
Chapter One

Introduction

If you walk the footsteps of a stranger, you will learn
things you never knew you never knew—*Pocahantas*,
by *Disney*

This chapter introduces many of the basic notions of parallel computation. In Section 1.1 we give a short overview of the book, and Section 1.2 attempts to define what we mean by parallel computing. Section 1.3 introduces the critical topic of performance, which is central to the entire subject. In Section 1.4 we describe some of the motivating factors for the development of parallel computers. This is followed by some examples of parallelizing computational problems. The first two examples (Section 1.5) are quite simple, but serve to introduce many of the most important concepts. The next examples (Section 1.6) help to introduce further concepts as well as to provide some numerical applications that will be developed more in the text. Section 1.7.1 puts into context the role of parallel computation in solving technical problems, and Section 1.7.2 (also see Section 2.7) considers parallelism in a broader context.

1.1 OVERVIEW

Parallel computing enables simulation in a variety of application areas which would not be possible with sequential processing alone. To use it effectively, there are diverse subjects that must be understood. This book focuses on three main areas that contribute to overall understanding of parallel computing: algorithms, architecture, and languages. All of these are essential contributors to solving problems of interest, which we refer to as “applications” in Figure 1.1.

A basic understanding of computer architecture is needed to understand and predict the behavior of programs on different machines. A variety of fundamentally different computer architectures are commercially available today. Some differ substantially in the way they are programmed and the performance that can result. An introduction to computer architecture is presented in Chapter 3 to allow us to compare and contrast existing options. Some designs can be seen to be less appropriate for certain applications based on the simple analysis presented there.

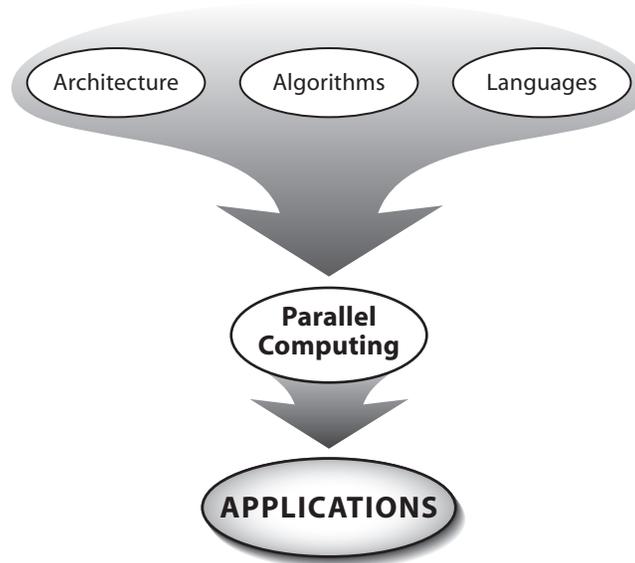


Figure 1.1 Knowledge of algorithms, architecture, and languages contributes to effective use of parallel computers in practical applications.

Central to the subject of parallel computing are algorithms. A general rule for scientific applications is that there is *no* parallelism of any significance occurring naturally. Parallelism must be created by removing *dependences* (see Chapter 4) that exist in most algorithms. Entirely new algorithms must be sought in some cases. Fortunately, the nature of scientific applications allows one to utilize a multitude of algorithms to solve the same problem, and highly efficient parallel algorithms can be found in most cases.

Several computer languages are in use today for programming parallel computers. Some languages can be used for different computer architectures with only a change at the compilation stage: the source application code does not change. Other languages are more closely aligned with a particular computer architecture. In Chapter 5, the essential features of various parallel computer languages are presented, and Chapters 8–10 introduce particular languages.

Applications are the driving force for all of computing, and much of the stimulus for parallel computing has come from scientific applications. We will introduce some simple prototypical application kernels later in this chapter as a basis for the discussion of programming languages and basic concepts of parallel computing. Once the elementary concepts are firmly established, basic algorithms of scientific computation are presented in Chapter 12. Later, more complete scientific applications will be developed and analyzed extensively, such as particle dynamics in Chapter 13. These can be used as models of projects that can be done by students as part of a

course. Later chapters on mesh-based computations (Chapter 14) and on sorting (Chapter 15) are similar in approach.

1.2 WHAT IS PARALLEL COMPUTING?

We define parallel computing to be the application of two or more processing units to solve a single problem. These units can be physical processors or logical processes. A *process* is an abstraction provided by the operating system to capture the state of a program in execution. Thus, a program containing parallel units of work, or tasks, is executed by two or more co-operating processes executing on one or more processors.

In the class of parallel programs of primary interest to this book, each process executes the same program, but with different data. In fact, typically the program is viewed as processes more or less moving through the same code, but with different values according to the chunk of work assigned to the process. Although there are exceptions, this is the guiding paradigm.

Since processes calculate values that are needed by other processes, we need a way to distinguish among those data at different processes. It is useful to use a process-centric view where one process is considered with respect to all other processes. It naturally follows that the data of the one process under consideration are considered *on-process data* and all the rest are *off-process data*. We define a *processing unit* as a process executing on some processor. In using multiple processing units to solve a problem, varying degrees of coordination are required. Coordination primarily revolves around accessing off-process data required by some process to compute its tasks. Already introduced, these are referred to as dependences.

Consider a real world example of a typing pool with the job to type the chapters of this book, one typist per chapter. After completion of a chapter, typists may want to exchange chapter page counts so to sequentially number the pages of the book. If we think of each typist as a processing element, then this exchange involves somehow getting access to off-process data. Specifically, the page count of a chapter is accessed after completion of the chapter and not before. Current methods in parallel computing achieve this data exchange typically in one of two ways. In one method, the process requesting information will access the off-process information directly from memory where it was written. This requires some form of *synchronization* so that the value in memory is accessed when it is valid. In the typist example, the page count of preceding chapters is retrieved when it is complete and not before. Another method to acquire off-process information uses *messages*. In a message, the required information is packaged, somehow identified, and sent from the process that defined it to the process that requires it. Thus, in this way messages have both the *information* (page counts in our example) and *synchronization*, since the page-count message is presumably sent after the chapter has been finished. That off-process data are required and that

these data may be on another physical processing unit introduces another dimension to the traditional memory hierarchy illustrated in Figure 1.2, that of off-process data. The term *communication* is often used in reference to accessing off-process data.

1.3 PERFORMANCE

The objective of the book is to help the reader achieve the best possible *performance* from parallel computers. Performance can be measured in many ways, but we will be interested primarily in the speed at which computation is done in floating point arithmetic.

Definition 1.3.1. The (floating point) **performance** of a computer code is the number of floating point operations that it can execute in a given time unit. Performance is often stated in terms of Millions of Floating Point Operations Per Second (megaFLOPS or MFLOPS), or Billions of Floating Point Operations Per Second (gigaFLOPS or GFLOPS).

Sometimes it will be necessary to refer to an amount of work in units of Floating Point Operations (a FLOP). The plural of this will be written FLOP's to distinguish it from the performance figure FLOPS (which is a number of FLOP's per second). In particular, 10 MFLOP's means ten million floating point operations, whereas 10 MFLOPS means a performance of ten million floating point operations per second. Since this combination of units is the one most critical to describing performance, we feel it is appropriate to introduce a new unit, the Cray.¹ We define a Cray to be 10^9 floating point operations per second (one floating point operation per nanosecond), since this was the level of performance being achieved by a single processor at Seymour Cray's untimely death.

To illustrate this definition, let us begin with a very simple example. A look at a book of mathematical tables tells us that

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} - \frac{1}{15} + \dots \quad (1.3.1)$$

This is not a rapidly converging series to use to compute π , but it serves as a good example for studying the basic operation of computing the sum of a series of numbers:

$$A = \sum_{i=1}^N a_i. \quad (1.3.2)$$

¹The development of the first "supercomputers" was largely the result of efforts lead by Seymour Cray (1925-1996). Seymour Cray earned a BS in engineering from the University of Minnesota in 1950. He co-founded Control Data Corporation (CDC) in 1957; the CDC 6600 is the primary candidate for the title of "first supercomputer." The series of supercomputers bearing Cray's name were produced by Cray Research. Started in 1972, this company was headquartered in Seymour's boyhood home, Chippewa Falls, Wisconsin, also home of the Jacob Leinenkugel Brewing Co.

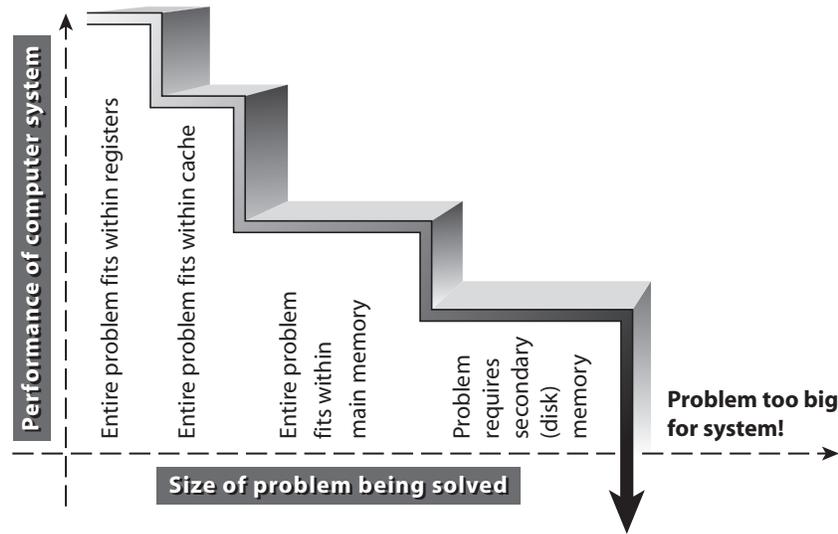


Figure 1.2 Hypothetical model of performance of a computer having a hierarchy of memory systems (registers, cache, main memory, and disk).

The computation of A requires $N - 1$ floating point additions and $N + 1$ memory references. If it takes T_N seconds to compute this for a given implementation (meaning for a given code compiled for a given computer), then the performance is $(N - 1)/T_N$ FLOPS. Since this is likely to be a very large number, we usually say that it is either $(N - 1)/(T_N \times 10^6)$ MegaFLOPS or $(N - 1)/(T_N \times 10^9)$ GigaFLOPS (or Crays).

1.3.1 Performance analysis

There is often a theoretical maximum floating point performance for a given computer, and the performance of any code run on it is guaranteed not to exceed this figure. The goal of **performance analysis** is to understand not only the performance being achieved by a given code, but the reasons why this may differ from the theoretical maximum. In many cases, this is quite difficult to do with sufficient precision since computer clocks have a finite resolution. Any given operating system may have different timers available. It is crucial to understand the advertised accuracy of the timer you are using, and to compare that with your own estimate of how small a time it can measure accurately. In timing the computation of the sum in (1.3.2), this may mean that the observed time $T_N = 0$ for $N < N_c$ for a critical size N_c depends on the resolution of the timer.

Even if the timer never returns zero, it may be that it reports $T_N = 1$ time unit for all $N < N_c$, which is equally uninformative. One can estimate the minimum measurable time in various ways, but one useful way is to plot the ratio of the observed time to a predicted time based on some model

of the computation. For example, we assume that the summation (1.3.2) should take an amount of time proportional to N . So the ratio T_N/N should be more or less constant. But when $T_N = 0$ it will drop to zero; before that happens, for slightly larger N , it may become erratic.

To get reliable information on performance for very short computations, it may be necessary to repeat computations during the timing cycle until the total time is large with respect to the smallest measurable time. Dividing by the number of repeats (see Exercise 1.7) can give an estimate of small times that would otherwise be too short to be measured.

Another feature of self-measurement is a kind of **computer uncertainty principle** similar in spirit to the Heisenberg Uncertainty Principle of quantum mechanics.² One cannot achieve arbitrary precision in measuring the performance of most computers since we usually use the computer itself to do the measurement. Introduction of timing analysis code can alter the time to completion by interfering with the calculation under measurement. In a parallel program it is important to assign time costs to sections of a program. In addition to determining the time for various procedures comprising a problem solution, it is frequently useful to separate out the costs to access remote data. The point is that timing a code can have subtle issues requiring careful engineering of a suitable timing strategy.

1.3.2 Memory effects

A critical feature of any algorithm is the relationship between the amount of work done and the amount of memory that must be accessed. In many cases, it is possible to quantify the relationship by a ratio measuring, at least in an asymptotic sense for sufficiently large problems, the number of floating point operations done per unit of memory accessed (via either a read or a write).

Definition 1.3.2. The **work/memory ratio** of an algorithm is the ratio ρ_{WM} of the number of floating point operations to the number of memory locations referenced (either reads or writes).

There is delicate wording in Definition 1.3.2. The denominator counts the *number of memory locations* referenced, L , not the *number of memory references* made, R . For example, should a program consist solely of the senseless but valid loop `for(i=0; i<1000; i++) j = i;` then $R \gg L$

²Werner Heisenberg (1901–1976) invented matrix mechanics in 1925, the first version of quantum mechanics, and in 1932 was awarded the Nobel Prize in physics for this work. Heisenberg is best known for the Heisenberg Uncertainty Principle which he discovered in 1927. The uncertainty principle states that if you become more certain about a particle's position, then you must become less certain about its momentum, and vice versa. The uncertainty in position, δx , and in momentum, δp , are related by $2\delta x\delta p = \hbar$, a very small number equal to 1.05×10^{-34} Joule seconds, not noticeable in everyday life.



Figure 1.3 A simple memory model with a computational unit with only a small amount of local memory (not shown) separated from the main memory by a pathway with limited bandwidth μ .

since two memory locations are referenced 1000 times. The number of memory references made by a program depends on the particular implementation used to do the computation, whereas the number of memory locations used measures the total number of data (or memory locations) involved. We will see that simplified but useful definition can require quite complex analysis with the introduction of compiler optimization and additional components to the memory hierarchy. This distinction is further illustrated in Example 1.3.5 and Exercise 1.3.

Example 1.3.3. The computation of A in equation (1.3.2) requires $N - 1$ floating point additions and involves $N + 1$ memory locations: one for A and n for the a_i 's. Therefore, the work/memory ratio for this algorithm is $\rho_{\text{WM}} = (N - 1)/(N + 1) \approx 1$ for large N . In most cases, we will only be interested in such quantities for large data sizes, so we will loosely say that $\rho_{\text{WM}} = 1$ for this algorithm. See Section 1.3.3 for a more precise way of simplifying such approximations.

It is uncommon to have ρ_{WM} much less than one, but it is not unusual to have it become arbitrarily large as a function of data size. In fact, we will see that certain computer architectures perform significantly better with algorithms having a large ρ_{WM} . One goal of this book is to help in either choosing or designing algorithms with large ρ_{WM} .

The observed performance of a computer system depends on its ability to access the memory required by a given computation. If the maximum speed that the memory system can deliver information is μ words per time unit, then the maximum performance that can be achieved is $\mu\rho_{\text{WM}}$. We formalize this observation in the following theorem. We will assume that the computer system has a very simple structure as indicated in Figure 1.3: the computational unit has only a small amount of memory available locally, with the main memory accessible only via a channel of limited bandwidth.

Theorem 1.3.4. Suppose that a given algorithm has a work/memory ratio ρ_{WM} , and it is implemented on a system as depicted in Figure 1.3 with a maximum bandwidth to memory of μ million floating point words per second. Then the maximum performance that can be achieved is $\mu\rho_{\text{WM}}$ MFLOPS.

Theorem 1.3.4 provides an upper bound on the number of operations

per unit time, by assuming the floating point operation blocks until data are available to the cpu. Therefore the cpu cannot proceed faster than the rate data are supplied, and it might proceed slower. Note again that ρ_{WM} measures the *amount* of memory referenced, not the *number* of memory references (the same memory location could be referenced multiple times in a given algorithm).

Example 1.3.5. Consider the computation of the product of a square matrix $\mathbf{A} = (a_{ij})$ and a vector $\mathbf{V} = (v_i)$, defined by

$$(\mathbf{AV})_i := \sum_{j=1}^n a_{ij}v_j \quad \text{for } i = 1, \dots, n. \quad (1.3.3)$$

Then the number n of floating point operations is n multiplies and $n - 1$ adds for each $(\mathbf{AV})_i$ for a total of $(2n - 1)n$ FLOP's to compute \mathbf{AV} . The number of memory locations involved is n^2 for \mathbf{A} , and n each for \mathbf{V} and \mathbf{AV} . Thus the ratio ρ_{WM} of the number of floating point operations to the number of data values (i.e., number of **memory locations**) involved in the algorithm is

$$\rho_{\text{WM}} = \frac{2n^2 - n}{n^2 + 2n} \approx 2$$

for n large. Note that the number of memory **references** can be larger. If we have to read v_j from memory every time we compute $a_{ij}v_j$ (for $i = 1, \dots, n$) then we read \mathbf{V} a total of n times. In this worst case, the ratio of work to memory *references* is one instead of two, worse by a factor of two.

Theorem 1.3.4 refers to a machine with a single path to memory, but this model is too simplistic for real systems. The typical memory subsystem is a multi-component system with a *hierarchy* of components organized from fast and costly to slower and less costly. In general, the memory components closest to the cpu are fastest, with the slower memory components feeding the faster ones using various strategies to amortize the cost to pull data from a slower component into a faster one. Memory cache (Section 3.1.2) supplies resident data to the cpu in a few machine cycles, and nonresident data are obtained from slower main memory in blocks consisting of contiguous memory locations. Consequently, programs that access data in some memory locale and reuse data will get more usage of data in the cache, resulting in better performance. In terms of the previous discussion, data reuse makes the most of data fetched to the cache by accessing the same memory location multiple times. For a cpu to access a datum already in cache is called a *cache hit*. Figure 1.4 presents a simple picture of such a system. Note that to access the same memory location multiple times in a program does not guarantee a cache hit. The size of the cache is much smaller than the size of main memory, so that it follows that many locations in main memory share



Figure 1.4 A memory model with a large local data cache separated from the main memory by a pathway with limited bandwidth μ .

only a few locations in the cache; we leave this important detail aside for the present discussion.

Consider the matrix-vector multiplication algorithm (1.3.3). Let us assume that after the vector \mathbf{V} has been read from memory once, it stays in cache. Then the number of memory *references* is the same as the amount of memory referenced, and the maximum performance in Theorem 1.3.4 can be achieved. That is, ρ_{WM} correctly measures the amount of memory traffic under these assumptions. Of course, this can only happen if the cache size is sufficiently large to hold all of the vector \mathbf{V} .

In addition to particular analyses for particular algorithms like (1.3.3), a general model of behavior for cache systems can be developed as in the next example based on average cache hit rates. We will use notation familiar from physics such as $\frac{\text{words}}{\text{second}}$ to mean “words per second” (denoting a rate of transmission). Of course, this notation is convenient for seeing how to cancel units of time, operations, or memory.

Example 1.3.6. The performance of a two-level memory model (as depicted in Figure 1.4) consisting of a cache and a main memory can be modeled simplistically as

$$\frac{\text{average cycles}}{\text{word access}} = \% \text{hits} \times \frac{\text{cache cycles}}{\text{word access}} + (1 - \% \text{hits}) \times \frac{\text{main memory cycles}}{\text{word access}}, \quad (1.3.4)$$

where $\% \text{hits}$ is the fraction of cache hits among all memory references. For a main memory access time of 100 cycles per word and a cache access time of 2 cycles per word, the average $\frac{\text{cycles}}{\text{word}}$ for 10% and 90% hit rates are 90.2 and 11.8, respectively. The average number of words per time unit is

$$\frac{\text{words}}{\text{second}} = \left(\frac{\text{average cycles}}{\text{word access}} \right)^{-1} \times \frac{\text{cycles}}{\text{second}}.$$

On a 2 GHz processor, the corresponding average memory rates would be 22.2 and 169.5 million words per second. The performance of programs exhibiting an average memory access time of n cycles per word on a 2 GHz processor is bounded by

$$\left(n \frac{\text{cycles}}{\text{word access}} \right)^{-1} \times \left(2 \times 10^9 \frac{\text{cycles}}{\text{second}} \right) \times \rho_{\text{WM}} = \frac{2 \times 10^9 \rho_{\text{WM}}}{n} \text{FLOPS}.$$

For an average of $\rho_{\text{WM}} = 2$ floating point operations per word, with 10% and 90% cache-hit cases, the bounds on performance are 44.3 MFLOPS and 330 MFLOPS, respectively.

Modeling the speed of access to memory is quite complex on modern high-performance computers. Figure 1.2 indicates the performance of a hypothetical application, depicting a decrease in performance as a problem increases in size and migrates into ever slower memory systems [6] [5]. Eventually the problem size reaches a point where it can not ever be completed for lack of memory.

Such degradation would, e.g., result for an algorithm that has a fixed and sufficiently small work/memory ratio ρ_{WM} , so that peak performance cannot be achieved for data not resident in cache. In this case, Figure 1.2 represents a plot of the bandwidth μ of the various memory systems, in view of Theorem 1.3.4. It is typical for a computer's various memory systems to be progressively slower, larger, and cheaper per word of memory. See Exercise 1.7 for a specific example of this behavior for the summation problem (1.3.2).

For computers having a performance profile as indicated in Figure 1.2, algorithms with a larger work/memory ratio ρ_{WM} will perform better than ones with smaller ones. It is sometimes possible to choose algorithms that produce equivalent answers but have vastly different work/memory ratios.

1.3.3 Asymptotic analysis

In much of our analysis of algorithms and performance we will be interested in the order of the growth of a function excluding the details of multiplicative constants and lower order terms. When we look at the growth of $f(n)$ in this way, we are considering the *asymptotic* growth in the limit of large n .

For our purposes we will use the \mathcal{O} -notation as defined below to describe an asymptotic upper bound of a function.

Definition 1.3.7. For a function $f(n)$ the **asymptotic upper bound** $\mathcal{O}(g(n))$ implies that there exists a constant c_1 , satisfying $0 < c_1 < \infty$, and an integer $n_0 \geq 0$ such that $f(n) \leq c_1 g(n)$ for all $n \geq n_0$.

Example 1.3.8. To show that the upper bound of function $f(n) = an^2 + bn$ is $\mathcal{O}(n^2)$ we seek c_1 so that the inequality

$$an^2 + bn \leq c_1 n^2$$

holds for large n . Provided that $c_1 > a$, this does hold for any $n \geq b/(c_1 - a)$.

It is important to understand that the asymptotic upper bound, say, $\mathcal{O}(n^2)$ for some performance metric, does not say the performance is $\mathcal{O}(n^2)$. Rather it does say that the worst case running time for any n is $\mathcal{O}(n^2)$. For

example, *bubblesort* (Figure 15.1) can sort an already sorted list with an operation count cn , although bubblesort has a worst-case asymptotic upper bound of $\mathcal{O}(n^2)$.

Example 1.3.9. In Example 1.3.3 we found the work/memory ratio for computing A in (1.3.2) to be $\rho_{\text{WM}} = (N - 1)/(N + 1)$, and we concluded $\rho_{\text{WM}} \approx 1$. Now let us compare $(N - 1)/(N + 1)$ to one:

$$\begin{aligned} \frac{N - 1}{N + 1} - 1 &= \frac{N - 1 - (N + 1)}{N + 1} \\ &= \frac{-2}{N + 1} = \mathcal{O}(N^{-1}). \end{aligned}$$

Thus we can say more precisely that $\rho_{\text{WM}} = 1 + \mathcal{O}(\frac{1}{N})$.

Example 1.3.10. In some instances asymptotic analysis can be misleading. A common source of confusion arises using the notation without taking into account the constant factors. Suppose we have two algorithms to solve the same problem. Algorithm F has running time $f(n) = an^2$, and algorithm G has running time $g(n) = bn$. By solving the inequality $an^2 < bn$ we find that $g(n) > f(n)$ for $n < \frac{b}{a}$. Therefore, for $n < \frac{b}{a}$, algorithm F is the better choice even though $f(n) = \mathcal{O}(n^2)$ and $g(n) = \mathcal{O}(n)$.

1.4 WHY PARALLEL?

Several factors spurred the development of parallel supercomputers. Many of these factors can be categorized as follows:

- physical limitations, e.g., the speed of light;
- economic factors (economies of scale);
- scalability (matching the computer to the problem size);
- architectural improvements (reflecting the increasing role of memory compared to processing power).

We will discuss examples of all of these, although our list is meant to be simply illustrative, not exhaustive.

1.4.1 Physical limits

At the most elementary level, physical limitations such as the speed of light³ (see Figure 1.5) make it difficult to construct large computers that operate

³The speed of light imposes a physical limit on signal propagation times across computer circuitry. This physical constant is approximately 3×10^{10} cm/sec in a vacuum, an upper limit on the speed of electromagnetic propagation in various physical media, such as copper wire. Grace Murray Hopper (1906–1992) was an early pioneer in computing who was famous for handing out a nanosecond’s worth of copper wire to students and admonishing them not to waste nanoseconds in their codes.

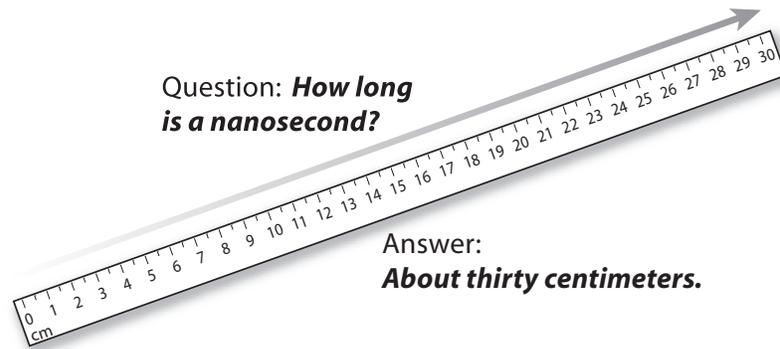


Figure 1.5 Light travels about one foot (in a vacuum) in a nanosecond.

with a clock speed on the order of a nanosecond or less. In this time, light travels only a foot or so in a vacuum (electrical signals are even slower in copper wire). This forces tolerances to be very precise in order to guarantee synchronization of even the simplest operations such as reading a word of data from memory, while demanding increasingly compact packaging of components.

Historically, processor speeds have increased exponentially, with a doubling time of something less than two years. In the last two decades, this has been exemplified by the rapid increase in performance of microprocessors. For example, Figure 1.6 depicts this increase for the Intel “86” family of computer chips which powered the personal computer revolution. Here, a fifteen-year period has led to a performance increase at the processor level by a factor of more than one hundred.

Due in part to speed-of-light limitations on the rate of data movement and lagging performance of memory subsystems, the increase in performance of vector supercomputers has been much less dramatic in the last three decades (see also Figure 1.6) than the increases in processor cycle speeds. Similarly, the performance increase of a uniprocessor workstation does not directly track processor cycle speed increases due in part to more slowly increasing memory performance with an increasing differential between processor speed and memory access time. The problem, however, has been more complex in vector processing systems due to the characteristically large memories and faster processing units.

1.4.2 Economic factors and scaling

It has always cost more to produce high-end computers than “commodity” single-processor workstations. Figure 1.7 shows this behavior with the family of workstations provided by Digital based on the SPECint92 perfor-

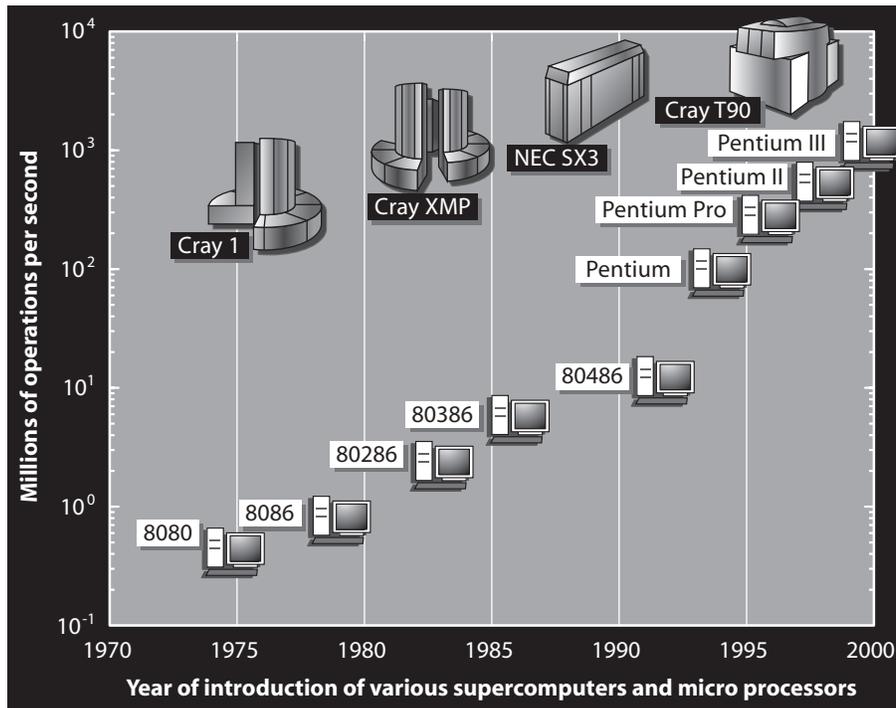


Figure 1.6 Performance of various “supercomputers” and the Intel “86” family of computer chips.

mance⁴ for the DEC “alpha” workstations. It is clearly more cost effective to buy three of the lowest performance workstations than one of the higher-performance ones, in terms of the aggregate performance available. It is natural to consider combining several less powerful processors to form a single, more cost-effective computer. Of course, in doing so there may be additional costs related to providing space, electricity, sufficiently fast communications, and other factors that complicate the price/performance equation. However, the economies of scale due to using many commodity computers will frequently compensate for this.

A significant topic of this book will be how to partition a large calculation into component parts which can then be performed by a collection of individual computers with minimal interaction. If this can be done, one extra benefit is that the individual calculations done by each of the computers are smaller, e.g., in terms of the size of the data set or the number of operations performed. Figure 1.2 indicates how performance can decrease as the problem size increases due to migration of data into ever slower memory systems. In this case, a problem which can be decomposed into smaller parts can potentially run with greater than linear speedup (see Definition 2.2.1)

⁴The System Performance Evaluation Cooperative (SPEC) is a nonprofit organization founded by computer vendors in 1988 with the goal to provide realistic, standardized performance tests.

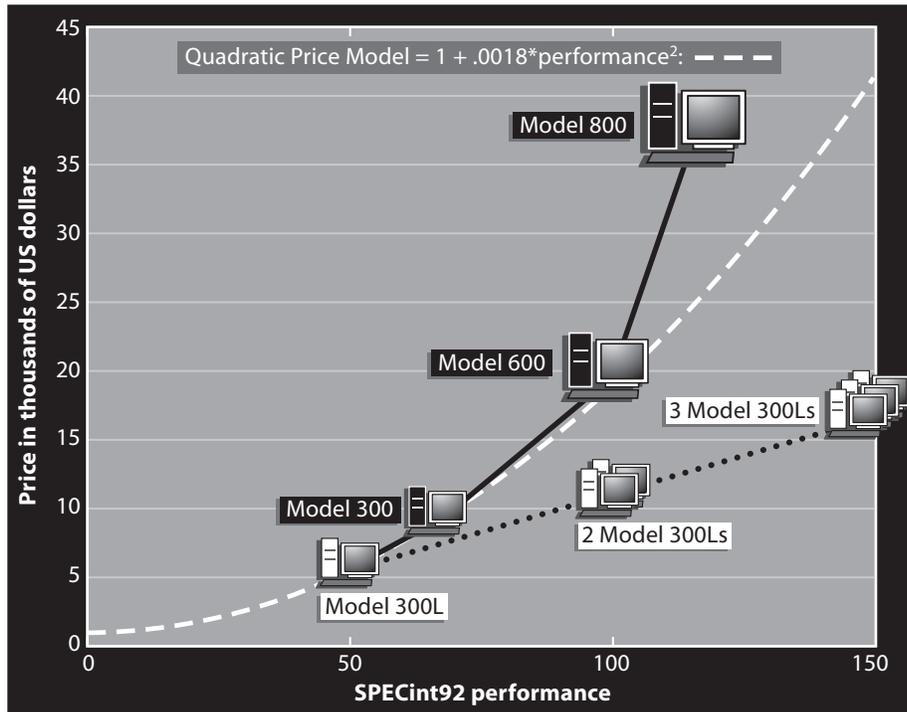


Figure 1.7 The cost of a particular computer architecture grows quadratically as a function of the performance of the computer. Plotted is the price of four Digital “alpha” workstation models (the circles on the graph) as a function of their SPECint92 performance. A quadratic curve has been fitted to the data to clarify its nonlinear behavior. Data were drawn from the 19 October 1993 *Wall Street Journal*. The dotted line indicates the price/performance associated with buying multiple machines of the cheapest model.

since the problem size for each individual computer is decreasing as the number of processors is increased. As indicated following Definition 1.3.2, the ratio of the amount of work done to the amount of memory that must be accessed is a critical factor determining performance. In any case, the ability to match the size of a parallel computer to a given problem (many processors for big problems, few for small ones) provides a way to “right size” the computer resources to ensure better resource utilization.

1.4.3 Memory

A recent but perhaps overarching reason for the success of parallel computers is that they allow the aggregate bandwidth between the processor(s) and memory to be made much larger at minimal expense. With a conventional pathway to memory, such as a bus (see Section 3.1.1), there are significant limits to how fast information can be moved between the processor and main memory. As one might expect, the cost of such a pathway also increases greater than linearly with the bandwidth achieved. A parallel computer

allows one to have many pathways (as many as the number of processors or more) and thereby to keep a balance between processing power and memory bandwidth at reasonable cost.

1.4.4 Parallel conclusions

In summary, parallel computers provide the following advantages which we have emphasized:

- they postpone the limitations of the speed of light that hampered the design and manufacture of conventional supercomputers;
- they allow cheaper components to be used to achieve comparable levels of aggregate performance;
- they allow problem sizes to be subdivided and thereby achieve a better match between algorithm and appropriate system components, such as cache and RAM, leading to better system performance;
- they allow aggregate bandwidth to memory to be increased together with the processing power at reasonable cost.

1.5 TWO SIMPLE EXAMPLES

For the sake of orientation, we give two extended but simple examples which exhibit key features of parallel computing without requiring much mathematical background. This will give us some basic examples to work with as we develop ideas in the first few chapters. They illustrate the concepts introduced so far in Definition 1.3.1 and Definition 1.3.2, and they introduce some additional key concepts, presented also in a series of definitions.

1.5.1 Simple sums

We begin with the summation problem (1.3.2):

$$A = \sum_{i=1}^N a_i.$$

This sort of operation is often called a *reduction*; it *reduces* the vector (a_1, \dots, a_N) to the scalar A . The formal definition of a reduction will be given in Chapter 6.

Assume for simplicity that N is an integer multiple, k , of P : $N = k \cdot P$. Then we can divide the reduction operation into P partial sums:

$$A_j = \sum_{i=(j-1)k+1}^{jk} a_i \quad (1.5.1)$$

for $j = 1, \dots, P$. Then

$$A = \sum_{i=1}^P A_i. \quad (1.5.2)$$

Leaving aside the last step (1.5.2), we have managed to create P parallel tasks (1.5.1) each having $k = N/P$ additions to do on $k = N/P$ data points.

Definition 1.5.1. A **task** is a part of a computation that can be thought of independently from other parts. We think of a task as something that can be computed by a separate **procedure**, such as a **subroutine** in Fortran, a **function** in C, or a **method** in Java.

Recall that the work/memory ratio (Definition 1.3.2) for these computations is 1 (for k sufficiently large); that is, there is only one floating point operation to be done for each data point a_i . The ratio $k = N/P$ is often called the *granularity* of the parallel tasks.

Definition 1.5.2. The **granularity** of a set of parallel tasks is the amount of work (of the smallest task) that can be done independently of any other computation.

The basic job of parallelizing scientific computation is to discover, create, or otherwise expose independent calculations that can be done in parallel with minimal communication. When no communication is required, we give these a special name.

Definition 1.5.3. Tasks that can be done independently of any other computation, without any communication required among them, are called **trivially parallel** or **embarrassingly parallel**.⁵

The tasks in (1.5.1) are trivially parallel. However, the final step of summing the A_i 's in (1.5.2) requires some form of cooperation, either communication of data or synchronization, among the P processors that computed them. We postpone detailed discussion of algorithms for forming this sum until Chapter 6, but simple algorithms will be given as examples in the sequel.

There are very few exceptions to the rule that any scientific program consists of loops, typically many. The computation of the sum (1.3.2) will

⁵Parallel computing for scientific and engineering applications remains a difficult undertaking. The act of parallelizing an application can involve Herculean efforts, or a sizable piece of a graduate student's career. Yet, there are some applications without data dependences (Chapter 4) along a richly parallel dimension, whereby the applicationists may avoid dramatic efforts in parallelization. The cliché *embarrassingly parallel* pokes fun at the good fortune of having readily available parallelism, permitting routine methods. Jay Boris, computational fluid dynamicist speaking at the 1997 DOD High-Performance Computing Modernization meeting, however, summed up the practical side of the matter: "I am not *embarrassed* about the [trivial] parallelism in my application, I am happy about it!"

be programmed typically in a *loop*, e.g., a `DO` loop in Fortran or a `for` loop in C. The concept of *iteration space* is useful to understand how to deal with parallelism in loops.

Definition 1.5.4. The **iteration space** of a given set of (possibly nested) loops is a subset of the Cartesian product of the integers consisting of the set of all possible values of loop indices. The dimension of the Cartesian product is the number of nested loops (it is one if there is only one loop). When the exact set of loop indices is not known without running the code, the iteration space is taken to be the smallest set known to contain all of the loop indices.

The iteration space for a single loop implementing (1.3.2) is simply the integers from 1 to N . The parallel algorithm embodied in (1.5.1) and (1.5.2) corresponds to dividing this set into P contiguous segments, which we call a *decomposition* of the iteration space:

Definition 1.5.5. An **iteration space decomposition** consists of a collection of disjoint subsets of the iteration space whose union is all of the iteration space.

The algorithm (1.3.2) has been parallelized using an approach referred to as *data parallelism*.

Definition 1.5.6. The concept of **data parallelism** refers to parallelizing a composite operation on a large data set in which the same (or similar) individual operations are carried out on each data item.

The data-parallel operation in (1.3.2) is summation. The key point is the homogeneity of the overall task, which allows it to be divided in arbitrary ways through multiple instances in execution of the same procedure where each instance operates on a portion of the iteration space. This kind of parallelism is very similar to the type of *loop* parallelism we will study in Section 4.2.

1.5.2 Load balancing

Figure 1.8 depicts graphically the iteration space for the summation problem (1.3.2) parallelized using the decomposition in (1.5.1). One requirement for a decomposition (which we have satisfied in the one depicted in Figure 1.8) is that the work to be done by each processor be *balanced* among all the processors. If the work is not distributed equally, then one processor may end up taking longer than the others. Since we are doing a cooperative project, the entire job cannot be finished until the slowest subtask is finished. We formalize the notion of balancing the work, or *load*, in the following definition.

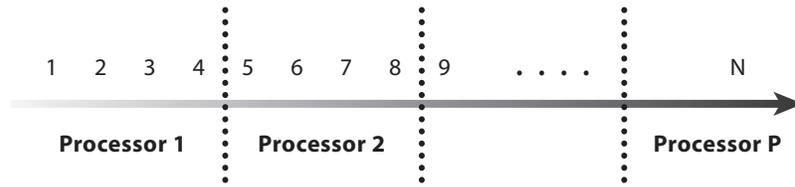


Figure 1.8 The iteration space for the summation problem with a simple decomposition indicated by dotted lines for a granularity of $k = 4$.

Definition 1.5.7. Suppose that a set of parallel tasks (indexed by $i = 1, \dots, P$) execute in an amount of time t_i . Define the average execution time

$$\text{ave} \{t_i : 1 \leq i \leq P\} := \frac{1}{P} \sum_{1 \leq i \leq P} t_i. \quad (1.5.3)$$

The **load balance** β of this set of parallel tasks is

$$\beta := \frac{\text{ave} \{t_i : 1 \leq i \leq P\}}{\max \{t_i : 1 \leq i \leq P\}}. \quad (1.5.4)$$

A set of tasks is said to be **load balanced** if β is close to one.

The amount the load balance β differs from the ideal case $\beta = 1$ measures the relative difference between the longest task and the average task, measured in terms of run time. That is,

$$1 - \beta = \frac{\max \{t_i : 1 \leq i \leq P\} - \text{ave} \{t_i : 1 \leq i \leq P\}}{\max \{t_i : 1 \leq i \leq P\}}. \quad (1.5.5)$$

A set of tasks is said to be load balanced if this difference is negligible. Note that we have compared the *average* time with the maximum time, not the minimum time. The relevance of this will become clearer below and in Section 2.3.3.

We have also defined load balance in terms of time of execution instead of amount of computational work to be done. This is because the performance (Definition 1.3.1) need not be the same for different tasks, and the cost of the computation is proportional to the time it takes, not to how many floating point operations get done. Of course, we will often try to achieve load balance by balancing the amount of work to be done, since we can frequently predict this in advance, whereas we rarely know the exact execution time in advance.

Example 1.5.8. Load balancing in the summation problem is effected by assigning the same number k of summands to each processor (cf. (1.5.1)). For perfect load balance, this requires that $N = Pk$. If N is not divisible by P , then this will not be possible. For the sake of argument, suppose that $N = k(P - 1) + 1$. One way to distribute the work is to let the first $P - 1$

processors sum k elements, with the last processor doing nothing. The time of execution is then proportional to k . For definiteness, suppose that the time units are chosen so that the constant of proportionality is one. Thus the execution time for process i is $t_i = k$ for $i = 1, \dots, P-1$ and $t_P = 0$, and the minimum time is zero. We can increase the minimum time by decreasing the load of other processors, but unless we can reduce them all there will be no reduction in total (parallel) run time. Therefore the minimum time plays very little role in determining the run time.

Our goal, then, is to create embarrassingly parallel sections of code (cf. (1.5.1)) with the largest possible granularity and the smallest possible amount of nonlocal memory references (often by communication) at the points where the independent results have to be merged (cf. (1.5.2)). Moreover, we must keep the load balanced among the different processors. Of course, it may not be possible to increase granularity and decrease communication simultaneously, and increasing the granularity often is equivalent to decreasing the parallelism, as in the summation problem parallelized via (1.5.1) and (1.5.2). Also, as the granularity gets smaller, the load balancing problem often becomes more difficult. For all of these reasons, the optimization problem for parallel computing can be quite complex and will likely entail significant compromise.

1.5.3 Prime number sieve

The prime number sieve⁶ provides an interesting example with a varying amount of parallelism. Recall that a prime number is an integer having no divisors other than itself and one. (An integer j is a divisor of another integer p if p/j is exactly an integer.)

The sieve works as follows. For a given integer k , suppose we have recorded the set $S(k)$ of prime numbers less than k (for example, $S(16) = \{2, 3, 5, 7, 11, 13\}$). Then we can check the primality of integers n less than k^2 by testing to see whether there are any divisors of n in $S(k)$ (if $n = j \cdot i < k^2$ then either i or j has to be less than k). So an algorithm for computing all primes is as follows.

Suppose $S(k)$ is known. Test the integers, n , in the range $k \leq n < k^2$ for divisors and if any primes are found, add them to $S(k^2)$. Note that each of these tests can be done independently of the others by a separate processor as long as it has access to all of $S(k)$. If done in parallel, the contributions to $S(k^2)$ have to be merged. Then set $k \leftarrow k^2$ and continue.

A pseudo-code for this algorithm is written as follows. It is a nested

⁶The sieve of Eratosthenes (276–196 B.C.) is a close cousin of the method we describe here. In addition to work in various areas of mathematics, Eratosthenes is also credited with a very accurate measurement of the diameter of the earth, and he was the third director of the famous library of Alexandria, which is thought to have contained hundreds of thousands of papyrus and vellum scrolls.

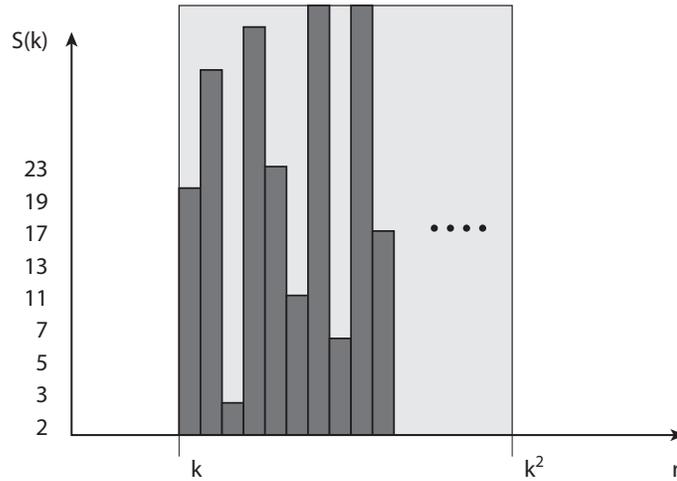


Figure 1.9 The iteration space for the prime number sieve. The n -axis has been modified to eliminate even numbers, numbers divisible by three, and so forth. The darkly shaded areas correspond to the actual values of n and π for which computation occurs, and the lightly shaded region is the maximum possible set of n and π values.

loop, three deep, but we omit the outermost loop which increments k .

```

loop on  $n = k, k + 1, \dots, k^2 - 1$ 
  loop on  $\pi \in S(k)$ 
    see if  $\pi$  divides  $n$  integrally
      if it does, exit the  $\pi$  loop since  $n$  is not prime
    end loop on  $\pi$ 
  if loop completes for all  $\pi \in S(k)$ , add  $n$  to  $S(k^2)$ 
end loop on  $n$ 

```

As written, this code will waste a great deal of time finding even values of n , so making the stride equal to two in the loop on n , and starting with k odd, will eliminate a substantial amount of unnecessary work. Further, we will assume that n divisible by three and five and perhaps more small primes are eliminated by adjusting the loop indices appropriately (see Exercise 1.9).

Note that the innermost operation of dividing π by n can be done independently of all others. Thus the loops can be parallelized in a number of ways. We could divide the “ n ” loop into P different parts, or we could divide the “ π ” loop into P different parts, or we could have a more complex organization. Recall the concept of iteration space in Definition 1.5.4. Figure 1.9 depicts graphically the iteration space for the prime number sieve. In this case, the iteration space is a subspace (shown in dark shading) of the Cartesian product (shown in light shading) of intervals $[k, k^2 - 1]$ and the set $S(k)$. The reason it is not all of the Cartesian product is that in

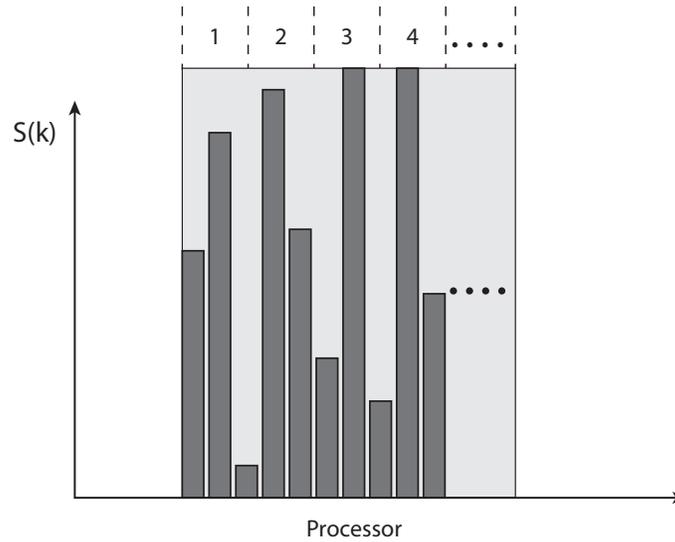


Figure 1.10 Parallelizing the prime number sieve with respect to the n loop using a strip decomposition.

executing the loops in the order indicated, the π index may not traverse all of its potential range (when a divisor is found). Note that in this case the precise iteration space is not known in advance but is determined as part of the computation.

The iteration space does not indicate *whether* loops are parallel or not. This can be only determined by studying the *dependences* in loops (see Chapter 4). However, for loops that are known to be trivially parallel, as in the sieve problem, different ways of parallelizing loops correspond to different geometric decompositions (Definition 1.5.5) of the iteration space. There are a variety of standard decompositions that are useful in parallelizing loops.

Definition 1.5.9. A **strip decomposition** or **slab decomposition** corresponds to a subdivision of only one of the dimensions, usually into segments of (about) equal length and consisting of contiguous elements in the iteration space. This corresponds to subdividing only one of the loops in a nested set of loops.

For example, Figure 1.10 shows a division of the “ n ” loop in the prime number sieve into P different strips. The corresponding pseudo-code looks like

```

 $b := (k^2 - k)/P$ 
for each processor  $p = 1, \dots, P$ 
  loop on  $n = k + (p - 1)b, \dots, k + p \cdot b$ 
    loop on  $\pi \in S(k)$ 

```

```
    see if  $\pi$  divides  $n$  integrally
      if it does, exit the  $\pi$  loop since  $n$  is not prime
    end loop on  $\pi$ 
  if loop completes for all  $\pi \in S(k)$ , add  $n$  to  $S_p(k^2)$ 
end loop on  $n$ 
end loop on  $p$ 
combine different  $S_p(k^2)$  into  $S(k^2)$ 
```

Note that the variable p plays the role of a unique processor identifier. $S_p(k^2)$ is the set of primes found by the p -th processor, and

$$S(k^2) = \bigcup_{p=1}^P S_p(k^2). \quad (1.5.6)$$

We did not describe an algorithm to form the union. This requires merging the separate sets into one set. It may not matter whether the primes come in any order, or one might decide smaller divisors are more likely and that it is more efficient to start checking with smaller primes first. In that case, the separate sets needed to be merged in sorted order. Parallel algorithms for sorting and merging will be discussed in Section 15.2.2.

Each processor will do different work (starting with the n loop) because p takes on different values. In this way, a *single program* executed by different processors can evaluate a portion of the iteration space described by the n and π loops, each processor producing different results even though the code is the same.

Definition 1.5.10. The **single program** parallel approach involves having only one program which will execute differently on different processors. The differences can occur through either implicit mechanisms such as compiler directives or explicit constructs such as references to the processor number executing the code. This approach is known as **SPMD (single program, multiple data)**.⁷

In the sieve example, the differences in execution on different processors all originate in the different values of the “processor I.D.” reflected in the value of p . The ability to use a single code to do parallel work greatly simplifies the task of writing parallel programs. This will be discussed more formally in Section 5.5. An alternate parallelization of the sieve is given in the following.

⁷Convention is that SPMD programs are written in per-process name spaces, in contrast to data parallel programs, which are written in a global name space. We will occasionally use the term *data parallel* loosely to include SPMD programs. Strictly speaking, HPF is a data parallel approach whereas **Pfortran** is an SPMD approach. These topics are discussed further in Chapters 5, 8 and 9.

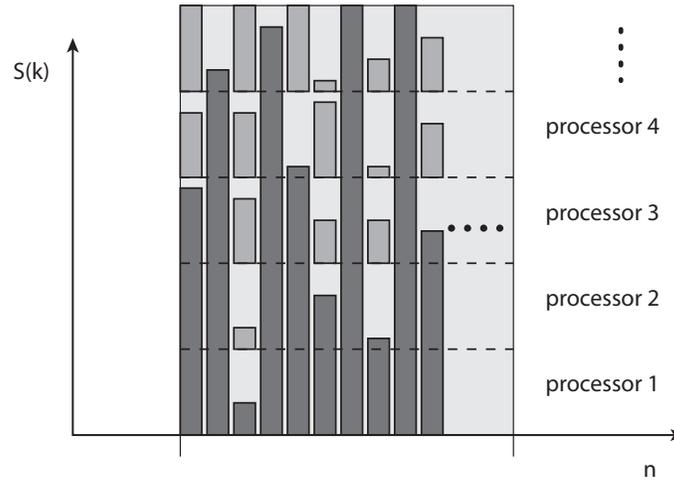


Figure 1.11 Parallelizing the prime number sieve with respect to the π loop using a strip decomposition.

Example 1.5.11. Dividing the π loop among processors as shown in Figure 1.11 corresponds to having each processor use only a restricted set of primes in its tests, but test all of the $n \in [k, k^2 - 1]$. In this case, some n will be divided by a larger set of primes π , involving more arithmetic work. The additional dark areas in Figure 1.11 indicate hypothetical extra work each processor would have to do (none of the integers in the dark area is a divisor, so the loop will not terminate until all in that area have been tested). Even if there is more arithmetic work done overall, it does not necessarily mean that the execution time is longer on a given parallel computer.

At the end, each processor has a set of candidate primes (numbers not divisible by its subset of primes), and the intersection of these must be formed to determine the actual set of primes. We postpone the discussion of parallel algorithms for forming set intersections until Chapter 6, Section 6.4.5.

The prime number sieve displays a number of interesting features. First of all, the amount of parallelism increases as k increases. Second, various parallelization strategies can be used, leading to quite different parallel algorithms for solving the same problem. Third, the amount of information that needs to be communicated (the merging of contributions to $S(k^2)$) cannot be predicted in advance. Finally, the load balance (Definition 1.5.7) is also unpredictable since the primality test may fail (when a divisor is found) after just a few tries, or it may succeed (the worst case: all potential divisors in $S(k)$ have to be tested). We leave the problem of load balancing the prime number sieve to the exercises (see Exercises 1.8 and 1.12).

The iteration space for the complete algorithm is of course three-dimensional. However, there is an essential *dependence* (see Chapter 4)

between the k iterations (which go $k^2, k^4, k^8, \dots, k^{2^i}, \dots$) since we do not know S for one iteration until all previous iterations have been completed. This dependence makes it impossible to parallelize the outermost (k) loop in a straightforward way.

1.6 MESH-BASED APPLICATIONS

We now give some examples of the type of applications that will be considered as one of the main topics of this book. We start with *ordinary differential equations* and standard types of discretizations. An ordinary differential equation (o.d.e.) provides a model in which the rate of change of a quantity is related to that quantity by an explicit function. The o.d.e. is solved for a function, rather than a number. Few such practical equations admit an explicit solution, making way for the application of numerical solutions.

Ordinary differential equations and their numerical solution will give us some more challenging examples to study and will also introduce more fundamental concepts. We will necessarily draw upon some ideas from elementary numerical analysis, but we will introduce the necessary notation and background as we go.⁸

1.6.1 Initial value problems

Let us begin with an ordinary differential equation

$$\frac{du}{dt} = f(t, u) \tag{1.6.1}$$

with an initial condition provided at $t = 0$:

$$u(0) = u_0$$

for some given data value u_0 . Under suitable smoothness conditions on f , this equation has a unique solution, u , which exists for some time interval $0 \leq t \leq T$.

We may be able to solve this analytically in terms of expressions that are familiar to us. For example, if

$$f(t, u) = f(u) := -u^2, \tag{1.6.2}$$

then

$$u(t) = \frac{1}{t + 1/u_0} \tag{1.6.3}$$

is the solution (see Figure 1.12).

⁸The differential calculus is credited to Sir Isaac Newton (1642–1727) and Gottfried Wilhelm von Leibniz (1646–1716). The controversy about priority regarding this development has been discussed in recent popular literature [137]. Newton's discoveries in physics and celestial mechanics culminated in the theory of universal gravitation, which will appear in Chapter 13.

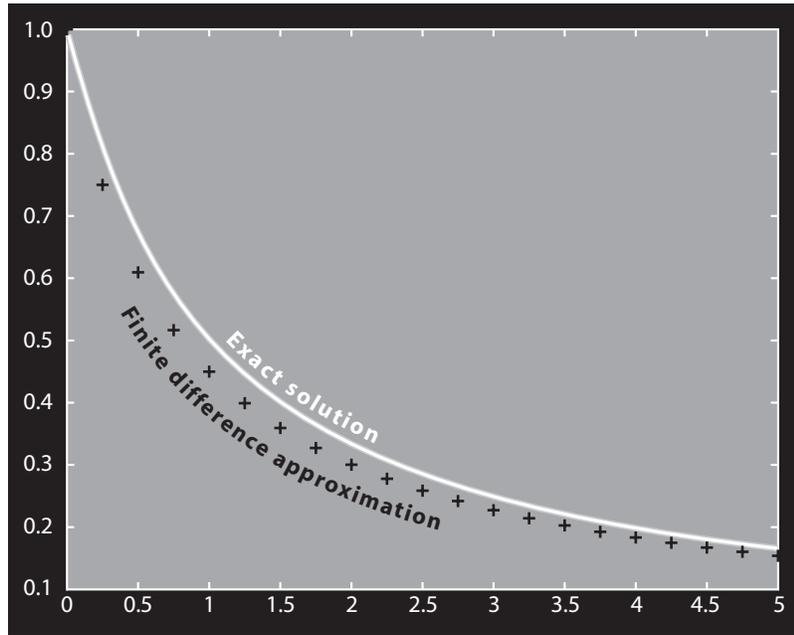


Figure 1.12 Solution of the ordinary differential equation $\frac{du}{dt} = -u^2$ and the finite difference approximation using the explicit Euler discretization.

More likely, we will not be able to recognize a solution as a combination of known functions. However, we may be primarily interested in only numerical values of u at specified points in time, or we may just want a graph of u that indicates its general behavior.

A *discretization* scheme can be used to generate approximations to the values of u at a discrete set of points. For example, let $\Delta t > 0$ be a fixed parameter, and use the definition of derivative to make the *finite difference approximation*

$$\frac{du}{dt}(t) \approx \frac{u(t + \Delta t) - u(t)}{\Delta t}. \quad (1.6.4)$$

Inserting this approximation into equation (1.6.1) we find

$$\frac{u(t + \Delta t) - u(t)}{\Delta t} \approx f(t, u(t)). \quad (1.6.5)$$

Define a sequence of time values $t_n := n\Delta t$, and using (1.6.5), a corresponding sequence of values u_n via the *explicit Euler method*

$$u_{n+1} = u_n + \Delta t f(t_n, u_n). \quad (1.6.6)$$

Here, u_n is intended to be an approximation to $u(t_n)$. Under suitable smoothness conditions on f , one can show that

$$|u_n - u(t_n)| \leq C\Delta t \quad \forall n \leq T/\Delta t, \quad (1.6.7)$$

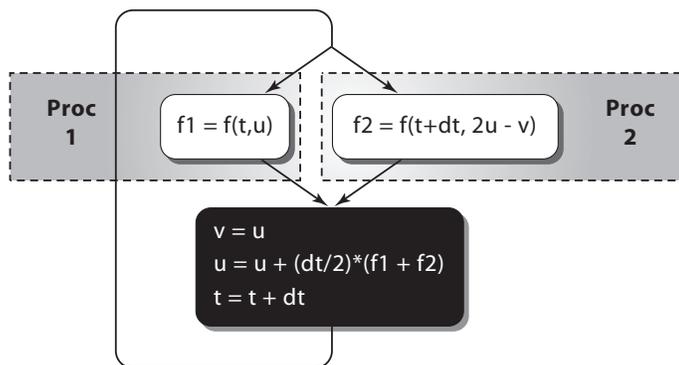


Figure 1.13 Flow chart for trapezoidal rule algorithm displaying independent tasks for $P = 2$ processors. Processor 1 computes the function f_1 , and processor 2 computes the function f_2 .

where C is a constant depending only on f and T . Figure 1.12 shows u_n for $0 \leq n \leq 20$ for $\Delta t = 0.25$ and f as defined in (1.6.2).

It is not at all necessary to have a fixed *time step* Δt . The approximation (1.6.4) allows one to define

$$u_{n+1} = u_n + (t_{n+1} - t_n)f(t_n, u_n). \quad (1.6.8)$$

for arbitrary sequences $0 = t_0 < t_1 < \dots < t_n$.

Note that u_n depends on all u_i for $i < n$ and there is no simple way to remove these dependences. One approach to creating parallelism is to switch to a more complex difference method, with the side benefit of having a higher order of convergence. Consider

$$u_{n+1} = u_n + \frac{t_{n+1} - t_n}{2} (f(t_n, u_n) + f(t_{n+1}, 2u_n - u_{n-1})). \quad (1.6.9)$$

Here $2u_n - u_{n-1}$ is a second order approximation to u_{n+1} , so (1.6.9) corresponds to a variant of the trapezoidal rule [27]. If the function f is expensive to calculate, then it can be useful to compute $f(t_n, u_n)$ and $f(t_{n+1}, 2u_n - u_{n-1})$ on separate processors. The resulting scheme can potentially use larger time steps since (Exercise 1.17)

$$|u_n - u(t_n)| \leq C \max_i (t_i - t_{i-1})^2 \quad \forall t_n \leq T. \quad (1.6.10)$$

The basic loop implementing (1.6.9) can be parallelized using an approach referred to as *task* or *procedural* parallelism.

Definition 1.6.1. An algorithm can be parallelized using **task parallelism** or **procedural parallelism** when there are independent parts that can be executed as separate procedures (Definition 1.5.1) without the need for communication between them.

The independent parts of an algorithm in task (procedural) parallelism

are trivially (embarrassingly) parallel in the terminology of Definition 1.5.3.

Using task or procedural parallelism to parallelize (1.6.9) is depicted in Figure 1.13. The two dotted boxes indicate the parts of the algorithm that can be done in parallel, namely, the function evaluations $f(\cdot, \cdot)$ with different arguments. Here we make the fundamental assumption that there are no *side effects* (see page 99) associated with evaluating $f(\cdot, \cdot)$. In many cases, this step can be a large part of the computation (see Exercises 8.7 and 10.3 for a contrived example).

In the case that the function f in (1.6.1) is an affine function independent of t (i.e., $f(t, x) = a + bx$), then difference methods such as (1.6.9) become a **linear recursion**. Solving such a system is equivalent to solving a banded triangular linear system of equations, something discussed at length in Chapter 12.

1.6.2 Boundary value problems

Ordinary differential equations of higher order can have more than one data value specified. An important special case of this occurs when data are given at different points which form the end points for the interval of interest for the solution. These points are then the *boundary* points, and the data are known as the *boundary values*. In such a case, the independent variable often connotes something distinct from “time,” so we will switch to calling the variable x instead of t .

Consider the simple second order ordinary differential equation

$$-\frac{d^2u}{dx^2} = f(x) \tag{1.6.11}$$

together with specified boundary values

$$u(0) = a, \quad u(1) = b \tag{1.6.12}$$

for some given data values a and b .

Using an approximation such as (1.6.4) twice we obtain a system of equations

$$-u_{n-1} + 2u_n - u_{n+1} = (\Delta x)^2 f(x_n). \tag{1.6.13}$$

Here we take $x_n = n\Delta x$ with $\Delta x = 1/(N + 1)$ for some integer N . Equation (1.6.13), for $n = 1, \dots, N$, forms a system of equations for u_1, \dots, u_N , where we interpret $u_0 := a$ and $u_{N+1} := b$ where they occur in (1.6.13) (see Exercise 1.14).

A more general set of equations can be derived, analogous to (1.6.8), on an arbitrary *mesh* $0 = x_0 < x_1 < \dots < x_n < \dots < x_{N+1} = 1$. These equations are of the form

$$-\frac{2}{x_{n+1} - x_{n-1}} \left(\frac{u_{n+1} - u_n}{x_{n+1} - x_n} - \frac{u_n - u_{n-1}}{x_n - x_{n-1}} \right) = f(x_n). \tag{1.6.14}$$

These equations can be simplified somewhat by collecting terms, introducing notation for the local mesh size, e.g.,

$$h_n := x_n - x_{n-1}, \quad (1.6.15)$$

and scaling the equations by the factor $\frac{h_{n+1}+h_n}{2}$ in the form

$$\begin{aligned} \frac{h_{n+1} + h_n}{2} f(x_n) &= -\frac{u_{n+1} - u_n}{h_{n+1}} + \frac{u_n - u_{n-1}}{h_n} \\ &= \left(\frac{1}{h_{n+1}} + \frac{1}{h_n} \right) u_n - \frac{1}{h_{n+1}} u_{n+1} - \frac{1}{h_n} u_{n-1}. \end{aligned} \quad (1.6.16)$$

This represents a symmetric, tridiagonal matrix, and it can be shown to be positive definite. In particular, the i -th row of the matrix has entries

$$a_{i,i-1} = \frac{-1}{h_i}, \quad a_{i,i} = \frac{1}{h_i} + \frac{1}{h_{i+1}}, \quad a_{i,i+1} = \frac{-1}{h_{i+1}} \quad (1.6.17)$$

for $1 < i < N$. See Exercise 1.15 regarding the remaining two equations.

The set of equations (1.6.16) (together with the two equations from Exercise 1.15) can be written succinctly in matrix form as

$$\mathbf{A}\mathbf{U} = \mathbf{F}, \quad (1.6.18)$$

where \mathbf{A} is the matrix with entries $(a_{i,j})$, \mathbf{F} is the vector whose i -th entry is $f(x_i)$, and \mathbf{U} is the vector whose i -th entry is u_i .

Gaussian elimination is a “direct” (non-iterative) method for solving the system (1.6.16) which will be discussed at length in Chapter 12. For the moment, we will turn to a discussion of iterative methods whose parallelizations are easier to describe.

1.6.3 Iterative equation solvers

The solution of the system (1.6.16) can be done by a variety of iterative methods. We will consider several techniques, such as stationary methods, conjugate gradients, and multigrid. One of the simplest is the Jacobi⁹ iteration. Equation (1.6.16) can be rewritten in “fixed point” form as

$$\left(\frac{1}{h_{n+1}} + \frac{1}{h_n} \right) u_n = \frac{1}{h_{n+1}} u_{n+1} + \frac{1}{h_n} u_{n-1} + \frac{h_{n+1} + h_n}{2} f(x_n). \quad (1.6.19)$$

Rescaling each equation yields

$$u_n = \frac{h_n}{h_{n+1} + h_n} u_{n+1} + \frac{h_{n+1}}{h_{n+1} + h_n} u_{n-1} + \frac{h_n h_{n+1}}{2} f(x_n). \quad (1.6.20)$$

⁹Carl Jacobi (1804–1851) is widely known for the determinant appearing in the formula for changing variables in integration, but he was also a pioneer in computational techniques and in studying first-order systems of partial differential equations.

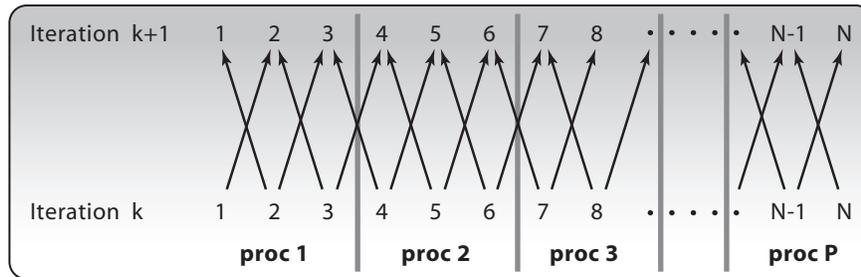


Figure 1.14 Data dependence in the Jacobi iteration.

This suggests an iterative method known variously as “fixed point” iteration or Jacobi’s method:

$$u_n^{k+1} = \frac{h_n}{h_{n+1} + h_n} u_{n+1}^k + \frac{h_{n+1}}{h_{n+1} + h_n} u_{n-1}^k + \frac{h_n h_{n+1}}{2} f(x_n). \quad (1.6.21)$$

When (1.6.21) converges (it does for arbitrary initial vectors (u_n^0) , e.g., $u_n^0 = 0$ for all n), then the limit naturally must solve (1.6.20) and equivalently (1.6.16).

The algorithm (1.6.21) can be parallelized using *data parallelism* as defined in Definition 1.5.6. The data-parallel operation in (1.6.21) can be represented as multiplying a matrix times the vector (u_n^k) , followed by a vector addition. Operations on vectors are typical data-parallel operations. They are parallelized simply by dividing the index space for the index n for the various vectors. This is depicted in Figure 1.14. Here, several indices are assigned to a given processor, p , say the interval $n_p \leq n < n_{p+1}$. Then processor p computes (1.6.21) for these n . To do so, it must know the values u_n^k for $n_p - 1 \leq n \leq n_{p+1}$ (we ignore for the moment the very ends of the index set, and assume some meaning is attached to $n = 0$ and $n = N + 1$).

In order to repeat the calculation (to perform the *iteration*), the values $u_{n_p-1}^k$ and $u_{n_{p+1}}^k$ will have to be obtained from neighboring processors as shown in Figure 1.14. The arrows indicate *dependences* (see Definition 4.1.3) from one iteration to the next.

There is no natural task parallelism as in Figure 1.13 since everything depends on something. On the other hand, there is data parallelism in the sense that the single operation of updating (u_n) acts on a large amount of data in parallel. Subdividing this single operation creates independent tasks, although communication between them is generated as a byproduct. However, the specific tasks are arbitrary, depending on the choice of subdivision of the index set, as opposed to being naturally part of the algorithm as in Section 1.6.1.

Data parallelism can be found in solving an ordinary differential equation (1.6.1) when the unknown \mathbf{u} denotes a vector of substantial size. In such a case, the computation of the vector quantity $\mathbf{f}(t, \mathbf{u})$ can often involve

opportunities for data parallelism. An example of this type can be found in Section 2.6.

Nonlinear problems can be solved by iterative methods as simply as linear ones. For example, the fixed-point iteration

$$u_n^{k+1} = \frac{h_n}{h_{n+1} + h_n} u_{n+1}^k + \frac{h_{n+1}}{h_{n+1} + h_n} u_{n-1}^k + \frac{h_n h_{n+1}}{2} f(x_n, u_n^k) \quad (1.6.22)$$

can be computed for an arbitrary nonlinear function $f(x, u)$. This corresponds to solving a finite difference equation for the boundary value problem

$$\begin{aligned} -\frac{du^2}{dx^2} &= f(x, u), \\ u(0) &= a, \quad u(1) = b. \end{aligned} \quad (1.6.23)$$

1.7 PARALLEL PERSPECTIVES

It is useful to understand the role of parallelism as a potential tool in problem solving. Parallelization is not a game unto itself, to be pursued in a vacuum. Rather it is a tool that can be effective in appropriate situations. If it does not provide significant performance gains, it may not be of interest. Here we give a brief example of how parallelization might play a role in a larger context, and we also review its historical role in computing.

1.7.1 Other options

Solving technical problems can be done in a variety of ways. We have described in Figure 1.15 a number of options that can occur. First, it is not necessary to do computation at all. Some paths in the tree refer to using analytical or experimental methods. The beginning point is not a sequential code, although often this is the case.

A good example of a problem of this type is **drug discovery**, the endeavor of finding new pharmaceuticals to fight disease. One popular approach involves combing the rain forests of the equatorial regions for exotic plants and testing them for efficacy. Another involves primarily laboratory work with test-tubes of chemicals. Another uses the computer to postulate and analyze new compounds at the molecular level. In creating any one pharmaceutical, all three of these approaches might be brought to bear in combination at various stages, so the real problem graph will not be a tree but something with more complex interactions.

Even when the computational option has been chosen for some reason, there still remain a variety of choices one can make at different stages. There are different models which provide the same level of accuracy for a given physical phenomenon, so it may be useful to choose different models for different purposes. Any given model will typically have different ways to

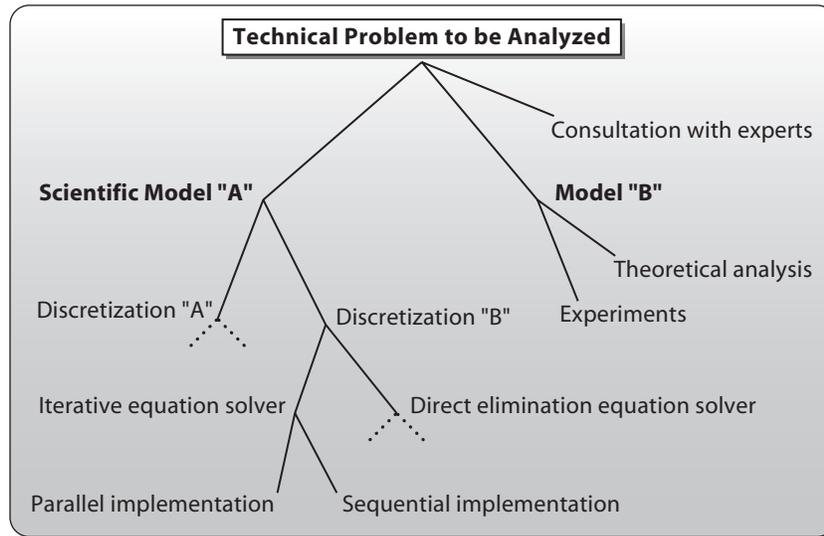


Figure 1.15 The “problem tree” for scientific problem solving. There are many options to try to achieve the same goal.

discretize it, and any discretization will have multiple ways to solve the resulting discrete equations.

It is always allowed to climb higher in the tree to find better path to the solution using new technology, such as parallel computation. Any given physical model, or discretization, or equation solver may lead to a difficult parallel computation. However, an equivalent result may be obtained more efficiently using a parallel computer by using a different model or discretization or equation solution strategy. This alternative approach should always be kept in mind when trying to utilize a parallel computer to solve a problem.

1.7.2 Parallelism in computing history

Parallelism has a long history in the development of computer systems. One early example is the representation of natural numbers as a collection of bits and the corresponding development of hardware that could work on such pairs of collections to do arithmetic. Such arithmetic units are in effect parallel computers. This type of parallelism is in use in microprocessors which now permeate everyday life, not only in pocket calculators, but in the “smart” electronics in automobiles, microwave ovens, and other home appliances.

Operating systems provided early impetus to parallel (or concurrent) programming and many of the early concepts in parallel programming languages [78, 46] were developed in this context. Much of the early work in synchronization using semaphores, guards, and critical sections had its

roots in operating system problems. Indeed, a steadfast component of Unix kernels is the message queue, an interprocess communication primitive not unlike its parallel scientific counterpart, the MPI library. Unlike operating systems, however, the scientific program parallelism we are primarily concerned with is the data parallel model defined above.

I/O devices have always functioned at a slower rate than the processors to which they are connected. The use of a separate processor (or just a separate process) to handle I/O is an example of parallel processing. These methods were employed as early as the 1950s in the Univac 1, the first computer to overlap program execution with some I/O [83].¹⁰ Early operating systems for personal computers were able to allow printers to queue data and allow control of the computer to return to the user.

Vector processor architectures introduced in the 1960s developed numerous innovations in addressing the problem of getting data to processors on time. Methods included memory interleaving to increase memory-to-processor bandwidth, overlap of computation with memory accesses (called prefetching), and out-of-order execution to improve reuse and tolerate memory delays such as with the Tomasulo algorithm developed for the IBM 360/91 [140]. Recent counterparts of the Tomasulo algorithm and CDC 6000/7000 series scoreboard [83] appear in the out-of-order instruction execution for the Intel Xeon and P4 processor lines [85]. With the benefits of these earlier developments, contemporary microprocessors execute multiple instructions simultaneously using pipelined functional units (Section 3.4).

It is natural that early vector processing methods have come full circle into today's modern commodity pipelined and multi-threaded processors. After all, computer performance depends fundamentally on having data available to the CPU with minimum delay. The methods of data reuse and tolerating memory access times appear in various guises in both architecture and software design. We will see that in order to get good performance on even a single processor of this type, it is necessary to understand basic parallel computing.

In Section 3.1.2, the notion of a memory *cache* will be discussed. As will be seen in Figure 3.6, this involves a type of parallel processing in the memory system. Moreover, one of the key issues in parallel computing is the communication that must be done between different processes. With current workstations, there are already several levels of cache, and the movement of data among them is a prime consideration in achieving good performance. Thus the issues we are studying with regard to parallelism in the large have

¹⁰The Univac 1, the Universal Automatic Computer, was the first commercial computer and successor to the ENIAC, the Electronic Numerical Integrator and Computer, a 30-ton computer developed at the University of Pennsylvania during World War II. Delivered to the Census Bureau in 1951, the Univac 1 weighed approximately 8 tons and could perform about 1,000 calculations per second. The first commercial sale by the Eckert-Mauchly Computer Corporation of Philadelphia was followed shortly by Remington-Rand's purchase of Eckert-Mauchly. A statistical model run on the Univac 1 and informed by early election returns predicted correctly Dwight Eisenhower's victory in the 1952 presidential election.

an important correspondence with different types of parallelism in single processors, in terms of both instruction parallelism and data movement to and from cache.

For a more detailed history of parallelism, we refer to [80] and [81]. For an intriguing look at an early proposal, see [113].

1.8 EXERCISES

Exercise 1.1. Beyond “mega” (10^6) and “giga” (10^9) are “tera” (10^{12}), “peta” (10^{15}) and “exa” (10^{18}). Determine the correct time unit (millisecond, microsecond, nanosecond, picosecond, femtosecond, ...) that it takes a single computer to do a floating point operation if it does (1) one megaflop per second, (2) one gigaflop per second, (3) one teraflop per second, and (4) one petaflop per second.

Exercise 1.2. What is the maximum number of floating point operations possible in Example 1.3.6 for a 99% cache-hit rate and for a 1% hit rate? Assume that $\rho_{\text{WM}} = 2$.

Exercise 1.3. What is the ratio ρ_{WM} of the number of floating point operations to the number of data values that have to be obtained from memory, or written to memory, in the computation of the product of two square matrices $\mathbf{A} = (a_{ij})$ and \mathbf{B} which is defined by

$$(\mathbf{AB})_{ij} := \sum_{k=1}^n a_{ik}b_{kj} \quad \text{for } i, j = 1, \dots, n \quad (1.8.1)$$

as a function of n ? (Assume that both \mathbf{A} and \mathbf{B} must be obtained from memory and the product is written back to memory. However, the denominator should just be the volume of the data, not the number of memory references that might occur in particular algorithm. For example, the term b_{11} occurs n times in (1.8.1) but should be counted only once.)

Exercise 1.4. What is the ratio of the number of floating point operations per processor to the number of data values that have to be obtained by the individual processors from other processors in the parallelization of (1.6.21) using P processors?

Exercise 1.5. Write a program to compute a summation as in (1.3.2). Test the program by computing π by the summation (1.3.1), i.e., with $a_i = (-1)^{i+1}/(2i - 1)$. Determine the performance on a single processor as a function of N . A model for expected time performance t_N for computing the summation is $t_N = a + bN$. Use your timing data to estimate the parameters a and b . What are the critical values of t and N below which the timing is uncertain (see page 6) for each part? Report what computer

and what timer you are using, and what its time resolution is purported to be. (Hint: plotting the observed time T_N as a function of N should, if $T_N \approx t_N$, give points lying nearly on a line with slope b ; the asymptote of this line to $N = 0$ gives an estimate of a . Then plotting $(T_N - a)/N$ as a function of N should be nearly constant. Often when nearing the resolution of the timer, the data will become erratic, giving a measure of where the limit is.)

Exercise 1.6. Write a program to compute a summation as in (1.3.2). Test the program by computing π by the summation (1.3.1), i.e., with $a_i = (-1)^{i+1}/(2i - 1)$. Determine the performance on a single processor as a function of N for both the computation of the a_i 's and the sum (i.e., give separate performance estimates for the two parts of the computation). Determine the relationship (if any) between these times for different values of N and explain why you think this is reasonable. Report what computer and what timer you are using, and what its time resolution is purported to be.

Exercise 1.7. Write a program to compute a summation as in (1.3.2). Test the program by computing π by the summation (1.3.1). Determine the performance on a single processor as a function of N and compare this with the diagram in Figure 1.2. To do so, you need to compute $a_i = (-1)^{i+1}/(2i - 1)$ separately and store it in an array. Just time the computation of the sum, not the computation of a_i . It may be necessary to repeat this k times to get an accurate timing, in such a way that $k \cdot N$ remains roughly constant as N increases. Be sure to divide by k to get the time for one sum. Try to find a virtual memory (e.g. Unix) machine with a small amount of memory so that you can do a computation involving paging to disk. Plot the results of your timings as in Figure 1.2 by plotting N/T_N as a function of N . (Explain why this gives a measure of performance for this calculation, and explain what the units are.) Choose values of N on a logarithmic scale, e.g., $n = 2^i$ for $i = 1, 2, \dots, I$ for a suitable value of I . Make sure that your code is compiled with an optimization level to give maximum performance. Explain what computer and what timer you are using, and what its time resolution is purported to be.

Exercise 1.8. In the prime number sieve, determine the number, P , of parallel tasks that can be done for a given k when parallelizing by subdividing the n loop. Describe a way to achieve a load balanced algorithm for P processors (with P fixed and k sufficiently large). How much communication will be involved in your approach?

Exercise 1.9. Modify the loop on n in the prime number sieve to eliminate considering n divisible by 2, 3, and 5. How would you do this for more small primes?

Exercise 1.10. Consider the Jacobi iteration depicted in Figure 1.14. Describe a strategy for load balancing this parallel algorithm.

Exercise 1.11. Other decompositions of a loop can be made besides the ones shown in Figures 1.8, 1.10, and 1.11. The **cyclic decomposition** or **modulo decomposition** refers to distributing the n -th loop index to processor number $n \pmod{P}$. Note that this assumes that the processors are numbered from 0 to $P - 1$ as is done in many systems, since this is the range of values in “modulo” arithmetic. Make a copy of the iteration space in Figure 1.8 and indicate the processor allocation for a cyclic decomposition of the n loop using $P = 3$ processors (numbered 0, 1, 2).

Exercise 1.12. Make a copy of the iteration space in Figure 1.9 and indicate the processor allocation for a cyclic decomposition (see Exercise 1.11) of the n loop using $P = 3$ processors (numbered 0, 1, 2).

Exercise 1.13. The **block decomposition** is a generalization of the strip decomposition shown in Figures 1.10 and 1.11. It corresponds to dividing the iteration space into blocks, in this case distributing both the n -th loop index and the π -loop index. Make a copy of the iteration space in Figure 1.9 and indicate the processor allocation for a block decomposition using $P = 4$ processors, where both the n -th loop and the π -loop are divided equally.

Exercise 1.14. Complete the definition of the matrix in (1.6.13) by making the indicated substitutions for u_0 and u_{N+1} . Show that matrix is symmetric.

Exercise 1.15. Complete the definition of the matrix (cf. (1.6.17)) in (1.6.14) by making the indicated substitutions for u_0 and u_{N+1} . Show that matrix is symmetric.

Exercise 1.16. Prove that the modified trapezoidal rule (1.6.9) is stable [27].

Exercise 1.17. Prove that the modified trapezoidal rule (1.6.9) satisfies (1.6.10). (Hint: use Exercise 1.16.)

Exercise 1.18. Show that matrix (1.6.17) has no negative eigenvalues. (Hint: use Gerschgorin’s theorem [139].)

Exercise 1.19. Show that (1.6.21) converges. (Hint: use Gerschgorin’s Theorem [139].)