Parallel algorithms for algebraic and numerical problems :

Iyad A. Ajwa
Lehigh University

PARALLEL ALGORITHMS FOR ALGEBRAIC AND NUMERICAL PROBLEMS: A SURVEY

by

IYAD A. AJWA

Thesis

Presented to the Graduate Committee

of Lehigh University

in candidacy for the degree of

Master of Science

in Computer Science

Lehigh University

1990

This thesis is accepted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

MAY 14, 1990
date

[illegible]
Professor Meghanad Wagh
Advisor in charge

Donald J. Hillman
Professor Donald J. Hillman
Computer Science Division Head

Lawrence J. Varnerin
Professor Lawrence J. Varnerin
CSEE Department Chairperson

بسم الله الرحمن الرحيم

In The Name of Allah, The Most Gracious, The Most Merciful

This thesis is dedicated with affection and respect to my parents:

Abdel-Rahim

and

Misa'deh

ACKNOWLEDGMENTS

I would like to express my greatest appreciation to my thesis advisor Professor Meghanad Wagh for sharing his experience, consistent support and encouragement. It is an honor for me to have him as my advisor.

High on my list of acknowledgements for financial support must be the Arab Student Aid International. In addition, I would like to express my gratitude to the staff of the Lehigh University Libraries. In particular Mrs. Stengel, Gail, Terry *the boss*, Sherilyn, Jan, Evelyn *the chief*, Marge, and Carol *the farmer* for their continued support.

I am greatly indebted to my friends especially *Omar Mohamed*, *Mohammad Al-Tuwaim*, and *Khaldoun Tahboub* for their continued support; *Moustafa Yousef* (*Uncle Abu-Ramie*), *Ali Jaber*, and *Adel Ali* for their support and for guiding me towards such a satisfying career. I owe special thanks to all of my brothers in the *Muslim Student Association* at Lehigh University.

Last, but by no means least, and most importantly, ***my parents Abdel-Rahim and Misa'deh*** to whom I dedicate this work, for their continual love, support, motivation, encouragement, understanding, patience which knew no bounds, and their willingness to endure so much for what seemed so little. I owe this to the inspiration of ***my brothers Yousef, Emad, and Ziad***, and to ***my sisters Raghdah and Rana.***

Table of Contents

List of Figures

List of Algorithms

ABSTRACT

The need for high computational throughput has always driven research in algorithm design. The recent advances in technology have made parallel machines a reality within reach of a large portion of scientific and engineering community. The parallel algorithms have, therefore, assumed increasing importance in numerical, algebraic, and number theoretic techniques.

This thesis presents a survey of parallel algorithms suitable for engineering and other applications requiring high computational throughput. In particular, we have presented algorithms for integer problems, polynomial computations, and matrix operations. Polynomial computation is important because polynomials are often used to represent transcendental functions and many diverse problems may be modeled through polynomials. We have presented algorithms for polynomial evaluation, for univariate and multivariate factoring, and for computing greatest common divisor polynomial. Matrix representation is used in almost all the engineering systems. Analysis of these systems then requires capability to manipulate matrices rather quickly. We have presented algorithms for matrix inversion, solving of linear systems, computing the characteristic polynomials, and determinants. We have also presented algorithms to deal with structured matrices such as triangular and tridiagonal. The integer algorithms deal mainly with the determination of the greatest common divisor. We have also presented general algorithms that convert

sequential in-line code to parallel code.

Chapter 1
INTRODUCTION

The ever increasing need for improved computational power has forced researchers to search for better computing environment. The recent strides in integrated circuit technology have made parallel computers a reality. Parallel computers allow concurrent processing of several parts of the problem thus speeding the overall solution. Many different parallel machines are now commercially available. However, the performance and usage of parallel computers is constrained by the availability of parallel algorithms. This thesis surveys the status research in parallel algorithms.

1. WHAT IS A PARALLEL ALGORITHM?

An algorithm can be defined in several ways. For the purposes of this thesis, it will be defined as a solution method. It is the primary ingredient in solving a computational problem on any computer. A parallel algorithm is a solution method for a given problem destined to be performed on a parallel computer.

Despite the admirable development in the computer industry, the majority of the existing computers have the same basic design principles formulated more than forty years ago: a memory unit, a control unit, and a processing unit. The control unit fetches an instruction from the memory unit to the processing unit. It sends back the result from the processing unit to the memory unit. These computers are usually refered to as uniprocessor computers. They contain only one unit of each kind, and hence only one instruction can be executed at a time. However, the past twenty five years have wittnessed the creation of new kinds of computers, namely the parallel computers. A parallel computer is one that consists of a collection of processing units, or processors, that cooperate to solve a problem by working simultaneously on different parts of that problem. The presence of many processors (the number of processors may reach several millions) significantly reduces the time required to solve the problem by a uniprocessor computer.

A person who is familiar with the computational problems notices that many of the solutions to these problems are of parallel nature, i.e., they have independent computations (a set of computations is called independent if each result variable appears in only one computation). This is one of the reasons which made parallel computing attractive. A second reason is the feasibility of the cost of parallel computers with large number of processors. Finally, parallel processing or computing makes it possible to choose a parallel computer architecture that is best suited to solve the problem under consideration.

Algorithms designed to work on uniprocessor computers are called sequential or serial algorithms, and algorithms designed to use parallel computing are called parallel algorithms. These are designed to exploit both the parallelism inherent in the problem and that available on the computer. Parallel algorithms depend on a simple yet crucial observation: independent computations may be executed simultaneously. For example, two n-vectors may be added in a single step using n parallel processors. The i-th processor performs the addition of the i-th components of the two vectors. This result is independent of the other computations. Therefore, it is said that vector addition exhibits inherent parallelism.

There are two basic approaches to the design of fast parallel algorithms. One is to start by recognizing the inherent parallelism of a good sequential algorithm, and to try to parallelize it. The second approach is to attempt to make the parallel time as small as possible, allowing an arbitrary number of processors. This approach is known as the asymptotic approach.

For a proper understanding of parallel algorithms, one needs to study the model of computation underlying the architecture of the parallel computers. Except for Section 4, the rest of this chapter is dedicated to this purpose. Section 4 briefly describes the organization of this survey.

2. PARALLEL ARCHITECTURE MODELS

In this section, a number of architecture models used in parallel computing are discussed. In Subsection 2.2.1., a description of characteristic features of parallel and pipeline computers is given. In 1966, Flynn [27] classified computers into four types: Single Instruction stream, Single Data stream (SISD);

Single Instruction stream, Multiple Data stream (SIMD); Multiple Instruction stream, Single Data stream (MISD); and Multiple Instruction stream, Multiple Data stream (MIMD). Only SIMD and MIMD models are of significant importance for this research. They will be discussed in Subsection 2.2.2. Loosely coupled and tightly coupled methods are explained in Subsection 2.2.3. In order to exchange data between different processing elements, some means of communication are necessary. Various network configurations are discussed in Subsection 2.2.4.

2.1. Pipeline and Array Processors

Pipeline and Array processors are dedicated parallel processors that are very good to solve certain kinds of problems. The idea behind pipeline computers is essentially that of an assembly line: if the same arithmetic operation is going to be repeated many times, throughput can be greatly increased by dividing the operation into a sequence of subtasks and maintaining a flow of operands in various stages of completion.

Array processors generally incorporate a large number of identical processing elements connected in a particular topology. These act synchronously under the control of a single unit issuing a set of identical instructions. Each of the processing elements may operate on a word (word-organized system), or on single-bit operands (bit-organized array processors). The speed of any given algorithm on an array processor is influenced by the routing network, which is discussed in Section 1.2.4.

Pipelining is used in many advanced microprocesors in the form of instruction lookahead or for specialized numerical operations. For example, by partitionning floating point operations into more basic suboperations, an assembly line structure or pipeline can be set up for repetitive calculations such as componentwise vector operations and inner products. Successive completed results leave the pipeline at a rate determined by the memory transfer rate and the internal stage delay, and not by the total time required for all the arithmetic operations together. This pipeline, sometimes called a vector processor, does not constitute a truly parallel system, it nevertheless provides a significant improvement in speed. Pipeline processor has a single CPU but with a limited amount of parallelism incorporated. Certain parts of the CPU which are responsible for seperate functions (fetching operands, executing arithmetic operations, outputting) can be instructed to operate simultaneously.

2.2. SIMD/MIMD Models

In Flynn's classification of computers, shown in Figure 1-1, the term SISD essentially designates the classical serial machine design. Each of the others refer to machines with a number of processors operating in parallel, to which multiple instruction and/or data streams are directed. For a discussion of the MISD design, the reader is refered to [2] and [51]. "Multiple Data stream" is now considered the most appropriate description for existing parallel machines.

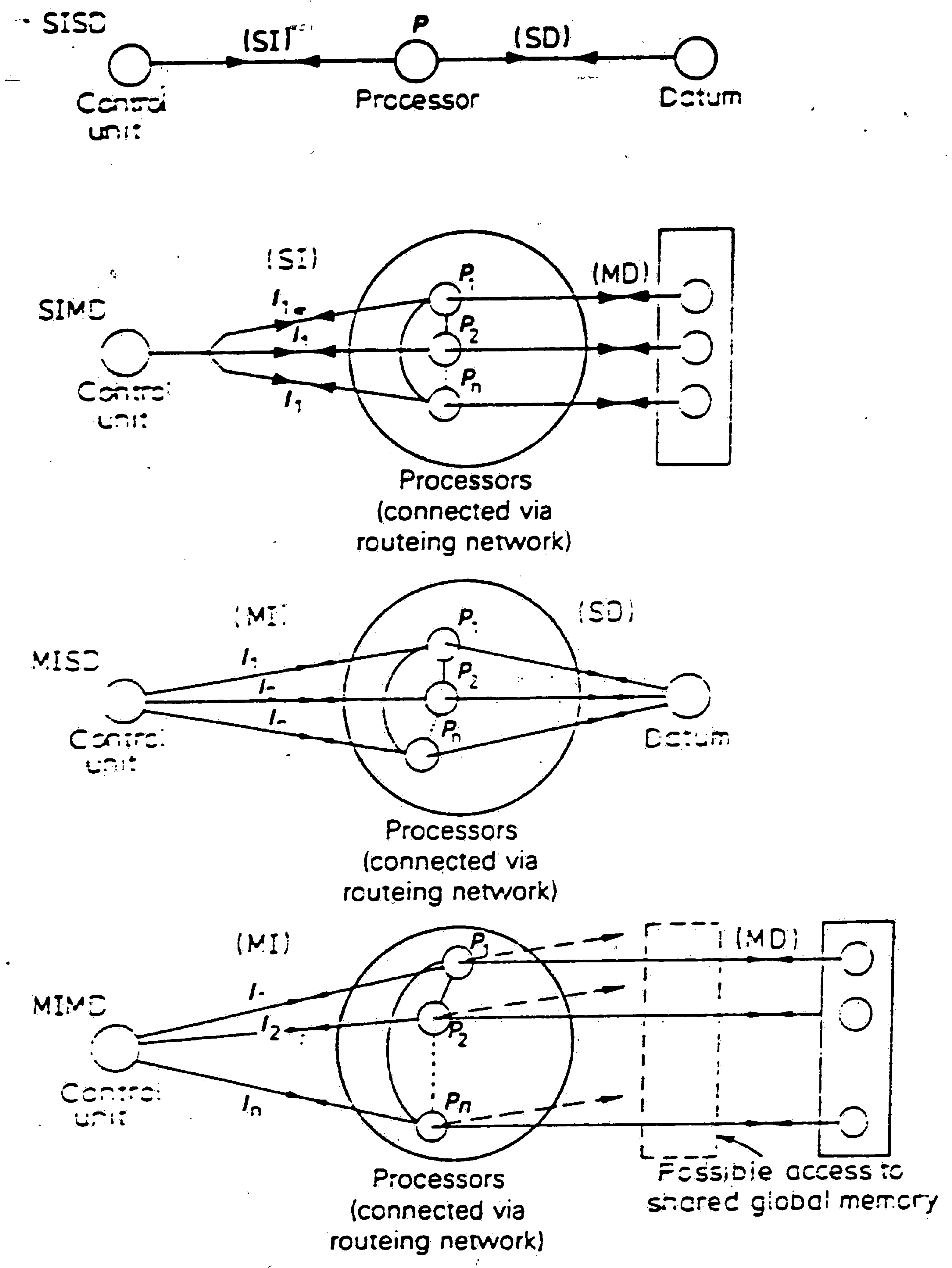

SISD
(SI)
P
(SD)
Control unit
Processor
Datum
SIMD
(SI)
(MD)
Control unit
Processors (connected via routeing network)
MISD
(MI)
(SD)
Control unit
Datum
Processors (connected via routeing network)
MIMD
(MI)
(MD)
Control unit
Processors (connected via routeing network)
Possible access to shared global memory

Figure 1-1. Flynn's Classification of Computers.

In the class of SIMD computers, a parallel computer consists of n identical processors, as shown in Figure 1-2. Each of the N processors possess a local memory where both programs and data are stored. Some systems may also provide an access from every processor to a global memory. All processors are under the control of a single instruction stream issued by a central control unit. The processors operate synchronously: at each step, all processors execute the same instruction, each on a different datum. Both of the instruction and the datum may be simple or complex. The instruction may include information such as which processors should be active or inactive. It is usually desirable for the processors to be able to communicate among themselves during the computation in order to exchange data or intermediate results. This communication can be done through a shared memory or via an interconnection network.

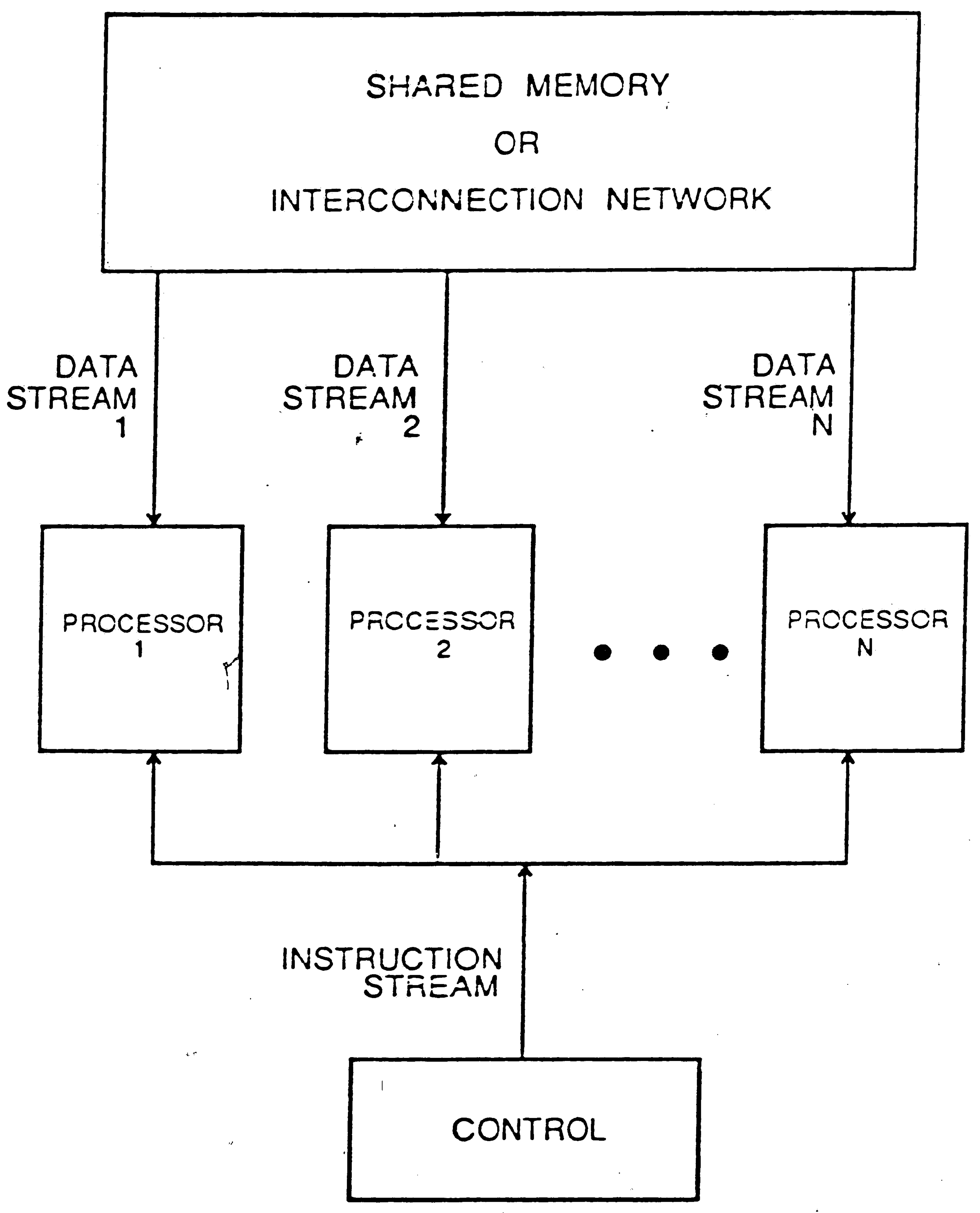


Figure 1-2. SIMD Computer Model.

The class of shared-memory SIMD computers is also known as the Parallel Random-Access Machines (PRAM) model. In this class, the N processors share a common memory in the same way a group of people may use a bulletin board. They use it to read input data, read or write intermediate results, and for writing final results. This class is fairly a powerful model of computation since it allows all available processors to gain access to the shared memory simultaneously. The class of interconnection-network SIMD computers is more powerful than the shared memory class. Here, the memory is distributed among the N processors, and every pair of processors are connected by a two-way line. At any step during the computation, processor P_i can receive a datum from processor P_j and send another one to processor P_k (or P_j). This model allows instantaneous communication between any pair of processors, and several pairs can communicate simultaneously.

It should be clear from the discussion of the SIMD computers that numerous problems covering a wide variety of applications can be solved by parallel algorithms on SIMD computers. Such algorithms are easy to design, analyze, and implement. The disadvantage of this class of computers is that it requires that the problems to be solved on them have a certain regular structure. These are the problems that can be subdivided into a set of identical subproblems, all of which can then be solved simultaneously by the same set of instructions. The computers used to solve those problems that lack the regular structure required of SIMD model are said to belong to the class of MIMD computers. In this case, the problem may be divided into subproblems that are not necessarily identical, and still may be solved parallely. The class of MIMD computers is the most general and most powerful class among the classes in Flynn's classification.

An MIMD computer has N processors, N streams of instruction and N streams of data, as shown in Figure 1-3. Each processor possesses its own control unit, local memory, and arithmetic and logic unit.

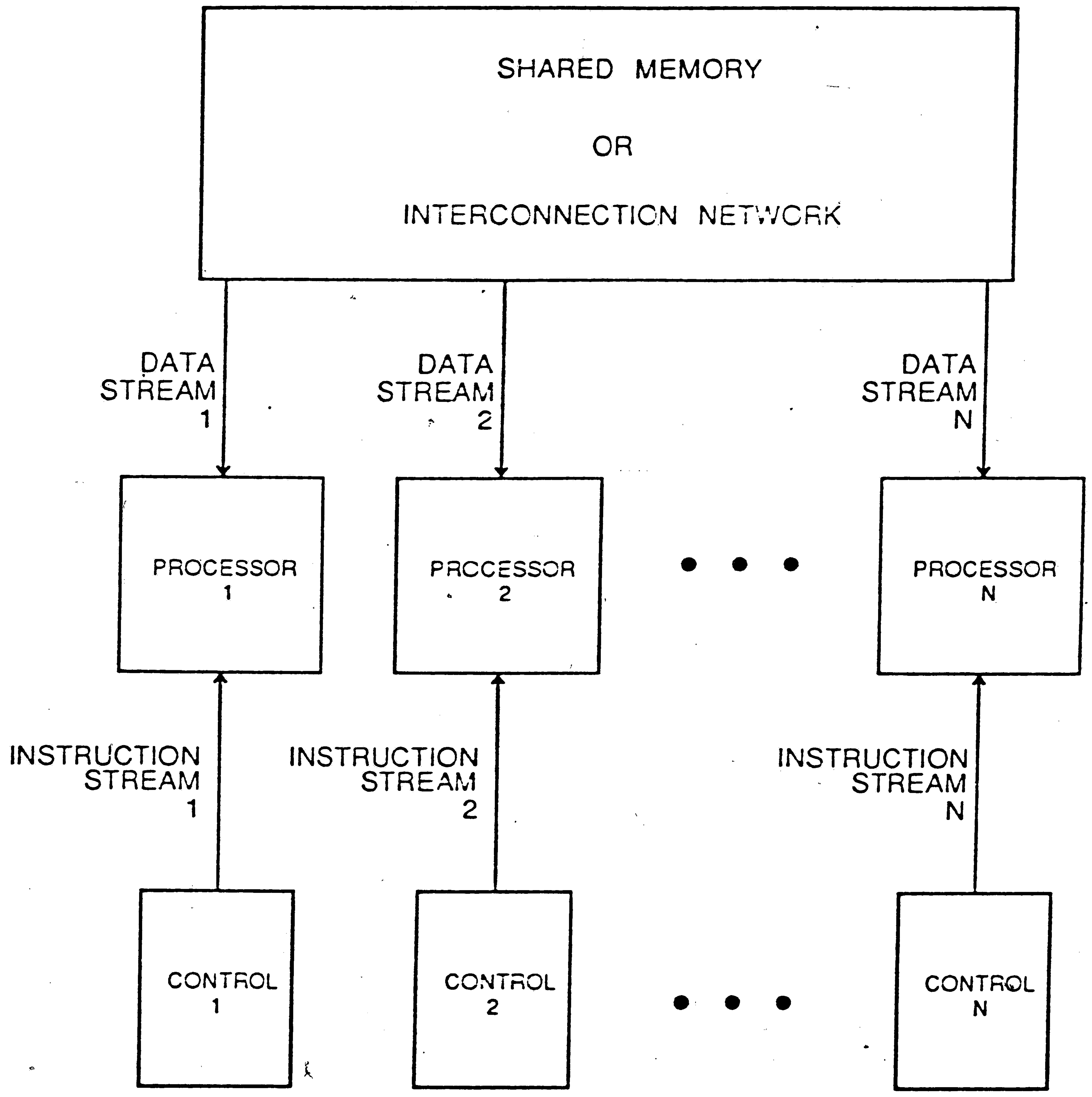


Figure 1-3. MIMD Computer Model.

The processors operate asynchronously, i.e., each processor has the potential to execute different programs on different data while solving different subproblems of a single problem. This is possible because each processor operates under the control of an instruction issued by its own control unit. In the class of MIMD computers, the communication between the processors is performed through either a shared memory or an interconnection network, as with the SIMD computers.

The design, evaluation, and implementation of algorithms on MIMD computers is considerably difficult because the processors operate asynchronously. In most MIMD computers, each processor has access to a global memory which may reduce processor communication delay.

2.3. Loosely Coupled and Tightly Coupled Machines

As it was mentioned in the previous subsection, the communication between the processors of an MIMD computer is performed either through a shared memory or an interconnection network. MIMD computers with a shared memory are called tightly coupled machines (or multiprocessors) while those with an interconnection network are called loosely coupled machines (or multicomputers).

In the basic model of tightly coupled machines, all processors are allowed to gain access to the shared memory simultaneously if the memory locations they are trying to write into or read from are different. However, two or more processors executing an asynchronous algorithm may wish to gain access to the same memory location. There are four models of this concurrent memory access:

1. No multiple-reading or writing. EREW (Exclusive-Read, Exclusive-Write).
2. Multiple-read, no multiple-writing. CREW (Concurrent-Read, Exclusive-Write).
3. Multiple-write, no multiple-read. ERCW (Exclusive-Read, Concurrent-Write).
4. Multiple-write, multiple-read. CRCW (Concurrent-Read, Concurrent-Write).

Multiple-read access poses no problem, but multiple-write accesses do. Several policies have been proposed to resolve the write conflicts [2].

Loosely coupled machines are sometimes referred to as distributed systems. The distinction is usually based on the physical distance seperating the processors. If all processors are in close proximity of one another, then they are loosely coupled systems; otherwise they are distributed systems. In these systems, the number of data exchanges among them is significantly more important than the number of computational steps performed by any of them.

2.4. Network Configurations

In order to exchange data between different processors, some means of communication is necessary. This may be effected by a routing network. Alignment of data with elements is carried out in parallel, to the extend that the algorithm and the nature of the routing network permit. In practice, many particular features of program and algorithm design arise out of limitations imposed by the network. Fortunately, in most applications a small subset of all pairwise connections is usually sufficient to obtain a good performance. The most popular of these networks are outlined in what follows. There are several other networks beside the ones to be described. The decision regarding which of these to use depends largely on the application, and in particular, on such factors as the kinds of computations to be performed, the desired speed of execution, and the number of processors available.

As it can be seen from this section, each network provides a different trade-off of the hardware complexity (characterized by number of links) and communication speed (characterized by the maximum distance between two processors in the network).

Linear Array and Cyclic Configurations, [2] & [51]:

The N processors P_i, $i = 0, 1, \cdots, N-1$, are connected to form a one-dimensional array, as shown in Figure 1-4. Any processor P_j can directly access data from its adjacent neighbours P_{j-1}, P_{j+1}. In practice, and for additional flexibility, the first and the last processors are also connected to provide a ring configuration. There are $N-1$ links in this network and the maximum distance between diametrically opposite processors is $N/2$.

Two-dimensional Array and Lattice Mesh, [2] & [51]:

A two-dimensional array network is obtained by arranging the N processors into an $\sqrt{N}\times\sqrt{N}$ array, as shown in Figure 1-5(a). A two-way communication line links P(i, j) to its neighbours P(i+1, j), P(i−1, j), P(i, j+1), and P(i, j−1). This network is also known as the mesh. If the processors at the edges are connected cyclically; one gets a two-dimensional lattice of Figure 1-5(b). There are N−1 links in this network and the maximum distance between diametrically opposite processors is N/2.

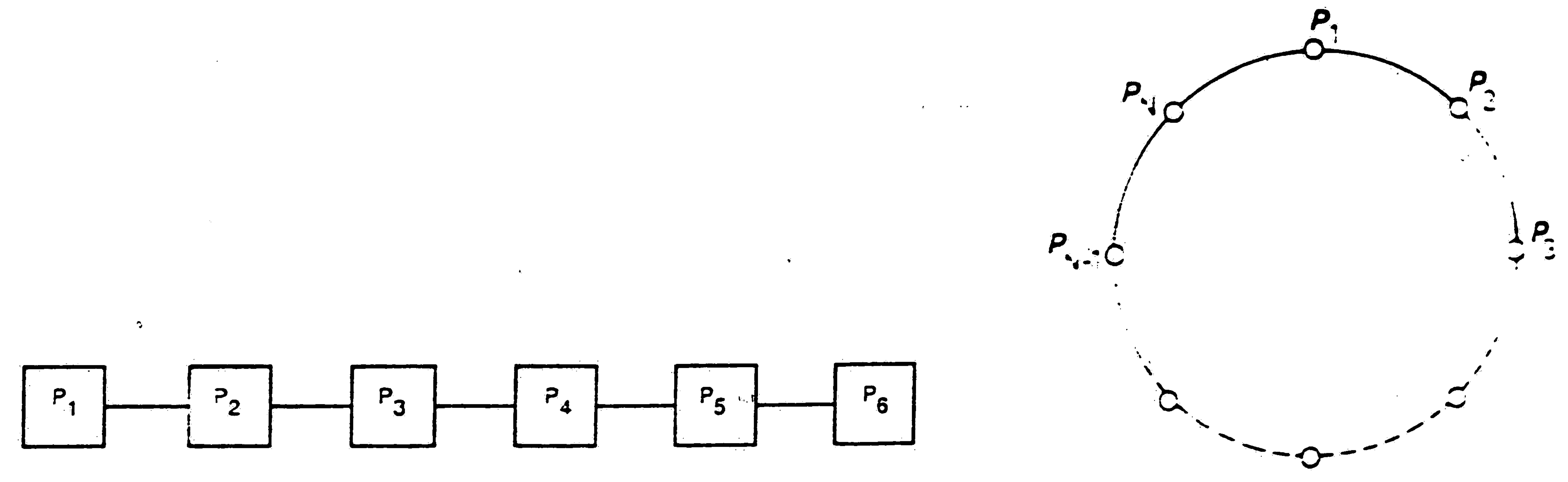


Figure 1-4. Linear Array and Cyclic Configurations.

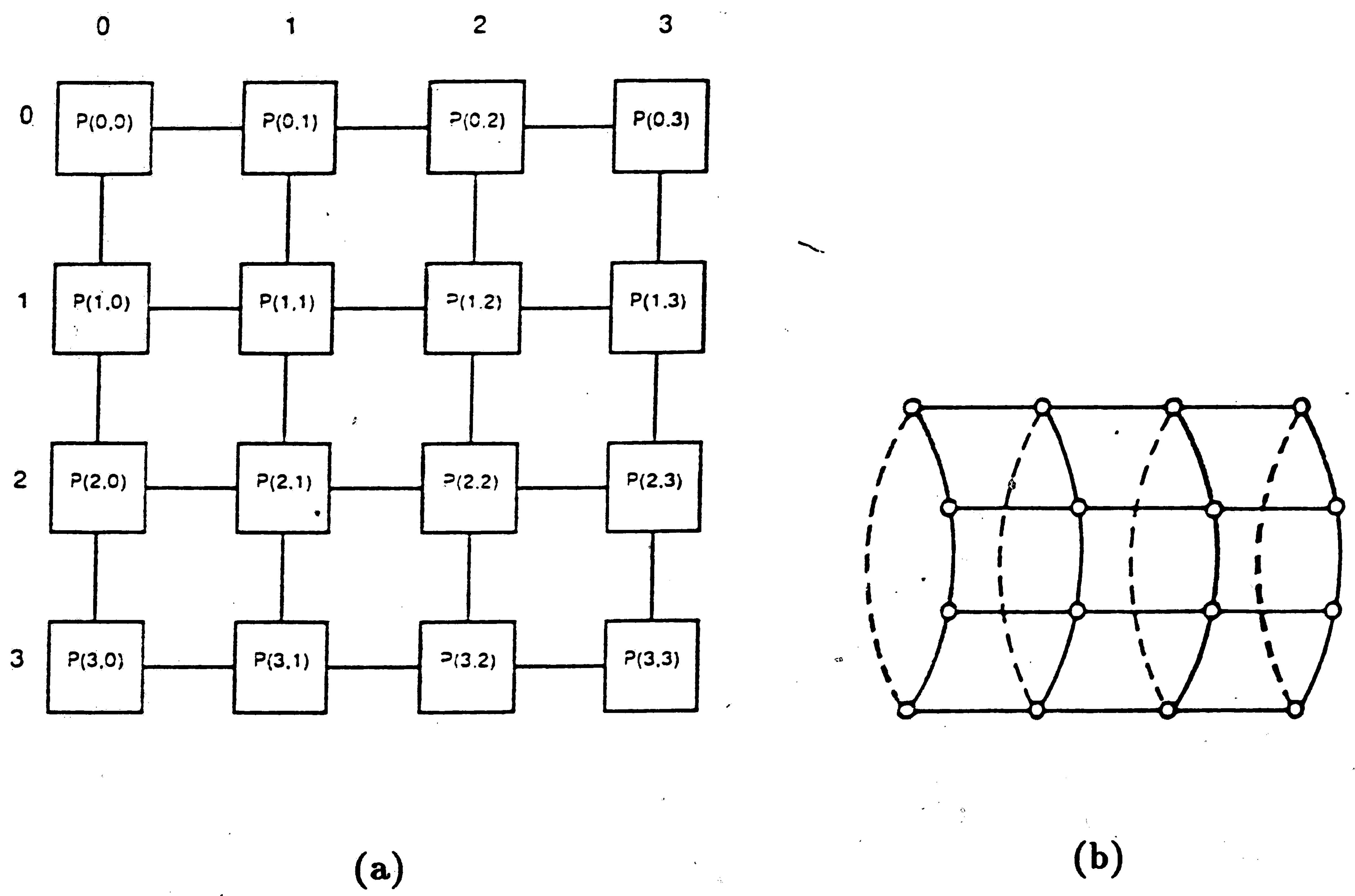


Figure 1-5. Two-dimensional Array and Lattice Mesh.

Tree Connection, [2] & [51]:

In this network, the processors form a complete binary tree. It has d levels, numbered 0 to d-1, and $N=2^d-1$ nodes each of which is a processor. Figure 1-6 shows a tree of 4 levels. One can see that in this configuration, each processor at level i is connected to its parent at level i+1 and its children at level i−1 by a bidirectional links. There are N−1 links in this network and the maximum distance between diametrically opposite processors is N/2.

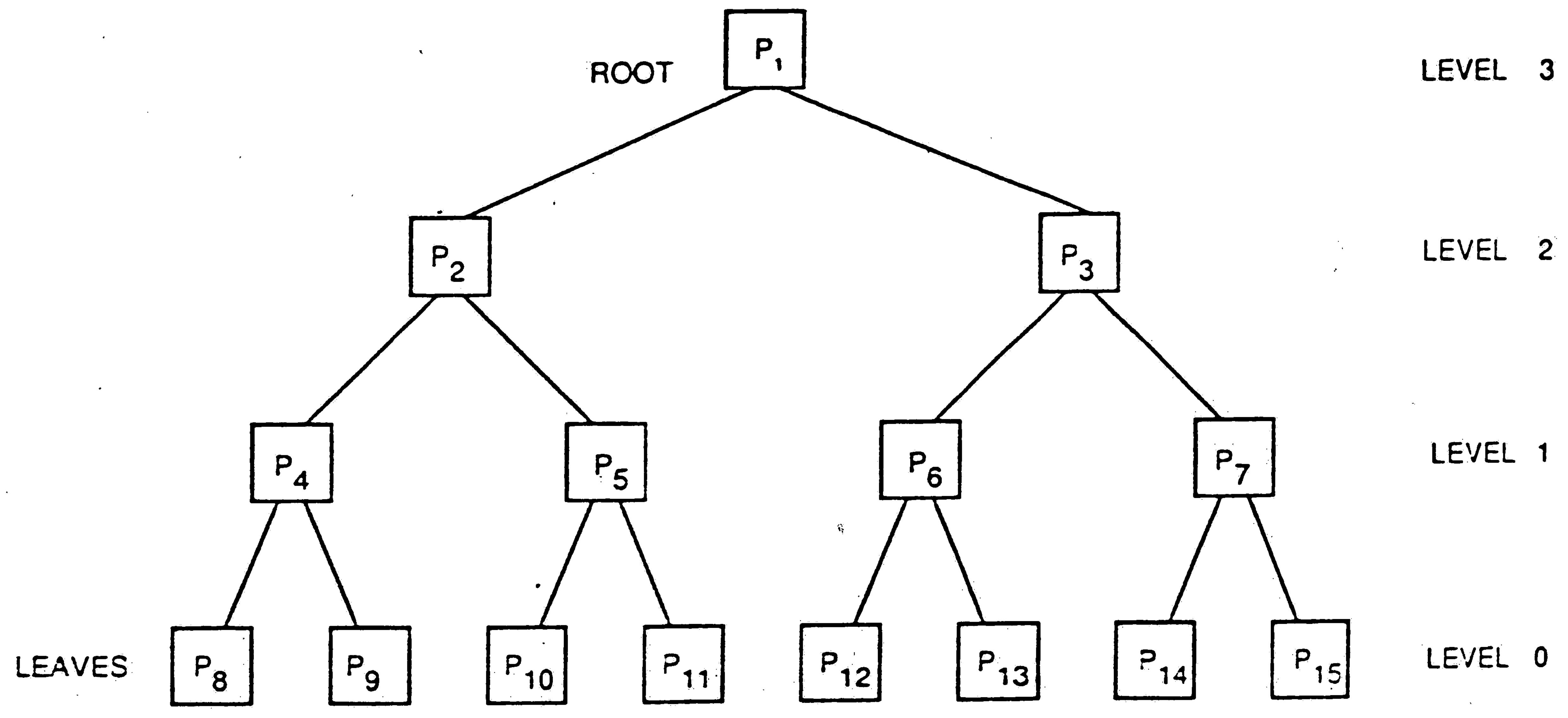


Figure 1-6. Tree Connection.

Perfect Shuffle, [2] & [51]:

In this topology, the number of processors $N = 2^m$ for some integer m. Figure 1-7 shows a perfect shuffle interconnection. In this network, unidirectional shuffle links (solid lines in the figure) go from processor P_i to processor P_j if

$$j=\begin{cases} 2i & \text{for } 0 \le i \le N/2-1, \\ 2i+1-N & \text{for } N/2 \le i \le N-1. \end{cases}$$

Additionally, bidirectional exchange links (dotted lines in the figure) link every even-numbered processor to its successor. There are N−1 links in this network and the maximum distance between diametrically opposite processors is N/2.

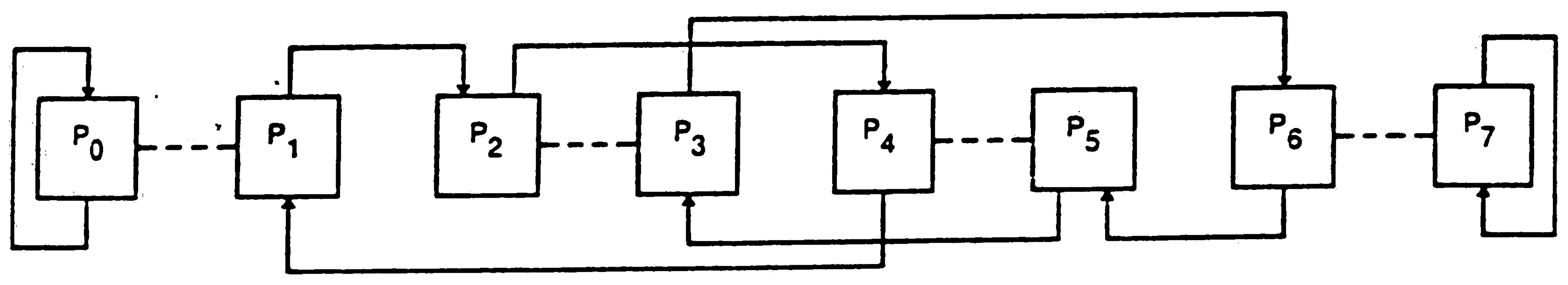


Figure 1-7. Perfect Shuffle.

Cube (Hypercube) Networks, [2] & [51]:

In this very popular network also, the number of processors is $N = 2^d$ where d is refered to as the degree of the cube. Two processors are connected here if and only if their binary representations differ in exactly one position. The 3-dimensional cube is shown in Figure 1-8. The processor indices are given in binary notation for simplicity. Parallel computers with hypercube topology of up to d = 10 are currently available commercially as integral parts of MIMD systems; higher dimensions will no doubt appear soon. There are N−1 links in this network and the maximum distance between diametrically opposite processors is N/2.

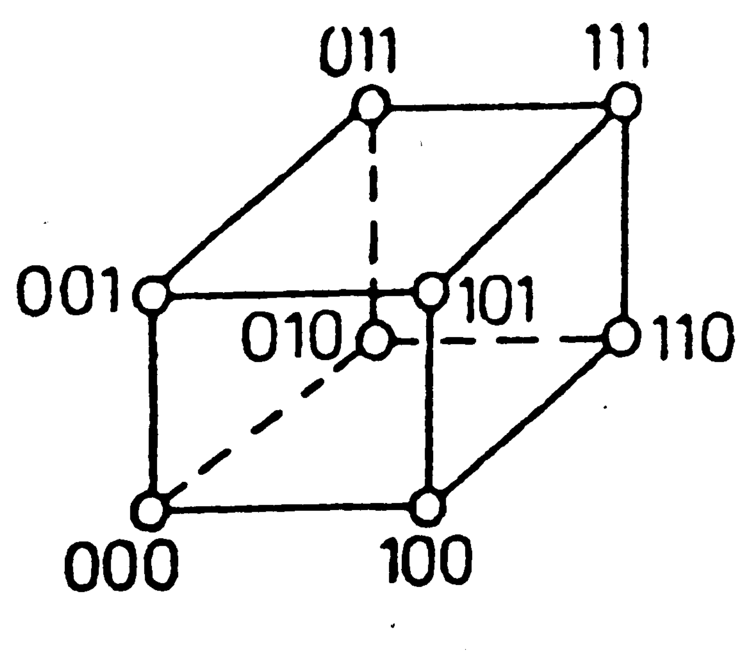


Figure 1-8. A Cube Network of dimension 3.

3. CHARACTERIZATION OF PARALLEL ALGORITHMS

Every parallel algorithm needs to be evaluated to its applicability: how much time and how many resources does it need, are these requirements minimal or worst case, and can the algorithm be restructured to use fewer resources and still have respectable running time. The following three criteria are the subject of this section: running time, number of processors, and the cost estimate.

Running time:

The running time is defined as the worst case total time taken by the algorithm to solve a problem on a parallel computer. Running time is the most important measure in evaluating a parallel algorithm. After all, higher computing speed is the main motivation for parallel computing. It is naturally estimated by counting the number of steps executed by the algorithm in the worst case. Two kinds of steps are usually considered in estimating the running time: computational and routing steps. The former is an arithmetic or logic operation performed on a datum within a processor. In the latter, a datum travels from one processor to another via shared memory or through the communication network.

Almost all existing problems have well-known lower bounds on the number of steps required to solve them in the worst case. If an algorithm executes a number of steps equal to the lower bound then it is called optimal (it is the fastest possible algorithm for the problem). The optimal algorithm is said to establish an upper bound on the number of steps required to solve that problem in the worst case. To determine the efficiency of a new algorithm, its running time is compared to the lower bound (to determine if it is the fastest algorithm for the problem) and to the upper bound (to compare it with other existing algorithms for the same problem). "Big O" notation is used for the upper bound. It is defined as follows: a function g(n) is said to be of order at most f(n), denoted by $O(f(n))$ if there are positive constants c and n_0 such that $g(n) \leq cf(n)$ for all $n \geq n_0$.

Another terminology used in evaluating a parallel algorithm is the speedup it produces. If T_s denotes the running time of the fastest known sequential algorithm, and T_p denotes the running time of a parallel algorithm using P processors, then the speedup, defined as $S_p = T_s/T_p$, measures the improvement in solution time using parallelism. Clearly, the larger the speedup, the better the parallel algorithm. A simple argument shows that $S_p \leq P$. The goal is to construct algorithms exhibiting linear (in P) speedup and hence utilizing the processors effeciently. However, linear speedup is not always possible. There are certain computations for which the maximal speedup is $S_p < d$ for a constant d independent of P, and such a computation clearly makes poor use of parallelism. For many important problems in linear algebra the best speedup is $S_p = c\ P/(\log P) - o(1)$, which is acceptable though less than linear.

Number of processors:

The second most important criterion in evaluating a parallel algorithm is the number of processors it requires to solve a problem. Due to the high cost of purchasing, maintaining, and running computers, it is preferred to keep the number of processors low. Therefore, the larger the number of processors an algorithm uses, the less desirable it becomes.

Cost:

One can intuitively see that the high speed and the low number of processors are conflicting requirements. Depending upon the application, one has to weigh one against the other. However, most commonly, the cost of a parallel algorithm is defined as the product of the parallel running time and the number of processors used. It is the number of steps executed collectively by all processors in solving a problem in the worst case. Another terminology related to the cost is the efficiency of a parallel algorithm. It is defined as $E_p = S_p / P$ where P is the number of processors used. Efficiency attempts to measure how well the processing power of the unit is being used. Usually $E_p \leq 1$; otherwise a faster sequential algorithm can be obtained from the parallel ones. In the case $E_p = 1$, it becomes unnecessary to choose the number of processors in order to maximize this function.

4. ORGANIZATION OF THE THESIS

The remainder of this thesis is organized in three chapters. Each chapter is devoted to the study of parallel algorithms for a fundamental computational problem area. The imortant subject of matrix computations and parallel algorithms is considered in Chapter 2. The essential algebraic topic of polynomials is discussed in Chapter 3. Finally, arithmetic computations are considered in Chapter 4.

This thesis is intended to provide a more complete and up-to-date survey of parallel algorithms for linear and algebraic problems. Original material from the papers and books surveyed has been included to create a unified treatment.

Chapter 2
PARALLEL ALGORITHMS FOR MATRIX COMPUTATIONS

1. INTRODUCTION

Numerical problems in Linear algebra such as calculation of determinants, matrix inversion, solutions of linear systems of equations, and eigenvalue expressions are of fundamental importance in scientific computing. This basic fact coupled with the onset of parallel architectures has implied a renewed interest in matrix calculations. In the past two decades, parallel algorithms for matrix computations have received strong emphasis from researchers in parallel computing.

This chapter surveys the parallel techniques for linear algebraic problems. Parallel algorithms to compute the determinant of a square matrix are discussed in Section 2. Section 3 describes parallel algorithms for finding the inverse of a square matrix. Parallel solutions to linear systems are related to matrix inversion and are also discussed in Section 3. These systems include the triangular, tridiagonal, and general dense systems. Finally, in Section 4, the problem of computing the characteristic polynomial of a square matrix is introduced.

2. DETERMINANTS

In this section, three parallel algorithms to compute the determinant of a square matrix are discussed. The concept of the determinant of a matrix is important in matrix theory. Aside from their effectiveness as a computational tool, determinants are important as a theoretical tool. For example, determinants provide a simple criterion for nonsingularity; a nonzero determinant. Determinants are used to derive conditions for the existence and uniqueness of solutions for systems of linear equations and are therefore very important in system theory. As a matter of fact, the notion of determinants has its origin in solving linear systems and today, most stability criteria in linear system theory involve use of determinants.

There are several equivalent ways to define the determinant of a matrix. The determinant is the function that maps square matrices into scalars. Let F be a field. Let $M_n(F)$ denote the set of all square matrices of order n over F. The determinant is defined by the mapping $\det : M_n(F) \rightarrow F$ such that for each $A = (a_{ij}) \in M_n(F)$, $\det(A) = \sum (\operatorname{sgn} \sigma) \; a_{1\sigma(1)} \, a_{2\sigma(2)} \cdots a_{n\sigma(n)}$, where the summation extends over all permutations σ of $\{1, 2, \cdots, n\}$. The functional value det(A) is called the determinant of the matrix $A = (a_{ij})$. Thus the determinant is a sum of n! terms where every row and every column is represented exactly once in each term of the sum.

Determinants satisfy several properties. Studying these properties leads to quicker means of computing the determinants. Following is a summary of these properties.

(P_1) If A^t is the transpose of a matrix A, then $\det(A^t) = \det(A)$.

(P_2) If A is a matrix with two identical rows (columns), then $\det(A) = 0$.

(P_3) If a row (column) of a matrix A consists entirely of zeros, then $\det(A) = 0$.

(P_4) If the matrix B is obtained from the matrix A by interchanging two rows (columns), then $\det(B) = -\det(A)$.

(P_5) If the matrix B is obtained from a matrix A by adding a scalar multiple of one row (column) to another row (column), then $\det(B) = \det(A)$.

(P_6) If the rows (columns) of a matrix A are linearly independent, then $\det(A) = 0$.

(P_7) If A is a traingular matrix (i.e., every element above or every element below the main diagonal is 0), then det(A) is the product of the elements on the main diagonal.

(P_8) For two matrices A and B, $\det(AB) = \det(A)\det(B)$.

(P_9) A matrix A is nonsingular if and only if $\det(A) \neq 0$.

(P_{10}) If A is a nonsingular matrix, then $\det(A)^{-1} = (\det(A))^{-1}$.

Computing the determinant has all along been thought of as only a sequential algorithm and several sequential algorithms to compute the determinant are available. Amongst these, the most popular technique is the Gauss's method. When the operation described in (P_5) is applied several times, the evaluation of the determinant can be reduced to that of a triangular matrix. Application of (P_7) then gives the determinant. This is the essence of Gauss's method. Another procedure that is effective for the purpose of evaluation of determinants consists of expressing a determinant in terms of those of lower order. This is in contrast to the method of Gauss. This method requires that one be familiar with the notion of a cofactor. Given a matrix $A = (a_{ij})$, the cofactor of the element a_{ij} is the scalar cof $a_{ij} = (-1)^{ij}\det(A_{ij})$, where A_{ij} is the matrix obtained from A by deleting its i-th row and j-th column. The matrix A_{ij} is sometimes called the minor of a_{ij} in A. The determinant can be written as follows: $\det(A) = \sum_{j=1}^{n} a_{ij} \text{ cof } a_{ij}$ for $i = 1, 2, \cdots, n$.

The adjoint of a matrix A can be defined as: $(\text{adj}(A))_{ij} = (-1)^{i+j}\det(A_{ji})$. It follows that if $\det(A) \neq 0$ then $(\text{adj}(A)) * A = A * (\text{adj}(A)) = I * \det(A)$, and adj(A) is unique.

A matrix A of order n×m is called Toeplitz if each diagonal has a value to which all the elements on that diagonal are equal.

In 1974, Csanky [24] gave the first parallel algorithm for fast determinant computation. Let A be a square matrix of order n. Let D_t denote an order t determinant, and let D_n be the determinant to be computed. Define $x_k = D_{n-k}/D_{n-k+1}$, where $1 \leq k \leq n-1$ and D_{n-k} is a properly chosen minor of D_{n-k+1}. Since $\prod x_k = D_1/D_n$ then $D_n = D_1/\prod x_k$. The following algorithm computes the determinant D_n:

Algorithm 2.1. Determinants over fields of characteristic 0.
Input: A square matrix, A, of order n.
Output: The determinant, D_n, of A.

1. *Compute x_k for $1 \leq k \leq n-1$ in parallel steps by solving the corresponding systems of equations.*

2. *Compute the product* $\prod x_k$ *for* $1 \leq k \leq n-1$ *by multiplying the* x_k*'s.*
3. *Divide the product of Step* 2 *by* D_1.

Denote by T(n) the parallel arithmetic complexity of computing det(A) (The parallel complexity of the computation is the least number of steps necessary to produce the result). It is easy to show that $2 \log n \leq T(n)$. To show that $T(n) \leq \log^2 n$: Step 1 requires $O(\log^2 n)$ steps. It will be shown later in the chapter that the solution to a system of n equations can be computed in $O(\log^2 n)$). Step 2 can be computed in at most $\log n + O(1)$ additional steps. Thus the determinant of a matrix can be computed in $O(\log^2 n)$ number of steps.

This method seems to require a division by n!, and therefore applies to fields of characteristic zero, but not to finite fields. Some applications such as factoring polynomials require an algorithm that works over arbitrary fields, in particular finite fields. Based on the general parallelization result by Valiant-Skyum-Berkowitz-Rackoff [67], Borodin-Gathen-Hopcroft [10] presented another algorithm for computing the determinant that works over arbitrary fields. The Borodin-Gathen-Hopcroft algorithm is asymptotic. It starts with the ordinary Gaussian elimination method performed on the matrix A of order n, with pivots chosen on the diagonal. This method computes the determinant of the matrix A sequentially in $O(n^3)$ steps of additions, multiplications, subtractions, and divisions. The algorithm consists of two steps which apply to any sequential computation to compute a polynomial $f \in F[a_1, a_2, \cdots, a_m]$ of degree n in time t. The algorithm may be stated as follows. A full description of the two steps follows the statement of the algorithm.

Algorithm 2.2. Determinants over arbitrary fields.
Input: **A square matrix, A, of order n.**
Output: **The determinant of A.**

1. ***Eliminate all divisions from the process.***
2. ***Apply the general parallelization result by Valiant-Skyum-Berkowitz-Rackoff*** **[67]** ***to parallelize the division-free algorithm.***

In the first step, the division can be avoided using Strassen's technique [64]. For each division g/h, we find $x_1, x_2, \cdots, x_m \in F$ such that $h(x_1, x_2, \cdots, x_m) \neq 0$. Since every rational function can be written as b/c where $b, c \in F[a_1, a_2, \cdots, a_m]$ and $\deg(b), \deg(c) \leq t2^t$, then the product d of all such denominators has degree less than or equal to $t2^t$. For every subset $P \subseteq F$ with $|P| > mt2^t$ there exists $x = (x_1, x_2, \cdots, x_m) \in P^m$ such that $d(x_1, x_2, \cdots, x_m) \neq 0$. There is a Monte Carlo algorithm to find such an x. After $x_1, x_2, \cdots, x_m$ are found, we shift the inputs by the negative of the x_i's, and consider new indeterminates $b_i = a_i - x_i$. Replacing every occurence of a_i by $b_i - x_i$, each division in the algorithm becomes a division by a rational function in $b_1, b_2, \cdots, b_m$ which has a nonzero value for $b_1 = b_2 = \cdots = b_m = 0$. These rational functions are invertible in the ring $R = F[b_1, b_2, \cdots, b_m]$, and $\bar{f} = f(b_1 + x_1, \cdots, b_m + x_m) \in R$ is a polynomial of degree n. Computing only the homogeneous part for each operation, one replaces every division by a multiplication in R. This yields a division-free sequential computation in R for $\bar{f}$

with time $O(tn^2)$. Back substituting, one gets an $O(tn^2)$-algorithm in $F[a_1, a_2, \cdots, a_m]$ that computes f without division. For the Gaussian elimination method we shift the matrix A by the negative of the identity matrix, and work on the new matrix $B = A - I$. Applying step 1 to the Gaussian elimination method yields a division-free straight-line algorithm that computes det(A) in time $O(n^5)$. The second step is to apply the Valiant-Skyum-Berkowitz-Rackoff [67] parallelization technique to obtain a parallel algorithm with parallel time $O(\log^2(tn))$ using a polynomial (in t and n) number of processors. Applying this step to the Gaussian elimination method gives a parallel algorithm for the determinant in time $O(\log^2 n)$ using $O(n^{15})$ processors. This algorithm works over finite fields, unlike Csanky's algorithm, Algorithm 2.1 above, which is useful only in fields of characteristic zero. The proof to this claim is as follows: Let F be a finite field, say of p elements. Take $u \in F[z]$, $\deg(u) = l$ such that $p^l \geq 2mt2^t$. Consider the extension field $G = F[z]/(u)$ of F. This field G has $2mt2^t$ elements. Apply a Monte Carlo procedure to find an appropriate $x \in G^m$. Each operation in G needs $O(l^2)$ operations in F. Thus the claim can be easily verified.

The above algorithm has several drawbacks. First it is not explicit. Secondly, the number of processors required is $O(n^{15})$. The reduction of this number was discussed by Berkowitz [6] in 1984. His new algorithm was based on Samuelson's method [60] which uses no divisions. This method relates the adjoint and the determinant of a matrix A in a very efficient way. Later in Section 2.4, the theorem and the parallel algorithm to find the characteristic polynomial of the matrix A will be discussed. The algorithm may be stated as follows:

Algorithm 2.3. Determinants using fewer processors.
Input: **A square matrix, A, of order n.**
Output: **The determinant of A.**

1. *Compute the characterestic polynomial $p(\lambda)$ of A.*
2. *Compute $p(0)$ to find the required determinant of the matrix A.*

This algorithm computes the determinant in time $O(\log^2 n)$ using $O(n^3)$ processors as it will be shown in Section 2.4.

3. LINEAR SYSTEMS

Solutions to general systems of equations require the study of matrix inversion and the study of solutions to more specific systems such as the triangular and tridiagonal systems. Subsection 3.1 studies the matrix inversion. The triangular systems and the tridiagonal systems are discussed in Subsections 3.2.1 and 3.2.2 respectively. In Subsection 3.3, the general dense systems are examined.

3.1 Matrix Inversion

One method to invert triangular matrices was presented by Heller [34] in 1974. The algorithm requires the factorization of the matrix A, which is

assumed to be of order n×n and a lower triangular matrix (i.e. all entries above the main diagonal are zeros). If A is not triangular, one may decompose it using efficient parallel LUD algorithms and then factorize it further as in this section. Let the matrix A be written as:

$$A = \begin{bmatrix} A_{11} & 0 \\ A_{12} & A_{21} \end{bmatrix}$$

where the submatrices A_{11} and A_{21} are lower triangular, and A_{12} is n/2×n/2. The algorithm given below can be used to compute A^{-1} which is written as

$$A^{-1} = \begin{bmatrix} B_{11} & 0 \\ B_{12} & B_{21} \end{bmatrix},$$

where $B_{11} = (A_{11})^{-1}$, $B_{12} = -(A_{21})^{-1} A_{12} (A_{11})^{-1}$, and $B_{21} = (A_{21})^{-1}$. Thus to calculate A^{-1}, first $(A_{11})^{-1}$ and $(A_{21})^{-1}$ are computed in parallel. Two matrix multiplications then give B_{12}. The algorithm may be stated formally as:

Algorithm 2.4. Triangular matrix inversion.
Input: A square lower triangular matrix, A, of order n.
Output: The inverse matrix of A, A^{-1}.

```
1. Compute $B_{11} = (A_{11})^{-1}$ and $B_{21} = (A_{21})^{-1}$ in parallel.
2. Compute $Y = (A_{21})^{-1} \times A_{12}$.
3. Compute $B_{12} = Y \times (A_{11})^{-1}$.
```

One can easily see that by using this parallel algorithm, the time to compute A^{-1} will be less than or equal to the time to compute the inverse of an

$n/2 \times n/2$ matrix + the time for two matrix multiplications. Denoting by T(n) the parallel arithmetic complexity of inverting order n matrices, one has $T(n) \leq T(n/2) + \lceil 2 \log n \rceil$, giving $T(n) = O(\log^2 n)$. Note, however, that to get the $n/2 \times n/2$ matrix multiplication time down to $\lceil \log n \rceil$, one has to use $n^3/8$ processors. This is easily explained: there are $n^3/8$ multiplications involved in this matrix product. All of them can be performed simultaneously in one time unit on $n^3/8$ processors. $n/2$ of these products are then added to an element of the product matrix. By using divide-and-conquer, this requires only $\lceil \log n/2 \rceil = (\lceil \log n \rceil - 1)$ time. Since calculations corresponding to distinct elements of the product matrix can be done concurrently, the total time is only $\lceil \log n \rceil$.

Csanky [23] also developed an algorithm to invert a triangular matrix using $O(n^3)$ processors.

Let A be a square matrix of order n. The inverse of A exists if only if it is nonsingualr (i.e. it has a nonzero determinant). Let $\lambda_1, \lambda_2, \cdots, \lambda_n$ be the roots of the characteristic polynomial p(A) of A. Let s_k be defined for $1 \leq k \leq n$ as: $s_k = \sum_{i=1}^{n} (\lambda_i)^k$. If tr(A) denotes the trace of the matrix A, then $s_k = tr(A^k)$. Further, if S deontes the $n \times n$ lower triangular matrix with elements 1, 2, $\cdots$, n on the main diagonal and s_i on the i-th diagonal, $C = [c_1, \cdots, c_n]^T$, $s = [s_1, \cdots, s_n]^T$ then from the Newton's identities on gets $SC = -s$.

Using the above ideas, Csanky [23] developed a parallel algorithm to compute the inverse of a nonsingular matrix in time of $O(\log^2 n)$. It uses polynomial number of processors. The algorithm may be described as follows:

Algorithm 2.5. Matrix inversion.

Input: A square matrix, A, of order n.

Output: The inverse matrix of A, A^{-1}.

1. *Compute* s_k *for* $1 \leq k \leq n$, *and* $s_k = tr(A^k)$.
2. *Invert the triangular matrix S.*
3. *Compute* c_i *for* $1 \leq i \leq n$ *from* $c = -S^{-1}s$.
4. *If* $c_n \neq 0$, *compute* A^{-1} *as follows using the Cayley-Hamilton Theorem:*

$$A^{-1} = -\frac{1}{c_n}\left(A^{n-1} + c_1A^{n-2} + \cdots + c_{n-1}I\right)$$

To determine the running time and the number of processors used in the given algorithm, one may proceed as follows. Firstly note that $\lceil \log n \rceil + 1$ steps are necessary and sufficient to multiply two n×n matrices using n^3 processors. Using the technique of evaluating x^i in $\lceil \log n \rceil$ steps using n/2 processors, for $1 \leq i \leq n$, one may compute A^k, for $1 \leq k \leq n$, in $\lceil \log n \rceil \left(\lceil \log n \rceil + 1\right)$ steps using $n(n^3)/2$ processors. Thus, Step 1 takes $\log^2 n + O(\log n)$ steps using $n^4/2$ processors. In Step 2, S^{-1} can be found in $O(\log^2 n)$ steps using $O(n^3)$ processors, since S is a triangular matrix. (Algorithm 2.4 can be used for this purpose). From the equation $c = -S^{-1}s$, the c_i's in Step 3 can be computed in $\lceil \log n \rceil + 1$ steps using n^2 processors. Finally, since $A^2, \cdots, A^{n-1}$ are already available, Step 4 requires $\lceil \log n \rceil + 1$ steps to complete using $O(n^3)$ processors. Thus, $T(n) \leq 2 \log^2 n + O(\log n) = O(\log^2 n)$ and the number of the processors used is less than or equal to $n^4/2$ processors.

The above result by Csanky is of great theoretical value, but is difficult to implement in practice because of the excessive number of processors used.

3.2 Algorithms for Structured Matrices

3.2.1 Triangular Systems

Let A be an n×n triangular matrix. Without loss of generality, it can be assumed that A is a lower triangular matrix. Heller [34] gave the first parallel algorithm to solve the system AX = B. His algorithm solved the system in $O(\log^2 n)$ steps using $O(n^4)$ processors. The algorithm used an expansion theorem for the determinant of a Hessenburg matrix, and was complicated. Later, the number of processors was improved to $O(n^3)$ processors. One may note that the original Heller's algorithm [34] is no better than the generalized inversion algorithm of the last section.

In 1975, Chen-Kuck [18] developed an interesting algorithm using recursion and doubling. This algorithm is a variation of Gauss-Jordan elimination method. The algorithm is applied to the augmented matrix of A rather than A itself (the augmented matrix of A, denoted by aug(A), is obtained by adding the column B to A). The algorithm uses the row operations to eliminate the entries of each diagonal below the main diagonal in one step, so it can be called elimination by diagonal. The algorithm may be described as follows:

Algorithm 2.6. Triangular systems solver (I).
Input: A triangular system of equations, AX = B.
Output: A solution X for the system.

1. *Divide each row of the matrix aug(A) by* a_{ii}.
2. *Eliminate the entries below the main diagonal by*

eliminating the entries of each subdiagonal using the following loop:

```
FOR $j=1$ STEP $j$ UNTIL $n-1$ DO
    row $i \leftarrow$ row $i - \sum_{k=j}^{2j-1} a_{i,\, i-k}\, row(i-k)$, for
    $j+1 \leq i \leq n$.
```

3. *Find* $x_i = a_{i,\, n+1}$.

The algorithm takes $O(\log^2 n)$ steps to solve the linear system. Step 1 can be performed in one step using n^2 processors where each processor will work on one element of aug(A). The multiplications in Step 2 can be done in parallel for each j, followed by log sum addition of j+1 rows, if $n^2(n+1)/2$ processors were used. Thus the total time is,

$$1 + \sum_{k=0}^{N-1} 1 + \lceil \log (2^k+1) \rceil = \frac{N^3 + 3N + 2}{2} = O(\log^2 n) \text{ where } N = \lceil \log n \rceil.$$

Since, the matrix is upper triangular, it contains $\sum_{m=1}^{n-1} (n-m)$ zero entries. This means that unnecessary multiplications are performed in Step 2. Considering this, and using the theorem proved by Kuck [43] in 1978 (the statement of the theorem will follow later in the section), only $n^3/68 + O(n^2)$ processors are sufficient.

If the matrix A is Toeplitz, then the number of processors is further reduced to $O(n^2/4)$.

Another parallel algorithm to solve a triangular system uses the idea of inverting a triangular matrix explained in Section 3.1 above and can be described as follows:

Algorithm 2.7. Triangular systems solver (II).

Input: **A triangular system of equations, AX = B.**

Otput: The solution X.

1. *Compute A^{-1} using Algorithm* 2.4.
2. *Compute* $X = A^{-1} \times B$.

Since A^{-1} can be computed in $O(\log^2 n)$ steps using $O(n^3)$ processors, and since Step 2 contains only one matrix and vector product which can performed in $O(\log n) + O(1)$ steps using $O(n^2)$ processors, the total time for the algorithm is $O(\log^2 n)$ using $O(n^3)$ processors.

If A has a unit diagonal (i.e. the entries of the main diagonal are 1's), then A can be written as $A = I - L$ where I is the identity matrix, and L is strictly lower triangular. Moreover,

$A = I + L + L^2 + \cdots + L^{n-1} = (I + L^{2n-1})(I + L^{2n-2}) \cdots (I + L)$.

A method that uses this idea to solve the system AX = B was presented by Orcutt [54] and Heller [34] independently and can be described as follows:

Algorithm 2.8. Unit diagonal systems solver.

Input: A system of equations, AX = B, where A has a unit diagonal.

Otput: The solution X.

1. *Divide each row of the matrix A by a_{ii} to make A a unit diagonal matrix.*
2. *Compute A^{-1} by repeatedly squaring the matrix L as above.*
3. *Multiply $A^{-1}B$ to get X.*

Step 1 of this algorithm can be performed in parallel in one step using $(n-1)^2/2$ processors by letting each nonzero matrix element be modified in a distinct processor. Step 2 consists of computing L^2 repeatedly, which can be done in parallel in $\lceil \log n \rceil$ steps. Each square itself takes $\lceil \log n \rceil + 1$ steps using n^3 processors. Thus Step 2 requires $\lceil \log n \rceil \left(\lceil \log n \rceil + 1 \right)$ steps. Finally, Step 3 is performed in $\lceil \log n \rceil + 1$ steps using n^2 processors. Thus the algorithm can be performed in at most $\lceil \log n \rceil^2 + \lceil \log n \rceil$ steps using $n^3 + n^2$ processors.

The following two algorithms to solve an $n \times n$ triangular system demonstrate the use of linear recurrence. A linear recurrence system $R(n, m)$ of order m for n equations is defined as:

$$x_k = \begin{cases} 0 & \text{if } k \leq 0 \\ b_k + \sum_{j=k-m}^{k-1} a_{kj} x_j & \text{if } 1 \leq k \leq n, \text{ and } m \leq n-1 \end{cases}$$

Equivelantly, if $A = [a_{ik}]$ where $a_{ik} = 0$ for $i \leq k$ or $i - k > m$, and $X = [x_1, \cdots, x_n]^t$, and $B = [b_1, \cdots, b_n]^t$ then the above definition can be written as $X = AX + B$.

The first of the two algorithms that use the idea of linear recurrence is called the Column-Sweep algorithm. It can be stated as follows.

Algorithm 2-9. Column-Sweep.
Input: A triangular system of equations, $AX = B$.
Output: The solution X.

1. *Evaluate in parallel the expressions of the form* $b_i^{(1)} = b_i + a_{i1} x_1$ *for* $i = 2, \cdots, n$ *where* $x_1 = b_1$ *is known. (Notice that only* $n-2$ *equations are left after*

this step).

2. *Evaluate in parallel the expressions of the form $b_i^{(2)} = b_i^{(1)} + a_{i2}\, x_2$ for $i = 3, \cdots, n$ where x_1 and x_2 are known. (Notice that only $n-3$ equations are left after this step).*

   ⋮

k. *Evaluate in parallel the expressions of the form $b_i^{(k)} = b_i^{(k-1)} + a_{ik}\, x_k$ for $i = k+1, \cdots, n$ where $x_1, \cdots, x_k$ are known.*

Clearly n−1 steps are required using n−1 processosrs at the first step, and fewer than that in the subsequent steps. So $O(n)$ steps and $O(n)$ processors are necessary and sufficient to solve a traingular system using the column-sweep algorithm.

The second algorithm is the recurrent-product algorithm presented by Sameh and Brent [58] in 1977. The algorithm proceeds by writing the equation $X = AX + B$ (as defined above) as $X = (I - A)^{-1} B$. The idea of Householder [36] can be used to express $(I-A)^{-1}$ as $(I-A)^{-1} = \prod_{i=1}^{n-1} M_{n-i}$, where M_i is defined as:

$$M_i = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & \mathbf{0} & \\ & & 1 & & & \\ & & a_{i+1,\,i} & 1 & & \\ \mathbf{0} & & \vdots & & \ddots & \\ & & a_{n,\,i} & \mathbf{0} & & 1 \end{bmatrix}$$

The problem is reduced to that of matrix multiplication which can be evaluated in parallel time of $O(\log^2 n)$ steps using $O(n^3)$ processors. This is, however, not practical for large n.

In 1978, Kuck [43] proved a useful theorem that can be stated as: A linear recurrence system R(n, m) can be evaluated on p processors in T_p steps where

$T_p < (2 + \log m) \log n - \frac{1}{2}(1 + \log m) \log m$

and

$p < \frac{1}{2} m (m + 1) n + O(m^3)$ for $1 \leq m \leq n/2$

and

$p < \frac{n^3}{68} + O(n^2)$ for $n/2 \leq m \leq n-1$

3.2.2. Tridiagonal Systems

Stone [63] was the first to discuss the solution to a system $AX = B$ where A is a tridiagonal matrix. Using recursive doubling algorithms, Stone related the LUD decomposition of A to a first and second recurrences. These algorithms compute the necessary terms in $O(\log n)$ time using n processors, assuming no pivoting is necessary. Let $L = (l_i, 1, 0)$, $D = (0, d_j, 0)$, and $U = (0, 1, u_j)$ be an LUD factorization of A. It is easy to see that $d_1 = b_1$, $d_j = b_j - a_j c_{j-1}/d_{j-1}$ where $2 \leq j \leq n$, $l_j = a_j/d_{j-1}$ where $2 \leq j \leq n$, $u_j = c_j/d_j$, where $1 \leq j \leq n$. Now L and U are completely determined by D. To compute D, define $p_0 = 1$, $p_1 = b_1$, $p_j = b_j p_{j-1} - a_j c_{j-1} p_{j-2}$, and $d_j = p_j/p_{j-1}$. Now $AX = B$ is solved by solving $LW = B$ and $UX = D^{-1}W$. The bidiagonal systems represent first order recurrences, and $D^{-1}W$ is computable in one parallel step using n processors.

The above algorithm fails if pivoting is necessary. Another decomposition for A that can be used is the QR decomposition discussed in Section 3.3. Consider the following algorithm:

```
Algorithm 2.10. Odd-even elimination.
Input: A tridiagonal system of equations, AX = B.
Output: The solution X.

1. For $k = 1$ step $k$ until $n$-1 do
      row $i$ ← row $i$ − $a_{i,\ i-k}$ (row $i-k$)/$a_{i-k,\ i-k}$ −
      $a_{i,\ i+k}$ (row $i+k$)/$a_{i+k,\ i+k}$  where $1 \leq i \leq n$.
2. $x_i \leftarrow a_{i,\ n+1}/a_{ii}$   $1 \leq i \leq n$.
```

If the loop in Step 1 was applied on the tridiagonal matrix A which has three nonzero diagonals, then these diagonals move further and further apart as the loop progresses. The result will be a diagonal matrix. If Step 2 is executed, the result is a solution to the tridiagonal system AX = B.

The above algorithm takes O(n log n) steps using n processors. This algorithm is known as the odd-even elimination which has a variation called odd-even reduction. This later algorithm generates a sequence of tridiagonal systems $A^{(i)}X^{(i)} = B^{(i)}$, each is half the size of the previous one and formed by eliminating the odd-indexed variables and saving the even-indexed variables. $X^{(i)}$ is obtained by back substitution to obtain $X^{(0)}$ which is the solution to the original problem.

Algorithm 2.11. Odd-even reduction.

Input: A tridiagonal system of equations, AX = B.

Output: The solution X.

```
1. For $k = 1$ step $k$ do
   row $i \leftarrow$ row $i - a_{i,\ i-k}(\text{row } i-k)/a_{i-k,\ i-k} -$
   $a_{i,\ i+k}(\text{row } i+k)/a_{i+k,\ i+k}$    $(i = 2k, 4k, \cdots, 2^n-2k)$;
2. For $k = 2^{n-1}$ step $-k/2$ until 1 do
   $x_i \leftarrow (a_{i,\ n+1} - a_{i,\ i-k}x_{i-k} - a_{i,\ i+k}x_{i+k})/a_{ii}$
   $(i = k, 3k, \cdots, 2^n-k)$
```

Only O(n) operations are performed in this algorithm as against O(n log n) in odd-even elimination. Another advantage of this algorithm is that it is equivalent to Gaussian elimination applied to PAP^T where P is a particular permutation matrix.

3.3 General Systems:

In 1974, Csanky [24] showed that parallel solution of any system AX = B can be obtained by inversion of A and then multiplying by B has a complexity T(n) which satisfies $2(\log(n)) \leq T(n) \leq O(\log^2 n)$. The number of processors used in the algorithm is polynomial in n.

One of the most familiar sequential algorithms to solve AX = B, which is suited to parallel computation, is the Gauss-Jordan elimination method. Let aug(A) be the augmented matrix of A, row(i) will be used to refer to the ith row of aug(A). Gauss-Jordan elimination method can be described by the following algorithm under the assumption that pivoting is not necessary. The algorithm eliminates the elements above the diagonal as well as below. So this method reduces the system quickly to a diagonal form.

Algorithm 2.12. Gauss-Jordan elimination (I).
Input: A system of equations, AX = B.
Output: The solution X.

```
1. For $1 \leq j \leq n$ compute
   $row(i) \leftarrow row(i) - a_{ij}/a_{jj}\ row(j)$ for $1 \leq i \leq n$ and $i \neq j$.
2. Compute $x_{ij} = a_{ij}/a_{ii}$ for $1 \leq i \leq n$, and $j = n + 1$.
```

For each j, there are n+1 multiplications, n divisions, and n+1 subtractions. Each group of operations can be performed in parallel in one step using at most n+1 processors. So Step 1 needs 3n steps using (n−1)(n+1) processors (since i = j is excluded). Step 2 needs only one step in pararllel and uses n processors. Thus using (n−1)(n+1) processors, the algorithm can be done in 3n+1 time units. However, if only n processors are available, the algorithm requires $n^2 + 2n + 1$ steps. This is because the above algorithm is altered as shown below. This illustrates the tradeoff between the number of processors used and the time required.

Algorithm 2-13. Gauss-Jordan elimination (II).
Input: A system of equations, AX = B.
Output: The solution X.

1. *For* $1 \le i \le n$ *and* $1 \le j \le n$ *compute* $t_i = a_{ij}/a_{jj}$.
2. *For* $j+1 \le k \le n+1$ *compute*

   $a_{ik} = a_{ik} - t_i a_{jk}, \ 1 \le i \le n, \ i \ne j.$

The disadvantage of the Gauss-Jordan elimination method is that it may sometimes prove numerically unstable, and some form of pivoting (such as column pivoting) should, therefore, be incorporated. If $a_{jj} = 0$ at some point in the algorithm, then $\lceil \log (n-1) \rceil$ additional steps are needed to find a nonzero pivot below the diagonal in column j.

Another method to solve a system of equations AX = B uses the LU decomposition. If there is a nonsingular lower triangular matrix L, and an upper triangular matrix U such that A = LU, then this is known as an LU decomposition of A.

When such a factorization is known, solving the linear system AX = B is relatively quick and simple. The system LY = B is solved for Y first. Since L is nonsingular then there is a unique solution vector Y, which is easily calculated (L is a lower triangular matrix). Then the equation UX = Y is solved to obtain the solution of AX = B. The derivation of X from Y is also simple because U is a triangular matrix.

An n×n matrix may be factorized as A = Q R where Q is an n×n orthogonal matrix and R is an n×n upper triangular matrix. Square-root-free Givens transformations can be applied to find Q and R. If Q is found, then R can be determined easily by calculating $Q^t A = R$ since $Q^t Q = I$ as Q is an orthogonal matrix. Q is computed implicitely as a product of simpler matrices.

As a matter of fact, Q is the product of a number of plane rotations, each of which eliminates an element of A below the diagonal without destroying the previously introduced zeros. The following loop can be used to achieve this goal. For notational purposes, Rotate(i, j) applies root-free Givens transformations to rows i and $i-1$ in order to eliminate the element a_{ij} where $1 \leq j < i \leq n$.

```
For $k = 1$ to $n-1$ do
   begin
      Rotate $(n-2p, k-p)$        $(0 \leq p \leq min(k-1, n-k-1))$
      Rotate $(n-2p-1, k-p)$      $(0 \leq p \leq min(k-1, n-k-2))$
   end.
```

Thus, in order to eliminate a_{ij} the i-th and the $(i-1)$-th rows are multiplied by

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}$$

where $c = \cos\theta$ and $s = \sin\theta$, the angles of rotation being determined by using the formulae

$$s = \frac{a_{ij}}{\sqrt{(a_{ij})^2 + (a_{i-1,\,j})^2}} \quad \text{and} \quad c = \frac{a_{i-1,\,j}}{\sqrt{(a_{ij})^2 + (a_{i-1,\,j})^2}} .$$

Several of these rotations can be applied simultaneously in parallel. Now to solve the system $A X = B$ where $A = Q R$, (the Q R decomposition), we proceed by writing $Q R X = B$, so $R X = Q^t B$. Computing $Y = Q^t B$ reduces the linear system to a triangular one, then $R X = Y$ may be solved using any of the techniques discussed in Section 3.2.

Pease [55] presented an interesting algorithm to solve a general system of equations. Any parallel computer with an interprocessor communication

network designed for FFT can implement it reasonably well.

Algorithm 2.14. General system solver.

***Input*: A system of equations, AX = B.**

***Output*: The solution X.**

```
Procedure P(n)
Begin
        Let $A = \begin{bmatrix} A_1 & E_1 \\ E_2 & A_2 \end{bmatrix}$,
            $X = (x_1\ x_2)^t$,
                $B = (b_1\ b_2)^t$ where $A_1$, $A_2$ are $2^{n-1} \times 2^{n-1}$.
    Solve $A_1(F_1, g_1) = (E_1, b_1)$ and
          $A_2\ (F_2, g_2) = (E_2, b_2)$
          in parallel by applying $P(n-1)$.
    Solve $(I - F_1F_2)\ x_1 = (g_1 - F_1g_2)$ and
          $(I - F_2F_1)\ x_2 = (g_2 - F_2g_1)$
          in parallel by applying $P(n-1)$.
end.
```

This algorithm requires $O(n^2 \log n)$ steps using n processors.

Several authors have discussed solutions of linear systems of equations. Among those one finds Boroding-Gathen-Hopcroft [10]. In their paper, they showed that Csanky's result works for any finite field. In 1984, Bini [9] used the concept of approximate algorithm to show that 6 log n + 6 parallel steps and 2n processors suffice to approximate, with any precision, the solution of a linear system with n×n triangular Toeplitz matrix A. Moreover, 7 log n + 7 steps are sufficient for an exact computation, whereas the number of processors is increased to $5n^2/2$. The inverse of A can be approximated with any precision by $6 \log n + 3$ steps and 2n processors. The author gave two applications of these results. First, if B is any matrix belonging to the algebra generated by a given $n \times n$ matrix over the Complex Numbers field, then the system $BX = b$ can be solved in no more than $9 \log n + 4$ steps using $O(n^2)$ processors. Second, given a Toeplitz matrix $A = (a_{ij})$ such that $a_{ij} = 0$ if $i-j > k$ or $j-i > h$, $a_{k1} \neq 0$ then the system $AX = b$ can be solved in $13 \log n + O(\log^2 k)$ steps using $\max \left\{ (5/2)\, n\, (k+h),\ n(n+1)/2 \right\}$ processors.

4. THE CHARACTERISTIC POLYNOMIAL OF MATRICES

The characteristic polynomial of a matrix A is the equation $\det(A - \lambda I)$, where I is the identity matrix of the same order of A. Eigenvalues are the roots of the characteristic polynomial and satsify $AX = \lambda X$ for some vector $X \neq 0$. Each X is called an eigenvector of A corresponding to eigenvalue λ. $AX = \lambda X$ holds if and only if $A - \lambda I$ is singular. Thus if the eigenvlaues are precisely the roots of the characteristic equation $\det(A - \lambda I) = 0$ denoted by $\lambda_1, \cdots, \lambda_n$ with multilpicity m_λ of any eigenvalue λ equal to that of the factor $\lambda - X$ of the X-polynomial $\det(A - X)$ I.

Let A be an $n \times n$ matrix and let R, S, and M be three of its sub-matrices of order $(n-1) \times 1$, $1 \times (n-1)$, and $(n-1) \times (n-1)$ respectively, as follows:

$$A = \begin{bmatrix} a_{11} & R \\ S & M \end{bmatrix}$$

The characteristic polynomials of A and M, defined above, are:

$$p(\lambda) = \det(A - \lambda * I) = \sum_{i=0}^{n} p_{n-i}\, \lambda^i,$$

and

$$q(\lambda) = \det(M - \lambda * I) = \sum_{i=0}^{n-1} q_{n-i-1}\, \lambda^i,$$

respectively.

In 1984, Berkowitz [6] proved that the characteristic polynomial of a matrix can be computed in $O(\log^2 n)$ steps. Before his main result can be discussed, four claims need to be stated. The proof of Calim 1 follows by expanding the det(A $-$ λ*I) by cofactors along the first row, and then along the first column. Claim 2 is obvious because the matrix M must satisfy its characteristic polynomial. The proof of Claim 4 can be found in Aho-Hopcroft-Ullman [1]. Only the proof to Claim 3 is presented here.

Claim 1: $p(\lambda) = (a_{11} - \lambda) * \det(M - \lambda * I) + R * \mathrm{adj}(M - \lambda * I) * S$.

Claim 2: $\mathrm{adj}(M - \lambda * I) = -\sum_{k=2}^{n} \left(M^{k-2} * q_0 + \cdots + I * q_{k-2}\right) * \lambda^{n-k}$.

Claim 3: Let R, M, and S be as defined above. Let $T = \left\{R * M^i * S\right\}_{i=0,\cdots,m}$.

Then T can be computed in time $O(\log^2 n)$ and a circuit size $O(n^{\alpha+\epsilon})$ where α is the exponent of n for the size of a circuit for multiplying two matrices using log n depth and ϵ is any positive real number. Currently $\alpha < 2.496$.

<u>Proof:</u> Any element of T can be computed as the dot product of vectors from U

$= \left\{R*M^i\right\}_{i=0,\cdots,n^{0.5}}$ and $V = \left\{M^{i*n^{0.5}}*S\right\}_{i=0,\cdots,n^{0.5}}$ since the exponent of k of the M term in any element from T can be uniquely expressed in the form $k = i + j * n^{0.5}$. Since each dot product can be computed in $O(\log n)$ time using $O(n)$ processors, then T can be computed from U and V in time $O(\log n)$ using $O(n^2)$ processors. It can be proved by induction on β that $U_\beta=\left\{R*M^i\right\}_{i=0,\cdots,n^\beta}$ can be computed in time $O(\log^2 n)$ using $O(n^{\alpha+\epsilon})$ processors for β a constant. Now, U can be computed in time $O(\log^2 n)$ using $O(n^{\alpha+\epsilon})$ processors since $U = U_{0.5}$. V can be computed similarily.

Claim 4: The product of two Toeplitz lower triangular matrices is also Toeplitz and lower triangular. Moreover, it can be computed in time $O(\log n)$ using $O(n^2)$ processors.

The main result of Berkowitz [6] can be stated as follows:

$\forall\ \epsilon > 0$, the coefficients of the characteristic polynomial can be computed in time $O(\log^2 n)$ in size $O(n^{\alpha+1+\epsilon})$ circuit.

To prove the theorem, define lower traingular Toeplitz matrices C_i of order $(n-t+1)\times(n-t)$ by:

$$C_{t\ 1,\, i} = \begin{cases} -1 & \text{if } i = 1, \\ \alpha_{tt} & \text{if } i = 2, \\ -R_t*M_t{}^{i-3}*S_t & \text{if } i > 2. \end{cases}$$

Samuelson's method relates the characteristic polynomilas of A and M. Using this method we have a linear relation between the coefficients of the two characteristic polynomials: $(p_0, p_1, \cdots, p_n)^t = C^1 * (q_0, q_1, \cdots, q_{n-1})^t$. Applying this recursively, we get the direct product of C_i for $1 \le i \le n$. The entries of the matrices $\{C_i\}$ can be copmuted by applying Claim 3 n-times in

$O(\log^2 n)$ time in size $O(n^{\alpha+1+\epsilon})$. Using Claim 4, the characteristic coefficients can be computed from $\{C_i\}$ with a balanced binary tree of matrix multiplies in time $O(\log^2 n)$ in size $O(n^3)$.

Chapter 3
PARALLEL ALGORITHMS FOR POLYNOMIAL COMPUTATIONS

1. INTRODUCTION

A polynomial over an algebraic system is an expression of the form

$$f(x) = f_n x^n + f_{n-1} x^{n-1} + \cdots + f_1 x + f_0,$$

where the coefficients $f_n, f_{n-1}, \cdots, f_1, f_0$ are elements of this algebraic system, and the variable x may be regarded as a formal symbol with an indeterminant value. It will be assumed that the algebraic system is a ring. This means that it admits the operations of addition, subtraction, and multiplication; satisfying the customary properties: addition and multiplication are associative and commutative binary operations, with well-defined identities; multiplication distributes over addition; and subtraction is the inverse of addition. The additive identity element is denoted by 0 giving $a + 0 = a$, and the multiplicative identity element is denoted by 1 giving $a\,1 = a$, for all elements a in the ring. f(x) is called a polynomial of degree n, denoted by deg(f), and leading coefficient f_n if $f_n \neq 0$. If the leading coefficient f_n is 1, the polynomial is called monic.

Arithmetic on polynomials consists primarily of addition, subtraction, and multiplication; in some cases, further problems such as division and exponentiation are important. There are a number of other important computational problems pertaining to finite fields such as determining the greatest common divisor of polynomials, factoring, finding the roots of a polynomial, and computing a polynomial. These computational problems are the topic of this chapter. Given two polynomials $f_1(x)$ and $f_2(x)$, algorithms to find the greatest common divisor of $f_1(x)$ and $f_2(x)$ are discussed in Section 3.2. Let F be a field, and let F[x] denote the field of polynomials with coefficients in F. Given a polynomial $f(x) \in F[x]$, one wants to find the factorization $f = f_1 f_2 \cdots f_n$ of f into its irreducible factors $f_i(x) \in F[x]$. This factorization problem is discussed in Section 3.3. Algorithms to find a root $\alpha \in F$ of $f(x) = 0$ (if it exists) are also discussed in Section 3.3. Finally, given a polynomial $f(x) \in F[x]$, one may want to compute the value of the polynomial for a given value for the indeterminant x. This is the computing or evaluation porblem, and will be considered in Section 3.4.

2. GREATEST COMMON DIVISOR AND LEAST COMMON MULTIPLE

A greatest common divisor of two elements is defined as a common divisor that is divisible by as many primes as possible. Given two polynomials f(x) and g(x) over a field, with $g(x) \neq 0$, one can divide f(x) by g(x) to obtain a quotient polynomial q(x) and a remainder r(x) satisfying the conditions: $f(x) = q(x)\, g(x) + r(x)$, and $\deg(r(x)) < \deg(g(x))$. If $g(x) = 0$, then the greatest common divisor of f(x) and g(x), denoted by gcd(f(x), g(x)), is f(x). If $g(x) \neq 0$, then $\gcd(f(x), g(x)) = \gcd(g(x), r(x))$ where r(x) is as defined above. Thus to find gcd(f(x), g(x)), f(x) is divided by g(x) to get r(x); and as long as $r_i(x) \neq 0$,

the division procedure of $g(x)$ by $r(x)$ continues. When $r_l(x) = 0$, then $\gcd(f(x), g(x)) = \gcd(g(x), 0) = g(x)$.

This is called the Euclid's gcd algorithm for polynomials over a field. The Euclid's algorithm and the other sequential algorithms for finding the gcd have the difficulty which makes them decidedly impractical if the coefficients of the polynomials are integers or polynomials themselves, Brown [13].

Borodin-Gathen-Hopcroft [10] presented an algorithm that avoids this difficulty by employing linear equations. Let F be a field, let $f(x)$ and $g(x)$ be any two polynomials with coefficients in F with degrees m and n respectively, and $m \leq n$. Write $f(x) = f_m x^m + f_{m-1}x^{m-1} + \cdots + f_0$ and $g(x) = g_n x^n + g_{n-1}x^{n-1} + \cdots + g_0$ where $f_m\, g_n \neq 0$. If $h(x) = \gcd(f(x), g(x))$ with degree d, then there exist two polynomials $u(x)$ and $v(x)$ in $F[x]$ such that $\deg(u(x)) < n - d$ and $\deg(v(x)) < m - d$ such that $h(x) = u(x)\, f(x) + v(x)\, g(x)$. So the gcd problem is reduced to computing the polynomials $u(x) = \sum u_i\, x^i$ and $v(x) = \sum v_i\, x^i$. For $0 \leq k \leq n$ and polynomials $s(x) = \sum s_i\, x^i$ and $t(x) = \sum t_i\, x^i$, the conditions "$s\, f + t\, g$ is monic of degree k, and $\deg(s) < n - k$" translate into the systems S_k, $(MN = L)$, of linear equations in the coefficients of s and t, where the coefficient matrix M, as defined below, is the $(n+m-2k)\times(n+m-2k)$-submatrix P_i of the Sylvester matrix of (f, g) which consists of the first $m-i$ columns of f_j's and the first $n-i$ columns of g_i's, N is the column matrix $N = [s_{n-k-1} \cdots s_0\ t_{m-k-1} \cdots t_0]^T$ of order $(n+m-2k)\times(1)$, and L is the column matrix $[0 \cdots 0\ 1]^T$ also of order $(n+m-2k)\times(1)$.

Solutions to these systems of equations yield the computation of the

polynomials s(x) and t(x).

$$M' = \begin{bmatrix} f_m & & & g_n & & \\ f_{m-1} & \ddots & & g_{n-1} & \ddots & \\ \vdots & & f_m & \vdots & & g_n \\ f_0 & & \vdots & g_0 & & \vdots \\ & \ddots & \vdots & & \ddots & \vdots \\ & f_0 \cdots & f_k & & g_0 \cdots & g_k \end{bmatrix}$$

Thus if f(x) and g(x) are two polynomials as defined above, then a greatest common divisor of f(x) and g(x) can be computed by the following algorithm:

Algorithm 3.1. Univariate gcd of two polynomials.
Input: Two univariate polynomials f(x) and g(x).
Output: A Greatest Common Divisor of f(x) and g(x).

```
1. Compute $a_0, \cdots, a_m$, where $a_k = det(P_k)$ and $P_k$ is the
   coefficient matrix of $S_k$.
2. Set $d = min \{k : a_k \neq 0\}$.
3. Solve the system $S_d$ to find $s(x)$ and $t(x)$.
4. Return $gcd(f, g) = sf + tg$.
```

The determinants in Step 1 can be computed in parallel using any of the methods explained in Section 2.2.3. The time used in this step is $O(\log^2 n)$. To solve the system S_d, which is nonsingular since $a_d \neq 0$, the methods discussed in Section 2.3 with time $O(\log^2 n)$ can be used. Thus the total time used to find the gcd using this parallel algorithm is $O(\log^2 n)$.

The above algorithm is limited to two polynomials. Gathen [28] generalized this algorithm to find the gcd of a finite number of polynomials. Let F be a field. Let $f_1(x)$, $f_2(x)$, $\cdots$, $f_n(x)$ be polynomials with coefficients in F. Assume the degree of $f_i(x)$ is at most n. Let g(x) be the greatest common divisor of $f_1(x)$, $\cdots$, $f_n(x)$. By Euclidean Theorem, it is easy to see that there exist polynomials $s_1(x)$, $\cdots$, $s_n(x)$ with coefficients in F such that $g(x) = \sum s_i(x)\, f_i(x)$. In addition, $\deg(s_i(x)) < n$. The proof of the latter claim is direct: reorder the polynomials such that $\deg(f_1(x)) \geq \deg(f_i(x))$ for all $i \geq 2$. Divide s_i by f_1 with remainder $s_i = q_i f_1 + \bar{s}_i$ and $\deg(\bar{s}_i) < \deg(f_1) \leq n$. Set $\bar{s}_1 = s_1 + \sum q_i f_i$, where the sum is taken over $i \geq 2$. Then $\sum \bar{s}_i\, f_i = g$, where the sum is taken over $1 \leq i \leq n$, and $\bar{s}_1\, f_1 = g - \sum \bar{s}_i\, f_i$, where the sum, over all $i \geq 2$, has a degree less than $n + \deg(f_i(x))$. Hence, $\deg(\bar{s}_i) < n$ for all i. Because of this, the degree of g(x), d, can be defined as follows:

$$d = \min \left\{ \deg(f) : \exists\ s_1, \cdots, s_n \in F[x],\ \deg(s_i) < n \text{ for all i and } f = \sum s_i f_i \neq 0 \right\}.$$

If we write $f_i = \sum f_{ij}\, x^j$, where the sum is taken over $0 \leq j \leq n$, then as in Algorithm 3.1: for $0 \leq k \leq n$ add polynomials $S_i = \sum s_{ij}\, x^j$, $0 \leq j \leq n$, the condition "$\sum s_i\, f_i$ is monic of degree k" now translates into the systems S_k of linear equations in the coefficients of s_{ij} such that:

$$\sum s_{ij}\, f_{i,l-j} = \begin{cases} 0 & \text{for } k < l < 2n \\ 1 & \text{for } l = k \end{cases}$$

Thus to compute g(x), the indeterminants s_{ij} must be computed. The system S_k has $2n-k$ equations with at most n^2 variables. Clearly, S_k has a solution if and only if $k \geq d$. In fact S_d has a solution and from a solution of S_d, g(x) can easily be computed.

If $f_1(x)$, $\cdots$, $f_n(x)$ are polynomials as above, then a greatest common divisor, $g(x)$, of $f_1(x)$, $\cdots$, $f_n(x)$ can be computed by the following algorithm:

Algorithm 3.2. Univariate gcd of many polynomials.
Input: The univariate polynomials $f_1(x)$, $\cdots$, $f_n(x)$.
Output: A Greatest Common Divisor of $f_1(x)$, $\cdots$, $f_n(x)$.

```
1. For all $k$, $0 \leq k \leq n$, determine whether $S_k$ has a
   solution, and if it has, compute a solution $(s_{ij}(k))$ of $S_k$.
2. Set $d = \min \left\{k : S_k \text{ has a solution}\right\}$.
3. Compute $gcd(f_1, \cdots, f_n) = g = \sum s_{ij}(d)\, x^j f_i$.
```

Step 1 of this algorithm can be computed using Algorithm 3.1 of Borodin-Gathen-Hopcroft, which takes $O(\log^2 n)$ parallel steps. Steps 2 and 3 need $O(\log n)$ steps each. So the algorithm works in parallel time of $O(\log^2 n)$.

It should be remarked here that if Algorithm 3.1 is used to compute the gcd of pairs of polynomials along a binary tree, the $\gcd(f_1, \cdots, f_n)$ can be computed in parallel time $O(\log^3 n)$.

Given two polynomials $f_1(x)$ and $f_2(x)$, $\exists$ a polynomial $m(x)$ such that $f_1(x)$ and $f_2(x)$ are factors of $m(x)$, and $m(x)$ has the smallest degree with this property. The polynomial $m(x)$ is called the least common multiple of $f_1(x)$ and $f_2(x)$, and is denoted by $\mathrm{lcm}(f_1(x), f_2(x))$. The following relation holds for $f_1(x)$ and $f_2(x)$ in $F[x]$: $\gcd(f_1(x), f_2(x)) \times \mathrm{lcm}(f_1(x), f_2(x)) = f_1(x)f_2(x)$. The $\mathrm{lcm}(f_1(x), f_2(x))$ can be computed in parallel time of $O(\log^2 n)$ if F is real, and $O(\log^3 n)$ if F is an arbitrary field. Once the lcm is known, the gcd may be determined from it.

Let $u_i = \sum u_{ij} x^j$, $0 \leq j \leq k-d$ be monic polynomials of degree $k-d$, let S_k be the system of linear equations that expresses $u_1f_1 - u_2f_2 = u_2f_2 - u_3f_3 = \cdots = u_{n-1}f_{n-1} - u_nf_n = 0$. The system S_k consists of $(n-1)k$ linear equations in the $\sum (k-d_i) = nk-s$ indeterminant coefficients u_{ij} $(1 \leq i \leq n, 0 \leq j \leq k-d_i)$. The following algorithm, due to Gathen [28], computes the $\mathrm{lcm}(f_1, f_2)$:

Algorithm 3.3. Univariate lcm of many polynomials.
Input: The univariate polynomials $f_1(x), \cdots, f_n(x)$.
Output: The Least Common Multiple of $f_1(x), \cdots, f_n(x)$.

```
1. Set $d_i = deg(f_i)$, $m = max\ d_i$, and $s = \sum d_i$, $1 \leq i \leq n$.
2. Replace each $f_i$ by $f_i/a_i$ where $a_i$ is the leading
   coefficient of $f_i$.
3. $\forall$ $k$, $m \leq k \leq s$, determine whether $S_k$ has a
   solution, and if it does, compute solution $u_{ij}(k)$.
4. Set $d = min \{k : S_k$ has a solution$\}$.
5. Set $u = \sum u_{1j}(d)\, x^j + x^{d-d_1}$.
6. Set $lcm\ (f_1(x), \cdots, f_n(x) = m(x) = u\, f_1$.
```

The methods discussed above apply to univariate polynomials, i.e., polynomials in one variable or indeterminant. Gathen-Kaltofen [31] presented an algorithm for the greatest common divisor of two bivariate polynomials. Given two polynomials $f, g \in F[x, y]$, where f is monic with respect to x and F is an arbitrary field, they used a modular approach to compute the, monic with respect to x, gcd $h \in F[x, y]$ of f and g. The algorithm can be stated as follows:

Algorithm 3.4. Bivariate gcd of two polynomials.
Input: Two bivariate polynomials f(x) and g(x).
Output: A Greatest Common Divisor of f(x) and g(x).

```
1. Set $d_x = max\{deg_x\ f,\ deg_x\ g\}$, $d_y = max\{deg_y\ f,\ deg_y\ g\}$, and $d = 2d_x d_y$. If $d = 0$, use a procedure for univariate gcd's. If $|F| = q < 3d$, then choose an irreducible monic polynomial $w \in F[t]$ of degree $\lceil log_q 3d \rceil$, and replace F by the extension field $F[t]/(w)$.
2. Choose any pairwise distinct $a_1, a_2, \cdots, a_{2d} \in F$ such that $g(x, a_i)$ has the same degree in x as g.
3. $\forall i$, $1 \le i \le 2d$ compute the monic $h_i = gcd(f(x, a_i), g(x, a_i)) = \sum h_{ij} x^j \in F[x], j \ge 0$.
4. Set $m = min\{deg\ h_i : 1 \le i \le 2d\}$, and choose some $M \subseteq \{1, \cdots, 2d\}$ with $|M| = d_y + 1$ and $deg\ h_i = m\ \forall i \in M$.
5. For $0 \le j \le m$, interpolate the $h_{ij}$'s: Compute $b_j \in F[y]$ of degree at most $d_y$ with $b_j(a_j) = h_{ij}\ \forall i \in M$.
6. Return $gcd(f, g) = h = \sum b_j x^j$, $0 \le j \le m$.
```

To estimate the timing of the algorithm, w in Step 1 can be found in $O(\log^4 d)$ operations in F, (Rabin [56]), since each monic polynomial $w \in F[t]$ of degree $l = \lceil \log_q 3d \rceil$ may be tested for irreducibility. There are at most $q^t \le 3dq < qd^2$ such polynomials, and each irreducible test takes $O(\log^2 d \log^2 \log d \log\log\log d \log q)$. Any operation in F[t]/(w) can be simulated by $O(\log^2 d)$ operations in F. This factor $\log^2 d$ has to be multiplied to the estimates for Steps 3 to 6 only if $q < 3d$. For each $f(x, a_i)$ and $g(x, a_i)$ in Step 3, the number of operations is $O(d)$. For each h_i, it is $O(d_x \log^2 d_x)$ (Aho-Hopcroft-Ullman [1]). Therefore, the total time necessary for Step 3 is $O(d\,(d + d_x \log^2 d_x))$

operations. Step 5 takes $\mathcal{O}(d_x(d_y \log^2 d_y))$ operations ($m < d_y$). The total time is $\mathcal{O}(d^2\log^2 d \log^2 d) = \mathcal{O}(d^2\log^4 d)$. If $q \geq 3d$, it is $\mathcal{O}(d^2\log^2 d)$ operations.

3. FACTORING POLYNOMIALS

Polynomials with coefficients from a finite field and their factoring techniques have been studied for a long time. In 1846, the Unique Factorization Property was proved for univariate polynomials over $\mathbb{Z}_p$. But no efficient algorithm to compute these factors was presented until the 1960's. Suppose F is a finite field of characteristic p (i.e., a prime p is the smallest element such that $pa = 0 \ \forall \ a \in F$) with $q = p^d$ elements, i.e., $F = GF(p^d)$. A fundamental computational task is to find the irreducible factors of a polynomial $f(x) = \sum_{i=0}^{n} f_i x^i$ in F[x]. This is called a univariate polynomial and will be the subject of Subsection 3.1. Polynomials over more indeterminants are called multivariate polynomials and will be discussed in Subsection 3.2. As it will be shown there, the problem of factoring multivariate polynomials over algebraic number field or over finite fields is eventually reduced to that of factoring univariate polynomials over finite fields, via a modular technique.

3.1. Factoring univariate Polynomials

Note first that the general factoring problem easily reduces to that of factoring a monic polynomial with no repeated factors (such a polynomial is called square-free). This is because one can divide each polynomial coefficient by

the leading coefficient to make the polynomial monic, and then use the following well-known method for finding repeated factors. Consider the case of a polynomial f(x) with repeated factor $f_2(x)$, i.e., $f(x) = f_1(x)(f_2(x))^n$. Differentiating f(x) one gets

$$f'(x) = f'_1(x)(f_2(x))^n + n\, f_1(x)\, (f_2(x))^{n-1}\, f'_2(x)$$
$$= (f_2(x))^{n-1} \left(f'_1(x)\, f_2(x) + n\, f_1(x)\, f'_2(x)\right) \times$$
$$(f_2(x))^{n-1} = \gcd(f(x), f'(x)),$$

and one may easily remove this gcd(f(x), f'(x)) from f(x) to convert f(x) to a monic square-free polynomial.

Berlekamp [7] devised the first complete factoring algorithm which factors univariate polynomials over a finite field F with q elements in $O(qn^3)$ operations where n is the degree of the polynomial. Let u(x) be the polynomial to be factored. The algorithm proceeds as follows:

Algorithm 3.5. Univariate factorization over a finite field I.
Input: A univariate polynomial $u(x) \in F[x]$ of degree n.
Output: The complete factorization of u(x).

1. *Ensure that $u(x)$ is square-free (i.e., if $gcd(u(x), u'(x)) \neq 1$, reduce the problem to factoring $u(x)/gcd(u(x), u'(x))$).*
2. *Form the matrix Q defined by*

$$Q = \begin{bmatrix} q_{0,\,0} & q_{0,\,1} & \cdots & q_{0,\,n-1} \\ \vdots & \vdots & \vdots & \vdots \\ q_{n-1,\,0} & q_{n-1,\,1} & \cdots & q_{n-1,\,n-1} \end{bmatrix}$$

where $x^{pk} \equiv q_{k,\,n-1}x^{n-1} + \cdots + q_{k,\,1}x + q_{k,\,0}$ (modulo $u(x)$). Thus, each row of Q consists of coefficients of powers of $x^p \bmod u(x)$.

3. *Triangularize the matrix $Q-I$, where $I = (\delta_{ij})$ is the $n \times n$ identity matrix, finding its rank $n-r$ and finding linearly independent row vectors $v^{[1]}, \cdots, v^{[r]}$ such that $v^{[j]}(Q-I) = (0, \cdots, 0)$ for $1 \leq j \leq r$. This triangularization can be done using appropraite column operations (Null space algorithm for instance).*
4. *Calculate $gcd(u(x), v^{[2]}(x)-s)$ for $0 \leq s \leq p$, where $v^{[2]}(x)$ is the polynomial represented by vector $v^{[2]}$. The result is a nontrivial factorization of $u(x)$, because $v^{[2]}(x) - s$ is nonzero and has degree less than $deg(u)$, and $u(x) = \prod gcd(v(x)-s,\ u(x))$, where it is understood that the product is taken over $0 \leq s \leq p$, whenever $v(x)$ satisfies $v(x)^p \equiv v(x)$ (modulo $u(x)$), and $deg(v) < deg(u)$.*

Moenck [52] gave the following analysis of the time-complexity of the Berlekamp's algorithm. Multiplying or dividing a polynomial of degree n by one of degree m can be done in $O(mn)$ field operations using the standard methods. As a corollary, one can see that squaring a polynomial of degree $n-1$ and computing the residue with respect to another polynomial of degree n can be done in $O(n^2)$ field operations. Since u(x) is monic, $x^p \bmod u(x)$ can be computed by repeatedly squaring in $O(n^2 \log p)$ steps. In the remaining $n-2$ rows of the matrix Q, $x^{pj} \bmod u(x)$ can be produced in $O(n^3)$ steps. Computing the null space of the matrix $Q - I$ can be done in $O(n^3 + n \log p)$ steps using a standard triangularization algorithm. The gcd operation can be performed in

$O(n^2 + n \log p)$ steps. (A parallel algorithm can be used to compute the gcd in parallel time $O(\log^2 n)$, see Section 3.2). If there are k factors, in the worst case, each v(x) will yield only one prime factor. To find this factor, one might have to try every element in the field. This means that the algorithm is bounded by the last step which requires $O(kp(n^2 + n \log p))$ field operations. If $k = O(n)$, the algorithm may require $O(n^3 p)$ steps. It is the factor p in this expression which restricts the application of the algorithm to small primes.

Berlekamp's algorithm for factoring polynomials over a finite field $\mathbf{Z}_p$ is a major milestone in the study of the factoring problem. One of the handicaps of his algorithm was the p term in the timing analysis. This restricts the method to relatively small fields. Later, Barlekamp [8] refined his method so that the factoring problem is reduced to computing the roots of a polynomial in a finite field. He showed how the latter problem could be solved in time proportional to $p^{1/4} \log p^{3/2}$. Moenck [52] gave a more direct reduction to the root finding problem and gave a method for finding the roots of a polynomial of degree n in $O(n^2 \log p + k \log^2 p)$ steps for special choices of p. These imply that Berlekamp's algorithm can be performed in $O(n^3 + n^2 \log p + n \log^2 p)$ steps for most cases. He also showed that a polynomial can be factored in $O(n^2(\log^2 n + \log n \log p))$ steps.

Algorithm 3.5 and the further improvements by Moenck [52] and Berlekamp [8] apply only when $q=p^d$ where $d=1$, and uses the calculations of resultants (or equivalently the solutions of linear equations) to reduce the problem to finding the roots of a polynomial which has all of its roots in F.

However, straightforward modifications given by Cantor-Zassenhaus [16] allow d > 1. They presented a probabilistic method which, when combined with the above algorithm and similar algorithms (e.g., folk method, Knuth [42]), avoids the need for both resultants and linear equations. It leads to algorithms which are conceptually simpler than the previous method. Moreover, it works equally well for all finite fields F, regardless of the magnitude of q. When used for factoring a quadratic, $x^2 - a$, it reduces to the Berlekamp's Algorithm 3.5.

Let p be a prime number, and let n be an integer. Let E be the Galois field $E = GF(p^n)$. Given a polynomial $f(x) \in E[x]$ of degree m, Rabin [56] presented a probabilistic algorithm to find the factorization $f = f_1 f_2 \cdots f_k$ of f into its irreducible factors $f_i(x) \in E[x]$. Both Rabin's and Cantor-Zassenhaus' algorithms are probabilistic ones and, therefore, will not be discussed in details in this thesis. However, Gathen [28] presented the Cantor-Zassenhaus probabilistic algorithm with the appropriate modification for parallel execution. We now discuss this modified algorithm.

Let F be a finite field with q elements, and let $f(x) \in F[x]$ be a monic polynomial of degree $n \geq 2$. In order to get a better complexity estimate in case q is not prime, let $G \subseteq F$ be another field with a prime number p of elements. Let $g \in G[x]$ be irreducible of degree d such that $F = G[t]/(g)$ and $q = p^d$. F is a vector space over G with basis $1, t, \cdots, t^{d-1}$, and $R = F[x]/(f)$ is a vector space over F with basis $1, x, \cdots, x^{n-1}$, and a dn-dimensional vector space over G with basis $\{t^i x^j: 0 \leq i < d, 0 \leq j < n\}$. The algorithm can then be stated as follows:

Algorithm 3.6. Univariate factorization over a finite field II.

Input: A polynomial $f(x) \in F[x]$ of degree n.

Output: The complete factorization of f.

1. *Replace f by its (unique) monic scalar multiple. Compute the matrix Q of the Frobenius mapping: $R \rightarrow R$ with $u \rightarrow u^p$.*
2. *Compute the dimension r of the nullspace K of Q − I, where, I is the* n×n *identity matrix, and $g_1, \cdots, g_r \in F[x]$ of degree less than n such that $\{g_i \bmod f \mid 1 \leq i \leq n\}$ forms a basis of K. If r = 1, set S = {f} and go to Step 5. r denotes the number of monic factors of f.*
3. *Let $m = \lceil \log r \rceil$, choose $v_{ij} \in F[x]$ for $1 \leq i \leq m$, $1 \leq j \leq r$ independently at random, and let $h_i = v_{ij}\, g_j \in F[x]$ for $1 \leq i \leq m$, $1 \leq j \leq r$.*
4. *For $1 \leq i \leq m$, compute $c_i = gcd\,(f,\ h_i^{(p-1)/2} - 1) \in F[x]$. If p is even, say $p = 2^k$, use $c_i = gcd(f, \sum h_i^{2j}, 0 \leq j < k)$. Compute the common refinement of these partial factorizations as follows. Let $M = \{0, 1\} \times \{1, \cdots, m\}$. For $I \subseteq M$ compute $s_I = gcd\left(\{c_i : (0, i) \in I\} \cup \left\{\frac{f}{c_i} : (1, i) \in I\right\}\right)$. Then compute the follwoing set T of "minimal I's":*

   $$T = \left\{ I \subseteq M : s_I \neq 1 \text{ and } \forall J \subseteq M \quad I \subseteq J \rightarrow s_J = 1 \text{ or } s_J = s_I \right\}.$$
5. *If $|S| \neq r$, then the algorithm fails. Otherwise, for each $a \in S$ do the following. Set b = a. While $b' = db/dx = 0$, replace $b = \sum b_k x^k$ for $k \geq 0$ by its p_0-th root $\sum b_{kp_0}{}^{q/p_0} x^k$, where $p_0 = char\,F$ is a prime number. If $b' \neq 0$, compute $g = b \,/\, gcd(b, b')$. Now g is an irreducible factor of f, and $e = deg\, a / deg\, g$ its multiplicity.*
6. *Return the set of all (g, e) computed above as the complete factorization of f.*

To estimate the parallel time of the algorithm, notice that if Q is an $n \times n$ matrix, then Step 1 requires $O(\log^2 n \log p)$ steps. Step 2 takes $O(\log^2 n)$ operations. This is because the computation of the quotient and the remainder of two polynomials of degree at most n needs parallel time $O(\log^2 n)$. The proof of the last assertion is rather simple. Let f, g $\in F[x]$. Let $k = \deg f - \deg g + 1 \leq n$. Their quotient $q \in F[x]$ is uniquely determined by the condition $\deg(f - qg) < \deg g$, which can be expressed by a nonsingular system of k linear equations in the k coefficients of q. This system can be solved in parallel time $O(\log^2 n)$ as in Section 2.3. Computation of the remainder takes $O(\log n)$ parallel steps. The fast parallel algorithm from Borodin-Gathen-Hopcroft [10] can be used to solve Step 2. This algorithm takes $O(\log^2 n)$ parallel steps also. Step 3 costs $O(\log r)$. In Step 4, each c_i can be computed with $O(\log^2 n \log p)$ operations. For each s_l, Gathen's algorithm to find the gcd of many polynomials from Section 3.2, Algorithm 3.2, can be applied eight times in parallel, with the same number n and using parallel time $O(\log^2 n)$. Unless all these applications fail, any of the answers can be taken. To compute the sets T and S, note that the number of subsets I of set M comes into play. This number of subsets is $2^{|M|} \leq 2^{2m} \leq 2^{12 \log r} = r^{12} \leq n^{12}$. Thus T and S can be computed in parallel time $O(\log n)$. Finally, each g in Step 5 can be computed in time $O(\log_{p_0} n \log q + \log^2 n \log p)$. Thus the total time is $O(\log^2 n \log q)$.

It should be noted that if the polynomial f(x) is square-free, and if $G \subseteq F$ is a subfield with p elements and $q = p^d$, then the complexity is $O(\log^2 n \log^2(d+1) \log p)$ operations in G. To show this, simply note that each operation in F can be simulated by operations in G in parallel time $O(\log^2(d+1))$. (d+1 rather than d is used to avoid getting log 1 = 0). Since all elements of R are represented by coefficients from G, one can consider Q as a dn×dn matrix over G. The computation of g in Step 5 is unnecessary since s is assumed to be squarefree. Thus, the time is $O(\log^2 n \log^2(d+1) \log p)$. In both cases, the number of processors is polynomial is n log q.

Lenstra-Lenstra-Lovasz [47] have presented a polynomial-time algorithm to solve the factoring problem over the field of rational numbers. If $f(x) \in \mathbb{Q}[x]$ is a univariate polynomial with rational coefficients, Lenstra-Lenstra-Lovasz [47] found the decomposition of f into irreducible factors in $\mathbb{Q}[x]$. Since this is equivalent to factoring primitive polynomials over $\mathbb{Z}[x]$, it can be considered a breakthrough in the factorization problem for univariate integer polynomials. (Here, by a primitive polynomial, we mean a polynomial $f(x) \in \mathbb{Z}[x]$ with content 1, i.e., the greatest common divisor of its coefficients is 1). An outline of the algorithm is as follows.

Algorithm 3.7. Univariate factorization over $\mathbb{Q}$.
Input: A polynomial $f(x) \in \mathbb{Q}[x]$ of degree n.
Output: The complete factorization of f.

1. *For a suitable small prime number p, find a p-adic irreducible factor h of f. Algorithm* 3.5 *can be used to find h.*
2. *Find an irreducible factor h_0 of f in $\mathbb{Z}[x]$ that is divisible by h.*
3. *Repeat the above two steps until all irreducible factors of f are found.*

The condition that h_0 is divisible by h means that h_0 belongs to a certain lattice, and the condition that h_0 divides f implies that the coefficients of h_0 are relatively small. Thus basically, the smallest element in that lattice needs to be found. The authors give a new reduction algorithm to do this. Therefore, this is a basic subroutine to compute short vectors in integer lattices. The running time of this algorithm, measured in bit operations, is $\mathcal{O}(n^{12} + n^9(\log |f|)^3)$, where n = deg(f).

It has been seen that a univariate polynomial of degree n over a finite field with q elements can be factored deterministically in $(nq)^{\mathcal{O}(1)}$ bit operations, and probabilistically in $(n \log q)^{\mathcal{O}(1)}$ bit operations (Berlekamp [7], Cantor-Zassenhaus [16], and Rabin [56]). For practical purposes, the probabilistic algorithms are quite satisfactory. However, the existance of a deterministic method to solve this problem in polynomial time, i.e., $(n \log q)^{\mathcal{O}(1)}$ bit operations is still an open question. Gathen [30] dealt with this question and proved that, for primes of a very special form (for those prime numbers p, for which all prime factors of $p-1$ are small), the factoring problem is deterministic polynomial-time equivalent to the more classical problem of finding primitive elements.

Given a prime number p, denote by w an irreducible monic polynomial $w \in \mathbb{Z}_p[y]$ of degree d, and by f(x) a monic polynomial $f(x) \in F[x]$ of degree n, where $F = \mathbb{Z}_p[y]/(w) = GF(p^d)$. The expected output of the factoring process is a factorization $f_1, f_2, \cdots, f_s \in F[x]$, where f_i's are disitinct irreducible monic polynomials, and $d_1, d_2, \cdots, d_s \geq 1$ such that $f = f_1^{d_1} \cdots f_s^{d_s}$.

Consider R = F[x]/(f) as an ld-dimensional vector space over $\mathbb{Z}_p$, with basis $\{x^i y^j \bmod (w, f) : 0 \le i < d,\ 0 \le j < l\} \subseteq R$, and residue class mapping g $\rightarrow \bar{g}$ from F[x] to R.

The main result of Gathen [30] can be stated as follows: "On input f ∈ GF(p^l)[x] of degree d and a primitive element modulo p, **the algorithm described** below can be executed with

$$O([(dl)^{l+\epsilon}\ ((dl)^{2.4} + \log^2 p) + S(p-1)\ (\log^2 p + l^{l+\epsilon} \log p + (dl)^{l+\epsilon})].\ \log^{l+\epsilon} p)$$

bit operations for any $\epsilon > 0$, or $O(n^8)$ bit operations, where S(p−1) is the greatest prime factor of p−1, and n = max {d, l, log p, S(p−1)}. If f is reducible, the algorithm returns a nontrivial factor of f."

The algorithm presents a deterministic polynomial-time reduction of factoring to the problem of finding primitive elements of special type of prime numbers as discussed above. It first computes polynomials $g_1 = 1, g_2, \cdots, g_s \in$ F[x] of degrees less than d such that the vectors formed by their coefficients, $\bar{g}_1, \cdots, \bar{g}_s$, form a basis of the $\mathbb{Z}_p$-vector space $B = \{u \in R : u^p = u\} \subseteq R$, the Berlekamp subalgebra of R. Once this basis is obtained, one proceeds as follows:

Algorithm 3.8. Univariate factorization over a finite field III.
Input: A polynomial f(x) ∈ GF(p^l)[x] *of degree d.*
Output: A nontrivial factor of f.

1. *If* $s = 1$, *return "f is irreducible" and stop, else set* $g = g_2$. *If gcd(f, g)* $\neq 1$, *return this nontrivial factor of f and stop.*
2. *For* $1 \le j \le r$, *compute* $w_j \in F[x]$ *of degree less than d as* $w_j \equiv g^{q_j} \bmod f$, *with* $q_j = (p-1)/p_j^{e_j}$. *Let i be the first*

value of j such that $w_j \notin \mathbb{Z}_p$, and set $h_1 = w_j$.

3. *For $t = 0, \cdots, e_i-1$, compute $y_t \in F[x]$ of degrees less than d such that $y_t \equiv h_1^{p_i^{e_t-1}}$ mod f. Set $m = max\{t : y_t \in \mathbb{Z}_p\}$.*
4. *Compute c in the multiplicative group $\mathbb{Z}_p$ such that $c^{p_i} = y_m$.*
5. *Compute $h_2 \in F[x]$ of degree less than d such that $h_2 \equiv h_1^{p_i^{e_i-m-1}} c^{-1} \bmod f$.*
6. *For $0 \leq v < p_i$, compute $z_v = a^{v(p-1)/p_i} h_2 - 1 \in F[x]$. Return $gcd(f, z_v)$ if it is nontrivial, and stop.*

The dominating computing times in this algorithm are as follows: Computing $g_1, \cdots, g_s$ requires $O((dl)^{2.4} + dl \log p)$ operations in R, using fast matrix arithmetic (see Coopersmith-Winograd [22]). Each of Steps 2, 3, and 5 requires $O(\log^2 p)$ operations in R. Step 4 uses $O(\log^2 p\ S(p-1))$ operations in $\mathbb{Z}_p$, and Step 6 requires $O(S(p-1)(\log p + d^{1+\epsilon}))$ operations in F for any $\epsilon > 0$. With fast integer and polynomial arithmetic, the time estimate mentioned earlier may be derived.

Thus, the factoring problem can be reduced to the problem of finding primitive elements. Conversely, the reduction of primitive elements to the factoring problem. (Interested reader is refered to Gathen [30]). Hence, the claim of Gathen [30] about the equivalence of the two problems is correct.

The problem of factoring polynomials over finite fields of characteristic p is important. The case of characteristic 2 is particularly important, e.g., in algebraic coding theory. Camion [15] has proved the existence of a polynomial-

time factoring procedure in GF(2^m). He showed that polynomials of degree d over a finite field GF(2^m) can be factored deterministically with $\mathcal{O}((dm)^w)$ operations in $\mathbb{Z}_p$, with w < 2.4. He has, however, not given the algorithm to do this factoring.

3.2. Factoring Multivariate Polynomials

In this subsection, algorithms for the factorization of multivariate polynomials with coefficients from a finite field will be discussed. Let f be a polynomial in $F[x_1, x_2, \cdots, x_t]$ of degree n_i in x_i, where $F = GF(p^m)$.

As it was shown in the previous subsection, Berlekamp's algorithm [7] factors univariate polynomials over a finite field with q elements in $\mathcal{O}(qn^3)$ field operations, where n is the degree of the polynomial. This execution time is polynomial in both n and q. Soon after this, Berlekamp modified the running time to be polynomial in the input size, i.e., using log q rather than q, at the expense of introducing a probabilistic rather than a deterministic method. It seems natural to ask whether this can be accomplished for multivariate polynomials over F. Given a bivariate polynomial of total degree n with coefficients in F, can one find (probabilistically) its factors in sequential running time polynomial in n and log q?.

Gathen-Kaltofen [31] have given a polynomial-time factorization algorithm for bivariate polynomials over a finite field. This algorithm, based on methods from Kaltofen [39] and [40], has three variants: a probabilistic one with running time $(n \log q)^{O(1)}$, a deterministic one with running time $(nq)^{O(1)}$, and a parallel one with running time $O(\log^2 n \log q)$ where n is the degree of the input polynomials and q is the cardinality of the coefficient field. In the deterministic case, q can be replaced by log q if one could factor univariate polynomials over finite fields in deterministic time polynomial in log q. The parallel variant is a generalization of the results of univariate factorization in Gathen [28].

Let F be a field with q elements and characteristic p, and f ∈ F[x, y] be a bivariate polynomial. f is called "nice" if f(x, 0) ∈ F[x] is square-free, and f is monic with respect to x. The following algorithm computes an irreducible factor g ∈ F[x, y] of a nice polynomial f:

Algorithm 3.9. Quick factoring.
Input: A nice polynomial f ∈ F[x, y].
Output: An irreducible factor g ∈ F[x, y] of f.

1. *Compute an irreducible monic factor* $h \in F[x]$ *of* $f(x, 0)$. *If* $h = f(x, 0)$, *then return* f.
2. *Set* $d_x = deg_x\, f$, $d_y = deg_y\, f$, *and* $d = 2d_x d_y$. *Set* $E = F[t]/(h(t))$, *and* $a_0 = (t \bmod h(t)) \in E$. *Use the Newton iterations to compute* $b \in E[y]$ *such that* $f(b, y) \equiv 0 \bmod y^{d+1}$ *in* $E[y]$.
3. *Set* $s = 1/f_x(a_0, \ 0) \in E$, *where* f_x *is the partial derivative of* f *with respect to* x. *This derivative is not zero because otherwise* a_0 *would be a double zero for* $f(x, 0)$, *contradicting its squarefreeness.*
4. *For* $k = 1, \cdots, d$ *compute* $a_k = a_{k-1} - s\, f(a_{k-1}, \ y) \in E[y]$.
5. *Find the minimal* i, $deg\ h \leq i \leq d_x$, *for which there exist*

$u_0, \cdots, u_{i-1} \in F[y]$ such that $\deg_y u_j \le d_y$ for $0 \le j < i$,
and $b^i + \sum u_j b^j \equiv 0 \bmod y^{d+1}$ for $0 \le j < i$.
Compute the corresponding $u_0, \cdots, u_{i-1}$.

6. *Return $g = x^i + \sum u_j x^j \in F[x, y]$ for $0 \le j < i$.*

The factoring of the univariate polynomial in Step 1 can be done as in Gathen [28] using $\theta(e) = O(\log^2 e \log q)$ steps, where e is the degree of h. Step 3 may be performed in $O(d_X)$ operations in E. In Step 4, each a_k takes $O(d_X)$ operations in E[y] (to compute mod y^{k+1}). Thus the total time for Step 4 is $O(d_X\ d \log^4 d)$ operations. In Step 5, compute $b^2, \cdots, b^{d_y}$ in $O(d_X)$ operations in E[y] or $O(d_X\ d \log^4 d)$ operations in E. A system of at most $(d+1)\, d_X$ linear equations in $d_X\,(d_Y+1)$ unknowns over F is to be solved. If the Gaussian elimination method is used, it takes $O((d_X(d_Y+1))^2(d+1)d_X)$ or $O(d^3 d_X)$ operations in F. Thus the total time for Step 5 is $O(d^3 d_X + d^2 d_X{}^2 \log^4 d \log^4 d_X)$ or $O(n^3 d_X{}^4)$ operations in F. Thus the algorithm can be used to factor a polynomial f of total degree n in $O(n^3 d_X{}^4) + \theta(d_X)$ or $O(n^7) + \theta(n)$ operations in F.

The above algorithm only dealt with square-free monic polynomials. But it can be easily generalized as follows to factor any polynomial $f \in F[x, y]$:

Algorithm 3.10. Bivariate factoring over finite fields I.
Input: A polynomial $f \in F[x, y]$.
Output: A nonconstant factor $g \in F[x, y]$ of f.

1. *Check primitivity: Set $d_x = \deg_x f$, and write $f = \sum f_t x^t$ for $0 \le t \le d_x$ with $f_t \in F[x]$. Compute the content, c*

$= cont_x(f) = gcd(f_0, \cdots, f_{d_x}) \in F[y]$. *If c is constant, then return c.*

2. *Check squarefreeness: Compute partial derivatives* f_x *and* f_y. *If* $f_x = f_y = 0$, *then write* $f = \sum f_{ij} x^{ip} y^{jp}$, $i, j \geq 0$. *Set* $g = \sum f_{ij}^{q/p} x^i y^j$ *and return g. If* $f_x = 0$ *and* $f_y \neq 0$, *then interchange the role of x and y and go to Step 1. Compute the monic* $g = gcd(f, f_x)$.
3. *Monic version of f: Let* $f_0 \in F[y]$ *be the leading coefficient of f with respect to x. Set* $v = f_0^{d_x} f(x/f_0, y) \in F[x, y]$. *Then v is monic of degree* d_x *with respect to x.*
4. *Extend F: Set* $d_y = deg_y v$, $m = max\{d_x, d_y\}$, *and* $d = 2d_x d_y$. *If* $q = |F| > d$, *set* $F^* = F$. *Otherwise, choose a prime number l with* $m < l \leq 2m$. *Choose monic polynomials* $w_1, \cdots, w_{ql_n} \in F[t]$ *of degree l at random, and test them for irreducibility. If none is irreducible, return "failure". Otherwise choose an irreducible* w_i, *and set* $F^* = F[t]/(w_i)$.
5. *Good evaluation point: Set* $r = res_x(v, v_x) \in F[y]$. *Choose* $c \in F^*$ *such that* $r(c) \neq 0$, *and set* $f^* = v(x, y-c) \in F^*[x, y]$. f^* *is nice.*
6. *Apply Algoritthm* 3.9 *to factor* $f^* \in F^*[x, y]$ *and get* $g^* \in F^*[x, y]$.
7. *Set* $e = deg_x g^*$, $g_1 = f_0^{e+1} g^*(xf_0, y+c) \in F^*[x, y]$, $g_0 = cont_x(g_1) \in F^*[y]$, $g = g_1/g_0 \in F[x, y]$, *and return g.*

To estimate the time complexity, first note that $d_X \leq n$, $d_Y \leq n^2$, $d = 2d_X d_Y \leq 2n^3$, $l \leq 2m \leq 2n^2$, and the total degree n^* of f^* is not more than n^2. Step 1 then requires $O(n^3)$ operations and Step 3, $O(n^4)$. In Step 2, the gcd can be computed in $O(d^2 \log^4 d)$ operations in F sequentially (for parallel time, see

Section 3.3.2.), and the p-th root in $O(d \log q/p)$ operations in F. The prime number l can be found deterministically in $O(m^{3/2}\log^2 m)$ bit operations, and w in $O(n^7\log^3 n \log q)$ operations in F (Rabin [56]). Steps 5 and 7 both take $O(d_x d^2)$ operations. The cost of the algorithm is dominated by the complexity of Step 6, which is $O(n^{10} + n^5 \log n \log q)$ operations in F^*. Each operation in F^* can be simulated by $O(l \log^4 l)$ operations in F, or $O(l \log^4 l \log^2 q)$ bit operations. Thus the total cost is $O(n^7\log^4 n \log^2 q\ (n^5 + \log n \log q))$ bit operations.

Once one nontrivial factor is found, the algorithm can be repeatedly applied to yield a complete factorization of the input polynomial.

The above algorithm can be implemented using parallel computing. Note that the basic subroutines for the algorithm are univariate factoring procedure over finite fields, computing univariate gcd's, and solving systems of linear equations over a finite field. All these tasks have been shown to be solvable in parallel with $O(\log^2 n)$ operations in F (respectively $O(\log^2 n \log q \log p)$ for factoring), where n is the total degree of the input polynomial, p is characterisitic of F, and $q = p^k = |F|$. For a complete factorization, one would lift all irreducible factors of f(x, 0) from Step 1 of Algorithm 3.9 in parallel, using a quadratic Newton procedure, and then discard duplicate roots. Also, a prime number l as in Step 4 of Algorithm 3.10 can be found in parallel with $O(\log^2 n)$ bit operations. The resulting algorithm returns the complete factorization of the input polynomial in parallel time $O(\log^2 n \log^2(kn) \log p + \log n \log q)$. The first summand from Step 1 of Algorithm 3.9, where a univariate polynomial of

degree at most n over a field with not more than p^{kn^2} elements has to be factored. In Step 4, each step of the quadratic Newton iteration has to compute $s \in E[y]$ such that $s\, f_k(a_k, y) \equiv 1 \bmod y^{2k}$. This congruence can be considered as a system of linear equations over the base field, and solved in parallel time $O(\log^2 n)$. The second summand comes from the computation of the p-th roots in Step 2 of Algorithm 3.10.

Other algorithms for the problem of factoring multivariate polynomials over finite fields are due to Chistov-Grigoryev [19] and Lenstra [46]. Lenstra [46] has described multivariate polynomial factorization algorithm over finite fields that is polynomial-time in the degree of the polynomial to be factored. The algorithm makes use of a new basis reduction algorithm for lattices over the field F[Y] containing q elements.

If the number of variables equals two, then the algorithm is similar to Algorithm 3.7 by Lenstra-Lenstra-Lavesz [47]. An outline of the algorithm for the factorization of $f \in \mathbf{F}_q[x, y]$ is as follows:

Algorithm 3.11. Bivariate factoring over finite fields II.
Input: A polynomial $f \in F[x, y]$.
Output: The factorization factoring of f.

1. *Calculate the resultant* $R(f, f_x) \in \mathbf{F}_q[y]$.
2. *Determine a positive integer u, and an irreducible polynomial* $F \in \mathbf{F}_q[y]$ *of degree u such that* $R(f, f_x)$ *is nonzero mod F. The reader is referred to Lenstra* [46] *for a method to find such u and F.*
3. *Apply Berlekamp's algorithm, Algorithm* 3.5, *to compute*

the irreducible factorization (h mod F) of (f mod F) in $\mathbf{F}_{q^u}[x]$.

4. *Since (h mod F)^2 does not divide (f mod F) in $\mathbf{F}_{q^u}[x]$, due to the choice of F and u, the complete factorization of f can be obtained by repeating application of a proposition given in Lenstra* [46], (*Proposition* 2.15).

Let k be a positive integer, let $d_x f^k$ denote the degree of f^k with respect to x. Step 1 requires $O(d_x f^6 d_y f^2)$ computations. Step 2 requires less than or equal to $d_y f\,(2d_x f-1)$ arithmetic operations in $\mathbf{F}_q$. Step 3 takes $O(d_x f^{4+\epsilon} d_y f^{1+\epsilon})$ arithmetic operations in $\mathbf{F}_q$. Finally, Step 4 requires $O(d_x f^6 d_y f^2)$ operations in $\mathbf{F}_q$. Hence, the factorization of f can be determined in $O(d_x\, f^6\, d_y\, f^2 + d_x\, f^3\, p\, m + d_y\, f^3\, p\, m)$, where $q = p^m$.

For factoring multivariate polynomials with more than two variables, ($f \in \mathbf{F}_q[x_1, \cdots, x_t]$ with $t > 2$), high powers of x_2 for x_3 up to x_t are computed first. This reduces the problem to factoring the polynomial in $\mathbf{F}_q[x_1, x_2]$. Let $d_i f = n_i$ denote the degree of f in x_i. Let $\tilde{f}_j \in \mathbf{F}_q[x_1, x_2, x_{j+1}, x_{j+2}, \cdots, x_t]$ be the polynomial f modulo $((x_3 - x_2^{k_3}), \cdots, (x_j - x\, x_2^{k_j}))$, for $2 \le j \le t$; i.e., $\tilde{f}$ is f with $x_2^{k_i}$ substituted for x_i, for $3 \le i \le j$. By choosing integers $k_3, \cdots, k_t$ such that $k_j = \prod_{i=2}^{j-1} (2nn_i - 1)$ for $3 \le j \le t$, one can ensure that $\tilde{f}$ is square-free. One may now compute the irreducible factor $\tilde{h}$ of $\tilde{f}$ of positive degree in x_1 using the earlier algorithm. The complete algorithm for factoring f is discussed in details in Lenstra [46].

There are a number of computational problems in which one wants the degrees of the factors of a polynomial over a finite field without needing the

factors themselves. Factorization of polynomials over the set of rational numbers Q provides one example. Gunji-Arnon [33] have presented an algorithm for determining the degrees of the factors of a polynomial over a finite field.

A strongly related problem to the factoring problem is the problem of determining the roots of a polynomial. This is a classical problem with applications in many branches of engineering. Although many sequential algorithms have been designed to obtain roots, not many fast parallel algorithms are known. BenOr-Feig-Kozen-Tiwari [4] have shown that this problem is in NC if all the roots of the polynomial are real. The basic strategy of root finding is to factor the given polynomial into its approximate linear factors, and hence approximately determine all its roots. This factorization is achieved by recursively factoring the given polynomial into two approximate factors of almost equal degree. These factors may be obtained by numerically evaluating a contour integral and then using the Newton identities.

We now present an outline of the algorithm for simultaneously determining all roots of a polynomial f(x). One may assume the polynomial to be square-free and monic. If the polynomial has multiple roots, well-known methods can be used to reduce the problem to that of determining the roots of a square free polynomial. If f(x) is not monic, it may be divided by the leading coefficient and then the algorithm is applied.

Algorithm 3.12. Roots.

Input: **A polynomial f of degree n.**

Output: **Approximations to the roots of f.**

Factor $f(x)$ recursively in the following manner until all monic linear factors are found.

1. ***Find a point w that seperates the roots of $f(x)$ into two sets L and R, those to the left and to the right of w, respectively, each containing between 1/4 and 3/4 of all roots of $f(x)$. w should not be too close to any root of $f(x)$.***
2. ***Using a numerically evaluated contour integral and the Newton identities, determine approximations to the two monic factors $f_1(x)$ and $f_2(x)$ of $f(x)$ with roots L and R, respectively.***

The authors [4] have shown that the above algorithm can be implemented in $\log^{O(1)}(m + n + v)$ steps using $(m + n + v)^{O(1)}$ processors on a PRAM, where m is the length of the integer coefficients in bits, n is the degree of the polynomial, and v is an error tolerance, and each processor in the PRAM machine is considered capable of perforing a real arithmetic operation in one step.

4. EVALUATION OF POLYNOMIALS

The evaluation of a polynomial is one of the most widely encountered operations in computing. The problem of efficient and accurate numerical evaluation of a polynomial had already received considerable attention in the

1950s when first realization of the computer power came about. This section explores the major results of these studies.

One of the most important results in this area is due to Valiant-Skyum-Berkowitz-Rackoff [67] and shows that any polynomial of degree d which can be computed sequentially in C steps can be computed in parallel in $O((\log d)(\log C + \log d))$ steps. This was an improvement of the earlier result by Hyafil [37] and Valiant [66].

Let F be a field, and let $F[x_1, \cdots, x_n]$ be the ring of polynomials over indeterminates $x_1, \cdots, x_n$ with coefficients from F. A program **P** over F is a sequence of instructions $v_i \leftarrow v_i' \circ v_i''$, $i = 1, \cdots, C$, where for each value of i, v_i', v_i'' are in $F \cup \{x_1, \cdots, x_n\} \cup \{v_1, \cdots, v_{i-1}\}$, and $\circ$ is one of the two ring operators + or $\times$. **P** is called a homogeneous program of degree d if

1. If $v_i \leftarrow v_i' + v_i''$ then v_i' and v_i'' are homogeneous polynomials of the same degree.
2. **P** has no division.
3. If $v_i \leftarrow v_i' + v_i''$ then v_i' and v_i'' are homogeneous and the degree of v_i is less than or equal to d.

If a homogeneous program, **P**, is used to compute the polynomials $f_1, \cdots, f_m \in F[x_1, \cdots, x_n]$ with $C_d(f_1, \cdots, f_m)$ denotes the minimum number of nonscalar multiplications necessary for this computing, then there exist two sets of homogeneous polynomials: $\left\{U_i \mid 1 \leq i \leq I\right\}$, where I is $n + C_d(f_1, \cdots, f_m)$; and $\left\{V_{i,\,j} \mid 1 \leq i \leq I, 1 \leq j \leq \lambda\right\}$, where λ is the number of operations of **P**. These two sets satisfy the following:

1. $d/3 \leq \deg(U_i) \leq 2d/3$ for $1 \leq i \leq I$.
2. $\deg(V_{i,j}) \leq 2d/3$ for $1 \leq i \leq I$, $1 \leq j \leq \lambda$.
3. If P computes f_i $(1 \leq i \leq \lambda)$ and $d/3 \leq \deg(f_i) \leq d$ then $f_i = \sum U_j V_{j,i}$ for $1 \leq j \leq I$.
4. $C_d(U_i) \leq C_d((f_1, \cdots, f_m)$ for $1 \leq i \leq I$.
5. $C_d(V_{i,j}) \leq C_d((f_1, \cdots, f_m)$ for $1 \leq i \leq I$, $1 \leq j \leq \lambda$.

The proof of these properties is constructed by induction on $L = (f_1, \cdots, f_m)$. Interested reader is refered to Hyafil [37].

Let f be a homogeneous polynomial of degree $\leq$ d in n indeterminants. By the above result, $f = \sum U_j V_j$, where $1 \leq j \leq I$ with $I \leq n + C_d(f)$ and U_j and V_j satisfy properties (1), (2), (4), and (5). The following algorithm computes f in $O(\lceil(1/\log 3 - 1) \log d\rceil)$ parallel multiplications and in

$$\left(1 + \frac{\log d}{\log 3 - 1}\right)\left(\lceil \log\, [C_d(f) + n]\rceil + 1\right)$$

parallel steps, and it is due to Hyafil [37].

Algorithm 3.13. Multivariate polynomials computation I.
Input: A multivariate homogeneous polynomial f.
Output: The computation of f.

1. *Write f as* $f = \sum U_j V_j$, where $1 \leq j \leq I$ as above.
2. *Compute* U_j *and* V_j *for* $1 \leq j \leq I$.
3. *Multiply* U_j *by* V_j.
4. *Sum up.*

Applying the induction hypothesis shows that Step 2 can be computed in

less than $(1/(\log 3 - 1)) \log (2d/3)$ parallel multiplicative steps, and $\log(2d/3)$ $(\log 3 - 1) (\lceil \log (C_d(f)+n) \rceil + 1)$ parallel steps. All the multiplications in Step 3 can be performed in parallel in one step. Step 4 requires $\lceil \log l \rceil \leq \lceil \log (n + C_d(f)) \rceil$ steps. Adding the total number of steps gives the stated complexity of computing f.

The main result of Hyafil [37] can be stated as follows:

A polynomial f of degree $\leq$ d in n indeterminates which can be computed with $C^*(f)$ multiplications/divisions can be computed with no more than

$$\lceil (1/[\log 3\ 1]) \log d \rceil$$

parallel multiplicative steps, and

$$\lceil 1 + \frac{\log d}{[\log 3\ -1]} \rceil \lceil \log \left(\left(\frac{d(d-1)}{2} \right)^2 C^*(f) + n \right) + 1 \rceil + \lceil \log d \rceil$$

total parallel steps.

The proof is direct from the observation that if $f_1, f_2, \cdots, f_d$ are the d homogeneous components of f, then $C_d(f_1, \cdots, f_d) \leq (d(d-1)/2)^2 C^*(f)$. To compute f in parallel, one may proceed as follows:

Algorithm 3.14. Multivariate polynomials computation II.
Input: A multivariate homogeneous polynomial f.
Output: The computation of f.

1. *Compute each of the d homogeneous components of f: f_1, $f_2, \cdots, f_d$.*
2. *Add these components in parallel.*

Step 2 requires $\lceil \log d \rceil$ additive steps. Clearly the total complexity is

given by the expressions stated earlier.

Valiant [66], later proved that an n-variables, degree d polynomial, f, that can be computed by some homogeneous program $\pi(C, d, n)$, having C nonscalar multiplications can be computed in $\lfloor \log_{3/2} d \rfloor$ parallel nonscalar multiplications and in $\lfloor \log_{3/2} d \rfloor (\lceil \log_{3/2} C \rceil + 1) + \lceil \log_2 n \rceil + 1$ total parallel steps, and that f has formula size less than $2n(2C)^{\log_{3/2} d}$.

If $f \in \pi(C, d, n)$ and $d \geq 2$ then $f = \sum_{i=1}^{C} g_i h_i$ for some g_i, $h_i \in \pi(C, \lfloor 2/3\, d \rfloor, n)$. Valiant's result [66] is then proved by using this fact to carry out induction on d. Clearly there are $\log_{3/2} d$ inductive steps, and each can be implemented in one parallel nonscalar multiplication and $\lceil \log_2 C \rceil$ parallel additions. $\pi(C, 1, n)$ consists of linear forms and can be computed in $\lceil \log_2 n \rceil + 1$ nonscalar operations. Similar induction can be used to prove the second result.

Unfortunately, the above two results by Hyafil [37] and Valiant [66] require $C^{\log d}$ processors. Thus even if C and d are both bounded polynomially in n, the number of processors required would not be. Valiant-Skyum-Berkowitz-Rackoff [67] have given an improved construction that achieves the same time bound but with only $(Cd)^{\beta}$ processors, for some constant β.

Let f be a homogeneous program. Let C and $f(v_C)$ denote the size (the number of instructions) and the polynomial under computation. Assume that f is the smallest possible program for computing $f(v_C)$. Let v, w be in $\{v_i\} \cup \{x_i\} \cup F$. Define $f(v; w) \in F[x_1, \cdots, x_n]$ by induction on the depth of w as:

$$
\begin{aligned}
f(v; w) &= 1 && \text{if } w = v; \\
&= 0 && \text{if } w \in F \cup \{x_i\};
\end{aligned}
$$

$$= f(v;\, w') + f(v;\, w'') \quad \text{if } w \leftarrow w' + w'';$$
$$= f(w'')\, f(v;\, w') \quad \text{if } w \leftarrow w' \times w''.$$

The main result of Valiant-Skyum-Berkowitz-Rackoff [67] can be stated as follows:

Let f be a homogeneous program of size C which computes a polynomial p of degree d. Then there is a program f' of size $O(C^3)$ which computes p such that the largest depth of any node is $O(\log C \log d)$.

To prove this result one may proceed in $\lceil \log d \rceil$ stages to construct f'. Each stage will add at most log C to the depth of any node.

Algorithm 3.15. Multivariate polynomials computation III.
Input: A multivariate homogeneous polynomial f.
Output: The computation of f.

1. *At stage* 0, *compute all $f(w)$ and $f(v;\, w)$ that have degree at most* $2^0=1$.
2. *At stage* $i+1$, *compute all $f(w)$ and $f(v;\, w)$ that have degree in the range* $(2^i, 2^{i+1}]$.

In Step 1, a depth of $2 + \lceil \log d \rceil$ is sufficient since the polynomials are linear in n indeterminates and $C \geq n-1$ if f is minimal. In Step 2, f(w) can be written as $f(w) = \sum f(t)\, f(t;\, w) = \sum f(t')\, f(t'')\, f(t;\, w)$, where t is such that $t \in V_a = \left\{t \in \{v_i\} \bigcup d(t) > a,\ t \leftarrow t' \times t'',\ d(t') \leq a, \text{ for some } a > 0\right\}$. Take $a = 2^i$. By definition of V_a, each $f(t')$, $f(t'')$, and $f(t;\, w)$ has already been computed. So f(w) can be computed adding $O(\log C)$ depth. Similarily, if $a = d(v) + 2^i$, then $f(v;\, w) = \sum f(v;\, t)\, f(t;\, w) = \sum f(t'')\, f(v;\, t')\, f(t;\, w)$. Each $f(v;\, t')$ and f(t;

w) has already been computed. So f(v; w) can be computed adding only $\mathcal{O}(\log C)$ depth. The size of the new program is dominated by the time to compute the f(v; w). There are C^2 choices of pairs (v, w) and the computation of each f(v; w) takes $\mathcal{O}(C)$ steps. The overall size is, therefore, $\mathcal{O}(C^3)$.

Even though nonhomogeneous programs can also be used for polynomial computing, Strassen has shown that forcing f to be homogeneous is not a serious restriction [64]. His result states that if a polynomial p of degree d is computed by a nonhomogeneous program f of size c, then there is a homogeneous program of size $\mathcal{O}(cd^2)$ which computes $d+1$ polynomials whose sum is the polynomial p. If this fact is combined with the main result of Valiant-Skyum-Berkowitz-Rackoff [67], one gets the following: Let f be a nonhomogeneous program of size C which computes a polynomial p of degree d, then there is a homogeneous program of size $\mathcal{O}((Cd^2)^3)$ and depth $\mathcal{O}(\log C + \log d) \log d)$ which computes p.

Finally, the following result was given by Valiant-Skyum-Berkowitz-Rackoff [67] without proof. Let f be a nonhomogeneous program of size C and degree d[1]. Then there is a program f' of size $\mathcal{O}(C^3)$ and depth $\mathcal{O}(\log C \log d)$ which computes the same polynomial.

[1]The degree of a program is defined as the maximum degree of any node. The degree of a multiplication node is the sum of the degrees of its inputs; the degree of an addition node is the maximum degree of its inputs. The degree of a field member is 0; the degree of an indeterminate is 1.

Chapter 4
PARALLEL ALGORITHMS FOR INTEGER ARITHMETICS

In this chapter, we review some of the important results on integer arithmetic operations performed in parallel. The GCD algorithm is discussed in Section 4.1. The parallel evaluation of straight-line code is considered in Section 4.2. Computing powers in parallel is discussed in Section 4.3. Because of the need for breivity, we are unable to include several other interesting results here. These include integer addition of two n bit numbers performed in $O(\log n)$ time using n processors as given by Ladner-Fischer [44]; integer multiplication given by Schonhage-Strassen [62] requiring $O(\log n)$ time with $(n \log \log n)$ processors; integer division requires $O(\log n)$ parallel time with a polynomial number of processors, Beame-Cook-Hoover [3], and the earlier algorithm by Cook [21] requiring $O((\log n)^2)$ time and n^2 processors.

1. THE GREATEST COMMON DIVISOR

If A and B are integers, not both zero, then their greatest common divisor, GCD(A, B), is the largest integer that evenly divides both A and B. One of the oldest and best known algorithms to calculate the greatest common

divisor of two integers without factoring them was discovered 2250 years ago; this is "Euclid's Algorithm". The algorithm can be stated as follows:

1. *Interchange A and B if $A < B$.*
2. *If $B = 0$, then the $GCD(A, B) = A$, and the algorithm terminates.*
3. *Set $B \leftarrow A \bmod B$ and $A \leftarrow B$ and go to Step* 1.

Since one can easily verify that the $GCD(A_1, A_2, \cdots, A_n) = GCD(A_1, GCD(A_2, \cdots, A_n))$, the Euclidean algorithm can be generalized to calculate the greatest common divisor of n integers. One may proceed as follows:

Algorithm 4-1. Euclid's algorithm for integer GCD.
Input: The integers $A_1, \cdots, A_n$.
Output: The greatest common divisor of $A_1, \cdots, A_n$.

```
1. Set $d = A_n$, $j = n-1$.
2. If $d \neq 1$ and $j > 0$, set $d = GCD(A_j, d)$ and $j = j-1$,
   and repeat this step. Otherwise, $d = GCD(A_1, \cdots, A_n)$.
```

Euclid's algorithm is an effective sequential algorithm for the GCD problem. Schonhage [61] has obtained the best known serial running time of $O(n \log^2 n \log \log n)$ for a sequential algorithm. Brent-Kung [12] have parallelized his algorithm and achieved a running time of $O(n)$ using n processors arranged in a systolic array. The parallelism reduces the bit operations, but it still requires n iterations. The parallel algorithm discussed here is by Kannan-Miller-Rudolph

[41]. It is sublinear and has a running time of $O(n \log\log n/\log n)$. Recently, Chor-Goldreich [20] have improved this running time by getting rid of the log log n term.

In the classical Euclidean algorithm, A is replaced with A mod B, or with A − q B, where q is the quotient when A is divided by B. Kannan-Miller-Rudolph's [41] algorithm computes $p A - q_p B$ in parallel for $p = 0$ to n, where q_p is the quotient when pA is divided by B. Since all of these integers are between 0 and B, then there are at least two that agree on leading log n bits by the pigeon-hole principle. Thus their difference is a nonnegative integer with at most $(n - \log n)$ bits. Replacing A by their difference would reduce the problem size by log n bits during each two iterations, thus requiring only $O(n/\log n)$ iterations.

The following lemmas handle the two problems that may arise in this situation. The first lemma characterizes the changes in the GCD(A, B) when A is replaced p A − q B. The second shows the application of the pigeon-hole principle to reduce the number of bits during an iteration.

Lemma 1: If $g = \mathrm{GCD}(A, B)$; $h = \mathrm{GCD}(p A - q B, B)$ then g divides h and (h/g) divides p.

Since p is at most n, the only extra factors that are introduced into the GCD when A is replaced by p A − q B are made up of powers of primes between 0 and n. At the outset of the algorithm, all prime factors of magnitude at most n between A and B can be removed (in $O(\log n)$ time), and the entire algorithm may be run. At the completion of the algorithm, the extra factors

introduced in the GCD by the replacement can be removed quickly.

Lemma 2: **If a, b, and n are positive integers and $a \leq b\, n$ then there exist integers p and q not both zero such that $|p| \leq n\, b/a$, $|q| \leq 2\, n$ and $0 \leq p\, a - qb \leq a/n$.**

We want to find p and q_p with p between $-n$ and n such that $(p\, A - q_p\, B)$ is an integer with at most $(n - \log n)$ bits. We thus need $p\, A - q_p\, B$ to satisfy $0 \leq p\, A - q_p\, B \leq \min(B, 2^{(n-\log n)})$. It should be noted that only $O(\log n)$ most significant bits of A and B are considered in order to find p and q_p that satisfy these conditions.

The algorithm makes use of the ordinary sequential Euclidean algorithm once the numbers get small. It also uses the long division which can be performed in parallel time of $O(\log^2 k)$ where k is the difference between the number of bits of A and that of B, since only $O(k)$ bit integers are to be dealt with. The long division is used only when k exceeds $(\log n)^2 + 1$. The algorithm can be stated as follows:

Algorithm 4.2. Integer GCD.
Input: Two integers A and B.
Output: The greatest common divisor of A and B.

<u>*MAIN*</u> <u>*PROGRAM*</u>

```
1. If $A < B$ then swap them.
2. Let $n$ and $m$ be the number of bits of $A$ and $B$
   respectively, (i.e., $n = \#A$, $m = \#B$)$^2$.
3. If $m \leq 2\,(\log n)^2$ then
```

```
4. Find $C = A \pmod{B}$ by long division.
5. Find the GCD(C, B) using the usual serial Euclidean
   algorithm, and return with result.
6. Remove small common factors from A and B, and call
   the product SF.
7. Repeat Procedure DoAPhase(A, B) until $m < 2(\log n)^2$.
8. Remove small factors from A and from B.
9. Find $C = A \pmod{B}$ by usual long division.
10. Run the serial Euclidean algorithm on C, B to get $g'$
11. Return $GCD \leftarrow SF * g'$.
```

<u>*PROCEDURE*</u> <u>*DoAPhase (A, B)*</u>

```
1. $k \leftarrow n$; $s \leftarrow 2(\log n)^2$.
2. If $n-m > (\log n)^2 + 1$ then call LongDivide(A, B), else
3. $a \leftarrow A[k : k - s + 1]$,   $b \leftarrow B[k : k - s + 1]$,   $T \leftarrow$
   identity matrix, $endsize \leftarrow \#a + \#b - (\log n)^2$.
4. Repeat Procedure DoAnIteration (a, b, T) until $(\#a +$
   $\#b) < endsize$.
5. $(A\ B)^t \leftarrow T(A\ B)$.
6. Replace A and B by their absolute values.
7. If $A < B$ then swap their values.
```

<u>*PROCEDURE*</u> <u>*LongDivision*</u>

```
0. $l \leftarrow \#B$.
1. $k \leftarrow \#A - \#B$.
2. $a \leftarrow$ most significant min(2k, k+l) bits of A.
3. $b \leftarrow$ most significant min(k, l) bits of B.
4. $q \leftarrow \lfloor a/b \rfloor$.
5. $C \leftarrow A - q\,B$.
6. If $C < 0$ then $C \leftarrow C + 4\,B$.
```

[2]We use #L to denote number of bits in L and L[u : v] to denote the integer formed by bits (u, u+1, ⋯, v) of L.

```
7. $A \leftarrow B$.
8. $B \leftarrow C$.
```

PROCEDURE DoAnIteration (a, b, T)

```
1. If $a/b \geq n$ then find a $q$ such that $q = \lfloor a/b \rfloor$, $p \leftarrow 1$.
2. else, find a pair $(p, q)$, where $|p| \leq n\, b/a$, and $|q| \leq 2\, n$,
   such that $0 \leq p\, a - q\, b \leq a/nm$.
3. $T \leftarrow \begin{bmatrix} 0 & 1 \\ p & -q \end{bmatrix} T$.
4. $(a \quad b)^t \leftarrow \begin{bmatrix} 0 & 1 \\ p & -q \end{bmatrix} (a \quad b)^t$.
```

The time complexity of this algorithm may be determined as follows. The repeat loop of the main program is executed at most $n/(\log n)^2$ times, since each iteration removes at least $(\log n)^2$ bits from the sum of the bits of A and B. However, each call of PROCEDURE DoAPhase (A, B) in the loop, involves a call of PROCEDURES LongDivide or DoAnIteration. Thus, it is important to determine the running time of each of these inner procedures.

PROCEDURE DoAnIteration can be executed in $O(\log \log n)$ parallel time using $n^2(\log n)^2$ processors. This can be shown as follows: Step 1 requires no more than $O(\log \log n)$ time because it is a multiplication of two $2(\log n)^2$-bits numbers. Finding the q in Step 1 can also be performed in this time bound by assigning $(\log n)^2$ processors to each of the n equations "a − b q". Step 2 also takes $O(\log \log n)$ time by using $(\log n)^2$ processors for each of the n^2 equations "p a − q b". Steps 3 and 4 can be computed in $O(\log \log n)$ time since the entries in the matrices are no graeter than $O((\log n)^2)$ bits.

PROCEDURE LongDivide can be executed in $O(\log n)$ time using no more than $O(n)$ processors. To show this, note that Step 1 of this procedure can easily be computed using a binary fan-in tree and n processors in $O(\log n)$ time. Steps 2 and 3 take constant time to identify the appropriate bits. In Step 4, the division of a 2k-bit number by a k-bit number can be done in $O((\log k)^2)$ time with k processors. Step 5 is simply a multiplication of a k-bit number by an n-bit number and this takes no more than $O(\log n)$ parallel time with n processors. The subtraction is also done within this time bound.

Since PROCEDURE DoAPhase may invoke PROCEDURE DoAnIteration (no more than (log n) times) or PROCEDURE LongDivide, from the previous discussion, it follows that it requires $O(\log n \log \log n)$ parallel time and uses $n^2(\log n)^2$ processors. Each of Steps 2 and 4 of the procedure requires $O(\log n)$ time using n processors. Therefore, each execution of the repeat loop in the main program takes no more than $O(\log n \log \log n)$ parallel time.

Hence, it can be seen that the GCD of two integers, each represented in at most n bits, requires parallel time $O(n \log \log n/\log n)$ using $n^2(\log n)^2$ processors. This follows immediately from the previous discussion of DoAnIteration, LongDivide, and DoAPhase provided we can remove the small common factors in $O(n/\log n)$ parallel time. Since the small prime factors of an n-bit number can be identified in $O((\log n)^2)$ time, the complexity result follows.

2. THE EVALUATION OF A STRAIGHT-LINE CODE

For arithmetic algorithms, the most basic models of computation are arithmetic circuits, using inputs, constants from the ground fields F or semi-ring R, and operations +, −, *, /. Straight-line programs are special cases of arithmetic circuits. An arithmetic circuit is an edge-weighted directed acyclic graph satisfying the following conditions:

1. Each node is labeled as one of three types: a leaf, a multiplication node, or an addition node.
2. Leaves are assigned a value in F or R, denoted value(v) for a leaf v.
3. The indegree of a leaf node is zero, a multiplication node is two, and an addition node is nonzero.
4. All edges are directed away from leaves.
5. There are no edges from multiplication nodes to multiplication nodes.

A straight-line program over a commutative semi-ring $R=(R, +, \times, 0, 1)$ is a sequence of assignment statements of the form $a \leftarrow b + c$ or $a \leftarrow b \times c$, where b and c are either elements of R or previously assigned variables.

Given a straight-line program, one may obtain its arithmetic circuit by constructing a node for each statement and for each input variable, and an edge from node i to node j if j is a statement that uses the variable evaluated at statement i. All edge weights are set to 1, and nodes corresponding to input variables are given values assigned to the corresponding variables.

Arithmetic networks use these arithmetic operations and also Boolean inputs, constants, and operations. The interface is given by "sign" gates, which

take an arithmetic input *a* in F or R and produce a Boolean value according to whether *a* is zero or not, and by "selection" gates, which produce the first or second of their two arithmetic inputs according to the value of the one Boolean input.

Strassen [65] and Ben-Or [5] have discussed sequential algorithms on the related model of "algebraic decision trees". Miller-Ramachandran-Kaltofen [49] have given a new and efficient parallel algorithm to evaluate a straight line program. The algorithm evaluates a program over a commutative semi-ring R of degree d and size n in time $O((\log n)(\log nd))$ using M(n) processors, where M(n) is the number of processors required for multiplying $n \times n$ matrices over R in $O(\log n)$ time. This result is a generalization of the result of Valiant-Skyum-Berkowitz-Rackoff [67] discussed in Chapter 3. That paper considers the problem of transforming a straight-line program into a program of "shallow" depth[3]. Their transformation is performed by a sequential polynomial time algorithm. As against the off-line algorithms presented in the previous papers, they show the construction of this "shallow" on-line program with the same size and time bounds and no preprocessing. Further, the algorithm does not need to know the degree of the circuit in advance. Let U be an upper triangular matrix with zero diagonal, representing an arithmetic circuit. An entry U_{ij} of this matrix U is the weight on the edge from node v_i to node v_j if the edge exists; it is zero otherwise. The following three submatrices may be derived from U:

[3]The depth (or height) of a coputational tree is the length of the longest path in it and in arithmetic circuits, it represents the time required for parallel execution of the computation.

$U(+, +)_{ij} = U_{ij}$ if v_i and v_j are addition nodes; it is zero otherwise,

$U(X, +)_{ij} = U_{ij}$ if v_j is an addition node, it is zero otherwise, and

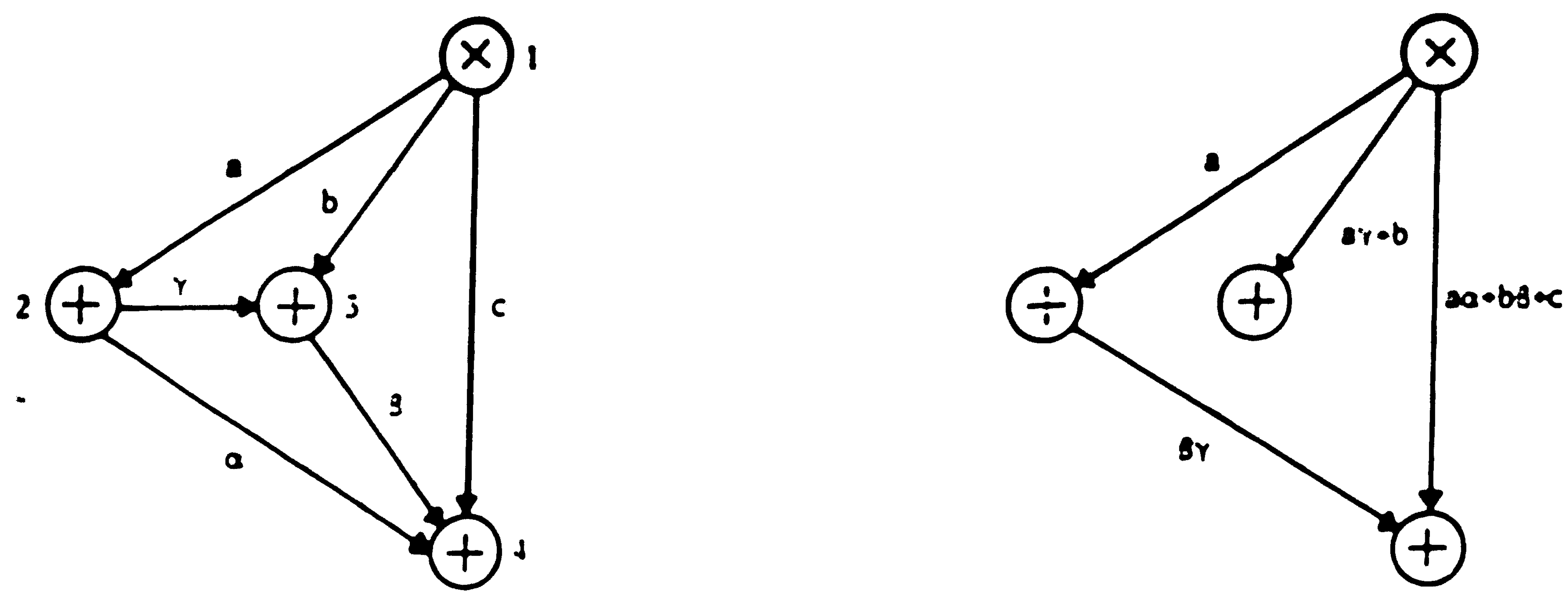
$U(X, X)_{ij} = U_{ij}$ if v_i or v_j is not an addition node, it is zero otherwise. The algorithm is described below.

Algorithm 4.3. Evaluation of straight-line programs.
Input: An arithmetic circuit.
Output: The evaluation of the arithmetic circuit.

```
Procedure Phase (U)
Begin
    $U \leftarrow MM(U)$
    $U \leftarrow Eval_{+}(U)$
    $U \leftarrow Eval_{X}(U)$
End.

Procedure MM(U)
$U \leftarrow U(X, +) * U(+, +) + U(X, X)$

Procedure Eval_+(U)
For all addition nodes $v_j$ whose children $v_k$ and $v_l$, both of
which are leaves, do
        $value(v_j) \leftarrow \sum value(v_i) * U_{ij}$  for $1 \leq i \leq n$.
        Set $v_j$ to a leaf  $U_{ij} \leftarrow 0$  for $i \in \{1, \cdots, n\}$.

Procedure Eval_X(U)
For all multiplication nodes $v_j$ with children $v_k$ and $v_l$, both
of which are leaves, do
        $value(v_j) \leftarrow value(v_k) * vlaue(v_l)$
        Set $v_j$ to a leaf  $U_{kj} \leftarrow 0$ and $U_{lj} \leftarrow 0$.
```

```
For all $U_{ji}$ where $v_j$ is a multiplication node with children
$v_k$ and $v_l$ and $v_k$ is a leaf and $v_l$ is not do
        $F_{lji} \leftarrow value(v_k) * U_{ji}$
For all pairs $(l, i)$ do
        $W_{li} \leftarrow \sum_j F_{lji}$                    (*)
        $U_{li} \leftarrow U_{li} + W_{li}$
        $U_{ji} \leftarrow 0.$
```

Procedure Phase takes as input an arithmetic circuit and returns a new circuit with the same nodes such that each node has the same value as before. Repeated application of procedure Phase eventually returns with the value of the circuit. Procedure MM, Matrix Multiplication, uses one matrix multiplication and one matrix addition over R. Thus it can be performed in $O(\log n)$ time using $O(n^{2.49})$ processors. Figure 4-1 below shows the effect of applying Procedure MM to an arithmetic circuit. Procedures Eval_{+}, plus evaluate, and $\mathrm{Eval}_{\times}$, multiplication evaluate, simply evaluate an addition node or a multiplication node if all its children have been evaluated. They can be performed in $O(\log n)$ time using only $O(n^2)$ processors. To see that $\mathrm{Eval}_{\times}$ can be performed with $O(n^2)$ processors, note that the number of terms F_{lji} in line (*) is at most equal to the number of edges. Thus, we simply sort these terms on their key (l, i) using a randomized parallel bucket sort or a deterministic comparison-based sorting algorithm, and then sum the terms using parallel list-ranking. Figure 4-2 shows the effect of applying $\mathrm{Eval}_{\times}$ to a circuit.



$$\begin{pmatrix} 0 & a & a\gamma+b & a\alpha+b\beta+c \\ 0 & 0 & 0 & \beta\gamma \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & a & b & c \\ 0 & 0 & \gamma & \alpha \\ 0 & 0 & 0 & \beta \\ 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & \gamma & \alpha \\ 0 & 0 & 0 & \beta \\ 0 & 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & a & b & c \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 4-1. An arithmetic circuit before and after an application of procedure MM, [49].

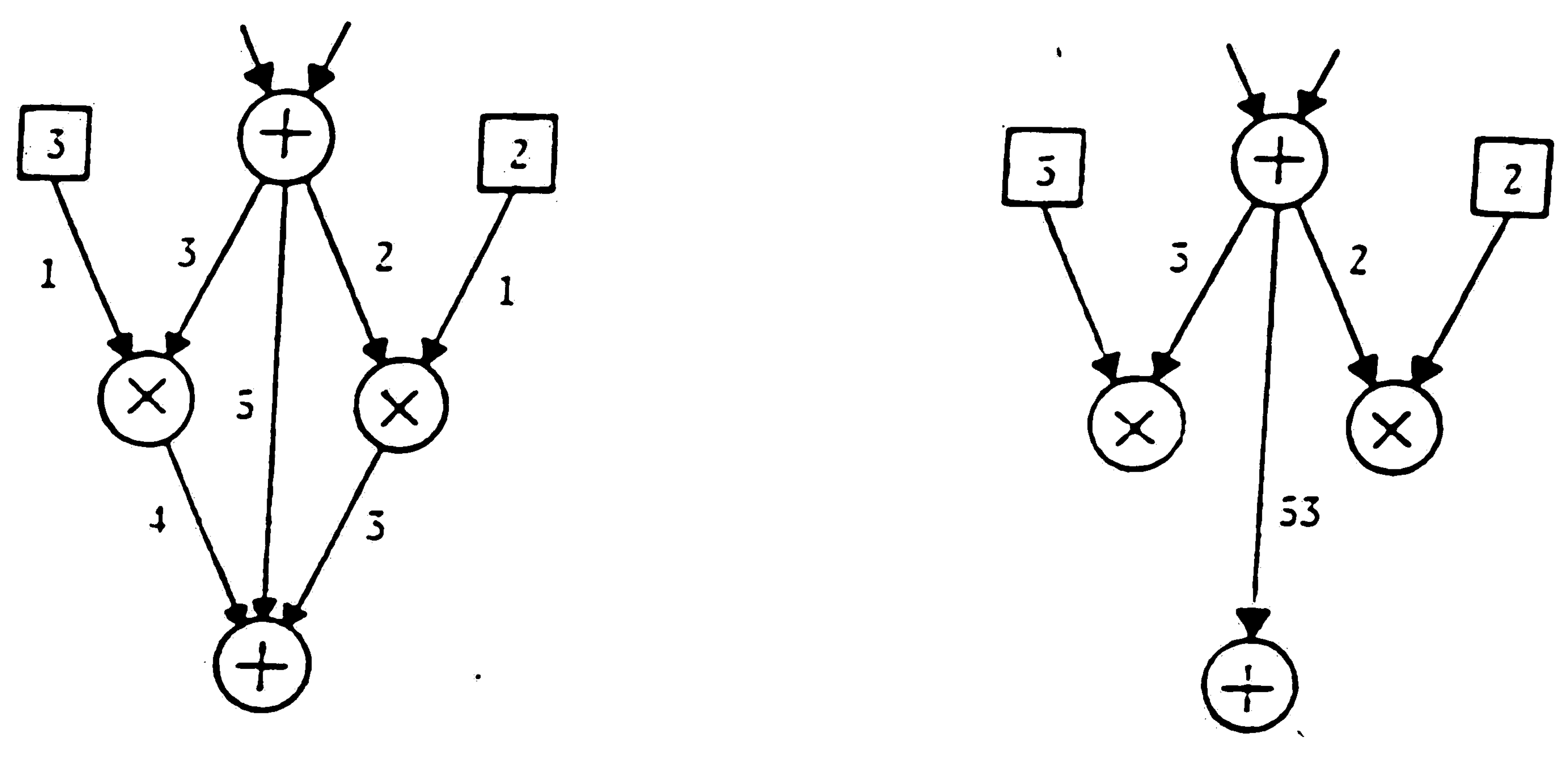


Figure 4-2. An arithmetic circuit before and after an application of prcoedure $\mathrm{Eval}_{\times}$, [49].

Before we state the main result of Miller-Ramachandran-Kaltofen [49], the definition of the height of an arithmetic circuit should be given. The height of a circuit U is the maximum height of any node in U. The height of a node is defined inductively by:

1. **A leaf has height 1.**

2. A multiplication node has height equal to the sum of the heights of its children.

3. If v is an addition node then the height of v equals max(a+1/2, m), where a equals the maximum height of any child v which is an addition node, and m equals the maximum of the heights of the children which are either a leaf or a multiplication node.

If a circuit has height h, then after $\lceil \log h \rceil$ applications of procedure Phase the rêsulting circuit will contain only leaves and output nodes. Thus, in one more application of Phase (only Eval_+ and $\mathrm{Eval}_\times$ are needed) all nodes will be leaves, i.e., the circuit has been evaluated. With a slightly more careful analysis the number of applications can be bounded by $\lfloor \log h \rfloor + 1$. Now the main result of Miller-Ramachandran-Kaltofen can be stated as follows [49]:

If U is an arithmetic circuit of degree d and size n then the value of V can be computed in time $O((\log n)(\log nd))$ using at most M(n) processors. To prove this, note that procedure Phase need only be applied $\lfloor \log h \rfloor + 1$ times, where h is the height of the circuit U. Now it can be easily shown that $h = O(e.$ $d)$, where e is the number of plus-plus edges. As a matter of fact $h \leq 1/2ed + d$, (see [49] for the proof). Thus, procedure Phase is applied $O(\log nd)$ times. Now, each application of Phase requires only (log n) parallel time. The processor-expensive step is the matrix multiplication in procedure MM, which can be performed using $O(M(n))$ processors.

In Figure 4-3, the effect of applying the different procedures to a circuit is shown. Starting with the circuit (a) and applying procedure MM, one obtains circuit (b), to which procedure Eval_{+} may be applied obtaining circuit (c), to which procedure $\text{Eval}_{\times}$ may then be applied obtaining circuit (d).

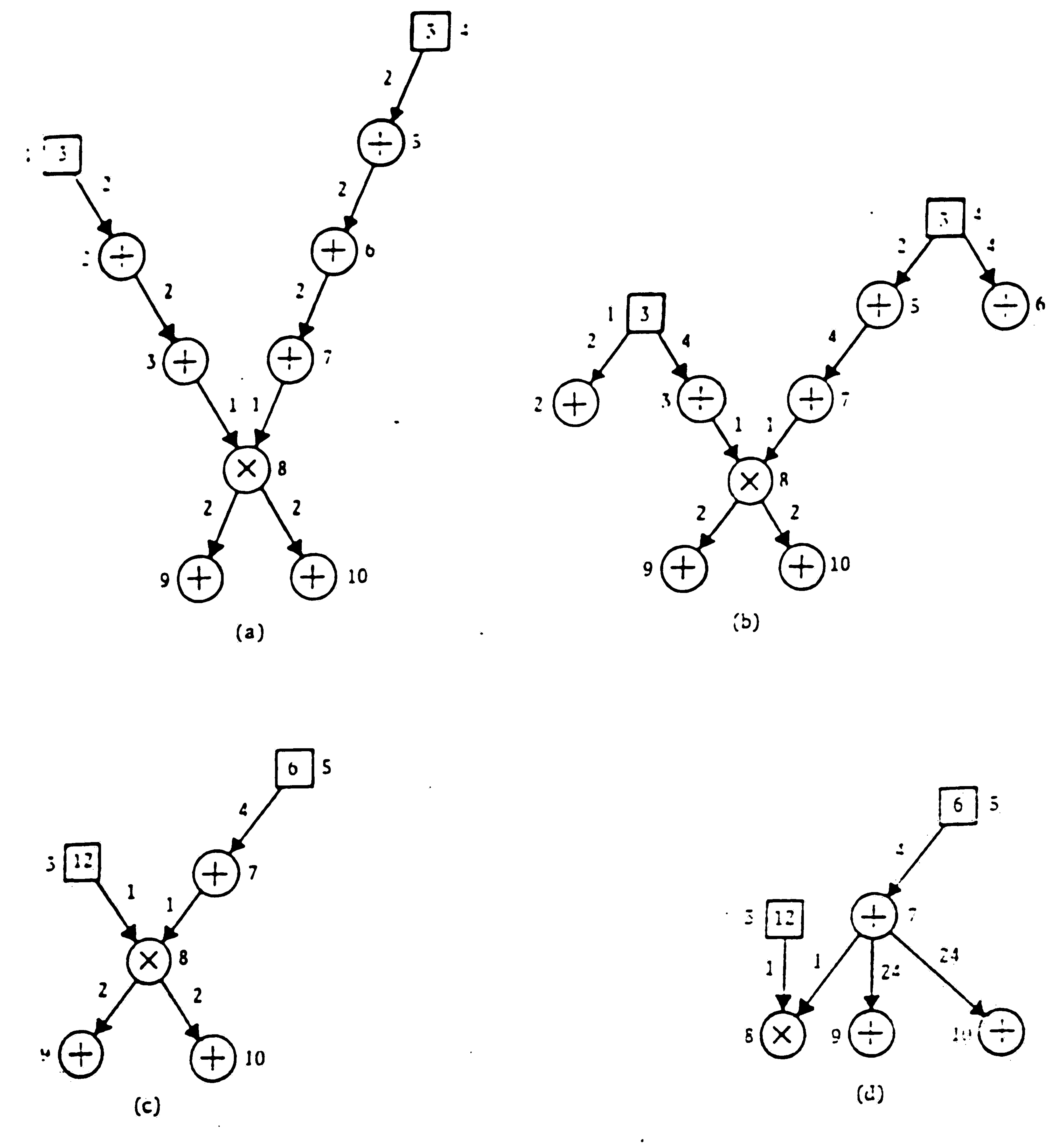


Figure 4-3. An arithmetic circuit after successive application of the procedures: MM, Eval_{+}, and $\text{Eval}_{\times}$, [49].

Several new related results have appeared since this work. Matrix multiplication can now be performed using $O(n^{2.367})$ processors as shown in Coppersmith-Winograd [22]. The ideas in [49] have been extended to more complex domains by Miller-Teng [50]. Finally, an analysis of the main theorem has been found that does not use the height metric, Mayr [48].

3. COMPUTING POWERS IN PARALLEL

In this section, the problem of computing a^b mod m is parallel, where a, b, and m are n-bit integers, is considered. This problem arises as a subroutine in many computational problems, e.g., factoring integers, primality tests, and factoring polynomials over finite fields. The common method of "repeated squares" does not yield fast parallel computations and, therefore, there has been a great deal of activity in this area to find better parallel algorithms.

Gathen [29] has presented fast parallel computations for large powers modulo an element that has only small prime factors. These parallel computations work for integers and polynomials over small finite fields. Subsequent to that paper, Fich-Tompa [26] obtained a fast parallel exponentiation algorithm in large finite fields of small characteristic. This lead to the surprising observation that for certain polynomial computations, Boolean circuits are exponentially more powerful than arithmetic circuits, Gathen-Seroussi [32]

Assume that n is an input size parameter, and that a, b, p, e $\in \mathbb{N}$ such

that $p, e \leq n$; $a, b < 2^n$, and $p \geq 3$ is a prime. The following algorithm computes $c \in \mathbb{N}$ such that $a^b \equiv c \bmod p^e$, and is due to Gathen [29].

Algorithm 4.4. Integer power modulo a prime power.
Input: **Integers $a, b, p, e \in \mathbb{N}$ such that $p, e \leq n$; $a, b < 2^n$, and $p \geq 3$ is a prime**
Output: **$c \in \mathbb{N}$ such that $a^b \equiv c \bmod p^e$.**

1. *If $a = 0$, then return $c = 0$ and stop. Compute $l \in \mathbb{N}$ such that p^l and p^{l+1} do not divide a. If $l\,b \geq e$, return $c = 0$ and stop. Otherwise replace a by a/p^l. Assume now that a is not congruent to 0 mod p.*
2. *Compute r such that $a \equiv r \bmod p$, $1 \leq r < p$.*
3. *Compute s such that $s \equiv \alpha(r) \equiv r^{p^{e-1}} \bmod p$, $1 \leq s < p^e$.*
4. *Compute h and u such that $h \equiv b \bmod p-1$, $0 \leq h < p-1$, and $u \equiv s^h \bmod p^e$, $1 \leq u < p^e$.*
5. *Compute the inverse t of s such that $s\,t \equiv 1 \bmod p^e$, $1 \leq t < p^e$.*
6. *Compute v and w such that $v \equiv a\,t \bmod p^e$, $w \equiv \sum \underset{i}{\overset{b}{C}} (v-1)^i \equiv w \bmod p^e$, and $1 \leq v$, $w < p^e$, where $0 \leq i < e$, and $\underset{i}{\overset{b}{C}}$ is the combination of b over i.*
7. *Return $c=p^{lb}uw$.*

In Step 3, α is an arithmetic curcuit computing u^b, $a^b \bmod m$, $a^b \bmod x^n$, $a^b \bmod x^n$ if a has constant term 1, where $u \in F$, $b \in \mathbb{N}$ with $2^{n-1} < b \leq 2^n$, a and m are in F[x] of degree n are input.

Let div(n) denote some function such that there exist Boolean circuits of depth div(n) and size of $n^{O(1)}$ that compute the division with remainder for n-bit integers. "Long division" yields the trivial bound div(n) = $O(\log^2 n)$. For P-uniform circuits, Beam-Cook-Hoover [3] gave the value $O(\log n)$ to div(n). By Reif [57], div(n) = $O(\log n \log\log n)$ for log-space uniform circuits. The "iterated" product of n n-bit integers can be computed in depth $O(\text{div}(n))$, Beam-Cook-Hoover [3].

Using the above notation of div(n), Gathen [29] showed that the above algorithm can be implemented on a Boolean circuit of depth $O(\log n \text{ div}(n))$ and size $n^{O(1)}$. Moreover, the algorithm works correctly as described. To prove these two claims, note that one can assume that a is not congruent to 0 mod p, i.e., $l = 0$. Also, note that $s \equiv r \equiv a \bmod p$ and $v \equiv 1 \bmod p$. For the details of the proof for the second claim, an interested reader is refered to Gathen [29]. To prove the depth, a quadratic Newton iteration to compute t in Step 5 is used.

$$t_0 = r,\ t_i \equiv t_{i-1} - (-t_{i-1} + t^2_{i-1} s) \bmod p^{2i},\ 1 \leq t_i < p^{2i}.$$

Each iteration step can be performed in depth $O(\text{div}(n))$. This depth is also sufficient for the iterated products required for the binomial coefficients and powers of $v-1$ in Step 6. The depth required by Step 3 is also $O(\text{div}(n))$. Therefore, the depth required by Steps 1, 2, 4, 6, and 7 is $O(\text{div}(n))$. The depths required by Steps 3 and 5 is $O(\log n \text{ div}(n))$. This proves the first claim.

By exploiting the power of P-uniformity, one can actually get Boolean circuits of optimal (up to constant factor) depth, $O(\log n)$, Gathen [29].

Chapter 5
SUMMARY AND CONCLUSIONS

Algorithms can be generally classified in various ways, such as algebraic vs. analytic, finite vs. infinite, and exact vs. approximate. Within recent years a new classification has become important: sequential vs. parallel, brought about by the development of parallel and pipeline computers. These devices allow concurrent arithmetic processing, can easily handle large volumes of information, and often provide hardware facilities for many inherently parallel operations found in numerical linear and polynomial algebras.

Recent surveys have given attention to research in areas such as numerical linear algebra and parallel arithmetic computations. None of these surveys, however, gave a complete and comprehensive survey on polynomial computations. It was our intention to provide a thorough and up-to-date discussion of parallel methods for matrix computations, polynomial operations, and integer arithmetics all in one survey, along with background information concerning the computer methods and fundamental techniques.

In the important subject of matrices and linear systems of equations, several parallel algorithms were presented. Three parallel algorithms to compute the determinant of a given matrix were discussed, along with two algorithms to compute the inverse matrix of a given one. In addition, several parallel algorithms to solve a linear system of equations were introduced. The cases where the coefficient matrix in the system of equation under consideration has a special structure were took into consideration.

Polynomial computation have several important applications. This topic was discussed in Chapter 3. Parallel algorithms to compute the polynomial gcd and finding the roots of a polynomial were discussed. Seven parallel algorithms for polynomial factorization were discussed thoroughly because of the importance of the subject. In addition, three algorithms for polynomial computation were presented.

In Chapter 4, the topic of integer arithmetics and parallel algorithms was discussed briefly. The need for breivity made us unable to include several interesting results here. However, three important integer problems were discussed. The integer GCD was discussed and algorithms to compute it were presented. The evaluation of straight line code was also discussed.

REFERENCES

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

[2] S. G. Akl, The Design and Analysis of Parallel Algorithms, Prentice Hall, Englewood Cliffs, NJ, 1989.

[3] P. W. Beame, S. A. Cook, and H. J. Hoover, "Log Depth for Division and Related Problems", 25th Annual Symposium on the Foundations of Computer Science, pp. 1-6, October 1984. *SIAM Journal on Computing*, vol. 15, pp. 994-1003, 1986.

[4] M. Ben-Or, E. Feig, D. Kozen, and P. Tiwari, "A Fast Parallel Algorithm for Determining all Roots of a Polynomial with Real Roots", *SIAM Journal on Computing*, vol. 17, No. 6, pp. 1081-1092, December 1988.

[5] M. Ben-Or, "Lower Bounds for Algebraic Computation Trees", Proceedings of the 15th Annual ACM Symposium on Theory of Computing, Boston, MA, pp. 80-86, 1983.

[6] S. Berkowitz, "On Computing the Determinant in Small Parallel Time Using a Small Number of Processors", *Information Processing Letters*, vol. 18, pp. 147-150, 1984.

[7] E. R. Berlekamp, "Factoring Polynomials Over Finite Fields", *The Bell System Technical Journal*, vol. 46, pp. 1853-1859, October 1967.

[8] E. R. Berlekamp, "Factoring Polynomials Over Large Finite Fields", *Mathematics of Computation*, vol. 24, No. 111, pp. 713-735, July 1970.

[9] D. Bini, "Parallel Solution of Certain Toeplitz Linear Systems", *SIAM Journal on Computing*, vol. 13, No. 2, pp. 268-276, May 1984.

[10] A. Borodin, J. von zur Gathen, and J. Hopcroft, "Fast Parallel Matrix and GCD Computations", *Information and Control*, vol. 52, pp. 241-256, 1982.

[11] A. Borodin and I. Munro, The Computational Complexity of Algebraic and Numeric Problems, Americal Elsevier, New York, NY, 1975.

[12] R. P. Brent and H. T. Kung, "Systolic VLSI Arrays for Linear Time GCD Computation", VLSI 83, International Federation of Information Processing, 1983.

[13] W. S. Brown, "On Euclid's Algorithm and the Computation of Polynomials Greatest Common Divisor", *Journal of the Association of Computer Machinery*, vol. 18, pp. 478-504, 1971.

[14] D. M. Burton, Abstract and Linear Algebra, Addison-Wesley Publications Company, Inc., Phillipines, 1972.

[15] P. Camion, "A Deterministic Algorithm for Factorizing Polynomials of $F_q[x]$", *Annals of Discrete Mathematics*, vol. 17, pp. 149-157, 1983.

[16] D. G. Cantor and H. Zassenhaus, "A New Algorithm for Factoring Polynomials Over Finite Fields", *Mathematics of Computation*, vol. 36, No. 154, pp. 587-592, April 1981.

[17] S. C. Chen, "Speedup of Iterative Programs in Multiprocessing Systems", Dissertation, Department of Computer Science, University of Illinois, Urbana, IL, 1975.

[18] S. C. Chen and D. J. Kuck, "Time and Parallel Processor Bounds for Linear Recurence Systems", *IEEE Transactions Computers*, pp. 701-717, 1975.

[19] A. L. Chistov and D. Yu Grigoryev, "Polynomial-time Factoring of Multivariable Polynomials Over a Global Field", *Lomi Preprints E*-5-82, Leningrad, 1982.

[20] B. Chor and O. Goldreich, "An Improved Parallel Algorithm for Integer GCD", MIT Laboratory for Computer Science, Cambridge, MA, April 1985, to appear.

[21] S. A. Cook, "The Classification of Problems Which Have Fast Parallel Algorithms", Lecture Notes in Computer Science, vol. 158, Springer-Verlag, New York, Berlin, Heidelberg, 1987.

[22] D. Coopersmith and S. Winograd, "Matrix Multiplication via Arithmetic Progressions (Behrend's Theorem)", Proceedings of the 19th Annual ACM Symposium on Theory of Computer Science, ACM, New York, pp. 1-6, May 1987.

[23] L. Csanky, "Fast Parallel Matrix Inversion Algorithms", *SIAM Journal on Computing*, vol. 5, No. 4, pp. 618-623, December 1976.

[24] L. Csanky, "On Parallel Complexity of Some Computational Problems", Ph.D. Dissertation, Computer Science Division, University of California, Berkley, CA, 1974.

[25] J. H. Davenport and B. M. Trager, "Factorization Over Finitely Generated Fields", Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation (P. Wang ed.), pp. 200-205, 1981.

[26] F. Fich and M. Tompa, "The Parallel Complexity of Exponentiating Polynomials Over Finite Fields", Proceedings of the 17th Annual ACM Symposium on The Theory of Computing, Providence, RI, pp. 38-47, 1985; *Journal of the Association of Computer Machinery*, to appear.

[27] M. J. Flynn, "Very High-Speed Computing Systems", Proceedings IEEE, vol. 54, pp. 1901-1909, 1966.

[28] J. von zur Gathen, "Parallel Algorithms for Algebraic Problems", *SIAM Journal on Computing*, vol. 13, No. , pp. 802-824, November 1984.

[29] J. von zur Gathen, "Computing Powers in Parallel", *SIAM Journal on Computing*, vol. 16, No. 5, pp. 930-945, October 1987.

[30] J. von zur Gathen, "Factoring Polynomials and Primitive Elements for Special Primes", *Theoretical Computer Science*, vol. 52, pp. 77-89, 1987.

[31] J. von zur Gathen and E. Kaltofen, "Factorization of Multivariate Polynomials Over Finite Fields", *Mathematics of Computation*, vol. 45, No. 171, pp. 251-261, July 1985.

[32] J. von zur Gathen and G. Seroussi, "Boolean Circuits Versus Arithmetic Circuits", Proceedings 6th International Conference on Computer Science, Santiago, Chile, pp. 171-184, 1986.

[33] H. Gunji and D. Arnon, "On Polynomial Factorization Over Finite Fields", *Mathematics of Computation*, vol. 36, No. 153, pp. 281-287, January 1981.

[34] **D. Heller, "A Determinant Theorem With Applications to Parallel Algorithms", *SIAM Journal on Numerical Analysis*, vol. 11, No. 3, pp. 559-568, June 1974.**

[35] D. Heller, "A Survey of Parallel Algorithms in Numerical Linear Algebra", *SIAM Review*, vol. 20, No. 4, pp. 740-777, October 1978.

[36] A. S. Householder, The Theory of Matrices in Numerical Analysis, Blaisdell, New York, 1974.

[37] L. Hyafil, "On the Parallel Evaluation of Multivariate Polynomials", *SIAM Journal on Computing*, vol. 8, No. 2, pp. 120-123, May 1979.

[38] L. Jamieson, D. Gannon, R. Douglass, editors.The Characteristics of Parallel Algorithms, The MIT Press, Cambridge, MA, 1987.

[39] E. Kaltofen, "A Polynomial-Time Reduction From Bivariate to Univariate Integral Polynomial Factorization", Proceedings of the 23rd Symposium on Foundations of Computer Science, IEEE, pp. 57-64, 1982.

[40] E. Kaltofen, "Polynomial-Time Reduction from Multivariate to Bivariate and Univariate Integer Polynomial Factorization", *SIAM Journal on Computing*, vol. 15, No. 2, 1985, vol. 14, pp. 469-489, 1984.

[41] R. Kannan, G. Miller, and L. Rudolph, "Sublinear Parallel Algorithm for Computing the Greatest Common Divisor of Two Integers", *SIAM Journal on Computing*, vol. 16, No. 1, pp. 7-16, February 1987.

[42] D. E. Knuth, The Art of Computer Programming, Seminumerical Algorithms, vol. 2, second edition, Addison-Wesley, Reading, MA, 1982.

[43] **D. J. Kuck, Structure of Computers and Computations, Wiley, New York, 1978.**

[44] **R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation",** ***Journal of the Association of Computer Machinery*****, vol. 27, pp. 831-838, 1980.**

[45] **P. Lancaster and M. Tismenetskt, The Theory of Matrices with Applications, second edition, Academic Press, Inc., 1985.**

[46] A. K. Lenstra, "Factoring Multivariate Polynomials Over Finite Fields", *Journal of Computer and System Sciences*, vol. 30, pp. 235-248, 1985.

[47] A. K. Lenstra, H. W. Lenstra, and L. Lovasz, "Factoring Polynomials with Rational Coefficients", *Mathematische Annalen*, vol. 261, pp. 515-534, 1982.

[48] E. W. Mayr, "The Dynamic Tree Expression Problem", Tech. Report STAN-CS-87-1156, Stanford University, Department of Computer Science, May 1987.

[49] G. L. Miller, V. Ramachandran, and E. Kaltofen, "Efficient Parallel Evaluation of Straight-Line Code and Arithmetic Circuits", *SIAM Journal on Computing*, vol. 17, No. 4, pp. 687-695, 1988.

[50] G. L. Miller and S. H. Teng, "Dynamic Parallel Complexity of Computational Circuits", Proceedings of the 19th Annual ACM Symposium on Theory of Computing, ACM, New York, pp. 254-264, May 1987.

[51] J. Modi, Parallel Algorithms and Matrix Computation, Clarendon Press, and Oxford University Press, 1988.

[52] R. T. Moenck, "On the Efficiency of Algorithms for Polynomial Factoring", *Mathematics of Computation*, vol. 31, No. 137, pp. 235-250, January 1977.

[53] D. R. Musser, "Algorithms for Polynomial Factorization", Ph.D.Thesis and TR 134, University of Wisconsin, 1971.

[54] S. E. Orcutt, Jr., "Computer Organization and Algorithms for High-Speed Computations", Dissertation, Department of Electrical Engineering, Stanford University, Stanford, CA, 1974.

[55] M. C. Pease, "Inversion of Matrices by Partitioning", *Ibid.*, vol. 16, pp. 302-314, 1969.

[56] M. O. Rabin, "Probabilistic Algorithms in Finite Fields", *SIAM Journal on Computing*, vol. 9, No. 2, pp. 273-280, May 1980.

[57] J. Reif, "Logarithmic Depth Circuits for Algebraic Functions", *SIAM Journal on Computing*, vol. 15, pp. 231-242, 1986.Extended Abstract in Proceedings 24th Annual IEEE Symposium on the Foundations of Computer Science, Tucson, AZ, pp. 138-145, 1983.

[58] A. H. Sameh and R. P. Brent, "Solving Triangular Systems on a Parallel Computer, *SIAM Journal on Numerical Analysis*, vol. 14, No. 6, 1977.

[59] A. H. Sameh and D. J. Kuck, "Linear System Solvers for Parallel Computers", Department of Computer Science, University of Illinois, Urbana, IL, 1975.

[60] P. A. Samuelson, "A Method for Determining Explicitely the Coefficients of the Characterestic Equation", *Ann. Math. Statist.*, vol. 13, pp. 424-429, 1942.

[61] A. Schönhage, "Schnelle Berechnung von Kettenbruchententwicklungen", *Acta Inform.*, vol. 1, pp. 139-144, 1971.

[62] A. Schönhage and V. Strassen, "Schnelle Multiplikation Grosser Zahlen", *Computing*, vol. 7, pp. 281-292, 1971.

[63] H. S. Stone, "An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations", *Journal of the Association of Computer Machinery*, vol. 20, pp. 27-38, 1973.

[64] V. Strassen, "Vermeiding von Division", *J. Reine Angew. Math.*, vol. 264, pp. 184-202, 1973.

[65] V. Strassen, "The Computational Complexity of Continued Fractions", *SIAM Journal on Computing*, vol. 12, pp. 1-27, 1983.

[66] L. Valiant, "Computing Multivariate Polynomials in Parallel", *Information Processing Letters*, vol. 11, No. 1, pp. 44-45, August 1980.

[67] L. Valiant, S. Skyum, S. Berkowitz, and C. Rackoff, "Fast Parallel Computation of Polynomials Using Few Processors", *SIAM Journal on Computing*, vol. 12, No. 4, pp. 641-644, November 1983.

VITA

Iyad A. Ajwa was born in Balata Camp, Jordan on April 3, 1959, to Mr. and Mrs. Abdel-Rahim and Misa'deh Ajwa. He attended public schools in Jordan and the United Arab Emirates (UAE), graduating from Dubai Secondary School, Dubai, UAE in 1977. He graduated from the University of Jordan, Amman, Jordan in June 1981, receiving a B.Sc. degree in Mathematics. Soon after graduation he began his teaching career as a high school teacher. He taught Mathematics from 1981 to 1983 in Jordan and the UAE. In 1983, Mr. Ajwa joined Lehigh University, Bethlehem, Pennsylvania to do an M.S. in Mathematics which he finished in 1985. Between 1983 and 1987 he worked as a Teaching Assistant in the Department of Mathematics, Lehigh University, Bethlehem, Pennsylvania. His teaching was well received and he was awarded two prizes by Lehigh University in 1986: the "Arthur E. Humphrey Teaching Assistant Award", and the "Teaching Assistant of the Year Prize". In 1987 he joined Northampton County Area Community College, Bethlehem, Pennsylvania as an Adjunct Professor. From January 1988 through June 1990, he was a recepient of a scholarship from the Arab Student Aid International. He is single *but planning to get married in July* 1990. He has three brothers: Yousef, Emad, and Ziad; and two sisters: Raghdah and Rana.