ICES REPORT 88-05

1988

Programming of Supercomputers

by

L. Hayes

The Institute for Computational Engineering and Sciences
The University of Texas at Austin
Austin, Texas 78712

Reference: L. Hayes, "Programming of Supercomputers", ICES REPORT 88-05, The Institute for Computational Engineering and Sciences, The University of Texas at Austin, 1988.

Programming of Supercomputers

Dr. Linda J. Hayes

Department of Aerospace Engineering & Engineering Mechanics

and

The Texas Institute for Computational Mechanics

The University of Texas at Austin
WRW 305
Austin, TX 78712

Acknowledgement: This report is funded in part by the National Science Foundation grant "Research and Development of Iterative Algorithms and Software on Supercomputers for Partial Differential Equations", CCR-8518722.

TABLE OF CONTENTS

NOMENCLATURE

α	overhead time (sec)
b	bandwidth (# words/sec)
g	fraction of asymptotic performance achieved
k	# levels in a pipeline
M	memory unit
N	# of data operands
$N_{\frac{1}{2}}$	half-performance length
p	# of data transfers between Pe's
Pe	processing unit
r	execution rate (MFLOP)
$\bar{r}$	average execution rate (MFLOP)
s	# of clock cycles to reconfigure the pipeline
S	speed up
t	total time (sec)
$\bar{t}$	average time (sec)
τ	clock cycle (sec)
v	fraction of total arithmetic done in parallel
w	# of Pe's

Sub-/Super-scripts

arith	arithmetic
b	break even
∞	asymptotic
comm	communication
ℓ	latch
p	parallel
s	Scaler
total	total
v	vector
w	for w Pe's

1. INTRODUCTION

Computers possessing **pipeline** and **parallel** architectures have been introduced into the engineering community, and their faster speed and larger memories, as compared to conventional mainframe computers, promise to greatly extend the range of problems which can be solved numerically. Using these machines it is now feasible to model large transient nonlinear three-dimensional problems. The demand for these machines has been increasing in the **applications** areas of weather forecasting, structural analysis, petroleum reservoir simulation, aerodynamics, artificial intelligence, fusion energy research, nuclear engineering, VLSI circuit design and many more. In order to take advantage of the potential offered by these machines, one must be aware of the machine architecture and must be willing to adapt computer programs so that they execute efficiently.[1-4]

1.1 Parallelism in Computation.

Parallel processing is a form of processing which takes advantage of concurrent steps in the processing sequence. This might include **pipelining** or overlapping events, **multiple events** taking place concurrently, or a single operation being performed on **multiple data sets** simultaneously. Parallelism arises in computations in many ways:

Overlapping CPU and I/O Operations. The Central Processing Unit (CPU) can perform its operations simultaneously with the input-output (I/O) units. Seperate I/O controllers, channels and processors can be established so that the CPU and I/O operations do not interfere with each other and can run concurrently.

Multiple Functional Units. Multiple, independent functional units may exist in the computer, and they may **execute independently** of each other and **simultaneously.** Typical functional units are floating point add, multiply, divide, square root, fixed point add, increment, Boolean shift and branch. A particular functional unit might be **replicated** in the system. For instance, several add units or multiply units might be available. These units can execute simultaneously on independent data or streams of data may be pipelined through the functional units to create variable-length pipeline configurations.

Pipelining in the CPU. Instructions can be broken into independent units as illustrated in Figure 1. Data streams through the pipeline in a manner very similar to an assembly line. Parallelism is achieved in that at any given time several instructions may be in various phases of operation fetch, decode, operand fetch, arithmetic logic execution and result store simultaneously.

Multi-processing. Multiple independent processors may execute independent portions of a program. Identical processors may execute the same instruction stream on independent data sets, or independent processors may concurrently execute different instruction streams within a single program.

Time-sharing. Multi-programming on a single processor can be achieved by allowing several programs to share the CPU and other resources. This is achieved by assigning **time slices** to individual programs. This concept requires some overhead in the operating system and is usually transparent to the single user.

1.2 History of Computing.

The advances in computing are often divided into **generations** which are determined by computer technology, architecture and languages that are available. The different generations of computers have been occurring roughly on a ten-year time span with some overlap between generations. Computers are often compared in terms of megaflops (**MFLOPS**) which are millions of floating point operations performed per second. The key aspects of the evolution of **computer generations** are given in Table 1, and the evolution of specific **computer systems** is given in Table 2. Table 2 is by no means a complete summary of every computer system which has been built. Rather, it is designed to give examples of the major evolutionary stages of computing.

The First Generation (1938 to 1953). This generation saw the **beginning** of computing. Electromechanical **relays, electronic valves** and tubes were used in the early computers. The devices were connected by insulated wires. On the first machines, binary coded languages were used. In 1950, the first stored computer program was developed, and this marked the beginning of higher-level computer programming. The hardware features were quite primitive and little or no software was available to ease the programming burden. These machines attained the range of 100 flops, and the electronic valves had gate delays of 1 μs .

Second Generation (1952 to 1963). The second generation is marked by the development of **discrete diodes** and **transistor** technology with gate delays of approximately .3 μs . The first transistorized digital computer, TRADIC, was built by Bell Labs in 1954. IBM built the 7090 in 1960, which used discrete germanium transistors. In 1959, Sperry Rand built the LARC

TABLE 1 -- GENERATIONS OF COMPUTER SYSTEMS

First Generation (1938-1953)

- Electronic Valves to Switch Gates
- 1 μs Gate Delay
- Arithmetic Was Done Bit-By-Bit
- Little or No System Software
- Binary Coded Language is Used

Second Generation (1952-1963)

- Discrete Germanium Transistor Technology is Used
- .3 μs Gate Delay
- Higher Level Programming Languages
  - FORTRAN (1956)
  - COBOL (1959)
  - ALGOL (1960)
- Interchangeable Disk Packs (1963)
- Batch Processing: Sequential Execution of Each Job
- Parallel CPU and I/O
- Program Look-Ahead and Memory Fetch
- Error Correction

Third Generation (1962-1972)

- Multilayer Printed Circuits
- Bipolar Planar Integrated Circuits with Several Gates Per Chip
- Gate Delays of 10 ns
- Functional Parallelism
- Independent Memory Banks
- Overlapping CPU and I/O
- Time Sharing
- Pipelined Instruction Stream
- Solid State and Cache Memories
- Intelligent Compilers for High Level Languages
- Multiprocessing or Parallel Processing
- Pipelined or Vector Processing

Fourth Generation (1972-1984)

- Large-Scale Integrated (LSI) Circuits
- High Density Packaging
- Extensions to High-Level Languages for Vector/Parallel Processing
- Virtual Memory
- Time Sharing
- Second Generation of Vector Machines
- Extensive Vareity of Parallel Designs
- Gate Delays of 1 ns

Fifth Generation (1984-

- Very Large-Scale Integrated Circuits (VLSI)
- High Density Modular Design
- Gallium Arsenide Technology
- Multiprocessors
- Giga-flop Execution Rates
- Several Hundred Million Words of Memory

TABLE 2 -- EVOLUTION OF COMPUTERS

Date	Machine	Feature
1938		First Electronic Analog Computer
1946	ENIAC	Electronic Numerical Integrator & Computer
1949	EDSAC 1	100 FLOPS, Stored Program (University of Cambridge)
1951	UNIVAC 1	First Commercial Computer Electronic Valves; Gate Delay = 1 μs
1952	EDVAC	Stored Program (University of Pennsylvania) Bit-by-Bit Arithmetic
1954	TRADIC	First Transitorized Digital Computer (Bell Labs)
1959	LARC	Overlapped I/O and 2 Functional Units (Sperry Rand)
1959	STRETCH	Instruction Look-Ahead & Memory Fetch (IBM) Banked Magnetic-Core Memory; Error Correction
1960	IBM 7090	Discrete Germanium Tarnsistors; Gate Delay = .3μs
1961	ATLAS	Complex Multiprogramming Operating System (University of Manchester) Large, Virtual One-Level Store Interrupt System; Banked Memory for Parallel Access 80,000 Discrete Germanium Transistors Gate Delay = 12 ns
1963	SOLOMON	First Parallel Processor (Westinghouse Corp.) Forerunner of Large-Scale Processors Array of Processors with Local Memory Nearest Neighbor Connections
1964	CDC 6600	First to Use Functional Parallelism 10 Separate Serial Functional Units 32 Independent Memory Banks Instruction Look-Ahead 3 MFLOPS; 100 ns clock
1966	IBM 360/91	Twice as Fast as CDC 6600 2 Execution Units (Floating Point and Scalar) Pipelined Instruction Stream with Look-Ahead
1969	CDC 7600	Ten Times Faster than the CDC 6600 8 Pipelined Function Units 10 MFLOPS; 27.5 ns clock
1970	IBM 360/195	Very Fast; Cache Memory; Comparable to CDC 7600

Date	Machine	Feature
1966	ILLIAC IV	First Commercial Parallel Processing Computer Dept. of Defense & University of Illinois (Burroughs Corp.) Array of 64 Processors with Local Memory Nearest Neighbor Communications 15 Layer Circuit Boards; Semi-Conductor Chips Computer-Aided Layout Design; 50 MFLOPS
1965	STAR-100	Pipelined or Vector Processor (Control Data Corp.) 3 Independent Arithmetic Pipelines Magnetic Core and Semi-Conductor MOS Memories Operates on Contiguous Data Slow Scalar Processor; Long Vector Start Up
1966	TI-ASC	Pipelined or Vector Processor (Texas Instruments) Reconfigurable Pipelined Instruction and Arithmetic Units Slow Scalar Processor Semi-Conductor & Bipolar Semi-Conductor Memories Constant Stride Vectors 4-Processor ASC for Multiprocessing; Shared Memory Extensions to FORTRAN; Automatic Vectorization
1973	CYBER-750	Control Data Corporation; Smaller Size, Reduced Electrical Power Single-bit Correction; Double-bit Detection; ECL Logic; Static RAMS Semi-Conductor Central Memory
1975	AMDAHL 470V/6	First Use of LSI Technology; 100 Circuits per Chip
1976	CRAY-1	Vector Processor; Single Processor 12.5 ns clock; 160 MFLOP
1982	CYBER 205	Vector Processor; 20 ns clocks; 2 processors Single Instruction-Multiple Processor; 400 MFLOPS
1983	CRAY-XMP	Multiple Processor; Vector Processing 9.5 ns clock; 210 MFLOPS Per Processor
1985	CRAY 2	Vector Processor; 4 Processors 4.1 ns clock; 1944 MFLOPS
1986-87	ETA-10	Vector Processor; 8 Processors; 10,000 FLOPS

system which had an independent I/O processor which could be overlapped with one or two processing units. Also in 1959, IBM started the STRETCH project which featured instruction look-ahead and error correction. During this period, FORTRAN (1956), ALGOL (1950) and COBOL (1959) were developed. Interchangeable disk packs appeared at the end of this generation, and batch processing was used with sequential execution of an entire single job.

Third Generation (1962 to 1972). This period saw tremendous gains in computational power. The generation began with the use of small-scale integrated (SSI) circuits and evolved to the use of medium-scale integrated (MSI) circuits. Multi-layered printed circuits were used, and this generation saw core memory replaced by faster, solid-state memories. High-level languages were developed with the advent of intelligent compilers. Time sharing became the standard, and virtual memory was developed during this period. Computer architectures began to take serious advantage of parallelism. Multiple-functional units became common, overlapping CPU and I/O operations were used, program look-ahead and pipelining of both the instruction stream and the data stream came into use.

This period also saw a tremendous increase in the speed attained with traditional architectures. In 1964, the CDC 6600, developed by Seymour Cray, was the first to use functional parallelism. It had ten separate functional units which could operate simultaneously, and a memory with 32 independent banks. In 1969, CDC released the 7600, also developed by Seymour Cray, which was ten times faster than the CDC 6600. During the sixties, these two machines set the standards for high-speed scientific computing. The CDC 7600 had a pipelined functional unit and attained a

speed of 10 MFLOPS. During this period, IBM released its 360 series; the IBM 360/91 was approximately twice as fast as the CDC 6600 and had instruction look-ahead, two instruction units (one for floating point operations, one for integer) and a pipelined instruction stream. The IBM 360-195 was roughly equivalent to the CDC 7600. It had a very fast cache memory. This period began the infancy of supercomputing as we know it today.

During this third generation, the very first multi-processors and vector processors were developed. The SOLOMON computer was the very first parallel processing project from Westinghouse Corporation and was a forerunner of the large-scale multi-processors of today. It was an array of processors with local memory and nearest-neighbor connections. In 1966, based on the success of the SOLOMON project, the Burroughs Corporation, the Department of Defense and the University of Illinois collaborated to develop the ILLIAC IV which was delivered in 1975. The ILLIAC IV was the first commercial parallel processing project. It was also an array of processors with local memory and nearest-neighbor connections. Although the architecture was attractive for finite difference type calculations, it was not usuable for general computing and data communications between processors was slow. In the late sixties and early seventies, the Texas Instruments Advanced Scientific Computer (TI-ASC) was developed, as was the Control Data Corporation's STAR-100. Both were pipelined, vector processors which set many of the standards for vector processors which are used today. The TI-ASC had a 4-processor model which combined both vector and parallel processing. Both machines suffered from slow scalar units.

The Fourth Generation (1972 to 1984). This generation is characterized by the use of **large-scale integrated** (LSI) circuits for both logic and

memory, as well as high-density packaging. Gate delays of 1 ns were achieved. High-level languages were extended to incorporate vector and parallel processing. Time sharing and virtual memories were widespread, high-speed mainframes appeared, as well as high-speed commercially available vector processors such as the CRAY 1, CRAY XMP and CYBER 205. Vector and parallel processing were becoming the standard for large-scale scientific computers, and a massively parallel machine was developed for image processing. Vector machines became commercially viable, and an extensive variety of parallel designs appeared.

The Fifth Generation (1984 to -). This generation is characterized by very large-scale integrated (VLSI) circuits with extremely high density modular design. Gallium arsenide technology may become viable, and there will be a great development in the area of multi-processing. Very high-speed machines with execution rates in the gigaflop range will be available and massive memory systems with several hundreds of millions of words will be available.

1.3 Advances in Materials

The historical evolution of the materials which are used in computers is illustrated in Table 3. In 1947, germanium was used in the first transistor, and in 1958 the first integrated circuit was made by putting several transistors on a single piece of germanium. However, this process did not lend itself to mass production. In 1959, the first integrated circuit of silicon was made. Silicon lends itself to growing a stable oxide film, and the oxide layer is used as a mask allowing the semiconductors to be selectively treated with impurities. This treatment, called doping is a necessary step in the manufacture of integrated circuits.

TABLE 3 -- EVOLUTION OF MATERIALS

1947	Germanium Used in First Transister
1958	First Integrated Circuit: Several Transistors on Germanium
1959	First Integrated Silicon Chip
1960's	Experiments with Various Semi-Conductor Materials
1966	First GaAs Transistor -- Difficult to Dope
1970's	Approach Performance Limit of Silicon-based Integrated Circuits
1974	First Integrated GaAs Circuit -- 4 Times the Speed of Silicon
1984	First Functional GaAs Integrated Circuits at CRAY
1985	Fully Functional 1028-bit GaAs Memory Circuits

Germanium, in contrast, grows an oxide layer that is water soluble and not useful as a mask.

In the 1960's, engineers experimented with many semi-conductor materials. A major development was a compound of galium, arsenic and phosporous which had the correct band gap to emit red light. This led to the development of the **red LED's** found in many types of electronic equipment and spurred the development of new techniques for processing compound semiconductors. In 1966, the first **Gallium Arsenide** (GaAs) transistor was made. It attracted little attention due to the difficulty in doping GaAs. Early GaAs tended to be unstable.

In the 1970's, the performance limits of silicon-based integrated circuits were reached. Computers became smaller with more transistors on each integrated circuit, but the gate delay had changed only by a factor of 2 in ten years. The speed with which electrons moved through silicon was a fundamental barrier to making faster circuits. Research actively ensued to develop **high-speed integrated circuits.** Josephson junctions were developed, but this type of silicon device requires cryogenic temperatures. In 1974, Hewlett Packard produced the first **GaAs integrated circuit,** demonstrating the material's ability to operate at four to five times the speed of silicon. In 1982, CRAY research established a formal GaAs development team which produced its first GaAs integrated circuits in 1984. GaAs offers a **power savings** over silicon, thus reducing heat which allows circuits to be packed more densely. **Denser circuitry** speeds signal propagation by shortening circuit interconnections. GaAs also exhibits greater **resistance** to radiation and temperature variations than does silicon. GaAs successfully operates in radiation levels of 10 to 100 million rads and in temperatures ranging from -200 to 200°C. Two significant areas where GaAs is inferior

to silicon are cost and transistor count capability. In 1985, CRAY research successfuly fabricated fully functional 124 bit GaAs memory circuits which contain approximately 11,000 transistors. A comparison of the properties of GaAs and silicon and silicon-CMOS are given in Table 4.

2 CLASSIFICATION OF COMPUTER ARCHITECTURES

2.1 Flynn's Taxonomy

Historically, parallelism has been used to improve the effectiveness of computers. Parallelism occurs at four levels:

1) Job Level
   - Between Jobs
   - Between Phases of a Job
2) Program Level
   - Between Parts of a Program
   - Within DO LOOPS
3) Instruction Level
   - Between Steps of Instruction Execution
   - Multiple Independent Instructions
4) Arithmetic and Bit Level
   - Between Elements of a Data String
   - Within Arithmetic Logic Circuits

Several taxonomy schemes have been proposed to classify parallel and pipeline architectures. The most standard was proposed by M. J. Flynn [5]. This classification scheme is based on the multiplicity of instruction streams and data streams in the computer system. This scheme will be discussed here. J. E. Shore [6] based his classification on how the computer

TABLE 4 -- COMPARISON OF GaAs , Si(NMOS) and Si(CMOS)

	GaAs	Silicon NMOS	Silicon CMOS/SOS
		Complexity	
Transistor count/chip	20-30K	300-450K	75-175K
Chip area (max.)	40,000 sq. mils.	90,000 sq. mils.	120,000 sq. mils.
		Speed	
Gate delay	50-150 ps	1-3 ns	1-1.25 ns
On-chip memory access	0.5-2.0 ns	10-20 ns	10-20 ns
On-chip/on-package memory access *	4-10 ns	40-80 ns	20-40 ns
Off-chip/off-package memory access *	20-80 ns	100-200 ns	100-200 ns
		IC Design	
Transistors/single gate	1 + fan-in	1 + fan-in	2 × fan-in
Transistors/memory cell			
Static	6	6	6
Dynamic	1	1	--
Fan-in (max.)	5	5	5
Fan-out (max.)	3-5	10-20	10-20
Gate delay increase for each additional fan-out relative to gate delay with fan-out = 1	25-40%	25-40%	10-20%

*Subject to change.

is organized from its constituate parts. Y. T. Feng [7] proposed a scheme based on serial versus parallel processing and W. Händler [8] proposed a classification determined by the degree of parallelism and pipelining in various subsystem levels.

Flynn's taxonomy recognizes four broad categories. Examples of machines in each of these categories are given in Table 5.

(1) SISD -- Single Instruction-Single Data. This organization is typical of the conventional, serial von Neumann computer and is found in most serial computers today. Instructions are executed sequentially but may be overlapped or pipelined in their execution stages. Each arithmetic instruction initiates one arithmetic operation on one data set. An SISD computer may have more than one functional unit. However all functional units are under the supervision of one sequential controller unit.

(2) (SIMD) -- Single Instruction-Multiple Data . This category corresponds to a machine with a single instruction unit which operates on multiple data sets. This classification includes both parallel processors, where a single instruction is issued to multiple function units, each operating on different data, and pipeline processors where each instruction initiates a given operation on multiple elements of a vector.

(3) (MISD) -- Multiple Instruction-Single Data Stream . This classification implies that several instructions are operating simultaneously on a single data set. The output of one processor becomes the input of the next processor in the system. This architecture has received much less attention than the others. Systolic arrays and data flow machines are examples of MISD organization.

(4) (MIMD) -- Multiple Instruction-Multiple Data . Here, multiple processing units operate independently on multiple data sets. This category

TABLE 5 -- CLASSIFICATION OF MACHINE ARCHITECTURE

SISD -- Single Instruction-Single Data

PDP Vax 11/780; FPS AP-120B
IBM 7090; IBM 360/91; IBM 370/168UP
CDC 6600; CDC 7600

SIMD -- Single Instruction-Multiple Data

ILLIAC-IV, PEPE, BSP, MPP
TI-ASC
CDC STAR-100, CYBER-205, ETA-10
CRAY-1, CRAY-XMP
FUJITSU VP-200, FPS-164

MISD -- Multiple Instruction-Single Data

Systolic Arrays
Data Flow Machines

MIMD -- Multiple Instruction-Multiple Data

CRAY-XMP; CRAY-2
Denelcor HEP; Intel Hypercube ZMOB
TRAC; PASM: NYU: CAL TECH: CEDAR

includes all multi-processor configurations where processors are executing independent segments of code. There may be a few high-performance processors or large arrays of microprocessors.

2.2 Memory Systems.

Memory systems must be organized so that the bandwidth or the rate at which data is brought from memory to the processor can match the processor speed and this must be done at a reasonable cost. Supercomputers can be used to solve very large three-dimensional, nonlinear, transient problems because of their very fast execution speed and large memory systems.

Hierarchical Memory

In a hierarchical memory system, several types of memory are used, and they are arranged in levels. Those at the highest level generally are the fastest, the most expensive and are smaller compared to lower-level memory systems. Memories can be classified into three main categories: random-access memory (RAM) where the access time of a memory word is independent of its location; sequential access memory (SAM) in which information is accessed serially or sequentially as in shift register memory with a first-in/first-out buffer; and direct access storage devices (DASD) which are rotational devices made of magnetic material where blocks of information can be accessed directly, usually via special interfaces called channels.

Hierarchical systems composed of two levels have two memories, often called primary memory and secondary memory. Primary memory usually is made of RAM and secondary memory is made of SAM and DASD. Three-level heirarchical systems also often consist of a local high-speed cache memory, as well as the primary and secondary memory systems. Characteristics of

common memory devices are given in Table 6. In designing an n-level hierarchical memory system, one wants to maximize the average memory access speed and minimize the total cost. The overall performance of a hierarchical memory system depends on several factors. These are the **access time** and **memory size** of each memory level, the amount of **information transferred** on each request, and the design of the processor memory **interconnection network.**

In a parallel environment, the machine performance can be severely degraded if several processors request access to the same memory location. When this **memory conflict** occurs, some processors must wait while one of them accesses memory. To avoid this, memories for supercomputers are frequently **banked** or divided into independent units, and data is distributed across these units rather than being in physically contiguous locations. This scheme is called **interleaving** and the interleaving of addresses across M data banks is called **M-Way Interleaving.** There are two basic techniques used for interleaving: **high-order** interleaving distributes data addresses into the M data banks so that the high-order M-bits are used to identify the bank number, and the low order bits in the address refer to a location within a given bank. High-order interleaving is attractive for parallel or **multi-processing** where each processor works on independent data sets. Another advantage to high-order interleaving is that if one memory bank fails, it affects only a local area of the data. However, in practice, most modules share data and there is interaction between parallel processes in a program. In **low-order interleaving,** the lowest order M-bits are used to determine the memory bank allocation, and the higher-order bits define an address within a given memory bank. Low-order interleaving is popular

TABLE 6 -- CHARACTERISTICS OF MEMORY DEVICES IN A MEMORY HIERARCHY*

Level i	Memory Type	Technology	Typical Size s_i	Average Access Time t_i	Unit of Transfer
1	Cache	Bipolar, HMOS, ECL	4K-128K byes	30-100 ns	1 word
2	Main or primary memory	MOS core	16K-1G bytes	0.25-1 μs	2-32 words
3 (optional)	Bulk memory (LCS, ECS)	Core	64K-16M bytes	5-10 μs	2-32 words
4	Fixed head disk or drums	Magnetic	8M-256M bytes	5-15 ms	1K-4K bytes
5	Moveable arm disk	Magnetic	8M-500M bytes	25-75 ms	4K bytes
6	Tape	Magnetic	16G bytes	1-5 s	1K-16K bytes

* Hwang and Brigs [2]

for vector processing because it distributes a given array across all data banks and accommodates high rates of data transmission to/from memory.

Virtual Memory

Virtual memory is a memory management technique used to intelligently allocate memory to users. Virtual memory management gives the programmer the impression that he has unlimited memory available even though the actual physical memory might be relatively small. The compiler translates the program and data structure into modules with unique identifiers. These unique identifiers define the **virtual address space**, and the portion of main memory which is allocated to a given problem defines the **physical memory space**. A dynamic memory management system translates the program-generated virtual addresses to the physical addresses in memory. This process is transparent to the programmer. In a multiple processing system which uses virtual memory, this memory allocation mechanism must be provided for each processor.

There are two common techniques for managing virtual memory space. The first is to partition virtual space into **pages** which are fixed sized blocks, and the other is to partition virtual memory into segments whose size may change dynamically. In a paged memory system, each virtual address consists of a page number and a displacement into that page. Data is brought into physical memory one page at a time. **Page tables** are used as address maps for the virtual memory pages. A pure paged system can be inefficient if the memory management system is poor. If virtual memory is large, the page table may be quite large, and since the size of the tables is fixed, there may be wasted portions of a page or **internal fragmentation**.

In a **segmented memory system**, a set of logically related contiguous memory spaces is called a segment and segments may grow and shrink

dyanmically. This lends itself to programs which are block structured and written in languages such as Pascal, C and ALGOL which have a high degree of modularity. Segmentation efficiently manages the virtual space, whereas paging manages the physical space. Due to its variable size, efficient implementation of segmentation is more difficult than paging. In mapping virtual addresses to physical addresses, the entire segment is moved and an appropriate amount of contiguous physical memory must to be located. In many cases, there are unused fragments or **holes** in physical memory during segmentation. These two concepts have been combined into **paged segments.** There are two implementations of paged segments: one uses **linear segmentation** in which the characteristics of paging dominate, and the other uses **segmented name space** in which the segmentation characteristics dominate. In each case, a segment is divided into pages, and a page table is used to access data within a segment.

2.3 Interconnection Networks

Data can be passed via **circuit switching** where a physical connection is made between a source and a destination or via **packet switching** where a packet of data is routed through a communications network. Circuit switching is preferable for transferring large amounts of data, and packet switching is preferable for transferring lots of small data packets.

Various interconnection networks are used in multiprocessor systems to connect processors, memory modules and I/O devices.

Bus. The most simple interconnection network is to have all functional units connected to a common or shared communications path as shown in Figure 2. This is an extremely simple interconnection device which is totally passive and contains no switching mechanisms. To send a message,

a functional unit must first gain availability of the bus, determine the availability of the target unit to receive a transfer, and initiate the data transfer. The receiving unit recognizes its address as being placed upon the bus, and it waits to receive the data. The bus organization is reliable and inexpensive. However, malfunction of a bus interface circuit can result in a complete system failure and total overall speed is limited by the bandwidth or speed of this single transmission path. An extension of this bus connection is a uni-directional bus in which two uni-directional paths are established between functional units. Multiple bi-directional busses have been developed which permit multiple, simultaneous bus transfers. However, this increases the complexivity of the system. The efficiency of a bus system decreases as more functional units are added to the bus.

Cross-bar Switch. In this sytem, a separate path is provided for each processor and for each memory unit, and a cross-bar switch is used to connect particular processors with specific memory devices. There is a separate bus associated with each memory module and simultaneous transfers can be made to and from each of the memory modules. A cross-bar switch has a very simple switch to functional unit interface. However, the hardware actions within the switch are complex. Each cross point must be capable of switching parallel transmissions, and it must be able to resolve requests to the same memory module during a given memory cycle. The conflicting requests are usually handled on a predetermined priority basis. Cross-bar switching networks can be generalized to much more complex structures, such as Banion networks and Delta networks.

3 VECTOR/PIPELINE PROCESSING

Pipelining or **vector processing** is an example of overlapped parallelism. In a vector processor, the functional unit is subdivided into independent units which can operate independently and concurrently as shown in Figure 22.1. At any given time, the arithmetic unit may be processing different phases of the arithmetic operation on several sets of operands simultaneously, as shown in Figure 3. In a **linear pipeline**, a task is divided into k independent sub- tasks and task j ($1 \leq j \leq k$) cannot begin until all earlier tasks have been completed. The **stages** of the pipeline perform arithmetic or logic operations, and they are separated by high-speed interface **latches**. The latches are fast registers for holding intermediate results between stages. Information flows between stages under the control of a **synchronizing** clock applied to all latches simultaneously. The **clock cycle** of a linear pipe is defined as the maximum time on each stage plus the corresponding delay of the interface latch. The reciprocal of this is called the asymptotic execution rate of the pipeline.

$$\tau = \max(\tau_j) + \tau_\ell \qquad 1 \leq j \leq k \qquad (1)$$

$$r_\infty = \frac{1}{\tau} \qquad (2)$$

On vector processors, there is a certain amount of overhead associated with setting up the pipeline. However, once the pipe has been filled, one resultant will be obtained on each clock cycle independent of the number of stages in the pipe. Ideally, a linear pipeline with k stages can process N-pieces of data in $k + (N - 1)$ clock cycles. Here, k cycles were used to fill up the pipeline and obtain the first result, and $N - 1$ cycles are used to obtain the remaining $N - 1$ results. The time for the total vector operation is

$$t = \left(k + (N - 1)\right)\tau \qquad 3$$

Figure 4 shows the typical timing comparison for a vector processor as compared to a traditional scalar processor. The vertical axis represents the total execution time, the horizontal axis represents the number of operands that are being processed, and α is typical of the vector start-up time. The different slopes of the two lines show the difference in computation rate of the pipeline versus a scalar processor. Figure 5 shows the time per operation versus the number of operands in the vector. This graph is constant for a scalar processor. However, on a pipelined processor, the cost of initializing the pipeline is evident and asymptotic computation rates are achieved as the vector length increases.

3.1 Classification of Pipelining

Pipelining can be achieved at several different levels.

Pipelining the Instruction Stream. The instruction stream can be pipelined, and the execution of several instructions can be done simultaneously with each instruction in various phases of the fetch, decode and operand fetch phase. This is frequently called **instruction look-ahead** and almost all high-performance computers have had this capability since the mid-1960's.

Pipelining the Arithmetic Unit. The arithmetic logic units of the computer can be subdivided and pipelined as illustrated in Figure 1. This is the most common type of pipelining in vector processors. Maximum computation speeds are attained as the vector or data length increases.

Pipelining of Processors. Independent processors can be cascaded or pipelined. The output of one processor immediately flows as input into the next processor in the pipeline. This is the basis of

systolic arrays and data flow machines. This architecture was used on the Denelcor Heterogeneous Element Processor, HEP, whereby the programmer could dynamically define a pipeline of processing units.

The following classification schemes have been proposed by Ramamoorthy and Lee [9].

Unifunction Vs. Multifunction Pipelines. A pipeline unit with a fixed function, such as a floating point add or multiply is called unifunctional. The CRAY machines have been designed with multiple unifunctional pipelines. A multi-functional pipeline can perform more than one operation by interconnecting different stages in the pipeline. The TI-ASC had multi-functional pipelines which were reconfigurable for a variety of arithmetic operations.

Static Vs. Dynamic Pipelines. A static pipeline may assume only one functional configuration at a time, and pipelining is made possible only if instructions are of the same type and are to be executed continuously. A static pipeline may be either unifunctional or multi-functional. A dynamic pipeline permits several functional configurations to exist simultaneously. A dynamic pipeline is a multi-functional pipeline, and it can reconfigure dynamically for various data sets. This requires a much more complex controller unit than does a static pipeline.

Scalar Vs. Vector Pipelines. A scalar pipeline processes a sequence of scalar operations usually under the control of a DO LOOP, and the scalar operands, which will be used for the repeated scalar instructions, are often moved into cache memories or registers so as to quickly supply the scalar pipeline with operands. Vector pipelines are designed to handle streams of data or vectors from banks of registers or from memory and to perform a single operation on each entry in the data stream. Interleaved

memory systems are usually used with pipelined processors in an attempt to match the memory bandwidth which is the average number of words retrieved from memory per second with the demand rate of the processor which is the number of words of data required per second.

3.2 Vector/Pipeline Performance

A supercomputer is characterized by extremely high computation speeds and the availability of massive amounts of main and secondary storage. Supercomputers are primarily designed to perform matrix and vector computations for very large-scale applications. It is very difficult to compare and evaluate various supercomputers, because their performance for a given application is highly dependent on both the machine architecture and the degree to which the programmer has taken advantage of the architecture. Supercomputers are often compared on the basis of the maximum rate at which results can be generated. This is often measured in terms of millions of floating-point operations per second (MFLOPS-Megaflops). Often, these peak performance rates are attained in very special circumstances and will not be sustained in an application.

Peak Performance Rates

Peak performance rates for a variety of machines are given in Table 7. In applications, these maximum execution rates are not attained, because the conditions required to attain these rates cannot be sustained. Each processor in the CYBER family can operate on 64-bit data or can produce two 32-bit floating-point results per clock cycle. A linked triad is a special combination of a scalar and vector operation

$$\underset{\sim}{w} = \underset{\sim}{x} + s\underset{\sim}{y} \qquad 4$$

where $\underset{\sim}{w}$, $\underset{\sim}{x}$ and $\underset{\sim}{y}$ are vectors and s is a scalar scalar. In effect,

TABLE 7 -- MAXIMUM EXECUTION RATES (MFLOPS)

(Millions of Floating-Point Operations/Second)

Machine	Conditions	Max. MFLOP
EDSAC 1	32-bit	0.0001 (100 FLOPS)
CDC 6600	60-bit	3
CDC 7600	60-bit	10
	First Generation Vector Computers	
TI-ASC	32-bit Single Operation	40
STAR-100	2 Units @ 25 MFLOPS 64-bit Single Operation	50
	32-bit Single Operation	100
	Second Generation Vector Computers	
CRAY-1S	12.5 ns Clock 64-bit Single Operation	80
	64-bit Linked Triad	160
CYBER 205	2 Units @ 50 MFLOPS 64-bit Single Operation	100
	64-bit Linked Triad	200
	32-bit Single Operation	200
	32-bit Linked Triad	400
	4 Units @ 50 MFLOPS 32-bit Linked Triad	800
Fujitsu VP-200	7.5 ns Clock	500

TABLE 7 (Continued)

Machine	Conditions	Max. MFLOP
CRAY X-MP-1	1 Processor: 9.5 ns Clock 64-bit Single Operation	105
	64-bit Linked Triad or Overlapped + , *	210
CRAY X-MP-2	2 Processors: 9.5 ns Clock 64-bit Single Operation	210
	64-bit Linked Triad or Overlapped +, *	420
CRAY X-MP-4	4 Processors: 64-bit Single Operation	420
	64-Bit Linked Triad or Overlapped +, *	840
CRAY-2	4 Processors: 4.1 ns Clock 64-bit Single Operation	972
	Simultaneous +, * Units (No linked triad available)	1,944
	Future Generation	
ETA-10	32-bit Linked Triad on 8 Processors	10,000
CRAY-YMP	- - -	- -
CRAY-3	- - -	- -

this allows pipelining between the addition and multiplication units. The CYBER family of machines is memory-to-memory so data is streamed to and from memory. The CRAY machines are register-to-register machines which contain banks of registers for fast vector operations. The CRAY family allows for an extenion of the linked triad operation called chaining. Chaining on a CRAY occurs whenever the results obtained from one pipelined operation can be input directly into the next functional unit. In other words, intermediate vector results reside in the vector registers and do not have to be transported to memory. Because only a fixed number of vector registers are available, in practice there is a limit to the gains which can be achieved through chaining.

The CYBER family of machines executes vector instructions only on contiguous data. The CRAY vector instructions can operate on data which is in constant stride through memory.

Performance Measures

In addition to peak performance rates, several other performance measures have been developed in order to evaluate supercomputer performance. Performance measures will be given for the operation of a single arithmetic instruction on a vector of length N. A simple model for determining machine performance contains two parameters:

(1) The half-performance length, $N_{\frac{1}{2}}$ which is the vector length required to achieve one-half the machine's maximum performance. $N_{\frac{1}{2}}$ is also the number of clock cycles required to reconfigure and fill the pipeline before the first result is obtained. If one had a vector of length $N_{\frac{1}{2}}$, exactly half the execution time would be spent in vector overhead and half in execution.

(2) **Maximum performance rate,** r_∞ , is the maximum number of computations which can be produced per second. In general, this is achieved **asymptotically** for vectors of very long length. The standard measurement unit is millions of floating-point operations per second (MFLOP). r_∞ can be used to measure the machine performance on long vectors, and $N_{1/2}$ gives a measure of performance on short vectors.

The **total time,** t , required to perform a vector operation of length N is

$$t = r_\infty^{-1} (N + N_{1/2}) \qquad 5$$

$N_{1/2}$ can also be derived from timing data supplied by manufacturers. This normally is in the form

$$t = \tau \left(s + k + (N - 1)\right) \qquad 6$$

where s is the number of clock cycles required to reconfigure the pipeline, and k is the number of levels in the pipeline for a given instruction. This gives the result that $N_{1/2}$ is equal to $s + k - 1$ and r_∞ is equal to τ^{-1} . Two other measures of performance can be defined. The **average performance** or the MFLOP rate expected on a vector of length N is given by

$$\bar{r} = N/t = r_\infty / \left(1 + \frac{N_{1/2}}{N}\right) \qquad 7$$

and the **vector efficiency** which is the fraction of the asymptotic performance achieved on a vector of length N ($0 \leq g \leq 1$) is given by

$$g = \frac{\bar{r}}{r_\infty} = \left(1 + \frac{N_{1/2}}{N}\right)^{-1} \qquad 8$$

Most vector computers have a scalar unit with a maximum processing rate of r_∞^s as well as a vector unit with a maximum processing rate of r_∞^v . The vector **break-even** point is the vector length N_b above which the

vector processing unit takes less time than the scalar processing unit. The break-even point is given by

$$N_b = N_{\frac{1}{2}}/(R - 1)$$

$$R = \frac{r_\infty^v}{r_\infty^s} \qquad 9$$

This is illustrated as the point β in Figure 22.4. In general, one desires N_b to be small. This can be achieved either by having a small value of $N_{\frac{1}{2}}$ or a large ratio between vector and scalar processing speeds. The ratio between vector to scalar speed is often large (a factor of 10). The overall performance of an entire program depends on the fraction v of arithmetic which is done in a vector processor as compared to arithmetic done in a scalar processor. The average time per operation is given as

$$\bar{t} = v\bar{t}_v + (1 - v)\bar{t}_s \qquad 10$$

where $\bar{t}_v$ and $\bar{t}_s$ are the average times required by the vector and scalar processing units. The rate of execution, $r = (\bar{t})^{-1}$, is a maximum for 100% vectorization ($v = 1$) and is some fraction of the machine potential when scalar operations are performed. The fraction of asymptotic performance which is achieved for a fixed value of v is given by

$$g = \frac{r}{r_\infty^v} = [R + v(1 - R)]^{-1}$$

$$R = \frac{r_\infty^v}{r_\infty^s} = \frac{t_s}{t_v} = R_\infty g \quad . \qquad 11$$

This data is tabulated in Table 8 which shows the fraction of the machine potential which is achieved for various values of v and R. Figure 6 shows the fraction of asymptotic performance a function of the ratio of scalar to vector speeds (R) and the percent of the arithmetic which vectorizes (v). This indicates that when the ratio of vector speed to scalar speed is large (R large), then a great deal of the computations must vectorize in order to achieve reasonable performance rate. As a measure of the amount of vectorization required, define $v_{1/2}$ as the fraction of arithmetic that must be vectorized in order to gain one-half the maximum machine potential speed.

$$v_{1/2} = (R - 2)/(R - 1) \qquad 12$$

Figure 7 shows the fraction of arithmetic which must vectorize in order order to realize one-half the machine potential speed as a function of the ratio to scalar speeds, R . If R = 10 , approximately ninety percent of the computations must vectorize in order to achieve one-half of the machine's performance rate.

3.3 Vector Language Constraints

In recent years, a tremendous amount of effort has been devoted to developing intelligent compilers which can detect and automatically vectorize sections of code written in a standard high-level language, usually FORTRAN. The advent of artificial intelligence and expert systems may eventually lead to the development of brilliant compilers with a very high level of automatic vectorization. Currently, the development in parallel languages falls into three categories.

TABLE 8 -- RATIO OF SCALAR TO VECTOR SPEEDS

Fraction of Machine Potential Achieved

R = Ratio of Vector Speed to Scalar Speed
V = % of Arithmetic which Vectorizes
G = 1/[R + V91 p R)]

0	1.	2.	5.	8.	10.	15.	20.
.10	1.00000	.50000	.20000	.12500	.10000	.06667	.05000
.20	1.00000	.52632	.21739	.13699	.10989	.07353	.05525
.30	1.00000	.55556	.23810	.15152	.12195	.08197	.06173
.40	1.00000	.62500	.29412	.19231	.15625	.10638	.08065
.50	1.00000	.66667	.33333	.22222	.18182	.12500	.09524
.60	1.00000	.71429	.38462	.26316	.21739	.15152	.11628
.70	1.00000	.76923	.45455	.32258	.27027	.19231	.14915
.80	1.00000	.83333	.55556	.41667	.35714	.26316	.20833
.85	1.00000	.86957	.62500	.48780	.42553	.32258	.25974
.90	1.00000	.90909	.71429	.58824	.52632	.41667	.34483
.92	1.00000	.92593	.75758	.64103	.58140	.47170	.39683
.94	1.00000	.94340	.80645	.70423	.64935	.54348	.46729
.96	1.00000	.96154	.86207	.78125	.73529	.64103	.56818
1.00	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000

Automatic Vectorization

The compiler investigates the data base to determine whether it matches the definition of a vector for a particular machine. If so, an instruction in a DO LOOP can generate a vector instruction. Figure 8 shows a simple case where instructions in a DO LOOP are executed on contiguous data and are independent of other in structions in the DO LOOP, so the instructions can be promoted out of the DO LOOP and replaced with vector instructions. On some machines, a vector is defined as data in contiguous memory locations, while on other machines, a vector may occupy a constant stride in memory. Table 9 lists the criteria for the CYBER 205 compiler to automatically vectorize an instruction within a DO LOOP. Each machine and compiler has a set of rules such as these which govern automatic vectorization. The vectorizing compiler must scan the FORTRAN code for the following items:

1. Data Base Matches Vector Definition
2. No Branching or Decision-Making within a Loop
3. Optimized Use of Vector Registers
4. Detect Pipeline Chaining or Linked Triad
5. Reorder Execution Sequence to Optimize Vector Execution
6. Storage of Temporary Arrays
7. Use of Vector Intrinsic Functions, such as Sine, Square Root, etc.

Extensions to High-Level Languages

Certain extensions which have been implemented into versions of FORTRAN and have been proposed for the standard FORTRAN language. The array and sub-array extensions have been proposed for the FORTRAN 8-X release. These extensions to FORTRAN are given to illustrate the types of

TABLE 9 -- CRITERIA FOR VECTORIZABLE LOOPS IN THE CYBER 205

language extensions which are being considered. At this point, there is not a uniform standard for vector compilers. One must check the specific compiler which is to be used to determine which language extensions have been incorporated and what syntax is required for using them.

(A) Sub-Array. The sub-array notation is used to combine the indices normally found on a DO LOOP with array notation. An * is used to denote that an index will run through its full range of values, incremented by one. For instance

```
      DIMENSION  A(100), B(100), C(100)
      A(*) = B(*) + C(*)
```

is equivalent to

```
      DIMENSION  A(100), B(100), C(100)
      DO  100  I = 1, 100
         A(I) = B(I) + C(I)
100   CONTINUE.
```

Indices normally found in the control of a DO LOOP can be incorporated into the sub-array notation using a : , such as $A(n_1:n_2:n_3)$ where

n_1 = initial value of index

n_2 = terminal value of index

n_3 = increment of index.

The sub-array notation can be combined with multi-dimensioned arrays. The following examples show uses of sub-arrays and the equivalent DO LOOP notation.

EXAMPLE 1

```
DIMENSION  A(10,10), B(10,10), C(10,10)

A(*,*) = B(*,*) + C(*,*)                DO   100  J = 1,10
                              or        DO   100  I = 1,10
                                        A(I,J) = B(I,J) + C(I,J)
                                  100   CONTINUE
```

EXAMPLE 2

```
DIMENSION  X(5,3), Y(2,5)

X(*,3) = Y(2,*)                         DO  100  I = 1,5
                              or        X(I,3) = Y(2,I)
X(1:5,3) = Y(2,1:5)                100  CONTINUE
```

EXAMPLE 3

```
DIMENSION  X(100), Y(100)

X(1:99:2) = Y(2:100:2)                  DO   100  I = 1,99,2
                              or        X(I) = Y(I + 1)
                                  100   CONTINUE
```

Certain other commands have been added to the FORTRAN language that are placed before DO LOOPS which will override decisions made by the automatic vectorizing compiler. These commands typically include

VECTORIZE THIS LOOP AND IGNORE ALL HAZARDS

DO NOT VECTORIZE THIS LOOP

ONLY VECTORIZE THIS LOOP IF THE NUMBER OF ELEMENTS ≥ N

THIS IS A SHORT LOOP (The compiler can often generate very fast code)

(B) **Explicit Vectors.** A vector may be declared explicitly by giving its name, starting address and length. The length is separated from the starting address with a ; . Examples of this are

EXAMPLE 4

```
DIMENSION  A(100), B(50,10)
```

Explicit Vector Notation	Equivalent Fortran
A(1;100)	(A(I), I = 1,100)
B(1,2;100)	(B(I,J) I = 1,50, J = 2,3)

Explicit vectors can be used in arithmetic statements, relational statements and sub-program calls.

EXAMPLE 5

```
DIMENSION  A(100), B(50,10), C(50,5)
A(1;100) = B(1,2;100) + C(1,1;100)
C(25,3;25) = A(10;25)/B(1,1;25)
B(1,1;50) = SQRT(A(1;50))
IF(A(1;100).GT.B(1,1;100))  C(1,1;100) = 0
```

(C) **Descriptor.** A descriptor is a **single-word** variable which identifies an array. It acts as a **software pointer** to a vector and may be used in place of an explicit vector. A portion of the descriptor word points to the **address** of the array and the rest of the word gives the **vector length.** In Example 6, AD, BD and CD point to arrays A, B and C, respectively. Once the descriptor has been assigned, it can be used in place of the explicit vector in arithmetic operations, function calls and logical operations. Descriptors can be assigned for multi-dimensioned arrays and can be reassigned during execution.

EXAMPLE 6

```
DESCRIPTOR  AD, BD, CD
DIMENSION   A(100), B(100), C(100)
READ, N
ASSIGN          AD, A(1;N)
ASSIGN          BD, B(1;N)
ASSIGN          CD, C(1;N)

AD = BD + CD
BD = SQRT(CD)
IF(AD.GT.BD)CD = 0
```

(D) **Special Library Programs.** Many machines offer special system library sub-programs. Vector system libraries usually contain **highly optimized** versions of standard arithmetic operations such as inner products, dot products, min's, max's, summation and saxpy which is

$$A(I) = B(I) + S*C(I)$$

where A, B and C are vectors and S is a scalar.

(E) **Vectorized Versions of Standard Intrinsic Functions.** Vectorized versions of standard intrinsic functions, such as sine, cosine, absolute value, logarithm, square root, etc., are usually available with all vector compilers. These routines normally have been optimized to assure fast vector operations. Either the user explicitly calls these routines or the compiler generates these calls.

(F) **Special Intrinsic Functions.** Often, special, non-standard, vector functions are available to help the programmer take advantage of machine architecture and to manipulate vectors and arrays. Gather and scatter routines are very common. As shown in Figure 9, the gather operation uses a bit vector, which is a vector of zeros and one, to control the packing or gather operation. The gather operation is necessary when the original data set does not conform to the machine definition of a vector. Entries are gathered from the original data set and packed into contiguous memory locations. Vector instructions are performed on the packed data, and, at the completion of these, the scatter operation is used to scatter the data into a resultant vector under the control of a bit vector. Gather/scatter operations are very expensive if they are implemented in software. However, the majority of supercomputers now implements gather/scatter in hardware, and they execute almost as fast as arithmetic computations. It is reasonable to perform gather/scatters when the original data set is sparsely populated.

An option offered on many supercomputers is the use of a control vector which is a vector of zeros and one. Arithmetic operations are performed on a full vector, including entries which one does not want to ultimately store. As shown in Figure 10, the control vector is used to

select the results which are to be stored. Only entries corresponding to a one in the control vector are actually stored. Special library functions are usually provided which allow the user to perform standard arithmetic and logical operations using a control vector. One can also use control vectors to avoid branchings in a DO LOOP so that the loop will vectorize. For instance, suppose there were two separate operations to perform, one for positive entries in a data set and one for negative entries in a data set. One can perform both operations on the data set and use two separate control vectors, one of which will store the results for the positive data and the other of which will be used to select the results corresponding to negative data. Since the two operations are done using vector instructions, this will execute much faster than performing scalar branching operations within a loop. In addition, branching within a loop often prevents the entire loop from vectorizing.

User Defined Subprograms

A user may define vector subroutines and vector functions. A user defined vector subroutine follows the standard conventions because all input and output variables are specified in the calling sequence. The argument list may contain standard array names, explicit vector notation, or descriptors. Example 7 illustrates a subroutine to add two vectors together, $\underset{\sim}{A} = \underset{\sim}{B} + \underset{\sim}{C}$ using array names, explicit vector notation and descriptors in the calling sequence. It is convention to put a V in front of a vector subprogram name.

EXAMPLE 7 VECTOR SUBROUTINE: $\underset{\sim}{A} = \underset{\sim}{B} + \underset{\sim}{C}$

DEFINITION

i) Array Names

```
SUBROUTINE  VADD(A,B,C,N)
DIMENSION   A(N), B(N), C(N)
DO  10    I = 1,N
     A(I) = B(I) + C(I)
10 CONTINUE
RETURN
END
```

ii) Explicit Vectors

```
SUBROUTINE  VADD(A,B,C,N)
DIMENSION   A(N), B(N), C(N)
     A(1;N) = B(1;N) + C(1;N)
RETURN
END
```

iii) Descriptors

```
SUBROUTINE  VADD(AD,BD,CD)
DESCRIPTOR  AD,BD,CD
     AD = BD + CD
RETURN
END
```

CALL

```
DIMENSION   A(100), B(100), C(100)
DESCRIPTOR  AD,BD,CD
READ,       N
ASSIGN      AD,  A(1;N)
ASSIGN      BD,  B(1;N)
ASSIGN      CD,  C(1;N)
```

i) Array Names

```
CALL        VADD(A,B,C,N)
```

ii) and iii) can accommodate either call command

```
CALL        VADD(A(1;N), B(1;N), C(1;N))
```

or

```
CALL        VADD(AD,BD,CD)
```

A user defined vector function must differ from a standard scalar function because the results will be an entire vector, not a single value. The function can be declared as a vector function either using a type declaration for the function

```
VECTOR FUNCTION   VADD(A,B)
DESCRIPTOR  A,B
```

or by explicitly typing the function name

```
FUNCTION    VADD(A,B)
DESCRIPTOR  VADD, A,B
```

Vector functions can be used as part of an arithmetic operation, so temporary space for intermediate vector results must be generated either by the compiler or by the user. Example 8 illustrates both of these options for a function which adds two vectors together. It also illustrates several calls for these routines.

EXAMPLE 8 VECTOR FUNCTION: $\underset{\sim}{B} + \underset{\sim}{C}$

Compiler Defined Temporary Space:	* = User Defined Temporary Space

DEFINITION

```
FUNCTION     VADD(B,C)
DESCRIPTOR   VADD, B,C
VADD = B + C
RETURN
END
```

```
FUNCTION    VADD(B,C;*)
DESCRIPTOR  VADD,B,C
VADD = B + C
RETURN
END
```

CALLS

```
      DESCRIPTOR     AD,BD,CD,DD
      AD = VADD(BD,CD)
```

```
      DESCRIPTOR    AD,BD,CD,DD,ED
      AD = VADD(BD,CD;AD)
```

or

```
      DD = VADD(BD,CD)**2/BD
```

```
      DD = VADD(BD,CD;ED)**2/BD
```

Conversion of Algorithms

If one is to achieve a large fraction of the machine potential of a supercomputer, it is necessary to insure that the numerical algorithm and the implementation of that algorithm are designed to take advantage of the specific machine architecture. There are distinct approaches to conversion of algorithms to supercomputers.

(1) **Implement Existing Code.** A computer program which was used on a serial machine is implemented directly on the supercomputer. Here, the user hopes that the optimizing compiler will do a good job of recognizing potential vector instructions and that the raw speed of the supercomputer will produce a speed-up in execution time. This is the most naive approach to programming a supercomputer and the speed-up which is attainable with this technique is very unpredictable. Certain existing

codes are quite compatible with vector processing and will execute quite well, while others contain apparent recursion and scalar operations and may execute an order of magnitude slower than the machine potential.

(2) **Simple Reordering of DO LOOPS.** In many programs, a tremendous gain in computation speed can be achieved by understanding the guidelines for automatic vectorization. Simple reordering of DO LOOPS and simple code conversion can be done so that the compiler will automatically recognize potential vector instructions.

(3) **Special Coding Techniques.** Additional speed-up can be achieved either by using special highly optimized library functions for operations such as inner products, dot products and matrix vector operations, or by coding critical sections of the program in assembly code to take full advantage of special machine architectural features. This last option is usually beyond the scope of a standard FORTRAN programmer.

(4) **Different Implementation of the Algorithm.** It is possible for additional speed-up to be achieved by rewriting or reexamining the total implementation of an algorithm. In this manner, one can often tailor the data structure and order of computations so as to take full advantage of the machine architecture.

Vectorizing an Algorithm. Computer manufacturers often compare supercomputers on the basis of maximum execution rates (MFLOPS). These rates are often achieved under very special circumstances, such as all processors fully operational with linked triad or chained operations on extremely long data sets. It is simply not reasonable to expect that these special conditions can be sustained in an actual application program. There are certain implementation factors which affect ultimate performance of an algorithm on a particular machine.

- The **data base** should match the intrinsic definition of a vector; either this is data in contiguous memory locations or at a constant stride in memory.
- The **vector lengths** must be long enough to generate efficient vector instructions; $N_{\frac{1}{2}}$ can be used as a guide for determining efficient vector length. Memory-to-memory machines usually require much longer vector lengths for optimum performance than do register-to-register machines. Operations on very **short vectors** can be replaced with scalar codes to avoid the overhead associated with issuing a vector instruction.
- **Gather and scatter commands** can be used to **pack data** into contiguous memory locations. **Control vectors** can be used to issue instructions on contiguous memory locations and to store only selected results.
- **Chaining** and **linked triad** operations can be used to double machine performance. Many vectorizing compilers can detect the occurence of chaining and linked triad operations.
- Many supercomputer memories are divided into independent subsections or **banks**. During one clock cycle, data can be retrieved from each of these banks independently. However, within a bank, once a memory request has been initiated, several clock cycles are required before a second request can be initiated. A **memory bank conflict** occurs and the machine performance can be degraded if one operates on a vector which has a constant stride in its index where the stride is chosen so that each memory request comes from the same memory bank.

The fastest execution will always be in contiguous memory locations.

- On a **register-to-register** machine, such as the CRAY family of machines, one wants to minimize requests from memory and maximize operations which are done on data which resides in very fast vector registers. Often, assembly coding is required to exercise complete control over data-register management. On a register-to-register machine, the vector registers have a fixed length, and the fastest execution speeds will be attained on vectors whose lengths match exactly that of the vector registers. This may be termed the **natural length** of the machine.
- In a **virtual memory** machine, memory management is transparent to the user. However, it can have a significant effect on machine performance. Inefficient use of computer pages and excess paging to and from main memory can result in a significant degredation on the overall performance of a given computer program.
- The choice of a **numerical algorithm** is critical for machine performance. On scalar or sequential machine, total execution time is directly related to the number of floating-point operations performed. This is not true on supercomputers because several operations can be executing simultaneously which resuls in a certain amount of calculations being done "for free." In addition, on a supercomputer, vector instructions may be an order of magnitude faster than scalar instructions. Often, improvements in execution speeds can be

attained if the algorithm is modified to take advantage of a particular architecture. The single most important factor in evaluating how well a given algorithm will vectorize is to recognize the degree of recursion which is inherent to the algorithm. A recursive section of code will not vectorize because each result must wait for the previous result to completely clear the arithmetic unit before it can begin execution. Recursion in an algorithm can be equated with scalar execution.

4. PARALLEL PROCESSING

Parallel processing consists of multiple processing elements (Pe) with multiple memory units (M) which may be executing under the control of a single instruction stream (SIMD) or under control of a multiple instruction machine (MIMD). Computational speeds in a parallel processor are increased by having several functional units operating simultaneously. Parallel processing is a much less advanced state than pipelined processing. No single machine design has emerged as being optimum for scientific computing, and a vast variety of parallel designs is available.

Processing Element (Pe). A variety of Pe's are used in machine design with tremendous differences in their speeds and computing powers. Some supercomputers, such as the CRAY and CYBER families use a small number of very high-powered processors, while other systems, such as the Hypercube and massively parallel processors (MPP), utilize a large number of Pe's with relatively low power. A premise of parallel processing is that low-cost simple Pe's can be replicated many times to achieve high performance and low cost in the computer system. The system must be designed for

detection and recovery from a faulty Pe, so the entire system is not inactive due to a single faulty processor.

Memory Configuration. Parallel systems are configured with global memory which can be accessed by all processors, with local memory which is dedicated to a single processor or with a combination of the two. In a global memory system, data bank conflicts due to requests to a common memory location by multiple Pe's can result in performance degredation. In a local memory system, data must be passed or shared between Pe's. The speed of data transmission is much slower than arithmetic computations, so excessive data handling can also degrade the performance of a parallel system.

Scheduling Jobs on a Parallel System. This is nontrivial and includes multiple jobs running on one system and partitioning a single job to use multiple processors. Deterministic and probabilistic models have been used to address the scheduling problem. However, obtaining an optimal scheduling algorithm for a multi-processor system is computationally intractable [2].

I/O to the Exterior World. This is often a problem in multi-processing systems. Systems with an extremely large number of processors require very high bandwidths for data transmission if each processor is capable of performing I/O operations. I/O speeds are often limiting factors in large multi-processing systems.

4.1 Communication Between Processors and Memory

Many network designs have been proposed for interconnecting Pe's and various combinations of local and shared memory. Typical architectural designs are shown in Figure 11. Communication links between processors are represented as solid lines.

In the **shared memory** design, independent processors are directly connected to a single global shared memory. This has the advantage that all processors have fast access to all data in memory. However, as the number of processors increases, the potential for data bank conflicts becomes great.

The **dance-hall version** connects independent Pe's with independent segments of memory via a switching network. This allows any Pe to access any location in memory. As the number of Pe's and M's increases, the switching network becomes very complicated and the potential for data conflict increases.

In a **binary tree design**, each Pe communicates with the two Pe's at the next lower level in the tree. General tree connections can be implemented. The binary tree design has the advantage that data can be collected from N Pe's in $\log_2(N)$ time. This is very efficient for performing summations on large data sets and for testing the **state** of each of the Pe's in the network.

A **ring design** is a wraparound linear array of Pe's where each Pe is connected to its nearest neighbor in the ring. A generalization of this design is to also connect Pe's along chords in the ring. This corresponds to connecting a specific Pe to neighbors a given increment away on the ring. The disadvantage of a ring design is that many data transmissions are required to pass data from one Pe to another Pe which is distant on the ring.

A **2-D array** of processors is shown with nearest-neighbor connections. This type of arrangement is very good for finite difference grid-generated algorithms. A disadvantage is that many data transmissions are required to pass data to remote locations in the array or to accumulate

data from each of the processors. A **generalized array** of processors can be developed with generalized interconnections between Pe's. This is attractive for finite element calculations. Completely generalized array connections can be developed.

The first-level **hypercube** design is an eight-noded cube with connections along the edges of the cube. A sixteen-node cube is generated by embedding a single cube into an exterior cube. Nodes in corresponding physical positions are then connected. A third-level or thirty-two node cube is generated by replicating a double cube and connecting nodes in similar locations. For problems arising from a grid stencil, a **binary reflected code** can be used to map the physical or computational geometry onto the node geometry of the hypercube [10].

4.2 Performance of Parallel Processors

Ideally, a speed-up of a factor of w is achieved with w Pe's operating in parallel. If w Pe's perform an arithmetic operation on N data sets, the total **arithmetic execution time** is

$$t_{arith} = \tau_{arith}[N/w] \qquad 13$$

where τ_{arith} is the execution speed for a single operation on a single Pe . This assumes that all data is in the local Pe memory and there is no data transmission. In the first clock cycle, results 1 through w are obtained, and in the second clock cycle results $w + 1$ through $2w$ are obtained. The **asymptotic execution rate** is

$$r_{\infty} = w/\tau_{arith} \qquad 14$$

which is a speed-up of a factor of w over a single processor.

Figure 12 shows total execution time versus the number of elements which are processed for w Pe's working in parallel. Typically,

there is an increase in execution time for arrays of length $nw + 1$ and execution time is then constant up to a length of $(n + 1)w$. Figure 13 shows the time per operand versus the number of operands for w Pe's working in parallel. There is an increase in time per operand at lengths of $nw + 1$, however, as the number of operands increase, this jump decreases, and the asymptotic limit of τ/w is attained.

When communication time between Pe's and between a Pe and memory is taken into consideration, the equation for the total time becomes more complicated. In a parallel system, where communication is not overlapped with I/O, the total time is the time for arithmetic computations plus the time for communications. The time required to pass p pieces of data from one Pe to another in a single move can be given by

$$t_{comm} = \left(\alpha_{comm} + \tau_{comm} \frac{p}{b}\right) \qquad 15$$

where α is the start-up time and b is the bandwidth of the data transfer between Pe's. The total time is given by the time for the arithmetic and communications time

$$t_{total} = t_{arith} + t_{comm} \qquad 16$$

$$t_{total} = q\tau_{arith}[N/w] + s\left(\alpha_{comm} + \tau_{comm} \frac{p}{b}\right) \qquad 17$$

where q is the total number of parallel operations performed, and s is the total number of parallel data transfers of length p.

If arithmetic computations are overlapped with I/O, then the total time is the maximum of the two.

$$t_{total} = MAX(t_{arith}, t_{comm}) \qquad 18$$

$$t_{total} = MAX\left(q\ \tau_{arith}[N/w],\ S(\alpha_{comm} + \tau_{comm} \frac{p}{b}\right) \qquad 19$$

Exact timing formulas are very difficult for complicated, overlapped multiprocessing systems.

Speed-up, S , using w processors can be defined as the ratio of the time required on one processor to the time required on w processors. A simple model based on this definition of speed-up and which ignores data transmissions and any overhead for synchronizing processors is given by

$$S = \frac{1}{g} = \frac{t_1}{t_w} = \frac{t_1}{t_s + t_p} \qquad 20$$

$$= \frac{t1}{\frac{t_1}{v\frac{}{w} + (1-v)t_1}} = \frac{1}{\frac{v}{w} + (1-v)}$$

where t_1 is the time required on one processor.

Table i0 gives the speed-up for various numbers of processors as a function of the fraction of arithmetic which can be done in parallel. If all of the arithmetic can be done in parallel, then the speed-up should be equal to the number of processors, w . If 90% of a code can be done in parallel, there is still a degradation in speed-up as the number of processors increases. For instance, a speed-up of less than nine is attained with 64 processors. Therefore, allocation of processors and parallel execution is a critical factor in obtaining high execution rates. A definition for speed-up of an algorithm is the ratio of the best sequential execution time to the time required with a parallel implementation

$$S = \frac{t(\text{Best Sequential Algorithm})}{t\ (\text{Parallel Algorithm})} \qquad 21$$

It is difficult to actually obtain quantitative measurements of this speed-up factor.

TABLE 10 -- FRACTION OF MAXIMUM MACHINE PERFORMANCE WHICH IS ATTAINED

Fraction of Time Multitasked	Number of Processors							
v	w=1	w=2	w=4	w=8	w=16	w=32	w=64	w=infinity
1.00	1.00	2.00	4.00	8.00	16.00	32.00	64.00	Infinity
.99	1.00	1.98	3.88	7.48	13.91	24.43	39.26	100.00
.98	1.00	1.96	3.77	7.02	12.31	19.75	28.32	50.00
.97	1.00	1.94	3.67	6.61	11.03	16.58	22.14	33.00
.96	1.00	1.92	3.57	6.25	10.00	14.29	18.18	25.00
.95	1.00	1.90	3.48	5.93	9.14	12.55	15.42	20.00
.94	1.00	1.89	3.39	5.63	8.42	11.19	13.39	16.67
.93	1.00	1.87	3.31	5.37	7.80	10.09	11.83	14.28
.92	1.00	1.85	3.23	5.13	7.27	9.19	10.60	12.50
.91	1.00	1.83	3.15	4.91	6.81	8.44	9.59	11.11
.90	1.00	1.82	3.08	4.71	6.40	7.80	8.77	10.00
.75	1.00	1.60	2.28	2.91	3.37	3.66	3.82	4.00
.50	1.00	1.33	1.60	1.78	1.88	1.94	1.97	2.00
.25	1.00	1.14	1.23	1.28	1.31	1.32	1.33	1.33
.10	1.00	1.05	1.08	1.09	1.10	1.11	1.11	1.11
.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

In theory, the total execution time of a given job should decrease as more and more Pe's are added to the system. However, in practice, one observes that eventually execution time actually increases as shown in Figure 14. This is due to the fact that the time spent doing arithmetic computations decreases with the number of Pe's which are added, whereas the time required for overhead and to synchronize Pe's increases as the number of Pe's are added. For any given computer architecture and application, there is an optimal number of Pe's which should be used. It is most difficult to quantitatively estimate the optimum number of Pe's.

4.3 Programming and Algorithm Conversion

In order to use a parallel processor efficiently, a suitable algorithm must be chosen for the problem. The degree of parallelism in an algorithm is the number of independent operations that may be performed simultaneously. The architecture of a particular machine is especially important in choosing an algorithm. A procedure or task is a collection of computational steps in a program. If an algorithm contains a number of independent or parallel tasks, it can be implemented on a multiprocessor system. Two major issues arise in converting an algorithm to a multiprocessor environment: Partitioning or identifying the independent parallel tasks in the algorithm; and assigning these tasks to the processors in the system. The architectural features which affect this process include arrangements of processors, processor speed, memory configuration, memory access time, the bandwidth to memory, and the synchronization procedure between processors.

The Pe speed, the interconnection between Pe's, and the speed of data communications greatly affect algorithm choice. Changes in a machine

architecture can have tremendous affects on algorithm implementation. Allocation of data to Pe's is a critical factor in parallel processing because it is possible for communication time to exceed computation time. In converting an algorithm to a multi-processing environment, the specific architecture often dictates the **granularity** of parallelism which one attempts to achieve. **Low-level granularity** is achieved by implementing parallelism at the single statement level. **High-level granularity** is achieved when parallelism occurs at the process, task, subroutine or procedure level. This results in a much larger quantity of instructions being assigned to a given Pe.

Parallel Language Constructs

Creation of efficient parallel compilers is very difficult. This has been approached at two levels: **explicit parallelism** - the programmer specifies the parallelism in the algorithm using language constructs; and **implicit parallelism** - the compiler automatically detects parallelism in the program and generates the appropriate parallel constructions.

Explicit Parallel Language. Adding explicit tasking language to a compiler is easy, and it gives the programmer the opportunity to exploit the multiprocessor architecture. However, in general, programmers find low-level languages **difficult to learn** and **inconvenient to use.** This also results in a code which is **machine specific** and **not transportable.** The user must define the multiple tasks for procedures and explicitly define all data transmissions in the system. In general, this approach does not support user libraries.

High-Level Parallel Languages. These languages are very easy for a programmer to use. There is a minimum of explicit parallel constructs

added to the language, and user libraries are supported. Creation of intelligent parallel compilers, which will do a good job of detecting parallelism in an algorithm, is extremely difficult. This normally results in long and complex compilations and execution performance is very poor if the compiler makes poor judgments.

Language Features. There is no proposed standard for language features which will exploit parallelism, and language features tend to be very machine dependent. However, all machines offer basic constructs. A simple mechanism for denoting parallelism is with the **fork** and **join** commands. A fork command either spawns a new task or activates more than one processor. A join command is used to synchronize processors, and all processors wait for the previously created tasks to terminate. This is also called a **barrier**. The fork command may spawn copies of a given task or it may spawn independent tasks. **Send** and **receive** commands can be used to pass data to a target Pe and to receive from another Pe.

Scheduling Parallel Tasks

Once **tasks** have been identified or assigned in an algorithm, the tasks and corresponding data must be allocated to Pe's in the system. **Pre-scheduling** is a static task allocation. Tasks and corresponding data are pre-assigned on a global basis to the Pe network. Tasks for static allocation are usually large increments of work, and the algorithm is partitioned with a high level of granularity. **Grid-generated algorithms** lend themselves to this type of scheduling. However, many applications do not. Prescheduling requires a synchronization controller and can result in the work being **unevenly distributed** in the Pe network with many processors remaining idle. **Self-scheduling** is a dynamic task allocation. Tasks are

identified and are assigned to a Pe as the Pe becomes available. These dynamic tasks are often small units of work, and the parallel partitioning is on a low-level granularity. Self-scheduling results in good load balancing with useful work being done in all processors. Data flags can be used for algorithm synchronization.

The factors which degrade parallel performance are

- Lack of Parallelism in the Algorithm
- Communication Time Between Pe's
- Memory Access Time
- Inadequate Bandwidth to Memory
- Memory Conflict Delays
- Synchronization Overhead
- Idle Processors .

4.4 Data Flow and Systolic Arrays

In standard processing units, the flow or control of computations is determined by a fixed stack of instructions which are processed sequentially. Branching statements are used to branch to a different point in the instruction stack and control again proceeds sequentially. Parallelism is achieved by assigning separate sequential instruction stacks to various processors. A given Pe may be a pipeline or vector processor, but the actual order of instructions which is executed on the Pe is determined in a sequential manner. By contrast, in a data flow machine, processors are fired or activated as soon as all of the required data is made available to them. This results in an asynchronous computing system which is controlled by the flow of data from one Pe to another. Systolic arrays have been generated using the data flow concept. Many low-cost very simple

Pe's are interconnected in an attempt to increase the performance-to-cost ratio of the system. VLSI technology can be used to create complex special purpose systolic arrays with a large number of processors on a single chip. Systolic arrays are often used as special purpose attached processors.

EXAMPLE 9 - Summation of Data with Systolic Arrays

A binary tree of systolic array processors, each of which receives two pieces of data, adds them together and passes the sum on, can be used to accumulate the sum of N numbers in $\log_2(N)$ time, as shown in Figure 15. The arrival of the first data set to the level-one processors triggers their operation. They calculate the sum of their two operands and pass that to level two, at which time the next data set enters level one which again triggers its action. The data sets control the pulse or operation of the systolic array system. At the top level, the summation of the N operands will appear. After the first summation appears, subsequent summations will be attained on every pulse of the system. No global synchronization is required.

For convenience, systolic arrays are often connected in regular patterns; however, irregular patterns are useful. Consider the sequence of operations

```
1.  C = A + B
2.  P = A + C
3.  S = C/B
4.  R = P + C
5.  T = R * C
6.  U = A/R
```

which can be implemented in a standard machine in several orders. For instance, statements 2 and 3 can be reversed without affecting the results.

Figure 16 shows an irregular systolic array which can be used to perform the same computations. This data flow graph is generated by analyzing dependencies without the calulation. Each Pe fires as soon as its two input data are availble.

5 Programming Techniques

In a parallel or vector environment, the performance of an algorithm is inversely proportional to the total CPU time. On parallel and vector computers, minimum execution time is not synonomous with performing the minimum number of arithmetic operations as it is on a serial processor. This is because extra computation can be attained at little or no increased cost, and there are new overhead costs associated with starting up vector instructions or processors and with communication and synchronization of parallel processors.

5.1 Guidelines

The overall performance depends very much on the algorithm, the implementation and the machine architecture. The programmer should exploit pipelining, parallel operations, and chaining, and should avoid data movement and recursive or scalar operations. Table 11 lists degregation factors which a programmer should consider. A programmer can often change the order of computations to increase parallelism and minimize data movement in an algorithm. Storage patterns can be changed, vector instructions lengthened, and the number of gather/scatter operations can be reduced. Sub-programs that contain short segments of code should be written in line so that the compiler can vectorize this. One should unroll short DO LOOPS

TABLE 11 -- DEGRADATION FACTORS

Vector	Parallel
Recursion	Lack of Parallelism
Scalar Operations	Synchronization of Pe's
Short Vector Instructions	Data Movement Between Pe's
Gather/Scatter to Rearrange Data	Communication to Shared Memory

to lengthen vectorization as shown in Example 10. The order of DO LOOPS can be reversed to make the order of arithmetic operations compatible with the data definition of a vector as shown in Example 11. Increases in vectorization may be obtained by reversing the order of DO LOOPS and inverting the order of their execution as shown in Example 12. The original problem does not vectorize on a system which requires vectors to be in contiguous memory locations. On a constant stride vector instruction, the performance will not be optimal. Reversal of the order of loops results in vectorization. However, storage must be allocated for a temporary vector. Reversing the order of the loops and inverting the order of one loop produces vectorization without introducing temporary storage requirements. Reversing the order of the loops and inverting both loops can greatly increase the vector length for some applications. For the third and fourth cases, the data would have to be stored in a reverse order memory.

EXAMPLE 22.10 -- UNROLLING DO LOOPS:

```
      Dimension  X(200,2), R(200)

Original Loop - Vector Length of 2

      DO 100 I = 1,N
         R(I) = 0
      DO 100  J = 1,2
         R(I) = R(I) + X(I,J)**2
100   CONTINUE

Unrolled Loop - Vector Length of  N

      DO  100  I = 1,N
         R(I) = X(I,1)**2 + X(I,2)**2
100   CONTINUE
```

EXAMPLE 11 -- REVERSE ORDER OF DO LOOPS

```
      Dimension  A(100,100), B(100,100)
```

Original Loop - May Not Vectorize
Does Not Access Contiguous Memory

```
      DO  100  I = 1,N
      DO  100  J = 1,N
        B(I,J) = 2.*A(I,J)
100   CONTINUE
```

Reverse Order of Loops - Vectorizes - Accesses Contiguous Memory

```
      DO  100  J = 1,N
      DO  100  I = 1,N
        B(I,J) = 2.*A(I,J)
100   CONTINUE
```

EXAMPLE 12 -- REVERSING AND INVERTING DO LOOPS

1. Original Problem - Does Not Access Contiguous Memory Locations

```
      DIMENSION A(25,25)
      DO 10  I = 1,20
      DO 10  J = 1,20
         A(I,J+1) = A(I+1,J)
 10   CONTINUE
```

2. Reversal of Order of Loops

Reversal of loops introduces fault. This can be circumvented by introduction of temporary vector.

```
      DO 10   J = 1,20                DO 20     J = 1,20
      DO 10   I = 1,20                DO 20     I = 1,20
         T(I,J) = A(I+1,J)               A(I,J+1) = T(I,J)
10    CONTINUE                    20    CONTINUE
```

3. Loops Reversed and One Inverted

This removes the fault without introducing temporary vector.

```
      DO 10  J = 1,20
      DO 10  I = 1,20
         A(I,22-J) = A(I+1, 21-J)
   10 CONTINUE
```

4. Loops Reversed and Both Inverted

This has properties of solution #3 but makes better use of memory.

```
      DO 10  J = 1,20
      DO 10  I = 1,20
         A(21-I, 22-J) = A(22-I, 21-J)
   10 CONTINUE
```

Divide-and-conquer schemes are often useful to break computations up into smaller pieces which can be done in parallel. Consider calculating the summation of $N = 2^n$ numbers. The data can be broken up into two pieces, and corresponding entries into the first data set and the second data set are added together using the vector instruction of length 2^{n-1}.

$$Sum_1 = a_i + a_{2^{n-1}+i} \qquad 1 \leq i \leq 2^{n-1} \qquad 23$$

These entries are then broken into two pieces and corresponding entries are summed using a vector instruction of length 2^{n-2}. This proceeds until finally the total summation is obtained. On a parallel processor, with w processors, the data entries can be divided into w pieces, and each processor performs a summation on its small data set. Then the data can be transferred and accumulated using a divide-and-conquer scheme. A reduced number of processors are active at each step of the accumulation process.

For grid-based algorithms, the allocation of computation or division of labor to individual Pe's may be algorithm dependent. For instance, on a rectangular grid one could proportion the work between four processors as shown in Figure 17. The row-wise allocation is very good for fast Fourier transforms and is not good for solving tridiagonal systems. The column-wise division is good for solving tridiagonal systems of equations and is poor for fast Fourier transforms. The block division represents a compromise between row-wise and column-wise division [11,12].

Stacking independent computations can often increase vectorization or parallelization. When many repetitive independent computations must be performed, they can either be assigned to independent Pe's or can be stacked as shown in Figure 18. Vector instructions can be generated across the data sets. The example shown in Figure 18 is the solution of many tridiagonal systems of equations. The straightforward solution technique is recursive and does not vectorize. However, by stacking the tridiagonal systems which must be solved, the steps in the algorithm generate the vector instructions and the length is the number of systems to be solved.

5.2 Vector and Parallel Algorithms

Tremendous progress is being made in developing vector and parallel algorithms. An indication of these will be given in this section and many examples can be found in the references [13-26].

Summation of a Series

Series of numbers can be summed using the previous divide-and-conquer technique, or a binary tree connection, as shown in Figure 22.15 can be used very efficiently to perform the summation. The binary tree network is particularly attractive if multiple summations are to be

performed. In summing many data sets which contain $N = 2^n$ entries, there is a start-up overhead of n clock cycles while the first data set propagates through the tree structure. After that, summations are attained on every clock cycle. For example, 10^6 is approximately 2^{20}. Once the initial overhead of 20 clock cycles has been done, summations of different sets of 10^6 numbers can be obtained on every clock cycle.

Matrix-Vector Multiplication

Multiplying a matrix times a vector is a good example where reordering data and arithmetic operations can increase vectorization. Consider multiplying

$$\underset{\sim}{b} = \underset{\sim}{\underset{\sim}{A}}\ \underset{\sim}{x} \qquad 24$$

The i^{th} entry of the result $\underset{\sim}{b}$ is given by

$$b_i = \sum_{j=1}^{N} A_{ij}\, x_j \qquad 25$$

and the standard implementation is a **row-oriented or inner product** form, with the FORTRAN code given by

```
      DO  100   I = 1,N
            B(I) = 0
      DO  100   J = 1,N
            B(I) = B(I) + A(I,J)*X(J)
100   CONTINUE
```

These calculations are done across a row of the matrix which is not in contiguous memory locations. The loops can be reversed to give a column oriented or outer product form.

```
      DO  100   J = 1,N
                B(I) = 0
      DO  100   I = 1,N
                B(I) = B(I) + A(I,J)*B(J)
100   CONTINUE
```

This corresponds to multiplying the i-th entry of the x vector by the entire i-th column of the matrix $\underset{\sim}{A}$ and then summing the products. This can be implemented using vector instructions of length N .

A diagonal product form is very attractive for operations with banded matrices. The diagonals are stored sequentially in memory and the diagonals are multiplied by the corresponding entries in the x vector. Pointers can be used to initiate the vector instruction at the correct location in the x vector. This technique can be used very efficiently to multiply a small dense matrix times a vector as shown in Figure 19. The individual entries, a, b, c and d, can be appended and calculated with one vector instruction. The summation can then be performed using a divide-and-conquer technique. This strategy, together with stacking, has been used for a matrix/vector multiply coming from finite element applications [26].

Iterative Methods

Supercomputers will be required to solve very large three-dimensional, nonlinear transient physical problems. At some point in the solution algorithm large linear systems of equations must be solved and iterative methods will be employed to solve them. The reader is referred to references [27,28] for a discussion of iterative methods. Iterative techniques are either parallel like a Jacobi-based method or recursive like a

Gauss-Seidel or SOR method. The effect of ordering of the nodes in a finite difference or finite-element grid can greatly increase or decrease the parallelism in the algorithm. For example, consider solving Poisson's equation on a unit square

$$-\Delta u = 1 \qquad 26$$

with zero as the boundary condition using a standard 5-point finite difference method. The natural ordering of the nodes and the structure of the resulting matrix problem are shown in Figure 20. The point-Jacobi method vectorizes for this matrix structure, and the algorithm can be accelerated using conjugate gradient techniques. However, the convergence rate is less than could be achieved using block iterative methods. A common block technique is to group points in a given row together. This division of the matrix is indicated by the solid lines in Figure 20. This results in several systems of tridiagonal equations which must be solved. The solution of tridiagonal equations is recursive, so this is not attractive for vector computation unless there are enough rows to make stacking reasonable. The nodes can be renumbered using a **red/black ordering** [27,28] as shown in Figure 21. If the red nodes and the black nodes are each taken as a block, the resulting systems of equations which must be solved are diagonal, which vectorizes very well. A **diagonal ordering** can be used to vectorize the Gauss-Seidel and SOR type algorithms. The nodes in the grid are numbered along diagonals. The computation for nodes along a diagonal involves the solution of diagonal matrices as shown in Figure 22, and this vectorizes. However, since the length of the diagonals varies, the vector instructions are of varying lengths and are very short in the lower and upper corners. A **diagonal red/black ordering** can be used as a variation of the previous red/black ordering as shown in

Figure 23. This ordering results in diagonal systems of equations being solved for the nodes in a given group.

Multiple iterations can be done simultaneously for Gauss-Seidel and SOR type algorithms using the diagonal node numbering. Iteration one is done for the first diagonal. These results are passed to the second diagonal. At this point, node 1 can begin computation of its second iterate as shown in Figure .24. In this manner, iterations proceed in **waves** across the grid. In practice, the question of convergence testing must be resolved for this type of scheme. One could test all of the nodes irregardless of the iteration number and stop when all nodes have reached the convergence criteria.

Asynchronous iterations have been suggested for multiprocessors. Here, nodes are assigned to individual Pe's and the Pe's begin iterating. When one Pe requests the current iterant from a neighboring Pe, data is passed with no iteration number. This data is used irregardless of its iteration number. In this manner, all Pe's perform the iteration process with no central controller locking them into step from iteration to iteration. This type of iterative procedure will converge as long as each node is updated at some time during the iterative procedure [29]. However, convergence rates have not been obtained for this type of technique.

Multicolorings are generalized red/black techniques where nodes are partitioned into more than two color groups. This is useful in extending the red/black techniques to finite-element grids and to irregular grid structures [30,31].

Tridiagonal Solvers

Solving tridiagonal systems of equations is a very important application in linear algebra. These equations arise in finite-difference

approximations to diffusion operators and in alternating-direction techniques. The standard procedure is to decompose the tridiagonal matrix into a lower and an upper diagonal matrix. Then, using forward elimination and back substitution, the solution vector can be obtained for a given right-hand side. The decomposition steps, as well as the forward elimination and back substitution, are recursive. Several techniques have been proposed to introduce parallelism into this algorithm. If multiple right-hand sides are used, the corresponding solutions can be calculated in parallel in separate Pe's, or the systems can be stacked and solved using vector instructions as shown in Figure 18. Recursive doubling can be used to introduce parallelism [32]. Here, cyclic reduction is used on all linear, first-order (two-term) recurrence relationships. A transformation is used on all linear second-order (three-term) recurrences, and cyclic reduction is performed on the transformed variables. The total algorithm requires order log(N) operations with parallelism N . Cyclic reduction or substructuring can also be used to solve tridiagonal equations [35,36]. In substructuring, the large tridiagonal matrix is divided into smaller pieces as shown in Figure 25. The elements marked with an X in Figure 25 which connect the diagonal blocks are eliminated which leads to the block structure after elimination shown. These independent block systems can either be assigned to individual processors or can be stacked and solved using vector instructions.

Differential Equation Solvers

In an ODE solver, explicit methods lead to Jacobi-like execution which can vectorize very well. Implicit time-marching techniques, which are more stable, are Gauss-Seidel like or recursive, and one must detect

parallelism within a time step. Within a time step, often the values at nodes are updated in a manner which can be vectorized. On a parallel machine, entries corresponding either to a single nodes or groups of nodes can be assigned to individual Pe's . The time history of these nodes is contained in that particular Pe , and this determines the communication patterns between Pe's. Runge-Kutta techniques vectorize very well because the new calculation at each time level is based on linear combinations of previous calculations which can be done in parallel. Nonlinear or implicit calculations, where new data is obtained by

$$Y^{n+1} = Y^n + f(Y^{n+1}) \qquad .27$$

can be parallelized if one wishes to introduce parallelism by solving for several sets of initial data.

Partial differential equations fall into three distinct categories. For parallel processing, the assignment of nodes to Pe's is usually done using a grid-oriented scheme.

Elliptic Equations. These result in systems of linear equations to be solved which can be done using iterative techniques or direct techniques as discussed earlier.

Hyperbolic Equations. Explicit marching techniques are usually used for hyperbolic equations, and these parallelize very well.

Parabolic Equations. Explicit techniques arise using finite-difference methods, and the time marching can be done very efficiently on both vector and parallel machines. Finite-element methods and implicit finite-difference methods result in linear systems of equations to be solved at each time step. This can be done using iterative methods or direct methods as discussed previously.

6 Discussion

Parallel and pipelined computer architectures are becoming widely available and are very attractive for large-scale computations. Advances in technology, as well as new concepts in architecture design, have offered orders-of-magnitude increase in computation speeds and available storage. However, to gain full advantage of the new architectures, the programmer must be much more aware of the architecture than was required in the past. These new computer designs will lead to the development of software which is tailor-made for a specific machine architecture. Algorithms can no longer be evaluated simply by the number of arithmetic operations. Vector or pipelined architectures follow fairly straightforward sets of rules for optimization so program evaluation is rather straightforward. Evaluating machine and algorithm performance is often quite difficult in a parallel environment. The programmer must not only design a computer code to take advantage of the computer architecture, but it must also efficiently transfer data between processors to avoid large communication delays. In the parallel arena, there are vast numbers of architectures being proposed. At this time, it is not clear which architecture will dominate future computing. Both machine and software development are very much in a state of evolution.

REFERENCES

1. R. W. Hockney and C. R. Jesshope, Parallel Computers, Adam Hilger, Ltd., 1981.

2. K. Hwang and F. A. Briggs, Computer Architectures and Parallel Computations, McGraw-Hill, 1984.

3. J. M. Ortega and R. G. Voight, "Solution of Parallel Differential Equations," ICASE Report 85-1, NASA Langley Research Center, Hampton Va, January 1985.

4. G. Rodrigue Parallel Computations, Academic Press, 1982.

5. M. J. Flynn "Some Computer Organizations and Their Effectiveness," IEEE Trans. Comput., Vol. C-21, pp. 948-960, 1972.

6. J. E. Shore, "Second Thoughts on Parallel Processing," Comput. Elect. Eng., Vol. 1, pp. 95-109, 1973.

7. Y. T. Feng, "Some Characteristics of Associative/Parallel Processing," Proc. 1972 Segamore Comp. Conf., Syracuse Univ., pp. 5-16, 1972.

8. W. Handler, "The Impact of Classification Schemes on Computer Architecture," Proc. 1977 Int'l. Conf. on Parallel Proc., pp. 7-15, 1977.

9. C. V. Ramamoorthy and H. F. Li, "Pipeline Architecture," ACM Computing Surveys, Vol. 9, No. 1, pp. 378-395, 1977.

10. T. F. Chan and Y. Saad, "Multigrid Algorithms on the Hypercube Multiprocessor," Yale Research Report, YALEU/DCS/RR-368, February 1985.

11. Y. Saad and M. Schultz, "Parallel Direct Methods for Solving Banded Linear Systems," Research Report YALEU/DCS/RR-387, Yale University, August 1985.

12. S. L. Johnsson, Y. Saad and M. Schultz, "Alternating-Direction Methods on Multiprocessors," Research Report YALEU/DCS/RR-382, Yale University, October 1982.

13. W. Gentzsch, Vectorization of Computer Programs with Applications to Computational Fluid Dynamics, Vieweg, 1984.

14. R. Numerische (Ed.), Proceedings: Supercomputer Applications Symposium, Plenum Press, 1985.

15. T. L. Jodan, "A Guide to Parallel Computation and Some CRAY-1 Experience," in Parallel Computations, Edited by G. Rodrigue, Academic Press, pp. 1-50, 1982.

16. J. I. Karush, N. K. Madsen and G. H. Rodrigue, "Matrix Multiplication by Diagonals on Vector/Parallel Processors," Report UCID 16899, LLNL, Livermore CA, 1975.

17. K. W. Fond and T. L. Jodan, "Some Linear Algebraic Algorithms and Their Performance on CRAY-1," Report LA-6774, LANL, Los Alamos NM, 1977.

18. D. A. Calahan, W. G. Ames and E. J. Sesek, "A Collection of Equation Solving Codes for the CRAY-1," Report SEL 133, University of Michigan, Ann Arbor, 1979.

19. A. Greenbaum and G. Rodrigue, "The Incomplete Cholesky Conjugate Gradient Method for the STAR (5-point Operator)," Report UCID-17574, LANL, Livermore CA, 1977.

20. T. L. Jordan, "A Performance Evaluation of Linear Algebra Software in Parallel Architectures," Report LA-8078-MS, LANL, Los Alamos NM, 1979.

21. P. N. Swarztrauber, "Vectorizing the FFTs, in Parallel Computations, Edited by G. Rodrigue, Academic Press, pp. 51-83, 1982.

22. Proceedings, LASL Workshop on Vector Parallel Processors, Report LA-74910, Los Alamos National Laboratory, Los Alamos NM, 1978.

23. Proceedings, NSF Workshop on Supercomputer Usage in Mechanics and Transport Phenomena, NCAR, Boulder CO, 1983.

24. Proceedings, NSF/NASA Workshop on Parallel Computations in Heat Transfer and Fluid Flows, College Park MD, 1984.

25. J. J. Dongarra, "Performance of Various Computers Using Standard Linear Equation Software in a FORTRAN Environment," Argonne Tech. Memor. No. 23, Argone Nat'l. Laboratory, August 1984.

26. T. C. Oppe and D. R. Kincaid, "A Comparison Study of Iterative Solution Methods for Sample Oil Reservoir Simulation Problems on Vector Computers," Center for Numerical Analysis Report, CNA-200, University of Texas at Austin, August 1985.

27. L. J. Hayes and P. Devloo, "A Fast Vectorized Matrix-Vector Multiply," Intl. J. Numer. Meth. Eng., to appear, 1986.

28. D. M. Young, Iterative Solution of Large Linear Systems, Academic Press, New York, 1971.

29. A. M. Ostrowski, "On the Linear Iteration Procedures for Symmetric Matrices," National Bureau of Standards Report No. 1844, p. 23, August 1952.

30. L. M. Adams, "An M-Step Preconditioned Conjugate Gradient Method for Parallel Computation," Proc. 1983 Intl. Conf. in Parallel Processing, Bellaire, MI, p. 36-43, August 1983.

31. L. M. Adams and H. F. Jordan, "Is SOR Color Blind?' SI SSC, to appear, 1985.

32. H. S. Stone, "Parallel Tridiagonal Solvers," Assoc. Comput. Math., Vol. 20, pp. 27-38, 1975.

33. R. W. Hockney, "A Fast Direct Solution of Poisson's Equation Using Fourier Analysis," J. Assoc. Comp. Mach., Vol. 12, pp. 95-113, 1965.

34. P. N. Swarztrauber, "A Direct Method for the Discrete Solution of Separable Elliptic Equations," SIAM J. Num. Anal., Vol. 11, pp. 1136-1150, 1974.

35. R. A. Sweet, "A Generalized Cyclic-Reduction Algorithm," SIAM J. Num. Anal., Vol. 11, pp. 506-520, 1974.

36. R. A. Sweet, "A Cyclic-Reduction Algorithm for Solving Block Tridiagonal Systems of Arbitrary Dimensions," SIAM J. Numer. Anal., Vol. 14, pp. 706-719, 1977.

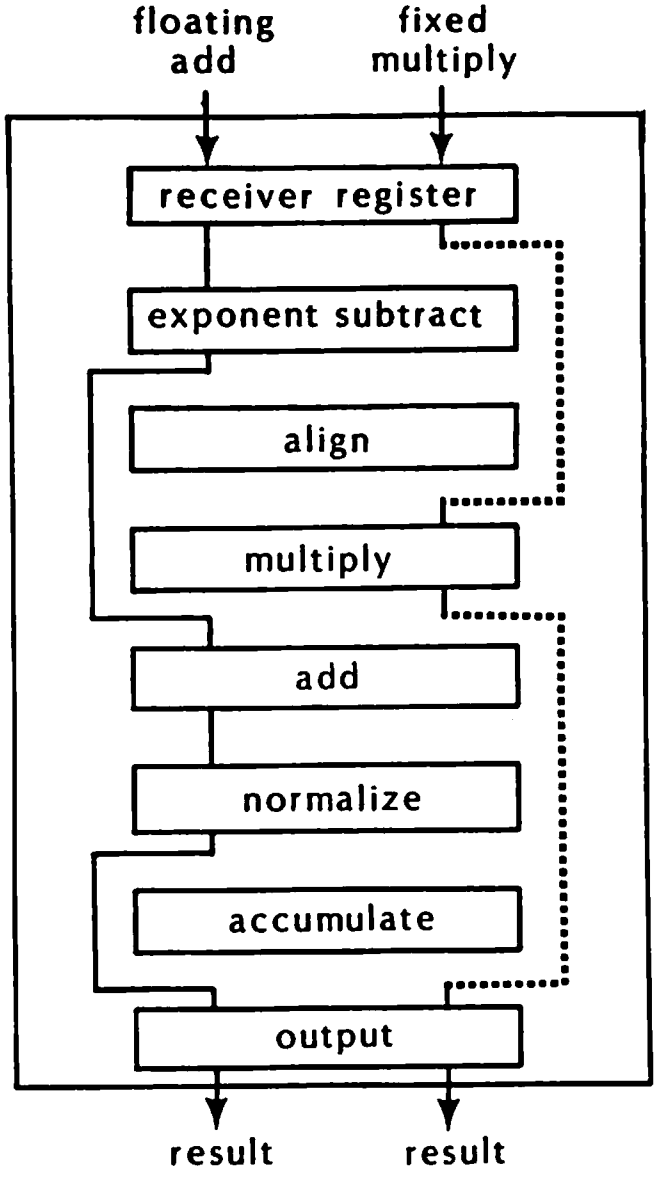


Figure 1 Pipelining the CPU

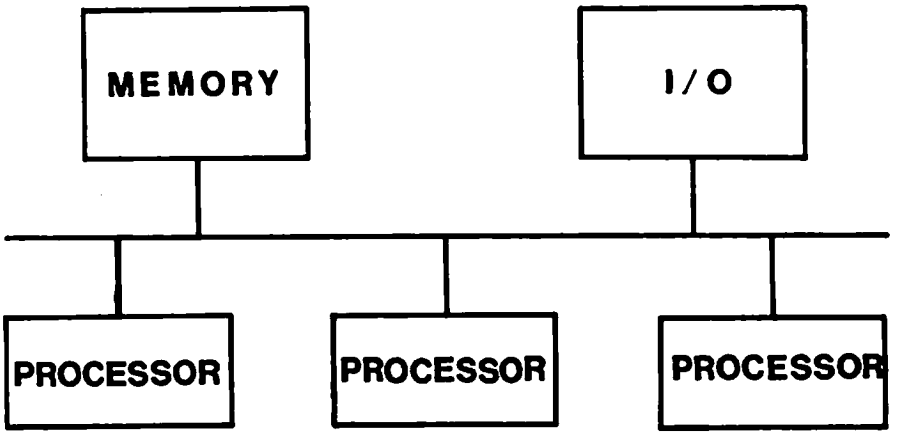
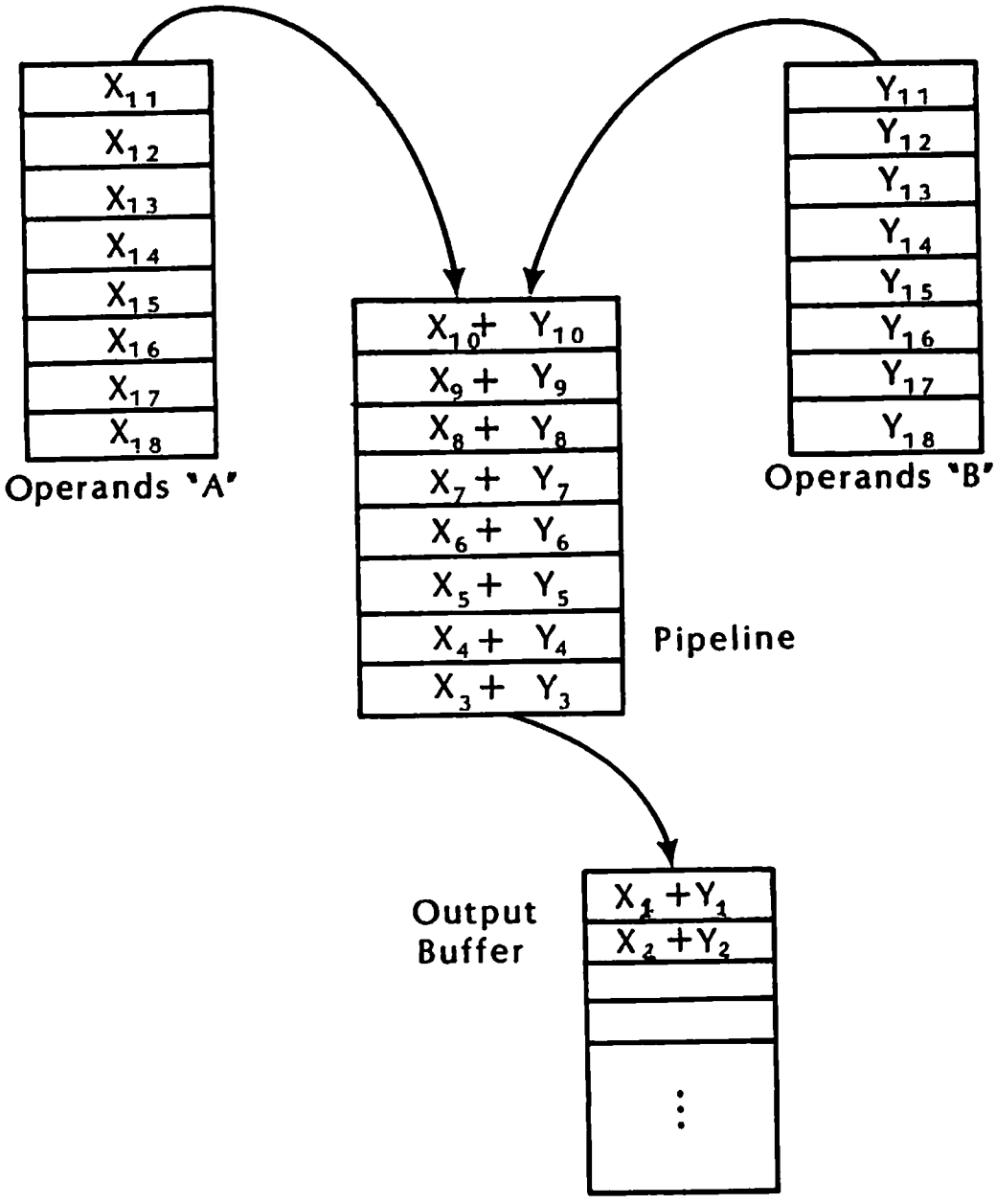


Figure 2 Shared Bus Interconnection



VECTOR INSTRUCTION

$Z = X + Y$

Figure 3 Data Flow in a Pipeline

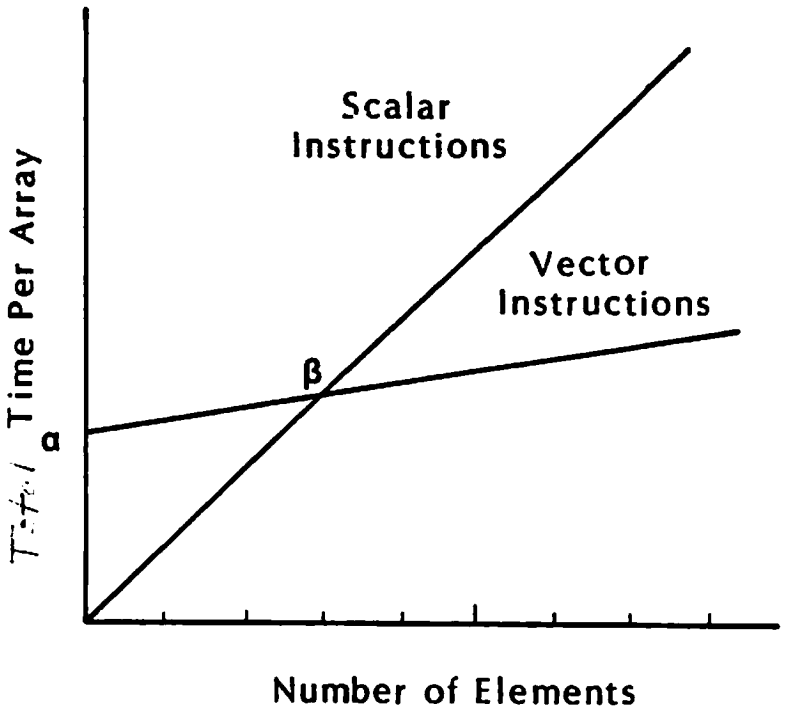


Figure 4 Total Time Versus Vector Length

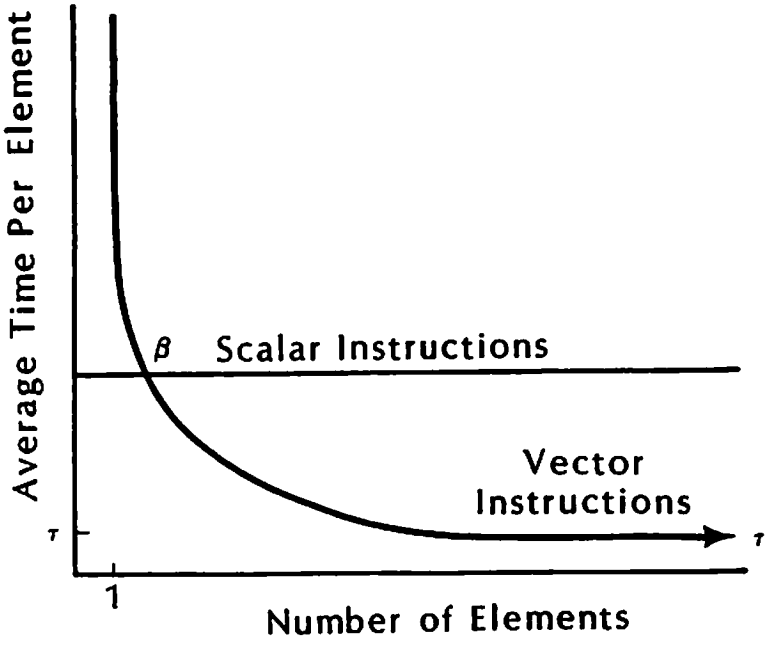


Figure 5 Time Per Operation Versus Vector Length

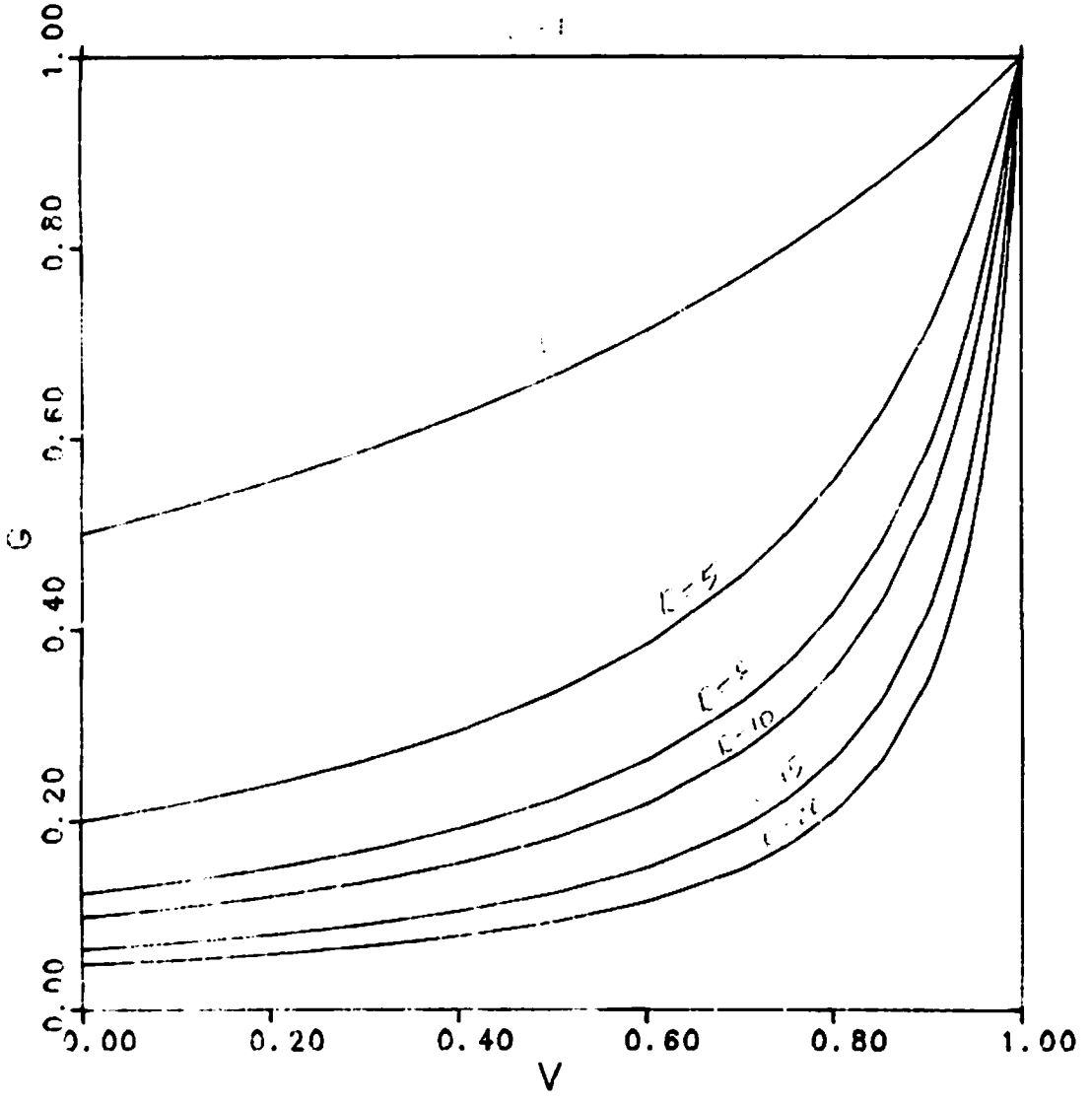


Figure 6 Percent of Performance Gained

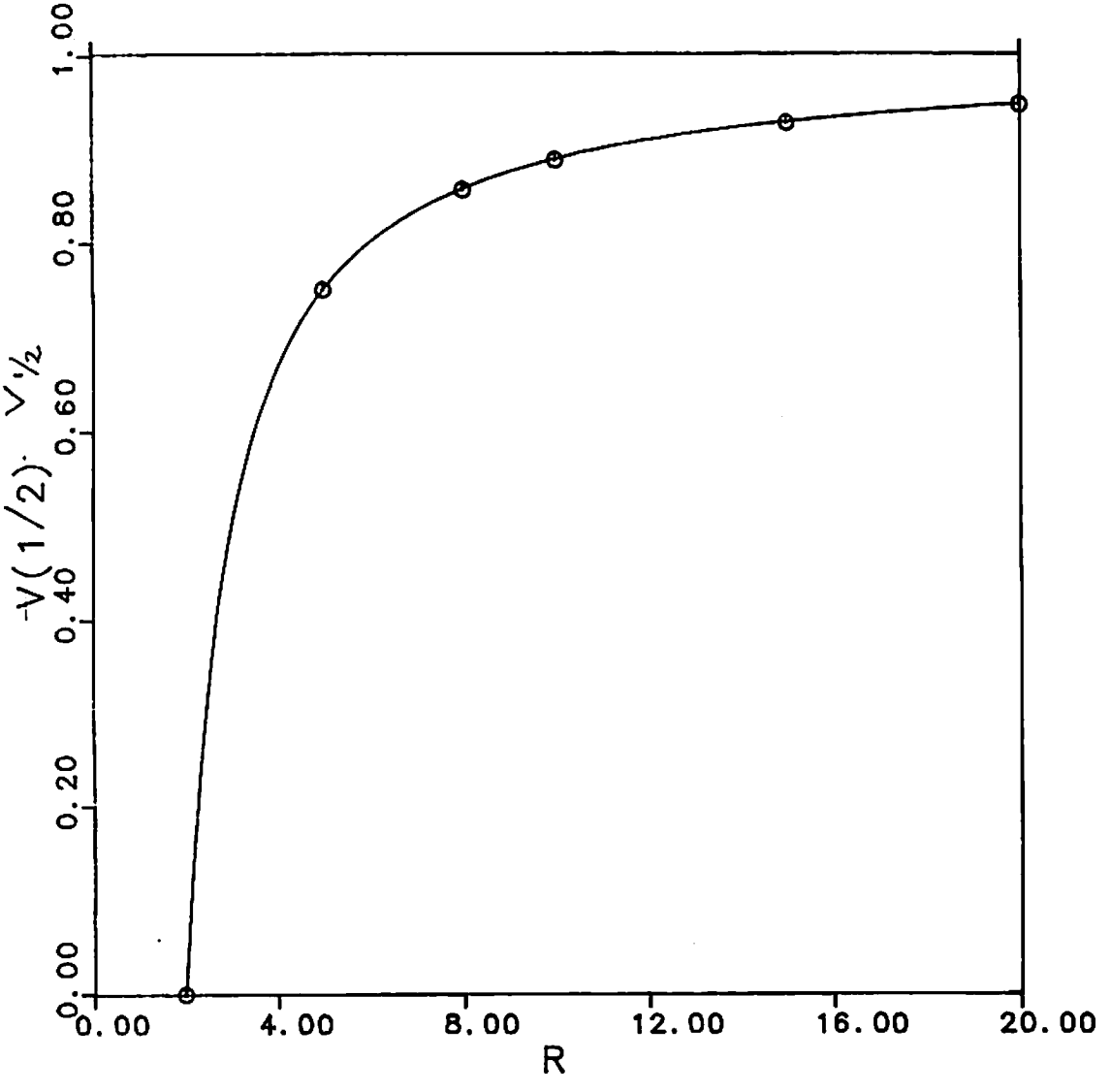


Figure 7 Percent of Arithmetic Which Must Vectorize in Order to Achieve One-Half Machine Performance

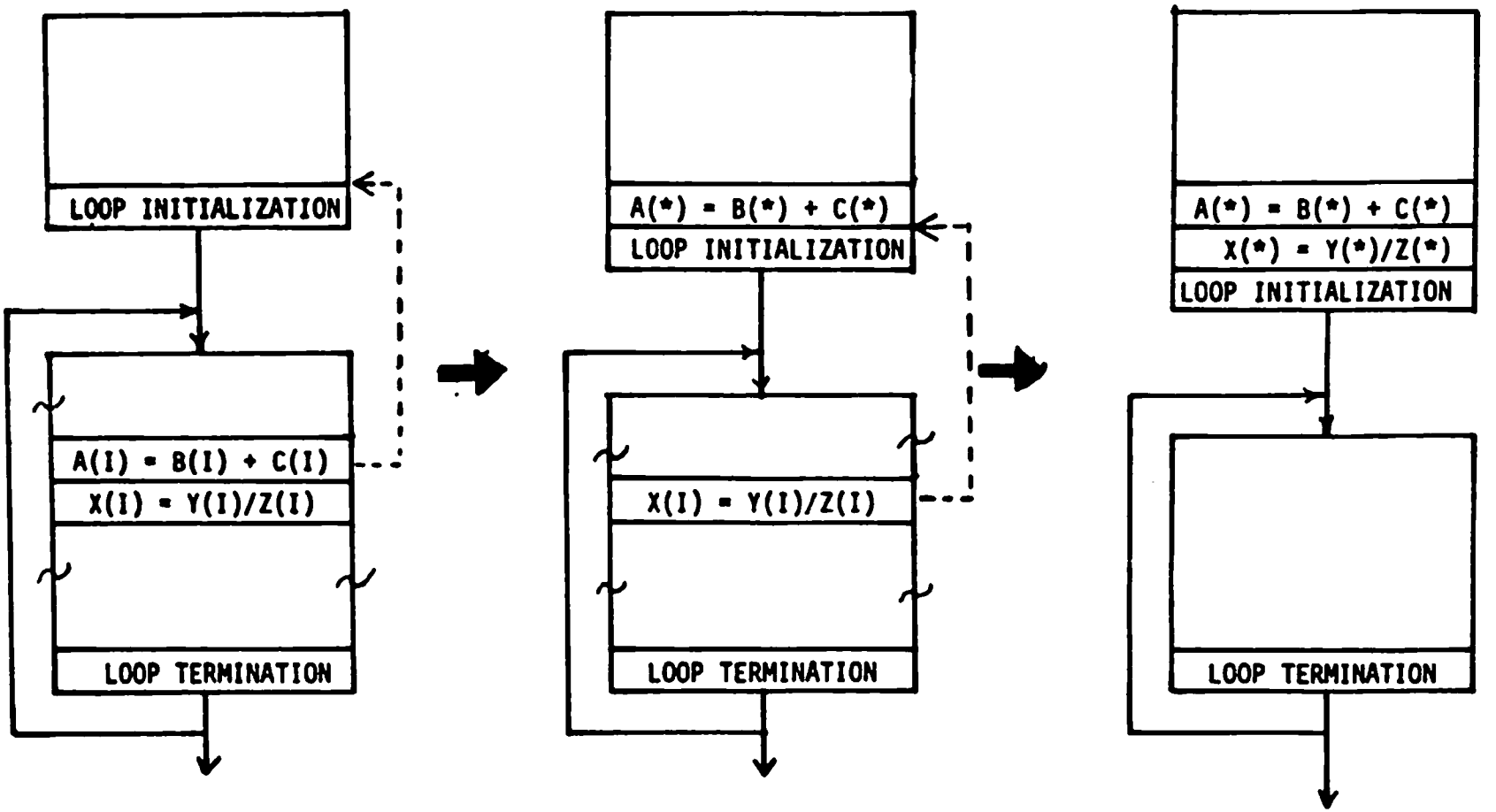


Figure 8 Promotion of Instructions Out of a DO LOOP

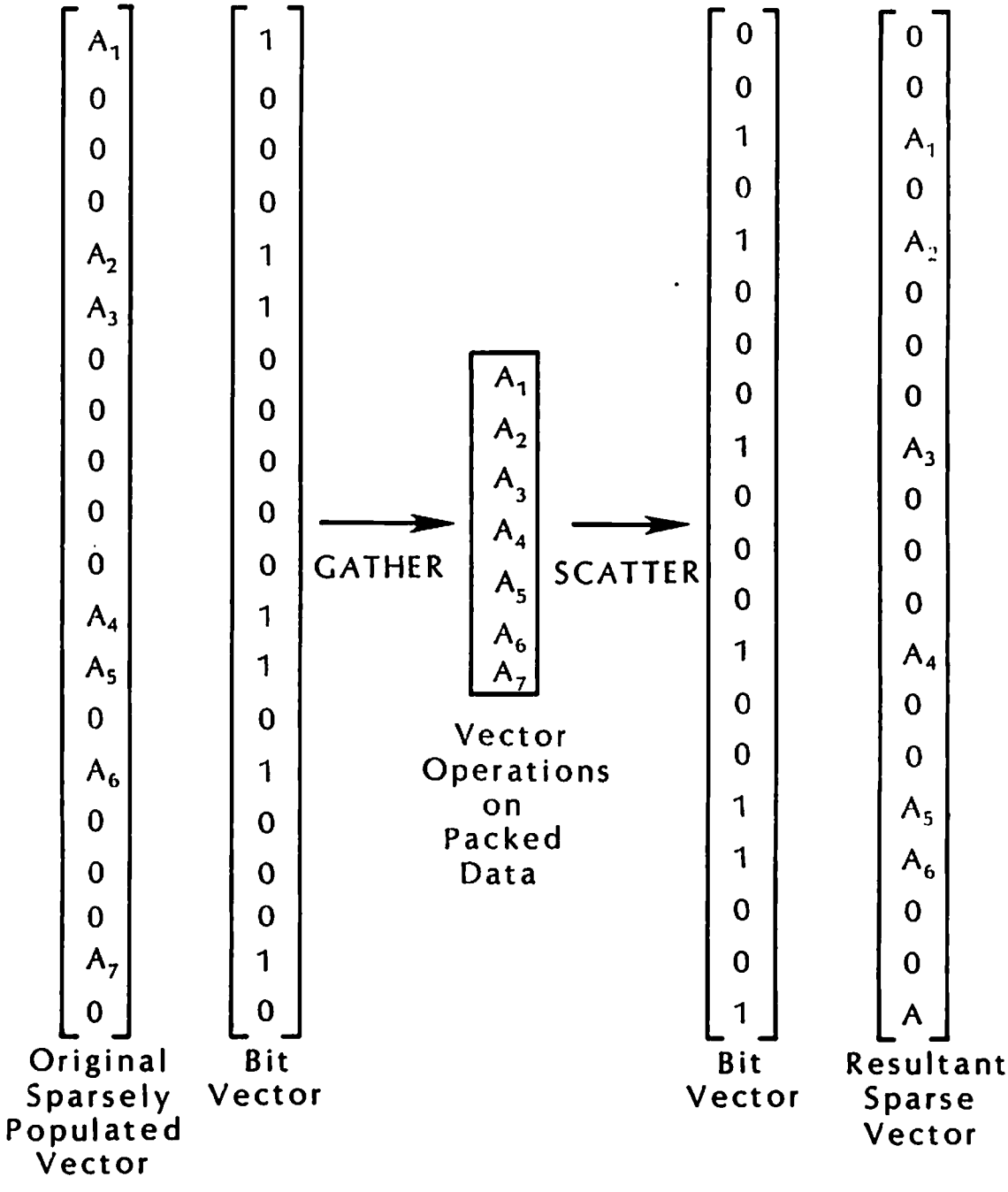


Figure 9 The Gather/Scatter Operation

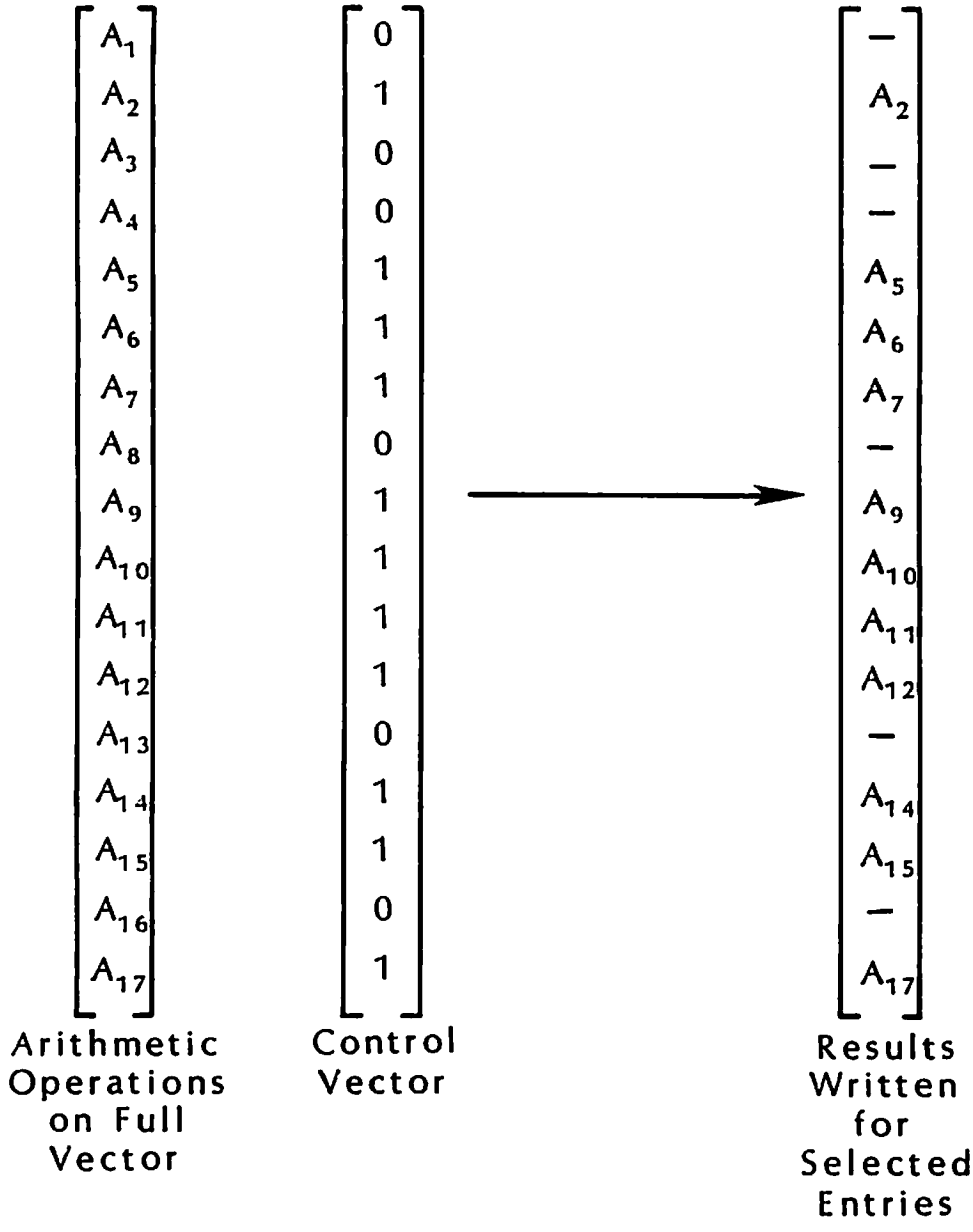
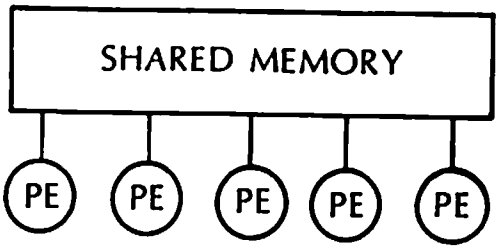
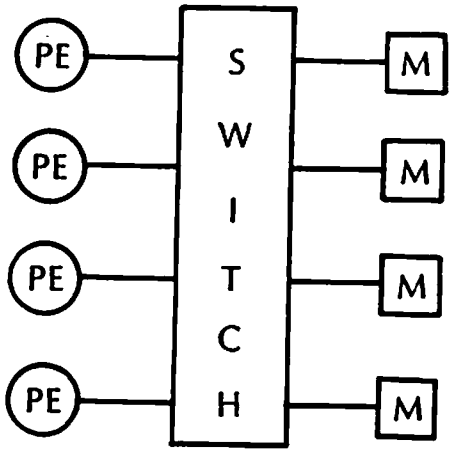


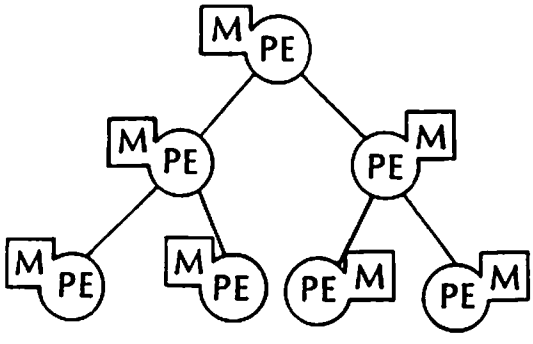
Figure 10 Operation of a Control Vector



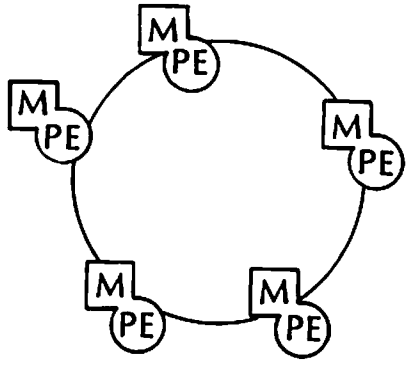
A. SHARED MEMORY DESIGN



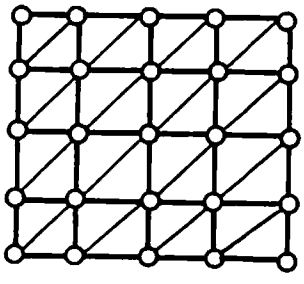
B. DANCE HALL VERSION



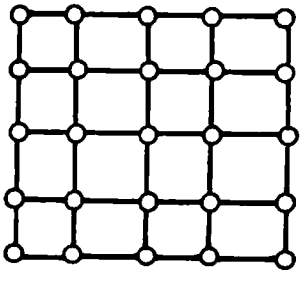
C. BINARY TREE DESIGN



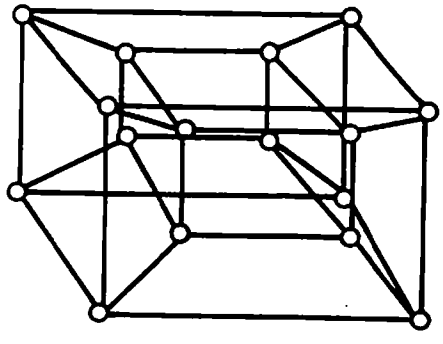
D. RING DESIGN



F. GENERALIZED ARRAY



E. 2-D ARRAY OF PROCESSORS



G. HYPER-CUBE DESIGN

Figure 11 Multi-Processor Design

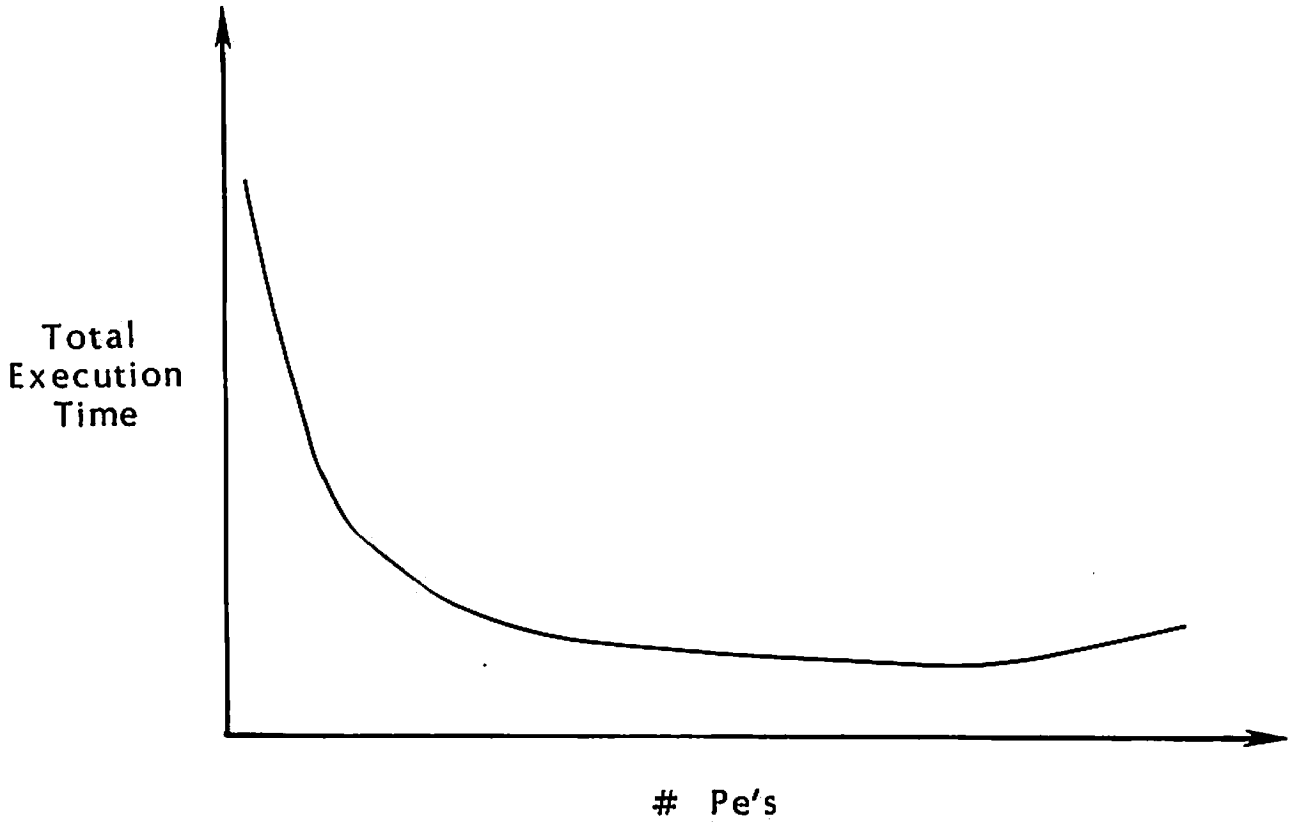


Figure 1.2 Total Execution Time Versus Number of Processors

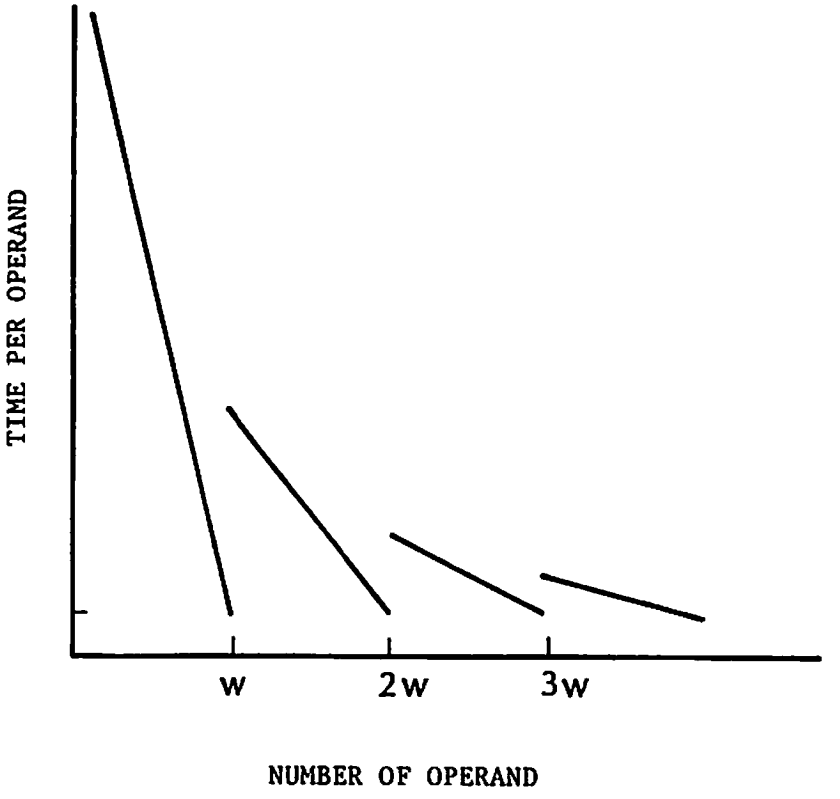


Figure 13 Time Per Operand for w Parallel Processors

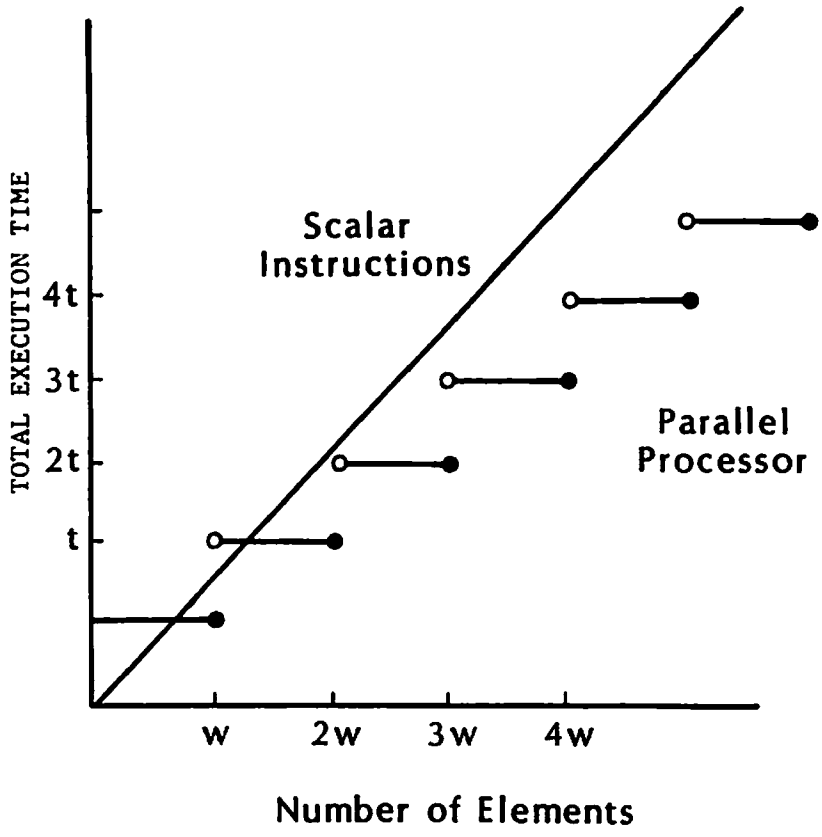


Figure 14 Total Execution Time for w Parallel Processors

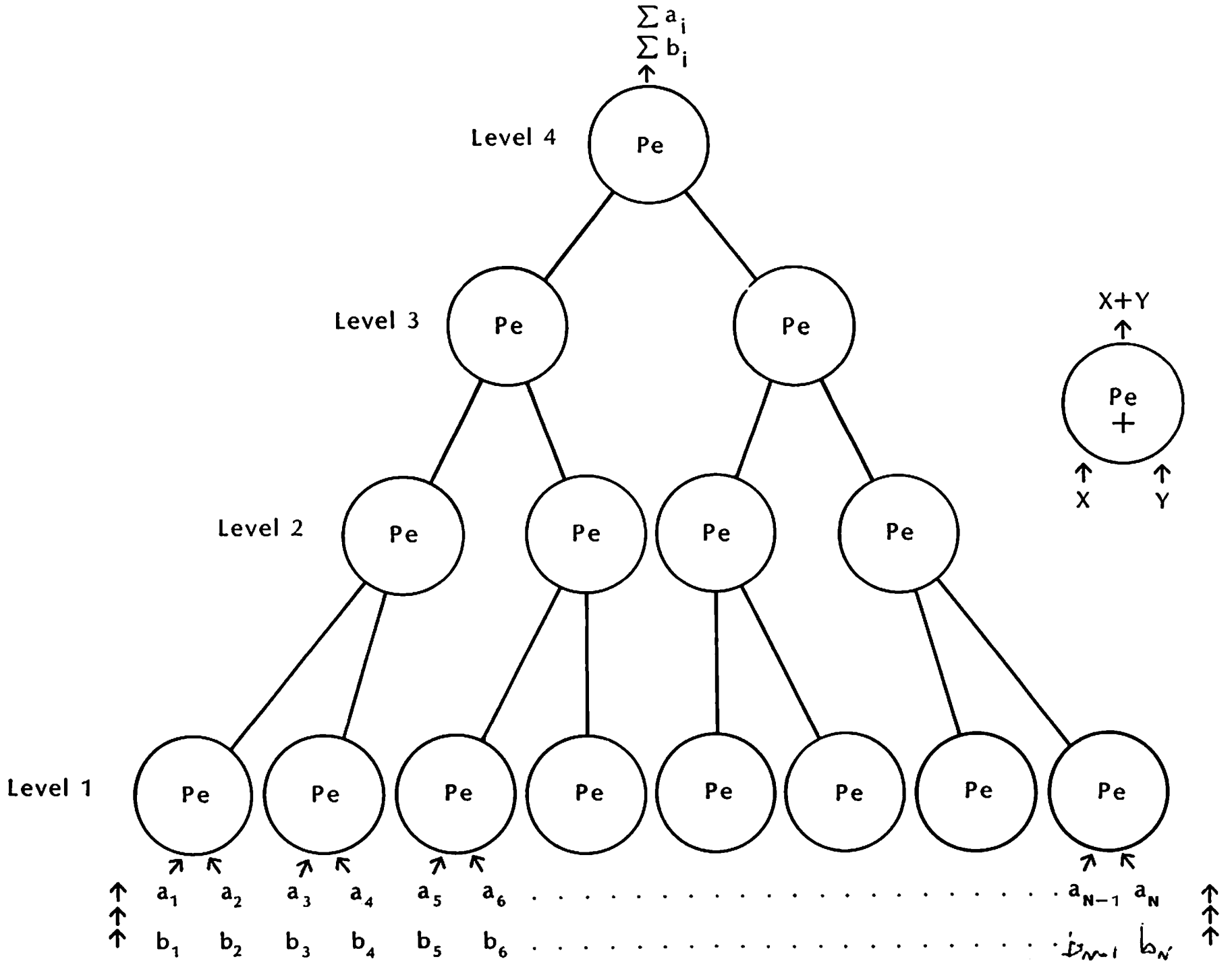


Figure 15 Summation Using a Binary Tree

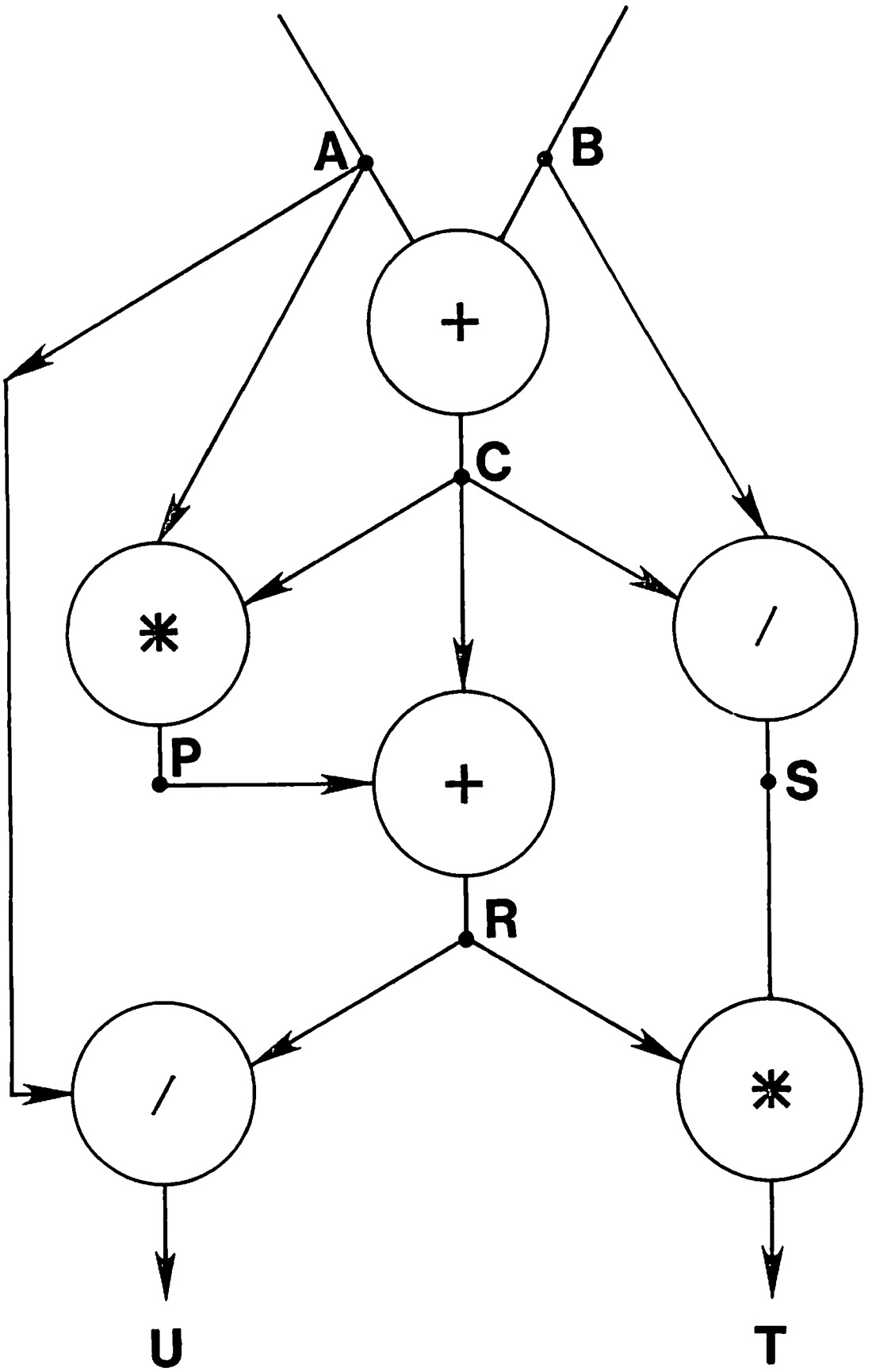


Figure 16 An Irregular Systolic Array Geometry

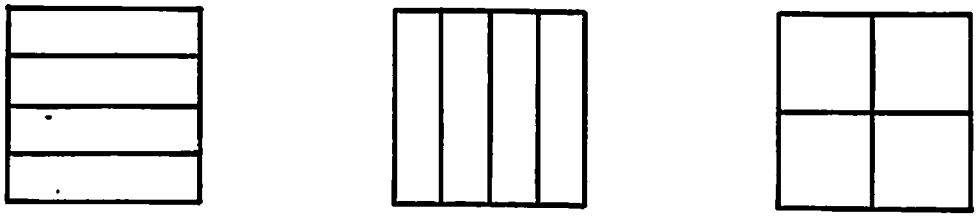


Figure 17 Division of Labor for Four Processors

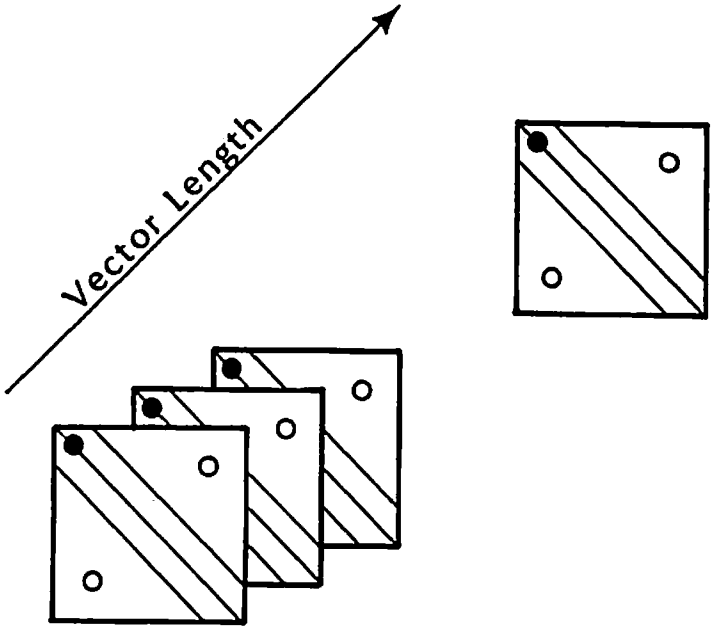


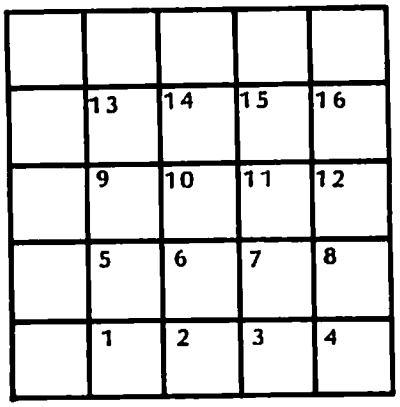
Figure 18 Stacking Independent Computations for Parallel Execution

$$\begin{bmatrix} A_1 & A_5 & A_9 & A_{13} \\ A_{14} & A_2 & A_6 & A_{10} \\ A_{11} & A_{15} & A_3 & A_7 \\ A_8 & A_{12} & A_{16} & A_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4^3 \end{bmatrix} =$$

$$\underbrace{\begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix}}_{a} + \underbrace{\begin{bmatrix} A_5 \\ A_6 \\ A_7 \\ A_8 \end{bmatrix} \begin{bmatrix} u_2 \\ u_3 \\ u_4 \\ u_1 \end{bmatrix}}_{b} + \underbrace{\begin{bmatrix} A_9 \\ A_{10} \\ A_{11} \\ A_{12} \end{bmatrix} \begin{bmatrix} u_3 \\ u_4 \\ u_1 \\ u_2 \end{bmatrix}}_{c}$$

$$\underbrace{\begin{bmatrix} A_{13} \\ A_{14} \\ A_{15} \\ A_{26} \end{bmatrix} \begin{bmatrix} u_4 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix}}_{d} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

Figure 19 Diagonal Product



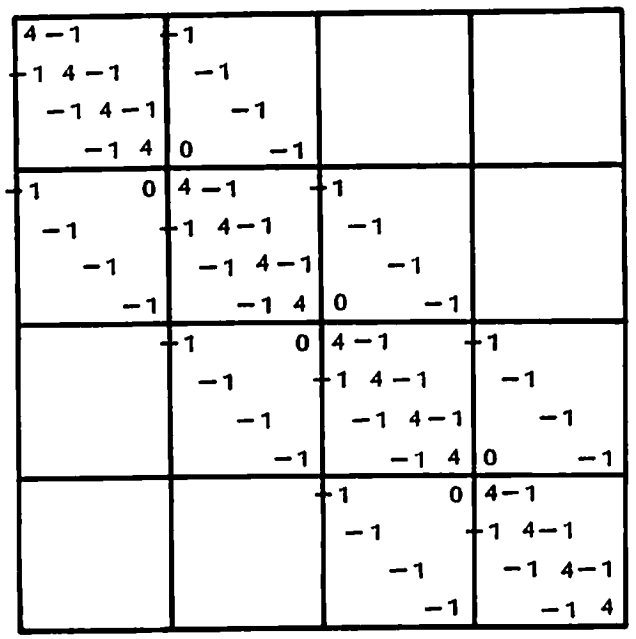
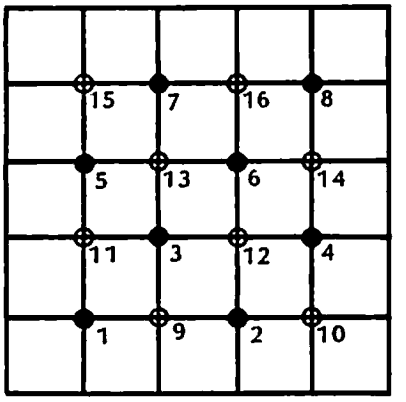


Figure 20 Natural Ordering and Resulting Matrix Problem



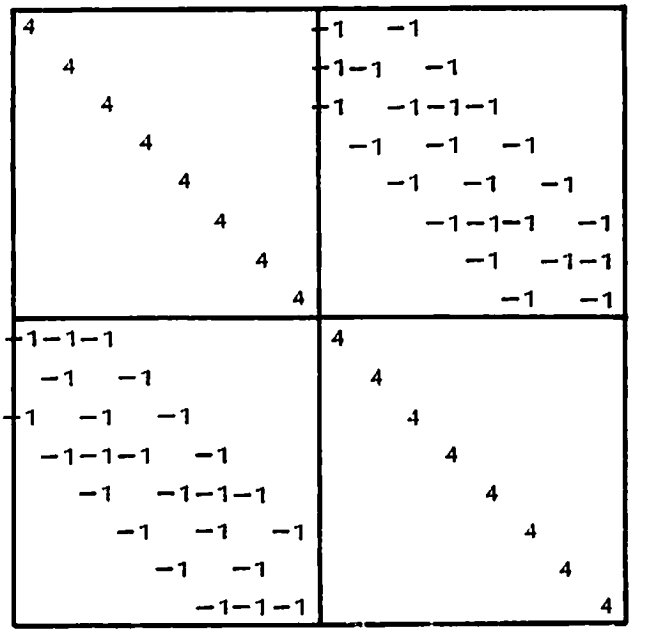


Figure .21 Red/Black Ordering and Resulting Matrix Problem

	7	11	14	16
	4	8	12	15
	2	5	9	13
	1	3	6	10

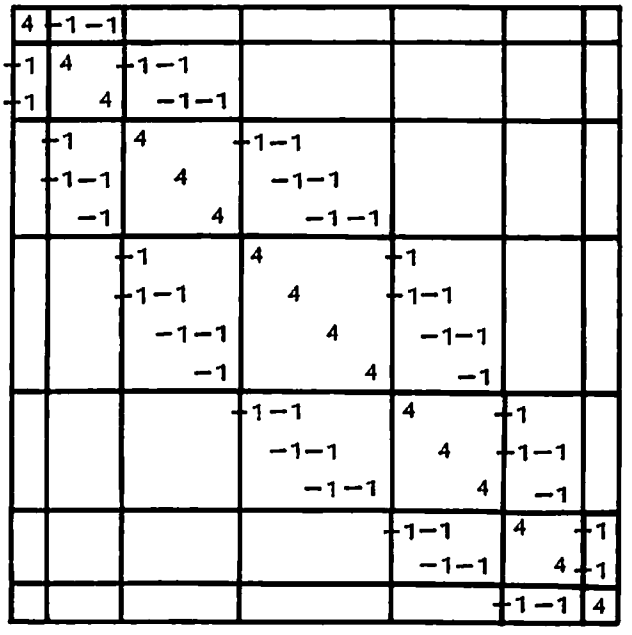
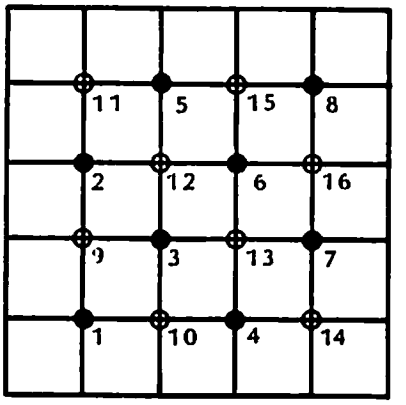


Figure 22 Diagonal Ordering and Resulting Matrix Problem



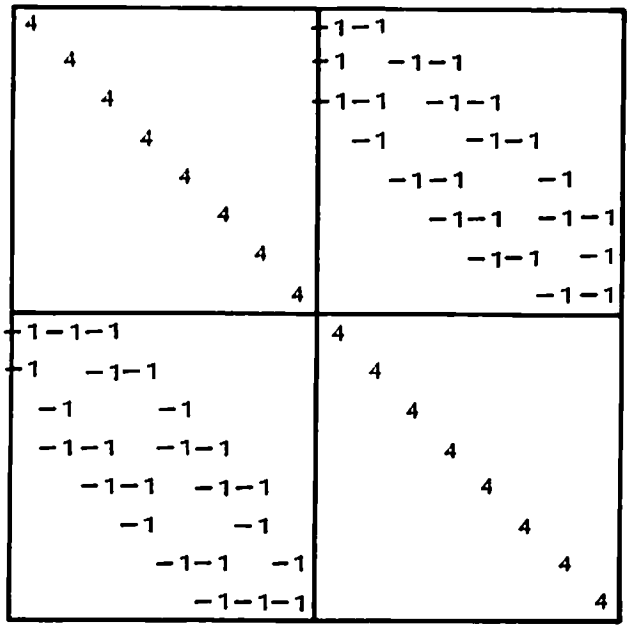


Figure 23 Diagonal Red/Black Ordering and Resulting Matrix Problem

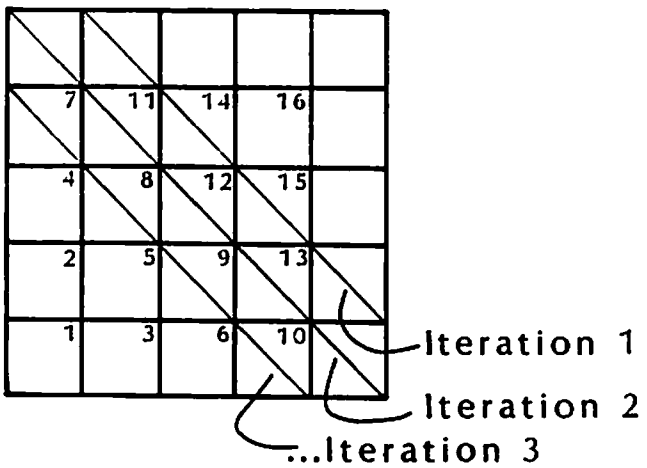


Figure 24 Multiple Concurrent Iterations

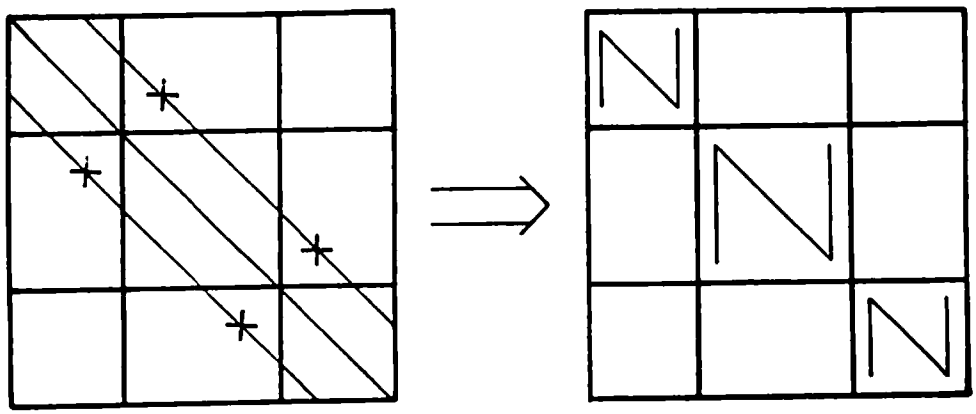


Figure 25 Substructuring a Tridiagonal Matrix