21
Parallel Computing

In this chapter, we examine parallel computers and one method used to program them. This subject continues to change rapidly, and, accordingly, the meaning of various terms, as well as the way to program them also changes. Our view is influenced by the overriding aim of this book, namely, how to use the computer to assist with science. By the same token, we view much of the present day developments in parallel computers as being of limited usefulness because it is so hard to implement general applications on these machines. In contrast, the computer manufacturers view their creations as incredibly powerful machines, as indeed they are for some problems.

This chapter contains a high-level survey of parallel computing, while Chap. 22 contains a detailed tutorial on the use of the Message Passing Interface (MPI) package. An earlier package, Parallel Virtual Machine (PVM), is still in use, but not as popular as MPI. If you have not already done so, we recommend that you read Chap. 13 before you proceed with this chapter. Our survey is brief and given from a practitioner's point of view. The text [51] surveys parallel computing and MPI from a computer science point of view, and provides some balance to this chapter. References on MPI include Web resources [52–54] and the texts [51, 55, 56]. References on parallel computing include [51, 57–60].

Problem: Start with the program you wrote to generate the bifurcation plot for bug dynamics in Chap. 19 and modify it so that different ranges for the growth parameter μ are computed simultaneously on multiple CPUs. Although this small problem is not worth investing your time to obtain a shorter turnaround time, it is worth investing your time to gain some experience in parallel computing. In general, parallel computing holds the promise of permitting you to get faster results, to solve bigger problems, to run simulations at finer resolutions, or to model physical phenomena more realistically; but it takes some work to accomplish this.

Computationyal Physics. Problem Solving with Computers (2nd edn).
Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu

ISBN: 978-3-527-40626-5

21.1 Parallel Semantics (Theory)

We have seen in Chap. 13 that many of the tasks undertaken by a high-performance computer are run in parallel by making use of internal structures such as pipelined and segmented CPUs, hierarchical memory, and separate I/O processors. While these tasks are run "in parallel," the modern use of *parallel computing* or *parallelism* denotes applying multiple processors to a single problem [51]. It is a computing environment in which some numbers of CPUs are running asynchronously and communicating with each other in order to exchange intermediate results and coordinate their activities.

For instance, consider matrix multiplication in terms of its elements

$$[B] = [A][B] \qquad \Rightarrow \qquad B_{i,j} = \sum_{k=1}^{N} A_{i,k} B_{k,j} \tag{21.1}$$

Because the computation of a $B_{i,j}$ for particular values of i and j is independent of the computation for all other values, each $B_{i,j}$ could be computed in parallel, or each row or column of $[B]$ can be computed in parallel. However, because the $B_{k,j}$ on the RHS of (21.1) must be the "old" values that existed before the matrix multiplication, some communication among the parallel processors is required to ensure that they do not store the "new" values of $B_{k,j}$ before all of the multiplications are complete. This $[B] = [A][B]$ multiplication is an example of *data dependency*, in which the data elements used in the computation depend on the order in which they are used. In contrast, the matrix multiplication $[C] = [A][B]$ would be a *data parallel* operation, in which the data can be used in any order. So already we see the importance of communication, synchronization, and understanding of the mathematics behind an algorithm for parallel computation.

The processors in a parallel computer are placed at the *nodes* of a communication network. Each node may contain one, or a small number of CPUs, and the communication network may be internal to, or external to the computer. One way of categorizing parallel computers is by the approach they employ to handle instructions and data. From this viewpoint there are three types of machines.

- **Single instruction, single data (SISD):** These are the classic (VonNeumann) serial computers executing a single instruction on a single data stream before the next instruction and next data stream are encountered.
- **Single instruction, multiple data (SIMD):** Here instructions are processed from a single stream, but the instructions act concurrently on multiple data elements. Generally the nodes are simple and relatively slow, but are large in number.

• **Multiple instructions, multiple data (MIMD):** In this category each processor runs independently of the others with independent instructions and data. These are the type of machines that employ *message passing* packages, such as MPI, to communicate among processors. They may be a collection of workstations linked via a network, or more integrated machines with thousands of processors on internal boards, such as the Blue Gene computer. These computers, which do not have a shared memory space, are also called *multicomputers*. Although these types of computers are some of the most difficult to program, their low cost and effectiveness for certain classes of problems have led to their being the dominant type of parallel computer at present.

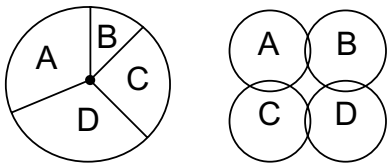

Fig. 21.1 *Left:* Multitasking of four programs in memory at one time. On a SISD computer the programs are executed in "round-robin" order. *Right:* Four programs in the four separate memories of a MIMD computer.

The running of independent programs on a parallel computer is similar to the multitasking feature used by Unix and PCs. In multitasking, symbolized (Fig. 21.1, left), several independent programs reside in the computer's memory simultaneously and share the processing time in a round-robin or priority order. On a SISD computer, only one program is running at a single time, but if other programs are in memory, then it does not take long to switch to them. In multiprocessing, these jobs may all be running at the same time, either in different parts of memory or on the memory of different computers (Fig. 21.1, right). Clearly, multiprocessing gets complicated if separate processors are operating on different parts of the *same* program, because then synchronization and load balance (keeping all processors equally busy) are concerns.

21.1.1 Granularity

In addition to instruction and data streams, another way to categorize parallel computation is by *granularity*. A *grain* is defined as a measure of the computational work to be done, more specifically, the ratio of computation work to communication work.

• **Coarse-grain parallel:** Separate programs running on separate computer systems with the systems coupled via a conventional communication network. To illustrate, six Linux PCs sharing the same files across a network but with a different central memory system for each PC. Each computer could

be operating on a different and independent part of one problem at the same time.

- **Medium-grain parallel;** Several processors executing (possibly different) programs simultaneously, while accessing a common memory. The processors are usually placed on a common *bus* (communication channel) and communicate with each other through the memory system. Medium-grain programs have different, independent, *parallel subroutines* running on different processors. Because the compilers are seldom smart enough to figure out which parts of the program to run where, the user must include the multitasking routines into the program.[1]
- **Fine-grain parallel:** As the granularity decreases and the number of nodes increases, there is an increased requirement for fast communication among the nodes. For this reason fine-grain systems tend to be custom-designed machines. The communication may be via a central bus or via shared memory for a small number of nodes, or through some form of high-speed network for massively parallel machines. In this latter case, the compiler divides the work among the processing nodes. For example, different `for` loops of a program may be run on different nodes.

21.2 Distributed Memory Programming

An approach to concurrent processing that, because it is built from commodity PCs, has gained dominance acceptance for coarse- and medium-grain systems is *distributed memory*. In it, each processor has its own memory and the processors exchange data among themselves through a high-speed switch and network. The data exchanged or *passed* among processors have encoded *To* and *From* addresses and are called *messages*. The *clusters* of PCs or workstations that constitute a *Beowulf*[2] are examples of distributed memory computers. The unifying characteristic of a cluster is the integration of highly replicated compute and communication components into a single system, with each node still able to operate independently. For a Beowulf cluster, the components are commodity ones designed for a general market, as is the communication

[1] Some experts define our medium as coarse, yet this fine point changes with time.

[2] Presumably there is an analogy between the heroic exploits of the son of Ecgtheow and the nephew of Hygelac in the 1000 C.E. poem *Beowulf*, and the adventures of us common folk assembling parallel computers from common elements that surpassed the performance of major corporations and their proprietary, multimillion dollar supercomputers.

network and its high-speed switch (special interconnects are used by major manufactures such as SGI and Cray, but they do not come cheaply). Note, a group of computers connected by a network may also be called a "cluster," but unless they are designed for parallel processing, with the same type of processor used repeatedly, and with only a limited number of processors (the *front end*) onto which users may log in, they would not usually be called a Beowulf.

The literature contains frequent arguments concerning the differences between clusters, commodity clusters, Beowulfs, constellations, massively parallel systems, and so forth [59, 61]. Even though we recognize that there are major differences between the clusters on the *Top500* list of computers, and the ones which a university researcher may set up in his or her lab, we will not distinguish these fine points in the introductory materials we present here.

For a messages-passing program to be successful, the data must be divided among nodes so that, at least for a while, each node has all the data it needs to run an independent subtask. When a program begins execution, data are sent to all nodes. When all nodes have completed their subtasks, they exchange data again in order for each node to have a complete new set of data to perform the next subtask. This repeated cycle of data exchange followed by processing continues until the full task is completed. Message-passing MIMD programs are also *single-program multiple-data* programs, which means that the programmer writes a single program that is executed on all of the nodes. Often a separate host program, which starts the programs on the nodes, reads the input files and organizes the output.

21.3 Parallel Performance

Imagine a cafeteria line in which all the servers appears to be working hard and fast, yet the ketchup dispenser has a seed partially blocking its output, and so everyone in line must wait for the ketchup lovers up front to ruin their food before moving on. This is an example of the slowest step in a complex process determining the overall rate. An analogous situation holds for parallel processing, where the "seed may be the issuing and communicating of instructions. Because the computation cannot advance until all instructions have been received, this one step may slow down or stop the entire process.

As we will soon demonstrate that the speedup of a program will not be significant unless you can get $\sim$90% of it to run in parallel. An infinite number of processors running a half serial program can, at best, attain a speedup of 2. This means that you need to have a computationally intense problem to make parallelization worthwhile, and that much of the speedup will probably be obtained with only a small number of processors. This is one of the

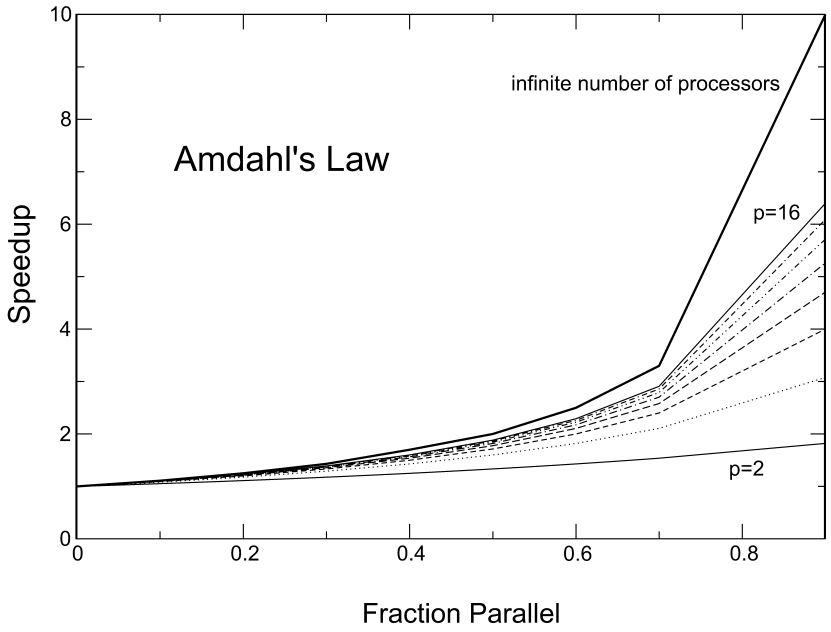

Fig. 21.2 The theoretical speedup of a program as a function of the fraction of the program that potentially may be run in parallel. The different curves correspond to different numbers of processors.

reasons why some proponents of parallel computers with thousands of processors suggest that you not apply the new machines to the old problems, but rather look for new problems which are both big enough and well suited for massively parallel processing to make the effort worthwhile.

The equation describing the effect on speedup of the balance between serial and parallel parts of a program is known as Amdahl's law [51, 62]. Let

$$p = \#\text{ of CPUs} \qquad T_1 = \text{1-CPU time} \qquad T_p = p\text{-CPU time} \tag{21.2}$$

The maximum speedup S_p attainable with parallel processing is thus

$$S_p^{\max} = \frac{T_1}{T_p} \to p \tag{21.3}$$

This limit is never met for a number of reasons: some of the program is serial, data and memory conflicts occur, communication and synchronization of the processors takes time, and it is rare to attain perfect load balance among all the processors. For the moment we ignore these complications and concentrate on how the *serial* part of the code affects the speedup. Let f to be the fraction of the program that potentially may run on multiple processors,

$$f = p\,\frac{T_p}{T_1} \qquad \text{(fraction parallel)} \tag{21.4}$$

The fraction $1 - f$ of the code that cannot be run in parallel must be run via serial processing, and thus takes time

$$T_s = (1 - f)T_1 \qquad \text{(serial time)} \tag{21.5}$$

The time T_p spent on the p parallel processors is related to T_s by

$$T_p = f\,\frac{T_1}{p} \tag{21.6}$$

That being the case, the speedup S_p as a function of f and the number of processors is

$$S_p \;=\; \frac{T_1}{T_s + T_p} \;=\; \frac{1}{1 - f + f/p} \qquad \text{(Amdahls's law)} \tag{21.7}$$

Some theoretical speedups are shown in Fig. 21.2 for different numbers p of processors. Clearly the speedup will not be significant enough to be worth the trouble unless most of the code is run in parallel (this is where we got the 90% figure from). Even an infinite number of processors cannot increase the speed of running the serial parts of the code, and so it runs at one processor speed. In practice, this means many problems are limited to a small number of processors, and that often for realistic applications, only 10–20% of the computer's peak performance may be obtained.

21.3.1 Communication Overhead

As discouraging as Amdahl's law may seem, it actually *overestimates* speedup because it ignores the *overhead* of parallel computation. Here we look at communication overhead. Assume a completely parallel code so that its speedup is

$$S_p = \frac{T_1}{T_p} = \frac{T_1}{T_1/p} = p \tag{21.8}$$

The denominator assumes that it takes no time for the processors to communicate. However, it take a finite time, called *latency*, to get data out of memory and into the cache or onto the communication network. When we add in this latency, as well as other times that make up the *communication time* T_c, the speedup decreases to

$$S_p \;\simeq\; \frac{T_1}{T_1/p + T_c} \;<\; p \qquad \text{(with communication time)} \tag{21.9}$$

For the speedup to be unaffected by communication time, we need to have

$$\frac{T_1}{p} \;\gg\; T_c \qquad \Rightarrow \qquad p \;\ll\; \frac{T_1}{T_c} \tag{21.10}$$

This means that as you keep increasing the number of processors p, at some point the time spent on computation T_1/p must equal the time T_c needed for communication, and adding more processors leads to greater execution time as the processors wait around more to communicate. This is another limit, then, on the maximum number of processors that may be used on any one problem, as well as on the effectiveness of increasing processor speed without a commensurate increase in communication speed.

The continual and dramatic increases in CPU speed, along with the widespread adoption of computer clusters, is leading to a changing view as to how to judge the speed of an algorithm. Specifically, CPUs are already so fast that the rate-determining step in a process is the slowest step, and that is often memory access or communication between processors. Such being the case, while the number of computational steps is still important for determining an algorithm's speed, the number and amount of memory access and interprocessor communication must also be mixed into the formula. This is currently an active area of research in algorithm development.