normal hypercube algorithms can be simulated on a butterfly with only constant slowdown*. Manuscript, 1989.
- [71] J.T. Schwartz. *Ultracomputers*. ACM Trans. Prog. Languages and Systems, Vol. 2, No. 4, Oct. 1980, pp. 484-521.
- [72] A. Seigel and D. Dolev. *Some Geometry for General River Routing*. SIAM J. COMPUT. Vol. 17, No. 3, June 1988, pp. 583-605.
- [73] C.L. Seitz. *The cosmic cube*. CACM 28, 1985, pp. 22-33.
- [74] Y. Shiloach and U. Vishkin. *An $O(\log n)$ parallel connectivity algorithm*. Jour. of Algorithms 3, 1982, pp. 57-67.
- [75] H.S. Stone. *Parallel Processing with The Perfect Shuffle*. IEEE Trans. on Comp., Vol. c-20, No. 2, Feb. 1971, pp. 153-161.
- [76] T. Szymanski. *Dogleg Channel routing is NP-complete*. Manuscript, Bell Laboratories, Murray Hill, NJ, Sep. 1981.
- [77] R.E. Tarjan and U. Vishkin. *An efficient parallel biconnectivity algorithm*. SIAM J. COMPUT., Vol. 14, No. 4, Nov. 1985, pp. 862-874.
- [78] C.D. Thompson and H.T. Kung. *Sorting on a mesh-connected parallel computer*. CACM, Vol. 20, No. 4, Apr. 1977, pp. 263-271.
- [79] M. Tompa. *An optimal solution to a wire routing problem*. Proc. 12th ACM Symp. on Theory of Computing, 1980, pp. 161-176.
- [80] Y.H. Tsin and F.Y. Chin. *Efficient parallel algorithms for a class of graph theoretic problems*. SIAM J. COMPUT., Vol 13, No. 3, Aug. 1984. pp. 580-599.
- [81] J.D. Ullman. *Computational aspects of VLSI*. Computer Science Press, Inc., 1984.

- [82] P. Varman and K. Doshi. *Sorting with linear speedup in a VLSI network*. Proc. 1988 International Conf. on Parallel Processing, 1988, pp. 202-206.
- [83] U. Vishkin. *Synchronous parallel computation - A survey*. TR-#71, Courant Institute, New York University, Apr. 1983.
- [84] U. Vishkin. *An optimal parallel connectivity algorithm*. Disc. App. Math. 9, 1984, pp. 197-207.
- [85] U. Vishkin. *On efficient parallel strong orientation*. Infor. Proc. Letters 20, June 1985, pp. 235-240.
- [86] A. Waksman. *A permutation network*. JACM, Vol. 15, No. 1, Jan. 1968, pp. 159-163.
- [87] J.C. Wyllie. *The complexity of parallel computation*. TR 79-387, Dept. of Computer Sci., Cornell Univ., Ithaca, NY, 1979.

Curriculum Vitae

Name: Kwan Woo Ryu

Permanent address: 228-4, Beomo-dong, Sooseong-ku,
Daegu, Korea

Degree and date to be conferred: Ph.D., 1990

Date of birth: November 2, 1957

Place of birth: Andong, Korea

Secondary education: Kyungpook High School, Daegu, Korea

Collegiate institutions attended:

Institution	Dates Attended	Degree	Date of Degree
University of Maryland College Park, MD 20742 U.S.A.	8/85	Ph.D.	5/90
Korea Advanced Institute of Science and Technology Seoul, Korea	3/80	M.S.	2/82
Kyungpook Natl. Univ. Daegu, Korea	3/76	B.S.	2/80

Major: Computer Science

Professional publications:

- [1] *Efficient Algorithms for List Ranking and for Solving Graph Problems on the Hypercube* (with J. JáJá), IEEE Trans. Parallel and Distributed Systems, Vol. 1, No. 1, Jan. 1990, pp. 83-90.
- [2] *Optimal Parallel Algorithms for One-Layer Routing* (with S.-C. Chang and J. JáJá), submitted to Journal of Algorithms. Also available as Technical Report UMIACS-TR-89-46, CS-TR-2239, Univ. of Maryland, April 1989.

- [3] *Load Balancing and Routing on the Hypercube and Related Networks* (with J. Jájá), submitted to Journal of Parallel and Distributed Computing. Also available as Technical Report UMIACS-TR-89-61, CS-TR-2264, Univ. of Maryland, June, 1989.
- [4] *List Ranking on the Hypercube* (with J. Jájá), Proc. of 1989 International Conference on Parallel Processing, Vol. III, pp. 20-23.
- [5] *Almost Uniformly Efficient Parallel Algorithms for Several Problems on the Network Model* (with J. Jájá), submitted to 1990 Symp. on Parallel Algorithms and Architectures.
- [6] *Efficient Techniques for Routing and for Solving Graph Problems on the Hypercube* (with J. Jájá), UMIACS-TR-89-33, CS-TR-2216, Univ. of Maryland, March 1989.

Professional positions held:

- | | |
|-------------|--|
| 6/90- | Assistant Professor
Kyungpook Natl. Univ.
Daegu, Korea. |
| 7/89 - 6/90 | Graduate Research Fellow
Institute for Advanced Computer Studies,
University of Maryland
College Park, MD 20742, U. S. A. |
| 8/85 - 6/89 | Graduate Assistant
Dept. of Computer Science,
Systems Research Center,
Institute for Advanced Computer Studies,
University of Maryland
College Park, MD 20742, U. S. A. |
| 3/82 - 7/85 | Full Time Lecturer
Dept. of Electrical Engineering
Kyungpook Natl. Univ., Daegu, Korea. |
| 3/80 - 2/82 | Research Assistant
Department of Computer Science
Korea Advance Institute of Science and Technology
Seoul, Korea |