30738

National Library of Canada
Bibliothèque nationale du Canada

CANADIAN THESES ON MICROFICHE

THÈSES CANADIENNES SUR MICROFICHE

NAME OF AUTHOR/*NOM DE L'AUTEUR* Robert K. Lee

TITLE OF THESIS/*TITRE DE LA THÈSE* Optimal Parallel Computations for SIMD Computers

UNIVERSITY/*UNIVERSITÉ* University of Alberta

DEGREE FOR WHICH THESIS WAS PRESENTED/ *GRADE POUR LEQUEL CETTE THÈSE FUT PRÉSENTÉE* Doctor of Philosophy

YEAR THIS DEGREE CONFERRED/*ANNÉE D'OBTENTION DE CE DEGRÉ* Fall 1976

NAME OF SUPERVISOR/*NOM DU DIRECTEUR DE THÈSE* Dr. Stanley Cabay

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.

DATED/*DATÉ* September 10, 1976 SIGNED/*SIGNÉ* Robert Lee

PERMANENT ADDRESS/*RÉSIDENCE FIXÉ* 8 Lane 132, Dragon River Road, Teipei, TAIWAN

THE UNIVERSITY OF ALBERTA

OPTIMAL PARALLEL COMPUTATIONS FOR SIMD COMPUTERS

by

© ROBERT KAI-SHUAN LEE

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

FALL, 1976

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled "Optimal Parallel Computations for SIMD Computers"

submitted by Robert K. Lee

in partial fulfilment of the requirements for the degree of Doctor of Philcsophy.

....
Supervisor

....

....

....

....
External Examiner

DATE August 16, 1976

TO MY PARENTS AND MY WIFE

ABSTRACT

Efficient parallel algorithms for SIMD (Single Instruction stream - Multiple Data stream) parallel computers are considered for a variety of problems. Included are efficient parallel algorithms for performing integer arithmetic, polynomial arithmetic and modular arithmetic. These parallel algorithms lead to efficient or (asymptotically) optimal methods for solving the following problems:

1) parallel integer arithmetic:
   addition, subtraction, multiplication and division of multiple-precision integers.
2) parallel polynomial arithmetic:
   addition, subtraction, multiplication, division and evaluation of polynomials, and evaluation of elementary symmetric functions.
3) parallel modular arithmetic:
   residue evaluation of multiple-precision integers and of polynomials, the Chinese remainder problem, polynomial interpolation, and the solution of full linear systems of equations by modular methods.

Among the parallel algorithms given in this thesis, three broad classes can be identified according to their level of parallelism:

1) parallel algorithms assuming linearly bounded parallelism, e.g., addition and subtraction of

integers, addition and subtraction of polynomials.

2) parallel algorithms assuming polynomially bounded parallelism, e.g., multiplication of integers, multiplication of polynomials, residue evaluation of polynomials, ~~polynomial~~ interpolation, and the evaluation of elementary symmetric functions.

3) parallel algorithms assuming exponentially bounded parallelism, e.g., division of integers, division of polynomials, residue evaluation of multiple-precision integers, the Chinese remainder problem, and the solution of full linear systems of equations.

ACKNOWLEDGEMENTS

I wish to express my gratitude to my supervisors, Dr. S. Cabay and Dr. I. N. Chen, for their guidance throughout the preparation of this thesis.

The financial assistance provided by the University of Alberta, in the form of a Dissertation Fellowship, is gratefully acknowledged.

TABLE OF CONTENTS

TABLE OF NOTATIONS

$\|A\|$	the cardinality of any set A
$A(i,j)$	the (i,j) cofactor of any matrix A
$\mathrm{Adj}(A)$	the adjoint matrix of any matrix A
$\det(A)$	the determinant of any matrix A
b	the base (or radix) of positional integers
$\deg(f)$	the degree of any polynomial f
F	an arbitrary algebraically complete field
$F[x,y]$	the ring of polynomials in x and y over F
$F(x,y)$	the field of rational functions in x and y over F
$GF(p)$	the field of integers modulo a prime p
$\log n$	the logarithm of n taken base 2
$n!$	the factorial of any non-negative integer n
$O(f(n))$	of order f(n)
$\lfloor f/g \rfloor$	the quotient of f divided by g, where f and g are in any Euclidean domain
$f \bmod g$	the remainder of f divided by g, where f and g are in any Euclidean domain
$s(n,k)$	the elementary symmetric function of degree k in n indeterminates
$\|x\|$	the absolute value of any number x
$\lfloor x \rfloor$	the greatest integer ≤ the number x
$\lceil x \rceil$	the smallest integer ≥ the number x

CHAPTER ONE

Introduction

The central theme of this thesis is optimal parallel computations for SIMD (Single Instruction stream - Multiple Data stream) parallel computers (Cf. Section 1.1). Many people think that parallel computations are extensions of sequential computations. This view is quite wrong. Parallel computations give rise to unique problems and issues beyond those of sequential computations [Stone 1973, Newell and Robertson 1975]. In fact, some mistakes of earlier research (and including some recent research) in parallel computations can be directly contributed to this view. For example, it is not surprising that adapting the classical algorithms such as Gauss-Jordan elimilation, LDU factorization, or Givens reduction on parallel computers is far from optimal.

In order to prepare discussion for subsequent chapters, this chapter is intended to clarify some important points and define our objectives.

In Section 1.1 we classify parallel computers according to Flynn's stream concept. Two broad categories can be obtained: the SIMD parallel computers and the MIMD (Multiple Instruction stream - Multiple Data stream) parallel computers. We explain why SIMD parallel computers are favored in this thesis. Also, in this section, the characteristics of idealized SIMD parallel computers, which serve as the basis of all parallel algorithms in the subsequent chapters, are described.

In Section 1.2 models of parallel computation and measurements of parallel complexity are given. Since the aim of this study is to gain insight into parallel computational complexity, we concentrate on the optimality and asymptotic behavior of parallel algorithms.

In Section 1.3 we briefly summarize the contributions of this thesis.

1.1 Classification of Parallel Computers

Many computers now exist which are capable of executing more than one instruction simultaneously [Barnes et al 1968, Hintz & Tate 1972, Johnson 1972, Rudolph 1972, Watson 1972, Wulf & Bell 1972]. Although all these computers are grossly termed as parallel computers, there are significant functional and architectural differences between any two parallel computers. A number of different approaches to classification of parallel computers are possible [Thurber & Wald 1975]. Most techniques use global architectural properties and are thus valid only within limited ranges. For the purpose of studying parallel computations, Flynn's stream concept [1966, 1972] will be adopted. Stream in this context simply means a sequence of instructions or data as executed or operated on by a processor. The advantage of this approach is that it describes a parallel computer from a macroscopic point of view, and yet avoids the pitfalls of relating such descriptions to a particular problem.

Using the stream concept, parallel computers are categorized by the magnitude (either in space or time multiplex) of interactions of their instruction and data streams. The two major categories are:

1) The Single Instruction stream - Multiple Data stream (SIMD) parallel computers have only one stream of instructions in execution at any time, but each instruction may affect many different data. These

parallel computers can be further categorized as being

a) parallel in space, as are structured array computers [1] (ILLIAC IV, OMEN-60, SIMDA);

b) unstructured linear array computers, or ensembles [2] (PEPE, Goodyear STARAN S);

c) parallel primarily in time, as are pipeline computers (CDC STAR-100, TI ASC);

d) associative computers whose memories are addressed by contents rather than by addresses (Goodyear STARAN S, PEPE, OMEN-60, SIMDA).

2) The Multiple Instruction stream - Multiple Data stream (MIMD) parallel computers have more than one stream of instructions, in fact, as many instruction streams as there are data streams. These parallel computers are essentially interconnected sequential computers, usually called multiprocessors (UNIVAC 1108 multiprocessor system, C.mmp [3], IMP).

The MIMD parallel computers that have been built so far

[1] A structured array computer has a high level of interconnectivity such as exhibited by ILLIAC IV's four nearest neighbor connections.

[2] An ensemble such as PEPE tends not to have much interprocessor connectivity except by transmission to the control unit and retransmission to the appropriate processor.

[3] C.mmp consists of 16 PDP-11 computers connected through a crosspoint switch to 16 primary-memory ports.

tend to have only 2, 4, or in rare instances 16 processors (C.mmp), whereas the SIMD parallel computers have an effective parallelism as small as 64 (ILLIAC IV) to upwards of 288 (PEPE) and 1024 (STARAN S) [4]. In addition, it is likely that the scale gap will become greater and greater due to the recent microprocessor / computer-on-a-chip revolution [Soucek 1976]. Unfortunately, most research previously done in the area of parallel computations is strongly oriented towards MIMD computations.

Since it is evident that technology for SIMD parallel computers is approaching maturity and a good number of large scale SIMD parallel computers are available, there is an urgent need for efficient parallel algorithms designed specifically for SIMD parallel computers. Without efficient parallel algorithms these computer giants are just ordinary dwarfs, and hardware worth millions of dollars is just another engineering spectacle. Therefore, in this thesis we are solely interested in designing parallel algorithms for SIMD parallel computers. [5] Proposed parallel algorithms are assumed to be executed on idealized SIMD parallel computers which can be characterized as follows:

[4] STARAN has a potential parallelism as large as 8192 in the most advanced model.

[5] Any parallel algorithm designed for SIMD parallel computers can easily e adapted for MIMD parallel computers, but not conversely.

1) There are an indefinite number [6] of identical processors, each able to execute the usual arithmetic, Boolean, relational (i.e., comparison), and control operations, and each with its own memory.

2) All processors obtain their instructions simultaneously from a single instruction stream broadcast by the control unit. Thus, all processors execute the same instruction, but operate on data stored in their own memories.

3) Each processor has a distinct number by which it may be identified by the control unit or referenced by an instruction.

4) Any processor may be disabled from performing an instruction. The enabling or disabling of individual processors is effected according to the result of some local test.

5) All processors can exchange data with each other over predefined data paths (ILLIAC IV) or through a data permutation network (STARAN S).

[6] However, this number is bounded [Kuck & Muraoka 1974, Kuck & Maruyama 1975, Chen & Kuck 1975].

1.2 Complexity of Parallel Algorithms

If algorithms for solving certain problems are to be analyzed and if lower bounds for solving these problems are to be obtained, models of computation and measurements of complexity must be first given. The following definition is evolved from a sequential computational model [Winograd 1970]. A parallel model similar to this one is proposed by Munro and Paterson [1973].

Definition 1.2.1 Given any three sets A, B and R, a computation of R in A given B is a finite sequence of non-empty sets $S_i \subset A$, $0 \leq i \leq t$, such that $S_0 = B$, $R \subset \bigcup_{i=0}^{t} S_i$ and for each $i > 0$, if $x \in S_i$ then $x = (y \text{ op } z)$, where $y, z \in \bigcup_{j=0}^{i-1} S_j$ and op $\in O_i$. The set O_i contains the types of all operations used for obtaining S_i. The integer t is the number of operations required for the computation. The elements in R are the results of the computation.

Note that in the above definition all the intermediate results S_i, $1 \leq i \leq t$, are contained in the set A. In fact, this is often a convenient way to express the types of operations which are allowed in a computation. For example, let F be an arbitrary algebraically complete field (7) and $\{ x_i \mid 1 \leq i \leq n \}$ be the input arguments. If arithmetic and only arithmetic operations are allowed for a computation,

[7] Throughout this thesis F denotes an algebraically complete field.

then the set A can be expressed as the (symbolic) rational field $F(x_1, x_2, \ldots, x_n)$. On the other hand, if the set A is specified as the ring $F[x_1, x_2, \ldots, x_n]$, then this is implicitly saying that division is not allowed. In the more general case, however, all arithmetic, Boolean and relational operations are allowed (Cf. Section 1.1) and a definition of the set A is normally omitted.

Example. Let w be a primit~~ive n-th root~~ of unity in F. Then the following identity holds:

$$\sum_{i=0}^{n-1} w^i / (x-w^i) = n / (x^n-1).$$

Based on this identity, the following sequence of sets is a computation for evaluating x^8 in F(x) given F U {x} [Kung 1974].

Sets	Comments
$S_0 = F \cup \{x\}$,	initially given.
$S_1 = \{A_i \text{<-} x-w^i \mid 1 \le i \le 8\}$,	$O_1 = \{-\}$, $x \in S_0$, $w^i \in S_0$.
$S_2 = \{B_i \text{<-} w^i / A_i \mid 1 \le i \le 8\}$,	$O_2 = \{/\}$.
$S_3 = \{C_i \text{<-} B_{2i-1} + B_{2i} \mid 1 \le i \le 4\}$,	$O_3 = \{+\}$.
$S_4 = \{D_i \text{<-} C_{2i-1} + C_{2i} \mid 1 \le i \le 2\}$,	$O_4 = \{+\}$.
$S_5 = \{E \text{<-} D_1 + D_2\}$,	$O_5 = \{+\}$.
$S_6 = \{G \text{<-} 8 / E\}$,	$O_6 = \{/\}$, $8 \in S_0$.
$S_7 = \{G+1\}$,	$O_7 = \{+\}$, $1 \in S_0$.

The number of operations required for this computation is 7. Also, note that all (and only) arithmetic operations are used in the computation. Thus, the computation is indeed in F(x).

Definition 1.2.2 A computation is called a sequential computation if and only if $|S_i| = 1$ for all $i > 0$; otherwise it is called a parallel computation. (8)

Note that in the above example, since $|S_1| = 8$, the computation is a parallel computation.

Definition 1.2.3 A parallel computation is called a SIMD computation if and only if $|O_i| = 1$ for all $i > 0$; otherwise it is called a MIMD computation.

Note that in the above example, since $|O_i| = 1$ for all i, the parallel computation is a SIMD computation.

Parallel algorithms can be measured by a variety of criteria (Cf. [Stone 1973] for alternative criteria). Primarily, we are interested in the rate of growth of the number of parallel operations required for solving very large instances of a problem. The number of parallel arithmetic and Boolean operations (9) required by a parallel algorithm expressed as an integer function of the attributes

(8) Given any set A, $|A|$ denotes the cardinality of A.

(9) We ignore all overhead [illegible] memory access and data exchange [Kung 1974, Brent 1974, Chen & Kuck 1975, Kuck & Muraoka 1975, Kuck & Maruyama 1975]. This assumption is justifiable because overhead is roughly proportional to the number of parallel operations performed and ignoring it affects the parallel complexity by a constant factor only.

(e.g., the precision of an integer, the size of a vector, the order of a matrix, etc.) of the input arguments is called the parallel complexity of the parallel algorithm.

Let $f(n)$ be the parallel complexity of a parallel algorithm for solving a certain problem with a single attribute and $g(n)$ be the parallel lower bound for solving the problem. The parallel algorithm is said to be optimal if and only if $f(n) = g(n) + O(1)$ and asymptotically optimal if and only if $f(n) = O(g(n))$.[10] The (asymptotic) optimality of a parallel algorithm with more than one attribute can be similarly defined.

We emphasize that, just as is the case in almost all research done in sequential computational complexity [Aho et al 1974, Borodin & Munro 1975], rather than attempting to provide the best practical parallel algorithms, the aim of this study is to gain insight into the intrinsic difficulty for solving various problems on SIMD parallel computers. This emphasis leads us to concentrate on the optimality and the asymptotic behavior of parallel algorithms. Thus, in certain instances, no matter how straightforward or impractical the parallel algorithms are, they are still given and analyzed simply to show that these parallel algorithms are (asymptotically) optimal or that the parallel

[10] Given two sequences (i.e., functions defined on integers) $f(n)$ and $g(n)$ such that $g(n) \geq 0$ for all n, we write $f(n) = O(g(n))$ if and only if there exists a constant $K > 0$ such that $|f(n)| \leq K \cdot g(n)$ for all n.

lower bounds can be attained. Of course, in other instances, the parallel algorithms given are not only intricate but also practically useful.

1.3 Contributions

The problems we are concerned with are the fundamental processes for doing different kinds of arithmetic including integer arithmetic, polynomial arithmetic and modular arithmetic.

On sequential computers, the subject has been thoroughly covered in Chapter 4 of Knuth's book [1969]. On parallel computers, except for a few isolated results, very little research has been done.

The main contributions of this thesis are efficient or (asymptotically) optimal parallel algorithms for the following problems:

1) parallel integer arithmetic:
   a) addition or subtraction of n-digit integers can be computed in O(log n) parallel operations.
   b) multiplication of an m-digit integer by an n-digit integer can be computed in O(log(m+n)) parallel operations.
   c) division of an (m+n)-digit integer by an n-digit integer can be computed in O(log(m+n)) parallel operations.
2) parallel modular arithmetic:
   a) polynomial interpolation at n points can be computed in O(log n) parallel operations.
   b) the Chinese remainder problem with n moduli can be solved in O(log n) parallel operations.
   c) full linear systems of order n can be solved

exactly in O(log n) parallel operations.

3) parallel polynomial arithmetic:

   a) division of a polynomial of degree m+n by a polynomial of degree n can be computed in O(log(m+1)) parallel operations.

Other contributions include the derivation of tight parallel lower bounds for the evaluation of polynomials and elementary symmetric functions.

CHAPTER TWO

Techniques for Proving Parallel Lower Bounds

Presently, techniques for proving parallel lower bounds are not only scarce but also immature. Among the few reported parallel lower bounds, results are either left unproved or proved in an ad hoc manner. More seriously, existing techniques fail to give tight parallel lower bounds for even very common problems. The purpose of this chapter is to discuss techniques for finding tighter parallel lower bounds.

The fan-in argument, which gives a relationship between the number of input arguments, or the minimal number of sequential operations, and the minimal number of parallel operations, is illustrated in Section 2.1; the growth argument, which determines a relationship between the growth of a result and the minimal number of parallel operations, in Section 2.2; and the substitution argument, which gives a relationship between classes of problems, in Section 2.3.

Section 2.3 is concluded with an example to demonstrate how the three arguments are combined to obtain a tighter parallel lower bound.

2.1 Fan-in Argument

Parallel computations are frequently represented by binary trees or, more generally, binary forests (i.e., collection of trees), in which the roots correspond to results, the leaves (or external nodes) correspond to input arguments, the (internal) nodes correspond to operations performed on previously available results, arguments, or constants and the height of the tree orresponds to the number of parallel operations performed. By this analogy, properties about binary trees can be immediately transformed into properties about parallel computations. The following fact about binary trees is well known.

Fact. The height of a binary tree with n leaves is at least $\lceil \log n \rceil$.[11] [12]

Using this fact, a simple relationship between the number of input arguments and the minimal number of parallel operations required can be established.

Lemma 2.1.1 At least $\lceil \log n \rceil$ parallel operations are required to compute a result which depends on n arguments.

The technique, generally called the fan-in argument, was first adopted by Winograd [1965, 1967] to prove lower

[11] Throughout this thesis all logarithms are taken base 2, unless otherwise stated.

[12] Given any number x, $\lceil x \rceil$ denotes the smallest integer $\geq x$, and $\lfloor x \rfloor$ denotes the greatest integer $\leq x$.

bounds for logical circuits. Variations of this lemma are either stated [Kuck & Muraoka 1974], proved [Munro & Paterson 1973], or simply used implicitly in many papers which deal with parallel computations. To illustrate the use of this lemma, we apply it to show some well-known fundamental results. The first is the summation of several numbers.

<u>Lemma 2.1.2</u> At least $\lceil \log n \rceil$ parallel operations are required to compute the sum of n numbers,

$$E = x_1 + x_2 + \dots + x_n,$$

in $F(x_1, x_2, \dots, x_n)$ given $F \cup \{x_1, x_2, \dots, x_n\}$. Furthermore, this bound can be attained.

<u>Proof</u>. Since E depends on n input arguments, by Lemma 2.1.1, at least $\lceil \log n \rceil$ parallel operations are required to compute E.

We construct a computation tree for E such that $\{ x_i \mid 1 \le i \le n \}$ are at the bottom level, $x_1 + x_2$, $x_3 + x_4$, ... are at the next level, $x_1 + x_2 + x_3 + x_4$, $x_5 + x_6 + x_7 + x_8$, ... are at the still next level, and so on, until the highest level is E. Obviously, the height of the tree is $\lceil \log n \rceil$. Thus, the parallel lower bound is attained. Q.E.D.

The parallel algorithm given in the above lemma is called the <u>log-sum algorithm</u> in the ILLIAC IV literature. Similar parallel algorithm, called the <u>log-product algorithm</u>, can be designed for the multiplication of several numbers.

Inner products are common operations in linear algebra. Matrix multiplication, for example, can be regarded as the computation of many inner products simultaneously. The following theorem gives a parallel lower bound for computing inner products, and consequently, also for computing the product of two matrices.

Theorem 2.1.3 At least $\lceil \log n \rceil + 1$ parallel operations are required to compute the inner product,

$$E = x_1 y_1 + x_2 y_2 + \dots + x_n y_n,$$

in $F[x_1, y_1, \dots, x_n, y_n]$ given $F \cup \{x_i\} \cup \{y_i\}$. Furthermore, this bound can be attained.

Proof. Since E depends on 2n input arguments, by Lemma 2.1.1, at least $\lceil \log (2n) \rceil = \lceil \log n \rceil + 1$ parallel operations are required.

The parallel algorithm to achieve the parallel lower bound is quite similar to the usual one. The n products, $x_i y_i$, $1 \le i \le n$, are computed in one parallel multiplication and the products are then summed in $\lceil \log n \rceil$ parallel additions by the log-sum algorithm. Q.E.D.

Corollary 2.1.4 At least $\lceil \log n \rceil + 1$ parallel operations are required to compute matrix-matrix, matrix-vector, or vector-matrix multiplications of order n. Furthermore, this bound can be attained.

Relationships between the number of sequential operations required and the number of parallel operations required can be established by first observing the following fact about binary trees.

Fact. The number of internal nodes in a binary tree is one less than the number of leaves.

Using this fact and Lemma 2.1.1, and also recalling the correspondence between binary trees and parallel computations, the following lemma follows immediately.

Lemma 2.1.5 At least $\lceil \log(n+1) \rceil$ parallel operations are required to compute a result which requires at least n sequential operations. (13)

Provided that the sequential lower bound for certain problem is known, the above lemma immediately yields a parallel lower bound. In fact, an alternative proof for Theorem 2.1.3 using this idea appears in [Borodin & Munro 1975].

The following lemma due to Kedem and Kirkpatrick [1974] gives the sequential lower bound for the summation of n numbers.

[13] It is interesting to note the dual of Lemma 2.1.5:

Lemma 2.1.5' At most 2^n-1 sequential operations are required to compute a result which can be computed in n parallel operations.

The lemma establishes a relationship between parallel upper bounds and sequential upper bounds, but we do not explore this direction any further.

Lemma 2.1.6 At least n-1 additions are required to compute

$$E = x_1 + x_2 + \ldots + x_n,$$

in $F(x_1, x_2, \ldots, x_n)$ given $F \cup \{x_1, x_2, \ldots, x_n\}$.

Proof. Cf. Borodin and Munro [1975, pp. 12-15]. Q.E.D.

Lemma 2.1.2 states that at least $\lceil \log n \rceil$ parallel operations are required to compute E. But on many occasions we are more interested in the specific types of operations involved, i.e. additions, subtractions, multiplications, divisions, or Boolean operations. On a SIMD parallel computer, although many operations can be performed simultaneously, only one type of operation can be performed at any time. This point is illustrated by proving a stronger version of Lemma 2.1.2.

Theorem 2.1.7 At least $\lceil \log n \rceil$ parallel additions are required to compute

$$E = x_1 + x_2 + \ldots + x_n,$$

in $F(x_1, x_2, \ldots, x_n)$ given $F \cup \{x_1, x_2, \ldots, x_n\}$.

Proof. The theorem is an immediate consequence of Lemma 2.1.5 and Lemma 2.1.6. Q.E.D.

2.2 Growth Argument

The fan-in argument as illustrated in the previous section is a ubiquitous yet simple technique for proving parallel lower bounds. However, there are severe limitations in using it alone. Very often the bounds obtained from it are too trivial to be useful. For example, the fan-in argument can be used to prove the following result:

At least log 1 = 0 parallel operation is required to compute x^n.

The result, however, is hardly useful.

The growth argument can complement the fan-in argument for obtaining more useful parallel lower bounds. Unlike the fan-in argument, it establishes a relationship between the growth of a result and the minimal number of parallel operations required. We proceed to illustrate the growth argument by proving the following lemma.

Lemma 2.2.1 At least $\lceil \log n \rceil$ parallel operations are required to compute x^n in $F(x)$ given $F \cup \{x\}$. Furthermore, this bound can be attained. (14)

Proof. The parallel lower bound is proved by induction on n, the degree of x^n. First, the parallel lower bound is obviously true for n = 1. Suppose that the lemma holds for n

(14) The sequential complexity of this problem is $\lceil \log n \rceil$ + O((log n)/(log log n)) [Knuth 1969]. This is one example of a problem where there is very little advantage in solving it on a parallel computer.

$\leq 2^k$, i.e., k parallel operations are required. Since only rational functions (including powers of x) of degree $\leq 2^k$ can be computed, then any function computed at the (k+1)-st parallel operation can not exceed degree 2^{k+1}. [15] Thus, the parallel lower bound holds for all n.

The parallel lower bound is attained by using the log-product algorithm. Q.E.D.

Theorem 2.2.2 At least $\lceil \log n \rceil$ parallel multiplications are required to compute x^n in F[x] given F ∪ {x}.

Proof. Since in the ring F[x], deg(f*g) = deg(f)+deg(g) and deg(f±g) ≤ {deg(f),deg(g)}. The only operation which can incur any increase in degree is multiplication. Q.E.D.

Note that the theorem is not true if the intermediate results are not confined to F[x]. In fact, Kung [1974] presents a parallel algorithm in F(x) given F ∪ {x} which computes x^n in 2 parallel divisions and $\lceil \log n \rceil$+2 parallel additions and no multiplication (Cf. example in Chapter 1).

[15] Any rational function f(x) in F(x) can be expressed as a formal quotient of two relatively prime polynomials f(x) = a(x)/b(x), and the degree of f is defined as deg(f) = max {deg(a),deg(b)}. By this definition the following inequality holds

$$\deg(f \text{ op } g) \leq \deg(f) + \deg(g),$$

for any rational functions f and g and where op is any of the operations +, -, *, and /.

2.3 Substitution Argument

The results obtained from the fan-in argument and the growth argument in the last two sections are very simple ones. For slightly more complicated problems the techniques are deemed insufficient. However, a simple idea which can boost the power of the previous techniques works as follows:

> Let A be any algorithm for solving a class of problems, P. If we substitute some of the input arguments of the algorithm A with specific values, then A is transformed into another algorithm A' for solving a class of reduced problems, P'. But, since A' does not use more operations than A, any lower bound for P' must also be a lower bound for P.

Thus, provided that the parallel lower bound for P' is known, it immediately yields a parallel lower bound for P. The technique, generally called the substitution argument, is a widely used technique for proving sequential lower bounds [Borodin & Munro 1975]. But the technique does not rely on any particular computation model, whether sequential or parallel. It primarily establishes a relationship between two classes of problems. To illustrate the technique, we use it to prove the parallel lower bound for multiplying n numbers.

Theorem 2.3.1 At least $\lceil \log n \rceil$ parallel multiplications are required to compute

$$E = x_1 * x_2 * \ldots * x_n,$$

in $F[x_1, x_2, \ldots, x_n]$ given $F \cup \{x_1, x_2, \ldots, x_n\}$. Furthermore, this bound can be attained.

Proof. By the substitution argument, let $x_i \leftarrow x$, $1 \le i \le n$. The problem is then reduced to evaluating x^n which, by Theorem 2.2.2, requires at least $\lceil \log n \rceil$ parallel multiplications.

The parallel lower bound is attained by using the log-product algorithm. Q.E.D.

In order to obtain tight parallel lower bounds, one must frequently combine all three techniques. To demonstrate this point, we will prove a parallel lower bound for evaluating Boolean recurrence equations which will be needed in Chapter 4.

Consider the following first-order Boolean recurrence problem. Given any 2n+1 Boolean constants $\{a_0, a_1, \ldots, a_n, b_1, \ldots, b_n\}$, evaluate the Boolean variables $\{ x_i \mid 0 \le i \le n \}$ defined by

$$x_0 = a_0,$$
$$x_i = a_i + b_i \bullet x_{i-1}, \; 1 \le i \le n.$$

where the operators '+' and '•' are naturally interpreted as being Boolean addition and multiplication, respectively. First x_i can be expanded as follows:

$$x_i = a_i + (b_i) a_{i-1} + \ldots + (b_i b_{i-1} \ldots b_2) a_1 + (b_i b_{i-1} \ldots b_1) a_0.$$

We observe that x is a sum of i+1 products. An obvious parallel algorithm for evaluating $\{ x_i \mid 0 \le i \le n \}$ is first to form the products in $\lceil \log(n+1) \rceil$ parallel Boolean multiplications. This is possible because no product has more than n+1 atoms and a Boolean version of the log-product algorithm will do. Next, we can form all the sums of products in $\lceil\log(n+1)\rceil$ parallel Boolean additions by a Boolean version of the log-sum algorithm. [16]

Since there are 2n+1 input arguments, by the fan-in argument, at least $\lceil\log(2n+1)\rceil \leq \lceil\log(n+1)\rceil+1$ parallel Boolean operations are required. On the other hand, since each term in any x_i contains no more than n+1 atoms, by the growth argument, at least $\lceil\log(n+1)\rceil$ parallel Boolean operations are required. Thus, neither the fan-in argument nor the growth argument can show that the number of Boolean operations required for the bove parallel algorithm is optimal.

<u>Theorem 2.3.2</u> At least $\lceil\log(n+1)\rceil$ parallel Boolean additions and $\lceil\log(n+1)\rceil$ parallel Boolean multiplications are required to evaluate $\{ x_i \mid 0 \leq i \leq n \}$ on SIMD parallel computers [17], if no other Boolean operations except addition and multiplication are permitted. Furthermore, this bound can be attained.

[16] This parallel algorithm uses $O(n^3)$ processors but another parallel algorithm based on the binary splitting technique uses only n+1 processors [Kogge 1974].

[17] On MIMD parallel computers, evaluation of Boolean recurrence equations has been studied by Brent [1970], and evaluation of general Boolean expressions by Preparata and Muller [1976].

Proof. First, substitute all the b_i's by the Boolean value 1 whereby evaluating x_n is reduced to evaluating $a_n + a_{n-1} + \ldots + a_1 + a_0$. This, by the fan-in argument, requires at least $\lceil \log(n+1) \rceil$ parallel Boolean operations. Furthermore, by the growth argument, since any Boolean multiplication would increase the number of atoms in a term, all of these operations must be Boolean additions.

Next, substitute all the a_i's, except a_0, by the Boolean value 0 whereby evaluating x_n is reduced to evaluating the single term $b_n b_{n-1} \ldots b_1 a_0$. By a similar reasoning, this requires at least $\lceil \log(n+1) \rceil$ parallel Boolean multiplications, since any Boolean addition would increase the number of terms.

Finally, by the definition of SIMD computation, Boolean additions and multiplications can not be performed simultaneously. Thus, the parallel lower bound is proved.

Q.E.D.

Note first that the fan-in argument, growth argument and substitution argument are all employed in the above proof. Also, a restriction is placed on the types of Boolean operations permitted. In fact, the last theorem is contrived strictly for illustration purposes. If arbitrary Boolean operations can be used, then Boolean additions or Boolean multiplications are not needed at all. For example, the NAND (or NOR) operation can be used exclusively to express any Boolean function.

CHAPTER THREE

Parallel Polynomial Arithmetic

In this chapter we consider univariate and multivariate polynomials with coefficients in F. On the one hand, polynomials may be viewed as being functions which can be evaluated, and on the other, as being algebraic entities which may themselves be added, subtracted, multiplied and divided. Both points of view are considered in this chapter.

The evaluation of polynomials is considered in Section 3.1. A tight parallel lower bound for this problem is given. From the proof it follows that a very early parallel polynomial evaluation algorithm designed by Estrin is optimal on SIMD parallel computers.

In Section 3.2 some simple polynomial arithmetic including addition, subtraction and multiplication is considered. Parallel algorithms for solving these problems are presented and their optimality proved. Most of these results are trivial and are included primarily for future references.

Division of polynomials is considered in Section 3.3. An efficient parallel algorithm for solving this problem is presented. A parallel lower bound for polynomial division, which is of the same complexity as the parallel lower bound for polynomial multiplication, is derived.

In the last section, Section 3.4, we consider the evaluation of elementary symmetric functions, which finds its application in parallel polynomial modular arithmetic in Chapter 5. An efficient parallel algorithm for evaluating all elementary symmetric functions is presented and its optimality proved.

3.1 Evaluation of Polynomials

Given a polynomial of degree n

$$f(x) = f_0 + f_1 x + \dots + f_n x^n,$$

the optimal sequential algorithm for evaluating f(x) is the well-known Horner's rule which requires n additions and n multiplications. In fact, this algorithm is uniquely optimal on sequential computers [Borodin 1971].

The first parallel algorithm for evaluating polynomials was designed by Estrin [1960]. It uses the binary splitting technique, namely, it first computes

$$f_0 + f_1 x,\ f_2 + f_3 x,\ \dots,\ x^2,$$

and then $(f_0+f_1x)+(f_2+f_3x)x^2,\ (f_4+f_5x)+(f_6+f_7x)x^2,\ \dots,\ x^4,$ $\dots,$ and so on.

This parallel algorithm requires $\lceil \log(n+1) \rceil$ parallel additions and $\lceil \log(n+1) \rceil$ parallel multiplications, assuming $\lfloor (n+1)/2 \rfloor$ processors are available.

Various other parallel algorithms are developed by Dorn [1962], Munro and Paterson [1973], Maruyama [1973] and Kung [1974]. Dorn's algorithm is essentially a generalization of Horner's rule. Munro and Paterson's algorithm and Maruyama's algorithm both assume a MIMD computational model.

In this section it is shown that, if division is not allowed, at least $\lceil \log(n+1) \rceil$ parallel additions and $\lceil \log(n+1) \rceil$ parallel multiplications are required to evaluate f(x) on SIMD parallel computers. Thus, on SIMD parallel computers, Estrin's oldest parallel algorithm is in fact optimal (in $F[x, f_0, f_1, \ldots, f_n]$). This is probably one good example of why a systematic approach to deriving parallel lower bounds is important.

<u>Theorem</u> <u>3.1.1</u> At least $\lceil \log(n+1) \rceil$ parallel additions and $\lceil \log(n+1) \rceil$ parallel multiplications are required to evaluate $f(x)$ in $F[x, f_0, f_1, \ldots, f_n]$ given $F \cup \{x, f_0, f_1, \ldots, f_n\}$ on SIMD parallel computers.

<u>Proof</u>. Using the substitution argument, first consider the case $x \leftarrow 1$. The problem is reduced to the summation of the n+1 indeterminates, $f_0 + f_1 + \ldots + f_n$. By Theorem 2.1.7, at least $\lceil \log(n+1) \rceil$ parallel additions are required to do the summation.

Next, consider the case $f_n \leftarrow x$ and $f_i \leftarrow 0$, for all $i < n$. The problem is now reduced to that of evaluating the power x^{n+1}. By Theorem 2.2.2, at least $\lceil \log(n+1) \rceil$ parallel multiplications are required to do the evaluation in $F[x, f_0, f_1, \ldots, f_n]$.

Finally, by the definition of SIMD computation, parallel additions and multiplications can not be performed simultaneously. Thus, at least $\lceil \log(n+1) \rceil$ parallel additions and $\lceil \log(n+1) \rceil$ parallel multiplications are required. Q.E.D.

Unlike Horner's rule for sequential computation, however, Estrin's parallel algorithm is not uniquely optimal. For example, a different parallel algorithm with the same operation bound can be obtained as follows: First, all the n+1 terms of $f(x)$ can be com uted in $\lceil \log(n+1) \rceil$

parallel multiplications and then the terms can be added in $\lceil \log(n+1) \rceil$ parallel additions.

In addition, if computation is not restricted to $F[x, f_0, f_1, \ldots, f_n]$, then a parallel algorithm which requires 6 parallel multiplications / divisions and $2\lceil \log n \rceil + 6$ parallel additions can be designed [Kung 1974].

3.2 Simple Polynomial Arithmetic

In this section we discuss some simple polynomial operations, including addition, subtraction and multiplication of polynomials. Some of the results are trivial, and are included for future reference and for the sake of completeness.

The simplest operations on polynomials are multiplication and division of a polynomial, say f, by a (positive) power of x, i.e., $f*x^k$ and f/x^k. These operations can be regarded as being "scaling" of polynomials, just as integers can be scaled by powers of b, where b is the base (or radix) [Cf. Aho et al 1974]. The highest (n+1)-st coefficients of $f*x^k$ are exactly the same as those of f, whereas the remaining k lower order coefficients are all zeros. The coefficients of the quotient of f/x^k are the same as the first n-k+1 coefficients of f and the coefficients of the remainder of f/x^k are exactly the lowest k terms of f.

There may be hidden costs associated with the multiplication and division by powers of x (as far as programming is concerned) but there are no explicit arithmetic operations performed. Indeed, when polynomials are scaled as part of a more complex process, the cost of this scaling will normally be negligible and can usually be ignored.

Multiplication and division of a polynomial by a constant polynomial are also easy to perform. The multiplication (or division) of a polynomial of degree n,

$$f(x) = f_0 + f_1 x + \ldots + f_n x^n,$$

by a constant polynomial c is the multiplication (or division) of its coefficients by c.

Theorem 3.2.1 The multiplication (or division) of the polynomial f by a constant polynomial c can be computed in 1 parallel multiplication (or division) on a SIMD parallel computer with n+1 processors. Furthermore, the parallel operation bound is optimal.

Proof. The parallel operation bound and processor requirements are obvious.

Since $f_i * c$ (or f_i / c) is not in $F \cup \{f_i\} \cup \{c\}$, at least one parallel operation is required. Thus, the parallel operation bound is optimal. Q.E.D.

The addition (or subtraction) of two polynomials

$$f(x) = f_0 + f_1 x + \ldots + f_n x^n$$

and
$$g(x) = g_0 + g_1 x + \ldots + g_n x^n$$

is the addition (or subtraction) of their corresponding coefficients.

Theorem 3.2.2 The addition (or subtraction) of the polynomials f and g can be computed in 1 parallel addition (or subtraction) on a SIMD parallel computer with n+1 processors. Furthermore, the parallel operation bound is optimal.

Proof. The parallel operation bound and processor requirements are obvious.

Since $f_i + g_i$ (or $f_i - g_i$) is not in $F \cup \{f_i\} \cup \{g_i\}$, at least one parallel operation is required. Thus, the parallel operation bound is optimal. Q.E.D.

The multiplication of general polynomials is also quite straightforward. Given two polynomials of degree m and n

$$f(x) = f_0 + f_1 x + \dots + f_m x^m$$

and $$g(x) = g_0 + g_1 x + \dots + g_n x^n,$$

their product (often called the <u>convolution product</u>) is a polynomial of degree m+n

$$h(x) = h_0 + h_1 x + \dots + h_{m+n} x^{m+n},$$

where $h_i = f_0 g_i + f_1 g_{i-1} + \dots + f_i g_0$, $0 \le i \le m+n$. Note that f_k (or g_k) is tacitly assumed to be zero if $k < 0$ or $k > m$ (or $k > n$).

Fast sequential polynomial multiplication algorithms are based on the Fast Fourier Transform (FFT) and are of complexity O(n log n) [Aho et al 1974, Borodin & Munro 1975]. These algorithms are much more complicated than the classical multiply-and-add algorithm. It is therefore important to note that on parallel computers the classical polynomial multiplication algorithm is indeed optimal.

<u>Theorem 3.2.3</u> If $m \ge n$, then the product of the two polynomials f and g can be computed in 1 parallel multiplication and $\lceil \log(n+1) \rceil$ parallel additions on a SIMD parallel computer with $(m+1)(n+1)$ processors. Furthermore, the parallel operation bound is optimal.

Proof. First compute the products, $f_i g_j$, $0 \le i \le m$, $0 \le j \le n$, in one parallel multiplication with $(m+1)(n+1)$ processors. Then compute the coefficients $h_i = f_0 g_i + f_1 g_{i-1} + \ldots + f_i g_0$ by the log-sum algorithm. Since $f_k g_{i-k} = 0$ whenever $i - k > n$, there are at most $n+1$ nonzero summands in each h_i, $0 \le i \le m+n$. Thus, the coefficients can be added in $\lceil \log(n+1) \rceil$ parallel additions. Obviously, $(m+1)(n+1)$ processors are sufficient for doing the summations.

Note that h_n may be viewed as being the inner product of two $(n+1)$-vectors. By Theorem 2.1.3, at least $\lceil \log(n+1) \rceil + 1$ parallel operations are required to compute just h_n. Thus, the parallel operation bound is optimal.

Q.E.D.

3.3 Division of Polynomials

Given two polynomials, one of degree $m+n$

$$f(x) = f_0 + f_1 x + \dots + f_{m+n} x^{m+n}$$

and one of degree n, $n \geq 1$, [18]

$$g(x) = g_0 + g_1 x + \dots + g_n x^n,$$

there exist unique polynomials q and r such that

$$f = q * g + r, \ \deg(r) < \deg(g),$$

where q is a polynomial of degree m,

$$q(x) = q_0 + q_1 x + \dots + q_m x^m.$$

The polynomial q is called the quotient and denoted by $\lfloor f/g \rfloor$, and r is called the remainder and denoted by f mod g. For any integer $k \geq 0$, the quotient $\lfloor x^{k+n}/g \rfloor$ is called the reciprocal of g of degree k. [19]

The existence of q and r can be proved constructively by 'long division', i.e., a multiple of g is successively subtracted from f until the final remaining polynomial has degree less than that of g. If there are $n+1$ processors available, then the number of parallel operations required by long division is proportional to $m+1$, the number of coefficients in the quotient q. Note particularly that explicit division of the coefficients is performed only by

[18] Since we have discussed division by constant polynomials and powers of x in Section 3.2, we assume that the divisors are non-trivial and of degree ≥ 1 throughout this section.

[19] This definition is slightly more general than that given in [Aho et al 1974].

g_n; so that if g is monic, no division is actually performed.

A faster parallel polynomial division algorithm can be obtained from the product of f and the reciprocal of g [20]. First, observe that it suffices to compute the quotient q, since the remainder r can be obtained in one polynomial multiplication and one polynomial subtraction. Next, only the first m+1 coefficients of f and g are relevant to the computation of q. More precisely, the quotient q is given by the following formula:

$$q = \lfloor(\lfloor f/x^n\rfloor * \lfloor x^{m+n}/g\rfloor)/x^m\rfloor,$$ [21]

i.e., the coefficients of q are the first m+1 coefficients of the product obtained by multiplying the first m+1 coefficients of f with the reciprocal of g of degree m.

<u>Examples</u>. Let g = 2x+1. Denote the reciprocal of g of degree k by $u^{(k)}$. Then, $u^{(0)} = 1/2$, $u^{(1)} = x/2-1/4$, $u^{(3)} = x^3/2-x^2/4+x/8-1/16$.

1. For f = 3x+1, let

   $$v = \lfloor f/x\rfloor * u^{(0)} = 3*(1/2) = 3/2.$$

   Then

[20] In fact, if the divisor will be used more than once (such as in the case of modular arithmetic), then it is more efficient to compute the reciprocal first and save it for future use. This is called <u>precomputed</u> <u>division</u> in [Kung 1973].

[21] This formulation is equivalent to those of [Kung 1973] and [Borodin & Munro 1975].

$q = \lfloor v/x^0 \rfloor = 3/2.$

2. For $f = 2x^2+3x+3$, let

$$v = \lfloor f/x \rfloor * u^{(1)} = (2x+3)(x/2-1/4) = x^2+x-3/4.$$

Then

$$q = \lfloor v/x \rfloor = x+1.$$

3. For $f = x^4+5x^2+3x+1$, let

$$v = \lfloor f/x \rfloor * u^{(3)} = (x^3+5x+3)(x^3/2-x^2/4+x/8-1/16)$$

$$= x^6/2-x^5/4+21x^4/8+3x^3/16-x^2/8+x/16-3/16.$$

Then

$$q = \lfloor v/x^3 \rfloor = x^3/2-x^2/4+21x/8+3/16.$$

The reciprocal $\lfloor x^{m+n}/g \rfloor$ of degree m can be computed iteratively as follows [Aho et al 1974]:

The reciprocal of degree 0 is given trivially by

$$\lfloor x^n/g \rfloor = 1/g_n.$$

Having obtained u, the reciprocal of degree $k=(2^{i-1})-1$, then, u', the reciprocal of degree $2k+1 = (2^i)-1$, can be computed by means of

$$v \leftarrow 2ux^{3k+1} - u^2 \lfloor g/x^{n-2k-1} \rfloor,$$

$$u' \leftarrow \lfloor v/x^{2k} \rfloor.$$

<u>Examples</u>. Let $g = x^7-x^6+x^5+2x^4-x^3-3x^2+x+4$. Denote the reciprocal of g of degree k by $u^{(k)}$. Then,

$$u^{(0)} = 1/g_7 = 1.$$

To find $u^{(1)}$:

$$v = 2u^{(0)}x^1 - (u^{(0)})^2 \lfloor g/x^6 \rfloor$$

$$= 2x - (x-1) = x+1.$$

$$u^{(1)} = \lfloor v/x^0 \rfloor = v = x+1.$$

To find $u^{(3)}$:

$$v = 2u^{(1)}x^4 - (u^{(1)})^2\lfloor g/x^4\rfloor$$

$$= 2(x+1)x^4 - (x+1)^2(x^3-x^2+x+2)$$

$$= x^5+x^4-3x^2-5x-2$$

$$u^{(3)} = \lfloor v/x^2\rfloor = x^3+x^2-3.$$

To find $u^{(7)}$:

$$v = 2u^{(3)}x^{10} - (u^{(3)})^2\lfloor g/x^0\rfloor$$

$$= x^{13}+x^{12}-3x^{10}-4x^9+3x^8+15x^7+12x^6$$

$$-42x^5-34x^4+39x^3+51x^2-9x-36.$$

$$u^{(7)} = \lfloor v/x^6\rfloor = x^7+x^6-3x^4-4x^3+3x^2+15x+12.$$

And so on.

Theorem 3.3.1 The reciprocal of g of degree m can be computed in $O(\lceil\log(m+1)\rceil, \lceil\log(n+1)\rceil)$ parallel additions / subtractions and $O(\lceil\log(m+1)\rceil)$ parallel multiplications / divisions on a SIMD parallel computer with $(m+1)(n+1)$ processors.

Proof. The iterative formula for computing the reciprocals of g, at each stage, entails two polynomial multiplications and one polynomial subtraction. Thus, the number of parallel operations $T(2^k=m+1)$ required to compute the the reciprocal satisfies, at the i-th stage,

$$T(2^i) \le T(2^{i-1}) + 2 \text{ multiplications} \tag{1a}$$
$$+ 2\lceil\log(n+1)\rceil \text{ additions} + 1 \text{ subtraction.}$$

In particular,

$$T(2^0) = 1 \text{ division.} \tag{1b}$$

From the recurrence relation (1), we conclude that the reciprocal of degree m can be computed in

$O(\lceil \log(m+1)\rceil \lceil \log(n+1)\rceil)$ additions / subtractions and $O(\lceil \log(m+1)\rceil)$ multiplications / divisions.

Within the parallel algorithm, the process using the greatest number of processors is polynomial multiplication. At the i-th stage, it is easy to observe that at most $2^i(n+1)$ processors (where $2^i \le m+1$) are used. Thus, $(m+1)(n+1)$ processors are sufficient for the entire computation. Q.E.D.

Corollary 3.3.2 The quotient q and remainder r of f divided by g can be computed in $O(\lceil \log(m+1)\rceil \lceil \log(n+1)\rceil)$ parallel operations on a SIMD parallel computer with $(m+1)(n+1)$ processors.

We now derive a parallel lower bound for polynomial division. Without loss of generality, assume that *the divisor g is monic*. From the identity $f = q * g + r$, it then follows that the coefficients of q are given by the linear recurrence equation:

$$\begin{aligned} q_m &= f_{m+n} \\ q_{m-1} &= f_{m+n-1} - g_{n-1} q_m \\ &\ldots \\ q_{m-i} &= f_{m+n-i} - g_{n-1} q_{m-i+1} - \ldots, \ i \le m. \end{aligned}$$

Observe that q_{m-i}, $0 \le i \le m$, is a polynomial of degree i in g_{n-1}. In particular, q_0 is a polynomial of degree m in q_{n-1}.

Theorem 3.3.3 At least $\lceil \log(m+1)\rceil$ parallel additions and $\lceil \log(m+1)\rceil$ parallel multiplications are required to compute the quotient q in $F[f_0, \ldots, f_{m+n}, g_0, \ldots, g_{n-1}]$

given $F \cup \{f_i\} \cup \{g_i\}$ on SIMD parallel computers.

Proof. [22] Since q_0 is a polynomial of degree m, by Theorem 3.1.1, to evaluate it alone requires at least $\lceil \log(m+1) \rceil$ parallel additions and $\lceil \log(m+1) \rceil$ parallel multiplications in $F[f_0, \ldots, f_{m+n}, g_0, \ldots, g_{n-1}]$. Q.E.D.

Note that the parallel lower bound can be attained only when g is of the first degree. In Chapter 5, however, we give a more efficient parallel algorithm (which depends on results yet to follow) for all n. The parallel algorithm given there is not practical since it assumes exponentially bounded parallelism. However, in the case of unbounded parallelism at least, we will have shown that polynomial multiplication and division are of the same parallel complexity.

[22] An alternative proof can be given by using the substitution argument.

3.4 Elementary Symmetric Functions

Let $s(n,k)$, $0 \leq k \leq n$, be the elementary symmetric function of degree k in n indeterminates $\{ y_i \mid 1 \leq i \leq n \}$. The elementary symmetric functions $\{ s(n,k) \mid 0 \leq k \leq n \}$ are formally defined to satisfy:

$$\begin{aligned} h(x) &= (x+y_1)(x+y_2)\ldots(x+y_n) \\ &= x^n + s(n,1)x^{n-1} + \ldots + s(n,n-1)x + s(n,n). \end{aligned}$$

(Note that $s(n,0)$ is always 1 regardless of the value of n.) Each $s(n,k)$ is the sum of all k-combinations (or products) formed from the n indeterminates. Therefore the number of products in $s(n,k)$ is $C(n,k) = n!/k!(n-k)!$. Moreover, $s(n,\lfloor n/2 \rfloor)$ has the greatest number of products, $C(n,\lfloor n/2 \rfloor) = O(2^n)$ products. Therefore, if the elementary symmetric functions are computed as the summation of products, then at least $O(n)$ parallel opeations are required, which on parallel computers is deemed inefficient.

A direct expansion of $h(x)$ leads to a more efficient parallel algorithm. If h is expanded by forming products of 2 factors, then products of 4 factors, and so on [23], then, using the polynomial multiplication method discussed in Section 2, the number of parallel operations required is $O(\log^2 n)$ on a parallel computer with n^2 processors. In fact, this parallel operation bound is the best previously reported result [Goyal 1975]. In Theorem 3.4.2 below, we

[23] Products formed in this manner are called supermoduli in [Moenck & Borodin 1972].

give a parallel algorithm which requires only $O(\log n)$ parallel operations.

<u>Lemma 3.4.1</u> Given n, the set of powers $\{x^2, x^3, \ldots, x^n\}$ can be computed in $\lceil \log n \rceil$ parallel multiplications on a SIMD parallel computer with $\lfloor n/2 \rfloor$ processors.

<u>Proof</u>. [Borodin & Munro 1975, p. 128]. Q.E.D.

The following theorem, due to Csanky [1974], gives an efficient parallel algorithm for computing the elementary symmetric functions. In his development, however, Ĉsanky ignores the cost of computing powers of the Fourier points.

<u>Theorem 3.4.2</u> The elementary symmetric functions $\{ s(n,k) \mid 0 \le k \le n \}$ of n indeterminates $\{ y_i \mid 1 \le i \le n \}$ can be computed in $F[y_1, y_2, \ldots, y_n]$ given $F \cup \{y_1, y_2, \ldots, y_n\}$ in $\lceil \log(n+1) \rceil + 1$ parallel additions, $\lceil \log n \rceil + 1$ parallel multiplications and 1 parallel scalar division on a SIMD parallel computer with $(n+1)^2$ processors. Furthermore, the parallel operation bound is optimal.

<u>Proof</u>. First, evaluate $h(x)$ and $\{x^2, x^3, \ldots, x^n\}$ at the n+1 Fourier points $\{w^i\}$, $0 \le i \le n$, where w is a primitive (n+1)-st root of unity. Obviously, this can be done in 1 parallel addition ($n(n+1)$ processors) and $\lceil \log n \rceil$ parallel multiplications ($2(n+1)\lfloor n/2 \rfloor$ processors). Next, the elementary symmetric functions are obtained from the following formula:

$$s(n,k) = \left[\sum_{i=0}^{n} h(w^i) * w^{-i(n-k)} \right] / (n+1), \quad 0 \le k \le n.$$

This can be computed in 1 parallel multiplication $((n+1)^2$ processors), $\lceil\log(n+1)\rceil$ parallel additions $((n+1)\lfloor(n+1)/2\rfloor$ processors) and 1 parallel scalar division ($n+1$ processors).

Since $s(n,1) = y_1 + y_2 + \cdots + y_n$, by Theorem 2.1.7, at least $\lceil\log n\rceil$ parallel additions are required. Similarly, since $s(n,n) = y_1 * y_2 * \cdots * y_n$, by Theorem 2.3.1 at least $\lceil\log n\rceil$ parallel multiplications are required. Thus, by the definition of SIMD computation, the optimality follows.

Q.E.D.

Note that the above parallel algorithm is based on discrete Fourier transforms which are special cases of evaluation and interpolation (Cf. Chapter 5) since the Fourier points $\{w^i\}$, $0 \le i \le n$, are special. Although discrete Fourier transforms in general require $2\lceil\log(n+1)\rceil$ parallel operations to compute (Cf. Section 3.1 on evaluation of polynomials), the evaluation and interpolation in the above theorem are made especially easy due to the fact that $h(x)$ is already factorized and that the powers of $\{w^i\}$ can be computed before the evaluation of $s(n,k)$ begins.

In addition, the above algorithm is not suitable for sequential computation, since it requires $O(n^2)$ sequential operations. The best sequential algorithm requires only $O(n \log^2 n)$ operations [Borodin & Munro 1975].

CHAPTER FOUR

Parallel Integer Arithmetic

Denote an n-digit, base-b (where $b \geq 2$) integer x by

$$x = \sum_{i=1}^{n} x(i) * b^{n-1}, \; 0 \leq x(i) \leq b-1.$$

The integer x given in the above form is said to be in positional notation.

In this chapter we study the parallel complexity of performing the following processes on SIMD parallel computers:

1) addition or subtraction of n-digit integers, giving an n-digit result plus a possible carry;
2) multiplication of an m-digit integer by an n-digit integer, giving an (m+n)-digit product;
3) division of an (m+n)-digit integer by an n-digit integer, giving an (m+1)-digit quotient and an n-digit remainder.

We assume that the following (binary) operations are primitive:

1) addition or subtraction of 1-digit integers, giving a 1-digit result and a carry;
2) multiplication of a 1-digit integer by another 1-digit integer, giving a 2-digit product;
3) Boolean addition (OR-ing) or multiplication (AND-ing) of two bits, giving a 1-bit result.

Thus, for example, if x and y are 1-digit integers, then it is assumed that addition yields the values

$$s = (x + y) \bmod b,$$

and $$c = \lfloor (x + y) / b \rfloor.$$

In Section 4.1, a parallel algorithm for performing integer addition is considered. The concept of pseudo-positional integers (integers whose digits are bounded by 2b-2 rather than b-1) is introduced. Using this concept, integer addition is reduced to conversion of integer from pseudo-positional to positional notation.

In Section 4.2, a parallel algorithm for performing integer multiplication is described. Pseudo-addition is crucial to the design of an efficient parallel integer multiplication algorithm.

In Section 4.3, parallel algorithms for performing integer division are considered. A tradeoff between the number of parallel operations required and the number of processors used is observed.

For all problems efficient or asymptotically optimal parallel algorithms are given.

4.1 Addition and Subtraction of Integers

Given two non-negative n-digit integers in positional notation,

$$x = \sum_{i=1}^{n} x(i) * b^{n-i},\ 0 \le x(i) \le b-1,$$

$$y = \sum_{i=1}^{n} y(i) * b^{n-i},\ 0 \le y(i) \le b-1,$$

their sum is an n-digit result plus a possible carry

$$s = \sum_{i=0}^{n} s(i) * b^{n-i},\ 0 \le s(i) \le b-1.$$

Let c(i) denote the carry at the i-th position. Then,

$$s(n) = [\ x(n) + y(n)\ ] \bmod b, \tag{1a}$$

$$c(n) = \lfloor [\ x(n) + y(n)\ ] / b \rfloor, \tag{1b}$$

and for i = n-1, ..., 2, 1, s(i) can be computed iteratively by means of:

$$s(i) = [\ x(i) + y(i) + c(i+1)\ ] \bmod b, \tag{1c}$$

$$c(i) = \lfloor [\ x(i) + y(i) + c(i+1)\ ] / b \rfloor. \tag{1d}$$

The 0-th digit, s(0), is given by

$$s(0) = c(1). \tag{1e}$$

This iterative addition algorithm requires O(n) parallel operations and therefore is too inefficient on parallel computers.

On the other hand, since polynomial addition is quite cheap, the sum s can be computed as if x and y were polynomials in b: (24)

$$s = \sum_{i=1}^{n} w(i) * b^{n-1},$$

where $w(i) = x(i) + y(i)$, $1 \le i \le n$. Note, in particular, that the coefficients of the polynomial sum are bounded by $2b-2$ rather than $b-1$. This motivates the notion of pseudo-positional integers:

<u>Definition 4.1.1</u> Let w be a positive integer given by

$$w = \sum_{i=1}^{n} w(i) * b^{n-1}, \quad 0 \le w(i) \le 2b-2.$$

The integer w given in the above form is said to be in <u>pseudo-positional notation</u>.

Thus, addition of integers is reduced to the conversion of integers from pseudo-positional notation to positional notation. This switching of views is particularly fundamental in the design of efficient parallel addition and multiplication algorithms discussed in sebsequent sections. It is then this problem to which we now turn attention.

Since all pseudo-positional integers appearing in this chapter are obtained as intermediate results, it is assumed that both digits in the representation of each $w(i)$,

[24] As a matter of fact, all arithmetic on multiple-precision integers can be viewed as being polynomial arithmetic [Aho et al 1974, Borodin & Munro 1975]. However, the correct results must be recovered from the polynomial results by releasing the carries [Borodin & Munro 1975 pp. 90-91].

$$s'(i) = w(i) \bmod b,$$

$$c'(i) = \lfloor w(i) / b \rfloor, \ 1 \le i \le n,$$

are available without any additional cost. The iterative addition algorithm (1) given above then can be interpreted as being a conversion algorithm:

$$s(n) = s'(n), \qquad (2a)$$

$$c(n) = c'(n), \qquad (2b)$$

for $i = n-1, \ldots, 2, 1,$

$$s(i) = [\ s'(i) + c(i+1)\ ] \bmod b, \qquad (2c)$$

$$c(i) = c'(i) + \lfloor [\ s'(i) + c(i+1)\ ] / b \rfloor, \qquad (2d)$$

and $s(0) = c(1).$ (2e)

This conversion algorithm is still inherently sequential due to its iterative nature. The first key to a fas[illegible]rallel conversion algorithm is the observation that at [illegible]t one carry is possible in each digit position during the addition of positional integers [Knuth 1969, p. 231]. This observation is rephrased in more general form in the lemma below.

<u>Lemma 4.1.2</u> Given any pseudo-positional integer, during the conversion to positional notation, the carry at [illegible] position is either 0 or 1.

The vector of carries $\underline{c} = (c(1), c(2), \ldots, c(n))$ can therefore be viewed as being a Boolean vector.

The next observation is that, by formulae (2b) and (2d), $c'(i)$ implies $c(i)$, $1 \le i \le n$, but not conversely. In other words, the vector $\underline{c}'$ is Boolean and indicates a <u>carry</u>

generating condition. Furthermore, by formula (2d), a carry at the (i+1)-st position, i.e., $c(i+1) = 1$, will propagate to the next position if and only if

$$s'(i) + c(i+1) = s'(i) + 1 = b;$$

or, equivalently, if and only if

$$s'(i) = b-1.$$

Denote this *carry propagating* condition as a Boolean vector

$$p(i) = 1 \text{ iff } s'(i) = b-1, \ 1 \le i \le n-1.$$

Then, formulae (2b) and (2d) can be rewritten as:

$$c(n) = c'(n) \qquad (3a)$$

$$c(i) = c'(i) + p(i) \bullet c(i+1), \ 1 \le i \le n-1. \qquad (3b)$$

That is, by using two auxiliary Boolean vectors, the carries $\{ c(i) \mid 1 \le i \le n \}$ have been shown to satisfy a Booleah first-order recurrence equation.

Lemma 4.1.3 The conversion of an n-digit pseudo-positional integer to its positional notation can be computed in $2\lceil \log n \rceil$ parallel Boolean operations and 1 parallel(*) addition on a SIMD parallel computer with n processors.

Proof. By Theorem 2.3.2, the Boolean recurrence equation (3) can be computed in $2\lceil \log n \rceil$ parallel Boolean operations with n processors.

Once the carries $\{ c(i) \mid 2 \le i \le n \}$ are available, by formula (2c), the digits $\{ s(i) \mid 1 \le i \le n-1 \}$ can be computed in 1 parallel addition (n-1 processors). Q.E.D.

Theorem 4.1.4 The sum of two non-negative n-digit integers can be computed in 2 parallel additions and $2\lceil \log n \rceil$ parallel Boolean operations on a SIMD parallel computer with n processors. Furthermore, the parallel operation bound is asymptotically optimal.

Proof. By Theorem 3.2.2, the polynomial sum of two integers can be computed in 1 parallel addition (n processors). Since the polynomial sum is a pseudo-positional integer, by Lemma 4,1.3, it can be converted to positional notation in $2\lceil \log n \rceil$ parallel Boolean operations and 1 parallel addition.

Since s(0) is a function of the 2n inputs $\{ x(i), y(i) \mid 1 \le i \le n \}$, by the fan-in argument, at least $\lceil \log(2n) \rceil = \lceil \log n \rceil + 1$ parallel operations are needed to compute just s(0). Thus, the asymptotic optimality follows. Q.E.D.

It is important to note that although the techniques used in this section is purely number theoretical, the parallel addition algorithm developed above includes the carry look-ahead adder [Flores 1963], which is derived from switching function theory, as a special case (i.e., when b = 2).

NOTE: The above results can easily be extended to include the addition or subtraction of negative integers. For example, the _true complement_ (i.e., b's complement) of any n-digit integer can be computed in 1 parallel single-precision subtraction and 1 n-precision addition. [25] [26]

[25] The sign-digit of the b's complement of any integer must be interpreted and operated on as a binary digit.

[26] Overflow is determined by the usual rule that if the carry out of the sign-digit position and the carry out of the first digit position disagree, then an overflow occurs.

4.2 Multiplication of Integers

Given two non-negative integers, namely, an m-digit multiplicand

$$x = \sum_{i=1}^{m} x(i) * b^{m-i}, \; 0 \le x(i) \le b-1,$$

and an n-digit multiplier

$$y = \sum_{i=1}^{n} y(i) * b^{n-i}, \; 0 \le y(i) \le b-1,$$

their product is an (m+n)-digit result

$$w = \sum_{i=1}^{m+n} w(i) * b^{m+n-i}, \; 0 \le w(i) \le b-1.$$

The classical paper-and-pencil algorithm for computing w is first to compute the n partial products

$$p(j) = x * y(j), \; 1 \le j \le n.$$

Each partial product p(j) is then normally converted to positional notation

$$p(j) = \sum_{i=1}^{m} p(i,j) * b^{m-i}, \; 0 \le p(i,j) \le b-1.$$

Next, the sum of the scaled partial products yields

$$w = \sum_{j=1}^{n} p(j) * b^{n-j}.$$

Since frequent conversions to positional notation are expensive, the strategy should be to avoid the use of positional integers during the calculation of intermediate.

results, thereby postponing conversions to positional integers until the last stage of the computation. This consideration leads us to take a more serious look at pseudo-positional arithmetic.

Definition 4.2.1 Define the *pseudo-sum* of two pseudo-positional integers

$$x = \sum_{i=1}^{n} x(i) * b^{n-i},\ 0 \le x(i) \le 2b-2,$$

and $$y = \sum_{i=1}^{n} y(i) * b^{n-i},\ 0 \le y(i) \le 2b-2,$$

to be the pseudo-positional integer

$$s = \sum_{i=0}^{n} s(i) * b^{n-i},\ 0 \le s \le 2b-2,$$

where $s = x + y$.

The process of obtaining the pseudo-sum s is called *pseudo-addition*.

Algorithm P. (Pseudo-Addition)

```
P1. Set $s(0) <- 0$,
        $s(i) <- x(i) + y(i),\ 1 \le i \le n$.
P2. If $s(i) \le 2b-2,\ 0 \le i \le n$, then terminate.
P3. Otherwise, release one round of carries:
      (let $c'(i) = \lfloor s(i) / b \rfloor$
       and $s'(i) = s(i) \bmod b$)
      Set $s(n) <- s'(n)$,
```

$$s(i) \leftarrow s'(i) + c'(i+1),\ 0 \le i \le n-1.$$

Go to step P2.

Lemma 4.2.2 The pseudo-addition of two pseudo-positional integers can be computed in at most 3 parallel additions (the first of which may be double-precision addition) on a SIMD parallel computer with n+1 processors. Furthermore, if $b \ge 3$, only 2 parallel additions are sufficient.

Proof. At the end of step P1,

$$0 \le s(i) = x(i) + y(i) \le 4b-4.$$

Thus,

$$\begin{aligned} c'(i) = \lfloor s(i) / b \rfloor &\le 2, \text{ if } b = 2, 3, \\ &\le 3, \text{ if } b \ge 4. \end{aligned}$$

Or, equivalently,

$$\begin{aligned} c'(i) &\le b, \text{ if } b = 2, \\ &\le b-1, \text{ if } b \ge 3. \end{aligned}$$

We next examine the number of times the loop P2 - P3 must be repeated. The first time through P3 yields

$$\begin{aligned} s(i) \le (b-1) + c'(i+1) &\le 2b-1, \text{ if } b = 2, \\ &\le 2b-2, \text{ if } b \ge 3. \end{aligned}$$

Thus if $b \ge 3$, only one iteration of steps P2 - P3 is sufficient.

If $b = 2$, one more loop through P2 - P3 results in (note that $c'(i) \le 1 = b-1$)

$$s(i) \le (b-1) + (b-1) = 2b-2.$$

Thus, for $b = 2$, at most two iterations of loop P2 - P3 are sufficient.

Now since step P1 can be computed in 1 parallel addition (double-precision, according to the definition of primitive addition given previously), and step P3 can be computed in 1 parallel (single-precision) addition, the lemma follows. Q.E.D.

Note that the number of parallel operations required to compute the pseudo-sum is a constant regardless of how many digits the pseudo-positional [illegible] have. Thus, by using pseudo-addition, not only is the growth of the intermediate results during multiplication controlled, but also the cost of combining the partial products is reduced.

The fastest sequential integer multiplication algorithm is one given by Schoenhage and Strassen [1971] and is of complexity O(n log n log log n). Basically it uses the idea of discrete Fourier transforms. A disadvantage of the algorithm is that it is difficult to code and its benefits are realized only for n sufficiently large [Knuth 1969, Aho et al 1974].

The fastest previous known parallel integer multiplication algorithm is one given by Atrubin [1965] and is of parallel complexity O(n) on a linear iterative array with n modules [Knuth 1969]. The parallel integer multiplication algorithm given below is essentially a parallelized version of the classical paper-and-pencil algorithm. It differs primarily in that, at all intermediate steps, conversion is made to pseudo-positional rather than positional notation.

<u>Algorithm M</u>. (Multiplication of Non-negative Integers)

M1. Compute the digit-by-digit products

$\{ x(i) * y(j), \mid 1 \le i \le m, 1 \le j \le n \}$.

M2. Convert the n partial products

$$p(j) = \sum_{i=1}^{m} x(i) * y(j) * b^{m-i}, \quad 1 \le j \le n$$

to pseudo-positional notation,

$$p'(j) = \sum_{i=0}^{m} p'(i,j) * b^{m-i}, \quad 0 \le p'(i,j) \le 2b-2.$$

M3. Compute the pseudo-sum $w = \sum_{j=1}^{n} p'(j) * b^{n-j}$.

M4. Convert w to positional notation.

<u>Theorem 4.2.3</u> The product of an m-digit multiplicand and an n-digit multiplier can be computed in 1 parallel multiplication, $3\lceil \log n \rceil + 2$ parallel additions and $2\lceil \log(m+n) \rceil$ parallel Boolean operations on a SIMD parallel computer with mn processors. Furthermore, the parallel operation bound is asymptotically optimal.

<u>Proof</u>. It is obvious that step M1 can be computed in 1 parallel multiplication with mn processors.

In step M2, $0 \le x(i) * y(j) \le (b-1)^2 < b(b-1)$ implies that all carries are bounded by $b-2$. Therefore, after one round of carry releasing, all digits are bounded by $(b-1)+(b-2) = 2b-3 < 2b-2$ (i.e., after one round of carry releasing, the result is in pseudo-positional notation). Thus, step M2 can be computed in 1 parallel addition with mn processors.

In step M3, using a generalized log-sum algorithm, the pseudo-sum w can be computed in $\lceil \log n \rceil$ pseudo-additions. By Lemma 4.2.2, each pseudo-addition can be computed in at most 3 parallel additions. Thus, step M3 can be computed in a total of $3\lceil \log n \rceil$ parallel additions. Since each pseudo-positional integer has at most m+n digits and at most $\lfloor n/2 \rfloor$ pseudo-sums are formed at any time, a total of $(m+n)\lfloor n/2 \rfloor \le mn$ (if $m \ge n$) processors is sufficient for computing w.

Finally, the conversion of w to positional notation in step M4, by Lemma 4.1.3, can be computed in 1 parallel addition and $2\lceil \log(m+n) \rceil$ parallel Boolean operations (m+n processors).

The total operation count is therefore 1 parallel multiplication, $3\lceil \log n \rceil + 2$ parallel additions and $2\lceil \log(m+n) \rceil$ parallel Boolean operations with mn processors.

The asymptotic optimality is obvious, since the product depends on all m+n inputs. Q.E.D.

4.3 Division of Integers

Let a and c be any (m+n)-digit and n-digit positive integers, respectively. We wish to find a quotient q and a remainder r such that

$$a = q * c + r, 0 \leq r < c.$$

As usual, once a fast parallel multiplication algorithm is given, it can be used for the design of a fast parallel division algorithm as follows: First, it suffices to compute the quotient q only, since then r can be obtained by means of r = a - q * c. Second, let 1/c denote the "reciprocal" of c. Then finding q = a * (1/c) can be reduced to multiplication and reciprocal approximation [Knuth 1969, Aho et al 1974, Borodin and Munro 1975]. The Newton iteration formula,

$$y_{i+1} = 2(y_i) - (y_i)^2c,$$

for example, can be used to compute the reciprocal of c.

Theorem 4.3.1 The division of an (m+n)-digit integer by an n-digit integer can be computed in $O(\log^2(m+1)) + O(\log(m+n))$ parallel operations on a SIMD parallel computer with (m+1)(m+n) processors.

Proof. The above Newton iteration formula entails two (multiple-precision integer) multiplications and one subtraction. In this case, the number of parallel operations $T(2^k=m+1)$ required to compute the reciprocal satisfies, at the i-th stage,

$$T(2^i) = T(2^{i-1}) + O(i),$$

where

$$T(2^0) = 1.$$

Thus, $T(2^k) = O(k^2)$, and the $(m+1)$-digit reciprocal can therefore be computed in $O(\log^2(m+1))$ parallel operations.

The number of processors used at the i-th stage is $2^i n$. Thus, a total of $(m+1)n \leq (m+1)(m+n)$ processors is sufficient for computing the reciprocal.

Next, by Theorem 4.2.3, the quotient $q = a * (1/c)$ can be computed in $O(\log(m+n))$ parallel operations with $(m+1)(m+n)$ processors.

Finally, the remainder $r = a - q * c$ can obviously be computed in $O(\log(m+n))$ parallel operations with $(m+1)(m+n)$ processors. Q.E.D.

Corollary 4.3.2 The division of an m-digit integer by a 1-digit integer can be computed in $O(\log^2 m)$ parallel operations on a SIMD parallel computer with m^2 processors.

The parallel integer division algorithm given above is slower than the parallel integer multiplication algorithm. This is in contrast with our experience on sequential computers, where integer division and multiplication are of the same complexity. However, the above iterative algorithm is not the fastest possible parallel algorithm.

Theorem 4.3.3 The division of an $(m+n)$-digit integer by an n-digit integer can be computed in $O(\log(m+n))$ parallel operations on a SIMD parallel computer with $b^{m+1}(m+1)n$

processors.

Proof. First recall that the division of an (m+n)-digit integer by an n-digit integer results an (m+1)-digit quotient and an n-digit remainder.

For all integers x, $0 \leq x < b^{m+1}$, i.e., x has at most m+1 digits, do the following:

1. Compute x * c.

Since we are concerned only with integers which are candidates for the quotient, we compute only the least m+n significant digits of x * c and discard all x such that x * c exceeds m+n digits. This, by Theorem 4.2.3, can certainly be done in O(log(m+n)) parallel operations (b^{m+1}(m+1)n processors).

2. Compute a - x * c.

This can certainly be done in O(log(m+n)) parallel operations (b^{m+1}(m+n) proces rs).

3. If $0 \leq a - x * c < c$ then q = x and r = a - x * c.

It is easily seen that comparison of n-digit integers can be reduced to subtraction of n-digit integers (b^{m+1}n processors).

The existence and uniqueness of q and r are guaranteed by the Integer Division Algorithm. Q.E.D.

Corollary 4.3.4 The division of an m-digit integer by a 1-digit integer can be computed in O(log m) parallel operations on a SIMD parallel computer with b^mm processors.

It may appear therefore that on parallel computers the question regarding the parallel complexity of integer division has finally been resolved. The above parallel algorithm requires O(log(m+n)) parallel operations and a fan-in argument can be used to show that at least log(m+n) parallel operations are required. The parallel algorithm, however, functions with a strange pecularity. In the final step (step 3 in the proof of Theorem 4.3.3), the parallel algorithm uses unrestricted fan-in to select one special processor from indefinitely many. This capability may be viewed as being a multiple operand operation and fan-in arguments are therefore no longer applicable. The lower bound of log(m+n) parallel operations for division on such a computational model is therefore no longer valid. However, the above integer division algorithm still serves a useful purpose. On parallel computers such as STARAN, the capability of determining the location of the first active processor is available. This parallel algorithm might therefore be an efficient one to implement on such parallel computers at least for sufficiently small problems.

CHAPTER FIVE

Parallel Modular Arithmetic

There are many applications in which it is much more efficient to do arithmetic in "modular representation". Examples of such applications include multiple-precision integer arithmetic [Schoenhage & Strassen 1971, Chapter 4], integer and polynomial linear systems [Young & Gregory 1973, McClellan 1973], polynomial GCDs [Brown 1971, Collins 1971, Moses & Yun 1973] and polynomial factorizations [Berlekamp 1968, Musser 1975, Wang & Rothschild 1973].

The advantages of modular representation are that addition, subtraction and multiplication are very simple. In addition, on parallel computers computations corresponding to different moduli can all be done at the same time. The same kind of efficiency can not be achieved by the usual arithmetic discussed in Chapter 4, since carry propagation must be considered.

The general scheme of a modular method consists of three steps and is outlined as follows:

1) Convert all the input arguments into their modular representation.
2) Perform the required computations in the modulo classes.
3) Convert the results back to the ordinary representation.

Thus the use of modular arithmetic can be justified only if efficient algorithms for conversion into and out of the modular representation are available.

In Section 5.1 we consider polynomial modular transforms. The forward polynomial transform (i.e., the evaluation of polynomials of degree $\leq n-1$ at n points) can be done in $\lceil \log n \rceil$ parallel additions and $\lceil \log n \rceil$ parallel multiplications and this parallel operation bound is optimal. The inverse polynomial transform (i.e., the polynomial interpolation at n points) can be done in $\lceil \log(n-1) \rceil + \lceil \log n \rceil + 1$ parallel additions and $\lceil \log(n-1) \rceil + 3$ parallel multiplications / divisions and this parallel operation bound is asymptotically optimal. Both parallel algorithms assume polynomially bounded parallelism.

In Section 5.2 we consider integer modular transforms. Efficient parallel algorithms are given to find the n residues of an m-digit integer and the inverse problem (i.e. the Chinese remainder problem) in O(log n) parallel operations. Again, as in the division of integers, a more powerful computational model is used. These parallel algorithms are distinguished by the fact that all assume exponentially bounded parallelism.

In Section 5.3 we consider the exact solution of linear systems. By using modular methods and the results obtained in Section 5.2, full linear systems of order n can be solved exactly in O(log n) parallel operations with exponentially bounded parallelism. Consequently, multiplication of matrices and solution of linear systems are of the same parallel complexity.

In Section 5.4, by using results of Section 5.3, an efficient parallel algorithm is given for the division of polynomials. It will be shown that the quotient and the remainder of a polynomial of degree m+n divided by a polynomial of degree n can be computed in O(log(m+1)) parallel operations with exponentially bounded parallelism. Consequently, polynomial multiplication and division are of the same parallel complexity.

5.1 Polynomial Modular Transforms

Given any polynomial $f(x)$ and any monic linear polynomial $x-a$, by the Polynomial Division Algorithm, there exist unique polynomials $q(x)$ and $r(x)$ such that

$$f(x) = q(x)(x-a) + r(x), \deg(r(x)) < \deg(x-a)=1.$$

The ren nder $r(x)$ is called the residue of f modulo the modulus $x-a$. Note that the degree of $r(x)$ is less than 1. Therefore the polynomial $r(x)$ is a constant. Substituting the constant a into both sides of the above equality, it follows immediately that

$$r(x) = f(a).$$

Thus finding the residue of a polynomial modulo a monic linear polynomial is equivalent to evaluating a polynomial. Furthermore, the process is reversible.

Interpolation Theorem. Given n residues $\{ r_i \mid 1 \leq i \leq n \}$ corresponding to n moduli $\{ x-a_i \mid 1 \leq i \leq n \}$, there exists a unique polynomial $f(x)$ of at most degree $n-1$ such that

$$f(a_i) = r_i, \ 1 \leq i \leq n.$$

Thus, any polynomial f of at most degree $n-1$ can be represented uniquely by a set of n residues. The set of residues $\{ r_i \mid 1 \leq i \leq n \}$ is called the modular representation of f modulo $\{ x-a_i \mid 1 \leq i \leq n \}$. The process of finding the modular representation for a polynomial is called the forward polynomial modular transform.

The classical sequential algorithm for evaluating polynomials at many points uses Horner's rule and requires a total of $2n^2$ operations. Fast sequential algorithms for the same problem perform successive divisions by supermoduli (Cf. Chapter 3) [Fiduccia 1972, Moenck & Borodin 1972] and require a total of $O(n \log^2 n)$ operations. On parallel computers, it turns out that straightforward evaluations are optimal.

<u>Theorem 5.1.1</u> The n residues of a polynomial f(x) of degree n-1 can be computed in $\lceil \log n \rceil$ parallel additions and $\lceil \log n \rceil$ parallel multiplications on a SIMD parallel computer with n^2 processors. Furthermore, the parallel operation bound is optimal.

<u>Proof</u>. Since the computation of each residue is independent of the other, the n residues can be evaluated simultaneously. Thus, the parallel operation bound and its optimality follow from Theorem 3.1.1 on evaluation of polynomials. A total of n^2 processors is sufficient because evaluation of f at each point uses only n processors. Q.E.D.

The process of recovering the polynomial from its modular representation, i.e. exact interpolation, is called the <u>inverse polynomial modular transform</u>. The classical sequential algorithms for polynomial interpolation normally use the Lagrangian or Newtonian interpolation formula and require $O(n^2)$ operations [Lipson 1971]. Fast sequential algorithms reduce polynomial interpolation to evaluation and multiplication and require $O(n \log^2 n)$ operations [Horowitz 1974, Moenck & Borodin 1972]. On parallel computers polynomial interpolation can be solved efficiently by means of the Lagrangian interpolation formula:

$$f(x) = \sum_{k=1}^{n} r_k \prod_{i\neq k}^{n} [(x-a_i)/(a_k-a_i)].$$

<u>Algorithm</u> <u>I</u>. (Polynomial Interpolation)

```
I1. Compute $0-a_k$, $a_k-a_i$, $i \neq k$ and $1 \leq i \leq n$,
    $1 \leq k \leq n$.

I2. Compute $A_k \leftarrow \prod_{i\neq k}^{n} (a_k-a_i)$, $1 \leq k \leq n$.

I3. Compute $B_k \leftarrow r_k/ A_k$, $1 \leq k \leq n$.

I4. Compute $C_k(x) \leftarrow \prod_{i\neq k}^{n} (x+(-a_i))$, $1 \leq k \leq n$.

I5. Compute $D_k(x) \leftarrow B_kC_k(x)$, $1 \leq k \leq n$.

I6. Compute $\sum_{k=1}^{n} D_k(x)$.
```

<u>Theorem</u> <u>5.1.2</u> Polynomial interpolation at n points can be computed in $\lceil\log(n-1)\rceil+\lceil\log n\rceil+2$ parallel additions / subtractions and $2\lceil\log(n-1)\rceil+4$ parallel multiplications / divisions on a SIMD parallel computer with n^3 processors. Furthermore, the parallel operation bound is asymptotically optimal.

<u>Proof</u>. Steps I1 - I3 can obviously be computed in 1 parallel subtraction (n^2 processors), $\lceil\log(n-1)\rceil$ parallel

multiplications ($n\lfloor(n-1)/2\rfloor$ processors) and 1 parallel division (n processors), respectively.

Step I4, by Theorem 3.4.2 on elementary symmetric functions, can be computed in $\lceil\log(n-1)\rceil+1$ parallel additions, $\lceil\log(n-1)\rceil+1$ parallel multiplications and 1 parallel division. Since the computation of each $C_k(x)$ uses n^2 processors, a total of n^3 processors is sufficient.

Step I5, by Theorem 3.2.1, can be computed in 1 parallel multiplication. Since the computation of each $D_k(x)$ uses n processors (note that each $D_k(x)$ has degree n-1), a total of n^2 processors is sufficient.

Step I6 can be obtained by computing the sums of the corresponding coefficients of all $D_k(x)$'s which in turn are obtained by using the log-sum algorithm. Therefore, this step can be computed in $\lceil\log n\rceil$ parallel additions ($n\lfloor n/2\rfloor$ processors).

On summing the parallel operations, the /parallel operation bound follows. A trivial lower bound (obtained, for example, by the fan-in argument) for this problem is $\lceil\log n\rceil$. Therefore, the para l operation bound is asymptotically optimal. Q.E.D.

The parallel algorithm given above faithfully reflects the Lagrangian interpolation formula. But on a close examination, one immediately observes that the computations in steps I1 - I3 are independent of the computations in step I4. A careful rearrangement of the parallel algorithm, requiring steps I1 - I3 to be meshed with step I4, yields a slightly faster parallel algorithm.

Theorem _5.1.2'_ The exact polynomial interpolation at n points can be computed in $\lceil \log(n-1) \rceil + \lceil \log n \rceil + 1$ parallel additions and $\lceil \log(n-1) \rceil + 3$ parallel multiplications / divisions on a SIMD parallel computer with n^3 processors.

5.2 Integer Modular Transforms

The previous discussion about polynomial modular arithmetic has a strong analogy in the integer setting. Given any integer a and any integer $c > 0$, by the Integer Division Algorithm, there exist unique integers q and r such that

$$a = q * c + r, \ 0 \le r < c.$$

The remainder r is called the residue of a, modulo the modulus c, and denoted by $r = a \bmod c$.

The analogue of the Interpolation Theorem in the integer case is the Chinese Remainder Theorem:

> Chinese Remainder Theorem. Given n residu $\{ r \mid 1 \le i \le n \}$ corresponding to n mutually relatively prime moduli $\{ m_i \mid 1 \le i \le n \}$, there exists a unique integer u, $0 \le u < M = m_1 m_2 \cdots m_n$, such that
>
> $$r_i = u \bmod m_i, \ 1 \le i \le n.$$

Thus, any integer can be represented uniquely by a sufficiently large set of residues. The set of residues $\{ r_i \mid 1 \le i \le n \}$ is called the modular representation of u modulo $\{ m_i \mid 1 \le i \le n \}$.

By analogy with the Lagrangian interpolation formula, the Lagrangian Chinese remainder formula can be written as:

$$u = \sum_{k=1}^{n} r_k a_k b_k \bmod M,$$

where $a_k = \prod_{i \neq k}^{n} m_i$

and $b_k = (\prod_{i \neq k}^{n} m_i)^{-1} \bmod m_k$.

Although the Chinese Remainder Theorem does not explicitly require the moduli to be prime, normally they are so chosen since in practice multiplicative inverses are required at intermediate steps. In addition, for the sake of effi ency, the primes are chosen to be large 1-digit (i.e., single precision) integers.

The process of finding the modular representation for an integer is called the forward integer modular transform and its recovery from the residues the inverse integer modular transform.

It is obvious that forward integer modular transform requires no more parallel operations than does integer division. Consequently, as for division, we give two parallel algorithms, one assuming polynomially bounded parallelism and the other exponentially bounded.

Theorem 5.2.1 The n residues of an integer with at most m digits modulo n 1-digit moduli can be computed in $O(\log^2 m)$ parallel operations on a SIMD parallel computer with m^2n processors.

Proof. Since, by Corollary 4.3.2, each residue can be computed in $O(\log^2 m)$ parallel operations on a SIMD parallel computer with m^2 processors, the n residues can be computed in $O(\log^2 m)$ parallel operations on a SIMD parallel computer with m^2n processors. Q.E.D.

<u>Theorem</u> <u>5.2.2</u> The n residues of an integer with at most m digits modulo n 1-digit moduli can be computed in $O(\log m)$ parallel operations on a SIMD parallel computer with $b^m mn$ processors.

<u>Proof</u>. Since, by Corollary 4.3.4, each residue can be computed in $O(\log m)$ parallel operations on a SIMD parallel computer with $b^m m$ processors, the n residues can be computed in $O(\log m)$ parallel operations on a SIMD parallel computer with $b^m mn$ processors. Q.E.D.

Next, we show that inverse integer modular transform requires no more parallel operations than does integer division. We give two parallel algorithms, again, one assuming polynomially bounded parallelism and the other exponentially bounded. First we prove the following lemma.

<u>Lemma</u> <u>5.2.3</u> The product of n 1-digit integers can be computed in $O(\log^2 n)$ parallel operations on a SIMD parallel computer with n^2 processors.

<u>Proof</u>. Using the "log-product" algorithm, form products of 2 factors, then products of 4 factors, and so on. At the k-th stage when forming terms with 2^k factors (i.e., when taking products of terms with 2^{k-1} factors), $\log(2^{k-1}+2^{k-1}) = k$ parallel operations are required. Thus, a total of

$$1 + 2 + \ldots + \lceil \log n \rceil = \lceil \log n \rceil^2/2 + \lceil \log n \rceil/2$$

parallel operations is required.

In addition, at the k-th stage, there are $\lfloor n/2 \rfloor$ factors each using $2^{k-1} * 2^{k-1} = 2^{2k-2}$ processors to compute. Thus, a total of $n * 2^{k-2} \leq n^2/4$ processors is sufficient.

Q.E.D.

<u>Theorem 5.2.4</u> Given n residues modulo n 1-digit moduli, the inverse integer modular transform can be computed in $O(\log^2(n-1))$ parallel operations on a SIMD parallel computer with n^3 processors.

<u>Proof</u>. Using the same notation given in the Lagrangian Chinese remainder formula above, each a_k is a product of n-1 1-digit integers. By Lemma 5.2.3, $\{ a_k \mid 1 \leq k \leq n \}$ can be computed in $O(\log^2(n-1))$ parallel operations ([illegible]1) processors). Since each a_k has at most n-1 digits, $\{ a_k \bmod m_k \mid 1 \leq k \leq n \}$ can be computed in another $O(\log^2(n-1))$ parallel operations ($(n-1)^2 n$ processors).

In addition, since each modulus m_k is prime, by Fermat's Theorem, b_k can be obtained from

$$\begin{aligned} b_k &= (a_k)^{-1} \bmod m_k \\ &= (a_k \bmod m_k)^{-1} \bmod m_k \\ &= (a_k \bmod m_k)^{m_k - 2} \bmod m_k . \end{aligned}$$

Thus $\{ b_k \mid 1 \leq k \leq n \}$ can be computed from $\{ a_k \bmod m_k \mid 1 \leq k \leq n \}$ in at most $\lceil \log (\max \{ m_k - 2 \mid 1 \leq k \leq n \}) \rceil$ parallel multiplications (n processors).

Finally, it is obvious that the products $\{ r_k a_k b_k \mid 1 \leq k \leq n \}$ and the summation of them can be computed in $O(\log n)$ parallel operations $(2(n-1)n$ processors). Since the sum has at most $n+\log_b n$ digits, its residue taken modulo $M = m_1 m_2 \quad m_n$ can be computed in at most $O(\log^2(n-1))$ parallel operations ($O(n^2)$ processors). Q.E.D.

<u>Theorem 5.2.5</u> The inverse integer modular transform of Theorem 5.2.4 can be computed in $O(\log n)$ parallel operations on a SIMD computer with $b^n n^2$ processors.

<u>Proof</u>. For all integer x, $0 \leq x < M = m_1 m_2 \ldots m_n$, do the following:

1. Define n residue functions,

$$u_i(x) = x \bmod m_i, \ 1 \leq i \leq n.$$

By Corollary 4.3.4, this can be computed in $O(\log n)$ parallel operations ($Mb^n n^2$ processors).

2. Define n Boolean functions,

$$b_i(x) = 1 \text{ iff } u_i(x) = r_i, \ 1 \leq i \leq n.$$

This can be computed in 1 parallel (single-precision) comparison (Mn processors).

3. Compute the Boolean product,

$$c(x) = b_1(x) \bullet b_2(x) \bullet \ldots \bullet b_n(x).$$

This can be computed in $\lceil \log n \rceil$ parallel ANDs ($M \lfloor n/2 \rfloor$ processors).

4. If $c(x) = 1$ then $u = x$.

The existence and uniqueness of u is guaranteed by the Chinese Remainder Theorem.

Summing up all the parallel operations we have $O(\log n)$

total parallel operations. Since M is bounded by b^n the processor bound follows. Q.E.D.

5.3 Exact Solution of Linear Systems

Full linear systems of order n can be solved on parallel computers in (n) parallel operations by Gauss-Jordan eliminatior [Borodin & Munro 1975], LDU factorization, or ens reduction [Sameh & Kuck 1975]. In addition, certai structured sparse linear systems can be solved much more efficiently, e.g., triangular linear systems can be solved in $O(\log^2 n)$ parallel operations [Heller 1974, Hyafil & Kung 1974, Chen & Kuck 1975, Orcutt 1974], and tridiagonal linear systems can be solved in $O(\log n)$ parallel operations [Buneman 1969, Stone 1973].

Despite these facts, researchers present no significantly faster parallel algorithm than the Gauss-Jordan elimination for the solution of general linear systems [Pease 1969, Sameh & Kuck 1975, Csanky 1974]. (Recently, Csanky presents a parallel algorithm which requires $O(\log^2 n)$ parallel operations.)

In this section we present an $O(\log n)$ parallel algorithm for solving full linear systems. Thus, as is the case for sequential computers [Borodin & Munro 1975], multiplication and inversion of matrices are of the same parallel complexity, both requiring $O(\log n)$ parallel operations. The parallel algorithm given below is a modular method which gives exact solution of linear systems. Using modular method, all computations performed, except the initial and final conversions, are modular arithmetic:

1) $(a \pm b) \bmod p = (a \bmod p \pm b \bmod p) \bmod p$.

2) $(a*b) \bmod p = [(a \bmod p) * (b \bmod p)] \bmod p$.

3) If $a \neq 0 \bmod p$, then a^{-1} exists and

$$a^{-1} \bmod p = (a \bmod p)^{p-2} \bmod p.$$

In doing modular arithmetic, it is not clear how many basic machine instructions are needed to implement one modular operation, since this is a highly machine dependent feature. But, for any particular computer, the total number of machine instructions executed must be proportional to the number of modular operations performed. Hence, in the rest of this chapter, all operations indicated are implicitly assumed to be modular operations.

Theorem 5.3.1 Given any n by n matrix A in GF(p), the inverse of A mod p can be computed in 1 parallel multiplication, $\lceil \log n \rceil$ parallel additions and $2\lceil \log n \rceil$ parallel ANDs on a SIMD parallel computer with $p^n n^2$ processors.

Proof. Let V(n) be the set of all n-vectors with components in GF(p). Obviously, $|V(n)| = p^n$. For each n-vector $\underline{x}$ in V(n), do the following:

1. Compute a vector function $\underline{v}(\underline{x}) = \underline{x} * A$.

By Corollary 2.1.4, this can be computed in 1 parallel multiplication and $\lceil \log n \rceil$ parallel additions. Since each vector-matrix product uses n^2 processors to compute, a total of $p^n n^2$ processors is sufficient.

2. Define n vector Boolean functions,

$$\underline{b}_k(\underline{x}) = [\underline{v}(\underline{x}) = \underline{i}(k)], \ 1 \leq k \leq n,$$

where $\underline{i}(k)$ is the k-th row vector of the identity matrix I (of order n). These functions can be obtained in 1 parallel comparison ($p^n n^2$ processors).

3. Define n scalar Boolean functions,

$c_k(\underline{x})$ = the Boolean product of the Boolean vector $\underline{b}_k(\underline{x})$, $1 \le k \le n$.

For each k and each $\underline{x}$, the Boolean product can be computed in $\lceil \log n \rceil$ parallel ANDs ($\lfloor n/2 \rfloor$ processors) by using a Boolean version of the log-product algorithm. A total of $p^n n \lfloor n/2 \rfloor$ processors is sufficient for computing these functions.

4. If $c_k(\underline{x}) = 1$ then $\underline{x}(k) = \underline{x}$ and $d(k) = 1$, $1 \le k \le n$, where $\underline{x}(k)$ constitutes the k-th row vector of a matrix X and $\underline{d} = (d(1), d(2), \ldots, d(n))$ is a Boolean vector (initially all of which are false).

5. Compute e = the Boolean product of the Boolean vector $\underline{d}$.

This can certainly be computed in $\lceil \log n \rceil$ parallel ANDs ($\lfloor n/2 \rfloor$ processors).

6. If e = 1 then the inverse of A exists and $A^{-1} = X$. Otherwise, the inverse of A does not exist.

Counting all the parallel operations and processors, the allel operation and processor bounds follows. Q.E.D.

The above parallel algorithm is restricted to finite fields and is due to Csanky [1974]. We use it below to design an efficient parallel algorithm over the rational field.

For any n by n matrix A, the following notations are adopted:

$a(i,j)$	the (i,j) entry of A
$A(i,j)$	the cofactor of $a(i,j)$
$Adj(A)$	the adjoint of A
$det(A)$	the determinant of A.

The following identity is well known:

$$A * Adj(A) = det(A) * I \qquad (1a)$$

or
$$A^{-1} = Adj(A) / det(A). \qquad (1b)$$

[illegible]w give an efficient parallel algorithm for co[illegible] the determinant of A. First, define n submatrices of A as follows:

$A^{(1)} = A$

$A^{(2)}$ = the minor of $a^{(1)}(1,1)$, i.e., the submatrix obtained from A by deleting the first row and the first column.

$A^{(k+1)}$ = the minor of $a^{(k)}(1,1)$, i.e., the submatrix obtained from A by deleting the first k rows and the first k columns.

Obviously,

$$A^{(n)} = (a(n,n)), \ det(A^{(n)}) = a(n,n)$$

and
$$A^{(k)}(1,1) = det(A^{(k+1)}), \ 1 \le k \le n-1.$$

<u>Theorem</u> <u>5.3.2</u> Let $B^{(k)} = (A^{(k)})^{-1}$, $1 \le k \le n-1$. If all the $\{ B^{(k)} \mid 1 \le k \le n-1 \}$ are known then the determinant of (A mod p) can be computed in $\lceil \log(n-1) \rceil$ additional parallel

multiplications and 1 parallel division.

Proof. Express (1b) in terms of the (1,1) entries,

$$\det(A) = A^{(1)}(1,1) / b^{(1)}(1,1)$$
$$= \det(A^{(2)}) / b^{(1)}(1,1).$$

Thus, the identity (1b) can be applied again and again

$$\det(A) = A^{(2)}(1,1) / b^{(1)}(1,1)*b^{(2)}(1,1) = \cdots$$
$$= a(n,n) / b^{(1)}(1,1)*b^{(2)}(1,1)* \ldots *b^{(n-1)}(1,1),$$

where each $b^{(k)}(1,1)$ is the (1,1) entry of $B^{(k)}$. This expression [27] can obviously be computed in $\lceil \log(n-1) \rceil$ parallel multiplicaitons and 1 parallel division. Q.E.D.

Once we have det(A) the solution of the linear system

$$A \, \underline{x} = \underline{b}$$

can be found immediately from:

$$\underline{x} = A^{-1} \, \underline{b} = (\mathrm{Adj}(A) * \underline{b}) / \det(A)$$
$$= \underline{y} / \det(A),$$

where $\underline{y} = \mathrm{Adj}(A) * \underline{b}$. Thus, an efficient solution of the linear system relies on an efficient solution of $\underline{y}$.

Lemma 5.3.3 Let $\underline{z} = (A \bmod p)^{-1} * (\underline{b} \bmod p)$. Then $(\underline{y} \bmod p) = (\det(A) \bmod p) * \underline{z}$. Hence $\underline{y}$ mod p can be computed from $\underline{z}$ in 1 parallel multiplication.

Proof. The first part can be found in [Young & Gregory 1973]. The second part is a trivial consequence of the first part. Q.E.D.

[27] This expression is true in any field, not just in GF(p).

The last remaining problem in a modular method is the number of prime moduli needed to guarantee the unique representation of det(A) and $\underline{y}$.

Let $\underline{a}(i)$ be the i-th row vector of A, then we can choose the moduli $\{ p_i \mid 1 \le i \le m \}$ such that [Young & Gregory 1973]

$$p_1 p_2 \cdots p_m \ge 2\, \|\underline{a}(1)\| * \|\underline{a}(2)\| * \cdots * \|\underline{a}(n)\| * |\underline{b}|,$$

where $\|\cdot\|$ is the 2-norm (i.e., Euclidean norm) and $|\cdot|$ is the 1-norm. In parallel computation the exact number of moduli needed is not as crucial here as in the sequential case, because computations corresponding to each modulus can be performed simultaneously. The important thing here is that the precision of the above bound is roughly proportional to n, the order of A. Thus, by Theorem 5.2.5, the number of parallel operations required to restore det(A) and $\underline{y}$ is proportional to O(log n).

We now can proceed to give the complete algorithm:

Algorithm L. (Exact Solution of Linear Systems)

For each modulus p, do the following:

L1. Find (A mod p) and ($\underline{b}$ mod p). [28]

L2. Compute $B^{(k)} = [(A \bmod p)^{(k)}]^{-1}$, $1 \le k \le n-1$.

L3. Compute det(A) mod p.

L4. Compute $\underline{z} = (A \bmod p)^{-1} * (\underline{b} \bmod p)$.

L5. Compute $\underline{y} \bmod p = (\det(A) \bmod p) * \underline{z}$.

L6. Recover det(A) and y.

The solution is the quotient of y and det(A), and the actual division may or may not necessarily be carried out.

Theorem 5.3.4 Full linear system of order n can be solved exactly in O(log n) parallel operations on a SIMD parallel computer with exponentially bounded parallelism.

Proof. The only multiple-precision arithmetic performed are in step L1 and step L6. Step L1 can be computed in a constant number of parallel operations. Step L6 can be computed in O(log n) parallel operations as commented previously.

Steps L2 - L4 each can be computed in O(log n) parallel operations by Theorem 5.3.1, Theorem 5.3.2, and Corollary 2.1.4, respectively. Step L5 can be computed in 1 parallel multiplication by Lemma 5.3.3.

Thus, the complete algorithm can be computed in O(log n) parallel operations. Both step L2 and step L6 assume exponentially bounded parallelism. Q.E.D.

Theorem 5.3.5 Matrix multiplication, matrix inversion, determinant calculation and solution of linear systems are of the same parallel complexity.

[28] The fact that A and b must be integral is no serious restriction due to the fact that if the entries of A and the components of b are rational numbers, they can be converted to integers simply by scaling.

Proof. It is known that inversion of matrices, solution of linear systems and calculation of determinants are all of the same parallel complexity [Csanky 1974, Borodin and Munro 1975]. By Theorem 5.3.4 and Corollary 2.1.4, the solution of linear systems and matrix multiplication are of the same parallel complexity; namely, O(log n). Therefore, all three of the above problems and matrix multiplication are of the same parallel complexity. Q.E.D.

5.4 Polynomial Division Revisited

Let f and g be any two polynomials of degree m+n and n, respectively, and q and r be the quotient and remainder, respectively, of f divided by g. By using the results obtained in the last section, a very efficient parallel algorithm for computing q and r can be derived.

Theorem 5.4.1 The quotient q and the remainder r of f divided by g can be computed in O(log(m+1)) parallel operations on a SIMD parallel computer assuming exponentially bounded parallelism.[29]

Proof. In Section 3.3, a linear recurrence equation is derived for the coefficients of q. As is well-known a linear recurrence equation can be viewed as being a triangular linear system:

$$G \underline{q} = \underline{f},$$

where $\underline{q}$ and $\underline{f}$ are the (m+1)-vectors formed by the coefficients of q and the first m+1 coefficients of f, respectively, and G is a given matrix of order m+1. Then, by Theorem 5.3.4, $\underline{q}$ (consequently the polynomial q) can be solved in O(log(m+1)) parallel operations (assuming exponentially bounded parallelism).

Next, g * q can be computed in 1 parallel

[29] In Corollary 3.3.2 a parallel algorithm requiring O(log(m+1)log(n+1)) parallel operations but assuming polynomially bounded parallelism is given.

multiplication and no more than $\lceil \log(m+1) \rceil$ parallel additions.

Finally, the remainder, $r = f - g * q$, can be computed in 1 additional parallel subtraction. Q.E.D.

Corollary 5.4.2 Polynomial multiplication and division are of the same parallel complexity.

CHAPTER SIX

Conclusions

The main objective of this thesis was to find the best possible parallel algorithms (in the sense defined in Chapter 1) for integer arithmetic, polynomial arithmetic and modular arithmetic. It turns out that there exists very efficient parallel algorithms for all problems studied. Thus, the main objective of this thesis has been achieved.

We observe, however, that in most cases there is a tradeoff between the number of parallel operations performed and the number of processors required. If the level of parallelism of a parallel algorithm (i.e., the number of processors required by the parallel algorithm) is used to classify parallel algorithms, then among all best possible parallel algorithms studied in this thesis there are clearly three broad classes:

1) parallel algorithms assuming linearly bounded parallelism, e.g., polynomial addition and integer addition.
2) parallel algorithms assuming polynomially bounded parallelism, e.g., polynomial multiplication and integer multiplication.
3) parallel algorithms assuming exponentially bounded parallelism, e.g., polynomial division and integer division.

The three classes of parallel algorithms can be considered increasingly more difficult if level of parallelism is adopted as the measure of parallel complexity.

For future research the following problems are therefore suggested:

1) Design (asymptotically) optimal parallel algorithms assuming polynomially (or linearly) bounded parallelism for polynomial division, integer division, the Chinese remainder problem and the solution of full linear systems.
2) Prove parallel lower bounds under *a priori* polynomially (or linearly) bounded parallelism for various problems.
3) Study tradeoffs between the number of parallel operations performed and the number of processors required for various problems (especially those listed in 1 above).

REFERENCES

1. A. V. Aho, J. E. Hopcroft and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley 1974

2. A. J. Atrubin, "A One Dimensional Real-Time Iterative Multiplier," IEEE EC-14 1965 pp 394-399

3. G. H. Barnes, R. A. Stokes, R. M. Brown, [illegible]. Kato, [illegible]. J. Kuck and D. L. Slotnick, "The ILLIAC IV Computer," IEEE C-17,8 Aug 1968 pp 746-757

4. K. E. Batcher, "STARAN Parallel Processor System Hardware," AFIPS Conference Proc 43 1974 NCC pp 405-410

5. E. R. Berlekamp, Algebraic Coding Theory, McGraw-Hill 1968

6. A. Borodin, "Horner's Rule is Uniquely Optimal," in Theory of Machines and Computation, Z. Kohavi and A. Paz (ed) Academic Press 1971

7. A. Borodin and I. Munro, The Computational Complexity of Algebraic and Numeric Problems, American Elsevier 1975

8. R. Brent, "On the Addition of Binary Numbers," IEEE C-19,8 Aug 1970 pp 758-759

9. R. Brent, D. Kuck and K. Maruyama, "The Parallel Evaluation of Arithmetic Expressions without Division," IEEE C-22,5 May 1973 pp 532-534

10. R. P. Brent, "The Parallel Evaluation of General Arithmetic Expressions," JACM 21,2 Apr 1974 pp 201-206

11. W. S. Brown, "On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors," JACM 18,4 Oct 1971 pp 478-504

12. O. Buneman, "A Compact Non-Iterative Poisson Solver," SUIPR Rep No 294 Institute for Plasma Reaearch Stanford Univ May 1969

13. S. C. Chen and D. J. Kuck, "Time and Parallel Processor Bounds for Linear Recurrence Systems," IEEE C-24,7 Jul 1975 pp 701-717

14. G. E. Collins, "The Calculation of Multivariate Polynomial Resultants," JACM 18,4 Oct 1971 pp 515-532

15. L. Csanky, "On the Parallel Complexity of Some Computational Problems," Ph D Thesis Dept Electrical Engineering and Computer Science Univ of California Berkeley 1974

16. W. S. Dorn, "Generalizations of Horner's Rule for Polynomial Evaluation," IBM J Research and Development 6 1962 pp 239-245

17. G. Estrin, "Organization of Computer System - The Fixed plus Variable Structure Computer," AFIPS Conference Proc 1960 WJCC pp 33-40

18. C. M. Fiduccia, "Polynomial Evaluation via the Division Algorithm - the Fast Fourier Transform Revisited," Proc 4th Annual ACM Symposium on Theory of Computing 1972 pp 88-93

19. I. Flores, The Logic of Computer Arithmetic, Prentice Hall 1963

20. M. J. Flynn, "Very High-Speed Computing Systems," Proc IEEE 54,12 Dec 1966 pp 1901-1909

21. M. J. Flynn, "Some Computer Organizations and Their Effectiveness," IEEE C-21,9 Sep 1972 pp 948-960

22. D. K. Goyal, "Parallel Evaluation of Elementary Symmetric Functions," Tech Rep No 184 Dept of Electrical Engineering Princeton Univ Apr 1975

23. D. E. Heller, "On the Efficient Computation of Recurrence Relations," ICASE Report NASA Langley Research Center Hampton Va Jun 1974

24. R. G. Hintz and D. P. Tate, "Control Data STAR-100 Processor Design," Digest of Papers COMPCON 72 Sep 1972 pp 1-4

25. E. Horowitz, "A Unified View of the Complexity of Evaluation and Interpolation," Acta Informatica 3 1974 pp 123-133

26. L. Hyafil and H. T. Kung, "Parallel Algorithms for Solving Triangular Linear Systems with Small Parallelism," Rep Dept Computer Science Carnegie-Mellon Univ Oct 1974

27. M. D. Johnson, "The Architecture and Implementation of a Parallel Element Processing Ensemble," Digest of Papers WESCON 72 1972

28. Z. Kedem and D. Kirkpatrick, "Addition Requirements of Rational Expressions," Unpublished manuscript 1974

29. D. E. Knuth, The Art of Computer Programming : Seminumerical Algorithms, Addison-Wesley 1969

30. P. M. Kogge and H. S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," IEEE C-22,8 Aug 1973 pp 786-793

31. P. M. Kogge, "Parallel Solution of Recurrence Problems," IBM J Research and Development 18 1974 pp 138-148

32. D. Kuck and Y. Muraoka, "Bounds on the Parallel Evaluation of Arithmetic Expressions Using Associativity and Commutativity," Acta Informatica 3,3 1974 pp 203-216

33. D. J. Kuck and K. Maruyama, "Time Bounds on the Parallel Evaluation of Arithmetic Expressions," SIAM J Computing 4,2 Jun 1975 pp 147-162

34. H. T. Kung, "Fast Evaluation and Interpolation," Rep Dept Computer Science Carnegie Mellon Univ Jan 1973

35. H. T. Kung, "New Algorithms and Lower Bounds for the Parallel Evaluation of Certain Rational Expressions," Proc 6th Annual ACM Symposium on Theory of Computing Apr 1974

36. J. J. Lambiotte Jr. and R. G. Voigt, "The Solution of Tridiagonal Linear Systems on the CDC STAR-100 Computer," ACM Transactions on Mathematical Software 1,4 Dec 1975 pp 308-329

37. J. Lipson, "Chinese Remainder and Interpolation Algorithms," Proc 2nd Symposium on Symbolic and Algebraic Manipulation 1971 pp 372-391

38. M. T. McClellan, "The Exact Solution of Systems of Linear Equations with Polynomial Coefficients," JACM 20,4 Oct 1973 pp 563-588

39. K. Maruyama, "On the Parallel Evaluation of Polynomials," IEEE C-22,1 Jan 1973 pp 2-5

40. W. L. Miranker, "A Survey of Parallelism in Numerical Analysis," SIAM Review 13,4 Oct 1971 pp 524-547

41. R. Moenck and A. B. Borodin, "Fast Modular Transforms via Division," Conference Record IEEE 13th Annual Symposium on Switching and Automata Theory 1972 pp 142-151

42. J. Moses and D. Y. Y. Yun, "The EZ GCD Algorithm," Proc 1973 ACM National Conference pp 159-166

43. I. Munro and M. Paterson, "Optimal Algorithms for Parallel Polynomial Evaluation," J Computer and System Sciences 7 1973 pp 189-198

44. Y. Muraoka and D. J. Kuck, "On the Time Required for a Sequence of Matrix Products," CACM 16,1 Jan 1973 pp 22-26

45. D. R. Musser, "Multivariate Polynomial Factorization," JACM 22,2 Apr 1975 pp 291-308

46. A. Newell and G. Robertson, "Some Issues in Programming Multi-Mini Processors," Research Rep Dept Computer Science Carnegie Mellon Univ Jan 1975

47. S. E. Orcutt Jr., "Parallel Solution Methods for Triangular Linear Systems of Equations," Report 77 Digital Systems Laboratory Stanford Univ

48. M. C. Pease, "Inversion of Matrices by Partitioning," JACM 16,2 Apr 1969 pp 302-314

49. F. P. Preparata and D. E. Muller, "Efficient Parallel Evaluation of Boolean Expressions," IEEE C-25,5 May 1976 pp 548-549

50. J. A. Rudolph, "A Production Implementation of an Associative Array Processor - STARAN," AFIPS Conference Proc 1972 FJCC pp 229-241

51. A. H. Sameh and D. J. Kuck, "Linear System Solvers for Parallel Computers," Rep No UIUCDCS-R-75-701 Dept Computer Science Univ of Illinois Feb 1975

52. A. Schoenhage and V. Strassen, "Schnelle Multiplikation grosser Zahlen," Computing 7 1971 pp 281-292

53. B. Soucek, Microprocessors and Microcomputers, John Wiley and Sons 1976

54. H. S. Stone, "An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations," JACM 20,1 Jan 1973 pp 27-38

55. H. S. Stone, "Problems of Parallel Computation," in Complexity of Sequential and Parallel Numerical Algorithms J. F. Traub (ed) Academic Press 1973 pp 1-16

56. H. S. Stone, "Parallel Tridiagonal Solvers," ACM Transactions on Mathematical Software 1,4 Dec 1975 pp 289-307

57. K. J. Thurber and P. C. Patton, "The Future of Parallel Processing," IEEE C-22,12 Dec 1973 pp 1140-1143

58. K. J. Thurber and L. D. Wald, "Associative and Parallel Processors," ACM Computing Surveys 7,4 Dec 1975 pp 215-255

59. P. Wang and L. Rothschild, "Factoring Multivariate Polynomials over the Integers," SIGSAM Bulletin 28 1973 pp 21-29

60. W. J. Watson, "The TI ASC - A Highly Modular and Flexible Super Computer Architecture," AFIPS Conference Proc 1972 FJCC pp 221-228

61. S. Winograd, "On the Time Required to Perform Addition," *JACM* *12,2* Apr 1965 pp 277-285

62. S. Winograd, "On the Time Required to Perform Multiplication," *JACM* *14,4* Apr 1967 pp 793-802

63. S. Winograd, "On the Number of Multiplications Necessary to Compute Certain Functions," *Communications on Pure and Applied Mathematics* 1970 pp 165-179

64. W. A. Wulf and C. G. Bell, "C.mmp - A Multi-Mini-Processor," *AFIPS Conference Proc* *1972 FJCC* pp 765-777

65. D. M. Young and R. T. Gregory, *A Survey of Numerical Mathematics*, Vol 2 Addison-Wesley 1973