INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in "sectioning" the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again – beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from "photographs" if essential to the understanding of the dissertation. Silver prints of "photographs" may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

Xerox University Microfilms

300 North Zeeb Road
Ann Arbor, Michigan 48106

77-288

HSU, Terry Tsai-Yuan, 1939-
ON PARALLELISM, SCHEDULING, AND DATA
COMMUNICATION IN PARALLEL PROCESSING
SYSTEMS.

City University of New York, Ph.D., 1976
Engineering, electronics and electrical

Xerox University Microfilms, Ann Arbor, Michigan 48106

ON

PARALLELISM, SCHEDULING, AND DATA COMMUNICATION IN PARALLEL PROCESSING SYSTEMS

by

TERRY T. HSU

A dissertation submitted to the Graduate Faculty in Engineering in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

1976

This manuscript has been read and accepted for the Graduate Faculty in Engineering in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

Aug. 23, 1976
date

Se Jeung Oh
Chairman of Examining Committee

Aug 23, 1976
date

Jacques E. Benveniste
Executive Officer

Prof. Se Jeung Oh

Prof. Morris Ettenberg

Prof. Ralph Mekel

Prof. Gerald E. Subak-Sharpe

Supervisory Committee

The City University of New York

Abstract

ON PARALLELISM, SCHEDULING, AND DATA COMMUNICATION
IN PARALLEL PROCESSING SYSTEMS

by

Terry T. Hsu

Adviser: Professor Se Jeung Oh

Three major aspects concerning the design and performance of a parallel processing system are investigated. These are detection and exploitation of parallelism in basic computations, task scheduling, and rearrangeable data connection networks. The objective is to make a unified study on these sub-areas of parallel processing and explore more efficient and practical ways to achieve a faster computation.

A general parsing technique to find parallelism within an arithmetic expression is discussed. In particular, a special binary number system is used to represent an expression's tree-height, and operations are introduced to compute the minimum tree height from its subexpressions. This approach, coupled with the concept of free nodes, is used to apply the distributive law to further reduce an expression's tree height. Inter-statement parallelism is also discussed.

Some important bounds and algorithms concerning deterministic and non-preemptive scheduling are investigated. A new expression for the lower bound on processing time and

some modified optimal and heuristic scheduling algorithms are presented. An analysis in cost and performance tradeoff is made to relate the theoretic parallelism exploitation to a real processing system.

Results of parallelism exploitation and scheduling are useful only if they can be realized in real-time processing by some rearrangeable data connection networks. A number of commonly encountered data connection requirements and rearrangeable connection networks are reviewed. Most importantly, a definitive decomposition algorithm for controlling a multistage rearrangeable switching network is presented. The new algorithm eliminates the problem-dependent nature of other previously known heuristic approaches, and has a high degree of parallelism imbedded for fast processing.

ACKNOWLEDGEMENTS

I wish to express my deepest gratitude to my adviser, Professor Se Jeung Oh of the Department of Electrical Engineering of the City College of New York, for his continuing guidance, inspiring suggestions, and enthusiastic discussions, which have led to a successful completion of this work.

I also wish to thank all other members of my guidance committee for their active interest and criticism, and the Department of Electrical Engineering and the City University of New York for the financial support I received.

Finally, I would like to thank my wife, Emily, for her invaluable help and understanding, and my son, Albert, for being patient during this period of my study and preparation of this work.

TABLE OF CONTENS

	Page
1. INTRODUCTION	1
2. DETECTION AND EXPLOITATION OF PARALLELISM IN GENERAL COMPUTATION PROGRAMS	12
2.1 Introduction	12
2.2 Sytactic Tree of An Arithmetic Expression	15
2.3 Minimization of Tree-height with Distribution	22
2.3.1 Notation	22
2.3.2 Hole Function and Space Function	29
2.3.3 Distribution Algorithm to Reduce Tree-height	37
2.4 Minimization of Tree-height with Unequal Operator Weights	39
2.4.1 Notation	39
2.4.2 Extension of Hole Function, Space Function and Tree-height Reduction	45
2.5 Interstatement Parallelism Exploitation	50
2.6 Summary	59
3. JOB GRAPH AND TASK SCHEDULING	61
3.1 Introduction and Job Graph	61
3.2 Optimal Non-preemptive Scheduling	65
3.2.1 Notation and Bounds	65
3.2.2 Optimal Scheduling Algorithms	68
3.3 Suboptimal Heuristic Scheduling Algorithms	76
3.4 Operational Considerations: Performance vs. Cost	82
4. REARRANGEABLE DATA-CONNECTION NETWORKS	90
4.1 Introduction: Data-Connection Requirements	90

Page

4.2 Data-Connection Networks 95

4.3 Decomposition of a 3-stage RSN 104

4.3.1 A Definitive Decomposition Algorithm 104

4.3.2 Complexity, Optimal Factoring, and Special-case Applications 123

4.4 Summary 127

5. CONCLUSION 129

LIST OF REFERENCES 132

AUTOBIOGRAPHIC STATEMENT 135

LIST OF ILLUSTRATIONS

Figure		Page
1.1	Connection network in (a) an SIMD system, (b) an MIMD system	10
2.1	Parallel Computation of an arithmetic expression	19
2.2	An example of holes and tree-height reduction	22
2.3	An example of space and tree-height reduction	23
2.4	Free nodes	24
2.5	An example of tree and free nodes	25
2.6	Another example of tree and free nodes	25
2.7	The tree structure of Fig. 2.5 rearranged	28
2.8	Generation of a higher-level free node from two low-level free nodes	29
2.9	An example of hole function in an expression of the form $E = \Sigma t_i$	32
2.10	An example of hole function in an expression of the form $E = \Pi t_i$	34
2.11	Creation of space	35
2.12	A free node	44
2.13	An example of holes with unequal operator weights	46
2.14	An example of spaces with unequal operator weights	49
2.15	A parse tree-complex of a BAS graph	53
2.16	A parse-tree complex for recursive equations	59
3.1	Job graph and precedence partition	64
3.2	An optimal scheduling of a job tree on two PE's	71
3.3	Preparation of a job graph for backward optimal scheduling	74

Figure	Page
3.4 Scheduling of the graph in Fig. 3.3	76
3.5 An example of heuristic scheduling	79
3.6 An example of worst-case heuristic scheduling	81
3.7 An arithmetic expression used for scheduling analysis	85
3.8 Optimal scheduling for the graph in Fig. 3.7	86
3.9 A plot of speed-up factor and cost-performance index vs. number of processors	87
4.1 Skewed storage of data array in memory modules	92
4.2 A perfect shuffle for N = 8	93
4.3 Row-broadcasting in matrix multiplication	94
4.4 A cross-bar switch	96
4.5 A 4-port barrel shifter	97
4.6 An 8 x 8 Ω-network and a connection path	98
4.7 A general 3-stage rearrangeable switching network	101
4.8 A flow chart for permutation matrix decomposition	111
4.9 An example on the decomposition of a 3-stage RSN	118
4.10 Illustration of possible situations leading to two critical elements in a matrix H	121

1. INTRODUCTION

In the past decade, a major innovation in computer design has been the development of systems with multiple processing capabilities. Many types of computer structures and organizations employing more than one processing unit have been known and often classified under categories such as "multicomputer system", "multiprocessor system", "array processor", and "associative processor", etc. [3],[8],[12], [20],[21],[23]. Each of these systems performs differently from others as the degree of interaction among a particular system's processing units varies from one system to another. Yet one basic common criterion which prevails in these systems is that each one is capable of performing multiple operations concurrently in time. This concurrence of operations can be different input/output activities, or execution of different parts of a program, or computation of different vector elements under the same instructions, etc. Since we are more concerned with the "principles" which constitute the feasibility of such concurrency, rather than identifying the different processes simultaneously going on, we shall simply use a general term "parallel processing systems" to cover these systems which are structurally different but conceptually identical in achieving multiple operations. For convenience, we shall also refer to each processing unit of a system as a PE (for processing element), although this does not necessarily mean that all PE's in a system are

identical.

Parallel processing systems have attracted much attention because they offer many desirable features which are not practically obtainable from a uniprocessor. For example, some of the main advantages are:

(a). Increased processing speed and power. This is achieved mainly due to the fact that a large job may be decomposed into many smaller ones and processed simultaneously on several appropriate units. In other words, this is an application of parallelism. Parallel processing systems allow the detection and exploitation of parallelism within a problem and execute many parts of a task concurrently, or simply have many (unrelated) tasks processed at the same time by sharing different resources of the system. In this way, the net throughput of a system is increased, and a higher utilization of the system resources is achieved.

(b). Reliability and graceful degradation. Parallel systems inherently offer co-operative interactions between their subsystem units, therefore are easier to be reconfigured for back-up or substitutive operation. In case of the occurrence of a subunit failure, the system will not have to be shut down completely but still can operate in a reduced but useful capacity until full recovery is made.

(c). Ease of system expansion. Since more subunits of a parallel system are similar in structure or consist of identical modules, it will be easier to expand the system's capability to meet the growth in processing demand.

(d). Application of new technology. The advancement in hardware technology, specially the prevalence of the LSI, has made it more practical to build many processing units into a system. The traditional concept of a large and mighty computer has now been challenged by the use of many smaller ones in a parallel system to do the same or even better jobs.

While parallel processing systems offer many advantages, they also present additional problems which are not encountered in a uniprocessor environment. The problems arise because in order to utilize a parallel processing system effectively, it is necessary to match the problem awaiting execution with the system configuration as closely as possible. Special treatment in the "pre-processing phase" is required to analyze the problem tasks and organize them in a way to suit the processing capability of the system before they are actually being executed. Then, in the "processing phase" of operation, a system should facilitate the flow of instructions and data communication among the active processors and memory modules, so as not to unduly hold up any piece of information. This requires a controllable connection network which allows information to be dynamically exchanged in a time period comparable to the processing speed. In short, the advantages of parallel processing and the power of such systems can be fully exploited only when these additional problems are properly handled.

It is the purpose of this thesis to investigate these important aspects of parallel processing from the viewpoint of rearrangement of instruction structure and data flow.

Specifically, we study the ways to increase the possibility of simultaneous execution of a given block of tasks through the detection of parallelism between and within the instructions, and how they should be aligned with processing units to minimize execution time by proper scheduling. We then examine the hardware connection networks which can be used to arrange this information flow as prescribed by previous ordering. Thus, detection and exploitation of parallelism, task scheduling, and rearrangeable data connection network are the topics of our discussion. We shall give attention to a "state-of-the-art" discussion on each topic, and present our results which include modification, improvement, and new development. We also show the linkage from topic to topic, so that they are considered as three phases of a problem rather three separated problems.

In the following, we briefly discuss these three areas and define the scope of our investigation.

(A). Parallelism detection and task decomposition

Let a task be any given amount of work to be processed by an appropriate processing unit, then two tasks are simultaneously processable if they do not depend on the processed outcome of each other. Informally we say that parallelism exists between the two tasks. In this sense, there are many possible levels of parallelism, depending on the size of a task one is concerned with:

parallelism between jobs,

parallelism between runs in a job,

parallelism between programs in a run,

parallelism between instructions in a program,

parallelism between suboperations in an instruction.

It is clear that the "principles" involved in detecting parallelism at all levels should be the same, although the actual rules or algorithms to be applied can be quite different. At higher levels above a program, parallelism generally exists because the tasks are simply unrelated, such as in a multiprogramming environment, or are associated with different resources of the system if related. To identify this parallelism and properly partition the tasks is part of the job of an operating system. At lower levels within a program, parallelism can be identified as inter-statement parallelism which exists between instruction statements, or as intra-statement parallelism which exists within a single instruction, particularly of arithmetic expression. We are specially interested in the detection and exploitation of parallelism at these lower levels for the following reasons:

(a). Parallelism at lower levels bears direct influence on the design of machine organizations such as in pipeline processors, array processors, and other types of special-purpose parallel processors. Advancement in hardware technology has promoted construction of such processors through more hardware implementation of low-level parallelism.

(b). For the vast mass of mathematical problems involving matrix computation, numerical analysis, or just arithmetic operation, parallelism essentially exists at lower levels.

Since achieving faster computation in solving mathematical problems is a major motive for the development of parallel processing systems, low-level parallelism detection and exploitation is vital to these application.

(c). Many conventional programs and programming techniques are sequential in nature. Parallelism detection at instruction level and lower is needed to translate them for parallel processing. A "DO" loop is a perfect example for lending itself to parallel processing, and arithmetic expressions have plenty of parallelism imbedded within.

(d). At low levels, one deals with only finite expressions and instructions which have well-defined characteristics and execution times, hence parallelism analysis using deterministic models gives meaningful and accurate results.

(e). Parallelism between and within instructions is fundamental to the design of efficient compilers.

Since arithmetic expressions are the basic building blocks of computation programs, their roles in parallel processing are first examined in Chapter 2. Various techniques have been developed to parse an arithmetic expression for compilation and parallel computation purposes [26],[27]. In particular, Baer and Bovet [1] proposed an efficient compilation algorithm for the case when an arithmetic expression is executed according to the usual rules of operator precedence and all operators are assumed to require the same amount of execution time. Muraoka [22] further considered the use of distributive law to reduce the total computation steps by

making use of some free nodes called holes and spaces. Kraska [15] extended the study to the case of unequal operator times, and introduced some binary operations to compute the minimum tree heights. However, their algorithms are rather lengthy and complicated for practical application. While preserving Muraoka' concept of holes and spaces and Kraska's computation technique, we shall in Chapter 2 give a new presentation on this subject, based on a series of redefined operations which operate directly on tree heights and free nodes. The result is a simpler procedure which is easier and more practical to apply.

Interstatement parallelism has been investigated by Muraoka [22], Kuck and others [16],[17], Ramamoorthy [26], and Bernstein [5], etc. We shall show specifically how various types of instructions can be handled and restructured to yield to concurrent operation.

(B). Scheduling of tasks for execution

Once a job or program is properly partitioned or restructured for execution, the many subtasks thus created have to be distributed to various PE's in a way such that most PE's can keep themselves busy working on the queues of tasks assigned to them and complete the job in the shortest possible time. Scheduling is therefore an important follow-up to link a set of well composed tasks to an actual processing system.

Like parallelism detection, scheduling of tasks can be considered at many levels, depending on the task size and

its processing units. Again, at lower levels, because the task sizes and execution times are known, scheduling can be analyzed from a deterministic approach. As part of a unified study, we shall in Chapter 3 consider the methods of transforming a block of basic operations, obtained from parallelism detection, into a job graph and loading these operations into the operational lists of various PE's for execution. Due to its broad application, scheduling has been investigated by many [10],[11],[13],[14],[28],[30]. We shall discuss some bounds and describe some "better" scheduling algorithms which produce either optimal or near optimal results.

Since scheduling is a practical problem which amounts to the operation cost of a system, it is appropriate for us to make a cost-performance analysis, so that one can have a more practical and total view of the combined merit of parallelism exploitation and scheduling on a real system.

(C). Memory-processor connection networks

Central to the operation of any parallel processing system is the dynamic role of a connection network which allows information to be transferred as required by the result of parallelism exploitation and proper scheduling. This aspect of parallel processing is the subject of investigation in Chapter 4.

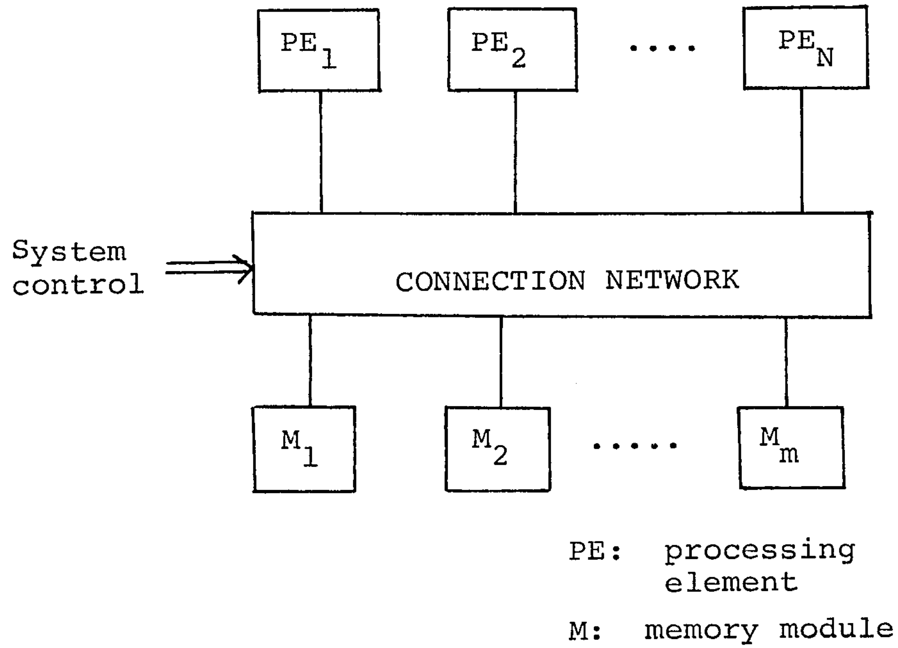
Parallel processing systems can be classified into two major types according to the ways that instruction streams and data streams are managed: i) single-instruction-multiple-data (SIMD), and ii) multiple-instruction-multiple-data (MIMD) .

In an SIMD system, the connection network serves to distribute the same instruction to all PE's at a time, fetch data from some memory modules, align a pair of data for each PE to execute, and then wait for next instruction or realign some data and send them back to the memory. There is also a need for data communication directly between the PE's, bypassing the memory, if some currently computed results are to be shared by other PE's. Connection networks in this type of systems are working mostly on a synchronous basis. Fig. 1.1(a) shows the role of a connection network in such a system.

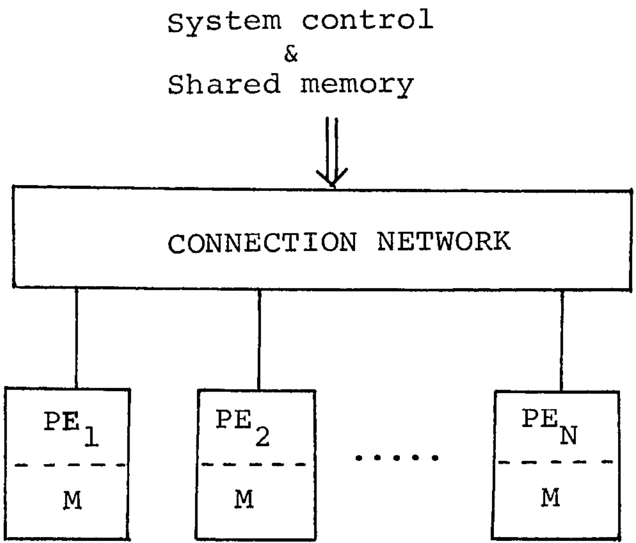
In an MIMD system, most PE's will be operating on an asynchronous basis, under the command of a system control unit. Here a connection network serves as a link between the system control unit and all PE's, and also as the channels for inter-PE communication. In this case, the connection network takes its basic role as shown in Fig. 1.1(b).

Connection networks are of course also needed in the sharing of other system resources such as input/output peripheral devices, although they may not require the same degree of dynamic connectivity.

The simplest and most popular type of controllable connection network is a cross-bar switch. It is simple to control and makes fast connections. The main disadvantage of this switch is its hardware cost, since the number of cross-points, N^2, rises sharply as the number of terminals, N, increases. As alternatives, many types of controllable



(a)



(b)

Fig. 1.1. Connection network in (a) an SIMD system, (b) an MIMD system.

connection networks have been proposed, some of which are:

multistage rearrangeable switching network (RSN) [4],[7],

barrel shifter,

Ω-network [18],[19],

sorting network [2].

Above connection networks differ considerably in structure, control scheme, and connection capability. We shall investigate the first three types in terms of these charcteristics.

In particular, we see a multistage RSN as a potential network which has full connection capability and allows some freedom in the choice of cost and speed trade-off. However, the decomposition and control scheme is not a simple process. Some algorithms of heuristic nature have been described [6], [24],[25],[29]. But we shall devote a main part of Chapter 4 to the description and proof of a new definitive decomposition algorithm. It will be shown to be an effective algorithm which generates the required path controls through all stages simultaneously to complete the prescribed connections.

2. DETECTION AND EXPLOITATION OF PARALLELISM IN GENERAL COMPUTATION PROGRAMS

2.1 Introduction

In this chapter we study the algorithmic detection of parallel processable operations within a program. Logically, this means that the program language and its structure under examination are basically sequential in nature. For example, the Fortran language and its programs represent a line of sequential operations which the programmer or user wishes to do, although in actuality, some of these operations may be performed concurrently. To detect a maximum amount of parallelism existing in a program and hence minimizing the program execution time is the goal of this chapter.

For convenience, expressions similar to the format of the Fortran language will be used, although practically the analysis should be independent of the particular language used. We shall consider three types of the so-called "executable" instruction statements which are data dependent. These are:

(a) Arithmetic expressions, e.g., assignment statements,

(b) Indexed computations, e.g., DO statements,

(c) Conditional branchings, e.g., IF statements.

We first consider the detection of parallel processable suboperations within a statement, particularly of an arithmetic expression. We shall use the terms "arithmetic expression" and "arithmetic statement" interchangeably to mean the same. This type of parallelism is referred to as intra-

statement parallelism. Then we will consider simultaneous execution of several statements either of the same type or different. This is referred to as inter-statement parallelism.

Decomposition of a general arithmetic expression to show the syntactic relations between its suboperations has been of great interest to the designers of compilers, even for uniprocessor systems, since it is essential to determine the allowable sub-operations and their sequencing. In this context an analysis is usually done with the aid of a special graph called a syntactic tree or parse tree. The same parse tree also shows the parallel processable subtasks if the execution times for the subtasks are also associated with. A number of different parsing techniques have been developed [26],[27], assuming equal time for all basic operations. Among them a procedure by Baer and Bovet [1] using operator precedence order is generally considered the best to produce a tree with minimum height, if the form of an expression is not altered other than through the use of associative and commutative properties. This scheme also bears more naturally to the way that arithmetic expressions are commonly executed (for example, compared to the use of a reverse Polish form).

Muraoka [22] considered the use of distribution to further reduce a tree height under the same unit operation-time assumption. This makes additional improvement in tree-height reduction possible, even though the preprocessing effort is also increased. Kraska [15] then extended the investigation

to the case with different operation times. While his analysis is more realistic, the complexity of his procedure prohibits practical application. Both Muraoka and Kraska based their computation of tree heights first on finding the maximal widths of subtrees (which they call effective lengths), which in turn are functions of subtree heights. In other words, if we consider tree heights as vertical quantities and their widths as horizontal ones, in general (except in very elementary cases) we go from vertical to horizontal and then back to vertical in doing this computation. This indirect approach appears to be inefficient. In this chapter, we describe a new approach which has two distinct advantages:

(1) A tree-height is directly computed from its subtree heights.

(2) All height functions are represented directly in a special binary number system in which simple operations can be easily performed.

As a result, the whole distribution algorithm becomes easier and more practical to be implemented in a parallel processing system.

Inter-statement parallelism has been investigated by Muraoka [22], Kuck and others [16],[17], Ramamoorthy [26], and Bernstein [5], etc. Based on the basic principle that statements with independent inputs are parallel processable, we shall examine how this is to be applied to the three different types of statements in a Fortran-like program so that the overall parallelism within the program is best exploited.

2.2 Syntactic Tree of an Arithmetic Expression

A common way to show the syntactic structure of an arithmetic expression for the purpose of compilation and parallel execution is to use a special tree in which each node represents a binary operation on two operands transmitted by the inward arcs. To facilitate the discussion, we first define a number of terms.

Definition 2.1:

A tree is a set of nodes and directed arcs (edges) in a leveled structure whose levels are numbered downward from 0 to, say, q in increasing order. Each node at some intermediate level j, $0 < j < q$, has two or more inward arcs emanating from some nodes at level i, where $i \leq j-1$, and each node at level j has only one outgoing arc to a node at some level k, where $j+1 \leq k \leq q$. Nodes at level 0 have no inward arcs and are called initial nodes. There is only one node at the bottom level q of the tree. It has no outgoing arcs, and is called the root of the tree, or the terminal node.

For brevity, unless otherwise specified, a node will in general refer to one between the initial and the terminal levels.

Definition 2.2:

Let the root of a tree T be at level q, then $h[T] = q$ is the height of the tree T. In general, if n_i is a node in T, then $h[n_i]$ indicates the level or height of the node n_i.

Definition 2.3:

If there exists at least one directed arc from n_i to

another node n_j, where $h[n_j] > h[n_i]$, n_j is called a successor of n_i, and n_i a predecessor of n_j, denoted by $n_i < n_j$. If there is only one arc between n_i and n_j, they are called immediate predecessor and immediate successor, respectively, to each other.

Definition 2.4:

A binary tree is a tree each of whose nodes has exactly two inward arcs.

Definition 2.5:

A syntactic tree T[E] of an arithmetic expression E is a binary tree each of whose nodes represents a binary operation in E and a datum as the result of the operation performed on the two data carried in from its two immediate predecessors. We define h[E] = h[T] as the height of the expression E represented by the tree T.

It is important to note that a node represents an "operator" for its predecessors and an "operand" for its successors. In this context, the term "node" may be used interchangeably with "operator" or "operand" when no confusion may arise.

In an arithmetic expression, operands are combined according to the rules of operator precedence which in decreasing priority are:

precedence 3 : $\uparrow$, (exponentiation)

precedence 2 : *,/, (multiplication, division)

precedence 1 : +,-, (addition, subtraction)

precedence 0 : ( ). (parentheses)

To bear useful information on parallelism exploitation, the construction of a syntactic tree must depend on the operator precedence and the operator weight which is the amount of time needed to perform the operation. In the simplest analysis only the four basic operators (*, /, +, -) are considered and they are assumed to have the same unit operator weight. Exponentiation and parentheses are treated in the usual way. That is, an expression is first compiled from inside the inner-most parentheses, until a single operand (datum) is obtained. This implies that only the associative law and commutative law will first be used here to construct a syntactic tree. Use of the distributive law will be considered in later sections.

Under these simplified conditions, we have the following basic algorithm by Baer and Bovet [1] for generating a syntactic tree of a parenthesis-free arithmetic expression, involving the four basic operators. It is based on operator-precedence order and multiple-pass scannings. During a pass, scanning procedes from left to right and each operand and each operator is scanned only once. Whenever two operands can be joined without violating the operator precedence rule, a temporary result is produced and inserted in the output string and as a new node for next higher level. After one pass, similar scanning is performed on the next higher level and so on, until the whole expression has been compiled. The total number of passes is thus equal to the number of levels in the syntactic tree.

In the following algorithm, ITEM represents the operand currently scanned, LOP, the operator to the left of ITEM, ROP, the operator to the right of ITEM, and P(variable), the precedence of an operator variable. When an operand cannot yet be included in the output string, it is stored in STACK(1) and its ROP in STACK(2).

Algorithm 2.1:

Step 1. Scan the next two symbols, ITEM and ROP.

Step 2. If (a) P(LOP) $<$ P(ROP) or (b) STACK is empty, store ITEM and ROP on top of STACK(1) and STACK(2), respectively and go to Step 1. (The content of LOP is initially assigned "+" and later assumes the value of the previous ROP).

If P(LOP) $\geq$ P(ROP), go to Step 3.

Step 3. If P(LOP) = P(STACK(2)), generate a temporary result of the form t_k = STACK(1).STACK(2).ITEM and add t_k and ROP to the output string. Push up the stack by one and go to Step 1.

If P(LOP) $>$ P(STACK(2)), add ITEM and ROP to the output string and go to Step 1.

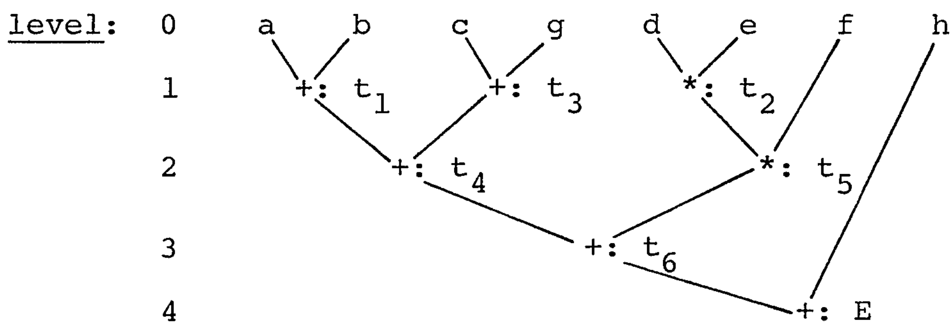
If all operands are scanned, one pass is completed. The output string generated becomes the input string for the next higher level. The procedure from Step 1 to Step 3 is repeated until the whole expression is compiled.

An example for the application of this algorithm is given in Fig. 2.1, which shows in part (a) the detail of the first pass, and in part (b) the parse tree of the expression.

Initial input string E = a + b + c + d * e * f + g + h

LOP	ITEM	ROP	STACK(1)	STACK(2)	Temporary Result	Output String
+	a	+	a	+		0
+	b	+			t_1 = a+b	t_1 +
+	c	+	c	+		t_1 +
+	d	*	d	*		t_1 +
			c	+		
*	e	*	c	+	t_2 = d*e	t_1 + t_2 *
*	f	+	c	+		t_1 + t_2 * f +
+	g	+			t_3 = c+g	t_1 + t_2 * f + t_3 +
+	h	.				t_1 + t_2 * f + t_3 + h.

(a)



(b)

Fig. 2.1. Parallel computation of an arithmetic expression. (a) Stack operation of the first pass. (b) The parse tree of E.

Algorithm 2.1 is complete if "/" and "-" occur only as the ROP of the first operand to be combined in a temporary result. However, if they occur at the end of the last output string, or as the LOP of the first operand, some modification in Algorithm 2.1 is necessary to change the operator. This is illustrated in the following.

(A) For subtraction operator "-".

a) If the last ROP in the output string is "-" and the LOP of the second operand has same precedence, then the second LOP is changed from "+" to "-" and vice versa. For example, $E_1 = a + b - c - d$ will be compiled as:

$t_1 = a + b, \quad t_2 = c + d$

$E = t_1 - t_2.$

b) Unary minus is avoided or delayed in the generation of the temporary result, if possible. For example, $E_2 = -a + b + c + d$ is compiled as:

$t_1 = b - a, \quad t_2 = c + d$

$E_2 = t_1 + t_2.$

Also, $E_3 = -a - b - c - d$ is compiled as:

$t_1 = a + b, \quad t_2 = c + d$

$t_3 = t_1 + t_2$

$E_3 = -t_3$, and

$E_4 = -a * b - c + d$ compiled as:

$t_1 = a * b, \; t_2 = d - c$

$E_4 = t_2 - t_1.$

(B) For division operator "/".

a) If the LOP of both operands are "/", the second LOP is changed to "*". For example,

E_5 = a / b / c / d is compiled as:

$t_1 = a / b$, $t_2 = c * d$

$E_5 = t_1 / t_2$.

b) If the LOP of the first operand is "/" and the LOP of the second is "*", the two operands and operators are interchanged in the order of execution. For example,

E_6 = a * b / c * d is compiled as :

$t_1 = a * b$, $t_2 = d / c$

$E_6 = t_1 * t_2$.

Algorithm 2.1 with above supplements then generates a syntactic tree of minimum height in a parenthesis-free arithmetic expression. All temporary results generated during a pass create new nodes at next higher level and their operations can be executed simultaneously to speed up the computation. If there are n operands at level 0, then by binary combination the minimum height it can have in the parse tree is $\lceil \log_2 n \rceil$, where the notation $\lceil x \rceil$ denotes the least integer such that $\lceil x \rceil \geq x$. (The notation $\lceil \ \rceil$ will be used consistently through-out the thesis.) If sufficient processing elements are used, parallel processing can yield a theoretic maximum speed-up factor of $(n-1) / \lceil \log_2 n \rceil$ over a uniprocessor system, assuming that all processors have equal speed.

2.3 Minimization of Tree-height with Distribution

2.3.1 Notation

In this section we investigate the use of distribution to further reduce the tree-height of an arithmetic expression assuming equal weight for all operators. Our discussion is evolutionary from those of Muraoka [22] and Kraska [15]. However, in our new approach, many definitions and operations are redefined. Our objective is to relate all height functions directly in a special binary system with some simple operations so that the overall distribution algorithm becomes easier to apply.

We first introduce the concept of holes and spaces and show the effect of distribution on tree-height reduction with simple examples. Consider the expression $E_1 = a(bcd + e)$ and its distributed form $E_1^d = abcd + ae$ with their snytactic trees shown in Fig. 2.2 as a result of the compilation algorithm stated in section 2.1.

Fig. 2.2. An example of holes and tree-height reduction.

While $h[E_1] = 4$, we see $h[E_1^d] = 3$ in the above example. The reason is that in E_1 the variables d and e are not combined with any other variables until level 2 and after, while they can actually take the operand "a" at level 1 and save

an extra step. The circles on some arcs show the idleness of the operands and are examples of "holes". E_1^d simply fills these holes with useful operations.

Next consider $E_2 = a(bc + d) + e$ and $E_2^d = abc + ad + e$ and their trees in Fig. 2.3.

Fig. 2.3. An example of space and tree-height reduction.

Again, we have $h[E_2] > h[E_2^d]$. In this case, the tree height is reduced by distribution because after "a" is distributed, the expression becomes a sum of product terms and it allows the variable "e" to be combined in an earlier step instead of waiting till the last step. The workable place which is created by distribution is an example of "space", as shown by the circled node operation.

For continuity, we show the follow definition and lemma [22].

<u>Definition 2.6:</u>

Let h[E] be the height of T[E]. Define $e[E] = 2^{h[E]}$ as the effective length of an expression E.

<u>Lemma 2.1:</u>

(a) If $E = \prod_{i=1}^{p} a_i$ or $E = \sum_{i=1}^{p} a_i$, where a_i are single variables and π and Σ are product and sum, respectively, then $h[E] = \lceil \log_2 p \rceil$.

(b) If $E = \sum_{i=1}^{p} t_i$, where t_i are single variables or products of variables, then $h[E] = \lceil \log_2 \ (\sum_{i=1}^{p} e[t_i]) \rceil$.

(c) If $E = \prod_{i=1}^{p} t_i$, where t_i are single variables or sums of variables, then $h[E] = \lceil \log_2 \ (\sum_{i=1}^{p} e[t_i]) \rceil$.

[Proof]: (a) is obviously true. Extending this to include a composite term t_i and treating it as a single expression, (b) and (c) are obtained. (Q.E.D.)

Now we introduce some definitions to facilitate further discussions.

Definition 2.7:

Let t_i and t_j be two subexpressions of an arithmetic expression E joined by any binary operator. Without loss of generality, let $h[t_i] = h[t_j] + s$. Then if $s > 0$, there will be s nodes on an arc to which no trees other than $T[t_j]$ are attached. These s nodes are called free nodes, whose heights are $h[t_j] + 1, \ldots, h[t_j] + s = h[E] - 1$. This is illustrated in Fig. 2.4.

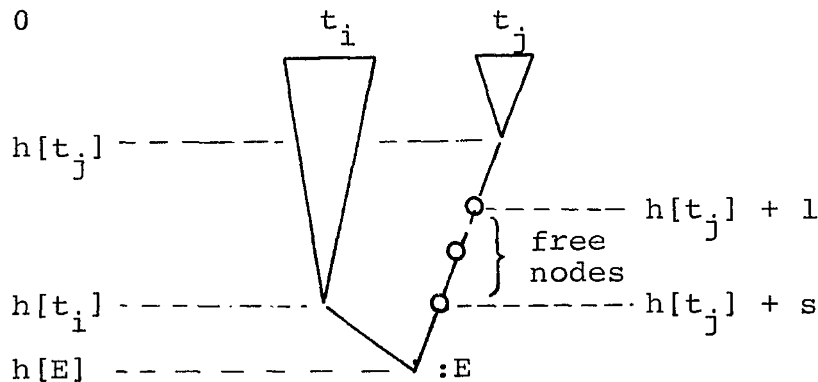


Fig. 2.4. Free nodes.

Definition 2.8:

Let $E = \sum_i t_i$ or $\prod_j t_j$ as in Lemma 2.1. Define $F[E,t_i]$ as the set of all free nodes which exist between the roots of $T[t_i]$ and $T[E]$, and $F[E]$ as the set of all free nodes which exist between the roots of $T[E]$ and all $T[t_i]$. The following examples illustrate these definitions.

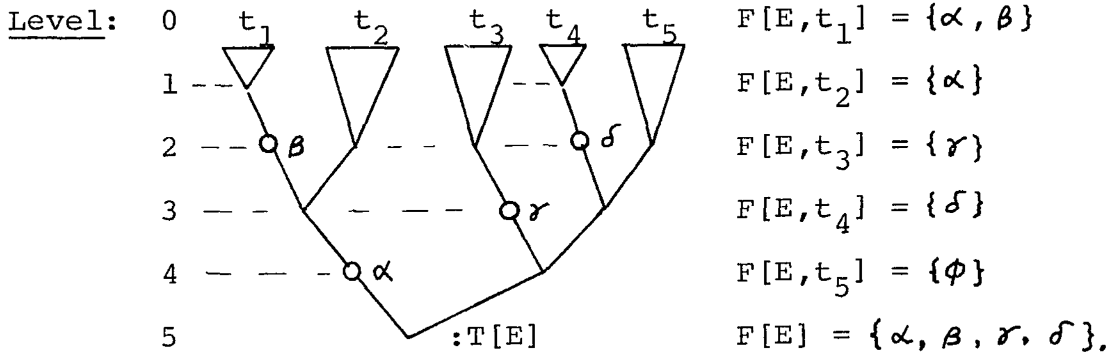


Fig. 2.5. An example of tree and free nodes.

For $E = (a + b) * (cde + f) = t_1 * t_2$

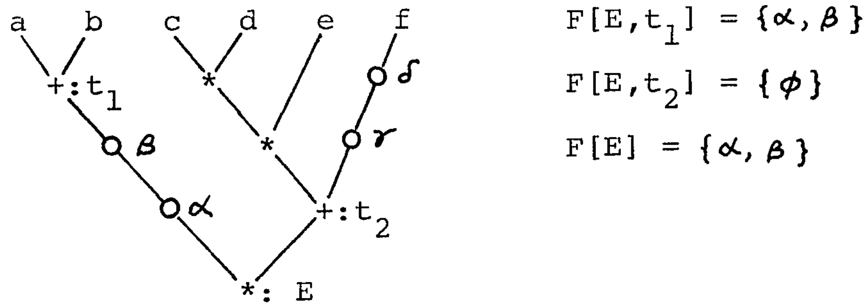


Fig. 2.6. Another example of tree and free nodes.

Definition 2.10:

Let M be a positive integer set. Define $[M]_b$ as a set of binary numbers $b_i b_{i-1} \ldots b_1 b_0$ where $b_i b_{i-1} \ldots b_1$ is the regular binary equivalent of m for every $m \in M$, and $b_0 = 0$. For example, if $M = \{9,13\}$, $[M]_b = \{10010, 11010\}$.

Definition 2.10:

The union operation U on s_1 and s_2 is a collection of all elements in sets s_1 and s_2.
for example, if $s_1 = \{1001\}$, $s_2 = \{1101\}$,

$s_1 \cup s_2 = \{1001, 1101\}$.

Definition 2.11:

Let I be a non-negative integer set. Define H[I] as a set of binary numbers so that for every $i \in I$, there is a binary number $b_i b_{i-1} .. b_1 b_0 \in H[I]$, where $b_i = 1$ and all other bits are 0. Since for any two different integers their binary representations in H[I] are also different (i.e., the "1" bits at different positions), we can, as a convenient short form for union operation, write the distinct numbers in a composite form. For example, let

$I = \{0, 1, 2, 2, 3, 4\}$, then

$H[I] = \{1, 10, 100, 100, 1000, 10000\}$

$= \{11111, 100\}$.

Definition 2.12:

Let a and b be two integers, then define a discrete sum operator "$+_d$" as follows:

$a +_d b = \text{Max}(a,b) + 1$, with the sum expressed in a binary number as defined in Definition 2.11.

In terms of the actual binary operation, the discrete sum operation means that the larger of (a,b) has its significant bit moved up to the left by one place, which then replaces both a and b. For example,

$10 +_d 100 = 1000$.

It is easy to see that when a = b, the discrete sum

operation is the same as the ordinary binary addition.

We use $H_+[I]$ to mean a discrete summation on a set of integers [I]. The two smallest numbers are first combined using the discrete sum operator $+_d$. The operation is then repeated on all remaining elements of the set and the intermediate result thus far obtained until a binary number with only a single non-zero bit is obtained.

We also use $\lceil H \rceil_+$ as a notation for performing discrete summation on the binary numbers enclosed in a set H. For example, if $I = \{0, 1, 2, 2\}$,

$H_+[I] = H_+[0, 1, 2, 2] = \lceil 1, 10, 100, 100 \rceil_+$.

To find the discrete sum, we first combine 1 and 10,

$\lceil 1, 10 \rceil_+ = 1 +_d 10 = 100$. Then,

$\lceil 100, 100 \rceil_+ = 1000$, and finally,

$\lceil 1000, 100 \rceil_+ = 10000$. Hence, $H_+[I] = 10000$.

Definition 2.13:

Define mxb H[I] as an integer representing the largest leading bit among the binary numbers in the set H[I]. For example, if $H[I] = \{1, 10, 100, 1000\}$,

$\text{mxb } H[I] = \text{mxb}\{1000\} = 3$.

The purpose of above definitions and operations is to allow a tree height to be computed more easily in lieu of Lemma 2.1 without actually constructing a tree. In this case let I be the set of subtree heights $h[t_i]$, and then h[E] is equivalent to $H_+[I]$. Or, $H[h[E]] = H_+[I]$.

As a short-form notation, when X is an expression, a tree, or a node, we define $H[X] = H[h[X]]$.

For example, in Fig. 2.5 let all operators be of the same type, we have $h[t_1] = h[t_4] = 1$, $h[t_2] = h[t_3] = h[t_5] = 2$. Thus, $I = \{1, 1, 2, 2, 2\}$, and

$$H_+[I] = \lceil 10, 10, 100, 100, 100 \rceil_+ = 10000.$$

Hence, $H[E] = \{10000\} = 10000$. Or, equivalently, $h[E] = 4$.

This actually means that the tree of Fig. 2.5 can be reconstructed as shown in Fig. 2.7 which has only 4 levels.

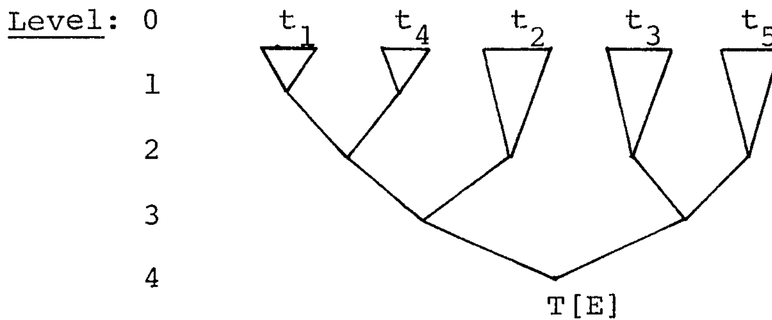


Fig. 2.7. The tree structure of Fig. 2.5 rearranged.

The ordinary sum operators + and Σ are used to combine some free nodes of lower level into higher nodes. This is presented as an important lemma.

Lemma 2.2:

Let $\{\alpha, \beta\} \in F[E]$ be two free nodes in $T[E]$, where $E = \Sigma\, t_i$ or $E = \Pi\, t_i$, and $h[\alpha] = h[\beta]$, then α and β can be replaced by one free node γ whose height is $h[\alpha] + 1$ without altering the tree height.

[Proof]: Assume $T[t_1]$ and $T[t_2]$ are the subtrees associated with α and β, respectively. Since operators of the same type are used to combine all subtrees, we can change the order

(by the associative law) and combine t_1 and t_2 first. This eliminates α and β and creates a new free node γ at level $h[\alpha] + 1$. The situation is illustrated in Fig. 2.8. (Q.E.D.)

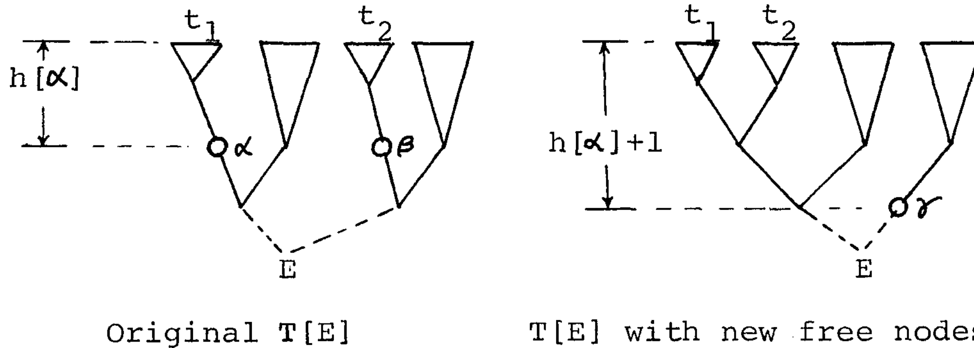


Fig. 2.8. Generation of a higher-level free node from two low-level free nodes.

Applying Lemma 2.2 repeatedly on a set of free nodes, we can obtain a binary number representing the highest level of an equivalent free node.

However, it is important to observe that free nodes within different subtrees cannot in general be combined in a similar way because the associative law does not apply among different operators. Hence, although some free nodes may be combined as shown above, some will have to be listed separately.

2.3.2. Hole Function and Space Function

We now examine how free nodes in different parts of a tree may be actively used for attaching additional subtrees without increasing its own height. We first give two fundamental lemmas. It is easy to see from the definition of

free nodes (Definition 2.7) that the following lemma is true.

Lemma 2.3:

Let t_i and t_j be joined by an operator &, where & = + or *, and α, a free node in $T[t_i \& t_j]$, then $h[t_i \& t_j] = h[t_i \& t_j \& t_k]$ for any t_k such that $h[t_k] \leq h[\alpha] - 1$, where & is used consistently throughout the expressions.

Since our objective is to use distribution to reduce a tree height, a basic property is shown next.

Lemma 2.4:

Let $E_1 = \sum_{i=1}^{m} t_i$ and $E_2 = \sum_{j=1}^{n} t'_j$ where $m,n \geq 2$. If $E = (\sum_i t_i) * (\sum_j t'_j)$ and

$$E^d = \sum_{i=1}^{m} \cdot \sum_{j=1}^{n} t_i t'_j = t_1 t'_1 + t_1 t'_2 + \ldots + t_m t'_n, \text{ then}$$

$$h[E^d] \geq h[E].$$

[Proof]: Without loss of generality let $h[E_1] \leq h[E_2]$, then $h[E] = h[E_2] + 1$. Since $m \geq 2$, E^d has m times as many terms as E_2 has and each $h[t_i t'_j] \geq h[t'_j]$. By Lemma 2.1,

$$h[E^d] \geq \lceil \log_2 (m \sum_{j=1}^{n} e[t'_j]) \rceil = \lceil \log_2 m + \log_2 (\sum_{j=1}^{n} e[t'_j]) \rceil$$
$$\geq 1 + h[E_2].$$

Hence $h[E^d] \geq h[E]$. (Q.E.D.)

Lemma 2.4 indicates that for an expression $E = (\sum_i t_i) * (\sum_j t'_j) = A * (\sum_j t'_j)$, distribution will help to reduce the tree height only if A is treated as a whole atom.

Now, a hole function which is a binary number representing levels of the existing active free nodes is derived. We discuss this according to different forms of an expression.

(a) If $E = \prod_{i=1}^{p} a_i$ where a_i are single variables. By the binary product operator, E must have $\lceil \log_2 p \rceil$ levels, that is, $h[E] = \lceil \log_2 p \rceil$. Since only p atoms are attached, this leavea $2^{\lceil \log_2 p \rceil} - p$ free nodes. Decomposing this number into binary corresponding to different levels, we obtain the hole function of E as

$$f_h[E] = [\, 2^{\lceil \log_2 p \rceil} - p]_b,$$

and, $f_h[E] = \emptyset$ if $2^{\lceil \log_2 p \rceil} - p = 0$. (2.1)

For example, if E = abcde,

$$f_h[E] = [2^3 - 5]_b = [3]_b = \{110\} = 110.$$

(b) If $E = \sum_{i=1}^{p} t_i$ where t_i are terms composed of product of atoms (an atom is any subexpression treated as a whole). Since when a factor A is distributed over Σt_i, A goes into each of the sum terms t_i, hence, in order to accomodate some A without increasing h[E], each t_i must have at least a set of free nodes whose maximum heights are greater than or equal to h[A] + 1 (see Lemma 2.3). In other words, the effective hole function is determined by the minimum of the set of free nodes existing within each $T[t_i]$ plus those between roots of $T[t_i]$ and T[E]. We therefore search successively from the highest level down, until some $T[t_i]$ has no more free nodes. So, the procedure to find $f_h[E]$ is as follows:

(1) For each t_i find the set of free nodes existing inside $T[t_i]$ and between the roots of $T[t_i]$ and $T[E]$. Let this be $f_h'[t_i]$. That is,

$$f_h'[t_i] = f_h[t_i] \cup \{ H[\alpha] \mid \alpha \in F[E,t_i]\}. \qquad (2.2a)$$

(2) Let x_i be the largest binary number in $f_h'[t_i]$ for each t_i, find the minimum of these, $m = \min_{\text{all } i} \{ x_i\}$. (2.2b)

Add m as an element in the set S, and eliminate or subtract m from each set $f_h'[t_i]$.

(3) Repeat Step (2) with the remaining $f_h'[t_i]$ until some $f_h'[t_i]$ becomes null. Then $f_h[E] = S$. (2.2c)

For example, in Fig. 2.9 let $E = ab + cde + ghi$

$= t_1 + t_2 + t_3$.

$f_h'[t_1] = \{\emptyset\} \cup \{100\} = \{100\}$

$f_h'[t_2] = \{10\} \cup \{\emptyset\} = \{10\}$

$f_h'[t_3] = \{10\} \cup \{1000\} = \{1010\}$.

Applying Step (2), we have $S = \{10\}$ and the new sets:

$f_h'[t_1] = \{10\}$

$f_h'[t_2] = \emptyset$

$f_h'[t_3] = \{1000\}$.

Since $f_h'[t_2] = \emptyset$, the procedure stops and $f_h[E] = \{10\} = 10$.

Which means the highest level of active free nodes is one.

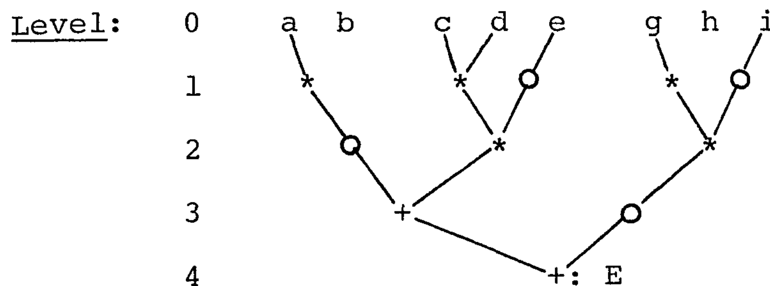


Fig. 2.9. An example of hole function in an expression of the form $E = \Sigma t_i$.

(c) If $E = \prod_{i=1}^{p} t_i$ where t_i are terms composed of sum of atoms.

When a factor A is distributed over $\prod_i t_i$, it goes into only one of the product terms. Hence, in order to accomodate A without increasing h[E], it is necessary and sufficient that at least one set of free nodes associated with a term t_i has a hole whose height is greater than or equal to h[A] + 1. Now the effective hole function is a union of all free nodes:

$$f_h[E] = \bigcup_{i=1}^{p} \{ (f_h[t_i]), \Sigma H[\alpha] \mid \alpha \in F[E] \}. \qquad (2.3)$$

For example, let E = (abc + d) * (efg + h) * (i + j)

$$= t_1 * t_2 * t_3.$$

Applying procedure from (b), we have for each term

$f_h[t_1] = \{10\}$

$f_h[t_2] = \{10\}$

$f_h[t_3] = \emptyset$.

By definition of F[E], we find free nodes at levels 2,3, and 4. Hence,

$$\begin{aligned} f_h[E] &= \{10\} \cup \{10\} \cup \{\Sigma H[2,3,4]\} \\ &= \{10, 10\} \cup \{100 + 1000 + 10000\} \\ &= \{10, 10\} \cup \{11100\} \\ &= \{10, 11110\}. \end{aligned}$$

The active free nodes are shown in Fig. 2.10.

E = (abc + d) * (efg + h) * (i + j).

Level: 0 a b c d e f g h i j

1

2

3

4

5 *: E

Fig. 2.10. An example of hole function in an expression of the form $E = \pi t_i$.

The following example further illustrates the combined application of above definitions and procedures.

$E = (a + b) * (c + def) + ghi = t_1 * t_2 + t_3 = t_4 + t_3$.

$f_h[t_1] = \emptyset$, $h[t_1] = 1$,

$f_h[t_2] = \{10\}$, $h[t_2] = 3$, hence $h[t_4] = 4$.

Since $F[t_4]$ contains free nodes at levels 2 and 3, then

$f_h[t_4] = \{10\} \cup \{\Sigma H[2,3]\} = \{10\} \cup \{100 + 1000\} = \{1110\}$.

Also $f_h[t_3] = \{10\}$ and $h[t_3] = 2$ and $F[E,t_3]$ contains free nodes at levels 3 and 4, we have thus

$f_h'[t_4] = \{10, 1100\} \cup \emptyset = \{1110\}$

$f_h'[t_3] = \{10\} \cup \{11000\} = \{11010\}$.

Applying equations (2.2b) and (2.2c) on $f_h'[t_4]$ and $f_h'[t_3]$ we have finally,

$f_h[E] = \{1110\}$.

Next we consider a different role in which free nodes are used to reduce tree height through distribution, namely, the use of space function. The space function is defined only for an arithmetic expression of the form

$$E = t * (\sum_i t_i) + t' = t * A + t'$$

and distribution of t over A is concerned. If we look again at the introductory example shown earlier, namely, $E = a(bc + d) + e$, we can generalize the tree structure as shown in Fig. 2.11.

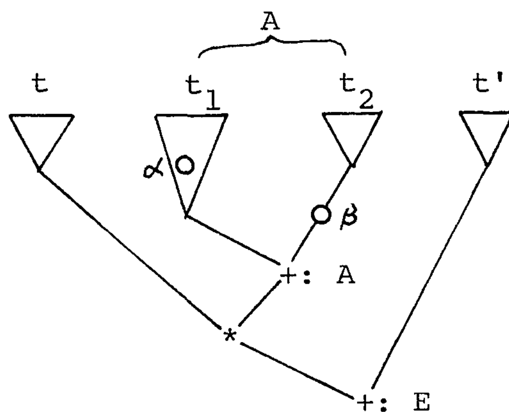


Fig. 2.11. Creation of spaces.

For the discussion of space function, we exclude the use of hole function in T[A], i.e., we assume T[A] does not have holes to accomodate t. Because if that were not the case, we would make use of the hole function to distribute t over A anyway. This implies that $h[t * A] \geq h[A] + 1$. Notice in the tree of Fig. 2.11, although there is a free node in T[A], we cannot directly use it to attach t'. However, if distribution of t over A is done, then the expression for E becomes a sum of terms, $E^d = \sum_i (t * t_i) + t'$

and t' may be absorbed by a created free node associated with some term $t * t_i$. This situation is examined in the following:

(a) If there is a hole α in some $T[t_i]$ to take $T[t]$ and $h[t_i] > h[t]$, then certainly t can be absorbed in $T[t_i]$ with $h[t * t_i] = h[t_i]$ and free node α' is created at a level $h[\alpha'] = h[t_i] + 1$, because one multiplication height is saved.

(b) If there is a free node β (not a hole) in T[A] such that $h[\beta] > h[t]$, then after distribution of t over A, each $h[t * t_i]$ is increased by one, and the free node β is also elivated by one level. That is, if the new free node is β' then $h[\beta'] = h[\beta] + 1$. (If $h[\beta] \leq h[t]$, then after distribution, $h[t * t_i] = h[t] + 1$. The free node is below the resultant subtree height $h[t] + 1$, and cannot be used.)

The created free nodes α' and β' in above situations are called spaces, and are available for attaching some other sum terms. Also these free nodes are accumulative (in the sense of Lemma 2.2) because now the whole expression is in a sum form for each subtree. Accordingly, we define the space function f_s of an arithmetic subexpression $A = \Sigma t_i$ with respect to another factor t in $E = t * (\sum_i t_i) + t'$ as the summed binary height of these spaces.

Definition 2.14:

For an expression $E = t * (\sum_i t_i) + t'$, the space function is $f_s[E] = f_s[A,t] = \Sigma \{ H[t_j], H[\beta] \}$, (2.4) where j is a term index such that $T[t_j]$ contains a hole α and $h[t_j] > h[\alpha] > h[t]$, and $\beta \in F[A]$ is a free node of

T[A] such that $h[\beta] > h[t]$. The hole α or its height $H[\alpha]$ is obtained from the hole function $f_h[A]$.

In the example $E = a(bc + d) + e = t(t_1 + t_2) + t'$ (see Fig. 2.3), $\alpha = \emptyset$, and there is a β with $h[\beta] = 1$. Hence, $f_s[E] = H[\emptyset, 1] = \{10\} = 10$.

We summarize the discussion of hole function and space function with the following theorem.

<u>Theorem 2.1</u>:

Let $E = t * A + t'$ and $E^d = (t * \bar{A})^d + t'$, where $(t * \bar{A})^d$ represents the expression when t is distributed over A, the following are true:

(a) $h[(t * \bar{A})^d] = h[A]$ if $h[t] \leq \text{mxb}\{f_h[A]\} - 1$.

(b) $h[E^d] = h[E] - 1$ if $h[t'] \leq \text{mxb}\{f_s[A,t]\}$.

[Proof]: Both parts are true because of the ways that hole function and space function are defined and derived. Namely, $f_h[A]$ contains the levels of the holes which can be used to attach other trees by product operations. Hence if $h[t] \leq \text{mxb}\{f_h[A]\} - 1$, part (a) is true. In part (b), $\text{mxb}\{f_s[A,t]\}$ itself indicates the maximum height of another tree which can be covered by the tree of $(t * \bar{A})^d$, hence the addition of t' does not show an additional step in the resultant tree and $h[E^d] = h[E] - 1$. (Q.E.D.)

2.3.3. Distribution Algorithm to Reduce Tree-height

With the hole and space functions created in previous section, we can describe their application in arithmetic expressions as two suboperations.

(a) Operation H (Hole-filling)

Let $E = t * A = (\prod_{i=1}^{p} t_i) * A$. Arrange t_i such that $h[t_1] \leq h[t_2] \leq \ldots \leq h[t_p]$. Then do:

(1) Let $i = 1$ and $E_0 = A$.

(2) Check if $h[t_1] \leq mxb\{f_h[E_{i-1}]\} - 1$. If so, distribute t_i over E_{i-1}. Call the result as E_i and go to step (3). Otherwise stop.

(3) Evaluate $f_h[E_i]$.

(4) If $i < p$, let $i = i + 1$ and go to step (2). Otherwise $(i = p)$, stop.

(b) Operation S (Space-filling)

Let $E = t * A + \sum_{j=1}^{q} t'_j$ and $h[t'_1] \leq h[t'_2] \leq \ldots \leq h[t'_q]$.

(1) Evaluate $f_s[A,t]$ from $f_h[A]$.

(2) Check if $h[t'_1] \leq mxb\{f_s[A,t]\}$. If so, distribute t over A. Otherwise stop.

(3) Find the maximum sum of k terms $\sum_{j=1}^{k} t'_j$ such that $h[\sum_{j=1}^{k} t'_j] \leq mxb\{f_s[A,t]\}$.

(4) Evaluate E in the form $E' = [(t * \bar{A})^d + \sum_{j=1}^{k} t'_j] + \sum_{j=k+1}^{q} t'_j$ and stop.

With above operations as substeps, the complete distribution algorithm is the following:

Algorithm 2.2:

(1) Go to the innermost parenthesis level in the expression.

(2) Obtain the hole function for the subexpression inside the parenthesis pair and apply Operation H for hole-filling.

(3) Find the space function and apply Operation S for the resultant expression from step (2) if possible.

(4) Go to the next innermost parenthesis-pair level.

(5) Repeat steps (2) ~ (4) until all parenthesis levels are checked, then stop.

2.4. Minimization of Tree-height with Unequal Operator Weights

2.4.1. Notation

In this section we extend the discussion of previous section to consider the more general case where operators do not have the same unit weight. We follow similar reasoning to arrive at various conclusions which in many cases are the modified version of previous results. First we extend a few definitions for some basic operations.

Let h and w be integers such that $h \geq 0$ and $w > 0$, then h can be expressed as $h = i * w + r$, where i and r are some integers determined by $i = \lceil (h + 1)/ w - 1 \rceil$, and $r = h - i * w$ with $0 \leq r \leq w - 1$.

That is, we can write h as the sum of some multiple integer of a modulus w and a residue r. Furthermore, h can be represented in a binary form $b_i b_{i-1} \ldots b_1 b_0$ with $b_i = 1$ and all other bits set to zero in a group where the values of w and r are explicitly or implicitly understood, thus giving b_i a decimal value equal to $i * w + r = h$. If fact, what we defined by H[I] in previous section is a special case when $w = 1$ and $r = 0$. For a fixed modulus, two numbers are said to be in the same residue group if they have the same residue. In other words, a residue group of r consists of numbers from the set $r, r + w, r + 2w, .., r + i*w, ..$.

Now we can extend the definition of H[I] as follows.

Definition 2.15:

Define $H[I]_w$ as a set of binary numbers such that for every $i \in I$, there is a binary number, as described above, belonging to a proper residue group. We write $H[I]_w$ in the following format with one residue group in each row:

$$\begin{bmatrix} \overline{0}: & (\qquad) \\ \overline{1}: & (\qquad) \\ \cdot & \\ \cdot & \\ \overline{w-1}: & (\qquad) \end{bmatrix}_w$$

The modulus may be omitted explicitly in above expression when no confusion may occur from its omission. As an example, let I = [0, 1, 3, 4, 5, 6]. With w = 3, the 0-residue group contains (0, 3, 6), the 1-residue group contains (1, 4) and the number 5 belongs to the 2-residue group. Writing these numbers in binary, with the short-form convention for union of unequal numbers, we have

$$H[I]_3 = \begin{bmatrix} \overline{0}: & 1,\ 10,\ 100 \\ \overline{1}: & 1,\ 10 \\ \overline{2}: & 10 \end{bmatrix}_3 = \begin{bmatrix} \overline{0}: & 111 \\ \overline{1}: & 11 \\ \overline{2}: & 10 \end{bmatrix}.$$

The definition for discrete sum operation now becomes $a +_d b = \max(a,b) + w$, with the sum in the group where the larger of a and b belonged to. In terms of the actual binary operation, this means the larger of (a,b) will have its significant bit moved to the left by one digit in the same group to replace a and b. It is to be noted that when a = b,

the discrete sum operation still performs like an ordinary binary addition.

Definition 2.16:

Define $H_+[I]_w$ or $\lceil H[I] \rceil_{w+}$ as a discrete summation on all elements of $H[I]_w$.

In general, the smallest numbers are combined first until a single binary number is obtained. However, if two numbers are identical (in the same residue group), they may be combined first. This is easily proved as follows.

Let $a < b = c$. By regular procedure, we have

$H_+[a,b,c]_w = (a +_d b) +_d (c) = (b + w) +_d (c) = c + 2w.$

If instead b and c are combined first, then

$H_+[a,b,c]_w = (c + w) +_d (a) = c + 2w.$ Hence, the two procedures are equivalent.

We use the notation $H_+[I]_w$ on an integer set, and use $\lceil\ \rceil_{w+}$ on a set of binary numbers undergoing the operation. For example, if $H[I]_3 = \begin{bmatrix} \bar{0}: & 111 \\ \bar{1}: & 11 \\ \bar{2}: & 10 \end{bmatrix}$,

the discrete sum is

$$H_+[I]_3 = \begin{bmatrix} \bar{0}: & 111 \\ \bar{1}: & 11 \\ \bar{2}: & 10 \end{bmatrix}_{3+} = \begin{bmatrix} \bar{0}: & 110 \\ \bar{1}: & 10 + 10 \\ \bar{2}: & 10 \end{bmatrix}_{3+}$$

$$= \begin{bmatrix} \bar{0}: & 110 \\ \bar{1}: & 100 \\ \bar{2}: & 10 \end{bmatrix}_{3+} = \begin{bmatrix} \bar{0}: & 100 \\ \bar{1}: & 100 \\ \bar{2}: & 100 \end{bmatrix}_{3+}$$

$$= \begin{bmatrix} \bar{0}: & 0 \\ \bar{1}: & 1000 \\ \bar{2}: & 100 \end{bmatrix}_{3+} = \begin{bmatrix} \bar{0}: & 0 \\ \bar{1}: & 10000 \\ \bar{2}: & 0 \end{bmatrix}_{3+} = [\bar{1}: 10000]_3.$$

Which shows the sum has a (decimal) value of 4 * 3 + 1 = 13.

The significance of the operation described above is shown in the following lemma.

Lemma 2.5:

$H_+[I]_w$ produces a minimum discrete sum on the integer set I or its equivalence, $H[I]_w$.

[Proof]: Let $h_1, h_2, h_3, \ldots$ be the elements of $H[I]_w$, and without loss of generality assume $h_1 \leq h_2 \leq h_3 \leq \ldots$. By the definition of discrete sum, $a +_d b = \max(a,b) + w$, hence the first minimum discrete sum is obtained with $h_1 +_d h_2 = h_2 + w = h_{1,2}$. Now we consider two cases:

(a) If $h_3 \leq h_{1,2}$, then $h_{1,2} +_d h_3 = h_{1,2} + w = h_2 + 2w$ $= h_{1,2,3}$

(b) If $h_3 > h_{1,2}$, then $h_{1,2} +_d h_3 = h_3 + w = h_{1,2,3}$.

Clearly, if h_3 is combined first, then $h_{1,2,3} = h_3 + 2w$. Therefore, in either case, the discrete sum operation by Definition 2.16 gives a minimum sum.

By recursively applying above argument on all remaining elements of $H[I]_w$, it is easily concluded that $H_+[I]_w$ as defined yields the minimum discrete sum. (Q.E.D.)

The direct application of the discrete sum operation is to compute a minimum tree-height of an arithmetic expression without actually constructing the tree. This is discussed next.

Let an arithmetic expression E be of the form $E = \prod_i t_i$ or $E = \sum_i t_i$, where, as before, Π and Σ are for product and sum, respectively, then the minimum tree-height for E is $H[E] = H_+[h[t_i]]_w$, where $w = w_a$ is the operator weight for addition if $E = \Sigma\, t_i$, and $w = w_m$ is the operator weight for multiplication if $E = \Pi\, t_i$. For example, let

$$E = a + b + c + def + g + h$$

$$= t_1 + t_2 + t_3 + t_4 + t_5 + t_6$$

and $w_a = 2$, $w_m = 3$.

Except for t_4, all other terms are single variables, hence their subtree heights are 0, i.e., $h[t_1] = 0$, etc.

We now compute $H[t_4]$. Its component subtrees all have zero height, hence $H[t_4] = H_+[0,\ 0,\ 0]_3$

$$= \left\lceil \begin{matrix} \bar{0}:\ 1 + 1 + 1 \\ \bar{1}:\ 0 \\ \bar{2}:\ 0 \end{matrix} \right\rceil_{3+} = \left\lceil \begin{matrix} \bar{0}:\ 11 \\ \bar{1}:\ 0 \\ \bar{2}:\ 0 \end{matrix} \right\rceil_{3+}$$

$$= \lceil \bar{0}:\ 11 \rceil_{3+} = [\bar{0}:\ 100]_3.$$

Which means, in decimal value, $h[t_4] = 6$. Since in the computation of $H[E]$, the operator is addition, we have to express $H[t_4]$ in terms of the new modulus w_a. It is readily seen then, $H[t_4] = [\bar{0}:\ 1000]_2$. Therefore, with $w_a = 2$,

$$H[E] = \left\lceil \begin{matrix} \bar{0}:\ 1 + 1 + 1 + 1000 + 1 + 1 \\ \bar{1}:\ 0 \end{matrix} \right\rceil_{2+}$$

$$= \lceil \bar{0}:\ 101 + 1000 \rceil_{2+} = \lceil \bar{0}:\ 1000 + 1000 \rceil_{2+}$$

$$= [\ \bar{0}:\ 10000]_2.$$

So the minimum tree for E has a height of 4 * 2 + 0 = 8.

It is to be noted that the order in which the elements of $H[t_i]_w$ are combined corresponds to the order in which the terms of E are combined to achieve the minimum tree T[E].

Also it is clear that whenever two unequal elements (representing two subtrees of unequal heights) are combined by the discrete sum operation, a free node whose height equals to the larger element is created in connection with the smaller subtree. For example, if $h[t_1] = 4$, $h[t_2] = 5$ and the two subtrees combined by $w_m = 3$, then

$$H_+[4,5]_{w=3} = \begin{bmatrix} \bar{0}: & 0 \\ \bar{1}: & 10 \\ \bar{2}: & 10 \end{bmatrix}_{3+} = [\bar{2}: 100]_3$$

Hence $h[t_1 * t_2] = 8$ and a free node with height $[\bar{2}: 10]_3$ is created in connection with $T[t_1]$. The tree diagram in Fig. 2.12 illustrates this.

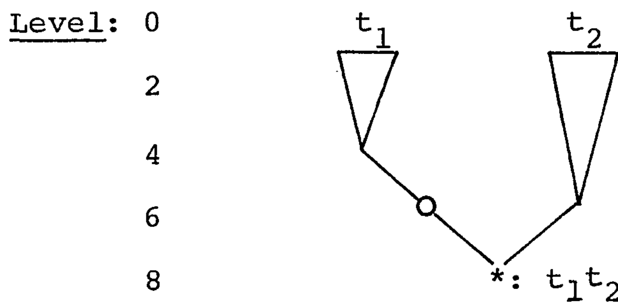


Fig. 2.12. Illustration of a free node.

2.4.2 Extension of Hole Function, Space Function and Tree-height Reduction

Now we consider the use of distribution to reduce a tree-height. In general, the principles involved in reducing tree height through distribution for the case of equal operator weights still apply, i.e., we find the hole function the space function for some subexpression and see if they can accomodate another factor or term. However, some explicit expressions will have to be modified.

First we revisit the definition of hole functions. For an arithmetic expression of the form $E = \prod_{i=1}^{p} a_i$ where a_i are single variables, equation (2.1) is repeated here:

$$f_h[E] = [2^{\lceil \log_2 p \rceil} - p]_b .$$

However, in this case all operations will take place only at some levels which are integer multiples of the multiplication operator weight, w_m. Hence it is meaningful to define free nodes and holes only at these levels. This leads to a modified expression for the hole function:

$$f_h[E] = [\ \overline{0}\colon\ [2^{\lceil \log_2 p \rceil} - p]_b\ ]_{w_m} . \qquad (2.5)$$

For arithmetic expressions of the form $E = \prod_i t_i$ or $E = \sum_i t_i$, equations (2.2) and (2.3) are still valid and can be used to find the hole functions if the H[I] function is replaced by $H[I]_w$. This is true because all the expressions for the holes and free nodes will have already been represented in the proper format as they are evaluated from each subtree.

While a hole can absorb another factor or term without increasing its own tree-height in the case of equal operator weight, the same may not hold here. We shall illustrate this point with an example.

Let $E = (a + b)cde = t_1 * t_2 * t_3 * t_4$ and $w_m = 3$, $w_a = 2$.

For t_1, $H[t_1] = \begin{bmatrix} \bar{0}: 1 + 1 \\ \bar{1}: 0 \end{bmatrix}_{2+}$

$= [\bar{0}: 10]_2 = [\bar{2}: 1]_3$,

and $h[t_2] = h[t_3] = h[t_4] = 0$.

Then ,

$$H[E] = \begin{bmatrix} \bar{0}: 1 + 1 + 1 \\ \bar{1}: 0 \\ \bar{2}: 1 \end{bmatrix}_{3+} = \begin{bmatrix} \bar{0}: 11 \\ \bar{1}: 0 \\ \bar{2}: 1 \end{bmatrix}_{3+} = \begin{bmatrix} \bar{0}: 10 \\ \bar{1}: 0 \\ \bar{2}: 10 \end{bmatrix}_{3+}$$

$= [\bar{2}: 100]_3$,

with 2 holes at $[\bar{2}: 1]_3$ and $[\bar{2}: 10]_3$.

Therefore, applying equation (2.3), we get

$f_h[E] = [\bar{2}: 10, 1]_3 = [\bar{2}: 11]_3 = 100100.$

The tree and holes are shown in Fig. 2.13.

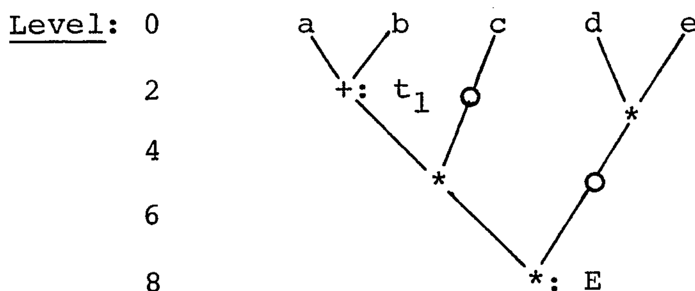


Fig. 2.13. An example of holes with unequal operator weights.

Since the highest hole is at level 5, it can accomodate another factor t_5 if $h[t_5] \leq 4$. However since $h[d * e] = 3$, if t_5 is attached such that $E_1 = t_1 * t_2 * t_3 * t_4 * t_5$, then the total tree height will be $h[E_1] = 9$, if $h[t_5] \leq 3$, and $h[E_1] = 10$ if $h[t_5] = 4$, while $h[E * t_5] = 11$ as can be seen from above tree diagram. Therefore in general, the resultant tree height depends on the distance (or difference) between the hole and the root of the subtree to which the hole is connected, and the height of the additional tree to be attached. But, even though a hole may not completely "contain" another factor, proper distribution nonetheless reduces the total tree height. That is, the possible reduction in tree-height through one application of hole-filling by distribution is now between 1 and w_m.

Next we examine the space function. Previously in Definition 2.14 we had for an expression

$$E = t * (\sum_i t_i) + t' = t * A + t'$$

its space function defined as

$$f_s[E] = f_s[A,t] = \sum \{ H[t_j], H[\beta] \},$$

where t_j is a term such that $T[t_j]$ contains a hole α with $h[\alpha] > h[t]$, and β is a free node in $T[A]$ such that $h[\beta] > h[t]$.

In either case, we assume $T[A]$ does not have holes to contain t. However, after distribution of t over A, a space is created for each t_j or β at a level which is higher by one multiplication weight and can take a term up to this level minus 1. That was why previously the space function

was defined at the same level of $h[\beta]$ (instead of the actual space level, $h[\beta] + 1$) to save a plus 1 and then minus 1 process. But now we shall modify the space function as follows:

$$f_s[E] = \Sigma\, H[\, h[t_j] + w_m,\ h[\beta] + w_m\,]_{w_a} \tag{2.6}$$

where E, t_j, and β are as defined in Definition 2.14 and w_m and w_a the operator weights as described above.

Again, similar to hole-filling, reduction of tree height through space-filling depends on the distance between the space and the root of the subtree to which the space is connected, and on the tree to be attached. Since now it is the addition operation whose height is to be saved, the possible reduction by each space-filling will be between 1 and w_a.

We give an example to illustrate space-filling.

Let E = (ab + c) * d + ef = A * d + t' and $w_a = 2$, $w_m = 3$. It is readily verified that A does not have holes to contain d, hence we consider space-filling.

$$H[A] = H_+[3,\ 0]_2 = \begin{bmatrix} \overline{0}: & 1 \\ \overline{1}: & 10 \end{bmatrix}_{2+} = [\overline{1}:\ 100]_2,$$

with two free nodes $\{\beta\}$ at $[\overline{1}:\ 11]_2$ or at levels 1 and 3. Therefore spaces are available at $1 + w_m = 4$ and $3 + w_m = 6$. Since h[ef] = 3, which is smaller than either of the spaces, we distribute d over A and let E^d = abd + cd + ef. It is then easily found that h[E] = 10 while $h[E^d] = 8$, as shown in Fig. 2.14, which also indicates the positions of spaces.

$E = (ab + c) * d + ef.$

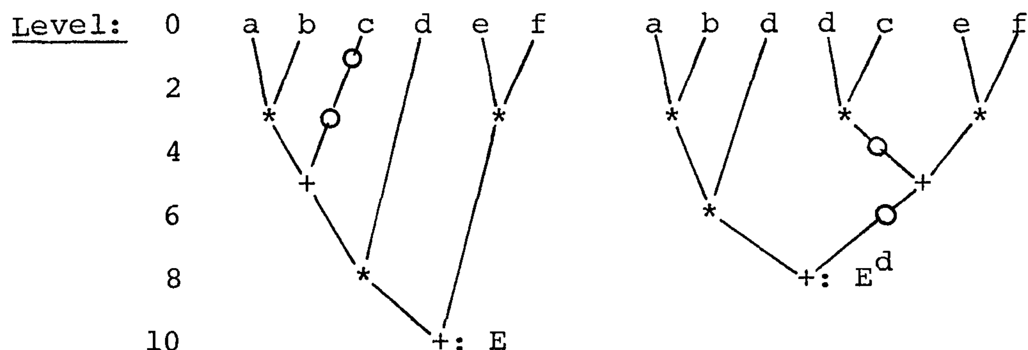


Fig. 2.14. An example of spaces with unequal operator weights.

In this example, because the highest space has a distance of more than w_a away from either of the subtrees attached, the resultant tree has its height reduced by $w_a=2$.

With hole and space functions revised, the distribution algorithm is entirely similar to that discussed in Section 2.3.3, hence will not be repeated. However we find that in this case the property expressed by Lemma 2.4 has to be modified . We show this in the following lemma.

Lemma 2.6:

Let $E_1 = \sum_{i=1}^{m} t_i$, $E_2 = \sum_{j=1}^{n} t'_j$ and

$E = (\sum_i t_i) * (\sum_j t'_j)$, $E^d = \sum_i \sum_j t_i t'_j$ as in Lemma 2.4, with $m,n \geq 2$. Then,

$h[E^d] \leq h[E]$ if $h[t_i t'_j] = h[t'_j]$ and $\lceil \log_2 m \rceil * w_a \leq w_m$.

[Proof]: By distribution, at best is when all t_i are absorbed by t'_j thus saving one multiplication height w_m. However, E^d has m times more terms than E has, thus it needs an additional height of $\lceil \log_2 m \rceil * w_a$ levels. If this height is

smaller than w_m, then $h[E^d] < h[E]$. (Q.E.D.)

We conclude this section by revising Theorem 2.1 to give the following corollary which is clear from above discussions.

Corollary 2.1:

Let $E = t * A + t'$ and $E^d = (t * \bar{A})^d + t'$, where $(t * \bar{A})^d$ represents the expression when T is distributed over A, and let $f_h[E]$, $f_s[E]$ be as defined by equations (2.2), (2.3), (2.5), and (2.6), and w_m and w_a the operator weights for multiplication and addition, respectively, then the following relations hold:

(a) $w_m \geq h[t * A] - h[(t * \bar{A})^d] \geq 1$ if $h[t] \leq mxb\{f_h[A]\}-1$.

(b) $w_a \geq h[E] - h[E^d] \geq 1$ if $h[t'] \leq mxb\{f_s[A,t]\}-1$.

2.5 Interstatement Parallelism Exploitation

Interstatement parallelism exploitation essentially covers two phases:

(a) If two statements are operationally independent of each other, then execute them simultaneously. Furthermore, if there exist some common suboperations, some of them can be performed only once.

(b) If two statements are not operationally independent from the way presented, modify the statements so that they become independent. Then execute as in part (a).

To define this operational dependency (or independency) more precisely, let s_i and s_j be two executable statements in a program, and s_i precedes s_j as presented in the program (we use $i < j$ to mean this). Define $I(s_i)$ as the set of

input variables contained in s_i, and $O(s_i)$ the output variable of s_i, and similarly for $I(s_j)$ and $O(s_j)$, then the following theorem is true:

Theorem 2.2:

Two statements s_i and s_j, where $i < j$ as presented, are operationally independent and can be executed simultaneously if and only if the following two conditions are met:

(1) $O(s_i) \notin I(s_j)$,

(2) $O(s_j) \notin I(s_i)$.

Theorem 2.2 is true because (1), if $O(s_i) \in I(s_j)$, then s_j has to be executed only after s_i is completed as s_j uses directly the result of s_i; and (2), if $O(s_j) \in I(s_i)$, some element of $I(s_i)$ will be updated later in the program by s_j, hence s_i must be executed before this happens to have a correct value for its input variable. However, by modifying the statements, it is possible to break the operational dependency between some statements to increase the parallelism.

In the following, we shall treat this problem separately with different types of statements, namely, assignment statements, IF statements, and DO statements.

(A) A block of assignment statements (BAS)

A BAS, by its name, contains only a sequence of assignment statements, screened out from the original program and preserves the general ordering sequence between the statements. If by an application of Theorem 2.2 some statements are found to be operationally dependent, two procedures can be taken:

(1) If $O(s_i) \in I(s_j)$, where $i < j$, replace $O(s_i)$ which appears in s_j by the RHS of s_i. Then s_i and s_j are operationally independent and some common suboperations can be shared by both s_i and s_j.

For example, if A = B * C * D

E = F * A

G = E + H

By repeated substitutions, we have

A = B * C * D

E = F * B * C * D

G = F * B * C * D * + H.

Then the modified statements can be computed in three steps of perations while the original block would take four.

(2) If $O(s_j) \in I(s_i)$, where $i < j$, replace $O(s_j)$ with a new variable name. Then s_i and s_j are operationally independent.

For example, assume two statements s_i and s_j as follows:

s_i A = B * C + D

.

.

s_j D = E + F

If we use a new output variable, say, D_1 for s_j, then clearly s_i and s_j are independent. Notice that in this case other occurrences of the variable D after s_j as an input variable should also be changed to D_1.

After a BAS is made independent for exploiting maximum interstatement parallelism, the techniques discussed in

previous sections are used to parse each statement for intrastatement parallelism exploitation. In order that results of common suboperations may be shared by more statements which contain them, parsing of s_j may have to depend on the way that s_i is parsed. This can be conveniently implemented by skipping over some suboperations in s_j which have already been encountered in s_i. For instance, in the first example above, the statements should be parsed as shown in Fig. 2.15.

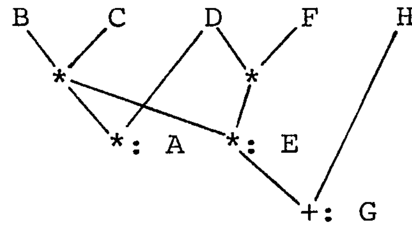


Fig. 2.15. A parse tree-complex of a BAS graph.

A few points are worth of note:

(1) A BAS parse tree-complex is a composite of interconnected parse trees. Therefore, a node can have more than one outgoing arc, and there is more than one terminal node. In order not to violate Definition 2.1, we shall call the resultant tree-complex as a BAS graph.

(2) A BAS graph shows all the interstatement and intrastatement parallelism. All nodes at a same level are independent and can be executed simultaneously.

(3) If a node has more than one outgoing arc, its processed result either has to be shared by more than one PE at the same time (if immediate successors are at same level),

or must be stored for later use (if immediate successors are at different levels).

(4) A BAS graph shows the optimal way that its statements may be executed in order to achieve the minimum computation time. However this may not actually be done. In general, the graph has to go through a stage of job-scheduling before it is executed by some PE's. This will be the central topic of the next chapter.

(B) IF statements

Given an IF statement of the form

IF (s) s_1, s_2, s_3

where s is an arithmetic or logic expression and s_1, s_2, and s_3 are branching instruction statements, speed-up in computation can be achieved by doing the following:

(1) Make s a separate assignment statement x.

(2) Apply the parsing technique of a BAS to x, s_1, s_2 and s_3 and possibly their follw-up statements, and execute them simultaneously.

(3) When the result of x is known, continue only with the correct branching path from s_1, s_2, or s_3 and discontinue the processing of the others.

For example, an IF statement

IF (B * B - 4 * A * C) 6, 8, 9

is converted to

x = B * B - 4 * A * C

IF (x) 6, 8, 9

and statements x, 6, 8, and 9 are all started to be executed.

Only the correct branch is continued after the value of x is obtained.

Above idea can be extended to include many IF statements and their associated assignment statements for more speedup. This will result in a binary decision tree and a larger BAS. However, the increased effort both in the preprocessing phase and the processing phase is quite enormous. It therefore requires a high occurrence rate of IF statements to justify this additional cost. This has been discussed by Davis [9] and shown by some simulation results in [17].

(C) DO statements

DO statements, by definition, contains a number of operations which are to be repeated in a similar or identical way. Hence this is the main area of interest where a high degree of interstatement parallelism may be achieved. We first define two types of DO statements.

Definition 2.17:

A DO statement is said to be recursive if it has the form $x_i = f(x_{i-1}, .., x_{i-m}, a_i)$ where m is any positive integer and a_i some vector of parameters. Otherwise the DO statement is nonrecursive and is called an array type.

A DO statement always involves a loop and sometimes the loops are nested. As is practically possible, we shall treat only a loop at a time, starting with the innermost loop if it is nested. Similar analyses can then be applied to the outer loops.

The principle in treating a DO loop is to separate the

loop into several completely independent DO loops and consider each one of them individually. To do this, precedence relations among the statements in the loop are examined. Because data can be used first and then updated for use in the next iteration, both conditions as considered in Theorem 2.2 have to be checked. This is more conveniently done with the aid of a graph in which each node corresponds to a statement in the DO loop, as described in the following procedures:

(1) Compare $I(s_j)$ with $O(s_i)$, where $i < j$. If two variables are identical with same or smaller index in $I(s_j)$, place an arc from node i to node j. Also compare $I(s_j)$ with $O(s_k)$, where $k > j$. If two variables are identical with the higher index in $O(s_k)$, place an arc from node k to node j. This is done for all statements in the loop.

(2) If some subgraphs are completely disconnected at the completion of step (1), treat each subgraph as a new DO loop and do the next step.

(3) Within each new DO loop:

(a) If statements are of array type, one statement can be generated for each index value, resulting in many parallel statements.

For example, the following loop

```
    DO  1   I = 1, 3, 1
  1 A(I) = A(I+1) + C(I) * D(I)
```

will generate three parallel statements:

A(1) = A(0) + B(1)

A(2) = A(0) + B(1) + B(2)

.

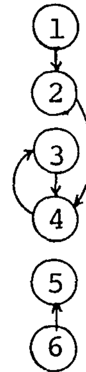
.

A(5) = A(0) +B(1) + B(2) +B(3) +B(4) + B(5).

It should be noted that in this case, A(1) through A(4) are actually included in A(5). Hence it suffices to parse only A(5), and in the process the appropriate intermediate results are obtained for A(1) through A(4).

An example of decomposing a DO loop is shown next to illustrate above procedures.

```
    DO  6   I = 1, 3, 1
1   T(I) = G(I) + M
2   G(I) = T(I) + D(I)
3   E(I) = F(I-1) + B(I)
4   F(I) = E(I) + G(I)
5   H(I) = A(I-1) + H(I-1)
6   A(I) = C(I) + N
```



In the accompanying graph, we observe:

1 ⟶ 2 because of T(I) in both statements,

2 ⟶ 4 because of G(I) in both statements,

3 ⟶ 4 because of E(I) in both statements,

4 ⟶ 3 because of F(I-1) in s_3 and F(I) in s_4,

6 ⟶ 5 because of A(I-1) in s_5 and A(I) in s_6.

Since there are two completely separated subgraphs, two new DO loops are created:

```
    DO  4   I = 1, 3, 1
 1  T(I) = G(I) + M
 2  G(I) = T(I) + D(I)
 3  E(I) = F(I-1) + B(I)
 4  F(I) = E(I) + G(I)

    DO  6   I = 1, 3, 1
 5  H(I) = A(I-1) + H(I-1)
 6  A(I) = C(I) + N.
```

We further decompose each new DO loop. For the first subgraph:

```
    DO  1   I = 1, 3, 1                (Comments)
 1  T(I) = G(I) + M              Generate 3 array equations
                                 T(1), T(2), T(3).
    DO  2   I = 1, 3, 1
 2  G(I) = T(I) + D(I)           Generate 3 array equations
                                 G(1), G(2), G(3).
    DO  4   I = 1, 3, 1
 3  E(I) = F(I-1) + B(I)         Generate 6 recursive equations
                                 E(1), F(1), E(2), F(2),
 4  F(I) = E(I) + G(I)           E(3), F(3).
```

The six array equations can be obtained in a straightforward way. The recursive equations are generated through repeated substitution:

```
    E(1) = F(0) + B(1)
    F(1) = F(0) + B(1) + G(1)
    E(2) = F(0) + B(1) + G(1) + B(2)
    F(2) = F(0) + B(1) + G(1) + B(2)
    E(3) = F(0) + B(1) + G(1) + B(2) + G(2) + B(3)
    F(3) = F(0) + B(1) + G(1) + B(2) + G(2) + B(3) + G(3).
```

Fig. 2.16 shows the parsing of above statements using the techniques discussed in previous sections.

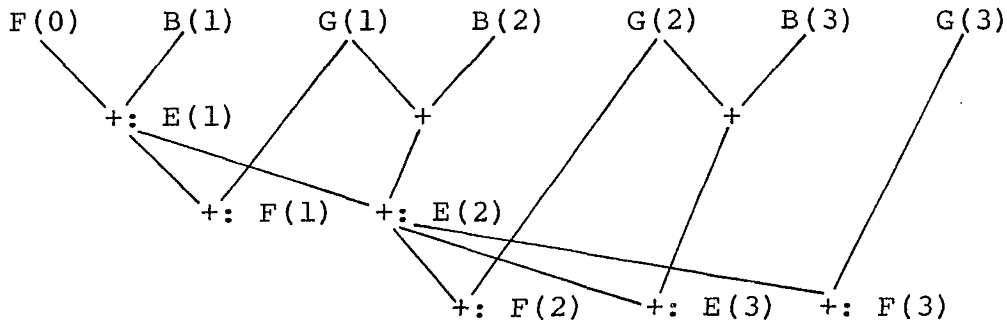


Fig. 2.16. A parse-tree complex for recursive equations.

For the second subgraph, three array equations and three recursive equations can be generated in a similar way for statements 6 and 5, respectively. We also observe that all three array equations of s_6 can be executed simultaneously and before s_5.

2.6 Summary

In this chapter we have discussed the detection and exploitation of parallelism in a computation-oriented program. Since the basic statements in this type of program are arithmetic expressions, we started by looking into the way that an arithmetic expression is parsed according to its syntactic structure. This leads to a syntactic tree showing the suboperations which can be processed simultaneously.

The concept of holes and spaces was then introduced, and use of the distribution law to fill up some free nodes and reduce a tree-height was discussed. By using a series of specially defined binary operations, useful information concerning tree height and free nodes can be computed

without having to construct a tree graphically. A distribution algorithm was then given to sum up the operations.

In Section 2.4 we further extended the discussion to the general case where operators do not have the same weights. Binary operations parallel to those defined in earlier sections were used to generalize the analysis. We found that with some revisions in the derivation of holes and spaces, the distribution algorithm could be readily applied.

Finally, interstatement parallelism was studied. Specific procedures for breaking the operational dependency between statements were given for assignment statements, IF statements, and DO statements. The result of these procedures is a tree-complex explicitly showing all combined intrastatement and interstatement parallelism.

This graph will be used as the basis for job scheduling before the program is executed. This leads us to the subject of scheduling which is to be discussed in the next chapter.

3. JOB GRAPH AND TASK SCHEDULING

3.1 Introduction and Job Graph

In Chapter 2 we saw that as a result of parallelism detection, a tree-complex showing all permissible operations was created. In order to show the height of a tree in an explicit way, we let the level distance between a node and its predecessors equal to the weight of the associated operation. It is obvious, however, that we can also let each node carry the information on its operator weight and let an arc between nodes indicate only the precedence relation. If this is done, the graph becomes a weighted-node directed graph. This graph can be used for scheduling all its node operations on a system of parallel processors for execution, so that the process represented by the graph is completed in a minimum time.

In above discussion, each node corresponds to a basic arithmetic expression, and maximun parallelism is theoretically obtainable if a sufficient number of processors are available. However, there are also occasions when it is more convenient or necessary to schedule the tasks on a higher unit basis. For example, a whole statement can be considered as a unit task if its internal suboperations are not to be executed by more than one PE. Then a node weight is the total combined time of all suboperations in the statement, and Theorem 2.2 can be used to determine the precedence relations among all nodes (i.e., an arc is placed from n_i to n_j if $O(s_i) \in I(s_j)$). Or on an even higher

basis, a whole program can be considered as a unit task. Certainly, a lower level scheduling can bring out more possible parallelism existing down to that level, but then the scheduling becomes complex and tedious. Hence, depending on the practicality of the application, one has to decide what level of scheduling he is concerned with, and determine the task size accordingly. In any case, since the principle of scheduling can be applied at all levels, for the purpose of analysis we assume a well-defined weighted-node directed graph has been obtained, and the graph is acyclic. That is, if any loop originally exists, the whole loop is treated as a single node with its node weight properly summed. Such a graph, when used for scheduling purpose, is called a job graph. And we shall use the terms "node weight" and "task time" interchangeably.

In a job graph, since the "length" of an arc does not bear any useful information, a node may sometimes be moved either closer to its predecessor or to its successor. If we place nodes only at some equally-spaced levels for convenient representation, then when all nodes are placed closest to their predecessors, we denote the graph by G_E (for earliest precedence partition). On the other hand, if all nodes are placed closest to their successors, we call it G_L (for latest precedence partition), or G_R (for relaxed graph). Since both forms are useful in the study of scheduling, we shall describe two simple systematic procedures for obtaining the two forms of a partitioned

graph, after a definition is first given.

Definition 3.1:

Let G be a job graph with N nodes. Define a connectivity matrix C(G) as an N x N matrix whose $c_{i,j} = 1$ if there is a directed arc from n_i to n_j, where $i \neq j$; otherwise $c_{i,j} = 0$. $i,j = 1, 2, .., N$.

To obtain G_E, do the following E-partition:

(1) In C(G), starting from column 1, cross out the columns which have all zero entries. The nodes which correspond to the indices of the crossed-out columns are put at the first level in G_E.

(2) Cross out equal number of rows which have the same indices in C(G).

(3) Repeat steps (1) and (2) with the remaining part of C(G), and place the collected nodes at the next higher level (numerically). This is done repeatedly until C(G) = 0.

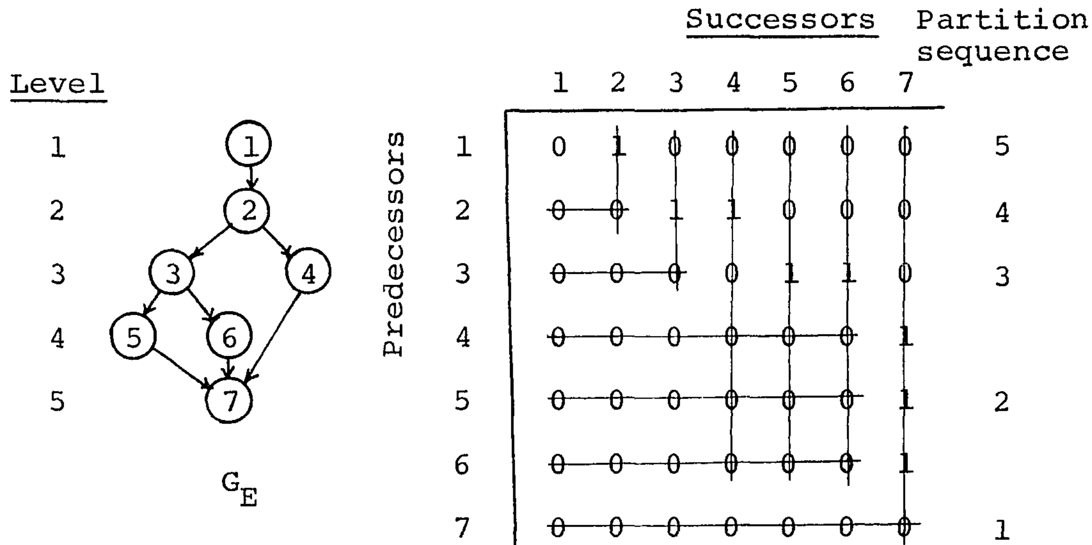
A similar procedure for L-partition is used to obtain G_L as follows:

(1) In C(G), starting from row N, cross out the rows which have all zero entries. The corresponding nodes are placed at the bottom level of G_L.

(2) Cross out equal number of columns which have the same indices in C(G).

(3) Repeat steps (1) and (2) with the remaining part of C(G) and place the collected nodes at the next lower level (numerically). Repeat until C(G) = 0.

An example on job graph, connectivity matrix, and partition procedure is shown in Fig. 3.1. Since the given graph is by itself in the earliest-partition form, we illustrate the second procedure to obtain G_L. We also use L_i to represent the set of nodes collected for level i in G_L.



(node weights not shown)

Collection of nodes at same level:

{(7), (6,5,4), (3), (2), (1)}

Fig. 3.1. Job graph and precedence partition.

As can be seen clearly from the given graph G_E, if node 4 is pushed down by one level, the graph is in G_L form. This is indicated by the collection of nodes in $L_4 = \{4,5,6\}$ from the partition procedure.

The direct meaning of above partition is that, if all node weights are same, all nodes at same level may be executed simultaneously. Also, G_E indicates how early a node may be executed, and G_L indicates the latest time by

which a node must be executed so that it does not unduly hold up further operations and increase the total processing time. For unequal node weights, a consistently partitioned graph makes it convenient for a scheduling algorithm to be applied, as we shall see in the next section.

3.2 Optimal Non-preemptive Scheduling

3.2.1 Notation and Bounds

By non-preemptive scheduling we mean that once a task is assigned to a PE and starts to be executed, it should be continued on that PE until the task is completed. Since we are primarily concerned with scheduling at a level not higher than a single program, a task is generally well-defined and not to be interrupted during its normal execution. Therefore, this is the only type of scheduling we shall be dealing with. We also assume that all participating PE's have same capability and thus can execute any task assigned to it equally well.

The study of scheduling involves the following questions:

(1) What are the upper and lower bounds on the number of PE's to be used so that the job (as represented by its job graph) can be completed in the least amount of time?

(2) Given a number of PE's less than the lower bound, what is the minimum time in which the job can be completed?

(3) What scheduling algorithms can one use?

These questions will be answered after some more definitions are given.

Definition 3.2:

Let w_i be the node weight of a node n_i, define the longest forward path length, D_i, of n_i as the largest sum of all node weights between n_i and the initial node among all possible paths.

That is, $D_i = w_i + \max \{ D_j \mid n_j < n_i \}$. (3.1)

Similarly, define the longest backward path length, R_i, of n_i as the largest sum of node weights between n_i and the terminal node among all possible paths:

$$R_i = w_i + \max \{ R_j \mid n_j > n_i \}. \quad (3.2)$$

Definition 3.3:

The critical-path length, C_q, of G having q levels is defined as $C_q = \max \{ D_i \mid n_i \in G \}$

$$= \max \{ R_i \mid n_i \in G \}.$$

The chain of arcs which yields C_q is naturally the critical path. It is clear from above definition, C_q is the absolute minimum time length in which a job graph can be processed. This is achieved if a sufficient number of PE's are available. It is interesting to find an upper bound on the number of PE's with which this C_q can be achieved. An upper bound based on the largest number of nodes at a level from E- or L-precedence partition was given by Ramamoorthy, et al. [28]:

$$m_u = \min \{ \max |L_i|, \max |E_i| \}, \ 1 \leq i \leq q \quad (3.3)$$

where L_i and E_i are sets of nodes at level i in G_L and G_E, respectively.

Since G_E or G_L may be used, clearly the minimum of the

two determines the upper bound.

The upper bound as expressed by equation (3.3) is readily applicable if all nodes have same unit weight. For unequal node weights, task times have to be extended accordingly (i.e., a node with $w_i = k$ becomes k nodes of unit weights and these nodes are extended over k levels). A similar but somewhat compact expression was given by Fernandez and Bussell [10], using the concept of load density function.

A sharp lower bound on the number of PE's required to process the job in C_q time was given by Kraska [15]:

$$m_L = \left\lceil \max\left\{\frac{W_i}{F_i}\right\} \right\rceil, \text{ for } 1 \leq i \leq q \qquad (3.4)$$

where $W_i = \sum_{j=1}^{i'} w_j$ is the sum of node weights for all nodes n_j existing from level 1 up to (including) level i, and $F_i = \max \{ D_j \mid n_j \text{ at level } i \}$ is the largest possible forward path length found on nodes at level i of G_L.

Equation (3.4) can be used first to determine a trial number of PE's which have to be used if a process time of C_q is desired. However, if by some optimal scheduling algorithm (which will be discussed in the next subsection), it is found that such critical-path time cannot be achieved with only m_L PE's, then $(m_L + 1)$ PE's are tried for scheduling.

The next question is a very practical one: if only m PE's, where $m \leq m_L$, are used, what is the shortest possible time to process the job graph? We present a new expression for answer by the following theorem:

Theorem 3.1:

With m PE's, where $m \leq m_L$, the lower bound on processing time to complete G is given by:

$$t_m = C_q + \left\lceil \max\left\{ \frac{W_i}{m} - F_i \right\} \right\rceil , \text{ for } 1 \leq i \leq q, \qquad (3.5)$$

where W_i and F_i are defined as in equation (3.4).

[Proof]: Clearly between any levels of nodes, the processing time is minimum if all tasks can be shared evenly by all m PE's. If this time, W_i / m, exceeds the largest forward path length, F_i, up to the level i, the extra time ($W_i/m - F_i$) has to be added to the minimum possible time of C_q. Hence the result in equation (3.5). (Q.E.D.)

Other expressions for lower bounds on processing time are also found in [10], [28]. However, Theorem 3.1 provides a direct and more convenient way to compute t_m.

Notice that in computing all the bounds discussed above, one actually has to go through all levels in the job graph for similar computations. The value of W_i is accumulative, hence it is more convenient to start systematically from the top level down, until all levels are examined. One should also keep in mind that these values represent only the bounds. The actual time obtainable and the number of PE's required will have to be found or verified through some scheduling process, which will be discussed next.

3.2.3 Optimal Scheduling Algorithms

In this section we present two major optimal scheduling algorithms based on the works of Hu [13] and Kraska [15]. The algorithms are modified to make use of the various

definitions given earlier and to emphasize the principles rather than just the procedural details.

Hu's algorithm is optimal only if it is applied to a tree graph with equal node weights. However, despite its seemingly restricted application, the algorithm is fundamentally important because it is a simple algorithm that one may intuitively like to use, whether or not it produces an optimal schedule. In fact, its application can be extended to a general job graph as a suboptimal scheduling technique with surprising effectiveness. We shall see this in the next subsection.

Kraska's algorithm is basically an iterative procedure to determine what sequence of assignment produces an optimal result. The complexity and operation time required are therefore considerably higher than the previous one. Nonetheless, it provides a systematic approach for one to obtain an optimal schedule on a general job graph. Another optimal scheduling algorithm, given by Ramamoorthy and others [28], basically involves the same principles although it starts the scheduling from the top end (initial terminals) of the graph in a forward way. Their algorithm does require the extension of a task time of n into n unit times and considers the "state" of a scheduling process at every time unit. Since this algorithm has no direct bearing on what we are going to discuss, it will not be considered in detail here. Other discussions on minimal-time scheduling are also found in [30].

In the following we describe the two algorithms. We shall assume that a job graph has q levels, and m PE's are used.

(A) An optimal scheduling algorithm for a job tree with unit node weights.

Algorithm 3.1:

Initially let $i = 0$, then $i = 1, 2, ..$ At any instant i, proceed as follows.

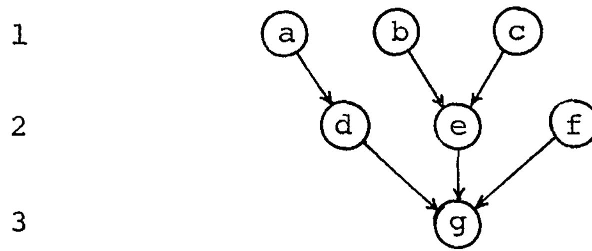
(1) Find the set of candidate nodes, D, whose predecessors have been assigned before. If $D = \emptyset$, stop, otherwise go to step (2).

(2) If $|D| \leq m$, assign all nodes $\{n_i\} \in D$ to m PE's in an arbitrary order. (However, if possible, a node is assigned to the same PE where one of its immediate predecessors has been assigned to minimize inter-PE communication.) If $|D| > m$, assign a subset of m nodes,

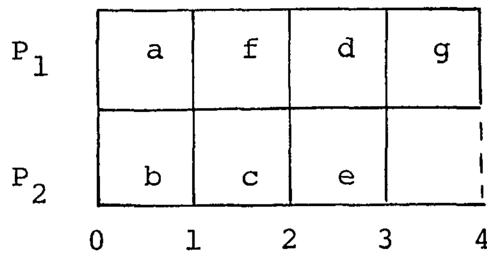
$B = \{n_i, n_{i+1}, .., n_{i+m-1}\} \subset D$ to m PE's in a similar or arbitrary order. The set B is collected in a way such that if $\{n_i, n_j\} \in D$, but $n_i \in B$, $n_j \in B$, then $n_i \in L_a$, $n_j \in L_b$ where L_a and L_b are set of nodes at levels a and b, respectively, in G_L and $a \leq b$.

(3) Let $i = i + 1$ and go to step (1).

As an example, consider the tree graph in Fig. 3.2 (a), in which all the task times are unity. The tree graph is to be assigned on two PE's.



(a) A job tree with equal task times.



(b) The scheduling diagram.

Fig. 3.2. An optimal scheduling of a job tree on two PE's.

At time instant 0, the candidate nodes are $D_0 = \{a, b, c, f\}$. But since $\{a, b, c\} \in L_1$, $\{f\} \in L_2$, by Algorithm 3.1, assign any two tasks in L_1 to the two PE's. Let $B_0 = \{a, b\}$, then $B_1 = \{c, f\}$. Next, $D_2 = \{d, e\}$ and finally $D_3 = \{g\}$. The resulting scheduling diagram , as shown in Fig. 3.2 (b) is also called a Gantt chart. Clearly the job needs 4 units of time to complete.

(B) A backward optimal scheduling algorithm for a general job graph.

This algorithm is based on a reduced graph G_L, and the procedure starts from the bottom level of G_L.

In the following discussion, let $T_{i(k)}$ be the accumulated task times (i.e., nodes weights) already assigned to the

k^{th} PE , P_k, at time instant t_i where $k = 1, 2, .., m$. P_k is said to be availabe for scheduling at a (reversed) time instant t_i if $T_{i(k)} \leq t_i$. Thus, a stage of scheduling begins at some t_i when at least one PE becomes available.

Algorithm 3.2:

Initially let $i = 1$, $G_1 = G_L$, $t_i = t_1 = 0$.

For stage $i = 1, 2, ...,$ the following steps are performed:

(1) At t_i find the set of terminal nodes, S, whose successors have been assigned, and find those terminal nodes $Q \subseteq S$ whose successors have been completed by the time t_i. Then Q is the set of nodes ready to be scheduled at t_i. If $Q = \emptyset$, stop, otherwise go to step (2).

(2) For every node $n_i \in S$, find its longest forward path length D_i, as defined by equation (3.1). Let

$$M = \max \{ D_j \mid n_j \in S \},$$

and n_i be the node such that $D_i = \max \{ D_j \mid n_j \in Q \}$.

(3) If n_i is on the critical path, go to step (4) directly, otherwise compute the possible increase in critical-path length if n_i is assigned by the following equation

$$d_i = t_i + w_i + D_j - C_q,$$

where $D_j = \max \{ D_f \mid n_f < n_i \}$ is the largest forward path length among the remaining nodes in S which become the predecessors of n_i (in operation, not due to actual functional precedence) if n_i is assigned. Go to step (5) next.

(4) Assign n_i to a PE, say P_k, whose $T_{i(k)} \leq t_i$, then let $T'_{i(k)} = t_i + w_i$ be the task time after assignment.

More than one node may be assigned in a similar way if it is possible, that is, if $|n_i| \geq |P_k| > 1$ and all $T_{i(k)} \leq t_i$. Let $Q' \subseteq Q$ be the set of nodes assigned in stage i, then

$$G_{i+1} = G_i - Q'.$$

Let $i = i + 1$ and go to step (1).

(5) Two situations are possible:

(a) If $d_i > 0$ and $D_i < M$, store d_i for later use. Let t_{i+1} be the time instant when next stage of scheduling can begin, then P_k is kept idle for a period of $\Delta_{i(k)} = t_{i+1} - t_i$. Let $i = i + 1$, include n_i in G_{i+1} and go to step (1).

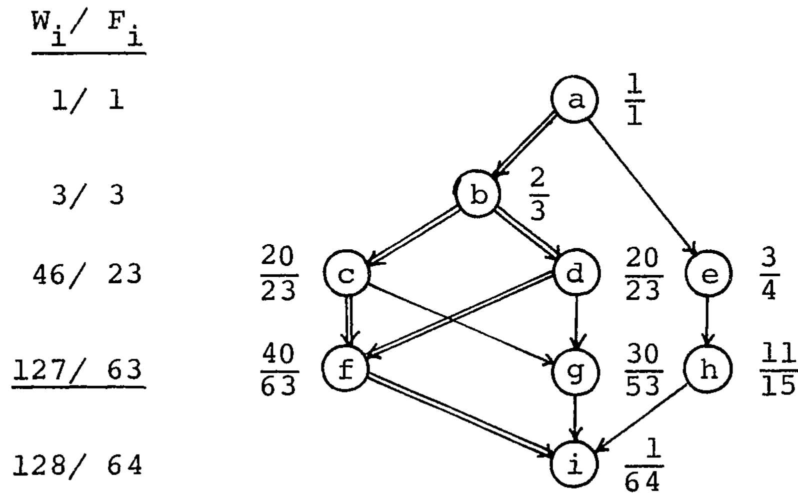
(b) If $d_i > 0$ and $D_i = M$, check and see if the node n_i has previously been examined for scheduling but postponed (i.e., see if step (5a) was previously performed on n_i in some stage $i' < i$). Let j be the stage such that for n_i,

$$d_j = \min \{ d_{i'} \} \quad \text{for all } i' < i,$$

return the scheduling process to stage j (i.e., let $i = j$), with G_j restored to what it was, and go to step (4) to assign the node n_i. If n_i has not been encountered before, go directly to step (4) next.

To illustrate Algorithm 3.2, consider the job graph in Fig. 3.3. The graph is already in the latest-partition form, and is to be scheduled on a system of two PE's. All the task times are shown as the upper numbers adjacent to each node.

As a practical approach, we first calculate the longest forward path length, D_i, associated with each node n_i.



Numbers adjacent to n_i are $\frac{W_i}{D_i}$

Fig. 3.3. Preparation of a job graph for backward optimal scheduling.

This is conveniently done downward from the highest level. The D_i are shown as the bottom numbers adjacent to a node. The longest path length for the terminal node certainly becomes the critical-path length C_q. In this example, $C_q = 64$. We also compile a column of W_i/ F_i for each level i. This is used to determine m_L, the least number of PE's for processing G in a time equal to the critical-path length, C_q. In this case,

$$m_L = \left\lceil \max \left\{ \frac{W_i}{F_i} \right\} \right\rceil = \lceil 127/ 63 \rceil = 3.$$

Hence it is not possible to process G in 64 units of time with only two PE's. However, we shall see what minimum processing time it takes for two PE's to do the job. For convenience, we also mark the arcs along the critical

path with broad arrows to distinguish them from others.

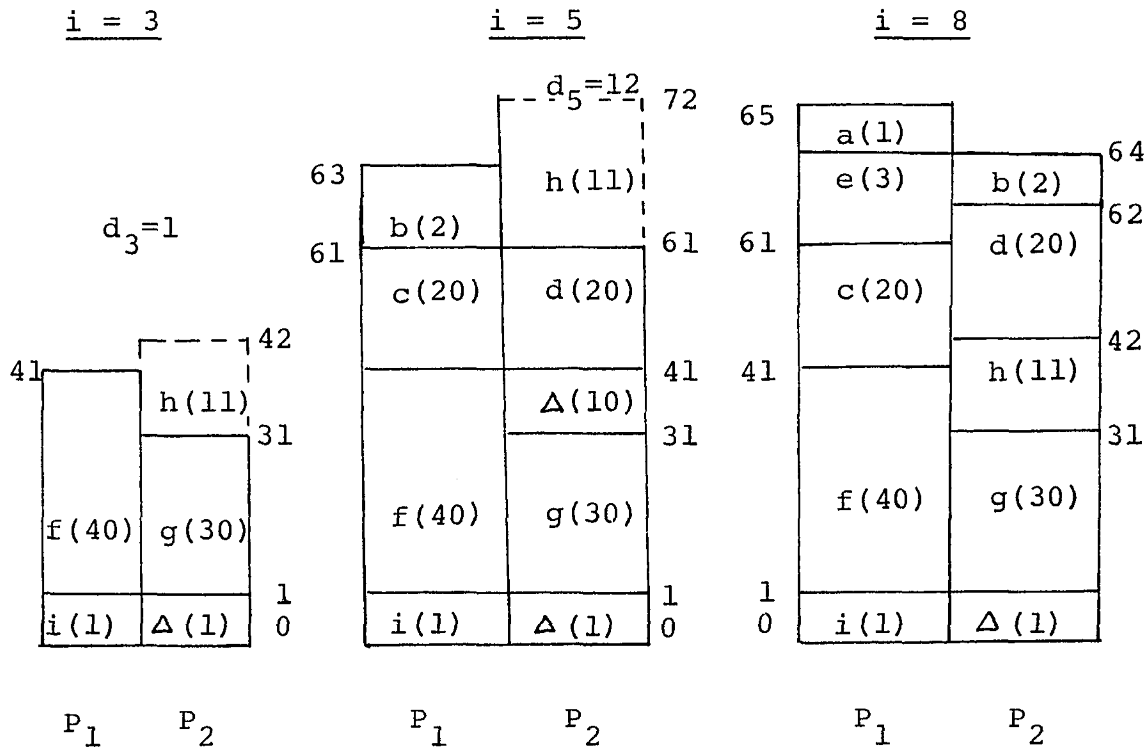
We now apply Algorithm 3.2. Since node i is the only terminal node, it is first assigned. Then nodes f, g, and h are included in the terminal set for the next stage. Since node f and g have higher forward path lengths than node h, f and g are assigned next. It is obvious that up to this point, no change in the critical-path length has been caused.

At the next instant, $t_3 = 31$, and $Q = \{h\}$. However, since $D_h = 15 < 23 = M$ (see Fig. 3.3), and
$d_3 = t_3 + w_h + D_d - C_q = 31 + 11 + 23 - 64 = 1$,
node h is not assigned yet, and P_2 is let idle till the next stage of scheduling is ready, i.e., when f is completed at $t_4 = 41$. So there is an idling time of $\Delta = 41 - 31 = 10$ for P_2.

The scheduling process continues until when i = 5 and node h is again coming up for assignment. At this time, $D_h = 15 = M$, and $d_5 = t_5 + w_h + D_e - C_q = 61 + 11 + 4 - 64 = 12$. Since $d_5 > d_3$ and $D_h = M$, we do not further postpone h but go back to stage i = 3, when h was last encountered, to schedule h, and a new schedule is built after it.

The scheduling process is illustrated in Fig. 3.4, with the final schedule done when i = 8. The schedule diagrams are shown with a reversed time axis going upward. Of course, in actual execution, the tasks are processed downward from the top. It is interesting to note that while it takes 3 PE's to process the job in its critical time, two PE's alone

can do it with only one extra unit of time.



Notes: Numbers within parentheses are node weights or idle times. Numbers along vertical axes indicate when a new scheduling stage is started or finished. Time scales are not in proportion.

Fig. 3.4. Scheduling of the graph in Fig. 3.3.

3.3 Suboptimal Heuristic Scheduling Algorithm

In view of the complexity and increased operations in an optimal scheduling algorithm universally applicable, it is desirable to consider some simpler heristic approaches and examine their effectiveness. Perhaps it is worthwhile to point out that in processing a job graph of equal unit

times, PE's can always be kept busy executing a series of tasks (if they are ready) according to an optimal schedule, but it is not so in a general job graph. However, in order to simplify the decision making,heuristic scheduling algorithms tend to allow a PE to stay idle only when there are simply no successor tasks ready for processing. This fact then sometimes precludes a PE's opportunity to process a more urgent task coming up in a short while, and prolongs the total job-processing time. As a result, a schedule produced from a simple heuristic approach will depend on the job graph: sometimes it may be optimal, other times not. Because of the random nature of a general graph in node weights and precedence relations, it is more practical to evaluate the effectiveness of a heuristic scheduling algorithm through simulation and experiment on test programs. Reports by Kaufman [14] and Graham [11] have indicated that the following heuristic scheduling algorithms generally produce schedules which are within 15% of the optimal schedules. Ramamoorthy's result [28] is even more promising: in ten tests out of eleven, the heuristic algorithms have produced optimal schedules.

We now give two heuristic scheduling algorithms.

Algorithm A (by latest partition):

At any time if a PE becomes free, assign to it a task in a way such that if both n_i and n_j are candidate nodes, but $n_i \in L_a$, and $n_j \in L_b$, with $a \leq b$, then n_i is assigned next.

Algorithm B (by longest backward path length):

At any time if a PE becomes free, assign to it a task in a way such that if n_i and n_j are candidates, but $R_i \geq R_j$, where R_i is the longest backward path length for n_i as defined in Definition 3.2, then n_i is assigned next.

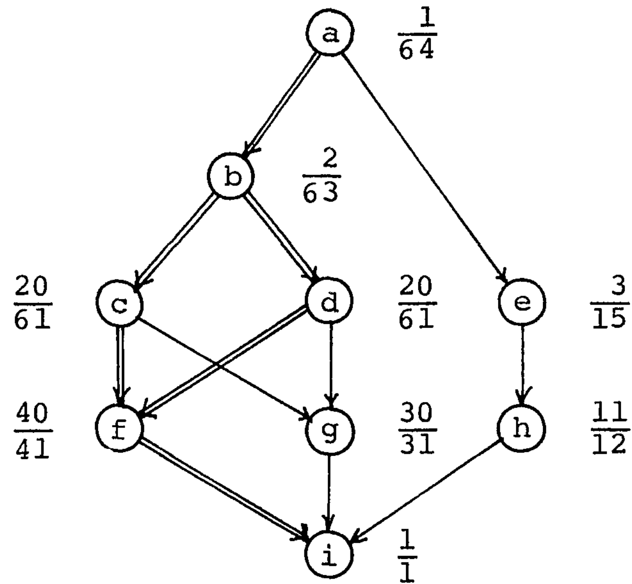
Algorithm A is actually just a direct copy of the optimal scheduling, Algorithm 3.1, discussed before, hence the node weights are ignored.

Algorithm B is also an extension of the same algorithm, with the partition level of a node essentially redefined by its longest backward path value. This algorithm is the more precise translation of Hu's algorithm into the case of unequal node weights.

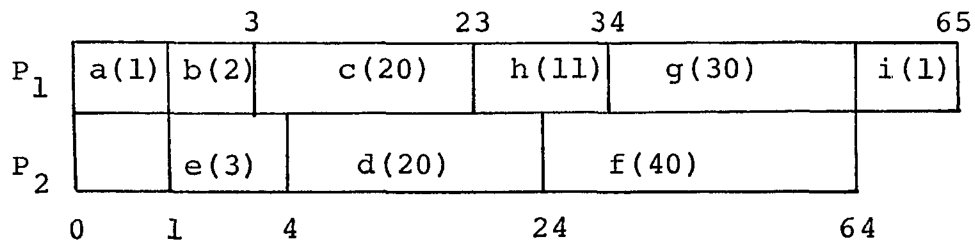
The rationale in these heuristics is clear. In Algorithm A, if a task belongs to a later partition, it is delayed to give way to another task whose latest-partition is due. In Algorithm B, a task with longer backward path is assigned first simply because it takes a longer time to reach the terminal node, hence should be taken care of first.

As an example, let us apply above heuristics to the job graph of Fig. 3.3 previously used for optimal scheduling. The graph is repeated in Fig. 3.5 (a) to show the longest backward path lengths associated with each node. In this case, both heuristics produce the same schedule as shown in Fig. 3.5 (b) when two PE's are used.

A quick comparison of the schedules in Fig. 3.4 and Fig. 3.5 (b) reveals that they are practically identical



(a) The job graph. The numbers adjacent to n_i are $\frac{w_i}{R_i}$.



(b) The Gantt chart.

Fig. 3.5. An example of heuristic scheduling.

(some irrelevant differences exist due to arbitrary choosing of equivalent tasks or PE's). However, the much less effort required in exercising the heuristic scheduling algorithms and the fast result obtainable certainly are quite appreciable.

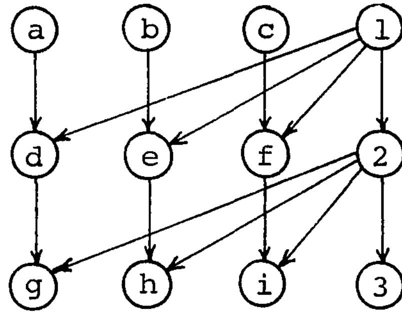
Graham [11] has given an upper bound on the effectiveness of the heuristic Algorithm B by the following ratio:

$$\frac{T}{T_o} \leq \frac{2m}{m + 1} = 2 - \frac{2}{m + 1}$$

where T is the processing time resulted from the longest path heuristic scheduling, and T_o is the optimal processing time, both with m PE's. To show that above bound is achievable, we give a worst-case example as shown in Fig. 3.6, where all tasks have unit weights, and three PE's are used.

The upper bound given above is quite prohibitive. However, as supported by other favorable experimental results mentioned earlier, application of heuristic algorithms with discretion should be valuable. One can always calculate the lower bound on processing time by equation (3.5), and compare it with the result of a heuristic scheduling and make a fair judgement on the effectiveness of the heuristic approach.

We also note from above example that additional information may be helpful to improve the results of a heuristic approach. For example, if the "sum of immediate successors' node weights" is used to break a tie among a group of candidate nodes, then above heuristic schedule also becomes an optimal one. Other decision factors are also possible.



(a) A job graph with unit node weights.

	0–1	1–2	2–3	3–4
P_1	1	2	3	i
P_2	b	a	e	h
P_3	c	f	d	j

(b) An optimal schedule.

	0–1	1–2	2–3	3–4	4–5	5–6
P_1	a	1	d	2	g	3
P_2	b		e		h	
P_3	c		f		i	

(c) A worst-case heuristic schedule.

Fig. 3.6. An example of worst-case heuristic scheduling.

However, adding too much to a heuristic approach may increase its complexity to such an extent that it is no longer a simple (but effective) algorithm.

3.4 Operational Considerations: Performance vs. Cost

In this section we deal with some questions which arise from optimal parsing and scheduling but have been neglected thus far. We shall give some answers and comments here.

It is obvious that up to now our only criterion for judging "optimality" of an algorithm or process is that a process be completed in the least possible time. The criterion certainly has its theoretic importance, besides the fact that very often time is the most important factor for considering the use of a parallel processing approach. However, from a more realistic point of view, we should also consider the additional cost in order to obtain this optimal performance. Therefore, a cost and performance trade-off is evaluated. For instance, in the example used to illustrate optimal scheduling (Figs. 3.3 and 3.4) we found that with two PE's the job graph can be processed in 65 units of time while it takes 64 units of time to complete if three PE's are used. This corresponds to an improvement of $1/65 \approx 1.5\%$ in processing time with a 50% increase in the number of PE's required. This certainly does not look like an investment one would like to make. Of course, here we have not really shown a complete picture of all the elements involved, but nonetheless, it serves to point out that some measurement of performance versus cost is needed.

We shall analyze this aspect.

Let G be a job graph, with other quantities defined as follows:

$W = \sum_i n_i \mid n_i \in G$ is the total node weights, and T_m is the time required to process all nodes in G using m PE's as determined from some (optimal) schedule.

If only one PE is used to process G, apparently it takes an amount of time equal to the sum of all tasks, i.e., $T_1 = W$. Therefore, a speed-up factor,s, when m PE's are used is determined by the following ratio:

$$s = \frac{T_1}{T_m} = \frac{W}{T_m} \geq 1. \qquad (3.6)$$

When m PE's participate in the processing of G, some PE's may be idling during certain periods of time. As a whole, we define a processor utilization factor, u, to indicate the overall percentage of time when all m PE's are doing useful work:

$$u = \frac{W}{m * T_m} = \frac{s}{m} \leq 1. \qquad (3.7)$$

Obviously, s indicates an improvement in performance by reducing the total processing time, and u relates the cost of operating m PE's to an overall system hardware utilization. Since a higher value in each factor indicates a better performance and a lower cost, we can use the product of these two factors as a figure of merit to measure the combined choice of a process being performed by a certain

system. We shall call this figure a cost-performance index, c.p.i., which is

$$c.p.i. = s * u = \frac{s^2}{m} \tag{3.8}$$

As an example, consider the execution of a single arithmetic expression E = A * B * (C - D) / (E + (F * G - H * I)) with its parse tree shown in Fig. 3.7 (a). If we let the operator weights be 2, 2, 3, and 5 for addition, subtraction, multiplication, and division, respectively, its job graph can be easily obtained as shown in Fig. 3.7 (b). The optimal schedules are shown in Fig. 3.8 for m = 4, 3, and 2, as can be easily verified. It is interesting to note that although a computation of the lower bound on the number of PE's required to process the tree in its critical-path length time (12) gives $m_L = 3$, this is not actually achievable.

If we apply equations (3.6) ~ (3.8), we have the following:

For m = 4, $s = \frac{23}{12} = 1.917$

$$u = \frac{23}{12*4} = 0.479$$

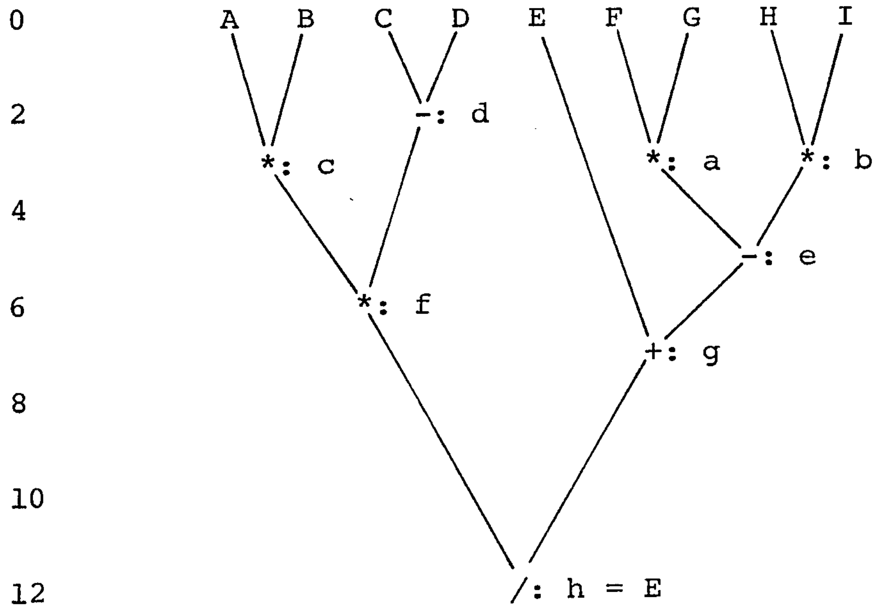
$$c.p.i. = s * u = 0.918.$$

For m = 3, s = 1.769, u = 0.589, and c.p.i. = 1.042.

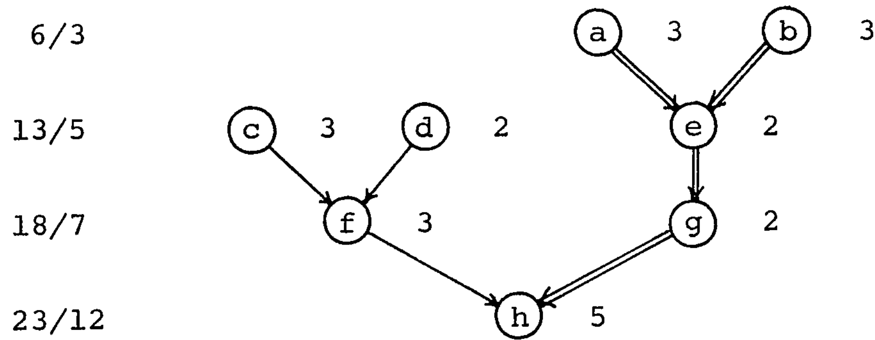
For m = 2, s = 1.643, u = 0.821, and c.p.i. = 1.349.

And, of course, for m = 1, s = u = c.p.i. = 1.

Above results have been plotted in Fig. 3.9. It is interesting to note that although s is always a non-decreasing function of m, if optimal scheduling is used, its slope

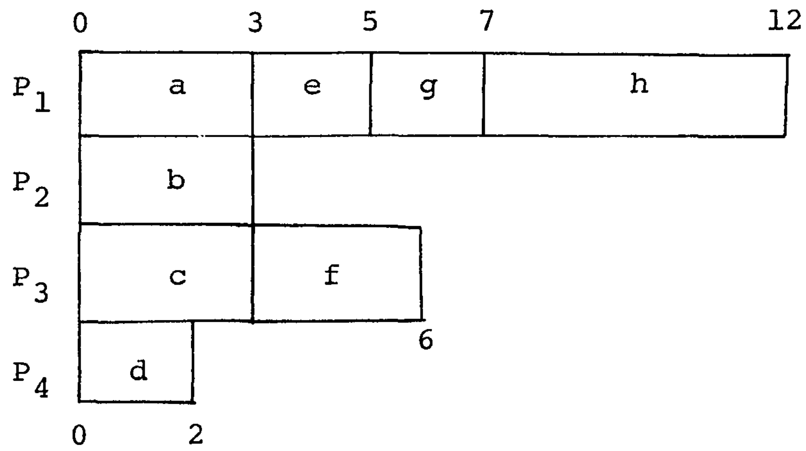


(a) A parse tree for E = A*B*(C-D)/ (E+(F*G-H*I)).

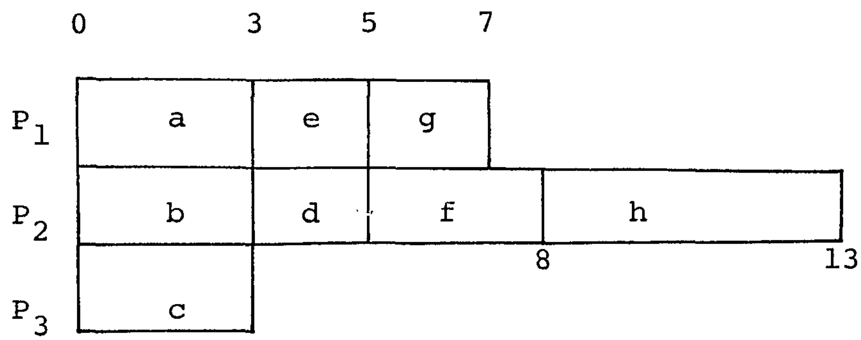


(b) The job graph. Numbers outside are node weights.

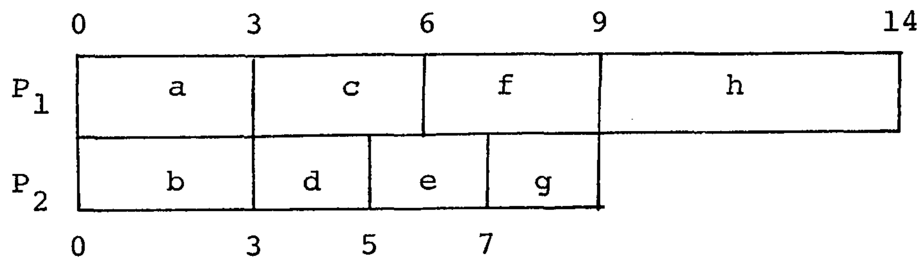
Fig. 3.7. An arithmetic expression used for scheduling analysis.



(a) m = 4



(b) m = 3



(c) m = 2

Fig. 3.8. Optimal schedules for the graph in Fig. 3.7.

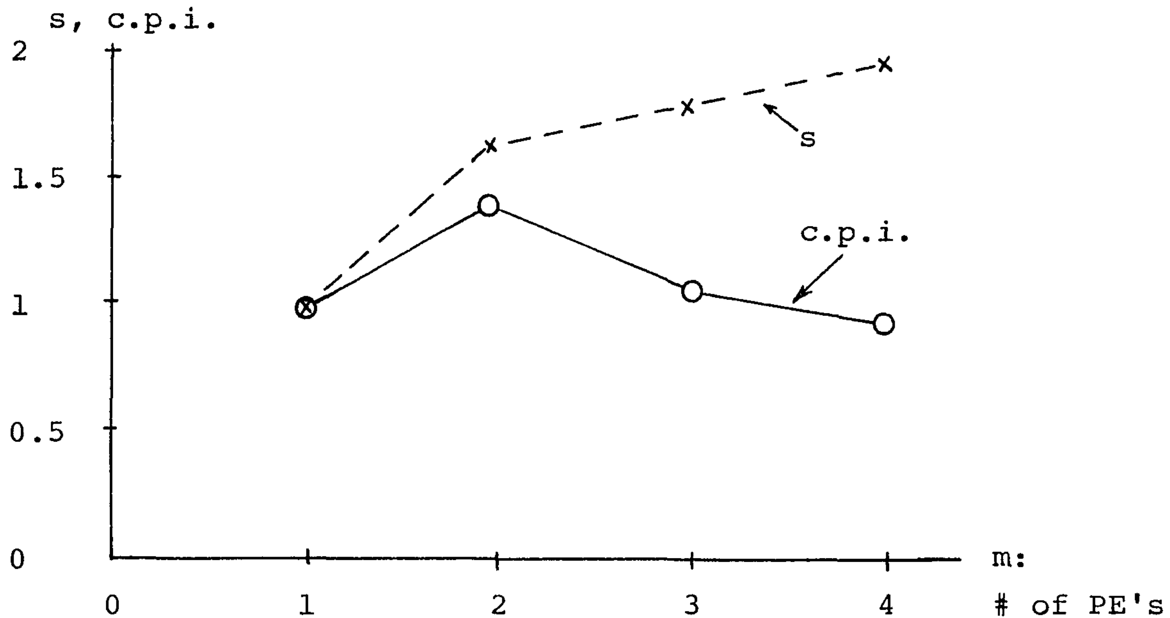


Fig. 3.9. A plot of speed-up factor, s, and cost-performance index, c.p.i., vs. the number of PE's.

levels off as more PE's are used, and the c.p.i. has a peak at m = 2. This makes it easy to determine the best choice on the number of PE's to use if c.p.i. is the prime factor of concern.

However, a word of caution is due for interpreting the above analysis. It should not be inferred that a system of many PE's improves little and costs more. What we have shown above is only a single statement, and may be a very small protion of a program. When all the interstatement and intrastatement parallelism are combined to give a total job graph of the program, a PE which finishes processing some suboperations of a statement s_i may find itself ready to process some portions of another statement s_j, even though the whole statement s_i has not yet been completed. For

example, in Fig. 3.8 for the case of $m = 3$, P_3 may be used for processing other available operations as soon as it finishes the subtask c. It is conceivable then, a large system of PE's will find its cost-performance index improving in a larger job, and the same technique of performing cost-performance analysis on a total program basis will yield a more meaningful and practical result.

We now look at another question from parsing and scheduling. So far we have implicitly assumed that the same job graph is used for various systems in comparing their cost-performance. That is, the total amount of a task is same regardless of which system it is scheduled on. However, a moment of reflection indicates that this may not be fair to a uniprocessor, or a smaller system, if some distribution is involved in minimizing a tree-height (as discussed in Chapter 2). When a distribution is performed in an arithmetic expression, the number of operations in general increases. For a uniprocessor, it certainly does not have to go through the trouble of performing the distribution and then finds itself executing a longer list of tasks than the original. Let us use the elmentary example again, $E = a(bcd + e)$ and $E^d = abcd + ae$ (see Fig. 2.2 for the parse trees). For a uniprocessor, it takes 3 multiplication times and 1 addition time to compute E, while it takes one extra multiplication time to compute E^d. But for a system with $m \geq 2$, E^d can be done in 2 multiplication times and 1 addition time. The question raised here is then:

"When should distribution be performed to minimize the processing time if the number of PE's is limited?"

Unfortunately there is no direct and easy answer to above question, since scheduling can be done only after a parse tree or job graph is obtained. And, as discussed above, it is further complicated by the fact that a statement, distributed or not, will finally be mingled with other statements to form a total program graph. Nonetheless, in the following we present a partial answer to it, which represents a local minimization.

Let E be an arithmetic expression, and E^d be its distributed form such that $h[E^d] < h[E]$, where h indicates the height of a tree as in Chapter 2. If the system has m PE's (or simply that m PE's are available for use), use this value of m in equation (3.5) of Theorem 3.1 to determine the lower bound on the processing time of E^d. Let this time be $t_m[E^d]$. Then if $t_m[E^d] < t_m[E]$, E^d is used in the program for scheduling and processing. Otherwise, simply use E in the program.

Then the analysis of cost-performance can be done as before. However, now the value of W, the total node weights, used in equations (3.6) and (3.7) should represent the actual tasks to be used for that configuration. That is, if it is decided from above discussion that an expression E^d does not help reduce computation time for $m < 2$, then E is used in the calculation of W for $m \leq 2$, but E^d is used to compute W for $m \geq 3$.

4. REARRANGEABLE DATA-CONNECTION NETWORKS

4.1 Introduction: Data-Connection Requirements

In previous two chapters we presented an analysis of parallel processing in the "preprocessing" phase so that when a task is eventually scheduled on a system of PE's for execution, the program and the system are well-matched and every suboperation may take place at the right time and be executed by the right processor. The whole task can then be processed smoothly with a minimum execution time and a high system utilization. This imposes a hard real-time requirement for the system in the processing phase: it must have a fine coordination among the PE's and memory modules in terms of the time and space control so that any piece of data, whether stored in some memory or produced by some PE, can reach a place under control at a time when it is needed. The question is then: how can these subsystems be physically connected so that data and instructions can be transmitted properly?

Naturally, this requires the use of a network whose input-output (I/O) connections can be changed dynamically under some control. Various names have been used for types of networks which can perform this versatility in one way or another: rearrangeable switching network, data-routing network, permutation network, etc. But before looking into the capability of some potential connection networks, we shall first examine some of the I/O connection requirements

most often encountered in parallel processing. For convenience, let I_i be an input terminal and O_i be an output terminal in an N-port connection network, where i = 1, 2, .., N (or 0, 1, .., N-1). We use the notation $I_i \rightarrow O_j$ to mean a one-way connection from I_i to O_j. (obviously, a two-way connection $I_i \leftrightarrow O_j$ can be achieved by duplicationg the simplex connection with I/O reversed. Hence, in principle, we shall consider only the one-way connections.)

The following basic types of connections are needed:

(A) One-to-one connection

(1) Arbitrary pair-wise connection: $I_i \rightarrow O_j$ for any permutation of i and j, where i, j = 1, 2, .., N.

This represents any possible pair-wise connections between the inputs and the outputs. For example, when N data have to be fetched from N different memory modules and sent to N different PE's in an arbitrary but predetermined way. (see Fig. 1.1 (a) for a configuration of the PE's and memory modules.)

(2) Linear shift or skewing connection:

$I_i \rightarrow O_{i+s \text{ (mod. N)}}$ where $s \geq 1$.

To illustrate the need for this type of connection, consider a matrix of data, $d_{i,j}$, where i, j = 1,..., N, to be stored in N memory modules M_1, .., M_N. If the jth column of data is stored in the corresponding memory, M_j, then later one can fetch a row of data $d_{i,1}$, .., $d_{i,N}$ simultaneously for processing. But all data in a column can only

be accessed sequentially one at a time. If the data are stored in a transposed form then simultaneous access to all row elements becomes impossible. However, if we store the data in a linearly skewed way, as shown in Fig. 4.1, then simultaneous access to all row elements or column elements becomes possible. But then we need a connection network from memories to PE's to "unskew" the ordering of data as each row is accessed. Reverse process will have to be performed when data are stored.

M_1	M_2	M_3	M_4
d_{11}	d_{12}	d_{13}	d_{14}
d_{24}	d_{21}	d_{22}	d_{23}
d_{33}	d_{34}	d_{31}	d_{32}
d_{42}	d_{43}	d_{44}	d_{41}

Fig. 4.1. Skewed storage of data array in memory modules.

(3) Perfect shuffle [31]

This type of connections resembles the order produced when a deck of sequentially numbered cards is cut in two and then shuffled in a perfect way such that only one card is drawn from each half-deck at a time. For N=8, Fig. 4.2 illustrates the numbers before and after a shuffle together with their binary representations.

Binary code $b_2\ b_1\ b_0$	Numbers before shuffle, J	Numbers after shuffle, I	Binary code $a_2\ a_1\ a_0$
000	0	0	000
001	1	4	100
010	2	1	001
011	3	5	101
100	4	2	010
101	5	6	110
110	6	3	011
111	7	7	111

Fig. 4.2. A perfect shuffle for N = 8.

Since after shuffling, a number $i \in I$ is at a new position indicated by the number $j \in J$ on the same row before shuffling, we can consider i as an input terminal to a network through which it is connected to an output terminal j. For example, 4 is connected to 1. In general, as can be verified from Fig. 4.2, the following connections are required to perform a perfect shuffle for N terminal pairs:

$$I_i \rightarrow O_{2i} \qquad \text{for } 0 \leq i \leq N/2 - 1,$$

$$I_i \rightarrow O_{2i+1-N} \qquad \text{for } N/2 \leq i \leq N - 1.$$

It is also to be noted that, if we represent the numbers in regular binary form, then to obtain a pair of above connections it merely involves a rotation of the binary bits. A rotation of $a_2a_1a_0$ to the left (which becomes $a_1a_0a_2$) gives

$b_2b_1b_0$. Similarly, a rotation of $b_2b_1b_0$ to the right restores the number to its original position.

Perfect shuffle can be used to arrange data in a sequence as may be desired , for example, in the computation of the fast Fourier transforms (FFT).

(B) One-to-many connection (broadcasting)

There are occasions when a piece of data or instruction must be shared by more than one PE. This may happen as a result of parsing and scheduling in a general computation program, or in a more orderly array computation in an SIMD system. Besides the arbitrary one-to-many connection, other basic connections are:

(1) Column broadcasting or row broadcasting:

$I_i \rightarrow O_j$, where $j = i, i+1, \ldots, i+k, \ldots$

For example, in a matrix multiplication, A x B = C, it may be desirable to broadcast an element of A to a whole row of B and perform a row of multiplications simultaneously, as illustrated in Fig. 4.3.

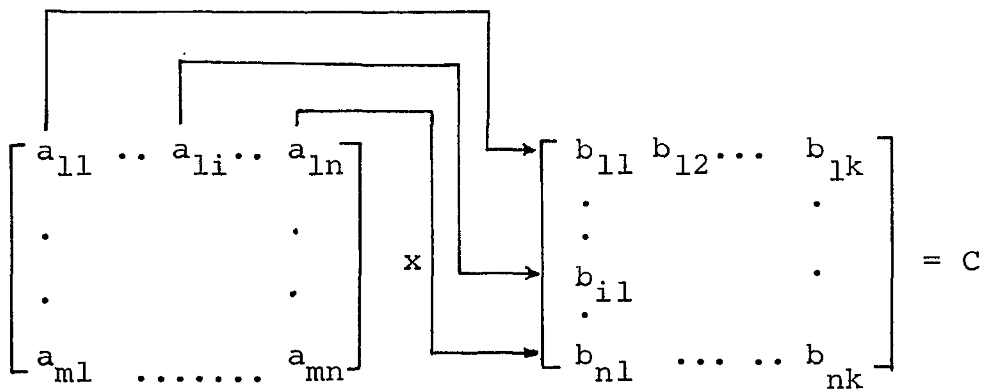


Fig. 4.3. Row broadcasting in matrix multiplication.

(2) 4-neighbor connection:

$I_i \rightarrow O_{i+j}$ where $j = \pm 1, \pm \sqrt{N}$.

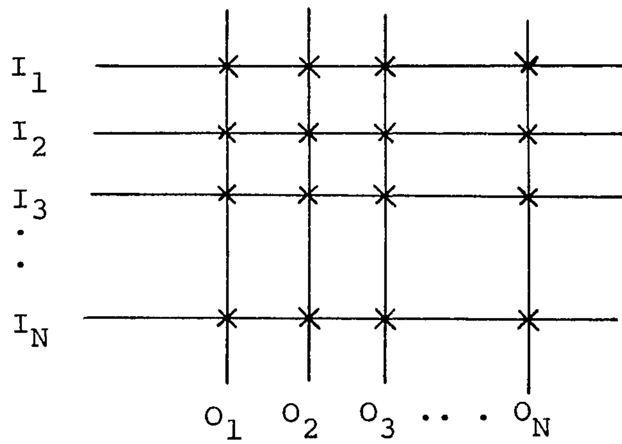
This type of connection is required, for example, in the solution of the Laplace's equation using iterative procedures in which an array point is updated by averaging its four neighboring values. If N PE's are used in a $\sqrt{N} \times \sqrt{N}$ array, then in general, the ith PE will have to communicate with its four neighbors whose coordinates are $i + 1$, $i - 1$, $i + \sqrt{N}$, and $i - \sqrt{N}$.

It is clear that if a network can do any arbitrary pair-wise connection, it can also do others listed under type (A). For a simultaneous one-to-many connection, enough channels will have to be duplicated, or allow some I/O terminals to share the same information. In the next section we shall describe networks which can realize all o1 some of the above connection requirements.

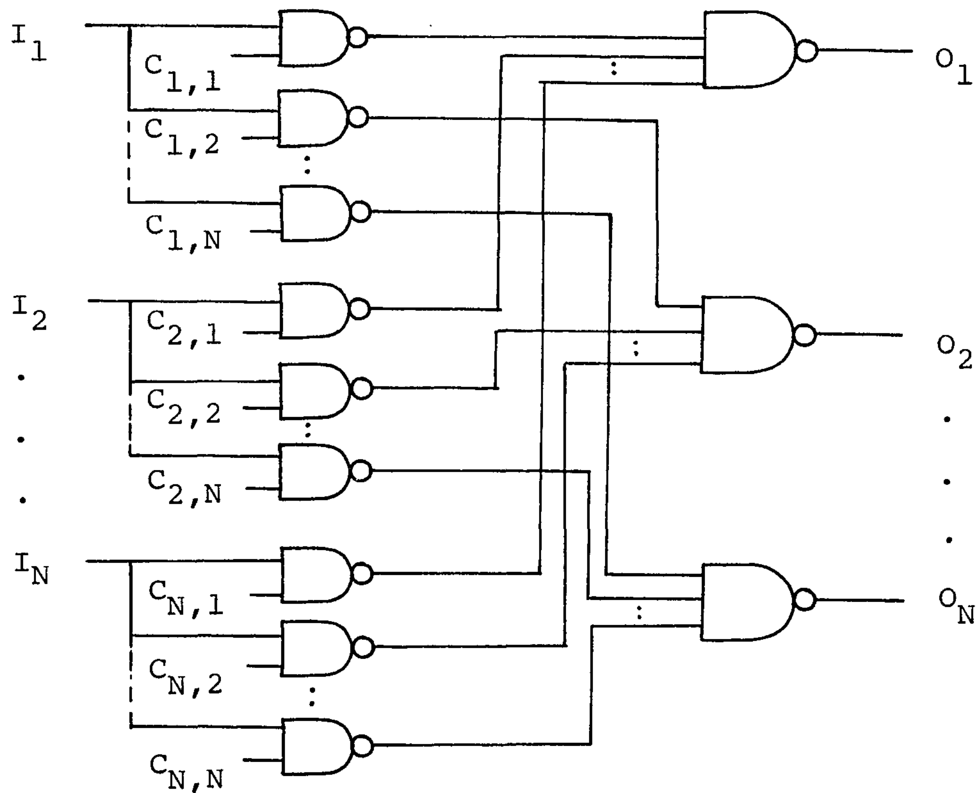
4.2 Data-Connection Networks

(1) Cross-bar switch

Conceptually the simplest type, the cross-bar switch is easy to construct and control. It can make any arbitrary connection in a minimum connecting time. The main disadvantage with this network is the large number of cross-point switches required, which amounts to a higher hardware cost. A basic diagram of a cross-bar switch is shown in Fig. 4.4(a). A digital version of the switch is shown in Fig. 4.4 (b), together with control lines such that a connection is made



(a). A basic diagram.



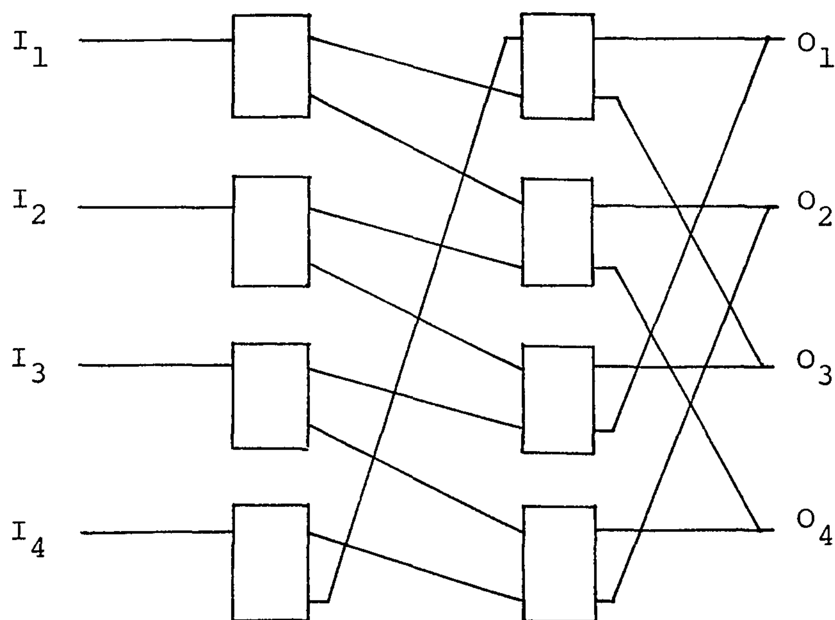
(b). With NAND-gate implementation.

Fig. 4.4. A cross-bar switch.

from I_i to O_j if $C_{i,j} = 1$. Notice that it is possible to make a one-to-many connection simply by allowing the corresponding control lines to become 1. If we assume that all gates have same complexity, then a digital N x N cross-bar switch has a data gate count of approximately N^2, and a propagation delay independent of the size N (i.e., only two levels of gates are involved in each data path).

(2) A barrel shifter

A barrel shifter employs binary switching elements in a number of stages with interstage connections made in such a way that stage i (i = 1, 2, ...) will shift its inputs by either 0 or 2^{i-1} positions to stage i+1. Thus a total number of $\log_2 N$ stages can perform any linear shift simultaneously for all inputs. Each binary switching element can be considered as a 2 x 2 programmable cross-bar switch. Fig. 4.5 shows the diagram of a 4-port barrel shifter.



Figl 4.5. A 4-port barrel shifter.

(3) Ω-network

A rather new type of data connction network called an Ω-network has been described by Lawrie []8],[19]. The network is based on the operation of a perfect shuffle. It has a reasonably modest structure and control scheme, and can perform most of the orderly data connection requirements encountered in array processing. However, it cannot do an arbitrary permutation as may be desired.

Fig. 4.6 shows a basic 8 x 8 Ω-network using 2 x 2 switches as basic elements, and an example of connection path from I_2 to O_6. As can be seen from the diagram, the interstage connections assume the pattern of a perfect shuffle, and there are as many stages as the number of bits when the terminals are expressed in their binary representations. Each of the 2 x 2 switches also allows a one-to-two broadcasting from one of its inputs to both outputs.

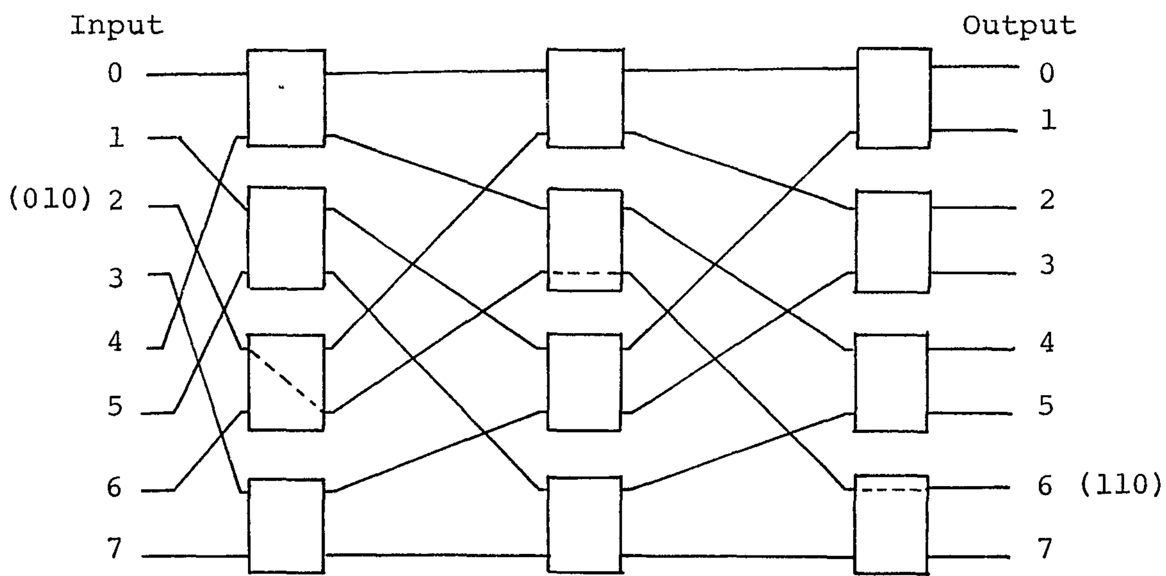


Fig. 4.6. An 8 x 8 Ω-network and a connection path.

To determine the path of a connection and hence the states of all basic switches associated, encode the I/O terminals in binaries (actually, only the output binary is sufficient to determine the state of an element). If an output terminal is coded as $d_1d_2d_3\ldots$, then starting from an input terminal s, go to a switching element s_i in stage 1 to which the input s is connected, and connect it to the upper output terminal of s_i if $d_1 = 0$, or to the lower output terminal of s_i if $d_1 = 1$. Then follow the fixed interstage connection to another element s_j in stage 2 and determine the state of s_j according to d_2 in a similar way. Continue this process until the final output is reached. The path shown in Fig. 4.6 illustrates a connection between I_2 (010) and O_6 (110).

From the fact that only the output code is used in determining the state of an switching element, it is clear that not all connections are possible, due to a possible merge (a conflict in connections). For example, if it is also desired to have a connection I_6 (110) $\rightarrow$ O_5 (101) in addition to the connection shown, then there is a merge in stage 1. Since the output can not distinguish the two different inputs, it is not possible for both connections to be realized simultaneously. Similar conflicts can occur at every switching element in every stage, hence the permissible simultaneous connections are restricted. However, it has been shown that most of the orderly data connections encountered in array processing can be realized with such an

Ω-network. It is also possible to implement an Ω-network with higher-order switching elements, but then the interstage connections and path-finding algorithm become much more complicated.

(4) Multistage rearrangeable switching network (RSN)

Multistage rearrangeable switching networks have been originally discussed by Clos [7] and Benes [4] for telephone switching application. This type of connection network has the main advantage of being able to make any pair-wise connection between the I/O terminals simultaneously. Fig. 4.7 shows a basic 3-stage RSN for N inputs and N outputs. The network is symmetric: the input and the output stages each consists of q subnetworks made up of d x d switching elements, and the middle stage has d q x q switching elements, where d and q are integer factors of N such that d * q = N. It is obvious that, in general, the middle stage can further be decomposed into another 3-stage RSN, and so on, resulting in a multistage RSN. In particular, if N is a number equal to a power of r, the network can be decomposed into $(2 \log_r N - 1)$ stages, each stage having N/r of the r x r switching elements. This particular form is referred to as a base-r structure. Since an r x r cross-bar switch has a gate count of approximately r^2, if we compare the one-sided (from the middle stage to either input or output) gate count of a base-r RSN with that of a base-2 system, we have the following ratio:

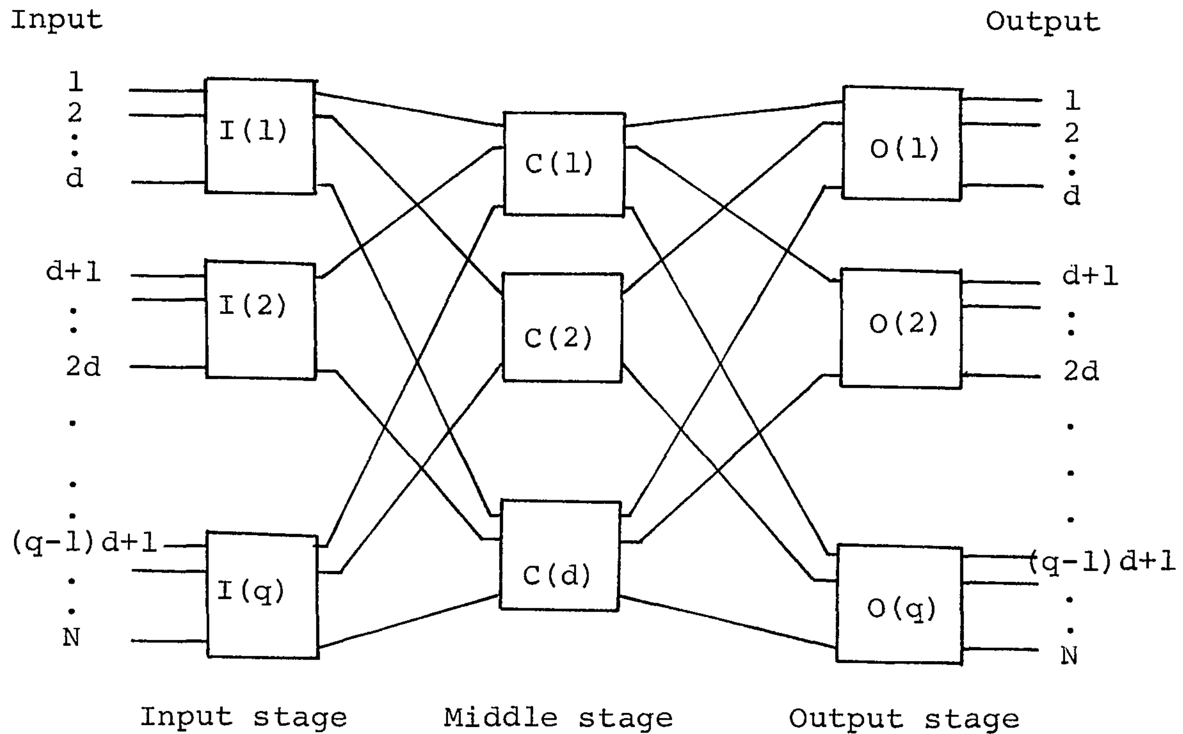


Fig. 4.7. A general 3-stage rearrangeable switching network.

$$\frac{N/r \cdot r^2 \cdot \log_r N}{N/2 \cdot 2^2 \cdot \log_2 N}$$

$$= \frac{r}{2} \cdot \frac{\frac{\log_2 N}{\log_2 r}}{\log_2 N} = \frac{r}{2 \log_2 r} .$$

Above ratio is 1 at r = 2 and 4, with a minimum slightly below 1 about r = 3, and increases its value for $r \geq 4$. For practical purposes, it is generally true that a lower-base structure is more economical in terms of the total hardware cost. On the other hand, since the propagation delay through the system is proportional to the number of stages, $2 \log_r N - 1$, a higher-base system will have a shorter

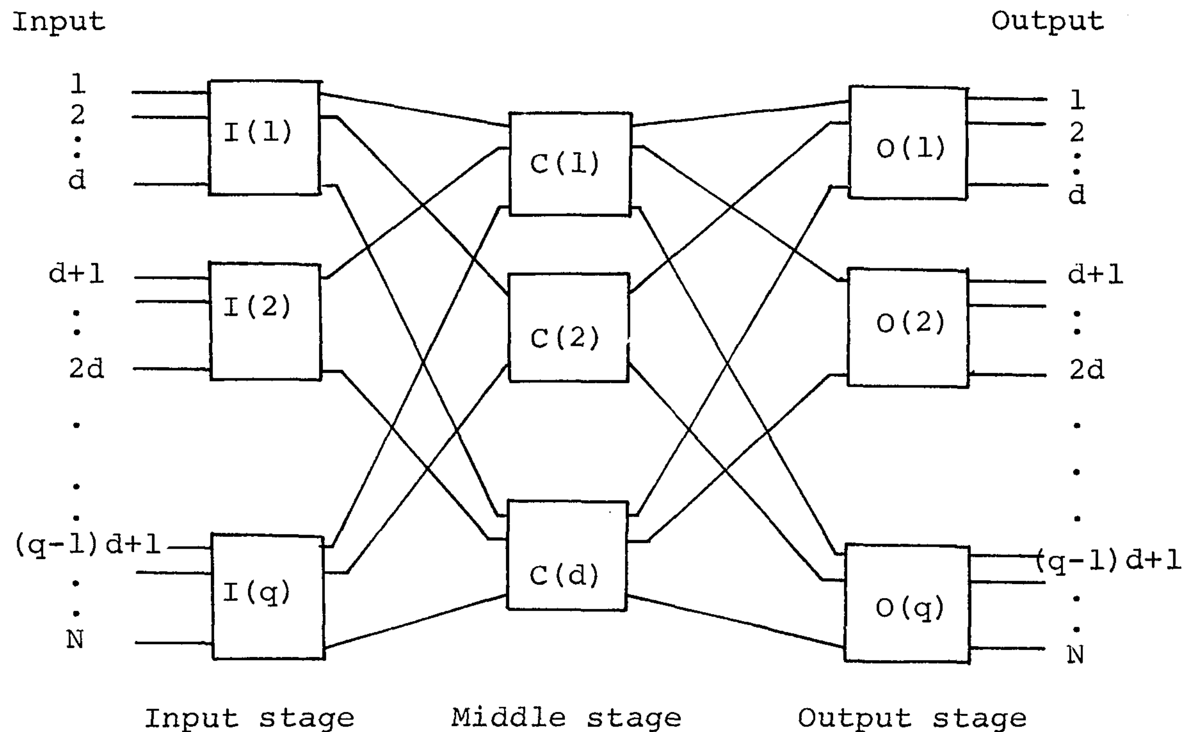


Fig. 4.7. A general 3-stage rearrangeable switching network.

$$\frac{N/r \cdot r^2 \cdot \log_r N}{N/2 \cdot 2^2 \cdot \log_2 N}$$

$$= \frac{r}{2} \cdot \frac{\dfrac{\log_2 N}{\log_2 r}}{\log_2 N} = \frac{r}{2 \log_2 r} .$$

Above ratio is 1 at $r = 2$ and 4, with a minimum slightly below 1 about $r = 3$, and increases its value for $r \geq 4$. For practical purposes, it is generally true that a lower-base structure is more economical in terms of the total hardware cost. On the other hand, since the propagation delay through the system is proportional to the number of stages, $2 \log_r N - 1$, a higher-base system will have a shorter

transmission delay and also requires fewer decompoditions to be performed. So the choice of a base amounts to a trade-off between the performance and the cost. However, with the advancing LSI technology, the cost factor is much lessened, and it may be desirable and more convenient to use a large size permuter as the basic switching element.

The controlling aspect (i.e., to find a connection path through the switching elements in various stages) of a multistage RSN is a nontrivial problem. As may be expected, a base-2 structure is the easiest one to control. Opferman and Tsao-Wu [25] gave control algorithms for systems with base-2 structures. However, for a general system with arbitrary integers of d and q , where $d * q = N$, their algorithm would involve a process of trials. Neiman [24], Ramanujam [29], and Chen and Frank [26] also described decomposition algorithms for a general case. But all these algorithms are of heuristic nature, i.e., some possible connection paths are assumed, and if conflicts should occur later, some corrective measures are taken to rearrange the connections. In contrast to this approach, we shall present a definitive control algorithm for an arbitrarily decomposed 3-stage RSN in the next section. The new algorithm is simple to use and eliminates the need to alter the already established connections, which would be required by other methods.

We summarize the discussion of above four data-connection networks in Table 4.1. It is clearly seen that

Network	no. of basic switching elements	no. of stages	Control scheme	I/O connection capabilities
Cross-bar switch	N^2	1	single stage	any permutation
Barrel shifter	$N \log_2 N$	$\log_2 N$	all stages simultaneous	uniform shifts
Ω-network	$\frac{N}{2} \log_2 N$	$\log_2 N$	stages in sequence or parallel	limited permutation
RSN (base-r)	$\frac{N}{r} (2 \log_r N - 1)$	$2 \log_r N-1$	all stages simultaneous	any permutation
RSN (3-stage)	$\sqrt{2N} \cdot 2N$ (optimal)	3	all stages simultaneous	any permutation

Table 4.1. Summary of four data-connection networks.

a properly chosen base-r RSN (or more generally, a d x q factored RSN) represents a compromise between hardware cost and speed (assuming the propagation delay proportional to the number of stages) compared to a cross-bar switch, if any permutation of connections is to be allowed.

4.3 Decomposition of a 3-stage RSN

4.3.1 A Definitive Decomposition Algorithm

Before we present the algorithm, it is helpful to review the structure of a 3-stage RSN as given in Fig. 4.7. Notice that for the interstage connections, the jth output terminal of the ith input block is connected to the ith input terminal of the jth middle-stage block, and the system is symmetrical around the middle stage. The factoring of $N = d \times q$ is arbitrary as far as the decomposition algorithm is concerned.

The overall connection requirement is specified by a $2 \times N$ matrix P, as shown in the following, where each column corresponds to one pair of I/O connection:

$$P = \begin{bmatrix} 1 & 2 & 3 & \ldots & i & \ldots & N \\ \pi(1) & \pi(2) & \pi(3) & & \pi(i) & & \pi(N) \end{bmatrix}$$

$$= \begin{bmatrix} p_i \\ \pi(p_i) \end{bmatrix} .$$

P can also be represented in an $N \times N$ matrix form, in which an element $p_{i,j} = 1$ if and only if an input terminal i is to be connected to an output terminal j, otherwise $p_{i,j} = 0$. In a one-to-one connection, there will be only one "1" in each row and each column. This form of connection

matrix is also known as a permutation matrix.

The goal of the decomposition algorithm is to generate permutation functions for all subnetworks so that the overall connections through the stages are same as those specified by the permutation matrix P. Since the input and the output terminals are each divided into q groups, the main effort of the algorithm is to obtain a set of permutations P_i for each middle stage block C(i) so that it contains as its input one and only one output terminal from each input block I(j), and as its output one and only one input terminal to each output block O(j), where $1 \leq i \leq d$ and $1 \leq j \leq q$. The permutations for the input stage and the output stage are then obtained from the combined knowledge of P_i and the original permutation P. This completes the connections of a 3-stage rearrangeable network as required by P.

If desired, each permutation P_i for the middle stage block C(i) can be further decomposed in a similar way. Then we would have a 5-stage network. Since the decomposition procedure is identical, we will not elaborate on this any further.

We now describe the decomposition algorithm with the aid of an example for clarity. The validity of the algorithm and the proof of the main permutation matrix decomposition will follow the algorithm. We shall use the same example as given in [24].

The permutttion is given by:

$$P = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 9 & 7 & 22 & 24 & 14 & 16 & 17 & 18 & 23 & 21 & 15 & 19 \\ 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \\ 1 & 20 & 13 & 2 & 10 & 5 & 3 & 12 & 6 & 8 & 11 & 4 \end{bmatrix},$$

and we are required to implement it with $d = 4$ and $q = 6$. Of course, here $N = d \times q = 24$.

In the following discussion of the algorithm, d, q, and N will all be fixed integers, not variables, for a given problem.

Step 1. Transform the string of numbers contained in P into a 2-dimensional $q \times d$ array so that each number m consists of two components (x,y). We call x the block index, and y the terminal number within block x. Hence, for any m such that $1 \leq m \leq N$ we have $x = \left\lceil \frac{m}{d} \right\rceil$, and $y = m - (x - 1)d = (m)_{\text{mod. } d}$, with elements of the set being $\{1, 2, .., d\}$.

This format allows us to identify an input or output terminal in terms of the actual subnetwork block number and the terminal within that block. Let the permutation matrix whose elements have been so transformed be denoted by P'. Then we have

$$P' = \begin{bmatrix} p'_i \\ \pi(p'_i) \end{bmatrix}, \quad i = 1, .., N.$$

Or, written in terms of x and y components with new indexing,

$$P' = \begin{bmatrix} (x_i, y_j) \\ \pi(x_i, y_j) \end{bmatrix},$$

for $i = 1, .., q$, and $j = 1, .., d$.

For our example,

$$P' = \begin{bmatrix} 11 & 12 & 13 & 14 & 21 & 22 & 23 & 24 & 31 & 32 & 33 & 34 & 41 & 42 & 43 & 44 & 51 & 52 & 53 & 54 & 61 & 62 & 63 & 64 \\ 31 & 23 & 62 & 64 & 42 & 44 & 51 & 52 & 63 & 61 & 43 & 53 & 11 & 54 & 41 & 12 & 32 & 21 & 13 & 34 & 22 & 24 & 33 & 14 \end{bmatrix}.$$ [note 1]

Next we partition P' into A_i and B_i sets, $i = 1, .., q$, according to the input block index, so that A_i contains the input terminals from input block i, and B_i contains all the outputs which are to be connected to input block i. That is,

$$A_i = \{ (x_i, y_j) \mid x_i = i \}$$

$$B_i = \{ \pi(x_i, y_j) \mid (x_i, y_j) \in A_i \}.$$

For our example, we have

$A_1 = \{ 11, 12, 13, 14 \}$, $A_2 = \{ 21, 22, 23, 24 \}$,

$B_1 = \{ 31, 23, 62, 64 \}$, $B_2 = \{ 42, 44, 51, 52 \}$,

$A_3 = \{ 31, 32, 33, 34 \}$, $A_4 = \{ 41, 42, 43, 44 \}$,

$B_3 = \{ 63, 61, 43, 53 \}$, $B_4 = \{ 11, 54, 41, 12 \}$,

$A_5 = \{ 51, 52, 53, 54 \}$, $A_6 = \{ 61, 62, 63, 64 \}$,

$B_5 = \{ 32, 21, 13, 34 \}$, $B_6 = \{ 22, 24, 33, 14 \}$.

Step 2. Construct a q x q matrix M whose element $m_{i,j}$ consists of a set obtained by the intersection of A_i and B_j:

$$M = [m_{i,j}] \quad \text{where } m_{i,j} = A_i \cap B_j, \text{ for } i, j = 1, .., q.$$

note 1: For convenience, we omit the comma between two component numbers. In our example, since both x and y are smaller than 10, this omission will not cause any confusion.

Then construct a matrix H whose elements $h_{i,j}$ are the cardinalities of the corresponding elements of M:

$$h_{i,j} = |m_{i,j}| , \ i, j = 1, .., q.$$

For our example,

$$M = \begin{bmatrix} \emptyset & \emptyset & \emptyset & \{11,12\} & \{13\} & \{14\} \\ \{23\} & \emptyset & \emptyset & \emptyset & \{21\} & \{22,24\} \\ \{31\} & \emptyset & \emptyset & \emptyset & \{32,34\} & \{33\} \\ \emptyset & \{42,44\} & \{43\} & \{41\} & \emptyset & \emptyset \\ \emptyset & \{51,52\} & \{53\} & \{54\} & \emptyset & \emptyset \\ \{62,64\} & \emptyset & \{61,63\} & \emptyset & \emptyset & \emptyset \end{bmatrix}$$

$$\text{and } H = \begin{bmatrix} 0 & 0 & 0 & 2 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 2 \\ 1 & 0 & 0 & 0 & 2 & 1 \\ 0 & 2 & 1 & 1 & 0 & 0 \\ 0 & 2 & 1 & 1 & 0 & 0 \\ 2 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}.$$

Notice that H is a composite permutation matrix, and

$$\sum_{i=1}^{q} h_{i,j} = \sum_{j=1}^{q} h_{j,i} = d, \text{ for all } i \text{ and } j.$$

<u>Step 3.</u> In this step we decompose the matrix H into d permutation matrices, one for each middle stage block at a time, and store the terminal information in a matrix C.

Let t be a sequence variable from 1 to d. Initially t = 1. Then follow these steps:

(1) Compute the number of non-zero elements in each row, H_i, where i = 1, .., q, and in each column, V_j, where j = 1, .., q.

(2) Let min $[H_i, V_j] = z$, where $H_i \neq 0$ and $V_j \neq 0$. Find a non-zero element, $h_{m,n}$, among the untagged elements of matrix H and circle it. The coordinates, namely, row m and column n, are determined as follows: [note 1]

(a) If $z = 1$, $h_{m,n}$ is found on row m with $H_m = 1$. Obviously, column n is then uniquely identified.

(b) If $z = 2$, two cases are possible:

i) If there exist at most two rows with $H_i = 2$, find $h_{m,n}$ on any row m with $H_m = 2$ and in an arbitrary column n.

ii) If there are three or more rows with $H_i = 2$, choose $h_{m,n}$ from a row and a column such that $H_m = V_n = 2$, provided that this is possible.

Otherwise find a column n such that $V'_n = \min [V'_j]$ where V'_j is the number of non-zero and untagged elements in column j across only the rows i which have $H_i = 2$. Select $h_{m,n}$ then on any row m such that $H_m = 2$.

(c) If $z \geq 3$, arbitrarily choose $h_{m,n}$ with $H_m \neq 0$, and $V_n \neq 0$.

note 1: In this description, we assume that z is found among H_i, i.e., $z = H_m$ for some m, and primarily use H_i for decision. If z is among V_j instead, similar procedures are followed with V_j primarily used for decision and the roles of a row and a column interchanged.

(3) Subtract 1 rom all $h_{m,n}$ circled in step 3.(2) and tag all other non-zero elements in row m and column n, if they have not been tagged yet.

(4) For some $j \neq n$, if $h_{m,j}$ on row m is tagged in step 3.(3), subtract 1 from the corresponding V_j and let $V_n = 0$.

Similarly, for some $i \neq m$, if $h_{i,n}$ in column n is tagged in step 3.(3), subtract 1 from the corresponding H_i and let $H_m = 0$.

(5) Remove one element of matrix M from row m and column n, determined from above substeps, and place it in a matrix C at row t and column n, where t is the sequence variable.

(6) If any $H_i \neq 0$, $i = 1, .., q$, go back to step 3.(2). Otherwise increment the value of t by one and continue.

(7) If $t \neq d$, erase all circles and tags in matrix H and go back to step 3.(1).

Otherwise $(t = d)$, assign the remaining elements of matrix M, $m_{i,j}$, to $c_{d,j}$ of matrix C, where $i, j = 1, \ldots, q$, and go to next step.

A simplified flow chart of Step 3 is shown in Fig. 4.8. For our example, we do the following:

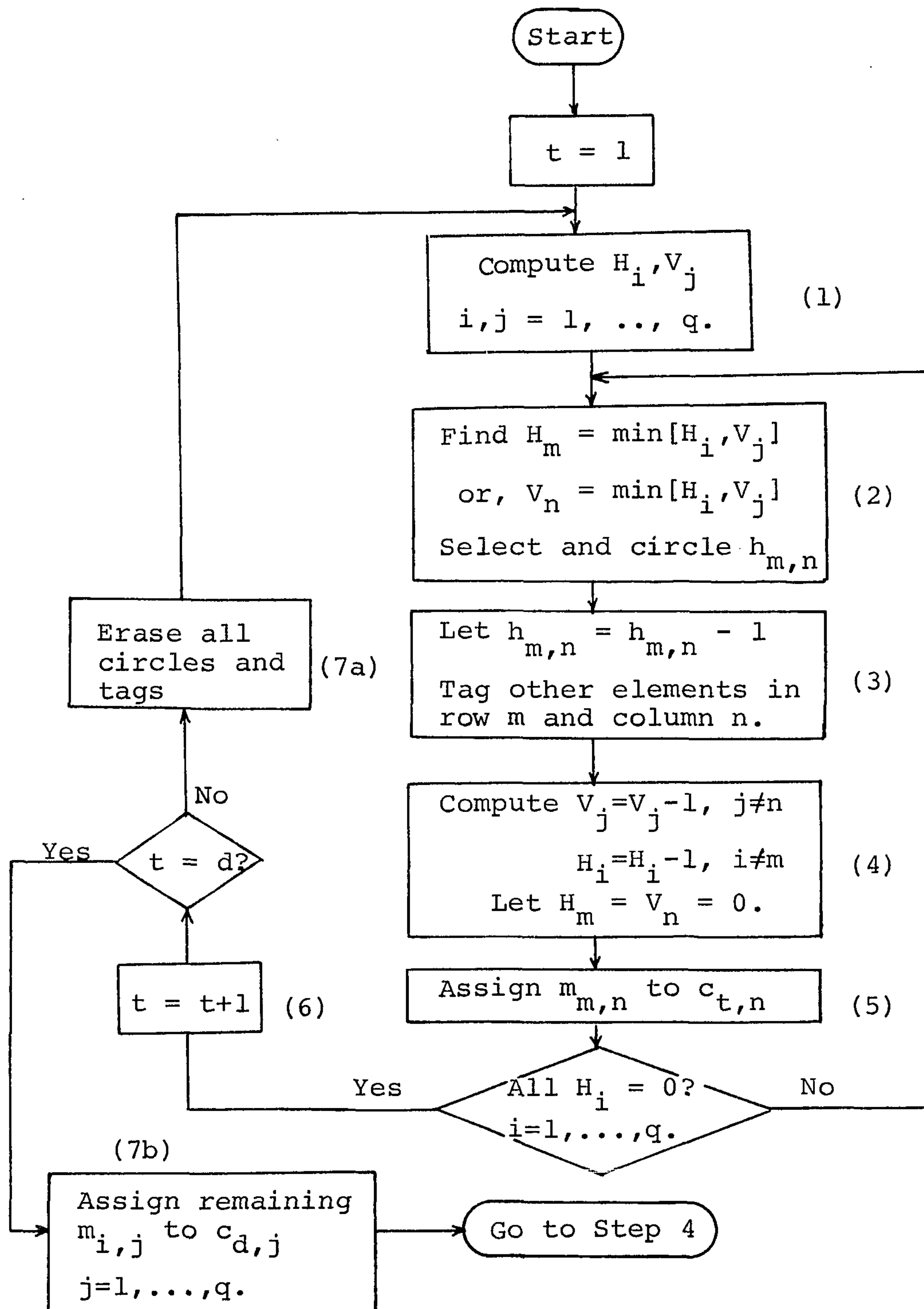


Fig. 4.8. A flow chart for permutation matrix decomposition.

At t = 1,

	1	2	3	4	5	6	H_i						Sequence
$H =$	0	0	0	$\textcircled{2}$	$\underline{1}$	$\underline{1}$	3	3	2	1	0		4
	$\underline{1}$	0	0	0	$\textcircled{1}$	$\underline{2}$	3	2	0				2
	$\underline{1}$	0	0	0	$\underline{2}$	$\textcircled{1}$	3	2	1	0			3
	0	$\textcircled{2}$	$\underline{1}$	$\underline{1}$	0	0	3	3	3	3	2	0	5
	0	$\underline{2}$	$\textcircled{1}$	$\underline{1}$	0	0	3	3	3	3	2	1	6
	$\textcircled{2}$	0	$\underline{2}$	0	0	0	2	0					1
V_j:	3	2	3	3	3	3							
	0	2	2	3	3	3							
		2	2	3	0	2							
		2	2	3		0							
		2	2	0									
		0	1										
			0										

Since initially $H_6 = 2$ is a minimum, we select $h_{6,1} = 2$ and circle it. After tagging elements on row 6 and col. 1, we have new $H_2 = H_3 = 2$. The new H_i and V_j are shown in a new column and row, respectively. Hence we select the first untagged element on row 2, $h_{2,5}$, and circle it. Then we tag two elements in col. 5, which resuls in $H_3 = 1$. So we select $h_{3,6}$ next. By the same process we then select $h_{1,4}$, $h_{4,2}$, and at last $h_{5,3}$. The numbers in the "sequence" column indicate this sequence.

We then place the elements from matrix M corresponding to the circled positions in the first row of matrix C. These are: $C_1 = \{ 62, 42, 53, 11, 21, 33 \}$.

After we subtract 1 from all circled $h_{i,j}$, the remaining H matrix is used for the next-round decomposition. By following same procedure, we obtain the following circled H matrices and other parts of matrix C.

At t = 2,

$$H = \begin{bmatrix} 0 & 0 & 0 & 1 & \textcircled{1} & 1 \\ 1 & 0 & 0 & 0 & 0 & \textcircled{2} \\ \textcircled{1} & 0 & 0 & 0 & 2 & 0 \\ 0 & \textcircled{1} & 1 & 1 & 0 & 0 \\ 0 & 2 & 0 & \textcircled{1} & 0 & 0 \\ 1 & 0 & \textcircled{2} & 0 & 0 & 0 \end{bmatrix},$$

$$C_2 = \{ 31, 44, 61, 54, 13, 24 \}.$$

At t = 3,

$$H = \begin{bmatrix} 0 & 0 & 0 & \textcircled{1} & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & \textcircled{2} \\ 1 & 0 & 0 & 0 & \textcircled{1} & 0 \\ 0 & 0 & \textcircled{1} & 1 & 0 & 0 \\ 0 & \textcircled{2} & 0 & 0 & 0 & 0 \\ \textcircled{1} & 0 & 1 & 0 & 0 & 0 \end{bmatrix},$$

$$C_3 = \{ 64, 51, 43, 12, 34, 22 \}.$$

The last row of matrix C is obtained directly from the remaining M-matrix.

$$C_4 = \{ 23, 52, 63, 41, 32, 14 \}.$$

Hence we have the complete matrix C:

$$C = \begin{bmatrix} 62 & 42 & 53 & 11 & 21 & 33 \\ 31 & 44 & 61 & 54 & 13 & 24 \\ 64 & 51 & 43 & 12 & 34 & 22 \\ 23 & 52 & 63 & 41 & 32 & 14 \end{bmatrix}.$$

<u>Step 4</u>. For each element in matrix C, we find its inverse permutation from the transformed permutation function P', and write them in d separate 2 x q matrices

$$G_i = \begin{bmatrix} g_{i,j} \\ \hat{g}_{i,j} \end{bmatrix}, \text{ for } i = 1, .., d, \text{ and } j = 1, .., q,$$

That is, $\hat{g}_{i,j} = c_{i,j} = (x_m, y_n)$ for some m and n, and $g_{i,j} = \pi^{-1}(x_m, y_n)$, where the inverse permutation function π^{-1} is defined as follows:

If $p' = (x,y)$ and $\pi(p') = (u,v)$ form a permutation pair in P', then $\pi^{-1}(u,v) = (x,y)$.

For our example,

$$G_1 = \begin{bmatrix} 13 & 21 & 34 & 41 & 52 & 63 \\ 62 & 42 & 53 & 11 & 21 & 33 \end{bmatrix},$$

$$G_2 = \begin{bmatrix} 11 & 22 & 32 & 42 & 53 & 62 \\ 31 & 44 & 61 & 54 & 13 & 24 \end{bmatrix},$$

$$G_3 = \begin{bmatrix} 14 & 23 & 33 & 44 & 54 & 61 \\ 64 & 51 & 43 & 12 & 34 & 22 \end{bmatrix},$$

$$G_4 = \begin{bmatrix} 12 & 24 & 31 & 43 & 51 & 64 \\ 23 & 52 & 63 & 41 & 32 & 14 \end{bmatrix}.$$

Notice that in each G_i, the first row elements have their block indices (first component of $g_{i,j}$) arranged in increasing order. This means that $g_{i,j}$ have to be searched only from the set A_j or the jth group of P'.

<u>Step 5.</u> Let $(x,y)_2 = y$ be the terminal number (second component) of any element (x,y) in G_i. Then rearrange the index of G_i such that $(g_{i,1})_2 = i$. That is, without

changing the elements of the matrices, each G_i is indexed according to the terminal number of the first element it contains in the first column.

For our example, the re-indexed matrices G_i are:

$$G_1 = \begin{bmatrix} 11 & 22 & 32 & 42 & 53 & 62 \\ 31 & 44 & 61 & 54 & 13 & 24 \end{bmatrix},$$

$$G_2 = \begin{bmatrix} 12 & 24 & 31 & 43 & 51 & 64 \\ 23 & 52 & 63 & 41 & 32 & 14 \end{bmatrix},$$

$$G_3 = \begin{bmatrix} 13 & 21 & 34 & 41 & 52 & 63 \\ 62 & 42 & 53 & 11 & 21 & 33 \end{bmatrix},$$

$$G_4 = \begin{bmatrix} 14 & 23 & 33 & 44 & 54 & 61 \\ 64 & 51 & 43 & 12 & 34 & 22 \end{bmatrix}.$$

<u>Step 6</u>. Let $(x,y)_1 = x$ be the block index (first component) of any element (x,y) in G_i obtained from Step 5. The permutations P_i for the middle stage block $C(i)$ are obtained from the block indices of the elements in G_i. That is,

$$P_i = \begin{bmatrix} p_{i,j} \\ \hat{p}_{i,j} \end{bmatrix}, \text{ for } i = 1, .., d, \text{ and } j = 1, .., q,$$

where $p_{i,j} = (g_{i,j})_1 = j$, and,

$$\hat{p}_{i,j} = (\hat{g}_{i,j})_1 .$$

For our example,

$$P_1 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 6 & 5 & 1 & 2 \end{bmatrix},$$

$$P_2 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 5 & 6 & 4 & 3 & 1 \end{bmatrix},$$

$$P_3 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 4 & 5 & 1 & 2 & 3 \end{bmatrix},$$

$$P_4 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 5 & 4 & 1 & 3 & 2 \end{bmatrix}.$$

<u>Step 7</u>. The permutation E_i for the input block I(i) are obtained from the terminal numbers of the elements of G along column i:

$$E_i = \begin{bmatrix} e_{i,1} & e_{i,2} & \cdots & e_{i,d} \\ 1 & 2 & \cdots & d \end{bmatrix}, \quad i = 1, .., q$$

where $e_{i,j} = (g_{j,i})_2$, $j = 1, .., d$.

For our example,

$$E_1 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}, \quad E_2 = \begin{bmatrix} 2 & 4 & 1 & 3 \\ 1 & 2 & 3 & 4 \end{bmatrix},$$

$$E_3 = \begin{bmatrix} 2 & 1 & 4 & 3 \\ 1 & 2 & 3 & 4 \end{bmatrix}, \quad E_4 = \begin{bmatrix} 2 & 3 & 1 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix},$$

$$E_5 = \begin{bmatrix} 3 & 1 & 2 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}, \quad E_6 = \begin{bmatrix} 2 & 4 & 3 & 1 \\ 1 & 2 & 3 & 4 \end{bmatrix}.$$

Notice that E_1 has straight connections for all terminals. This is always true as a result of re-indexing performed in Step 5.

<u>Step 8.</u> Rearrange elements in the second rows of G_i obtained from Step 5 so that their block indices are in increasing order. Let the resulting row matrices be denoted by

$$T_i = [t_{i,j}], \; i = 1, .., d, \text{ and } j = 1, .., q,$$

with $(t_{i,1})_1 < (t_{i,2})_1 < \cdots < (t_{i,q})_1$.

For our example,

$$T_1 = [13 \quad 24 \quad 31 \quad 44 \quad 54 \quad 61]$$

$$T_2 = [14 \quad 23 \quad 32 \quad 41 \quad 52 \quad 63[$$

$$T_3 = [11 \quad 21 \quad 33 \quad 42 \quad 53 \quad 62]$$

$$T_4 = [12 \quad 22 \quad 34 \quad 43 \quad 51 \quad 64].$$

Step 9. The permutations F_i for the output blocks O(i) are obtained from the terminal numbers of the elements of the matrix T along column i:

$$F_i = \begin{bmatrix} 1 & 2 \;.. & .\; d \\ f_{i,1} & f_{i,2} \;.. & f_{i,d} \end{bmatrix}, \qquad i = 1, .., q,$$

where $f_{i,j} = (t_{j,i})_2$, $j = 1, .., d$.

For our example,

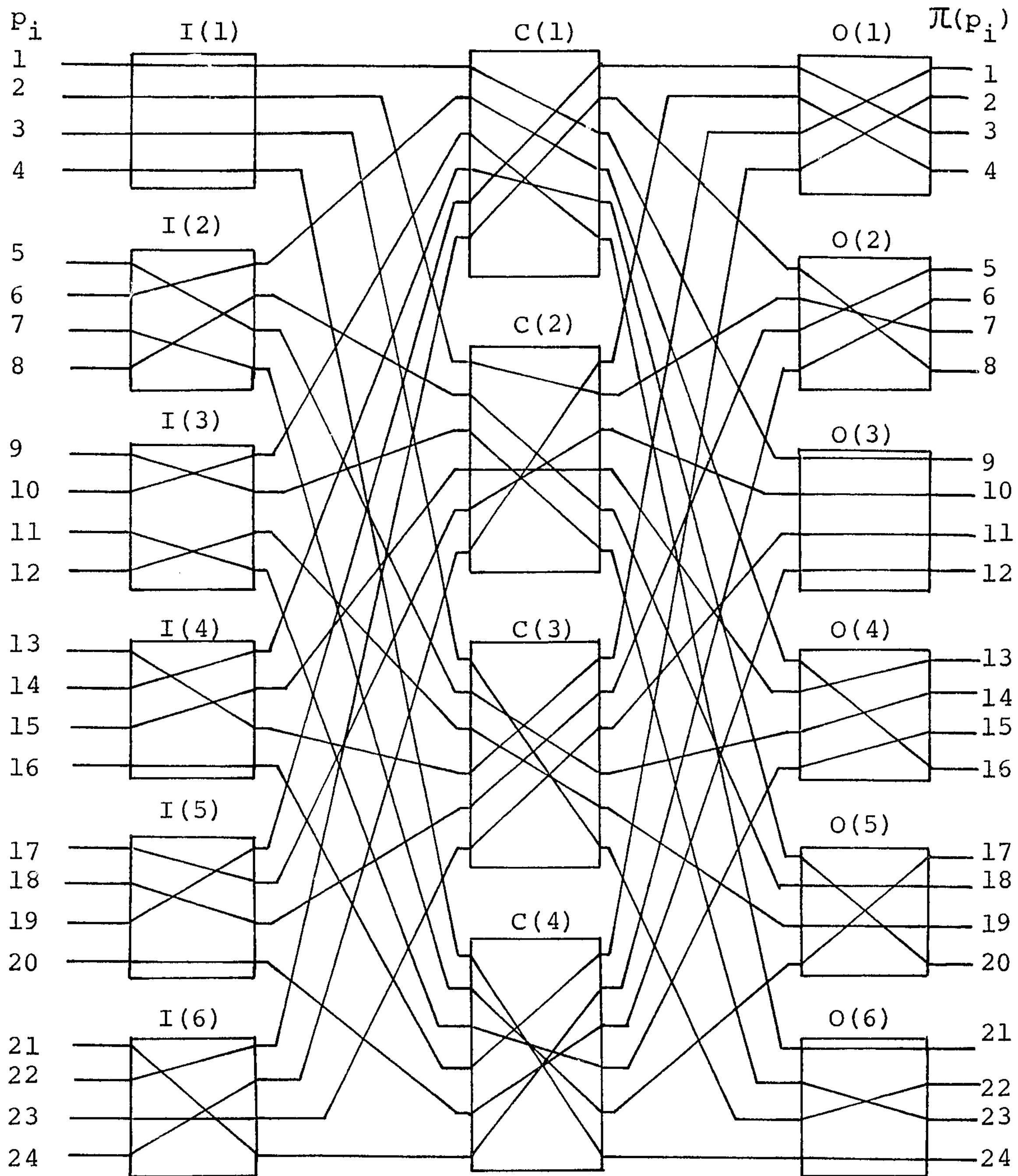
$$F_1 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 \end{bmatrix},, \qquad F_2 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 1 & 2 \end{bmatrix},$$

$$F_3 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}, \qquad F_4 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 2 & 3 \end{bmatrix},$$

$$F_5 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 2 & 3 & 1 \end{bmatrix}, \qquad F_6 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 3 & 2 & 4 \end{bmatrix}.$$

This completes our algorithm and example. The complete connection diagram of the example is shown in Fig. 4.9.

Next we prove the validity and the definitiveness of the algorithm. The essence of the algorithm is in the decomposition of the matrix H into d permutation matrices performed in Step 3. However, before we show that this can always be done according to the procedures described, we



$$P = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \\ 9 & 7 & 22 & 24 & 14 & 16 & 17 & 18 & 23 & 21 & 15 & 19 & 1 & 20 & 13 & 2 & 10 & 5 & 3 & 12 & 6 & 8 & 11 & 4 \end{bmatrix}.$$

Fig. 4.9. An example on the decomposition of a 3-stage RSN.

briefly explain the whole procedures of the algorithm.

Step 1 renames all input and output terminals so that each one is represented by a block index and a terminal number within that block. Notice that while the set A_i clearly identifies the input terminals from the input block I(i), it also represents the group of output terminals in the output block O(i) as well. This is so because the output terminals are coded exactly in the same way as input terminals. The set B_i contains all the output terminals which are to be connected to the input block I(i). Therefore, intersection of A_i and B_j in Step 2 produces the set of output terminals $m_{i,j}$ which are from output block O(i) and to be connected to input block I(j). It is clear then, if we can successfully decompose matrix M, or equivalently matrix H, into d permutation matrices such that each of them contains only one non-zero element in each row and column, then each new permutation matrix defines a set of connections for one middle stage block, because it has connections to all different input blocks and different output blocks. This is exactly what Steps 3 to 6 accomplish.

Since the jth output terminal of I(i) is connected to the ith input terminal of C(j) by the network structure, Step 7 collects one input element $g_{j,i}$ from each middle stage block C(j) for the permutation of the input block I(i). Also as mentioned before, the arrangement of indexing in Step 5 makes all connections straight-through in the first input block and eliminates one permuter.

Finally Steps 8 and 9 produce the output permutations in a way similar to the input counterparts.

Now we prove the correctness of Step 3, i.e., the matrix H can indeed always be decomposed according to the procedures shown.

For convenience, we define a term:

Definition 4.1:

An element $h_{i,j}$ of matrix H is called a critical element if the corresponding H_i of V_j, or both, as defined in Step 3 of the algorithm, are 1.

We also give a lemma:

Lemma 4.1:

At any time, matrix H can have at most one critical element in a row or in a column, if Step 3 is followed.

[Proof]: First we show that initially matrix H, as derived from matrix M, can not have two or more critical elements in a row or in a column. This is easily seen as initially $\sum_{i=1}^{q} h_{i,j} = \sum_{j=1}^{q} h_{j,i} = d$ for every row i and column j. Hence if $h_{m,n}$ is a critical element in row m and column n, then $h_{m,n} = d$, and it requires all other elements $h_{i,j} = 0$, where $i \neq m$ in column n, and $j \neq n$ in row m. Since the algorithm requires that critical elements be selected first and their corresponding rows and columns be out of further consideration, we are left with a square matrix H of size (q-c), where c is the number of critical elements initially present in H. And we have $H_i \geq 2$, $V_j \geq 2$ for all remaining rows i

and columns j. From this point on, an element may become critical only if a row with $H_i = 2$ or a column with $V_j = 2$ has one of its elements tagged.

Therefore, assume at some time t, two rows i and j have $H_i = H_j = 2$ with untagged non-zero elements in columns k and l. Now, if for any reason elements of one column, say k, are tagged thus causing $H_i = H_j = 1$, we will have two critical elements in the same column l. But this means an element $h_{r,k}$ in some other row r is being circled. Since we must have a square matrix before $h_{r,k}$ is circled, assume another column s which has a non-zero and untagged element only in row r (because $H_i = H_j = 2$). Notice that this also makes $H_r = 2$. But then $V_s = 1$ and $h_{r,s}$ is a critical element. Hence at this time t, we should have circled $h_{r,s}$, instead of $h_{r,k}$, according to the algorithm. And the assumed situation of having two critical elements in column l would not have happened. This situation is depicted in Fig. 4.10 (a).

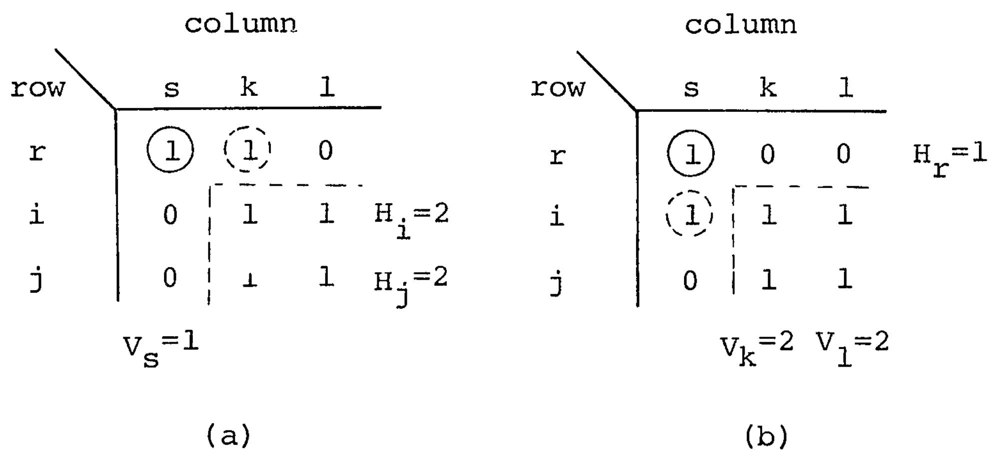


Fig. 4.10. Illustration of possible situations leading to two critical elements in a matrix H in same column l (a), or in same row j (b).

However, we note in part (a) of the above figure, if additional elements exist in some rows other than those depicted in a way such that each column has three or more elements, it is then necessary to look through only those rows which contain two elements in each row in order to pick $h_{r,s}$ correctly and avoid the creation of two critical elements in the same column.

A similar situation which might lead to two critical elements in the same row is shown in Fig. 4.10 (b). By the same argument, $h_{r,s}$ should have been circled instead of $h_{i,s}$. Hence the assumed situation can not occur.

Above reasoning can be extended to show that three or more critical elements can not be in the same row or column, thus completing the proof of the lemma. (Q.E.D.)

With above lemma, we can conclude the definitiveness of the algorithm by the following theorem.

Theorem 4.1:

The decomposition procedure of matrix H is definitive.

[Proof]: Lemma 4.1 guarantees successful selection of the circled elements in the construction of a permutation matrix from H without trial. Then as each permutation matrix is subtracted from H, the new (remaining) H-matrix still retains the property that each row and column sums up to the same value $(d - p)$, where p is the number of permutation matrices already completed $(p = t - 1)$. Hence further decomposition is always possible and definitive. (Q.E.D.)

4.3.2 Complexity, Optimal Factoring and Special-case Applications

We now look briefly at the operational complexity of the algorithm. The whole decomposition process is by no means a very simple one. However, all the steps, or variations of them, are basically required to generate the various permutation functions. Therefore, different algorithms will vary mainly in the ways that a composite matrix H, or its equivalent, is decomposed into the many unit permutation matrices, namely for what is performed in Step 3 of the given algorithm. A quick review then reveals that compared to a basic heuristic approach, which would require one to search in any arbitrary way for a non-zero element in a different row and in a different column at a time, the algorithm given here computes or counts the number of non-zero elements in each row and column before a search is initiated. Basically this is the only "extra" work this algorithm requires. Since all these computations are independent, they can be performed in a single step if such parallel processing can be done. The number of simultaneous computations starts at most from 2q, for a q x q H-matrix, and decreases by at least 2 with each element chosen. Thus, at the cost of q extra computation times, each unit permutation matrix is obtained in a definitive way, which may not be achievable with other heuristic approaches where repeated trials or more complicated corrective procedures are required after an unseccessful search.

It is also to be noted that all steps of the algorithm are well amenable to parallel processing with a degree of parallelism varying between d and N from step to step, as can be easily verified. For example, in Step 2 where we perform the intersection of A_i and B_j to form $m_{i,j}$, where $i, j = 1, .., q$ and $|A_i| = |B_j| = d$, the number of operations is in the order of $O(q \times d^2) = O(N \times d)$. However, the apparent parallelism available is at least $q \times d = N$. It is also found that the number of operations each step has is at most in the order of $O(Nd)$.

Next we consider the factoring of $N = d \times q$. In previous discussion we assumed this to be any possible but arbitrary one. However, it is possible to obtain an optimal factoring in terms of the minimum switching points required, which reflects the hardware cost. We now analyze this aspect.

Since a basic $d \times d$ switching element has a gate count or number of switching points proportional to d^2, we shall use this number as a basis for evaluating the overall system complexity. We also assume $q \geq d$ (so that if necessary or desired, the middle stage can be further decomposed). Therefore, a $d \times d$ permuter is the smallest switching element in the system.

Since there are q d-permuters in each of the I/O stages, and d q-permuters in the middle stage, the total number of equivalent switching points in a 3-stage RSN is, denoted by S, as follows:

$$S = 2 \times q \times d^2 + d \times q^2$$
$$= 2N \times d + N \times q$$
$$= 2Nd + N^2/ d. \qquad (4.1)$$

Differentiating S with respect to d, and setting the result to zero, we obtain a minimum in S if

$$2Nd^2 - N^2 = 0$$

Or, $$d = \sqrt{\frac{N}{2}}. \qquad (4.2)$$

With this optimal value of d, equation (4.1) gives

$$S_{min} = 2N\sqrt{\frac{N}{2}} + N^2/\sqrt{\frac{N}{2}} = 2N \cdot \sqrt{2N}. \qquad (4.3)$$

Equation (4.2) gives the optimal factoring of N in a 3-stage RSN. For example, if $N = 128$, then $d = 8$ and $q = 16$ result in a most economical system. For a value of N whose factoring is not exactly realizable by equation (4.2), one can choose a nearby value of d and compare the gate counts resulting from equations (4.1) and (4.3) to decide a near-optimal value. For instance, if $N = 100$, the optimal factor requires $d \approx 7$ with $S_{min} = 200 \cdot \sqrt{200}$ $= 2.8 \times 10^3$.

However if we use a factoring of $100 = 10 \times 10$, then from equation (4.1), $S_{(d=10)} = 200 \times 10 + 100^2/ 10 = 3 \times 10^3$, which is quite close to the theoretic minimum.

Finally, we consider the application of a 3-stage RSN in some special cases where the data connections assume some orderly patterns as described in Section 4.1. The purpose of this is to demonstrate that under these situations the permutation matrices are very simple and have some sort

of symmetry such that the application of the previous decomposition algorithm actually requires very little effort. For example, in all of the following cases, it is sufficient just to make row computations in an H-matrix in order to obtain the permutation matrices for middle stage blocks. We shall use a 16-port, 3-stage RSN with d = q = 4 for all cases. The formation of all matrices can be easily verified and is thus omitted. The dotted lines in H matrices indicate a collection of elements to make up one permutation matrix for a middle stage block.

(1) Skew-1 connection

$$M = \begin{bmatrix} (12,13,14) & \emptyset & \emptyset & (11) \\ (21) & (22,23,24) & \emptyset & \emptyset \\ \emptyset & (31) & (32,33,34) & \emptyset \\ \emptyset & \emptyset & (41) & 42,43,44) \end{bmatrix}$$

$$H = \begin{bmatrix} 3 & 0 & 0 & 1 \\ 1 & 3 & 0 & 0 \\ 0 & 1 & 3 & 0 \\ 0 & 0 & 1 & 3 \end{bmatrix}.$$

(2) $+\sqrt{N}$ linear shift

$$M = \begin{bmatrix} \emptyset & \emptyset & \emptyset & (11,12,13,14) \\ (21,22,23,24) & \emptyset & \emptyset & \emptyset \\ \emptyset & (31,32,33,34) & \emptyset & \emptyset \\ \emptyset & \emptyset & (41,42,43,44) & \emptyset \end{bmatrix}$$

$$H = \begin{bmatrix} 0 & 0 & 0 & 4 \\ 4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 4 & 0 \end{bmatrix}.$$

(3) Perfect shuffle

$$M = \begin{bmatrix} (11,13) & \emptyset & (12,14) & \emptyset \\ (21,23) & \emptyset & (22,24) & \emptyset \\ \emptyset & (31,33) & & (32,34) \\ \emptyset & (41,43) & \emptyset & (42,44) \end{bmatrix}$$

$$H = \begin{bmatrix} 2 & 0 & 2 & 0 \\ 2 & 0 & 2 & 0 \\ 0 & 2 & 0 & 2 \\ 0 & 2 & 0 & 2 \end{bmatrix}.$$

4.4 Summary

In this chapter we first described some of the data-connection requirements encountered in parallel processing for various purposes, and examined four potential data-connection networks which can realize all or part of these requirements. Although a cross-bar switch is by far the most popular choice due to its full connection capability, simple control, and fast connection , a multistage RSN allows some freedom in system design and represents a compromise between speed and cost while preserving all connection capability.

The main part of this chapter was then devoted to the description and proof of a new definitive decomposition algorithm for the control of a 3-stage RSN. This algorithm establishes simultaneous connection paths through all stages, and eliminates the iterative correcting procedures encountered in other previously known algorithms. The increase in the complexity of the algorithm is shown to be little beyond that of a heuristic approach, specially when parallel processing is used. Also the decomposition effort is greatly reduced when the required connection pattern has some regularity. This has been demonstrated by several examples.

An optimal choice on the size of the switching elements was also discussed in terms of minimum hardware complexity. The optimal size is $d = \sqrt{N/2}$.

5. CONCLUSION

In this thesis we have made a unified study on three major aspects which influence the design and performance of a parallel processing system, namely, parallelism exploitation, task scheduling, and data connection networks. In essence, these three phases of parallel processing are linked together by the flow of instructions and data. While proper treatment in the preprocessing phase on a problem helps to increase the possibility of simultaneous information flow, the actual performance is realized and controlled by some rearrangeable connection networks in the system. Our objective is to find more efficient and practical ways to achieve a faster computation in a parallel processing system.

In Chapter 2 we first laid the ground for discussion of trees and introduced Baer and Bovet's basic parsing algorithm to find parallel processable operations in an arithmetic expression in the form of a syntactic tree. This algorithm is important because of its simplicity and the minimum tree-height it produces, when distribution is not to be considered. We then considered the use of distribution to further reduce a tree height. Using a new direct approach, we presented a thorough discussion on this subject, from the simple case of equal operator weights to the general case of arbitrary operator times. Some binary operations were defined and algorithms developed to allow a systematic computation of tree heights directly in terms of their subtree heights

and available free nodes. The discussion was summed up with the description of distribution algorithms to reduce the tree heights. We next discussed the basis for different instructions of a program to be operationally independent. Specific treatment for different types of instructions was described.

In Chapter 3 we investigated some important bounds associated with the problem of scheduling. A new form for computing the lower bound in processing time was given. We then presented two major optimal scheduling algorithms and described two heuristic suboptimal alternatives. The heuristic approaches have been experimentally tested to produce optimal or near-optimal schedules in most cases, however we also gave a possible worst-case example as a caution against its careless application. The practical aspect of scheduling is reflected by a cost-performance analysis which was given to allow one to evaluate the overall merit of performing parallelism exploitation on a real system.

In Chapter 4 we first introduced some types of data-connection requirements more commonly encountered in a parallel processing environment, and some typical rearrangeable connection networks which may be used to realize all or part of these connections. Among the networks, a cross-bar switch and a multistage rearrangeable switching network (RSN) were seen to possess full connection capability. The control of a multistage RSN requires a proper decomposition process. For this purpose, we have presented a new algorithm. Its

definitive procedure is a distinguished feature different from other previously known algorithms. We proved its definitiveness and showed its complexity to have increased little beyond that of a heuristic approach. We also found an optimal size for the switching elements of a 3-stage RSN in order that the number of switching points be minimized. Finally, we demonstrated that the application of the algorithm could be simplified if the connection patterns assumed some regularity as found in most array processing.

LIST OF REFERENCES

[1] Baer, J.L. and Bovet, D.P., "Compilation of Arithmetic Expressions for Parallel Computations," Proc. IFIP, 1968.

[2] Batcher, K.E., "Sorting Networks and Their Applications," Proc. SJCC, 1968.

[3] Bell, C.G. and Newell, A., Computer Structure: Readings and Examples, McGraw Hill Book Co., 1971.

[4] Benes, V.E., Mathematical Theory of Connecting Networks and Telephone Traffic, New York: Academic Press, 1965.

[5] Bernstein, A.J., "Analysis of Programs for Parallel Processing," IEEE Trans. Comp., vol. EC-15, no. 5, Oct. 1966.

[6] Chen, C.J. and Frank, A.A., "On programmable Parallel Data Routing Networks via Cross-bar Switches for Multiple Element Computer Architectures," Proc. of the Sagamore Computer Conference, Aug. 1974.

[7] Clos, C., "A Study of Non-blocking Switching Networks," B.S.T.J., vol. 32, no. 2, Mar. 1953.

[8] Curtin, W.A., "Multiple Computer System," in Advances in Computers, edited by F. Alt, et al., vol. 4. Academic Press, New York, 1963.

[9] Davis, E., "Concurrent Processing of Conditional Jump-Trees," COMPCON Digest, 1972.

[10] Fernandez, E. and Bussell, B., "Bounds on the Number of Processors and Time for Multiprocessor Optimal Schedules," IEEE Trans. Comp., vol. C-22, no. 8, Aug. 1973.

[11] Graham, R.L., "Bounds on Multiprocessing Anomalies and Related Packing Algorithms," AFIPS, SJCC, vol. 40, 1972.

[12] Hobbs, L.C. and Theis, D.J., Parallel Processor Systems, Technologies, and Applications. Spartan Books, 1970.

[13] Hu, T.C., "Parallel Sequencing and Assemble Line Problems," Oper. Res., vol. 9, Nov. 1961

[14] Kaufman, M.T., "An Almost-Optimal Algorithm for the Assembly Line Scheduling Problem," IEEE Trans. Comp., vol. C-23, no. 11, Nov. 1974.

[15] Kraska, P., "Parallelism Exploitation and Scheduling," Ph.D. Dissertation, Univ. Illinois, Urbana, 1972.

[16] Kuck, D., Muraoka, Y., and Chen, S., "On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup," IEEE Trans. Comp., vol. C-21, no. 12, Dec. 1972.

[17] Kuck, D., et al., "Measurements of Parallelism in Ordinary Fortran Programs," Computer, Jan. 1974.

[18] Lawrie, D.H., "Memory-Processor Connection Networks," Ph.D. Dissertation, Univ. Illinois, Urbana, 1973.

[19] Lawrie, D.H., "Access and Alignment of Data in an Array Processor," IEEE Trans. Comp., vol. C-24, no. 12, Dec. 1975.

[20] Lehman, M., "A Survey of Problems and Preliminary Results Concerning Parallel Processing and Parallel Processors," Proc. IEEE, vol. 54, no. 12, Dec. 1966.

[21] Lorin, H., Parallelism in Hardware and Software, Prentice-Hall Inc., 1972.

[22] Muraoka, Y., "Parallelism Exposure and Exploitation in Programs," Ph.D. Dissertation, Univ. Illinois, Urbana, 1971.

[23] Murtha, J.C., "Highly Parallel Information Processing Systems" in Advances in Computers, edited by F. Alt, et al., vol. 7, Academic Press, New York, 1966.

[24] Neiman, V.I., "Structure Et Commande Optimales De Reseaux De Connexion Sans Blocage," Annales de Telecommunications, Jul./Aug. 1969.

[25] Opferman, D.C. and Tsao-Wu, N.T., "On a Class of Rearrangeable Switching Networks," B.S.T.J., vol. 50, no. 5, May-June 1971.

[26] Ramamoorthy, C.V. and Gonzalez, M.J., "A Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs," Proc. Fall Joint Computer Conference, 1969.

[27] Ramamoorthy, C.V., et al., "Compilation Techniques for Recognition of Parallel Processable Tasks in Arithmetic Expressions," IEEE Trans. Comp., vol. C-22, no. 11, Nov. 1973.

[28] Ramamoorthy, C.V., et al., "Optimal Scheduling Strategies in a Multiprocessor System," IEEE Trans. Comp., vol. C-21, no. 2, Feb. 1972.

[29] Ramanujam, H.R., "Decomposition of Permutation Networks," IEEE Trans. Comp., vol. C-22, no. 7, Jul. 1973.

[30] Sethi, R., "Algorithms for Minimal-Length Schedules" in Computer and Job-Shop Scheduling Theory, edited by E.G. Coffman, John Wiley and Sons, Inc. New York, 1976.

[31] Stone, H.S., "Parallel Processing with the Perfect Shuffle," IEEE Trans. Comp., vol. C-20, no. 2, Feb. 1971.

AUTOBIOGRAPHICAL STATEMENT

Terry T. Hsu was born in Tainan, Taiwan, on August 25, 1939. He graduated from National Taiwan University, Taipei, Taiwan, in Electrical Engineering in June 1962.

He started his graduate study at the University of Tennessee in Knoxville, Tennessee and received the degree of Master of Science in Electrical Engineering in December, 1966.

From April of 1967 to August of 1970 he was an elctronics engineer with Pocket Fone and Cro-Med Bionics, both divisions of American Chromalloy Corporation in New York City.

Since September, 1970, he has been in the Ph.D. program at the City University of New York, and on the instructional staff of the Electrical Engineering Department of the City College of New York.

Mr. Hsu is a member of the Institute of Electrical and Electronics Engineers and its Computer Society.