# Highly Parallel Computing

**George S. Almasi**

*IBM Thomas J. Watson Research Center*

**Allan Gottlieb**

*NYU Courant Institute of Mathematical Sciences*


**The Benjamin/Cummings Publishing Company, Inc.**
Redwood City, California • Fort Collins, Colorado • Menlo Park, California
Reading, Massachusetts • New York • Don Mills, Ontario • Wokingham, U.K.
Amsterdam • Bonn • Sydney • Singapore • Tokyo • Madrid • San Juan

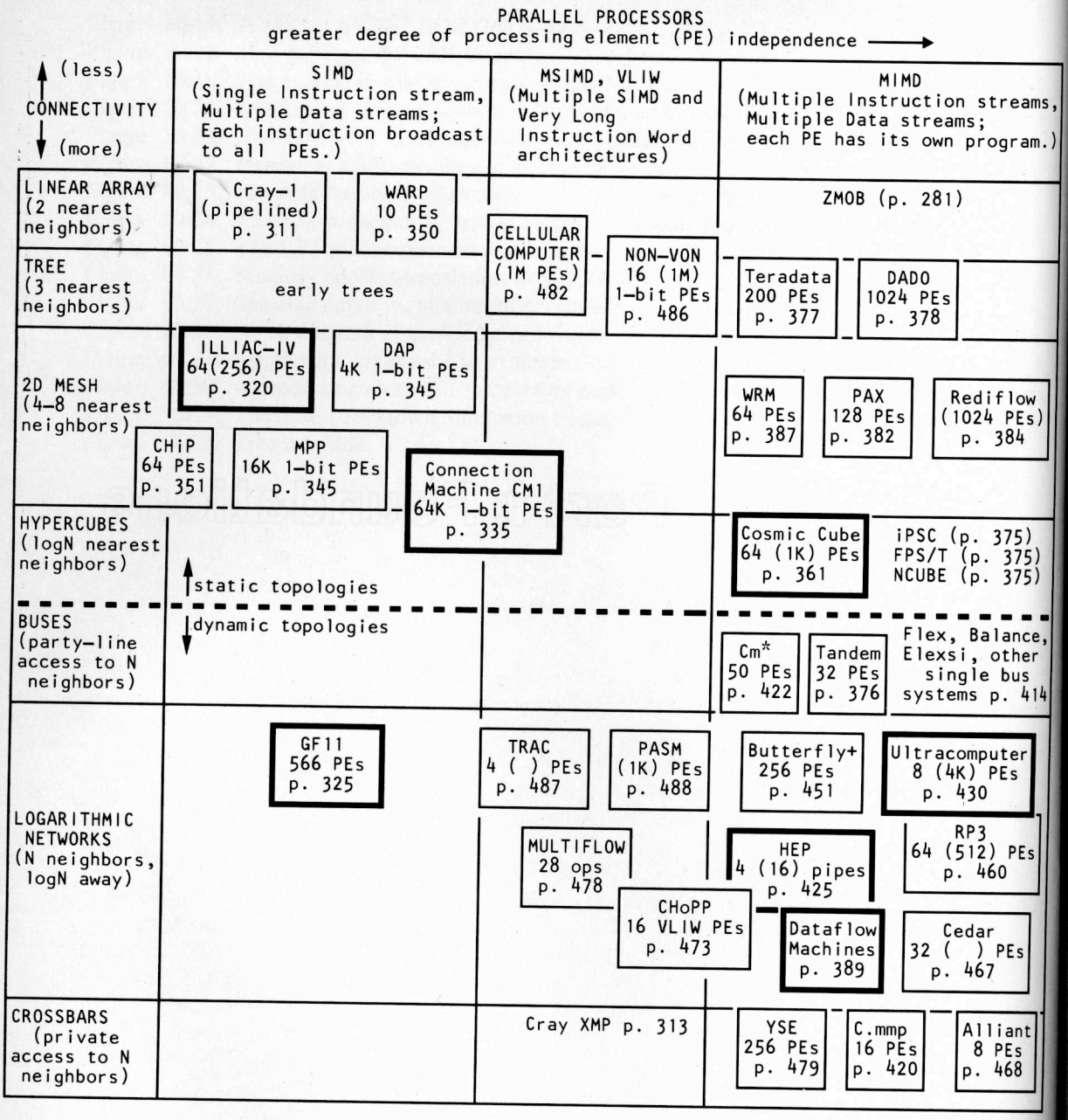


***Figure 1.1.*** *Parallel architecture projects grouped according to the independence and connectivity of the processing elements (PEs). Heavy boxes denote projects that we treat as detailed case studies, light boxes denote less detailed decriptions, and names without boxes correspond to brief summaries. The page numbers are where they appear in this book. The nominal number of PEs is given in parentheses, preceded by the number built so far.*

# 1 OVERVIEW

We are on the threshold of a new era in computer architecture. It is becoming increasingly difficult to obtain more performance from the time-honored von Neumann model, and many of the technological constraints that influenced its design over thirty years ago have changed drastically. Many of the arguments for processing a single instruction at a time no longer apply, and a number of enthusiastic parallel processing projects (Figure 1.1) are working on various ways to allow many processors to cooperate on a single problem. However, this reopens a Pandora's box of questions about how computation should be done, a box closed four decades ago by the von Neumann model. Some of the strengths of this model become especially apparent when one tries to replace it. The field is at an interesting juncture. Much work has been done, and ideas now exist for putting it all together. But large experiments are needed to provide real results from real programs if the pace of progress is to be maintained. To quote John Hopcroft, winner of the 1986 Turing Award, "I don't think people have a good understanding yet of parallelism and what it is going to buy us. We have to develop ways of thinking about parallelism and languages for expressing parallel algorithms. There will be major activity in that area for 5 or 10 years to develop the science base that's needed to exploit it."

This chapter presents a fairly substantial overview of parallel processing and of this book. It is written at such a level that readers not interested in nitty-gritty details should still be able to derive from this chapter a knowledge of what the main problems are and what solutions have been and are being pursued. However, sufficient annotation and examples are provided so that this chapter can also be used as a road map to the more detailed treatment of these topics in subsequent chapters.

This chapter is organized as follows: "Definition and Driving Forces" begins to define what we mean by parallel processing and why the interest in it is growing so rapidly. "Questions" shows that, on reflection, our definition raises a number of additional questions. "Emerging Answers" starts to answer some of these questions, weaving in a bird's-eye preview of projects that are providing some of the answers and that are treated in detail later in the book. Finally, in "Previous Attempts: Why Expect Success Now?", we discuss why parallel processing is more likely to succeed now than it was, say, ten years ago.

## 1.1 Overview and Scope of This Book

As we said in the Preface, this book is organized into three parts called "Foundations", "Parallel Software", and "Parallel Architectures", each with several chapters.

It is tempting to begin with parallel architectures ("the fun stuff"), and those who succumb to this temptation are referred to page 275 where the third and largest part of our book begins. Four chapters there cover interconnection schemes, parallel computers whose processing elements each have their own programs, those that do not, and hybrids of these two.

However, parallel computing is a much broader subject. Its major software components form the subject of the second part of this book, which starts on page 147. The three chapters involved discuss parallel languages, compilers, and operating systems.

But we are dealing here with a fundamental expansion in the possible ways of thinking about and doing computations, one that reopens many questions that arose during the forty years that electronic computers have been with us, questions that were answered under technological constraints that no longer exist. To put into perspective the opportunity and adventure that parallel processing represents, we begin this book with a four-chapter first part titled "Foundations". It provides a preview of the whole book and a perspective on potential parallelism in applications, technological limitations and opportunities, and formulation of parallel solutions.

Some important topics that are absent from this book help to clarify its focus. We are interested in computers that use a high degree of parallelism to speed the computation required to solve a single large problem. This leaves out much of the "COBOL" world of business programs, where I/O rather than computing power is typically the bottleneck. It leaves out most commercial multiprocessors, whose added processing units are used to increase the *number* of jobs that can be handled at a time, rather than to speed up a single job (in other words, to improve *throughput* rather than *turnaround time*). And it leaves out distributed systems [208] such as a network of personal workstations, because, although the number of processing units can be quite large, the communication in such systems is currently too slow to allow close cooperation on one job.

## 1.2 Definition and Driving Forces

The following definition describes many highly parallel processor designs[1] :

> **"A large collection of processing elements that can communicate and cooperate to solve large problems fast."**

One could include other important factors such as reliability and ease of programming. However, we shall soon see that even this simple definition raises more questions than it answers.

### 1.2.1 Driving Forces and Enabling Factors

The basic driving force for parallel processing is the desire and prospect for greater performance: users have ever bigger problems, and designers have ever more gates. Historically, most of the impetus has come from people with three different kinds of concerns:

1. How to solve large problems that run too slowly even on the fastest contemporary supercomputer. Typical examples have been weather modeling, design automation, and other scientific/engineering applications. More recently, artificial intelligence has become a source of such problems.
2. How to obtain solutions to problems that *could* be done on one of today's supercomputers when one's budget is limited.
3. How to increase programmer productivity. Proposed solutions often call for increased processing power and, perhaps, new parallel architectures.

In short, parallel processing is perceived as having the potential to improve *performance, cost/performance,* and *productivity.* (There are other factors as well, including reliability/availability and reduced part numbers.)

The key enabling factors have been hardware technology improvements (chiefly VLSI) that have led to a repeal of "Grosch's law" [262], a long-valid observation that the best price/performance was obtained with the most powerful uniprocessor. Thus it is no longer automatically true that a collection of smaller processors will always have less performance than a single large processor of the same total cost. At the same time, physical effects such as the finite speed of light are making it increasingly difficult for technology alone to speed up the fastest serial

---

1 Some terminology: A *functional unit* is a logic circuit like an adder, multiplier, or shifter. A *processing element* (PE) is the same as a CPU; it contains functional units, registers, clock circuits, and processor and bus control circuits. In some cases (MIMD computers) the PE contains an instruction decoder and program counter. A *computer* contains PE(s), memory, I/O, and control for all. Hence, the definition of "parallel processor" given in the text leaves out computers whose only parallelism consists of multiple pipelined functional units, like the CRAY-1. These are interesting but are not the main emphasis of this book.

processors (chapter 3 on technology discusses this). In these circumstances, a parallel computer made up of a number of small cooperating processing elements becomes an attractive proposition for many problems. Our applications chapter (page 31) gives some examples of such problems.

From the software side, John Backus, the inventor of FORTRAN, argues strongly that programmer productivity is declining due to the anachronistic "von Neumann bottleneck"[38] and that new languages and new architectures to support them are needed. This is discussed in our languages chapter (page 151).

## 1.3 Questions

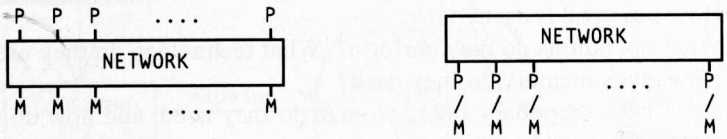


***Figure 1.2.*** *Generalized parallel processor showing two ways to arrange the processing elements (P) and memory modules (M). The "dancehall" configuration at left is used primarily for designs with a "shared memory" model of computation in which every P has equal access to every M. The "boudoir" configuration at right is used primarily for "message-passing" designs in which the processing elements each have their own memory and communicate by exchanging messages. Shared memory is a more powerful communication mechanism, but costs more to implement.*

Let us return now to the definition we gave for a parallel processor. We said it was

> **"A collection of processing elements that can communicate and cooperate to solve large problems fast."**

Two of the implementations we might conceive for this definition are sketched in Figure 1.2. However, a few minutes spent thinking about this definition in a very common-sense sort of way shows that it raises quite a list of fundamental questions. In fact, readers are encouraged to pause here, make their own lists of questions, compare them with the list on the next page, and then keep these questions in mind while reading the rest of the book.

Some of the key questions raised by the different parts of our definition are as follows:

1. "A COLLECTION OF PROCESSING ELEMENTS..."
   - How many?
   - How powerful is each?
   - What operations do they perform? What technology do they use?
   - How much memory do they need?
   - How much secondary (disk) storage do they need, and how do they perform I/O access to it?
2. "...THAT CAN COMMUNICATE..."
   - How will they communicate? What protocol will they use?
   - What sort of interconnection network will they need?
3. "...AND COOPERATE..."
   - How will they coordinate (synchronize) their efforts?
   - How large should each processor's task be?
   - How autonomous should each processor be?
   - How will the operating system coordinate these efforts?
4. "...TO SOLVE LARGE PROBLEMS FAST."
   - Which problems (applications) are amenable to parallel processing?
   - How? That is, what computational model will be used?
   - How general-purpose vs. special-purpose should our approach be?
   - What algorithm is best now?
   - How much speedup can be expected?
   - What is the programmer's view of the machine?
     - What programming language and software tools will be available?
     - How is a problem decomposed?
     - How is concurrent execution specified?
     - Who does each of the above? The user or a compiler?
     - What about reliability, availability, serviceability (RAS)?

Almost as interesting as the questions themselves is the high degree to which they are intertwined. The answers are far from independent of each other. If one divides the world into architecture issues and software issues, then item 1 in the list above can be thought of as a set of questions related primarily to architecture and technology, whereas item 4 relates mostly to software. However, items 2 and 3 fall under both umbrellas, and the umbrellas themselves interact. Is computer design driven by problems looking for solutions, or by solutions looking for problems? The answer is, by both. This is nicely illustrated by the deceptively simple diagram in Figure 1.3 on page 9, which shows how the key elements of a computing system are related. This relationship is discussed in more detail in chapter 4 on computational models, but it is interesting to note that both the problem and the technology appear as driving forces.

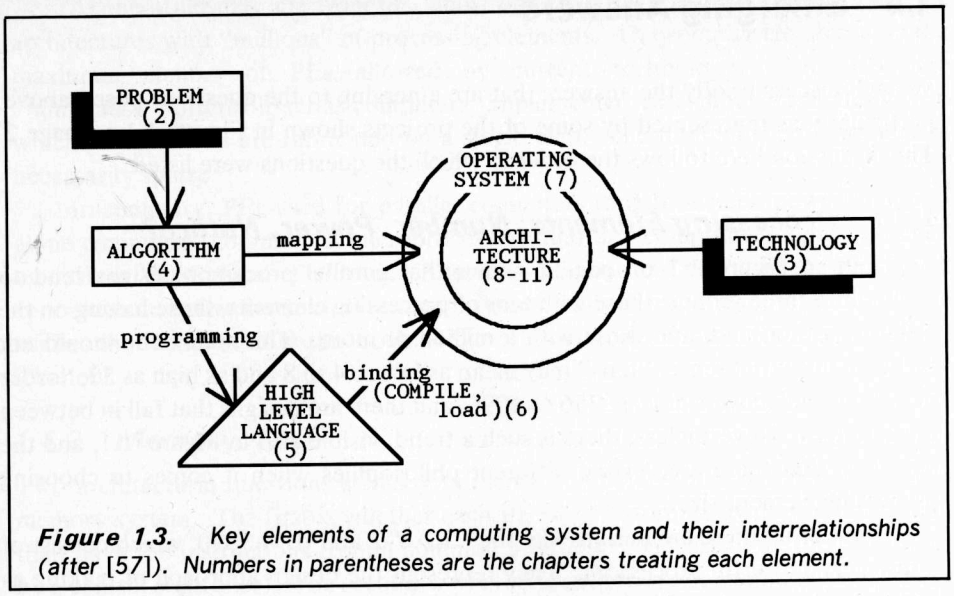


***Figure 1.3.*** *Key elements of a computing system and their interrelationships (after [57]). Numbers in parentheses are the chapters treating each element.*

These key elements are also reflected in the flow of this book: the two driving forces of progress, namely, the problem to be solved (i.e., the application) and the technology (i.e., what is possible to build) are discussed in chapters 2 and 3. At a higher level of abstraction, computational models (or "rules of the game") available for problem solving are treated in chapter 4 along with parallel algorithms (or recipes) for selected problems. Chapter 5 on languages describes how the programmer can choreograph the dance that the hardware elements of the computing system are supposed to perform. Chapter 6 on compilers describes how the programmer's instructions are translated into terms the dancers can understand, and 7 on operating systems discusses the issue of directing the dancers during the performance. Chapter 8 on interconnections describes how the elements stay in touch while they perform the dance. And chapters 9, 10, and 11 on architectures give examples of real dancers and real dances. The emphasis is on the new elements introduced by the decision to do things in parallel.

The rest of the present chapter is intended to provide the reader with a brief summary of the emerging answers to the questions listed above, and also a glimpse of the parallel processing projects and activities that are providing these answers.

## 1.4 Emerging Answers

We will discuss briefly the answers that are emerging to the questions posed above, particularly as represented by some of the projects shown in Figure 1.1 on page 2. The discussion here follows the order in which the questions were listed.

### 1.4.1 Processing Elements: Number, Power, Nature?

A glance at Figure 1.1 on page 2 shows that parallel processor designs tend to cluster into three groups: those with tens of processing elements, those having on the order of a thousand, and those with a million or more. This statement should not be taken in a strict sense: "tens" may mean as low as 4 to 8 and as high as 32, "order of a thousand" covers at least 256 to 4096, and there are designs that fall in between these ranges. Nevertheless, there is such a trend, visible also in Figure 1.1, and the reason is that there are several different philosophies when it comes to choosing processing element size.

First of all there are designs like the CRAY-2 and IBM 3090, which consist of a few very powerful processors. They represent the design approach of making as fast a computer as possible (normally a pipelined vector machine; see page 301) and then combining as many of them as is practical. This group contains the bulk of today's high-end commercial computers as well as holders of the land speed record for scientific computations. They tend to use high-speed, high-power bipolar transistor technology (page 81) requiring sophisticated cooling techniques (page 86 ff.). By and large, however, the parallelism of these designs has not been applied to speed up a single job,[2] but rather to increase the *throughput* of the system, that is, to allow it to handle *more* jobs each day. As we explained earlier, this book is about parallelism used for *speedup* rather than for throughput, and so we say relatively little about such systems despite their commercial importance and high absolute performance.

The other designs in the "tens of PEs" category contain *bus based* machines that emphasize cost/performance (rather than raw performance). The hardware is built from slightly slower but smaller, lower-power, and more easily cooled NMOS or CMOS transistors (page 77). Although the bus is economical and allows all PEs to communicate directly, its fixed bandwidth limits the number of PEs.

Computers with "thousands of PEs" are mostly designed to take advantage of available *microprocessors* or other VLSI computing engines. The bandwidth required for the larger number of processors is obtained by using more powerful networks, which are discussed in chapter 8. At least one design (page 460) uses bipolar technology for this network.

---

2 Here we are referring to the "high level" hardware parallelism arising from the multiple processors. Certainly the lower levels of hardware parallelism arising from multiple, pipelined functional units concurrently operating on wide data items is applied to a single job.

At the other extreme from the "most powerful PE possible" approach are the architectures with "millions" of processing elements. They emphasize obtaining the maximum *number* of PEs allowed by current technology. The PEs are "minimalist", often bit-serial computers, implemented as custom VLSI designs in which multiple PEs are fabricated on a single chip. The interconnection scheme is necessarily sparse.

In summary, PEs used for parallel computers to date include powerful stand-alone computers, commercial microprocessors, and tiny computing engines grouped on custom VLSI chips. The tradeoff between number and power of the PEs is still very much an active research area, and is discussed further in the representative case studies that come later in this book. Figure 1.1 on page 2 provides a map to their locations.

### 1.4.2 Memory, I/O?

Two architectural questions strongly influence the details of a parallel computer's memory system. The first is whether each PE should store its own program (rather than have the instructions from a common program broadcast to all the processors by a central controller). The second is whether each processor should have exclusive access to its part of the data (rather than pooling all the data and adopting a Turkish bath philosophy of letting everyone see what everyone else has got). The three answers given (no, yes-yes, and yes-no)[3] map to the left half, right upper quadrant, and right lower quadrant of Figure 1.1, and are discussed in detail in chapters 9 and 10. In the much briefer but nearer discussion on page 19, we explain that these three possibilities are named SIMD, MIMD with private memory, and MIMD with shared memory. These three possibilities represent three progressively greater degrees of autonomy and interactivity for the PEs.

Here, we are going to talk about the gross memory and I/O requirements of the entire parallel system. For want of a better way, the relationships between memory capacity, disk I/O bandwidth, and processing rate are being extrapolated from rules of thumb established for serial processors (one current example is "about one MByte of memory and one MByte/sec I/O rate per MIPS"). Chapter 3 discusses these briefly. The memory and I/O structure will be influenced by a number of factors, including:

1. The power of the processors
2. The ratio of processing to communication
3. The frequency and nature of the I/O (sequential, random, paging, etc.)

---

3 Although four answers are theoretically possible, a design with pooled data and a single copy of a program (shared memory SIMD) has not been implemented to date. With a shared memory, the tight synchronization favored by SIMD enthusiasts becomes difficult because the possibility of memory access conflicts leads to unpredictable instruction execution times [265].

The last two items will depend strongly on the application, and can be expected to be quite different for problems in, say, numeric computation, symbolic computation, and database operations. Chapter 2 treats these applications.

### 1.4.3 How Do the Processors Communicate?

The interconnection networks that allow communication among the PEs have ranged from the elaborate to the simple, representing a tradeoff between communication power and cost. At one end of this spectrum is the crossbar network, which allows complete connectivity and minimum delay among the processors, but costs a great deal to implement. At the other end are easy-to-build tree networks in which the PEs are connected to only their three nearest neighbors. Intermediate tradeoffs, such as mesh, cube, and multistage networks, are shown along the left edge of Figure 1.1 on page 2 and discussed in more detail in chapter 8.

The *mode* of communication in the network must also be specified. In *circuit switching*, a connection is established that lasts at least until the entire message has been transmitted. The classic example is the telephone system, in which a circuit is established when a call is made and remains until the call is terminated. The lines and switches being used are unavailable to anyone else in the interim. In a *packet-switched* network, the message is broken up into packets, each bearing a destination address (and often a source address). These packets then steer their way through the network releasing resources just after they are used (see Figure 8.7 on page 294). This process avoids the overhead of setting up individual connections and is more economical if the communication consists mostly of short messages scattered to different destinations. Both modes are represented among the projects in Figure 1.1.

These communication *mechanisms* (networks and protocols) can be used to build either a shared memory or a message-passing communication *system*. Page 430, for example, describes a MIMD shared memory system using a packet-switched omega network, page 325 describes a SIMD system using a circuit-switched Beneš network, and page 361 describes a MIMD private memory system using a (static-topology) cube network. In the shared memory case, PEs communicate via common access to a shared global memory. On the other hand, if each PE has access to only its own private memory, a separate message mechanism is needed to communicate with the other PEs. (The shared memory can, of course, also be used for message passing.) Message passing and shared memory are compared briefly on pages 24 and 356, and in much more detail on pages 360 ff. and 412 ff., respectively. Communication is discussed further in chapter 4 on computational models, chapter 8 on interconnection networks, and chapters 9, 10, and 11 on specific architectures.

### 1.4.4 How do the processors cooperate?

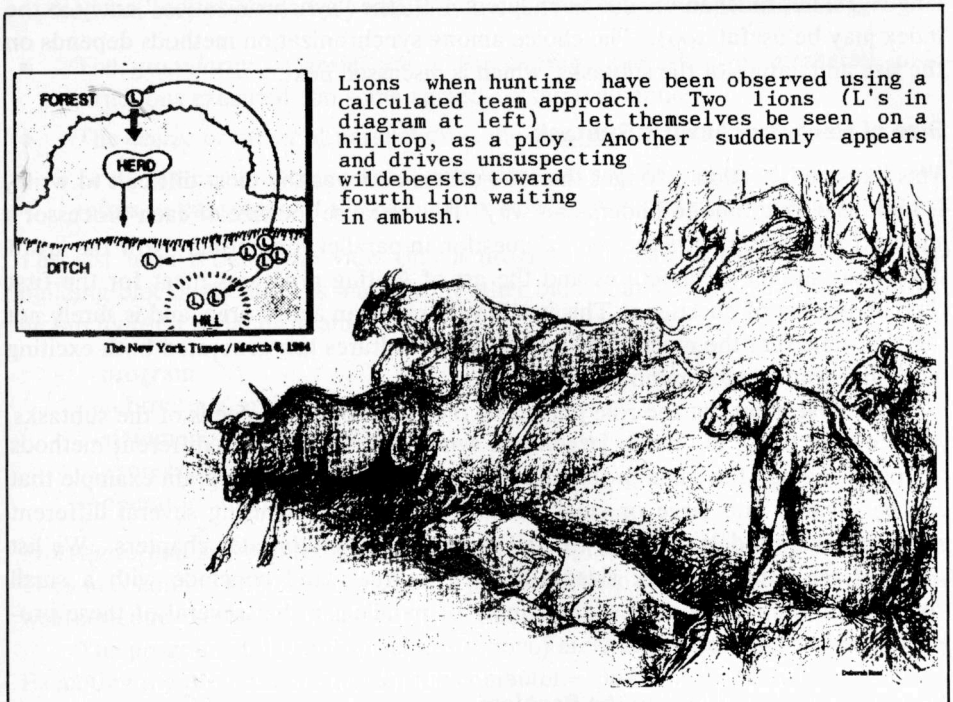


***Figure 1.4.*** *An illustration of the need for synchronization during a cooperative task. The plan succeeds only if the tasks are performed in the right sequence.*

#### Synchronization

Things must happen in a certain sequence if our parallel computer's cooperating processors are to produce the desired results. This requirement introduces the need for *synchronization*. An interesting example of this was given in the *New York Times* in an article [341] describing how lions seem to use a calculated team approach when hunting a herd of wildebeests, as shown in Figure 1.4. One lion slinks, unseen by the herd, along the bottom of a ditch into a position of ambush. Two lions then climb a hill close enough to the herd to insert themselves into the herd's consciousness and make it nervous, but not close enough to panic the herd yet. Then a fourth lion charges the herd. Remembering the other two lions, the herd charges away from the new lion *and* away from the hill and toward the lion waiting in ambush. All four lions then share the kill.

The success of this plan depends strongly on the exact *order* in which the lions perform their tasks. Analogous situations exist in many programs, which is why the language used to write such programs must be able to *express* such synchronization

requirements[4] , and why hardware and/or software means are needed to *implement* them. There are many ways of doing this, as discussed primarily in chapter 5 on languages, but touched on also in chapters 4–9 (the "synchronization" entry in the index may be useful, too). The choice among synchronization methods depends on the size and nature of the subtasks, which is discussed next.

### Size of Each Processor's Subtask

This "answer" section is longer than the others and was the most difficult to write. It took us a long time to understand why. It is because the size of each processor's subtask[5] is in many ways the *central* question in parallel computing. Now the art of designing parallel architectures and the art of writing programs meet for the first time on our list of questions. This is still very much an active area, and is surely not the first time that the meeting of two different cultures has generated both exciting work and confusing terminology.

We first point out the role played by the *communication* needs of the subtasks. After a brief pause to discuss jargon and synonyms, we compare different methods of "parallelizing" the same problem by examining a common program example that can be divided into large or small subtasks and executed using several different computational models, in ways treated more extensively in later chapters. We list some of the other subtask sizes and their sources, and conclude with a small "parallelism map" that depicts the supply of parallelism that several of these program constructs represent.

#### *Choosing and Partitioning the Problem*

Problems can sometimes be divided into almost completely independent, non-communicating pieces (see "easy parallelism" on page 21), but in general the subtasks need to communicate in order to exchange data and coordinate their activities. For good efficiency, the time spent on such communication and on such overheads as starting up the subtasks should be small compared to the time spent executing the subtasks. For example, a problem that has been divided into a few large subtasks may be well matched to a parallel computer consisting of a few powerful processors, and can afford the overhead of the Ada language's "rendezvous" mechanism (see chapter 5) to control the interaction of these tasks. However, this combination of architecture and language would be a very poor way to obtain speedup for a problem divided into a much larger number of much smaller pieces. Although there are some simple, useful theoretical models (page 223), the general topic of finding the best workload for a given parallel computer is a complex subject occupying much of the rest of this book.

---

4 Dataflow language programs (page 189) are an example of an exception.

5 We use this term loosely here. The precise definition of "task" in languages like PL/1 and Ada is treated later.

To probe the question of subtask size further, let us examine what is available. We can start constructing a parallelism map by analyzing existing (serial) programs to see what sort of pieces they naturally break into. First, some terminology:

- The *granularity* or *grain size* is the average subtask size, measured in instructions executed (on some agreed-on serial computer).
- The *degree* or *extent* of parallelism is the number of subtasks available.
- *Level* (as in "procedure-level parallelism, expression-level parallelism," etc.) refers to the source of the parallelism in what was originally a serial program.

The last term reflects the view that a program consists of certain characteristic building blocks, themselves made from other blocks, and so on, as in the following hierarchy for a serial program [8]:

-program
-subroutines and blocks
-statements
-expressions
-operators and data references

*Procedures* and *subroutines* perform specific computations needed by the program. They may contain *program loops*, a very popular source of parallelism (see the NASA weather code example on page 42).

The program and its substructures (above) are static quantities — always there. Executing the program creates for the computer a chunk of work called a *job*, a dynamic quantity that comes and goes. As candidates for *parallel* workloads go, jobs are the largest and least communicative units of work on a computer. There is no shared memory and perhaps not even shared disk space among jobs, and any communication among them is limited to exchanging messages or perhaps only files. Such loose coupling corresponds more to distributed processing or throughput-oriented parallelism, whereas this book concerns the use of parallelism to speed up a single job, like the computation of tomorrow's weather, which takes a significant amount of time when performed at the precision desired.

To utilize the multiple processors found on an MIMD computer, a job is broken into a number of pieces called *processes* or *tasks*, terms that are used interchangeably in this book[6] to mean executions of a program or multiple parts of a program. An important distinction that we *do* make is that programs and subprograms are static, that is, once written they exist forever; jobs and tasks/processes, however, are dynamic quantities that come and go. In some systems, tasks and jobs have the same

6 Many efforts have been made to give these two terms different definitions that distinguish between the resources needed for execution (i.e., a *protection domain*) and the execution itself (the thread of control), or, more loosely, between the ability to do something and the actual doing of it. Unfortunately, different authors have used different definitions, and none is accepted universally. Instead, we take "process" and "task" to be synonyms that mean an execution of a (sub)program. Where finer distinctions are needed, we discuss them explicitly.

lifetime, so a job has a fixed number of tasks; in others tasks are "more dynamic", with their number varying during the job's execution. This topic is discussed further in chapter 7 on operating systems.

### *Four Ways to Do Matrix Multiplication in Parallel*

The best-known source of parallel workloads may be the loops in FORTRAN scientific programs. As an example, consider the matrix multiplication

$$C(i,j) = \sum_{k=1}^{N} A(i,k) \times B(k,j)$$

which is often coded as

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

(the Cs are initialized to 0). The two outer loops (the ones in which I and J vary) generate $N^2$ independent computations, which is as it should be since each one computes a different element of C. These $N^2$ iterations of the outer loops could *each* be a task for a PE that has the equipment to store and execute the program represented by the inner (K) loop, as is true for the PEs in an MIMD parallel computer. In practice, however, a larger task consisting of several iterations may be more favorable due to communication overheads and other factors. For example, in a 400x400 matrix multiplication on the 256-PE Butterfly (page 457), assigning each PE a different iteration created too much traffic in the network leading to the memory, and 6x6 subsets of the matrix were parceled out instead. Thus each PE produced 36 elements at the cost of fetching 6 rows and 6 columns from memory. By contrast, computing 36 elements one at a time requires fetching 1 row and 1 column *each time*, or six times more memory traffic.

The foregoing partitioning is appropriate for an ensemble of reasonably powerful, more or less conventional (serial, von Neumann) processing elements — microprocessors or larger — able to operate in asynchronous MIMD mode, that is, having provisions for the likelihood that some of the subtasks will take longer than others. In some of the designs in chapter 10 that fit this description, these subtasks are scheduled at the time the program is executed, while in others it is done earlier. A simple, useful theoretical model for the maximum useful parallelism in such machines appears on page 223.

This same problem can also be sliced to yield a much finer granularity of parallelism, that is, into much smaller parallel tasks. The most thoroughly explored of these levels is that of machine instructions. We describe three approaches that use this fine-grained parallelism: dataflow, vector processors, and systolic arrays. The thinking behind all three is that surely at most moments of a program's exe-

cution there is a huge backlog of instructions that could be done in parallel, even though the von Neumann model does them one at a time. Dataflow computers are sort of an extreme effort to capitalize on this.

As described on page 389, dataflow computers are MIMD, but use a radically different computational model in which instructions may execute as soon as their input data are ready. (This is the "synchronization mechanism" in this model.) Figure 5.12 on page 189 shows a dataflow method for solving the matrix multiplication example above.

SIMD, a more modest approach to exploiting parallelism at the instruction level, has had more commercial success to date in the form of the vector processor, which is the basis of the present generation of supercomputers. As discussed on page 303, a vector processor has some extra computing units and fast memory designed for efficient execution of long sequences of identical operations. An added restriction in a vector processor is that the identical operations are performed on data stored in a highly regular manner (i.e., the data can be considered as a vector). Thus ILLIAC IV (page 320) is a parallel vector machine, the CRAY-1 (page 311) is a pipelined vector machine,[7] but GF11 (page 325) and CM1 (page 335) are more general SIMD parallel computers. All use instruction-level parallelism. By moving the K loop to the outside, the iterations of the inner loops become independent, as explained on page 216, and the program above is then ideal for a vector processor. Vectorizable and non-vectorizable loops from a weather code are shown in Figure 2.5 on page 42.

The type of VLSI-oriented systolic array shown in Figure 9.14 on page 348 can be regarded as a low-cost vector processor designed to perform one particular algorithm, in this case the matrix multiplication above. Subtasks must still be independent, as with a more general-purpose vector processor, but are now "scheduled" at hardware design time instead of at compile time.

Vector machines can afford to use subtasks as small as an instruction because the synchronization is performed by the hardware automatically. The "cost" is the necessity to cast algorithms in vectorizable form. Dataflow hopes to be able to afford instruction level parallelism for essentially the same reason: the machine is self-synchronized, in this case by the arrival of data. The key is a large supply of other subtasks that can execute while one communicates.

---

7 Pipelining may be considered SIMD by viewing the stages of the pipeline as the multiple PEs.

### A Parallelism Map

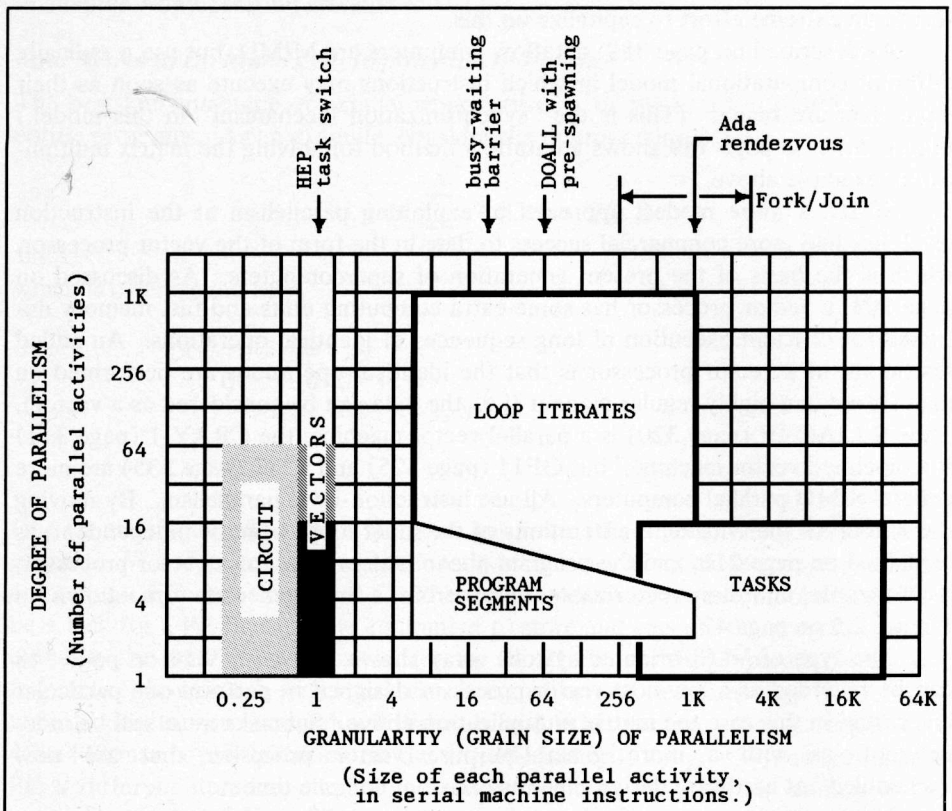


***Figure 1.5.*** *Parallelism map showing degree of parallelism available at a given granularity. The total parallelism can be multiplicative: 16 tasks could be running, each with 1000 independent iterations of a loop, etc. Open areas denote sources of parallelism in a program; shaded areas denote hardware parallelism: ILLIAC IV had 64 PEs, a pipelined vector processor might have 16 independent functional units, etc. "CIRCUIT" area refers to horizontal microcode and multiple-bit logic circuits. Speedup is bounded by the degree of parallelism and determined by the size of the overhead. Several examples of such overhead are shown on the top edge of this map. Chapter 5 treats these parallel coordination constructs in more detail.*

### How Autonomous Should Each Processor Be?

Should each processing element execute its own program, or should they all receive the same instructions from a central source? These two possibilities (robots or puppets?) are called MIMD and SIMD, respectively, from a classification originally made by Flynn [126] and still widely used. The idea is that the von Neumann model of computation (page 111) can be viewed as a stream of instructions and a stream of data being knitted together, one instruction at a time, to produce a useful result. Since there is a Single Instruction stream and a Single Data stream, its category receives the name SISD. One can take a step into the world of parallel execution by having this Single Instruction stream manipulate Multiple Data streams (SIMD), and a further step by adding Multiple Instruction streams (MIMD).

SIMD/MIMD debates are sometimes heated. This book devotes a chapter to each of these architectures. SIMD processors have been around for a while in the form of vector processors and array processors (see page 303), and even though there is more research on MIMD designs these days, there is still a substantial amount of work being done on SIMD designs.

SIMD offers simpler synchronization: the processors are kept in lockstep by the broadcast instructions. Therefore, the processors need not talk to *each other* for synchronization purposes. Since they need not store their own programs, the processors can also be smaller and more numerous.

MIMD is seen as a more general design capable of performing well over a broader range of applications. However, in an MIMD machine each processor needs enough memory to store its own copy of at least part of the program and enough logic to decode instructions and manage its program counter; this makes designs with more than a few thousand processors seem difficult to achieve. But some researchers are inherently interested in machines with millions of processors, because of the nature of their applications or because of perceived analogies with the brain; SIMD is their only alternative, given the technologies that are available now. SIMD designs tend to be associated with smaller-grain parallelism than MIMD, since the overhead costs per task are lower.[8] Examples are given in chapters 9 and 10.

[8] But there are also MIMD designs, such as dataflow, that are aimed at fine-grain parallelism.

### How Will the Operating System Coordinate These Efforts?

We discuss operating systems for parallel processors in chapter 7, which begins on page 247. To paraphrase the "beautification principle" from that chapter, an operating system provides a thin veneer of civilization between the machine hardware and the programmers who wish to use it.[9] Key functions include the creation and scheduling of tasks, and the management of resources such as memory, I/O devices, and file systems, with an eye toward various kinds of system load balancing and protection. As pointed out in the introduction to chapter 7, even an operating system for a uniprocessor has parallel aspects to the extent that it allows multiple programs to timeshare the machine or allows, say, computing, I/O, and printing to go on concurrently. Now, however, we contemplate a vast increase in the number of parallel activities, plus the operating system itself is being asked to run on a parallel machine and take advantage of its multiple computing resources.

Not surprisingly, this increased activity creates new difficulties. For example, a common way to prevent unconstrained accesses to a shared resource in a multiprogrammed uniprocessor is to serialize the accesses by making each a critical section (see Figure 5.5 on page 163). But in a parallel computer, critical sections become more and more of a problem as the degree of parallelism increases (see page 273). Chapter 7 talks about the critical sections in the Hydra operating system developed for the early, 16-processor C.mmp parallel computer (see also page 420). As an example of a more recent approach that anticipates much higher degrees of parallelism, the Ultracomputer section starting on page 430 describes parallel task queues, etc., based on an atomic "fetch-and-add" memory operation developed specially to eliminate critical sections. Such constructs are also being implemented in the RP3 (page 460). For a contrasting approach, the Rediflow section (page 384) sketches an interesting but yet untried distributed control scheme in which a pressure-like mechanism rather than a central task queue is used to balance the load among processors.

---

9 Working with one of the unfriendlier operating systems, a friend said "You ask it to do something, and it replies 'Let's arm-wrestle'; and two out of three times, you lose."

### 1.4.5 Choosing and Attacking the Problem

#### Applications Amenable to Parallel Processing

Which applications will provide enough parallelism to keep all these processors busy? In other words, "where's the parallelism?"

*Easy parallelism.* There are many examples of "easy parallelism" in which the work to be done can be divided into subtasks that have negligible communication with each other. One trivial example is ADP preparing the payroll for several independent companies; another is a group of people independently keeping track of the time. In both cases, the overall task can be assigned to multiple hardware entities (computers or watches) without worrying about communication among the entities. A more serious (scientific) application is some Monte Carlo application in which a large calculation is to be performed many times using different random inputs and the outputs are to be accumulated in a simple manner. Some database applications have the flavor of multiple, largely independent transactions, applied to a central database, which could be performed in parallel.[10] In general, however, problems are not so easily divided into independent pieces,[11] and the communication requirements between the subtasks play a key role. In these cases more care is needed to recast applications in parallel form. Chapter 2 discusses several representative application areas in which this has been done.

*Scientific and engineering calculations*, including flow dynamics (incompressible fluid), particle behavior, weather prediction (compressible fluid), and seismic modeling. Analysis at NYU [214] has shown that levels of parallelism in the thousands are available from the first three types of these calculations. The Cosmic Cube project at Caltech (page 361) and the users of the DAP machine at Queen Mary College (page 345) offer encouraging analytical and experimental results in this area. The 576-processor GF11 (page 325) at IBM Yorktown uses a parallel approach for a problem in this area (quantum chromo-dynamics or QCD; see QCD in the index).

*VLSI design automation*, including logic simulation, circuit simulation, shape checking, placement and wiring, logic synthesis (silicon compilation), and test generation, fault simulation, timing analysis, and logic equivalency analysis. The YSE experience (see page 479) demonstrates that at least 256-way parallelism is available in logic simulation, with a corresponding speedup that can approach 150 [266]. The degree of available parallelism is estimated to be in the thousands for this application as well as for circuit simulation, shape checking, and wiring (the WRM on page 387 demonstrated 64-way parallelism in chip wiring). Logic synthesis, with a higher degree of interdependencies, is felt to offer parallelism in the hundreds.

---

10 Obtaining the necessary I/O bandwidth may limit the number of concurrent transactions that can be processed, as may the coordination required to ensure that each transaction is atomic.

11 A favorite example, from Bob Newhart's old television show: Bob staggers home from a night with the boys and finds he forgot to defrost the turkey. "Put it in the oven at 2,000 degrees," suggests one friend, but the dial only goes up to 500. "OK, then use four ovens!"

*Database operations*, including parallelism among as well as within transactions. Teradata and Tandem are two parallel machines associated with this application.

*Near-term artificial intelligence* forms the fourth group of applications. The search procedures employed in "expert" (production) systems and other logic programming applications are expected by some to yield on the order of thousandfold parallelism. Present programs contain much lower degrees of parallelism [157], but this lower degree is said to be an artifact of being written for a serial computer (see page 63). DADO (page 378) is one project aimed at this application.

*Long-term artificial intelligence* applications, including perception/planning systems, visual recognition, natural language processing, motor control (robotics), and learning. These are all problems that appear suitable for a highly parallel computational model called "Computing with Connections" (page 67); parallelism on the order of tens of thousands appears to be available.

### What Parallel Computational Models Are Available?

As discussed in chapter 4, a computational model describes, at a level removed from hardware or software details, what a computer can do, i.e., which primitive actions can be performed, when these actions are performed, and by which methods data can be accessed and stored. In other words, a computational model gives the essential "rules of the game" for performing computations. As Tanenbaum [319] points out, a modern computer system consists of up to six levels of abstraction (including the machine hardware, operating system, and programming language), each representing a virtual computer with its own computational model. Thus, one must consider not only the model at each level, but also how well it can represent the actions of its neighbor in the hierarchy (see page 110).

SIMD and MIMD (Figure 1.1 and page 19) represent two major parallel computational models. In the MIMD case, in which a PE gets its instructions from its own stored program rather than from a central broadcast source, there is a further division into private memory and shared memory models, depending on whether each PE has exclusive access to a separate chunk of memory and must therefore communicate with other PEs via message passing[12] , or whether there is a memory to which *all* the PEs have direct access.

However, we have addressed only the data mechanism; there is also a control mechanism to be considered. Treleaven [328] breaks the latter down into control-driven, pattern-driven, demand-driven, and data-driven approaches, representing progressively less explicit control. The von Neumann model (page 108), for example, is control-driven, and instructions are executed exactly on schedule ("fire on command!"), whereas in dataflow (page 106), a data-driven model, instructions may execute as soon as their data are ready ("fire at will!").

These two data mechanisms and four control mechanisms can be used to classify most parallel architectures (see Figure 1.6 on page 23) as well as languages (Figure 5.8 on page 171). The advantages of the various models are discussed in

12 Fahlman [116] further subdivides message passers according to the size of the message.

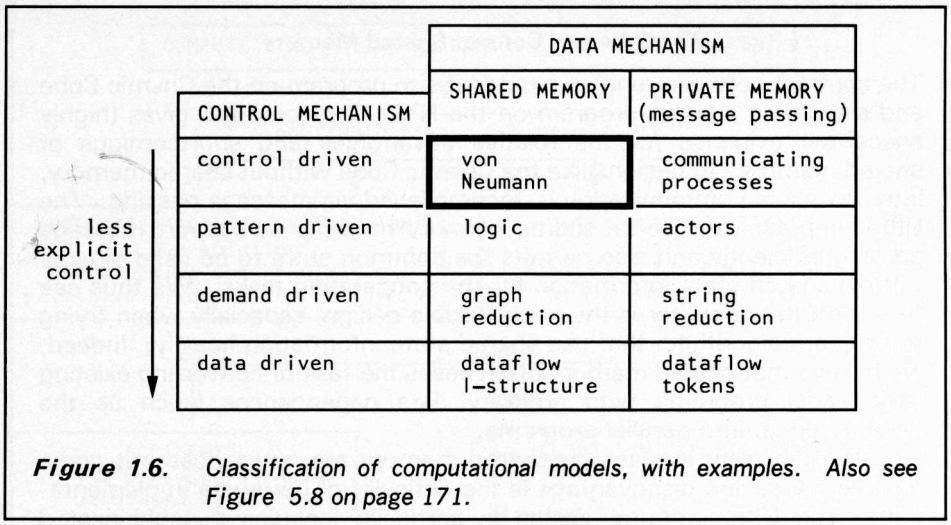


***Figure 1.6.*** *Classification of computational models, with examples. Also see Figure 5.8 on page 171.*

chapters 9, 10, and 11 in the context of specific embodiments. Shared and private memory designs are compared briefly in the box on page 24.

### What Algorithms Should Be Used?

As in the serial case, the choice of parallel algorithm depends on the computational model to be used and the problem to be solved. As pointed out on page 107, the computational model gives the costs associated with algorithms, including the time required for primitive operations and the space used for data objects. For parallel processing, there are additional charges that depend on the number of processors and the required interprocessor communication.

A parallel algorithm is a recipe for solving a given problem using a given computational model, and hence parallel algorithms too come in SIMD and MIMD varieties. In the MIMD case, there are further subdivisions: the distinction between shared-memory vs. message-passing communication that was introduced on page 23, and another distinction between synchronous vs. asynchronous algorithms that is discussed in chapter 4 on page 131.[13]

A fair amount of analysis has been published on parallel algorithms. Given N processors, the choice of algorithm can determine whether the speedup is a discouraging logN or a very encouraging N. The still-acceptable N/logN is frequently the best that one can do. Most of the analysis is for SIMD (Sameh [282], Heller [167], Gottlieb and Kruskal [152]). MIMD algorithms are discussed in an excellent paper by Kung [227].

13 These represent two different methods of handling communication among parallel processes, given that the process execution times are not predictable.

### The Pros and Cons of Shared Memory

The contrast between running a modest size program on the Cosmic Cube and simulating a larger program on the NYU Ultracomputer gives (highly anecdotal) evidence for the relative advantages and shortcomings of shared memory. In designs like the Cosmic Cube without shared memory, interprocessor communication is accomplished via message passing. The Ultracomputer can use its shared memory machine to support message passing efficiently and also permits the common store to be used directly to hold shared state information for the cooperating tasks. We thus see that the Ultracomputer is the more flexible design, especially when trying to program algorithms that use shared state information heavily. Indeed, we believe that shared memory often eases the task of converting existing large serial programs with arbitrary data dependences (such as the weather code) into parallel programs.

The flexibility available in shared memory machines does not come for free. A severe disadvantage is the difficulty of hardware implementation. The Ultracomputer design in particular includes a sophisticated processor to memory interconnection network. Since this network is not yet available, all the large scale parallelism results on the weather code are from a software simulator. In comparing the NYU and Caltech situations, one of the Ultracomputer designers was heard to say:

> "We're smart, they're smart, but while we're still designing chips, they're doing real science."

One must also note that even when the designs are complete, the Cosmic Cube will retain an advantage. For a given size configuration (i.e., for a given cost or component count) the shared memory will offer noticeably less peak computational power since a considerable number of its components will be devoted to the network. (The designs differed in other respects as well. For example, the Cosmic Cube's microprocessor, an INTEL 8086, does not support large programs as easily as the Ultracomputer's Motorola 68000. On the other hand, the 8086 had a floating point coprocessor well before the 68000.)

In summary, shared memory offers increased flexibility and programming ease at the cost of additional hardware complexity and thus lower peak performance for a given machine cost. Software implementation is easier, and hardware implementation is harder.

## How Much Speedup Can Be Expected?

As important as the algorithm is, it is only one factor that determines the actual program's running time. As Amdahl noted, the compute time can be divided into the *parallel portion* and the *serial portion*, and no matter how high the degree of parallelism in the former, the speedup will be asymptotically limited by the latter, which must be performed on a single processing element. Consider, for example, a sequence of 100 operations, 80 of which can be done in parallel, but 20 of which

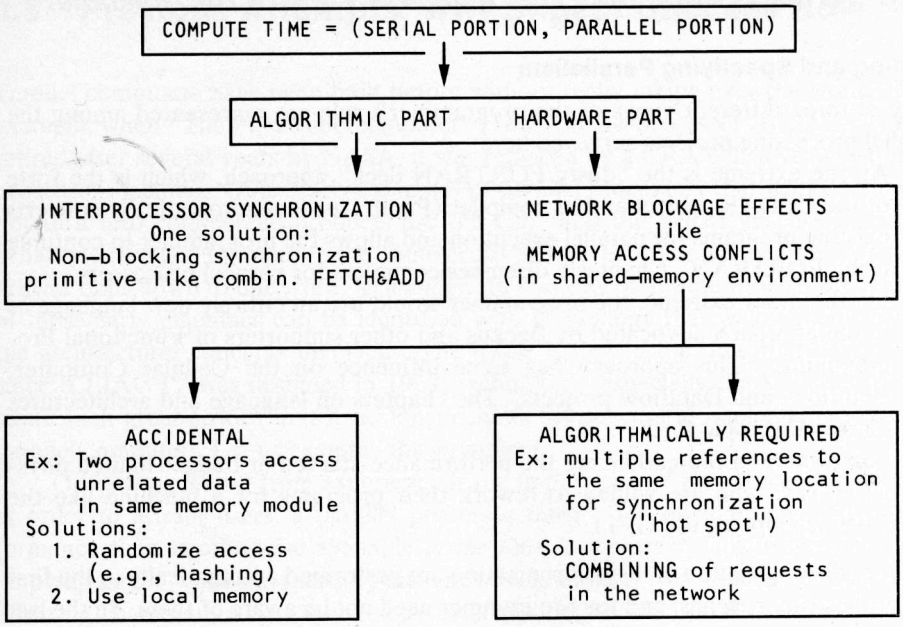


***Figure 1.7.*** *The serial portion of a parallel program's compute time. The program's speedup due to parallelism is limited by the serial portion, which is influenced by both the algorithm and the hardware design. This example shows the contributions in a shared memory environment, and lists the set of solutions used by the NYU Ultracomputer design.*

must be done in sequence. Compared to a single processing element, an 80-PE machine would attain a speedup of not 80 but somewhat under 5, even though the PE cost is 80 times higher.

Pfister [265] subdivides the serial portion into an *algorithmic* part, which refers to serialization intrinsic to the algorithm, and a *hardware-induced* part, which refers to serialization not required by the algorithm but induced by the hardware. Figure 1.7 shows these relationships in a diagrammatic way for a shared-memory computational model; also listed is one set of solutions, namely those used for the NYU Ultracomputer. However, these observations are quite general, and all computational models must supply some such set of solutions. A speedup of 1000 demands that the *total* serial portion be less than 0.1%. Evidence that this speedup is achievable for at least some important applications has been reported by several groups, including the Caltech Cosmic Cube and NYU Ultracomputer projects.

### 1.4.6 What Will Be the Programmer's View of the Machine?

#### Finding and Specifying Parallelism

At least three different views of the programmer's role are represented among the parallel processing projects surveyed here:

1. At one extreme is the "dusty FORTRAN deck" approach, which is the forte of the CEDAR project and its compiler (Parafrase) that automatically converts existing programs for parallel execution and allows the programmer to continue using FORTRAN. Chapter 6 discusses compilers for parallel processors.
2. On the other extreme, the programmer would use an entirely new language as in the approach advocated by Backus and other supporters of Functional Programming. This approach has some influence on the Cellular Computer, Rediflow, and Dataflow projects. The chapters on language and architectures discuss this topic further.
3. Somewhere in the middle are the performance-starved or budget-limited physicists, say, who are willing to rework their programs for a machine like the Cosmic Cube or the GF11.

Program decomposition and synchronization are performed automatically in the first two of these approaches, and the programmer need not be aware of them. In the last approach, however, some specific language constructs or synchronization primitives will be needed to specify how the computer is to apply parallel processing to our problem. Chapter 5 (Languages) treats this matter in some detail.

#### What About Reliability, Availability, Serviceability?

A fair amount of work has been done on fault-tolerant interconnection networks, and we show one example (the "extra-stage cube") in Figure 8.11 on page 299. But this is only part of what is needed. A parallel system introduces two new elements: it can fail in new ways, and potentially, at least, the multiple processors permit it to keep running (at degraded performance) in the face of many failures that would cripple a uniprocessor. Both of these new factors pose significant new challenges to the system software. We discuss some of these points further in our section on the Tandem system (page 376). However, a great deal of work is needed in this area.

## 1.5 Previous Attempts; Why Expect Success Now?

Parallel computers have been built before without really taking over the world. For example, when "The First Supercomputer" [182], the 64-processor ILLIAC IV, was retired after several years by NASA, it was replaced by a CRAY-1, a very fast one-processor machine. Other early examples of parallel processors that worked but did not turn into commercial products are the Cm* and the WRM, discussed below. What has changed to improve the commercial prospects of parallel computers? The answer boils down to one word: *technology.* In one way or another, the shortcomings of these early machines can be identified with technology that was inadequate for the architectural concepts involved. The thesis is that in the time that has passed since ILLIAC IV was designed in 1967, technology (especially the VLSI chip) has made such great strides that it no longer limits the successful implementation of a parallel computer. Let us examine this argument.

One strong lesson from experience is the importance of raw processing power; in order to attract users, a parallel processor must offer substantially better performance than is otherwise available *at the time it appears.*[14] This is similar to the "moving target" problem faced by most new technologies: like the task of unseating a boxing champion, a new technology must not only catch but also *surpass* an entrenched technology in order to replace it, which is very difficult while the entrenched technology is still making good progress. The von Neumann computer will probably always be with us, but at the high-performance end, the rate of progress has slowed down as physical effects like the speed of light have become more and more of a problem. And the designer of a parallel computer has ammunition now that wasn't available before.

The designer has a choice between off-the-shelf and custom-built processing elements. In the past, commercially available microprocessors were so slow that a huge number would have been needed for high performance. On the other hand, a smaller assemblage suffered not only in raw performance but also in the interconnection technology that could be justified with this lower performance, a sort of double whammy. This dilemma affected both the Cm*, a collection of 50 16-bit microcomputers, and the WRM, a collection of 64 8-bit microprocessors. The Cm* also suffered from the high communication overhead of its microprogrammed hierarchical network, and from the small memory space (16 address bits, 64KB) of its processors; the latter condition meant that programs too large to fit into the memory had to be loaded in and out in pieces using "overlay" techniques. The WRM suffered from the inflexible communications imposed by a two-dimensional mesh interconnection as well as from the fact that only Z80 assembly language was available for programming it, a consequence of the fact that its performance did not surpass the best serial computers of its day. A compiler to a high level language would

14 This remark does not apply to a system on which existing application programs can be run without modification.

have degraded the performance further, and hence the work required to write one was hard to justify.

These two computers were built primarily for experimental purposes. The ILLIAC IV, on the other hand, was designed to be a computing workhorse, a true supercomputer. It actually preceded the other two, and the arguments above called for a custom processor design to get the necessary performance. Both the processor and the memory used very aggressive technology for the day. The processor was able to perform a floating point addition in 250 nanoseconds, corresponding to a peak instantaneous processing rate of 4 million floating point operations per second (MFLOPS). Theoretically, the full 256-processor machine would have had a peak instantaneous performance over 1000 MFLOPS. (The *sustained* performance, of course, was much less.) The processor was built from discrete components; in that pre-VLSI era, the designers did not have to consider the alternate choice of a VLSI design and the extra delays associated with early VLSI design systems. The large number of discrete components, however, resulted in a serious reliability problem. Today, microprocessors are so powerful that many fewer are required to achieve high performance, and better interconnection networks are more justified (as well as more affordable).

To keep pace with the processors, the ILLIAC IV memory had to be much faster than the predominant core memories of the day; magnetic films were planned first, but the emerging semiconductor memory technology was eventually chosen. The amount of memory was amazingly small by today's standards: each processor had only 16KB, giving the whole machine just 1 MB. To put this into perspective: the NEC SX (page 317) has roughly 10 times the peak performance of ILLIAC IV, but has 256 times as much memory, a testimony to the progress in memory technology over the last fifteen years.[15]

Another lesson from experience is the importance of flexible, general, fast communications. Past parallel processors either assumed more regularity (two-dimensional mesh) than there really is in most problems (ILLIAC IV, WRM), or provided global communication that was too slow (Cm*). Today, the communications problem is easier to solve because a much smaller number of processors is needed for a given performance. Better communications networks with functions like buffering and combining of messages at reasonable cost have been made possible by VLSI. And communications concepts not previously applied to computers are now available (binary N-cube, omega network, etc.).

It seems, therefore, that hardware no longer represents a major stumbling block to parallel processing and that the question becomes how practical is a parallel model of computation? How much parallelism is there in applications, and how well can software exploit it? These questions are explored in the next several chapters.

---

15 One might expect this to lead to a substantially better ratio of sustained to peak performance. Interestingly enough, this is not so in this case. SX obtains about 290 MFLOPS on the "Livermore loops" benchmark, or about 22% of peak, in the same range as ILLIAC IV. The SIZE of the problem that can be handled, of course, is much larger.

## 1.6 Conclusions and Future Directions

To summarize the foregoing discussion, we need a new way to think about computing. The sequential architecture devised by von Neumann was a brilliantly successful match to the technology of the 1940s and several decades thereafter, when the processor and memory were both expensive, precious resources. By now, however, there have been million-fold improvements in the speed, cost, and power dissipation of processors as well as a million-fold improvement in the cost of memories, and these two technologies no longer stand in the way of a many-processor parallel computer. The networks needed to provide adequate connections between the elements of such a computer are still a formidable undertaking, but this situation, too, is rapidly being improved by the progress in technology. Now we have to learn the best way to capitalize on all this.

Parallel processing is essentially a divide-and-conquer approach to problem solving, and there are many ways to divide up a problem. Each has its own overhead costs. Put another way, parallelism can be perceived at a number of levels of program execution. An important consideration is the *cost of achieving this parallelism*, which will set the lower bound on the size of the code segment that is profitable to run on a given system. As Lorin [240] points out, it is manifestly unprofitable to execute 50 microseconds of task establishment code to enable the execution of a 3-microsecond sequence in parallel with a 5-microsecond sequence in a multiple processor system. A subtask profitably done in parallel on an MIMD multiprocessor like the Butterfly comprises a reasonably large number of instructions. In a parallel SIMD vector processor, on the other hand, the unit of work is considerably smaller (an instruction stage) because there is essentially no cost of parallelism, no overhead involved in starting up a successor instruction. The cost here is the expense of designing and implementing the capability. In between these extremes lie a number of potential subtask sizes, some not previously seen in existing serial computers.

To make a choice among these possible task sizes or "levels of granularity", we need to determine the best ways to detect their existence and then express and control their parallel execution. Translated into computer terminology, major work is required in compilers, languages, and operating systems. These will be the critical research areas of the future in parallel computing. Some of their outline was sketched in the last sections. Meaningful progress will require bringing together significantly large programs and significantly powerful parallel processors. This is the challenge facing the search for the best way to do parallel processing.