Introduction to Parallel and Vector Solution of Linear Systems

James M. Ortega

Introduction to Parallel and Vector Solution of Linear Systems

James M. Ortega
University of Virginia
Charlottesville, Virginia

Plenum Press • New York and London

Preface

Although the origins of parallel computing go back to the last century, it was only in the 1970s that parallel and vector computers became available to the scientific community. The first of these machines—the 64 processor Illiac IV and the vector computers built by Texas Instruments, Control Data Corporation, and then CRAY Research Corporation—had a somewhat limited impact. They were few in number and available mostly to workers in a few government laboratories. By now, however, the trickle has become a flood. There are over 200 large-scale vector computers now installed, not only in government laboratories but also in universities and in an increasing diversity of industries. Moreover, the National Science Foundation's Supercomputing Centers have made large vector computers widely available to the academic community. In addition, smaller, very cost-effective vector computers are being manufactured by a number of companies.

Parallelism in computers has also progressed rapidly. The largest supercomputers now consist of several vector processors working in parallel. Although the number of processors in such machines is still relatively small (up to 8), it is expected that an increasing number of processors will be added in the near future (to a total of 16 or 32). Moreover, there are a myriad of research projects to build machines with hundreds, thousands, or even more processors. Indeed, several companies are now selling parallel machines, some with as many as hundreds, or even tens of thousands, of processors.

Probably the main driving force for the development of vector and parallel computers has been scientific computing, and one of the most important problems in scientific computing is the solution of linear systems of equations. Even for conventional computers, this has continued to be an active area of research, especially for iterative methods. However, parallel and vector computers have necessitated a rethinking of even the most basic algorithms, a process that is still going on. Moreover, we are in the most

turbulent period in the history of computer architecture. It will certainly be several years, if ever, before a single parallel architecture emerges as the machine of choice.

It follows that a book on this topic is destined to be obsolete almost before it is in print. We have attempted to mitigate this problem by not tying the book to particular machines, although the influence of the older CDC CYBER 205 and CRAY-1 will be noted. However, there are many basic approaches that are essentially machine independent and that, presumably, will survive even though particular algorithms based on them may need to be modified. It is these approaches that we have tried to stress.

This book has arisen from a second-semester first-year graduate course begun in the early 1980s. Originally the course was directed primarily toward the analysis of iterative methods, with some attention to vector computers, especially the CYBER 205. Over the years more on parallel computers has been added, but the machines used for projects have been the CYBER 205 and, more recently, the CRAY X-MP. This is reflected in the exercises, which are biased heavily toward the CYBER 205.

The organization of the book is as follows. Chapter 1 discusses some of the basic characteristics of vector and parallel computers as well as the framework for dealing with algorithms on such machines. Then many of these concepts are exemplified by the relatively simple problem of matrix multiplication. Chapter 2 treats direct methods, including LU, Choleski, and orthogonal factorizations. It is assumed that the reader has had at least a first course in numerical methods and is familiar with most of these methods. Thus, the emphasis is on their organization for vector and parallel computers. Chapter 3 deals with iterative methods. Since most introductions to numerical methods deal rather lightly, if at all, with iterative methods, we devote more time in this chapter to the basic properties of such methods, independently of the computer system. In addition, many of the standard convergence theorems and other results are collected in two appendixes for those who wish a more detailed mathematical treatment. Other than the above background in numerical methods, the main prerequisites are some programming experience and linear algebra. Very basic background material in linear algebra is summarized briefly in Appendix 4.

Many important topics are not covered, and, as mentioned previously, rather little attention is given to algorithms on current particular machines. However, each section ends with "References and Extensions," which give short summaries of related work and references to the literature. It is hoped that this will help the reader to pursue topics of interest. References are given by the format Author [year] (for example, Jones [1985]) and may be found accordingly in the bibliography.

The book should be read in the spirit of a mathematics book, not as a "how-to-do-it" manual. The incomplete code segments that are given are

meant to be illustrative, not the basics of a running code. Anyone wishing to use a linear equation solver for a particular parallel or vector machine is strongly advised *not* to start from the material in this book, especially for direct methods. Rather, see what routines, especially LINPACK, are already available on that machine.

The following conventions are used. Vectors are denoted by lower case bold and matrices by upper case italic. Equation numbers are given by chapter and section; thus (3.2.4) is the fourth numbered equation in Section 3.2. Theorems and definitions are likewise numbered within a section by, for example, 3.2.6.

I am indebted to Sandra Shifflett, Beverly Martin, and especially, B. Ann Turley for typing the manuscript, and to many students and reviewers for their comments.

Charlottesville, Virginia James M. Ortega

Contents

Chapter 1. Introduction 1

1.1. Vector and Parallel Computers 1
1.2. Basic Concepts of Parallelism and Vectorization 20
1.3. Matrix Multiplication 36

Chapter 2. Direct Methods for Linear Equations 59

2.1. Direct Methods for Vector Computers . . 59
2.2. Direct Methods for Parallel Computers . . 85
2.3. Banded Systems 108

Chapter 3. Iterative Methods for Linear Equations 133

3.1. Jacobi's Method 133
3.2. The Gauss-Seidel and SOR Iterations . . 156
3.3. Minimization Methods 185
3.4. The Preconditioned Conjugate Gradient Method 196

Appendix 1. The *ijk* Forms of *LU* and Choleski Decomposition 235

Appendix 2. Convergence of Iterative Methods 253

Appendix 3. The Conjugate Gradient Algorithm . . . 269

Appendix 4. Basic Linear Algebra 281

Bibliography 285

Index 299

1

Introduction

1.1. Vector and Parallel Computers

In the early 1970s, computers began to appear that consisted of a number of separate processors operating in parallel or that had hardware instructions for operating on vectors. The latter type of computer we will call a *vector computer* (or *processor*) while the former we will call a *parallel computer* (or *processor*).

Vector Computers

Vector computers utilize the concept of *pipelining*, which is the explicit segmentation of an arithmetic unit into different parts, each of which performs a subfunction on a pair of operands. This is illustrated in Figure 1.1-1 for floating point addition.

In the example of Figure 1.1-1, a floating point adder is segmented into six sections, each of which does one part of the overall floating point addition. Each segment can be working on one pair of operands, so that six pairs of operands can be in the pipeline at a given time. The advantage of this segmentation is that results are being computed at a rate that is 6 times faster (or, in general, K times, where K is the number of segments) than an arithmetic unit that accepts a pair of operands and computes the result before accepting the next pair of operands. However, in order to utilize this capability, the data must reach the arithmetic units rapidly

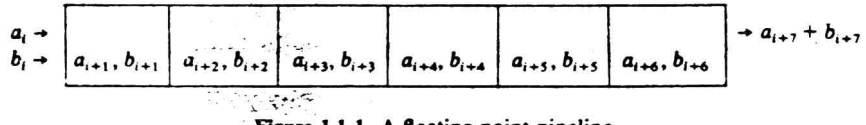

Figure 1.1-1. A floating point pipeline.

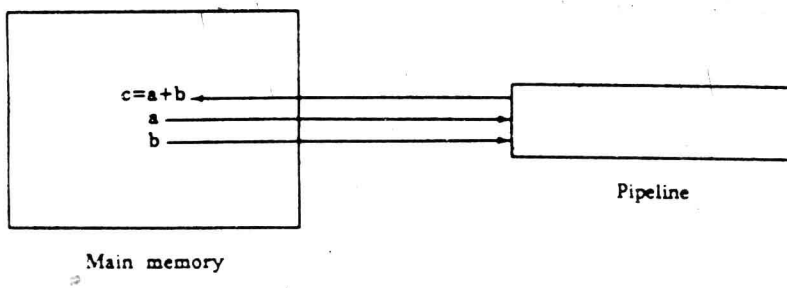

Figure 1.1-2. Memory-to-memory addition operation.

enough to keep the pipeline full. As one aspect of this, the hardware instructions for, say, vector addition eliminate the need for separate load and store instructions for the data. A single hardware instruction will control the loading of the operands and storing of the results.

Memory-to-Memory Processors

Control Data Corporation (CDC) has produced a line of vector processors starting with the STAR-100 in 1973. This machine evolved into the CYBER 203 in the late 1970s and then the CYBER 205 in the early 1980s. These machines were all *memory-to-memory* processors in that vector operations took their operands directly from main memory and stored the result back in main memory. This is illustrated in Figure 1.1-2 for a vector addition.

Register-to-Register Processors

Cray Research, Inc. has produced vector processors since the mid 1970s that are examples of *register-to-register* processors. By this we mean that the vector operations obtain their operands from very fast memory, called *vector registers*, and store the results back into vector registers. This is illustrated in Figure 1.1-3 for vector addition. In Figure 1.1-3, each vector

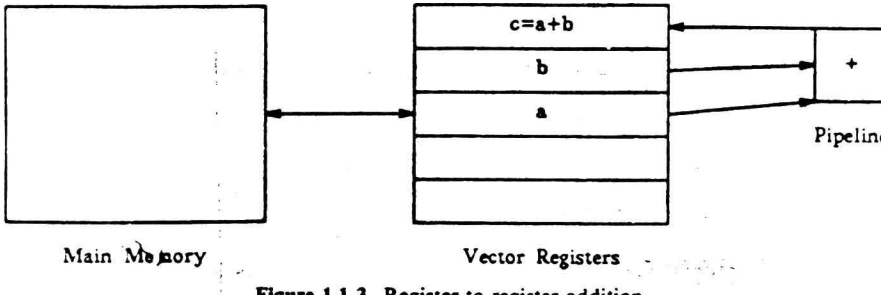

Figure 1.1-3. Register-to-register addition.

register is assumed to hold a certain number of words. For example, on the CRAY machines, there are eight vector registers, each of which holds 64 floating point numbers. Operands for a vector addition are obtained from two vector registers and the result is put into another vector register. Prior to the vector addition, the vector registers must be loaded from main memory, and, at some point, the result vector is stored back into main memory from a vector register. It is usually desirable on this type of computer to use data in the vector registers as much as possible while they are available; several examples of this will be given in later sections.

Memory Hierarchies

Vector registers play somewhat the same role as cache memory on conventional computers. More recent vector computers may have a more complex memory hierarchy. For example, the CRAY-2, in addition to vector registers, has a 16,000-word fast local memory in each processor. Other machines such as the CRAY X-MP series can have a back-up storage (the Solid-State Storage Device), which is slower than main memory but considerably faster than disk. And, of course, all the machines will have disk units. The challenge is to use these various types of storage in such a way as to have the data ready for the arithmetic units when it is needed.

Arithmetic Units

As discussed above, vector processors will have pipelined arithmetic units to handle vector operations. The form of these may differ, however. The CDC machines have used *reconfigurable* units in which a single pipeline can perform the different arithmetic operations but it must be configured for a particular operation, say addition, before that operation can begin. The CRAY machines, on the other hand, have used separate pipelines for addition and multiplication, as well as other functions. And some Japanese machines (for example, the NEC SX-2) have multiple separate pipelines (for example, four pipelines for addition, four for multiplication, etc.). There may also be multiple reconfigurable pipelines. For example, the CYBER 205 allows 1, 2, or 4 pipelines, which are used in unison on a given vector operation (rather than processing separate operations.)

Vector hardware operations are always provided for the addition of two vectors, the elementwise product of two vectors, and either the elementwise quotient of two vectors or the reciprocals of the elements of a vector. There may also be vector instructions for more complex operations such as the square roots of the elements of a vector, the inner product of two vectors, the sum of the elements of a vector, sparse vector operations, and so on. There may also be certain operations that can be handled very

efficiently. A *linked triad* is an operation of the form $\mathbf{a} + \alpha\mathbf{b}$, where **a** and **b** are vectors and α is a scalar. Other forms of a linked triad are also possible such as $(\mathbf{a} + \alpha)\mathbf{b}$, where $\mathbf{a} + \alpha$ is the vector with α added to all of its components. The CYBER 205 can perform these linked triad operations at almost the same speed as a vector addition or multiplication operation. The linked triad $\mathbf{a} + \alpha\mathbf{b}$ is also known as a *saxpy* operation, and this is the more common designation, especially amongst CRAY users. We will, however, use the term "linked triad."

Machines with separate arithmetic pipelines usually allow the possibility of *chaining* arithmetic units together so that results from one unit are routed directly to another without first returning to a register. This is illustrated in Figure 1.1-4 for a linked triad operation.

Most vector computers provide separate units for scalar arithmetic. These units may also be pipelined but do not accept vector operands as the vector pipelines do. They can run concurrently with the vector pipelines, and usually produce scalar results 5–10 times slower than the maximum rates of the vector pipelines.

Vector Start-up Times

The use of vector operations incurs an overhead penalty as shown in the following approximate formula for the time T for a vector operation:

$$T = S + KN \tag{1.1.1}$$

In (1.1.1), N is the length of the vectors involved, K is the time interval at which results are leaving the pipeline, and S is the *start-up time*. S is the time for the pipeline to become full, and includes the time to initiate the fetch of the operands. This is typically much larger for memory-to-memory machines than register to register machines provided that the time to load the vector registers from main memory is not included. S also includes the time for configuring the pipeline on those machines with reconfigurable pipelines.

The result rate K is closely related to the *cycle time* (also called the clock period, clock time, or minor cycle time) of the machine. One result

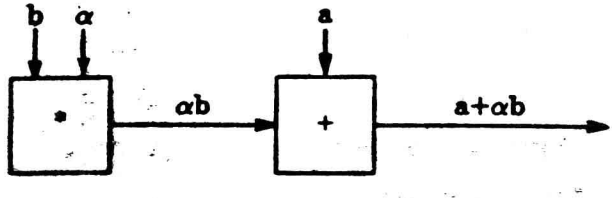

Figure 1.1-4. Chaining.

will leave a pipeline unit every cycle time once the pipeline is full. Hence, on many machines K is just the cycle time. However, on those machines with multiple reconfigurable pipelines, or multiple addition and multiplication units, K will be the cycle time divided by the number of multiple units. For example, the cycle time of the CYBER 205 is 20 ns (nanoseconds = 10^{-9} seconds) so that on a two-pipeline 205, K is 10 ns for addition and multiplication and 5 ns on a four-pipeline 205. Similarly, the cycle time of the NEC SX-2 is 6 ns and there are four units for addition and multiplication. Hence, K = 1.5 ns for these operations. The value of K may be larger for more complex operations such as square root or inner product.

From (1.1.1), the time per result is

$$T_R = K + S/N \tag{1.1.2}$$

which has the graph shown in Figure 1.1-5 as a function of the vector length N. Figure 1.1-5 illustrates the need to use sufficiently long vectors so as to amortize the start-up time over many results. Result rates on current vector processors are on the order of a few nanoseconds, while start-up times S range from several tens of nanoseconds on register to register machines to several hundreds of nanoseconds on memory to memory machines.

Another way of plotting the result rate is to use the number of results per time unit given by

$$R = T_R^{-1} = \frac{N}{S + KN} \tag{1.1.3}$$

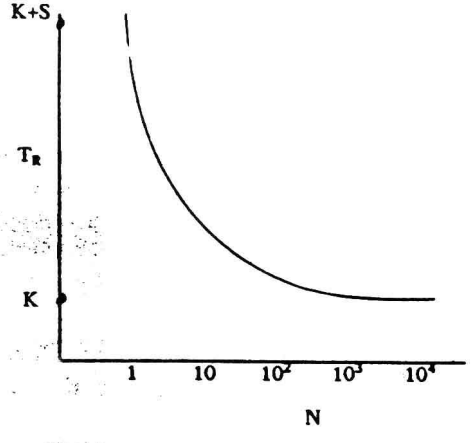

Figure 1.1-5. Time per result relation.

If $S = 0$, or as $N \to \infty$, (1.1.3) yields

$$R_\infty = \frac{1}{K} \tag{1.1.4}$$

which is called the *asymptotic result rate*. This is the (not achievable) maximum result rate when the start-up overhead is ignored. For example, if K is 10 ns, then the asymptotic rate is $R_\infty = 10^8$ results per second, or 100 mflops, where "mflops" denotes megaflops or one million floating point operations per second.

The function R of (1.1.3) is plotted in Figure 1.1-6 as a function of N under the assumption that $K = 10$ ns and $S = 100$ ns as well as $S = 1000$ ns. As illustrated in Figure 1.1-6, both curves tend to the asymptotic result rate of 100 mflops as N increases, but the approach is much more rapid initially for smaller S.

A number of some interest is $N_{1/2}$, which is defined to be the vector length for which half the asymptotic rate is achieved. For example, if $K = 10$ ns, it follows from (1.1.3) that $N_{1/2} = 100$ for $S = 1000$, while if $S = 100$, then $N_{1/2} = 10$. Another important number is the *cross-over point*, N_c, at which vector arithmetic becomes faster than scalar arithmetic. Suppose that scalar arithmetic can be done at an average rate of 10 mflops. Then the crossover point N_c is the value of N for which $R \geq 10$ mflops. For $S = 1000$, using (1.1.3), this is the minimum value of N for which

$$\frac{N}{(1000 + 10N)10^{-9}} \geq 10 \times 10^6$$

or $N_c = 12$. Thus, for vectors of length less than 12, the use of vector

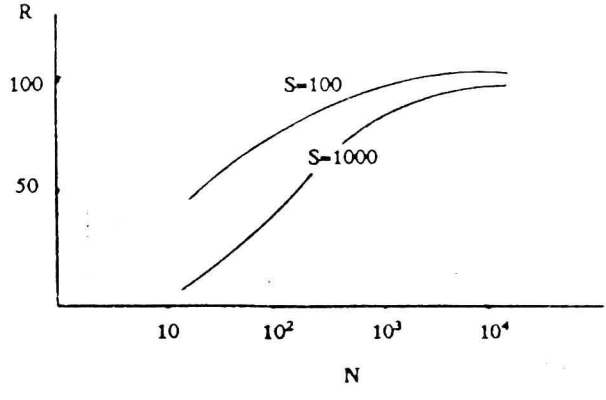

Figure 1.1-6. Result rates in mflops.

arithmetic is slower than scalar arithmetic. On the other hand, for $S = 100$, $N_c = 2$; in this case, vector operations are more efficient for all vector lengths except the trivial case of vectors of length 1. Note that the cross-over point is highly dependent on the scalar arithmetic rate. In the previous examples, if the scalar arithmetic rate was only 5 mflops, then $N_c = 6$ if $S = 1000$ and $N_c = 1$ if $S = 100$.

Vectors

On vector computers, there are restrictions on what constitutes a vector for purposes of vector arithmetic operations. On register to register machines, a vector for arithmetic instructions will be a sequence of contiguous elements in a vector register, usually starting in the first element of a register. An important consideration for these machines, then, is what constitutes a vector in main memory for the purpose of loading a vector register. This is essentially the same consideration as what constitutes a vector for arithmetic operations on memory to memory computers.

Elements that are sequentially addressable always constitute a suitable vector, and on some machines (for example, CDC machines) these are the only vectors. In the sequel, we will use *contiguous* as a synonym for sequentially addressable, although sequentially addressable elements are usually not physically contiguous in memory; rather, they are stored in different memory banks. (However, on CDC machines data are accessed in "superwords" of eight words that are physically contiguous. Here, it is superwords that are in different memory banks.) On other machines, elements with a constant stride form a vector. By *stride* we mean the address separation between elements. Thus, elements with addresses a, $a + s$, $a + 2s$, ... have a constant stride equal to s. In the special case that $s = 1$, the elements are sequentially addressable.

For elements that are not stored with a constant stride, or for elements stored with a constant stride greater than one on those machines for which a vector consists only of sequentially addressable elements, it is necessary to use auxiliary hardware or software instructions to reformat the data to be an acceptable vector. A *gather* operation will map a given number of elements specified by a list of their addresses into a vector. A *merge* operation will combine two vectors into a single vector. A *compress* operation will map elements at a constant stride into a vector. All of these operations, of course, require a certain amount of time, which adds to the overhead of the vector arithmetic operations. Moreover, after the vector arithmetic is done, it may be necessary to store the results in a nonvector fashion. A *scatter* operation, the inverse of gather, stores elements of a vector into positions prescribed by an associated address list. A primary consideration in developing algorithms for vector computers is to have the data arranged

so as to minimize the overhead that results from having to use the above data management operations.

Parallel Computers

The basic idea of a parallel computer is that a number of processors work in cooperation on a single task. The motivation is that if it takes one processor an amount of time t to do a task, then p processors can do the task in time t/p. Only for very special situations can this perfect "speedup" be achieved, however, and it is our goal to devise algorithms that can take as much advantage as possible, for a given problem, of multiple processors.

The processors of a parallel computer can range from very simple ones that do only small or limited tasks to very powerful vector processors. Most of our discussions will be directed towards the case in which the processors are complete sequential processors of moderate power, although attention will also be paid to having vector processors.

MIMD and SIMD Machines

A first important dichotomy in parallel systems is how the processors are controlled. In a Single-Instruction-Multiple-Data (SIMD) system, all processors are under the control of a master processor, called the *controller*, and the individual processors all do the same instruction (or nothing) at a given time. Thus, there is a single instruction stream operating on multiple data streams, one for each processor. The Illiac IV, the first large parallel system (which was completed in the early 1970s), was an SIMD machine. The ICL DAP, a commercial machine introduced in England in 1977, the Goodyear MPP, especially constructed for NASA in the early 1980s, and the Connection Machine, a commercial machine of the mid-1980s, are also machines of SIMD type, although the individual processors are relatively simple 1-bit machines: 4096 in the DAP, 16,484 in the MPP, and 64,936 in the Connection Machine. Vector computers may also be conceptually included in the class of SIMD machines by considering the elements of a vector as being processed individually under the control of a vector hardware instruction.

Most parallel computers built since the Illiac IV are Multiple-Instruction-Multiple-Data (MIMD) systems. Here, the individual processors run under the control of their own program, which allows great flexibility in the tasks the processors are doing at any given time. It also introduces the problem of synchronization. In an SIMD system, synchronization of the individual processors is carried out by the controller, but in an MIMD system other mechanisms must be used to ensure that the processors are doing their tasks in the correct order with the correct data. Synchronization will be discussed more in later sections.

Shared versus Local Memory

Another important dichotomy in parallel computers is *shared* versus *local* memory. A shared memory system is illustrated in Figure 1.1-7. Here, all the processors have access to a common memory. (In the sequel, we will use the terms "shared memory" and "common memory" interchangeably.) Each processor can also have its own local memory for program code and intermediate results. The common memory would then be used for data and results that are needed by more than one processor. All communication between individual processors is through the common memory. A major advantage of a shared memory system is potentially very rapid communication of data between processors. A serious disadvantage is that different processors may wish to use the common memory simultaneously, in which case there will be a delay until the memory is free. This delay, called *contention time*, can increase as the number of processors increases.

An alternative to shared memory systems are local memory systems, in which each processor can address only its own memory. Communication between processors takes place by *message passing*, in which data or other information are transferred between processors.

Interconnection Schemes

Probably the most important and interesting aspect of parallel computers is how the individual processors communicate with one another. This is particularly important for systems in which the processors have only local memory, but it is also important for shared memory systems since the connection to the shared memory can be implemented by different communication schemes. We shall next discuss briefly a number of the more common interconnection schemes.

Completely Connected. In a completely connected system, each processor has a direct connection to every other processor. This is illustrated

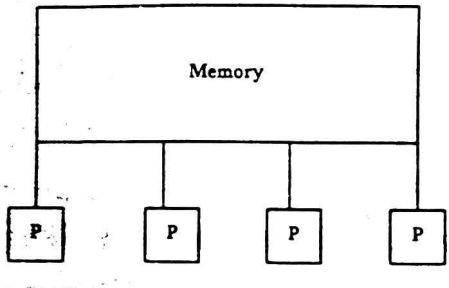

Figure 1.1-7. A shared memory system.

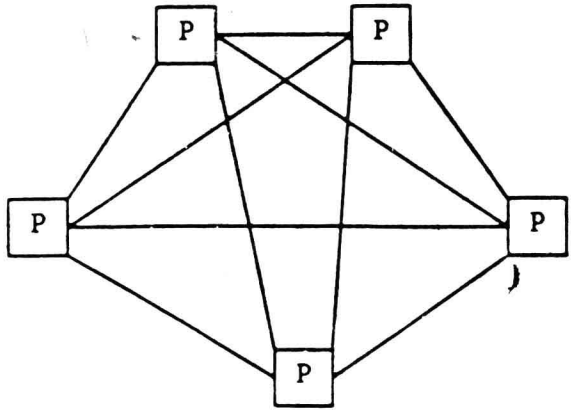

Figure 1.1-8. A completely connected system.

in Figure 1.1-8. A completely connected system of p processors requires $p - 1$ lines emanating from each processor, which is impractical if p is large.

Crossbar Switch. Another approach to a completely connected system is through a *crossbar switch* as illustrated in Figure 1.1-9. As shown there, each processor can be connected to each memory, in principle, through switches that make the connection. This has the advantage of allowing any processor access to any memory with a small number of connection lines. But the number of switches to connect p processors and p memories is p^2, which becomes impractical for large p. One early parallel system, the C.mmp developed at Carnegie-Mellon University in the early 1970s, used this scheme to connect 16 PDP-11 minicomputers.

Bus and Ring. A bus network is illustrated in Figure 1.1-10. Here, all processors are connected by a (high-speed) bus. An advantage is a very

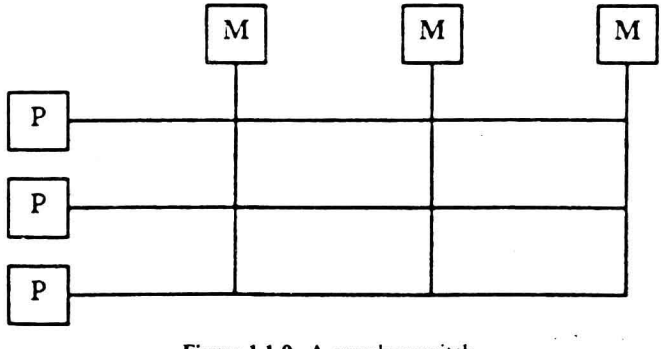

Figure 1.1-9. A crossbar switch.

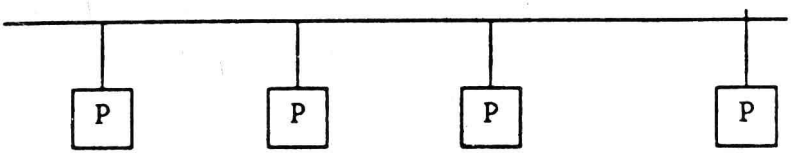

Figure 1.1-10. A bus network.

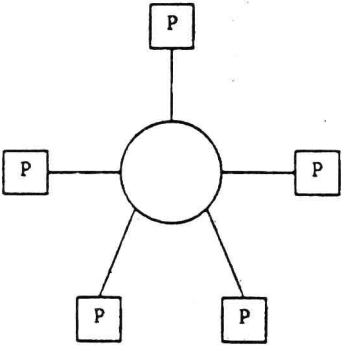

Figure 1.1-11. A ring network.

small number of connection lines, but there may be contention (*bus contention*) for use of the bus by different processors; this can become a severe problem as the number of processors increases.

A *ring network* is a closed bus network as illustrated in Figure 1.1-11. Here data move around the ring and are available to each processor in turn. Several parallel computers have used bus or ring connections of various types. Some systems have used a bus to connect processors to a global memory; an example of the mid 1980s is the Flexible Computer Corporation FLEX/32, a system with 20 processors. Other systems have used a bus to implement a local memory message passing system. An example was ZMOB, an experimental system developed at the University of Maryland in the early 1980s.

Mesh Connection. One of the most popular interconnection schemes historically has been to have each processor connected to only a few neighboring processors. The simplest example of this is a *linear array* illustrated in Figure 1.1-12. Here, each processor is connected to two nearest neighbors (except the end processors, which are connected to only one).

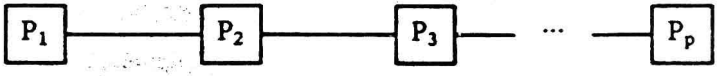

Figure 1.1-12. A linear array.