Introduction to the Principles of Parallel Computation

By

Dr. Daniel C. Hyde
Department of Computer Science
Bucknell University
Lewisburg, PA 17837
hyde@bucknell.edu

Minor Update Jan 14, 1998

Updated January 16, 1995

DRAFT

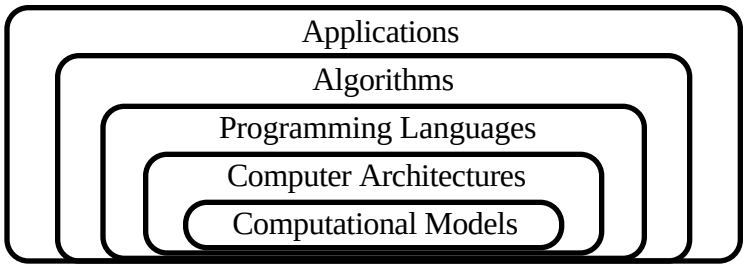
Page left blank intentionally.

Table of Content

Preface

A few years ago, parallel computers could be found only in research laboratories. Today they are widely available commercially. The field of parallel processing has matured to the point where undergraduate textbooks are needed. This text has the primary goal of introducing the principles of parallel processing to undergraduates. The principal audience is intended to be junior and senior computer science majors in an elective course on parallel processing. However, the practicing professional, who is new to the field, will find it valuable to study the text. Also, the scientist who is interested in computational science would benefit from reading the text. The prerequisite is a course in computer organization. Courses in programming language design and operating systems will be useful but are not absolutely essential.

Parallelism covers a wide spectrum of material, from hardware design of adders to the analysis of theoretical models of parallel computation. In fact, aspects of parallel processing could be incorporated into every computer science course in the curriculum. In order to deal with this massive amount of subject material, we need a way to organize it. This textbook is organized around a five layered model of parallel processing.



The Five Layers of Parallel Processing

Central to the material are parallel computational models or theoretical ways to view computation. Selecting a computational model, we implement a parallel architecture based on the computational model. We can design a parallel programming language based on the computational model which generates code for the architecture. Or we could first design a programming language based on a computational model, then implement an architecture based on the language. We develop and analyze parallel algorithms written in the programming language. Finally, we use the algorithms in parallel applications. These five layers span a large portion of computer science.

Chapter One of the text introduces important terms and concepts, e. g., interleaved memory and pipelining. To motivate the students as well as set the stage, a brief history of parallelism is included. Chapter Two covers the important and sometimes tricky topic of how one measures performance on a parallel computer. Historically important, pipelined vector processors and vectorization are covered in Chapter Three by using the Cray-1 as a case study. Chapters Four through Seven discuss pertinent concepts and issues in the inner four layers as described above, namely parallel computational models, parallel computer architectures, parallel programming languages and parallel algorithms. Chapter Eight deals with the often ignored topic of computing environments on parallel computers. If an instructor needs more material, he or she can choose several of the parallel machines discussed in Chapter Nine. The last chapter attempts to predict the future of parallel processing.

To help the student overcome the jargon of the field, a glossary is included after Chapter Ten. The glossary is important because many terms are not defined or are ill defined in the area of parallel processing. For the student's further study, an extensive bibliography is included as well.

Undergraduates need to practice what is presented in lecture. Therefore, students should attempt the exercises at the end of each chapter, as they are an integral part of the learning process. When I teach the course, my students have a two-hour formal laboratory each week where they perform experiments or explore specific concepts. I highly recommend a series of laboratories as well as written homework and programming assignments. Of great importance to undergraduates is programming on a real parallel machine. If no parallel machine is available on your campus, contact one of the National Supercomputer Centers and arrange computer time for your class.

Chapter 1 Introduction

1.1 What is Parallelism?

A few years ago, parallel computers could be found only in research laboratories. Today they are widely available commercially. The field of parallel processing has matured to the point where computer science undergraduates should study it. Parallelism covers a wide spectrum of material, from hardware design of adders to the analysis of theoretical models of parallel computation. In fact, aspects of parallel processing could be incorporated into every computer science course in the curriculum. This text introduces the important principles of parallel processing.

This chapter introduces important terms and concepts. Also, to set the stage for later chapters, a brief history of parallelism is included. The first term to define is "parallelism." When computer scientists talk of parallel processing, they are not talking of processing "two straight lines that never meet." They are discussing several computational activities happening at the same time.

parallelism - several activities happening at the same time.

Parallelism need not refer to computing. For example, writing notes while listening to a lecture are parallel activities.

However, we must be careful because the phrase "at the same time" is imprecise. For example, a uni-processor VAX might appear to be computing at the same time for several time-shared users because the processor handles instructions so fast. Here we have the illusion of instructions being executed simultaneously, i. e., the illusion of parallelism. However, since the processor is executing instructions for only one job at a time, it is not true parallelism.

1.2 Parallelism vs. Concurrency

What is concurrency and how is it different from parallelism?

concurrency - capable of operating at the same time.

It appears that "concurrency" and "parallelism" are synonyms. But there is a subtle difference between the two. We reserve "parallelism" to refer to situations where actions truly happen at the same time, as in four processes (tasks) executing on four processing agents (CPUs) simultaneously. "Concurrency" refers to both situations -- ones which are truly parallel and ones which have the illusion of parallelism, as in four processes (tasks) time-sliced on one processing agent (CPU).

Unfortunately, the two terms "concurrency" and "parallelism" are not used consistently in computer literature. Further, even though many times the term "concurrency" or "concurrent" would be more appropriate, "parallel" is used since it is a current buzz word with more appeal. Therefore, we hear of "parallel programming languages" and "parallel architectures" as opposed to "concurrent programming languages" and "concurrent architectures," which would be more accurate. In this book we will use parallelism for simultaneous actions and concurrency if there is a possibility of illusion of simultaneous actions.

1.3 Levels of Parallelism

In modern computers, parallelism appears at many levels. At the very lowest level, signals travel in parallel along parallel data paths. At a slightly higher level of functionality, functional units such as adders replicate components which operate in parallel for faster performance. Some computers, e. g., the Cray-1, have multiple functional units which allow the operations of addition and multiplication to be done in parallel. Most larger computers overlap I/O operations, e. g., a disk read, for one user with execution of instructions for another user. For faster accesses to memory, some computers use memory interleaving where several banks of memory can be accessed simultaneously or in parallel.

At a higher level of parallelism, some computers such as the Cray Y-MP have multiple Central Processing Units (CPUs) which operate in parallel. At an even higher level of parallelism, one can connect several computers together, for example in a local area network. It is important to understand that parallelism is pervasive through all of computer science and used across many levels of computer technology.

1.4 Why Use Parallelism?

The main reason to use parallelism in a design (hardware or software) is for higher performance or speed. All of today's supercomputers use parallelism extensively to increase performance. Computers have increased in speed to the point where computer circuits are reaching physical limits such as the speed of light. Therefore, in order to improve performance, parallelism must be used.

Raw speed is not the only reason to use parallelism. A computer designer might replicate components to increase reliability. For example, the Space Shuttle's guidance system is composed of three computers that compare their results against each other. The Shuttle can fly with just one of the computers working and the other two as backups. A system of this nature will be more tolerant to errors or faults and is, therefore, called fault tolerant.

Parallelism might be used to decentralize control. Rather than one large computer, a bank might have a network of smaller computers at the main and branch offices. This distributive approach to computing has the advantage of local control by the branch managers as well as gradual degradation of services if a computer should fail.

Parallelism (really concurrency) is an important problem solving paradigm. Nature is parallel. As you talk with your friends, your heart pumps, your lungs breathe, your eyes move and your tongue wags, all in parallel. Many problems are inherently parallel and to solve them sequentially forces us to distort the solution. For example, consider the actions of a crowd of people waiting for and riding on an elevator (activities). Individuals at different floors may push a button (an event) at the same time as someone inside the elevator. To model properly the behavior of the elevator, for example, by a computer program, you must deal with these parallel activities and events.

1.5 Why Study Parallel Processing?

In recent years, parallel processing has revolutionized scientific computing and is beginning to enter the world of every day data processing in the form of distributed databases. Scientific programmers need to understand the principles of parallel processing to effectively program the computers of the future.

All of today's supercomputers rely heavily on parallelism. Parallelism is used at the software level as well as in the architectural design of hardware. The United States' success in building most of the supercomputers has become an issue of national pride and a symbol of technological leadership. The race to hold its lead over the Japanese and European competition is intense. In order to compete in a world economy, countries require innovative scientists, engineers and computer scientists to utilize the supercomputers in an effective manner.

The use of parallel programming has seen a dramatic increase in recent years. Not only may the principles of parallelism be used to increase the performance of hardware, but the ideas of parallelism may also be incorporated into a programming language. Using such a language is called parallel programming.

parallel programming - programming in a language which has explicit parallel (concurrent) constructs or features.

These new parallel programming languages reflect several important developments in computer science. First, the realization that the real world is parallel, especially if the problem has asynchronous events. To properly model the world, we need expressibility beyond what is available in current sequential programming languages. Second, parallelism (or better, concurrency) is a fundamental element of algorithms along with selection, repetition and recursion. The study of parallel algorithms is an interesting topic in its own right. Third, parallelism is an important abstraction for the design of software and understanding of computation. In fact, one can treat all of sequential programming as a special case of parallel programming. Research exploring these issues in parallel programming is very active at universities today.

1.6 What is Parallel Processing?

We have used the phrase "parallel processing" several times in the text so far. Now it is time to distinguish the phrase from several others, notably "parallel programming" and "parallel computing." Unfortunately, the discipline of computer science has problems with the meaning of terms. There are no generally accepted definitions of parallel processing, etc. Therefore, we need to describe how we will use these terms.

parallel processing - all the subject matter where the principles of parallelism are applicable.

Parallel processing is a sub-field of computer science which includes ideas from theoretical computer science, computer architecture, programming languages, algorithms and applications areas such as computer graphics and artificial intelligence.

What is a parallel computer? If we say that a parallel computer is one that uses parallelism then we must include all computers! All computers use parallelism at the bit level for data paths. But clearly, we say some machines are parallel ones and other are not. To distinguish, we need a test. We propose a test that is imprecise, but which is generally close to common usage. If a user can write a program which can determine if a parallel architectural feature is present, we say the parallelism is visible. For example, one cannot write a program to determine the width of a data bus. But, one can write a program to determine if vectorization, as on the Cray-1, is available. (More on vectorization in Chapter 3.)

parallel computing - computing on a computer where the parallelism is "visible" to the applications[1] programmer.

Do we perform parallel programming on a parallel computer? Not necessarily! The programming language may have parallel language features which can be executed on a sequential computer by simulation. Therefore, we would have concurrency, since it would be an illusion of parallelism.

parallel programming - programming in a language which has explicit parallel (concurrent) constructs or features.

The parallel features could be constructs in the language like PAR in the language Occam or could be extensions to a traditional sequential language, e. g., FORTRAN 77 or C, using library routines.

Note, that one could be engaged in parallel computing but not parallel programming, for example, a programmer writing traditional FORTRAN 77 on a Cray-1. Here there are no parallel features in the FORTRAN, but the user still must be aware of the parallel architecture of the machine for effective performance. We will explore this more later.

1.7 Parallel Computing vs. Distributed Computing

In parallel computing, we limit the situations to solving one problem on multiple processing elements, for example, eight processors of a Cray Y-MP. The processors are tightly coupled for fast coordination. In distributed computing, the processors are loosely coupled, usually separated by a distance of many feet or miles to form a network; the coordination is usually for network services, e. g., a database search request; and not the solution to a single problem. A major concern in the study of parallel processing is performance of algorithms, whereas a major concern in the study of distributed computing is dealing with uncertainty. The student of distributed systems has to realize that the exact state of the whole system cannot be instantaneously computed. Every distributed operation may involve network traversals, which cannot be performed instantaneously [Plouzeau, 1992]. Nevertheless, we must be aware that parallel computing/distributed computing is really a continuum of networks based on the average distance between nodes.

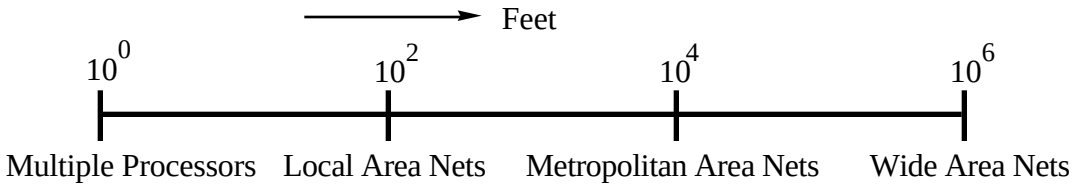


Fig. 1.1 Spectrum of Parallel Computing and Distributed Computing

There is no reason why parallel computing could not be performed on a Local Area Network (LAN) of workstations, e. g., ten SPARC Stations. If the solution to one end-user problem is spread over many workstations, we would say we are engaged in parallel computing. However, the communication over a network is much slower than between tightly coupled processors.

---

[1] We say "applications programmer" because a programmer developing an operating system needs intimate knowledge of the architecture of a machine including low level parallel operations such as overlapping I/O with CPU activity. Similarly with compiler writers, they need intimate knowledge of the machine's architecture, for example, details of the instruction pipeline. Therefore, we restrict our definition to application programmers.

Therefore, in order to be effective, the solution will require very little communication compared to the computation.

1.8 Two Themes of Parallelism: Replication and Pipelining

One way to organize activities in parallel is to replicate agents to do part of the activities. If we want to lay a brick wall faster, we hire more brick layers.

replication - organizing additional agents to perform part of the activity.

This form of parallelism is easy to understand and is widely used. We replicate data wires to allow signals to travel in parallel. We replicate functional units as in the Cray-1. We replicate memory banks in an interleaving scheme. We replicate CPUs as in the Cray Y-MP, which can have up to eight CPUs.

The second way to organize activities in parallel is to specialize the agents into an assembly line. Henry Ford revolutionized industry by using assembly line methods in producing his low-priced Model T Ford (1908). Instead of each worker building a car from scratch, the worker specialized in doing one task very well. The cars ride on a conveyor belt which moves through the factory. Each worker performs his or her task on a car as it moves by. When finished, the worker works on the next car on the conveyor belt. As long as the assembly line moves smoothly with no interruptions, a finished car is rolled off the line every few minutes. For historical reasons, the idea of assembly line is called "pipelining" in computer science.

pipelining - organizing agents in an assembly line where each agent performs a specific part of the activity.

Replication and pipelining (assembly lining) are the two ways of organizing parallel activities. We will see these two themes used over and over again as we study parallel processing.

1.9 Speedup

How much faster are the parallel organizations? If it takes one unit of time for one bricklayer to build a wall, how long for *n* bricklayers? Let us assume the best case where the bricklayers do not interfere with each other, then *n* bricklayers should build the wall in 1/*n* time units.

$$\textbf{speedup} = \frac{\text{the time for the sequential case}}{\text{the time for the parallel case}}$$

Here the sequential case is the time for one bricklayer and the parallel case is the time for *n* bricklayers.

$$\text{speedup}_{\text{replication}} = \frac{1}{\left(\frac{1}{n}\right)} = n$$

Therefore, with our assumptions, the *n* bricklayers are *n* times faster than one.

In general, with *n* replications, we have the potential for a speedup of *n*.

Let us assume a mythical assembly line where four tasks need to be performed to construct a widget.

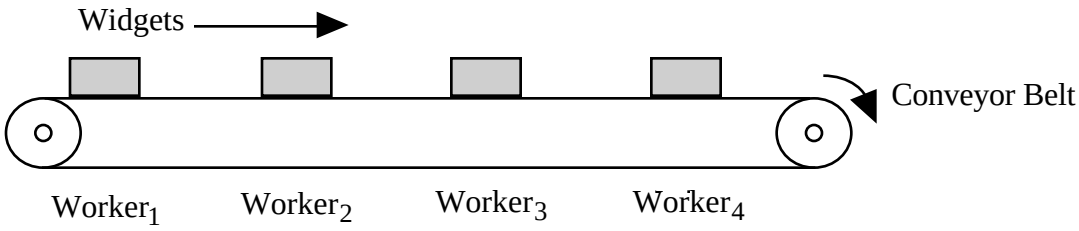


Fig. 1.2 Four Station Assembly Line

Let us further assume that each of the four tasks takes T time units. In the first T time unit, $worker_1$ performs $task_1$ on $widget_1$. In the second T time unit, $worker_1$ performs $task_1$ on the second widget while $worker_2$ performs task2 on $widget_1$ and so on.

After 4T time units, the first widget rolls off the conveyor belt. After that a widget rolls off the conveyor belt every T time unit. Let us assume we want to manufacture 10 widgets, what is the speedup of the assembly line over one worker doing all four tasks?

$$time_{\text{one worker}} = 10 \cdot 4 \cdot T = 40T$$

It takes the worker 4T to build each widget and he has 10 to build.

On the assembly line, it takes 4T for the first widget to ride the length of the conveyor belt and one T for each widget after that.

$$time_{\text{assembly line}} = 4 \cdot T + (10 - 1) \cdot T$$

$$speedup_{\text{assembly line}} = \frac{\text{time for one worker}}{\text{time for assembly line}} = \frac{40 \cdot T}{13 \cdot T} = 3.07$$

For building the ten widgets, the four workers on the assembly line are a little over three times faster than the one worker.

Let us assume then we want to manufacture k widgets:

$$time_{\text{one worker}} = k \cdot 4 \cdot T$$

$$time_{\text{assembly line}} = 4 \cdot T + (k - 1) \cdot T$$

What is the asymptotic behavior of the speedup if we produce many widgets?
Or as k approaches ∞?

Simplifying the two expressions:

$$speedup_{\text{assembly line}} = \frac{k \cdot 4 \cdot T}{4 \cdot T + (k - 1) \cdot T} = \frac{4 \cdot k}{3 + k}$$

Divide the top and bottom by k.

$$\text{speedup}_{\text{assembly line}} = \frac{4}{\frac{3}{k} + 1}$$

$$\lim_{k \to \infty} \frac{4}{\frac{3}{k} + 1} = 4$$

as the $\frac{3}{k}$ term will go to zero as $k \to \infty$.

Therefore, for a four station assembly line (pipeline), the asymptotic speedup is four.

Generalizing: Assuming equal time at each station, *n* stations in an assembly line will have an asymptotic speedup of *n*.

1.10 Examples of Pipelining in Computers

The idea of using an assembly line (pipelining) to organize components in parallel is common in computer architecture. However, the actual use can be disguised among a vast array of detail. This section discusses two uses of pipelining to help us identify pipelining when we see it.

1.10.1 Pipelined Functional Unit

As an example of a pipelined functional unit, we will describe a floating point adder. First some background on floating point numbers. Floating point refers to numbers that are in scientific notation, i. e., with an exponent and a data mantissa[2]. For example,

$$2.3 \times 10^2$$

has an exponent of 2 and a mantissa of 2.3. If we want to add two floating point numbers together, what do we have to do first?

$$\begin{array}{r} 2.3 \times 10^2 \\ \underline{+1.1 \times 10^3} \end{array}$$

First, we must compare the exponents. If equal, we can add the mantissas. If not, we need to adjust the exponent of one of them. Here we adjust the exponent of the first number and add the mantissas.

$$\begin{array}{r} 0.23 \times 10^3 \\ \underline{+1.10 \times 10^3} \\ 1.33 \times 10^3 \end{array}$$

Consider the following pair of numbers where one is negative.

---

[2] Modern computers use bases of 2 or 16. For simplicity, we will assume a base of 10.

$$\begin{array}{r} 1.234 \times 10^3 \\ +(-1.233 \times 10^3) \\ \hline 0.001 \times 10^3 \end{array}$$

After we add the mantissas, we need to adjust the exponent in the result. This is called normalizing. It is important to normalize the result to retain as much precision in the answer as possible.

$$0.001 \times 10^3 = 1.000 \times 10^0$$

Now we have the four tasks we need to form our pipeline.

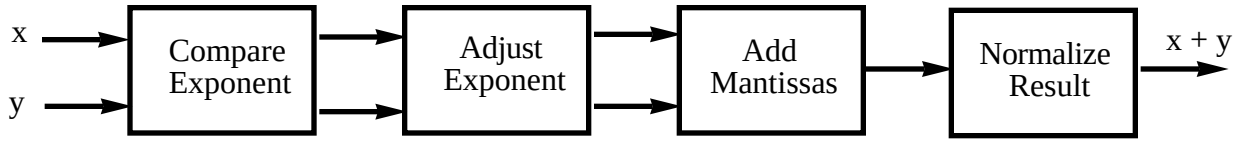


Fig. 1.3 Four Stage Pipeline

Assume each stage takes T units of time to perform its task. Given many pairs of numbers to add, we can feed the first pair to the "Compare Exponent." After T time units, the first pair is passed to the "Adjust Exponent" stage at the same time the pipeline is comparing the exponents of the second pair and so on. Once the pipe is full, we receive a result every T time units.

As an example, let us analyze the speedup for ten pairs of numbers. For the serial case, we need to do all four tasks to add a pair of numbers which takes 4T. The pipeline case takes 4T for the first pair to be added; after the pipe is full, each of the other pairs takes T.

$$\textbf{speedup} = \frac{\text{the time for the sequential case}}{\text{the time for the parallel case}}$$

$$\text{speedup}_{\text{pipelined adder}} = \frac{10 \cdot 4 \cdot T}{4 \cdot T + (10 - 1) \cdot T} = \frac{40}{13} = 3.07$$

The speedup is exactly the same as with the widgets on the conveyor belt of section 1.9. Therefore, we have an asymptotic speedup of 4 with our four stage floating-point adder as long as we keep the pipeline busy. The key question is "How do we keep the pipeline busy?" The answer is with "vectors" and that is the topic of Chapter 3 in which we discuss vector processors such as the Cray-1.

1.10.2 Instruction Lookahead

In order for a typical CPU to execute an instruction, the instruction is first fetched from main memory and then executed. A way to speed up the CPU is to overlap the fetching of the next instruction with the execution of the current instruction. This technique is called *instruction lookahead*. Let us compare the times for three instructions with and without instruction lookahead.

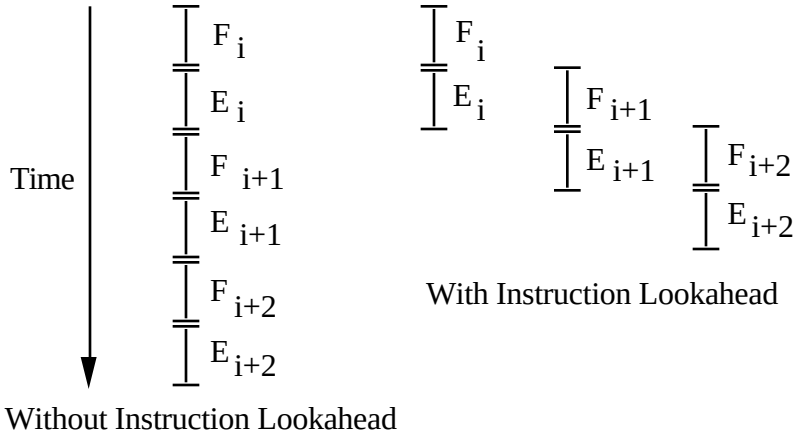


Fig. 1.4 Fetch and Execute Timings of Instruction Lookahead

Without lookahead, the three instructions take 6 time units to complete. With lookahead, they take only 4.

This overlapping of work is none other than a two stage pipeline, where instructions flow by a station to fetch and a station to execute.

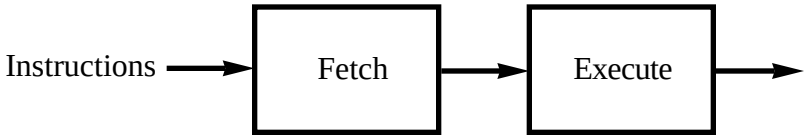


Fig. 1.5 Two Stage Pipeline

If the time to perform each task is equal, the potential speedup is two. Keeping the pipeline busy is easy, since we have millions of instructions to push through the pipeline.

However, there is a complication with the jump and similar instructions. With instruction lookahead, the CPU just fetched the next instruction as indicated by the program counter. If the current instruction is a jump to a new location somewhere in memory, the CPU fetched the wrong next instruction! One solution to this problem is to flush the pipeline and fetch the instruction indicated by the jump. This works. However, flushing the pipeline slows down the machine, since it takes time for the wrong information to flow out of the pipe. Other solutions, allow the pipeline to flow at full rate.

Instruction lookahead is very common. Even the cheaper microprocessors, e. g., the 6502 of the Apple II, used this technique. As the need for faster CPUs arose, an obvious extension to instruction lookahead was to search for ways to increase the number of stages in the instruction fetch-and-execute cycle. The modern RISC (Reduced Instruction Set Computer) microprocessors, e. g., the SPARC, do just that by dividing the fetch-and-execute cycle of the instructions into multiple stages to overlap in a pipeline.

In instruction lookahead, we overlap the fetching of instructions with the execution of the instructions. In the pipelined functional unit, we overlap the operations “compare exponent,” “adjust exponent,” “add mantissas” and “normalize result.” Whenever we observe an overlapping of activity, we should suspect a pipeline at work.

1.11 Classification of Parallel Computers

In the last two decades many parallel computers have been designed and constructed. We would like to classify them into groups by common characteristics. Such a classification scheme would allow us to study one or two representative machines in each group in order to understand the group. Unfortunately, researchers have not found a satisfactory classification scheme to cover all parallel machines. However, one scheme by Flynn [Flynn, 1972] has gained acceptance and is widely used, even though it does not cover all machines.

1.11.1 Flynn's Classification Scheme

Flynn's classification scheme or taxonomy is based on how many streams of instructions and data exist in a machine. A stream in this context simply means a sequence of items (instructions or data) as executed or operated on by a processor. For example, some machines execute a single stream of instructions, while others execute multiple streams. Similarly, some machines reference single streams of data, and others reference many streams of data. Flynn places a machine in one of four classes depending on the existence of single or multiple streams.

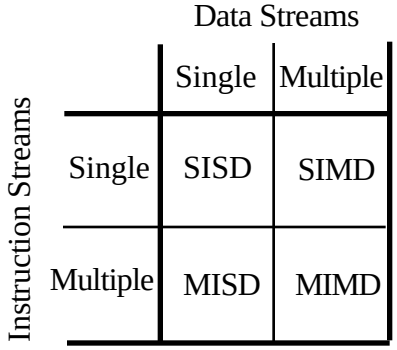


Fig. 1.6 Flynn's Taxonomy

We will discuss each of the four classes and give examples in the following sections.

1.11.2 SISD

Single Instruction stream and Single Data stream (SISD) is the class of traditional serial computers with which we are all familiar, e.g., a DEC VAX or an Apple Macintosh. Machines in this class fetch an instruction from memory then execute it, typically using a data value referenced from memory. Then they fetch another instruction from memory and so on. This class is also known as the von Neumann architecture, after the work of John von Neumann in the late 1940s and early 1950s. The computer industry has had over forty years of experience with this class and much of our programming languages, e. g., FORTRAN and C, compilers, operating systems and programming methodology, are based on this class. Therefore, when we investigate some of the newer parallel machines, or the so-called "Non Vons" (Non von Neumann architectures), we must remember that our bias is towards this long history of serial-based software and hardware.

All the serial computers are in the SISD class. Further, researchers place some parallel computers in this class. For example, the vector processors, such as the Cray-1, are considered SISD machines because even though some instructions operate on vectors of data values, they still have only one stream of instructions. Also, machines that use instruction lookahead, i. e., where the fetch of the next instruction is overlapped with the execution of the current instruction, are

classified as SISD. Even though there is more than one instruction being handled at a time, there is still only one instruction stream.

1.11.3 SIMD

The Single Instruction stream and Multiple Data Stream (SIMD) class includes computers where one instructional unit issues instructions to multiple processing elements (PEs). Since each PE operates on its own local data, there are multiple data streams. Typically, the instruction unit issues the same instruction to all the PEs in lockstep fashion. For example, all the PEs do an ADD instruction, then a STORE instruction, etc. Such a machine is analogous to a sergeant barking out orders to a precision drill team.

The ILLIAC IV is a good illustration of an SIMD class machine. One instruction unit issues the same instructions in lockstep manner to 64 PEs. Each PE has 2K words of local memory to load, manipulate and store data. The 64 PEs are connected in a two-dimensional mesh, eight on a side, where nearest neighbors can send and receive messages. The connections to the processors on the top edge wrap around to the bottom, and the connections on the left side wrap around to the right. (The wraparounds are not shown in Figure 1.7). This forms a structure called a torus.

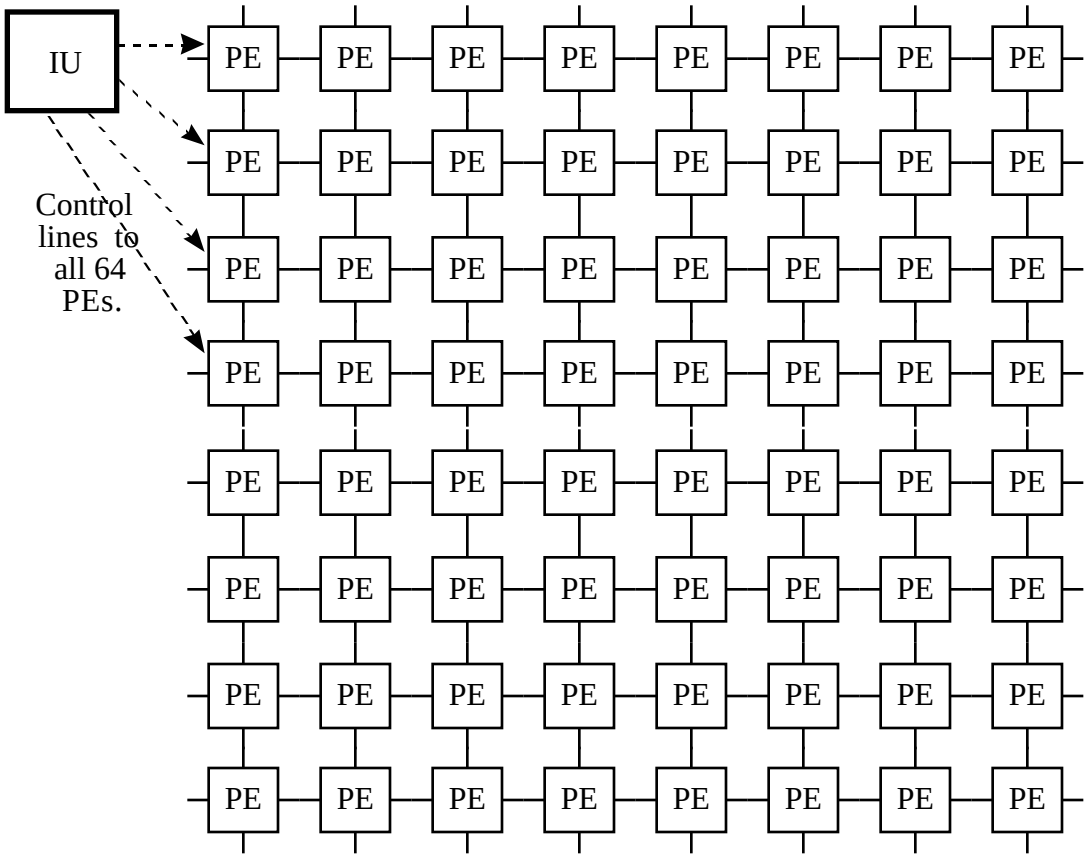


Fig. 1.7 ILLIAC IV with One Instruction Unit (IU) and 64 PEs

Since each PE can send in four directions, the connections are usually labeled with the compass directions -- North, East, South and West. This is also referred to as a NEWS connection network. With one instruction, the 64 PEs can each pass a message in one direction, e. g., North.

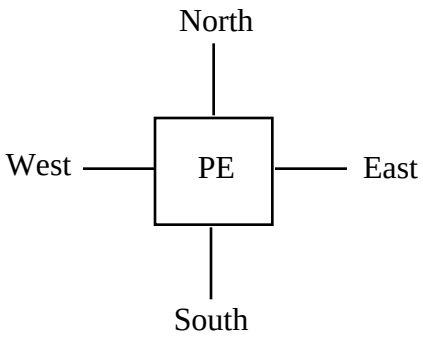


Fig. 1.8 Labeling in a NEWS Mesh

The ILLIAC IV was not designed to be a general purpose computer, but rather a special purpose computer for solving partial differential equations. Such problems, e. g., weather forecasting, require a lot of data in a three-dimensional space. A typical solution divides the space into 64 sections and places a section on each PE. That way, the solutions for the 64 sections are computed in parallel. Whenever a PE needs data from a neighboring section, a communication must be initiated on the connection network.

Important SIMD machines are the ILLIAC IV, the ICL DAP and Thinking Machines Corporation's Connection Machine.

1.11.4 MISD

The Multiple Instruction stream and Single Data stream (MISD) class is void of machines.

1.11.5 MIMD

Computers in the Multiple Instruction stream and Multiple Data Stream (MIMD) class have multiple instruction units issuing instructions to multiple processing units. Within this rich class of machines, there are two important sub classes -- shared memory and message passing.

1.11.6 MIMD Shared Memory

With shared memory MIMD, any of the processors, which includes an instruction unit and an arithmetic unit, may read from or write to a shared memory address space. Although the name given to this class of machines is "shared memory," it is more properly "shared memory address space." The processors do not need to share a single memory unit, for example, with physical connections, but need only to share the same address space, which could be across many memory units.

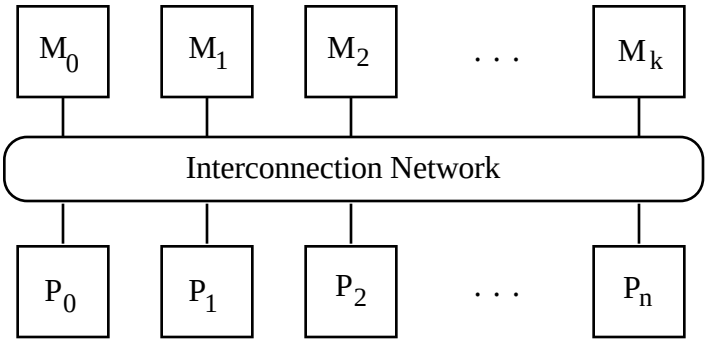


Fig. 1.9 Shared Memory MIMD

In this model, the processors are connected to the memory units by an interconnection network, which may take many forms, depending on the machine. The interconnection network can be fixed, as in a ring or a mesh; or can be switched, as in a crossbar switch. The time for a memory reference through the interconnection network is critical to the performance of the machine.

This model is analogous to a committee which uses a common blackboard for all its communication. Any committee member can read any part of the blackboard, but only one person can write on a particular section of the board. In our shared memory model, we may have memory conflicts when two processors try to write to the same memory bank at the same time. Also, processors may interfere with each other when writing in the same shared memory cell, and corrupt the computation. To insure that interference is not a problem, the hardware must provide locks, semaphores, or some other synchronization mechanism to guarantee that only one processor updates a shared memory cell at a time (the mutual exclusion principle).

We will briefly describe three commercial shared memory machines to show the wide range of approaches. You should especially notice the variety in the interconnection networks.

The Cray X-MP supercomputer may have up to four processors, each with four ports into a common memory accessing up to 64 memory banks.

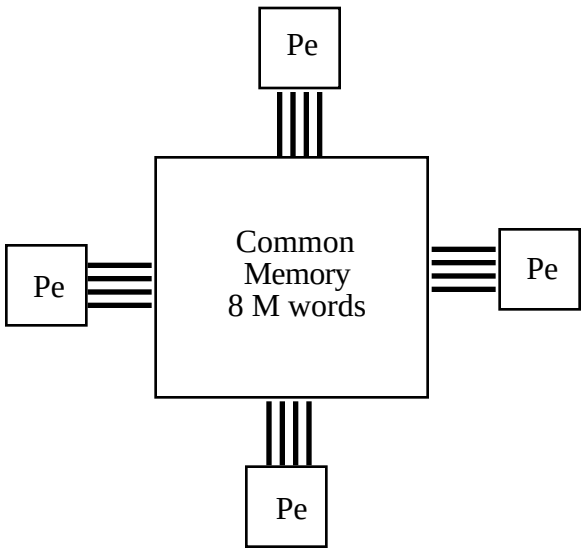


Fig. 1.10 Each Processor of the Cray X-MP/48 Has Four Ports to a Common Memory

The Alliant FX/8 minisupercomputer may have up to eight computational elements (CEs) sharing a common memory. The CEs are connected by way of a crossbar switch to two 64 Kbyte caches with a bandwidth of 376 Mbytes/sec. The caches in turn access the shared memory via a 188 Mbytes/sec memory bus [Hockney, 1988].

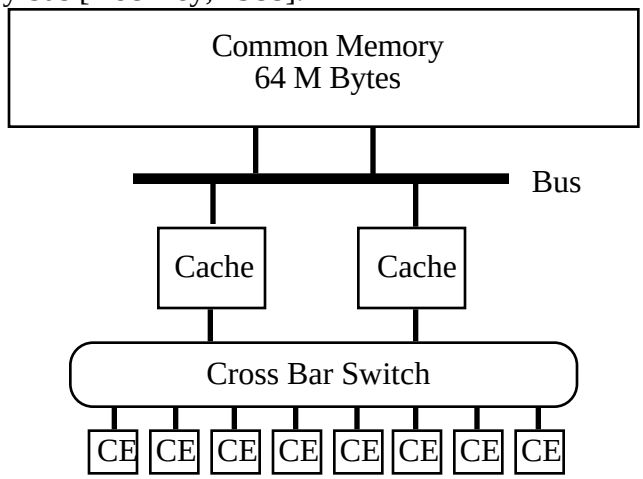


Fig. 1.11 The Alliant FX/8 with Eight Processors (CEs) Which Share Memory

The Bolt, Beranek and Newman (BBN) Butterfly has a distributed memory approach, but retains the shared memory address space. Up to 256 processors are connected to memory units by way of a multi-stage switch called a butterfly switch.

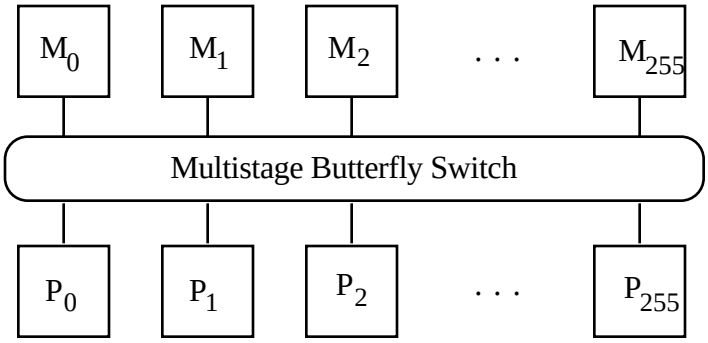


Fig. 1.12 The BBN Butterfly Allows Any PE to Be Switched to Any Memory Unit.

All the shared memory MIMD machines share a common advantage. The shared memory model provides a uniform view of storage to the programmer. This matches the traditional view of storage with which most programmers are familiar. Therefore, machines using the shared memory model are easier to program than, for example, the message passing model.

A common disadvantage is the possibility of “hot spots” in the common memory, where many processors try to write to the same memory cell. Because the processors must wait until the memory cell is available, the hot spots can significantly degrade performance. Another disadvantage is that the programmer, the compiler, or the operating system must decide how to allocate the program across the many processors. Effective processor allocation which minimizes processor idle time is non trivial.

1.11.7 MIMD Message Passing

In the message passing MIMD model, all the processors have their own local memory. In order to communicate, the processors send messages to each other by way of an interconnection network.

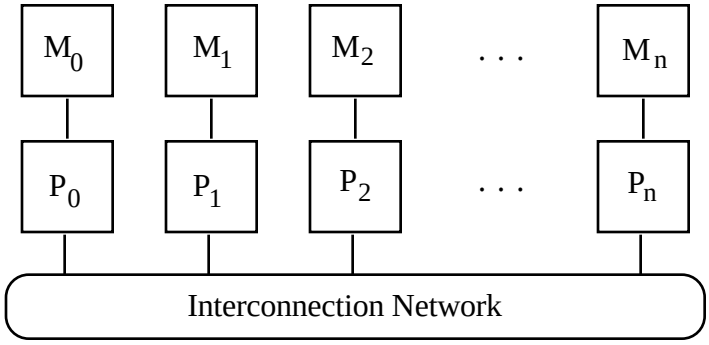


Fig. 1.13 Message Passing MIMD

As with the shared memory model, the interconnection network may take many different forms. A popular interconnection network for message passing machines is the k-dimensional binary hypercube, where two processors are fixed in each of the k dimensions. For example, in a 3-dimensional hypercube, the processors are at the corner of a cube, as shown in Fig. 1.14.

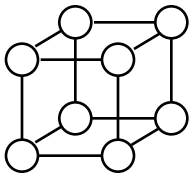


Fig. 1.14 A 3-Dimensional Hypercube

The message passing model is analogous to a committee where the members only communicate by writing notes to each other. The routing strategy and the speed of a message are critical issues in the performance of this style of machine. If the communication is two orders of magnitude slower than the arithmetic, as it was in early message passing machines, the machine acts much like a committee that functions by sending messages through campus mail!

The message passing approach has several advantages. Because there is no shared memory, the problems of interference and memory conflicts of the shared memory model disappear. Therefore, memory access is typically faster and the overhead associated with the semaphores, etc. disappears.

The message passing mechanism serves two purposes -- first, as communication to pass data values between processors; and second, as a synchronization mechanism in parallel algorithms. The synchronizing event is the arrival of the sent message.

Since the processors are not sharing a common memory, the message passing approach has the promise of being more scalable to very large numbers of processors, e. g., in the thousands, than shared memory systems. For example, if one tries to add more processors to the Cray X-MP, the number of possible ports to the common memory (See Fig. 1.10) is the limiting factor. This approach probably can't extend beyond eight or sixteen processors. Whereas the Cray machines use only a small number of high-powered -- and, therefore, very expensive -- processors, the

message passing approach may achieve reasonable performance by using many cheap off-the-shelf microprocessors. For example, the Intel iPSC hypercube uses 128 Intel 80286 microprocessor chips, the same microprocessor used in the IBM PC AT personal computers.

The main disadvantage of the message passing model is the extra burden placed on the programmer. Not only must the programmer partition the program across many processors, he or she must distribute the data as well. Programming a message passing machine requires programmers to rethink their algorithms in order to effectively use the machine.

The Intel iPSC hypercube machine has 128 processors ($128 = 2^7$ Therefore, it is a 7-dimensional hypercube.) Another company, nCUBE, is marketing hypercube machines with up to 8,192 processors ($8192 = 2^{13}$ Therefore, a 13-dimensional hypercube).

Although most commercial computers fall into one of the three classes of Flynn's taxonomy -- namely SISD, SIMD, or MIMD -- several interesting designs do not. Some, like the ICL DAP, seem to fit into several categories. Others, like the dataflow machines and the reduction machines, which we will study later, don't fit well into any category.

1.12 A Brief History of Parallelism

This section will present a brief history of parallelism and its use in computers. This history covers developments up to about 1985. Developments after that will be discussed in later chapters. For a more detailed history of parallel computers see Hockney and Jesshope's book *Parallel Computers 2* [Hockney, 1988]. We will focus on important firsts in the field as well as discuss general trends. Even though parallel processing is a new sub-discipline of computer science, you will discover that many of the ideas of parallelism have been around for a long time.

1.12.1 Parallel or Concurrent Operations

Many computer designs achieve higher performance by operating various parts of the computer system concurrently. Even Charles Babbage, considered the Father of Computing, utilized parallelism in the design of his Analytical Engine. His Analytical Engine was designed in the 1820s and 1830s and was to be a mechanical device operated by hand crank. The arithmetic unit was designed to perform 50-digit calculations at the following speeds: add or subtract in one second; multiply or divide in one minute. To achieve such high speeds with mechanical parts, Babbage devised, after years of work, a parallel addition algorithm with anticipatory carry logic. [Kuck, 1978]. At a higher level of parallelism, the Analytical Engine performed indexing arithmetic on an index register for loop counting in parallel, with its arithmetic unit [Kuck, 1978]. Unfortunately, Babbage was never able to build his Analytical Engine because technology took over one hundred years to advance to the point where his design could be implemented.

After Babbage, the world had to wait over a hundred years for significant activity in the design of computers. In the 1940s, the first modern computers were built, and soon after parallel operations were used to increase performance. With the availability of static random-access memories from which all the bits of a word could be read in parallel, bit-parallel arithmetic became a practical part of computer design. In 1953, the International Business Machines (IBM) 701 was the first commercial computer with parallel arithmetic. In 1955, the designers of the IBM 704 added floating point hardware to the parallel arithmetic to make the IBM 704 the fastest machine then in production. In 1958, six I/O channels were added to the IBM 704 and it was renamed the IBM 709. The I/O channels allowed the I/O to be overlapped with the execution of the CPU and is an early case of multiprogramming.

In 1955, IBM embarked on its Project Stretch to specifically achieve the highest performance possible within limits of time and resources, i.e., stretch the limits -- hence the project's name [Buchholz, 1962]. An overall performance of one hundred times the 704, the fastest machine then in the world, was set as a target. The purpose of setting so ambitious a goal was to stimulate innovation in all aspects of computer design. During the lifetime of the project, the technological advancements of electrical circuits and memory were projected to be only six to ten times faster in speed than 1954 technology. Therefore, in order to attain their stated goal, the designers had to provide for parallel operation in the system wherever possible. The Stretch computer was the first to use instruction lookahead, i.e., the fetch and updating of the next instruction is overlapped in time with the data fetch and execution of the current instruction.

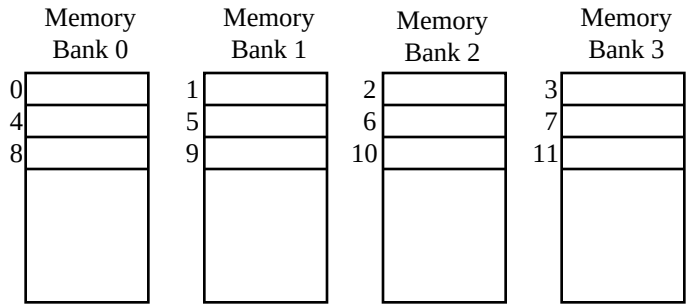


Fig. 1.15 Interleaving of Addresses on the IBM Stretch

Also, Stretch was the first computer to use parallelism in memory by using a memory interleaving scheme to speed up accesses to memory [Hockney, 1988]. To achieve a high degree of overlap in memory accesses, addresses were interleaved across four memory units so that four consecutive word addresses lie in the four different memory units. This four-way interleaving scheme was able to achieve rates of accessing one full word every 0.5 microseconds for 2 microsecond technology. The Stretch computer, renamed the IBM 7030, was first delivered to Los Alamos Scientific Laboratory in 1961 [Buchholz, 1962].

By the end of the 1950s, most of the fundamental ideas of parallel or concurrent operation, i. e., parallel arithmetic, pipelining of instructions, overlapping I/O with CPU execution and interleaved memory, had been incorporated into a commercial machine. Since 1960, large scale serial computers have extensively used these ideas.

In 1964, Control Data Corporation (CDC) introduced its CDC 6600, designed by Seymour Cray [Thornton, 1970]. The CDC 6600 was the fastest machine in its day, due partly to a new use of parallelism in the CPU design. The CDC 6600's CPU contained ten independent functional units -- floating-point adder, two floating-point multipliers, floating-point divider, fixed-point adder, two incrementors, Boolean, shift and branch. The functional units were independent in the sense that more than one of them could be operating simultaneously. From an Instruction Stack of up to 32 instructions, several instructions could be selected and issued to the ten functional units. Since the goal was to maintain the same order of execution as the serial computer program, i. e., as a single instructional stream, a sophisticated control mechanism was required. This control mechanism, called the Scoreboard, reserved and controlled the twenty-four registers and the ten functional units. Since many instructions could be issued at the same time, the Scoreboard had to block instructions which required the results of other instructions still in progress or not even started (in order to follow the order imposed by the data dependencies). Also, the Scoreboard had to resolve conflicts in multiple instructions simultaneously, using the same resource, i. e., a functional unit or register (called resource management). The Scoreboard required a sophisticated set of flags to guarantee that instructions were performed in the proper order and to resolve the

resource conflicts. During a typical program, two or three of the ten functional units could be kept busy. The FORTRAN compiler writers found generating high quality code for the 6600 to be quite a challenge. It was several years before CDC released a version of the FORTRAN compiler which effectively utilized the machine's multiple functional units.

In this section, we saw that parallelism was used to increase the performance of computers while maintaining the serial program model. This is reiterated by Seymour Cray in the Forward of Thornton's book [Thornton, 1970], "This book describes one of the early machines attempting to explore parallelism in electrical structure without abandoning the serial structure of the computer programs." Even though all the computers in this section used parallel operations extensively, we still would classify them as serial and as SISD in Flynn's Taxonomy. In the next section, we will continue by examining the history of parallel computers.

1.12.2 Parallel Computers

Of the hundreds of parallel machines[3] proposed in the last two decades, many were the object of university and industrial research projects. We will limit our discussion to important commercial parallel computers and will mention a research project only when appropriate. We will focus on the first machine in a class, e. g., vector processor, SIMD and MIMD, and its influence on later machines.

Vector Processors

In the early 1970s, computer architects of high performance machines (supercomputers) turned to pipelined vector processors. A vector processor has special machine instructions to perform arithmetic on two vectors, element by element, to produce an output vector. Thus, if **A**, **B** and **C** are vectors, each with N elements, a vector processor can perform the operation

$$\mathbf{C} := \mathbf{A} + \mathbf{B}$$

much faster than a serial machine which would require executing a loop of many instructions N times.

The first vector processor was the CDC STAR 100 which was conceived as a processor with an instruction set based on Iverson's (1962) vector based APL language [Hockney, 1988]. Using pipelined arithmetic functional units, the STAR 100 obtained high performance on long vectors of data which are common to many scientific problems. The machine was operational in 1973. In 1979, the STAR 100 was completely re-engineered in Large Scale Integration (LSI) and renamed the CDC Cyber 203E. In 1981, after further enhancements, the machine was renamed the CDC 205.

In 1972, Seymour Cray left Control Data Corporation to start his own company, Cray Research, Inc., with the aim of producing the fastest computer in the world. In the extraordinarily short time of four years, the Cray-1 computer was designed and built (1976). The Cray-1 follows the evolutionary trend of Seymour Cray's CDC 6600 and his upgrade, the CDC 7600. Though Seymour Cray did not design the CDC STAR 100, he liked the idea of the pipelined vector processor, introduced by the CDC STAR 100, and incorporated it into the Cray-1.

By the early 1980s, supercomputer designers, discovering that not much more performance could be achieved by serial vector processors, incorporated more parallelism by using multiple

---

[3] In section 1.6, we defined a "parallel computer" as one where the parallelism is "visible" to the applications programmer.

processors. For example, in 1982 Cray Research introduced the Cray X-MP which consisted of two Cray-1-style pipelined vector processors sharing a common memory. Later, the Cray X-MP, was upgraded to four processors. This multiple processor approach continued in the Cray line with the Y-MP (1987), which has up to sixteen processors.

In 1989, Seymour Cray left Cray Research to start another start-up company, Cray Computer Co., where he pursues the gallium arsenide (GaAs) technology integral to the Cray-3. Cray has demonstrated that gallium arsenide integrated circuits can be fabricated to execute three times faster than silicon. The Cray-3 is to be delivered in 1992.

SIMD Computers

In the 1960s, The ILLIAC IV, the first SIMD machine, was designed at the University of Illinois (It was the fourth major computer designed at the University -- hence the IV.). The ILLIAC IV consisted of a two-dimensional array of sixty-four processors under control of a single instruction stream in a central instructional unit. Each processor contained a fast 64-bit floating-point unit and its own local memory. Contrary to the evolutionary development of the serial computer to the pipelined vector computer, the ILLIAC IV concept was a radical change in thinking in computer architecture and had a substantial influence on computer science research as well as computer design.

Though the ILLIAC IV was delivered to NASA Ames Research Center in 1972, it was not until 1975 that usable service could be offered. The machine, due to the pioneering of the new and faster emitter-coupled logic as well as the low level of integration (only 7 gates per integrated chip), was plagued with reliability problems. In order to improve the reliability, the original clock period was lengthened from 40 nanoseconds to 80 nanoseconds, with the consequence of a significantly slower machine. The ILLIAC IV was, like Babbage's Analytical Engine, too ambitious for the technology of its time [Falk, 1976].

The influence of the ILLIAC IV was profound and spawned many research projects in the 1970s and 1980s. Several of the research projects have resulted in commercial SIMD machines. In 1980, the British company ICL delivered its Distributed Array Processor (DAP). It was comprised of a 64 by 64 array of processors and formed a memory module in a host ICL 2980. As with the connections on the ILLIAC IV, the 4096 processors were connected in a two-dimensional network with nearest-neighbor connections. Unlike the expensive floating point arithmetic units of the ILLIAC IV, the DAP contained 4096 simple one-bit processors which performed arithmetic on 4096 numbers in parallel in a bit-serial fashion. Having a massive number of simple processors is very effective on certain problems. For example, the DAP was especially suited to special application areas such as image processing where a processor could be assigned a natural partition of the problem, in the image processing case, a region of the image.

Another important SIMD machine, the Connection Machine, is also based on this idea of a massive number of simple processors (65,536). Originally, the Connection Machine was developed in the early 1980s by Danny Hillis [Hillis, 1985] and others at MIT for use as a parallel artificial intelligence (AI) engine. Later, after users found it to be a serious numerical engine, its focus and development have been towards scientific computations and not AI. The Connection Machine has been implemented by Thinking Machines Corporation(TMC), a company co-founded by Hillis, and is called the CM-1. The CM-1 comprises 65,536 one-bit serial processors each with 4096 bits of memory. Sixteen processors were implemented on a custom Very Large Scale Integrated (VLSI) chip. On the chip, the processors are connected by a two-dimensional mesh such as used in the ILLIAC IV and DAP. The 4096 chips are connected in a twelve-dimensional hypercube ($2^{12} = 4096$). The first CM-1 was delivered in 1986. Two years later, TMC introduced the CM-2 which has more memory per 1-bit processor (64 K bits) and groups of 32

processors share a fast floating-point chip for a total of 2096 such chips. The CM-2 can be viewed as a machine with 2048 floating-point processors.

MIMD Computers

Because of the advancements of VLSI technology and the cheap microprocessor, the 1980s became the decade of the MIMD computer. MIMD computers are controlled by more than one instruction stream. We limit the term to tightly coupled systems in which the instruction streams cooperate on the solution of one problem. Therefore, we exclude loosely coupled collections of workstations in a local area network. Also, we exclude multiprocessor configurations, such as a four processor IBM 3081, where the user is unable to utilize more than one processor to solve a problem. We have already met several small scale MIMD examples, such as the four CPU Cray X-MP. However, we usually think of MIMD computers as a large collection of cheap minicomputers or microprocessors connected together by an interconnection network, e. g., the Intel iPSC Hypercube with 128 Intel 80286 microprocessors.

An important early research example of MIMD computing was the C.mmp computer at Carnegie Mellon University [Wulf, 1972]. Completed in 1975, the C.mmp was comprised of 16 DEC PDP-11 minicomputers connected to 16 memory modules by a 16x16 crossbar switch. The crossbar switch provided a direct connection between every minicomputer and every memory module. The C.mmp, a shared memory MIMD system, provided valuable early experience in the issues of programming an MIMD system.

A second important research project in MIMD computing was the Cm*, also developed at Carnegie Mellon University (1977) [Swan, 1977] [Fuller, 1978]. The Cm* was a shared memory MIMD system with "compute modules" consisting of a DEC LSI-11 microprocessor, 64 K bytes memory and, perhaps, other peripherals. The compute modules were connected by a hierarchical bus structure. Up to 14 compute modules could be attached to a bus to form a tightly coupled "cluster". The system is constructed by linking clusters together by an intercluster bus. All the compute modules shared the same virtual address space of 256 Mbytes, but there was a penalty for accessing another compute module within the cluster (9 microseconds compared to 3 microseconds for a local access), and a stiffer penalty for accessing a computer module in another cluster (26 microseconds). In 1980, a five-cluster Cm* containing 50 compute modules was operational [Satanarayanan, 1980] [Jones, 1980].

The first commercially available MIMD computer was the Denelcor HEP designed by Burton Smith [Smith, 1978]. Delivered in 1982, the HEP was not a collection of microprocessors as we described above. In a clever scheme, multiple instruction streams were passed through a single instruction processing unit. Up to 50 user instruction streams could be created, for example, from several user programs or all from one program. Instructions are taken in turn from each stream and placed into an eight-stage instruction pipeline. Since the instructions are taken from different independent user streams, the problems caused by the data dependencies, which we saw in the CDC 6600's Scoreboard, is drastically reduced. The method of implementing MIMD computing on the HEP (called multi-threading) is much more flexible than that used in designs based on a fixed number of microcomputers. The number of instruction streams can be changed from one problem to the next by appropriate programming, thus the number of streams can be chosen to suit the problem being solved. The multi-thread approach is being used in Burton Smith's Tera Computer [Alverson, 1990] still in development.

An important commercial shared memory MIMD machine was the BBN Butterfly (1985) [Hockney, 1988]. It contained up to 256 processors based on the Motorola 68000 microprocessor with up to 256 one Mbyte memory modules. The processors were connected to the memory modules by a multi-stage switch called a butterfly switch. BBN upgraded the Butterfly architecture to use Motorola 68020s, then, later, Motorola 88000 microprocessors.

The first practical message passing MIMD machine was the Cosmic Cube built at Cal Tech by Geoffrey Fox and Charles Seitz in 1984 [Seitz, 1985]. Each node of the Cosmic Cube was built from an Intel 8086 microprocessor with an Intel 8087 floating point coprocessor and 128 Kbytes of memory. The Cosmic Cube had 64 nodes and, therefore, was a six-dimensional hypercube. The Cal Tech group was able to demonstrate the feasibility of message passing MIMD computing by porting many scientific programs to the Cosmic Cube. The Cosmic Cube's commercial derivative, the Intel iPSC (intel Personal SuperComputer) (1985) used faster chips, the Intel 80286 with 80287 coprocessor, and was expanded to 128 nodes. In 1988, the second generation hypercube Intel iPSC/2 replaced the slow store-and-forward software approach to communication with high speed circuit switching in hardware. In 1990, the Intel iPSC/860 used the high performance RISC chip, the Intel i860.

The trends of the 1990s appear to be massively parallel MIMD machines and heterogeneous computing. By massively parallel (MP) we usually mean more than 1000 processors. In 1991, Thinking Machines Corporation (TMC), Alliant, Intel Supercomputer, and Kendall Square Research all introduced MP supercomputers. These MP machines are based on the latest high performance RISC microprocessors, for example, TMC's CM-5 is based on the SPARC microprocessor. In the next decade, the MP computers promise to surpass the performance of the traditional supercomputers (small number of vector processors as typified by the Cray Y-MP). Even Cray research, the bastion of vector supercomputers, is designing an MP machine based on Digital Equipment Corporation's (DEC) Alpha microprocessor. Each Alpha microprocessor is reported to have the performance of a 1976 Cray-1.

Heterogeneous computing refers to connecting several dissimilar supercomputers together by way of a very high speed network, e. g., the High-Performance Parallel Interface (HIPPI) which transfers data at 800 Mbits per second (80 times the speed of ethernet), to solve a single problem. With heterogeneous computing, the user can increase the computing power available to solve a single problem without having to buy a new and larger machine.

1.12.3 Parallel Programming[4]

APL

In 1958, Ken Iverson of IBM developed a hardware description language called APL (A Programming Language). Using his APL to describe the IBM 360 architecture, Iverson was the first to describe a computer by a formal notation. His APL reflects the view that modern computers move vectors of information around. Further, with APL, Iverson could describe the low level parallelism in the IBM 360. In 1962, IBM developed a subset of Iverson's original APL to create the general purpose programming language APL we know today. The innovations in APL, especially its rich set of vector operations, have had profound influence on computer architecture and programming languages. For example, designers of the first pipelined vector computer, the CDC STAR 100, (1973) based its vector instruction set on APL.

Distributed Computing

Operating systems were the first significant examples of concurrent programs[5] and remain among the most interesting. With the advent of independent I/O device controllers in the 1960s, it was natural to organize an operating system as a concurrent program, with processes managing

---

[4] In Section 1.5 we defined parallel programming as programming in a language which has explicit parallel (concurrent) constructs or features.

[5] See Section 1.2 on the differences between concurrency and parallelism.

devices and execution of user tasks. Later, with the desire to handle multiple users at the same time, operating systems designers expanded the concept of process to include user processes. These operating systems were implemented on a single CPU system by multiprogramming, i. e., processes were executed one at a time in an interleaved manner (time sharing).

In the 1970s, operating systems were expanded to handle multiple CPUs. Since a user could utilize only one CPU to solve a problem, we do not consider these multiprocessor systems to be parallel computers. However, researchers obtained valuable experience in designing complex concurrent programs.

In the mid 1970s, the United States Department of Defense (DOD) was concerned with the tremendous cost of programming embedded computer systems in assembly language. Embedded computer systems are ones that are embedded in a larger system such as an aircraft or a weapons system. Whenever a system was upgraded by a new computer, all the assembly programs had to be rewritten in its new assembly language. To reduce the cost, the DOD proposed and mandated a new high level language, Ada[6], in which to program embedded systems (1983). Because of the need to handle multiple CPUs in embedded systems, Ada has language constructs to facilitate the development of concurrent programs. Because it was mandated by the DOD as the language to use in defense contracts, Ada has had a major impact on the computer industry.

At the same time, developers of non-military concurrent systems also moved away from assembly languages to high level languages, e. g., PL/1 or C, with special library routines to access the concurrency facilities of the operating system. Programming in a high level language, a programmer's productivity was higher. Also, a company could reclaim much of its investment when porting the high level code to a new hardware platform.

In the early 1980s, researchers were developing operating systems for local area networks (LAN) where programs are spread over many workstations. The result was an extensive effort in the research area called distributed computing [Andrews, 1991]. Some researchers [Bal, 1989] argued that programming languages especially designed for distributed programming were better than serial languages with library routines that invoke the operating systems primitives, e. g., C. As a consequence, many new distributed languages, e. g., SR [Andrews, 1982], were designed in the 1980s.

By 1985, the operating systems community researching distributed computing realized that their research overlapped ideas with the parallel programming of the scientific computation community. Today, the distributed computing and the scientific computation communities are still distinct, e. g., each has its own journals and conferences. However, in the future, distributed computing and parallel computing will probably blend into one.

Scientific Computing

Parallel programming in the scientific computation arena started in the late 1960s with the ILLIAC IV project at the University of Illinois. This radical approach to computing, where one instruction stream controls 64 processors in lockstep manner (SIMD), demanded novel approaches to programming and programming languages. Four computer languages that could express the parallelism of the machine were developed: ALGOL-like TRANQUIL [Abel, 1969], GLYPNIR [Lawrie, 1975], the Pascal-like ACTUS [Perrott, 1978] and CFD FORTRAN [Stevens, 1975]. Except for FORTRAN, the other languages have long since disappeared from use.

FORTRAN has been and still is the predominant programming language for scientific computations. Unfortunately, with its heavy dependence on the architectural features of the

[6] Ada is a trademark of the U. S. Department of Defense.

ILLIAC IV, CFD FORTRAN started a trend still continued in supercomputers today. In order to squeeze the most performance out of a supercomputer, FORTRAN compilers have relied heavily on machine dependent features. Thus, FORTRAN programs developed on one supercomputer will have dismal performance on another or not run at all. For example, vendors have added non-standard, and, therefore, non portable, features for vectorization and message passing to their versions of FORTRAN. The new FORTRAN 90 Standard [Metcalf, 1990], which contains parallel programming constructs, and the Parallel Computing Forum's (PCF) FORTRAN extensions standards [PCF, 1991] promise to alleviate much of this non-portability. The High Performance FORTRAN Forum (HPFF) continues this effort.

CSP

In 1978, Tony Hoare of the operating systems research community published an influential paper on Communicating Sequential Processes (CSP) [Hoare, 1978]. For over a decade the operating systems research community had struggled with the problems of using shared variables between processes. With CSP, Hoare proposed a mathematical model of concurrency based on message passing which eliminated shared variables and, therefore, the problems associated with them. In 1983, a British company, Inmos Limited, developed the concurrent programming language Occam [May, 1983] based on Hoare's CSP model. Occam[7] in association with Inmos' Transputer microprocessor chip allows the easy construction of message passing MIMD systems specifically designed for an application, for example, fingerprint recognition.

Linda

Several researchers have proposed new coordination models to relieve the programmer of worrying about the specifics of how processes coordinate or communicate as one must do in Occam or Ada. Linda [Gelernter, 1985] is one model which is based on a Tuple space where any process can add a Tuple to the space and any process can read a tuple (analogous to a bulletin board). Linda is not a programming language but a coordination model which is added to existing languages such as FORTRAN and C. Linda offers high expressibility and convenience to the programmer as well as portability, i. e., a program written in Linda will execute on many different parallel architectures. Such portability means reusability and, therefore, reduced cost.

Functional Languages

Some researchers believe that imperative (algorithmic) languages, for example, C, FORTRAN, Ada and Occam, are not the best ones for dealing with parallelism. In 1978, John Backus [Backus, 1978] (developer of FORTRAN in the 1950s) claimed in his Turing Award Lecture that "one-word-at-a-time" imperative languages are inherently sequential. Backus explained that imperative languages are based on von Neumann architectures which fetch one computer word at a time for instruction or data. He dubbed this the "von Neumann Bottleneck." Backus proposed an alternative -- functional languages which contain inherent parallelism. His paper stimulated intense research efforts in functional languages, e. g., FP [Backus, 1978], Miranda [Turner, 1985] and SISAL [Feo, 1990], and computer architectures to realize their inherent parallelism, i. e., dataflow machines and reduction machines.

Logic and Object-Oriented Languages

Agreeing with Backus that imperative languages are inappropriate for parallelism, other researchers are investigating parallel versions of logic and object-oriented languages. In logic languages, different parts of a proof procedure can be worked on in parallel, as exemplified by Concurrent PROLOG [Shapiro, 1986] and PARLOG [Clark, 1986]. Parallelism can also be

---

[7] Occam and Transputer are registered trademarks of Inmos Limited.

introduced into object-oriented languages by making objects active, as done in Emerald [Black, 1986].

1.12.4 Theory of Parallel Algorithms

Research in parallel algorithms has been in two different arenas: scientific computing and the theory of algorithms. Researchers in scientific computing are interested in the design and analysis of numerical algorithms [Cheney, 1985] [Press, 1986]. Researchers in the theory of algorithms have developed theoretical models of parallelism and analyzed the behavior of parallel algorithms based on these models [Corman, 1990].

Scientific Computing

Before the advent of the electronic digital computer (pre-1945), numerical methods were highly parallel and designed to be executed by a team of human analysts working together on a large tableau with many mechanical calculators. Ironically, the algorithms were "sequentialized" for use on early digital computers [Carey, 1989]. The early days at Los Alamos before the electronic computer while he was working on the Atomic Bomb are described by Richard P. Feynman in his delightful book "*Surely You're Joking Mr. Feynman!*"[8] Feynman explains how the calculations were cycled many times through a pipeline of several individuals rather than every single person doing all the steps. Also, several energy calculation problems, encoded on different colored cards, were passed through the pipeline, but out of phase, to keep the human analysts busy. (Reminds one of the multi-threading scheme used in the HEP many years later (1978).) By this means, Feynman and his group could solve two or three problems at a time in parallel. Although an energy calculation took over a month to solve this way, Feynman's group could average a solution every two weeks because of the parallelism.

After over two decades of "sequentializing" numerical algorithms, parallel algorithms were first designed for the ILLIAC IV at the University of Illinois (late 1960s). Kuck and Sameh [Sameh, 1977] discovered that parallelizing traditional stable sequential numerical algorithms might become unstable numerically. Hence, numerical analysts started the search for numerically stable parallel algorithms as well as ones with optimal speedup. In the last two decades, many parallel numerical algorithms have been designed and analyzed [Carey, 1989] [Bertsekas, 1989].

Theory of Parallel Algorithms

The theory of parallel algorithms began in the late 1940s when John von Neumann [Burks, 1966] introduced a restricted model of parallel computing called cellular automata, which is essentially a two-dimensional array of finite-state processors interconnected in a mesh-like fashion.

The popular theoretical model, the Parallel Random-Access Machine (PRAM, pronounced "PEE-ram") was formalized in 1978 by Fortune and Wyllie [Fortune, 1978], although many other authors had previously discussed essentially similar models. The basic architecture of PRAM is p serial processors that share a global memory. All the processors, in a tightly synchronous manner, can read or write to the global memory in parallel (basically a shared memory SIMD). Although the PRAM model ignores many important aspects of real parallel machines, it has become an important model for analyzing the behavior of parallel algorithms [Corman, 1990].

In the previous four sections on the history of parallelism, we have seen that aspects of parallelism have been studied for many years. Also, we have seen that many research communities have been involved. Algorithm theorists develop models of parallel computation and analyze the

---

[8] Feynman, Richard P., "*Surely You're Joking Mr. Feynman!*," Bantum Books, 1985, pp. 108-114.

behavior of parallel algorithms. Computer designers and computer architects use parallelism to achieve higher performance in the machine's hardware. Operating system researchers develop and study distributed systems. Program language designers create and investigate parallel programming languages. Numerical analysts and scientists develop practical parallel numerical algorithms to solve problems.

1.13 Layered Model of Parallel Processing

The previous sections show that parallelism covers a wide spectrum, from hardware design to the theory of computation. In fact, aspects of parallel processing could be incorporated into every computer science course in the curriculum. In order to deal with this massive amount of subject material, we need a way to organize it. This textbook is organized around a five-layered model of parallel processing (See Figure 1.15).

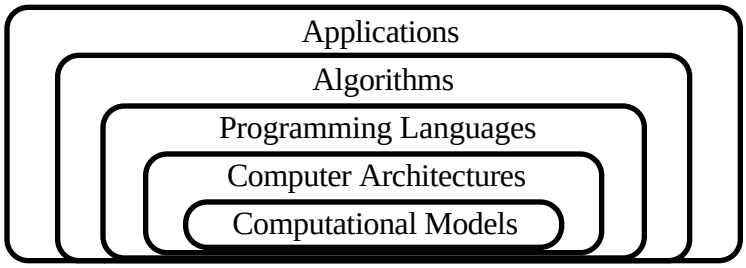


Fig. 1.15 The Five Layers of Parallel Processing

Central to the material are parallel computational models or theoretical ways to view computation. Selecting a computational model, we implement a parallel architecture based on the computational model. We can design a parallel programming language based on the computational model which generates code for the architecture[9]. We develop and analyze parallel algorithms written in the programming language. And finally, we use the algorithms in parallel applications. These five layers span a large portion of computer science.

In the next several sections, we raise some of the questions pertaining to each layer that we will investigate in later chapters.

1.13.1 Parallel Computational Models

A computational model attempts to define and clarify one of the many ways we view computation. By proposing an abstract machine or a mathematical model, a researcher tries to characterize the essence of one way of computation to form a theoretical basis for other research. With a parallel computational model, we ask: "How is this different from other parallel models and the serial model, that we all know?" Some parallel models are mind expanding because they require us to think in a very different way from serial processing. The parallel computation models we will study in later chapters include Communicating Sequential Processes (CSP), Parallel Random-Access Machine (PRAM), dataflow, demand flow and systolic processing. We will want to ask such questions as: "What are the issues?" "Which models are important, and why?"

1.13.2 Parallel Computer Architectures

[9] Or we could first design a programming language based on a computational model, then implement an architecture based on the language.

A machine architecture is a physical realization of a parallel computational model's abstract machine. The implementation depends heavily on the technology available, For example, the characteristics of light and the advancements in optics may favor an optical architecture totally different from today's electronic-based computers. Even within the framework of one parallel model, the designer has many ways to exploit the parallelism. In the later chapters, we ask such questions as: "What are the architectural issues?" "What are the engineering tradeoffs?" "How do we manage the resources effectively?" "How easily does the machine interface with people in the form of operating systems, programming environments, programming languages and training needs?"

1.13.3 Parallel Programming Languages

The semantics of a programming language reflects the underlying computational model. Also, a compiler for the language must generate effective code for the underlying machine architecture. With a programming language, we are concerned with its expressibility. Are we able to express parallel solutions in a natural and concise manner? What is the language's effectiveness in exploiting the parallelism in the problem? Is the language portable across different machines? In later chapters, we will study several practical parallel languages: FORTRAN 90, Ada, and Logical Systems C,. We will also study several languages with a more formal theoretical basis: Occam, ID and SISAL.

1.13.4 Parallel Algorithms

The notation (programming language) we use to describe an algorithm profoundly influences our design. But more importantly, a parallel algorithm's design depends heavily on the underlying computational model and/or architectural model.

For a specific class of architecture, e. g., SIMD, researchers have established successful design paradigms to develop parallel algorithms. These modes of thinking allow the programmer to develop algorithms with one eye on a high performance solution and the other eye on the common pitfalls encountered in the class, e. g., deadlock. In later chapters, we will explore several of these design paradigms and discuss their strengths and weaknesses.

Unfortunately, because of the vastly different approaches in the parallel computational models and/or the parallel architecture, an algorithm developed under one design paradigm, e.g., designed for SIMD, probably won't work under another architectural structure. In fact, the situation may be so bad that you should spend the time and effort to discover a new algorithm rather than try to rework the old algorithm into the new computational/architectural structure. This sad state of affairs has prompted some researchers to search for a universal design paradigm which would work for all parallel architectural structure at the cost of reduced performance. In a later chapter, we will study Linda, which attempts to fulfill this universal paradigm.

As we study the design paradigms, we will study several algorithms of interest to computer scientists.

1.13.5 Parallel Application Areas

Many application areas have inherent parallelism, e. g., image processing, fluid flow, thermo flow, and weather forecasting. Although discussing parallel applications is beyond the scope of this book, we will discuss several in later chapters to indicate the range of possible applications.