Report No. 424

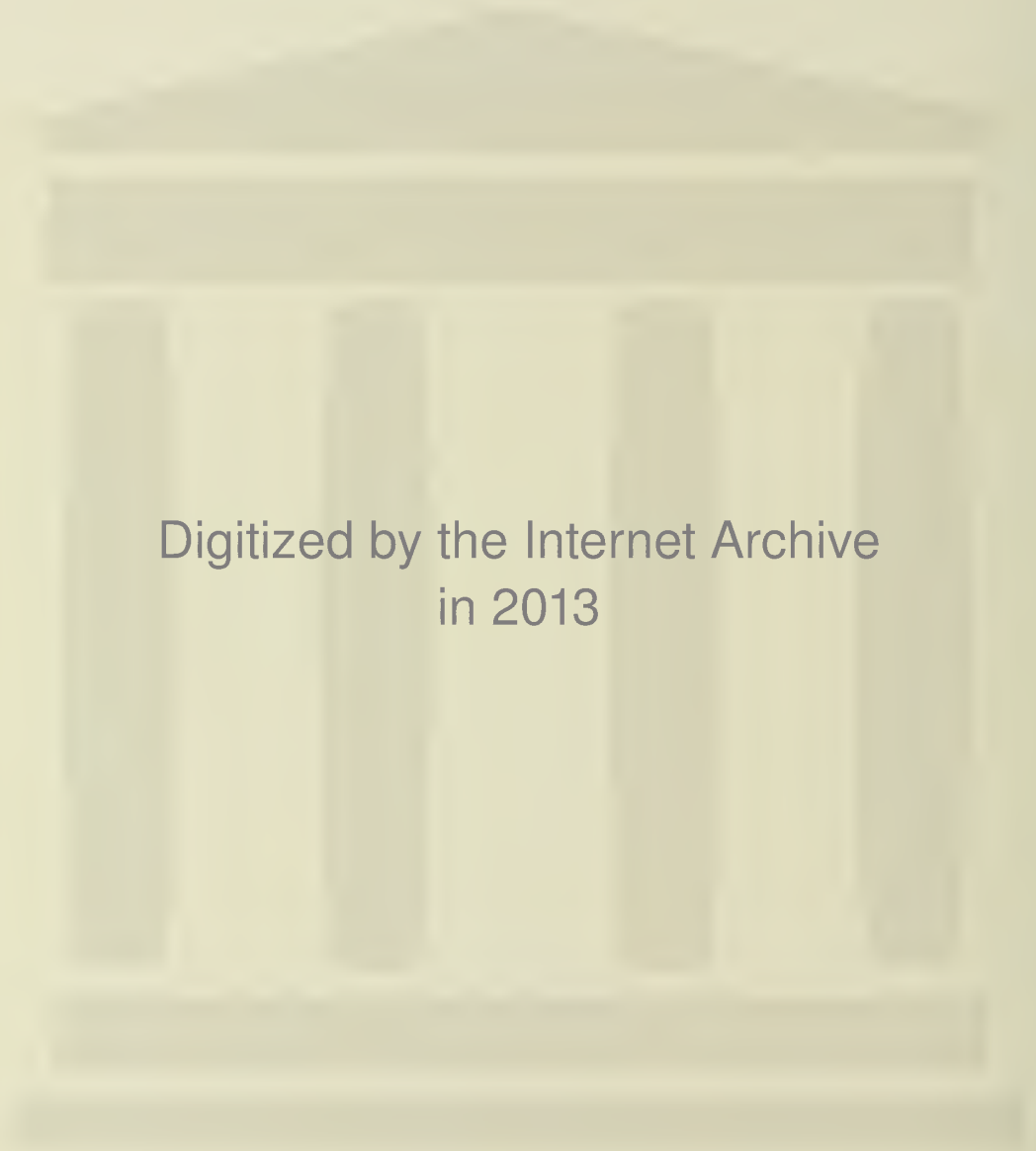
PARALLELISM EXPOSURE AND EXPLOITATION IN PROGRAMS

by

Yoichi Muraoka

February, 1971

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS



PARALLELISM EXPOSURE AND EXPLOITATION IN PROGRAMS

BY

YOICHI MURAOKA
B.Eng., Waseda University, 1965
M.S., University of Illinois, 1969

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1971

Urbana, Illinois

ACKNOWLEDGEMENT

The author would like to express his deepest gratitude to Professor David J. Kuck, the Department of the Computer Science of the University of Illinois, whose encouragement and good advice have led this work to the successful completion. Also Paul Kraska read the thesis and provided valuable comments.

Special thanks should go to Mrs. Linda Bridges without whose excellent job of typing, the final form would have never come out. Thanks are also extended to Mrs. Diana Mercer who helped in getting the thesis finished on time.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

1. INTRODUCTION

1.1 Introduction

The purpose of this research is to study compiling techniques for parallel processing machines.

Due to remarkable innovations of technology today such as the introduction of LSI, it has become feasible to introduce more hardware into computer systems to attain otherwise impossible high speeds. For example Winograd [42] showed that the minimum amount of time required to add two t bit numbers is $\lceil \log_2 t \rceil d$ ($\lceil x \rceil$ denotes the smallest integer not smaller than x), where we assume that an adder consists of two input binary logic elements, e.g. AND or OR gates and d is a delay time per gate. An adder which realizes this speed requires a huge number of gates, e.g. approximately 1300 gates for t = 6 [12], and it has been out of the question to build such an adder. However, the introduction of LSI has reduced the cost of a gate significantly, e.g. it has been anticipated that by 1974 the cost of LSI would be reduced to 0.7 cent per gate [33]. Another example is a class of parallel processing machines. The Illiac IV [7], the CDC 6600 [4] and the D825 [41] are included in this class. A machine in this class has e.g. many arithmetic units to allow simultaneous execution of arithmetic operations. As an extreme case it has been suggested to include special arithmetic units, e.g. a log taking unit (ln x) and an exponent unit (x^n) [16]. (Such being the case, this decade may be marked as a "computer architecture" race, reminiscent of the cycle-time and multiprogramming races of the 60's [12].) We shall not go into the details of machines further. An extensive survey of parallel processing machines is found in [30].

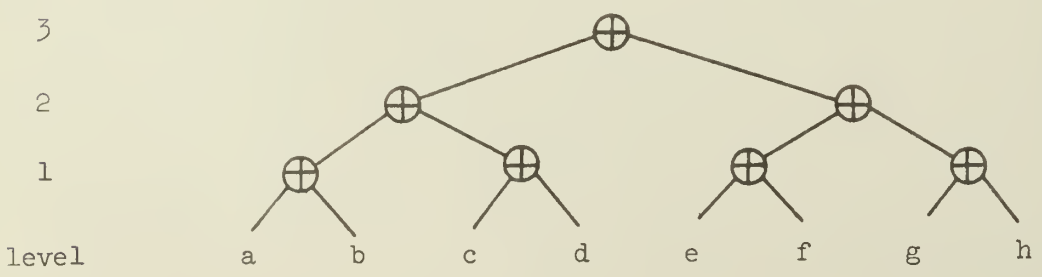
Having a parallel processing machine which is capable of processing many operations simultaneously, we are faced with the problem of exploiting parallelism from a program so that computational resources be kept as busy as possible to process the program in the shortest time. We now discuss the problem in detail.

In this thesis, by a parallel (processing) machine P we understand a set of arbitrarily many identical elements called processing elements (PE). A PE is assumed to be capable of performing any binary arithmetic operations, e.g. addition and multiplication, in the same amount of time. Furthermore we assume that data can be transferred between any PE's instantaneously. Also we write P(m) if P has only m PE's. A machine of this nature may be considered as a generalization of the Illiac IV.

To date two types of parallelism exploitation techniques are known to compile a program written in a conventional programming language (e.g. ALGOL) for the parallel processing machine [36]. They may be termed as intra-statement parallelism and inter-statement parallelism exploitation techniques. The first technique is to analyze the parallelism which exists within a statement, e.g. an arithmetic expression and this has been explored by Stone [40], Squire [39], Hellerman [20], and Baer and Bovet [6]. For example consider the arithmetic expression:

a + b + c + d + e + f + g + h

and a syntactic tree for it:



The tree is such that operations on the same level may be done in parallel. The height of a tree is the maximum level of the tree and indicates the number of steps required to evaluate an arithmetic expression in parallel. Note that there may be many different syntactic trees for an arithmetic expression, and among them the tree with the minimum height should be chosen to attain the minimum parallel computation time. Baer and Bovet's algorithm is claimed to achieve this end, i.e. build the minimum height syntactic tree for an arithmetic expression [6].

Exploitation of inter-statement parallelism has also been studied [10], [37]. An outcome of these works is an algorithm (the dependence relation detection algorithm [10]) which detects the dependence relation between statements in a loop- and jump-free sequence of statements. The dependence relation between S and S' holds if S proceeds S' in a sequence and S' uses the output of S as an input to S'. For example the algorithm dects that the statement S1 in Figure 1.1 must be computed before the statement S2, but it may be computed simultaneously with S3.

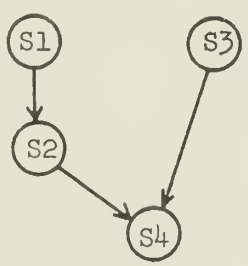
S1: $x := f_1(y);$

S2: $u := f_2(x);$

S3: $v := f_3(w);$

S4: $z := f_4(v,u);$

(a) program



(b) dependence relation

Figure 1.1. Statement Dependence Relation

Since in a real program the major part of the execution time is spent within loops if it is executed sequentially, the major effort should be directed toward detecting inter-statement parallelism in loops. For example we would like to find

out that all fifty statements, A[1] := f(B[1]), ..., A[50] := f(B[50]), in a loop

```
E:  for I := 1 step 1 until 50 do
        A[I] := f(B[I])
```

may be executed simultaneously to reduce the computation time to one fiftieth of the original. A technique available now which detects inter-statement parallelism inside a loop requires a loop to be first replaced with (expanded to) a sequence of statements, e.g. E in the above example must be replaced with the sequence of fifty statements, A[1] := f(B[1]), ..., A[50] := f(B[50]), so that the dependence relation detection algorithm can be applied [10]. Obviously this approach obscures an advantage of the introduction of loops into a program because essentially all loops are required to be removed from a program and replaced with straight-line programs so that the dependence relation detection algorithm can be applied on them.

The techniques described above find out parallelism inside and between statements as they are presented. If the size of a machine (i.e. the number of PE's) is unlimited, however, then it becomes necessary to exploit more parallelism from a program than the above approaches provide. One obvious strategy is to write a completely new program using e.g. parallel numerical methods [32], [38]. The other approach which we will pursue here is to transform a given program to "squeeze" more parallelism from it. While the first approach requires programmers (or users) to reanalyze problems and reprogram, the second approach tries to accept existing sequential programs written in e.g. ALGOL and execute them in parallel. First we study parallel computation of an arithmetic expression more carefully along this line.

For the sake of argument let us assume that an arithmetic expression consists of additions, multiplications and possibly parentheses. Then the associative, the commutative and the distributive laws hold. The first and second

laws have been already used to exploit more parallelism from an arithmetic expression. For example the associative law allows one to compute the arithmetic expression a + b + c + d as ((a+b) + (c+d)) in two steps rather than as (((a+b)+c)+d) which requires three steps.

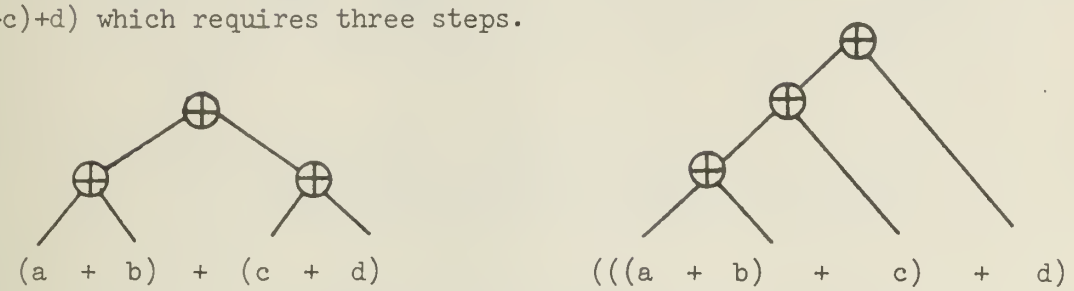


Figure 1.2. Trees for ((a+b)+(c+d)) and (((a+b)+c)+d)

Also it has been recognized that the commutative law together with the associative law gives a lower height tree. For example ((a + b × c) + d) requires three steps while (b × c + (a + d)) requires two [39].

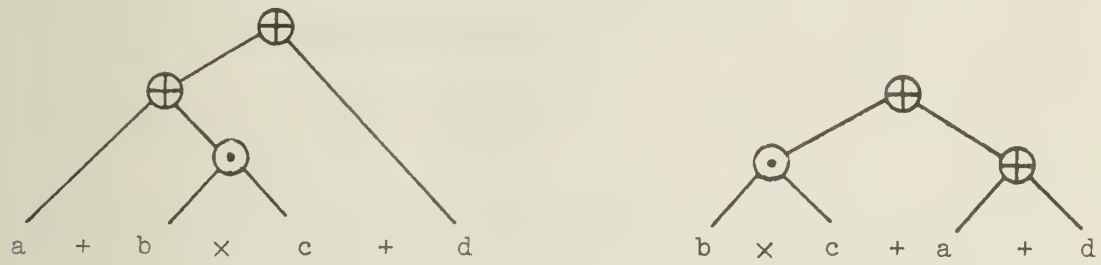


Figure 1.3. Trees for a + b × c + d and b × c + a + d

Now we turn our interest to the third law, i.e. the distributive law and see if it can help speeding up computation. As we can readily see there are cases when distribution helps. For example a(bcd + e) requires four steps while its equivalent abcd + ae which is obtained by distributing a over bcd + e can be computed in three steps.

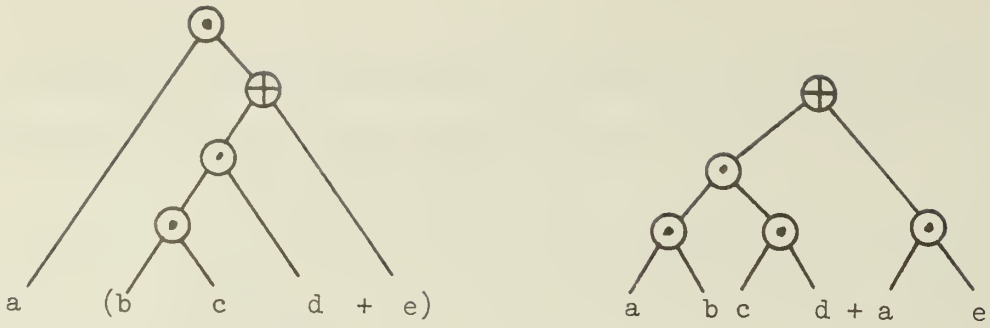


Figure 1.4. Trees for a(bcd+e) and abcd + ae

However, distribution does not necessarily always speed up computation. For example the undistributed form ab(c+d) can be computed in fewer steps than the distributed form abc + abd. Hence nondiscriminative distribution is not the solution to the problem. Chapter 3 of this thesis studies this situation and gives an algorithm which we call the distribution algorithm. Given an arithmetic expression A the distribution algorithm derives the arithmetic expression A^d by distributing multiplications over additions properly so that the height of A^d (we write $h[A^d]$ for this) is minimized. The algorithm works from the innermost parenthesis level to the outermost parenthesis level of an arithmetic expression and requires only one scan through the entire arithmetic expression. Chapter 3 concludes by giving a measure of the height of the minimum height tree for A as well as A^d as a function of fundamental values such as the number of single variable occurrences in A.

The idea is extended to handle a sequence of assignment statements in Chapter 4. The distribution algorithm is applied on the arithmetic expression which is obtained by backsubstituting a statement into one another. Suppose we have a sequence of n assignment statements A_1, A_2, ..., A_n and we get the assignment statement A from this sequence by back substitution. If the

sequence is computed sequentially, i.e. one statement after another, but each statement is computed in parallel, then it will take $h[A_1] + h[A_2] + \ldots + h[A_n]$ steps to compute the sequence (where $h[A_i]$ is the height of the minimum height tree for A_i). Instead we may compute the back substituted statement A in parallel which requires $h[A]$ steps. Obviously $h[A_1] + \ldots + h[A_n] \geq h[A]$ holds. Chapter 4 discusses cases when the strict inequality in the above equation holds. The cases include iteration formulas such as $x_{i+1} := a \times x_i + b$.

Next we study inter-statement parallelism in terms of program loops. Chapter 6 first establishes a new algorithm which detects inter-statement parallelism in a loop. The algorithm is such that it only examines index expressions and the way index values vary in a loop to detect parallel computability. For example the algorithm checks index expressions I and I + 1 as well as the clause "I := 1 step 1 until 20" in the loop

```
for I := 1 step 1 until 20 do
    A[I] := A[I+1] + B
```

and detects that all twenty statements, A[1] := A[2] + B, ..., A[20] := A[21] + B, may be computed simultaneously. Thus it is not necessary to expand a loop into a sequence of statements as was required before to check inter-statement parallelism. In general, the amount of work (i.e. the time) required by the algorithm is proportional to the number of index expression occurrences in statements in a loop.

Having established the algorithm, Chapter 6 further introduces two techniques which help to exploit more inter-statement parallelism in loops. These are the introduction of temporary locations and the distribution of a loop. The second technique resembles the idea introduced in Chapter 3, i.e. reduction of tree height for an arithmetic expression by distribution. Let us write

I, J, K(S1, S2, S3)

for an ALGOL--like program

for I := i_1, i_2, ..., i_m do

for J := j_1, j_2, ..., j_n do

for K := k_1, k_2, ..., k_p do

begin S1; S2; S3 end.

Furthermore by e.g. [I,J], K(S1,S2,S3) we understand a loop#

for (I,J) := (i_1,j_1), (i_1,j_2), ..., (i_1,j_n), (i_2,j_1), ..., (i_n,j_n) do

for K := k_1, k_2, ..., k_p do

begin S1; S2; S3 end.

Then as in the case of arithmetic expressions we may establish the following:

(a) Association: Introduction of brackets, e.g. I,[J,K](S1,S2,S3).

(b) Commutation: Change of the order of I,J,K e.g. I,K,J(S1,S2,S3).

(c) Distribution: Distribution of I,J,K over S1,S2,S3, e.g.
I,J,K(S1),I,J,K(S2,S3).

Then while the associative law always holds, e.g. I,J,K(S) = [I,J],K(S), the commutative and the distributive laws do not necessarily hold for all loops, e.g. I,J(S) $\neq$ J,I(S) if I,J(S) represents a loop

for I := 1, 2, 3 do

for J := 1, 2, 3 do

A[I,J] := A[I+1,J-1].

#This is equivalent to a TRANQUIL expression [2]
for (I,J) seq $((i_1,j_1),(i_1,j_2),\ldots,(i_m,j_n))$ do.

In short, Chapter 6 shows that commutation indicates the possibility of computing a loop in parallel as it is and distribution indicates the possibility of introducing more parallelism into a program. For example if I,J,K(S) = K,I,J(S), then S can be computed simultaneously for all values of K while I and J vary sequentially. Next suppose a loop I(S1,S2) cannot be computed in parallel for all values of I. Then in a certain case it is possible to distribute and obtain two loops I(S1), I(S2) which are equivalent to the original loop, I(S1,S2), and execute each of two loops in parallel for all values of I separately. Chapter 6 gives an algorithm to distribute to attain this end.

The thesis, thus, introduces new techniques which transform a given program to expose hidden parallelism. All results in this thesis are also readily applicable to another type of machines, i.e. machines with a pipe-line arithmetic unit such as CDC STAR [18] (we regard this type of machines as a special type of parallel machines and call them serial array machines). Each stage of a pipe-line unit may be regarded as an independent PE in the sense that an operation being processed in one stage of a pipe-line unit must not depend on an operation being processed in a different stage. Hence exploiting parallelism results in busying many stages at once.

Two more chapters are included in this thesis to make it complete. Chapter 2 studies parallel computation of special cases of arithmetic expressions, e.g. powers and polynomials, in detail to give a measure of the power of a parallel processing machine.

As was mentioned before, unless specially mentioned, it will be assumed that there are a sufficient number of PE's available to perform the desired task. In reality, however, that may not be the case and non trivial scheduling problems

may arise. To give some insight to this problem Chapter 7 discusses a solution to the two processor-equally weighted job scheduling problem.

We conclude this chapter by defining the following symbols:

$\lceil x \rceil$ ... the smallest integer not smaller than x,

$\lfloor x \rfloor$ ... the largest integer not larger than x, and

$\lceil x \rceil_2$ ... the smallest power of 2 not smaller than x.

Also unless specified, the base of logarithms is assumed to be 2, e.g. log n is $\log_2 n$.

2. PARALLEL COMPUTATION OF SUMMATIONS, POWERS AND POLYNOMIALS

2.1 Introduction

In this chapter, we study the parallel computation of summations, powers and polynomials. We first assume that m processors (PE) are available. Then the parallel computation times for the summation ($\sum_{i=1}^{n} a_i$) and the power (x^n) evaluation are given as functions of m and n. The minimum time to evaluate $\sum_{i=1}^{n} a_i$ or x^n as well as the minimum number of PE's required to attain it is also derived.

Polynomial computation is first studied assuming the availability of an arbitrary number of PE's. The lower bound on the computation time for a polynomial of degree n ($p_n(x)$) is presented. A scheme which computes $p_n(x)$ in lesser time than any known scheme is obtained. Because of its simplicity in scheduling, the k-th order Horner's Rule is studied further in detail. It is shown that for this algorithm the availability of more PE's sometimes <u>increases</u> the computation time.

Table 2.1 summarizes a part of results of this chapter.

Before we go further a few comments are in order. The base of logarithms in this chapter is 2, e.g. log n is actually $\log_2 n$. The following lemma will be frequently referred to in the text.

	h_{min}	$h(m,n)$	M
$\sum_{i=1}^{n} a_i$	$\lceil\alpha\rceil$	(1): $n - 1$ $(m = 1)$ (2): $\lfloor n/m \rfloor - 1 + \lceil \log_2 (m + n - \lfloor n/m \rfloor m) \rceil$ $(m \geq 2)$	(1): $1 + \lfloor (n-1)/2 \rfloor - 2^{k-2}$ $(2^k < n \leq 2^k + 2^{k-1})$ (2): $n - 2^k$ $(2^k + 2^{k-1} < n \leq 2^{k+1})$
x^n	$\lceil\alpha\rceil$	$\lfloor\alpha\rfloor + N - 1$ * $(m = 1)$	2
all x^i $(1 \leq i \leq n)$	$\lceil\alpha\rceil$	(1): $n - 1$ $(m = 1)$ (2): $\lfloor\beta\rfloor + 1 + \lceil (n - 2^{\lfloor\beta\rfloor} + 1)/m \rceil$ $(m \geq 2)$	(1): $n - 2^{\lceil\alpha\rceil - 1}$ $(n - 2^{\lceil\alpha\rceil - 1} \geq 2^{\lceil\alpha\rceil - 2})$ (2): $\lceil (n - 2^{\lceil\alpha\rceil - 2})/2 \rceil$ $(n - 2^{\lceil\alpha\rceil - 1} < 2^{\lceil\alpha\rceil - 2})$
$p_n(x)$	$\lceil\alpha\rceil + 1 + \lceil \log_2 (n+1) \rceil$	(1): $2n$ $(m = 1)$ (2): $2\lceil\beta\rceil + 2\lfloor n/m \rfloor + 1$ $(m \geq 2)$	(1): $n + 1$ $(n = 2^g)$ (2): $\lceil (n+1)/3 \rceil$ $(2^g < n \leq 2^g + 2^{g+1})$ (3): $\lceil (n+1)/2 \rceil$ $(2^g + 2^{g-1} \leq n < 2^{g+1})$

h_{min} : The minimum computation time for $f_n(x)$.

M : The number of PE's required to attain h_{min}.

$h(m,n)$: The computation time for $f_n(x)$ on $P(m)$.

$\alpha = \log_2 n$, $\beta = \log_2 m$. *) See Lemma 3.

Table 2.1. The Parallel Computation Time for Summation, Power and Polynomial

Lemma 1:

(1) $\lceil \log a \rceil - \lceil \log b \rceil \leq \lceil \log a - \log b \rceil$

where a and b are non zero positive integers.

(2) $\lceil a \pm b \rceil = \lceil a \rceil \pm b$,

$\lfloor a \pm b \rfloor = \lfloor a \rfloor \pm b$,

$\lceil b - a \rceil = b - \lceil a \rceil + 1$ and

$\lfloor b - a \rfloor = b - \lfloor a \rfloor - 1$

where b is a positive integer and a is a positive real number (not an integer);

(3) $a + 1 > \lceil a \rceil \geq a$ and

$b - 1 < \lfloor b \rfloor \leq b$

where a and b are non zero positive real numbers.

Proof:

(1) Let $a = 2^h + k$ and $b = 2^f + g$ where $0 \leq k \leq 2^h - 1$ and $0 \leq g \leq 2^f - 1$. Now the proof is divided into four cases, i.e. (i) k, $g > 0$, (ii) $k = g = 0$, (iii) $k = 0$, $g > 0$, and (iv) $k > 0$, $g = 0$.

(i): k, $g > 0$.

Then $\lceil \log a \rceil - \lceil \log b \rceil = h - f$. Also let $\log a = h + x$ and $\log b = f + y$ where $0 \leq x, y < 1$. Then $\lceil \log a - \log b \rceil \geq h - f$. Thus $\lceil \log a \rceil - \lceil \log b \rceil \leq \lceil \log a - \log b \rceil$.

Other three cases may be proved similarly and the details are omitted.

(2)(3) Proofs for (2) and (3) follow from the definition.

(Q.E.D.)

2.2 Summation of n Numbers

Theorem 1:

The minimum number of steps, $h^a(m,n)$, to add n numbers on P(m) is

$$h^a(m,n) = \begin{cases} (1): & n - 1 & (m = 1) \\ (2): & \lfloor n/m \rfloor - 1 + \lceil \log (m + n - \lfloor n/m \rfloor m) \rceil & \\ & & (\lfloor n/2 \rfloor > m \geq 2) \\ (3): & \lceil \log n \rceil & (m \geq \lfloor n/2 \rfloor) \end{cases}$$

Proof:

(1) is self evident. (3) uses the so-called log sum method [22] or the tree method (see Theorem 1 of Chapter 3). It is clear that $\lceil \log n \rceil$ steps are required and also that n numbers cannot be added in fewer steps.

Now we prove (2). First each PE adds $\lfloor n/m \rfloor$ numbers independently. This takes $\lfloor n/m \rfloor - 1$ steps and produces m partial sums. Then there will be $m + (n - \lfloor n/m \rfloor \cdot m)$ numbers left. Clearly $m + n - \lfloor n/m \rfloor \cdot m < 2m$. Then those numbers are added by the log sum method, which takes $\lceil \log(m + n - \lfloor n/m \rfloor m) \rceil$ steps.

(Q.E.D.)

Now we show that for a fixed n, $\lfloor n/2 \rfloor \geq m \geq m'$ implies that $h^a(m,n) \leq h^a(m',n)$. To prove this it is enough to show that $h^a(m + 1, n) \leq h^a(m,n)$ where $m + 1 \leq \lfloor n/2 \rfloor$. There are two cases:

(1) $\lfloor n/m \rfloor = \lfloor n/(m + 1) \rfloor = k \geq 1$.

Let $n = km + p$ $(p < m)$. Then

$$m + n - \lfloor n/m \rfloor m = m + p$$

and

$$(m + 1) + n - \lfloor n/(m + 1) \rfloor (m + 1) = m + p + 1 - k.$$

Hence we have

$$\lceil \log(m + n - \lfloor n/m \rfloor m) \rceil \geq \lceil \log((m + 1) - n - \lfloor n/(m + 1) \rfloor (m + 1)) \rceil, \text{ or } h^a(m,n) \geq h^a(m + 1,n).$$

(2) $\lfloor n/m \rfloor > \lfloor n/(m + 1) \rfloor$.

Let $\lfloor n/m \rfloor = k$ and $\lfloor n/(m + 1) \rfloor = k - g$, where $k, g \geq 1$. Then

$$n = km + p \qquad (p < m) \tag{1}$$

and $$n = (k - g)(m + 1) + p' \qquad (p' < m + 1). \tag{2}$$

Suppose $h^a(m,n) < h^a(m + 1,n)$, i.e.

$$\left\lfloor \frac{n}{m} \right\rfloor - 1 + \left\lceil \log(m + n - \left\lfloor \frac{n}{m} \right\rfloor m) \right\rceil < \left\lfloor \frac{1}{m+1} \right\rfloor - 1 + \left\lceil \log((m + 1) + n - \left\lfloor \frac{n}{m+1} \right\rfloor (m + 1)) \right\rceil. \tag{3}$$

By substituting Eq. (1) and (2) into Eq. (3) and by rearranging we get

$$g < \lceil \log(m + 1 + p') \rceil - \lceil \log(m + p) \rceil. \tag{4}$$

If we can prove

$$g < \lceil \log((m + 1 + p')/(m + p)) \rceil, \tag{5}$$

then by Lemma 1(1), we can prove Eq. (4). Eq. (5) holds if

$$2^g < (m + 1 + p')/(m + p) \tag{6}$$

holds. Since

$$2 + 1/m \geq (m + 1 + p')/(m + p), \tag{7}$$

Eq. (6) holds if and only if $g = 1$ (remember that $g \geq 1$). By letting $g = 1$ in Eq. (2), we get

$$p' = km + p - (k - 1)(m + 1).$$

Then by substituting this and $g = 1$ into Eq. (6) and by proper

rearrangement, we get

$$p < 2 - k.$$

This only holds if $k = 1$ and $p = 0$, which implies that $n = m$ (see Eq. (1)) and contradicts our assumption that $m + 1 \leq \lfloor n/2 \rfloor$.

Hence $h^a(m,n) < h^a(m+1,n)$ never holds.

The above two cases (1) and (2) prove that $h^a(m+1,n) \leq h^a(m,n)$. Thus we have the following lemma.

Lemma 2:

$h^a(m',n) \geq h^a(m,n)$ if $m' < m \leq \lfloor n/2 \rfloor$.

The above lemma may seem insignificant. In Section 2.4.2, however, it will be shown that for a certain algorithm to compute an n-th degree polynomial on P(m), the computation time step, $h^p(m,n)$, is not a nonincreasing function of m, i.e., $m \geq m'$ does not necessarily imply that $h^p(m,n) \leq h^p(m',n)$. As it will be described later, the algorithm is such that all PE's are forced to participate in the computation. It is true that if we are allowed to "turn off" some PE's, then we always get $h^p(m,n) \leq h^p(m',n)$ if $m \geq m'$. Then a question is how many PE's are to be turned off. These problems will be studied in Section 2.4.2.

It should be noted that the minimum number of PE's required to achieve the minimum computation time is not necessarily $\lfloor n/2 \rfloor$. For example let $n = 17$. Then $\lfloor 17/2 \rfloor = 8$ and $h^a_{min}(17) = h^a(8,17) = 5$. But also $h^a(6,17) = 5$.

As we know, the minimum computation time to add n number is $\lceil \log n \rceil$. Now we present the minimum number of PE's, M, which achieves this bound.

Theorem 2:

For a fixed n, let

$$M = \mu_m \left(\lfloor n/m \rfloor - 1 + \lceil \log(m + n - \lfloor n/m \rfloor m) \rceil = \lceil \log n \rceil \right)^{\#}$$

Then

$$M = \begin{cases} (1) & 1 + \lfloor (n-1)/2 \rfloor - 2^{k-2} & (2^k < n \leq 2^k + 2^{k-1}) \\ (2) & n - 2^k & (2^k + 2^{k-1} < n \leq 2^{k+1}) \end{cases}$$

where $k = \lfloor \log(n-1) \rfloor$.

Proof:

For $k \leq 3$, the direct examination shows that the theorem holds (see Table 2.2). Therefore we assume that $k > 3$.

The proof is divided into two parts, i.e. (1) $2^k < n \leq 2^k + 2^{k-1}$ and (2) $2^k + 2^{k-1} < n \leq 2^{k+1}$. In either case we first prove that $h^a(M,n) = \lceil \log n \rceil$. Then for $m < M$ we show that $h^a(m,n) > \lceil \log n \rceil$.

It should be clear that in both cases $2^k < n \leq 2^{k+1}$ and $\lceil \log n \rceil = k + 1$.

(1) $2^k < n \leq 2^k + 2^{k-1}$

Let $n = 2^k + p$. $(1 \leq p \leq 2^{k-1})$ (8)

Now we first show that $h^a(m,n) = \lceil \log n \rceil$ where

$$M = 1 + \lfloor (n-1)/2 \rfloor - 2^{k-2} \tag{9}$$

μ_m(condition) denotes the minimum value of m which satisfies the condition.

		m							
		1	2	3	4	5	k	M	Case
	2	1					0	1	2
	3	2	2				1	1	1
	4	3	2				1	2	2
n	5	4	3	3	3		2	2	1
	6	5	3	3	3		2	2	1
	7	6	4	3	3		2	3	2
	8	7	4	4	3		2	4	2
	9	8	5	4	4	4	3	3	1
	10	9	5	4	4	4	3	3	1

(1): $2^k < n \leq 2^k + 2^{k-1}$

(2): $2^k + 2^{k-1} < n \leq 2^{k+1}$

where $k = \lfloor \log(n-1) \rfloor$.

n: The degree of a polynomial

m: The number of PE's

M: The minimum number of PE's

Table 2.2. The Number of Steps Required to Compute $\sum_{i=1}^{n} a_i$ on P(m), $h^a(m,n)$, for $n \leq 10$.

Now

$$h^a(m,n) = \lfloor n/M \rfloor - 1 + \lceil \log(M-n-\lfloor n/M \rfloor M) \rceil . \qquad (10)$$

We then show that $\lfloor n/M \rfloor = 3$ for all n. Then we get $(M - n - \lfloor n/M \rfloor M) = n - 2M$. The value of $n - 2M$ is evaluated in three ways, i.e. (i) $p = 2g$, (ii) $p = 2g + 1$ $(g \geq 1)$ or (iii) $p = 1$. In any case $n - 2M \leq 2^{k-1}$ holds. Thus we can prove that

$$\lceil \log (M - n - \lfloor n/M \rfloor M) \rceil \leq k - 1.$$

and

$$h^a(M,n) \leq k + 1 = \lceil \log n \rceil .$$

Then we prove that for all $m < M$, $h^a(m,n) > \lceil \log n \rceil$. This is proved by showing that $h^a(M - 1, n) > \lceil \log n \rceil$. Then by Lemma 2, $h^a(m,n) > \lceil \log n \rceil$ for all $m < M$.

Now let us show the details. First we prove that $h^a(M,n) = \lceil \log n \rceil$. From Eq. (8) and (9), we get

$$M = 2^{k-2} + 1 + \lfloor (p - 1)/2 \rfloor, \qquad (11)$$

and by Lemma 1(2), we get

$$\left\lfloor \frac{n}{M} \right\rfloor = 3 - \lfloor P \rfloor \qquad (12)$$

where

$$P = \frac{4\lfloor (p - 1)/2 \rfloor + 4 - p}{2^{k-2} + 1 + \lfloor (p - 1)/2 \rfloor}.$$

By Lemma 1(3), we get

$$P \leq P' = \frac{2p - 4}{2^{k-1} + p - 1}.$$

Now we show that for all $p(1 \leq p \leq 2^{k-1})$, $P' < 1$. Since

$$\frac{\partial}{\partial p}P' = \frac{2^k + 2}{(2^{k-1} + p - 1)^2} > 0$$

we have

$$\max P' = \frac{2^k - 4}{2^k - 1} < 1 \quad \text{for } 1 \le p \le 2^{k-1} \ (k \ge 4).$$

Thus $P < 1$ and by Eq. (12) we have $\lfloor n/M \rfloor = 3$. Substituting this into Eq. (10), we get $h^a(M,n) = 2 + \lceil \log(n - 2M) \rceil$. Now subtracting two times Eq. (11) from Eq. (8) we have

$$n - 2M = 2^{k-1} + p - 2 - 2\lfloor (p - 1)/2 \rfloor. \qquad (13)$$

Eq. (13) is evaluated in three different ways according to the value of p, i.e. (i) $p = 2g$, (ii) $p = 2g + 1$ $(g \ge 1)$ or (iii) $p = 1$ (in every case g is an integer).

(i) $p = 2g$ (From Eq. (1), $g > 1$).

$$n - 2M = 2^{k-1}.$$

(ii) $p = 2g + 1$, $(g \quad 1)$.

$$n - 2M = 2^{k-1} + 2g + 1 - 2 - 2g < 2^{k-1}.$$

(iii) $p = 1$.

$$n - 2M = 2^{k-1} + 1 - 2 < 2^{k-1}.$$

Hence in any case $n - 2M \le 2^{k-1}$ or $\lceil \log(n-2M) \rceil \le k - 1$. Thus

$$h^a(M,n) \le k + 1 = \lceil \log n \rceil.$$

This proves the first part of (1). Next we prove the latter part, i.e. for $m < M$, $h^a(m,n) > h^a(M,n)$.

First we show that $h^a(M - 1, n) > h^a(M, n)$. From Eq. (8) and (9), and using Lemma 1(2) we get

$$\left\lfloor \frac{n}{M - 1} \right\rfloor = \lfloor 4 - Q \rfloor = 3 - \lfloor Q \rfloor, \tag{14}$$

where

$$Q = \frac{4\lfloor (p - 1)/2 \rfloor - p}{2^{k-2} + \lfloor (p - 1)/2 \rfloor}.$$

As we showed for P, we can also prove that for all p ($1 \leq p \leq 2^{k-1}$), $Q < 1$. From Eq. (14), we have $\lfloor n/(M - 1) \rfloor = 3$. Then $h^a(M - 1, n) = 2 + \lceil \log(n - 2(M - 1)) \rceil$. From Eq. (8) and (11), we get

$$n - 2(M - 1) = 2^{k-1} + p - 2\lfloor (p - 1)/2 \rfloor > 2^{k-1} + 1$$

or $\lceil \log(n - 2(M - 1)) \rceil \geq k$. Hence

$$\lceil \log n \rceil = k + 1 < k + 2 \leq h^a(M - 1, n).$$

Thus for all $m < M$, $h^a(m,n) > \lceil \log n \rceil$ by Lemma 2, and this proves (1).

(2) $2^k + 2^{k-1} < n \leq 2^{k+1}$.

Let $n = 2^k + 2^{k-1} + p$ ($1 \leq p \leq 2^{k-1}$). (15)

We first sketch the proof. We first show that $h^a(M,n) = \lceil \log n \rceil$. To show this we prove that $\lfloor n/M \rfloor = 2$ for all n ($2^k + 2^{k-1} < n \leq 2^{k+1}$). Then using this we get $M + n - \lfloor n/M \rfloor M = n - M$. We further show that $n - M = k$. Thus we get $h^a(M,n) = k + 1 = \lceil \log n \rceil$. Then we prove that $h^a(M - 1, n) > h^a(M, n)$ which together with Lemma 2 completes the proof of (2).

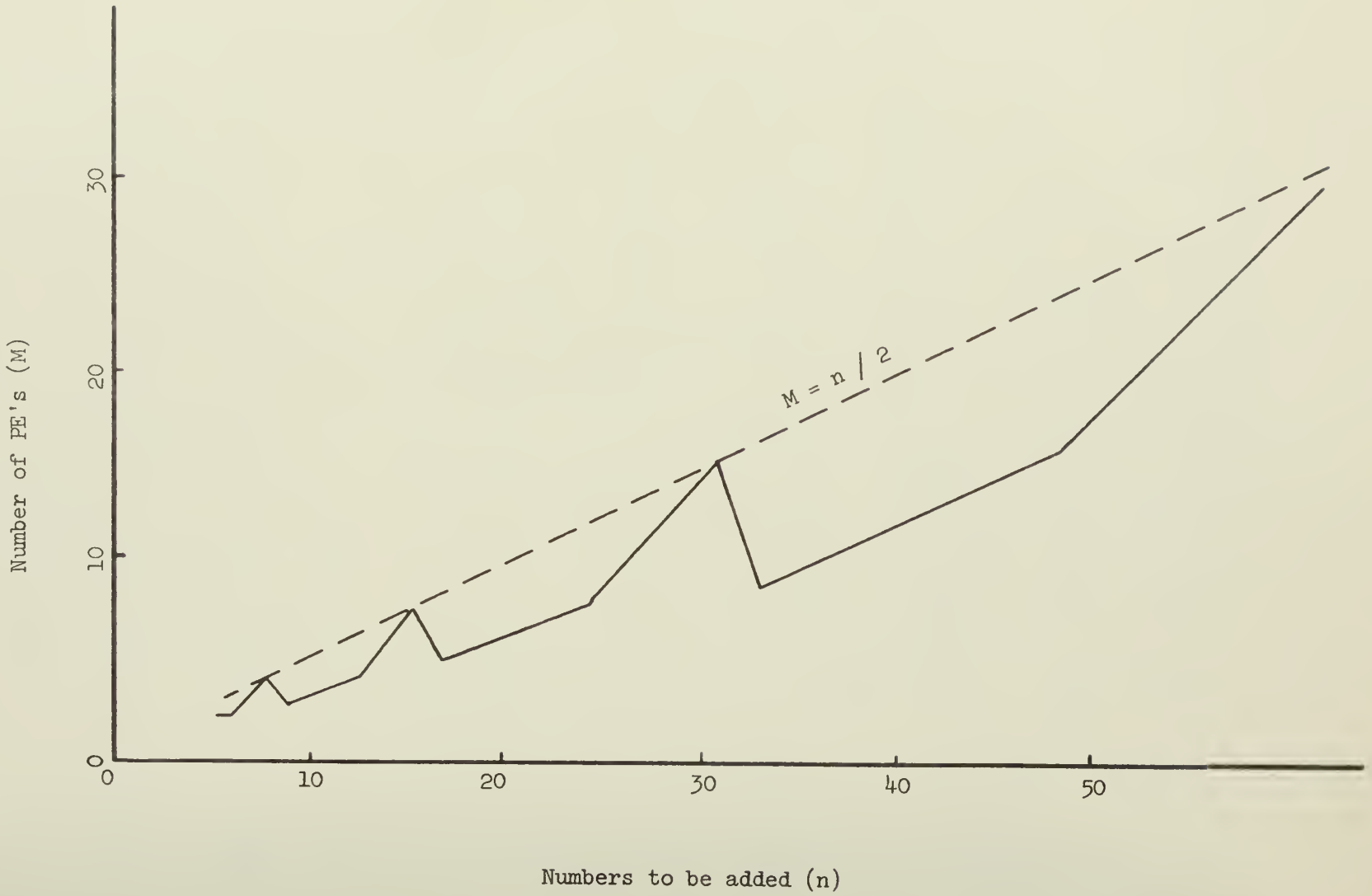


Figure 2.1. The Minimum Number, M, of PE's Required to Add Numbers in the Minimum Time

The details are similar to (1) and will not be given here.

(Q.E.D.)

Theorems 1, 2 and Lemma 2 also apply to the case of multiplication of n numbers, and to avoid duplication, the corresponding lemmas for multiplication shall not be presented.

Next let us consider the power computation, e.g. $x^n (n \geq 2)$.

2.3 Computation of Powers

Lemma 3: [23]

Let N be the number of ones in the binary representation for n.

Then the near minimum number of steps to compute x^n on P(1), $h^e(1,n)$, is

$$h^e(1,n) = \lfloor \log n \rfloor + N - 1.$$

Up to now, there is no result about the minimum computation time to evaluate x^n [23]. Thus we shall settle for an approximation. For example let $n = 15$. Then $h^e(1,15) = \lfloor \log 15 \rfloor + 4 - 1 = 6$. On the other hand x^{15} can be evaluated in fewer steps, e.g.

$$(x)^2 = x^2 \rightarrow (x^2)(x) = x^3 \rightarrow (x^3)(x^2) = x^5$$

$$\rightarrow (x^5)^2 = x^{10} \rightarrow (x^5)(x^{10}) = x^{15}.$$

This takes only 5 steps. For $n \leq 70$, this lemma gives the correct values for more than 70% of the cases.

While we cannot give the definite answer for the sequential case, we can prove the following.

Theorem 3:

The minimum number of steps to compute x^n on P(m) (m > 1), $h^e(m,n)$, is

$$h^e(m,n) = \lceil \log n \rceil .$$

Proofs of Lemma 3 and Theorem 3:

Let $\alpha = \log_2 n$ and $\beta = \log_2(n+1)$ for convenience. Let I_j be the j-th most significant bit in the binary representation for n. If m = 1, then x^n is computed as follows. First let us write

$$X_j = (x^{2^j})^{I_{j+1}}.$$

We first compute all x^i ($i = 2^1, 2^2, \ldots, 2^{\lfloor \alpha \rfloor}$) in $\lfloor \alpha \rfloor$ steps. Then

$$x^n = (X_0) \times (X_1) \times \ldots \times (x^{2^{\lfloor \alpha \rfloor}})^{I_{\lceil \beta \rceil}}$$

and this computation takes N - 1 steps (note that if $I_j = 0$, then $X_{j-1} = 1$). Thus in total $\lfloor \alpha \rfloor + N - 1$ steps are needed.

If two PE's are used, then x^n is computed as follows. Again let us write

$$x^n = (X_0) \times (X_1) \times \ldots \times (x^{2^{\lfloor \alpha \rfloor}})^{I_{\lceil \beta \rceil}}.$$

Now this can be computed by the following two recursive equations.

$$t_0^{(1)} = x$$

$$t_0^{(2)} = 1$$

$$t_k^{(1)} = t_{k-1}^{(1)}\, t_{k-1}^{(2)}$$

$$t_k^{(2)} = t_{k-1}^{(2)} \left(t_{k-1}^{(1)}\right)^{I_k}$$

and

$$x^n = t_{\lceil \alpha \rceil}^{(2)}.$$

Two PE's are required for the simultaneous computation of $t_k^{(1)}$ and $t_k^{(2)}$.

That the above process for P(2) is optimum is clear, because $x^{2^{\lfloor \log n \rfloor}}$ cannot be computed in less than $\lfloor \log n \rfloor$ steps and at least $\lfloor \log n \rfloor + 1$ $(= \lceil \log n \rceil)$ steps are required to compute x^n.

(Q.E.D.)

From the above discussion, we have the following corollary.

Corollary:

$h^e(m,n) = h^e(2,n)$ for all $m > 2$.

Now let us study simultaneous computation of all x^i $(i=1,2,\ldots,n)$.

Theorem 4:

The minimum number of steps, $h^w(m,n)$, required for simultaneous evaluation of all x^i $(i=1,2,\ldots,n)$ on P(m) is

$$h^w(m,n) = \begin{cases} (1) \quad n-1 & (m=1) \\ (2) \quad \lfloor \log m \rfloor + 1 + \lceil (n - 2^{\lfloor \log m \rfloor + 1})/m \rceil & \\ \qquad (\max(n-2^{\lceil \log n \rceil - 1},\ 2^{\lceil \log n \rceil - 2}) > m \geq 2) & \\ (3) \quad \lceil \log n \rceil & (m \geq \max(n - 2^{\lceil \log n \rceil - 1},\ 2^{\lceil \log n \rceil - 2})). \end{cases}$$

Proof:

(1) is obvious. (3) is illustrated in Figure 2.2. At the k-th step, the x^i ($i = 2^{k-1} + 1, 2^{k-1} + 2, \ldots, 2^k$) are computed using the results of earlier steps, e.g. $x^{2^k} = x^{2^{k-1}} \times x^{2^{k-1}}$. The number of PE's required at this step is then $2^k - (2^{k-1} + 1) + 1 = 2^{k-1}$.

step \ PE	P_1	P_2	P_3	P_4					No. of PE's required
1	x^2								1
2	x^3	x^4							2
3	x^5	x^6	x^7	x^8					4
:			..	..					:
k	x^{a+1}	x^{a+2}	..	..	..	x^{2a}			2^{k-1}
:			..	..					
$\lceil \log n \rceil - 1$	x^{b+1}		..	..			x^{2b}		b
$\lceil \log n \rceil$	x^{c+1}		..	..			..	x^n	n - 2c

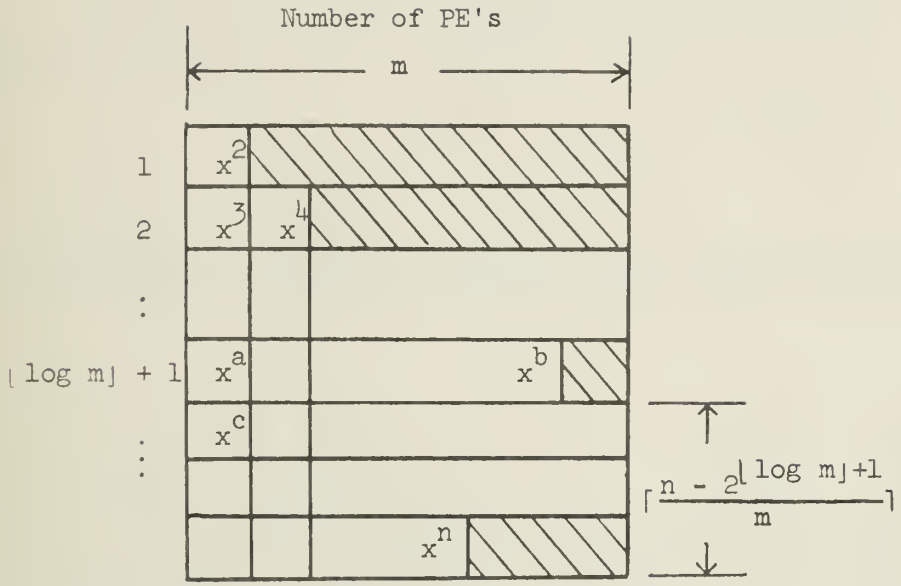
$a = 2^{k-1}$ $\quad b = 2^{\lceil \log n \rceil - 2}$

$c = 2b$

Figure 2.2. Computation of x^n (1)

The maximum number of PE's required is the larger one of $n - 2^{\lceil \log n \rceil - 1}$ (the number of PE's required at the last step) or $2^{\lceil \log n \rceil - 2}$ (the number of PE's required at the $(\lceil \log n \rceil - 1)$-th step). This proves (3). Clearly this procedure is optimum in the sense that it gives the minimum computation time.

Next suppose that the number of PE's available, m, is less than $\max(n - 2^{\lceil \log n \rceil - 1}, 2^{\lceil \log n \rceil - 2})$. Then first all $x^i (1 \leq i \leq 2^{\lfloor \log m \rfloor + 1})$ are computed in $\lfloor \log m \rfloor + 1$ steps in the same manner as the above procedure.



$$a = 2^{\lfloor \log m \rfloor} + 1$$

$$b = 2^{\lfloor \log m \rfloor + 1}$$

$$c = b + 1 = 2^{\lfloor \log m \rfloor + 1} + 1$$

Figure 2.3. Computation of x^n (2)

Now there are $n - 2^{\lfloor \log m \rfloor + 1}$ x^i left ($2^{\lfloor \log m \rfloor + 1} < i \leq n$) to be computed. This takes $\lceil (n - 2^{\lfloor \log m \rfloor + 1})/m \rceil$ steps on P(m). Clearly at each step, all necessary data to perform operations are available. To show this, let us take two successive steps. Assume that the first step computes $x^a \sim x^b$ where $b - a + 1 = m$. Then the second step computes $x^{b+1} \sim x^{b+m}$. Since $b + m = 2b + 1 - a \leq 2b$ ($a \geq 1$), all inputs required at the second step are available from the first step. Thus in total $\lfloor \log m \rfloor + 1 + \lceil (n - 2^{\lfloor \log m \rfloor + 1})/m \rceil$ steps are required. This proves (2).

(Q.E.D.)

Clearly for fixed n, $m \geq m'$ implies that $h^W(m,n) \leq h^W(m',n)$. Thus we have:

Lemma 4:

For fixed n, $h^W(m,n)$ is a non-increasing function of m.

We again call the reader's attention that the number of PE's required to compute all x^i in the minimum number of steps, i.e. $\lceil \log n \rceil$, is not necessarily $\max(n-2^{\lceil \log n \rceil - 1}, 2^{\lceil \log n \rceil - 2})$. For example, let n = 18. Then $\max(18 - 2^4, 2^3) = 8$, and $\lceil \log 18 \rceil = 5$. Yet P(5) achieves the same result, i.e. $h^W(5,18) = 5$.

Lemma 5:

The minimum number of PE's, M, necessary to compute all x^i ($1 \leq i \leq n$) simultaneously in the shortest time is

$$M = \begin{cases} (1) \quad n - 2^{\lceil \log n \rceil - 1} & (n - 2^{\lceil \log n \rceil - 1} \geq 2^{\lceil \log n \rceil - 2}) \\ (2) \quad \lceil (n - 2^{\lceil \log n \rceil - 2})/2 \rceil & (\text{otherwise}) \end{cases}$$

Proof:

Let $\alpha = \lceil \log n \rceil$.

(1) $n - 2^{\alpha-1} \geq 2^{\alpha-2}$.

Suppose that there are only m PE's where $m < n - 2^{\alpha-1}$.

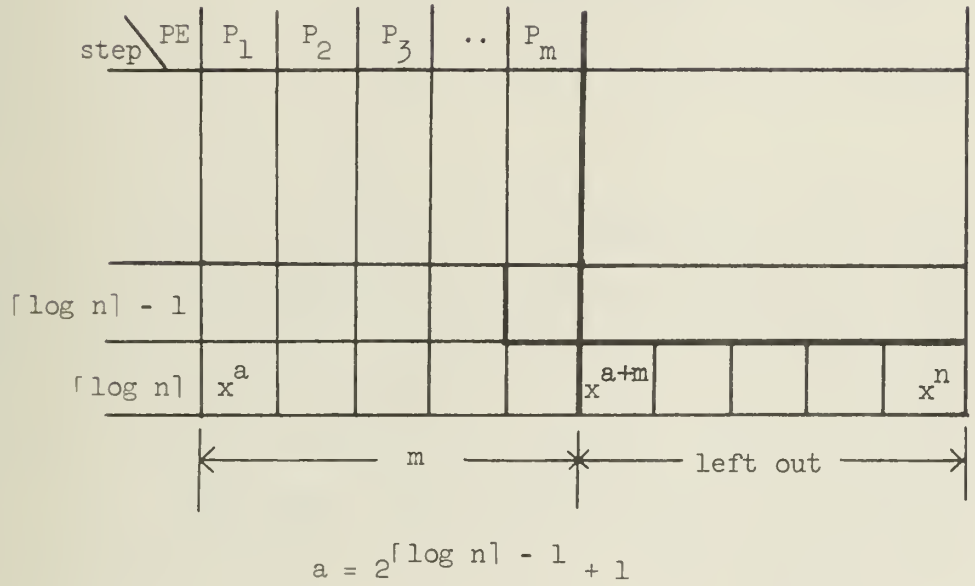


Figure 2.4. Computation of x^n (3)

Then $x^i (2^{\alpha-1} + 1 + m \leq i \leq n)$ cannot be computed at the α-th step. Also none of them can be computed at an earlier step because their inputs are not ready. This proves (1).

(2) $n - 2^{\alpha-1} < 2^{\alpha-2}$.

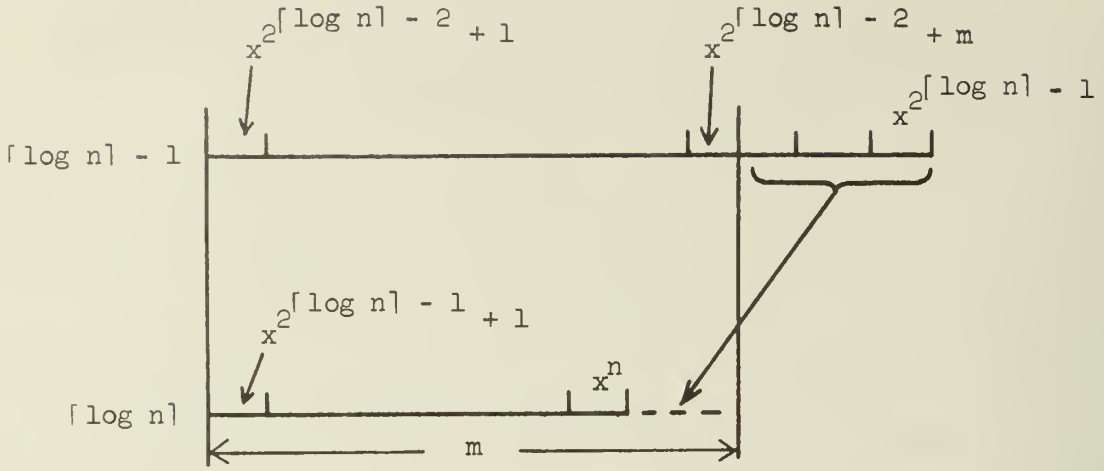


Figure 2.5. Computation of x^n (4)

Then we delay the computation of all $x^i (2^{\alpha-2} + m + 1 \leq i \leq 2^\alpha - 1)$ which are originally scheduled to be computed at the $(\alpha - 1)$-th step till the next step, i.e. the α-th step (i.e. the last step). At the α-th step we compute all remaining $x^i (2^{\alpha-2} + m + 1 \leq i \leq n)$. The value of m has been chosen in such a way that

$$(2^\alpha - 1) - (2^{\alpha-2} + m + 1) + 1 \geq n - (2^{\alpha-2} + m + 1) + 1$$

i.e.

$$m = \lceil (n - (2^{\alpha-2} + 1) - 1)/2 \rceil.$$

Also since

$$2(2^{\alpha-2} + m) \geq 2(2^{\alpha-2} + n/2 - 2^{\alpha-3})$$

$$= n + 2^{\alpha-2} \geq n,$$

all inputs required at the α-th step are ready at the $(\alpha - 1)$-th step or earlier steps.

(Q.E.D.)

2.4 Computation of Polynomials

In this section, we study polynomial computation. First we assume that there are arbitrarily many PE's. Then four schemes are studied and compared. Two of them are known as Estrin's method [16] and the k-th order Horner's rule [15]. Two new methods are also introduced. They are called a tree method (see Chapter 3) and a folding method. It is shown that if there are arbitrarily many PE's, then the folding method gives a faster computation time than any known method.

Then we study the case where only a limited number of PE's are available. Because of the simplicity of scheduling, the k-th order Horner's rule is studied in detail. It is shown that on P(m) the m-th order Horner's rule does not necessarily guarantee the fastest computation, i.e., there is a case where the m'-th order Horner's rule ($m' < m$) gives a better result. Thus availability of more computational resources does not necessarily "speed up" the computation for a certain class of feasible parallel computation algorithms.

2.4.1 Computation of a Polynomial on an Arbitrary Size Machine

Definition:

We write $p_n(x)$ for a polynomial of degree n

$$p_n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0.$$

2.4.1.1 k-th Order Horner's Rule [15]

The details will be presented in Section 2.4.2. Theorem 5 shows that the minimum time required to compute $p_n(x)$ by this method is

$$h^p_{min} = \lceil \log n \rceil + \lceil \log (n+1) \rceil + 1$$

2.4.1.2 Estrin's Method [15][16]

We first compute

$$c^0_i = a_i + xa_{i+1} \qquad i = 0, 2, \ldots, 2\lfloor n/2 \rfloor .$$

Then successively compute

$$c^1_i = c^0_i + x^2 c^0_{i+2} \qquad i = 0, 4, \ldots, 4\lfloor n/4 \rfloor$$

$$c^2_i = c^1_i + x^4 c^1_{i+4} \qquad i = 0, 8, \ldots, 8\lfloor n/8 \rfloor$$

$$\ldots$$

$$c^m_i = c^{m-1}_i + x^{2m} c^{m-1}_{i+2m} \qquad i = 0, 2^{m+1}, \ldots, 2^{m+1}\lfloor n/2^{m+1} \rfloor$$

where $m = \lfloor \log n \rfloor$ and more over

$$p_n(x) = c^m_0 .$$

The procedure may be illustrated by

$$\begin{aligned} p_n(x) = {} & a_0 + a_1 x + x^2(a_2 + a_3 x) \\ & + x^4(a_4 + a_5 x + x^2(a_6 + a_7 x)) \\ & + x^8(a_8 + a_9 x + x^2(a_{10} + a_{11} x) + x^4(a_{12} + a_{13} x + x^2(a_{14} + a_{15} x))) \\ & + \ldots . \end{aligned}$$

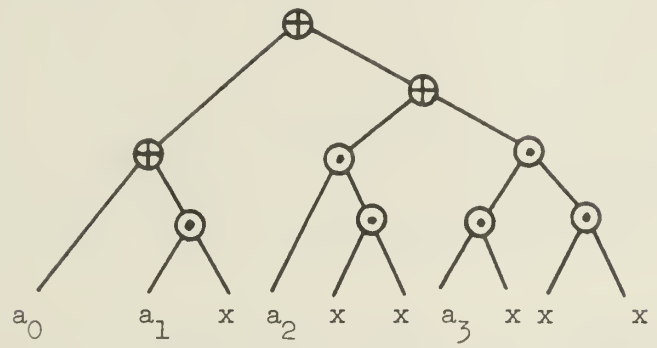
Now notice that for each j all c^j_i may be computed simultaneously.

Assuming the availability of an arbitrary number of PE's, it is easy to show that the minimum number of steps to compute $p_n(x)$ by this method is

$$h^E_{min} = 2\lceil \log (n + 1) \rceil .$$

2.4.1.3 Tree Method

In this method a tree similar to the one for arithmetic expressions is built for a polynomial $p_n(x)$. For example $p_3(x)$ is computed as:



Computation by this method consists of two stages, computation of $b_i = a_i x^i$ $(0 \leq i \leq n)$ and computation of $\sum_{i=0}^{n} b_i$.

(1) Computation of $b_i = a_i x^i$ $(0 \leq i \leq n)$.

This requires $\lceil \log(i+1) \rceil$ steps by Theorem 3 (see Figure 2.6).

(2) Computation of $\sum_{i=0}^{n} b_i$.

As soon as b_i's become available they are added in the log sum way. Suppose b_i becomes available at the k-th step. Having a variable at the k-th step is equivalent to having 2^{k-1} variables originally at the first step (cf. the effective length in Chapter 3).

Step	Terms Which Are Computed	No. of Terms Computed
1	a_0	1
2	a_1x	1
3	a_2x^2, a_3x^3	$2(=2^1)$
4	a_4x^4, a_5x^5, a_6x^6, a_7x^7	$4(=2^2)$
⋮	⋮	⋮
$\lceil \log(n+1) \rceil$		$2^{\lceil \log(n+1) \rceil - 2}$
$\lceil \log(n+1) \rceil + 1$	$a_{2^{\lceil \log(n+1) \rceil - 1}}x^{2^{\lceil \log(n+1) \rceil - 1}}, \ldots, a_nx^n$	$n-2^{\lceil \log(n+1) \rceil - 1}+1$

Figure 2.6. Computation of a_ix^i

Thus, for example, two variables at the third step are reduced to 8 variables at the first step. Repeating this procedure we get

$$n' = 1 + \sum_{i=1}^{\alpha-1} (2^{i-1}2^i) + 2^\alpha(n - 2^{\alpha-1} + 1)$$

$$= 1 + \sum_{i=1}^{\alpha-1} 2^{2i-1} + 2^\alpha(n - 2^{\alpha-1} + 1)$$

variables on the first level where

$$\alpha = \lceil \log(n + 1) \rceil .$$

To add these n' numbers by the log sum method, it takes

$$h^t_{min} = \lceil \log n' \rceil \text{ steps.}$$

2.4.1.4 Folding Method

This is a method which computes $p_n(x)$ in shorter time than any known method.

Assume that $p_{t-1}(x)$ can be computed in h - 1 steps, $p_i(x)$ $(t \leq i \leq s - 1)$ are computed in h steps and $p_s(x)$ can be computed in h + 1 steps.

Steps	Degree
h - 1	$\sim t - 1$
h	$t \sim s - 1$
h + 1	$s \sim (s + t - 1)$
h + 2	$s + t \sim$

Then we show that all $p_j(x)$ $(s \leq j \leq s + t - 1)$ can be computed in h + 1 steps and further $p_{s+t}(x)$ can be computed in h + 2 steps.

Proof:

(1) First we show that $p_{s+t-j}(x)$ $(1 \leq j \leq t)$ can be computed in h + 1 steps.

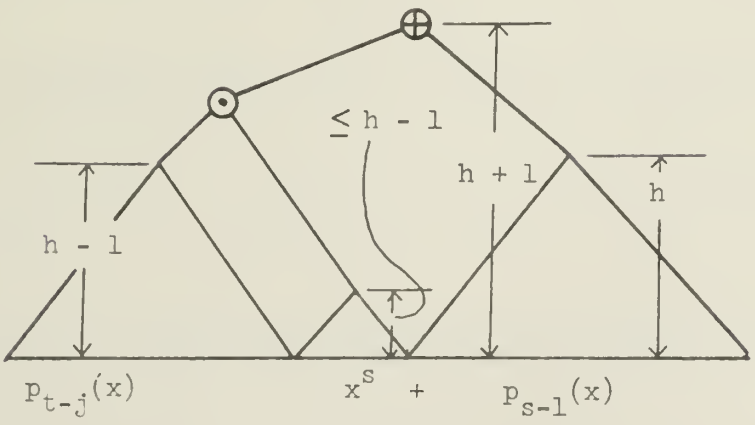


Figure 2.7. A Tree for $p_{s+t-j}(x)$

We write $p_{s+t-j}(x)$ as

$$p_{s+t-j}(x) = (a_{t+s-j}x^{t-j} + \ldots + a_s)x^s + a_{s-1}x^{s-1} + \ldots + a_o$$

$$= p_{t-j}(x)x^s + p_{s-1}(x).$$

Now we show that x^s can be computed in less than h steps.

From Theorem 1 we know that x^n can be computed in $\lceil \log n \rceil$ steps. Suppose that the computation of x^s takes longer than h, i.e.

$$h \leq \lceil \log s \rceil . \tag{16}$$

Now from the assumption, $p_{s-1}(x)$ takes h steps to compute. Also (see Section 2.4.1.5)

$$h \geq \lceil \log(2(s-1) + 1) \rceil = \lceil \log(2s-1) \rceil . \tag{17}$$

From Eq. (16) we have $s \geq 2^{h-1} + 1$ and from Eq. (17) we have $2^h \geq 2s - 1$ or $2^{h-1} \geq s$. Thus we have $s \geq 2^{h-1} + 1 > 2^{h-1} \geq s$ which is a contradiction. Thus $h > \lceil \log s \rceil$.

From the assumption $p_{t-j}(x)$ can be computed in h - 1 steps and $p_{s-1}(x)$ can be computed in h steps. Hence $p_{t-j}(x) \cdot x^s$ takes h steps and $p_{s+t-j}(x)$ can be computed in h + 1 steps.

(2) Next we show that $p_{s+t}(x)$ can be computed in h + 2 steps.

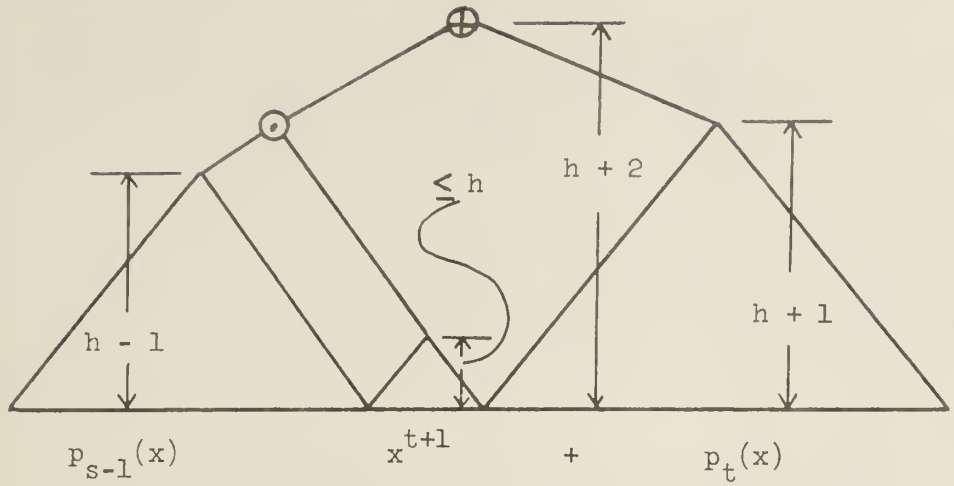


Figure 2.8. A Tree for $p_{s+t}(x)$

We write

$$p_{s+t}(x) = (a_{s+t}x^{s-1} + \ldots + a_{t+1})x^{t+1} + a_t x^t + \ldots + a_0$$

$$= p_{s-1}(x) \cdot x^{t+1} + p_x(x).$$

Then from the previous proof we know that x^{t+1} can be computed in less than h + 1 steps. Since $p_{s-1}(x)$ and $p_t(x)$ take h - 1 and h + 1 steps respectively, we can compute $p_{s-1}(x) \cdot x^{t+1}$ in at most h + 1 steps and $p_{s+t}(x)$ in at most h + 2 steps.

(Q.E.D.)

It is easy to check that $p_2(x)$ takes 3 steps, $p_3(x)$ and $p_4(x)$ can be computed in 4 steps and $p_5(x)$ can be computed in 5 steps. By induction we obtain the following table for h = 2, 3, ..., 10.

Minimum Steps	Degree of Polynomial
3	2
4	3 - 4
5	5 - 7
6	8 -12
7	13-20
8	21-33
9	34-54
10	55-87
11	89-143

Table 2.3. Computation of $p_n(x)$ by Folding Method

For example, $p_{35}(x)$ takes 9 steps to be computed. Note that the first numbers in the right column form a Fibonacci sequence.

2.4.1.5 Comparison of Four Methods

It has been proved that at least 2n operations are required to compute $p_n(x)$. Proofs appeared in several papers. We owe Ostrowski [34] and Motzkin [29] for their original works. Pan [35] summarized the results. An excellent review of the problem appears in [23]. Also Winograd [43] generalized results of Ostrowski and Motzkin.

Now assume that to compute $p_n(x)$ in parallel h steps are required. Then Theorem 5 gives $h \leq \lceil \log n \rceil + \lceil \log(n+1) \rceil + 1 \leq 2\lceil \log(n+1) \rceil + 1$. Also since $2^k - 1$ operations can be performed in a parallel computation tree of

height k, we have $2^h - 1 \geq 2n$ or $h \geq \lceil \log(2n+1) \rceil$. Thus

$$2\lceil \log(n+1) \rceil + 1 \geq h \geq \lceil \log(2n+1) \rceil.$$

In Figure 2.9, these upper and lower limits are plotted together with the results from the previous section. It is clear that the folding method is the best in terms of the computational speed. It is yet an open question if there is a better method.

2.4.2 Polynomial Computation by the k-th Order Horner's Rule

Now let us study computation of a polynomial by the k-th order Horner's rule.

A polynomial is computed by the k-th order Horner's rule as shown by the following procedure. We use k PE's. First we compute all $x^i (1 \leq i \leq k)$ simultaneously. Then we compute k polynomials $p_k'(x)$ on k PE's simultaneously, where

$$p_k'(x) = a_i + x^k(a_{i+k} + x^k(a_{i+2k} + \dots))) \qquad (0 \leq i \leq k - 1).$$

Then we get k partial results which are added to get $p_n(x)$. Figure 2.10 illustrates this.

This scheduling may not be the best, yet it is easy to implement and also adaptable to any number of k.

Theorem 5:

The minimum number of steps, $h^p(m,n)$, required to compute a degree n polynomial $p_n(x)$ on P(m) by the m-th order Horner's rule is

$$h^p(m,n) = \begin{cases} (1) \quad 2n & (m = 1) \\ (2) \quad 2\lceil \log m \rceil + 2\lfloor n/m \rfloor + 1 & (n + 1 > m \geq 2) \\ (3) \quad \lceil \log n \rceil + \lceil \log(n+1) \rceil + 1. & (m \geq n + 1) \end{cases}$$

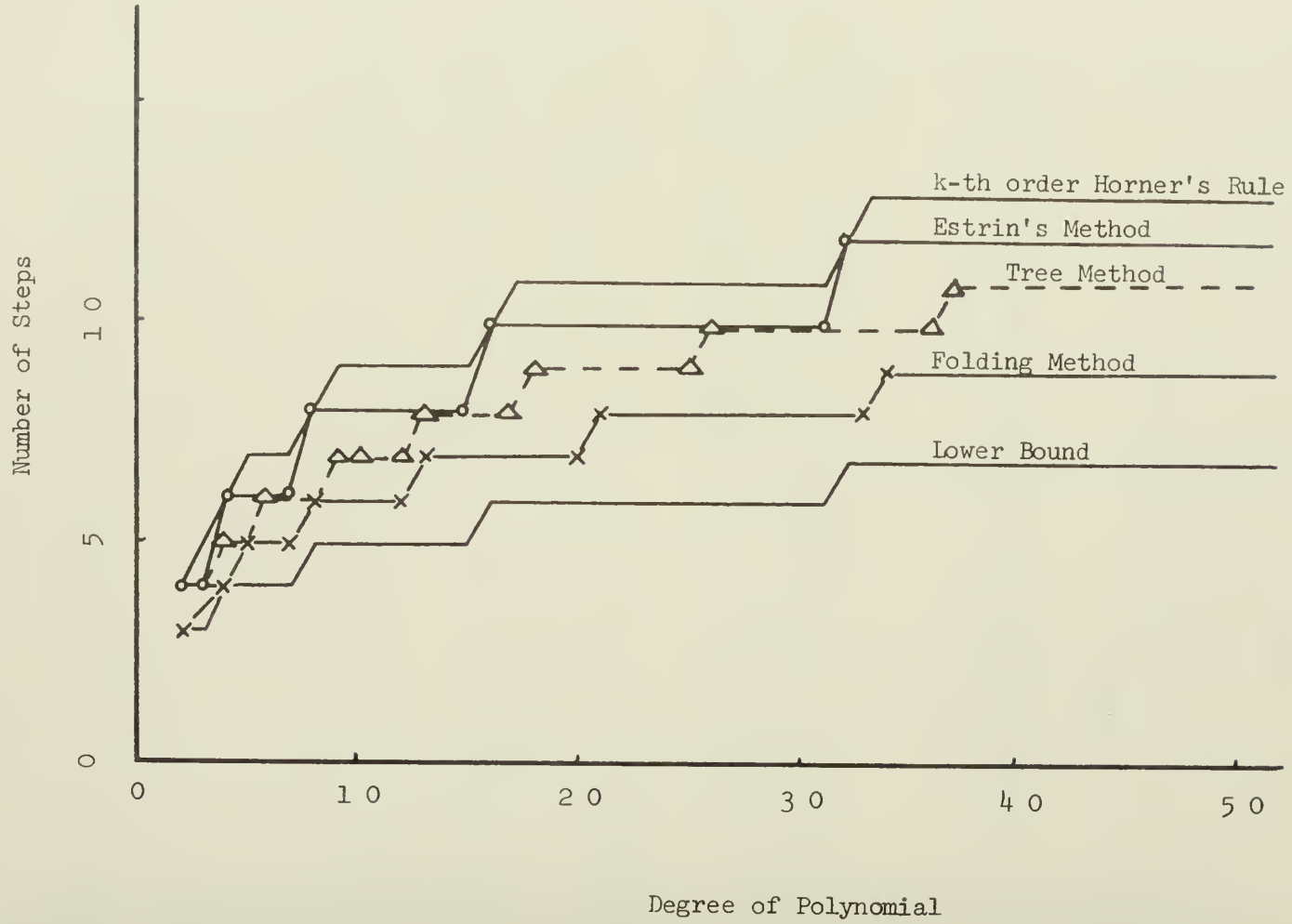


Figure 2.9. Comparison of the Four Parallel Polynomial Computation Schemes

Proof:

A proof for (1) can be found in [43]. (2) and (3) are self-evident from the above discussion.

(Q.E.D.)

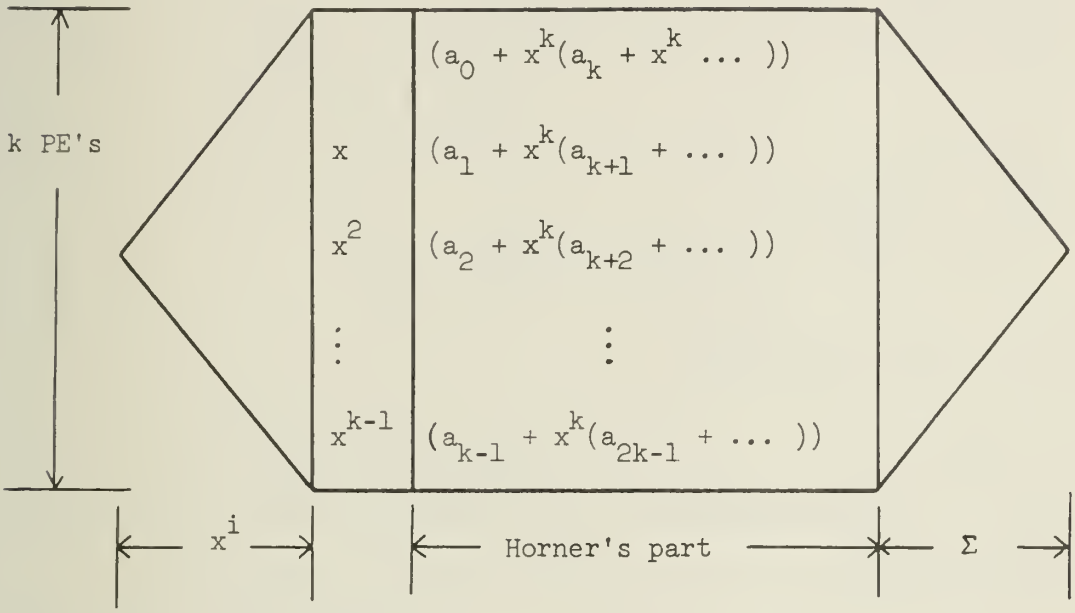


Figure 2.10. k-th Order Horner's Rule

Lemma 6:

The minimum number of steps h^p_{min} to compute a polynomial $p_n(x)$ of degree n by the k-th order Horner's rule ($1 \leq k \leq n + 1$) is

$$h^p_{min} = \lceil \log n \rceil + \lceil \log(n+1) \rceil + 1.$$

Proof:

It is enough to show that

$$2 \lceil \log m \rceil + 2 \lfloor n/m \rfloor + 1 \geq 2 \lceil \log(n+1) \rceil + 1$$

or

$$\lceil \log m \rceil + \lfloor n/m \rfloor \geq \lceil \log(n+1) \rceil$$

for $m \leq n$, because $2\lceil \log(n+1) \rceil \geq \lceil \log n \rceil + \lceil \log(n+1) \rceil$ and if $m \geq n + 1$, then by Theorem 5 $h^p(n+1,n) = h^p_{min}$.

Assume that $n = 2^g + t$ $(1 \leq t \leq 2^g)$ and $m = 2^k + s (1 \leq s \leq 2^k)$. Then we have two cases, i.e. (1) $g = k$ or (2) $g > k$. If $g = k$, then $t \geq s$ since $n \geq m$.

(1) $g = k$ and $t \geq s$.

Then $\lceil \log m \rceil = k + 1 = g + 1$, and

$$\left\lfloor \frac{n}{m} \right\rfloor = \left\lfloor \frac{2^g + t}{2^g + s} \right\rfloor = 1.$$

Hence

$$\lceil \log m \rceil + \lfloor n/m \rfloor = g + 2 \geq \lceil \log(n+1) \rceil .$$

(2) $g > k$.

We have further two subcases, i.e. (i) $1 \leq t < 2^g$ or (ii) $t = 2^g$.

(i) $1 \leq t < 2^g$

Then $\lceil \log(n+1) \rceil = g + 1$, $\lceil \log m \rceil = k + 1$

and

$$\left\lfloor \frac{n}{m} \right\rfloor = \left\lfloor \frac{2^g + t}{2^k + s} \right\rfloor \geq 2^{g - k - 1}.$$

Hence

$$\lceil \log m \rceil + \lfloor n/m \rfloor \geq k + 1 + 2^{g-(k+1)}$$

Now we show that for all $k < g$

$$f(k) = k + 1 + 2^{g-(k+1)} > g + 1.$$

Since for all $k < g$,

$$\frac{\partial}{\partial k} f(k) = 1 - (\log_e 2)\, 2^{g-(k+1)} < 0$$

$$\min f(k) = g + 1 \quad \text{where } 0 \leq k < g.$$

Hence $f(k) \geq g + 1$ or

$$\lceil \log m \rceil + \lfloor n/m \rfloor \geq g + 1 = \lceil \log(n+1) \rceil .$$

(ii) $t = 2^g$

Similarly to the above, we can show that

$$\lceil \log m \rceil + \lfloor n/m \rfloor > \lceil \log(n+1) \rceil .$$

The details are omitted.

(Q.E.D.)

Unlike the case of $h^a(m,n)$ and $h^e(m,n)$, $h^p(m,n)$ is not necessarily a non-increasing function of m. (A few curves in Figure 2.11 illustrate this.)

Therefore it becomes important to choose an appropriate m for a given n to compute in an optimum way.

Theorem 6:

Given an n-th degree polynomial.

Let

$$M = \mu_m(h^p(m,n)) = \lceil \log n \rceil + \lceil \log(n+1) \rceil + 1),$$

where

$$h^p(m,n) = 2 \lceil \log m \rceil + 2 \lfloor n/m \rfloor + 1.$$

Then

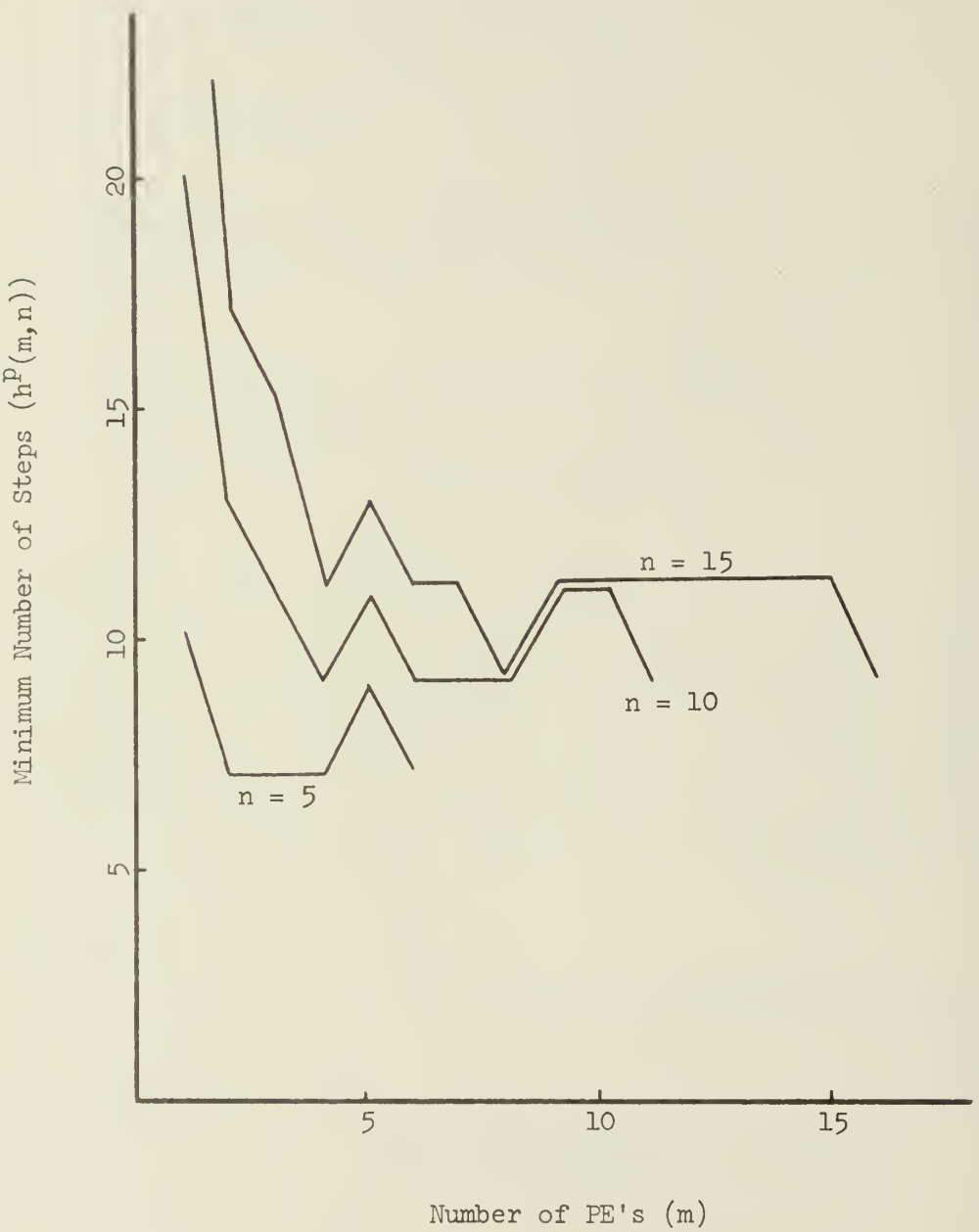


Figure 2.11. The Number of Steps, $h^p(m,n)$, to Compute $p_n(x)$ on P(m) by the m-th Order

$$M = \begin{cases} (1) \quad n + 1 & (n = 2^g) \\ (2) \quad \lceil (n+1)/3 \rceil & (2^g < n < 2^g + 2^{g-1}) \\ (3) \quad \lceil (n+1)/2 \rceil & (2^g + 2^{g-1} \leq n < 2^{g+1}) \end{cases}$$

where $g = \lfloor \log n \rfloor$.

Proof:

Proof is given for each case independently.

(1) $n = 2^g$.

The proof is divided into two parts. First it is clear that if we have $n + 1$ PE's, then $p_n(x)$ can be computed in h^p_{min} steps (see Theorem 5). Next we show that if the number of PE's, m, is less than or equal to $M - 1$ $(m \leq n)$, then $h^p_{min} < h^p(m,n)$, where

$$h^p_{min} = \lceil \log n \rceil + \lceil \log(n+1) \rceil + 1 = 2g + 2.$$

Let $m = 2^k + p \quad (k < g,\ 1 \leq p \leq 2^k)$. Then

$$h^p(m,n) = 2k + 3 + 2 \left\lfloor 2^{g-k} - \frac{2^{g-k} p}{2^k + p} \right\rfloor. \tag{18}$$

Let

$$P = \frac{2^{g-k} p}{2^k + p}.$$

Then

$$\frac{\partial}{\partial p} P = \frac{2^g}{(2^k+p)^2} \geq 0,$$

and

$$\max P = 2^{g-k-1} \qquad \text{for } 1 \le p \le 2^k.$$

From this and Eq. (18), we get

$$h^p(m,n) \ge 2k + 3 + 2\lfloor 2^{g-k-1} \rfloor. \tag{19}$$

Since $g > k$, Eq. (19) becomes

$$h^p(m,n) \ge 2k + 3 + 2^{g-k}.$$

Now let

$$f(k) = (2k + 3 + 2^{g-k}) - (2g + 2) = 2(k - g) + 1 + 2^{g-k}.$$

Since $2^a + 1 - 2a > 0$ (note that $\frac{\partial}{\partial a}(2^a + 1 - 2a) = \log_e 2 \cdot 2^a - 2 > 0$ for $a \ge 1$), we have $f(k) > 0$ for all $k < g$. Since

$$h^p(m,n) - (2g + 2) \ge f(k) > 0,$$

$$h^p(m,n) > 2g + 2 = h^p_{min}.$$

This proves (1).

Now since $\lceil \log n \rceil = \lceil \log(n + 1) \rceil$ if $n \ne 2^g$ for some g, we use

$$h^p_{min} = 2\lceil \log(n + 1) \rceil + 1$$

for

$$h^p_{min} = \lceil \log(n + 1) \rceil + \lceil \log n \rceil + 1$$

to prove (2) and (3). Then it is enough to show that

$$M = \mu_m(\lceil \log(n + 1) \rceil = \lceil \log m \rceil + \lfloor n/m \rfloor)$$

instead of

$$M = \mu_m(2\lceil \log(n + 1) \rceil + 1 = 2\lceil \log m \rceil + 2\lfloor n/m \rfloor + 1).$$

Since $2^g < n < 2^{g+1}$ for (2) and (3), we have $\lceil \log(n + 1) \rceil = g + 1$. By direct

computation, we can show that the theorem holds for $n \leq 10$ (see Table 2.4).

(2) $2^g < n < 2^g + 2^{g-1}$.

Now we show that

$$h^p_{min} = 2\lceil \log M \rceil + 2 \lfloor n/M \rfloor + 1.$$

To show this we first show

$$\lceil \log M \rceil + \lfloor n/M \rfloor = g + 1.$$

Since $2^g < n < 2^g + 2^{g-1}$, we have

$$2^{g-2} < \frac{n+1}{3} \leq 2^{g-1} \tag{20}$$

and

$$\lceil \log M \rceil = \lceil \log \lceil (n+1)/3 \rceil \rceil = g - 1. \tag{21}$$

Now let $\lceil (n+1)/3 \rceil = k$ ($k \geq 3$ as we assumed). Then

$$n + 1 = 3k - p \qquad (p \leq 2)$$

or

$$n = 3k - p - 1. \tag{22}$$

Using this and the relation $0 < (p+1)/k \leq 1$, we have

$$\lfloor n/M \rfloor = 2. \tag{23}$$

Thus from Eq. (21) and (23)

$$\lceil \log M \rceil + \lfloor n/M \rfloor = g + 1 = \lceil \log(n+1) \rceil,$$

or

$$h^p(M,n) = h^p_{min}.$$

Next we show that if $m' < \lceil (n+1)/3 \rceil$, then $\lceil \log m' \rceil + \lfloor n/m' \rfloor > g + 1 = \lceil \log(n+1) \rceil$ (or equivalently $h^p(m',n) > h^p_{min}$).

n \ m	1	2	3	4	5	6	7	8	9	10	11	g	M	Case
2	4	4	4									4	1	1
3	6	5	6	5								5	2	2
4	8	7	7	7								7	2	1
5	10	7	7	9	7							7	2	1
6	12	9	9	7	9	9	7					7	4	2
7	14	9	9	7	9	9	9	7				7	4	2
8	16	11	9	9	9	9	9	9	9			9	3	1
9	18	11	11	9	9	9	9	9	11	9		9	4	1
10	20	13	11	9	11	9	9	9	11	11	9	9	4	1

(1): $2^g \leq n < 2^g + 2^{g-1}$

(2): $2^g + 2^{g-1} \leq n < 2^{g+1}$

where $g = \lfloor \log n \rfloor$.

n: The degree of a polynomial

m: The number of PE's

M: The minimum number of PE's

Table 2.4. The Number of Steps Required to Compute $p_n(x)$, $h^p(m,n)$, for $n \leq 10$

We have two cases, i.e.

(i) $2^i < m' \leq 2^{i+1}$ where $i + 1 \leq g - 2$

and (ii) $2^{g-2} < m' < \lceil (n + 1)/3 \rceil$.

(i) $2^i < m' \leq 2^{i+1}$.

Since $i + 1 \leq g - 2$, we write

$$g - 2 = i + 1 + j \qquad (j \geq 0).$$

Then

$$\lceil \log m' \rceil = i + 1 = g - 2 - j,$$

and since $2^g < n < 2^g + 2^{g-1}$,

$$\lfloor n/m' \rfloor \geq 2^{g-i-1} = 2^{2+j}.$$

Thus

$$\lceil \log m' \rceil + \lfloor n/m' \rfloor \geq (g - 2 - j) + 2^{2+j} \geq g + 1$$
$$= \lceil \log(n + 1) \rceil$$

because $2^a \geq a + 1$ if $a \geq 1$.

(ii) $2^{g-2} < m' < \lceil (n + 1)/3 \rceil$.

Let us write

$$m' = \lceil (n + 1)/3 \rceil - q = k - q \qquad (q \geq 1).$$

Then by Eq. (20) and (22), we get

$$\lceil \log m' \rceil + \left\lfloor \frac{n}{m'} \right\rfloor = g - 1 + \left\lfloor \frac{3k - p - 1}{k - q} \right\rfloor$$
$$\geq g + 1 = \lceil \log(n + 1) \rceil,$$

because $q \geq 1$, $p \leq 2$ and $3q - p - 1 \geq 0$.

This ends a proof for (2).

(3) $(2^g + 2^{g-1} \leq n < 2^{g+1})$ can be proved in a similar manner and the details are omitted.

(Q.E.D.)

It should be noted that the function $h^p(m,n)$ is not a non-increasing function of m even if $m < M$ for some n. However if $n \leq 50$, then for more than 70% of the cases, $h^p(m,n)$ turn out to be non-increasing functions. (The only cases where $h^p(m,n)$ is not a non-increasing function are the cases n = 15, 27, 28, 29, 30, 31, 36, 37, 38, 39, 45, 46, and 47. In any case, $h^p(m,n)$ increases by at most one.)

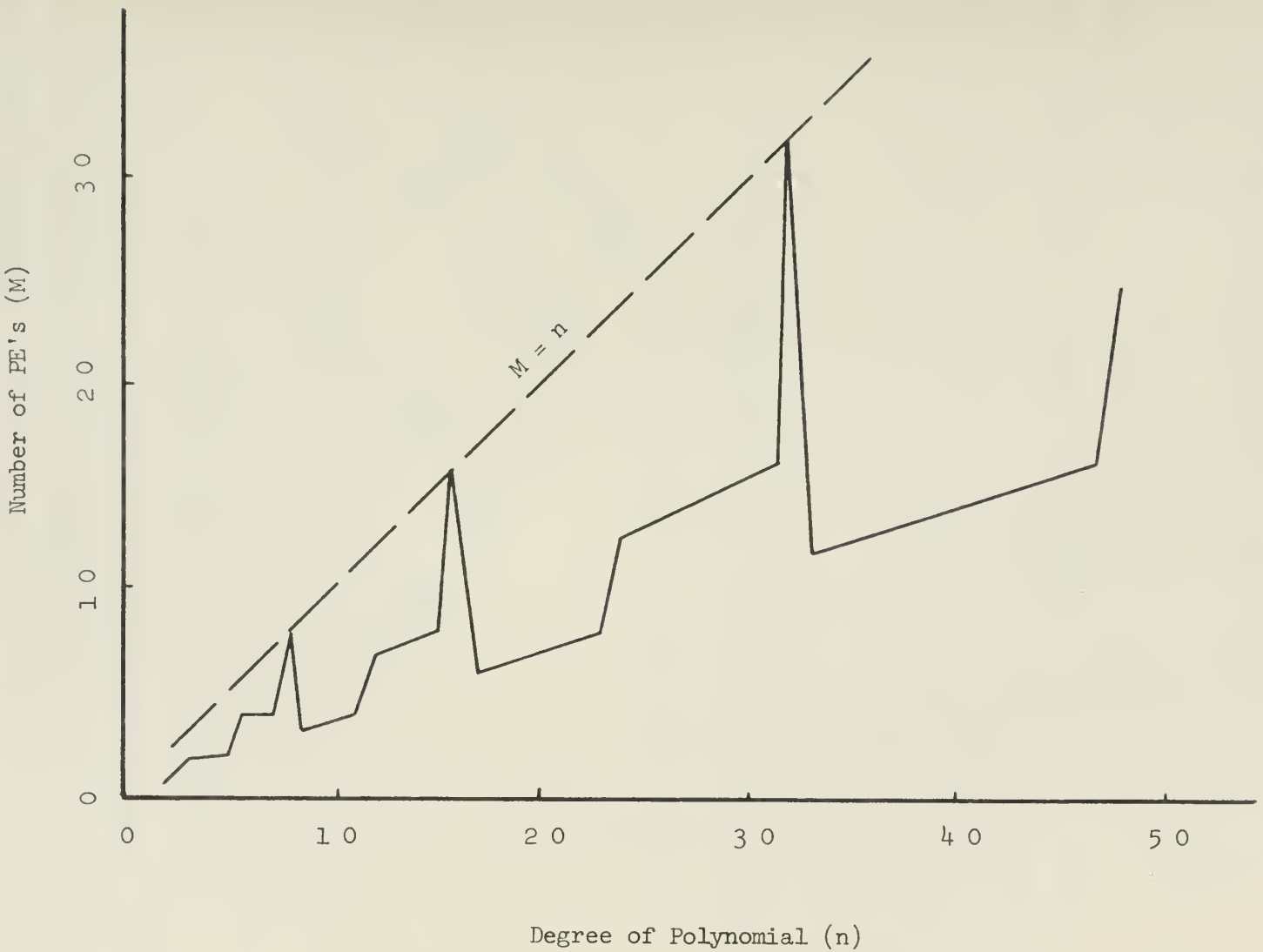


Figure 2.12. The Minimum Number, M, of PE's Required to Compute $P_n(x)$ in the Minimum Time

3. TREE HEIGHT REDUCTION ALGORITHM

3.1 Introduction

In this chapter, recognition of parallelism within an arithmetic statement or a block of statements is discussed. There are several existing algorithms which produce a syntactic tree to achieve this end. The tree is such that operations on the same level can be done in parallel. Among them, the algorithm by Baer and Bovet [6] is claimed to give the best result. For example, a statement

$$a + b + c + d \times e \times f + g + h$$

can be computed in four steps by their algorithm (Figure 3.1).

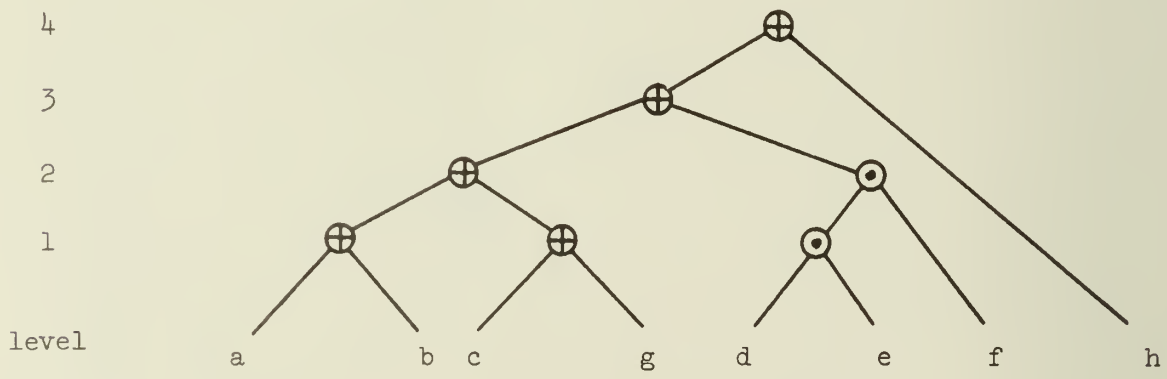


Figure 3.1. An Arithmetic Expression Tree (1)

The algorithm reorders some terms in a statement to decrease tree height. However, this algorithm does not always take advantage of distributions of multiplication over addition. An arithmetic expression

a(bcd+e) takes four steps as it is, whereas the equivalent distributed expression (abcd+ae) requires only three steps. A further example is Horner's rule. To compute a polynomial

$$p_n(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n, \qquad (1)$$

Horner's rule

$$p_n(x) = a_0 + x(a_1 + x(a_3 + \ldots x(a_{n-1}+x\ a_n))) \ldots) \qquad (2)$$

gives a good result for serial machines. However, if a parallel machine is to be used, (1) gives a better result than (2). Namely, if we apply Baer and Bovet's algorithm [6] on (2), we get 2n steps whereas (1) requires only $2\lceil \log_2(n+1)\rceil$ steps (see Chapter 2). Thus it is desirable for the compiler to be able to obtain (1) from (2) by distributing multiplications over additions properly. An algorithm to distribute multiplications properly over additions to obtain more parallelism (henceforth called the distribution algorithm or the tree height reduction algorithm) is discussed now.

3.2 Tree Height and Distribution

Definition 1:

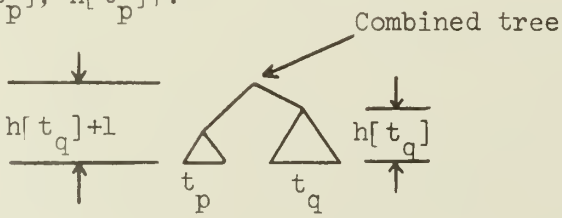
An arithmetic expression A consists of additions, multiplications, and possibly parentheses. We assume that addition and multiplication require the same amount of time (see Chapter 1). Subtractions and divisions will be introduced later. Small letters (a,b,c,...), possibly with subscripts, denote single variables. Upper case letters and t, possibly with subscripts, denote arbitrary arithmetic expressions, including single variables. t is used to single out particular subexpressions, i.e. terms.

Then A can always be written as either (1) $A = \sum_{i=1}^{n} t_i$ or

(2) $A = \prod_{i=1}^{n} (t_i)$, e.g. $A = abc + d(ef+g) = t_1 + t_2$ where $t_1 = abc$ and $t_2 = d(ef+g)$, or $A = (a+b)c(de+f) = (t_1)t_2(t_3)$ where $t_1 = a + b$, $t_2 = c$ and $t_3 = de + f$. Note that when we write $A = \prod_{i=1}^{n} (t_i)$, we implicitly assume that for each i $t_i = \sum_{j=1}^{n(i)} t_j$'. h[A] denotes the height of a tree T for A, which is of the minimum height among all possible trees for A in its presented form.

A <u>minimum height tree</u> (henceforth by a tree we mean a minimum height tree) for A, T[A], is built as follows [6].

Let us assume that $A = \sum_{i=1}^{n} t_i$ or $A = \prod_{i=1}^{n} (t_i)$ and that for each i, a minimum height tree $T[t_i]$ has been built. Then first we choose two trees, say $T[t_p]$ and $T[t_q]$, each of whose height is smaller than height of any other tree. We combine these two trees and replace them by the new tree whose height is one higher than max $\{h[t_p], h[t_p]\}$:#



This procedure is repeated until all trees are combined into one tree, which is T[A]. The procedure is formalized as follows:

#Figures are in scale as much as possible.

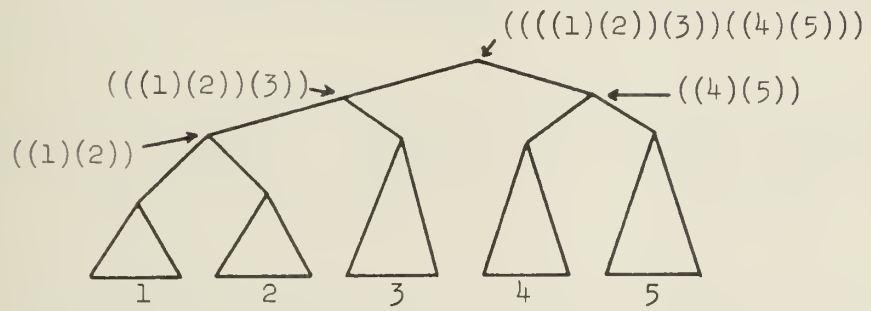
(1) Let ST = $\{(1), (2), (3), \ldots, (n)\}$ and let h'[i] = h[t_i] for all iϵST.

(2) Choose two elements of ST, say p and q such that h'[p], h'[q] $\leq$ M[#] where M = min $\{h'[u]\}$ for all uϵST- $\{p,q\}$.

(3) Now let ST = $\{ST - \{p,q\}\} \cup \{(p,q)\}$ and h'[(p,q)] = max $\{h'[p], h'[q]\}$ + 1.

(4) If | ST | = 1, then stop else go to Step 2.

After we apply the above procedure on A, we get e.g. ST = $\{((((1)(2))(3))((4)(5)))\}$ where a pair (ab) indicates that trees corresponds to a and b are to be combined. Thus in this case we get:



as a minimum height tree of A. In general the procedure is applied from the lowest parenthesis level (see Definition 3) to the higher parenthesis levels.

Example 1:

Let A = (a+bc)(d+efg) + hi

= $t_1 + t_2$

[#]If there are many choices, then choose those subtrees with smaller #(Sh[t_i]) values first (see Definition 8).

where $t_1 = (t_3)(t_6)$

$t_3 = a + bc = t_4 + t_5$

and $t_6 = d + efg = t_7 + t_8$.

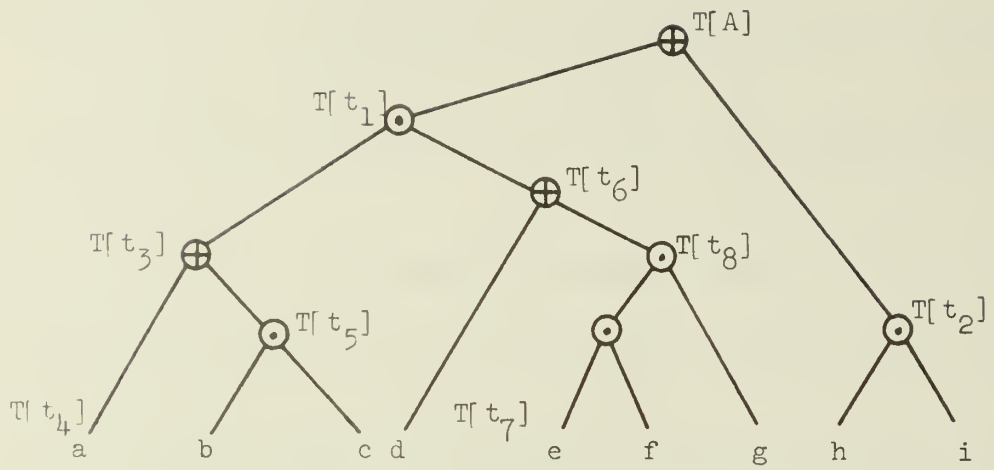


Figure 3.2. An Arithmetic Expression Tree (2)

The effective length e of an arithmetic expression is defined as $e[A] = 2^{h[A]}$.

The number of single variables in an arithmetic expression is the number of single variable occurences in it.

The height of a minimum height tree can be obtained without actually building a tree.

Theorem 1:

(1.1) If $A = \prod_{i=1}^{p} a_i$ or $A = \sum_{i=1}^{p} a_i$ then $h[A] = \text{Log}_2\lceil p \rceil_2$

(1.2) If $A = \sum_{i=1}^{p} t_i$ then $h[A] = \log_2 \left\lceil \sum_{i=1}^{p} e[t_i] \right\rceil_2$.

(1.3) If $A = \prod_{i=1}^{p} a_i \times \prod_{j=1}^{\ell} (t_j)$ then $h[A] = \log_2 \left\lceil \lceil p \rceil_2 + \sum_{j=1}^{\ell} e[t_j] \right\rceil_2$.

Given an arithmetic expression A, to obtain the height of a tree for A, Theorem 1 is applied from the inner most parts of A to the outer parts, recursively.

Proof:

(1) It is obvious that if $A = \sum_{i=1}^{p} a_i$ or $A = \prod_{i=1}^{p} a_i$ then $h[A] = \log_2 \lceil p \rceil_2$.

(2) Now let $A = \sum_{i=1}^{p} t_i$. Then we can replace each t_i by a product of $e[t_i]$ single variables without affecting the total tree height h[A]. (Note that each t_i must be computed before the summation over t_i is taken.) Thus A becomes $A' = \sum_{i=1}^{p} \prod_{j=1}^{e[t_i]} a_j$ and $h[A] = h\lceil A' \rceil$. Let us call a tree for $\prod_{j=1}^{e[t_i]} a_j$ a subtree. Then a tree for A' is built using subtrees in the increasing order of their heights. Since a binary tree of height h cannot accomodate more than 2^h leaves, we have $2^{h[A']} \geq \sum_{i=1}^{p} e[t_i] > 2^{h[A']-1}$ or $h[A'] = h[A] = \log_2 \left\lceil \sum_{i=1}^{p} e[t_i] \right\rceil_2$.

(3) can be proved in a similar manner.

(Q.E.D.)

Definition 2:

The additive length a and the multiplicative length m of an arithmetic expression A is defined as follows:

(2.1) If $A = \prod_{i=1}^{p} a_i$, then

(i) $a[A] = e[A]$ and

(ii) $m[A] = p$.

(2.2) If $A = \sum_{i=1}^{p} t_i$, then

(i) $a[A] = \sum_{i=1}^{p} e[t_i]$ and

(ii) $m[A] = e[A]$.

(2.3) If $A = \prod_{i=1}^{p} a_i \times \prod_{j=1}^{\ell} (t_j)$, then

(i) $a[A] = e[A]$ and

(ii) $m[A] = p + \sum_{j=1}^{\ell} e[t_j]$.

It is to be noted that

(1) $\frac{e[A]}{2} < \left\{ \begin{matrix} a[A] \\ m[A] \end{matrix} \right\} \leq e[A]$, and

(2) $h[A] = \log_2 \left\lceil a[A] \right\rceil_2 = \log_2 \left\lceil m[A] \right\rceil_2$;

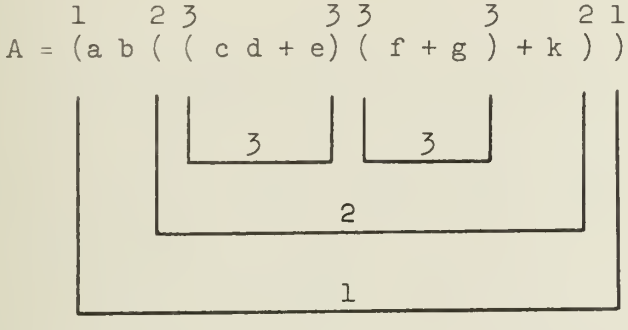
compare this definition with Theorem 1.

Definition 3:

The level ℓ of a parenthesis pair in an arithmetic expression is defined as follows:

First we start numbering parentheses at the left of the formula, proceeding from left to right, counting each left parenthesis, (, as +1 and each right parenthesis,), as -1 and adding as we go. We call the maximum number m the depth of parentheses. Now the level ℓ of each parenthesis pair is obtained as $\ell = p$, where p is the count for each parenthesis. The arithmetic expressions enclosed by the level ℓ parenthesis pair are called the level ℓ arithmetic expressions, A^{ℓ}. Also for convenience we assume that there is an outermost parenthesis pair which encloses A.

Example 2:



Now several lemmas are in order.

Lemma 1:

Let $A = \sum_{i=1}^{n} t_i$ or $A = \prod_{i=1}^{n} (t_i)$. Also let $A' = t_1 + t_2 + \ldots + t_i' + \ldots + t_n$ or $A' = (t_1) \times (t_2) \times \ldots \times (t_i') \times \ldots \times (t_n)$, and $A'' = A + t_{n+1}$ or $A'' = A \times (t_{n+1})$. Then

(i) $h[A'] \geq h[A]$ if $h[t_i'] \geq h[t_i]$.

(ii) $h[A''] \geq h[A]$.

Proof:

Obvious from Theorem 1.

What Lemma 1 implies is that the height of the tree for an arithmetic expression is a non-decreasing function of term heights, and the number of terms involved.

In an arithmetic expression, there are four possible ways of parenthesis occurence:

P_1) ... + (A) + ...

P_2) ... $\theta(t_1 \times t_2 \ldots \times t_n) \times (t_1' \times t_2' \ldots \times t_m') \theta$ ...

P_3) ... $\theta\ a_1 \times a_2 \times \ldots a_n \times (A)\ \theta$ ...

P_4) ... $\theta(t_1 + t_2 + \ldots + t_n) \times (t_1' + t_2' + \ldots + t_m')\ \theta$ ...

where θ represents +, $\times$, or no operation.

Lemma 2:

Let $D = B + (A) + C$ and $D_1 = B + (t_1 \times \ldots \times t_n) \times (t_1' \times \ldots \times t_m') + C$. Also let $D^d = B + A + C$ and $D_1^d = B + t_1 \times \ldots \times t_n \times t_1' \times \ldots \times t_m' + C$. Then $h[D] \geq h[D^d]$ and $h[D_1] \geq h[D_1^d]$.

Proof:

Obvious from Theorem 1.

As an example, let $D = (a + b + c) + d$ and $D_1 = (abc)(defgh)$. Then $D^d = a + b + c + d$ and $D_1^d = abcdefgh$, $h[D] = 3 > h[D^d] = 2$ and $h[D_1] = 4 > h[D_1^d] = 3$.

Lemma 3.

Let $D = \left(\sum_{i=1}^{n} t_i\right)\left(\sum_{j=1}^{m} t_j'\right)$ and $D^d = t_1t_1' + t_1t_2' + \ldots + t_nt_1' + \ldots + t_nt_m'$. Then $h[D^d] \geq h[D]$.

Proof:

Let $D = (A)(B)$ where $A = \sum_{i=1}^{n} t_i$ and $B = \sum_{i=1}^{m} t_i'$. Also without losing generality, assume that $h[A] \geq h[B]$. Then $h[D] = h[A] + 1$. For each j, let $d_j = t_j' t_1 + t_j' t_2 + \ldots + t_j' t_n$. It is clear that $h[t_j' t_i] \geq h[t_i]$ for all i and j. Thus from Lemma 1, we have $h[d_j] \geq h[A]$ for all j. Since $D^d = \sum_{j=1}^{m} d_j$, $h[D^d] \geq \min(h[d_j]) + \log_2\lceil m\rceil_2 \geq h[A] + \log_2\lceil m\rceil_2$, or since $h[D] = h[A] + 1$, $h[D^d] \geq h[D]$.

(Q.E.D.)

Note that the above lemma does not imply necessarily that if $D = \left(\sum_{i=1}^{n} t_i\right)(B)$, and $D^{d''} = t_1(B) + t_2(B) + \ldots + t_n(B)$, then $h[D^{d''}] \geq h[D]$. Actually, it can be shown that there is a case when $h[D] > h[D^{d''}]$. What Lemma 3 says is that $D = (A)(B)$ should not be fully distributed, but partial distribution, as in $D^{d''}$, may be done in some cases.

Lemmas 2 and 3 together indicate that distribution in case (P_3) and partial distribution for case (P_4) are the only cases which should be considered for lowering tree height. In cases (P_1) and (P_2), removal of parenthesis leads to a better result or at least gives the same tree height. Full distribution in case (P_4) always increases tree height and should not be done. Also it should be clear that in any case tree height of an arithmetic expression can not be lower than that of a component term even after distributions are done. For example, let $D = t(A) = t \times \left(\sum_{i=1}^{n} t_n \right)$ and $D^d = \sum_{i=1}^{n} (t \times t_i)$. Then from Lemma 1, we have $h[tt_i] \geq h[t_i]$ for each i. Thus $h[D^d] \geq h[A]$.

The same argument holds for all four cases. This assures that evaluation of distributions can be done locally. That is, if some distribution increases tree height for a term then that distribution should not be performed because once tree height is increased, it can never be remedied by further distributions. Actually, there are two cases where distribution pays. For example, if A = a(bcd + e), then $h[A] = 4$. However, if we distribute a, then we get A^d = abcd + ae and $h[A^d] = 3$. The idea is to balance a tree by filling the "holes" because a balanced tree can accommodate the largest number of variables among equal height trees. The situation is, however, not totally trivial, because by distribution, the number of variables in an expression is also increased. Next let A = a(bc + d) + e = t + e and A^d = abc + ad + e = t^d + e. In this case $h[A] = 4$ but $h[A^d] = h[t^d] = 3$. What happened here is that t is "opened" by distributing a over (bc + d) and the "space" to put e in is created.

At each level of parenthesis pair, cases (P3) and (P4), i.e., instances of "holes" and "spaces", are checked and proper distribution is performed. Next we give definitions of holes and spaces, and formalize these ideas.

3.3 Holes and Spaces

3.3.1 Introduction

Before we proceed further, let us study trees for arithmetic expressions more carefully.

Let $A = \sum_{i=1}^{p} t_i$. By Definition 1 we first build minimum height trees $T[t_i]$ for all i, and $T[A]$ is built by combining these $T[t_i]$. Once $T[t_i]$ is built the details of t_i do not matter, and the only thing that matters is its height $h[t_i]$.

Suppose $T[t_i]$ and $T[t_j]$ are combined to build $T[A]$. Assume also that $h[t_i] = h[t_j] + s$. Then we will get s nodes to which no trees are attached other than $T[t_j]$. We call these free nodes whose heights are $h[t_j] + 1$, $h[t_j] + 2$, ..., $h[t_i]$.

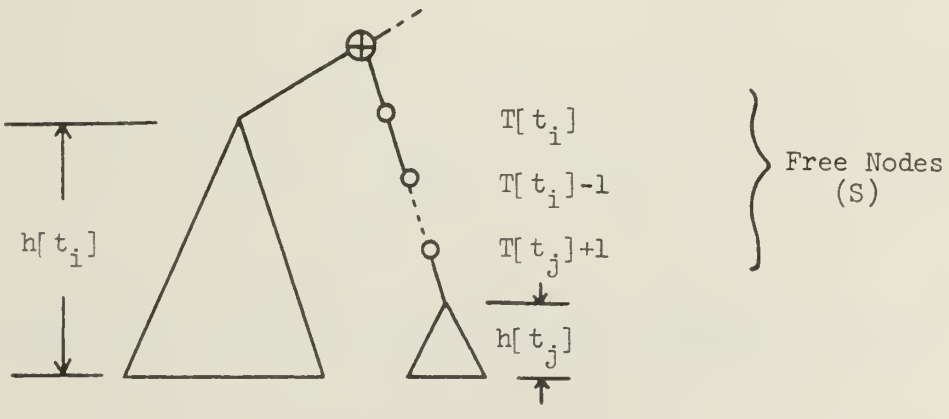


Figure 3.3. Free Nodes

Similarly we can enumerate all free nodes in T[A] with their heights.

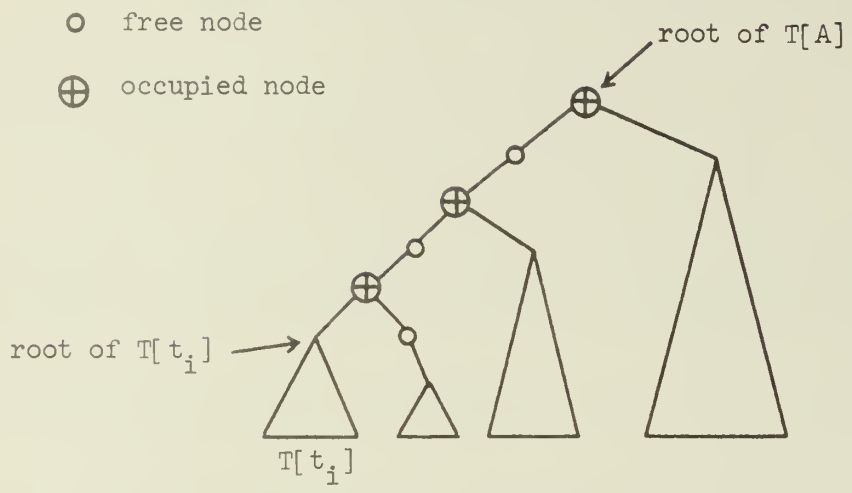
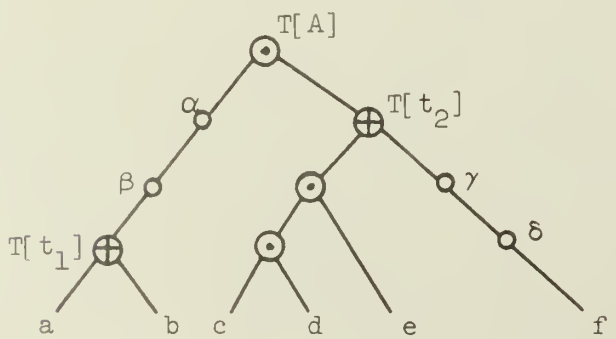


Figure 3.4. Free Nodes in a Tree

Free nodes in a tree $T\left[\pi_{i=1}^{p}(t_i)\right]$ are defined similarly.

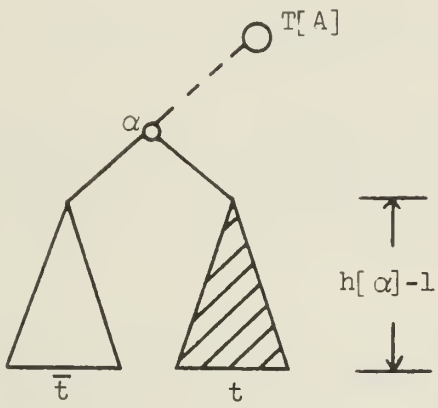
Let us emphasize that once we get T[t_i] we treat it as a whole and do not care about its details when we build T[A]. That is, when we consider free nodes in T[A] we mean free nodes "in" T[A] but "outside" of T[t_i]. For example let A = (a+b)(cde+f) = $(t_1)(t_2)$. Then



α and β are free nodes in T[A] while γ and δ are free nodes in $T[t_2]$ and not in T[A].

Now suppose there are m free nodes in T[A]. We number them arbitrarily from 1 to m. Also let us denote the height of a free node α by $h[\alpha]$.

Given a free node α whose height is $h[\alpha]$ in $T[A = \Sigma t_i]$ (or $T[A = \pi(t_i)]$), by definition we can attach a tree T[t] whose height is $h[\alpha]-1$ (or whose effective length is $2^{h[\alpha]-1}$) to α without affecting the height of A:



Definition 4:

For $A = \pi(t_i)$ or $A = \Sigma t_i$, we build a tree. Then

(4.1) define $F_A[A]$ to be a set of all free nodes in T[A], and

(4.2) for each i define $F_R[A, t_i]$ to be a set of all free nodes which exist between the roots of T[A] and $T[t_i]$, i.e. the free nodes which we encounter when we traverse T[A] down to the root of $T[t_i]$.

For example let us consider the following tree (see Figure 3.5).

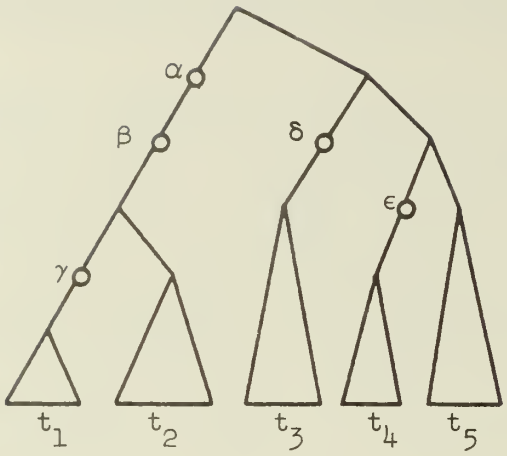


Figure 3.5. An Example of F_A and F_R

Then $F_A[A] = \{\alpha, \beta, \gamma, \delta, \epsilon\}$ and $F_R[A, t_1] = \{\alpha, \beta, \gamma\}$, $F_R[A, t_2] = \{\alpha, \beta\}$ etc.

Lemma 4:

Suppose $h[\alpha] = h[\beta]$ for some free nodes α and β in $T[A]$. Then without changing the tree height $h[A]$, we can replace two free nodes α and β by one free node β' whose height is $h[\alpha]+1$.

Proof:

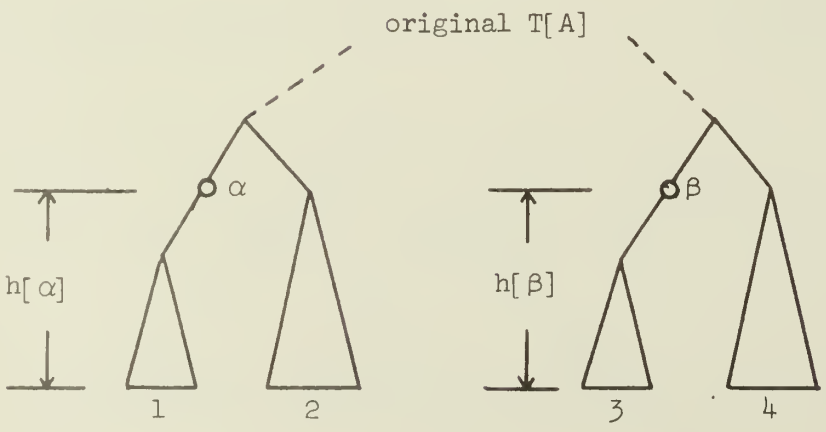


Figure 3.6. Elimination of a Free Node

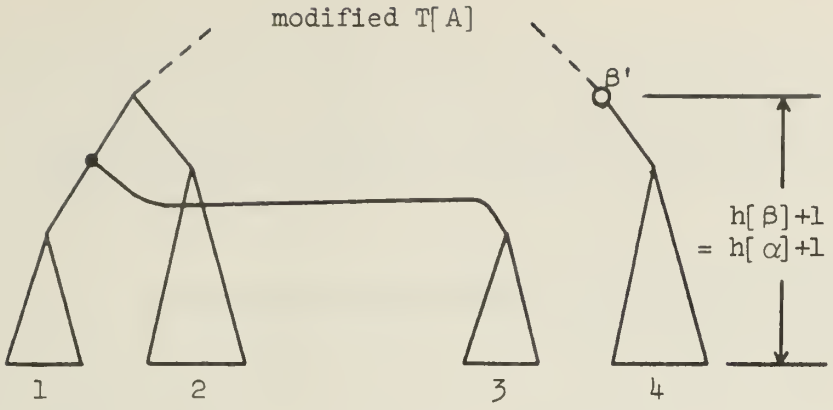


Figure 3.6. (continued)

We can combine subtrees 1 and 3, and hence eliminate free nodes α and β and create a new free node β' (see Figure 3.6).

(Q.E.D.)

Given $F_A[A]$, two free nodes α and β of equal height can be replaced by one free node β' whose height is $h[\alpha] + 1$. Repeating this procedure finally we get a new $F_A'[A]$ in which no two free nodes have the same height. Let γ and δ be free nodes in $F_A[A]$ and $F_A'[A]$ respectively. Assume that for all free nodes α in $F_A[A]$ (or $F_A'[A]$), $h[\gamma] \geq h[\alpha]$ (or $h[\delta] \geq h[\alpha]$). Then obviously $h[\gamma] \leq h[\delta]$. However, it is clear that $h[\delta] < h[A]$, i.e. if $h[\delta] = h[A]$ then $h[A]$ is not the height of a minimum tree (see Figure 3.7). Hence we have the following corollary.

Corollary 1:

In a minimum height tree $T[A]$, $\sum_{\alpha \in F_A[A]} 2^{h[\alpha]-1} < \frac{1}{2} e[A]$.

Proof:

Assume that $\sum_{\alpha \in F_A[A]} 2^{h[\alpha]-1} = \frac{1}{2}e[A]$. Then by Lemma 4, all α in $F_A[A]$ can be replaced by one free node β whose height is h[A] (note $2^{h[A]-1} = e[A]/2$). This, however, implies that we can build a tree T'[A] whose height is h[A]-1, which contradicts our assumption that T[A] is a minimum height tree for A.

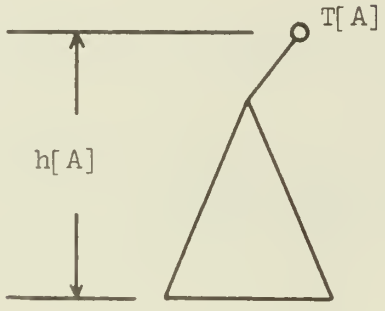


Figure 3.7. A Minimum Height Tree

(Q.E.D.)

The following definitions are also used in the next section.

Definition 5:

An integer set is a set of integers with possibly duplicated elements, e.g. {2,2,2,4,8,8}. If an integer x occurs in an integer set Y at least once then we write x in Y, e.g. 2 in {2,2,2,4,8,8}. Let Y and Z be two integer sets. Then by Y uni Z we get an integer set where if an integer x occurs i times in Y and j times in Z, then x occurs i + j times in Y uni Z, e.g. {2,4,4} uni {2,4,8} = {2,2,4,4,4,8}. Also if Y is an integer set, then #(Y) = (the sum of values of all elements of Y), e.g. #({2,4,4}) = 10, and le Y = (the value of the largest

element in Y), e.g. le {2,4,4} = 4. Furthermore if x in Y, then by Y del {x} we mean the integer set which is obtained by deleting one occurence of x from Y, e.g. {2,4,4,8,16,16} del {4} = {2,4,8,16,16}.

Let $Y_1, Y_2, \ldots, Y_p$ be integer sets. Then by $\underset{i=1}{\overset{p}{M_u}}$ (d) Y_i we mean the set Y constructed as follows.

(1) Let Y = $\emptyset$ (empty) and s = d.

(2) If s = 0, then stop else go to (3).

(3) Let u = min{le Y_i} and k be an index of Y_i such that u = le Y_k.

If there are more than one k which satisfy this, then pick an arbitrary one. Now let Y = Y uni {u} and s = s - u. Also let $Y_k = Y_k$ del {u} and Y_i = (Y_i del {le Y_i}) uni {le Y_i-u} for all i ($i \neq k$). Go to (2).

For example let Y_1 = {1,2,2,8}, Y_2 = {4,4,8} and Y_3 = {16}. Then Y = $\underset{i=1}{\overset{3}{M_u}}$ (13) Y_i is constructed as follows.

(1) Y = $\emptyset$ and s = 13.

(2) le Y_1 = 8, le Y_2 = 8, le Y_3 = 16. Hence u = 8 and k = 1. Then Y = {8} and s = 13 - 8 = 5. Also Y_1 = {1,2,2}, Y_2 = {4,4} and Y_3 = {8}.

(3) le Y_1 = 2, le Y_2 = 4, le Y_3 = 8. Hence u = 2 and k = 1. Then Y = {2,8} and s = 5 - 2 = 3. Also Y_1 = {1,2}, Y_2 = {4,2}, Y_3 = {6}.

(4) Repeating the procedure we finally get Y = {1,2,2,8}.

Definition 6:

Let m be an integer. Now write m as a sum of powers of 2 in which each power appears at most once. Then by $\Sigma b(m)$ we mean the integer set of powers of 2 which appear in the above sum.

For example since $13 = 2^3 + 2^2 + 2^0 = 8 + 4 + 1$, $\Sigma\, b(13) = \{1,4,8\}$ but $\Sigma\, b\,(13) \neq \{1,4,4,4\}$ since 4 appears more than once.

3.3.2 Holes

Now we discuss holes in an arithmetic expression. Intuitively, if an arithmetic expression A has a hole of size u then an arithmetic expression t' with $e[t'] \leq u$ may be distributed over A without increasing tree height.

Definition 7:

For each $A = \pi(t_i)$ we define a total hole function H_T and for each t_i in $A = \Sigma\, t_i$ we define a relative hole function H_R as follows:

(7.1) For each $A = \pi(t_i)$, build a tree T[A]. Then define $H_T[A] = \Sigma\, 2^{h[\alpha]-1}$ for all $\alpha \in F_A[A]$.

(7.2) For each $A = \Sigma\, t_i$, build a tree T[A]. Then for each t_i define $H_R[A,t_i] = \Sigma\, 2^{h[\alpha]-1}$ for all $\alpha \in F_R[A,t_i]$. If $F_R[A,t_i] = \emptyset$, then let $H_R[A,t_i] = 0$.

As stated before if α is a free node in T[A], then $2^{h[\alpha]-1}$ is the effective length of a term t' whose tree T[t'] can be attached to α without changing the tree height h[A]. Also let u in $\Sigma\, b(H_T[A])$. Then this implies that there is a free node α in $F_A[A]$ such that $2^{h[\alpha]-1} = u$ (see Lemma 4). Similarly if u in $\Sigma\, b(H_R[A,t_i])$ then there is a free node α in $F_R[A,t_i]$ such that $2^{h[\alpha]-1} = u$. Thus in general

(1) if $A = \pi(t_i)$, then $h[(t')\times(A)] = h[A]$ if $m[t'] \leq u$, and

(2) if $A = \Sigma\, t_i$, then $h[t'+A] = h[A]$ if $a[t'] \leq u$, where $u = \sum_{\alpha \in F_A[A]} 2^{h[\alpha]-1}$.

Definition 8:

The set of holes Sh[A] for an arithmetic expression A is an integer set defined as follows:

(8.1) If $A = \pi_{i=1}^{p} a_i$, then we let $Sh[A] = \{\Sigma b(e[A]-p)\}$, e.g. Sh[abcde] = $\{\Sigma b(3)\} = \{1,2\}$.

(8.2) If $A = \pi_{i=1}^{p} (t_i)$, then we let $Sh[A] = \underset{i=1}{\overset{p}{\text{uni}}}(Sh[t_i])$ uni $\Sigma\, b(H_T[A])$,

e.g. Sh[(abc+d)(efg+h)(i+j)] = Sh[abc+d] uni Sh[efg+h] uni Sh[i+j] uni $\{\Sigma b(H_T[A])\} = \{1\}$ uni $\{1\}$ uni $\Sigma\, b(14) = \{1,1,2,3,8\}$.

(8.3) If $A = \Sigma_{i=1}^{p} t_i$, then we first let $Sh'[t_i] = Sh[t_i]$ uni $\Sigma\, b(H_R[A,t_i])$

and $d = \min_i (\#(Sh'[t_i]))$. Then we let $Sh[A] = \underset{i=1}{\overset{p}{M_u}}(d)\ Sh'[t_i]$.

Example 2:

Let $A = (a+b)(c+def) + ghi = (t_1)(t_2) + t_3 = t_4 + t_3$. Then $Sh[t_2] = \{1\}$ and $Sh[(t_1)(t_2)] = \{1\}$ uni $\Sigma\, b(6) = \{1,2,4\}$. Also $Sh[t_3] = \{1\}$. Hence $Sh'[t_4] = \{1,2,4\}$ and $Sh'[t_3] = \{1\}$ uni $\Sigma\, b(12) = \{1,4,8\}$. Now $d = \min((\#Sh'[t_4]), \#(Sh'[t_3])) = 7$.

Hence $Sh[A] = \underset{i=3,4}{M_u}(7)\ Sh'[t_i] = \{1,2,4\}$.

In (8-3) above note that for every i

(1) if u in Sh[A], then there is u_i in Sh'[t_i] such that $u \leq u_i$,

(2) if u in Sh[A], then $u < \#(Sh'[t_i])$. Also there is at least one k such that le Sh[A] = le Sh'[t_k].

Given an arithmetic expression A, and $t' = \sum_{i=1}^{q} t_i'$, if $e[t'] \leq$ le Sh[A], then t' may be distributed over A without increasing tree height. Informally we say that A has a hole which can accomodate t'.

Now we show that the above assertion is indeed valid. First we observe that if $t' = \pi_{i=1}^{q} (t_i')$, then each t_i' may be distributed independently over A. Thus in the following we can assume that $t' = \sum_{i=1}^{q} t_i'$. Note that if t can be distributed over A without increasing tree height, then any t'($e[t'] \leq e[t]$) can be distributed over A as well.

Lemma 5:

Let $A = \pi_{i=1}^{p} a_i$. Then $h[A] = h[(A) \times (t')]$ if $e[t'] \leq$ le Sh[A].

Proof: Obvious by Theorem 1.

Theorem 2:

(1) Let $A = \pi_{i=1}^{p} (t_i)$. Then $h[A] = h[(t'\bar{A})^d]$ if $e[t'] \leq$ le Sh[A].

(2) Let $A = \sum_{i=1}^{p} t_i$. Then $h[A] = h[(t'\overline{A})^d]$ [#] if $e[t'] \leq \underline{le}\ Sh[A]$.

Proof:

We use a mathematical induction to prove the above theorem. Lemma 5 serves as a basis.

Let $A = \pi_{i=1}^{p} (t_i)$ or $A = \sum_{i=1}^{p} t_i$. Assume that the theorem holds for t_i.

(1) $A = \pi_{i=1}^{p} (t_i)$.

We show that if $e[t'] \leq \underline{le}\ Sh[A]$ then A can be multiplied by t' without increasing tree height. There are two cases:

(i) There is k such that $e[t'] \leq \underline{le}\ Sh[t_k]$. Then we distribute t' over t_k without increasing tree height.

(ii) $\forall_k\ e[t'] > \underline{le}\ Sh[t_k]$. In this case $e[t'] \leq \underline{le}\ \Sigma\, b(H_T[A])$. Then there is a free node α in $F_A[A]$ such that $u = 2^{h[\alpha]-1}$ and $e[t'] \leq u$ means that $T[t']$ can be attached to α without increasing tree height.

Hence A can be multiplied by t' without increasing tree height.

(2) $A = \sum_{i=1}^{p} t_i$.

We show that if $e[t'] \leq \underline{le}\ Sh[A]$ then t can be distributed over A without increasing tree height.

[#] We write $(t'\overline{t})^d$ for the expression obtained after distributing t' over t such that the tree height is deduced, e.g. $(a, \overline{b+cd})^d = ab + acd$.

First note that for all i, if $\alpha \in F_R[A, t_i]$ then $2^{h[\alpha]-1} > \underline{le}\ Sh[t_i]$. Now assume that $e[t'] \leq \underline{le}\ Sh[A]$. For fixed k, that $u = \underline{le}\ Sh[A]$ implies either

(i) $u \leq \underline{le}\ Sh[t_k]$

or (ii) there is α in $F_R[A, t_k]$ such that $u \leq 2^{h[\alpha]-1}$ (or equivalently $u \leq \underline{le}\ \Sigma\, b(H_R[A, t_k])$).

In the first case we have $h[t_k] = h[(t'\bar{t}_k)^d]$ by assumption and

$$h[A] = h[\sum_{i \neq k} t_i + (t'\bar{t}_k)^d].$$

In the second case we attach $T[t']$ to α (Figure 2.8(a)) without increasing the tree height h[A], i.e.

$$h[A] = h[\sum_{i \neq k} t_i + (t') \times (t_k)].$$

In general let T' be a subtree in T[A] whose root is α (Figure 3.8(b)). Then

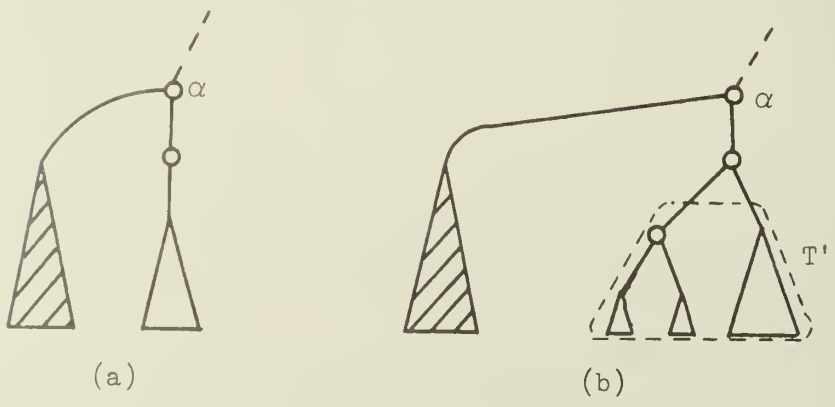


Figure 3.8. Attachment of T[t'] to a Free Node

there may be other term trees besides $T[t_k]$ in T', e.g. $T[t_j]$ and $T[t_h]$ in Figure 3-8(b). Hence we get,

$$h[A] = h[\sum_{i\neq k,j,h} t_i + (t') \times (t_k+t_j+t_h)]$$

in this case. Note that α is also in $F_R[A,t_j]$ and $F_R[A,t_h]$.

Repeating this procedure for all k we can get an arithmetic expression equivalent to $\sum_{i=1}^{p} (t') \times (t_i)$ or $(t')(A)$ without increasing tree height. This proves (2).

It is obvious that in both (1) and (2) if $e[t'] > \underline{le}\ Sh[A]$, then t' can not be distributed (multiplied) over A without increasing tree height.

(Q.E.D.)

Lemma 6:

Let $A = \Gamma\ t_i$ and $e[t'] \leq \underline{le}\ Sh[A]$. Then after t' is distributed over A, we have

$$Sh[(t'\overline{A})^d] = (Sh[A]\ \underline{del}\ \{u\})\ \underline{uni}\ \Gamma\ b(u-e[t'])\ \underline{uni}\ Sh[t']$$

where u is the smallest element in Sh[A] bigger than e[t'].

Now let us summarize what we have so far. Let A and t' be arithmetic expressions where $A = \Gamma\ t_i$ and $t' = \Gamma\ t'_i$. If $e[t'] \leq \underline{le}\ Sh[A]$, then t' can be distributed over A without increasing tree height, i.e. $h[(t') \times (A)] > h[(t'\overline{A})^d] = h[A]$. A set of holes in $(t'\overline{A})^d$ is given by the above lemma. Since it is obvious that $\underline{le}\ Sh[A] < e[A]$, we have the following lemma.

Lemma 7:

Let $t' = \Sigma t_i'$. Then $h[(t')(A)] > h[(t'\bar{A})^d]$ implies that $h[t'] < h[A]$.

In general let $t' = \pi^{q}(t_i')$. For convenience let us assume that $e[t_1'] \leq e[t_2'] \leq e[t_3'] \leq \ldots \leq e[t_q']$. Then if the following procedure can be accomplished successfully, we say that A has holes to accomodate all t_i' $(i=1, 2,\ldots,q)$.#

Procedure:

(1) Let $V_i = (t_i'(t_{i-1}',\ldots,(t_1'\bar{A})^d)^d\ldots)^d$ and $V_0 = A$. Let $i = 1$.

(2) Check if $e[t_i'] \leq \underline{le}\ Sh[V_{i-1}]$.

(3) If so, then distribute t_i' over V_{i-1}, and we have V_i.

(4) Evaluate $Sh[V_i]$.

(5) If $i=q$ then stop, else let $i = i + 1$ and go to (1).

The procedure may be accomplished successfully if $m[t'] \leq \#(Sh[A])$.

3.3.3 Space

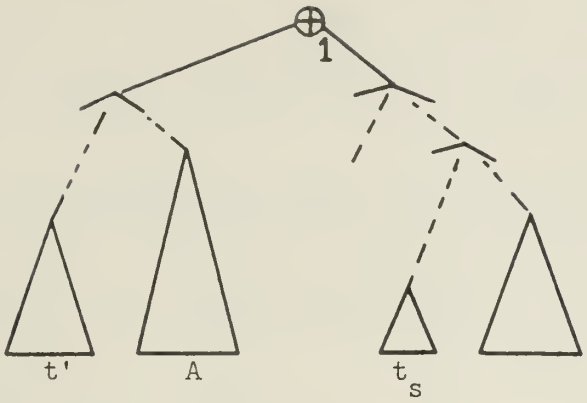
Now the second possible distribution case, i.e. space is studied.

The idea of the second distribution case is that given an arithmetic expression D of the following form

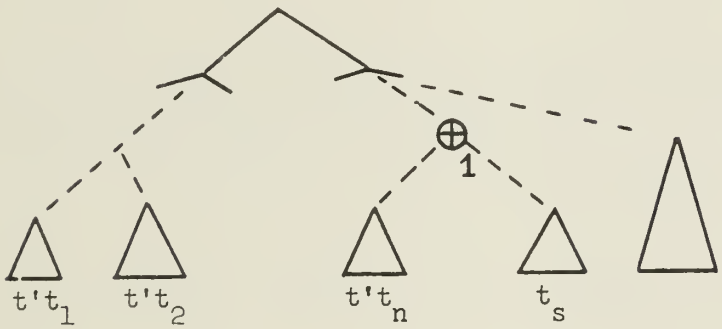
$$D = \ldots \theta(t') \times (A) +_1 t_s \theta \ldots$$

distribute t' over A so that t_s can be hidden under the combined tree as shown in Figure 3.9.

We assume that each t_i' does not have any holes, i.e. $Sh[t_i'] = \emptyset$ for all i. Hence $Sh[(t'\bar{A})^d] = (Sh[A]\ \underline{del}\ \{u\})\ \underline{uni}\ b(u\text{-}e[t'])$ for example.



(a)



(b)

Figure 3.9. An Example of Space (1)

In other words, in case of D, the addition $+_1$ cannot be done before $(t')(A)$ (we write $t'(A)$) is computed while in case of D^d it may be done earlier.

Note that if $h[t'(A)] > h[(t'\bar{A})^d]$ then A has enough holes to accomodate t' and the distribution of t' over A is done anyway. Henceforth throughout the rest of this section we assume that A does not have any holes to accomodate t'. Thus we now deal with the case when $h[t'(A)] \leq h[(t'\bar{A})^d]$. However, if $h[t'(A)] < h[(t'\bar{A})^d]$ holds, then clearly there is no way to

get $h[D^d] < h[D]$ by Lemma 1. Thus we have:

Lemma 8:

(1) $h[t'(A)] = h[(t'\bar{A})^d]$ must hold to get $h[D^d] \leq h[D]$.

(2) Let $A = \sum_{i=1}^{n} t_i$. Then $h[t'(A)] \geq h[(t'\bar{A})^d]$ if and only if $e[t'] < e[A]$ (i.e. $h[t'] < h[A]$). This implies that $h[t'(A)] = h[A] + 1$.

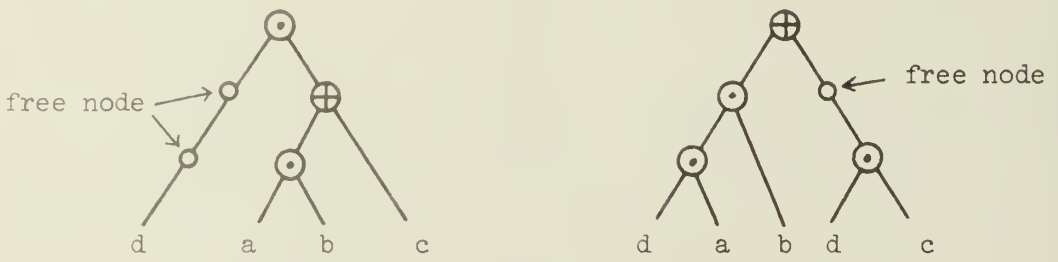
Proof:

By inspection.

Intuitively the space Sp in an arithmetic expression $A = \sum_{i=1}^{p} t_i$ with respect to t' is defined as

$$Sp[A,t'] = e[(A) \times (t')] - \sum_{i=1}^{p} e[(t') \times (t_i)].$$

For example let A = ab + c and t' = d. Then $S_p[A,t'] = 2$.



(a) (b)

Figure 3.10. An Example of Space (2)

Let $D = t'(A) + t = (t'\bar{A})^d + t$. Note that a free node in $T[t'(A)]$ cannot be used to attach T[t] while a free node in $T[(t'\bar{A})^d]$ may be used to attach T[t].

Now the formal definition of space follows.

Definition 9:

Given arithmetic expressions $A = \sum_{i=1}^{p} t_i$ and $t' = \sum_{i=1}^{q} t_i'$, the space function Sp of A w.r.t. t' is defined as follows. First we build trees T[A] and T[t'], and in T[A] let F be a set of free nodes f higher than T[t'] (h[f] > h[t']). Also we define a set I as follows. We let $i \in I$ if $h[t_i] > h[t']$ and $e[t'] \leq$ le $Sh[t_i]$. Now the space function is obtained as:

$$Sp[A,t'] = \sum_{f \in F} 2^{h[f]} + \sum_{i \in I} 2^{h[t_i]}.$$

To show how Definition 9 works, we first describe how to build $T[(t'\overline{A})^d]$ by attaching T[t'] to T[A] properly (i.e. by distributing t' over A properly). Since $h[(t'\overline{A})^d] = h[A] + 1$ (Lemma 8(1)), we first study to build T'[A] which is obtained by replacing $T[t_i]$ in T[A] by $T'[t_i]$ whose height is $h[t_i] + 1$. Then the height of T'[A] is h[A] + 1. Building $T[(t'\overline{A})^d]$ from T[A] may be explained in an analogous way.

As stated before the only case to be considered is when $h[t'(A)] = h[(t'\overline{A})^d] = h[A] + 1$ holds (Lemma 8(1)). Suppose that all $T[t_i]$ in T[A] are replaced by $T'[t_i]$ whose height is $h[t_i] + 1$. Then the new tree T'[A], whose height is h[A] + 1, is obtained. Note that a free node α in T[A] now becomes a free node α' in T'[A] with height $h[\alpha] + 1$. In T'[A] if $T'[t_i]$ is replaced by $T[t_i]$ again, then a new free node β', whose height is $h[t_i] + 1$, is created. Having these facts in mind, now we describe the way t' is distributed over A to create space.

The tree $T[(t'\overline{A})^d]$ is built from T[A] as follows (note $h[(t'\overline{A})^d] = h[A] + 1$). Depending on the height of $T[t_j]$, we have two cases.

(1) $h[t_j] \geq h[t']$.

If $T[t_j]$ has a hole to put $T[t']$, then we fill it by $T[t']$. In this case $h[t'(t_j)] = h[t_j]$ and a new free node β' whose height is $h[t_j] + 1$ is created in $T[(t'\bar{A})^d]$. Otherwise $h[t'(t_j)] = h[t_j] + 1$.

(2) $h[t_j] < h[t']$.

Find the tree $T[\Sigma t_s]$ whose height is $h[t']$ and which includes $T[t_j]$. We multiply Σt_s by t' and get $h[t'(\Sigma t_s)] = h[\Sigma t_s] + 1$.

Note that t_j and Σt_s are treated as terms of A. In the resultant tree $T[(t'\bar{A})^d]$, those free nodes in T[A] whose heights are less than or equal to $h[t']$ (i.e. free nodes in $T[\Sigma t_s]$) do not appear.

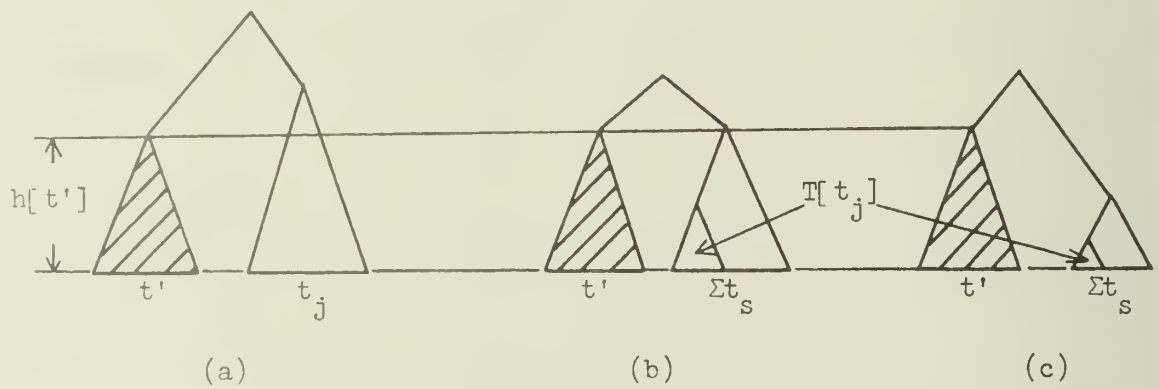


Figure 3.11. Distribution of t' over A

A free node α in T[A] ($h[\alpha] > h[t']$) appears in $T[(t'\bar{A})^d]$ as a free node α' where $h[\alpha'] = h[\alpha] + 1$. Thus $T[(t'\bar{A})^d]$ has those α' and β' described in (1) as free nodes.

If δ' is a free node in $T[(t'\bar{A})^d]$, then a tree $T[\bar{t}]$ $(h[\bar{t}] \leq h[\delta'] - 1)$ can be attached to $T[(t'\bar{A})^d]$ without increasing tree height (i.e. $h[(t'\bar{A})^d] = h[(t'\bar{A})^d + \bar{t}]$). Since $h[\delta']$ is either $h[t_j] + 1$ or $h[\alpha] + 1$, we have $e[\bar{t}] = 2^{h[t_j]}$ or $2^{h[\alpha]}$.

In general if $a[\bar{t}] \leq Sp[A,t']$ then we have $h[t'(A)] = h[(t'\bar{A})^d] = h[(t'\bar{A})^d + \bar{t}]$.

Definition 9 may be generalized to include the case where $t' = \pi^q(t_i')$. In this case we first obtain $h[t']$ and $e[t_i']$ $(i=1,2,\ldots,q)$, and build $T[A]$. In $T[A]$ let F be a set of free nodes f which are higher than $h[t']$ $(h[f] > h[t'])$. Also let I be a set such that i is in I if $h[t_i] > h[t']$ and $T[t_i]$ has enough holes to put all $T[t_i']$ (i.e. $a[t'] \leq \#(Sh[t_i])$). Then $Sp[A,t'] = \sum_{f\in F} 2^{h[f]} + \sum_{i\in I} 2^{h[t_i]}$.

Informally we say that space to put $\bar{t}$ is created by distributing t' over A if $h[t'(A) + \bar{t}] < h[(t'\bar{A})^d + \bar{t}]$. Then the procedure is called space filling. Now we study how much we can reduce tree height by space filling. Let $B = \sum_i (t_i')(A_i) + \sum_j \bar{t}_j$ and assume that by distributing t_i' over A_i space can be created for all i.

Lemma 9:

Let $B = \sum_i (t_i')(A_i) + \sum_j \bar{t}_j$ and $B^d = \sum_i (t_i'\bar{A}_i)^d + \sum_j \bar{t}_j$. Then $h[B^d] = h[B] - 1$ if $\sum_i Sp[A_i,t_i'] \geq a[B] - e[B]/2$, otherwise $h[B^d] = h[B]$.

Proof:

First note that to lower the height of a tree for B, some terms must be removed from B so that an effective length of a resultant expression becomes e[B]/2. Hence $\Sigma Sp[A_i,t_i']$ must be greater than or equal to a[B] - e[B]/2. Next we show that $h[B^d]$ cannot be smaller than h[B] - 1.

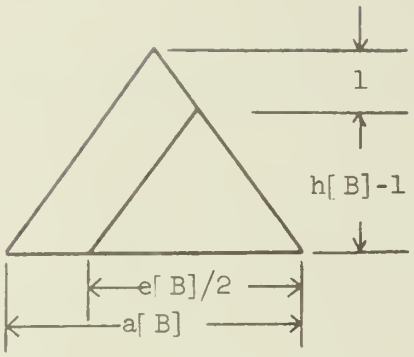


Figure 3.12. Tree Height Reduction by Hole Creation

As before we assume that $h[t_i'(A_i)] = h[(t_i'\bar{A}_i)^d]$ for all i (see lemma 8(1)). It is equivalent to the assumption that $Sp[A_i,t_i'] < e[t_i'(A_i)]/2 = e[(t_i'\bar{A}_i)^d]/2$. First we get B' from B as follows. We replace every $(t_i'\bar{A}_i)^d$ in B^d by a product P_i of $e[(t_i'\bar{A}_i)^d]/2$ single variables. This amounts to assuming that $Sp[A_i,t_i'] = e[(t_i'\bar{A}_i)^d]/2$. Futher we get B'' from B' by replacing every $\bar{t}_j$ in B' by a product Q_j of $e[\bar{t}_j]/2$ single variables. Then it is clear that $h[B] \geq h[B^d] \geq h[B'] \geq h[B''] = h[B] - 1$. If $\Sigma Sp[A_i,t_i'] \geq a[B] - e[B]/2$, then $h[B] > h[B^d]$. Hence $h[B^d] = h[B] - 1$.

(Q.E.D.)

What the lemma implies is that by space filling we can lower tree height at most by one. In other words to see if the space creation by distribution is effective it is enough to see if total tree height can be lowered by one, and we know if tree height is once lowered by one it is not necessary

(i.e. useless) to try to lower tree height further by creating more space by further distribution.

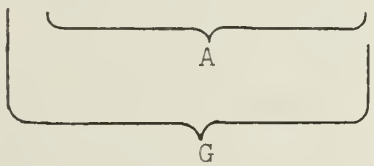
Unlike a set of holes (Theorem 2), the space function for $A = \Sigma t_i$ does not carry any information about space in components of A, t_i. For example let

$$B = a(b(c+defg) + \pi 16) + \pi 16 + \pi 4 = a(A) + \pi 16 + \pi 4$$

where πi is a product of i single variables. Then h[B] = 7. Now Sh[A] = $\emptyset$ and space creation is tried. Note that Sp[A.a] = 16 < a[B] - e[B]/2 = 20, but Sp[c+defg,ab] = 4. That is, a as well as b should be distributed over c+defg. Thus we get B' = abc + abcdefg + a(π16) + π16 + π4 where h[B'] = 6.

Now this situation is studied in detail. In general we have a form:

$$F = \ldots + t'(\ldots + t''(C) + \ldots + D + \ldots) + \ldots + E + \ldots$$



Then if Sp[A,t'] is not enough to reduce the tree height h[F], we have to further check components of A, e.g. Sp[C,t't"]. As we will show later (see Substep 2 of Step 2 of Algorithm given in Section 3.4.1) an arithmetic expression is examined from the inner most pairs of parentheses to the outer most pair. In the above diagram, the distribution of t" over C is first checked to see if it reduces the tree height h[A] and then the distribution of t' over A is examined. If the distribution of t" over C creates space and reduces the tree height h[A], then there is no problem. However if that distribution does not lower the tree height h[A], then t" will not be distributed over C (see Algorithm). As we showed in the above example when we check the possibility of reducing the tree height h[F] by creating space by the distribution of t' over A, it may be necessary to check Sp[C,t't"] as well.

Let $A' = \ldots + (t''\overline{C})^d + \ldots + D + \ldots$, $G' = (t'\overline{A})^d$ and $G'' = (t'\overline{A'})^d$. Now we show that if $Sp[A,t'] = 0$, then it is not necessary to examine components of A, e.g. $Sp[C,t't'']$ further. This helps to reduce the number of checks required.

Lemma 10:

(1) If $Sp[A,t'] = 0$, then $h[\ldots+G''+\ldots+E+\ldots] < h[\ldots+G'+\ldots+E+\ldots]$ never holds.

(2) If $Sp[A,t'] = 0$, then $h[G''] < h[G']$ never holds.

Proof:

(1) We prove this by showing that if $Sp[A,t'] = 0$ then $Sp[A',t'] = 0$. Note that $Sp[A,t'] = 0$ implies that either

(i) $h[t'] > h[t''(C)]$

or (ii) $h[t''] = h[C]$ (Definition 9).

By Lemma 9, in either case we get $h[(t'(\overline{t''\overline{C})^d})^d] > h[t't''(C)]$. Note that the only difference between G' and G'' is that a term $t't''(C)$ in G' is being replaced by $(t'(\overline{t''\overline{C})^d})^d$ in G''. Since $h[(t'(\overline{t''\overline{C})^d})^d] > h[t't''(C)]$, G'' cannot have more free nodes than G'. Hence $Sp[A',t'] = 0$.

(2) This may be proved in a similar way and the details are omitted.

(Q.E.D.)

Thus in $F = \ldots + t'(\ldots+t''(C)+\ldots+D+\ldots) + \ldots + E \ldots$ the distribution of t'' over C should be done if it reduces the tree height of T[A], otherwise it should be left untouched. In the latter step when the distribution of t' over A is examined, the possibility of distributing t'' over C as well shall be checked if and only if $Sp[A,t'] \neq 0$. Otherwise we shall leave t'(A)

as it is and we need not check inside of A again.

3.4 Algorithm

Having these results, an algorithm to reduce tree height of an arithmetic expression is now described. Given an arithmetic expression, the algorithm works from the inner most pairs of parentheses to the outer most pair. We assume that cases (P_1) and (P_2) (see Lemma 2) are already taken care of. At each level of a parenthesis pair, first upon finding a form t'(A), a hole of A is tried to be filled by t' (Theorem 2). After all holes are filled, a form t'(A) + t" is checked, i.e. if the distribution of t' over A creates enough space to accomodate t", then the distribution is made. Note that it is not necessary to fully distribute $(A)(B) = (\Sigma t_i)(\Sigma t_i')$ (see Lemma 3).

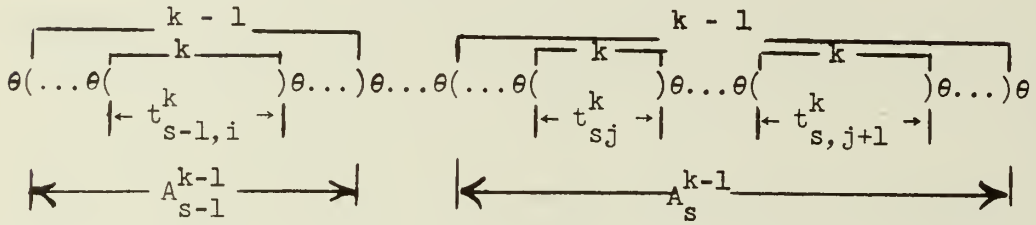
It is not necessarily true that reducing tree height of a term t of an arithmetic expression A reduces tree height of A. However, we show that reduction of tree height should be made in any case to help later steps of the distribution algorithm.

Let $A = \sum_{i=1}^{n-1} t_i + t_n$ (or $\prod_{i=1}^{n-1} (t_i) \times (t_n)$). Assume that the distribution algorithm somehow reduced the height of $T[t_n]$, i.e. the distribution algorithm reduced A to $A' = \sum_{i=1}^{n-1} t_i + t_n'$ (or $\prod_{i=1}^{n-1} (t_i) \times (t_n')$) where $h[t_n] > h[t_n']$. Also assume that h[A] = h[A']. Yet it is obvious that #(Sh[A]) $\leq$ #(Sh[A']) (i.e. le Sh[A] $\leq$ le Sh[A']) and also for any t", Sp[A,t"] $\leq$ Sh[A,t"]. That is, even if distribution only reduces the tree height $h[t_n]$ and does not reduce the tree height h[A], that distribution does not cause any bad effect on the later steps when A appears as a term of a bigger expression Q with respect to holes and spaces.

The arithmetic expression thus obtained may give the lowest tree height, i.e. the fewest number of computation time steps.

3.4.1 Distribution Algorithm

In the steps below we refer to the notation:



$$A^{k-1}_s = \ldots + t^k_{s,j-1} \prod_{h=j}^{j+n} (t^k_{sh}) + \ldots$$

where $t^k_{s,j-1} = \prod_{\ell=1}^{p} a_\ell$ or empty.

Step 1:

Go to an arithmetic expression enclosed in an innermost parenthesis pair which is not checked. Let this level be k-1. In the above diagram we are now working on, say, A^{k-1}_s.

Step 2:

Obtain a set of holes for all t^k_{sj} which are enclosed in the k-th parenthesis pair and are components of A^{k-1}_s, as well as their heights h and effective lengths e.

Step 3:

In this step, the (k-1)th parenthesis pair level A^{k-1}_s is examined.

Substep 1: Hole filling (see Theorem 2)

Let

$$A_s^{k-1} = \ldots + \underbrace{t^k_{s,j-1} \times \pi_{h=j}^{j+n} (t^k_{sh})}_{t} + \ldots$$

where $t^k_{s,j-1} = \pi_{\ell=1}^{p} a_\ell$ or empty. Also without loss of generality we assume that $e[t^k_{s\ell}] \leq e[t^k_{s,\ell+1}]$ for $\ell = j, j+1, \ldots, j+n-1$.

(1) Find an occurrence of a form $\pi(t_j)$ or $\pi a_i \times \pi(t_j)$ in A_s^{k-1}. If there is no such occurrence, then go to Substep 2. If an occurence of a form $\pi(t_j)$ is found, then skip (2) and (3). Otherwise go to (2).

(2) Suppose we find t in A_s^{k-1} as an instance of $\pi a_i \times \pi(t_j)$.

Fill holes in $Sh[t^k_{sh}]$ $(h=j, j+1, \ldots, j+n)$ using a_i's in t^k_{sh}. If there are many holes to be filled in, fill the smallest ones first, i.e. in order of increasing size. Reevaluate Sh by Lemma 8 for those t^k_{sh} whose holes are filled.

(3) If $t^k_{sh}(h=j, j+1, \ldots, j+n)$ do not have enough holes to accomodate all a_i's then go back to (1) to find out another occurrence of $\pi(t_j)$ or $\pi a_i \times \pi(t_j)$ form. Otherwise we work on $\pi_{h=j}^{j+n} (t'^k_{sh})$ which we get from

$$\pi_{\ell=1}^{p} a_\ell \times \pi_{h=j}^{j+n} (t^k_{sh})$$ after (2).

(4) We start from h = j. Check if t'^k_{sh} can fill in one of holes in

$Sh[t'^{k}_{s\ell}]$ (ℓ=h+1,...,j+n). If there are many holes which can accomodate t'^{k}_{sh}, fill them in order of increasing size. Continue the procedure until all t'^{k}_{sh}(h=j+1,...,j+n-1) are put in some holes or there is no hole to accomodate t'^{k}_{sh}. Go back to (1) to find out another occurrence of $\pi(t_j)$ or $\pi a_i \times \pi(t_j)$ form.

Substep 2: Space filling

After Substep 1, we again check A^{k-1}_s, where all holes in t^k_{sj} have been filled in as much as possible by Substep 1.

(1) Let Ex = $a[A^{k-1}_s] - e[A^{k-1}_s]/2$ (see Lemma 9).

(2) Let

$$A^{k-1}_s = \ldots + \underbrace{t^k_{s,j-1} \times \prod_{h=j}^{j+n-1} (t^k_{sh})}_{t'} \times \underbrace{(t^k_{s,j+n})}_{t} + \ldots$$

where $t^k_{s,j-1} = \prod_{\ell=1}^{p} a_\ell$ or empty. We also assume that $e[t^k_{s,j+n}]$ is the largest among all $e[t^k_{sh}]$ (h=j,...,j+n). Let $t' = t^k_{s,j-1} \times \prod_{h=j}^{j+n-1} (t^k_{sh})$. If $h[t'] < h[t^k_{s,j+n}]$, then evaluate $Sp[t^k_{s,j+n}, t']$. Otherwise leave it as it is.

(3) Repeat (2) for every occurrence of a form $\pi(t_j)$ (or $\pi a_i \times \pi(t_j)$) in A^{k-1}_s. Assume that there are m such occurrences. Arrange all $Sp[t,t']$ in order of decreasing size. For convenience we write $Sp_1, Sp_2, \ldots, Sp_m$($Sp_i \geq Sp_{i+1}$).

(4) If $\sum_{i=1}^{m} Sp_i \geq Ex$ then let d be such that $\sum_{i=1}^{d-1} Sp_i < Ex$ and $\sum_{i=1}^{d} Sp \geq Ex$.

(5) Let $\underbrace{t^k_{s,j-1} \times \prod_{h=j}^{j+n-1} (t^k_{sh})}_{t'} \times \underbrace{(t^k_{s,j+n})}_{t}$ be a form which corresponds to $Sp_i (i \leq d)$. Then distribute t' over t, and create space Sp_i.

Repeat the same procedure for all i = i, 2, ..., d.

(6) In the case where enough space to accomodate Ex (i.e. $\sum_{i=1}^{m} Sp_i < Ex$) is not found, a check is made against the component terms of $t^k_{s,j+n}$ (see Lemma 11).

For example let

$$t^k_{s,j-1} = a_1 a_2 \cdots a_p, \quad n = 1$$

and $$t^k_{s,j} = b_1 b_2 \cdots b_q (t^{k+1}_{sf}) + \sum_{\substack{i=1 \\ i \neq f}}^{m} t^{k+1}_{si}.$$

Then

$$A^{k-1}_s = \cdots + t^k_{s,j-1} \times (b_1 b_2 \cdots b_q (t^{k+1}_{sf}) + \sum_{\substack{i=1 \\ i \neq f}}^{m} t^{k+1}_{si}) + \cdots.$$

Then the distribution is done if the sum of Sp_i and $Sp[t^{k+1}_{sf}, t^k_{s,j-1} \times b_1 \cdots b_q]$ is greater than or equal to Ex. Here the distribution of $a_1 \cdots a_p b_1 \cdots b_q$ over Σt^{k+1}_{si} is to be made as well as the distribution of $a_1 \cdots a_p$ over t^{k+1}_{sf}. This checking is to

be made until enough space to accomodate Ex is found or else until the innermost level of parenthesis pair is reached.

Step 4:

Mark A_s^{k-1} as checked.

Step 5:

If all levels are checked, then stop, otherwise go back to Step 1.

For example let us consider the following

$$A = \ldots + \underbrace{a_1a_2a_3(t_1)(t_2)(t_3)}_{t} + \ldots.$$

Further assume that

$$Sh[t_1] = \{4\}, \quad e[t_1] = 16$$

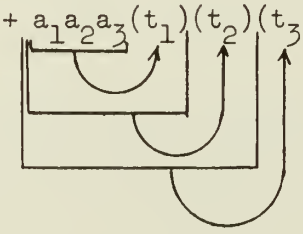
$$Sh[t_2] = \{16,2\}, \quad e[t_2] = 64$$

and

$$Sh[t_3] = \emptyset \qquad e[t_3] = 64.$$

Then $a_1a_2a_3$ can be distributed over t_1, and in turn this whole thing can be distributed over t_2

$$\ldots + a_1a_2a_3(t_1)(t_2)(t_3) + \ldots$$



and we get $h[t'] = 7$ whereas $h[t] = 8$.

3.4.2 Implementation

A few words about implementation of the algorithm described above are given as well as the total number of checks required to process an arithmetic expression. Suppose we are given the aritmetic expression

$$A = \ldots + (\pi 2+\pi 1)(\pi 10+(\pi 4+\pi 1)(\pi 17+\pi 2)+\pi 3) + \ldots$$

$$= \ldots + (d_1+d_2)(e_{11}+(e_{21}+e_{22})(e_{31}+e_{32}) + e_{41}) + \ldots$$

$$= \ldots + (D)(E_1+(E_2)(E_3) + E_4) + \ldots$$

$$= \ldots + (D)(E) + \ldots$$

where πi represents a product of i single variables. Then we build nested stacks as shown in Figure 3.13(a). Note that a new stack is created for each form $\pi(t_i)$ or $\Sigma\, t_i$.

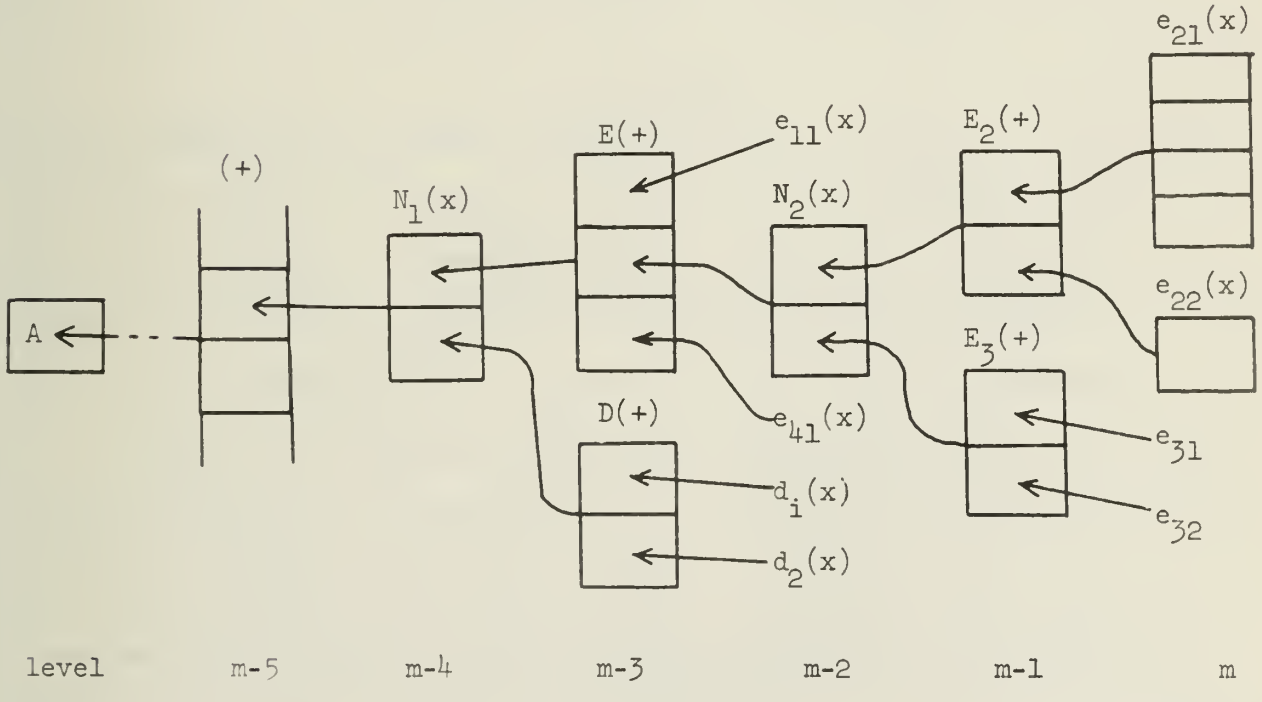
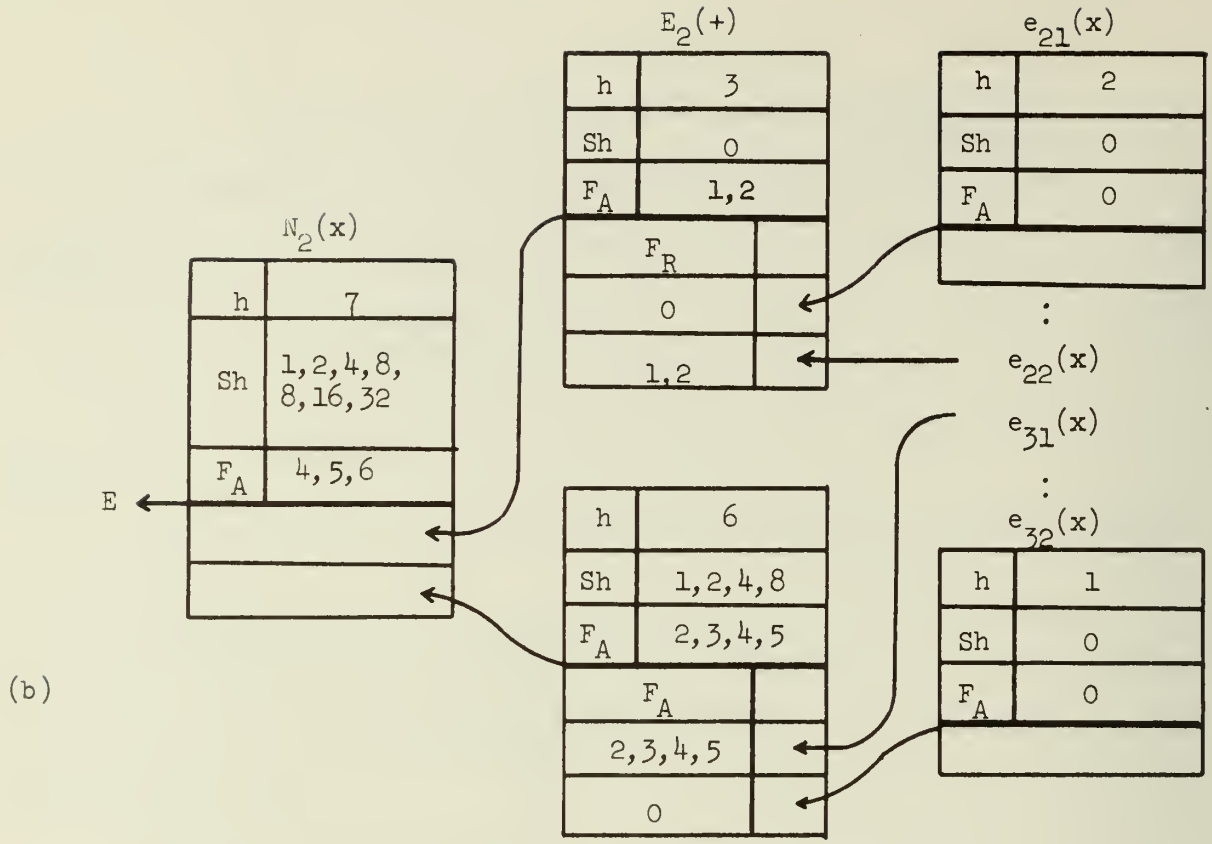


Figure 3.13. Stacks for an Arithmetic Expression



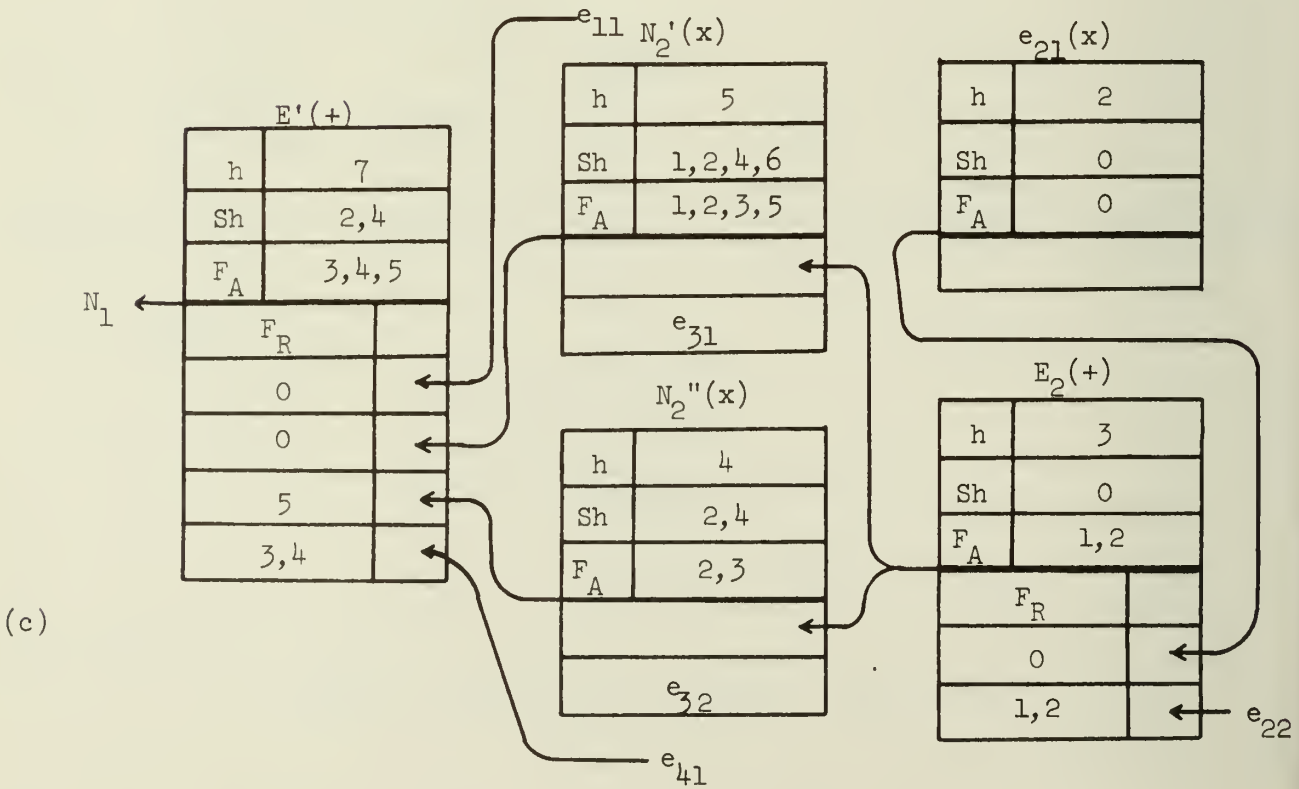


Figure 3.13. (continued)

Each stack is assigned a level number (cf. Definition 3) where the first stack which corresponds to A receives the level number 0 (Figure 3.12(a)).

We start working from a stack with the largest level number, say m. For each stack t, where $t = \Sigma\, t_i$ or $t = \pi(t_i)$, h[t] is evaluated. Also if a stack represents a form $t = \Sigma\, t_i$, then Sh[t], $F_A[t]$ and $F_R[t,t_i]$ are evaluated. If a stack represents a form $t = \pi(t_i)$, then Sh[t] and $F_A[t]$ are evaluated. These values are obtained by Definitions 1, 4 and 8. Note that this information is sufficient to evaluate Sp. Figure 3.12(b) gives an illustration.

Upon finding a form $\pi(t_i)$ (or $\pi a_j \times \pi(t_i)$) (e.g. the stack N_2), we apply the distribution algorithm and decide if distribution is to be made. If a stack represents a form $t = \pi(t_i)$, then Substep 1 of the distribution algorithm, i.e. hole filling, is tried. Otherwise a stack represents a form $t = \Sigma\, t_i$ and Substep 2 of the distribution algorithm, i.e. space filling is applied. In our example E_2 is distributed over $E_3(e[E_2] \leq \underline{le}\ Sh[E_3])$. Then stacks are revised as shown in Figure 3.12(c). Note that the stack N_2 is replaced by two new stacks N_2' and N_2'', and the stack E_2 disappears.

If all stacks with the level number k have been checked, then stacks with the level number k-1 will be checked. In our example, stacks E(or E' since it has been revised) and D are now checked.

The total number of checks required to process a whole arithmetic expression thus depends on the number of parenthesis occurrences in it. Assume that there are p parenthesis pairs in an arithmetic expression A. For each pair, space creation should be examined. Hence in total p space creation checks are required. Now for each $\pi(t_i)$ form hole filling should be tried. The number of

occurrences of a form $\pi(t_i)$ in A is obviously less than p. Hence the total number of checks required is less than 2p (i.e. the order of p).

3.5 Discussion

3.5.1 The Height of a Tree

Given a tree for an arithmetic expression, the distribution algorithm tries to lower tree height by distribution if possible. However, in general it may not give the minimum tree height. For example let A = ac + ad + bc + bd whose tree height is 3, and since no further distribution is possible, the distribution algorithm yields the same value. There is, however, an equivalent expression A' = (a+b)(c+d), whose tree height is lower than 3, i.e. 2. That is, even though factorization lowers tree height sometimes, the distribution algorithm does not take care of it.

The question we ask now is how much the distribution algorithm lowers tree height. Before giving an answer to this question let us study tree height in more detail. Given an arithmetic expression, Theorem 1 gives the exact height of a tree obtained by Bovet and Baer's algorithm. It is also desirable if we can get an approximate tree height without actually building a tree for an arithmetic expression. Since the number of single variable occurrences (the number one less than this gives the number of operators in an arithmetic expression) and the depth of parenthesis nesting may well represent the complexity of arithmetic expressions, let us try to approximate tree height in terms of them.

Let A be an arithmetic expression with n single variable occurrences and depth d of parenthesis nesting. Now build a tree for A by Bovet and Baer's algorithm. Then it can be proved that:

Lemma 11:

$$\log_2 \lceil n \rceil_2 \leq h[A] \leq n - 1.$$

Moreover we can prove the following theorem.

Theorem 3:

$$h[A] \leq 1 + 2d + \log \lceil n \rceil_2 .$$

The following lemma is helpful to prove Theorem 3.

Lemma 12:

(1) $2a > \lceil a \rceil_2$

(2) $\lceil 2a \rceil_2 = 2 \lceil a \rceil_2$

(3) $$\log \left\lceil \sum_{i=1}^{p} \lceil m_i \rceil_2 \right\rceil_2 < \log \left(2 \cdot \left\lceil \sum_{i=1}^{p} m_i \right\rceil_2 \right).$$

Proof:

(1) and (2) are obvious and (3) can be proved by (1) and (2).

(Q.E.D.)

Proof of Theorem 2:

Proof is given by induction on d. First let us prove the theorem for d = 0. Then A has the following pattern:

$$A = \sum_{i=1}^{p} \prod_{j=1}^{q_i} a_j^i .$$

Then by Theorem 1,

$$h[A] = \log \left\lceil \sum_{i=1}^{p} \lceil q_i \rceil_2 \right\rceil_2$$

$$< \log\left(2 \cdot \left\lceil \sum_{i=1}^{p} q_i \right\rceil_2\right) \text{ by Lemma 5(3)}$$

$$= 1 + \log\lceil n\rceil_2.$$

Now assume that the theorem holds for $d \leq f$.

Let t_j^i be an arithmetic expression with depth $d_j^i (\leq f)$ parenthesis nesting and n_j^i single variable occurences. Then by assumption $h[t_j^i] \leq 1 + 2d_j^i + \log\lceil n_j^i\rceil_2$. Now an arithmetic expression with $f + 1$ parenthesis nesting can be built from t_j^i as follows:

$$A = \sum_{i=1}^{p} \left(\prod_{j=1}^{q_i} a_j^i \prod_{k=1}^{m_i} (t_k^i) \right).$$

where a_j^i are single variables and at least one of t_j^i has f nested parentheses. Now each t_j^i can be replaced with a product of $e[t_j^i]$ single variables without affecting the total tree height. Instead of using the value $e[t_j^i]$, let us use the value $2 \cdot 2^{2d_j^i} \cdot \lceil n_j^i\rceil_2$. (Note that $h[t_j^i] \leq 1 + 2d_j^i + \log\lceil n_j^i\rceil_2 = \log(2 \cdot 2^{2d_j^i} \cdot \lceil n_j^i\rceil_2)$ and $e[t_j^i] = 2^{h[t_j^i]} \leq 2 \cdot 2^{2d_j^i} \cdot \lceil n_j^i\rceil_2$.) Since $d_j^i \leq f$, $e[t_j^i] \leq 2 \cdot 2^{2f}\lceil n_j^i\rceil_2$. Now from Theorem 1, we have

$$h[A] \leq \log \left\lceil \sum_{i=1}^{p} \left\lceil \lceil q_i\rceil_2 + \sum_{j=1}^{m_i} 2 \cdot 2^{2f \lceil n_j^i\rceil_2} \right\rceil_2 \right\rceil_2$$

$$\leq \log 2 \left\lceil \sum_{i=1}^{p} \lceil q_i + \sum_{j=1}^{m_i} 2 \cdot 2^{2f} n_j^i \rceil_2 \right\rceil_2$$

$$\leq \log\left(2\cdot 2\cdot 2\cdot 2^{2f}\left\lceil \sum_{i=1}^{p}(q_i + \sum_{j=1}^{m_i} n_j^i)\right\rceil_2\right)$$

$$= 1 + 2(f+1) + \log\lceil n\rceil_2.$$

Thus the theorem holds for d = f + 1 and this proves the theorem.

(Q.E.D.)

Now let us examine the original question i.e. how effective is the algorithm presented in this chapter. Let A and A^d be arithmetic expressions where A^d is the resultant expression obtained from A after the application of the distribution algorithm. Now build trees for A and A^d by Bovet and Baer's algorithm. Then it should be clear that $h[A] \geq h[A^d]$. Moreover experience suggests that:

Conjecture:

$$h[A^d] \leq 2\ \log_2\lceil n\rceil_2$$

where n is the number of single variable occurrences in the original arithmetic expression A.

Note that the distribution algorithm speeds up a Horner's rule polynomial in a logarithmic way. Also note that the distribution algorithm does no distributions in the case of

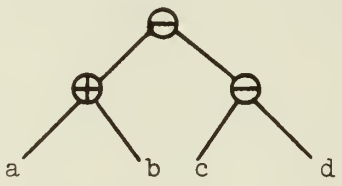
$$\prod_{i=1}^{n}\left(\sum_{j=1}^{n} x_{ij}\right)$$

which takes $2\log\lceil n\rceil_2$ steps as it is presented but which would take $(n+1)\ \log\lceil n\rceil_2$ steps if fully distributed. Thus the algorithm can save a factor of n/2 steps over a scheme which would distribute indiscriminately and in some cases achieves a logarithmic speed up.

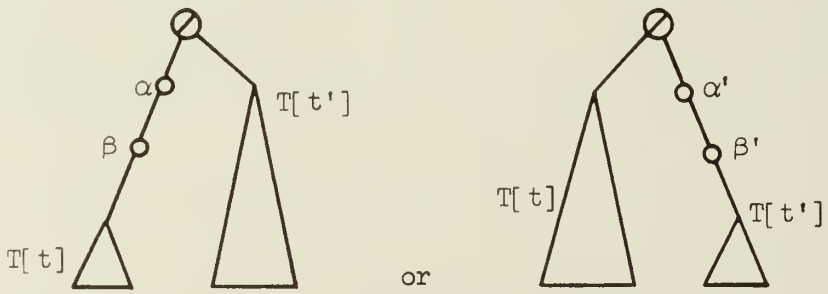
3.5.2 Introduction of Other Operators

3.5.2.1 Subtraction and Division

Subtractions can be introduced into an arithmetic expression without causing any effect on the distribution algorithm. It may be necessary to change operators to build a minimum height tree. For example let A = a + b - c + d. This will be computed as A = a + b - (c-d):

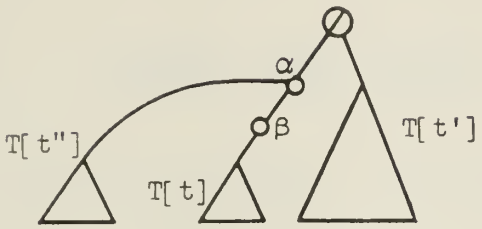


Divisions may require special treatment since the distributive law does not hold in certain cases, e.g. (a+b+c)/d = a/d + b/d + c/d but a/(b+c+d) ≠ a/b + a/c + a/d. Hence in general minimization of the height of trees for a numerator and a denominator is tried independently, and then distribution of a denominator over a numerator is tried if appropriate. Also let A = t/t'. Then T[A] is built from T[t] and T[t'] as follows:



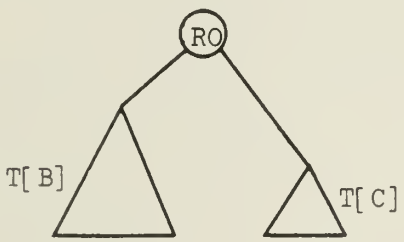
If h[t] ≠ h[t'], then we get nodes to which only one tree is attached, e.g. α and β if h[t] < h[t'], and α' and β' if h[t] > h[t']. Then α and β are treated as free nodes in T[A] while α' and β' are <u>not</u> treated as free nodes in T[A], because later when another expression, say t", is multiplied to A, t must

be multiplied by t" not t', i.e. t"(A) = t"(t/t') = (t"t)/t' ≠ t/(t't"):

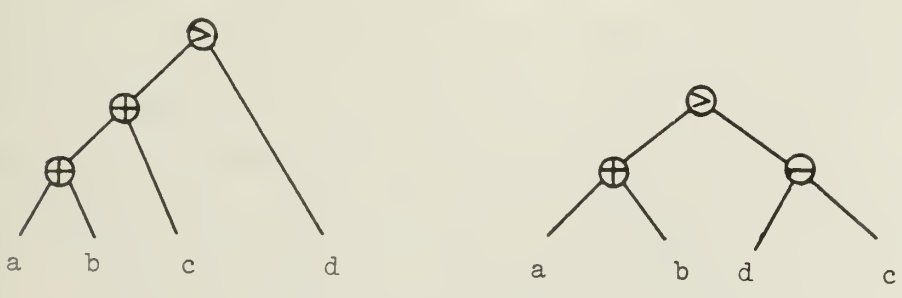


3.5.2.2 Relational Operators

If an arithmetic expression A contains relational operators e.g. A = B RO C where RO = {>, <, =, ≥, ≤, ...}, trees can be built for B and C independently:



If h[B] ≠ h[C], then operators may be moved from one side to the other to balance two trees. For example let A = a + b + c > d. Then we modify this as A' = a + b > d - c and get h[A'] = 2 while h[A] = 3:



4. COMPLETE PROGRAM HANDLING

Chapter 3 presented the algorithm which reduced tree height for a single arithmetic expression by distributing multiplications over additions properly. In this chapter we will discuss some ideas about how to handle complete programs, i.e. given one program, how can it be executed in the shortest time by building a tree as well as executing a statement in a for statement simultaneously for all index values. Ideas include back substitution. We do not have the solution to the problem, but this chapter presents some details of the problem and some ways to attack them.

We conclude this chapter by comparing serial and parallel computations in terms of a generated error. It is shown that in general we could expect less error from parallel computation than serial computation. It is also shown that distribution would not increase the size of an error significantly.

4.1 Back Substitution - A Block of Assignment Statements and an Iteration

While the distribution algorithm in the previous chapter discusses tree height reduction for a single arithmetic expression, it can be used for any jump free block of assignment statements. If we define those variables which appear only on the right hand sides of assignment statements or in read statements in a block as inputs to the block, and those variables which appear only on the left hand sides of assignment statements or in write statements as outputs from the block, then we can rewrite the block with one assignment statement per output by substitution of assignment statements into one another. For example

a := b + c;

d := e × f;

g := a + c;

h := a + g × d

can be rewritten as

g := (b+c) + c;

h := (b+c) + ((b+c) + c) × (e×f).

After such a reduction only input variables appear on the right hand sides of assignment statements. At this point, the distribution algorithm could be applied to each remaining assignment statement and if sufficient computer resources were available, all of the reduced assignment statements could be executed at once. In the above example if each statement is computed in parallel (by building a tree) independently then 5 steps are required, while if the back substitution is done then the computation requies only 4 steps. Suppose we have assignment statements, $A_1, A_2, \ldots, A_n$. Also suppose that by back substitution we can rewrite this block as A. We build minimum height trees for $A_1, A_2, \ldots, A_n$ and A. Now we apply the distribution algorithm on those trees. Let the resultant tree heights be $h[A_1], \ldots, h[A_n]$ and h[A]. Then obviously $h[A_1] + \ldots + h[A_n] \geq h[A]$, i.e. back substitution <u>never</u> increases the computation time (in the sense of tree height) (Figure 4.1).

Our main interest here is the case where strict inequality in the above relation holds, because that $h[A_1] + \ldots + h[A_n] > h[A]$ holds is equivalent to a speed up of the computation by back substitution. Note that back substitution amounts to symbol manipulation (i.e. replacement) and should not be confused with arithmetic simplification. For example from

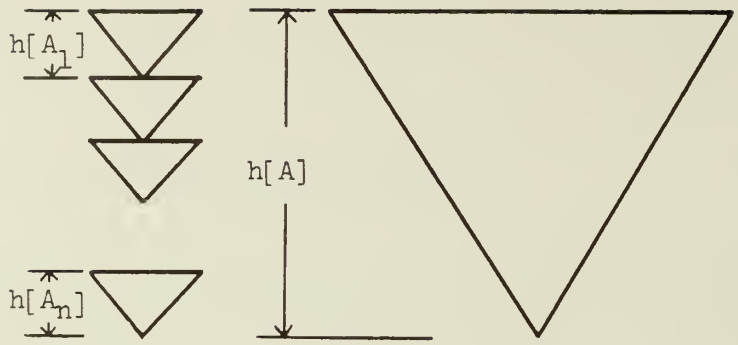


Figure 4.1. A Back Substituted Tree

a := x + y

b := a + y

we get

b := (x+y) + y

or b := x + y + y

but we do not get

b := x + 2y .

Now we shall study this kind of speed up.

We shall discuss a limited class of assignment statements, i.e. an iteration. This may serve to give some insight to the problem of speed up by back substitution in the general assignment statement case.

By an iteration we mean a statement

$y_i = f(y_{i-1})$.

Usually a statement is executed repeatedly for I = 1,2,...,n. An example is:

for I := 1 step 1 until 10 do

A[I] := A[I-1] + A[I];

Also a block of assignment statements such as:

S_1: a := h + i + j;

S_2: b := a + k + m;

S_3: c := b + n + p;

S_4: d := c + q + r;

falls into this category (note that all statements have a form

output of S_i := output of S_{i-1} + x + y

where x and y are pure inputs in the sense that they do not appear as outputs).

Assume that we are only interested in the value of y_n (the other results, i.e. y_{n-1}, y_{n-2}, ..., y_1 may be obtained similarly to y_n but in less time). Then instead of n statements, i.e. $y_1 = f(..)$, $y_2 = f(..),\ldots,y_n = f(..)$, we may obtain one statement for y_n by back substitution. For example, let $y_i = a_{i-1}y_{i-1}$. Then $y_n = a_{n-1}y_{n-1} = a_{n-1}(a_{n-2}y_{n-2}) = a_{n-1}(a_{n-2}(a_{n-3}y_{n-3}))$ $= \ldots = y_0 \prod_{k=0}^{n-1} a_k$. We use the superscript "b" to distinguish the back substituted form from an iteration form, e.g. $y_n = a_{n-1}y_{n-1}$ and $y_n^b = y_0 \prod_{k=0}^{n-1} a_k$. Then instead of computing each y_i repeatedly for i = 1,2,..,n, y_n^b may be computed directly. In the above example y_i can be computed in one step, and to get y_n n steps are required while y_n^b can be computed in $\lceil \log n \rceil$ steps in parallel (i.e. by building a tree for y_n^b). The following table summarizes the results for some primitive yet typical iteration formulas.

y_i	y_n^b	T_s	nT_s	T_p
ay_{i-1}	$a^n y_0$	1	n	$\lceil \log_2(n+1) \rceil$
$y_{i-1} + b$	$y_0 + \overbrace{b + \ldots + b}^{n}$	1	n	$\lceil \log_2(n+1) \rceil$
$a_{i-1} y_{i-1}$	$\prod_{k=0}^{n-1} a_k y_0$	1	n	$\lceil \log_2 n \rceil$
$y_{i-1} + a_{i-1}$	$\sum_{k=0}^{n-1} a_k + y_0$	1	n	$\lceil \log_2 n \rceil$
$ay_{i-1} + b$	$a^n y_0 + p_{n-1}(a)$ *	2	2n	$\approx 2\lceil \log_2(n+1) \rceil$
$ay_{i-1} + x_{i-1}$	$p_n'(a)$ **	2	2n	$\approx 2\lceil \log_2(n+1) \rceil$
$y_{i-1} + bx_{i-1}$	$\sum_{k=0}^{n-1} bx_k + y_0$	2	2n	$\approx \lceil \log_2 n \rceil + 1$
$ay_{i-1} + bx_{i-1}$	$p_n''(a)$ ***	3	3n	$\approx 2\lceil \log_2(n+1) \rceil$

* $p_{n-1}(a) = ba^{n-1} + ba^{n-2} + \ldots + b$

** $p_n'(a) = a^n y_0 + a^{n-1} x_0 + a^{n-2} x_1 + \ldots + a x_{n-2} + x_{n-1}$

*** $p_n''(a) = a^n y_0 + ba^{n-1} x_0 + ba^{n-2} x_1 + \ldots + bax_{n-2} + bx_{n-1}$

T_s: The time required to compute y_i in parallel, i.e. $h[y_i]$.

T_p: The time required to compute y_n^b in parallel, i.e. $h[y_n^b]$.

Table 4.1. Comparison of Back Substituted, y_n^b, and Non-Back Substituted Computation, y_i--Iteration Formulas

From Table 4.1, the following lemma is obtained by exhaustion.

Lemma 1:

Let $y_i = f(y_{i-1})$ be linear in y_{i-1} where we assume that in the presented form additions are reduced to multiplications as much as possible, e.g. $y_i = 2y_{i-1}$ instead of $y_i = y_{i-1} + y_{i-1}$. Then $n \times h[y_i] > h[y_n^b]$.

Thus if we have enough PE's, then instead of computing each y_i repeatedly for $i = 1,2,\ldots,n$, we should obtain y_n^b by back substitution and compute it by building a minimum height tree.

If an iteration $y_i = f(y_{i-1})$ is not linear in y_{i-1}, e.g. $y_i = a\, y_{i-1}^3 + b\, y_{i-1}^3 + c$, or if it is linear in y_{i-1} but there are some additions not being reduced to multiplications, e.g. $y_i = y_{i-1} + a\, y_{i-1}$, then it is not clear if back substitution speeds it up. For example, back substitution does not speed up the computation of $y_i = y_{i-1} + y_{i-1} + y_{i-1} + y_{i-1}$. Also let $y_i = f(y_{i-1})$ be a polynomial of y_{i-1} where in the presented form additions are again reduced to multiplications as much as possible. Then it is not likely that we can speed up the computation by back substitution. Let

$$f(y_{i-1}) = a_m\, y_{i-1}^m + \cdots$$

where y_{i-1}^m is the highest power of y_{i-1} among those which appear in $f(y_{i-1})$. Note that $f(y_{i-1})$ is not necessarily a dense polynomial (a polynomial in which all powers of y_{i-1}, i.e. y_{i-1}, y_{i-1}^2, $\ldots$, y_{i-1}^m, appear). While the exact height of $T[f(y_{i-1})]$ depends on $f(y_{i-1})$, we may content ourselves with (see Chapter 2)

$$h[f(y_{i-1})] \approx 2\lceil \log_2 m \rceil .$$

Hence $2n\lceil \log_2 m \rceil$ steps are required to compute y_n.

Now let us consider y_n^b. Then

$$\begin{aligned} y_n^b &= a_m (y_{n-1}^b)^m + \ldots \\ &= a_m (a_m (y_{n-2}^b)^m + \ldots)^m + \ldots \\ &\quad \ldots \\ &= a_m (a_m (\ldots a_m y_0^m \ldots)^m \ldots)^m + \ldots \\ &= a_m^{n-1} m^n y_0 + \ldots . \end{aligned}$$

That is, y_n^b becomes a polynomial of y_0 of degree m^n. Leaving the computation of a_m^{n-1} out of consideration, we have (see Chapter 2)

$$\begin{aligned} h[y_n^b] &\approx 2\lceil \log_2 m^n \rceil \\ &\approx 2n\lceil \log_2 m \rceil . \end{aligned}$$

Hence back substitution does not help to speed up computation significantly in this case.

To gain a better understanding of more general cases, let us study the situation from a different point of view. Given an iteration $y_i = f(y_{i-1})$, let us consider the number of single variable occurrences in y_n^b as a measure of the complexity. We study two cases separately, i.e. (i) y_{i-1} appears only once in y_i and (ii) y_{i-1} appears k times in y_i. In both cases we assume that there

are m single variable occurrences (including the occurrences of y_{i-1}) in y_i. For convenience we write N(y) for the number of single variable occurrences in y, e.g. N(a+b+cd+a-e) = 6.

(1) y_{i-1} appears only once in y_i. In this case we have

$$N(y_1) = m$$

$$N(y_2^b) = N(y_1) + m - 1 = 2m - 1$$

$$N(y_3^b) = N(y_2^b) + m - 1 = 2m - 2$$

$$\cdots$$

$$N(y_n^b) = N(y_{n-1}^b) + m - 1 = mn - n + 1 \approx mn.$$

(2) y_{i-1} appears k times in y_i. In this case we have

$$N(y_1) = m$$

$$N(y_2^b) = k \cdot N(y_1^b) + m - k = m + k(m-1)$$

$$N(y_3^b) = k \cdot N(y_2^b) + m - k = m + (k+k^2)(m-1)$$

$$\cdots$$

$$N(y_n^b) = k \cdot N(y_{n-1}^b) + m - k$$

$$= m + (k + \ldots + k^{n-1})(m-1)$$

$$= 1 + \frac{k^n - 1}{k - 1}(m-1).$$

If $k^n \gg 1$ and $m \gg 1$, then $N(y_n^b) \approx k^{n-1}m$.

Now if we use $2\lceil \log_2 N(y) \rceil$ as a measure of the height of a tree, then we have (see Section 3.5.1 of Chapter 2):

	$h[y_i]$	$n \times h[y_i]$	$h[y_n^b]$
(1)	$2\lceil \log_2 m \rceil$	$2n\lceil \log_2 m \rceil$	$2\lceil \log_2 (mn) \rceil \approx 2(\lceil \log_2 m \rceil + \lceil \log_2 n \rceil)$
(2)	$2\lceil \log_2 m \rceil$	$2n\lceil \log_2 m \rceil$	$2\lceil \log_2 (k^{n-1} m) \rceil \approx 2((n-1)\lceil \log_2 k \rceil + \lceil \log_2 m \rceil)$

Table 4.2. Comparison of Back Substituted, y_n^b, and Non-Back Substituted Computation, y_i--General Cases

For example let $m = 5$, $k = 2$ and $n = 20$ in (2). Then we have

$$n \cdot h[y_i] = 40 \cdot \lceil \log_2 5 \rceil = 120$$

and $h[y_n^b] = 2(19\lceil \log_2 2 \rceil + \lceil \log_2 5 \rceil) = 44.$

Also if we let $m = 5$ and $n = 20$ in (1), then we get

$$n \cdot h[y_i] = 40\lceil \log_2 5 \rceil = 120$$

and $h[y_n^b] = 2(\lceil \log_2 5 \rceil + \lceil \log_2 20 \rceil) = 16.$

Now a few comments about implementation are in order. As for back substitution of a block of assigment statements, the step by step substitution is the only possible scheme. In case of an iteration formula, we may use the z-transformation technique to obtain y_n^b [8]. For example let $y_i = y_{i-1} + x_i$. Then by applying z-transformation on it, we get $Y(z) = zY(z) + X(z)$ or $Y(z) = \frac{X(z)}{1 - z}$. Hence $Y(z) = X(z)(1 + z + z^2 + z^3 + \ldots)$

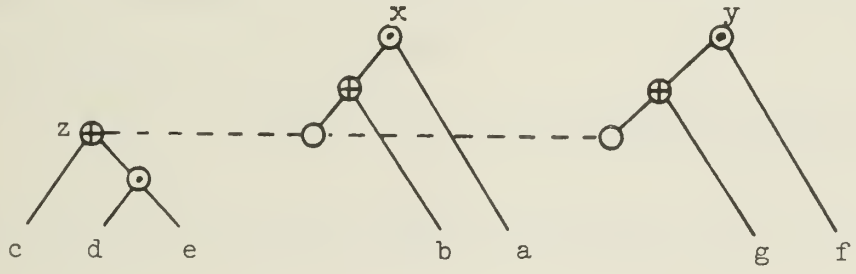
$$= (x_0 + x_1 z + x_2 z^2 + \ldots)(1 + z + z^2 + \ldots)$$

$$= x_0 + (x_1 + x_0)\, z + (x_2 + x_1 + x_0)\, z^2 + \ldots$$

or $y_i^b = \sum_{k=0}^{i-1} x_k$.

Two other related problems become evident in the example presented above. First is algebraic simplification. For example, a := b + 4c could be executed more quickly than a := b+c+c+c+c. We shall not discuss this subject further here. A second problem is the discovery of common subexpressions. In our example, (b+c) appears twice in the right hand side of it.

If we had an algorithm, e.g. [11] which discovered common subexpressions in one (or more) tree which could be simultaneously evaluated, the number of PE's required could be reduced by evaluating the common subexpressions once for all occurrences. On the other hand, by removing common subexpressions the execution time (the height of a tree) may be increased in some cases. For example, if we have x := a(b+c+de) and y := f(g+c+de), then we might try to replace c + de in x and y by z as follows to save the number of PE's required:



However, note that x = a(b+z) or y = f(g+z) takes 4 steps while the original x and y require only 3 steps, i.e. h[a(b+c+de)] = 3 and h[a(b+z)] = 4. Thus an overall strategy must be developed for the use of a common subexpression discovery algorithm in conjunction with overall tree height reduction.

4.2 Loops

This section is included here to complete this chapter, and discusses the subject superficially. Details will be presented in the following chapters.

Consider the following example.

```
E1:  for I := 1 step 1 until 10 do
     for J := 1 step 1 until 10 do
     S3:  A[I,J] := A[I,J-1] + B[J];
```

In this case ten statements, A[1,J] := A[1,J-1] + B[J], A[2,J] := A[2,J-1] + B[J], ..., A[10,J] := A[10,J-1] + B[J] can be computed simultaneously while J takes values 1,2,...,10 sequentially. We say that S3 can be computed in parallel with respect to I. Note that originally the computation of E1 takes 100 steps. (One step corresponds to the computation of S3, i.e. addition. For the sake of brevity we only take arithmetic operations into account and shall not concern with e.g. operations involved in indexing.) By computing S3 simultaneously for all values of I(I = 1,2,...,10) the computation time can be reduced to 10 steps. Finally by building a minimum height tree for $A^b[I,10](:= A[I,0] + \sum_{J=1}^{10} B[J])$ for each I (I = 1,2,...,10), we can compute all ten trees simultaneously in 4 steps.

To help understanding, let us further consider

```
L:  for I := 1 step 1 until N1 do
    for J := 1 step 1 until N2 do S;
```

Then Figure 4.2(a) shows the execution of L as it is presented. The total computation time required (T) is $N_1 \times N_2 \times m$, where we assume that m arithmetic operators are in S. Now suppose S can be computed in parallel with respect to

I_1 (Figure 4.2(b)). In Figure 4.2(b), each box has the form shown in Figure 4.2(c). Here S is still computed sequentially, i.e. $T_1 = mN_2$. Now let us compute S in parallel i.e. by building a tree (Figure 4.2(d)). Then we have $T_2 = N_2 h[S]$. Note that $m \geq h[S]$. Further if we back substitute S for J = $1,2,\ldots,N_2$ and get S^b, we have $T_3 = h[S^b]$. As stated before (Section 4.1), $N_2 n[S] \geq h[S^b]$, or $T_0 > T_1 \geq T_2 \geq T_3$.

In general we have

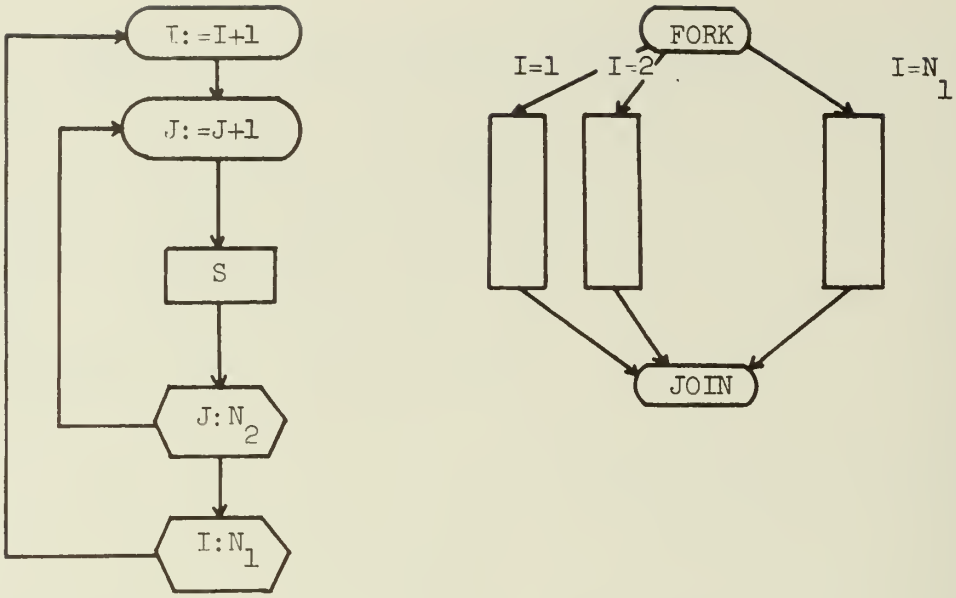
L: for I_1 := 1 step 1 until N_1 do

for I_2 := 1 step 1 until N_2 do

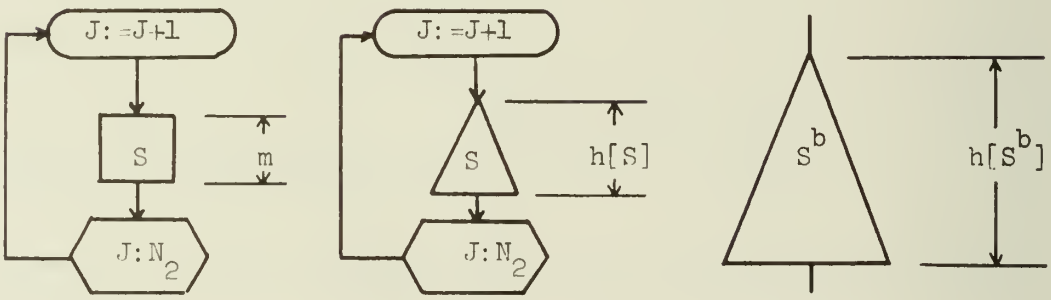
$\vdots$

for I_n := 1 step 1 until N_n do S;

where S is an assignment statement. Then the computation of L takes $T = \left(\prod_{i=1}^{n} N_i\right)m$ steps as it is presented, where we assume that m arithmetic operations are involved in the computation of S. If S can be computed for all values of I_k ($I_k = 1,2,\ldots,N_k$) simultaneously, then the computation time can be reduced to $T_k = T/N_k\left(=\left(\prod_{\substack{i=1 \\ i\neq k}}^{n} N_i\right)m\right)$ steps, i.e. N_k statements can be computed simultaneously while $I_1, I_2, \ldots, I_{k-1}, I_{k+1}, \ldots, I_n$ change sequentially. In general there are n possibilities, i.e. we examine if S can be computed in parallel with respect to I_k for k = 1,2,...,n. Let P = {k|S can be computed in parallel with respect to I_k}. Then we would compute S in parallel with respect to I_g where T_g =



(a) (b)



T_1 T_2 T_3

(c) (d) (e)

Figure 4.2. Loop Analysis

$\min_{k \in P} T_k$. Clearly each statement of the resultant N_g statements can be computed by building a tree. Further if it is appropriate we perform back substitution and obtain a big tree as the above example (E1) illustrates.

If a loop is a limit loop which terminates when $\epsilon < \delta$ for some predetermined δ and computed ϵ, it may be approximated by a counting loop (e.g. for I := 1 step 1 until N do) which is executed a fixed number of times before the $\epsilon < \delta$ test is made, and then repeated if the test fails.

4.3 Jumps

Consider a program containing n two way forward jump statements (or if statements). Let the tests for jumps be Boolean expressions $B_1, B_2, \ldots, B_n$. Assume that there are m output variables from the program given as expressions $A_1, A_2, \ldots, A_m$, where parts of A_i may depend on B_j's. In a program when B_j is encountered, one of two choices is taken depending on the value of B_j. It is possible to start computing all of these possible alternatives at the earliest time, and choose proper results as soon as values of B_j's become available.

For example

```
a := g + c;
B := (a ≥ 0);
if B then d := e + f + s else d := a + g + t;
h := d + f + i × j × k × p × q;
```

yield

```
h := B × (e+f+s) + (not B) × ((g+c)+g+t) + f + i × j × k × p × q
```

or

$$h := ((g+c) \geq 0) \times (e+f+s) + ((g+c) < 0) \times ((g+c)+g+t) + f + i \times j \times k \times p \times q,$$

where we let B = 1 for true, B = 0 otherwise. Then we may build a tree for h as follows.

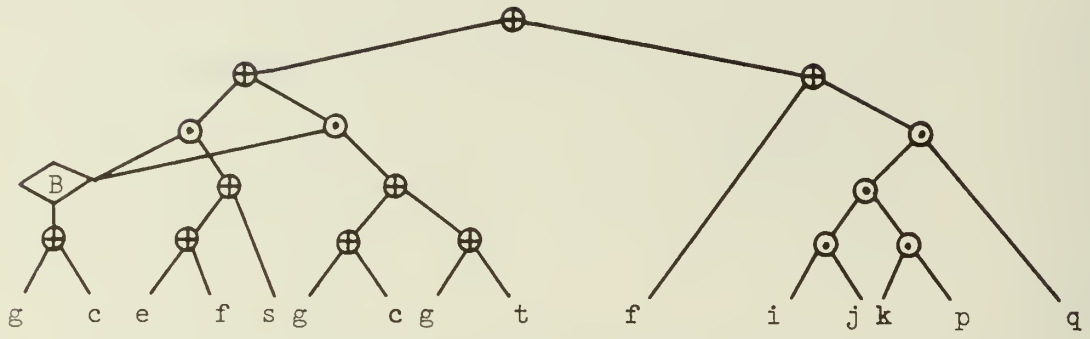


Figure 4.3. A Tree with a Boolean Expression

The box \B · produces 0 or 1 depending on the value of $(g+c \geq 0)$.

In general, Boolean expressions can be embedded in arithmetic expressions as shown in the above example, and a minimum height tree can be built for it.

4.4 Error Analysis

In this section parallel and serial computation are compared in terms of error. We are only concerned with a generated error, i.e. an error which is introduced as a result of arithmetic operations. It is shown that in general parallel computation would produce less error than serial computation. It is also shown that distribution would not increase the size of an error significantly. Let ω represent any arithmetic operation. In general, we do not perform the operation ω exactly but rather a pseudo-operation $\textcircled{\omega}$. Hence instead of obtaining the result $x\omega y$, we obtain a result $x\textcircled{\omega}y$. We may write

$$x \textcircled{\omega} y = (x \omega y)(1+\epsilon_\omega) \tag{1}$$

where ϵ_ω represents an error introduced by performing a pseudo-operation. For example, we have

$$x \textcircled{+} y = (x+y)(1+\epsilon_a)$$

and

$$x \textcircled{\times} y = (xy)(1+\epsilon_m).$$

Let us write A^* for an approximation to an arithmetic expression A with an error obtained by computing A using pseudo operations, e.g. $\textcircled{\times}$ or $\textcircled{+}$. Then Eq. (1) can also be written as

$$(x \omega y)^* = (x \omega y)(1+\epsilon_\omega).$$

Now let us consider the computation

$$A = \sum_{i=1}^{N} a_i.$$

First we compute A serially, i.e.

$$A = (\ldots(((a_1+a_2)+a_3)+a_4)+\ldots+a_N).$$

Then we have

$$A_2^* = a_1 \textcircled{+} a_2 = (a_1+a_2)(1+\epsilon_a)$$

$$= a_1 + a_2 + \epsilon_a(a_1+a_2),$$

$$A_3^* = A_2^* \textcircled{+} a_3 = (a_1+a_2+\epsilon_a(a_1+a_2)+a_3)(1+\epsilon_a)$$

$$= a_1 + a_2 + a_3 + \epsilon_a(2a_1+2a_2+a_3).$$

Note that higher terms of ϵ_a are neglected.

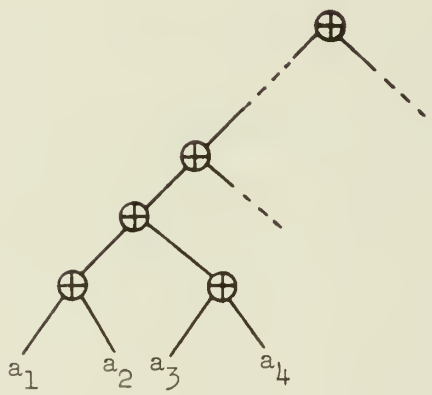
$$A_4^* = A_3^* \oplus a_4 = a_1 + a_2 + a_3 + a_4 + \epsilon_a(3a_1+3a_2+2a_3+a_4),$$

. . .

$$A_N^* = A_{N-1}^* \oplus a_N = \sum_{i=1}^{N} a_i + \epsilon_a((N-1)a_1+(N-1)a_2+(N-2)a_3+\ldots+a_N)$$

$$\approx \sum_{i=1}^{N} a_i + \epsilon_a(Na_1+(N-1)a_2+(N-2)a_3+\ldots+a_N).$$

We let $E_a^s = \epsilon_a(Na_1+(N-1)a_2+(N-2)a_3+\ldots+a_N)$. Next let us compute A in parallel, i.e. by building a tree:



Without loss of generality we assume that N is a power of 2. Then

$$A_{12}^* = a_1 \oplus a_2 = a_1 + a_2 + \epsilon_a(a_1+a_2)$$

$$A_{1-4}^* = A_{12}^* \oplus A_{34}^* = a_1 + a_2 + a_3 + a_4 + 2\epsilon_a(a_1+a_2+a_3+a_4)$$

$$A_{1-8}^* = A_{1-4}^* \oplus A_{5-8}^* = a_1 + a_2 + \ldots + a_8 + 3\epsilon_a(a_1+a_2+\ldots+a_8)$$

...

$$A_{1-N}^* = A_{1-N/2}^* \oplus A_{N/2+1-N}^* = \sum_{i=1}^{N} a_i + \lceil \log_2 N \rceil \epsilon_a \sum_{i=1}^{N} a_i$$

We let $E_a^p = \lceil \log N \rceil \epsilon_a \sum_{i=1}^{N} a_i$. To compare E_a^s with E_a^p, let $a = a_1 = a_2 = \ldots = a_N$.

Then we get

$$E_a^s = \frac{N(N+1)}{2} a \cdot \epsilon_a$$

and

$$E_a^p = \lceil \log_2 N \rceil a \cdot \epsilon_a,$$

or

$$E_a^s > E_a^p.$$

An error for $B = \prod_{i=1}^{N} b_i$ can be analyzed in a similar manner. In this case we get

$$E_m^s = E_m^p = (N-1) \epsilon_m \prod_{i=1}^{N} b_i$$

Hence, in general, we could expect that parallel computation produces less error than serial computation.

Note that if higher terms of e_a and e_m are neglected, then A^* can be written as

$$A^* = A + \epsilon_a \times E_a(A) + \epsilon_m \times E_m(A)$$

where $E_a(A)$ and $E_m(A)$ are arithmetic expressions consisting of variables in A. For example, if we compute A = a(bc+d) serially (i.e. A = a((bc)+d)) then we get

$$A^* = (a\times((b\times c)(1+\epsilon_m)+d)(1+\epsilon_a))(1+\epsilon_m)$$

$$\approx a(bc+d) + \epsilon_a a(bc+d) + \epsilon_m(2abc+ad),$$

and $E_a(A) = a(bc+d)$

and $E_m(A) = 2abc+ad.$

Usually $E_a(A)$ and $E_m(A)$ depend on how A is computed as we have shown for $A = \Sigma a_i$.

Now let us compare parallel computation of two arithmetic expressions A and A^d, where A^d is the resultant expression obtained by applying the distribution algorithm on A, in terms of a generated error. Note that we can write

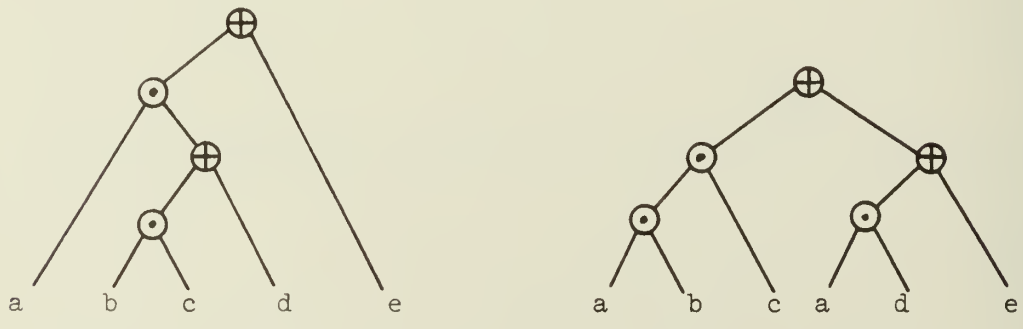
$$A^* = A + \epsilon_a \times E_a(A) + \epsilon_m \times E_m(A)$$

and

$$A^{d*} = A^d + \epsilon_a \times E_a(A^d) + \epsilon_m \times E_m(A^d)$$

$$= A + \epsilon_a \times E_a(A^d) + \epsilon_m \times E_m(A^d).$$

As an example let us study $A = a(bc+d) + e$ and $A^d = abc + ad + e$.



(a) $A = a(bc+d) + e$ (b) $A^d = abc + ad + e$

Figure 4.4. Trees for a(bc+d) + e and abc + ad + e

Then we have

$$A^* = (a(bc(1+\epsilon_m)+d)(1+\epsilon_a)(1+\epsilon_m)+e)(1+\epsilon_a)$$

$$= (a(bc+d)+e)+\epsilon_a(2abc+2ad+e)+\epsilon_m(2abc+ad)$$

and

$$A^{d*} = (abc(1+\epsilon_m)^2 + (ad(1+\epsilon_m)+e)(1+\epsilon_a))(1+\epsilon_a)$$

$$= (abc+ad+e) + \epsilon_a(abc+2ad+2e) + \epsilon_m(2abc+ad).$$

Note that $E_m(A) = E_m(A^d)$ in the above example, which is not mere chance. We can show that this holds for all cases.

Lemma 2:

$$E_m(A) = E_m(A^d)$$

Proof:

First let us consider

$$t^* = t_1^* \oplus t_2^* \oplus \cdots \oplus t_n^*$$

where

$$t_i^* = t_i + \epsilon_a E_a(t_i) + \epsilon_m E_m(t_i).$$

Then clearly

$$E_m(t) = \epsilon_m \sum_{i=1}^{n} E_m(t_i)$$

regardless of the order of additions whereas $E_a(t)$ depends on the order of additions. Hence we may write

$$t^* = \sum_{i=1}^{n} t_i + \epsilon_a E_a(t_\Sigma) + \epsilon_m \sum_{i=1}^{n} E_m(t_i)$$

where Σ indicates that $E_a(t)$ depends on the order of additions.

Now let us consider

$$A^* = t^* \otimes (t_1^* \oplus t_2^* \oplus \cdots \oplus t_n^*)$$

and

$$A^{d*} = t^* \otimes t_1^* \oplus t^* \otimes t_2^* \oplus \cdots \oplus t^* \otimes t_n^*$$

where

$$t^* = t + \epsilon_a E_a(t) + \epsilon_m E_m(t)$$

and

$$t_i^* = t_i + \epsilon_a E_a(t_i) + \epsilon_m E_m(t_i).$$

Then we have

$$A^* = (t+\epsilon_a E_a(t)+\epsilon_m E_m(t))(\sum_{i=1}^{n} t_i + \epsilon_a E_a(t_\Sigma) + \epsilon_m \sum_{i=1}^{n} E_m(t_i)(1+\epsilon_m))$$

$$= t \sum_{i=1}^{n} t_i + \epsilon_a(\sum_{i=1}^{n}(t_i E_a(t)) + tE_a(t_\Sigma)) + \epsilon_m(\sum_{i=1}^{n}(tE_m(t_i)+t_i E_m(t)+tt_i))$$

and

$$A^{d*} = (t+\epsilon_a E_a(t)+\epsilon_m E_m(t))(t_1+\epsilon_a E_a(t_1)+\epsilon_m E_m(t_1))(1+\epsilon_m) \oplus \cdots$$

$$= (tt_1 + \epsilon_a(tE_a(t_1)+t_1 E_a(t)) + \epsilon_m(tE_m(t_1)+t_1 E_m(t)+tt_1)) \oplus \cdots$$

$$= t \sum_{i=1}^{n} t_i + \epsilon_a E_a(A_\Sigma^d) + \epsilon_m(\sum_{i=1}^{n}(tE_m(t_i)+t_i E_m(t)+tt_i)).$$

Hence

$$E_m(A) = E_m(A^d).$$

(Q.E.D.)

As for $E_a(A)$ and $E_a(A^d)$, they depend on the order of additions and cannot be compared simply. However, they may not differ significantly. As a simplified case, let us study the following:

$$A = t\left(\sum_{i=1}^{N} a_i\right)$$

and

$$A^d = \sum_{i=1}^{N} (ta_i).$$

Again we assume that N is a power of two. Then to compute A, we first compute $\sum_{i=1}^{N} a_i$ in parallel. As we showed before, $\left(\sum_{i=1}^{N} a_i\right)^* = \sum_{i=1}^{N} a_i + \lceil \log_2 N \rceil \epsilon_a \sum_{i=1}^{N} a_i$.

Hence

$$A^* = t\left(\sum_{i=1}^{N} a_i + \lceil \log_2 N \rceil \epsilon_a \sum_{i=1}^{N} a_i\right)(1+\epsilon_m)$$

$$= t \sum_{i=1}^{N} a_i + \epsilon_a \lceil \log N \rceil t \sum_{i=1}^{N} a_i + \epsilon_m t \sum_{i=1}^{N} a_i.$$

On the other hand, we have

$$A_i^* = (ta_i)^* = ta_i + \epsilon_m ta_i$$

and A^{d*} is obtained by summing A_i in parallel, i.e.

$$A^{d*} = (..(((A_1^* \oplus A_2^*) \oplus (A_3^* \oplus A_4^*)) \oplus ((A_5^* \oplus A_6^*) \oplus (A_7^* \oplus A_8^*)))...)$$

$$= t \sum_{i=1}^{N} a_i + \epsilon_a \lceil \log_2 N \rceil t \sum_{i=1}^{N} a_i + \epsilon_m t \sum_{i=1}^{N} a_i.$$

Hence in this case $E_a(A) = E_a(A^d)$ as well as $E_m(A) = E_m(A^d)$.

5. PARALLELISM BETWEEN STATEMENTS

This chapter should be read as an introduction to the following chapter which discusses loops in a program. In this chapter we study parallelism between statements, i.e. inter-statement parallelism. Given a loop and jump free sequence of statements (we call this a program), it is expected that they are executed according to the given (i.e. presented) order. However if two statements do not depend on each other, they may be executed simultaneously in hopes of reducing the total computation time. In general, statements in a program may be executed in any order other than the given order as long as they produce the same results as they will produce when they are executed in accordance with the given sequence. In this chapter we give an algorithm which checks if the execution of statements in a program by some sequence gives the same results as the execution of statements by the given sequence does. Also a technique which exploits more parallelism between statements by introducing temporary locations is introduced.

5.1 Program

A program P with a memory M is a sequence of assignment statements S(i), i.e. $P = (S(1); S(2); \ldots; S(i); \ldots; S(r))$ where i is a statement number and r is the length of a program P (we write $r = lg(P)$). The memory M is a set of all variables (or identifiers) which appear in P.

Associated with each S(i) is a set of input variables, IN(S(i)) and an output variable, OUT(S(i)). Then $M = \bigcup_{i=1}^{r} (IN(S(i)) \cup OUT(S(i)))$. Further

we define two regions in a memory; a primary input region M_I and a final output region M_O as

$$M_I = \{m \mid m \in IN(S(i)) \text{ and } \forall k < i,\ m \notin OUT(S(k))\}.$$

and $M_O = \{m \mid m \in OUT(S(i))\}$.

A program uses the values of those variables in M_I as primary input data and puts final results into M_O.

C(m) refers to a content (value) of a variable m. C(M) refers to the contents of variables in the memory M as a whole and is called a configuration of M. Also $C_I(m)$ refers to a value which m has before a computation (i.e. an initial value of m). Thus $C_I(M_I)$ refers to primary input data given to a program. We call it an initial configuration.

The following relations are established among statements in P.

A triple (id, i, j) where id $\in$ M (id for an identifier) and i, j $\in$ $\{0, 1, \ldots, r, r+1\}$ ($r = lg(P)$) is in the dependence relation DR(P) if and only if:

(1) (i) $i < j$ and

(ii) id $\in$ OUT(S(i)) and id $\in$ IN(S(j)) and

(iii) $\forall k,\ i < k < j$, id $\notin$ OUT(S(k)),

or (2) (i) $i = 0$ and

(ii) id $\in$ IN(S(j)) and

(iii) $\forall k,\ 0 < k < j$, id $\notin$ OUT(S(k)),

(S(j) is the first statement to use id).

or (3) (i) $j = r + 1$ and

(ii) id $\in$ OUT(S(i)) and

(iii) $\forall k,\ i < k < r + 1$, id $\notin$ OUT(S(k)).

(S(i) is the last statement to update id).

Similarly a triple (id,i,j) is in the locking relation LR(P) if and only if:

(i) $i < j$ and

(ii) id $\in$ IN(S(i)) and id $\in$ OUT(S(j)) and

(iii) $\forall k,\ i < k < j,$ id $\notin$ OUT(S(k)).

Example 1 (The notations follow ALGOL 60[3]):

Let P be

S(1): a := b + c;

S(2): d := a + e;

S(3): f := g + d;

S(4): g := h + i.

Then

DR(P) = {(b,0,1),(c,0,1),(e,0,2),(g,0,3),(h,0,4),(e,0,4),(a,1,2),
(d,2,3),(f,3,5),(g,4,5)} and

LR(P) = {(g,3,4)}.

Since we are only interested in meaningful programs, we assume that there is no superfluous statement, i.e. there is no id $\in$ M such that

(i) id $\in$ OUT(S(i)) and id $\in$ OUT(S(j)) where $i < j$, and

(ii) $\forall k,\ i < k < j,$ id $\notin$ IN(S(k)).

Also we assume that there is no statement that has no inputs other than constant numbers, e.g. "a := 5".

Now we define an execution order E of a program P as:

$E(P) = \{(i,j) \mid i \in \{1,2,\ldots,lg(P)\}, j \in \{1,2,\ldots\}\}$.

We also write $E_P(i) = j$ if $(i,j) \in E(P)$.# To execute a program by E(P) means that at step j, all statements with statement number $E_P^{-1}(j)$ are computed simultaneously using data available before the j-th computation as inputs. A pair (P,E) is used to denote this execution. Also by $E_0(P)$, we understand the execution order given by a program, i.e.

$$E_0(P) = \{(i,i) \mid \forall i \in \{1,2,\ldots,lg(P)\}\}.$$

E_0 is called a primitive execution order.

We assume that at each time step at least one statement of P must be executed. That is, for any E there is k such that $\forall j > k$, $E_P^{-1}(j) = \text{empty}$ and $\forall j \leq k$, $E_P^{-1}(j) \neq \text{empty}$. We call k the length of an execution and write lg(E).

As stated before, C(OUT(S(i))) refers to the contents of a variable OUT(S(i)). This value, as we expect, varies from time to time throughout an execution. Thus it is essential to specify the time when a variable is referred to.

S(i)(P,E) refers to a computation of S(i) of P in an execution (P,E). C(m) after S(i)(P,E) refers to the value of a variable m right after S(i)(P,E). C(m) after (P,E) refers to the value of a variable m after an execution of a whole program.

5.2 Equivalent Relations Between Executions

Now we define two equivalent relations between executions.

For convenience we define that $\forall i$, $E_P(0) < E_P(i)$ and $E_P(i) < E_P(lg(P)+1)$.

Definition 1:

Given a program P and two execution orders E_1 and E_2, (P,E_1) and (P,E_2) (or simply E_1 and E_2) are said to be output equivalent if and only if: for all initial memory configurations $C_I(M_I)$,

$\forall i\ C(OUT(S(i)))$ after $S(i)(P,E_1) = C(OUT(S(i)))$ after $S(i)(P,E_2)$.

We write $(P,E_1) \overset{O}{=} (P,E_2)$ if (P,E_1) is output equivalent to (P,E_2).

Definition 2:

Given two programs P_1 and P_2, let their execution orders be E_1 and E_2 respectively. Also let their memories be M_1 and M_2. Then two executions (P_1,E_1) and (P_2,E_2) are said to be memory equivalent if and only if:

(1) there is a one-to-one function

$f:\{M_{1I} \cup M_{1O}\} \rightarrow \{M_{2I} \cup M_{2O}\}$

such that

$f(M_{1I}) = M_{2I}$

and $f(M_{1O}) = M_{2O}$,

and (2) for all initial memory configuration pairs $C_I(M_{1I})$ and $C_I(M_{2I})$ such that

$\forall m \in M_{1I},\ C_I(m) = C_I(f(m))$.

$\forall n \in M_{1O}$,

$C(n)$ after $(P_1,E_1) = C(f(n))$ after (P_2,E_2).

We write $(P_1,E_1) \overset{M}{=} (P_2,E_2)$ if (P_1,E_1) is memory equivalent to (P_2,E_2).

In principle, a program is written assuming that it will be executed sequentially, i.e. by E_0. It, however, need not necessarily be executed by E_0 as long as it produces the same results as (P,E_0) when it terminates, i.e. it may be executed by any E as long as $(P,E)\overset{M}{=}(P,E_0)$ holds.

Now the following theorems can be proved directly from the above definitions.

Theorem 1

$(P,E)\overset{O}{=}(P,E_0)$ if and only if:

(1) $\forall i$, $(\underline{id},i,j)\in DR$ implies that $E_P(i) < E_P(j)$.

and (2) for any two triples ($\underline{id}$, i, j) and ($\underline{id}$, i', j') in DR with the same identifier $\underline{id}$, either $E_P(j') < E_P(i)$ or $E_P(j) < E_P(i')$ holds.

What condition (1) implies is that variables must be properly updated before used, and condition (2) prevents variables from being updated before they are used by all pertinent statements.

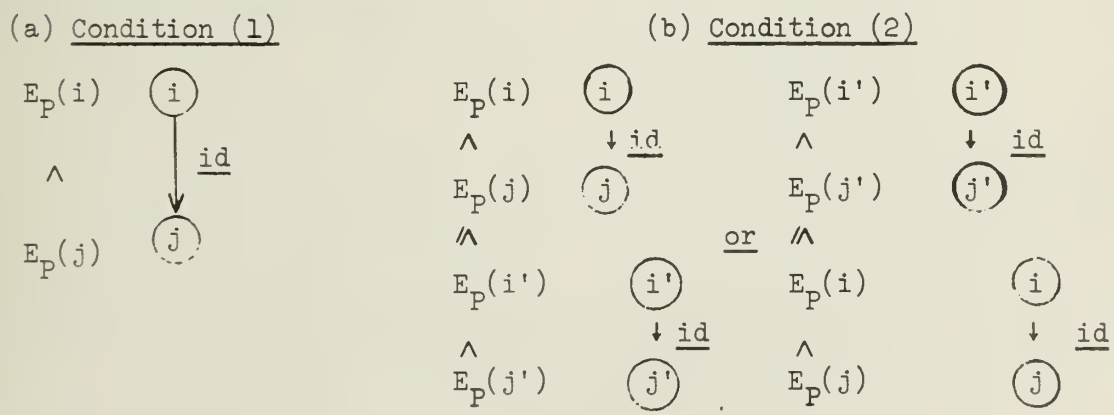


Figure 5.1. Conditions for the Output Equivalence

Proof of Theorem 1:

(1) if part:

Assume that a statement S(i) receives data from statements $S(i_1), S(i_2), \ldots, S(i_k)$, i.e. for each pair i and i_s $(s=1,2,\ldots,k)$ there is an identifier $\underline{id}_s$ such that $(\underline{id}_s, i_s, i) \in DR$. Now let E be an execution order which guarantees that (1) before S(i) is computed, all $S(i_s)$ are computed, and (2) between the computation of $S(i_s)$ and S(i), no statement updates $\underline{id}_s$, then it is clear that (C(OUT(S(i))) after S(i)(P,E) = C(OUT(s(i))) after S(i)(P,E_0) providing that all $OUT(S(i_s))$ have appropriate values. Note that the above two requirements are equivalent to conditions (1) and (2) of the theorem. Then by induction, we can show that if conditions (1) and (2) hold for all statements, then $(P,E) \overset{0}{=} (P,E_0)$.

(2) only if part:

We give an example to show that if an execution order violates condition (1) or (2) then we cannot get an output equivalent execution. Now let P be

S(1): a := b;

S(2): c := a;

S(3): b := e.

Then DR = {(b,0,1),(e,0,1),(a,1,2),(c,2,4),(b,3,4)}, and (P,E_0) gives

$C(OUT(S(1)))$ after $S(1)(P,E_0) = C_I(b)$,

$C(OUT(S(2)))$ after $S(2)(P,E_0) = C_I(b)$,

and $C(OUT(S(3)))$ after $S(3)(P,E_0) = C_I(b)$.

Now let $E_1(P) = \{(1,2),(2,1),(3,3)\}$ which violates the first condition of the theorem, and $E_2(P) = \{(1,2),(2,3),(3,1)\}$ which violates the second condition. Then

$C(OUT(S(2)))$ after $S(2)(P,E_1) = C_I(a)$

and $C(OUT(S(1)))$ after $S(1)(P,E_2) = C_I(e)$

which do not agree with corresponding values produced by (P,E_0).

(Q.E.D.)

Theorem 1 gives more meaningful executions compared to the previous results [5] [10]. For example let P be:

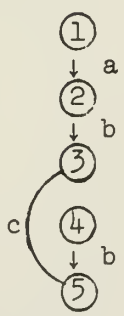
S(1): $a:=f_1(x)$;

S(2): $b:=f_2(a)$;

S(3): $c:=f_3(b)$;

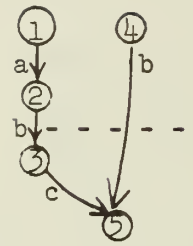
S(4): $b:=f_4(x)$;

S(5): $d:=f_5(b,c)$.



Fisher [10], for example, would give the following execution (P,E) as an "equivalent" execution to (P,E_0).

Step



$E = \{(1,1), (2,2), (3,3), (4,1), (5,4)\}$.

This, however, does not give correct results unless P is properly modified. Note that the variable b carries two different values between steps 2 and 3 which is physically impossible. Theorem 1 does not recognize such an execution as "equivalent" to (P,E_0).

Theorem 2

$(P,E) \overset{M}{=} (P,E_0)$ if and only if

(1) $\forall i$, $(\underline{id}, i, j) \in DR$ implies that $E_p(i) < E_p(j)$,

and (2) $\forall_j$, $(\underline{id}, i, j) \in LR$ implies that $E_p(i) < E_p(j)$.

Example 2:

P: S(1): a:=b+c;

S(2): d:=a+e;

S(3): a:=q+r;

S(4): h:=a+s.

Let $E(P) = \{(3,1), (4,2), (1,3), (2,4)\}$. Then $(P,E) \overset{0}{=} (P,E_0)$.

E, however, violates the second condition of Theorem 2, i.e. $(a,2,3) \in LR$ but $E_p(2) \not< E_p(3)$.

The following lemma is helpful to prove the theorem.

Lemma 1:

If two conditions of Theorem 2 hold for an execution order E, then $(P,E)\overset{O}{=}(P,E_0)$.

Proof:

We show that if conditions (1) and (2) of Theorem 2 (we write C(2-1) and C(2-2) for them) hold, then conditions (1) and (2) of Theorem 1 (C(1-1) and C(1-2)) follow.

First note that C(2-1) is identical to C(1-1). Next we show that C(2-2) together with C(2-1) satisfy C(1-2). Assume that $(a,i_1,j_1),(a,i_2,j_2) \in DR$ where $j_1 < i_2$. Then there exist statements $S(h_1),S(k_1),S(h_2),S(k_2),\ldots,S(h_m),S(k_m)$ such that $(a,j_1,h_1),(a,k_1,h_2),(a,k_2,h_3),\ldots,(a,k_g,h_{g+1}),\ldots,(a,k_m,i_2) \in LR$ and $(a,h_1,k_1),(a,h_2,k_2),\ldots,(a,h_g,k_g),\ldots,(a,h_m,k_m) \in DR$. Then if C(2-1) and C(2-2) hold, then $E_P(j_1) < E_P(i_2)$. Thus C(1-2) follows.

(Q.E.D.)

Proof of Theorem 2:

(1) if part:

Let E be an execution order which satisfies C(2-1) and C(2-2). Then by Lemma 1, $(P,E)\overset{O}{=}(P,E_0)$. Now let i be a statement number such that $(\underline{id},i,r+1) \in DR$ where r = lg(P), i.e. S(i) is the last statement in (P,E_0) which updates id. Then

$$C(\underline{id}) \text{ } \underline{after} \text{ } S(i)(P,E_0) = C(\underline{id}) \text{ } \underline{after} \text{ } (P,E_0). \qquad (1)$$

Also by a similar argument used to prove Lemma 1, we can show that for all j such that id ϵ OUT(S(j)), $E_P(j) < E_P(i)$ holds. Thus S(i) is the last statement to update id in (P,E), too. Thus

C(id) after S(i)(P,E) = C(id) after (P,E). (2)

Also since $(P,E_0) \overset{0}{=} (P,E)$, we have C(OUT(S(i))) after S(i)(P,E_0) = C(OUT(S(i))) after S(i)(P,E) or

C(id) after S(i)(P,E_0) = C(id) after S(i)(P,E). (3)

Thus from (1), (2) and (3), we have C(id) after (P,E) = C(id) after (P,E_0). Using the same argument for all i such that (id,i,r+1) ϵDR, we can show that for all m ϵ M_0, C(m) after (P,E) = C(m) after (P,E_0).

(2) only if part:

This part is again proved by giving a counter example. It is easy to show that a program

S(1): a := e;

S(2): c := b;

S(3): b := a.

together with execution orders E_1 = {(1,3),(2,1),(3,2)} and E_2 = {(1,1),(2,3),(3,2)} serves as a counter example. The details are omitted.

(Q.E.D.)

Note that Lemma 2 can be modified as:

Corollary 1:

If $(P,E)\overset{M}{=}(P,E_0)$, then $(P,E)\overset{O}{=}(P,E_0)$.

Now let us study the memory equivalence relation between two different programs and execution orders, (P_1,E_1) and (P_2,E_2), in detail. As a subcase of this let us consider (P_1,E_0) and (P_2,E_0). In general it is impossible to show whether $(P_1,E_0)\overset{M}{=}(P_2,E_0)$. That is, this problem can be reduced to the Turing machine halting problem which is known to be recursively unsolvable [28]. In our discussion we have put the restriction on P so that a program is a loop-free block of assignment statements. Even with this restriction it may still be practically impossible to show memory equivalence between (P_1,E_0) and (P_2,E_0). For example if P_1 and P_2 are different polynomial approximations for the same function, then (P_1,E_0) and (P_2,E_0) are likely to produce different results due to e.g. a truncation error. We do not pursue this problem further.

Finally let us consider the following example:

Example 3:

```
P:  S(1):  k := a;
    S(2):  b := k;
    S(3):  k := c;
    S(4):  d := k;
    S(5):  k := e;
    S(6):  f := k.
```

Let $E(P) = \{(1,1),(2,2),(3,2),(4,3),(5,3),(6,4)\}$. Then $(P,E_0)\overset{M}{=}(P,E)$ and $lg(E) = 4$.

However, the following program P'

P': S(1): k' := a;

S(2): b := k';

S(3): k" := c;

S(4): d := k";

S(5): k := e;

S(6): f := k.

together with an execution order $E(P') = \{(1,1),(2,2),(3,1),(4,2),(5,1),(6,2)\}$ gives a memory equivalent execution to (P,E_0), and $lg(E(P')) = 2$.

This suggest the introduction of the following transformation, which when applied on a program P, produces a new program P' such that $(P,E_0) \overset{M}{=} (P',E_0)$.

Transformation T_1

Let $S_1 = \{S(i),S(i+1),\ldots,S(j)\}$ and $S_2 = \{S(k),S(k+1),\ldots,S(m)\}$ where $j < k$. Assume that there is an identifier id such that $(id,j,k) \in LR$, and id $\in$ OUT(S(i)), and $\forall u$, $i < u \leq j$, id $\notin$ OUT(S(u)). Also assume that for any v and w, $i \leq v \leq j$ and $k \leq w \leq m$, there is no id' such that $(id',v,w) \in DR$. Then replace every occurrence of id in S_1 by id' where id' $\notin$ M.

Gold [17] presented a similar transformation to describe his model for linear programming optimization of programs.

After the transformation is applied S_1 and S_2 can be processed in parallel, and still $(P,E_0) \overset{M}{=} (P',E_0)$ holds where P' is the resultant program after the application of T_1 on P.

This shows that the second condition of Theorem 2 is not essential, i.e. it can be removed by introducing extra locations if necessary.

6. PARALLELISM IN PROGRAM LOOPS

6.1 Introduction

6.1.1 Replacement of a for Statement with Many Statements

Using the results from the previous chapter, now let us study loops in a program, e.g. ALGOL for statements or FORTRAN DO loops, to extract potential parallelism among statements. Given a loop P, we seek an execution order E with the minimum length among all possible ones. Sometimes it may be appropriate to get a loop P' from P by the previously introduced transformation for which there is an execution order E' such that $(P',E') \overset{M}{=} (P,E_0)$ and lg(E') is the minimum (For the definition of $\overset{M}{=}$, see Chapter 5).

As stated before, in this chapter our main concern is the parallelism among statements (interstatement parallelism). For example, we are interested in finding out that all 10 statements (A[I] := A[I + 1] + FUNC(B[I]); (I=1,2,..., 10)) in F1 can be computed in parallel, whereas statements in F2 cannot be (The notation follows ALGOL 60 [3]):

```
F1:  for I := 1 step 1 until 10 do
         A[I] := A[I + 1] + FUNC(B[I]);
F2:  for I := 1 step 1 until 10 do
         A[I] := A[I - 1] + FUNC(B[I]).
```

First several notations are presented. According to the ALGOL 60 report [3], a for statement has the following syntax:

< for statement > ::= < for clause > + < statement > +[#]

< for clause > ::= for < variable > := < for list > do.

An instance of this is:

for I_1 := ... do
...
for I_n := ... do

begin S_1; S_2; ...; S_m end.

For the sake of brevity, we shall write $(I_1 \leftarrow L_1, I_2 \leftarrow L_2, \ldots, I_n \leftarrow L_n)$ $(S_1, S_2, \ldots, S_m)$ or $(I_1, I_2, \ldots, I_n)(S_1, S_2, \ldots, S_m)$ for the above for statement instance where I_k is called a loop index, L_k is an ordered set and called a loop list set, and S_p is called a loop body statement with a statement identification number p (which is different from a statement number (see Chapter 5)).

As its name suggests, a loop list set represents a < for list >, e.g. $L_k = (1,2,3,4,5,6)$ represents "I_k := 1 step 1 until 6." In general we write $L_k(i)$ for the i-th element of L_k thus $L_k(|L_k|)$ is the last element of L_k.

Now to facilitate later discussions we introduce the following notation.

Let $B = (b_1, b_2, \ldots, b_n)$ ($b_i > 0$ for all i) and $(i_1, i_2, \ldots, i_n)$ be n-tuples of integers. Then we define the value of $(i_1, i_2, \ldots, i_n)$ w.r.t. B as follows.[##]

[#] < A > + ≡ < A > < A >* where * is the Kleene star.

[##] For convenience we write i(s..t) for (t-s+1) integers i_s, i_{s+1}, ..., i_{t-1}, i_t, e.g. (i(1..s), i(s+2..n)) means $(i_1, \ldots, i_s, i_{s+2}, \ldots, i_n)$. Also (|L(s..t)|) means $(|L_s|, |L_{s+1}|, \ldots, |L_t|)$. Finally (i(n)) means n i's e.g. (1(3)) = (1,1,1).

$$V((i(i..n))|B) = \sum_{j=1}^{n} i_j B_j - \sum_{j=1}^{n} B_j + 1$$

where $B_j = \prod_{k=j}^{n+1} b_{k+1}$ and $B_n = b_{n+1} = 1$.

This notation is introduced so that the relations

$$V((1,1,\ldots,1,1)|B) = 1,$$

$$V((1,1,\ldots,1,2)|B) = 2,$$

$$\vdots$$

$$V((1,1,\ldots,1,b_n)|B) = b_n,$$

$$V((1,1,\ldots,1,2,1)|B) = b_n + 1,$$

$$V((1,1,\ldots,1,2,2)|B) = b_n + 2,$$

$$\vdots$$

and $V((b_1,b_2,\ldots,b_n)|B) = b_1 \times b_2 \times \ldots \times b_n$

hold.

For example $V((2,3,1)|(3,4,5)) = 31$. An n-tuple B is called a base.

The inverse function of V is also defined as $V^{-1}(t|B) = (i(1..n))$ if $V((i(1..n))|B) = t$. Note that V^{-1} is not one-one e.g. $V^{-1}(15|(3,4,5)) = (2,0,0)$ or $V^{-1}(15|(3,4,5)) = (1,4,0)$. An n-tuple $(i(1..n))$ is said to be normalized if $b_j \geq i_j > 0$ for all j. Let $(i(1..n))$ be normalized. Then $1 \leq V((i(i..n))|B) \leq \sum_{j=1}^{n} b_j B_j - \sum_{j=1}^{n} B_j + 1$.

If $1 \leq t \leq \sum_{j=1}^{n} b_j B_j - \sum_{j=1}^{n} B_j + 1$, then $V^{-1}(t|B)$ has unique normalized $(i(1..n))$ as its value.

We say that normalized (i(i..n)) ranges over B = (b(1..n)) in increasing order if V((i(i..n))|B) takes all values between 1 and $\sum_{j=1}^{n} b_j B_j$ - $\sum_{j=1}^{n} B_j$+1 in increasing order as (i(1..n)) changes. Notationally we write

(1(n)) $\leq$ (i(1..n)) $\leq$ (b(1..m)).

Finally we let

(i(1..n)) > (j(1..n)) if V((i(1..n))|B) > V((j(1..n))|B)

and

(i(1..n)) = (j(1..n)) if V((i(1..n))|B) = V((j(1..n))|B)

The following lemma is an immediate consequence of the above definition.

Lemma 1:

Let B = (b(1..n)) where $\forall_i b_i \geq 2$. Then

(1) V((a(1..n))|B) $\leq$ V((a'(1..n))|B) implies that

$$V((a_1 - a_1', \ldots, a_n - a_n')|B) \leq - \sum_{j=1}^{n} B_j \text{ or } V((a_1 - a_1', \ldots, a_n - a_n')|B) \leq V((0(n))|B).$$

(2) $V((a_1'+c_1', \ldots, a_n'+c_n')|B) = V((a_1+c_1, \ldots, a_n+c_n)|B)$ and V((a'(1..n))|B) $\geq$ V((a(1..n))|B) imply that

$$V((c_1 - c_1', \ldots, c_n - c_n')|B) \leq - \sum_{j=1}^{n} B_j \text{ or } V((c_1 - c_1', \ldots, c_n - c_n')|B) \leq V((0(n))|B).$$

(3) Let $0 \leq |a_j| < b_j$ for all j. Then V((a(1..n)|B) $\leq$ V((0(n))|B) if and only if there is h such that $\forall k$ ($1 \leq k \leq h$), $a_k = 0$ and $a_h < 0$.

A loop must be replaced with a sequence of statements so that we can use the results of the previous chapter. For example we replace

```
for I := 1 step 1 until 10 do
S1:  A[I] := A[I] + B[I];
```

with the sequence of ten statements

```
A[1] := A[1] + B[1];
A[2] := A[2] + B[2];
        ...
A[10] := A[10] + B[10].
```

In general we will get $\left(\prod_{j=1}^{n} |L_j| \cdot m\right)$ statements after the replacement of a loop $P:(I_1,I_2,\ldots I_n)(S_1;S_2;..;S_m)$. Any statement in the set of replaced statements can be identified by an n-tuple (i(1..n)) which corresponds to values of $I_1,I_2,\ldots,I_n$ (i.e. $L_1(i_1),L_2(i_2),..,L_n(i_n)$), and p which represents a statement identification number. Thus an (n + 1)-tuple (i(1..n),p) serves as a statement number, and we write S((i(1..n),p)) to denote a particular statement in the set of replaced statements, e.g. in the above example S((3,1)) ≡ A[3] := A[3] + B[3]. The actual statement which corresponds to this is the statement S_p with $L_1(i_1),\ldots,L_n(i_n)$ substituted into every occurrence of $I_1,\ldots,I_n$ in S_p, and we also write $S_p[L_1(i_1),\ldots,L_n(i_n)]$ for this.

These $\left(\prod_{j=1}^{n} |L_j| \, m\right)$ statements are to be executed according to the presented order (i.e. the order specified by for loop lists). In other words, the statement S((1,1,...,1,1)) is executed first, S((1,1,...,1,2)) second, ...,

the statement S((i(1..n),p)) is executed V((i(1..n),p)|(|L(1..n)|,m))-th,..., and the statement S((|L_1|,...,|L_n|,m)) is executed lastly. Formally, as the essential execution order we have:

$$E_0(P) = \{((i(1..n),p),V((i(1..n),p)|B)) \mid (l(n),1) \leq (i(1..n),p) \leq (|L(1..n)|,m)\}$$

where B = (|L(1..n)|,m).

Example 1:

```
for I1 := 1 step 1 until 10 do
for I2 := 1 step 1 until 10 do
begin
S1: A1[I1,I2] := A2[I1-1,I2] + B1[I1,I2];
S2: B2[I1,I2-1] := A3[I1+1,I2] + B3[I1,I2+1];
end
```

is executed as

```
S((1,1,1)): A1[1,1] := A2[0,1] + B1[1,1];
S((1,1,2)): B2[1,0] := A3[2,1] + B3[1,2];
S((1,2,1)):            ...
S((1,2,2)):            ...
    ...
S((10,10,2)): B2[10,9] := A3[11,10] + B3[10,11];
```

The superscript is used to distinguish different occurences of A and B.

$A[(i(1..n))]$ represents a form in which $L_1(i_1),\ldots,L_n(i_n)$ are substituted into $I_1,\ldots,I_n$ in index expressions, e.g. in the above example

$$A^2[(i_1,i_2)] = A^2[i_1-1,i_2]$$

and $A^2[(3,2)] = A^2[2,2]$.

Finally a set of inputs to a statement $S((i(1..n),p))$ is denoted by $IN(S((i(1..n),p)))$. Similary $OUT(S((i(1..n),p)))$ represents a set of outputs from $S((i(1..n),p))$. From the above example we have, e.g.

$$IN(S(1,1,2)) = \{A^3[2,1], B^3[1,2]\}$$

and $OUT(S(1,1,2)) = \{B^2[1,0]\}$.

6.1.2 A Restricted Loop

In what follows, we mainly deal with a restricted class of for statements. Two restrictions are introduced. Let a loop with m body statements be

$$P^m = (I_1,I_2,\ldots,I_n)(S_1;S_2;\ldots;S_m).$$

Restriction 1:

A for list set L_j must be an arithmetic sequence, i.e.

$$L_j = (1,2,3,\ldots,t_j) \qquad (1)$$

for all j.

Restriction 2:

Let $\{A_1,A_2,\ldots,A_s\}$ be a set of all array identifiers in P^m where the h-th occurrence of A_k in P^m has the following form (where the superscript h is

used only if it is important to distinguish different occurrences of A_k):

$$A_k^h[F(k,h,1),\ F(k,h,2),\ \ldots,\ F(k,h,n)]. \qquad (2)$$

For fixed k and j, F(k,h,j) has an identical form for all h, i.e. either

$$F(k,h,j) = I_j + w(k,h,j)$$

or $F(k,h,j) = \emptyset$ (i.e. vacant).

w(k,h,j) is a constant number. Also we assume that each A_k appears on the left hand side of statements at most once.

An example of a restricted loop is:

for I_1 := 1 step 1 until 20 do

for I_2 := 1 step 1 until 30 do

for I_3 := 1 step 1 until 40 do

begin

S1: $A_3[I_1-1, I_2+3, \emptyset] := A_3[I_1, I_2-3, \emptyset] + A_1[\emptyset, \emptyset, I_3]$;

S2: $A_2[\emptyset, I_2, I_3-1] := A_3[I_1-1, I_2, \emptyset] + A_1[\emptyset, \emptyset, I_3-1]$;

S3: $A_1[\emptyset, \emptyset, I_3+1] := A_2[\emptyset, I_2-1, I_3]$;

end;

Note that, for example, A_3 always appears as

$A_3[I_1+w(3,h,1),\ I_2+w(3,h,2), \emptyset]$, thus the first occurrence of A_3 is

$A_3[F(3,1,1),\ F(3,1,2), \emptyset] = A_3[I_1+w(3,1,1),\ I_2+w(3,1,2), \emptyset] = A_3[I_1-1,\ I_2+3, \emptyset]$.

If there is no ambiguity, we write e.g. $A_3[I_1-1, I_2+3]$ for $A_3[I_1-1, I_2+3, \emptyset]$ (which is the conventional form).

We also write $F(k,h,j)(i)$ for the resultant expression obtained by substituting i into I_j in $F(k,h,j)$, e.g. $A_3[F(3,1,1)(2), F(3,1,2)(3),\emptyset] = A_3[1,6,\emptyset]$ $(= A_3[1,6]$ conventionally).

A single variable may be introduced as a special case of array indentifiers, e.g. we write

```
for I := 1 step 1 until 1 do
 A[I] :=        ...
```

for

```
    T :=        ... .
```

6.2 A Loop With a Single Body Statement

6.2.1 Introduction

First we shall deal with the case where a loop has only one body statement (i.e. m = 1). Let a loop with a single body statement be $P^1 = (I_1, I_2, \ldots, I_n)S$. Since there is only one statement we may drop the statement identification number. Then a statement number for a replaced statement becomes (i(1..n)) and as the essential execution order we have:

$$E_0(P^1) = \{((i(1..n)),\ V((i(1..n))|(|L(1..n)|)) \mid$$

$$(1(n)) \leq (i(1..n)) \leq (|L(1..n)|)\}.$$

Also in this case we only have to consider the array identifier which appears on the left hand side of S. Hence instead of s array identifiers we only have one array identifier (see Restriction 2 of Section 6.1.2). Hence we drop k and write

$$A^h[F(h,1),\ F(h,2),\ \ldots,\ F(h,n)]$$

and

$$F(h,j) = I_j + w(h,j)$$

for the h-th occurrence of A for Eq. (2) of Section 6.1.2 (the superscript is used if it is necessary to distinguish the different occurrences of A). Furthermore we assume that $F(h,j) \neq \emptyset$ for any h and j.

Now let us study the following two examples.

```
G1:  for I := 1 step 1 until 10 do
         A[I] := A[I] + 5;
G2:  for I := 1 step 1 until 2 do
     for J := 1 step 1 until 10 do
         A[I,J] := A[I-1,J+1] + 5;
```

Assume that an arbitrary number of PE's are available. Then:

G1: All ten statements (A[I] := A[I] + 5) can be computed simultaneously by 10 PE's.

G2: A[1,J] and A[2,J-2] can be computed simultaneously by two PE's at the J-th step (J=1,2,...,10).

In what follows, the above two types of the interstatement parallelism are studied.

Before we go into the details, a few comments are in order with regard to real programs. A for statement with a single body statement, $(I_1,\ldots,I_n)S$, can be classified from several different points of view. First of all let us take a for list set L_j. As a simplified case we have $L_j = (s_j, s_j+1, \ldots, t_j)$ $(t_j = (|L_j|-1) + s_j)$ which is equivalent to an ALGOL statement "for $I_j := s_j$ step 1 until t_j do". Knuth stated [9] that examination of published algorithms showed that well over 99 percent of the use of the ALGOL for statement, the

value of the step was '+1', and in the majority of the exceptions the step was a constant. This statement was confirmed by checking all Algorithms published in the Communications of the ACM in 1969. There were 23 programs and 263 for statements used. Only six uses were exceptions ($\approx$ 3 percent).

Next let us examine a body statement S. Then either (1) the left hand side variable of S (i.e. OUT(S)) is a single variable, or (2) OUT(S) is an array identifier. In case of (2) S is of a form

$$A^0[F(0,1),\ldots,F(0,n)] := f(A^1[F(1,1),\ldots,F(1,n)],\ldots,A^p[f(p,1),\ldots]).$$

Now S has either one of the following five forms.

(1) OUT(S) is a single variable t.#

(i) t := a function which does not depend on t,

e.g. t := a + 5,

(ii) t := f(t), e.g. t:= t + a,

(2) OUT(S) is an array variable A:

(i) $A^0[F_1^0,\ldots,F_a^0]$:= a function which does not depend on A,

e.g. A[I,J] := b + 5,

(ii) for all h F(0,j) - F(h,j) is a constant for each j,

e.g. A[I,J] := A[I-5,J+3] + A[I+1,J-3] + 5

(iii) other cases, e.g. A[I,J] := A[2I,J-5] + a.

Note that if S is of Form (1-i), then

$P^1 \equiv S[L_1(|L_1|), L_2(|L_2|), \ldots, L_n(|L_n|)]$. For example let P^1 be

---

We use a lower case letter for a single variable and an upper case letter for an array variable.

for I := 1 step 1 until 5 do t := A[I] - 1.

Then after the execution of P^1, t = A[5] - 1.

Again all Algorithms published in the CACM were checked (this time the check was made against Algorithms published in 1968 and 1969.) There were 52 programs altogether and 117 for statements with a single body statement. The details were:

		No. of Examples	Percentage
(1)	(i)	0	0
	(ii)	42	35.8
(2)	(i)	18	15.4
	(ii)	33	28.2
	(iii)	24	20.6
		117	100.0

In what follows we deal with Forms (2-i), (2-ii) and (2-iii). Form (1-ii) has been discussed in Chapter 4.

6.2.2 Type 1 Parallelism

6.2.2.1 General Case

As stated in Chapter 5, a block of statements P need not be executed according to the essential execution order E_0 and may be executed by any execution order E as long as $(P,E_0)\overset{M}{=}(P,E)$ holds. In this section we study a special class of execution orders called type 1 execution orders. This execution order is defined for each loop index I_u(u=1,2,...,n) and hence there are n of these.

Definition 1:

A type 1 parallel execution order with respect to I_u (we write 1-p w.r.t. I_u) is given by

$$E(P) = \{((i(1..n)),V((i(1..u-1),i(u+1..n))|(|L(1..u-1)|,|L(u+1..n)|)) \\ |(1(n)) \leq (i(1..n)) \leq (|L(1..n)|)\},$$

and is represented by $E[I_u]$.

Figures 6.1 and 6.2 illustrate execution orders E_0 and $E[I_u]$.

Note that $E[I_u]((i(1..n))) = E[I_u]((i'(1..n)))$ if $i_k = i'_k$ for all $k = 1,2,\ldots,u-1,u+1,\ldots,n$. Furthermore note that if

$$V((i(1..u-1))|(|L(1..u-1)|)) > V((i'(1..u-1))|(|L(1..u-1)|)),$$

then $E[I_u]((i(1..n))) > E[I_u]((i'(1..n)))$.

By introducing extra $|L_u|$ PE's, the computation time becomes one $|L_u|$-th of the original, i.e. $\pi_{\substack{j=1 \\ j\neq u}}^{n} |L_j|$ steps instead of $\pi_{j=1}^{n} |L_j|$ steps, where one step corresponds to the computation of a body statement.

We now introduce TRANQUIL notation [2] to illustrate Definition 1. In TRANQUIL

for (I) seq (L) do S

stands for

for I := (for list set) do S.

Also in TRANQUIL for (I) sim (L) do S indicates that statements S(L(i)) are executed simultaneously for all L(i) in L. Then Definition 1 amounts to obtaining

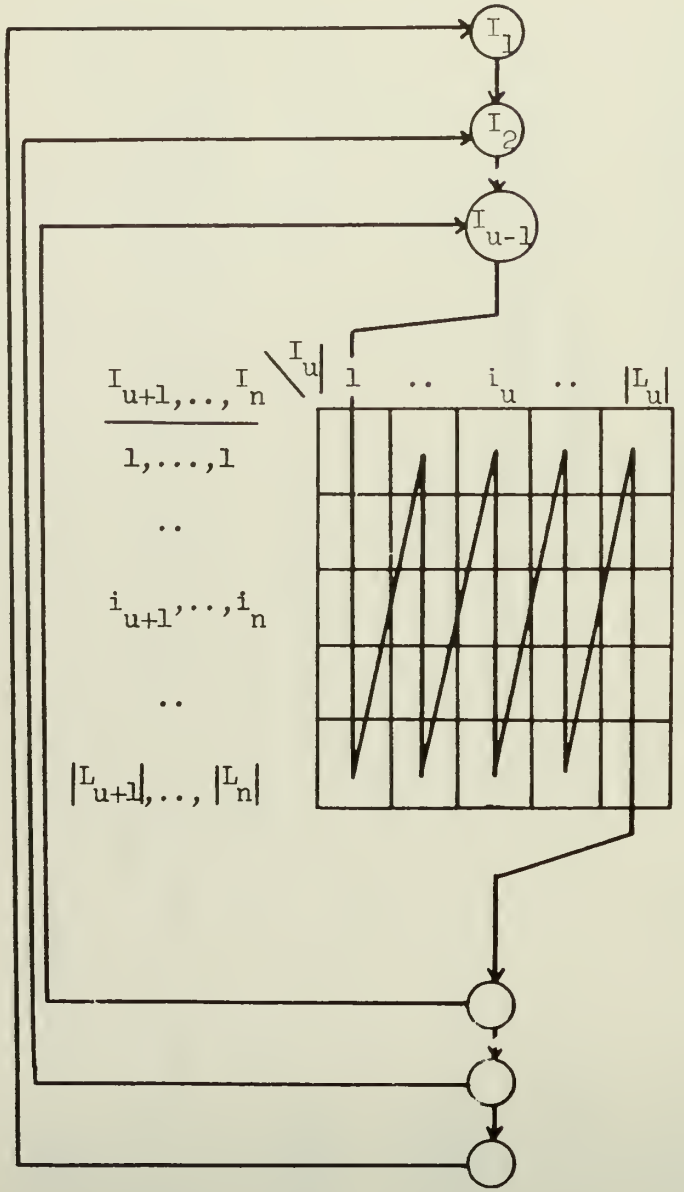


Figure 6.1. E_0

1) The grid is labeled by $1,2,..,i_u,...,$etc. rather than by $L_u(1),...,L_u(i_u)$, ...,etc. for convenience.
2) FORK→(a set of statements) →JOIN indicates the parallel execution of a set of statements[13].

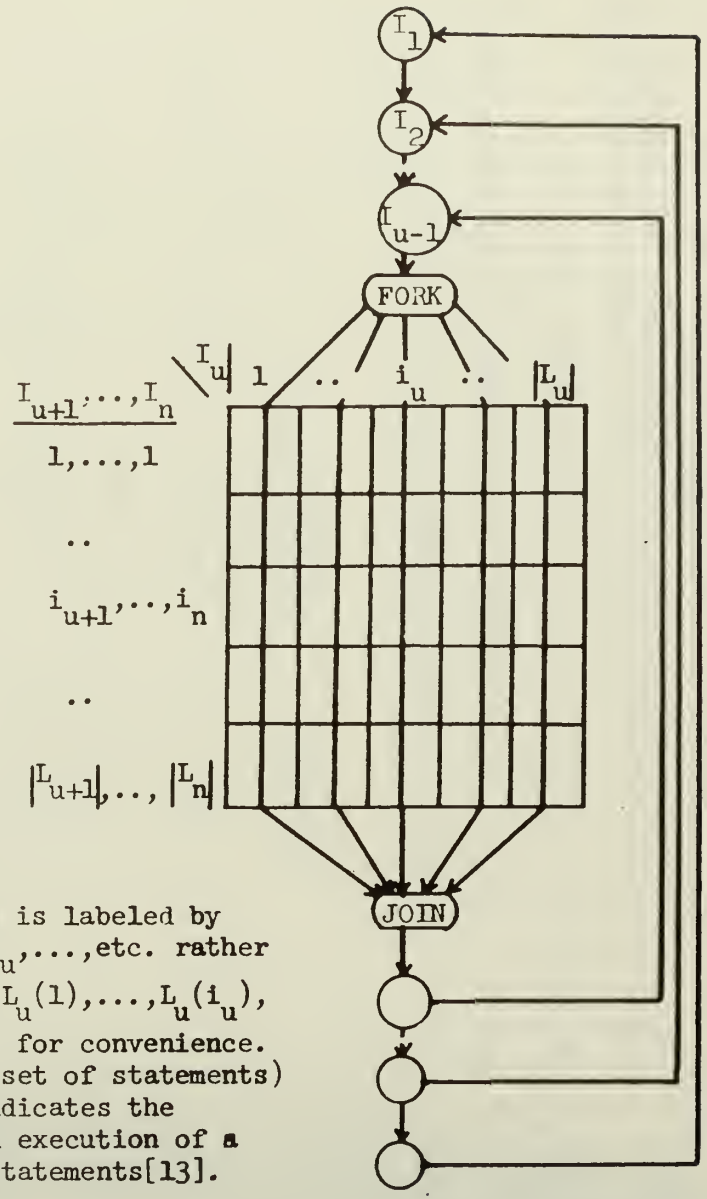


Figure 6.2. $E[I_u]$

for (I_1) seq (L_1) do

...

for (I_{u-1}) seq (L_{u-1}) do

for (I_u) sim (L_u) do

for (I_{u+1}) seq (L_{u+1}) do

...

for (I_n) seq (L_n) do S

from

for (I_1) seq (L_1) do

...

for (I_u) seq (L_u) do

...

for (I_n) seq (L_n) do S.

First we study a type 1 parallel execution order for a general loop in detail. Let the two-dimensional plane $I_u - I_{u+1} \times \cdots \times I_n$ be an $|L_u|$ by $\pi_{j=u+1}^{n} |L_j|$ grid (see Figure 6.3).

The grid is labeled by $1,2,\ldots,i_u,\ldots$etc. rather than by $L_u(1), L_u(2), \ldots, L_u(i_u), \ldots$ etc. for convenience.

Note that each square of the plane represents the computation $S((i(1..u-1),i(u..n)))$ for some $(i(1..u-1))$. If P' is executed by E_0, then the

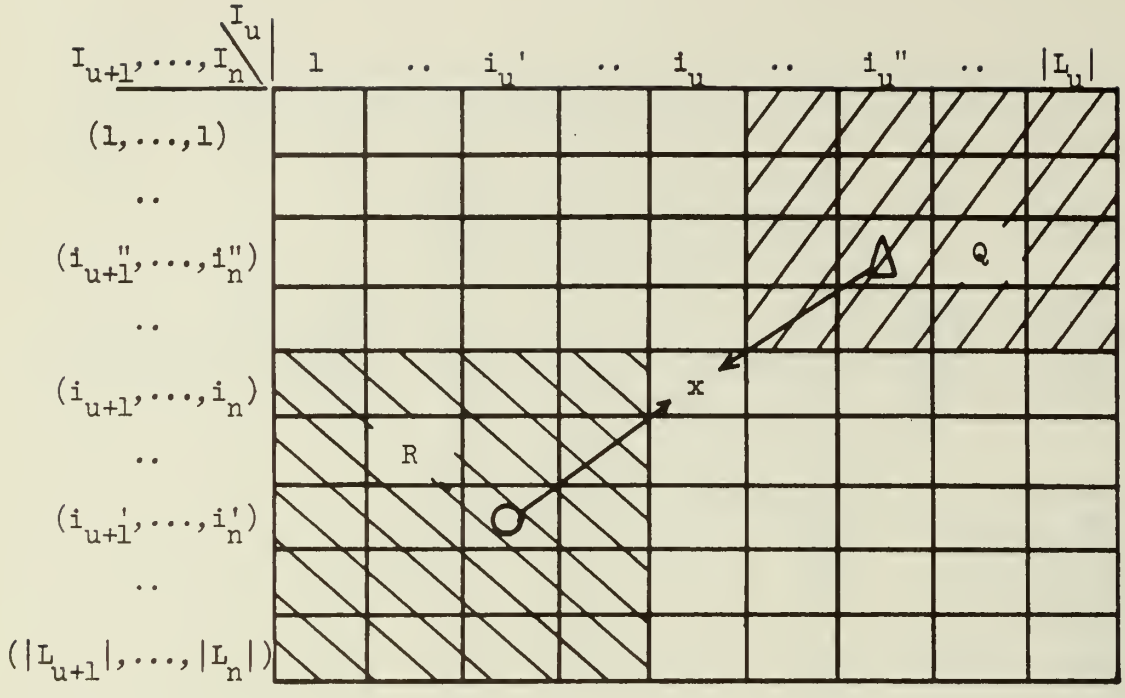


Figure 6.3. Conditions of Parallel Computation in a Loop

computation proceeds from the leftmost column to the rightmost column while in a column the computation proceeds from the top to the bottom sequentially. On the other hand if P^1 is executed by $E[I_u]$ then we proceed to compute from the top row to the bottom row while we perform computation in each row simultaneously. Each computation S((i(1..u-1),i(u..n))) uses inputs IN(S((i(1..n))) and updates the output OUT(S((i(1..n)))). Then as we studied in Chapter 5, we have to make sure that the computation S((i(1..n))) (marked x in Figure 6.3) does not receive any data which are to be updated by the computation in the region R, i.e. there must be no id such that

$$(\underline{id},\ (i(1..u-1),i'(u..n)),(i(1..n))) \in DR$$

holds where $(i'(u..n)) \geq (i(u..n))$ and $i_u' < i_u$.

Similarly the computation $S(\{i(1..n)\})$ must not use any data which are to be updated by the computation in the region Q, i.e. there must be no $\underline{id}'$ such that

$$(\underline{id}', (i(1..n)), (i(1..u-1), i''(u..n))) \in LR$$

holds where $(i''(u..n)) < (i(u..n))$ and $i_u'' > i_u$.

The above observation gives the following theorem.

Theorem 1:

Let $E[I_u]$ be a type 1 parallel execution order w.r.t. I_u. Then $(P^1, E[I_u]) \overset{M}{=} (P^1, E_0)$ if and only if there are no $\underline{id}$, $\underline{id}'$, $(i(1..n))$, $(i(1..u-1), i'(u..n))$ and $(i(1..u-1), i''(u..n))$ for which either[#]

(1) (i) $i_u' < i_u$ and

(ii) $(i'(u+1..n)) \geq (i(u+1..n))$ and

(iii) $(\underline{id}, (i(1..u-1), i'(u..n)), (i(1..n))) \in DR$, where $\underline{id} \in OUT(S((i(1..u-1), i''(u..n))))$ and $\underline{id} \in IN(S((i(1..n)))$

or (2) (i) $i_u'' > i_u$ and

(ii) $(i''(u+1..n)) < (i(u+1..n))$ and

(iii) $(\underline{id}', (i(1..n)), (i(1..u-1), i''(u..n))) \in LR$ where $\underline{id}' \in OUT(S((i(1..u-1), i''(u..n))))$ and $\underline{id} \in IN(S((i(1..n)))$

hold.

[#] $(i'(u+1..n)) \geq (i(u+1..n))$, for example, means $V((i'(u+1..n))|B) \geq V((i(u+1..n))|B)$ where $B = (|L(u+1..n)|)$. Unless specified, the base $(|L(s..n)|)$ $(=(|L_s|, \ldots, |L_r|))$ is to be understood for $(i(s..n))$.

Let S be of a form $A^0 := f(A^1,\ldots,A^p)$ where A is an array identifier and the superscript is used to distinguish different occurrences of A. Then id in the first condition of Theorem 1 corresponds to those $A^h[(i(1..n))]$ for which $A^0[(i(1..u-1),i'(u..n))] = A^h[(i(1..n))]$ holds together with the three conditions (i), (ii), and (iii) of (1) between (i'(u..n)) and (i(u..n)). Similarly id' corresponds to those $A^h[(i(1..n))]$ for which $A^0[(i(1..u-1),$ $i'(u..n))] = A^h[(i(1..n))]$ holds. Thus $A^h(1 \leq h \leq p)$ can be classified into three groups:

$C1 = \{h | A^h$ satisfies the first condition$\}$

$C2 = \{h | A^h$ satisfies the second condition$\}$

$C3 = \{1,2,\ldots,p\} - C1 - C2.$

Note that $C1 \cap C2 = \emptyset$.

Example 2:

Let P^1 be

$$(I_1 \leftarrow (1,2,3),\ I_2 \leftarrow (1,2,3))(A^0[I_1,I_2] := f(A^1[I_1\text{-}1, I_2\text{+}1])).$$

Then for $i_1' = 1 < i_2 = 2$ and $(i_2') = (3) > (i_2) = (2)$, we have $A^0[(i_1',i_2')] = A^1[(i_1,i_2)] = A[1,3]$, or $(A[1,3], (1,3), (2,2)) \in DR$. Thus P^1 cannot be computed in 1-p w.r.t. I_1, and $C1 = \{1\}$.

From this argument it should be clear that if a body statement is of Form (2-i) (see Section 6.2.1), then the loop can be computed in 1-p w.r.t. any $I_u (u=1,2,\ldots,n)$.

6.2.2.2 A Restricted Loop

If a loop is a restricted loop, then Theorem 1 may be simplified. First we define a vector R(h) for each h = 1,2,...,p:

$$R(h) = (R_1(h),\ldots,R_n(h)),$$

where

$$\begin{aligned} R_j(h) &= F(0,j) - F(h,j) \\ &= I_j + w(0,j) - (I_j + w(h,j)) \\ &= w(0,j) - w(h,j). \end{aligned}$$

For example we get R(1) = (-1,8) from a statement

$$A^0[I_1-1, I_2+3] := f(A^1[I_1, I_2-5]).$$

Then we use these vectors to check parallel computability as follows. Also for convenience we write

$$R'(u,h) = (R_1(h),\ldots,R_{u-1}(h))$$

and

$$R''(u,h) = (R_{u+1}(h),\ldots,R_n(h)).$$

Theorem 2:

If one of the following two holds for any of R(h) (h = 1,2,...,p), then P^1 cannot be computed in 1-p w.r.t. I_u.

(1) (i) $R'(u,h) = (0,\ldots,0)$ and

(ii) $R_u(h) > 0$ and

(iii) $R''(u,h) \leq (0,\ldots,0)$ and $\forall_j(u+1 \leq j \leq n) \mid R_j(h)| \leq |L_j| - 1$.

(2) (i) $R'(u,h) = (0,\ldots,0)$ and

(ii) $R_u(h) < 0$ and

(iii) $R''(u,h) > (0,\ldots,0)$ and $\forall_j(u+1 \leq j \leq n) \mid R_j(h)| \leq |L_j| - 1$.

That the theorem is valid is the direct consequence of Theorem 1, i.e. the first check of the theorem corresponds to the first condition of Theorem 1 and the second check corresponds to the second condition. For example the first condition of Theorem 1 says that if

$(\underline{id}, (i(1..u-1), i'(u..n)), (i(1..n))) \in DR$ holds for $i_u' < i_u$ and

$(i'(u+1..n)) \geq (i(u+1..n))$, then P^1 cannot be computed in 1-p w.r.t. I_u, where

$\underline{id} \in OUT(S((i(1..u-1), i'(u..n)))$

and $\underline{id} \in IN(S((i(1..n))))$.

Then $\underline{id}$ represents the element of A for which

$$A^h[(i(1..n))] = A^O[(i(1..u-1), i'(u..n))]$$

holds. Now this implies that

$$F(h,j)(L_j(i_j)) = F(0,j)(L_j(i_j))$$

for $j < u$ and

$$F(h,j)(L_j(i_j)) = F(0,j)(L_j(i_j'))$$

for $j \geq n$. Hence

$$L_j(i_j) + w(h,j) = L_j(i_j) + w(0,j)$$

or $R_j(h) = 0$ for $j < u$, and

$$L_j(i_j) + w(h,j) = L_j(i_j') + w(0,j)$$

or

$$i_j' = i_j - R_j(h)$$

for $j \geq u$. Also

$$(i'(u+1..n)) \geq (i(u+1..n))$$

with

$$B = (|L(u+1..n)|)$$

becomes

$$V((i_{u+1}-R_{u+1}(h),\ldots,i_n-R_n(h))|B) \geq V((i(u+1..n))|B).$$

Then by Lemma 1,

$$V((R_{u+1}(h),\ldots,R_n(h))|B) \leq V((0,\ldots,0)|B).$$

Thus the first check of Theorem 2 is verified. The second check can be varified similarly.

Now let us consider the number of checks required. For each A^h(h=1,2, ...,p) which appears on the right hand side of S, we first obtain a vector R(h). Then for each loop index I_u, we perform the two checks given by Theorem 2 for all R(h) (h=1,2,...,p). Since there are n loop indicies, in total we perform 2np checks.

The procedure described in this section can be extended to cover nonrestricted loops, too. Let S be of a form

$$A^0[F(0,1),\ldots,F(0,n)] := f(A^1[F(1,1),\ldots,F(1,n)],\ldots,A^p[(F(p,1),\ldots])$$

and we define a vector R(h) for each h = 1,2,...,p as we did before, i.e.

$$R(h) = (R_1(h), \ldots, R_n(h))$$

and

$$R_j(h) = F(0,j) - F(h,j).$$

Since a loop is not restricted, F(0,j) and F(h,j) may take any form and hence $R_j(h)$

may not be a constant number but rather a function of loop indicies, e.g. $R_j(h) = I_3 + 2I_4 - 5$. Hence, in the most general case, it is necessary to check the two conditions of Theorem 2 for all values of $(i(1..n))$ (i.e. $(1(n)) \leq (i(1..n)) \leq (|L_1|, |L_2|, \ldots, |L_n|)$) to examine type 1 parallel computability, i.e. $2(\prod_{j=1}^{n} |L_j|)$ checks are required for each R(h) (h=1,2,...,p). In many cases, we can expect that the number of checks required is far smaller than that. For example if

$$R(1) = (2I_1, 2I_1 - I_3, I_3, 2I_3 + I_1),$$

then only $2(|L_1| \times |L_3|)$ checks are required, i.e. it is not necessary to check for those loop indicies, e.g. I_2, which do not appear in $R_j(1)$ (j=1,2,3,4).

6.2.2.3 Temporary Locations

In this section we mean a restricted loop by a loop. The second condition of Theorem 1 (or 2) may be dropped by introducing extra temporary locations by applying Transformation T_1 of Chapter 5 on P^1, i.e. if $C1 = \emptyset$ and and $C2 \neq \emptyset$, then temporary locations may be set up so that P^1 can be computed in parallel (for C1 and C2, see Section 6.2.2.1). Let $h \in C2$. This implies that there are $(i(1..n))$, $(i(1..u-1), i'(u..n))$ and $\underline{id}$ (see Figure 6.4) for which

$$(\underline{id}, (i(1..n)), (i(1..u-1), i'(u..n))) \in LR$$

holds and

$$\underline{id} = A^h[(i(1..n))] \in IN(S((i(1..n))))$$

and

$$\underline{id} = A^{0}[(i(1..u-1),i'(u..n))] \in OUT(S((i(1..u-1),i'(u..n))))).$$

If a loop is computed in l-p w.r.t. I_u, then we have

$$E[I_u]((i(1..u-1),i'(u..n))) < E[I_u]((i(1..n)))$$

while

$$E_0((i(1..u-1),i'(u..n))) > E_0((i(1..n))).$$

Hence $A^{h}[(i(1..n))]$ will be updated by $S((i(1..u-1),i'(u..n)))$ before being used by $S((i(1..n)))$. That is, if we compute P^{1} in l-p w.r.t. I_u we must keep the old value of $A^{h}[(i(1..n))]$ which otherwise will be updated by $S((i(1..u-1),i'(u..n)))$ at the $E[I_u]((i(1..u-1),i'(u..n)))$-th step separately until it is used by $S((i(1..n)))$ at the $E[I_u]((i(1..n)))$-th step. The period of time, t_h, through which the old value of $A^{h}[(i(1..n))]$ must be kept for the computation $S((i(1..n)))$ is given by

$$t_h = E[I_u]((i(1..n))) - E[I_u]((i(1..u-1),i'(u..n)))$$

$$= V((i(u+1..n))|B) - V((i'(u+1..n))|B),$$

where $B = (|L(u+1..n)|)$. Then as we showed in Section 6.2.2.2, in case of a restricted loop, we can show that

$$t_h = V(R''(u,h) \mid B) + \sum_{s=u+1}^{n} B_s - 1$$

where $B_s = \prod_{t=s}^{n} |L_{t+1}|$. The details are omitted.

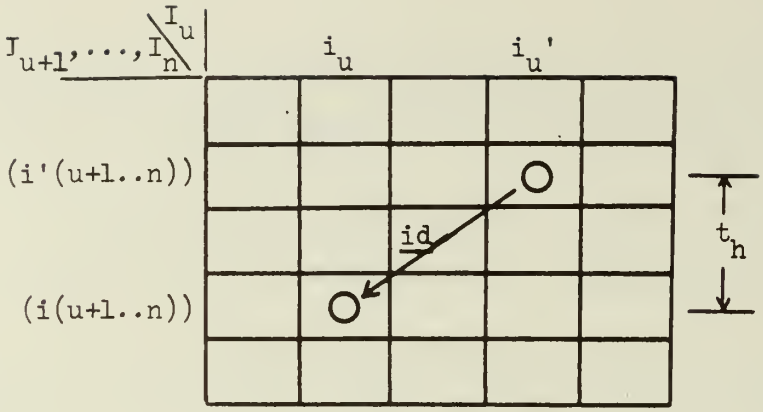


Figure 6.4. An Illustration of t_h

Now $\max_{h \in C2} t_h$ gives the maximum period of time through which $A^h[(i(1..n))]$ must be kept. Since we have $|L_u|$ of them (i.e. $|L_u|$ statements are computed simultaneously), the total amount of temporary locations required will be $|L_u| \times \max t_h$. Additional $|L_u|$ locations are required for buffering (see Example 3). Hence we have the following theorem.

Theorem 3:

The maximum number of temporary storage locations required is

$$|L_u| \times (\max_{h \in C2} [V((R_{u+1}(h), \ldots, R_n(h))|B) + \sum_{s=u+1}^{n} B_s])$$

where $B = (|L(u+1..n)|)$ and $B_s = \prod_{t=s}^{n} |L_{t+1}|$ and $B_n = 1$.

Example 3:

Let P^1 be

for (I_1) seq (1,2,...,40) do

for (I_2) seq (1,2,...,40) do

$A[I_1,I_2] := A[I_1+2,I_2-3] + 2;$

P^1 as it is cannot be computed in 1-p w.r.t. I_1 because it violates the second condition of Theorem 2. Now we modify P^1 as follows by introducing temporary arrays T1(40×1) and T2(40×3).

for (I_1) seq (1,2,...,40) do

for (I_2) seq (1,2,...,40) do

begin S1: $T1[I_1] := A[I_1+2,I_2]$;

S2: $A[I_1,I_2] := T2[I_1, I_2 \text{ mod } 3] + 2$;#

S3: $T2[I_1, I_2 \text{ mod } 3] := T1[I_1]$;

end.

Then all three statements can be computed in 1-p w.r.t. I_1, i.e. we can replace seq in the first for statement by sim. The original P^1, if executed sequentially, takes 1600 steps whereas the modified P^1 takes only 120 steps if executed in parallel with respect to I_1.

$a \text{ mod } b \equiv a - \left\lceil \frac{a}{b} \right\rceil \cdot b$

Also we assume that T2 is properly initialized before the computation of the loop, i.e. store A[1,*], A[2,*] and A[3,*] in T2[1,*], T[2,*] and T[0,*].

6.2.3 Type 2 Parallelism

In this section we mean a restricted loop by a loop. Since the conflict between two statements S(i) and S(j) due to the existence of an identifier id such that (id,i,j) ϵLR may be resolved by introducing temporary locations (see Chapter 5 and the previous section), such conflict will not be taken into account to check parallel computability throughout the rest of this chapter.

This section describes the second type of parallelism, i.e. type 2 parallelism, in a for statement with a single body statement. Type 2 parallelism is introduced to resolve the conflict due to the first condition of Theorem 1. The following example illustrates it.

Example 4:

P: for I_1 := 1 step 1 until 40 do

for I_2 := 1 step 1 until 40 do

$A^0[I_1,I_2]$:= $A^1[I_1-1,I_2+1]$ + $A^2[I_1,I_2-1]$;

Since $R_1(1) = 1 > 0$ and $(R_2(1)) = (-1) < (0)$ hold, P cannot be computed in 1-p w.r.t. I_1.

Now let us consider the I_1-I_2 plane (Figure 6.5). Suppose that all $S((i_1,i_2))$ in the shaded area have been computed. Then at the next step those $S((i_1',i_2'))$ marked as (1) can be computed simultaneously, and at the following step all (2) can be computed simultaneously, and so forth. We can see that a heavy zigzag line travels from left to right like a "wave front" indicating that all statements on that front can be computed simultaneously.

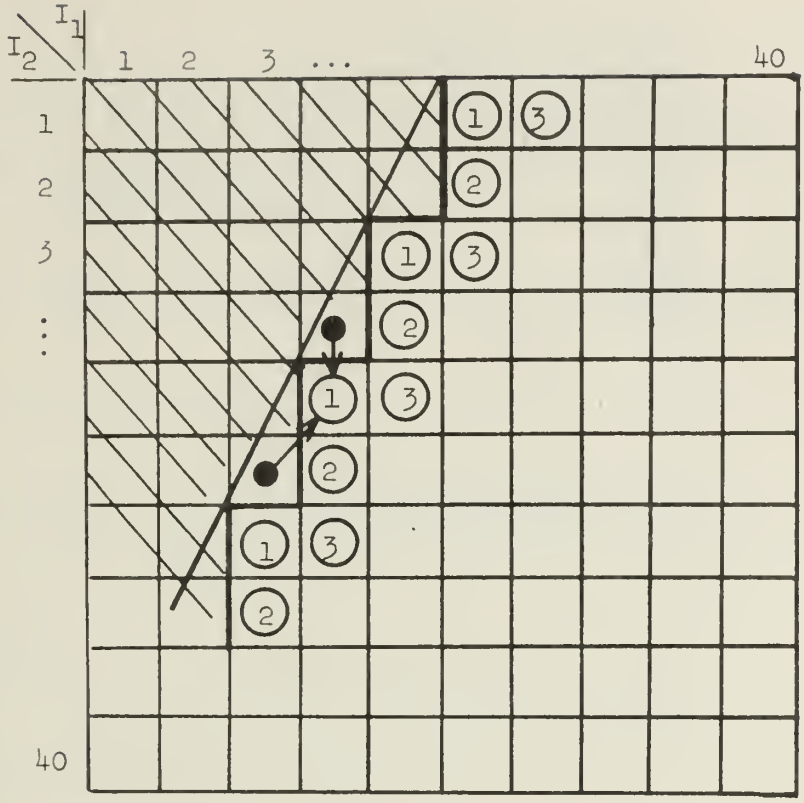


Figure 6.5. Wave Front

Note that computation of P by this scheme takes approximately 120 steps, while if P is computed sequentially it takes 40 × 40 = 1600 steps.

Given a loop P^1, if P^1 is computed in l-p w.r.t. I_u, then a "wave front" is in parallel with the I_u axis of the I_u - $I_{u+1} \times \cdots \times I_n$ plane, and it travels in the increasing order of (i(u+1..n)).

If P^1 cannot be computed in l-p w.r.t. I_u then it may be possible to find a "wave front" which is diagonal rather than horizontal as in Example 4 on the I_u - $I_{u+1} \times \cdots \times I_n$ plane.

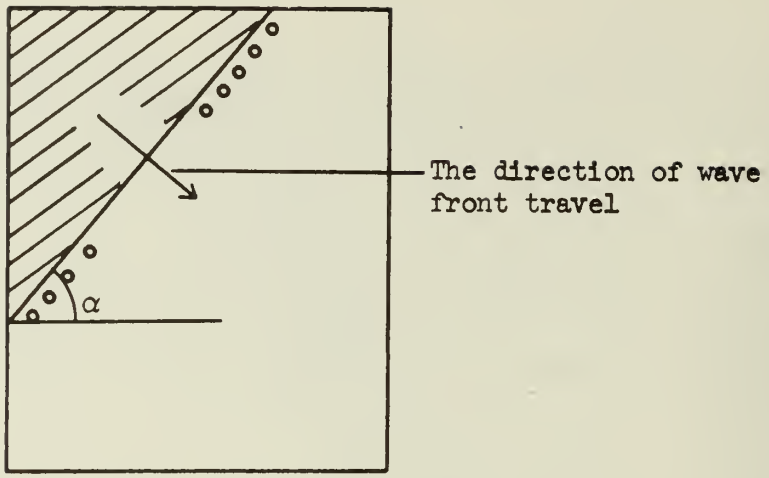


Figure 6.6. Wave Front Travel

This wave front is such that all computations S((i(1..n))) which corresponds to points $(i_u, i_{u+1}, \ldots, i_n)$ which lie right next to a wave front can be computed simultaneously. In other words all necessary data to compute S((i(1..u-1), i(u..n))) have been already computed in the shaded area. The direction of a wave front's travel is perpendicular to the wave front.

Now let us obtain the slope of a possible wave front for a restricted loop.

Let P^1 be a restricted loop. Assume that P^1 cannot be computed in 1-p w.r.t. I_u. Then according to Theorem 2, this means that there are R(h) for which $R_u(h) > 0$ and $(R_{u+1}(h), \ldots, R_n(h)) \leq (0, \ldots, 0)$ hold (i.e. $Cl \neq \emptyset$).

Theorem 4:

The slope of a possible wave front in the $I_u - I_{u+1} \times \ldots \times I_n$ plane is given by

$$\max_{h \in Cl} \left[\frac{1}{R_u(h)} \left((-V((R_{u+1}(h), \ldots, R_n(h)) \middle| B) - \sum_{s=u+1}^{n} B_s + 2 \right) \right]$$

where $B = (|L(u+1..n)|)$ and $B_s = \prod_{t=s}^{n} L_{t+1}$ and $B_n = 1$.

In Example 4, the slope of the wave front is $\frac{1}{1}(-(-1-1+1)-1+2) = 2$.

Proof of Theorem 4:

Let us consider $S((i(1..u-1), I(u..n)))$ on the $I_u - I_{u+1} \times \ldots \times I_n$ plane. Assume that there is a variable id such that (Figure 6.7)

$$(\underline{id}, (i(1..u-1), i'(u..n)), (i(1..u-1), i(u..n))) \in DR$$

holds together with

$$i_u' < i_u \text{ and } (i'(u+1..n)) \geq (i(u+1..n)),$$

i.e.

$$\underline{id} \in IN(S((i(1..u-1), i(u..n))))$$

and

$$\underline{id} \in OUT(S((i(1..u-1), i'(u..n)))).$$

This implies that there is $h \in Cl$ such that

$$A^h[(i(1..u-1), i(u..n))] = A^0[(i(1..u-1), i'(u..n))]$$

holds.

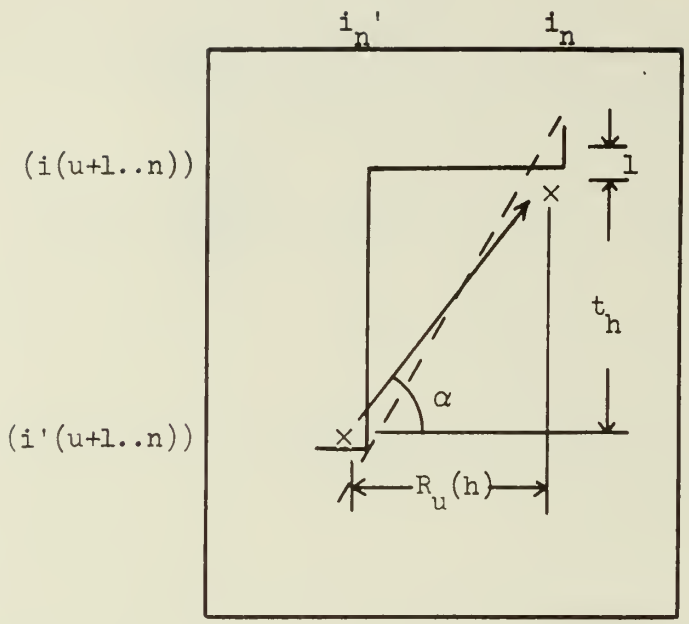


Figure 6.7. An Illustration for Theorem 4

In case of a restricted loop, we have

$$i_j' = i_j - R_j(h)$$

for $j \geq u$. Now let

$$t_h = V((i'(u+1..n))|B) - V((i(u+1..n))|B)$$

then we get

$$t_h = -V((R_{u+1}(h),\ldots,R_n(h))|B) - \sum_{s=u+1}^{n} B_s + 1 \text{ where } B_s = \prod_{t=s}^{n} |L_{t+1}| \text{ and}$$

$B_n = 1$. Now if we let the slope of a wave front be equal to

$$\frac{t_h + 1}{i_u - i_u'} = \frac{t_h + 1}{R_u(h)}$$

then $A^h[(i(1..n))]$ and $A^0[(i(1..u-1),i'(u..n))]$ will be separated by it (Figure 6.7).

The actual wave front is a zigzag line, rather than a straight line as shown in Figure 6.7.

If there are more than one h in Cl, then we choose α to be large enough so that all inputs to S((i(1..u-1),i(u..n))) be inside of a wave front, i.e.

$$\tan \alpha = \max_{h \in Cl} \frac{t_h + 1}{R_u(h)}.$$

(Q.E.D.)

Now suppose we compute P^1 in parallel w.r.t. I_u using a diagonal wave front whose slope is D = tan α. Then how many steps (one step corresponds to the computation of a body statement S) does it take to compute P^1?

Theorem 5:

The total number of steps required to compute P^1 in parallel w.r.t. I_u using a diagonal wave front whose slope is D is given by

$$T_p = \left(\prod_{j=1}^{u-1} |L_j| \right) \left(\prod_{j=u+1}^{n} |L_j| + |L_u| D \right).$$

Proof:

Let us consider the $I_u - I_{u+1} \times \ldots \times I_n$ plane.

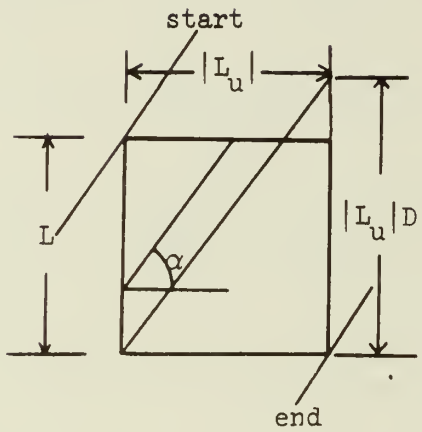


Figure 6.8. An Execution by a Wave Front

Wave front W must travel from the start position to the end position on the plane. How long does it take? It takes $L + |L_u|D$ steps where $L = \pi_{j=u+1}^{n} |L_j|$.

Since we have to process $\pi_{j=1}^{u-1} |L_j|$ $I_u - I_{u+1} \times \ldots \times I_n$ planes, in total it becomes $T_p = \pi_{j=1}^{u-1} |L_j|(L + |L_u|D)$.

(Q.E.D.)

Note that if a wave front is horizontal (i.e. if P^1 can be computed in 1-p w.r.t. I_u), then $D = 0$ and $T_p = \pi_{\substack{j=1 \\ j \neq u}}^{n} |L_j|$.

6.2.4 Conclusion

Assume that there are an arbitrary number of PE's available. Given a restricted single body statement loop,

$$P^1 = (I_1,\ldots,I_n)(A^0[F_1^0,\ldots,F_n^0] := f(A^1[F_1^1,\ldots,F_n^1],\ldots,A^p[F_1^p,\ldots,F_r^p]))$$

we can check if P^1 can be computed in l-p w.r.t. $I_u(u=1,\ldots,n)$ by Theorem 2. If it cannot be, then we can check for type 2 parallel computability w.r.t. I_u, i.e. find a possible wave front. In either case we obtain the number of computational steps required, i.e. $T_u = \prod_{\substack{j=1 \\ j\neq u}}^{n} |L_j|$ or $T_u = (\prod_{j=1}^{u-1} |L_j|)(\prod_{j=u+1}^{n} |L_j| + |L_u|\cdot D)$, where one step corresponds to the computation of the body statement S. Then among all possible choices, we would choose to compute in parallel w.r.t. I_u where $T_u = \min_{1\leq j\leq n} T_j$.

6.3 A Loop With Many Body Statements

6.3.1 Introduction

In what follows we mean a restricted loop by a loop. Again a check against all published Algorithms in 1968 and 1969 CACM issues has been done, and it has been revealed that well over 50 percent of the cases of for statement usage (with more than two-body statements) are instances of restricted loops.

Also as stated in Section 6.2.2.3 and Chapter 5, the LR relation may be disregarded by introducing temporary locations. Hence it will not be taken into account throughout the rest of this chapter.

Given a loop with m body statements, P^m, there are three different approaches to compute it in parallel. First it is possible to extend the procedure described in Section 6.2 by treating m body statements as if they were one statement. That is, we consider body statements as a function

OUT(S(i(1..n))) = f(IN(S((i(1..n))))).

For example

```
S1:  A[I,J] := f(A[I,J-1],B[I-1,J-1]);
S2:  B[I,J] := g(A[I-1,J-1],B[I,J-1])
```

yield

```
S:  {A[I,J],B[I,J]} := f'(A[I,J-1],A[I-1,J-1],B[I-1,J-1],B[I,J-1]).
```

Then we can apply Theorem 1 directly to check if e.g. S can be computed in 1-p w.r.t. I.

The second and the third approaches can be illustrated by the following two examples.

```
E1:  for I := 1 step 1 until 40 do
     begin
          S1:  A[I] := f(A[I],B[I]);
          S2:  B[I] := g(A[I],B[I-1]);
     end;
E2:  for I := 1 step 1 until 40 do
     begin
          S1:  A[I] := f(A[I-1],B[I-2]);
          S2:  B[I] := g(A[I]);
     end.
```

In E1, note that S1 and S2 cannot be computed in parallel for all values of I because S2 has an iteration form

B[I] := g'(B[I-1]).

However, E1 may be replaced with two for statements:

```
for I := 1 step 1 until 40 do
    S1:  A[I] := f(A[I],B[I]);
for I := 1 step 1 until 40 do
    S2:  B[I] := g(A[I],B[I-1]);
```

Now the first loop can be computed in parallel for all values of I while the second for statement is still an iteration. In general by replacing a single for statement with two or more for statements the parallel part may be exposed.

In the second example, S1 and S2 can not be computed in parallel for all values of I, nor can they be separated into two independent loops because S1 uses values which are updated by S2 (i.e. B[I-2]), and S2 uses values being updated by S1 (i.e. A[I]). However S1 and S2 could be computed simultaneously while I varies sequentially if the index expression in S2 is "skewed" as follows.#

```
E2' for I := 1 step 1 until 40 do
    begin
        S1:  A[I] := f(A[I-1],B[I-2]);
        S2':  B[I-1] := g(A[I-1]);
    end.
```

#Strictly speaking S2' should not be executed when I = 1 and an extra statement S2": B[40] := g(A[40]) is required after this loop. For the sake of brevity those minor boundary effects are ignored throughout this section.

Figure 6.9 illustrates the computation of the modified loop as well as the original loop.

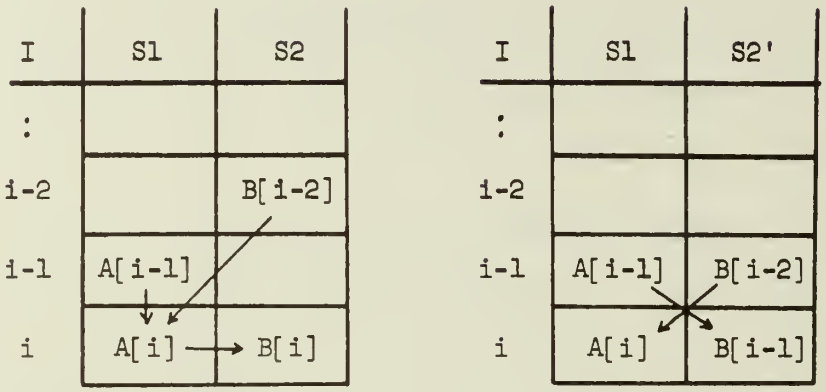


Figure 6.9. Simultaneous Execution of Body Statements

In general, the above three approaches could be tried in any combination. For example, we may first try the first approach, i.e. we try to execute body statements simultaneously for all values of some loop index. If this fails, then we may use the second approach, i.e. we separate a loop or we replace a loop with as many for statements as possible. On a resultant for statement we again try the first approach (If it has only one body statement, then the results of the previous section can be used). If we fail again, then the third approach can be taken.

We now describe each approach separately.

Before we go further, we define the following notations. Without loss of generality we assume that the p-th occurrence of A_k appears in S_p and also assume that S_p and S_q have forms

$$S_p \equiv (A_k^p[F(k,p,1),\ldots,F(k,p,n)] := f_p(\ldots))$$

and

$$S_q \equiv (\ldots := f_q(\ldots,A_k^q[F(k,q,1),\ldots,F(k,q,n)],\ldots)).$$

Then we define a vector $R(k,p,q)$ as follows

$$R(k,p,q) = (R_1(k,p,q),\ldots,R_n(k,p,q))$$

where

$$R_j(k,p,q) = F(k,p,j) - F(k,q,j).$$

$$= w(k,p,j) - w(k,q,j).$$

If $F(k,p,j) = F(k,q,j) = \emptyset$, then we let $R_j(k,p,q) = \emptyset$. Finally we write

$$R'(u,k,p,q) = (R_1(k,p,q),\ldots,R_{u-1}(k,p,q))$$

and

$$R''(u,k,p,q) = (R_u(k,p,q),\ldots,R_n(k,p,q)).$$

6.3.2 Parallel Computation with Respect to a Loop Index

We first study the first approach described in the previous section, i.e. we treat body statements as if they were one statement and try to execute them in parallel with respect to some loop index.

Let us consider $P^m = (I_1,I_2,\ldots,I_n)(S_1;\ldots;S_m)$. Then we treat m body statements as one statement S where

$$OUT(S((i_1,i_2,\ldots,i_n))) = \bigcup_{p=1}^{m} OUT(S_p((i_1,i_2,\ldots,i_n)))$$

and

$$IN(S((i_1,i_2,\ldots,i_n))) = IN(S_1((i_1,i_2,\ldots,i_n))) \cup \bigcup_{p=2}^{m} [IN(S_p((i_1,i_2,$$

$$\ldots,i_n))) - \bigcup_{\ell=1}^{p-1} OUT(S_\ell((i_1,i_2,\ldots,i_m)))].$$

Having these two sets, we can use results of Section 6.3 directly. For example let us consider Theorem 2. Then we may modify Theorem 2 as follows. First suppose an array A_k^p appears in $OUT(S((i(1..n))))$ and A_k^q appears in $IN(S((i(1..n))))$. Then obtain $R(k,p,q)$.

Theorem 6: (cf. Theorem 2)

For each A_k in $OUT(S((i(1..n))))$, we obtain $R(k,p,q)$ for all q such that A_k^q is in $IN(S((i(1..n))))$. Then if there is any $R(k,p,q)$ which satisfies all three conditions described below, then S cannot be computed in type 1 parallel w.r.t. I_u. Conditions:

(1) $R_j(k,p,q) = 0$ or $\emptyset$ for all $j = 1,2,\ldots,u-1$,

(2) $R_u(k,p,q) > 0$, and

(3) there is $\ell (u+1 \leq \ell \leq n)$ such that $\forall_j (u+1 \leq j \leq \ell-1), R_j(k,p,q) = 0$ and $R_\ell(k,p,q) = \emptyset$ or $R_\ell(k,p,q) < 0$.

The above theorem can be proved similarly to Theorem 2, and the details will not be given.

Since we have to apply the above check for all $R(k,p,q)$ vectors, the number of checks required is proportioned to the total number of $R(k,p,q)$ vectors, $\#R(k,p,q)$. Also since we can try to compute S in type 1 parallel in n ways, i.e. with respect to $I_u (u=1,2,\ldots,n)$, the total number of checks we would perform is given by $n \times (\#R(k,p,q))$.

6.3.3 Separation of a Loop

6.3.3.1 Introduction

In this section we study replacement (or separation) of a single *for* statement with two or more *for* statements. Let

$$P^m = (I_1, I_2, \ldots, I_n)(S_1; S_2; \ldots; S_m)$$

$$= (I_1, \ldots, I_{u-1})\{(I_u, \ldots, I_n)(S_1; \ldots; S_m)\}$$

$$= (I_1, \ldots, I_{u-1})P_u^m.$$

For fixed values of $I_1, \ldots, I_{u-1}$, let us consider P_u^m.

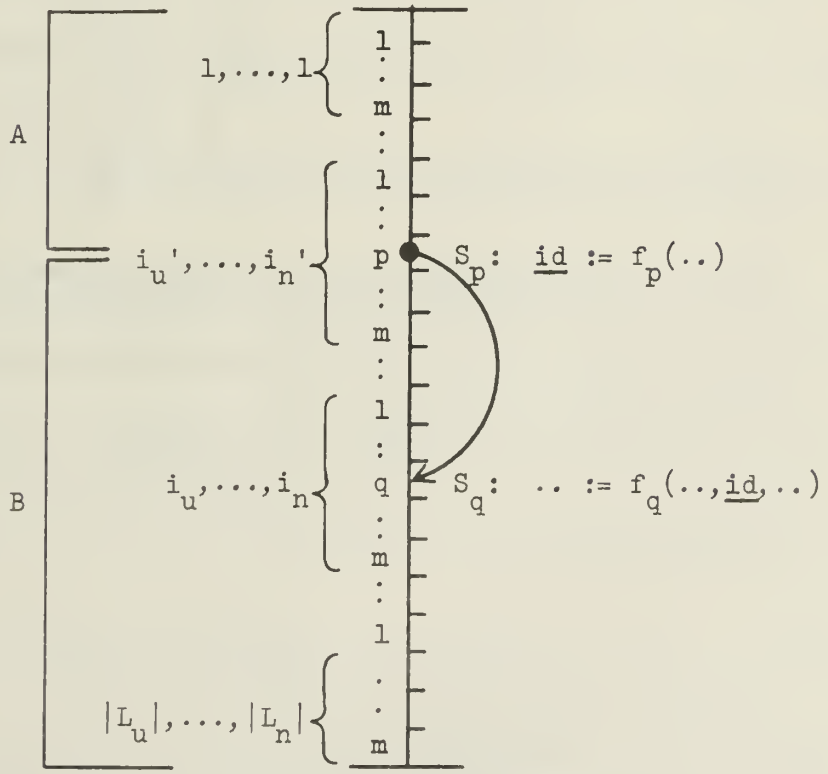


Figure 6.10. Execution of P_u^m

If we write down the primitive execution order $E_0(P_u^m)$, it can be represented by the straight line as shown in Figure 6.10. Now let us consider a statement S((i(1..n),q)) (See Figure 6.10). If it does not receive any computed results from part A, i.e. if there is no id and a statement S((i(1..u-1),i'(u..n),p)) for which

$$(\underline{id},(i(1..u-1),i'(u..n),p),(i(1..n),q))\epsilon DR$$

and $(i'(u..n),p) < (i(u..n),q)$

hold, then it may be computed independently of part A. If this holds for all p and all (i(u..n)) for some fixed q, then we can compute S_q before any S_p, i.e.

$$P_u^m \overset{M}{=} (I_u,\ldots,I_n)(S_q);(I_u,\ldots,I_n)(S_1;\ldots;S_{q-1};S_{q+1};\ldots;S_m).$$

Similarly if S((i(1..n),q)) does not give any output to part B, then we have

$$P_u^m \overset{M}{=} (I_u,\ldots,I_n)(S_1;\ldots;S_{q-1};S_{q+1};\ldots;S_m);(I_u,\ldots,I_n)(S_q).$$

6.3.3.2 The Ordering Relation (θ_u) and Separation of a Loop

Now we study how P_u^m may be replaced with several for statements. We first define the relation θ_u between body statements. The relation is such that if $\theta_u(p,q)$ holds, then for any given (i(u..n)) there are (i'(u..n)) and id such that

$$(\underline{id},\ (i(1..u-1),i'(u..n),p),(i(1..n),q))\epsilon DR$$

and $(i'(u..n)) < (i(u..n))$

hold. That is, for some fixed q, if there is no p for which $\theta_u(p,q)$ holds, then S_q can be computed before any S_p and

$$P_u^m \overset{M}{=} (I)(S_q);(I)(S_1;\ldots;S_{q-1};S_{q+1};\ldots;S_m)$$

where

$$(I) = (I_u,\ldots,I_n).$$

Definition 2:

Between two body statements, S_p and S_q we first obtain $R(k,p,q)$. Then $\theta_u(p,q)$ holds if and only if

(1) $R_j(k,p,q) = 0$ or $\emptyset$ for all $j = 1,2,\ldots,u-1$.

and (2) the first nonzero element of $R''(u,k,p,q)$ is either $\emptyset$ or a positive number. Also if all elements of $R''(u,k,p,q)$ are 0 then $p < q$ holds.

We also write $\theta_u = \{(p,q) \mid \theta_u(p,q) \text{ holds}\}$.

If A_k appears more than twice in S_q, then we modify Definition 2 as follows. Suppose A_k appears twice in S_q as q_1-th and q_2-th occurrences. Then we construct two vectors $R(k,p,q_1)$ and $R(k,p,q_2)$. For each vector we check the above two conditions. If at least one of two vectors satisfies the two conditions, then we let $\theta_u(p,q)$ hold.

Example 6:

S1: $A_1[I_1-1, I_2+3] := A_2[I_2, I_3] + A_4[I_1-1];$

S2: $A_2[I_2+1, I_3-1] := A_1[I_1, I_2-3] + A_3[I_1, I_3];$

S3: $A_3[I_1-5, I_3-1] := A_3[I_1, I_3] + A_1[I_1, I_2+1];$

S4: $A_4[I_1] := A_4[I_1-1] + A_2[I_2, I_3];$

give

$$R(1,1,2) = (-1,6,\emptyset), R(1,1,3) = (-1,2,\emptyset),$$

$$R(2,2,1) = (\emptyset,1,-1), R(2,2,4) = (\emptyset,1,-1),$$

$$R(3,3,2) = (-5,\emptyset,-1) \text{ and } R(4,4,1) = (1,\emptyset,\emptyset).$$

Then we have $\theta_1 = \{(2,1),(2,4),(4,1)\}$.

Now let us study Definition 2 in detail. First we note that id in Figure 6.10 corresponds to those A_k for which

$$F(k,p,j)(i_j) = F(k,q,j)(i_j)$$

holds for all $j = 1,2,\ldots,u-1$ and

$$F(k,p,j)(i_j') = F(k,q,j)(i_j)$$

holds for all $j = u,u+1,\ldots,n$. The former implies that

$$F(k,p,j) = F(k,q,j)$$

or

$$R_j(k,p,q) = 0 \text{ (or } \emptyset)$$

for $j = 1,2,\ldots,u-1$ which is equivalent to the first condition of Definition 2. Next note that $(i'(u..n),p) < (i(u..n),q)$ implies that either

(i) $(i'(u..n)) < (i(u..n))$

or (ii) $(i'(u..n)) = (i(u..n))$ and $p < q$.

In the first case

$$(F(k,h^p,u),\ldots,F(k,h^p,n)) > (F(k,h^q,u),\ldots,F(k,h^q,n))$$

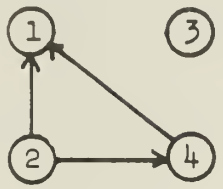
must hold. In the second case

$$(F(k,h^p,u),\ldots,F(k,h^p,n)) = (F(k,h^q,u),\ldots,F(k,h^q,n))$$

must hold. These two make up the second condition.

From θ_u we can construct a dependence graph D_u with m nodes each of which represents a body statement, e.g. from Example 6 we get:

D_1



In D_u we call a series of θ_u, $\theta_u(p_1,p_2),\theta_u(p_2,p_3),\ldots,\theta_u(p_i,p_{i+1}),\ldots,$ $\theta_u(p_{k-1},p_k)$ a chain and write $ch(p_1,p_k)$ for it. If $p_k = p_1$ then it is called a mesh M. We say that a node p_i is in the chain $ch(p_1,p_k)$ (or in the mesh M), or the chain $ch(p_1,p_k)$ (or the mesh M) includes p_i. Note that for nodes p and q there may be more than one chain which connects p to q.

Now let

$$Z_s = \{p \mid \text{there is no q such that } \theta_u(q,p) \text{ holds}\}$$

and

$$Z_e = \{p \mid \text{there is no q such that } \theta_u(p,q) \text{ holds}\}.$$

Furthermore let

$$PD(p) = \{q \mid ch(q,p) \text{ exists}\} \cup \{p\}$$

and

$$SC(p) = \{q \mid ch(p,q) \text{ exists}\} \cup \{p\}.$$

(PD for predecessors and SC for successors). Then we classify nodes in D_u as follows:

$$Z_1 = \{p \mid \text{For all } r \in PD(p), \text{ there is no mesh in } D_u \text{ which includes r}\},$$

$$Z_3 = \{p \mid \text{For all } r \in SC(p), \text{ there is no mesh in } D_u \text{ which includes r}\},$$

and

$$Z_2 = N - Z_1 - Z_3.$$

Let Z_1(or Z_3) = $\{p_1, p_2, \ldots, p_u\}$. Then we can order this set as $p_1', p_2', \ldots, p_u'$ in such a way that $\theta_u(p_i', p_j')$ does not hold if $i > j$.# Let us write Z_1^θ(or Z_3^θ) for a resultant ordered set. Also we order $Z_2 = \{q_1, q_2, \ldots, q_v\}$ as $Z_2^\theta = (q_1', q_2', \ldots, q_v')$ in such a way that $q_i' < q_j'$ if $i < j$.

Now given a loop P^m where

$$P^m = (I_1, I_2, \ldots, I_n)(S_1; S_2; \ldots; S_m)$$

$$= (I_1, \ldots, I_{u-1})\{(I_u, \ldots, I_n)(S_1; S_2; \ldots; S_m)\}$$

$$= (I_1, \ldots, I_{u-1})P_u^m,$$

we build the dependence graph D_u and obtain sets Z_1, Z_2 and Z_3, say $Z_1 = \{p_1, p_2, \ldots, p_u\}$, $Z_2 = \{q_1, q_2, \ldots, q_v\}$ and $Z_3 = \{r_1, r_2, \ldots, r_w\}$. (m=u+v+w).

From Z_1 and Z_3 we obtain ordered sets Z_1^θ and Z_3^θ, say $Z_1^\theta = (p_1', p_2', \ldots, p_u')$ and $Z_3^\theta = (r_1', r_2', \ldots, r_w')$. Also we have $Z_2^\theta = (q_1', q_2', \ldots, q_v')$. Then

$$P_u^m \overset{M}{=} (I)(S_{p_1'}); (I)(S_{p_2'}); \ldots; I(S_{p_u'}); (I)(S_{q_1'}; \ldots; S_{q_v'});$$

$$(I)(S_{r_1'}); (I)(S_{r_2'}); \ldots; (I)(S_{r_w'})$$

where $(I) = (I_u, I_{u+1}, \ldots, I_n)$.

#Note that Z_1(or Z_3) together with θ_u makes a graph which does not contain any mesh. To order Z_1 (or Z_3) the technique discussed in Chapter 7 may be used.

Thus we have replaced a loop P_u^m with as many for statements as possible. We say that p is separable from P^m if $p \epsilon Z_1$ (or $p \epsilon Z_3$) with respect to u, and that P^m is separable with respect to I_u. Also we say that P^m is separated with respect to I_u if P_u^m is replaced by many for statements as we showed above.

6.3.3.3 Temporary Storage

Now let us study the following:

P^2: for I_1 := 1 step 1 until 1 do

for I_2 := 1 step 1 until 40 do

begin

S1: $A_1[I_1] := A_2[I_2] + A_3[I_2]$;

S2: $A_4[I_2] := A_1[I_1] + A_4[I_2]$;

end

Then we have $R(1,1,2) = (0,\emptyset)$ or $\theta_2(1,2)$ holds and:

D_2:

Hence we get $Z_1 = \{1,2\}$ and

$P_1'^2$: (I_1,I_2)(S1: $A_1[I_1] := A_2[I_2] + A_3[I_2]$);

(I_1,I_2)(S2: $A_4[I_2] := A_1[I_1] + A_4[I_2]$).

This, however, does not give the same results as produced by the original P^2. Note that after the execution of the first loop $(I_1,I_2)(S1)$, the only outcome is $A[I_1]$ (i.e. $A[1]$) = $A_2[40] + A_3[40]$. However the second loop, $(I_1,I_2)(S2)$, requires forty different inputs, i.e. $A_2[1] + A_3[1],..,A_2[40] + A_3[40]$. Hence it becomes necessary to modify $P_1'^2$ as follows:

$P_1'^2$: $(I_1,I_2)(S1: \ A_1[I_1,I_2] := A_2[I_2] + A_3[I_2]);$

$(I_1,I_2)(S2: \ A_4[I_2] := A_1[I_1,I_2] + A_4[I_2]).$

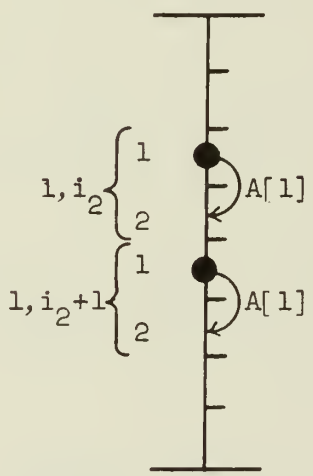


Figure 6.11. An Introduction of Temporary Locations

In general we apply the following transformation rule on a loop when it is separated. Assume that S_p and S_q are body statements in a loop P^m, and $\theta_u(p,q)$ holds. Further assume that p is separated from P^m with respect to

I_u (i.e. $p \epsilon Z_1, q \epsilon Z_2$). Now let us consider the vector $R''(u,k,p,q)$. Let the value of the first element which is neither 0 nor ϕ be e and its position be ℓ. Then we let

$$\overline{R}(u,k,p,q) = \{j \mid u \leq j < \ell \text{ and } R_j(k,p,q) = \phi\}.$$

We order elements of $\overline{R}(u,k,p,q)$ by their positions in $R''(u,k,p,q)$ and write

$$\overline{R}(u,k,p,q) = (\overline{r}(1),\overline{r}(2),\ldots,\overline{r}(t)).$$

Then we apply the following on the loop P^m.

Transformation T_2:

Transformation T_2 is defined for the cases $e < 0$ and $e > 0$ separately.

(1) $e > 0$.

Change $F(k,p,r(j))$ and $F(k,q,r(j))$ to $I_{r(j)}$ for $j = 1,2,\ldots,t$.

(2) $e < 0$.

(i) Change $F(k,p,r(j))$ to $I_{r(j)}$ for $j = 1,\ldots,t$.

(ii) Change $F(k,q,r(j))$ to the following ALGOL program for $j = 1,\ldots,t$. "if $(I_{r(j+1)} = 1)$ and $(I_{r(j+2)} = 1)$ and ... $(I_{r(t)} = 1)$ then (if $I_{r(j)} = 1$ then $|L_{r(j)}|$ else $I_{r(j)} - 1$) else $I_{r(j)}$." Also change $F(k,h^q,r(t))$ to the following ALGOL program: "if $(I_{r(t)} = 1)$ then $|L_{r(t)}|$ else $I_{r(t)} - 1$."

Example 7:

Let $R''(5,k,p,q) = (R_5(k,p,q),\ldots,R_9(k,p,q)) = (0,\phi,0,\phi,-1)$.

Then we get $e = -1$ and $\ell = 9$ and $\overline{R}(5,k,p,q) = (6,8)$. Also assume that $|L_6| =$

$|L_8| = 3$. Originally S_p and S_q may look like

S_p: $A_k[I_1, I_5, I_7, I_9] := f_p(..);$

S_q: $.. := f(..., A_k[I_1, I_5, I_7, I_9-1], ...);$

Now after Transformation T_2 is applied, S_p and S_q become:

S_p': $A_k[I_1, I_5, I_6, I_7, I_8, I_9] := f(..);$

S_q': $.. := f_q(..., A_k[I_1, I_5, B_6, I_7, B_8, I_9], ...),$

where

B_6 = *if* $I_8 = 1$ *then* (*if* $I_6 = 1$ *then* 3 *else* I_6-1) *else* I_6

and B_8 = *if* $I_8 = 1$ *then* 3 *else* $I_8 - 1$.

Note that by applying Transformation T_2, temporary locations are eventually introduced. For example in Example 7, A_k is changed to a seven-dimensional array from a four dimensional array by Transformation T_2.

6.3.4 Parallelism Between Body Statements

6.3.4.1 Introduction

Now we describe the parallelism between body statements. As stated before it becomes necessary to modify index expressions. In this section we give an algorithm which modifies index expressions properly.

We first describe the algorithm in terms of a restricted loop with only one loop index, i.e. $P^m = (I_1)(S_1;S_2;...;S_m)$. Accordingly every array identifier in P^m is of a form $A_k[F(k,h,1)] (= A_k[I_1+w(k,h,1)])$ where this is the h-th occurrence of A_k in P^m. For convenience we drop the subscript of

loop index. The primitive execution order for P^m becomes

$$E_0(P^m) = \{((i,p),V((i,p)|(|L|,m))|(1,1) \leq (i,p) \leq (|L|,m))\}.$$

For a given loop P^m, we consider the I-S plane which is an L by m grid. For example we have the following 40 x 3 grid for:

```
for I := 1 step 1 until 40 do
begin
S1: A1[I-1] := A2[I] + A3[I-1];
S2: A2[I] := A1[I+1] - A3[I];
S3: A3[I] := A1[I] + A2[I];
end.
```

On this grid, we only show the relation DR, e.g. $(A_3[i-1],(i-1,3),(i,1)) \in DR$.

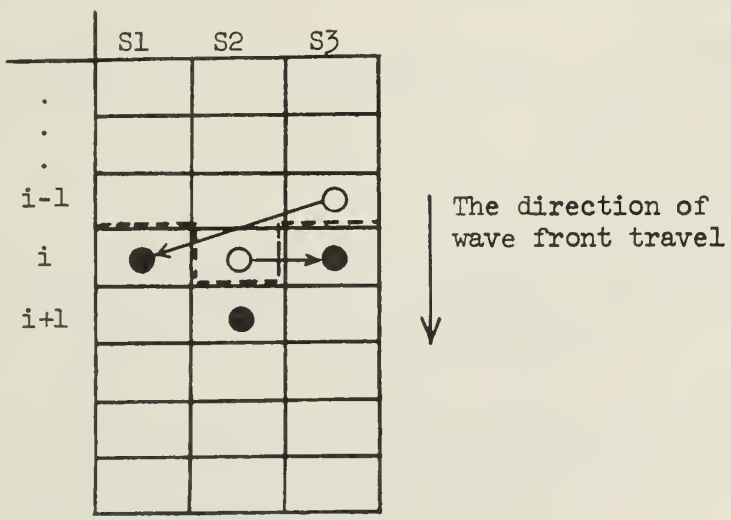


Figure 6.12. Wave Front for Simultaneous Execution of Body Statements

Then the objective of this section is to discover a wave front W (cf. Section 6.2.4) which separates all inputs from the computation, e.g. in Figure 6.12 inputs (shown by O) to S((i,1)),S((i+1,2)) and S((i,3)) (shown by •) lie above the wave front indicated by a dotted line. Hence S((i,1)),S((i+1,2)) and S((i,3)) can be computed simultaneously while I takes values 1,2,...,40 sequentially. In general to discover a wave front is equivalent to discovering a constant C(p) for each body statement S_p so that all statements S((i-C(1),1)), ...,S((i-C(p),p)),...,S((i-C(m),m)) can be computed simultaneously.

6.3.4.2 The Statement Dependence Graph and the Algorithm

Let us consider the I-S plane again and consider the computation S((i,p)). Assume that there is id such that

(id,(j,q),(i,p))ϵDR

where either (i) j = i and q < p, or (ii) j < i and p $\neq$ q, then clearly S((i,p)) and S((j,q)) cannot be computed simultaneously.

Definition 3:

The statement dependence graph (cf. the dependence graph in Section 6.3.3), $D(P^m)$, is defined by a set N of nodes 1,2,...,m each of which corresponds to a body statement of P^m and the arrow relation a. From node p to q there is an arrow a(p,q) if and only if either one of the following two conditions hold.

(1) For fixed i, there is k such that

$$A_k^h[F(k,h,1)(i)] \in OUT(S((i,p))),$$

$$A_k^g[F(k,g,1)(i)] \in IN(S((i,q))),$$

$$F(k,h,1)(i) = F(k,g,1)(i)$$

and $p < q$.

(2) For fixed i, there exist k and i' such that

$$A_k^h[F(k,h,1)(i')] \in OUT(S((i',p))),$$

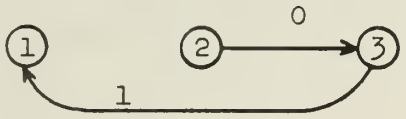
$$A_k^g[F(k,g,1)(i)] \in IN(S((i,q))),$$

$$F(k,h,1)(i') = F(k,g,1)(i)$$

and $i' < i$.

In the first case we label the arrow 0 and write $\ell(p,q) = 0$. In the second case the arrow is labeled 1 and we write $\ell(p,q) = 1$.

The statement dependence graph for the previous example is:



A chain of arrows, $a(p_1,p_2),a(p_2,p_3),\ldots,a(p_{k-1},p_k),a(p_k,p_1)$ in $D(P^m)$ is called a mesh M and we say e.g. $a(p_i,p_{i+1})$ is in M. If $\ell(p_i,p_{i+1}) = 0$ for some arrow in M, then M is called a part zero mesh. The following lemma is obtained immediately.

Lemma 2:

If $D(P^m)$ contains a part zero mesh, then there is no wave front for P^m.

Henceforth we assume that $D(P^m)$ has no part zero mesh. Given $D(P^m)$, we define a subset Z of N as follows:

$$Z = \{p \mid \text{there is no } q \text{ such that } \ell(p,q) = 0 \text{ or } \ell(q,p) = 0\}.$$

Z together with arrows gives a subgraph D_s of $D(P^m)$. Further we let

$Z_h = \{p \mid p \in Z$ and there is no q such that $\ell(q,p) = 0\}$.

Now we give an algorithm to find a wave front for $D(P^m)$.

Algorithm 1:

(1) Let $C(p) = +\infty$ for all $p \in N$.

(2) (i) Take any p from Z_h. If $Z_h = \emptyset$, then go to Step (5).

(ii) Let $C(p) = 0$.

(3) (i) If there are nodes s and t such that $a(s,t)$ exists, $\ell(s,t) = 1$ and $C(s) > C(t)$, then we let the value of $C(s)$ be equal to $C(t)$.

(ii) If there are nodes s and t such that $a(s,t)$ exists, $\ell(s,t) = 0$ and $C(s) \geq C(t)$, then let the value of $C(s)$ be equal to $C(t) - 1$.

Repeat (i) and (ii) until there are no s and t which satisfy (i) or (ii) in $D(P^m)$.

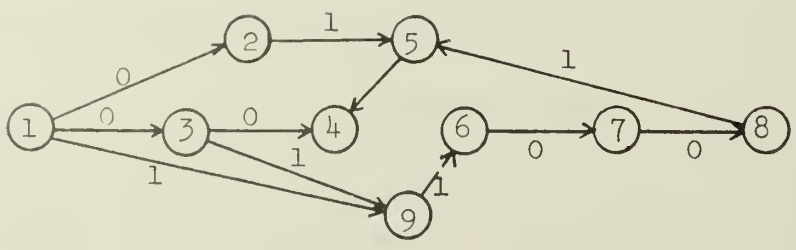
(4) (i) If for all p in Z $C(p) \neq +\infty$, then go to Step (5).

(ii) Otherwise take any p from Z for which there is q in Z such that $a(p,q)$ exists and $C(q) \neq +\infty$. Let $M = \max\{C(s)\}$ where $s \in Z$ and $C(s) \neq +\infty$. Then let $C(p) = \max\{C(q) + 1, M\}$. Go to Step (3).

(5) For all p in Z with $C(p) = +\infty$, let $C(p) = M$ where $M = \max_{s \in Z}\{C(s)\}$ and $c(s) \neq +\infty$. If $Z_h = \emptyset$, then let $C(p) = 0$ for all p in Z.

Example 8:

(1) D:

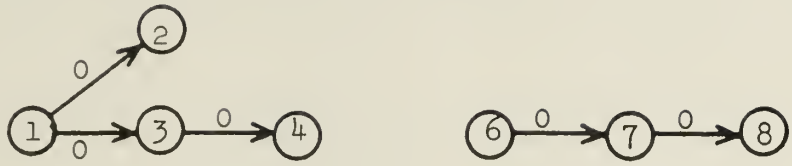


(2) $Z = \{1,2,3,4,6,7,8\}$,

$Z_h = \{1,6\}$

and

D_s:



(3) Let $C(1) = 0$ and apply Step (3) of Algorithm 1. Then we get $C(1) = 0$ since there is no q such that $a(q,1)$ exists.

(4) Let $C(2) = 1$.

(5) Let $C(3) = 1$.

(6) Let $C(4) = 2$. Then we apply Step (3) of Algorithm 1 on $a(5,4), a(8,5), a(5,2), a(2,1), a(7,8), a(6,7), a(9,6), a(3,9), a(3,1)$ and $a(9,1)$. And we get $C(5) = C(8) = 2$. $C(2) = 1$. $C(7) = 1$, $C(6) = 0$, $C(9) = 0$, $C(3) = 0$, and $C(1) = -1$.

(7) There is no p in Z with $C(p) = +\infty$. Hence Algorithm 1 terminates and we get

$C(1) = -1$, $C(2) = 1$, $C(3) = 0$, $C(4) = 2$, $C(5) = 2$, $C(6) = 0$, $C(7) = 1$, $C(8) = 2$ and $C(9) = 0$.

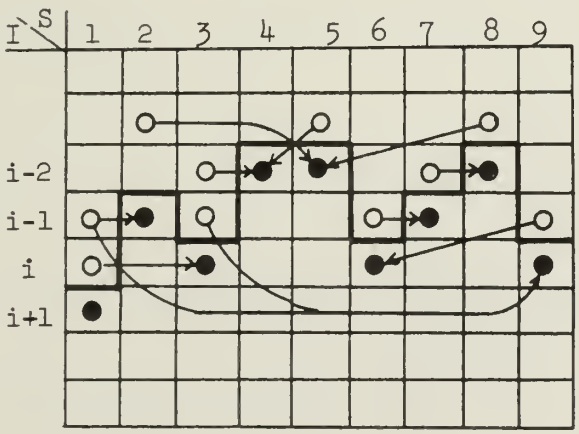


Figure 6.13. A Wave Front for Example 10

Now we show that Algorithm 1 gives a valid wave front. To prove this first we show that Algorithm 1 is effective, i.e. every step of Algorithm 1 is always applicable and terminates.

Lemma 3:

Algorithm 1 is effective.

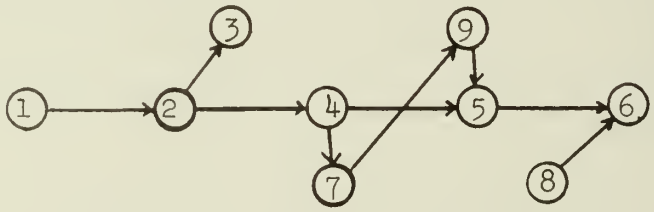
Proof:

That Step (1),(2),(4) and (5) are effective is clear. Now we show that Step (3) is effective. First we define U(p) to be a set of nodes such that

$$U(p) = \{q \mid \text{there is a chain of arrows } a(p_1,p_2), a(p_2,p_3), \ldots,$$

$$a(p_{n-1},p_n) \text{ exist where } p_1 = q \text{ and } p_n = p\} \cup \{p\}, \text{ e.g.}$$

$D(P^8)$:



and $U(5) = \{1,2,4,5,7,9\}$. $U(p) = U(q)$ implies that there is a mesh which includes p and q. By assumption this mesh is not a part zero mesh and C(q) will be assigned the same value as C(p) in a finite number of steps after C(p) has been assigned a value.

If $U(p) \supset U(q)$, then C(q) will be assigned a value less than or equal to C(p) in a finite number of steps after C(p) has been assigned a value.

Thus after a finite number of applications Step (3) eventually terminates. Hence Algorithm 1 is effective.

(Q.E.D.)

Theorem 6:

Algorithm 1 gives a valid wave front.

Proof:

To prove this, it is enough to show that (i) if $\ell(p,q) = 0$, then $C(p) < C(q)$ and (ii) if $\ell(p,q) = 1$, then $C(p) \leq C(q)$. However, from Steps (3) and (4) of Algorithm 1, clearly the above conditions hold. Also if p is assigned a value C(p) by Step (5) it implies that either (i) there is $r \in U(q)$ where $q \in Z_h$ and $\ell(r,q) = 1$ or (ii) there is no such r. In the second case C(p) may take any value (S_p may be computed at any time), and in the first case $C(q) \geq C(r)$ must hold. Hence we let $C(q) = \max\{C(s)\}$.

(Q.E.D.)

To handle a restricted loop with more than one loop indicies, we modify Definition 3 as follows. For each S_p and S_q in P^m we first obtain a vector R(k,p,q).

Definition 3':

The statement dependence graph of P^m, $D(P^m)$, is defined by a set N of nodes 1,2,...,m each of which corresponds to a body statement of P^m and the arrow relation a. There is an arrow a(p,q) if and only if either one of the followings holds:

(1) $R_j(k,p,q) = 0$ or $\emptyset$ for all $j = 1,2,..,n$ and $p < q$. We let $\ell(p,q) = 0$.

(2) $V((R_1(k,p,q),\ldots,R_n(k,p,q)|B) \geq V((0,\ldots,0)|B)$ where $B = (|L_1|,\ldots,|L_n|)$.

We let $\ell(p,q) = 1$.

From Definition 3' clearly

(1) $\ell(p,q) = 0$ if $S_q((i_1,i_2,\ldots,i_n))$ uses the output of $S_p((i_1,i_2,\ldots,i_n))$ and $p < q$.

(2) $\ell(p,q) = 1$ if $S_q((i_1,i_2,\ldots,i_n))$ uses the output of $S_p((i_1',i_2',\ldots,i_n'))$ where $(i_1',i_2',\ldots,i_n') \leq (i_1,i_2',\ldots,i_n)$.

Algorithm 1 is then applied on $D(P^m)$. For example let P^3 be

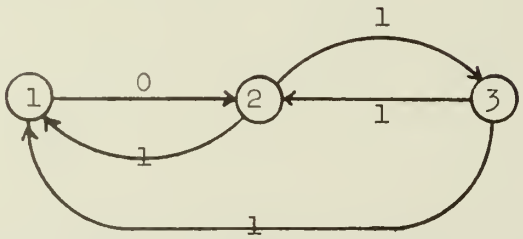
S1: $A_1[I_2,I_2] := A_2[I_2] + A_3[I_2-1,I_3+1]$;

S2: $A_2[I_2+1] := A_1[I_1,I_2] + A_3[I_2,I_3]$;

S3: $A_3[I_2,I_3] := A_2[I_2-1]$;

Then we have

$D(P^3)$:



6.3.5 Discussion

Given a loop $P^m = (I_1,\ldots,I_n)(S_1;\ldots;S_m)$, we first try to execute body statements in parallel with respect to some loop index. If this fails for any loop index, or if this does not give a satisfactory result, then we try to replace the loop with many for statements. Then we can attempt to execute a body statement (or body statements) of a resultant for statement in parallel with respect to some loop index. If this fails, then we may try the third

approach, i.e. we try to execute all body statements simultaneously while loop indices vary sequentially. Often the number of loop indicies, n, is very small (typically n = 2), and it will be easy to try all variations.

7. EQUALLY WEIGHTED--TWO PROCESSOR SCHEDULING PROBLEM

7.1 Introduction

This chapter gives a solution to the so-called equally weighted—two processor scheduling problem. Informally the problem may be stated as follows. Given a set of tasks along with a set of operational precedence relationships that exist between certain of these tasks, and given two identical processors (PE), P(2), how does one schedule these tasks on the two processors so that they execute in the minimum time? It is assumed that either one of two processors is capable of processing any task in the same amount of time, say 1 unit of time. Informally a set of tasks together with procedence relations forms a graph.

Clearly the problem of scheduling any given equally weighted task graph on k identical processors, P(k), in an optimal way is effectively solvable by exhaustion. But this is far from possible in practice. The only practical solution so far obtained is a result for scheduling a rooted tree (a restricted class of graphs) with equally weighted tasks on k identical processors, P(k) [21].

Now let us study how the equally weighted—two processor scheduling problem is related to the computation of arithmetic expressions on a parallel machine.

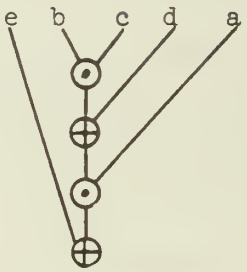
In Chapter 3, the parallel computation of an arithmetic expression by building a syntactic tree was studied. There we were only concerned with the height of a tree and reducing it by distribution, and we did not introduce any

physical restrictions of a machine. For example, in reality, the size of a machine, i.e. the number of PE's is limited rather than arbitrarily big. One problem which will arise immediately is whether the distribution algorithm should be applied or not to reduce tree height since distribution introduces additional operations. For example assume that we have a two PE machine, P(2). Now let us consider two arithmetic expressions, A = a(bc+d) + e and B = abc(defgh+i). Then we have h[A] = 4, h[B] = 5, $h[A^d]$ = h[abc+ad+e] = 3 and $h[B^d]$ = h[abcdefgh+abci] = 4. Thus distribution reduces the height of T[A] and T[B]. On the other hand Figure 7.1 shows that if A,B,A^d and B^d are computed on P(2), A^d is still computed in less time than A while B^d now takes more time than B.

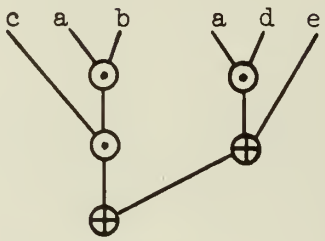
Assume that we get A^d from A by the distribution algorithm. If the size of a machine is limited, then it may not necessarily be true that A^d can be computed in less time than A even if $h[A^d] < h[A]$ holds. Actually it is a nontrivial problem to decide whether distribution is to be made or not to reduce computation time (which is different from tree height) if the size of a machine is limited. It depends on the form of an arithmetic expression as well as the machine organization. We will not go into this problem any further.

Now let us look at the situation from a different point of view. Given an arithmetic expression A and its minimum height tree, it is possible to take advantage of common expressions to reduce the number of operations to be performed in hopes of reducing computation time. For example let us consider the computation of A = (a+b+c+d)ef + (a+b)g on P(2). If we evaluate

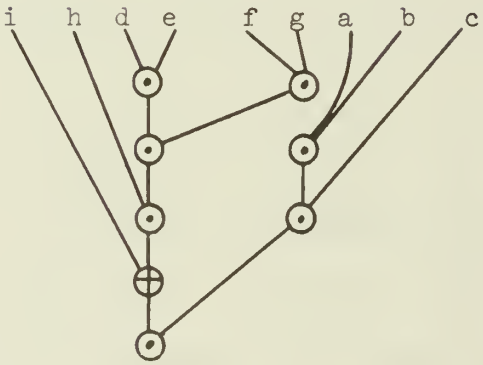
(a+b) only once then A can be computed in 4 steps on P(2) while if (a+b) is evaluated twice, then it takes 5 steps to compute A (see Figure 7.2).



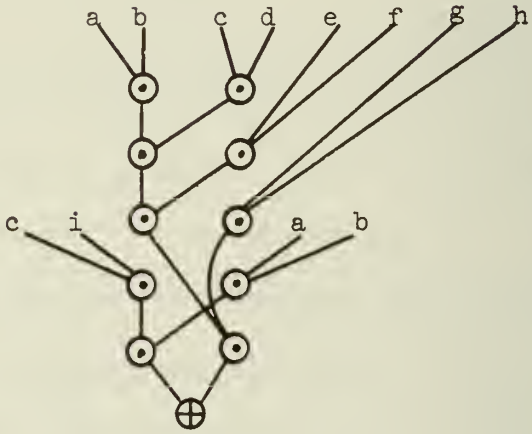
(a) A (4 steps)



(b) A^d (3 steps)



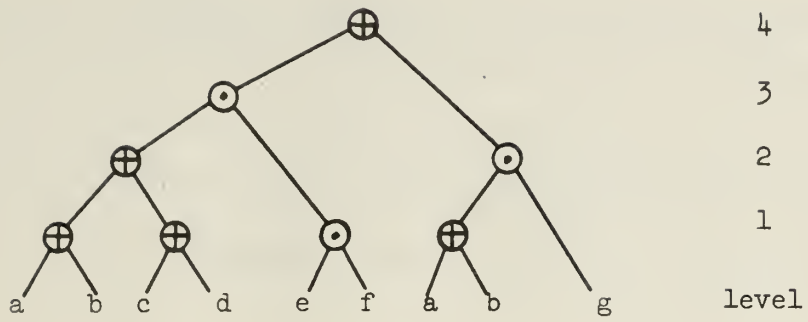
(c) B (5 steps)



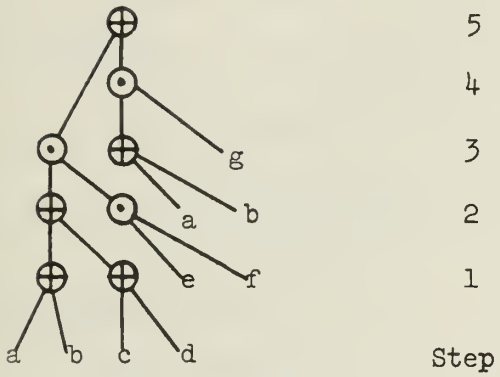
(d) B^d (6 steps)

Figure 7.1. Computation of Nondistributed and Distributed Arithmetic Expressions on P(2)

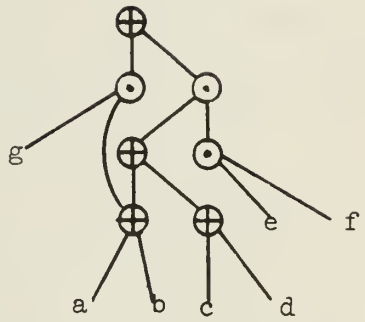
Our main concern in Chapter 2 was to reduce tree height assuming that the size of a machine is unlimited. Hence we were not interested in reducing the number of operations. As mentioned there, it was an open problem to find out common expressions while keeping the height of a tree minimum. However, if we could take advantage of common expressions while



(a) A Minimum Height Tree for A



(b) (a+b) computed twice



(c) (a+b) computed once

Figure 7.2. Common Expression

keeping the height of a tree minimum, then we would obtain a graph of operations rather than a tree for an arithmetic expression (see Figure 7.2. (c)).

While we do not know how to compute an arithmetic expression A on P(2) in the minimum time (e.g. should distribution be done?), the scheduling algorithm presented in this chapter schedules a <u>given</u> graph of operations for an arithmetic expression on P(2) so that the <u>given</u> graph is processed in the

minimum time, assuming that each PE of P(2) may perform addition or multiplication independently but in the same amount of time, say 1 unit of time. Note that we may be able to construct many graphs for A. Hence while the scheduling algorithm schedules a given graph for A on P(2) in an optimal way, the algorithm does not necessarily compute A itself in the minimum amount of time.

7.2 Job Graph

Let G be an acyclic graph with nodes N_i $(i=1,2,\ldots,n)$ and a set of directed arrows connecting pairs of nodes. For nodes N and N' we write $N \to N'$ if there is an arrow from N to N'. We say that N is an immediate predecessor of N' and N' is an immediate successor of N. Also we let

$SR(N) = \{N' | N \to N'\}$ (a set of successors of N)

and $PR(N) = \{N' | N' \to N\}$ (a set of predecessors of N).

Nodes which have no incoming arrows are called initial nodes, and nodes which have no outgoing arrows are called terminal nodes. For the sake of simplicity we assume that a graph has one initial node and one terminal node. If there are more than two, then we can add a dummy initial/terminal node. We write N_I and N_T for them, respectively. We also write $N \Rightarrow N'$ if there is a chain $N_1, N_2, \ldots, N_m$ such that $N \to N_1 \to \ldots \to N_m \to N'$, or $N \to N'$. Furthermore we write $N \not\to N'$ or $N \not\Rightarrow N'$ to show that the relation $N \to N'$ or $N \Rightarrow N'$ does not hold.

Definition 1:

The forward distance (or level) from the initial node to a node N, $d_I(N)$, is the length of the longest path from the initial node to N, thus

$d_I(N_I) = 0$. Similarly the backward distance from the terminal node to N, $d_T(N)$, is defined, thus $d_T(N_T) = 0$.

Thus a node N cannot be initiated before time $d_I(N)$ but may be initiated at $d_I(N)$ or at any time after that.

Definition 2:

The height of a graph G, h(G), is defined as

$$h(G) = d_I(N_T).$$

Then we say that a graph G is tight if for all nodes N,

$$d_I(N) + d_T(N) = h(G).$$

Otherwise we say that a graph G is loose.

Example 1:

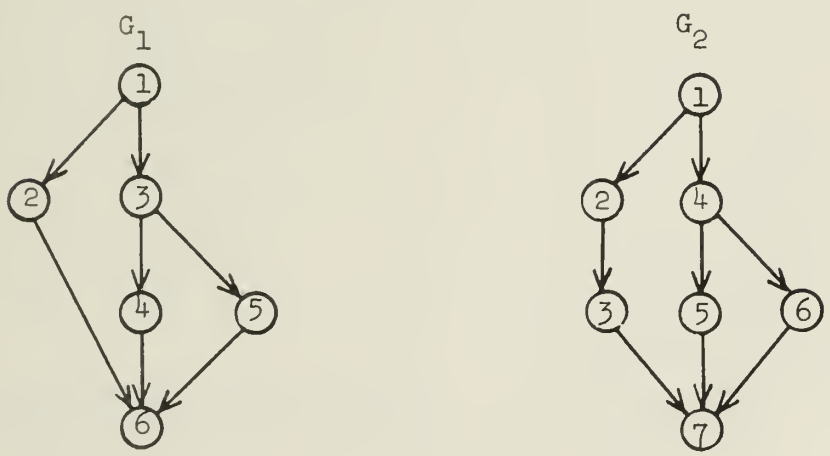


Figure 7.3. A Loose Graph and a Tight Graph

The graph G_1 is a loose graph because $d_I(N_2) + d_T(N_2) = 2 \neq h(G_1)$, whereas the graph G_2 is a tight graph.

First we shall study an optimum scheduling for a tight graph. A scheduling for a loose graph will be discussed in a latter section. In what follows, we use words "process" and "schedule" interchangably.

Definition 3:

Let A(i) be the set of all nodes of forward distance i, i.e.

$A(i) = \{N | d_I(N) = i\}$.

This is called a node set.

All nodes in A(i) can be scheduled independently of each other since there can be no precendence relations between nodes in A(i). In other words, if $N \Rightarrow N'$ then N and N' cannot be processed simultaneously.

Now we have the following lemma which characterizes a tight graph.

Lemma 1:

If a graph G is tight, then for every node N, there exists $N' \in SR(N)$ and $N'' \in PR(N)$ such that $d_I(N') = d_I(N) + 1$ and $d_I(N'') = d_I(N) - 1$. (For the terminal node $SR(N_T) = \emptyset$ and for the initial node $PR(N_I) = \emptyset$. Those are exceptions.)

Proof:

Obvious by Definition 2.

(Q.E.D.)

Corollary 1:

Let G be a tight graph. Let N be a node of G. Let $N \in A(t)$. Then for every i, $0 \leq i \leq t - 1$, there is at least one node $N' \in A(i)$ such that $N' \Rightarrow N$. Also for every j, $t + 1 \leq j \leq h(G)$, there is at least one node $N'' \in A(j)$ such that $N \Rightarrow N''$.

Definition 4:

To p-schedule a set Q of nodes is to partition Q into subsets of size 2 in an arbitrary way (if |Q| is odd, there will be one subset of size 1) and to order those subsets in an arbitrary way.

A node N is said to be available if all predecessors of N have been processed.

7.3 Scheduling of a Tight Graph

Having these definitions, now we discuss a scheduling of a tight graph G on two processors. The idea of this scheduling scheme is rather simple. We start checking |A(i)| from i = 1 to h(G). For each i, if |A(i)| is even, then we p-schedule A(i), and no processor time will be wasted. If |A(i)| is odd, and if we simply p-schedule A(i), then there will be one node, N, left which cannot be processed in parallel with another node in A(i). Thus we will waste processor time. Therefore a node which can be processed in parallel with the above left out node N must be found. Where can that node be found? It will be shown that we have to look as far as the smallest i' larger than i with |A(i')| = odd to find it. Thus the amount of work to look ahead is always bounded.

Before we go further, a few more definitions are in order.

For some t and n, let us consider a set $A_n^t = \bigcup_{i=0}^{n} A(t+i)$. Now let us take a node from each of A(t+j) and A(t+i) (j < i). Let them be N^j and N^i. If $N^j \not\triangleright N^i$, then N^j and N^i may be processed simultaneously, providing that all predecessors of N^j and N^i have been processed.

Now we establish this relation formally on A_n^t.

Definition 5:

The p-line relation ($\underline{p}$) between two nodes N and N' in A_n^t is defined as follows.

$N \underline{p} N'$ if and only if

(1) $N \not\geq N'$ and $N' \not\geq N$

and (2) $d_I(N) \neq d_I(N')$.

We also write (N,N') for $N \underline{p} N'$. A pair (N,N') is called a p-line pair. Note that in general (N,N') and (N',N") do not necessarily imply (N,N").

Further we define $A_n^t(p) = \{(N,N') \mid N \in A(t+i), N' \in A(t+j), 0 \leq i,j \leq n\}$, i.e. $A_n^t(p)$ is a set of all pairs of nodes in A_n^t between which the p-line relation holds.

Since $(N,N') \in A_n^t(p)$ implies that $(N',N) \in A_n^t(p)$, we shall in general put only one of them in $A_n^t(p)$ and drop the other.

An algorithm to find $A_n^t(p)$ is given in Section 7.5.

Definition 6:

A p-line set L_n^t on A_n^t is an ordered set of p-line pairs

$$L_n^t = ((N_0,N_1'),(N_1,N_2'),\ldots,(N_k,N'_{k+1}))$$

where

(1) $N_0 \in A(t)$ and $N'_{k+1} \in A(t+n)$,

and (2) for all $g(1 \leq g \leq k)$, $d_I(N_g') = d_I(N_g)$ but $N_g' \neq N_g$.

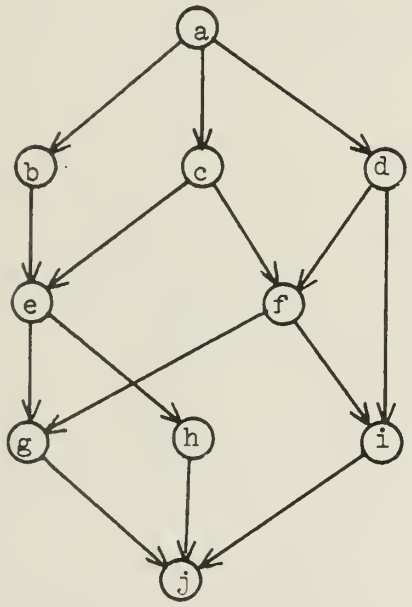
We say that A_n^t is p-connectable if there is a p-line set L_n^t on A_n^t.

Also we write $L_n^t(N_0, N'_{k+1})$ when the first and the last nodes in L_n^t are of special interest.

Further, we write $L_n^t(N_0, N'_{k+1}) = L_i^t(N_0, N_g') \cup L_{n-i}^{t+i}(N_g, N'_{k+1})$ if (N_{g-1}, N'_g) and (N_g, N'_{g+1}) are two adjacent elements of $L_n^t(N_0, N'_{k+1})$ and $d_I(N'_g) = d_I(N_g) = t + i$.

An algorithm to build a p-line set for A_n^t is given in Section 7.5.

Example 2:



$A(1) = \{b, c, d\}$

$A(2) = \{e, f\}$

$A(3) = \{g, h, i\}$

Figure 7.4. A Graph G

$A_2^1 = \bigcup_{i=0}^{2} A(1+i) = \{b, c, d, e, f, g, h, i\}$. $A_2^1(p) = \{(b, f), (d, e), (f, h), (e, i)\}$. A

typical $L_2^1 = ((b,f),(e,i))$. Hence A_2^1 is p-connectable.

Further we define a special p-line set called a p-line (1) set.

Definition 7:

Given a set A_n^t. We call a p-line set

$$L_n^t = ((N_0,N_1'),(N_1,N_2'),\ldots,(N_k,N_{k+1}'))$$

a p-line (1) set (write $L_n^t(1)$) if

$$d_I(N_i) + 1 = d_I(N_{i+1}') \text{ for all } i(0 \leq i \leq k).$$

Note that in this case $k = n - 1$.

Also we write $L_n^t(1)(N_0,N_{k+1}')$ when the first and last nodes are of particular interest.

Now a few lemmas are in order.

Lemma 2:

Suppose $N \in A(t)$ and $N' \in A(t+n)$ for some t and n in graph G. Assume that (N,N') holds. Then there is a p-line (1) set

$$L_n^t(1)(N,N') = ((N_0,N_1'),(N_1,N_2'),\ldots,(N_{n-1},N_n'))$$

where $N_0 = N$ and $N_n' = N'$.

Proof:

A proof is given by an induction on n.

First note that $|A(t+i)| \geq 2$ for all i, $1 \leq i \leq n - 1$. Otherwise $N \Rightarrow N'$ holds and (N,N') does not.

(1) Now let n = 2. Then there must be two distinct nodes $N_1, N_2 \in A(t+1)$ such that $N \rightarrow N_1$ and $N_2 \rightarrow N'$. Otherwise the graph G is not tight. Hence (N_1, N') and (N, N_2). Thus $L_2^t(1) = ((N, N_2), (N_1, N'))$.

(2) Now assume that the lemma holds for $n \leq i$.

Let $n = i + 1$. Let $N \in A(t)$, $N' \in A(t+i+1)$, and (N, N'). Then there must be two distinct nodes $N_1, N_2 \in A(t+i)$ such that (N, N_1) and $N \Rightarrow N_2$. This follows from Lemma 1 and Corollary 1. Then (N_2, N'), since otherwise $N \Rightarrow N'$ holds and this contradicts the assumption. By the induction hypothesis, there is a

$$L_{n-1}^t(1)(N, N_1) = ((N, N^{1\prime}), (N^1, N^{2\prime}), \ldots, (N^{n-2}, N_1)).$$

Thus there is a p-line (1) set

$$L_n^t(1)(N, N') = L_{n-1}^t(1)(N, N_1) \cup (N_2, N') = ((N, N^{1\prime}), (N^1, N^{2\prime}), \ldots, (N^{n-2}, N_1), (N_2, N')).$$

(Q.E.D.)

Lemma 3:

Suppose that $N \in A(t), N^1, N^2 \in A(t+i), N^3, N^4 \in A(t+j)$, and $N' \in A(t+n)$ where $i < j < n$. Also assume that $(N, N^3), (N^4, N')$, and (N^2, N') hold.

Then there is a p-line (1) set $L_n^t(1)(N, N') = ((N, N_1'), \ldots, (N_{n-1}, N'))$.

Proof:

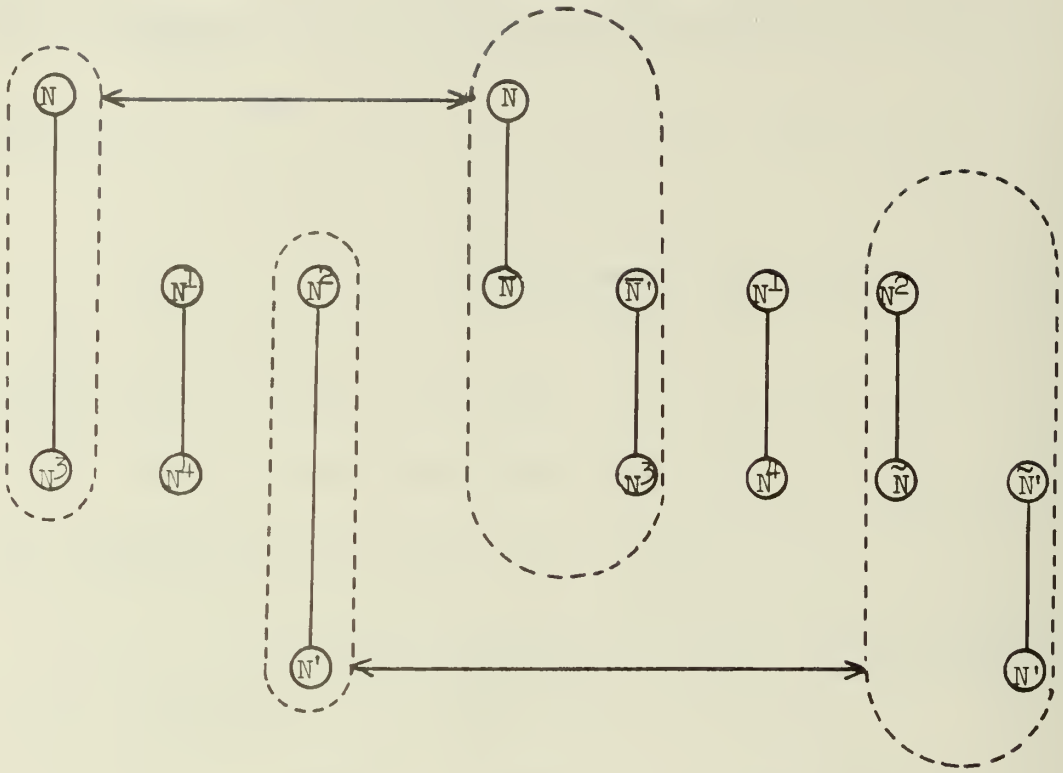


Figure 7.5. An Illustration for Lemma 3

Since (N,N^3) by the previous lemma, there is a p-line (1) set

$$L_j^t(1)(N,N^3) = L_i^t(1)(N,\bar{N}) \cup L_{j-1}^{t+i}(1)(\bar{N}',N^3),$$

and since (N^2,N'), there is a p-line (1) set

$$L_{n-i}^{t+i}(1)(N^2,N') = L_{j-i}^{t+i}(1)(N^2,N) \cup L_{n-j}^{t+j}(1)(\tilde{N}',N').$$

By definition, $\bar{N} \neq \bar{N}'$, $\tilde{N} \neq \tilde{N}'$, $N^3 \neq N^4$ and $N^1 \neq N^2$.

Now we have two cases.

(1) $N^3 = \tilde{N}'$

Then $N^4 \neq \tilde{N}'$. Now let

$$L_n^t(1)(N,N') = L_i^t(1)(N,\bar{N}) \cup L_{j-i}^{t+i}(1)(N^1,N^4) \cup L_{n-j}^{t+g}(1)(\tilde{N}',N').$$

(2) $N^3 \neq \tilde{N}'$.

Then let

$$L_n^t(1)(N,N') = L_j^t(1)(N,N^3) \cup L_{n-j}^{t+j}(1)(\tilde{N}',N').$$

Thus in either case there is a p-line (1) set on A_n^t.

(Q.E.D.)

From Lemmas 2 and 3, we can prove the following lemma.

Lemma 4:

If A_n^t is p-connectable, then there is a p-line (1) set $L_n^t(1)(N,N')$ where $N \in A(t)$ and $N' \in A(t+n)$.

What Lemma 4 implies is the following.

Let $L_n^t(1)(N,N') = ((N,N_1'),(N_1,N_2'),(N_2,N_3'),\ldots,(N_{n-1},N'))$, i.e. $d_I(N_i') = d_I(N_i) = t + i$ and $d_I(N_i) + 1 = d_I(N_{i+1}')$.

Now assume that a set A_n^t is p-connectable. Then we have $L_n^t(1)(N,N') = ((N,N_1'),(N_1,N_2'),\ldots,(N_{n-1},N'))$ where $N \in A(t)$ and $N' \in A(t+n)$. Since (N_i,N_{i+1}'), we can process both at the same time. To do this we first process $A(t+i) - \{N_i\}$

(notice that $d_I(N_i) = t+i$). Then process $\{N_i, N'_{i+1}\}$. Finally process $A(t+i+1) - \{N'_{i+1}\}$. This leads us to the following scheduling.

Definition 8:

Assume A_n^t is p-connectable. Then by to p-line schedule A_n^t by $L_n^t(1)$, we mean the following scheduling.

Let $L_n^t(1) = ((N_0, N_1'), (N_1, N_2'), \ldots, (N_{n-1}, N_n'))$.

(1) p-schedule $A(t) - \{N_0\}$.

(2) $g = 1$

(3) p-schedule $\{N_{g-1}, N_g'\}$

(4) p-schedule $A(t+g) - \{N_g', N_g\}$.

(5) $g = g + 1$. If $g < n$, then go to (3).

(6) p-schedule $\{N_{n-1}, N_n'\}$.

(7) p-schedule $A(t+k) - \{N_n'\}$.

Now an algorithm to schedule a tight graph for two processors is described.

Scheduling is done according to node sets A(i), for $i = 1,2,\ldots,h(G)$.

All nodes in A(i) can be processed independently of each other, i.e., in any order.

If $|A(i)|$ is even, then two processors can be kept busy all the time to process A(i) and A(i) can be processed in time $|A(i)|/2$.

If $|A(i)|$ is odd, then there is a possibility that a machine becomes idle, i.e., one node will be left out from A(i) which does not have any partner

to be processed with. Let it be N. Then a partner must be found from some A(j), j > i. First we may try to find N' $\in$ A(i+1) which can be a partner of N. However if |A(i+1)| is even, then |A(i+1) - {N'}| becomes odd and we have the same problem again, and may try to find a node from A(i+2) to fill an idle machine, and so on. If this cycle is ever to stop, it must stop when A(i+k) is hit where |A(i+k)| is odd. Otherwise there is no way to remedy the cycle, and machine time must be wasted.

Tight Graph Scheduling Algorithm:

Step 1: t = 0

Step 2: If t = h(G) then p-schedule A(t) and stop, else go to Step 3.

Step 3: If |A(t)| is even, then

(3-1) p-schedule A(t) and

(3-2) go to Step 7.

Step 4: |A(t)| is odd.

Find A(t+1).

Step 5: If $\forall N \in A(t)$ |SR(N)| = |A(t+1)|, then

(5-1) p-schedule A(t) and

(5-2) go to Step 7.

Step 6: There is N' $\in$ A(t) such that |SR(N')| < |A(t+1)|.

(6-1) If |A(t+1)| is odd, then

(6-1-1) p-schedule A(t) - {N'}.

(6-1-2) p-schedule {N'} $\cup$ {N''} where N'' $\in$ A(t+1) - SR(N').

(6-1-3) p-schedule A(t+1) - {N''}.

(6-1-4) go to Step 7.

(6-2) $|A(t+1)|$ is even.

(6-2-1) Find out the smallest k greater than 1 such that $|A(t+k)|$ is odd.

(6-2-2) If there is no such k (i.e., we have checked up to A(h(G))) then p-schedule A(i) $t \leq i \leq h(G)$ individually, and stop.

(6-2-3) Else we have a set $\mathcal{A} = \{A(t), A(t+1), \ldots, A(t+k)\}$ where $|A(t)|$ and $|A(t+k)|$ are odd and other $|A(t+i)|$ are all even. Check p-connectivity of A_k^t.

(1) A_k^t is not p-connectable.

(1-i) p-schedule $A(t), A(t+1), \ldots, A(t+k-1)$ individually.

(1-ii) Let $t = t + k - 1$.

(1-iii) go to Step 7.

(2) A_k^t is p-connectable.

(2-i) Find a p-line (1) set $L_k^t(1) = ((N_0, N_1'), (N_1, N_2'), \ldots, (N_{k+1}, N_k'))$.

(2-ii) p-line schedule A_k^t by $L_k^t(1)$.

(2-iii) $t = t + k$.

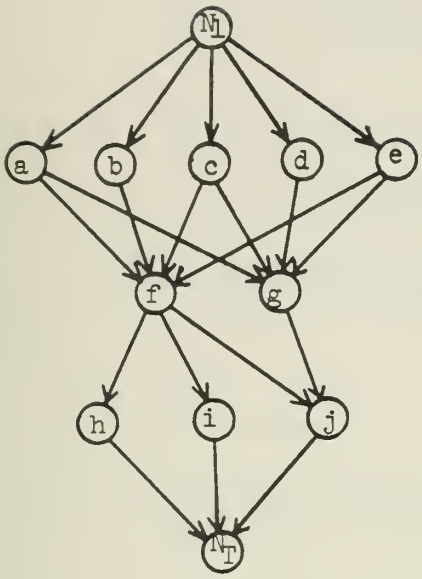
(2-iv) go to Step 7.

Step 7: $t = t + 1$.

Go to Step 2.

Example 3:

(1) $|A(1)|$ and $|A(3)|$ are odds, and $|A(2)|$ is even. Thus we have $\mathcal{A} = \{A(1), A(2), A(3)\}$. (by Step 6 of Algorithm)



A(1) = {a,b,c,d,e}

A(2) = {f,g}

A(3) = {h,i,j}.

Figure 7.6. An Example of a Tight Graph Scheduling

(2) For A_2^1, we have $L_2^1(1) = ((d,f),(g,h))$.

(3) According to Step (6-2-3)(2), we schedule as follows.

(i) p-schedule A(1) - {d} = {a,b,c,e}.

(ii) p-schedule {d,f}.

(iii) p-schedule A(2) - {f,g} = $\emptyset$.

(iv) p-schedule {g,h}.

(v) p-schedule A(3) - {h} = {i,j}.

Thus we have an optimum schedule:

Step	1	2	3	4	5
machine A	a	c	d	g	i
B	b	e	f	h	j

Proof for the algorithm:

Lemma 5:

Step 3 is optimum and whatever p-schedule is made, it does not affect the later stages.

Lemma 6:

Step 5 is optimum and whatever p-schedule is made, it does not affect the later stages.

Proof:

First note that after A(t) has been processed, nodes in A(t+1) only can become available. Since $\forall N \in A(t), |S(N)| = |A(t+1)|$, all nodes in A(t) must have been processed before any node in A(t+1) can be processed.

(Q.E.D.)

Lemma 7:

Step 6-1 is optimum and whatever p-schedule is made, it does not effect the later stages.

Proof:

The algorithm actually processes A(t) and A(t+1) (where $|A(t)|$ and $|A(t+1)|$ are odd) in time $(|A(t)| + |A(t+1)|)/2$, which is optimum.

(Q.E.D.)

Lemma 8:

Step 6-2-2 is optimum.

Proof:

Let $\mathcal{A} = \{A(t), A(t+1), \ldots, A(h(G))\}$. Since $|A(t)|$ is odd and all other $|A(i)|$ is even $(t < i \leq h(G))$, it takes at least time

$$\left\lceil \frac{|A(t)|+|A(t+1)|+\ldots+|A(h(G))|}{2} \right\rceil$$

to process $\mathcal{A}$.

Step 6-2-2 achieves this.

(Q.E.D.)

Lemma 9:

Assume $|A(t)|$ and $|A(t+k)|$ are odd and $|A(t+i)|$ are all even $(1 \leq i < k)$. If A_k^t is p-connectable, then Step 6-2-3 (2) is optimum.

Proof:

$\mathcal{A} = \{A(t), A(t+1), \ldots, A(t+k)\}$ can be processed in time

$$\frac{|A(t)|+|A(t+1)|+\ldots+|A(t+k)|}{2}.$$

Step 6-2-3 (2) achieves this.

(Q.E.D.)

Lemma 10:

Assume $|A(t)|$ and $|A(t+k)|$ are odd and $|A(t+i)|$ are all even $(1 \leq i < k)$. If A_k^t is not p-connectable, then Step 6-2-3 (1) is optimum.

Proof :

Let $Q = \{A(t), A(t+1), \ldots, A(t+k)\}$. Since A_k^t is not p-connectable, there is no p-line set $L_k^t(1)$. Thus there will be N in some A(t+i) $(0 \leq i \leq k)$ which does not have any partner to be processed with it, thus making a machine idle. Now if this situation could be remedied, then it would be only because there is $N' \in A(t+n+j)$ $(j > 0)$ which can be done in parallel with N.

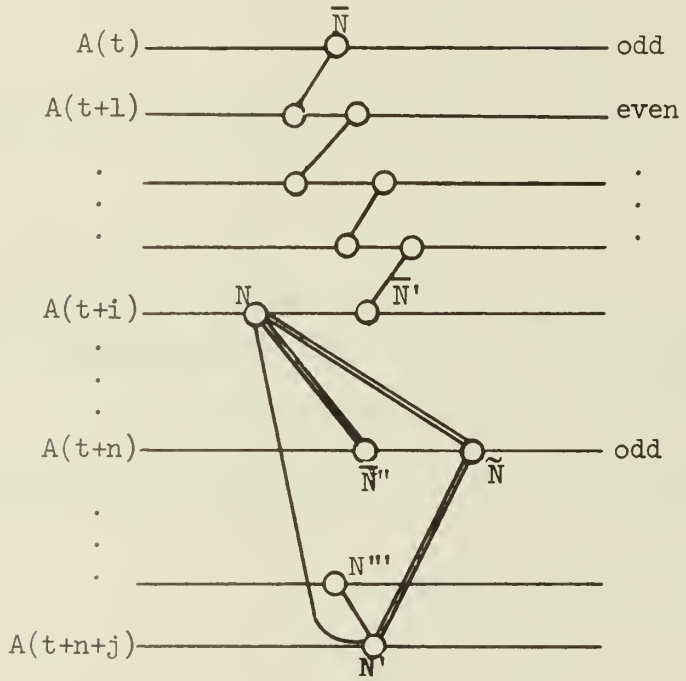


Figure 7.7. An Illustration for Lemma 11

Suppose that the above could be done. The parallel processing of N and N' can, however, be advantageous only if A_i^t is p-connected and there is a p-line (1) set $L_i^t(1)(\bar{N}, \bar{N}')$ where $\bar{N} \in A(t), \bar{N}' \in A(t+i)$ and $N' \neq N$. Otherwise

processors cannot be kept busy for $A(t), A(t+1), \ldots, A(t+i)-\{N\}$ and we do not gain at all by delaying a processing of N. Now from the assumption, $N \not\Rightarrow N''$ for all $N'' \in A(t+n)$ because otherwise A_n^t becomes p-connectable. Also by Corollary 1, there is a node $\tilde{N}$ in $A(t+n)$ such that $\tilde{N} \not\Rightarrow N'$. This implies that $N \not\Rightarrow N'$. Thus N cannot be processed in parallel with N. This proves the lemma.

(Q.E.D.)

Essentially what the above algorithm does is upon finding A(t) where $|A(t)|$ is odd to try to delay the processing of a node N in A(t) till the next node set A(t+1). If $|A(t+1)|$ is even, then again the algorithm tries to delay the processing of a node N' in A(t+1) till the next level A(t+2), etc. until it finds some A(t+k) where $|A(t+k)|$ is odd, or $|A(t+k) \cup \{N''\}|$ is even where N'' is a node whose processing is being delayed from the previous stage. In other words, the algorithm tries to establish p-connectability between two adjacent node sets both of which have an odd number of elements. Note that it is not necessary to try to establish p-connectability among more than two odd node sets, say $A(t), A(t+k)$ and $A(t+m)(m > k)$, because A_m^t cannot be p-connectable if A_k^t is not (see Lemma 10).

Now the above argument together with Lemmas 5-10 prove the following theorem.

Theorem 1:

The algorithm gives an optimum schedule of a tight graph.

7.4 Scheduling of a Loose Graph

Now let us extend the above algorithm so that it can handle a loose graph. To facilitate presentation, a few definitions are in order.

(From now on by "a graph", a "loose graph" is to be understood.)

Definition 9:

A node N in a graph G for which $d_I(N)+d_T(N) \neq h(G)$ is called a loose node. Otherwise a node is tight. Let N be a loose node. Then we define the far distance $\bar{d}_I(N)$ as $\bar{d}_I(N) = h(G) - d_T(N)$.

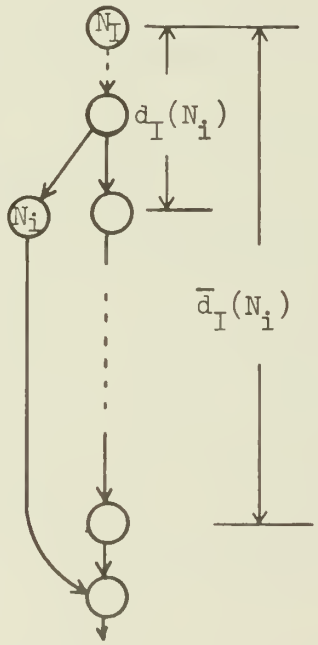


Figure 7.8. A Loose Node

A set B(i) is a node set (cf. Definition 4) such that

$$B(i) = \{N \mid (N \text{ is a tight node}) \text{ and } (d_I(N) = i)\} \cup \{N \mid (N \text{ is a loose node}) \text{ and } (\bar{d}_I(N) = i)\}.$$

We write Bt(i) and Bℓ(i) for the above two component sets respectively, i.e.

$$B(i) = Bt(i) \cup B\ell(i).$$

Note that a loose node N can be processed anywhere between $d_I(N)$ and $\bar{d}_I(N)$ without affecting the rest of a graph. Definition 9 says that loose nodes are pushed far down and classified in terms of the far distance rather than the forward distance.

Note that a loose node N receives two attributes, the far distance $\bar{d}_I(N)$ and the forward distance $d_I(N)$, and is classified in terms of $\bar{d}_I(N)$,

e.g. we say that a loose node N with the forward distance $d_I(N)$ is in B(i) where $i = \overline{d}_I(N)$.

We also define a set C(i).

Definition 10:

A set C(i) of loose nodes is defined as follows.

$C(i) = \{N \mid N \text{ is a loose node and } d_I(N) \leq i < \overline{d}_I(N)\}$.

C(i) is a set of those loose nodes which may be processed in parallel with a node in B(i). If $\overline{d}_I(N) = i$, then N is put in B(i) rather than in C(i). C(i) is a set of loose nodes which will be used to fill up waste processor time if necessary.

Scheduling a loose graph is similar to that for a tight graph. A loose graph is scheduled in accordance with B(i) for i=1,2,...,h(G). A loose node N, even though it is in $B(\overline{d}_I(N))$, may be scheduled with any B(k) where $d_I(N) \leq k \leq \overline{d}_I(N)$. All tight nodes are scheduled first and loose nodes are scheduled as late as possible. If it becomes inevitable to waste processor time if only tight nodes are used, then loose nodes are used to fill those otherwise wasted times.

In what follows B(i) plays a similar role to that of A(i) in the previous discussion. All definitions for A(i) are also applicable to B(i) with a few modifications.

Now the p-line relation ($\overset{p}{=}$) between two nodes N and N' in B_n^t is re-defined as follows.

Definition 5':

Let N and N' be two nodes in B_n^t. $N \overset{p}{\frown} N'((N,N'))$ holds if either one of:

(1) (i) N' is not a loose node (N may be a loose node),

and (ii) $d_I(N)$ (or $\bar{d}_I(N)$ if N is a loose node) + 1 = $d_I(N')$,

(In other words, $N \in B(t+k)$ and $N' \in Bt(t+k+1)$, $0 \leq k < n$.)

and (iii) $N \not\sim N'$.

or (2) (i) N' is a loose node (N may be a tight node),

and (ii) $d_I(N') \leq d_I(N)$ (or $\bar{d}_I(N)$ if N is a loose node) $< \bar{d}_I(N')$

(In other words, $N' \in B\ell(t+k)$ and $N \in B(t+j)$ where $d_I(N') \leq t+j < t+k \leq t+n$.) If $d_I(N') < t$, then the above inequality becomes

$$t \leq d_I(N) \text{ (or } \bar{d}_I(N)) < \bar{d}_I(N')$$

holds.

Then L_n^t on B_n^t is defined similarly to Definition 6.

Example 4 (see Figure 7.9):

$(2,5),(3,4),(4,8),(5,7) \in B_n^t(p)$ by (1). Since $d_I(N_6) = t + 1$ and $\bar{d}_I(N_6) = t + 3$, $(2,6),(3,6),(4,6),(5,6) \in B_n^t(p)$ by (2).

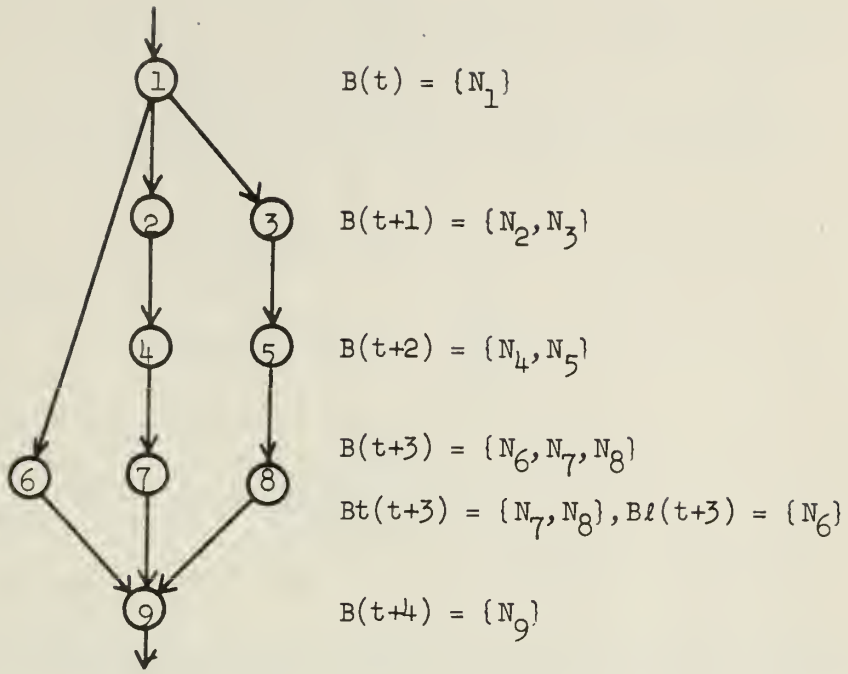
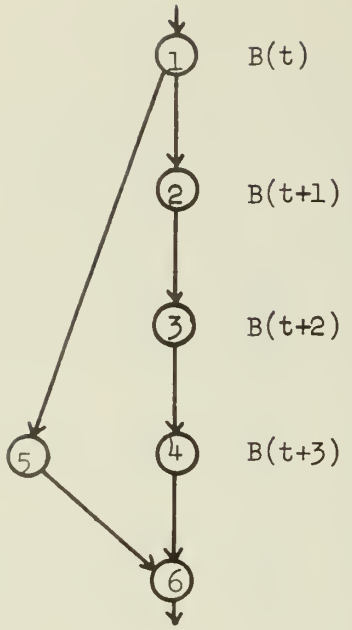


Figure 7.9. The p-line Relation in B_n^t

(1) of the above definition resembles Definition 5 and takes care of tight nodes. As Lemmas 2, 3 and 4 showed Definition 5 is more general than necessary. That is, the p-line relation need only to be defined between adjacent levels, i.e. A(i) and A(i+1). And Definition 5' follows this simplification.

(2) , on the other hand, takes care of loose nodes. If a loose node N is in B(i) then this means that $\overline{d}_i(N) = i$ and N may be processed in parallel with any node $N' \in B(j)$ where $d_I(N) \leq j \leq i$. Because of this addition, Lemmas 2, 3 and 4 do not hold anymore. For example let us consider the graph G:



It is easy to see that $(N_1,N_5) \in B_3^t(p)$ by (2) of Definition 5'. Since $(N_1,N_2) \notin B_3^t(p)$, there is no p-line (1) set on $B_3^t(p)$. This, however, does not bother us. Assume that B_n^t is p-connectable, and let $L_n^t = ((N_0,N_1'),(N_1,N_2'),\ldots,(N_i,N_{i+1}'),\ldots,(N_{k-1},N_k'))$, where $N_0 \in B(t)$ and $N_k' \in B(t+n)$. Further assume that $N_i \in B(t+a_i)$ and $N_{i+1}' \in B(t+a_{i+1})$. If $a_i + 1 \neq a_{i+1}$, then by Definition 5' N_{i+1}' is a loose node and $d_I(N_{i+1}') \leq t + a_i$. Thus N_{i+1}' can be processed in parallel with N_i without affecting the rest of a graph, i.e. we first process $B(t+a_i) - \{N_i\}$ and then process $\{N_i,N_{i+1}'\}$. If $a_i + 1 = a_{i+1}$, then the old strategy can be applied, i.e. we first process $B(t+a_i) - \{N_i\}$, then $\{N_i,N_{i+1}'\}$, then $B(t+a_{i+1}) - \{N_{i+1}'\}$.

This leads us to modify Definition 8 as follows.

Definition 8':

Given B_n^t and L_n^t, where $L_n^t = ((N_0,N_1'),(N_1,N_2'),\ldots,(N_i,N_{i+1}'),\ldots,(N_{k-1},N_k'))$. Then by to p-line schedule B_n^t by L_n^t, we mean the following scheduling.

(1) p-schedule $B(t) - \{N_0\}$.

(2) $g = 1$.

(3) p-schedule $\{N_{g-1},N_g'\}$.

(4) Let $N_{g-1} \in B(t+a_{g-1})$ and $N_g' \in B(t+a_g)$.

(i) If $a_{g-1} + 1 = a_g$, then p-schedule $B(t+a_g) - \{N_g',N_g\}$.

(ii) If $a_{g-1} + 1 \neq a_g$, then p-schedule $B(t+b)$ where $b = a_{g-1} + 1, a_{g-1} + 2, \ldots, a_g - 1$. Then p-schedule $B(t+a_g) - \{N_g',N_g\}$.

(5) $g = g + 1$. If $g < k$, then go to (3).

(6) p-schedule $\{N_{k-1},N_k'\}$.

(7) p-schedule $B(t+k) - \{N_k'\}$.

Finally in connection with Definition 8', we define the following. Suppose that B_n^t is not p-connectable, i.e. there is no p-line set L_n^t on B_n^t. It is, however, possible to find a p-line set L_s^t on B_s^t for some s, $0 \leq s < n$.

Definition 11:

Given B_n^t which is not p-connectable. Now we check if B_s^t is p-connectable for $s = 1,2,\ldots,n-1$. Let m be the smallest s such that B_s^t is p-

connectable but B^t_{s+1} is not. We call m the maximum p-connectable distance, and L^t_m a maximum p-line set.

The following example illustrates the above definition.

Example 5:

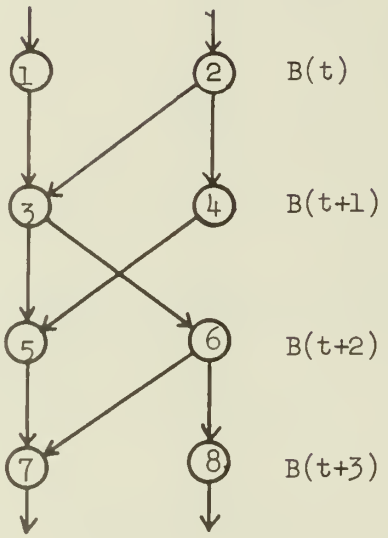


Figure 7.10. An Example of the Maximum p-connectable Distance

Let us consider p-connectability of B^t_3. It is clear that $B^t_3(p) = \{(N_1,N_4),(N_4,N_6),(N_5,N_8)\}$, and B^t_3 is not p-connectable. Since B^t_1 is p-connectable (N_1, N_4) but B^t_2 is not, the maximum p-connectable distance in the above B^t_3 is 1 and $L^t_1 = ((N_1,N_4))$.

Using a similar technique to the one described in Section 7.5, we can check p-connectability of B_n^t.

An algorithm to schedule a loose graph for two processors is now given. The algorithm resembles the one for a tight graph. The major difference lies in the treatment of loose nodes. Loose nodes are scheduled as late as possible. They will be used when it becomes inevitable to waste processor time.

In what follows we modify the definitions for B(i) and C(i) so that they do not include those loose nodes which have been already scheduled.

Loose Graph Scheduling Algorithm

Step 1: $t = 0$.

Step 2: If $t = h(G)$ then p-schedule all unscheduled nodes and stop, else go to Step 3.

Step 3: If $|B(t)|$ is even, then

(3-1) p-schedule B(t)

(3-2) go to Step 7.

Step 4: $|B(t)|$ is odd.

Find B(t+1).

Step 5: If $\forall N \in B(t)$ $|SR(N)| = |B(t+1)|$,

then

(5-1) Check C(t). If $C(t) = \emptyset$, then p-schedule B(t) and go to Step 7.

(5-2) Otherwise pick N with the minimum $\bar{d}_I(N)$ in C(t). (If there are more than one such N, pick any N.) Now p-schedule $B(t) \cup \{N\}$ and go to Step 7.

Step 6: There is $N' \in B(t)$ such that $|SR(N')| < |B(t+1)|$.

(6-1) If $|B(t+1)|$ is odd, then

(6-1-1) p-schedule $B(t) - \{N'\}$.

(6-1-2) p-schedule $\{N'\} \cup \{N''\}$ where $N'' \in B(t+1) - SR(N')$.

(6-1-3) p-schedule $B(t+1) - \{N''\}$.

(6-1-4) go to Step 7.

(6-2) $|B(t+1)|$ is even.

(6-2-1) Find out the smallest k greater than 1 such that $|B(t+k)|$ is odd.

(6-2-2) If there is no such k (i.e. we have checked up to $B(h(G))$), then p-schedule $B(i)$ $(t \leq i \leq h(G))$ individually and stop.

(6-2-3) Else we have a set $\mathcal{A} = \{B(t), B(t+1), \ldots, B(t+k)\}$ where $|B(t)|$ and $|B(t+k)|$ are odd and other $|B(t+i)|$ are all even. Check if B_k^t is p-connectable.

(1) B_k^t is not p-connectable.

(1-i) Find the maximum p-connectable distance in B_k^t. Let it be m.

(1-ii) Check $C(t+m)$. If $C(t+m) = \emptyset$, then p-schedule $B(t)$, $B(t+1), \ldots, B(t+k-1)$ individually. Let $t = t+k-1$. Go to Step 7.

(1-iii) Otherwise let $B'(t+m) = B(t+m) \cup \{N\}$ where $N \in C(t+m)$ and has the minimum $\bar{d}_I(N)$. Let $B'^t_m = \bigcup_{i=0}^{m-1} B(t+i) \cup B'(t+m)$. Then p-line schedule B'^t_m by a maximum

p-line set L_m^t. Let t = t + m. Go to Step 7.

(2) B_k^t is p-connectable.

(2-i) Find a p-line set L_k^t.

(2-ii) p-line schedule B_m^t by L_k^t.

(2-iii) t = t + k.

(2-iv) Go to Step 7.

Step 7: t = t + 1.

Go to Step 2.

Proof for the Algorithm:

That the above algorithm is optimum can be proved by a similar argument used to prove the previous algorithm. For example, we can show that Steps 3, 5-1, 6-1, 6-2-2 and 6-2-3(2) are optimum easily by previous Lemmas. Now we have to show that Steps 5-2 and 6-2-3(1) are optimum.

Lemma 11:

Step 5-2 is optimum.

Proof:

Suppose that we do not use N in C(t) to fill up otherwise wasted processor time, where N is the node with the minimum $\bar{d}_I(N)$ in C(t). Saving N for later use, however, does not improve the situation because a node N cannot be used more effectively than to fill up wasted processor time anyway.

Also the choice of N from C(t) is optimum. Suppose for example that N' with $\bar{d}_I(N') > \bar{d}_I(N)$ is also in C(t). Now let us consider those two nodes (See Figure 7.12). Whether we use N or N' to schedule with B(t) will not make

any difference to schedule $B(t+1), B(t+2), \ldots, B(\bar{d}_I(N))$ because a node N or N' is available if necessary. Suppose we used N' with B(t). Then it is not possible to fill a later request which may arise when B(u) ($\bar{d}_I(N) + 1 \leq u < \bar{d}_I(N')$) is scheduled, whereas if we used N with B(t) then we can fill the request. Thus this proves the lemma.

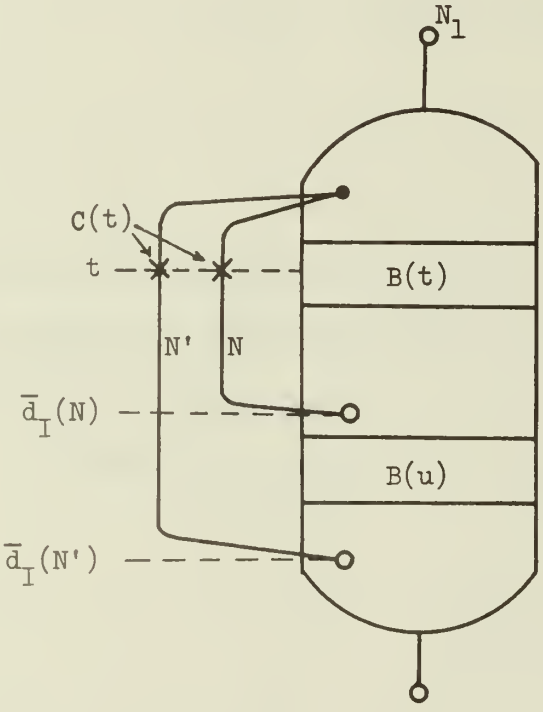


Figure 7.11. An Illustration for Lemma 13

(Q.E.D.)

That Step 6-2-3(1) is optimum is proved similarly.

Now we have the following theorem.

Theorem 2:

The algorithm gives an optimum schedule of a loose graph.

7.5 Supplement

(1) An algorithm to establish $A_n^t(p)$ on A_n^t is now discussed.

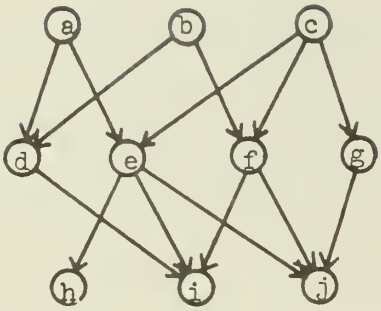
Let $m = \sum_{i=0}^{n} |A(t+i)|$, and B be an $m \times m$ connection matrix where the first $|A(t)|$ columns and rows are labeled by nodes in A(t), the second $|A(t+1)|$ columns and rows by nodes in A(t+1), and so forth. An element b_{ij} of B is 1 if and only if $N_i \to N_j$ where N_i and N_j are labels for the i-th row and the j-th column.

Now define the multiplication of the connection matrices as follows. Let $A = B \times C$ where A, B and C are $m \times m$ connection matricies. Then $a_{ij} = \bigvee_{k=1}^{m} (b_{ik} \wedge C_{kj})$. Now complete $\bar{B}^m = \bigvee_{k=1}^{m} B^k$. In $\bar{B}^m$, $\bar{b}^m_{ij} = 1$ implies that $N_i \geqslant N_j$ and $\overline{b_{ij}}^m = 0$ implies that (N_i, N_j). For example, let us consider the graph in Figure 7.13. Then we have

$$A_2^1 = \{(a,f),(a,g),(b,e),(b,h),(b,g),(c,d),(d,h),(d,j),(f,h),(g,h),$$
$$(g,i)\}.$$

(2) Given $A_n^t(p)$ for A_n^t, an algorithm to find p-connectivity and $L_n^t(1)$ is described.

According to Lemma 4, if A_n^t is p-connectable, then there is a p-line (1) set $L_n^t(1)$. Thus to check p-connectability it is enough to examine if there is a p-line (1) set $L_n^t(1)$.



(a) G

	a	b	c	d	e	f	g	h	i	j
a				1	1					
b				1		1				
c					1	1	1			
d									1	
e								1	1	1
f									1	1
g										1
h										
i										
j										

(b) B

	a	b	c	d	e	f	g	h	i	j
a				1	1			1	1	1
b				1		1			1	1
c					1	1	1	1	1	1
d									1	
e								1	1	1
f									1	1
g										1
h										
i										
j										

(c) $\overline{B}^2$

Figure 7.12. An Example for $A^t(\underline{\underline{p}})$

First let

$$A_n^t(p)(1) = A_n^t(p) - \{(N,N') \mid |d_I(N) - d_I(N')| > 1\}$$

Now we construct a graph $\bar{A}_n^t$ as follows.

(1) $\bar{A}_n^t$ has following nodes:

(i) an initial node N_s,

(ii) a terminal node N_E,

(iii) all nodes in A_n^t,

(iv) for each node N ($N \in A_n^t$, $N \notin A(t)$, $N \notin A(t+n)$) a new duplicate node N'.

(2) $\bar{A}_n^t$ has following verticies:

(i) a vertex from N_s to every node N which was in A(t),

(ii) a vertex from every node N which was in A(t+n) to N_E,

(iii) if $(N_1,N_2) \in A_n^t(p)(1)$, then a vertex from N_1 to N_2.

(iv) for every $A(t+k), 1 \leq k \leq n-1$, for every $N \in A(t+k)$, a vertex from N to every N' which is a duplicate of $N'' \in A(t+k)$ but is not identical to N.

To illustrate the above definition, let us consider the following example.

Example:

Let $A_2^t = \{a,b,c,d,e,f,g\}$

where $a,b \in A(t)$

$c, d, e \in A(t+1)$

and $f, g \in A(t+2)$.

Further let

$$A_2^t(p)(1) = \{(b,c),(d,f),(d,g),(c,f)\}$$

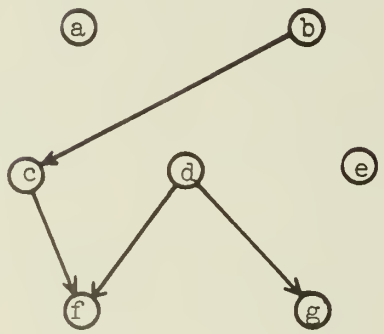
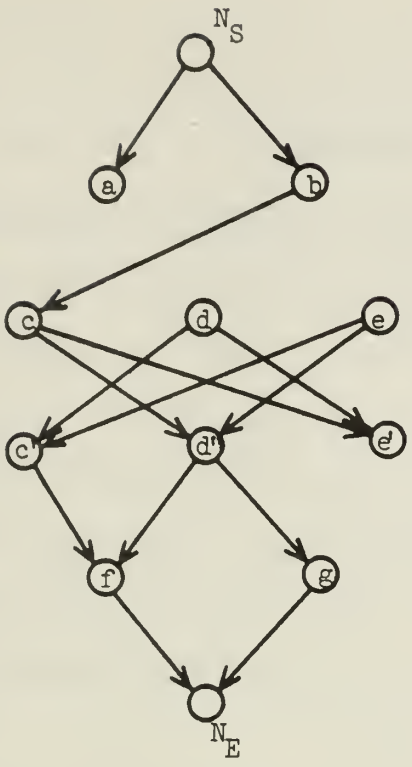


Figure 7.13. An Example for p-connectivity Discovery

Then $\bar{A}_n^t$ has nodes = {N_s, N_E a,b,c,d,e,f,g (which are nodes in A_n^t), c',d',e' (which are duplicates of nodes in A(1))} and verticies = {$(N_s,a),(N_s,b),(f,N_E)$, $(g,N_E),(b,c),(d,f),(d,g),(c,f)$(which are original verticies in A_n^t),(c,d'),(c,e'), (d,c'),(d,e'),(e,c'),(e,d')(which are verticies from N to a duplicate N' of N" which is not identical to N).}

Now it is clear that A_n^t has a p-line (1) set $L_n^t(1)$ if and only if there is a path from N_s to N_E in $\bar{A}_n^t$. There is a well-known algorithm for path finding, e.g. [19].



(b)

Figure 7.13. (continued)

8. CONCLUSION

This thesis introduced new techniques to expose hidden parallelism in a program. Techniques included the use of one of the fundamental arithmetic laws, i.e., the distributive law, extensively. Furthermore it was suggested that with the help of these techniques computation of a program might be speeded up logarithmically in the sense that computation time became a logarithmic function of the number of single variable occurrences in a program rather than its linear function. Even though discussions were based on an ILLIAC IV type machine, as mentioned before, they are readily applicable to pipe-line machines such as CDC STAR.

Chapter 2 of the thesis studied parallel computation of summations, powers and polynomials. The minimum time to evaluate summations or powers as well as the minimum number of PE's required to attain it was given. A scheme which computed a polynomial in parallel in lesser time than any known scheme was also introduced. Because of its simplicity in scheduling, the k-th order Horner's rule for parallel polynomial computation was studied in detail. It was shown that for this algorithm the availability of more PE's sometimes increased the computation time. The algorithm was such that all PE's were forced to participate in computation.

Chapter 3 presented an algorithm which reduced tree height for an arithmetic expression by distribution. The algorithm worked from the inner most parenthesis pair to the outer most one and scanned an arithmetic expression only once. A measure for the height of the minimum height tree for an arith-

metic expression was given as a function of the depth of parenthesis pair nesting and the number of single variable occurrences in it.

Chapter 4 extended the above idea to cover a sequence of arithmetic expressions. It was shown that by replacing a sequence of arithmetic expressions with an arithmetic expression by back substitution, the computation time could be speeded up in a logarithmic way for a certain class of iteration formulas, e.g., $x_{i+1} := a \times x_i + b$. The chapter also showed that parallel computation was in general more favorable than sequential computation in terms of the round off error. Furthermore it was shown that distribution would not introduce the significant amount of the round off error.

Chapter 5 studied inter-statement parallelism as an introduction to the following chapter. An algorithm which checked if the execution of statements in a program by some sequence gave the same results as the execution of statements by the given sequence did was given. The algorithm was new in the sense that it prevented variables from being updated before they were used. This had not been taken into account by the previous works. Also a technique which exploited more parallelism between statements by introducing temporary locations was introduced.

Chapter 6 presented an algorithm which checked if a statement in a loop could be executed simultaneously for all values of a loop index. The algorithm checked index expressions and the way the values of indices varied only and did not require a loop to be replaced with a sequence of statements. In case a statement in a loop could not be executed in parallel with respect to a loop index as it was, the algorithm "skewed" the computation of a statement with respect to a loop index so that the statement could be executed in parallel for all values of the loop index. Also to expose hidden parallelism

from a loop, replacement of a loop with several loops was discussed.

A solution for the equally weighted-two processor scheduling problem was given in Chapter 7. The only practical work so far obtained was a result for scheduling a rooted tree with equally weighted tasks on k identical processors. The solution given in Chapter 7 scheduled a graph with equally weighted tasks on two identical processors. If we considered common expressions in an arithmetic expression then we would obtain a graph of operations rather than a tree for the arithmetic expression and the scheduling algorithm was readily applicable for scheduling that graph on P(2).

Suggestions for further research have been given in several places throughout the thesis and need not be repeated here. We conclude by giving two possible extensions that deserve brief mention.

(1) The design of a better machine.

Even though we assumed that a PE can communicate with any other PE instantaneously, this may not be the case in reality because it is costly and impractical to provide data paths between every PE pair. Hence it is necessary to design PE interconnection which is economical yet powerful enough to simulate the above idealized interconnection [25], [26].

(2) Generalization of the idea given in this thesis.

The three laws of arithmetic were utilized in this thesis in terms of parallelism exploitation. We should, however, pay more attention on these laws even in terms of serial computation. For example suppose an arithmetic expression which involves matrices, row and column vectors is given. Then by the appropriate application of the associative law, the number of multiplications required may be reduced drastically.

LIST OF REFERENCES

[1] Abrahams, P. W., "A Formal Solution to the Dangling else of ALGOL 60 and Related Languages", Comm. ACM, 9 (September, 1966), pp. 679-682.

[2] Abel, N. E., et al., "TRANQUIL: A Language for an Array Processing Computer", Proc. of the Spring Joint Computer Conference (1969), pp. 57-73.

[3] Naur, P., et al., "Revised Report on the Algorithmic Language ALGOL 60", Comm. ACM, 6 (January, 1963), pp. 1-17.

[4] Allard, R. W., Wolf, K. A. and Zemlin, R. A., "Some Effects of the 6600 Computer on Language Structures", Comm. ACM, 7 (February, 1964), pp. 112-119.

[5] Baer, J. E., "Graph Models of Computations in Computer Systems", Ph.D. Dessertation, University of California, Los Angeles, Report No. 68-46 (October, 1968).

[6] Baer, J. E. and Bovet, D. P., "Compilation of Arithmetic Expressions for Parallel Computations", Proc. of IFIP Congress (1968), pp. 340-346.

[7] Barnes, G. H., et al., "The Illiac-IV Computer", IEEE Trans. of Computers, C-17 (August, 1968), pp. 746-757.

[8] Beightler, C. S., et al., "A Short Table of z-Transforms and Generating Functions", Operations Research, 9 (July-August, 1961), pp. 574-578.

[9] Bingham, H. W., Reigel, W. E. and Fisher, D. A., "Control Mechanisms for Parallelism in Programs", Burroughs Corporation, ECOM-02463-7 (October, 1968).

[10] Bingham, H. W. and Reigel, W. E., "Parallelism Exposure and Exploitation in Digital Computing Systems", Burroughs Corporation, ECOM-02463-F (June, 1969).

[11] Brewer, M. A., "Generation of Optimal Code for Expressions via Factorization", Comm. ACM, 12 (June, 1969), pp. 333-340.

[12] "Newsdata", Computer Decision (March, 1970), p. 2.

[13] Conway, M. E., "A Multiprocessor System Design", Proc. of the Fall Joint Computer Conference (1963), pp. 139-146.

[14] Conway, R. W., Maxwell, L. W. and Miller, L. W., Theory of Scheduling, Addison-Wisley Publishing Company, Inc., New York (1967).

[15] Dorn, W. S., "Generalizations of Horner's Rule for Polynomial Evaluation", IBM Journal of Research and Development, 6 (April, 1962), pp. 239-245.

[16] Estrin, G., "Organization of Computer Systems--the Fixed plus Variable Structure Computer", Proc. of Western Joint Computer Conference (May, 1960), pp. 33-40.

[17] Gold, D. E., "A Model for Linear Programming Optimization of I/O--Bound Programs", M.S. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Report No. 340 (June, 1969).

[18] Graham, W. R., "The Parallel and the Pipeline Computers", Datamation (April, 1970), pp. 68-71.

[19] Harary, F., Norman, R. Z. and Cartwright, D., Structural Model: An Introduction to the Theory of Directed Graphs , John-Wiley and Sons, Inc., New York (1966).

[20] Hellerman, H., "Parallel Processing of Algebraic Expressions", IEEE Trans. of Electronic Computers, EC-15 (January, 1966), pp. 82-91.

[21] Hu, T. C., "Parallel Sequencing and Assembly Line Problems", Operation Research, 9 (November-December, 1961), pp. 841-848.

[22] Knowls, M., et al., "Matrix Operations on Illiac-IV", Department of Computer Science, University of Illinois at Urbana-Champaign, Report No. 222 (March, 1967).

[23] Knuth, D. E., The Art of Computer Programming, Vol. 2, Addison-Wesley Publishing Company, Inc., New York (1969).

[24] Kuck, D. J., "Illiac-IV Software and Application Programming", IEEE Trans. of Computers, C-17 (August, 1968), pp. 758-769.

[25] Kuck, D. J. and Muraoka, Y., "A Machine Organization for Arithmetic Expression Evaluation and an Algorithm for Tree Height Reduction", unpublished (September, 1969).

[26] Kuck, D. J., "A Preprocessing High Speed Memory System", to be published.

[27] Logan, J. R., "A Design Technique for Digital Squering Networks", Computer Design (February, 1970), pp. 84-88.

[28] Minsky, L. M., Computation: Finite and Infinite Machines, Prentice-Hall, Inc., New Jersey (1967).

[29] Motzkin, T. S., "Evaluation of Polynomials and Evaluation of Rational Functions", Bull. A.M.S., 61 (1965), p. 163.

[30] Murtha, J. C., "Highly Parallel Information Processing System", in Advances in Computers, Academic Press, Inc., New York, 7 (1966), pp. 2-116.

[31] Muntz, R. R. and Coffman, E. G., "Optimal Preemptive Scheduling on Two Processor Systems", IEEE Trans. of Computers, C-18 (November, 1969), pp. 1014-1020.

[32] Nievergelt, J., "Parallel Methods for Integrating Ordinary Differential Equations", Comm. ACM, 7 (December, 1964), pp. 731-733.

[33] Noyce, R. N., "Making Integrated Electronics Technology Work", IEEE Spectrum, 5 (May, 1968), pp. 63-66.

[34] Ostrowski, A. M., "On Two Problems in Abstract Algebra Connected with Horner's Rule", in Studies in Mathematics and Mechanics Presented to R. von Mises, Academic Press, New York (1954), pp. 40-48.

[35] Pan, V. Ya., "Methods of Computing Values of Polynomials", Russian Mathematical Surveys, 21 (January-February, 1966), pp. 105-136.

[36] Ramamoorthy, C. V. and Gonzalez, M. J., "A Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs", Proc. of the Fall Joint Computer Conference (1969), pp. 1-15.

[37] Russel, E. C., "Automatic Program Analysis", University of California, Los Angeles, Report No. 69-72 (March, 1969).

[38] Shedler, G. S. and Lehman, M. M., "Evaluation of Redundancy in a Parallel Algorithm", IBM System Journal, 6, 3 (1967), pp. 142-149.

[39] Squire, J. S., "A Translation Algorithm for a Multiple Processor Computer", Proc. of the 18th ACM National Conference (1963).

[40] Stone, H. S., "One-pass Compilation of Arithmetic Expressions for a Parallel Processor", Comm. ACM, 10 (April, 1967), pp. 220-223.

[41] Thompson, R. N. and Wilkinson, J. A., "The D825 Automatic Operating and Scheduling Problem", in Programming Systems and Languages, McGraw-Hill, New York (1967), pp. 647-660.

[42] Winograd, S., "On the Time Required to Perform Addition", JACM, 12 (April, 1965), pp. 277-285.

[43] Winograd, S., "The Number of Multiplications Involved in Computing Certain Functions", Proc. of IFIP Conference (1968), pp. 276-279.

VITA

Yoichi Muraoka was born in Sendai, Japan, on July 20, 1942. He graduated from Waseda University, Tokyo, Japan, in Electrical Engineering in March, 1965 and started his graduate study at the graduate college, Waseda University.

Since September 1966, he has been a research assistant with the project of Illiac IV computer in the Department of Computer Science of the University of Illinois at Urbana-Champaign. In 1969 he received his degree of Master of Science in Computer Science.

He is a member of the Association for Computing Machinery and the Institute of Electrical and Electronics Engineering.


BOUND BY
HNM