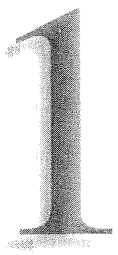
CHAPTER



Motivation and History

Well done is quickly done.
Caesar Augustus

1.1 INTRODUCTION

Are you one of those people for whom "fast" isn't fast enough? Today's workstations are about a hundred times faster than those made just a decade ago, but some computational scientists and engineers need even more speed. They make great simplifications to the problems they are solving and still must wait hours, days, or even weeks for their programs to finish running.

Faster computers let you tackle larger computations. Suppose you can afford to wait overnight for your program to produce a result. If your program suddenly ran 10 times faster, previously out-of-reach computations would now be within your grasp. You could produce in 15 hours an answer that previously required nearly a week to generate.

Of course, you *could* simply wait for CPUs to get faster. In about five years single CPUs will be 10 times faster than they are today (a consequence of Moore's Law). On the other hand, if you can afford to wait five years, you must not be in that much of a hurry! Parallel computing is a proven way to get higher performance *now*.

What's parallel computing?

Parallel computing is the use of a parallel computer to reduce the time needed to solve a single computational problem. Parallel computing is now considered a standard way for computational scientists and engineers to solve problems in areas as diverse as galactic evolution, climate modeling, aircraft design, and molecular dynamics.

What's a parallel computer?

A **parallel computer** is a multiple-processor computer system supporting parallel programming. Two important categories of parallel computers are multicomputers and centralized multiprocessors.

As its name implies, a **multicomputer** is a parallel computer constructed out of multiple computers and an interconnection network. The processors on different computers interact by passing messages to each other.

In contrast, a **centralized multiprocessor** (also called a **symmetrical multiprocessor** or **SMP**) is a more highly integrated system in which all CPUs share access to a single global memory. This shared memory supports communication and synchronization among processors.

We'll study centralized multiprocessors, multicomputers, and other parallel computer architectures in Chapter 2.

What's parallel programming?

Parallel programming is programming in a language that allows you to explicitly indicate how different portions of the computation may be executed concurrently by different processors. We'll discuss various kinds of parallel programming languages in more detail near the end of this chapter.

Is parallel programming really necessary?

A lot of research has been invested in the development of compiler technology that would allow ordinary Fortran 77 or C programs to be translated into codes that would execute with good efficiency on parallel computers with large numbers of processors. This is a very difficult problem, and while many experimental parallelizing[1] compilers have been developed, at the present time commercial systems are still in their infancy. The alternative is for you to write your own parallel programs.

Why should I program using MPI and OpenMP?

MPI (Message Passing Interface) is a standard specification for message-passing libraries. Libraries meeting the standard are available on virtually every parallel computer system. Free libraries are also available in case you want to run MPI on a network of workstations or a parallel computer built out of commodity components (PCs and switches). If you develop programs using MPI, you will be able to reuse them when you get access to a newer, faster parallel computer.

Increasingly, parallel computers are being constructed out of symmetrical multiprocessors. Within each SMP, the CPUs have a shared address space. While MPI is a perfectly satisfactory way for processors in different SMPs to communicate with each other, OpenMP is a better way for processors within a single SMP

[1] **parallelize** *verb*: to make parallel.

to interact. In Chapter 18 you'll see an example of how a hybrid MPI/OpenMP program can outperform an MPI-only program on a practical application.

By working through this book, you'll learn a little bit about parallel computer hardware and a lot about parallel program development strategies. That includes parallel algorithm design and analysis, program implementation and debugging, and ways to benchmark and optimize your programs.

1.2 MODERN SCIENTIFIC METHOD

Classical science is based on observation, theory, and physical experimentation. Observation of a phenomenon leads to a hypothesis. The scientist develops a theory to explain the phenomenon and designs an experiment to test that theory. Usually the results of the experiment require the scientist to refine the theory, if not completely reject it. Here, observation may again take center stage.

Classical science is characterized by physical experiments and models. For example, many physics students have explored the relationship between mass, force, and acceleration using paper tape, pucks, and air tables. Physical experiments allow scientists to test theories, such as Newton's first law of motion, against reality.

In contrast, contemporary science is characterized by observation, theory, experimentation, *and numerical simulation* (Figure 1.1). Numerical simulation is an increasingly important tool for scientists, who often cannot use physical experiments to test theories because they may be too expensive or time-consuming, because they may be unethical, or because they may be impossible to perform. The modern scientist compares the behavior of a numerical simulation, which

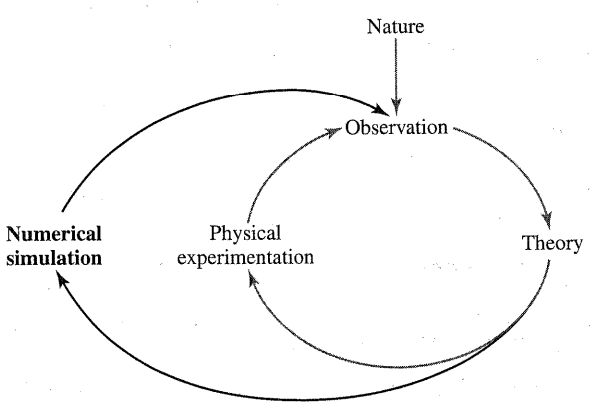


Figure 1.1 The introduction of numerical simulation distinguishes the contemporary scientific method from the classical scientific method.

implements the theory, to data collected from nature. The differences cause the scientist to revise the theory and/or make more observations.

Many important scientific problems are so complex that solving them via numerical simulation requires extraordinarily powerful computers. These complex problems, often called **grand challenges** for science, fall into several categories [73]:

1. Quantum chemistry, statistical mechanics, and relativistic physics
2. Cosmology and astrophysics
3. Computational fluid dynamics and turbulence
4. Materials design and superconductivity
5. Biology, pharmacology, genome sequencing, genetic engineering, protein folding, enzyme activity, and cell modeling
6. Medicine, and modeling of human organs and bones
7. Global weather and environmental modeling

While grand challenge problems emerged in the late 1980s as a stimulus for further developments in high-performance computing, you can view the entire history of electronic computing as the quest for higher performance.

1.3 EVOLUTION OF SUPERCOMPUTING

The United States government has played a key role in the development and use of high-performance computers. During World War II the U.S. Army paid for the construction of the ENIAC in order to speed the calculation of artillery tables. In the 30 years after World War II, the U.S. government used high-performance computers to design nuclear weapons, break codes, and perform other national security–related applications.

Supercomputers are the most powerful computers that can be built [60]. (As computer speeds increase, the bar for "supercomputer" status rises, too.) The term *supercomputer* first came into widespread use with the introduction of the Cray-1 supercomputer in 1976. The Cray-1 was a pipelined vector processor, not a multiple-processor computer, but it was capable of more than 100 million floating point operations per second.

Supercomputers have typically cost $10 million or more. The high cost of supercomputers once meant they were found almost exclusively in government research facilities, such as Los Alamos National Laboratory.

Over time, however, supercomputers began to appear outside of government facilities. In the late 1970s supercomputers showed up in capital-intensive industries. Petroleum companies harnessed supercomputers to help them look for oil, and automobile manufacturers started using these systems to improve the fuel efficiency and safety of their products.

Ten years later, hundreds of corporations around the globe were using supercomputers to support their business enterprises. The reason is simple: for many

businesses, quicker computations lead to a competitive advantage. More rapid crash simulations can reduce the time an automaker needs to design a new car. Faster drug design can increase the number of patents held by a pharmaceutical firm. High-speed computers have even been used to design products as mundane as disposable diapers!

Computing speeds have risen dramatically in the past 50 years. The ENIAC could perform about 350 multiplications per second. Today's supercomputers are more than a billion times faster, able to perform trillions of floating point operations per second.

Single processors are about a million times faster than they were 50 years ago. Most of the speed increase is due to higher clock rates that enable a single operation to be performed more quickly. The remaining speed increase is due to greater system concurrency: allowing the system to work simultaneously on multiple operations. The history of computing has been marked by rapid progress on both these fronts, as exemplified by contemporary high-performance microprocessors. Intel's Pentium 4 CPU, for example, has clock speeds well in excess of 1 GHz, two arithmetic-logic units (ALUs) clocked at twice the core processor clock speed, and extensive hardware support for out-of-order speculative execution of instructions.

How can today's supercomputers be a billion times faster than the ENIAC, if individual processors are only about a million times faster? The answer is simple: the remaining thousand-fold speed increase is achieved by collecting thousands of processors into an integrated system capable of solving individual problems faster than a single CPU; i.e., a parallel computer.

The meaning of the word *supercomputer,* then, has changed over time. In 1976 *supercomputer* meant a Cray-1, a single-CPU computer with a high-performance pipelined vector processor connected to a high-performance memory system. Today, *supercomputer* means a parallel computer with thousands of CPUs.

The invention of the microprocessor is a watershed event that led to the demise of traditional minicomputers and mainframes and spurred the development of low-cost parallel computers. Since the mid-1980s, microprocessor manufacturers have improved the performance of their top-end processors at an annual rate of 50 percent while keeping prices more or less constant [90]. The rapid increase in microprocessor speeds has completely changed the face of computing. Microprocessor-based servers now fill the role formerly played by minicomputers constructed out of gate arrays or off-the-shelf-logic. Even mainframe computers are being constructed out of microprocessors.

1.4 MODERN PARALLEL COMPUTERS

Parallel computers only became attractive to a wide range of customers with the advent of Very Large Scale Integration (VLSI) in the late 1970s. Supercomputers such as the Cray-1 were far too expensive for most organizations. Experimental parallel computers were less expensive than supercomputers, but they were still relatively costly. They were unreliable, to boot. VLSI technology allowed

computer architects to reduce the chip count to the point where it became possible to construct affordable, reliable parallel systems.

1.4.1 The Cosmic Cube

In 1981 a group at Caltech led by Charles Seitz and Geoffrey Fox began work on the Cosmic Cube, a parallel computer constructed out of 64 Intel 8086 microprocessors [34]. They chose the Intel 8086 because it was the only microprocessor available at the time that had a floating-point coprocessor, the Intel 8087. The complete 64-node system became operational in October 1983, and it dramatically illustrated the potential for microprocessor-based parallel computing. The Cosmic Cube executed its application programs at about 5 to 10 million floating point operations per second (5 to 10 **megaflops**). This made the Cosmic Cube 5 to 10 times the speed of a Digital Equipment Corporation VAX 11/780, the standard research minicomputer of the day, while the value of its parts was less than half the price of a VAX. In other words, the research group realized a price-performance jump of between 10 and 20 times by running their programs on a "home-made" parallel computer rather than a VAX. The Cosmic Cube was reliable, too; it experienced only two hard failures in its first year of operation.

Intel Corporation had donated much of the hardware for the Cosmic Cube. When it sent employee John Palmer to Caltech to see what Seitz and Fox had done, Palmer was so impressed he left Intel to start his own parallel computer company, nCUBE. Intel's second delegation, led by Justin Rattner, was equally impressed. Rattner became the technical leader of a new Intel parallel computer division called Intel Scientific Supercomputing.

1.4.2 Commercial Parallel Computers

Commercial parallel computers manufactured by Bolt, Beranek and Newman (BBN) and Denelcor were available before the Cosmic Cube was completed, but the Cosmic Cube stimulated a flurry of new activity. Table 1.1 is a list of just a few of the many organizations that jumped into the fray [116].

Companies from around the world began selling parallel computers. Intel's Supercomputer Systems Division and small start-up firms, such as Meiko, nCUBE, and Parsytec, led the way, while more established computer companies (IBM, NEC, and Sun Microsystems) waited until the field had become more mature. It is interesting to note that even Cray Research, Inc., famous for its custom, very high-performance, pipelined CPUs, eventually introduced a microprocessor-based parallel computer, the T3D, in 1993.

Other companies produced parallel computers with a single CPU and thousands of arithmetic-logic units (ALUs) implemented in VLSI. The most famous of these computers was the Connection Machine, built by Thinking Machines Corporation. This massively parallel computer, first shipped in 1986, contained 65,536 single-bit ALUs.

By the mid-1990s most of the companies on our list had either gotten out of the parallel computer business, gone bankrupt, or been purchased by larger

Table 1.1 Some of the organizations that delivered commercial parallel computers based on microprocessor CPUs in the 10-year period 1984–1993 and their current status.

Company	Country	Year	Status in 2001
Sequent	U.S.	1984	Acquired by IBM
Intel	U.S.	1984	Out of the business*
Meiko	U.K.	1985	Bankrupt
nCUBE	U.S.	1985	Out of the business
Parsytec	Germany	1985	Out of the business
Alliant	U.S.	1985	Bankrupt
Encore	U.S.	1986	Out of the business
Floating Point Systems	U.S.	1986	Acquired by Sun
Myrias	Canada	1987	Out of the business
Ametek	U.S.	1987	Out of the business
Silicon Graphics	U.S.	1988	Active
C-DAC	India	1991	Active
Kendall Square Research	U.S.	1992	Bankrupt
IBM	U.S.	1993	Active
NEC	U.S.	1993	Active
Sun Microsystems	U.S.	1993	Active
Cray Research	U.S.	1993	Active (as Cray Inc.)

*"Out of the business" means the company is no longer selling general-purpose parallel computer systems.

Hewlett-Packard, IBM, Digital Equipment Corporation, Silicon Graphics, and Sun Microsystems all had parallel computers in their product lines by the mid-1990s.

These commercial systems ranged in price from several hundred thousand dollars to several million dollars. Compared to a commodity PC, the price per CPU in a commercial parallel computer was high, because these systems contained custom hardware to support either shared memory or low-latency, high-bandwidth interprocessor communications.

Some commercial parallel computers had support for higher-level parallel programming languages and debuggers, but the rapid evolution in the underlying hardware of parallel systems, even those manufactured by the same vendor, meant their systems programmers were perpetually playing catch-up. For this reason systems programming tools for commercial parallel computers were usually primitive, with the consequence that researchers found themselves programming these systems using the "least common denominator" approach of C or FORTRAN combined with a standard message-passing library, typically PVM or MPI. Vendors focused their efforts on penetrating the large commercial market, rather than serving the needs of the relatively puny scientific computing market. Hence computational scientists seeking peak performance from commercial parallel systems often felt they received inadequate support from vendors, and so they adopted a do-it-yourself attitude.

1.4.3 Beowulf

Meanwhile, driven by the popularity of personal computing for work and enter-

performance and razor-thin profit margins. The dynamic PC marketplace set the stage for the next breakthrough in parallel computing.

In the summer of 1994, at NASA's Goddard Space Flight Center, Thomas Sterling and Don Becker built a parallel computer entirely out of commodity hardware and freely available software. Their system, named Beowulf, contained 16 Intel DX4 processors connected by multiple 10 Mbit/sec Ethernet links. The cluster ran the Linux operating system, used GNU compilers, and supported parallel programming with the MPI message-passing library—all freely available software.

The high-performance computing research community rapidly embraced the Beowulf philosophy. At the Supercomputing '96 conference, both NASA and the Department of Energy demonstrated Beowulf clusters costing less than $50,000 that achieved greater than 1 billion floating point operations per second (1 **gigaflop**) performance on actual applications. At the Supercomputing '97 conference, Caltech demonstrated a 140-node cluster running an n-body simulation at greater than 10 gigaflops.

Beowulf is an example of a system constructed out of commodity, off-the-shelf (**COTS**) components. Unlike commercial systems, commodity clusters typically are not balanced between compute speed and communication speed: the communication network is usually quite slow compared to the speed of the processors. However, for many applications that are dominated by computations, clusters can achieve much better performance per dollar than commercial parallel computers. Because the latest CPUs typically appear in PCs months before they are available in commercial parallel computers, it is possible to construct a commodity cluster with newer, higher-performance CPUs than those available in a commercial parallel system. Commodity clusters have the additional, significant advantage of a low entry cost, which has made them a popular platform for academic institutions.

1.4.4 Advanced Strategic Computing Initiative

Meanwhile, the United States government has created an ambitious plan to build a series of five supercomputers costing up to $100 million each. This effort is motivated by the moratorium on underground nuclear testing signed by President Bush in 1992 and extended by President Clinton in 1993, as well as the decision by the United States to halt production of new nuclear weapons. As a result, the U.S. plans to maintain its stockpile of existing weapons well beyond their originally planned lifetimes. Sophisticated numerical simulations are required to guarantee the safety, reliability, and performance of the nuclear stockpile. The U.S. Department of Energy's Advanced Strategic Computing Initiative (**ASCI**) is developing a series of ever-faster supercomputers to execute these simulations.

The first of these supercomputers, ASCI Red, was delivered to Sandia National Laboratories in 1997. With just over 9,000 Intel Pentium II Xeon CPUs, it was the first supercomputer to sustain more than 1 trillion operations per second (1 **teraop**) on production codes. (Intel dropped out of the supercomputer busi-

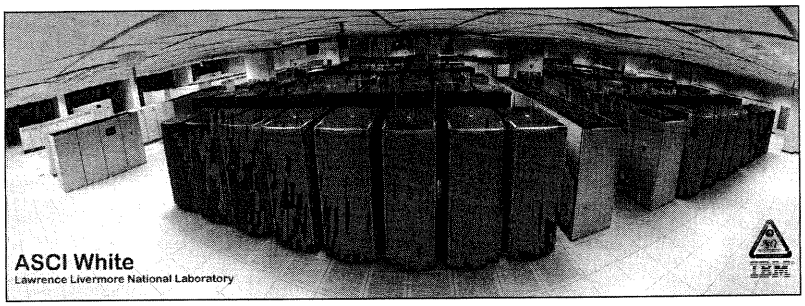


Figure 1.2 The ASCI White supercomputer at Lawrence Livermore National Laboratory contains 8192 PowerPC CPUs and is capable of sustaining more than 10 trillion operations per second on production programs. It was the fastest computer in the world in the year 2000. (Photo courtesy Lawrence Livermore National Laboratory)

California received delivery of the second supercomputer in the series, ASCI Blue Pacific, from IBM in 1998. It consists of 5,856 PowerPC CPUs and is capable of sustained performance in excess of 3 teraops.

In 2000 IBM delivered the third ASCI supercomputer, ASCI White, to the Lawrence Livermore National Laboratory (Figure 1.2). ASCI White actually is composed of three separate systems. The production system is an SMP-based multicomputer. It has 512 nodes; each node is an SMP with 16 PowerPC CPUs. The aggregate speed of the 8,192 CPUs has been benchmarked at more than 10 teraops on a computation of interest.

If the U.S. Department of Energy maintains this pace, tripling the performance of its ASCI supercomputers every two years, it will meet its goal of installing a 100 teraops computer by 2004.

1.5 SEEKING CONCURRENCY

As we have seen, parallel computers are more available than ever, but in order to take advantage of multiple processors, programmers and/or compilers must be able to identify operations that may be performed in parallel (i.e., concurrently).

1.5.1 Data Dependence Graphs

A formal way to identify parallelism in an activity is to draw a data dependence graph. A **data dependence graph** is a directed graph in which each vertex represents a task to be completed. An edge from vertex u to vertex v means that task u must be completed before task v begins. We say that "Task v is dependent on task u." If there is no path from u to v, then the tasks are **independent** and may be performed concurrently.

As an analogy, consider the problem of performing an estate's weekly land-

Landscape, Inc. (Figure 1.3a). His goal is to complete the four principal tasks—mowing the lawns, edging the lawns, weeding the gardens, and checking the sprinklers—as quickly as possible. Mowing must be completed before the sprinklers are checked. (Think of this as a dependence involving the four lawns as shared "variables." The lawns may take on the value "wet and cut" only after they have taken on the value "cut.") Similarly, edging and weeding must be completed before the sprinklers are checked. However, mowing, edging, and weeding may be done concurrently. Someone must also turn off the security system before the crew enters the estate and turn the system back on when the crew leaves. Allan represents these tasks using a dependence graph (Figure 1.3b).

Knowing the relative sizes of the respective jobs and the capabilities of his employees, Allan decides four crew members should mow the lawn while two crew members edge the lawn and two other crew members weed the gardens (Figure 1.3c).

Three different task patterns appear in Figure 1.3c. Figure 1.4 illustrates each of these patterns in isolation. The labels inside the circles represent the kinds of tasks being performed. Multiple circles with the same label represent tasks performing the same operation on different operands.

1.5.2 Data Parallelism

A data dependence graph exhibits **data parallelism** when there are independent tasks applying the same operation to different elements of a data set (Figure 1.4a).

Here is an example of fine-grained data parallelism embedded in a sequential algorithm:

```
for i ← 0 to 99 do
  a[i] ← b[i] + c[i]
endfor
```

The same operation—addition—is being performed on the first 100 elements of arrays b and c, with the results being put in the first 100 elements of a. All 100 iterations of the loop could be executed simultaneously.

1.5.3 Functional Parallelism

A data dependence graph exhibits **functional parallelism** when there are independent tasks applying different operations to different data elements (Figure 1.4b).

Here is an example of fine-grained functional parallelism embedded in a sequential algorithm:

$a \leftarrow 2$
$b \leftarrow 3$
$m \leftarrow (a + b)/2$
$s \leftarrow (a^2 + b^2)/2$
$v \leftarrow s - m^2$

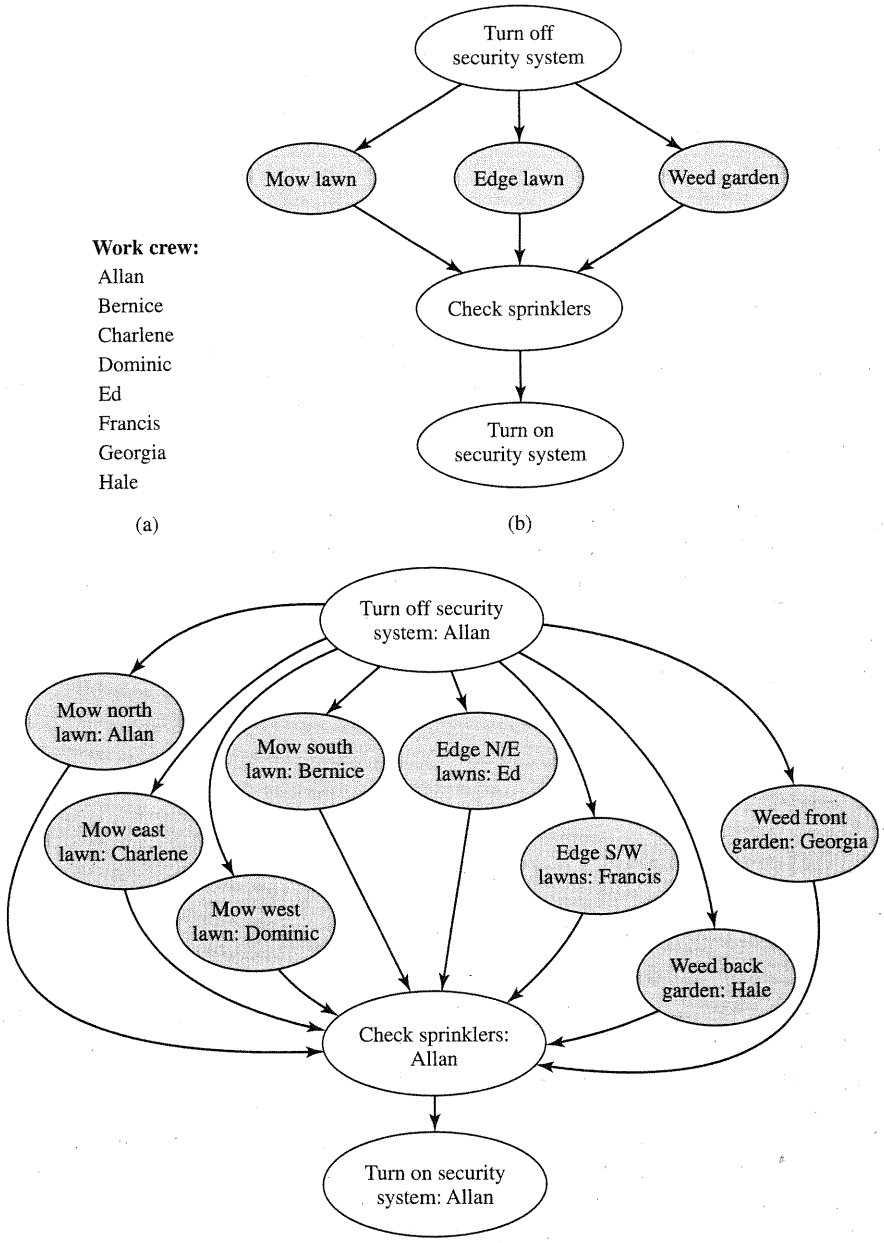


Figure 1.3 Most realistic problems have data parallelism, functional parallelism, and precedence constraints between tasks. (a) An eight-person work crew is responsible for landscape maintenance at Medici Manor. (b) A data dependence graph shows which tasks must be completed before others begin. If there is no path from vertex u to vertex v, the tasks may proceed concurrently (functional parallelism). (c) The larger tasks have been divided into subtasks, in which several employees perform the same activity on different portions of the estate (data

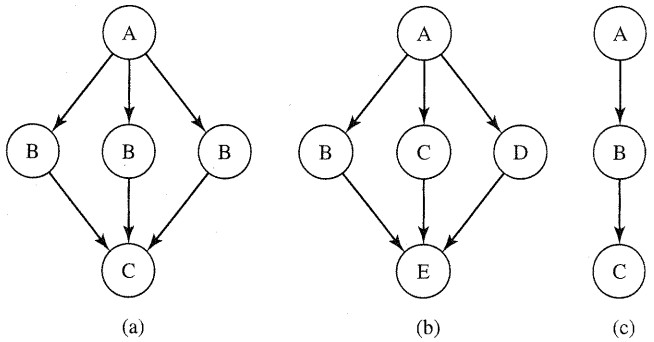


Figure 1.4 Parallelism in data dependence graphs. Vertices represent tasks. The letter inside a vertex indicates the operation being performed. Edges denote dependences among tasks. (a) A graph exhibiting data parallelism. Three tasks may concurrently apply operation B to different operands. (b) A graph exhibiting functional parallelism. Tasks performing operations B, C, and D may be performed concurrently. (c) A purely sequential dependence graph. However, if all tasks take the same amount of time to execute and multiple problem instances need to be processed, operation C may be performed on instance i while operation B is performed on instance $i + 1$ and operation A is performed on instance $i + 2$. This structure is called pipelining.

The third and fourth statements—the assignments to m and s—depend solely on values a and b and could be performed concurrently.

1.5.4 Pipelining

A data dependence graph forming a simple path or chain (as in Figure 1.4c) admits no parallelism if only a single problem instance must be processed. However, a computation that has been divided up into stages can support concurrency equal to the number of stages if multiple problem instances need to be processed. A **pipelined computation** is analogous to an assembly line (Figure 1.5). At any moment each stage is working on a particular part of the computation. The output of one stage is the input of the next.

For example, suppose automobiles really did get constructed on a four-stage assembly line, where each stage required one hour. Beginning with an empty assembly line, it would take four hours for the first automobile to be assembled. However, at that point the assembly line would be full, and the second automobile would be completed one hour later. The kth automobile would be completed at hour $3 + k$.

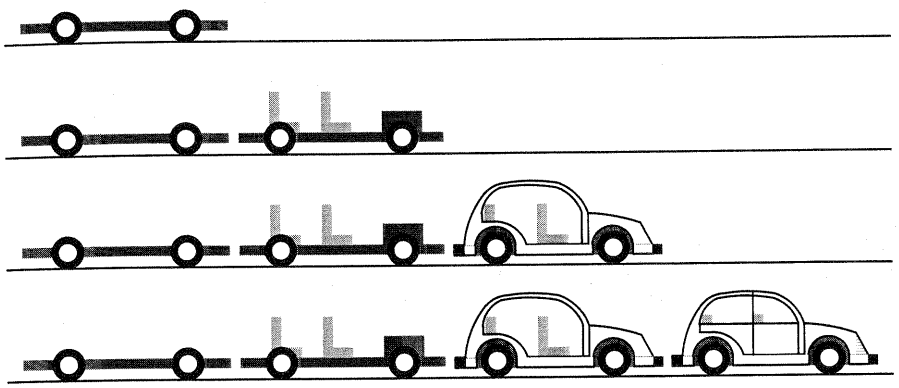


Figure 1.5 An automobile assembly line is an example of a pipeline.

Let's consider pipelining in the context of a **for** loop computing partial sums

$$\begin{aligned} p_0 &\leftarrow a_0 \\ p_1 &\leftarrow a_0 + a_1 \\ p_2 &\leftarrow a_0 + a_1 + a_2 \\ p_3 &\leftarrow a_0 + a_1 + a_2 + a_3 \end{aligned}$$

This can be accomplished by the loop:

```
p[0] ← a[0]
for i ← 1 to 3 do
  p[i] ← p[i − 1] + a[i]
endfor
```

The loop is not data parallel, because computing $p[i]$ depends on the value of $p[i-1]$. However, we can break the loop into individual stages or segments:

```
p[0] ← a[0]
p[1] ← p[0] + a[1]
p[2] ← p[1] + a[2]
p[3] ← p[2] + a[3]
```

The resulting pipeline (Figure 1.6) could be used to compute multiple sets of partial sums in an assembly-line fashion.

1.5.5 Size Considerations

These examples of data parallelism, functional parallelism, and pipelining are for expository purposes only. So few operations are involved that it would not be worth the trouble to make them run on different CPUs of a parallel computer. In

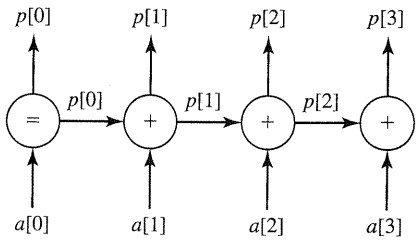


Figure 1.6 A pipeline to compute partial sums. Each circle represents a process. The leftmost stage inputs $a[0]$, outputs its value as $p[0]$, and passes $p[0]$ to the next stage. All other stages i input $a[i]$, collect $p[i-1]$ from their predecessors, add the two values, and output the sum as $p[i]$.

this book, we'll be looking for sources of parallelism in problems requiring far more computations.

1.6 DATA CLUSTERING

Let's consider a practical example of a computationally intensive problem and try to find opportunities for parallelism.

Modern computer systems are capable of collecting and storing extraordinary amounts of data. For example, the World Wide Web contains hundreds of millions of pages. U.S. Census data constitute another very large dataset. Using a computer system to access salient facts or detect meaningful patterns is variously called **data mining** or **scientific data analysis**. Data mining is a compute-intensive, "off-line" operation, in contrast to data retrieval, which is an I/O-intensive, "on-line" operation.

Multidimensional data clustering is an important tool for data mining. **Data clustering** is the process of organizing a dataset into groups, or clusters, of "similar" items. Clustering makes it easier to find additional items closely related to an item of interest.

Suppose we have a collection of N text documents. We will examine each document to come up with an estimate of how well it covers each of D different topics, and we will assign each document to one of K different clusters, where each cluster contains "similar" documents. See Figure 1.7. A performance function indicates how well clustered the documents are. Our goal is to optimize the value of the performance function.

Figure 1.8 contains a high-level description of a sequential algorithm to solve the data clustering problem. How could parallelism be used to speed the execution of this algorithm?

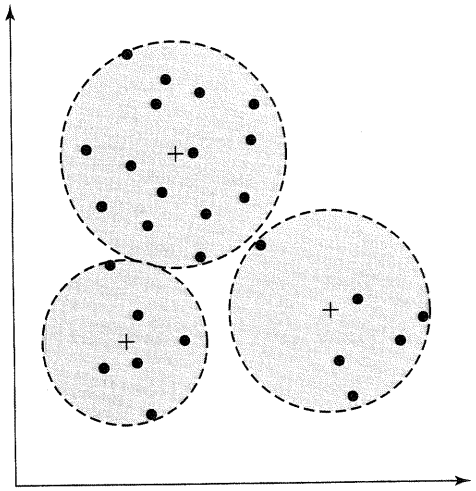


Figure 1.7 An example of document clustering for which $N = 20$, $D = 2$, and $K = 3$. There are 20 documents (represented by black dots). We measure each document's coverage of two topics (hence each document is represented by a point in two-dimensional space). The documents are organized into three clusters (centered around the crosses). Can you find a better clustering?

Data Clustering:

1. Input N documents
2. For each of the N documents generate a D-dimensional vector indicating how well it covers the D different topics
3. Choose the K initial cluster centers using a random sample
4. Repeat the following steps for I iterations or until the performance function converges, whichever comes first:
   (a) For each of the N documents, find the closest center and compute its contribution to the performance function
   (b) Adjust the K cluster centers to try to improve the value of the performance function
5. Output K centers

Figure 1.8 A sequential algorithm to find K centers that optimally categorize N documents.

Our first step in the analysis is to draw a data dependence graph. While we could draw one vertex for each step in the pseudocode algorithm, it is better to draw a vertex for each step of the algorithm *for each document or cluster center,* because it exposes more opportunities for parallelism. The resulting data dependence graph appears in Figure 1.9.

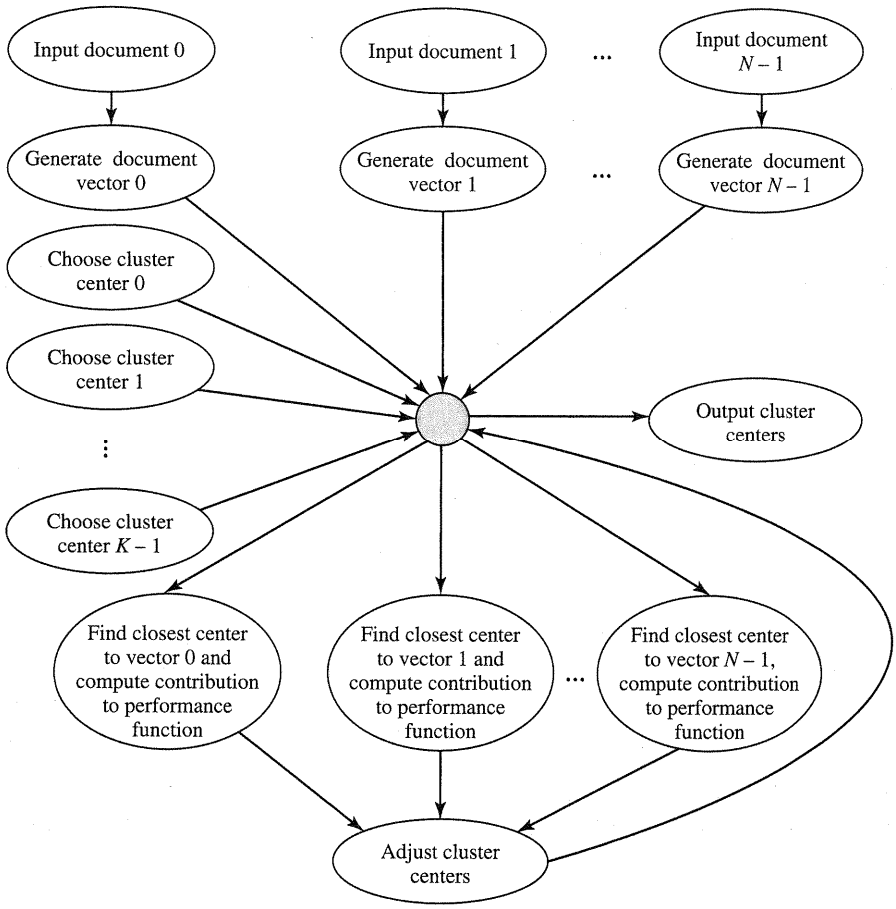


Figure 1.9 Dependence diagram for the document clustering algorithm. The small, unlabeled vertex in the middle is a "null task" containing no operations. Its purpose is to reduce the number of directed edges and make the diagram easier to read.

A good data dependence graph makes data and functional parallelism easy to find. That's certainly true in this case.

First, let's list the opportunities for data parallelism:

- Each document may be input in parallel.
- Each document vector may be generated in parallel.
- The original cluster centers may be generated in parallel.
- The closest cluster center to each document vector and that vector's contribution to the overall performance function may be computed in parallel.

Next, let's look for functional parallelism. The only independent sets of vertices are those representing the document input and vector generation tasks and those representing the center generation tasks. These two sets of tasks could be performed concurrently.

After we have identified parallelism in a problem, our next step is to develop an algorithm implemented in a programming language. Let's take a look at the variety of ways in which parallel computers have been programmed.

1.7 PROGRAMMING PARALLEL COMPUTERS

In 1988 McGraw and Axelrod identified four distinct paths for the development of applications software for parallel computers [85]:

1. Extend an existing compiler to translate sequential programs into parallel programs.
2. Extend an existing language with new operations that allow users to express parallelism.
3. Add a new parallel language layer on top of an existing sequential language.
4. Define a totally new parallel language and compiler system.

Let's examine the advantages and disadvantages of each of these alternatives.

1.7.1 Extend a Compiler

One approach to the problem of programming parallel computers is to develop parallelizing compilers that can detect and exploit the parallelism in existing programs written in a sequential language.

Much research has been done into the parallel execution of functional or logic programs, which can contain a good deal of intrinsic parallelism. However, most of the focus has been on the imperative programming language Fortran. Proponents of the development of parallelizing compilers for Fortran point out that existing Fortran programs represent the investment of billions of dollars and millenia of programmer effort. While not all of these programs would benefit from execution on a parallel computer, some organizations (such as the national laboratories run by the U.S. Department of Energy) would like to speed the execution of many sophisticated Fortran codes. The time and labor that could be saved from the automatic parallelization of these programs makes this approach highly desirable. In addition, parallel programming is more difficult than programming in Fortran, leading to higher program development costs. For these reasons some believe it makes more sense for programmers to continue to use simpler, sequential languages, leaving the parallelization up to a compiler.

The development of parallelizing compilers has been an active area of research for more than two decades, and many experimental systems have been developed. Companies such as Parallel Software Products have begun offering

compilers that translate Fortran 77 code into parallel programs targeted for either message-passing or shared-memory architectures.

This approach does have its detractors. For example, Hatcher and Quinn point out that the use of a sequential imperative language "pits the programmer against the compiler in a game of hide and seek. The algorithm may have a certain amount of inherent parallelism. The programmer hides the parallelism in a sea of DO loops and other control structures, and then the compiler must seek it out. Because the programmer may have to specify unneeded sequentializations when writing programs in a conventional imperative language, some parallelism may be irretrievably lost" [49].

One response to these concerns is to allow the programmer to annotate the sequential program with compiler directives. These directives provide information to the compiler that may help it correctly parallelize program segments.

1.7.2 Extend a Sequential Programming Language

A much more conservative approach to developing a parallel programming environment is to extend a sequential programming language with functions that allow the programmer to create and terminate parallel processes, synchronize them, and enable them to communicate with each other. There must also be a way to distinguish between public data (shared among the processes) and private data (for which each process has a copy).

Extending a sequential programming language is the easiest, quickest, least expensive, and (perhaps for these reasons) the most popular approach to parallel programming, because it simply requires the development of a subroutine library. The existing language and hence its compiler can be used as is. The relative ease with which libraries can be developed enables them to be constructed rapidly for new parallel computers. For example, libraries meeting the MPI standard exist for virtually every kind of parallel computer. Hence programs written with MPI function calls are highly portable.

Giving programmers access to low-level functions for manipulating parallel processors provides them with maximum flexibility with respect to program development. Programmers can implement a wide variety of parallel designs using the same programming environment.

However, because the compiler is not involved in the generation of parallel code, it cannot flag errors. The lack of compiler support means that the programmer has no assistance in the development of parallel codes. It is surprisingly easy to write parallel programs that are difficult to debug.

Consider these comments from parallel programming pioneers circa 1988:

"Suddenly, even very simple tasks, programmed by experienced programmers who were dedicated to the idea of making parallel programming a practical reality, seemed to lead inevitably to upsetting, unpredictable, and totally mystifying bugs" (Robert B. Babb II) [6].

"The behavior of even quite short parallel programs can be astonishingly complex. The fact that a program functions correctly once, or even one hundred times,

with some particular set of inputs, is no guarantee that it will not fail tomorrow with the same inputs" (James R. McGraw and Timothy S. Axelrod) [85].

1.7.3 Add a Parallel Programming Layer

You can think of a parallel program as having two layers. The lower layer contains the core of the computation, in which a process manipulates its portion of the data to produce its portion of the result. An existing sequential programming language would be suitable for expressing this portion of the activity. The upper layer controls the creation and synchronization of processes and the partitioning of the data among the processes. These actions could be programmed using a parallel language (perhaps a visual programming language). A compiler would be responsible for translating this two-layer parallel program into code suitable for execution on a parallel computer.

Two examples of this approach are the Computationally Oriented Display Environment (CODE) and the Heterogeneous Network Computing Environment (Hence). These systems allow the user to depict a parallel program as a directed graph, where nodes represent sequential procedures and arcs represent data dependences among procedures [12].

This approach requires the programmer to learn and use a new parallel programming system, which may be the reason it has not captured much attention in the parallel programmer community. While research prototypes are being distributed, the author knows of no commercial systems based on this philosophy.

1.7.4 Create a Parallel Language

The fourth approach is to give the programmer the ability to express parallel operations explicitly.

One way to support explicit parallel programming is to develop a parallel language from scratch. The programming language occam is a famous example of this approach. With a syntax strikingly different from traditional imperative languages, it supports parallel as well as sequential execution of processes and automatic process communication and synchronization.

Another way to support explicit parallelism is to add parallel constructs to an existing language. Fortran 90, High Performance Fortran, and C* are examples of this approach.

Fortran 90 is an ANSI and ISO standard programming language, the successor to Fortran 66 and Fortran 77. (Outside of the United States, Fortran 90 replaced Fortran 77. Within the U.S., Fortran 90 is viewed as an additional standard.) It contains many features not incorporated in Fortran 77, including array operations. Fortran 90 allows entire, multidimensional arrays to be manipulated in expressions. For example, suppose `A`, `B`, and `C` are arrays of real variables having 10 rows and 20 columns, and we want to add `A` and `B`, assigning the sum to `C`. In Fortran 77 we would need to write a doubly nested DO loop to accomplish

this. In Fortran 90 we can write the simple assignment statement

```
C = A + B
```

High Performance Fortran is an extension to Fortran 90 that supports the development of data-parallel programs in a number of ways. One important extension is the `FORALL` statement, which extends Fortran 90's array operations, allowing the programmer to specify more complex data-parallel operations. Another key feature of High Performance Fortran is the compiler directives that enable the programmer to specify how data should be mapped to processors.

C*, developed by Thinking Machines Corporation, extends the C programming language with the notion of a **shape**, which specifies the way in which parallel data are organized. A shape is an array-like object, but while array elements must be manipulated one at a time, elements of a shape may be manipulated simultaneously. For example, the declaration

```
shape [10][20]matrix;
```

sets up a template for parallel data. The subsequent declaration

```
float:matrix a, b, c;
```

specifies that variables `a`, `b`, and `c` are 10×20 array-like objects upon which parallel operations may be performed. To add `a` and `b` and assign the sum to `c`, you would write

```
with (matrix) { a = b + c; }
```

Earlier we noted the irony of a programmer encoding an inherently parallel algorithm in a sequential programming language and a compiler working to rediscover the parallelism. A language that supports explicit parallelism changes the relationship between the programmer and the compiler from adversaries to allies.

Adding parallel constructs to an existing programming language or creating a completely new parallel programming language requires the development of new compilers. Even after a language standard is adopted, it typically takes years for vendors to develop high-quality compilers for their parallel systems.

Some parallel languages, such as C*, were not adopted as a standard. In this situation many competing vendors may decide not to provide compilers for the language on their machines. When this happens, the portability of codes is severely compromised.

Another barrier to the adoption of new programming languages is user resistance. When new parallel constructs are added to a programming language, programmers must learn how to use these new constructs. Many programmers are reluctant to make the transition.

1.7.5 Current Status

While work continues to be done on parallelizing compilers and higher-level parallel programming languages, the most popular approach to parallel programming continues to be the approach of augmenting an existing sequential language with low-level constructs expressed by function calls or compiler directives. Parallel programming in C with MPI and/or OpenMP is an example of this approach. Low-level parallel programming can produce programs that exhibit high efficiency and are portable to a wide range of parallel systems. They have the disadvantage of being more difficult to code and debug than programs written in a higher-level parallel language.

1.8 SUMMARY

Driven by national security–related concerns, the United States government played an important role in early high-performance computing research and development. Later, capital-intensive industries, led by petroleum companies and automakers, began purchasing high-performance systems. The billion-fold increase in the speed of supercomputers between the 1940s and today is due to increasing clock speeds, increasing the amount of concurrency within processors, and using multiple processors to solve individual problems faster. Today's supercomputers are parallel computers containing thousands of CPUs.

Most parallel computers are not supercomputers. Parallel computers of more modest size still have enough speed to solve problems much more quickly than single-processor systems. Parallel computers have a wide range of uses, including powering Web search engines, helping Wall Street firms evaluate financial instruments, and predicting the weather.

If an organization can find a competitive advantage by either reducing the time needed to perform a computation or solving more sophisticated problems in the same amount of time, then parallel computing is worth exploring. Many problems are inherently parallel and are amenable to solution by parallel computers.

Commercial parallel computers typically range in price from a few hundred thousand dollars to several million dollars. Many research organizations and universities are constructing parallel computers using commodity, off-the-shelf components. While these systems do not have the balance between raw compute speed and interprocessor communication speed exhibited by commercial parallel computers, they can be assembled on modest budgets.

While the CPUs inside today's parallel computers are more than five hundred times as fast as the CPUs of 1985, there has been little improvement in parallel programming environments. Fortunately, two standards are available: MPI and OpenMP. The MPI library supports parallel programming through message passing. It allows processors that do not share a memory to cooperate in performing a parallel computation. OpenMP is a set of compiler directives. It helps a compiler to generate multithreaded code that can take advantage of multiple processors

available inside a shared memory multiprocessor. Together, MPI and OpenMP can be used to program a cluster of multiprocessors.

1.9 KEY TERMS

ASCI
centralized multiprocessor
COTS
data clustering
data dependence graph
data mining
data parallelism
functional parallelism
gigaflops
grand challenge problem
independent tasks
megaflops
multicomputer
multiprocessor
parallel computer
parallel computing
parallelize
parallel programming
pipelined computation
scientific data analysis
shape
supercomputer
symmetrical multiprocessor (SMP)
teraops

1.10 BIBLIOGRAPHIC NOTES

Supercomputing and the Transformation of Science, written by Kaufmann and Smarr, is an attractive, easy-to-read book describing how supercomputers have spurred new scientific discoveries [60]. It contains many extraordinary color images. Two articles in the April 1994 issue of *Communications of the ACM* give examples of using parallel computers to solve a variety of science and engineering applications. Camp et al. describe a variety of problems addressed at Sandia National Laboratory [13]. Tentner et al. discuss using parallel computers to solve a variety of problems related to nuclear reactor design [106]. Sabot has edited *High Performance Computing: Problem Solving with Parallel and Vector Architectures* [99]. The contributors give many examples of parallel programs efficiently solving nontrivial problems, such as weather prediction, chess, and mortgage-backed financing.

A more detailed history of parallel computing can be found in Appendix B of Wilson's monograph *Practical Parallel Computing* [116]. Patterson and Hennessy also have a section on the history of parallel computing in their textbook, *Computer Architecture: A Quantitative Approach* [90].

Researchers continue to explore compiler technologies supporting the automatic parallelization of sequential imperative programs. Bacon et al. have written a survey article describing a wide variety of compiler optimizations [7]. A fairly recent monograph devoted to the subject is *High Performance Compilers for Parallel Computing,* by Michael Wolfe [117].

Techniques for the automatic extraction of parallelism from logic programs are surveyed by de Kergommeaux and Codognet [20].

High Performance Fortran (HPF) attracted more interest than any other parallel programming language in the early 1990s. Unlike most other parallel programming languages, a reasonable variety of commercial compilers are available for HPF. To learn more about the language, see *The High Performance Fortran Handbook* by Koelbel et al. [62].

The parallel programming language occam is of particular interest because it served, more or less, as the "assembly language" of the INMOS Transputer, a single-chip computer designed for integration into parallel computer systems. Pountain and May have written *A Tutorial Introduction to Occam Programming* [93].

Skillicorn and Talia have surveyed models and languages for parallel computation [102]. They divide parallel programming models into six categories, according to the amount of abstraction they provide, and provide examples of programming languages at each of these levels. It is interesting to note that MPI and OpenMP would be categorized at the lowest level, where everything is explicit.

Jain et al. have written a survey of algorithms for data clustering [57].

In the original formulation of Moore's Law, Gordon Moore observed that the logic density of silicon-based integrated circuits closely followed the curve 2^{t-1962}, where t is the year [87]. In other words, density was doubling every year. Actual semiconductor bit densities grew in this fashion until the late 1970s, at which point the doubling period slowed to 18 months. By this time Moore's Law was so well established that people were happy to revise it without renaming it.

1.11 EXERCISES

1.1 Try these experiments with a few friends.

a. Shuffle a deck of cards, then determine how long it takes one person to sort the cards into the order A♠, 2♠, ..., K♠, A♡, 2♡, ..., K♡, A♣, 2♣, ..., K♣, ..., A◇, 2◇, ..., K◇.

b. How long does it take p people to sort p decks of shuffled cards?

c. How long does it take p people to sort one deck of cards? Try this experiment for $p = 1, 2, \ldots, 6$. How does your algorithm change as p changes?

1.2 Suppose you own a hole punch capable of putting a hole in an arbitrarily thick stack of paper. If you insert the paper into the hole punch and activate it, you will have a piece of paper with one hole in it. If you fold the paper in half before inserting it into the hole punch, you will have a piece of paper with two holes in it. If you can only use the hole punch once, how many times must you fold a piece of paper in order to put n holes in it? Prove that your answer is correct and optimal.

1.3 You have been assigned the task of computing the sum of 1,000 four-digit numbers as rapidly as possible. You hold in your hands a stack of 1,000 index cards, each containing a single number, and you are in charge of 1,000 expert accountants, each with a calculator. You may choose to use the services of any number of these accountants. The accountants are sitting at desks in a cavernous room. The desks are organized into 25 rows and 40 columns. Each accountant is able to pass cards to the four accountants nearest her—in front, in back, to her left, and to her right.

a. Describe a fast method of distributing cards to accountants.
b. Describe a fast method of accumulating the subtotals generated by the active accountants into a grand total.
c. Draw a graph that plots your best estimate of the time needed to compute the grand total as a function of the number of accountants you choose to use.
d. Add another curve to the graph you drew in part **c,** estimating the time needed to compute the grand total of 10,000 numbers, given 1, . . . , 1,000 accountants.
e. Explain why 1,000 accountants cannot perform the task 1,000 times faster than one accountant.

1.4 Refer to the problem of adding 1,000 numbers posed in Exercise 1.3. Find another way to arrange the desks of the accountants to reduce the time needed to distribute cards and collect subtotals. Describe the new desk arrangement, the new communication pattern, and the new estimate for time spent distributing cards and accumulating subtotals.

1.5 Given a task that can be divided into m subtasks, each requiring one unit of time, how much time is needed for an m-stage pipeline to process n tasks?

1.6 A copy machine's feeder tray holds pages to be copied. Assume that it takes 5 seconds to load a new group of pages into the feeder before copying and 10 seconds to unload the originals and the copies after copying. If the copier takes 4 seconds to print the first page of a group and 1 second to print every subsequent page, what is the minimum capacity of the feeder tray necessary to ensure that the effective throughput of the copier asymptotically approaches 40 pages per minute as the length of the original document increases?

1.7 Are all of today's supercomputers parallel computers? Is every parallel computer a supercomputer? Explain your answers.

1.8 If the performance of CPUs increases by a factor of 10 every five years, how long does it take for performance to double?

1.9 a. Using the performance of Caltech's Cosmic Cube and its Supercomputing '97 cluster as your guide, estimate the improvement in microprocessor speeds between 1984 and 1997.
b. Assuming the rate of change in microprocessor performance was constant between 1984 and 1997, what was the annual microprocessor performance improvement needed to account for the speed increase calculated in part **a**?

1.10 The complexity of an algorithm often becomes more, not less, significant as computer speeds increase. For example, an aircraft designer runs one simulation each evening between 5 P.M. and 8 A.M. When she gains access to a faster computer, she uses this extra speed to run a larger simulation (one that can provide more detailed results) in the same

amount of time. Suppose her current computer can solve a problem of size 100,000 in 15 hours. Assume that execution time is determined solely by CPU speed; i.e., all other resources such as I/O bandwidth and primary memory are not a constraint on performance. How large a problem can be solved in 15 hours by a computer that is 100 times faster if the simulation program's time complexity is

a. $\Theta(n)$
b. $\Theta(n \log_2 n)$
c. $\Theta(n^2)$
d. $\Theta(n^3)$

1.11 Name two advantages commodity clusters have over commercial parallel computers. Name one advantage a commercial parallel computer has over a commodity cluster.

1.12 For each of the following activities, illustrate the tasks needed to complete the activity with a data dependence graph. Each illustration should contain at least six tasks.

a. Building a house
b. Cooking and serving a spaghetti dinner
c. Cleaning an apartment
d. Detailing an automobile

1.13 Consider the data dependence graph in Figure 1.10. Identify all sources of data parallelism. Identify all sources of functional parallelism.

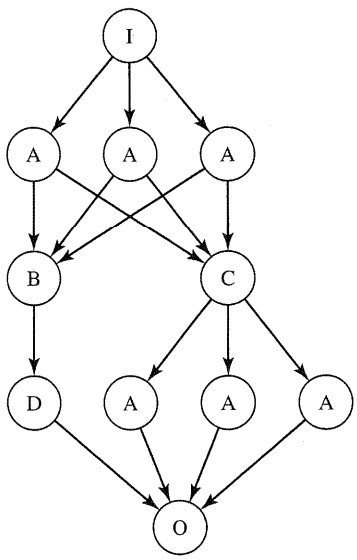


Figure 1.10 Dependence diagram for Exercise 1.13.

1.14 Suppose we are going to speed the execution of the data clustering algorithm by using p processors to generate the D-dimensional vectors for each of the N documents. One approach would be to preallocate about N/p documents to each processor. Another approach would be to put the documents on a list and let processors remove documents as fast as they could process them. Discuss one advantage of each approach.

1.15 Consider the vector-generation step of the data clustering algorithm described in this chapter. Assume the time required to perform this step is directly proportional to document size. Suggest an approach for allocating documents to processors that may avoid the problems associated with either preallocating N/p documents to each processor or having processors retrieve unprocessed documents from a central list.