MULTISELECTION ON SOME INTERCONNECTION NETWORKS

BY

Adel Fadhl Noor Ahmed

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **Adel Fadhl Noor Ahmed** under the direction of his thesis advisor, and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE IN COMPUTER SCIENCE

Thesis Committee:

[illegible]
Dr. Mohammed Alsuwaiyel (*Thesis Advisor*)

[illegible]
Dr. Muhammad Sarfraz (*Member*)

[illegible]
Dr. Nasir Al-Darwish (*Member*)

Kanaan Faisal 19/9/2001
Dr. Kanaan Faisal
Department Chairman

[illegible]
Dr. Osama Ahmed Jannadi
Dean of Graduate Studies

عمادة الدراسات العليا
DEANSHIP OF GRADUATE STUDIES

22/9/2001
Date

To my parents

Acknowledgment

Acknowledgement is due to King Fahd University of Petroleum and Minerals for supporting this research.

I would like to express my deep gratitude to my thesis committee chairman, Dr. Muhammad Alsuwaiyel for his continuous assistance and support. I am indebted to my thesis committee Dr. Mohammed Sarfraz and Dr. Naser Darwish for their time, guidance and suggestions.

I am thankful to the dean of College of Computer Science and Engineering Dr. Jaralla Al-Ghamdi and the chairman of department of Information and Computer Science, Dr. Kanan Faisal and other faculty members for their cooperation and support.

I would also like to express my gratitude to my father Dr. Fadhl N. M. Ahmed for his guidance to build a better future, my mother for taking pains to fulfill my academic pursuits and building my personality, my siblings and all the members of my family for their emotional and moral support through my academic career.

Last but not least let me mention my sincere appreciation to my friends for their help and encouragement: Hani Al-Arifi, Mohammed Al-Sheheri, Talal Al-Bakr, Purushothaman, and especially Mutlaq Al-Mutlaq for giving me the final push to finish this work.

Table of Contents

Table of Figures

Thesis Abstract

NAME	: ADEL FADHL NOOR AHMED
TITLE	: MULTISELECTION ON SOME INTERCONNECTION NETOWRK
MAJOR FIELD	: COMPUTER SCIENCE
DATE OF DEGREE	: JULY, 2001

As time passes by, more and more computers get connected to the Internet creating the environment ancient researches in parallel and distributed applications only dreamt of. Programmers and application developers turned to distributed processing to accomplish tasks with greater speeds. Unlike the classic single processor machine, the interconnection network facilitates simultaneous execution for multiple instructions. This assisted the rapid expansion of parallel algorithms and architectures.

Several parallel algorithms are devised such that they depend on preprocessing phases to prepare data for processing. Selection and Multiselection arise very often is such cases. Unlike selection, the multiselection problem did not get its share in studies. Some special case solutions were suggested but no detailed or universal solution for the problem was to be found in the literature. In this study we present a parallel multiselection algorithm suitable for execution on some interconnection network. After building the proper background to pave the way for the solutions, the multiselection problem is defined. Then the universal algorithm for multiselection is studied in detail. Compared with other less-direct ways to accomplish the task of multiselection, our algorithm proved to perform better than any other on a fairly large range in the problem domain.

خلاصة الرسالة

الإســـــــــــم	: عـــادل بن فضـــل نور أحمـــد
عنـــوان الرسالة	: الاختيار المتعدد على بعض الشبكات المتصلة
التخصـــــــــص	: علوم الحاسب الآلي
تاريخ التخـــــرج	: يوليو ٢٠٠١م

مع ازدياد عدد الحاسبات التي تتصل بالإنترنت أصبح حلم الباحثين في مجال التطبيقات المتوازية و المتوزعة ببيئة مناسبة لتطوير و اختبار تطبيقاتهم أقرب إلى الواقع. و ذلك ادى إلى ظهور برامج بصفتها المتوزعة و مبنية على الخوارزميات المتوازية تعالج البيانات بسرعات فائقة جداً. بخلاف الحاسبات ذات المعالجات واحدة ، الشبكات المتصلة تسهل تنفيذ خطوات متعددة بالتزامن مما ساعد على سرعة تطور مجال الخوارزميات و التصاميم المتوازية.

يعتمد تصميم كثيراً من الخوارزميات المتوازية على خطوات تحضيرية لتجهيز البيانات قبل معالجتها. الاختيار و الاختيار المتعدد من أمثلة بعض هذه الخوارزميات التى غالباً ما تظهر في تلك الخطوات. فمسألة الاختيار المتعدد هي من ضمن المسائل التي لم تحصل على نصيبها في الدراسات على عكس الخوارزميات الأخرى . رغم وجود حلول لحالات خاصة ، تفتقر المراجع المختصة لحل عام و مفصل للاختيار المتعدد. نستعرض في هذه الرسالة خوارزمية الاختيار المتعدد القابلة التنفيذ على أغلب انواع الشبكات المتصلة . فبعد إعطاء تعريف دقيق لمسألة الاختيار المتعدد نورد دراسة شاملة لخوارزمية عامة الاختيار المتعدد و من ثم نعرض مقارنة ادائها مع خوارزميات أخرى يمكن باستخدامها الغير مباشر حل مسألة الاختيار المتعدد على الشبكات المتصلة .

Chapter 1

Introduction

The *selection* problem comes under a category of problems called *Order Statics*. The kth order static of a set of n elements is the kth smallest element. The *multiselection* problem is to find the elements of ranks $k_1, k_2, \ldots k_r$, that is, the k_1th, k_2th, up to the k_rth smallest element of a set of n elements, where $1 \le r \le n$. This chapter is an introductory chapter. In section 1.1 we present a formal introduction of the selection and the multiselection problems. In section 1.2, a brief survey of various computation models is presented. The analysis notations and conventions used in the literature and in this material will be followed in section 1.3. Literature overview, section 1.4, contains a history of previous results and solutions to the selection and multiselection problems on several computation models and specifies the scope and goals of this material. Section 1.5 presents the classical solution to the selection problem on the Random Access Machine (RAM) model. See 1.2.1 for more detail on the RAM model.

1.1 SELECTION AND MULTISELECTION

1.1.1 The Selection Problem

The problem –as mentioned in [25] is:

Input: A set A of n (distinct) numbers and a number k, with $1 \le k \le n$.

Output: The element $x \in A$ that is larger than exactly $k-1$ other elements of A.

A more general definition of the problem is found in [23] that states: Given a sequence $s = \{s_1, s_2, ..., s_n\}$ of numbers listed in arbitrary order and an integer k, where $1 \le k \le n$ the problem of selection calls for determining the kth smallest element of S. Here we include the situation where equal elements exist in the sequence. In this case ties between elements are broken using their indices: If two elements of S are equal, then the one with the smaller index is considered to be smaller (i.e. if $s_i = s_k$ then s_j is considered smaller than s_k if $j < k$.

The selection problem arises in many applications in computer science and statistics. Before we look into the solution of the selection problem we establish a lower bound of the problem. This will draw a guideline for our initial goal of devising a fast efficient algorithm for the problem. In [14] the selection problem was shown to belong to a family of problems known as *comparison problems*. These problems are usually solved by comparing pairs of elements of an input sequence. Three special cases arise when $k = 1$, $k = n$, and $k = \lceil n/2 \rceil$. In the first two cases we are looking for the smallest and largest elements of S, respectively. In the third case we are looking for the *median*. An element s_k is said to be the median of a set S if half of the elements of S are smaller than (or equal to) it and the other half is larger (or equal). Putting these special cases aside, and examining the general case it is certain that regardless of the value of k, in order to determine the kth smallest element, we must examine each element of S at least once [14]. This establishes a lower bound of $\Omega(n)$ on the number of sequential steps required to solve the problem.

One simple way to solve the selection problem is by using heap sort or merge sort and then simply extract the kth element in the output set. But this requires $\Omega(n \log n)$ time. This approach, however, solves the problem for every value of k from 1 to n, implying that perhaps a more efficient algorithm may exist that finds the kth smallest element only for the given k [23]. The minimum and maximum can be found separately with $n-1$ comparisons, and both can be found in $\frac{3n}{2}$ comparisons.

The upper bound for finding the minimum can be easily obtained: examine each element of the set in turn and keep track of the smallest element seen so far.

Finding the minimum using Algorithm 1 will take $n-1$ comparisons. By modifying the algorithm above, the maximum can also be found with $n-1$ comparisons. This is the best algorithm that can be devised for the problem and Algorithm 1 is optimal with respect to the number of comparisons performed.

Algorithm MINIMUM

```
1.  $min = A[1]$
2.  **for** $i = 2$ **to** $length[A]$ **do**
3.      **if** $min > A[i]$
4.          $min = A[1]$
5.      **end if**
6.  **end for**
7.  **return** $min$
```

Algorithm 1.1-1 Algorithm *Minimum*

It is not too difficult to devise an algorithm that can find both the maximum and minimum of n elements. One simple solution is to find the minimum and the maximum element independently, using $n-1$ comparisons for each, for a total of $2n-2$ comparisons. But in fact, only $3\lceil \frac{n}{2} \rceil$ comparisons are necessary to find both the minimum and the maximum. To do this, we maintain the minimum and maximum elements seen thus far. Rather than processing each element of the input by comparing it against the current minimum and maximum, however, at a cost of two comparisons per element, we process elements in pairs. We compare pairs of elements from the input first with *each other*, and then compare the smaller to the current minimum and the larger to the current maximum, at a cost of three comparisons for every two elements.

For the general selection, as mentioned above, sorting is not the optimal solution. Instead, the idea behind selection is to find the median of the element set. The median m is an element with the property that the number of elements smaller than m are equal to the number of elements larger than m in the elements set. Then the element set is partitioned into three subsets. One containing elements that are smaller than the median, one subset containing all the elements equal to the median, and the last subset containing elements larger than the median. Next, elements are discarded from future consideration by counting the size of the subsets and recursively do the same procedure again with the remaining sets. The selection problem and its solution are discussed in more details in section 1.5

1.1.2 The Multiselection Problem

Multiselection problem is an extension to the selection problem in which, we are to find not only one element, namely the *k*th smallest element, but a set of elements of ranks $k_1, k_2, \ldots k_r$. That is, we have to find the k_1th, k_2th, up to the k_rth smallest element of a set of n elements, where $1 \le r \le n$. We saw earlier that finding the minimum and the maximum element can be done in $3\lceil \frac{n}{2} \rceil$ comparison steps, this is a case of the multiselection problem where $k_1 = 1$ and $k_2 = n$.

A simple solution to the multiselection problem is to sort the set of element in $O(n \log n)$ steps and index the elements at positions $k_1, k_2, \ldots k_r$. But again, this solution is not optimal since the lower bound for selection problem, as will be established in section 1.5, is $\Omega(n)$. A divide-and-conquer approach is used to solve this problem, where the problem space is divided into two sub problems, in which the element set and the rank set is partitioned into two halves, then each sub problem is solved separately as two smaller multiselection problems. The final solution is combined at the end to get the elements required. As it was for the selection problem, the multiselection problem is bounded by $\Omega(n \log r)$ where n is the number of elements, and r is the number of ranks. Chapter 2 contains detailed analysis of the process.

1.2 COMPUTATIONAL MODELS

Before searching for a suitable solution for any problem the type of machine used to solve the problem has to be determined. In the following, some of the most popular architectures are presented.

1.2.1 Random Access Machine

The solution provided above is suitable to a model of computation called Random Access Machine RAM, which consists of a memory with M locations, a processor operating under a sequential algorithm, and a memory access unit (MAU) whose purpose is to create a path from the processor to an arbitrary location in the memory. Each step of the algorithm consists of up to three phases as mentioned in [23].

1. A READ phase, in which the processor reads a datum from an arbitrary location in memory into one of its registers
2. A COMPUTE phase, in which the processor performs a basic operation on the contents of one or two of its registers, and
3. A WRITE phase, in which the processor writes the contents of one register into an arbitrary memory location.

1.2.2 Parallel Random Access Machine

Another model that became very popular recently is the Parallel Random Access Machine (PRAM). This model consists of a number of identical processors $P_1, P_2, \ldots, P_N$ where *N* is arbitrarily large. A common memory with *M* locations, arbitrarily large such that $M \geq N$. And a memory access unit (MAU) that allows the processors to gain access to memory.

Each step of the algorithm for the PRAM also consists of (up to) three phases:

1. A READ phase, in which (up to *N*) processors read simultaneously from (up to *N*) memory locations.
2. A COMPUTE phase, in which (up to *N*) processors perform basic arithmetic or logical operations on their local data
3. A WRITE phase, in which (up to *N*) processors write simultaneously into (up to *N*) memory locations.

There are number of different ways for the processors to gain access to memory

a) **Exclusive Read (ER):** Processors gain access to memory locations for the purpose of reading in a one-to-one fashion.

b) **Concurrent Read (CR):** Two or more processors can read from the same memory location at the same time.

c) **Exclusive Write (EW):** Processors gain access to memory locations for purpose of writing in a one-to-one fashion.

d) **Concurrent Write (CW):** Two or more processors can write into the same memory location at the same time. Several extensions are available for the CW instruction

   a. **Priority CW:** The processors are assigned certain priorities. Of all the processors attempting to write in a given memory location, only the one with the highest priority is allowed to do so.

   b. **Common CW:** Processors will succeed in writing to the same memory location if they attempt to write the *same* value. This is further specified as:

      i. **Fail Common:** The content of the memory is unchanged if the written values are not all equal.

      ii. **Collision Common:** A "failure" label is stored in the memory location in case the CW does not succeed

      iii. **Fail-Safe Common:** The algorithm must be designed to overcome a CW failure situation.

   c. **Arbitrary CW:** Any processor can succeed in writing arbitrarily to the memory location. The algorithm must decide how this will be done.

   d. **Random CW:** Choose at random which processor will write to the memory location.

   e. **Combining CW:** All the values are combined into a single value, which is then stored in the memory location. The following variants are available:

i. **Arithmetic functions:** Using sum, product, average ...etc.

ii. **Logical functions:** Using the OR, XOR, NAND ...etc.

iii. **Selection functions:** Using Min, Max ...etc.

1.2.3 Interconnection Networks

A network can be viewed as a graph $G = (N, E)$ where each node $i \in N$ represents a processor, and each edge $(i, j) \in E$ represents a two-way communication link. There is no shared memory. Instead, the M memory is distributed among N processors. Two processors directly connected by a link are said to be *neighbors*. As described earlier, the processors in the PRAM model use shared memory. A PRAM model can also be modeled as a complete graph where each processor is connected with every other processor by a communication link i.e. each processor is the neighbor of every other processor. In an N processor PRAM model each processor P_i has exactly N-1 neighbors.

Interconnected networks differ in various aspects; network topology, number of neighbors for each processor, message size, transmission delay, transmission path, processors ...etc. All of them have a major role in shaping the and designing of a suitable algorithm for a certain problem.

There are several parameters used to evaluate the topology of a network G. **Diameter,** which is the maximum distance between any pair of nodes, **maximum degree** of any node in G, and edge connectivity of G. In describing algorithms for the network model, we need additional construct for describing communication between processors. *send*() and *receive*() are used to send and receive a copy of a variable to and from a processor respectively.

Linear Array

We now turn to a description of the various network topologies that are studied in theory and practice. The simplest and most fundamental topology, which consists of p processors $P_1, P_2, \ldots, P_N$ connected in a **linear array**; that is, processor P_i is connected to P_{i-1} and to P_{i+1}, whenever they exist. A ring is linear array of processors with an end-round connection; that is, processor P_1 and P_N are directly connected. See Figure 1.2-1

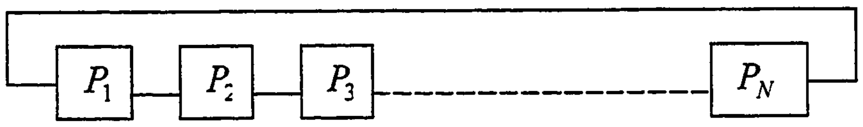


Figure 1.2-1 A ring (Linear Array)

Mesh

The two-dimensional **mesh**, Figure 1.2–2 , is a two-dimensional version of the linear array. It consists of $N = m^2$ processors arranged into an $m \times m$ grid such that processor $P_{i,j}$ is connected to processors $P_{i\pm1,j}$ and $P_{i,j\pm1}$, whenever they exist. The mesh model can be generalized to dimensions higher than two. In a d-dimensional mesh, each processor is connected to two neighbors in each dimension, with processors on the boundary having fewer connections

Hypercube

A **hypercube** consists of $N = 2^d$ processors interconnected into a *d*-dimensional hypercube such that each processor is connected to exactly q neighbors. The d neighbor of P_i are those processors P_j such that the binary representations of the indices i and j differ in exactly one bit. Figure 1.2–3 shows a 3 dimensional hypercube.

Butterfly

A *d*–dimensional **butterfly** consists of $(d+1)2^d$ nodes and $d2^{d+1}$ connection. Each processor is connected to at most 4 neighbors, independent of *d*. This property makes butterfly networks more desirable to work with rather than the hypercubes. In addition, algorithms devised on the butterfly network can be easily adapted to work on the hypercube. This makes butterfly networks a suitable candidate to develop algorithms for the hypercubes. Figure 1.2–4 shows a butterfly network.

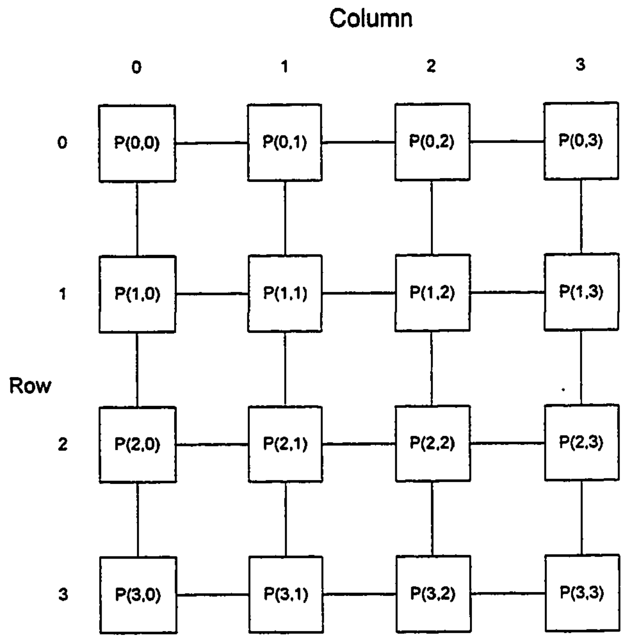


Figure 1.2-2 Mesh network

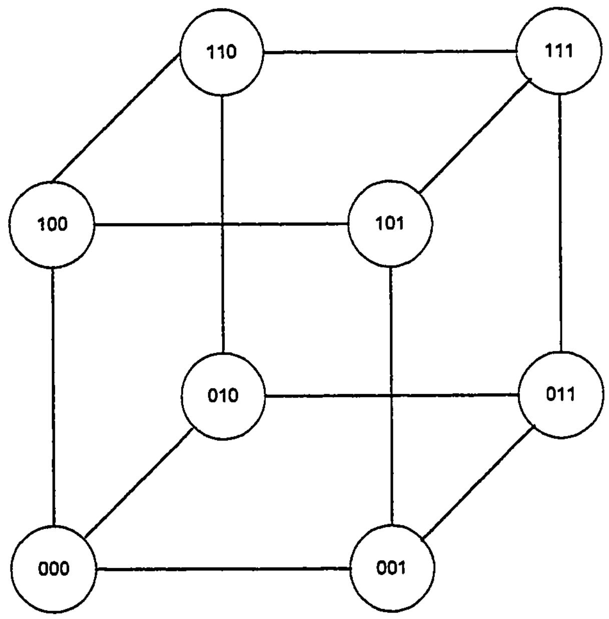


Figure 01.2-3 Hypercube network

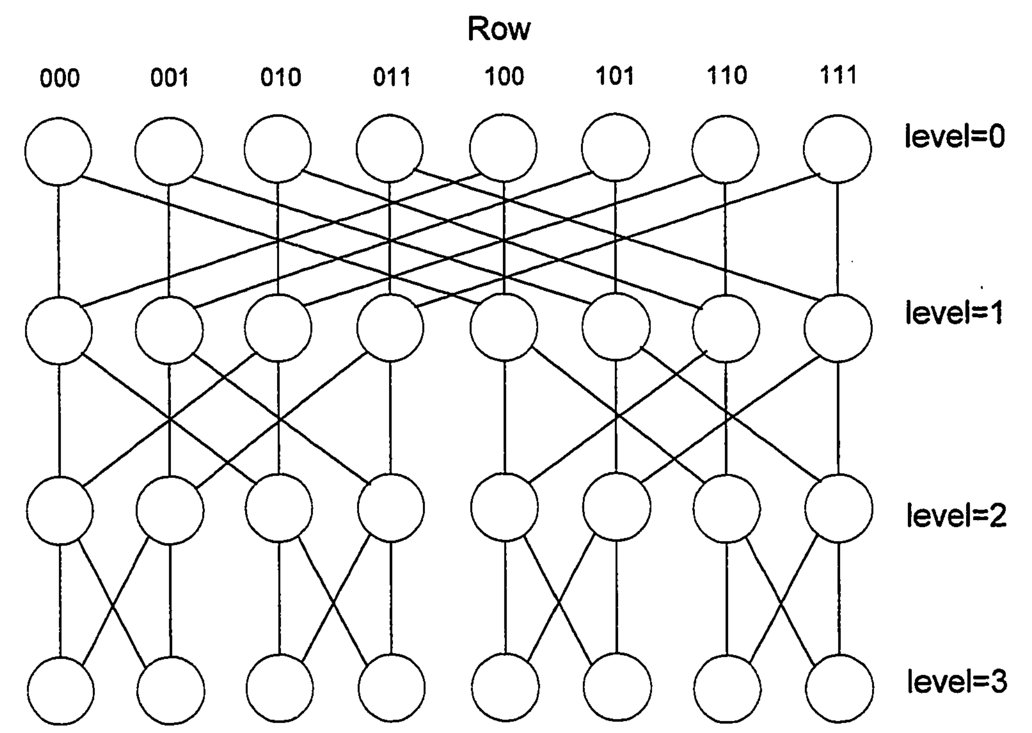


Figure 01.2-4 Butterfly network

1.3 PERFORMANCE ANALYSIS AND NOTATIONS

1.3.1 Space Complexity

Definition 1.3-1 Space/Time Complexity

The *space complexity* of an algorithm is the mount of memory it needs to run to completion. The *time complexity* of an algorithm is the amount of computer time it needs to run to completion.

■

The space needed by any algorithm is seen to be the sum of the following components:

1. A fixed part that is independent of the characteristics (e.g., number, size) of the inputs and outputs. This part typically includes the instruction space (i.e., space for the code), space for simple variables and fixed-size component variables (also called *aggregate*), space for constants, and so on.
2. A variable part that consists of space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that this depends on instance characteristics), and the recursion stack space (insofar as this space depends on the instance characteristics).

The space requirement $S(P)$ of any algorithm P may therefore be written as $S(P) = c + S_P$, where S_p is the instance characteristics and c is a constant.

When analyzing the space complexity of an algorithm, we concentrate solely on estimating S_P (instance characteristics). For any given problem, we need first to determine which instance characteristics to use to measure the space requirements. This is very problem specific, and we resort to examples to illustrate the various possibilities. Generally speaking, our choices are limited to quantities related to the number and magnitude of the inputs to and output from the algorithm. At times, more complex measures of the interrelationships among the data items are used.

1.3.2 Time Complexity

The time $T(P)$ taken by a program P is the sum of the compile time and the run (or execution) time. The compile time does not depend on the instance characteristics. Also, we may assume that the compiled program will be run several times without recompilation. Consequently, we concern ourselves with just the run time of a program. This time is denoted by t_P (instance characteristics).

Because many of the factors, t_P depend on, are not known at the time a program is conceived, it is reasonable to attempt only to estimate t_P. If we knew the characteristics of the compiler to be used, we could proceed to determine the number of additions, subtractions, multiplications, divisions, compares, loads, stores, and so on, that would be made by the code for *P*. So, we could obtain an expression for $t_P(n)$ of the form

$$t_P(n) = c_a ADD(n) + c_s SUM(n) + c_m MUL(n) + c_d DIV(n) + \cdots$$

where *n* denotes the instance characteristics, and c_a, c_s, c_m and so on, respectively, denote the time needed for an addition, subtraction, multiplication, and so on, and *ADD*, *SUB*, *MUL*, and so on, are functions whose values are the numbers of addition, subtractions, multiplications, and so on, that are performed when the code of *P* is used on an instance with characteristic *n*.

Obtaining such an exact formula is in itself an impossible task, since the time needed for an addition, subtraction, multiplication, and so on, often depends on the numbers being added, subtracted, multiplied, and so on. The value of $t_P(n)$ for any given *n* can be obtained only experimentally. The program is typed, compiled, and run on a particular machine. In a multi-user system, the execution time depends on such factors as system load, the number of other programs running on the computer at the time program *P* is run, the characteristics of these other programs, and so on.

Given the minimal utility of determining the exact number of additions, subtractions, and so on, that are needed to solve a problem instance with characteristics given by *n*, we might as well lump all the operations together (provided that the time required by each is relatively independent of the instance characteristics) and obtain a count for the total number of operations. We can go one step further and count only the number of program steps.

A *program step* is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics. For example, the entire statement

$$\textbf{return } a+b+b*c+(d-a)/3*b;$$

of an algorithm could be regarded as a step since its execution time is independent of the instance characteristics (this statement is not strictly true, since the time for multiply and divide generally depends on the numbers involved in the operation)

The number of steps any program statements is assigned depends on the kind of statement. For example, comments count as zero steps; an assignment statement which does not involve any calls to other algorithms is counted as one step; in an iterative statement such as the **for, while,** and **repeat-until** statements, we consider the step counts only for the control part of the statement. The control parts for **for** and **while** statements have the following forms:

```
for $i$ = <expr> to <expr1> do
while (<expr>) do
```

Each execution of the control part of a **while** statement is given a step count equal to the number of step counts assignable to *<expr>*. The step count for each execution of the control part of a **for** statement is one, unless the counts attributable to *<expr>* and *<expr1>* are functions of the instance characteristics. In this latter case, the first execution of the control part of the **for** has a step count equal to the sum of the counts for *<expr>* and *<expr1>* (note that these expressions are computed only when the loop is started). Remaining executions of the **for** statement have a step count of one; and so on.

We determine the number of steps needed by a program to solve a particular problem instance in one of two ways. In the first method, we introduce a new variable, *count*, into the program. This is a global variable with initial value 0. Statements to increment *count* by the appropriate amount are introduced into the program. This is done so that each time a statement in the original program is executed, *count* is incremented by the step count of that statement.

When analyzing a recursive program for its step count, we often obtain a recursive formula for the step count, for example

$$t_P(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_p(n-1) & \text{if } n > 0 \end{cases}$$

These recursive formulas are referred as *recurrence relations.* One way to solving any such recurrence relation is to make repeated substitutions for each occurrence of the function t_p on the right-hand side until all such occurrences disappear:

$$\begin{aligned} t_P(n) &= 2 + t_P(n-1) \\ &= 2 + 2 + t_P(n-2) \\ &= 2(2) + t_P(n-2) \\ &\vdots \\ &= n(2) + t_P(0) \\ &= 2n + 2 \end{aligned}$$

So the step count for the recurrence is $2n + 2$.

The step count is useful in that it tells us how the run time of a program changes with changes in the instance characteristics. From the step count of the recurrence above, we see that if n is doubled, the run time also doubles (approximately); if n increases by a faction of 10, the run time increases by a factor of 10; and so on. So the run time grows *linearly* in n. If A is an algorithm that produced the recurrence above, we say, A is a linear time algorithm (the time complexity is linear in the instance characteristic n).

Definition 1.3-2 Input Size

The input size of any instance of a problem is defined to be the number of words (or the number of elements) needed to describe that instance.

■

The input size for the problem of summing an array with n elements is $n+1$, n for listing the n elements and 1 for the value of n. If the input to any problem instance is a single element, the input size is normally taken to be the number of bits needed to specify that element.

One method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributed by each statement. This figure is often arrived at by first determining the number of steps per execution (s/e) of the statement and the total number of times (i.e., frequency) each statement is executed. *The s/e of a statement is the amount by which the count changes as a result of the execution of the statements.* By combining these two quantities, the total contribution of each statement is obtained. By adding the contributions of all statements, the step count for the entire algorithm is obtained.

1.3.3 Asymptotic Notation

The notation used to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers $\mathbf{N} = \{0,1,2,\ldots\}$. Such notations are convenient for describing the worst-case running-time function $t(n)$, which is usually defined only on integer input sizes. It is sometimes convenient, however, to *abuse* asymptotic notation in a variety of ways. For example, the notation is easily extended to the domain or real numbers or, alternatively, restricted to a subset of the natural numbers. It is important, however, to understand the precise meaning of the notation so that when it is abused, it is not *misused.*

Θ-notation

Definition 1.3-3

For a given function $g(n)$, we denote by $\Theta(g(n))$ the *set of functions*

$$\Theta(g(n)) = \{f(n) : \text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}.$$

■

A function $f(n)$ belongs to the set $\Theta(g(n))$ if there exists positive constants c_1 and c_2 such that it can be "sandwiched" between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large *n*. Although $\Theta(g(n))$ is a set, we write "$f(n) = \Theta(g(n))$" to indicate that $f(n)$ is a member of $\Theta(g(n))$, or "$f(n) \in g(n)$". This abuse of equality to denote set membership may appear confusing, but it has advantages, as we will see later.

For all values of n to the right of n_0, the value of $f(n)$ lies at or above $c_1 g(n)$ and at or below $c_2 g(n)$. In other words, for all $n \ge n_0$, the function $f(n)$ is equal to $g(n)$ to within a constant factor. We say that $g(n)$ is an *asymptotically tight bound* for $f(n)$.

The definition of $\Theta(g(n))$ requires that every member of $\Theta(g(n))$ be *asymptotically nonnegative*, that is, that $f(n)$ be nonnegative whenever n is sufficiently large. Consequently, the function $g(n)$ itself must be asymptotically nonnegative, or else the set $\Theta(g(n))$ is empty. We shall therefore assume that every function used within Θ-notation is asymptotically nonnegative.

O-notation

The Θ-notation asymptotically bounds a function from above and below. When we have only an asymptotic upper bound, we use O-notation.

Definition 1.3-4

For a given function $g(n)$, we denote by $O(g(n))$ the set of functions

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$

■

The O-notation is used to give an upper bound on a function, to within a constant factor. For all values n to the right of n_0, the value of the function $f(n)$ is on or below $g(n)$.

To indicate that a function $f(n)$ is a member of $O(g(n))$, we write $f(n) = O(g(n))$. Note that $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$, since Θ-notation is a stronger notation than O-notation. Since O-notation describes an upper bound, when we use it to bound the worst-case running time of an algorithm, by implication we also bound the running time of the algorithm on arbitrary inputs as well.

Ω-notation

Just as *O*-notation provides an asymptotic *upper* bound on a function, Ω-notation provides an *asymptotic lower bound.*

Definition 1.3-5

For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions

$$\Omega(g(n)) = \{f(n) : \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}.$$

■

From the definitions 1.3-3, 1.3-4 and 1.3-5, it is easy to prove the following theorem. Further details of the proof can be found in [25].

Theorem 1.3-1

For any two functions $f(n)$ and $g(n)$, $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

■

Asymptotic notation in equations

Asymptotic notations can be used within mathematical formulas. When the asymptotic notation stands alone on the right-hand side of an equation, as in $n = O(n^2)$, we have already defined the equal sign to mean set membership: $n \in O(n^2)$. In general, however, when asymptotic notation appears in a formula, we interpret it as standing for some anonymous function that we do not care to name. For example, the formula $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that $2n^2 + 3n + 1 = 2n^2 + f(n)$, where $f(n)$ is some function in the set $\Theta(n)$. In this case, $f(n) = 3n + 1$, which indeed is in $\Theta(n)$.

Using asymptotic notation in this manner can help eliminate inessential detail and clutter in an equation. For example, we express the running time of a recurrence as $t(n) = 2t(n/2) + \Theta(n)$. If we are interested only in the asymptotic behavior of $t(n)$, there is no point of specifying all the lower-order terms exactly; they are all understood to be included in the anonymous function denoted by the term $\Theta(n)$. The number of anonymous functions in an expression is understood to be equal to the number of times the asymptotic notation appears.

In some cases, asymptotic notation appears on the left-hand side of an equation as in $2n^2 + \Theta(n) = \Theta(n^2)$. We interpret such equations using the following rule: *No matter how the anonymous functions are chosen on the left of the equal sign, there is a way to choose the anonymous functions on the right of the equal sign to make the equation valid.* Thus, the meaning of our example is that for *any* function $f(n) \in \Theta(n)$, there is *some* function $g(n) \in \Theta(n^2)$ such that $2n^2 + f(n) = g(n)$ for all n. In other words, the right-hand side of an equation provides coarser level of detail than the left-hand side.

o-notation

The asymptotic upper bound provided by the O-notation may or may not be asymptotically tight. The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not. We use o-notation to denote an upper bound that is not asymptotically tight.

Definition 1.3-6

We formally define $o(g(n))$ ("little-oh of g of n") as the set

$$o(g(n)) = \{f(n) : \text{there exist positive constants } c > 0 \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}.$$

■

The definition of O-notation and o-notation are similar. The main difference is that in $f(n) = O(g(n))$, the bound $0 \leq f(n) \leq cg(n)$ holds for *some* constant $c > 0$, but in $f(n) = o(g(n))$, the bound $0 \leq f(n) < cg(n)$ holds for *all* constants $c > 0$. Intuitively, in the o-notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity; that is, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

ω-notation

By analogy, ω-notation is to Ω-notation as o-notation is to O-notation, we use ω-notation to denote a lower bound that is not asymptotically right. One way to define it is by

$$f(n) \in \omega(g(n)) \text{ if and only if } g(n) \in o(g(n))$$

Definition 1.3-7

Formally, however, we define $\omega(g(n))$ ("little-omega of g of n") as the set

$$\omega(g(n)) = \{f(n) : \text{there exist positive constants } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le cg(n) < f(n) \text{ for all } n \ge n_0\}.$$

■

For example, $n^2/2 = \omega(n)$, but $n^2/2 \ne \omega(n^2)$. the relation of $f(n) = \omega(g(n))$ implies that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, if the limit exists. That is, $f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity.

1.4 LITERATURE OVERVIEW

The sequential selection algorithm, found in introductory literature, has a linear run time, that is, the performance is $O(n)$, where n denotes the number of elements –the size of the problem.

Blum, *et. al.* [7] were the first to show that selection could be performed in linear time. On parallel machines, the same selection problem can be performed in $O(\log n \log\log n)$ parallel time.

Richard John Cole [5] gave an optimally efficient parallel algorithm for selection on the EREW PRAM. It requires a linear number of operations and $O(\log n \log^* n)$ time, where $\log^* n$ is defined to be the least I such that $\log^{(i)} n \leq 1$. A modification of the same algorithm runs on the CRCW PRAM that requires a linear number of operations and $O(\log n \log^* n / \log \log n)$ time.

Valiant [2] introduced the parallel computation tree model for studying these tasks where only the cost of making comparisons is considered. The extremal selection result implies that $\log \log n$ parallel steps are necessary for the median. In the deterministic case, Valiant gave upper and lower bounds of $O(\log \log n)$ for extermal selection and also showed how to merge two lists of size n in $O(\log \log n)$.

Cole and Yap [4] gave a deterministic algorithm for finding the k*th* smallest item in a set of n items, running in $O((\log \log n)^2)$ parallel time on $O(n)$ processors in Valiant's comparison model.

Ajtai, *et. al.* [1] showed that in the deterministic comparison model for parallel computation, In n processors can select the k*th* smallest item form a set of n numbers in $O(\log \log n)$ parallel time. With this result all comparison tasks (selection, merging, sorting), now have upper and lower bounds of the same order in both random and deterministic models.

Gereb-Graus and Krizanc [3] contains a survey of complexity results for comparison problems.

Cole and Yap [4] provided a bound of $O((\log\log n)^2)$ for sorting elements. Cole-Yap algorithm recursively returns the *j*th smallest number in the problem set. Cole and Yap agreed: they suggested that their $O((\log\log n)^2)$ upper bound might be optimal because the recursive calls are inherently sequential.

Krizanc, *et. al.* [6] presented a deterministic algorithm for selecting the element of rank k among $N = n^2$ elements, $1 \le k \le N$, on an $n \times n$ mesh-connected processor arrya in $1.45n$ parallel computation steps, using constant sized queues (for large enough n), which was a considerable improvement over the best previous deterministic algorithm that was based upon sorting and requires $2n + o(n)$ steps. They mentioned that their algorithm could be generalized to solve the problem of selection on higher-dimensional meshes.

Plaxton [8] proved an $\Omega(\frac{N}{p}\log p + \log p)$ lower bound for selection on networks that satisfy a particular low expansion property, where N is the size of the problem and p is the number of processors in the network. The class of networks satisfying this property includes such common network families as trees, multidimensional meshes, hypercube, butterfly, and shuffle-exchange networks.

Plaxton [9] and Rajasekaran [10] give efficient algorithms for selection on the hypercube.

Selection problem is closely related to the sorting problem. The extreme case of selection problem turns into sorting. That is, selecting the 1st, 2nd,..., nth elements. Kaufmann, *et. al.* [11] have shown a $2n+o(n)$-step algorithm to sort the elements of the mesh into block snake-like row-major order, using constant-size queues at every node. In the same area, Knude [12] imply a lower bound of $2n-o(n)$ steps for selecting the median at the meddle processor in the mesh, in a model that puts no limit on the power of the processors but requires each processors to hold exactly one packet at all times.

Aggarwal, *et. al.* [13] presented a generalized selection algorithm on a two-dimensional pyramid model. The algorithm finds the kth largest element out on N elements and has a time complexity of $O(N^{\varepsilon})$ and a cost of $O(N^{1+\varepsilon})$, for some ε between 0 and 1.

There exist parallel algorithms for finding the kth largest element from N elements $x_1, x_2, \ldots, x_N$ on Shared Memory Model [14], Mesh-Connected Computers [6, 15], Hypercube and Perfect Shuffle Computer [16]. The algorithm due to Aggarwal [17] has time complexity of $O(\log^2 N)$ using $O(N)$ processors to select the kth largest element on both tree and pyramid models.

Shen [18] presented an optimal parallel algorithm from multiselection that runs in time $O(n^{\varepsilon} \log r)$ on the EREW PRAM with $n^{1-\varepsilon}$ processors, $0 < \varepsilon < 1$. In the special case, when $\varepsilon = \log(\log n \log^* n) / \log n$, his algorithm runs in time $O(\log n \log^* n \log r)$.

The lower bound for the multiselection problem on a sequential machine is $\Theta(n \log r)$ and was established by Fredman and Spencer [19] in the context of heap operations.

Alsuwaiyel [20] presented a simple optimal algorithm to solve the multiselection problem that runs in $O(n^{\varepsilon} \log r)$ time on the EREW PRAM with $n^{1-\varepsilon}$ processors, $0 < \varepsilon < 1$. In the case when $\varepsilon = O(\log \log n / \log n)$, the algorithm's running time becomes $O(\log n \log r)$ using $O(n / \log n)$ processors.

Alsuwaiyel presented for the first time in [21] an optimally efficient algorithm to solve the multiselection problem that runs in time $O(\log n \log r)$ on the EREW PRAM with $O(n / \log n)$ processors.

1.5 SELECTION PROBLEM ON THE RAM

In this section, the selection problem is formally defined and an optimal solution, algorithm, for the selection problem is presented.

1.5.1 The Selection Problem

As it is mentioned earlier, the *selection* problem comes under a category of problems called *Order Statics*. The *k*th order static of a set of *n* elements is the *k*th smallest element. The problem –as mentioned in [2] is :

Input: A set A of n (distinct) numbers and a number k, with $1 \leq k \leq n$.

Output: The element $x \in A$ that is larger than exactly $k-1$ other elements of A.

Definition 1.5-1 [Selection Problem]

Given a sequence $S = \{s_1, s_2, ..., s_n\}$ of numbers listed in arbitrary order and an integer k, where $1 \leq k \leq n$ the problem of selection calls for determining the kth smallest element of S.

■

If S were presented in sorted order, that is, $S = \{s_{(1)}, s_{(2)}, \ldots, s_{(n)}\}$, then selection would be trivial: In one step we could obtain $s_{(k)}$. Of course, we do not assume that this is the case. Not do we want to sort S first and then pick the kth element: This appears to be (and indeed is) a computationally far more demanding task than we need (particularly for large values of n) since sorting would solve the selection problem for *all* values of k, not just one.

Regardless of the value of k, one face is certain: In order to determine the kth smallest element, we must examine each element of S at least once. This establishes a lower bound of $\Omega(n)$ on the number of steps required to solve the problem.

1.5.2 Sequential Selection

The algorithm presented as the solution for the selection problem is recursive in nature. It uses the *divide-and-conquer* approach an $\Theta(n)$ time algorithm was devised. At each stage of the recursion, a number of elements of S are discarded from further consideration as candidates for being the kth smallest elements. This continues until the kth element is finally determined [10]. By $|S|$ we denote the size of a sequence S. Also, let q be a small integer constant.

```
SEQUENTIAL SELECTION ($S$, $k$)

1.  if $|S| < q$
2.          sort $S$ and return the $k$th element directly
3.  else
4.          subdivide $S$ into $|S|/q$ subsequences of $q$ elements each with
5.  end if
6.  Sort each subsequence and determine its median.
7.  Call SEQUENTIAL SELECT recursively to find $m$ the median of the $|S|/q$
    medians found in step 2
8.  Create three subsequences $S_1, S_2$, and $S_3$ of elements of $S$ smaller than, equal to,
    and larger than $m$, respectively
9.  if $|S_1| \geq k$ then
10.         {$k$th smallest element must be in $S_1$}
11. else
12.         if $|S_1| + |S_2| \geq k$ then
13.                 return $m$
14.         else
15.                 call SEQUENTIAL SELECT recursively to find the
                    $(k = |S_1| - |S_2|)$th element of $S_3$
16.         end if
17. end if
```

Algorithm 1.5-1 Sequential Selection on the RAM

Analysis

Lines 1-5: Since q is a constant, sorting S when $|S| < q$ takes constant time. Otherwise, subdividing S requires $c_1 n$ time for some constant c_1

Line 6: Since each of the $|S|/q$ subsets consists of q elements, it can be sorted in constant time. Thus, $c_2 n$ time is also needed for this step for some constant c_2.

Line 7: There are $|S|/q$ medians; hence the recursion takes $t(n/q)$ time.

Line 8: One pass through S creates S_1, S_2, and S_3 given m. therefore this step is completed in $c_3 n$ time for some constant c_3.

Line 9-17: Since m is the median of $|S|/q$ elements, thee are $|S|/2q$ elements larger than or equal to it. Each of the $|S|/q$ elements was itself the median of a set of q elements, which means that it has $q/2$ elements larger than or equal to it. It follows that elements of S are guaranteed to be larger than or equal to m. Consequently, $|S_1| \leq 3|S|/4$. By similar reasoning, $|S_3| \leq 3|S|/4$. A recursive call in this step to SEQUENTIAL SELECTION therefore requires $t(\frac{3n}{4})$.

From the preceding analysis we have

$$t(n) = c_4 n + t(\tfrac{n}{q}) + t(\tfrac{3n}{4}), \text{ where } c_4 = c_1 + c_2 + c_3.$$

If we choose q so that

$$\frac{n}{q} + \frac{3n}{4} < n$$

then the two recursive calls in the procedure are performed on ever-decreasing sequence Any value of $q \geq 5$ will do. Take $q = 5$; thus

$$t(n) = c_4 n + t(\tfrac{n}{5}) + t(\tfrac{3n}{4}).$$

This recurrence can be solved by assuming that

$$t(n) \leq c_5 n \qquad \text{for some constant } c_5.$$

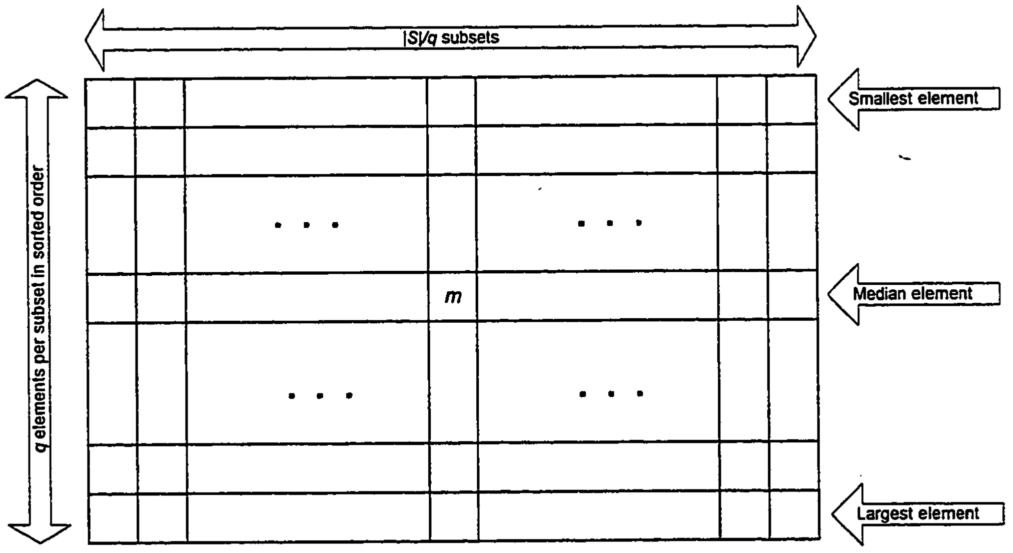


Figure 1.5-1 Main idea behind algorithm Sequential Selection

Substituting, we get

$$t(n) \le c_4 n + c_5(\tfrac{n}{5}) + c_c(\tfrac{3n}{4}) = c_4 n + c_5(\tfrac{19n}{20}).$$

Finally, taking $c_5 = 20c_4$ yields

$$t(n) \le c_5(\tfrac{n}{20}) + c_5(\tfrac{19n}{20}) = c_5 n .$$

thus confirming our assumption. In other words, $t(n) = O(n)$, which is optimal in view of the lower bound derived earlier.

1.6 SUMMARY

The selection problem deals with finding the *k*th smallest element from a set S containing n elements. In the multiselection problem, however, instead of searching for one element, we have to search for a set of elements having the ranks $k_1, k_2, \ldots, k_r$.

Random Access Machine (RAM) consist of a memory with M locations, A processor operating under a sequential algorithm, and a memory access unit (MAU) whose purpose is to create a path from the processor to an arbitrary location in the memory.

Parallel Random Access Machine (PRAM) consist of a number of identical processors $P_1, P_2, \ldots, P_p$ where p is arbitrarily large. A common memory with M locations, arbitrarily large such that $M \ge p$. And a memory access unit (MAU) that allows the processors to gain access to memory.

A network can be viewed as a graph $G = (N, E)$ where each node $i \in N$ represents a processor, and each edge $(i, j) \in E$ represents a two-way communication link. The memory, M, is distributed among N processors. Interconnected networks differ in various aspects; network topology, number of neighbors for each processor, message size, transmission delay, transmission path, processors ...etc. The topologies of interest to this material are the Linear array, the Mesh, the Hypercube, and the Butterfly.

The space complexity of an algorithm is the mount of memory it needs to run to completion. The time complexity of an algorithm is the amount of computer time it needs to run to completion.

The notation used to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers $\mathbf{N} = \{0,1,2,\ldots\}$.

The Θ-notation bounds a function from above and below.

$$\Theta(g(n)) = \{f(n) : \text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}.$$

The O-notation bounds the function from above.

$$O(g(n)) = \{f(n) : \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le c g(n) \text{ for all } n \ge n_0\}.$$

The Ω-notation bounds a function from below.

$$\Omega(g(n)) = \{f(n): \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}.$$

Selection on the RAM machine (sequential machine) can be solved using the divide-and-conquer approach. The original element set S is divided into subsets. The median of each subset is found, then the median of medians is found. The element set is partitioned into three subsets S_1, S_2, and S_3. Two of the three sets are discarded from further consideration, and the same process continues with the remaining elements. All this can be done in $t(n) = O(n)$ time.

Chapter 2

Parallel Computation on PRAM

In the previous chapter, in section 1.2, some computational models were presented. The Parallel Random Access Machine (PRAM) was the first parallel model introduced. In this chapter, this model will be presented in more details, and some PRAM algorithms will be studied. In section 2.1 some definitions and algorithm analysis and notations are presented. In section 2.2, five fundamental PRAM algorithms are presented, namely, the Prefix Computation, Broadcasting, List Ranking, Data Concentration, and Packet Routing. Sorting on the PRAM is followed in section 2.3. The selection problem and its solution are presented in section 2.4. The multiselection problem is tackled in section 2.5. And finally a summary of this chapter is presented in section 2.6.

2.1 INTRODUCTION AND DEFINITIONS

Parallel computers are used primarily to speed up computations. A parallel algorithm can be significantly faster than the best possible sequential solution. This is true in the majority of computational problems. But are such fast solutions truly needed? Indeed, they are. There is a growing number of applications –for example, in science, engineering, business, and medicine– requiring computing speeds that cannot be delivered by any current or future conventional computer. These applications involve processing huge amount of data, or performing a large number of iterations, or both, thus leading to inordinate running times. Parallel computation is the only approach known today that would make these computations feasible.

There is, however, a second less well understood, reason for using parallel computers whose importance has been recently recognized. Consider the following situation:

In a real-time application, a computer needs certain data from the outside world in order to solve a computational problem. It receives s independent data streams at the same time sent by s sources). The data within each stream arrive at a rate that makes it impossible for a sequential computer (even one which operates at the speed of light!) to process more than one stream at a time. Furthermore, it is not feasible to store data arriving from the other $s-1$ streams for later processing, as the data become meaningless if not used immediately. Suppose that precisely one stream contains data useful in solving the current instance of the computational problem, all other streams containing spurious data. The "good" stream is not known in advance and can only be recognized at the end of the computation –that is, when the data it contains leads to a solution.

Clearly a sequential computer has a probability $\frac{1}{s}$ of choosing the good data stream and hence succeeding in solving the problem. In the other hand, a parallel computer with s processors, each dedicated to monitoring one stream always succeeds.

The foregoing example is representative of a host of situations in which the probability of success in performing a computational task is increased through the use of parallel computer. In some extreme cases, a parallel approach can make the difference between guaranteed success and guaranteed failure. It these situations, tackling a problem through parallel computation are not simply the best approach, but rather the only way to obtain a solution.

2.1.1 Algorithm Analysis

A number of criteria are commonly used in evaluation the goodness of an algorithm. The most important of these are the algorithm's running time, how many processors it uses, and the total number of steps it performs. A less widely used, but no less important, criterion is the algorithm's probability of success in completing the task. In those situations were such a criterion is meaningful. These four criteria and the techniques employed in measuring them and interpreting the results of such evaluations are referred to collectively as *algorithm analysis*.

As used in the study of sequential algorithms (RAM algorithms), deriving and expressing measures of an algorithm's behavior are greatly simplified if certain notations are used. Let $f(n)$ and $g(n)$ be functions from the positive integers to the positive real numbers. Then

- The function $g(n)$ is said to be of *order at least* $f(n)$, denoted $\Omega(f(n))$, if there are positive constants k and n_0 such that $g(n) \geq kf(n)$ for all $n \geq n_0$.
- The function $g(n)$ is said to be of *order at most* $f(n)$, denoted $O(f(n))$, if there are positive constants k and n_0 such that $g(n) \leq kf(n)$ for all $n \geq n_0$.

A parallel algorithm typically uses two kinds of elementary steps:

- *Computational steps*: A computational step is a basic arithmetic or logical operation performed on one or two data within a processor. Examples include adding, comparing, and swapping two numbers.
- *Routing steps*: A routing step is used by an algorithm to move a datum of constant size from one processor to another, via the shared memory through the links connecting the processors.

In general, a computational step requires a constant number of *time units*, where as a routing step depends on the distance between the processors. Exactly how long does a routing step take? The answer depends on whether the processors share a common memory or communicate via direct links.

We first consider the case where the processors communicate through a shared memory. If processor P_i wishes to send a datum d to processor P_j, it writes d in some memory location, which is then read by. This involves two memory accesses. IN *uniform analysis*, a memory access is assumed to require a constant number of time units, and consequently, so does a routing step. This assumption, through unrealistic, often simplifies the analysis and is most widely used. In *non-uniform* or *discriminating analysis*, by contrast, a memory access is assumed to require $O(\log M)$ time units, where M is the number of memory locations in the shared memory. This assumption is justified by the way memory access mechanisms are actually built.

Let us not turn to the case where processors communicate by sending data across the links joining them. If two processors P_i and P_j are directly connected by a link, then a routing step from P_i to P_j is assumed to take a constant number of time units. If, on the other hand, the two processors are not directly connected, then routing a datum from P_i to P_j require a number of time units linear in the number of links on the shortest path from P_i to P_j.

2.1.2 Running Time

The *running time* of a parallel algorithm is defined as the time required by the algorithm to solve a computations problem. More precisely, it is the time elapsed between the moment the first processor on the parallel computer to begin operating on the input starts and the moment the last processor to end producing the output terminates. We are interested in the *worst case* running time –that is, the time needed by the algorithm when applied to the most difficult instance of the problem (the one which takes most time to solve).

Running time is measured by counting the number of consecutive elementary steps performed by the algorithm (in the worst case), from the beginning to the end of the computation.

Since each step (computational or routing) is assumed to take a constant number of time units, the number of steps is a good theoretical estimate of the actual amount of time that the algorithm will take to solve the problem on a real parallel computer. The number of steps, and hence the running time, of a parallel algorithm is a function of the size of the input and the number of processors used. Moreover, the number of processors is often itself a function of the size of the input. Therefore, for a problem of size n, the worst case running time of a parallel algorithm is denoted by $t(n)$. Henceforth, when we say that an algorithm has a running time of $t(n)$, or takes $t(n)$ time, we mean that $t(n)$ is the number of time units required by the algorithm.

In some cases, we denote by t_p the running time of an algorithm that uses p processors. This is particularly useful when comparing algorithms that solve the same problem with different number of processors. In these cases, the size n of the problem is usually omitted to simplify the notation. Once the running time of an algorithm for a given problem has been derived, it is instructive to compare it to existing lower and upper bounds for the problem.

A *lower bound* on a certain problem gives the minimum number of steps required by an algorithm to solve the problem in the worst case. In parallel computation, a lower bound usually depends on the size and nature of the problem, the type of parallel computer used, and the number of processors involved. The best algorithm known to solve the problem – that is, the algorithm using the fewest number of steps in the worst case, on the other hand, establishes an *upper bound.*

When the upper and lower bounds for a problem coincide (up to a constant multiplicative factor), the algorithm setting the upper bound is said to be asymptotically *time optimal* for the problem, on that particular parallel computer. Otherwise, a faster algorithm may have to be found, or a lower bound of higher value needs to be derived.

2.1.3 Speedup and Slowdown

The primary reason for using parallel algorithms is to speed up sequential computations. It is therefore quite natural to compare the running time of a parallel algorithm designed for a certain problem to that of the best available sequential algorithm for the same problem. Computing a ratio known as the *speedup* usually does this.

Definition2.1-1 [Speedup]

Let t_1 denote the worst case running time of the fastest known sequential algorithm for the problem, and let t_p denote the worst case running time of the parallel algorithm using p processors. Then the speedup provided by the parallel algorithm is

$$S(1,p) = \frac{t_1}{t_p}$$

A good parallel algorithm is one for which this ratio is large.

For example, suppose that we wish to add n numbers stored in memory. In a sequential computer, a number can be read from memory and added to a running sum in one time unit. Therefore, the sum of the n numbers can be computed in n time units. this is optimal, since n access to memory and $n-1$ additions are required.

On a binary tree of processors with $\frac{n}{\log n}$ leaves, and hence and total number of processors equal to $\frac{2n}{\log n}-1$, the parallel algorithm consists of two phases. In the first phase, with all leaves operation in parallel, each leaf sequentially reads $\log n$ numbers from the input and computes their sum. This takes $\log n$ time units. In the second phase, partial results are sent up the tree: Each node receives two sums from its children, adds them, and sends the result to its parent. This continues until the final sum emerges from the root. The second phase takes $\log(\frac{n}{\log n})$ time units. The time required by the algorithm is therefore $k_1 \log n$, for some constant k_1, where $1 < k_1 < 2$. A speedup of $\frac{k_2 n}{\log n}$ is therefore achieved by the algorithm, for some constant k_2, where $\frac{1}{2} < k_2 < 1$.

In the preceding example, the speedup equals (up to a constant factor) the number of processors used. for many computational problems, this is the largest speedup possible; that is, the speedup is at most equal to the number of processors used by the parallel computer. Because this condition is satisfied by so many traditional problems, it has become part of the folklore of parallel computation and is usually formulated as a theorem:

Theorem 2.1-1 [Speedup Folklore Theorem]

For a given computational problem, the speedup provided by a parallel algorithm using *p* processors, over the fastest possible sequential algorithm for the problem, is at most equal to *p*; that is, $S(1,p) \leq p$. ■

Proof:

Let the fastest sequential algorithm for the problem require time t_1, and let the parallel algorithm require time t_p. Proceeding by contradiction, assume that $\frac{t_1}{t_p} > p$. Since any parallel algorithm can be simulated on a sequential computer by having the single processor execute the parallel steps serially, the simulation requires $p \times t_p$ time. Because $p \times t_p < t_1$ by assumption, the simulation yields a faster sequential algorithm, thus leading to a contradiction.

▲

The speedup folklore theorem is true, and its "proof" holds, for the majority of *standard* problems in computer science. These problems typically obey very restrictive constraints on input, computation, and output. Examples of such problems are provided by operations on a list of numbers stored in memory, such as adding the numbers, searching for a particular number, sorting the numbers, and so on. In fact, for many of these problems, the speedup provided by a parallel algorithm using p processors is much smaller than p either

- because the problem cannot be decomposed into an appropriate number of independent computations to be executed simultaneously, while keeping all processors sufficiently busy, or
- because the structure of the parallel computer used imposes restrictions that render the desired running time unattainable. Specifically, the communications required

amount the processors within a given model of computation may unduly delay the completion of the task.

For many nontraditional problems, however, the speedup folklore theorem does not hold. In other words, there are situations in which a speedup larger than the number of processors used can be obtained. In order to see this, we must look beyond the narrow perspective provided by conventional computations. There is evidence today that the *nature of computing* is changing and must be viewed in a context much broader than before. With increasing frequency, computers are being asked to process data in applications not conceived of until recently, wherein they interact with their environment, affect it, and often move about it freely and autonomously. In these situations, computation can no longer be regarded solely as the process of evaluating a function of a given input, the traditional definition. For example, it may be the case that each input arrives in real time or varies with time, or it may be that each output affects the next input or has to meet a certain deadline. In these conditions, it is obvious that the speedup folklore theorem fails simply because it no longer makes sense.

Another concept that is useful in studying the running time of parallel algorithms is what we call *slowdown* (by contrast with speedup). Slowdown measures the effect on running time of reducing the number of processors on a parallel computer. Naturally, one would expect the running time of an algorithm to increase as the number of processors decreases. The question is, how much slower is a parallel algorithm solving a problem with fewer processors? The traditional answer to this question has given rise to a second folklore theorem:

Theorem 2.1-2 [Slowdown Folklore Theorem]

If a certain computation can be performed with p processors in time t_p and with q processors in time t_q, where $q < p$, then $t_p \leq t_q \leq t_p + \frac{pt_p}{q}$.

■

Proof:

Let W_i denote the number of elementary steps performed simultaneously during the *i*th time unit by the *p*-processor algorithm such that

$$\sum_{i=1}^{t_p} W_i = W$$

In other words, W is the total number of elementary steps performed collectively by the p processors to complete the computation in time t_p. Since not all p processors are necessarily busy at the time, it follows that $W \leq pt_p$. On a smaller computer with only q processors, we can simulate the *i*th time unit of the *p*-processor algorithm by distributing the W_i elementary steps among the q processors, so that each executes $\left\lceil \frac{W_i}{q} \right\rceil$ such steps. The simulation of the entire algorithm requires t_p time units, where

$$t_q \leq \sum_{i=1}^{t_p} \left\lceil \frac{W_i}{q} \right\rceil = \sum_{i=1}^{t_p} \left(\left\lfloor \frac{W_i}{q} \right\rfloor + 1 \right) \leq t_p + \left\lfloor \frac{W_i}{q} \right\rfloor \leq t_p + \frac{pt_p}{q}$$

This completes the "proof".

▲

The slowdown folklore theorem essentially puts an upper bound on the running time of the machine with fewer processors. It says that the running time of the machine with p processors increases at worst by a factor of $1+\frac{q}{p}$ when the number of processors is reduced to q. As with the speedup folklore theorem, evidence from standard computations supports the slowdown folklore theorem: For most conventional problems, the theorem holds.

2.1.4 Cost, Work and Efficiency

Suppose that a parallel algorithm runs in time $t(n)$ in the worst case and uses $p(n)$ processors to solve a problem of size n. An upper bound on the total number of elementary steps execut4ed by this algorithm is given by its cost $c(n)$, which is defined as $c(n) = p(n) \times t(n)$. In other words, the cost of a parallel algorithm is equal to the product of its running time and the number of processors it uses. We say that the cost is an upper bound on the number of steps, since it may be the case that not all $p(n)$ processors are active throughout the $t(n)$ time units. If they are, then of course, $c(n)$ equals the total number of steps executed.

Sometimes the total number of steps performed by the processors of a parallel algorithm can be obtained exactly. This is known as the *work* of the parallel algorithm and is equal to the sum of the steps executed individually by the various processors.

The cost of a parallel algorithm can be used to assess the performance of the algorithm. Consider first those problems to which the speedup folklore theorem applies. In other words, we restrict our attention to those problems for which a *p*-processor parallel algorithm running in time t_p can be simulated on a sequential computer in time $p \times t_p$.

- Assume that a lower bound of $\Omega(f(n))$ is known on the number of steps required in the worst case to solve one such problem of size *n*. If the cost of a parallel algorithm for that problem is $O(f(n))$, then the algorithm is said to be asymptotically *cost optimal*. This is due to the fact that the parallel algorithm can be simulated on a sequential computer. If the total number of steps executed during the simulation matches the lower bound (to within a constant multiplicative factor), then when it comes to *cost*, this parallel algorithm cannot be improved upon: It executes the minimum number of steps possible. One can, of course, *use more processors* in order to *reduce the running time* of a cost-optimal parallel algorithm. Alternatively, one can use *fewer processors*, while retaining cost optimality, if the resulting *higher running time* is acceptable.
- A parallel algorithm is *not cost optimal* if a sequential algorithm exists for solving the same problem, whose worst case running time is smaller than the parallel algorithm's cost. Note that this is true regardless of whether the speedup folklore theorem holds or not.
- Sometimes it is not know whether a parallel algorithm is cost optimal. Let the cost of a parallel algorithm for a given problem match the running time of the

fastest existing sequential algorithm for the same problem. Furthermore, assume that it is not known whether the sequential algorithm is time optimal. In this case, the status of the parallel algorithm with respect to cost optimality is unknown.

A simple way to measure the goodness of a parallel algorithm's cost is to compute a quantity called the *efficiency*. Let t_1 be the worst case running time of the fastest known sequential algorithm for a given problem. Similarly, let t_p be the worse case running time of a p-processor parallel algorithm for the same problem. Then the latter algorithm has a cost of pt_p, and its efficiency is

$$E(1,p)=\frac{t_1}{pt_p}$$

For the problems under consideration –that is, those or which the speedup folklore theorem holds –efficiency is usually at most equal to 1:

- If $E(1,p)<1$, then the parallel algorithm is not cost optimal
- If $E(1,p)=1$, then the parallel algorithm is cost optimal
- If $E(1,p)>1$, them a faster sequential algorithm can be obtained by simulating the parallel one.

Note that if a sequential algorithm is discovered which is faster than the one used to compute $E(1,p)$, then this quantity must be recomputed for all parallel algorithms for the same problem. Suppose that this faster sequential algorithm is obtained by simulating a parallel algorithm. In this special case, the recomputed $E(1,p)$ is 1 for the parallel algorithm that was simulated. However, the later is cost optimal only if the sequential algorithm that it yields is time optimal. When $\frac{t_1}{pt_p} = O(1)$, we take $E(1,p)$ as equal to 1, for simplicity.

2.2 FUNDAMENTAL PRAM ALGORITHMS

A formal description of the PRAM model of computation was presented in sub-section 1.2.2 in the previous chapter. In the following, Prefix Computation and Broadcasting algorithms on the PRAM are presented.

2.2.1 Broadcasting on the PRAM

Let D be a location in memory holding a datum that all p processors need at a given moment during the execution of an algorithm. The broadcasting algorithm assumes the presence of an array A of length p in memory. The array is initially empty and is used by the algorithm as a working space to distribute the contents of D to the processors. Its ith position is denoted by $A(i)$.

Algorithm BROADCAST (D, N, A)

```
1.  Processor $P_i$ reads the value in $D$ stores in its own memory and writes it in $A(1)$
2.  for $i = 0$ to $(\log p - 1)$ do
3.          for $j = 2^i + 1$ to $2^{i+1}$ do in parallel
4.                  Processor $P_j$ reads the value in $A(j - 2^i)$, stores it in its own
```

```
          memory and writes it in $A(j)$
5.      end for
6.  end for
```

Algorithm 2.2-1 Broadcasting on the PRAM

The working of the broadcasting algorithm is illustrated in Figure 2.2–1. The analysis of Algorithm 2.2-1 is as follows; since the number of processors having read D doubles with every iteration, the algorithm terminates in $O(\log n)$ time.

2.2.2 Prefix Computation on the PRAM

We will first formally define the prefix problem, then we present two prefix algorithms, the later one is a cost-optimal algorithm on the PRAM.

Prefix Problem Definition

A set Ψ is given, together with an operation $\oplus$ defined on the elements of Ψ such that:

- The operation $\oplus$ is *binary*; that is, $\oplus$ applies to pairs of elements of Ψ.
- The set Ψ is *closed* under the operation $\oplus$; that is, if x_i and x_j are elements of Ψ, then so is $x_i \oplus x_j$.
- The operation $\oplus$ is *associative*; that is, if x_i, x_j, and x_k are elements of Ψ, then

$$(x_i \oplus x_j) \oplus x_k = x_i \oplus (x_j \oplus x_k) = x_i \oplus x_j \oplus x_k$$

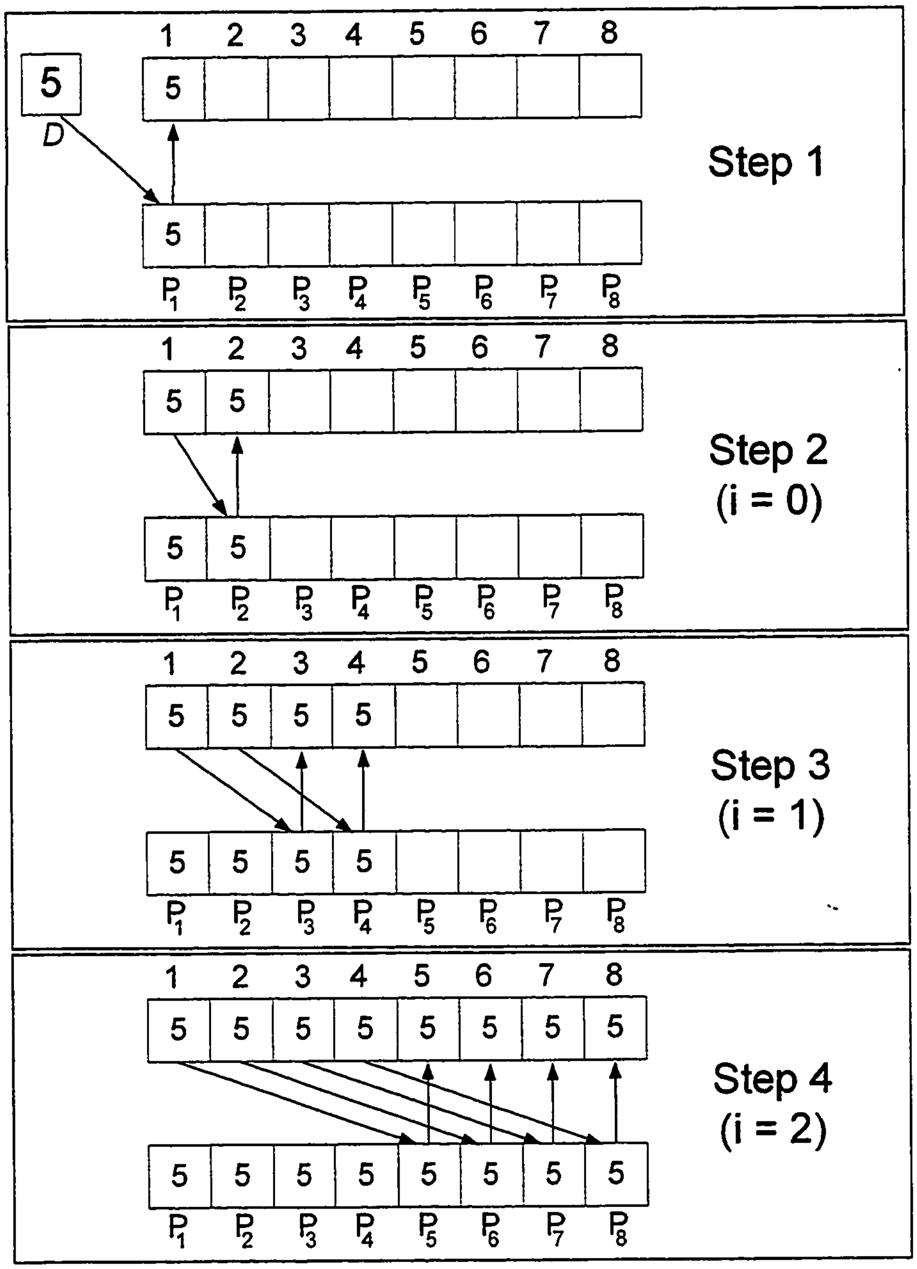


Figure 2.2-1 Distributing a datum to eight processors using the PRAM broadcast algorithm

Definition 2.2-1 [Prefix Computation]

Let $X = \{x_0, x_1, \ldots, x_{n-1}\}$ be a set of *n* different elements. And let

$$\begin{aligned}
s_0 &= x_0 \\
s_1 &= x_0 \oplus x_1 \\
s_2 &= x_0 \oplus x_1 \oplus x_2 \\
&\vdots \\
s_{n-1} &= x_0 \oplus x_1 \oplus \cdots \oplus x_{n-1}
\end{aligned}$$

The process of obtaining $S = \{s_0, s_1, \ldots, s_{n-1}\}$ from $X = \{x_0, x_1, \ldots, x_{n-1}\}$ is known as *prefix computation.*

■

A computation symmetric to prefix computation and referred to as *suffix computation* is defined similarly. Here, a sequence $\{a_0, a_1, \ldots, a_{n-1}\}$, is computed, where

$$\begin{aligned}
a_{n-1} &= x_{n-1} \\
a_{n-2} &= x_{n-1} \oplus x_{n-2} \\
&\vdots \\
a_0 &= x_{n-1} \oplus x_{n-2} \oplus \cdots \oplus x_0
\end{aligned}$$

Unless otherwise stated, we assume in what follows that $\oplus$ takes constant time to be executed. Since computing s_{n-1} involves combining all the x_i, a lower bound on the number of operations required for prefix computation is $\Omega(n)$.

Prefix Computation Algorithm

The prefix computation problem can be solved in $O(n)$ time sequentially. Fortunately, work-optimal algorithms are known for the prefix computation problem on many models of parallel computing. We present a CREW PRAM algorithm that uses $\frac{n}{\log n}$ processors and runs in $O(\log n)$ time. Let n processors $P_0, P_1, \ldots, P_{n-1}$ be available on the PRAM, where n is a power of 2. Initially, the sequence X is stored in shared memory by having P_i read x_i from the input, for $0 \le i \le n-1$, and store it in memory. Processor P_i also sets s_i equal to x_i for $i = 0,1,2,\ldots,n-1$. This takes constant time. The algorithm consists of $\log n$ iterations; during each step, the binary operation $\oplus$ is performed by pairs of processors whose indices are separated by a distance twice that in the previous iteration.

Thus, in the first iteration we compute

$$s_1 \leftarrow s_0 \oplus s_1, s_2 \leftarrow s_1 \oplus s_2, \cdots, s_{n-1} \leftarrow s_{n-1} \oplus s_{n-1}.$$

In the second iteration we compute

$$s_2 \leftarrow s_0 \oplus s_2, s_3 \leftarrow s_1 \oplus s_3, \cdots, s_{n-1} \leftarrow s_{n-3} \oplus s_{n-1}.$$

In the final iteration we compute

$$s_{\frac{n}{2}} \leftarrow s_0 \oplus s_{\frac{n}{2}}, s_{\frac{n}{2}+1} \leftarrow s_1 \oplus s_{\frac{n}{2}+1}, \cdots, s_{n-1} \leftarrow s_{\frac{n}{2}-1} \oplus s_{n-1}.$$

Algorithm PREFIX COMPUTATION

```
1.   for $j = 0$ to $(\log n) - 1$ do
2.       for $i = 2^j$ to $n - 1$ do in parallel
3.           $s_i \leftarrow s_{i-2^j} \oplus s_i$
4.       end for
5.   end for
```

Algorithm 2.2-2 Prefix Computation on the PRAM

It is easy to see that each iteration yields twice as many final values s_i as the previous one. The algorithm therefore runs in $O(\log n)$ time. Since $p(n) = n$, the algorithm's cost is

$$\begin{aligned} c(n) &= p(n) \times t(n) \\ &= n \times O(\log n) \\ & \quad O(n \log n) \end{aligned}$$

The cost is not optimal, in view of the $O(n)$ operations that are sufficient to solve the problem on the RAM model.

Cost-Optimal Prefix Computation Algorithm

Now we will present a cost optimal prefix computation algorithm. Let $k = \log n$ and $m = \frac{n}{k}$, where k and m are rounded appropriately. We use a PRAM with m processors $P_0, P_1, \ldots, P_{n-1}$ and think of input set $\{x_0, x_1, \ldots, x_{n-1}\}$ as being split into m subsets, each of size k, namely,

$$\begin{array}{lcl} Y_0 & = & x_0, x_1, \ldots, x_{k-1} \\ Y_1 & = & x_k, x_{k+1}, \ldots, x_{2k-1} \\ & \vdots & \\ Y_{m-1} & = & x_{n-k}, x_{n-k+1}, \ldots, x_{n-1} \end{array}$$

Processor P_i first reads the sequence Y_i for $0 \le i \le m-1$, stores it in memory, and then applies to it the sequential prefix computation algorithm to obtain

$$s_{ik}, s_{ik+1}, \ldots, s_{(i+1)k-1}$$

where

$$s_{ik+j} = x_{ik} \oplus x_{ik+1} \oplus \cdots \oplus x_{ik+j}$$

for $j = 0,1,\ldots,k-1$. This step is executed simultaneously by all processors. Since each processor executes k iterations, the step requires $O(k)$ time.

The parallel algorithm for prefix computation is now applied by processors $P_0, P_1, \ldots, P_{n-1}$ to the set $\{s_{k-1}, s_{2k-1}, \ldots, s_{n-1}\}$. When this step is completed, s_{ik-1} will be replaced by

$$s_{k-1} \oplus s_{2k-1} \oplus \cdots \oplus s_{ik-1}$$

for $i = 1,2,\ldots,m$. The time required is $O(\log m)$. Finally, P_i, for $1 \le i \le m-1$, performs the step $s_{ik+j} \leftarrow s_{ik-1} \oplus s_{ik+j}$ for $j = 01,2,\ldots k-2$. This step is executed sequentially by all processors operating in parallel (except P_0) and takes $O(k)$ time. The algorithm is given next.

Algorithm COST-OPTIMAL PREFIX

```
1.  for $j = 0$ to $m-1$ do in parallel
2.      $s_{ik} \leftarrow x_{ik}$
3.      for $j = 1$ to $k-1$ do
4.          $s_{ik+j} \leftarrow s_{ik+j-1} \oplus x_{ik+j}$
5.      end for
6.  end for
7.  for $j = 0$ to $(\log m)-1$ do
8.      for $i = 2^j + 1$ to $m$ do in parallel
9.          $s_{ik-1} \leftarrow s_{(i-2^j)k-1} \oplus s_{ik-1}$
10.     end for
11. end for
12. for $i = 1$ to $m-1$ do in parallel
13.     for $j = 0$ to $k-2$ do
14.         $s_{ik+j} \leftarrow s_{ik-1} \oplus s_{ik+j}$
15.     end for
16. end for
```

Algorithm 2.2-3 Cost Optimal Prefix Computation Algorithm on the PRAM

The analysis of the algorithm follows; Lines 1-5 and 12-16 require $O(k)$ time, while lines 7-10 runs in $O(\log m)$ time. Since $k = \log n$ and $\frac{m}{\log n}$, we have

$$\begin{aligned} t(n) &= O(\log n) + O(\log(\tfrac{n}{\log n})) \\ &= O(\log n) \end{aligned}$$

In other words, the reduction in the number of processors from n to $\frac{n}{\log n}$ does not affect the running time. This is due to the associative property of the $\oplus$ operation, which allows the computation to be divided among the processors in the manner described. The cost is

$$\begin{aligned} c(n) &= p(n) \times t(n) \\ &= \tfrac{n}{\log n} \times O(\log n) \\ &= O(n) \end{aligned}$$

which is optimal in the view of the $\Omega(n)$ lower bound derived earlier.

2.3 SORTING ON THE PRAM

First we will present an optimal merging algorithm, then an optimal sorting algorithm is followed, which makes use of the merging algorithm.

2.3.1 Finding the Median of Two Sorted Sequences

Given two sorted sets $A = \{a_1, a_2, \ldots a_r\}$ and $B = \{b_1, b_2, \ldots b_s\}$, where $r, s \geq 1$ let *A.B* denote the set of length $m = r + s$ resulting from merging *A* and *B*. It is required to find the median, that is, the $\lceil \frac{m}{2} \rceil$th element, of *A.B*. Without actually forming *A.B*, the algorithm we are about to describe returns a pair (a_x, b_y) that satisfies the following properties:

- Either a_x or b_y is the median of *A.B*, that is, either a_x or b_y is larger than precisely $\lceil \frac{m}{2} \rceil - 1$ elements and smaller than precisely $\lfloor \frac{m}{2} \rfloor$ elements
- If a is the median, then b_y is either:
    - the largest element in *B* smaller than or equal to a or
    - the smallest element in *B* larger than or equal to a .

Alternatively, if b_y is the median, then a is either

- o the largest element in A smaller than or equal to b_y or
- o the smallest element in A larger than or equal to b_y.

- If more than one pair satisfies 1 and 2, then the algorithm returns the pair for which $x+y$ is smallest.

We shall refer to (a_x, b_y) as the median pair of $A.B$. Thus x and y are the *indices of the median pair*. Not that a is the median of $A.B$ if either

- $a_x > b_y$ and $x+y-1 = \lceil \frac{m}{2} \rceil - 1$ or
- $a_x < b_y$ and $m-(x+y-1) = \lfloor \frac{m}{2} \rfloor$.

otherwise b_y is the median of $A.B$.

The algorithm, described in algorithm 2.3-1, proceeds in stages. At the end of each stage, some elements are removed from consideration from both A and B. We denote by n_A and n_B the number of elements of A and B, respectively, still under consideration at the beginning of a stage and by w the smaller of $\lfloor \frac{n_A}{2} \rfloor$ and $\lfloor \frac{n_B}{2} \rfloor$. Each stage is as follows: The medians a and b of the elements still under consideration in A and B, respectively are compared. If $a \geq b$, then the largest (smallest) w element of $A(B)$ are removed from consideration. Otherwise, that is, if $a < b$, then the smallest (largest) w elements of AB) are removed from consideration. This process is repeated until there is only one element left still under consideration in one or both of the two sets. The median pair is then determined from a small set of candidate pairs. The procedure keeps track of the elements still under consideration by using two pointers to each set: low_A and $high_A$ in A and low_B and $high_B$ in B.

Algorithm TWO-SEQUENCE MEDIAN (A, B, x, y)

```
1.  $low_A \leftarrow 1$
2.  $low_b \leftarrow 1$
3.  $high_A \leftarrow$ r
4.  $high_B \leftarrow$ s
5.  $n_A \leftarrow r$
6.  $n_B \leftarrow s$
7.  while $n_A > 1$ and $n_B > 1$ do
8.          $u \leftarrow low_A + \lceil (high_A - low_A - 1)/2 \rceil$
9.          $v \leftarrow low_B + \lceil (high_B - low_B - 1/2 \rceil$
10.         $w \leftarrow \min\left(\left\lfloor \frac{n_A}{2} \right\rfloor, \left\lfloor \frac{n_B}{2} \right\rfloor\right)$
11.         $n_A \leftarrow n_A - w$
12.         $n_b \leftarrow n_b - w$
13.         if $a_u \geq b_u$
14.                 $high_A \leftarrow high_A - w$
15.                 $low_B \leftarrow low_B + w$
16.         else
17.                 $low_A \leftarrow low_A + w$
18.                 $high_B \leftarrow high_B - w$
19.         end if
20. end while
21. return as $x$ and $y$ the indices of the pair from $\{a_{u-1}, a_u, a_{u+1}\} \times \{b_{v-1}, b_v, b_{v+1}\}$
    satisfying the properties of the median pair mentioned above.
```

Algorithm 2.3-1 Two-Sequence Median

The analysis of algorithm 2.3-1 is as follows. Lines 1-6 and line 21 require constant time. For the while loop, each iteration reduces the smaller of the two sequences by half. For constants c_1 and c_2 the algorithm thus requires $c_1 + c_2 \log(\min\{r,s\})$ time, which is $O(\log n)$ in the worst case.

2.3.2 Fast Merging on EREW PRAM

We now make use of the algorithm presented in section 2.3.2 to construct a parallel merging algorithm for the EREW mode. The algorithm presented in what follows has the following properties:

- It requires a number of processors that is sub-linear in the size of the input and adapts to the actual number of processors available on the EREW computer.
- Its running time is small and varies inversely with the number of processors used.

Given two sorted sets $A = \{a_1, a_2, \ldots a_r\}$ and $B = \{b_1, b_2, \ldots b_s\}$, the algorithm assumes the existence of p processors $P_1, P_2, \ldots, P_p$ where p is a power of 2 and $1 \leq p \leq r+s$. It merges A and B into a sorted set $C = \{c_1, c_2, \ldots, c_{r+s}\}$ in two stages as follows:

Stage 1: Each of the two sets A and B is partitioned into p (possibly empty) subsets $A_1, A_2, \ldots, A_p$ and $B_1, B_2, \ldots, B_p$ such that

- $|A_i| + |B_i| = \frac{(r+s)}{p}$ for $1 \leq i \leq p$

- All elements in $A_i.B_i$ are smaller than or equal to all elements in $A_{i+1}.B_{i+1}$ for $1 \leq i \leq p$.

Stage 2: All pairs A_i and B_i, $1 \leq i \leq p$, are merged simultaneously and placed in C.

The first stage can be implemented efficiently with the help of algorithm 2.3-1. Stage 2 is carried out using a sequential merging algorithm. In what follows, $A[i,j]$ is used to denote the subset $\{a_i, a_{i+1}, \ldots, a_j\}$ of A if $i \leq j$; otherwise $A[i,j]$ is empty. We define $B[i,j]$ similarly.

Algorithm PRAM MERGE(*A*, *B*, *C*)

```
1.   Processor $P_1$ obtains the quadruple (1, r, 1, s)
2.   for $j = 1$ to $\log p$ do
3.       for $i = 1$ to $2^{j-1}$ do in parallel
             // Processor $P_i$ having received the quadruple $(e, f, g, h)$
             // Finds the median pair of two sets
4.           Two-Sequence Median ($A[e, f]$, $B[g, h]$, x, y)
             // Computes four pointers $p_1, p_2, q_1,$ and $, q_2$ as follows
5.           if $a_x$ is the median
6.               $p_1 \leftarrow x$
7.               $q_1 \leftarrow x + 1$
8.               if $b_y \leq a_x$
9.                   $p_2 \leftarrow y$
10.                  $q_2 \leftarrow y + 1$
11.              else
12.                  $p_2 \leftarrow y - 1$
13.                  $q_2 \leftarrow y$
```

```
14.                     end if
15.                 else
16.                     $p_2 \leftarrow y$
17.                     $q_2 \leftarrow y+1$
18.                     if $a_x \le b_y$
19.                         $p_1 \leftarrow x$
20.                         $q_1 \leftarrow x+1$
21.                     else
22.                         $p_1 \leftarrow x-1$
23.                         $q_1 \leftarrow x$
24.                     end if
25.                 end if
26.                 Communicate the quadruple $(e, p_1, g, p_2)$ to $P_{2i-1}$
27.                 Communicate the quadruple $(q_1, f, q_2, h)$ to $P_{2i}$
28.         end for
29. end for
30. for $i = 1$ to $p$ do in parallel
        // Processor $P_i$ having received the quadruple $(a, b, c, d)$
31.     $w \leftarrow 1 + ((i-1)(r+s))/p$
32.     $z \leftarrow \min\{i(r+s)/p, (r+s)\}$
33.     Call Sequential Merge($A[a,b]$, $B[c,d]$, $C[w,z]$)
34. end for
```

Algorithm 2.3-2 PRAM Merging Algorithm

It should be clear that at any time during the execution of algorithm 2.3-2, the subsets on which processors are working are all disjoint. The analysis of the algorithm goes as follows: In line 1, processor $P1$ reads from memory in constant time. During the *j*th iteration of the *for* loop in line 2, each processor involved has to find the indices of the median pair of $\frac{r+s}{2^{j-1}}$ elements. This is done using algorithm Two-Sequence Medina (algorithm 2-3) in $O(\log[\frac{r+s}{2^{j-1}}])$ time, which is $O(\log(r+s))$. The two other operations in lines 26 and 27 take constant time as they involve communications among processors through the shared memory. Since there are $\log p$ iterations of the *for* loop in line 2, lines 1-29 are completed in $O(\log p \times \log(r+s))$ time.

In lines 30-34, each processor merges at most $\frac{r+s}{p}$ elements. This is done using a sequential merge algorithm, which takes $O(\frac{r+s}{p})$. Altogether, the algorithm takes $O(\frac{r+s}{p} + \log p \times \log(r+s))$ time. In the worst case, when $r = s = n$, the time required can be expressed as $t(2n) = O(\frac{n}{p} + \log^2 n)$, yielding a cost of $c(2n) = O(n + p\log^2 n)$, which is optimal when $p \le \frac{n}{\log^2 n}$.

2.3.3 Sorting on the PRAM

In sequential computation, a very efficient approach to sorting is based on the idea of merging successively longer subsets of sorted elements. This approach is more attractive in parallel computation. Algorithm 2.3-3 uses the parallel merging algorithm to sort n number of elements. The idea is simple. Assume that we have a PRAM with p processors $P_1, P_2, \ldots, P_p$ is used to sort a set $S = \{s_1, s_2, \ldots, s_n\}$, where $p \le n$. We begin by distributing the elements of S evenly among the p processors. Each processor sorts its allocated subsequence sequentially using a sequential sorting algorithm, like *quicksort*. The p sorted subsets are now merged pair-wise, simultaneously, using the PRAM Merge algorithm for each pair. The resulting subsets are again merged pair-wise and the process continues until one sorted set of length n is obtained. In what follows, we denote the initial subsets of S allocated to processor P_i by S_i. Subsequently, S_i^k is used to denote the subset obtained by merging two subsets and P_j^k the set of processors that performed the merging.

Algorithm PRAM SORT

```
1.  for $i = 1$ to $p$ do in parallel
        // Processor $P_i$
2.      Reads a distinct subset of $S_i$ of $S$ of size $\frac{n}{p}$
3.      Performs quicksort on $S_i$
4.      $S_i^1 \leftarrow S_i$
5.      $P_i^1 \leftarrow \{P_i\}$
6.  end for
7.  $u \leftarrow 1$
8.  $v \leftarrow p$
9.  while $v > 1$ do
10.     for $m = 1$ to $\lfloor \frac{v}{2} \rfloor$ do in parallel
11.         $P_m^{u+1} \leftarrow P_{2m}^{u} \cup P_{2m}^{u}$
12.         The processors in the set $P_m^{u+1}$ perform PRAM
            Merge($S_{2m-1}^{u}, S_m^{u}, S_m^{u+1}$)
13.     end for
14.     if $v$ is odd
15.         $P_{\lceil v/2 \rceil}^{u+1} \leftarrow P_v^u$
16.         $S_{\lceil v/2 \rceil}^{u+1} \leftarrow S_v^u$
17.     end if
18.     $u \leftarrow u + 1$
19.     $v \leftarrow \lceil \frac{v}{2} \rceil$
20. end while
```

Algorithm 2.3-3 PRAM Sort

The dominating operation is the call to *quicksort* in line 3, which requires $O(\frac{n}{p}\log\frac{n}{p})$ time. During each iteration of the *while* loop in line 9, $\lfloor\frac{v}{2}\rfloor$ pairs of subsets with $n/\lfloor\frac{v}{2}\rfloor$ elements per pair are to be merged simultaneously using $p/\lfloor\frac{v}{2}\rfloor$ processors per pair. Thus, the merging step requires $O([n/\lfloor\frac{v}{2}\rfloor/p/\lfloor\frac{v}{2}\rfloor] + \log(n/\lfloor\frac{v}{2}\rfloor))$, that is, $O(\frac{n}{p}+\log n)$ time. Since the *while* loop in line 9 iterates $\lfloor\log p\rfloor$ times, the total running time of algorithm 2-5 is

$$\begin{aligned} t(n) &= O(\tfrac{n}{p}\log\tfrac{n}{p}) + O(\tfrac{n}{p}\log p + \log n \log p) \\ &= O(\tfrac{n}{p}\log n + \log^2 n) \end{aligned}$$

The cost is given by $c(n) = O(n\log n + p\log^2 n)$ which is optimal for $p \le \frac{n}{\log n}$.

2.4 SELECTION ON THE PRAM

We are now ready to study an algorithm for parallel selection on an EREW PRAM model. The algorithm makes the following assumptions

- A sequence of integers $S = \{s_1, s_2, \ldots s_n\}$ and integer *k*, $1 \le k \le n$, are given, and it is required to determine the *k*th smallest element of *S*.
- The parallel computer consists of *p* processors $P_1, P_2, \ldots, P_p$.
- Each processor has received *n* and computed ε from $p = n^{1-\varepsilon}$, where $0 < \varepsilon < 1$.

- Each of the $n^{1-\varepsilon}$ processors is capable of sorting a set of n^{ε} elements in its local memory.
- Each processor can execute the sequential selection algorithm as well as the broadcasting and the prefix algorithms.
- M is an array in shared memory of length p whose ith position is $M(i)$.

2.4.1 Parallel Selection

The parallel selection algorithm is similar to the sequential one. As in the sequential algorithm, the elements are divided among the processors. The basic idea is to distribute the problem set over all the processors in the machine and try to obtain partial solutions and then compose the final solution from the sub-solutions. Assuming that the problem size is n, that is the input set $S=\{s_1,s_2,...,s_n\}$ contains n elements. On a PRAM with $O(n)$ processors, S can be sorted in $O(\log n)$ time [1]. This means that the cost of finding the kth smallest element is $O(n\log n)$ which is not optimal in view of the $O(n)$ running time of the sequential selection algorithm. The following algorithm uses $O\left(\frac{n}{\log n}\right)$ processors that runs in $O(\log n \log\log n)$ time.

The elements of S are distributed among $\frac{n}{\log n}$ processors, by dividing S into $\frac{n}{\log n}$ subsets $A_1, A_2,..., A_{\frac{n}{\log n}}$ each subset contains $\log n$ elements. Each processor P_i, $1 \le i \le \frac{n}{\log n}$ finds the median m_i of A_i. Then the median m of medians $\{m_1, m_2,..., m_{\frac{n}{\log n}}\}$ is found and the whole set S is partitioned into three subsets S_l, S_e, and S_g and depending on the value of k, two of the sets will be ignored and the same process is repeated.

Algorithm PRAM SELECT (S, k, a)

```
1.  found ← false
2.  while($|S| > \frac{n}{\log n}$ and not found)do
3.          Divide $S$ into subsequences $A_i, i = 1,2,\ldots,\frac{|S|}{\log|S|}$ each consisting of $\log|S|$
            elements of $S$.
4.          for $i = 1$ to $\frac{|S|}{\log|S|}$ do in parallel
5.                  Find the median $m_i$ of $A_i$
6.          end for
7.          Find the median $m$ of $\{m_1, m_2, \ldots, m_{\frac{n}{\log n}}\}$
8.          Create the subsequences $S_l, S_e$, and $S_g$
9.          if $k = |S_l| + 1$ then
10.                 $a \leftarrow m$
11.                 found ← true
12.         else
13.                 if $k < |S_l|$ then
14.                         $S \leftarrow S_l$
15.                 else
16.                         $S \leftarrow S_g$
17.                         $k \leftarrow k - |S_l| - 1$
18.                 end if
19.         end if
20. end while
21. if not found then
22.         (3.1) Sort the sequence $S$
23.         (3.2) $a \leftarrow$ kth element of $S$
24. end if
```

Algorithm 2.4-1 Parallel Selection PRAM SELECT(S, k, a)

The analysis of the algorithm follows: Steps in lines 1, 3 and 9-19 take constant time. The *for* loop in line 4 uses $\frac{|S|}{\log|S|}$ processors each executing the sequential selection algorithm on a subset of length $\log|S|$ and runs in $O(\log|S|)$ time. For finding the median of medians in line 7 we must sort a set of $\frac{|S|}{\log|S|}$ elements using the PRAM Sorting algorithm using $O(\frac{|S|}{\log|S|})$ processors. This requires $O(\log(\frac{|S|}{\log|S|})) = O(\log|S|)$ time. Creating each of S_1 and S_3 by means of array packing is a prefix computation requiring $O(\frac{S}{S})$ processors to complete in $O(\log|S|)$ time. Therefore, each iteration of the *while* loop in line 2 uses $O(\frac{S}{S})$ processors and runs in $O(\log|S|)$ time.

Since $|S| \leq n$, it follows that the processor and time requirements of one iteration of the *while* loop in line 2 are $O(\frac{n}{\log n})$ and $O(\log n)$ respectively. Because the *while* loop in line 2 is iterated $O(\log\log n)$ times, its total running time is $O(\log n \log\log n)$. Similarly, the number of elementary steps (operations) executed during one iteration of the *while* loop is

$$O\left(\frac{|S|}{\log|S|}\right) \times O(\log|S|) = O(|S|)$$

Therefore, the total number of operations executed over all iterations of the *while* loop is

$$O\left(n + \left(\tfrac{3}{4}\right)n + \left(\tfrac{3}{4}\right)^2 n + \left(\tfrac{3}{4}\right)^3 n + \cdots + \left(\tfrac{3}{4}\right)^{O(\log\log n)} n\right) = O(n)$$

Lines 21-24 require $O(\log n)$ time and $O(\frac{n}{\log n})$ processors and executes a total of $O(n)$ elementary steps. To sum up, algorithm 2 has a running time of $t(n) = O(\log n \log\log n)$ and uses $O(\frac{n}{\log n})$ processors with a total cost of $O(n \log\log n)$.

2.5 MULTISELECTION ON THE PRAM

Let S be a set of n elements drawn from a linearly ordered set, and let $K = \{k_1, k_2, ..., _r\}$ be a set of positive integers between 1 and n, that is a set of ranks. We aim to select the k_i th smallest element for all values of $i, 1 \le i \le r$. It is clear that multiselection problem is an extension to the classical selection problem. By setting $r = 1$ in the multiselection problem we obtain the classical selection problem. If $r = n$, then the problem is tantamount to the problem of sorting. It is not difficult to see that $\Omega(n \log r)$ is the lower bound for the sequential multiselection problem. From the previous section it is clear that an optimal parallel algorithm for multiselection that runs in time $O(n^{\varepsilon} \log r)$ on the EREW PRAM with $n^{1-\varepsilon}$ processors where $0 < \varepsilon < 1$. A slight modification or the parallel quick-sort algorithm results in an optimal algorithm for the multiselection problem.

2.5.1 Adaptive Multiselection

In the following *select(S, w)* refers to the sequential selection algorithm presented in section 1.5.

Algorithm MSELECT(S, K)

```
1.  **if** $K$ is not empty
2.          **if** $K = \{k\}$ **then**
3.                  **return** *select*($S$, $k$)
4.          **else**
5.                  $r = |K|$
```

```
6.              $w = k_{r/2}$
7.              SELECT($S$, $w$)
8.              $S_1 = \{x \in S \mid x < S[w]\}$
9.              $S_2 = \{x \in S \mid x > S[w]\}$
10.             $K_1 = \{k_1, k_2, \ldots, k_{w-1}\}$
11.             $K_2 = \{k_{r/2+1} - w, k_{r/2+2} - w, \ldots, k_r - w\}$
12.             MSELECT($S_1, K_1$)
13.             MSELECT($S_2, K_2$)
14.        end if
15.  end if
```

Algorithm 2.5-1 Multiselection Algorithm MSELECT(*S*, *K*)

2.5.2 The PRAM Parallel Algorithm

The PRAM algorithm for multiselection we will discuss now makes use of the parallel selection algorithm found in [18] which runs in time $O(n^{\varepsilon})$ using $p = n^{1-\varepsilon}$ processors. We first start with a set of elements $S = \{x_1, x_2, \ldots, x_n\}$ and a set of ranks $K = \{k_1, k_2, \ldots, k_r\}$. Let q be an appropriately chosen small positive integer greater than 1. Both the set of elements S and the set of ranks K are divided into q parts then the algorithm is recursively called in parallel on the q pairs (S_j, K_j) where $1 \leq j \leq q$.

Algorithm PRAM MULTISELECT(*S*, *K*, *N*)

```
1.    If $|K| \leq q$ then
2.         for $j \leftarrow 1$ to $|K|$ do
3.              SELECT$(S, k_j, p)$
4.              output $S[k_j]$
```

```
5.            end for
6.      else
7.            $u \leftarrow |K|/q$
8.            $w \leftarrow k_0$
9.            for $j \leftarrow 1$ to $q-1$ do
10.                 SELECT$(S, k_{ju}, p)$
11.                 output $S[k_{ju}]$
12.                 $S_j = \{x \in S \mid S[k_{(j-1)u}] < x < S[k_{ju}]\}$
13.                 $K_j = \{k_{(j-1)u+1} - w, k_{(j-1)u+2} - w, \ldots, k_{ju-1} - w\}$
14.           end for
15.           $S_q = \{x \in S \mid x > S[k_{(q-1)u}]\}$
16.           $K_q = \{k_{(q-1)u+1} - w, k_{(q-1)u+2} - w, \ldots, k_r - w\}$
17.           for $j \leftarrow 1$ to $q$ do in parallel
18.                 MSELECT$(S_j K_j, p|S_j|/|S|)$
19.           end for
20.     end if
```

Algorithm 2.5-2 Parallel Multiselection on the PRAM

It is not hard to see that the algorithm above works correctly. We now analyze its time complexity. Each call to Algorithm *select*() in lines 3 and 10 take $O(n^{\varepsilon})$ time using $n^{1-\varepsilon}$ processors. After each call $select(S, k_{ju}, p)$, $1 \le j < q$, in line 10, we extract S_j by marking those elements between (and not including) $S[k_{(j-1)u}]$ and $S[k_{uj}]$, and extracting them using the parallel prefix and compaction using all allocated processors. That is, each processor works on n^{ε} elements and marks those elements between $S[k_{(j-1)u}]$ and $S[k_{uj}]$. Applying parallel prefix and compaction follows this. hence, the time required to construct all S_j's is $O(q(n^{\varepsilon} + \log n^{1-\varepsilon})) = O(qn^{\varepsilon})$. Since K is sorted, K_j is constructed by extracting those elements greater than j_{u-1} and less than j_u in $O(q)$ time. For each recursive call, the number of processors is

$$\frac{p\,|S_j|}{n}=\frac{n^{1-\varepsilon}\,|S_j|}{n}=\frac{|S_j|}{n^{\varepsilon}}.$$

Hence, the ratio of the number of elements to the number of processors is

$$\frac{|S_j|}{|S_j|/n^{\varepsilon}}=n^{\varepsilon}.$$

Each call to the sequential selection algorithm takes $O(n^{\varepsilon})$ time. It follows that the overall running time of the multiselection algorithm is governed by the recurrence:

$$t(r,n)=t(\tfrac{r}{q},n)+O(qn^{\varepsilon})+O(q\log n)$$

The solution to the recurrence is

$$t(r,n)=O(qn^{\varepsilon}\log_q r)=O(n^{\varepsilon}\log r)$$

and hence the cost of the algorithm is $O(n\log r)$.

2.6 SUMMARY

Parallel computers are used primarily to speed up computations. A parallel algorithm can be significantly faster than the best possible sequential solution. A number of criteria are commonly used in evaluation the goodness of an algorithm. The most important of these are the algorithm's running time, how many processors it uses, and the total number of steps it performs.

The *running time* of a parallel algorithm is defined as the time elapsed between the moment the first processor on the parallel computer to begin operating on the input starts and the moment the last processor to end producing the output terminates.

A *lower bound* on a certain problem gives the minimum number of steps required by an algorithm to solve the problem in the worst case. In parallel computation, a lower bound usually depends on the size and nature of the problem, the type of parallel computer used, and the number of processors involved.

Speedup of an algorithm is provided by

$$S(1,p)=\frac{t_1}{t_p}$$

where t_1 denotes the worst case running time of the fastest known sequential algorithm for the problem, and let t_p denotes the worst case running time of the parallel algorithm using p processors.

The cost of a parallel algorithm is equal to the product of its running time and the number of processors it uses. In other words, $c(n)=p(n)\times t(n)$. The total number of steps performed by the processors of a parallel algorithm is known as the *work* of the parallel algorithm. The efficiency of a parallel algorithm is given by $E(1,p)=\frac{t_1}{pt_p}$.

Broadcasting a datum on a PRAM machine requires $O(\log n)$ time. Prefix computation on the PRAM machine can also be done in $O(\log n)$ time with a cost of $O(n)$. Finding the median of two sorted sets requires $O(\log n)$ time in the worst case. Merging on a PRAM machine can be done in $O(\frac{n}{p} + \log^2 n)$ time with a cost of $O(n + p\log^2 n)$. Whereas, merge sorting a set *S* on the PRAM can be performed in $O(\log n)$ time.

Selection on the PRAM machine with $\frac{n}{\log n}$ processors requires $O(\log n \log\log n)$ time with a cost of $O(n \log\log n)$. Multiselection on the other hand requires $O(n^{\varepsilon} \log r)$ with cost $O(n \log r)$.

Chapter 3

Parallel Computation on the Interconnection Networks

A detailed study of the PRAM model was presented in Chapter 2. This chapter covers another kind of parallel computers, which are fairly different from the PRAM. The models presented in this chapter are known as Interconnection networks. Section 3.1 presents the basic difference between the PRAM model of computation and Interconnection Network model of computation. The various different topologies of interconnection networks along with their properties are also presented in Section 3.1. The three main topologies of our interest are the Linear Array, the Mesh, the Butterfly and the Hypercube. Section 3.2 presents a detailed study of some fundamental algorithms on the interconnection networks, namely the prefix computation, broadcasting, data concentration and merging algorithms. A study of optimal sorting algorithms on the three topologies is presented in Section 3.3. Once the necessary background is built up, the reader can proceed to Section 3.4, which presents a universal selection algorithm for the interconnection network. The universal multiselection algorithm on the three types of interconnection networks follows in Section 3.5 (the main goal of this study). The chapter is concluded with a brief summary of the chapter in Section 3.6

3.1 DIFFERENCE BETWEEN PRAM MODEL AND THE INTERCONNECTION NETWORKS

As described in chapter 2, in the PRAM model of computation, a number (say p) of processors work synchronously. They communicate with each other using the common block of global memory that is accessible by all. This global memory is also called common or shared memory. Communication is performed by writing to and/or reading from the common memory. Any two processors i and j can communicate in two steps. In the first step, processor i writes its message into memory cell j, and in the second step, processor j reads from this cell. In contrast, in a fixed connection machine, or the interconnection networks, the communication time depends on the length of the paths connecting the communicating processors. A PRAM model with p processors can be graphically represented as a complete graph as illustrated in the Figure 3.1-1. Each processor is connected to every other processor in the graph. Whereas, in the interconnection networks, a processor is connected to q other processors in the network where $q < p$. In interconnection networks, there are no longer a shared memory; instead, the M locations

A network can be viewed as a graph $G = (N, E)$ where each node $i \in N$ represents a processor, and each edge $(i, j) \in E$ represents a two-way communication link. In the PRAM, all exchanges of data among processors take place through the shared memory. Another way for processors to communicate is via direct links connecting them. There is no longer a shared memory; instead, the M locations of memory are distributed among the p processors. The local memory of each processor now consists of $\frac{M}{p}$ locations. When a processor P_i in an interconnection network wishes to send a datum to processor P_j, it uses the network to route the datum from its memory to that of P_j. The analogy here is a network of roads connecting cities: A datum sent from one processor to another follows a path through the network in the same way that a car travels from one city to another.

Two processors directly connected by a link are said to be *neighbors*. The links used in an interconnection networks *two-way* communication lines: Two processors connected by a link can exchange data simultaneously. In other words, in reality, the link between P_i and P_j in an interconnection network represents two links, namely, one from P_i to P_j and one from P_j to P_i.

The most obvious, and most general, way to connect p processors is to connect each pair by a two-way link, as shown in Figure 3.1-1 for $p = 5$. Each processor has $p - 1$

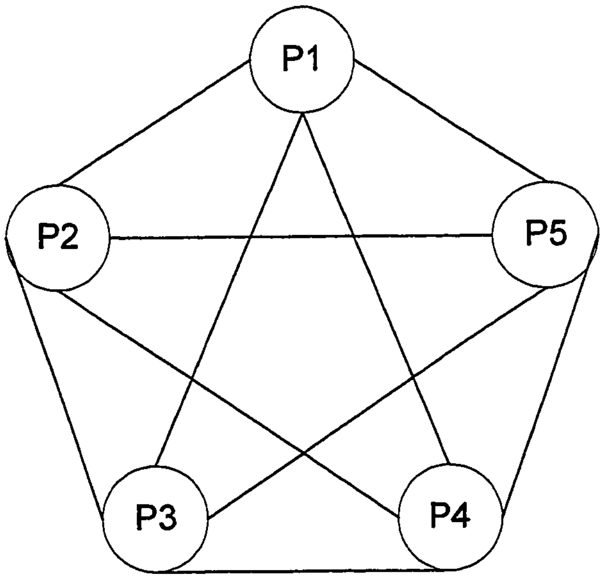


Figure 3.1-1 Graphical Representation of a 5-processor PRAM

neighbors and can send a datum directly to, or receive a datum directly from, any of its neighbors. The complete graph in Figure 3.1-1 simulates a PRAM model with 5 processors, that is, the PRAM model can be viewed as an interconnection network that represents a complete graph. Figure 3.1-1 is known as a complete network. While convenient, such a network is costly and unrealistic for all but the small values of p. Indeed, there are $\frac{p(p-1)}{2}$ links in the network, rendering it infeasible in practice for two reasons, namely, the expense associated with the total number of links and the limit on the number of links that can be physically connected to a processor. As a consequence, more reasonable networks are sought. Fortunately, a small subset of all pair wise connections usually suffices to obtain efficient algorithms in most applications. In some models the number of processor's neighbors is constant, while in other it is a function of p.

Whether the number of neighbors of a processor is constant of function of p, we assume that a processor can send and/or receive data to and from a constant number of neighbors in one time unit. Every message that a processor wishes to transmit is considered a datum of fixed size. If m data are to be sent from one processor to another, then m transmissions are required. Suppose that a processor has x neighbors (where x is either a constant of a function of p). In order to select one of its neighbors for a transmission, the processor needs time that is a function of x. For example, an address of $\log x$ bits requires $O(\log x)$ time to be decoded.

If the link connection two processors has length k, then the time to traverse the link is a function of l. Again, for simplicity, we assume that the datum can travel form one processor to any of its neighbors in constant time. If a processor has x neighbors (where x is either a constant or a function of p), then it needs time that is a function of x to select a neighbor from which to receive data. We take this time to be constant. Each processor has a memory of size $\frac{M}{p}$. We assume this time to be constant too. When processor P_i wishes to send a datum d to processor P_j, which it is not directly connected, the following scheme is used: First P_i sends d to one of its neighbors –for example, P_k. Now P_k receives and stores d and then relays it to one of its own neighbors –for example,

P_l. This continues until d reaches P_j. Since each datum is of constant size, it makes no difference whether this scheme is used or another whereby a path is first established from P to P_j and *then* d is sent.

Unless otherwise stated, we assume that the paths are predefined. In other words, the address of the destination processor is used to find a shortest path from the source processor. This is the responsibility of the algorithm designer. Specifically, the algorithm defines the required path when a processor P is to send a datum to another processor P_j at any given step. Thus, a path from P to P_j is fully specified from beginning to end, prior to that step. When the step is executed, each processor on the path from P to P_j, upon receipt of a datum d, simply forwards d to that specific neighbor indicated by the algorithm. All processors are assumed to "know" p, the total number of processors, as well as the topology of the network. Furthermore, no processor is isolated: There is always at least one path from a source processor to a destination processor.

While handshaking may be a useful feature in practice, it serves no purpose to explicitly include it in the model and allow time for it. We assume that handshaking is done implicitly as part of the communication between P and P_j. In fact, the messages involved in a handshake can themselves be viewed as data, and communicating them involves the same routing mechanisms and requires the same amount of time as sending a regular datum.

We assume that all processors operate synchronously. In one step, requiring constant time, a processor can receive data from a constant number of neighbors, perform a computation, and send data to a constant number of neighbors. Each processor holds a copy of the common algorithm, and all processors execute this algorithm in lockstep fashion. This algorithm may indicate that only a subset of the processors is active, using the indices of the processors. Active processors execute the same step at the same time.

Each processor in the network may be viewed as a RAM: It can perform a number of basic arithmetic or logical operations and has access to a random access memory. Each basic operation requires constant time to be preformed. In addition to a RAM, however, each processor has a number of special registers (called *ports*) that allow it to communicate with its neighbors. Each port is physically connected by a link to a port in another processor.

A number of criteria are used to help determine which topology is best suited for a certain application.

- The *degree* of a processor in a given network topology is defined as the number of neighbors of that processor. The degree of the network is the maximum of all processors degrees in that network. For example, the degree of a binary tree of processors is 3. Degree is an important criterion for assessing a topology and must be considered carefully. In the one hand, a large degree is interesting from a theoretical point of view, since many processors are one step away from any given processor. In the other hand, a small degree is preferable to a large one from a practical point of view, for the reasons given earlier: Having many neighbors is not only expensive, but many also be infeasible.

- The *distance* between two processors P and P_j in a given network topology is the number of links on the shortest path from P to P_j. The *diameter* of the network is the length of the longest distance among all distances between pairs of processors in that network. Since processors need to communicate among themselves, and since the time for a message to go from one processor to another depends on the distance separating them, a network with a small diameter is better than one with a large diameter.
- Although the models that we will study are abstract objects, some of them may represent parallel computers to be implemented. In this light, one network topology is more desirable than another if it is more efficient, more convenient, and more extendable than the other. One particular criterion is the length of the longest link in the network. A network whose links have constant length is usually easier and more efficient to implement.

Next we will study four topologies of networks and their properties. The selection algorithm will be later studies on these topologies, and suitable efficient multiselection algorithm will be developed on them.

3.1.1 Linear Array

In a linear array, each interior processor in a liner array is connected with bi-directional links to its *left-neighbor* and its *right-neighbor*. The outermost processors may have just one connection each, and may serve as input/output points for the entire network.

A linear array, shown in figure 3.1-2, is the simplest example for a *fixed-connection network*. Each processor in the array has a local program control and local storage. The complexity of the local program control and the size of the local storage may vary, although we will usually assume that the local control is simple. (i.e., that it consists of a few operations) and that the local storage is small (i.e., that it can hold a few words of data). At each step, each processor

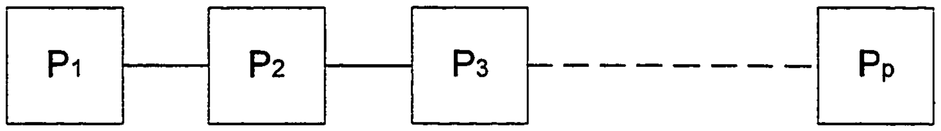


Figure 3.1-2 Linear Array of Processors

- receives input from its neighbors,
- inspect its local storage,
- performs the computation indicated by its local control,
- generates output for its neighbors, and

- updates its local store.

Time is partitioned into *steps* by a *global clock*, so that the entire array operates synchronously. Computation in this fashion is commonly know as *systolic computation*, because data pulses through the network in a manner analogous to the way blood pulses through the body. An array used in this fashion is called a *systolic array*.

The degree of each internal processor in a linear array is 2. While the degree of the two outer most processors is 1. The diameter of a p-processor linear array network is $p-1$ or $O(p)$, because in the worst case, a packet originated from processor P_1 destined to P_p will traverse $p-1$ links.

3.1.2 Mesh

A mesh is an $a \times b$ grid in which there is a processor at each grid point. The edges correspond to the communication links and are bi-directional. Each processor of the mesh can be labeled with a tuple (i, j), where $1 \le i \le a$ and $1 \le j \le b$. Every processor of the mesh is a RAM with some local memory. Hence each processor can perform any of the basic operations such as addition, subtraction, multiplication, comparison, local memory access, and son on, in one unit of time. The computation here is also assumed to be synchronous; that is, there is a global clock and in every time unit each processor completes its intended task. This arrangement can be obtained by arranging the p

arranging the p processors $P_0, P_1, \ldots, P_{p-1}$ into an $m \times m$ array, where $m = p^{\frac{1}{2}}$, as shown in Figure (XXX). The processors in row j and column k are denoted by $P(j,k)$, where $0 \le j \le m-1$ and $0 \le k \le m-1$. A two-way communication line links $P(j,k)$ to its neighbors $P(j+1,k)$, $P(j-1,k)$, $P(j,k+1)$, and $P(j,k-1)$. Processors on the boundary rows and columns have fewer that four neighbors and, hence, fewer connections.

A number of indexing schemes are used for the processors in a mesh. For example, in *row-major* order, processor P_i is placed in row j and column k of the two-dimensional array, where $i = jm + k$ for $0 \le i \le p-1$, $0 \le j \le m-1$, and $0 \le k \le m-1$. In *snakelike row-major* order, processor P_i is placed in row j and column k of the processor array such that $i = jm + k$ when j is even, and $i = jm + m - k - 1$ when j is odd, where i, j, and k are as before.

Like the linear array, the model is simple from a theoretical point of view, as well as being appealing in practice. In it, the maximum degree of processor is four. The topology is *regular*, as all rows and columns are connected to their successors in exactly the same way. The topology is also modular, in the sense that any of its regions can be implemented with the same basic components. The diameter of the mesh is $2p^{\frac{1}{2}} - 2$ or $O(\sqrt{p})$ (i.e., the number of links on the shortest path from the processor in the top left

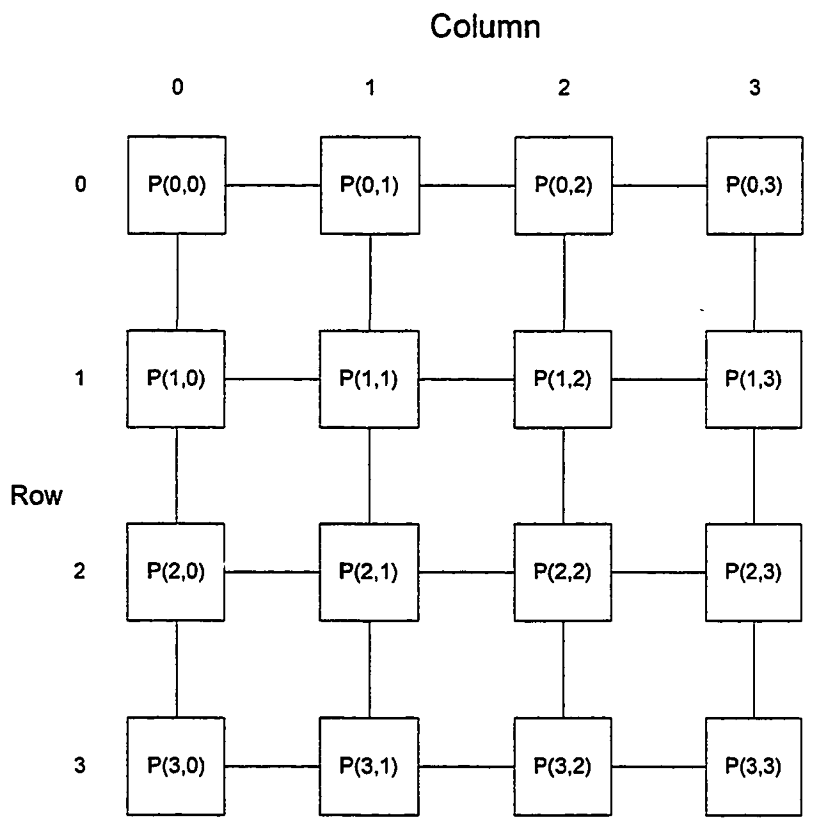


Figure 3.1-3 16-processor Mesh

corner to the processor in the bottom right corner). Another configure of the mesh arranges the processors into *m* rows and *n* columns, where $m \neq n$ and $p = m \times n$.

3.1.3 Hypercube

The *d-dimensional hypercube* has $p = 2^d$ nodes (processors) and $d2^{d-1}$ edges (communication links, or simply, links). In what follows, when the graph of a hypercube is discussed, the words *nodes* and *edges* are used to describe its components; on the other hand, if the object of discussion is the hypercube network, the words *processors* and *links* are used instead. Each processor in the hypercube can be labeled with a d-bit binary number, and two processors are linked with an edge if and only if their binary number differ in precisely one bit. A hypercube of dimension d = 1, 2, 3, and 4 are shown in figure (3.1-3). If v is a d-bit binary number, then the *first bit* of v is the most significant bit of v. The *second bit* of v is the net-most significant bit. And so on. The *dth bit* of v is its least significant bit. Let $v^{(i)}$ stand for the binary number that differs from v only in the ith bit. For example, if v is 1011, then $v^{(3)}$ is 1001. In the same manner, the edges of the hypercube can be naturally partitioned according to the dimensions that they traverse. In particular, an edge is called a *dimension i edge* if it links two nodes that differ in the ith bit position. Any processor v in a hypercube of dimension d is connected only to the processors $v^{(i)}$ for $i = 1,2,\ldots,d$. In a hypercube of dimension 3, for example, the processor 110 is connected to the processors 010, 100, and 111, see figure 3.1-3. The link (or edge) $(v, v^{(i)})$ is called a *level i link.* The edges of any dimension i in a hypercube form a perfect matching for each i, $1 \le i \le \log p$. Note that $d = \log p$. Recall that a perfect matching for any p-node graph is a set of $p/2$ edges that do not share any node. Moreover, removal of the dimension i edges for any $i \le \log p$ leaves two disjoint copies of a $\frac{p}{2}$-node hypercube. Conversely, a p-node hypercube can be constructed from two $\frac{p}{2}$-node hypercubes by simply connecting the jth node of one $\frac{p}{2}$-node hypercube to the jth node of the other for $0 \le j \le \frac{p}{2}$.

Since each processor in the hypercube of dimension *d* is connected to exactly *d* other, the degree of such hypercube is *d*. The hamming distance between two binary numbers *u* and *v* is defined to be the number of bit positions in which they differ.

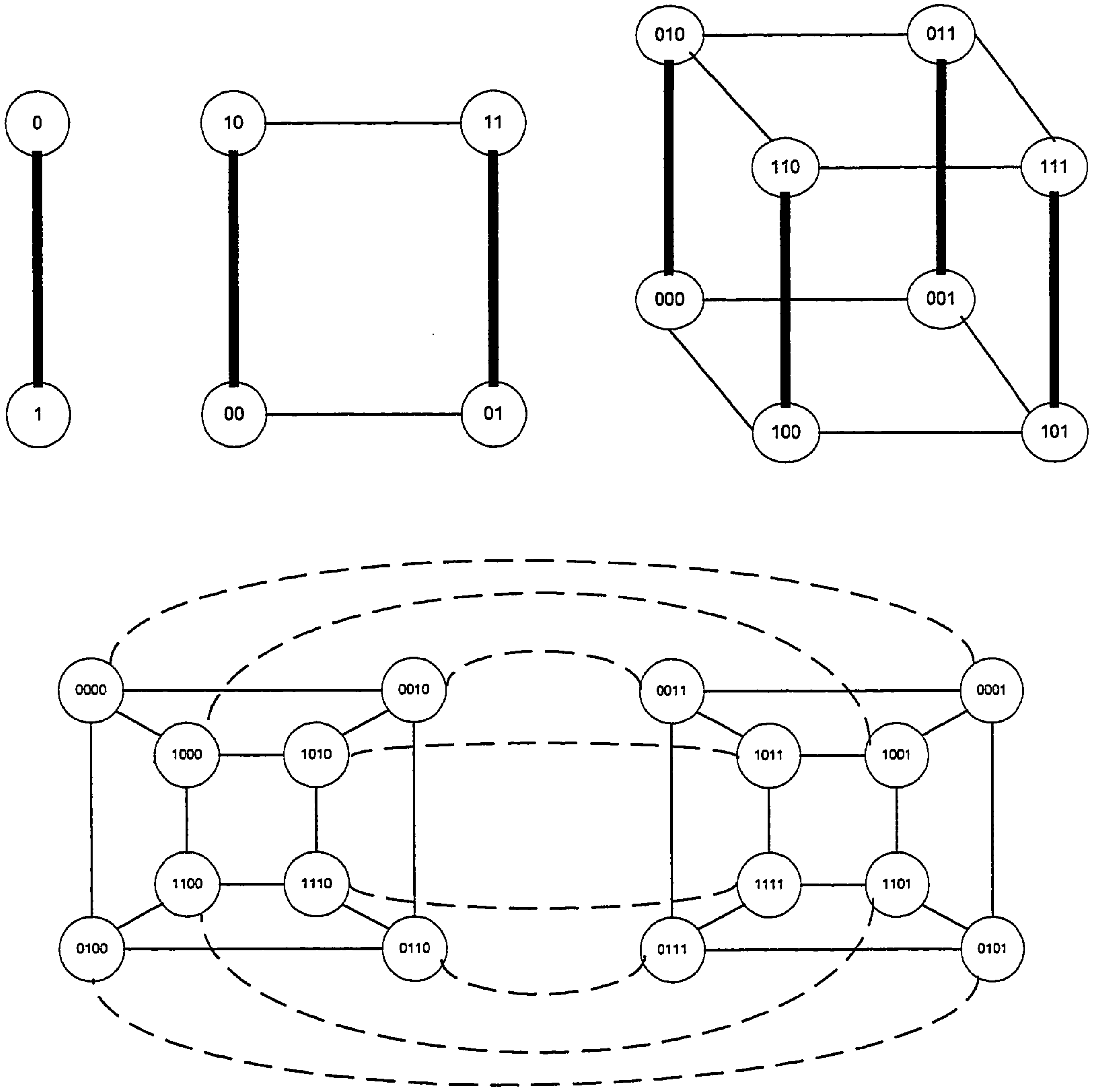


Figure 3.1-4 Hypercubes of dimensions 1, 2, 3, and 4

For any two processors u and v in a hypercube, there is a path between them of length equal to the *Hamming distance* between u and v. For example, there is a path of length 4 between the processors 10110 and 01101 in a five-dimensional hypercube: 10110, 00110, 01110, 01100, 01101. In general, if u and v are any two processors, a path between them (of length equal to their Hamming distance) can be determined in the following way. Let $j_1, j_2, \ldots j_k$ be the bit positions (in increasing order) in which u and v differ. Then, the following path exists between u and v: $u, w_{j_1}, w_{j_2}, \ldots, w_{j_k}, v$, where w_{j_l} has the same bit as v in position 1 through j_l and the rest of the bits are the same as those of u (for $1 \le l \le k$). In other words, for each step in the path, one bit of u is "corrected" to coincide with the corresponding bit of v.

In addition to a simple recursive structure, the hypercube also has many of the other nice prosperities that we would like a network to have. In particular, it has low diameter ($\log p$) and high bisection width ($\frac{p}{2}$). The *diameter* of a network is the maximum distance between any pair of processors. Recall that the *distance* between any pair of processors is the smallest number of links that have to be traversed in order to get from one processor to the other. The *bisection width* of a network is defined as the minimum number of links that have to be removed in order to disconnect the network into two halves with identical (within one) numbers of processors. An interesting aside, it is worth noting that a hypercube can be bisected by removing far fewer than $\frac{p}{2}$ edges are required to bisect the p-node hypercube. For example, consider the partition formed by removing all nodes with size $\left\lfloor \frac{\log p}{2} \right\rfloor$ and $\left\lfloor \frac{\log p}{2} \right\rfloor$. (The *size*, or *weight*, of a processor in the hypercube is the number of 1s contained in its binary string.) A simple calculation reveals that removal of these nodes forms a bisection with $\Theta(\frac{p}{\sqrt{\log p}})$ processors, which is the best possible. It is also worth noting that the hypercube possesses much symmetry. For example, it is *node* and *edge symmetric*. IN other words, by just re-labeling nodes, we can map any node onto any other node, and any edge onto any other edge. More precisely, for any pair of edges (u, v) and (u', v') in a p-node hypercube H, there is an automorphism σ of H such that $\sigma(u) = u'$ and $\sigma(v) = v'$. An *automorphism* of a graph is a one-to-one mapping of the nodes to the nodes such that edges are mapped to edges.

Every processor of the hypercube is a RAM with some local memory and can perform any of the basic operations such as addition, subtraction, multiplication, comparison, local memory access, and so on, in one unit of time. Inter-process communication happens with the help of communication links in a hypercube. If there is no link connecting two given processors that desire to communicate, then communication is enabled using any of the paths connecting them and hence the time for communication depends on the path length. There are two variants of the hypercube. In the first version, know as the *sequential hypercube* or *single-port hypercube*, it is assumed that in one unit of time a processor can communicate with only one of its neighbors. In contrast, the second version, known as the *parallel hypercube* or *multi-port hypercube*, assumes that in one unit of time a processor can communicate with all its d neighbors. Both these versions assume synchronous computations that is, in every time unit, each processor completes its intended task.

Containment (Embedding) of other graphs

One of the most interesting properties of the p-node hypercube network is that it contains p-node linear array as a subgraph. This result holds true even for high-dimensional arrays (meshes) and even wraparound edges are allowed. For example, the embedding of a 4×4 mesh in a 16-node hypercube is shown in Figure (mesh-in-hypercube). Note that p should be a power of 2. As a consequence, the p-node hypercube contains a p-cell linear array (with wraparound) as a subgraph for $p \geq 4$.

The sequence of nodes traversed by a Hamiltonian cycle of a hypercube forms what is known as a *Gray code*. Formally, an r-bit Gray code is an ordering of all r-bit numbers so that consecutive numbers differ in precisely on bit position. A Hamiltonian cycle of an r-dimensional hypercube forms a Gray code since it visits every r-bit binary number in sequence, and since consecutive numbers are linked by an edge of the hypercube (implying that they differ in just one bit). For example, a Hamiltonian cycle of an 8-node hypercube and the associated Gray code are shown in Figure (3-6 black book).

Going back to the embedding of a p-node mesh it is noticeable that not all p-node meshes are subgraphs of $\lceil \log p \rceil$-dimensional hypercubes, it is still possible to find an embedding of any p-node mesh in a $\lceil \log p \rceil$-dimensional hypercube provided that we are allowed to "stretch" the edges of the mesh. The maximum amount that we must stretch any edge to achieve the embedding is called the *dilation* of the embedding. For example, a 3×5 mesh can be embedded in a 16-node hypercube provided that we allow some edges in the mesh to be stretched across two edges of the hypercube. Hence, the mesh can be embedded with dilation 2 in the 16-node hypercube. See Figure (3-7,3-8 black book). In fact, any p-node mesh can be embedded in a $\lceil \log p \rceil$-dimensional hypercube with dilation 2.

Embedding of a binary tree

There are many ways in which a binary tree can be embedded into a hypercube. A q-leaf full binary tree T (where $q = 2^d$ for some integer d) can be embedded into a d-dimensional hypercube. Note that a q-leaf full binary tree has a total of $2q-1$ processors. Hence the mapping cannot be one-to-one. More than one processor of T may have to be mapped into the same processor of the d-dimensional hypercube. If the tree leaves are $0,1,\ldots,p-1$, then leaf i is mapped to the ith processor of the d-dimensional hypercube. Each internal processor of T is mapped to the same processor of the d-dimensional hypercube as its leftmost descendant leaf. See figure (15.7 red book). This embedding could be used to simulate tree algorithms efficiently on a sequential hypercube. If any step of computation involves only one level of the tree, then this step can be simulated in on step on the hypercube [26]. Next we will use this embedding to explain how we can efficiently broadcast on the d-dimensional hypercube.

3.1.4 Butterfly

Although the hypercube is quiet powerful from a computation point of view, there are some disadvantages to its use as and architecture for parallel computation. One of the most obvious disadvantages is that the node degree of the hypercube grows with its size. Several variations of the hypercube have been devised that have similar computational properties but bounded degree. The butterfly network is closely related to the hypercube. Algorithms designed for the butterfly can easily be adapted for the hypercube and vice-versa. In fact, for several problems it is easier to develop algorithms for the butterfly and then adapt them to the hypercube. Which is the case with sorting on the hypercube.

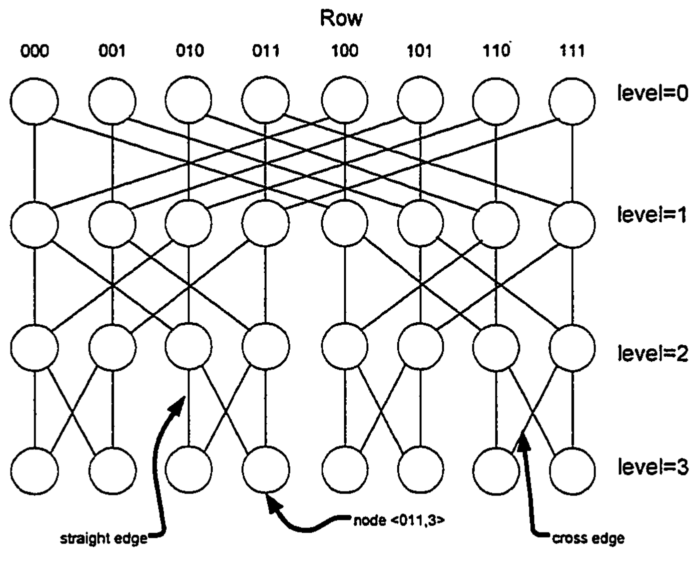


Figure 3.1-5 The Butterfly Network

The d-dimensional butterfly has $(d+1)2^d$ nodes and $d2^{d+1}$ edges, see Figure 3.1-4. The nodes correspond to pairs $\langle w,i \rangle$ where i is the *level* or *dimension* of the node ($0 \le i \le d$) and w is a d-bit binary number that denotes the *row* of the node. Two nodes $\langle w,i \rangle$ and $\langle w',i' \rangle$ are linked by an edge if and only if $i' = i+1$ and either:

1. w and w' are identical, or
2. w and w' differ in precisely the i' th bit.

If w and w' are identical, the edge is said to be a *straight edge*. Otherwise, the edge is a *cross edge*. The butterfly and the hypercube are quite similar in structure. In particular, the *i*th node of the d-dimensional hypercube corresponds naturally to the *i*th row of the r-dimensional butterfly, and an *i*th dimension edge (u, v) of the hypercube corresponds to cross edges ($\langle u,i-1 \rangle$, $\langle v,i \rangle$) and ($\langle v,i-1 \rangle$, $\langle u,i \rangle$) in level i of the butterfly. In effect, the hypercube is just a folded up butterfly (i.e., we can obtain a hypercube from a butterfly by merging all butterfly nodes that are in the same row and then removing the extra copy of each edge). Hence, any single step of p-node hypercube calculation can be simulated in $\log p$ steps on a $p(\log p+1)$-node butterfly by having the *i*th row of the butterfly simulate the operation of the *i*th node of the hypercube for each i. A butterfly can be converted into the hypercube by collapsing each row into a single processor and preserving all the links.

The butterfly has a simple recursive structure. A d-dimensional butterfly contains two $(r-1)$-dimensional butterflies as subgraphs. In figure (3-19 black book), just by removing the level 0 nodes of the d-dimensional butterfly we realize that the resulting graph is simply two $(r-1)$-dimensional butterflies.

The following two properties are important in the analysis of an algorithm in section 3.2.3.2.

Property 1: If the level d processors and incident links are eliminated from the d-dimensional butterfly, two copies of $(d-1)$-dimensional butterfly result. One of these butterflies consists of only even rows and the other consists of only odd rows. We call them even subbutterfly and odd subbutterfly respectively.

Property 2: All processors at level d are connected by a full binary tree. For example, if we trace all the descendants of the processor 00...0 of level zero, the result is a full binary tree with the processors of level d as its leaves. In fact this is true for each processor at level zero.

Another useful property of the d-dimensional butterfly is that the level 0 node in any row w is linked to the level d node in any row w' by a unique path of length d. The path traverses each level exactly once, using the cross edge from level i to level $i+1$ if and only if w and w' differ in the $(i+1)$st bit.

Like the hypercube, the butterfly also has a large bisection width. In particular, the bisection width of the p-node butterfly is $\Theta(\frac{p}{\log p})$. To construct a bisection of this size, simply remove the cross edges from a single level. As a result, we get the following lemma.

Lemma 3.1-1

Each step of a d-dimensional butterfly can be simulated in one step on the parallel version of the d-dimensional hypercube. Also, each step of a d-dimensional butterfly can be simulated in d steps on the sequential version of the d-dimensional hypercube.

■

Any algorithm that runs on a d-dimensional butterfly is said to be a *normal butterfly algorithm* if at any given time, processors in only one level participate in the computation.

Lemma 3.1-2

A single step of any normal algorithm on a d-dimensional butterfly can be simulated in one step on the sequential d-dimensional hypercube.

■

3.2 FUNDAMENTAL ALGORITHMS

In this section we will present four fundamental algorithms on the three interconnection network topologies. These algorithms will be will simplify the analysis of the discussion selection and the multiselection algorithms presented later.

3.2.1 Broadcasting

Broadcasting on the Linear Array and the Mesh

The problem of broadcasting in an interconnection network is to send a copy of a message that originates from a particular processor to a specified subset of other processors. Unless otherwise specified, this subset is assumed to consist of every other processor. Broadcasting is a primitive form of inter-processor communication and is widely used in the design of several algorithms. Let Γ be a linear array with processors $P_1, P_2, \ldots, P_p$. Also let *M* be a message that originates from processor P_1. Message *M* can be broadcast to every other processor as follows. Processor P_1 sends a copy of *M* to processor P_2, which in turn forwards a copy to processor P_3, and so on. This algorithm takes $p-1$ steps and this run time is the best possible. If the processor of message origin is different from processor P, a similar strategy could be employed. If processor P_i is the origin, P_i could start by making two copies of *M* and sending a copy in each direction to each of it neighbors. From the above analysis we get the following lemma.

Lemma 3.2-1 [Broadcasting on the Linear Array]

Broadcasting on a p-processor linear array requires $p-1=O(p)$ steps.

■

In the case of a $\sqrt{p}\times\sqrt{p}$ mesh broadcasting can be done in two phases. If $P_{i,j}$, is the processor of message origin, in phase 1, M could be broadcast to all processors in tow i. IN phase 2, broadcasting of M is done in each column. This algorithm takes $\leq 2(\sqrt{p}-1)$ steps. This can be expressed in the following theorem.

Theorem 3.2-1 [Broadcasting on the Linear Array and the Mesh]

Broadcasting on a p-processor linear array can be completed in $O(p)p$ steps. In a $\sqrt{p}\times\sqrt{p}$ mesh the same can be performed in $\leq 2(\sqrt{p}-1)=O(\sqrt{p})$ time.

■

Broadcasting on the Hypercube

To perform broadcasting on a d-dimensional hypercube, we employ the binary tree embedding. In a binary tree network, assume that the message M to be broadcast is at the root of the tree (i.e., at the processor 00…0). The root makes two copies of M and sends a copy to each of its two children in the tree. Each internal processor, on receipt of a message from its parent, makes two copies and sends a copy to each of its children. This proceeds until all the leaves have a copy of M. Note that the height of this tree is $d=\log p$. Thus in $O(\log p)$ steps, each leaf processor has a copy of M. In this algorithm, computation happens only at one level of the tree at any given time. Thus each step of this algorithm can be run in one time unit on the sequential d-dimensional hypercube. This can be expressed in the following theorem

Theorem 3.2-2 [Broadcasting on the Hypercube]

Broadcasting on a p-processor d-dimensional hypercube can be performed in $O(\log p)$ steps, where $d = \log p$.

■

3.2.2 Prefix Computation

Prefix computation on the Linear array

Prefix computation is a very useful tool that can be used for studying algorithms on parallel topologies. Recall that if we have a set $S = \{s_1, s_2, ..., s_n\}$ a set of elements and a binary *associative* unit time computable operator $\oplus$. Then the prefix computation problem on S has as input n elements. The problem is to compute the n elements

$$x_1, x_1 \oplus x_2, x_1 \oplus x_2 \oplus x_3, \ldots, x_1 \oplus x_2 \oplus x_3 \oplus \cdots \oplus x_n$$

In the case of the linear array with p processors, assume there is an element x_i at processor i (for $i = 1,2,\ldots,p$). We have to compute the prefix of $x_1, x_2, \ldots, x_p$. After this computation, processor i should have the value $\sum_{j=1}^{i} x_j$. One way of performing this computation is as follows. In step 1, processor 1 sends x_1 to the right. In step 2, processor 2 computes $x_1 \oplus x_2$, stores this answer and sends a copy to its right neighbor, and so on, see algorithm 3.2-1.

Linear-Array-Prefix

```
1. for $i = 1$ to $p$ do in parallel
2.         if $i = 1$
3.                 Processor $P_i$ sends $x_i$ to the right in step 1
4.         else if $i = n$
5.                 Processor $P_n$ receives an element (call it $z_{n-1}$) in step $n$ from
                   processor $P_{n-1}$, computes and stores $z_{n-1} \oplus z_n$.
6.         else
7.                 Processor $P$ receives an element (call it $z_{i-1}$) in step $i$ from processor
                   $P_{i-1}$, computes and stores $z_i = z_{i-1} \oplus x_i$, and sends $z_i$ to processor
                   $i+1$
8.         end if
9. end for
```

Algorithm 3.2-1 Prefix Computation on the Linear Array

In general step *i*, processor *i* adds the element received from its left neighbor to x_i, stores the answer, and sends a copy to the right. This will take *p* steps to compute all prefixes. Thus prefix computation on linear array takes $O(p)$ time.

Lemma 3.2-2 [Prefix Computation on the Linear Array]

Prefix computation on a *p*-processor linear array can be performed in $O(p)$ time

■

Prefix computation on the Mesh

An algorithm similar to the one that computes the prefix sum on the linear array can be adopted on the mesh. Consider a $\sqrt{p}\times\sqrt{p}$ mesh in which there is an element of S at each processor. Since the mesh is a two-dimensional structure, there is no natural linear ordering of the processors. We could come up with many possible orderings. Any such orderings of the processors is called an *indexing scheme* like the *row-major*, *column-major*, *snakelike row-major*, etc.

The problem of prefix computing on the mesh can be reduced to three phases in each of which the computation is local to the individual rows or columns.

Phase 1 Row i (for $i=1,2,\ldots,\sqrt{p}$) computes the prefixes of its $\sqrt{p}$ elements. At the end, the processors (i,j) has $y_{(i,j)}=\sum_{q=1}^{j}x_{(i,q)}$

Phase 2 Only column $\sqrt{p}$ computes prefixes of sums computed in phase 1. Thus at the end, processor $(i,\sqrt{p})$ has $z_{(i,\sqrt{p})}=\sum_{q=1}^{i}y_{(q,\sqrt{p})}$. After the computation of prefixes shift them down by one processor; i.e., have processor $(i,\sqrt{p})$ send $z_{(i,\sqrt{p})}$ to processor $(i+1,\sqrt{p})$ (for $i=1,2,\ldots\sqrt{p}-1$).

Phase 3 Broadcast $z_{(i,\sqrt{p})}$ in row i+1 (for $i=1,2,\ldots\sqrt{p}-1$). Node j in row i+1 finally updates its result to $z_{(i,\sqrt{p})}\oplus y_{(i+1,j)}$

Algorithm 3.2-2 Prefix computation on the Mesh

The algorithm assumes the *row-major* indexing scheme. From lemma 3.2-2, the prefix computation in phases 1 and 2 take $\sqrt{p}$ steps each, which is clear if we look at each row as a linear array. The shifting in phase 2 takes one step, and the broadcasting in phase 3 takes $\sqrt{p}$ steps. The final update of the answers needs an additional step. A slight modification to the above algorithm yields prefix computation in snakelike row major order, or any other indexing scheme.

Theorem 3.2-3 [Prefix Computation on the Mesh]

Prefix computation on a $\sqrt{p} \times \sqrt{p}$ mesh in row major order can be performed in $3\sqrt{p} + 2 = O(\sqrt{p})$ steps.

■

Prefix Computation on the Hypercube

For a hypercube model of computation with p processors $P_0, P_2, \ldots, P_{p-1}$ and a number x_i at each processor P_i, $0 \le i \le p-1$. The following algorithm uses a technique called recursive doubling to compute the prefix computation. Each processor P_i has two registers A_i and B_i. Initially, both A_i and B_i contain x_i, $0 \le i \le p-1$. When the algorithm terminates, A_i contains $x_0 \oplus x_1 \oplus \ldots \oplus x_i$. Let i and $i^{(j)}$ be two integers of $\log p$ bits each that differ in the jth bit, where $0 \le j \le (\log p) - 1$, the ith bit being the least significant bit. Algorithm [3.2-3] consists of $\log p$ iterations.

Algorithm HYPERCUBE PREFIX COMPUTATION

```
1.  Step 1: for $j = 0$ to $(\log p) - 1$ do
2.              for all $i < i^{(j)}$ do in parallel
3.                  $A_{i^{(j)}} \leftarrow A_{i^{(j)}} + B_i$
4.                  $B_{i^{(j)}} \leftarrow B_{i^{(j)}} + B_i$
5.                  $B_i \leftarrow B_{i^{(j)}}$
6.              end for
7.      end for
```

Algorithm 3.2-3 Prefix computation on the hypercube

Theorem 3.2-4 [Prefix Computation on the Hypercube]

Prefix computation on the p-processor hypercube can be performed in $O(\log p)$ time. When $p = n$, the cost of the algorithm is $c(n) = O(n \log n)$.

■

Prefix Computation Using Binary Tree Embedding on the Hypercube

Now we make use of the binary tree embedding to perform prefix computation on the d-dimensional hypercube. Let x_i be input at the ith leaf of a 2^d-leaf binary tree. There are two phases in the algorithm, namely, the *forward phase* and the *reverse phase*. In the forward (reverse) phase, data items flow from bottom to top (top to bottom). In each step of the algorithm only on level of the tree is active. In the forward phase of the algorithm, each internal processor computes the sum of all the data in its subtree. Let v be an internal processor and v' be the leftmost leaf in the subtree rooted at v. Then, in the reverse phase of the algorithm, the datum q received by v can be seen to be $\sum_{i=0}^{v'-1} x_i$.

That is, q is the sum of all input data items to the left of v'.

Algorithm: PREFIX COMPUTATION ON A BINARY TREE

> Let $\oplus$ be a binary operation
> **Forward phase**
> The leaves start by sending their data up to their parents. Each internal processor on receipt of two items (say y from its left child and z from its right child) computes $w = y \oplus z$, stores a copy of y and w, and sends w to its parent. At the end of d steps, each processor in the tree has stored in its memory the sum of all the data items in the subtree rooted at this processor. In particular, the root has the sum of all the elements in the tree.
> **Reverse phase**
> The root starts by sending zero to its left child and its y to its right child. Each internal processor on receipt of a datum (say q) from its parent sends q to its left child and $q \oplus y$ to its right child. When the ith leaf gets a datum q from its parent, it computes $q \oplus x_i$ and stores it as the final result.

Algorithm 3.2-4 Prefix Computation on a Binary Tree (as well as on a Hypercube)

The correctness of the algorithm follows. Also both the forward phase and the reverse phase take d steps each. Moreover, at any given time unit, only one level of the tree is active. Thus each step of the algorithm can be simulated in one step on the d-dimensional hypercube. This result agrees with Theorem 3.2-4. Therefore, the prefix computation on a 2^d-leaf binary tree as well as the d-dimensional hypercube can be performed in $O(d) = O(\log p)$ time.

3.2.3 Data Concentration

In a p-processor interconnection network assume that there are $d < p$ data items distributed arbitrarily with at most one data item per processor. The problem of data concentration is to move the data into the first d processors of the network one data item per processor. This problem is also known as *packing*.

In the case of a p-processor linear array, we have to move the data into the processors $P_1, P_2, \ldots, P_d$. On a mesh, we might require the data items to move according to any indexing scheme of our choice. For example, the data could be moved into the first $\left| \frac{d}{\sqrt{p}} \right|$ rows.

Data Concentration on the Linear Array and the Mesh

First performing a prefix computation to determine the destination of each packet and then routing the packet using an appropriate packet routing algorithm achieve data concentration on any network. Note, broadcasting is a special case of packet routing.

Let Γ be a p-processor linear array with d data items. To find the destination of each data item, we make use of a variable x. If processor P_i has a data item, then it sets $x_i = 1$; otherwise it sets $x_i = 0$. Let the prefixes of the sequence $x_1, x_2, \ldots, x_p$ be $y_1, y_2, \ldots, y_p$. If processor P_i has a data item, then the destination of this item is y_i. The destinations for the data items having been determined, they are routed. Prefix computation from lemma 3.2-2 takes p time. Also, from lemma 3.2-1, broadcasting requires p time steps. Thus the total runtime for data concentration on the linear is $2p$.

On the mesh, the same strategy of computing prefixes followed by packet routing can be employed. Prefix computation and packet broadcasting can be done in $O(\sqrt{p})$ steps (c.f. Theorem 3.2-1 and Theorem 3.2-3).

Theorem 3.2-5 [Data Concentration on the Linear Array and the Mesh]

Data concentration (packing) on a p-processor array takes $O(p)$ time. On a $\sqrt{p} \times \sqrt{p}$ mesh, it takes $O(\sqrt{p})$ time.

■

Data Concentration on the Hypercube

In a d-dimensional hypercube, assume that there are $k < p$ data items distributed arbitrarily with at most one datum per processor. The problem of data concentration is to move the data into the processors $P_0, P_1, \ldots, P_{k-1}$ of the d-dimensional hypercube one data item per processor. We will present a normal butterfly algorithm then invoke lemma 3.1-2. The two properties for the butterfly network, mentioned in 3.1.5, will be useful in the analysis of the algorithm.

Assume that the $k \le 2^d$ data items are arbitrarily distributed in level d of the d-dimensional butterfly. At the end, these data items have to be moved to successive rows of level zero. For example, if there are five items in level 3, row $001 \to a$ (this notation means that the processor $\langle 001{,}3 \rangle$ has the item a)., row $010 \to b$, row $100 \to c$, row $101 \to d$, and row $111 \to e$, then at the end, these items will be at level zero and row $000 \to a$, row $001 \to b$, row $010 \to c$, row $011 \to d$, and row $100 \to e$. There are two phases in the algorithm. In the first phase a prefix sums operation is performed to compute the destination address of each data item. In the second phase each packet is routed to its destination using the greedy path from its origin to its destination.

From Theorem 3.2-4, the prefix computation takes $O(d)$ time. During this process the prefix sums are computed on a sequence, $x_0, x_1, \ldots, x_{2^d-1}$, of zeros and ones. The leaf processor P_i sets x_i to one if it has a datum, otherwise to zero.

In the second phase packets are routed using the greedy paths. The claim is that no packet gets to meet any other and hence there is not possibility of link contentions. Consider the first step in which the packet travel from level d to level $d-1$. If two packets meet at level $d-1$, it could be only because they originated from two successive processors of level d. If two packets originate from two successive processors, then they are also destined for two successive processors. In particular, one has an odd row as its destination and the other has an even row. That is, one belongs to the odd subbutterfly and the other belongs to the even subbutterfly. Without the loss of generality assume that the packets that meet at level $d-1$ meet at a processor of the odd subbutterfly. Then it is impossible for one of these two to reach any processor of the even subbutterfly. In summary, no two packets can meet at level $d-1$.

After the first step, the problem of concentration reduces to two sub-problems: concentrating the packets in the odd subbutterfly and concentrating the items in the even subbutterfly. But these subbutterflies are of dimension $d-1$. Thus by induction it follows that there is no possibility of any two packets meeting in the whole algorithm.

The first phase as well as the second phase of this algorithm takes $\Theta(d)=\Theta(\log p)$ time each. Also note that the whole algorithm is normal. We get this lemma.

Lemma 3.2-3 [Data Concentration on the Hypercube]

Data concentration can be performed on a *d*-dimensional butterfly as well as the sequential *d*-dimensional hypercube in $\Theta(d)=\Theta(\log p)$ time.

■

3.2.4 Merging

The problem of merging is to take two sorted subsets as input and produce a sorted set of all elements. To merge two sorted lists $A=a_0,a_1,\ldots,a_{M-1}$ and $B=b_1,b_2,\ldots,b_{M-1}$ into a single list *L* (where *M* is a power of 2), we first partition *A* and *B* into odd and even index sublists. In particular we set

$$\text{even}(A) = a_0, a_1, \ldots, a_{M-2}, \qquad \text{odd}(A) = a_0, a_1, \ldots, a_{M-1}$$

$$\text{even}(B) = b_0, b_1, \ldots, b_{M-} \text{ , and odd}(B) = b_0, b_1, \ldots, b_{M-1}$$

Note that because *A* and *B* are sorted, so are the odd and even index sublists. We next use recursion to merge even(*A*) with odd(*B*) to form a sorted list *C*, and odd(*A*) with even(*B*) to form another sorted list *D*. To form *L* we still have to merge *C* and *D*. At first glance, it appears that the formation of *C* and *D* does not make any progress at all since we will still have to merge these two *M*-element lists to form *L*. The task of merging *C* and *D* is much easier than the task of merging *A* and *B*, however. In particular,

$$C = c_0, c_1, \ldots, c_{M-1} \text{ and } D = d_1, d_2, \ldots, d_{M-1}$$

can be merged by first interleaving the lists to form

$$L' = c_0, d_0, c_1, d_0, \ldots, c_{M-1}, d_{M-1}$$

and then comparing each c_i with the following d_i, and switching the values if they are out of order. The resulting list is sorted.

Odd-Even Merge on the Linear Array

The odd-even merge algorithm is described in algorithm 3.2-5. On a ($p = 2m$)-processor linear array assume that X_1 is input in the first *m* processors and X_2 is input in the next *m* processors. In line 1 of algorithm 3.2-5, X and X_2 are separated into odd and even parts (call them O_1, E_1, O_2, and E_2). This takes $\frac{m}{2}$ steps of data movement.

Algorithm: ODD-EVEN MERGE

1. If $m=1$, merge the sequence with one comparison.
2. Partition X and X_2 into their odd and even parts. That is, partition X into $X_1^{odd}=\{k_1,k_3,\ldots,k_{m-1}\}$ and $X_1^{even}=\{k_2,k_4,\ldots,k_m\}$. Similarly, partition X_2 into X_2^{odd} and X_2^{even}.
3. Recursively merge X_1 with X_2 using m processors. Let $L_1=\{l_1,l_2,\ldots l_m\}$ be the result. At the same time merge X_1 with X_2 using the other m processors to get $L_2=\{l_{m+1},l_{m+2},\ldots,l_{2m}\}$.
4. Form the sequence $L=\{l_1,l_{m+1},l_2,l_{m+1},\ldots,l_m,l_{2m}\}$. compare every pair (l_{m+i},l_{i+1}) and interchange them if they are out of order.

Algorithm 3.2-5 Odd-Even Merge

Next, E_1 and O_2 are interchanged. This also takes $\frac{m}{2}$ steps. In line 2, O is merged recursively with O_2 to get O. At the same time E is merged with E_2 to get E. In line 3, O and E are shuffled in a total of $\leq m$ data movement steps. Finally, adjacent elements are compared and interchanged if out of order. If $t(m)$ is the run time of this algorithm on two sequences of length m each, then we have $t(m)\leq t(\frac{m}{2})+2m+1$ which solves to $t(m)=O(m)$. That is for a linear array of p-processors, odd-even merge requires $O(p)$ time.

Lemma 3.2-4 [Odd-Even Merge on the Linear Array]

Two sorted sequences of length m each can be merged on a p-processor linear array in $O(p)$ time, where $p=2m$.

■

Odd-Even Merge on the Mesh

Consider a $\sqrt{p} \times \sqrt{p}$ mesh. Assume that the two sequences to be merged are input in the first and second halves of the mesh in snakelike row major order where each snake S_1 and S_2 has $\frac{\sqrt{p}}{2}$ columns and $\sqrt{p}$ rows. Algorithm 3.2-6 merges two snakes with l columns each. In what follows we will S denotes the set of elements, S and S_2 denotes two snake-like subsets of the element set S.

Odd-Even Merge on the Mesh

1. If $l = 1$, merge the two snakes using linear array merge.
2. Partition S into its odd and even parts, O_1 and E_1, respectively. Similarly partition S_2 into O_2 and E_2. Parts O, E, O_2, and E_2 are snakes with $\frac{l}{2}$ columns each.
3. Interchange O_2 with E
4. Recursively merge O with O_2 to get the snake O. At the same time merge E with E_2 to get the snake E.
5. Shuffle O with E. compare adjacent elements and interchange them if they are out of order.

Algorithm 3.2-6 Odd-Even Merge on the Mesh

Let $t(l)$ be the run time of the algorithm on two sorted snakes with l columns each. In step 0, we have to merge two sorted columns. Since data can be moved from one column to the other in one step, which takes $O(\sqrt{p})$ [25]. Steps 1, 2, and 4 take $\leq \frac{l}{2}$, $\frac{l}{2}$, and l steps of data movement, respectively. Step 3 takes $t(\frac{l}{2})$ time. Thus, $t(l)$ satisfies $t(l) \leq t(\frac{l}{2}) + 2l$, which on solution implies $t(l) \leq 4l + t(1)$; that is $t(\frac{\sqrt{p}}{2}) = O(\sqrt{p})$.

Therefore, two sorted snakes of size $\sqrt{p} \times \frac{\sqrt{p}}{2}$ each can be merged in time $O(\sqrt{p})$ on a $\sqrt{p} \times \sqrt{p}$ mesh.

Theorem 3.2-6 [Odd-Even Merge on the Mesh]

Two sorted snakes of size $\sqrt{p} \times \frac{\sqrt{p}}{2}$ each can be merged in time $O(\sqrt{p})$ on a $\sqrt{p} \times \sqrt{p}$ mesh.

■

3.3 SORTING ON INTERCONNECTION NETWORK

We are slowly paving our way to simplify the analysis of the selection and multiselection algorithms presented at the end of this chapter. In this section, we present sorting algorithms on the three interconnection network topologies. The main sorting technique that we will study is the Odd-Even Merge sort. Algorithms for this sorting technique are given for the Linear Array, the Mesh, and the Hypercube. First we start by explaining how odd-even merge sort works.

Odd-Even Merge Sort works by recursively merging larger and larger sorted lists. Given an unsorted list of n items, the algorithm starts by partitioning the items into n sublists of length 1. Next, we merge pairs of the unit-length lists in parallel to form $\frac{n}{2}$ sorted lists of length 2. These lists are then merged into $\frac{n}{4}$ sorted lists of length 4, and so on. At the end we merge two sorted lists of length $\frac{n}{2}$ into the final sorted list of length n.

Next we present methods for performing sorting on different topologies.

3.3.1 Sorting on the Linear Array

Odd-Even Transposition Sort

The Odd-Even Transposition Sort Algorithm consists of two steps that are performed repeatedly. In the first step, all odd-numbered processors P_i obtain s_{i+1} from P_{i+1}. If $s_i > s_{i+1}$, then P_i and P_{i+1} exchange the elements they held at the beginning of this step. In the second step, all even-numbered processors perform the same operations as did the odd-numbered ones in the first step. We need $\lceil n/2 \rceil$ repetitions of these two steps in this order. Hence the algorithm terminates with $s_i < s_{i+1}$ for all $1 \le i \le n-1$.

Algorithm ODD-EVEN TRANSPOSITION SORT (*S*, *k*)

```
1.  Step 1: for $j \leftarrow 1$ to $\lceil n/2 \rceil$ do
2.          for $i \leftarrow 1,3,\ldots,2\lfloor n/2 \rfloor - 1$ do in parallel
3.              if $x_i > x_{i+1}$ then
4.                  (1.1.1.1) $x_i \leftrightarrow x_{i+1}$
5.              end if
6.          end for
7.          for $i = 2,4,\ldots,2\lfloor (n-1)/2 \rfloor$ do in parallel
8.              if $x_i > x_{i+1}$ then
9.                  (1.2.1.1) $x_i \leftrightarrow x_{i+1}$
10.             end if
11.         end for
12.     end for
```

Algorithm 3.3-1 ODD-EVEN Transposition Sort

Steps (1.1) and (1.2) require constant time, and they are executed $\lceil n/2 \rceil$ times. The running time of the Algorithm is $t(n) = O(n)$ with a very high cost of

$$c(n) = p(n) \times t(n) = O(n^2)$$

which is not optimal.

The Odd-Even Transposition Sort Algorithm is not attractive because of many reasons. Mainly because it uses a number of processors equal to the size of the input, which is unreasonable. Also, because its cost is too high with a very huge running time when compared with quick sort algorithm.

Merge-Split Sort on Linear Array

Now let us consider the case where $p < n$, that is, the number of processors is less than the problem size. For the linear array topology of processors the MERGE-SPLIT algorithm can achieve better performance than the Odd-Even Transposition Sorting algorithm. In the MERGE-SPLIT algorithm, algorithm 3.3-2, each two neighboring processors P_i and P_{i+1}, initially holding sorted subsequences S_i and S_{j+1} -where $|S_i| = |S_{i+1}| = \frac{n}{p}$, merge their subsequences to produce a sorted sequence $S' = \{s'_1, s'_2, \ldots, s'_{\frac{2n}{p}}\}$. Processor P_i keeps the first $\frac{n}{p}$ elements and processor P_j keeps the rest.

In the first step, each processor P_i sorts S_i using *quicksort*(). In the next step, each odd-numbered processors P_i merges the two subsequences S_i and S_{i+1} into a sorted sequence $S_i' = \{s_1', s_2', \ldots, s_{\frac{2n}{p}}'\}$. It retains the first half of S_i' and assigns its neighbor P_{i+1} the second half. The even-numbered processors in the next step perform the same process. These two steps are preformed alternately $\lceil p/2 \rceil$ times.

Algorithm MERGE SPLIT SORT (*S*, *K*, *N*)

```
1.  for $i \leftarrow 1$ to $p$ do in parallel
2.          quicksort ($S_i$)
3.  end for
4.  for $j \leftarrow 1$ to $\lceil p/2 \rceil$ do
5.      for $i = 1,3,\ldots,2\lfloor p/2 \rfloor - 1$ do in parallel
6.          SEQUENTIAL MERGE ($S_i, S_{i+1}, S_i'$)
7.          $S_i \leftarrow \{s_1', s_2', \ldots, s_{n/p}'\}$
8.          $S_{i+1} \leftarrow \{s_{(n/p)+1}', s_{(n/p)+2}', \ldots, s_{2n/p}'\}$
9.      end for
10.     for $i = 2,4,\ldots,2\lfloor (p-1)/2 \rfloor$ do in parallel
11.         SEQUENTIAL MERGE ($S_i, S_{i+1}, S_i'$)
12.         $S_i \leftarrow \{s_1', s_2', \ldots, s_{n/p}'\}$
13.         $S_{i+1} \leftarrow \{s_{(n/p)+1}', s_{(n/p)+2}', \ldots, s_{2n/p}'\}$
14.     end for
15. end for
```

Algorithm 3.3-2 Merge-Split Sort

The analysis of algorithm 3.3-2 is as follows: Lines 1-3, *quicksort* requires $O((n/p)\log(n/p))$ steps. Sequential Merge and data transformation in Step 2 requires $O(n/p)$. Thus the total running time is

$$t(n) = O((n/p)\log(n/p)) + \lceil p/2 \rceil \times O(n/p)$$

Since $p < n$ the total running time is $t(n) = O((n/p)\log(n/p)) + O(n)$ and thus the cost is $c(n) = (n\log n) + O(np)$ which is optimal when $p \leq \log n$

Lemma 3.3-1 [Odd-Even Transposition Sort on the Linear Array]

The odd-even transposition sort runs in $O(p)$ time on a *p*-processor linear array.

■

Lemma 3.3-2 [Merge-Split Sort on the Linear Array]

The merge-split sort runs in $O(p)$ time on a *p*-processor linear array.

■

3.3.2 Odd-Even Merge Sort on the Mesh

Let $S = \{x_1, x_2, \ldots, x_n\}$ a given set of *n* elements. In brief, odd-even merge sort partitions *S* into two subsets $S_1' = x_1, x_2, \ldots, x_{\frac{n}{2}}$ and $S_2' = x_{\frac{n}{2}+1}, x_{\frac{n}{2}+2}, \ldots, x_n$ of equal length. Subsets S_1' and S_2' are sorted recursively assigning $n/2$ processors to each, and them finally merge using the odd-even merge algorithm given in the previous subsection.

Initially the elements x_i are distributed on the $\sqrt{p} \times \sqrt{p}$ mesh, where $p = n$. We can partition the mesh into four equal parts of size $\frac{\sqrt{p}}{2} \times \frac{\sqrt{p}}{2}$ each. Then we sort each part recursively into snakelike row major order. Then we merge the top two snakes using merge sort at the same time we merge the bottom two snakes using the same algorithm. These merges take $O(\sqrt{p})$. Finally merge these two snakes. This also takes $O(\sqrt{p})$. If $t(l)$ is the time needed to sort an $l \times l$ mesh using the above divided-and-conquer algorithm then we have $t(l) = t(\frac{l}{2}) + O(l)$ which solves to $S(l) = O(l)$.

Theorem 3.3-1 [Odd-Even Merge Sort on the Mesh]

Sorting p elements can be done in $O(\sqrt{p})$ time on a $\sqrt{p} \times \sqrt{p}$ into snakelike row major order.

■

3.3.3 Sorting on the Hypercube

It is easier to implement the Odd-Even merge sort on a butterfly than on a hypercube. Once the technique is developed on the butterfly, using lemma 3.1-2 we can adapt the same algorithm to work on the hypercube.

Odd-Even Merge Sort on the Butterfly

Although it is a bit surprising at first glance, there is a very simple implementation of Odd-Even Merge Sort on a butterfly. To see how the implementation works, we will first show how to merge two $\frac{M}{2}$-element lists $A = a_0, a_1, \ldots, a_{\frac{M}{2}-1}$ and $B = b_1, b_2, \ldots, b_{\frac{M}{2}-1}$ on a $\log M$ -dimensional butterfly.

We start by inputting a_i into node $\langle 0 \mid \text{bin}(i), \log M \rangle$ of the butterfly and b_i into node $\langle 1 \mid \text{bin}(i), \log M \rangle$ of the butterfly for $0 \le i < \frac{M}{2}$. Next we pass the value of a along the straight edge to level $\log M - 1$, and the value of b along the cross edge to level $\log M - 1$, see Figure 3.3-2. It is now time for the recursive part of the algorithm –i.e., merging even(*A*) with odd(*B*) to form *C*, and merging odd(*A*) with even(*B*) to form *D*. A quick look at Figure 3.1-5 reveals that the even rows of our $\log M$ -dimensional butterfly contain a ($\log M - 1$)-dimensional butterfly, as do the odd rows. Moreover, we have already entered the data to the level $\log M - 1$ nodes so that the subbutterfly in the even rows can merge even(*A*) with odd(*B*) to form *C* and so that the subbutterfly in the odd rows can merge odd(*A*) with even(*B*) to form *D*, see Figure 3.3-1, 3.3-2.

Once the lists *C* and *D* are formed, it only remains to interleave them, and then switch the c , d_i pairs that are out of order. As can be seen in Figure 3.3-2, these tasks are easily accomplished in a single step on the butterfly. The reason is that the *C* and *D* lists are already interleaved, and so we only need to have each even-row ($\log M$)-level node pick the minimum of the values output by its ($\log M - 1$)-level neighbors, and each odd-row ($\log M$)-level node picks the maximum of the values output by its ($\log M - 1$)-level neighbors.

If we unwind the recursion, we find that the entire merge takes just $2 \log M$ steps. In the first $\log M$ steps, the data makes one pass through the butterfly, using the straight and cross edges in a preordained way. In particular, we always use straight edges in the top half of the butterfly and cross edges in the bottom half of the butterfly (except when crossing level 1 edges, where we use only straight edges). Thus the net effect of the first $\log M$ steps of the algorithm is to reverse the order of the items in the *B* list. During the last $\log M$ steps, the items make a second pass back through the butterfly, always switching across cross edges whenever adjacent pairs are out of order.

It is straightforward to adapt the previous algorithm so that $\frac{p}{M}$ *M*-elements lists using $2\log M$ steps on a ($\log p$)-dimensional butterfly. The reason is that the first $\log M + 1$

levels of a ($\log p$)-dimensional butterfly (i.e., levels $\log p - \log M$ through $\log N$ form a collection of $\frac{p}{M}$ *M*-dimensional butterflies, so we can assign one merge to each $\log M$ - dimensional butterfly. Bu recursively merging the lists into larger and larger lists, we can thereby sort *p* numbers in

$$2\log 2 + 2\log 4 + 2\log 8 + \cdots + 2\log p = \log p(1 + \log p) = \log^2 p + \log p$$

steps overall on a $\log p$-dimensional butterfly.

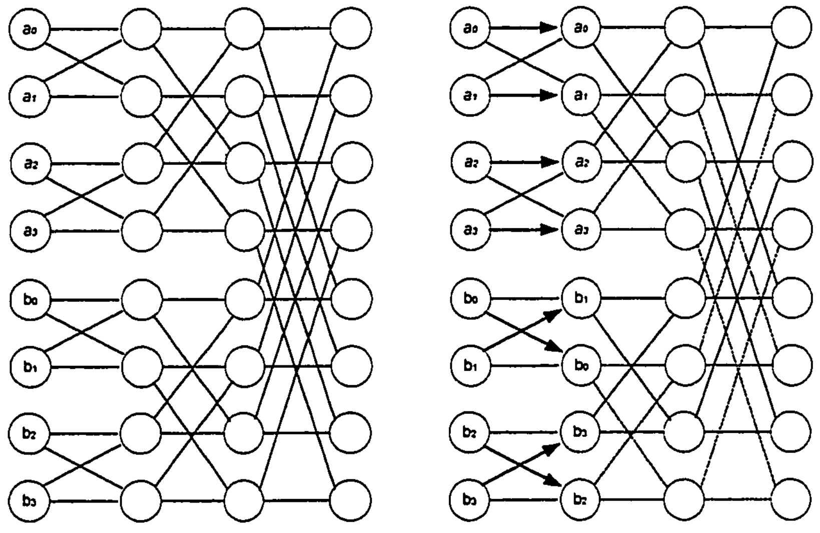


Figure 3.3-1 Implementation of the Odd-Even Merge algorithm on a butterfly

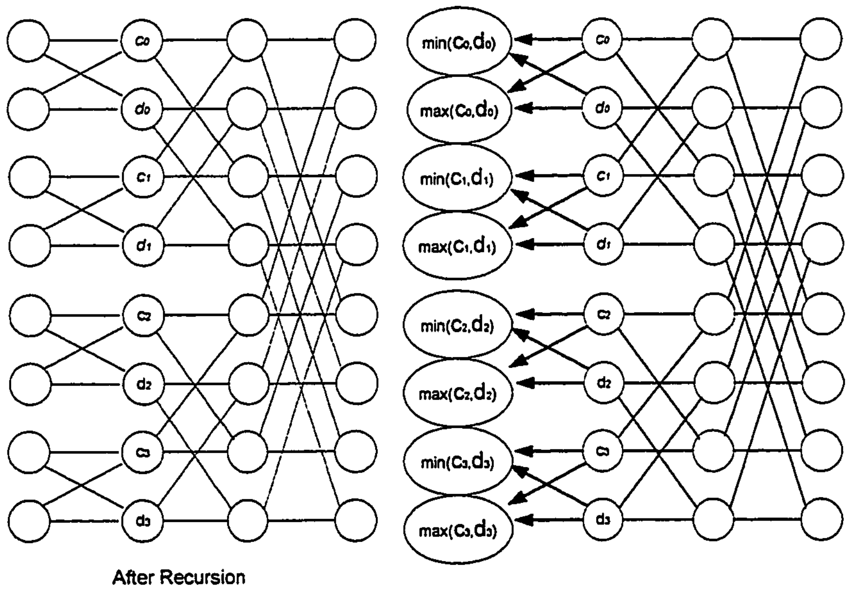


Figure 3.3-2 Implementation of the Odd-Even Merge algorithm on a butterfly

The algorithm just described uses only one level of butterfly edges at any step, and it uses consecutive steps, and thus the algorithm is normal. Thus it can be implemented to run in $O(\log^2 p)$ steps on any p-node hypercube network. For several decades, this algorithm was the fastest algorithm known for sorting on a hypercube network. Next we describe a faster algorithm.

A Deterministic O(log*p* loglog*p*)-Step Sorting on the Hypercube

This algorithm is based on an algorithm for merging $\sqrt{p}$ lists of $\sqrt{p}$ items in $O(\log p \log\log p)$ steps on a p-node hypercube. In particular, we start by partitioning the p items to be sorted into $\sqrt{p}$ groups with $\sqrt{p}$ items each. We then recursively sort the items within each group to form $\sqrt{p}$ sorted list of length $\sqrt{p}$, and finish by merging the $\sqrt{p}$ lists. If $t_s(p)$ denotes the time needed to sort p items on a p-node hypercube network, and $t_M(\sqrt{p},\sqrt{p})$ denotes the time needed to merge $\sqrt{p}$ sorted lists of length $\sqrt{p}$, then $t_S(p) \le t_S(\sqrt{p}) + t_M(\sqrt{p},\sqrt{p})$. We are assuming that p is a power of two (without loss of generality). We also assume that p is a perfect square, which is not the case always. So in the case when p is not a perfect square (i.e., when $\log p$ is an odd integer), we partition the p items into $\sqrt{2p}$ groups of size $\sqrt{\frac{p}{2}}$. We then recursively sort the items within each group in $t_S(\sqrt{\frac{p}{2}})$ steps to form $\sqrt{2p}$ sorted lists of length $\sqrt{\frac{p}{2}}$. We also partition the hypercube into two $(\frac{p}{2})$-node subcubes, each containing $\sqrt{\frac{p}{2}}$ of the lists. Each subcube can merge the $\sqrt{\frac{p}{2}}$ lists (of length $\sqrt{\frac{p}{2}}$) that it contains in $t_M(\sqrt{\frac{p}{2}},\sqrt{\frac{p}{2}})$ steps. After the merge, we are left with two sorted lists of length $\sqrt{\frac{p}{2}}$, which can be merged in $2\log p$ steps using the Odd-Even Merge algorithm. Hence, in the case when $\log p$ is an odd integer,

$$t_S(p) \le t_S(\sqrt{\tfrac{p}{2}}) + t_M(\sqrt{\tfrac{p}{2}},\sqrt{\tfrac{p}{2}}) + 2\log p$$

we will rely on the fact that merging x sorted lists of length x each can be done in $O(\log x \log\log 2x)$ steps on an x^2-node hypercube for any $x \ge 1$ that is a power of two, provided that we are allowed to perform some off-line pre-computation. Proof of this fact is given in appendix A. As a consequence, we will find that the time needed to sort p items is at most

$$t_S(p) \le t_S(2^{\left\lfloor \frac{\log p}{2} \right\rfloor}) + O(\log p \log\log p)$$

solving the recurrence we find that

$$\begin{aligned} t_S(p) &= O\left(\sum_{i=o}^{\log\log p} \log p^{2^{-i}} \log\log p^{2^{-i}}\right) \\ &= O\left(\sum_{i=0}^{\log\log p} 2^{-i} \log p \log\log p\right) \\ &= O(\log p \log\log p) \end{aligned}$$

If the pre-computation is not allowed, then the time needed to merge is $O(\log p \log\log^2 p)$, which results in an $O(\log p \log\log^2 p)$-step bound on the time needed to sort. The proof of the following theorem can be found in [27].

Theorem 3.3-2 [Sorting on the Hypercube]

Sorting p items on a p-processor hypercube can be performed in $O(\log p \log\log p)$ time if some offline pre-computation is allowed.

■

3.4 SELECTION ON INTERCONNECTION NETWORKS

3.4.1 The Universal Selection Algorithm

The algorithm presented by Rajasekaran, Chen, and Yooseph is an efficient deterministic algorithm for selection on any interconnection network when the number of input keys n is greater than the number of processors p.

The basic idea behind the algorithm is same as the sequential selection algorithm. The sequential algorithm partitions the input into groups (of say 5), finds the median of each group, and computes recursively the median (call it M) of these group medians. Then the rank r_M of M in the input is computed and as a result, all the elements from the input which are either $\leq M$ or $> M$ are dropped, depending on whether $i > M$ or $i \leq M$ respectively. Finally, an appropriate selection is performed from out of the remaining keys recursively. Call *M* the *splitter key* and the process of finding *M*, partitioning the remaining keys around *M*, and deleting unwanted keys a *phase* of the algorithm. In section 1.5.2 we saw that the running time of the algorithm is $O(n)$.

The same algorithm can be used in parallel, for instance on a PRAM, to obtain an optimal algorithm. If one must employ this algorithm on a network, it seems like one must perform periodic load balancing (i.e., distribute remaining keys uniformly among the processors). Load balancing is a costly operation to perform. The algorithm that we will describe now employs the same sequential selection algorithm with a twist. To begin with each node has exactly $\frac{n}{p}$ keys. As the algorithm proceeds, keys get dropped from future consideration. The algorithm never performs any load balancing. The remaining keys from each node will form the groups. The algorithm identifies the median of each group. If one attempts to use the median of these medians as the splitter key, not enough keys might get deleted in every phase and hence the algorithm might take a very long time to terminate. The objective is to eliminate constant fraction of the remaining keys in each phase of the algorithm. So instead of picking the median of medians as the splitter key *M*, the algorithm chooses a weighted median of medians. Each group median is weighted with the number of remaining keys in that node. This modification suffices to ensure that a constant fraction of keys get eliminated in each phase.

Algorithm WSELECT(*S*, *k*)

```
1.  $N \leftarrow n \leftarrow |S|$
2.  if $\log(\frac{n}{p})$ is $\le \log\log p$ then
3.          Sort the elements at each node
4.  else
5.          Partition the keys at each node into nearly $\log p$ approximately equal
            parts such that keys in one part will be $\le$ keys in the parts to the right
    end if
6.  repeat
7.          (2.1) for $i \leftarrow 1$ to $p$ do in parallel
8.                  (2.1.1) Find the median $M_i$ and their weights $N_i$
9.          end for
10.         (2.2) Find the weighted median $M$ of $M_1, M_2, \ldots, M_p$ medians
11.         (2.3) Count the rank of $M$. Let this rank be $r_m$
12.         (2.4) if $k \le r_m$ then
13.                 (2.4.1) Eliminate all remaining keys that are $> M$
14.         else
15.                 (2.4.2) Eliminate all remaining keys that are $\le M$
16.         end if
17.         (2.5) Compute $E$, the number of eliminated keys
18.         (2.6) if $k > r_m$ then
19.                 (2.6.1) $k \leftarrow k - E$
20.                 (2.6.2) $N \leftarrow N - E$
21.         end if
22. until $N \le c$, $c$ being a constant
```

Algorithm 3.4-1 Universal Selection Algorithm

The analysis of algorithm 3.4.1 follows; lines 1-5 take $(\frac{n}{p})\min\{\log(\frac{n}{p}), \log\log p\}$. At the end of line 5, the keys in any node have been partitioned into nearly $\log p$ approximately equal parts. Call each such part a *block*.

Line 7, we could find the median at any node in $O(\log p + (\frac{n}{p \log p}))$ time. This can be done by first identifying the group that has the median and then performing an appropriate selection in this group. In line 10, we could sort the median and thereby compute the weighted median. If $M'_1, M'_2, ..., M'_p$ is the sorted order of the medians, then we need to identify j such that $\sum_{k=1}^{j} N'_k \geq \frac{N}{2}$ and $\sum_{k=1}^{j-1} N'_k < \frac{N}{2}$. Such a j can be computed with an additional prefix computation. Thus M, the weighted median, can be identified in time $O(T_p^{sort} + T_p^{prefix})$, where T_p^{sort} is the time needed to perform a prefix computation on a p-node network. Line 11 requires a scan through the locally remaining keys followed by a prefix sums computation. Thus this step takes $O(\frac{n}{p \log p} + T_p^{prefix})$ time. Lines 12-16 also take $O(\frac{n}{p \log p} + T_p^{prefix})$ time. Lines 18-21 take $O(T_p^{prefix})$ since they involve just a prefix computation. Thus each run of the *repeat* loop takes $O(\frac{n}{p \log p} + T_p^{prefix} + T_p^{sort})$.

To determine the number of keys getting eliminated in each run assume that $i > r_M$ in a given run. (The other case is argued similarly.) The number of keys eliminated is at least $\sum_{k=1}^{j} \left\lceil \frac{N'_k}{2} \right\rceil$ which is $\geq \frac{N}{4}$. Therefore, it follows that the *repeat* loop is executed $O(\log n)$ times. Thus we get (assuming that $\log n$ is asymptotically the same as $\log p$).

Theorem 3.4-1 [Universal Selection Algorithm]

Selection on any p-node interconnection network can be performed in time $O(\frac{n}{p}\log\log p + [T_p^{prefix} + T_p^{sort}]\log n)$. If $T_p^{prefix} \le T_p^{sort}$ then this time bound is $O(\frac{n}{p}\log\log p + T_p^{sort}\log n)$.

■

3.4.2 Selection on the Linear Array

For the linear array, from lemma 3.2-1 we know that prefix computation can be performed in $O(p)$ time. And from lemma 3.3-1 and 3.3-2 we know that sorting p keys on a p-node linear array also requires $O(p)$ time. Thus $T_p^{prefix} = T_p^{sort} = O(p)$ for the linear array, and we get

Theorem 3.4-2 [Selection on the Linear Array]

Selection on a p-node linear array can be performed in time $O(\frac{n}{p}\log\log p + p\log n)$.

■

3.4.3 Selection on the Mesh

For the mesh interconnection network, from theorem 3.2-3 clearly shows that prefix computation can be performed in $O(\sqrt{p})$. And according to theorem 3.3-1, sorting p

keys on a p-node mesh also requires $O(\sqrt{p})$. Therefore, for the mesh, $T_p^{prefix} = T_p^{sort} = O(\sqrt{p})$. Thus we get the following theorem.

Theorem 3.4-3 [Selection on the Mesh]

Selection on a p-node square mesh can be performed in time $O(\frac{n}{p}\log\log p + \sqrt{p}\log n)$.

■

3.4.4 Selection on the Hypercube

As presented in section 3.3.3.2, the fastest known value for T_p^{sort} on a hypercube is $O(\log p \log\log p)$. Thus we have the following theorem (assuming that $n = p^c$ for some positive constant c then $\log n$ is asymptotically the same as $\log p$).

Theorem 3.4-4 [Selection on the Hypercube]

Selection on a p-node hypercube can be performed in time

$$O(\tfrac{n}{p}\log\log p + \log^2 p \log\log p) = O((\tfrac{n}{p} + \log^2 p)\log\log p).$$

■

3.5 MULTISELECTION ON INTERCONNECTION NETWORKS

Recall that in the multiselection problem, a set $S = \{x_1, x_2, \ldots, x_n\}$ of elements and a set $K = \{k_1, k_2, \ldots, k_r\}$ of ranks are given, where $r \le n$. The objective is to find the elements in *S* of ranks k_i, $1 \le i \le r$. In other words, find the k_i th smallest elements of *S*.

In section 2.5.1, an adaptive multiselection algorithm is given, which uses the divide-and-conquer approach, which splits the problem into two subproblems by partitioning both *S* and *K* into two sub sets, then recursively solving the subproblems.

In this section we will combine algorithm 2.5-1 and algorithm 3.4-1 to devise a universal multiselection algorithm.

3.5.1 Universal Multiselection Algorithm

The idea behind our algorithm is same as of algorithm 3.4-1. We make use of the weighted median to avoid load balancing and eliminate almost equal amount of keys from each processor. We assume that *K* is sorted, if not, *K* can be sorted in $O(r \log r)$ time using *quicksort()*. Initially, each processor P_i has a subset S_i of *S*, where $|S_i| = \frac{n}{p}$ and a copy of *K*.

Algorithm: **UMSELECT(*S*, *K*)**

```
1.  if K is not empty then
2.      if $K = \{k\}$ then
3.          return WSELECT(S, k)
4.      else
5.          $r = |K|$
6.          $w = k_{r/2}$
7.          WSELECT(S, w)
8.          for $i = 1$ to p do in parallel
9.              $S_i^1 = \{x \in S_i \mid x < S[w]\}$
10.             $S_i^2 = \{x \in S_i \mid x > S[w]\}$
11.             $K^1{}_i = \{k_1, k_2, ..., k_{w-1}\}$
12.             $K^2{}_i = \{k_{r/2+1} - w, k_{r/2+2} - w, ..., k_r - w\}$
13.         end for
14.         UMSELECT($S_i^1, K_i^1$)
15.         UMSELECT ($S^2{}_i, K^2{}_i$)
16.     end if
17. end if
```

Algorithm 3.5-1 Universal Multiselection UMSELECT(*S*, *K*)

The analysis of Algorithm 3.5-1 is similar to the MSELECT algorithm given in algorithm 2.5-1. In line 7 *WSELECT* takes $O(\frac{n}{p}\log\log p + [T_p^{prefix} + T_p^{sort}]\log n)$. Lines 9 and 10 both can be performed using prefix computation in T_p^{prefix} time. In lines 14 and 15the depth of the recursion is $\lceil \log r \rceil$ thus the running time of the algorithm is

$$O((\tfrac{n}{p}\log\log p + [T_p^{prefix} + T_p^{sort}]\log p)\log r).$$

Theorem 3.5-1 [Universal Multiselection]

Multiselection on a p-node interconnection network can be performed in

$$O((\tfrac{n}{p}\log\log p + [T_p^{prefix} + T_p^{sort}]\log p)\log r).$$

where T_p^{prefix} is the time required to do prefix computation and T_p^{sort} is the time required to sort p keys on the p-node interconnection network.

■

3.5.2 Multiselection on the Linear Array

From theorem 3.5-1 it is clear that if we determine T_p^{pr} and T_p^{sort} for any network, then we can determine the running time of multiselection algorithm for that network. From lemma 3.2-1 we know that $T_p^{prefix} = O(p)$. $T_p^{sort} = O(p)$, (c.f. lemma 3.3-1 and 3.3-2). Thus multiselection on the linear array can be performed in

$$O((\tfrac{n}{p}\log\log p + p\log n)\log r)$$

It appears that the multiselection is not suitable for the interconnection array when n is comparable in size to the number of processors p, e.g. $p = \frac{n}{\log n}$. For this reason, when n

is large, say $p = n^{\varepsilon}$, the running time of the algorithm reduces to $O(n^{\varepsilon} \log n \log r)$, which is more efficient than sorting. But for extreme cases when $r \to n$, the algorithm reduces to $O(n^{\varepsilon} \log^2 n)$. In this case we are using the multiselection as a sorting algorithm, that is, we are selecting 1st smallest, 2nd smallest... nth smallest elements. For such extreme cases, the Merge-Split Sorting algorithm (algorithm 3.3-2) is more suitable.

3.5.3 Multiselection on the Mesh

From the previous discussions we get that T_p^{prefix} and T_p^{sort} are both equal to $O(\sqrt{p})$. Therefore, from the universal multiselection theorem, multiselection on a p-processor mesh interconnection network can be preformed in $O((\frac{n}{p} \log\log p + \sqrt{p} \log n) \log r)$ time.

It is apparent that the multiselection algorithm is not suitable for cases where number of elements n is far less than the number of processors p. For such a cases, one can use a sub-mesh of the original mesh to solve the problem. In addition, if n, the number of elements, is equal to the number of processors p, that is, if $n = p$ then the running time of the algorithm will reduce to

$$O((\log\log n + \sqrt{n} \log n) \log r) \text{ or } O((\log\log p + \sqrt{p} \log p) \log r)$$

which is not suitable because multiselection can be done in an indirect way –in this case, by simply sorting the elements using Odd-Even merge sort on the mesh in $O(\sqrt{p})$ time and letting the processors P_j where $j \in K$ publish their elements.

Now let us consider the case where we have n elements and we want to do multiselection on a p processor mesh such that $n >> p$. For sorting the elements on the mesh and publishing the kth elements it will require $O(\frac{n}{p}\log\frac{n}{p}+\frac{n}{p}\sqrt{p})$ steps. Simply because initially each processor will start by sorting $\frac{n}{p}$ elements in $O(\frac{n}{p}\log\frac{n}{p})$ time then, using Odd-Even merge on the mesh will require $O(\frac{n}{p}\sqrt{p}) = O(\frac{n}{\sqrt{p}})$ time, of total sorting time being

$$O(\tfrac{n}{p}\log\tfrac{n}{p}+\tfrac{n}{\sqrt{p}}) \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \qquad (2)$$

In the multiselection algorithm, if $r = n$, the problem turns in to the sorting problem, which also resembles the extreme (worst) case for the multiselection problem. Now we will compare the extreme case of multiselection on the mesh with the upper-limit average case of sorting on the mesh to establish our result.

Take $p = n^{\varepsilon}$ where $0 < \varepsilon < 1$. By substituting in (1) and (2) we get

Multiselection: $O((n^{1-\varepsilon}\log\log n^{\varepsilon} + n^{\varepsilon/2}\log n)\log n)$

Let $f(n,\varepsilon) = (n^{1-\varepsilon}\log\log n^{\varepsilon} + n^{\varepsilon/2}\log n)\log n \ldots\ldots$ (3)

Sorting: $O(n^{1-\varepsilon}\log n^{1-\varepsilon} + n^{1-\varepsilon/2})$

Let $g(n,\varepsilon) = (1-\varepsilon)n^{1-\varepsilon}\log n + n^{1-\varepsilon/2} \ldots\ldots\ldots\ldots$ (4)

f and g are both are function in two variables n and ε where $n \in (c,\infty)$ and $\varepsilon \in (0,1)$ where c is a positive constant. Figure 3.5-1 shows the actual surface of f. The z-axis represents the number of steps $t(n)$ against the number of processors p on the x-axis and the number of elements n on the y-axis. The number of processors is a function of ε and n. Figure 3.5-2 shows the actual surface of g, while Figure 3.5-3 shows the same surface of g with logarithmic z-axis.

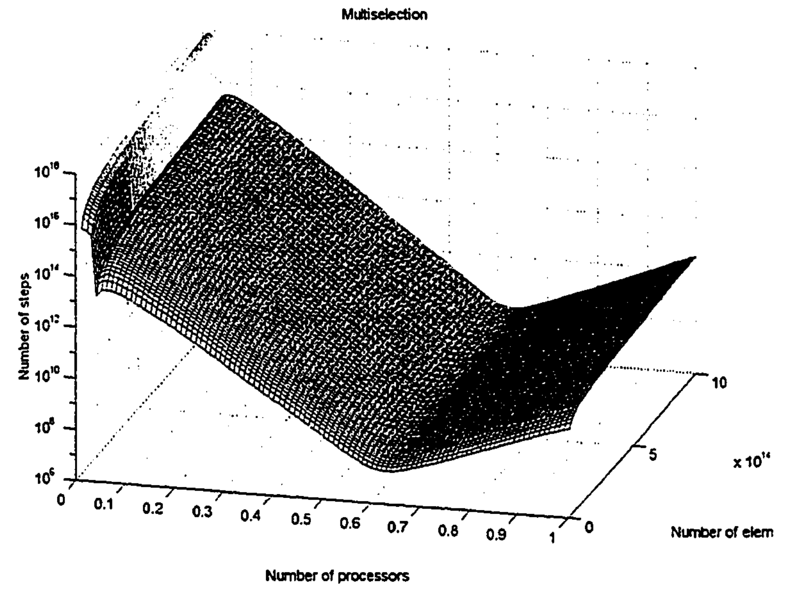


Figure 3.5-1 Multiselection surface

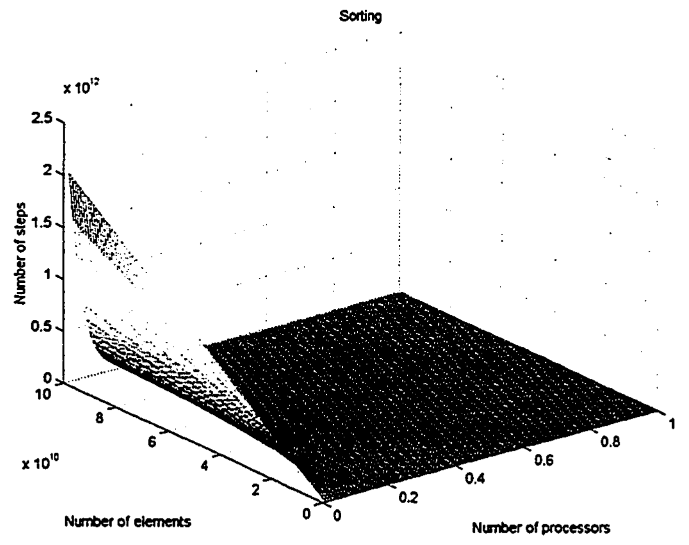


Figure 3.5-2 Sorting surface with linear z-axis

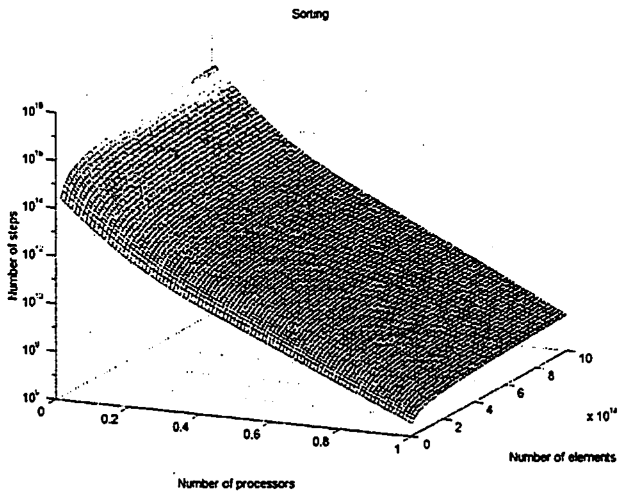


Figure 3.5-3 Sorting surface

Taking c a large positive integer, it is clear from the figures that f is less than g in most of the region $R = \{(\varepsilon, n) \mid 0 < \varepsilon < 1 \text{ and } c < n < \infty\}$. This is clearer from the difference surface d in Figure 3.5-4. The multiselection algorithm performs better than the sorting algorithm in the area where the surface d is non-zero. By closely examining Figure 3.5-4 we see that the multiselection algorithm performs better than the sorting algorithm in two regions.

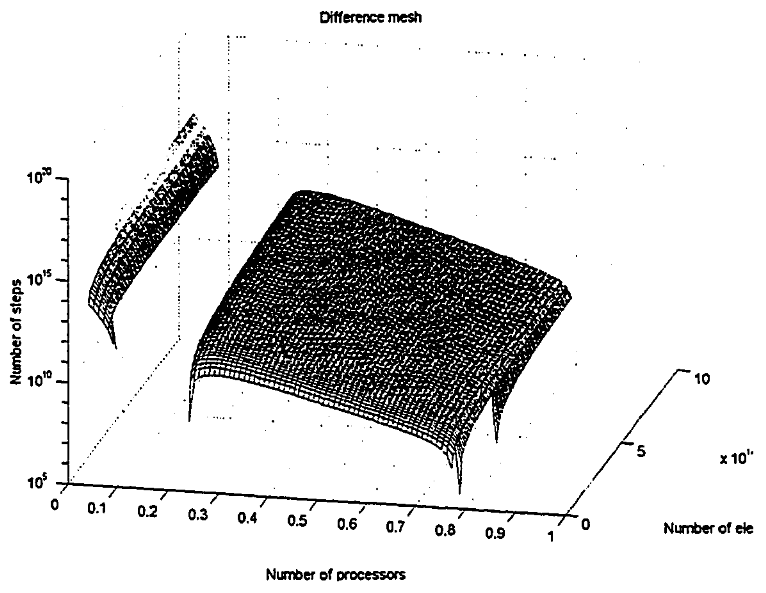
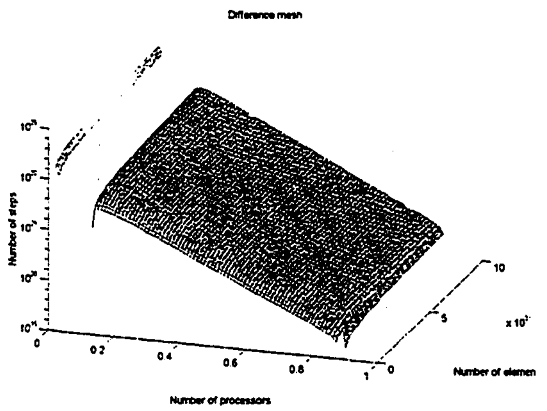
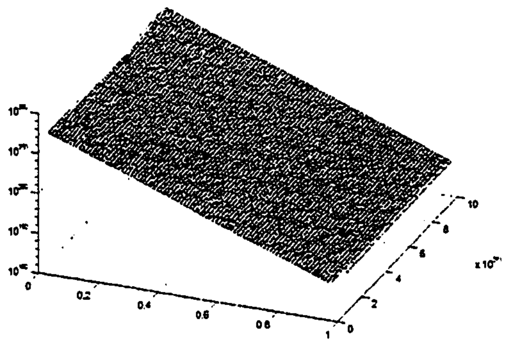


Figure 3.5-4 Difference surface. $[1 \le n \le 10^{14}]$





Difference surface $[10^{20} \le n \le 10^{30}]$ Difference surface $[10^{250} \le n \le 10^{280}]$

Figure 3.5–5 Difference surfaces

A small region for small values of ε, and a large region as ε grows larger. This region grows larger when the numbers of elements grow significantly. The multiselection algorithm proves to show better performance than the sorting algorithm for very large number of elements n.

Furthermore, if we study a cross section of both f and g at different values of n, it appears that our multiselection algorithm shows a very good performance across a large area of ε's interval.

In fact, the multiselection algorithm shows a very good performance in the middle of ε's interval for very values of n. This can be established by analyzing the behavior of the multiselection surface cross section with the behavior of the sorting surface cross section in Figures 3.5-6 and 3.5-7. Figure 3.5-6 is a plot of a cross section of f and g when the number of elements is small, and Figure 3.5-7 shows two plots of the cross sections of *f* and *g*.

When the number of processors is small, i.e. ε is small compared to the number of elements, the multiselection and sorting takes almost the same time for sorting *n* number of elements for any *n*. The sudden drop in the graph of *f* is due to the change of definition of *f* from $(\frac{n}{p}\log\frac{n}{p}+\sqrt{p}\log n)\log n$ to $(\frac{n}{p}\log\log p+\sqrt{p}\log n)\log n$.

This later definition of *f* shows a drop in the number of steps for sorting using multiselection then rises up again as ε approaches to 1. The point of intersection of *f* with *g* at large values of ε is logical because as the number of processors grows larger

and larger, fewer elements will be assigned to individual processors and as $\varepsilon \to 1$, $p \to n$. So for the multiselection we get

$$\lim_{p\to n}(\tfrac{n}{p}\log\log p+\sqrt{p}\log n)\log n = (\log\log n+\sqrt{p}\log n)\log n$$

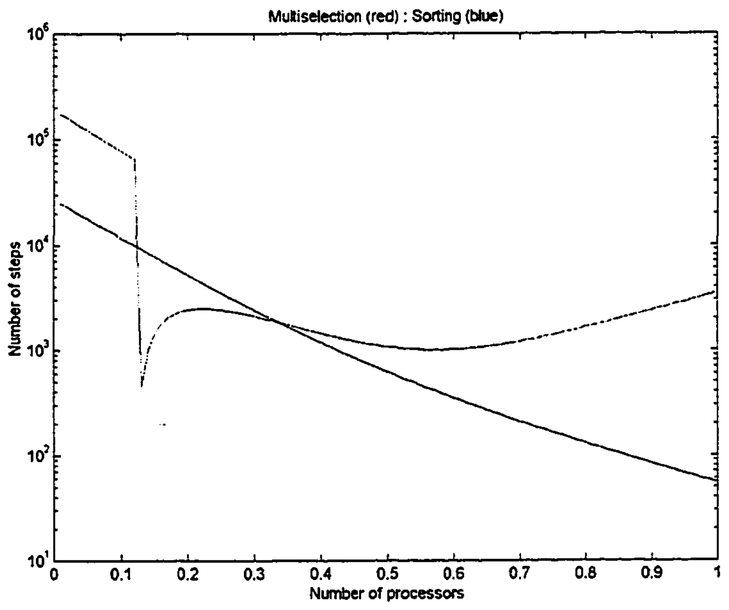
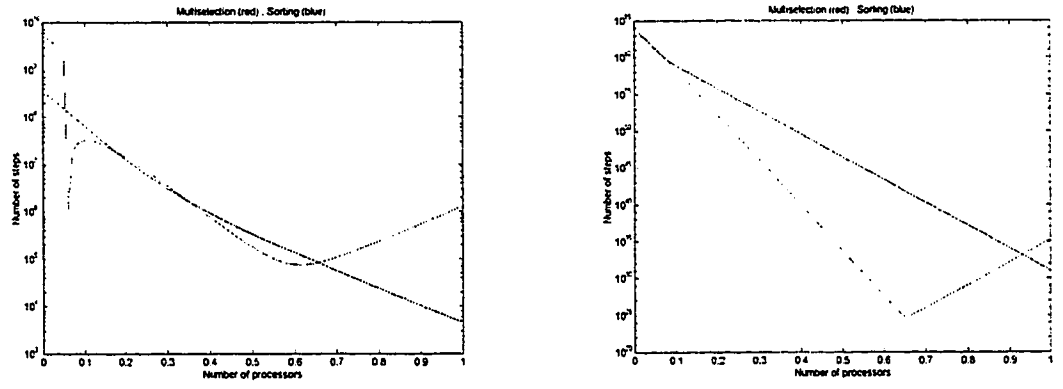


Figure 3.5-6 A cross section of the Multiselection mesh and the Sorting mesh when the number of elements is very small



Small *n*.

Large *n*

Figure 3.5-7 A cross section of the Multiselection mesh and the Sorting mesh when the number of elements is large

Whereas for sorting we get

$$\lim_{p \to n}(\tfrac{n}{p}\log\tfrac{n}{p}+\tfrac{n}{\sqrt{p}}) = c_1 + \sqrt{n}$$

It is clear that $\sqrt{n}$ is far less than $(\log\log n + \sqrt{p}\log n)\log n$. Which means, that if the number of processors is equal to the number of elements, it is better to sort all the elements and pick the k_ith element by letting processor P_{k_i} publish element it has.

Summarizing the result, it appears from the discussion given above that multiselection is not suitable when the number of processors p is comparable to the number of elements n. On the other hand, when n is large, say $p = n^{\varepsilon}$, the running time of the algorithm reduces to $O(n^{\frac{\varepsilon}{2}}\log n \log r)$, which is more efficient than sorting. For optimized performance consider the running time of the multiselection algorithm in terms of p and n. That is,

$$O((\tfrac{n}{p}\log\log p + \sqrt{p}\log n)\log r)$$

By taking $p = n^{\frac{2}{3}}$ the algorithm's running time reduces to $O(n^{\frac{1}{3}}\log n \log r)$.

3.5.4 Multiselection on the Hypercube

From theorem 3.3-2 we get that sorting p elements on a p-node hypercube can be performed in $O(\log p \log\log p)$ time. Thus, $T_p^{sort} = O(\log p \log\log p)$. From theorem 3.2-4 prefix computation can be performed on the same setup in $O(\log p)$. Thus

$$T_p^{sort} + T_p^{prefix} = O(\log p \log\log p) + O(\log p) = O(\log p \log\log p)$$

Therefore, the running time for the multiselection algorithm on the hypercube will reduce to

$$O((\tfrac{n}{p}\log\log p + \log^2 p \log\log p)\log r) = O((\tfrac{n}{p} + \log^2 p)\log\log p \log r)$$

In the following, we will compare the worst case of multiselection, when $r = n$, with the sorting algorithm on the hypercube.

From [22], we know that the selection on the hypercube is nearly matches that of [8]. But if we use the same comparison technique as we did with the analysis of the multiselection on the mesh, that is, we take the worse case of the multiselection when $r = n$ and use the multiselection algorithm for sorting. The running time of the algorithm turns to

$$f : O((\tfrac{n}{p} + \log^2 p)\log\log p \log n).$$

A plot of the performance surface of the multiselection algorithm can be seen in Figure 3.2-1.

Sorting n keys on a p-processor hypercube when $n > p$ would take on the average

$$g : O(\tfrac{n}{p}\log\tfrac{n}{p} + \tfrac{n}{p}\log p \log\log p).$$

A plot of the performance surface of the sorting algorithm can be seen in Figure 3.2-2. The difference surface $(g - f)$ is shown in Figure 3.2-3. Form the difference surface it is clear that the multiselection algorithm shows the same performance as the sorting algorithm in the worse case. That is, when the multiselection algorithm is used for sorting. Looking at the general case we find that when $n >> p$ and $r < n$, for multiselection:

$$O((\tfrac{n}{p} + \log^2 p)\log\log p \log r) \rightarrow O(n \log r)$$

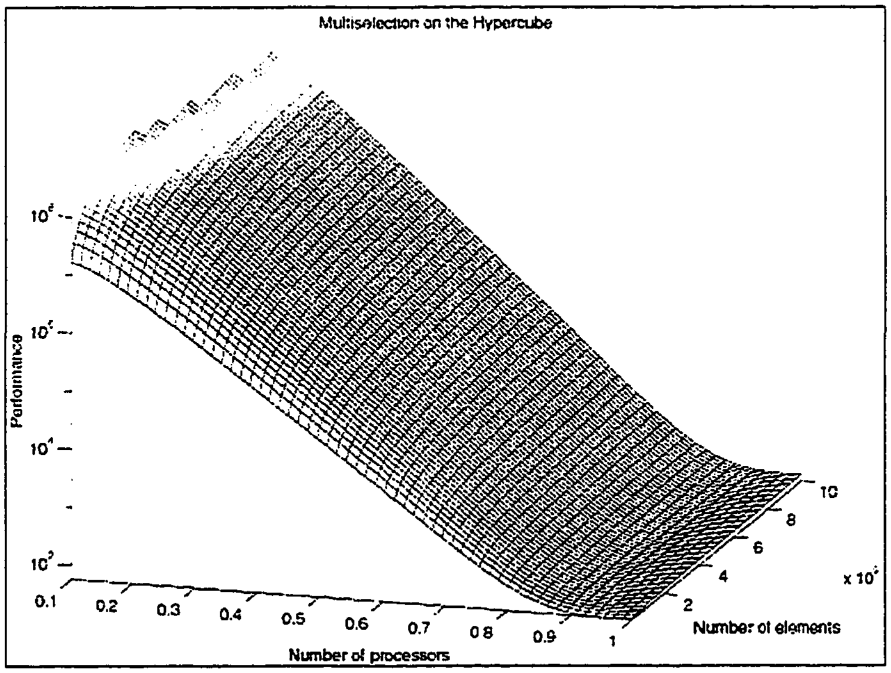


Figure 3.5-8 Multiselection performance surface on the Hypercube

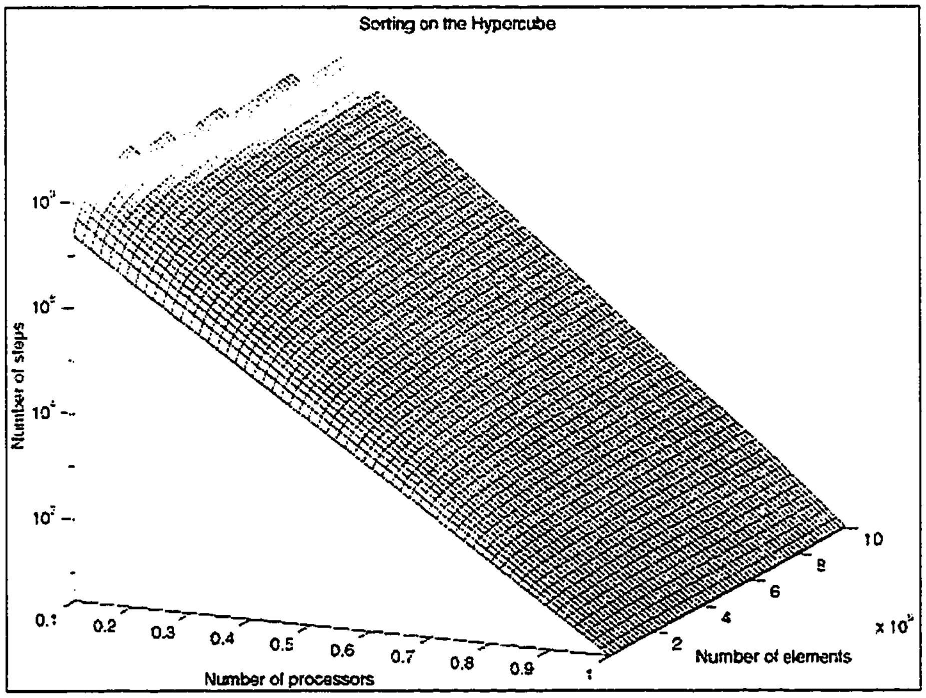


Figure 3.5-9 Sorting performance surface on the Hypercube

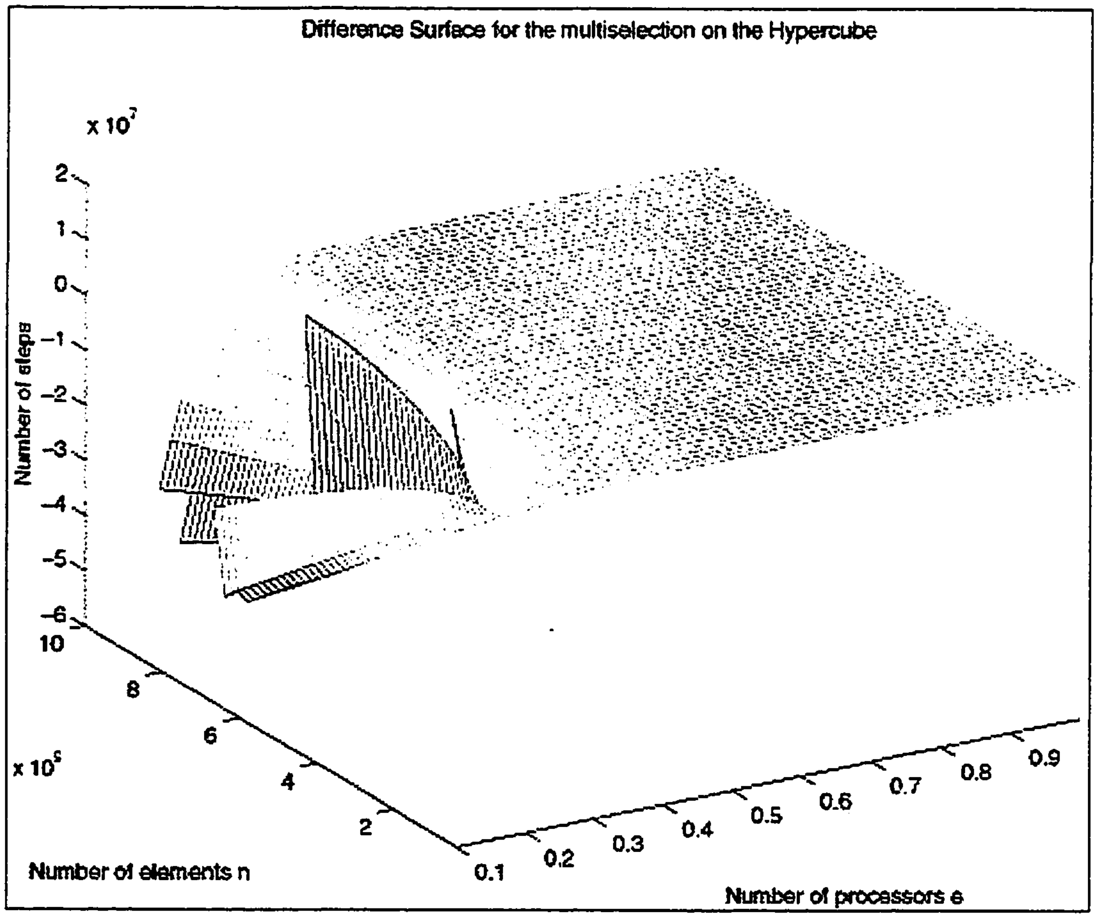


Figure 3.5-10 Difference surface between

while for sorting:

$$O(\tfrac{n}{p}\log\tfrac{n}{p}+\tfrac{n}{p}\log p\log\log p)\rightarrow O(n\log n)$$

Thus the multiselection algorithm gives better performance on the hypercube in the average case. Therefore we conclude that the multiselection algorithm we produced is good when the number of elements is far larger than the number of processors in a hypercube.

3.6 SUMMARY

A network can be viewed as a graph $G = (N, E)$ where each node $i \in N$ represents a processor, and each edge $(i, j) \in E$ represents a two-way communication link. In an interconnection network, memory is distributed among the p processors. Communication is done through links between processors. When a processor P_i in an interconnection network wishes to send a datum to processor P_j, it uses the network to route the datum from its memory to that of P_j.

Two processors, in an interconnection network, directly connected by a link are said to be *neighbors*. The *degree* of a processor in a given network topology is defined as the number of neighbors of that processor. The *distance* between two processors P_i and P_j in a given network topology is the number of links on the shortest path from P_i to P_j. The *diameter* of the network is the length of the longest distance among all distances between pairs of processors in that network.

In a linear array, each interior processor in a liner array is connected with bi-directional links to its *left-neighbor* and its *right-neighbor*. The outermost processors may have just one connection each, and may serve as input/output points for the entire network.

A mesh is an $a \times b$ grid in which there is a processor at each grid point. The edges correspond to the communication links and are bi-directional. Whereas a *d-dimensional hypercube* has $p = 2^d$ nodes (processors) and $d2^{d-1}$ edges (communication links, or simply, links).

The d-dimensional butterfly has $(d+1)2^d$ nodes and $d2^{d+1}$ edges. The nodes correspond to pairs $\langle w,i \rangle$ where i is the *level* or *dimension* of the node ($0 \le i \le d$) and w is a d-bit binary number that denotes the *row* of the node. Two nodes $\langle w,i \rangle$ and $\langle w',i' \rangle$ are linked by an edge if and only if $i' = i+1$ and either:

- w and w' are identical, or
- w and w' differ in precisely the i' th bit.

If w and w' are identical, the edge is said to be a *straight edge*. Otherwise, the edge is a *cross edge*.

Broadcasting on a p-processor linear array can be completed in $O(p)p$ steps. In a $\sqrt{p} \times \sqrt{p}$ mesh the same can be performed in $\le 2(\sqrt{p}-1) = O(\sqrt{p})$ time. While broadcasting on a p-processor d-dimensional hypercube can be performed in $O(\log p)$ steps, where $d = \log p$.

Prefix computation on a p-processor linear array can be performed in $O(p)$ time. On the p-processor mesh it requires $O(\sqrt{p})$ time. While on the p-processor hypercube, it requires $O(\log p)$ time.

Data concentration (packing) on a p-processor array takes $O(p)$ time. On a $\sqrt{p} \times \sqrt{p}$ mesh, it takes $O(\sqrt{p})$ time. On the hypercube, it can be performed in $O(\log p)$ time.

Two sorted sequences of length m each can be merged on a p-processor linear array in $O(p)$ time, where $p = 2m$. Two sorted snakes of size $\sqrt{p} \times \frac{\sqrt{p}}{2}$ each can be merged in time $O(\sqrt{p})$ on a $\sqrt{p} \times \sqrt{p}$ mesh.

The merge-split sort runs in $O(p)$ time on a p-processor linear array. Sorting p elements can be done in $O(\sqrt{p})$ time on a $\sqrt{p} \times \sqrt{p}$ into snakelike row major order. Merge sorting can be implemented to run in $O(\log^2 p)$ steps on any p-node hypercube network. Sorting p items on a p-processor hypercube can be performed in $O(\log p \log\log p)$ time if some offline pre-computation is allowed.

Selection on any p-node interconnection network can be performed in time $O(\frac{n}{p}\log\log p + [T_p^{prefix} + T_p^{sort}]\log n)$. If $T_p^{prefix} \le T_p^{sort}$ then this time bound is

$O(\frac{n}{p}\log\log p + T_p^{sort}\log n)$. Selection on a p-node linear array can be performed in time $O(\frac{n}{p}\log\log p + p\log n)$. Selection on a p-node square mesh can be performed in $O(\frac{n}{p}\log\log p + \sqrt{p}\log n)$ time. Selection on a p-node hypercube can be performed in time $O(\frac{n}{p}\log\log p + \log^2 p\log\log p) = O((\frac{n}{p} + \log^2 p)\log\log p)$.

Multiselection on a p-node interconnection network can be performed in

$$O((\frac{n}{p}\log\log p + [T_p^{prefix} + T_p^{sort}]\log p)\log r).$$

where T_p^{prefix} is the time required to do prefix computation and T_p^{sort} is the time required to sort p keys on the p-node interconnection network. Therefore, on the linear array, it can be performed in $O((\frac{n}{p}\log\log p + p\log n)\log r)$ time. While on a p-processor mesh it requires $O((\frac{n}{p}\log\log p + \sqrt{p}\log n)\log r)$ time, and on a p-processor hypercube the running time for the multiselection algorithm is

$$O((\frac{n}{p}\log\log p + \log^2 p\log\log p)\log r) = O((\frac{n}{p} + \log^2 p)\log\log p\log r)$$

For the cases when the number of elements is far larger than the number of processors in the hypercube, that is $n >> p$, our algorithm is more suitable for multiselection.

References:

1. Miklos Ajtai, Janos Komlos, W.L. Steiger, Endre Szemerdi, Deterministic Selection in $O(\log\log n)$ Parallel Time, *ACM*, 188-195, 1986.
2. Valiant, L., Parallel in Comparison Problems, *SIAM J. Computing*, 4, 348-355, 1975.
3. Gereb-Graus, Mihaly and Krizanc, Danny. A Lower Bound of $\Omega(\log\log n)$ for Randomized Parallel Merging (draft). Harvard University, 1985.
4. Cole, Richard and Yap, Chee K. A Parallel Median algorithm. *Inf. Proc. Letters 20*, 137-139, 1985.
5. Cole, Richard J., An Optimally Efficient Selection Algorithm, *Information Processing Letters 26*, pp.295-299, 1987/1988
6. D. Krizanc, L. Narayanan, and R. Raman, Deterministic Selection on Mesh-Connected Processor Arrays, *Algorithmica* 15: 319-332, 1996
7. M. Blum, R. Floyd, V. R. Prat, R. Rivest, and R. Tarajan, Time Bounds for Selection. *Journal of Computer and System Science,* 7(4):488-461, 1972
8. G. Plaxton. On the Network Complexity of Selection. *Proceedings of the Symposium on the Foundations of Computer Science*, pp. 396-401, 1989.
9. G. Plaxton. Load Balancing, Selection and Sorting on the Hypercube. *Proceedings of the Symposium on Parallel Algorithms and Architecture*, pp. 64-73, 1990.
10. S. Rajasekaran. Randomized Parallel Selection. *In Foundations of Software Technology and Theoretical Computer Science*, pp. 176-184. Lecture Notes in Computer Science, Vol 472. Springer-Verlag, Berlin. 1990.
11. M. Kaufmann, J. Sibeyn, and T. Suel. Derandomizing Algorithms for Routing and Sorting on Meshes. *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 669-679, 1994.

12. M. Kunde. l-Selection and Related Problems on Grids of Processors. *Journal of New Generation Computer Systems*, 2:129-143, 1989.

13. C. C. Aggarwal, N. Jain, P. Gupta, An Efficient Selection Algorithm on the Pyramid, *Information Processing Letters* 53:37-47, 1995.

14. S. G. Akl, *Design and Analysis of Parallel Algorithms*, Printice Hall, Englewook Cliffs, NJ, 1989.

15. C. D. Thompson and H. T. Kung, Sorting on a Mesh Connected Parallel Computer, *Comm. ACM*, 20, 1997

16. R. Cypher and G. Plaxton, Deterministic Sorting in Nearly Logarithmic Time on the Hypercube and Related Computers, *Proc. STOC*, 1990.

17. A. Aggarwal, A Comparative Study of X-tree, Pyramid and Related Machines, *Proc. 25th Ann. IEEE Symp. On Foundations of Computer Science*, 1984.

18. H. Shen, Optimal Parallel Multiselection on EREW PRAM, *Parallel Computing*, 23 (1997), 1987-1992.

19. M. L. Fredman and T. H, Spencer, Refined Complexity Analysis for Heap Operations, *Journal of Computer and System Sciences*, 269-284, 1987.

20. M. H. Alsuwaiyel, An Optimal Parallel Algorithm for the Multiselection Problem, *Parallel Computing*, to appear.

21. M. H. Alsuwaiyel, On the Multiselection Problem, *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications*, 2000, 1439-1441.

22. S. Rajasekaran, W Chen, and S. Yooseph, Unifying Themes for Selection on Any Network, *Journal of Parallel and Distributed Computing*, 46, 1997, 105-111.

23. S. G. Akl, *Parallel Computation Models and Methods*, Prentice-Hall, Inc. 1997

24. J. Jaja, *An Introduction to Parallel Algorithms*, Addison-Wesely Publishing Company, 1992.

25. Cormen T., Leiserson C. and Rivest R., *Introduction to Algorithms*, MIT Press McGraw Hill, 1994.

26. F. Thomson Leighton, *Introdction to Parallel Algorithms and Architectures: Arrays – Trees – Hypercubes*, Morgan Kaufmann Publishers Inc., 1992

Vita

Name:	Adel Fadhl Noor Ahmed.
Date of Birth:	April 6, 1973.
Place of Birth:	Riyadh, Saudi Arabia.
Nationality:	Saudi.

- Received Bachelor of Science (B.Sc.) degree in Mathematics from King Saud University (KSU), Riyadh Saudi Arabia in 1995.

- Joined the Department of Information and Computer Science at King Fahd University of Petroleum and Minerals (KFUPM), Dhahran, Saudi Arabia as a Graduate Assistant in 1996.

- Received Master of Science (M.Sc.) degree in Computer Science from King Fahd University of Petroleum and Minerals (KFUPM), Dhahran, Saudi Arabia in 2001.