Chapter 1

Introduction

1.1 INTRODUCTION

The idea of a single-processor computer is fast becoming archaic and quaint. We now have to adjust our strategies when it comes to computing:

- It is impossible to improve computer performance using a single processor. Such processor would consume unacceptable power. It is more practical to use many simple processors to attain the desired performance using perhaps thousands of such simple computers [1].
- As a result of the above observation, if an application is not running fast on a single-processor machine, it will run even slower on new machines unless it takes advantage of parallel processing.
- Programming tools that can detect parallelism in a given algorithm have to be developed. An algorithm can show regular dependence among its variables or that dependence could be irregular. In either case, there is room for speeding up the algorithm execution provided that some subtasks can run concurrently while maintaining the correctness of execution can be assured.
- Optimizing future computer performance will hinge on good parallel programming at all levels: algorithms, program development, operating system, compiler, and hardware.
- The benefits of parallel computing need to take into consideration the number of processors being deployed as well as the communication overhead of processor-to-processor and processor-to-memory. Compute-bound problems are ones wherein potential speedup depends on the speed of execution of the algorithm by the processors. Communication-bound problems are ones wherein potential speedup depends on the speed of supplying the data to and extracting the data from the processors.
- Memory systems are still much slower than processors and their bandwidth is limited also to one word per read/write cycle.

Algorithms and Parallel Computing, by Fayez Gebali

- Scientists and engineers will no longer adapt their computing requirements to the available machines. Instead, there will be the practical possibility that they will adapt the computing hardware to solve their computing requirements.

This book is concerned with algorithms and the special-purpose hardware structures that execute them since software and hardware issues impact each other. Any software program ultimately runs and relies upon the underlying hardware support provided by the processor and the operating system. Therefore, we start this chapter with some definitions then move on to discuss some relevant design approaches and design constraints associated with this topic.

1.2 TOWARD AUTOMATING PARALLEL PROGRAMMING

We are all familiar with the process of algorithm implementation in software. When we write a code, we do not need to know the details of the target computer system since the compiler will take care of the details. However, we are steeped in thinking in terms of a single central processing unit (CPU) and sequential processing when we start writing the code or debugging the output. On the other hand, the processes of implementing algorithms in hardware or in software for parallel machines are more related than we might think. Figure 1.1 shows the main phases or layers of implementing an application in software or hardware using parallel computers. Starting at the top, *layer 5* is the application layer where the application or problem to be implemented on a parallel computing platform is defined. The specifications of inputs and outputs of the application being studied are also defined. Some input/output (I/O) specifications might be concerned with where data is stored and the desired timing relations of data. The results of this layer are fed to the lower layer to guide the algorithm development.

Layer 4 is algorithm development to implement the application in question. The computations required to implement the application define the tasks of the algorithm and their interdependences. The algorithm we develop for the application might or might not display parallelism at this state since we are traditionally used to linear execution of tasks. At this stage, we should not be concerned with task timing or task allocation to processors. It might be tempting to decide these issues, but this is counterproductive since it might preclude some potential parallelism. The result of this layer is a dependence graph, a directed graph (DG), or an adjacency matrix that summarize the task dependences.

Layer 3 is the parallelization layer where we attempt to extract latent parallelism in the algorithm. This layer accepts the algorithm description from layer 4 and produces thread timing and assignment to processors for software implementation. Alternatively, this layer produces task scheduling and assignment to processors for custom hardware very large-scale integration (VLSI) implementation. The book concentrates on this layer, which is shown within the gray rounded rectangle in the figure.

Layer 2 is the coding layer where the parallel algorithm is coded using a high-level language. The language used depends on the target parallel computing platform. The right branch in Fig. 1.1 is the case of mapping the algorithm on a general-purpose parallel computing platform. This option is really what we mean by *parallel programming*. Programming parallel computers is facilitated by what is called *concurrency platforms*, which are tools that help the programmer manage the threads and the timing of task execution on the processors. Examples of concurrency platforms include Cilk++, openMP, or compute unified device architecture (CUDA), as will be discussed in Chapter 6.

The left branch in Fig. 1.1 is the case of mapping the algorithm on a custom parallel computer such as systolic arrays. The programmer uses hardware description language (HDL) such as Verilog or very high-speed integrated circuit hardware (VHDL).

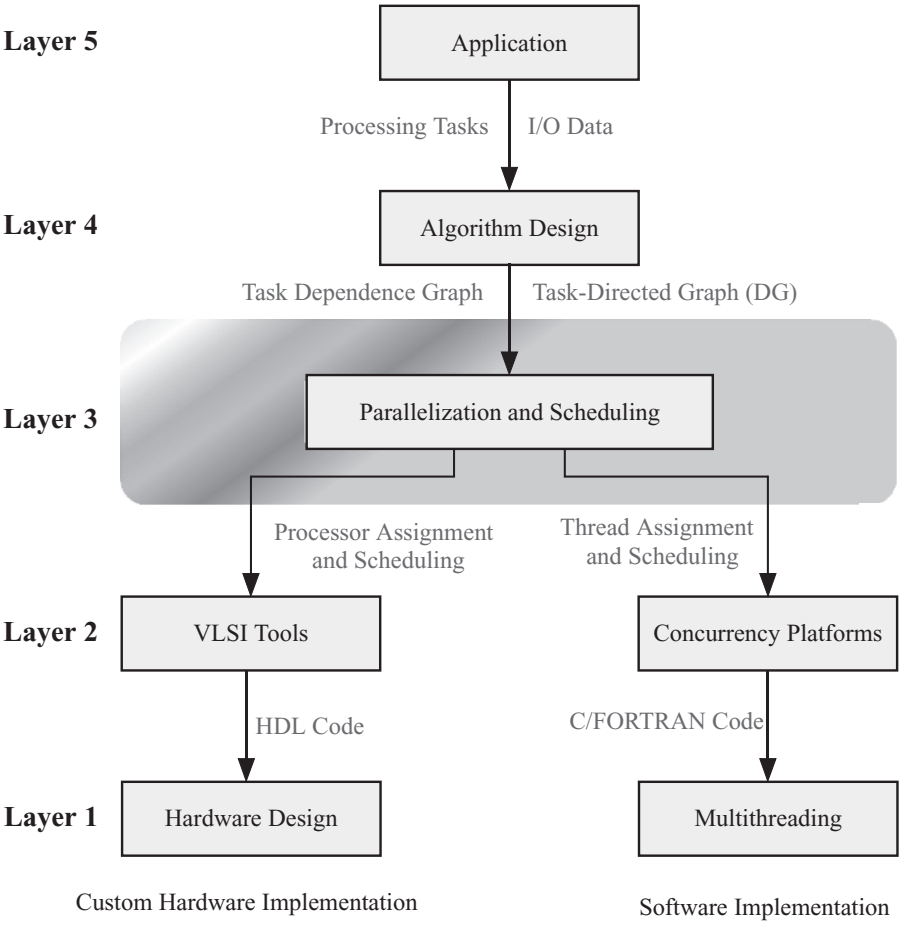

Figure 1.1 The phases or layers of implementing an application in software or hardware using parallel computers.

Layer 1 is the realization of the algorithm or the application on a parallel computer platform. The realization could be using multithreading on a parallel computer platform or it could be on an application-specific parallel processor system using application-specific integrated circuits (ASICs) or field-programmable gate array (FPGA).

So what do we mean by automatic programming of parallel computers? At the moment, we have automatic serial computer programming. The programmer writes a code in a high-level language such as C, Java, or FORTRAN, and the code is compiled without further input from the programmer. More significantly, the programmer does not need to know the hardware details of the computing platform. Fast code could result even if the programmer is unaware of the memory hierarchy, CPU details, and so on.

Does this apply to parallel computers? We have parallelizing compilers that look for simple loops and spread them among the processors. Such compilers could easily tackle what is termed *embarrassingly parallel algorithms* [2, 3]. Beyond that, the programmer must have intimate knowledge of how the processors interact among each and when the algorithm tasks are to be executed.

1.3 ALGORITHMS

The IEEE Standard Dictionary of Electrical and Electronics Terms defines an algorithm as "A prescribed set of well-defined rules or processes for the solution of a problem in a finite number of steps" [4]. The tasks or processes of an algorithm are interdependent in general. Some tasks can run concurrently in parallel and some must run serially or sequentially one after the other. According to the above definition, any algorithm is composed of a serial part and a parallel part. In fact, it is very hard to say that one algorithm is serial while the other is parallel except in extreme trivial cases. Later, we will be able to be more quantitative about this. If the number of tasks of the algorithm is W, then we say that the *work* associated with the algorithm is W.

The basic components defining an algorithm are

1. the different tasks,
2. the dependencies among the tasks where a task output is used as another task's input,
3. the set of primary inputs needed by the algorithm, and
4. the set of primary outputs produced by the algorithm.

1.3.1 Algorithm DG

Usually, an algorithm is graphically represented as a DG to illustrate the data dependencies among the algorithm tasks. We use the DG to describe our algorithm in preference to the term "dependence graph" to highlight the fact that the algorithm

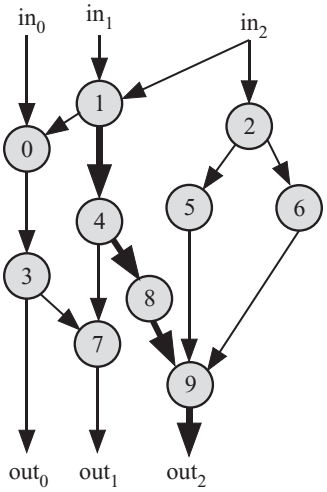

Figure 1.2 Example of a directed acyclic graph (DAG) for an algorithm.

variables flow as data between the tasks as indicated by the arrows of the DG. On the other hand, a dependence graph is a graph that has no arrows at its edges, and it becomes hard to figure out the data dependencies.

Definition 1.1 A dependence graph is a set of nodes and edges. The nodes represent the tasks to be done by the algorithm and the edges represent the data used by the tasks. This data could be input, output, or internal results.

Note that the edges in a dependence graph are undirected since an edge connecting two nodes does not indicate any input or output data dependency. An edge merely shows all the nodes that share a certain instance of the algorithm variable. This variable could be input, output, or I/O representing intermediate results.

Definition 1.2 A DG is a set of nodes and directed edges. The nodes represent the tasks to be done by the algorithm, and the directed edges represent the data dependencies among the tasks. The start of an edge is the output of a task and the end of an edge the input to the task.

Definition 1.3 A directed acyclic graph (DAG) is a DG that has no cycles or loops.

Figure 1.2 shows an example of representing an algorithm by a DAG. A DG or DAG has three types of edges depending on the sources and destinations of the edges.

Definition 1.4 An input edge in a DG is one that terminates on one or more nodes but does not start from any node. It represents one of the algorithm inputs.

Referring to Fig. 1.2, we note that the algorithm has three input edges that represent the inputs in_0, in_1, and in_2.

Definition 1.5 An output edge in a DG is one that starts from a node but does not terminate on any other node. It represents one of the algorithm outputs.

Referring to Fig. 1.2, we note that the algorithm has three output edges that represent the outputs out_0, out_1, and out_2.

Definition 1.6 An internal edge in a DG is one that starts from a node and terminate one or more nodes. It represents one of the algorithm internal variables.

Definition 1.7 An input node in a DG is one whose incoming edges are all input edges.

Referring to Fig. 1.2, we note that nodes 0, 1, and 2 represent input nodes. The tasks associated with these nodes can start immediately after the inputs are available.

Definition 1.8 An output node in a DG is whose outgoing edges are all output edges.

Referring to Fig. 1.2, we note that nodes 7 and 9 represent output nodes. Node 3 in the graph of Fig. 1.2 is not an output node since one of its outgoing edges is an internal edge terminating on node 7.

Definition 1.9 An internal node in a DG is one that has at least one incoming internal edge and at least one outgoing internal edge.

1.3.2 Algorithm Adjacency Matrix A

An algorithm could also be represented algebraically as an *adjacency matrix* **A**. Given W nodes/tasks, we define the 0–1 adjacency matrix **A**, which is a square $W \times W$ matrix defined so that element $a(i, j) = 1$ indicates that node i depends on the output from node j. The source node is j and the destination node is i. Of course, we must have $a(i, i) = 0$ for all values of $0 \leq i < W$ since node i does not depend on its own output (self-loop), and we assumed that we do not have any loops. The definition of the adjacency matrix above implies that this matrix is asymmetric. This is because if node i depends on node j, then the reverse is not true when loops are not allowed.

As an example, the adjacency matrix for the algorithm in Fig. 1.2 is given by

$$\mathbf{A} = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{0} & 0 & \mathbf{0} \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & \mathbf{0} & 0 & \mathbf{0} \\ 0 & 0 & 1 & 0 & 0 & 0 & \mathbf{0} & \mathbf{0} & 0 & \mathbf{0} \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & \mathbf{0} & 0 & \mathbf{0} \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & \mathbf{0} & 0 & \mathbf{0} \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & \mathbf{0} & 0 & \mathbf{0} \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & \mathbf{0} & 1 & \mathbf{0} \end{bmatrix}. \tag{1.1}$$

Matrix **A** has some interesting properties related to our topic. An input node i is associated with row i, whose elements are all zeros. An output node j is associated with column j, whose elements are all zeros. We can write

$$\text{Input node } i \Rightarrow \sum_{j=0}^{W-1} a(i, j) = 0 \tag{1.2}$$

$$\text{Output node } j \Rightarrow \sum_{i=0}^{W-1} a(i, j) = 0. \tag{1.3}$$

All other nodes are internal nodes. Note that all the elements in rows 0, 1, and 2 are all zeros since nodes 0, 1, and 2 are input nodes. This is indicated by the bold entries in these three rows. Note also that all elements in columns 7 and 9 are all zeros since nodes 7 and 9 are output nodes. This is indicated by the bold entries in these two columns. All other rows and columns have one or more nonzero elements to indicate internal nodes. If node i has element $a(i, j) = 1$, then we say that node j is a parent of node i.

1.3.3 Classifying Algorithms Based On Task Dependences

Algorithms can be broadly classified based on task dependences:

1. Serial algorithms
2. Parallel algorithms
3. Serial–parallel algorithms (SPAs)
4. Nonserial–parallel algorithms (NSPAs)
5. Regular iterative algorithms (RIAs)

The last category could be thought of as a generalization of SPAs. It should be mentioned that the level of data or task granularity can change the algorithm from one class to another. For example, adding two matrices could be an example of a serial algorithm if our basic operation is adding two matrix elements at a time. However, if we add corresponding rows on different computers, then we have a row-based parallel algorithm.

We should also mention that some algorithms can contain other types of algorithms within their tasks. The simple matrix addition example serves here as well. Our parallel matrix addition algorithm adds pairs of rows at the same time on different processors. However, each processor might add the rows one element at a time, and thus, the tasks of the parallel algorithm represent serial row add algorithms. We discuss these categories in the following subsections.

1.3.4 Serial Algorithms

A serial algorithm is one where the tasks must be performed in series one after the other due to their data dependencies. The DG associated with such an algorithm looks

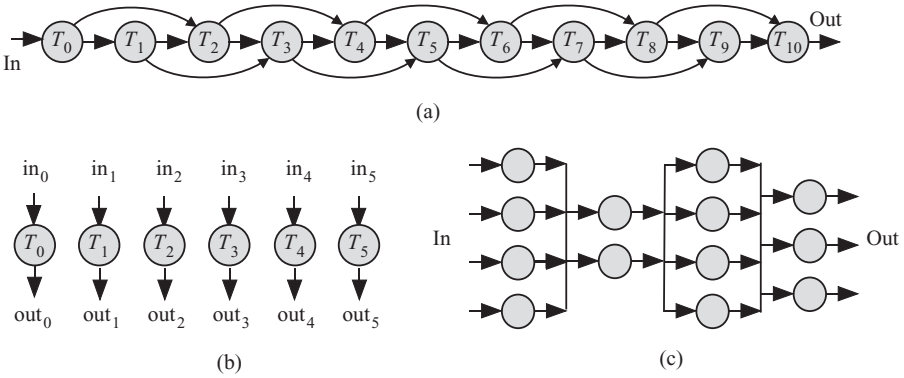

Figure 1.3 Example of serial, parallel, and serial–parallel algorithms. (a) Serial algorithm. (b) Parallel algorithm. (c) Serial–parallel algorithm.

like a long string or queue of dependent tasks. Figure 1.3a shows an example of a serial algorithm. The algorithm shown is for calculating Fibonnaci numbers. To calculate Fibonacci number n_{10}, task T_{10} performs the following simple calculation:

$$n_{10} = n_8 + n_9, \tag{1.4}$$

with $n_0 = 0$ and $n_1 = 1$ given as initial conditions. Clearly, we can find a Fibonacci number only after the preceding two Fibonacci numbers have been calculated.

1.3.5 Parallel Algorithms

A parallel algorithm is one where the tasks could all be performed in parallel at the same time due to their data independence. The DG associated with such an algorithm looks like a wide row of independent tasks. Figure 1.3b shows an example of a parallel algorithm. A simple example of such a purely parallel algorithm is a web server where each incoming request can be processed independently from other requests. Another simple example of parallel algorithms is multitasking in operating systems where the operating system deals with several applications like a web browser, a word processor, and so on.

1.3.6 SPAs

An SPA is one where tasks are grouped in stages such that the tasks in each stage can be executed concurrently in parallel and the stages are executed sequentially. An SPA becomes a parallel algorithm when the number of stages is one. A serial-parallel algorithm also becomes a serial algorithm when the number of tasks in each stage is one. Figure 1.3c shows an example of an SPA. An example of an SPA is the CORDIC algorithm [5–8]. The algorithm requires n iterations and at iteration i, three operations are performed:

$$\begin{aligned} x_{i+1} &= x_i + m y_i \delta_i \\ y_{i+1} &= y_i - x_i \delta_i \\ z_{i+1} &= z_i + \theta_i, \end{aligned} \tag{1.5}$$

where x, y, and z are the data to be updated at each iteration. δ_i and θ_i are iteration constants that are stored in lookup tables. The parameter m is a control parameter that determines the type of calculations required. The variable θ_i is determined before the start of each iteration. The algorithm performs other operations during each iteration, but we are not concerned about this here. More details can be found in Chapter 7 and in the cited references.

1.3.7 NSPAs

An NSPA does not conform to any of the above classifications. The DG for such an algorithm has no pattern. We can further classify NSPA into two main categories based on whether their DG contains cycles or not. Therefore, we can have two types of graphs for NSPA:

1. DAG
2. Directed cyclic graph (DCG)

Figure 1.4a is an example of a DAG algorithm and Fig. 1.4b is an example of a DCG algorithm. The DCG is most commonly encountered in discrete time feedback control systems. The input is supplied to task T_0 for prefiltering or input signal conditioning. Task T_1 accepts the conditioned input signal and the conditioned feedback output signal. The output of task T_1 is usually referred to as the error signal, and this signal is fed to task T_2 to produce the output signal.

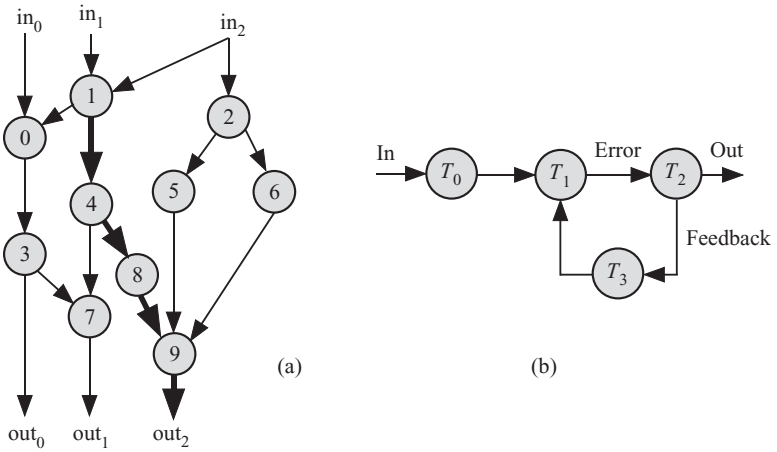

Figure 1.4 Example directed graphs for nonserial–parallel algorithms. (a) Directed acyclic graph (DAG). (b) Directed cyclic graph (DCG).

The NSPA graph is characterized by two types of constructs: the *nodes*, which describe the tasks comprising the algorithm, and the *directed edges*, which describe the direction of data flow among the tasks. The lines exiting a node represent an output, and when they enter a node, they represent an input. If task T_i produces an output that is used by task T_j, then we say that T_j depends on T_i. On the graph, we have an arrow from node i to node j.

The DG of an algorithm gives us three important properties:

1. *Work (*W*)*, which describes the amount of processing work to be done to complete the algorithm
2. *Depth (*D*)*, which is also known as the *critical path*. Depth is defined as the maximum path length between any input node and any output node.
3. *Parallelism (*P*)*, which is also known as the *degree of parallelism* of the algorithm. Parallelism is defined as the maximum number of nodes that can be processed in parallel. The maximum number of parallel processors that could be active at any given time will not exceed B since anymore processors will not find any tasks to execute.

A more detailed discussion of these properties and how an algorithm can be mapped onto a parallel computer is found in Chapter 8.

1.3.8 RIAs

Karp et al. [9, 10] introduced the concept of RIA. This class of algorithms deserves special attention because they are found in algorithms from diverse fields such as signal, image and video processing, linear algebra applications, and numerical simulation applications that can be implemented in grid structures. Figure 1.5 shows the *dependence graph* of a RIA. The example is for pattern matching algorithm. Notice that for a RIA, we do not draw a DAG; instead, we use the dependence graph concept.

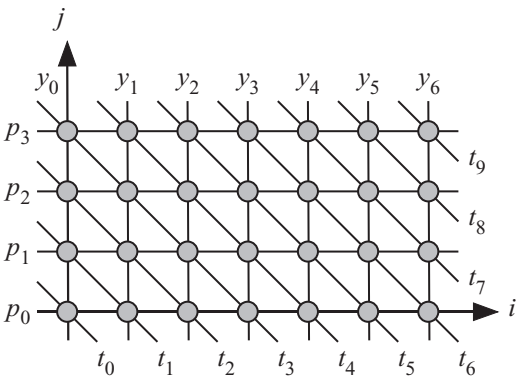

Figure 1.5 Dependence graph of a RIA for the pattern matching algorithm.

A dependence graph is like a DAG except that the links are not directed and the graph is obtained according to the methodology explained in Chapters 9, 10, and 11.

In a RIA, the dependencies among the tasks show a fixed pattern. It is a trivial problem to parallelize a serial algorithm, a parallel algorithm, or even an SPA. It is not trivial to explore the possible parallelization options of a RIA. In fact, Chapters 9–11 are dedicated to just exploring the parallelization of this class of algorithms.

A simple example of a RIA is the matrix–matrix multiplication algorithm given by Algorithm 1.1.

Algorithm 1.1 Matrix–matrix multiplication algorithm.

Require: Input: matrices A and B

for $i = 0 : I - 1$ **do**
  for $j = 0 : J - 1$ **do**
    $temp = 0$
    for $k = 0 : K - 1$ **do**
      $temp = \text{temp} + A(i, k) \times B(k, j)$
    end for
    $C(i, j) = temp$
  end for
end for
RETURN C

The variables in the RIA described by Algorithm 1.1 show regular dependence on the algorithm indices i, j, and k. Traditionally, such algorithms are studied using the dependence graph technique, which shows the links between the different tasks to be performed [10–12]. The dependence graph is attractive when the number of algorithm indices is 1 or 2. We have three indices in our matrix–matrix multiplication algorithm. It would be hard to visualize such an algorithm using a three-dimensional (3-D) graph. For higher dimensionality algorithms, we use more formal techniques as will be discussed in this book. Chapters 9–11 are dedicated to studying such algorithms.

1.3.9 Implementing Algorithms on Parallel Computing

The previous subsections explained different classes of algorithms based on the dependences among the algorithm tasks. We ask in this section how to implement these different algorithms on parallel computing platforms either in hardware or in software. This is referred to as parallelizing an algorithm. The parallelization strategy depends on the type of algorithm we are dealing with.

Serial Algorithms

Serial algorithms, as exemplified by Fig. 1.3a, cannot be parallelized since the tasks must be executed sequentially. The only parallelization possible is when each task is broken down into parallelizable subtasks. An example is to perform bit-parallel add/multiply operations.

Parallel Algorithms

Parallel algorithms, as exemplified by Fig. 1.3b, are easily parallelized since all the tasks can be executed in parallel, provided there are enough computing resources.

SPAs

SPAs, as exemplified by Fig. 1.3c, are parallelized by assigning each task in a stage to a software thread or hardware processing element. The stages themselves cannot be parallelized since they are serial in nature.

NSPAs

Techniques for parallelizing NSPAs will be discussed in Chapter 8.

RIAs

Techniques for parallelizing RIAs will be discussed in Chapters 9–11.

1.4 PARALLEL COMPUTING DESIGN CONSIDERATIONS

This section discusses some of the important aspects of the design of parallel computing systems. The design of a parallel computing system requires considering many design options. The designer must choose a basic *processor architecture* that is capable of performing the contemplated tasks. The processor could be a simple element or it could involve a superscalar processor running a multithreaded operating system.

The processors must communicate among themselves using some form of an *interconnection network*. This network might prove to be a bottleneck if it cannot support simultaneous communication between arbitrary pairs of processors. Providing the links between processors is like providing physical channels in telecommunications. How data are exchanged must be specified. A bus is the simplest form of interconnection network. Data are exchanged in the form of words, and a system clock informs the processors when data are valid. Nowadays, buses are being replaced by *networks-on-chips* (NoC) [13]. In this architecture, data are exchanged on the chip in the form of *packets* and are routed among the chip modules using *routers*.

Data and programs must be stored in some form of *memory system*, and the designer will then have the option of having several memory modules shared among

the processors or of dedicating a memory module to each processor. When processors need to share data, mechanisms have to be devised to allow reading and writing data in the different memory modules. The order of reading and writing will be important to ensure data integrity. When a shared data item is updated by one processor, all other processors must be somehow informed of the change so they use the appropriate data value.

Implementing the tasks or programs on a parallel computer involves several design options also. *Task partitioning* breaks up the original program or application into several segments to be allocated to the processors. The level of partitioning determines the workload allocated to each processor. *Coarse grain partitioning* allocates large segments to each processor. Fine grain partitioning allocates smaller segments to each processor. These segments could be in the form of separate *software processes* or *threads*. The programmer or the compiler might be the two entities that decide on this partitioning. The programmer or the operating system must ensure proper *synchronization* among the executing tasks so as to ensure program correctness and data integrity.

1.5 PARALLEL ALGORITHMS AND PARALLEL ARCHITECTURES

Parallel algorithms and parallel architectures are closely tied together. We cannot think of a parallel algorithm without thinking of the parallel hardware that will support it. Conversely, we cannot think of parallel hardware without thinking of the parallel software that will drive it. Parallelism can be implemented at different levels in a computing system using hardware and software techniques:

1. *Data-level parallelism*, where we simultaneously operate on multiple bits of a datum or on multiple data. Examples of this are bit-parallel addition multiplication and division of binary numbers, vector processor arrays and systolic arrays for dealing with several data samples. This is the subject of this book.
2. *Instruction-level parallelism (ILP)*, where we simultaneously execute more than one instruction by the processor. An example of this is use of instruction pipelining.
3. *Thread-level parallelism (TLP).* A thread is a portion of a program that shares processor resources with other threads. A thread is sometimes called a lightweight process. In TLP, multiple software threads are executed simultaneously on one processor or on several processors.
4. *Process-level parallelism.* A process is a program that is running on the computer. A process reserves its own computer resources such as memory space and registers. This is, of course, the classic multitasking and timesharing computing where several programs are running simultaneously on one machine or on several machines.

1.6 RELATING PARALLEL ALGORITHM AND PARALLEL ARCHITECTURE

The IEEE Standard Dictionary of Electrical and Electronics Terms [4] defines "parallel" for software as "simultaneous transfer, occurrence, or processing of the individual parts of a whole, such as the bits of a character and the characters of a word using separate facilities for the various parts." So in that sense, we say an algorithm is parallel when two or more parts of the algorithms can be executed independently on hardware. Thus, the definition of a parallel algorithm presupposes availability of supporting hardware. This gives a hint that parallelism in software is closely tied to the hardware that will be executing the software code. Execution of the parts can be done using different threads or processes in the software or on different processors in the hardware. We can quickly identify a potentially parallel algorithm when we see the occurrence of "FOR" or "WHILE" loops in the code.

On the other hand, the definition of parallel architecture, according to *The IEEE Standard Dictionary of Electrical and Electronics Terms* [4], is "a multi-processor architecture in which parallel processing can be performed." It is the job of the programmer, compiler, or operating system to supply the multiprocessor with tasks to keep the processors busy. We find ready examples of parallel algorithms in fields such as

- scientific computing, such as physical simulations, differential equations solvers, wind tunnel simulations, and weather simulation;
- computer graphics, such as image processing, video compression; and ray tracing; and,
- medical imaging, such as in magnetic resonance imaging (MRI) and computerized tomography (CT).

There are, however, equally large numbers of algorithms that are not recognizably parallel especially in the area of information technology such as online medical data, online banking, data mining, data warehousing, and database retrieval systems. The challenge is to develop computer architectures and software to speed up the different information technology applications.

1.7 IMPLEMENTATION OF ALGORITHMS: A TWO-SIDED PROBLEM

Figure 1.6 shows the issues we would like to deal with in this book. On the left is the space of algorithms and on the right is the space of parallel architectures that will execute the algorithms. Route A represents the case when we are given an algorithm and we are exploring possible parallel hardware or processor arrays that would correctly implement the algorithm according to some performance requirements and certain system constraints. In other words, the problem is given a parallel algorithm, what are the possible parallel processor architectures that are possible?

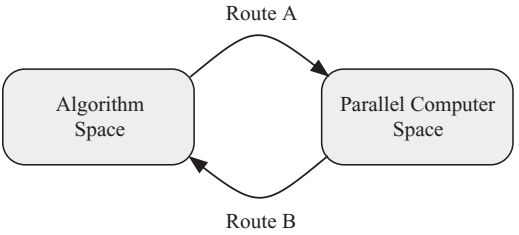

Figure 1.6 The two paths relating parallel algorithms and parallel architectures.

Route B represents the classic case when we are given a parallel architecture or a multicore system and we explore the best way to implement a given algorithm on the system subject again to some performance requirements and certain system constraints. In other words, the problem is given a parallel architecture, how can we allocate the different tasks of the parallel algorithm to the different processors? This is the realm of parallel programming using the multithreading design technique. It is done by the application programmer, the software compiler, and the operating system.

Moving along routes A or B requires dealing with

1. *mapping* the tasks to different processors,
2. *scheduling* the execution of the tasks to conform to algorithm data dependency and data I/O requirements, and
3. *identifying* the data communication between the processors and the I/O.

1.8 MEASURING BENEFITS OF PARALLEL COMPUTING

We review in this section some of the important results and benefits of using parallel computing. But first, we identify some of the key parameters that we will be studying in this section.

1.8.1 Speedup Factor

The potential benefit of parallel computing is typically measured by the time it takes to complete a task on a single processor versus the time it takes to complete the same task on N parallel processors. The speedup $S(N)$ due to the use of N parallel processors is defined by

$$S(N) = \frac{T_p(1)}{T_p(N)}, \tag{1.6}$$

where $T_p(1)$ is the algorithm processing time on a single processor and $T_p(N)$ is the processing time on the parallel processors. In an ideal situation, for a fully

parallelizable algorithm, and when the communication time between processors and memory is neglected, we have $T_p(N) = T_p(1)/N$, and the above equation gives

$$S(N) = N. \tag{1.7}$$

It is rare indeed to get this linear increase in computation domain due to several factors, as we shall see in the book.

1.8.2 Communication Overhead

For single and parallel computing systems, there is always the need to read data from memory and to write back the results of the computations. Communication with the memory takes time due to the speed mismatch between the processor and the memory [14]. Moreover, for parallel computing systems, there is the need for communication between the processors to exchange data. Such exchange of data involves transferring data or messages across the interconnection network.

Communication between processors is fraught with several problems:

1. *Interconnection network delay.* Transmitting data across the interconnection network suffers from bit propagation delay, message/data transmission delay, and queuing delay within the network. These factors depend on the network topology, the size of the data being sent, the speed of operation of the network, and so on.
2. *Memory bandwidth.* No matter how large the memory capacity is, access to memory contents is done using a single port that moves one word in or out of the memory at any give memory access cycle.
3. *Memory collisions*, where two or more processors attempt to access the same memory module. Arbitration must be provided to allow one processor to access the memory at any given time.
4. *Memory wall.* The speed of data transfer to and from the memory is much slower than processing speed. This problem is being solved using memory hierarchy such as

   register ↔ cache ↔ RAM ↔ electronic disk ↔ magnetic disk ↔ optic disk

To process an algorithm on a parallel processor system, we have several delays as explained in Table 1.1.

1.8.3 Estimating Speedup Factor and Communication Overhead

Let us assume we have a parallel algorithm consisting of N independent tasks that can be executed either on a single processor or on N processors. Under these ideal circumstances, data travel between the processors and the memory, and there is no

Table 1.1 Delays Involved in Evaluating an Algorithm on a Parallel Processor System

Operation	Symbol	Comment
Memory read	$T_r(N)$	Read data from memory shared by N processors
Memory write	$T_w(N)$	Write data from memory shared by N processors
Communicate	$T_c(N)$	Communication delay between a pair of processors when there are N processors in the system
Process data	$T_p(N)$	Delay to process the algorithm using N parallel processors

interprocessor communication due to the task independence. We can write under ideal circumstances

$$T_p(1) = N\tau_p \tag{1.8}$$

$$T_p(N) = \tau_p. \tag{1.9}$$

The time needed to read the algorithm input data by a single processor is given by

$$T_r(1) = N\tau_m, \tag{1.10}$$

where τ_m is memory access time to read one block of data. We assumed in the above equation that each task requires one block of input data and N tasks require to read N blocks. The time needed by the parallel processors to read data from memory is estimated as

$$T_r(N) = \alpha T_r(1) = \alpha N\tau_m, \tag{1.11}$$

where α is a factor that takes into account limitations of accessing the shared memory. $\alpha = 1/N$ when each processor maintains its own copy of the required data. $\alpha = 1$ when data are distributed to each task in order from a central memory. In the worst case, we could have $\alpha > N$ when all processors request data and collide with each other. We could write the above observations as

$$T_r(N)\begin{cases} = & \tau_m & \text{when} & \text{Distributed memory} \\ = & N\tau_m & \text{when} & \text{Shared memory and no collisions} \\ > & N\tau_m & \text{when} & \text{Shared memory with collisions.} \end{cases} \tag{1.12}$$

Writing back the results to the memory, also, might involve memory collisions when the processor attempts to access the same memory module.

$$T_w(1) = N\tau_m \tag{1.13}$$

$$T_w(N) = \alpha T_w(1) = \alpha N\tau_m. \tag{1.14}$$

For a single processor, the total time to complete a task, including memory access overhead, is given by

$$\begin{aligned} T_{\text{total}}(1) &= T_r(1) + T_p(1) + T_w(1) \\ &= N\left(2\tau_m + \tau_p\right) \end{aligned}. \tag{1.15}$$

Now let us consider the speedup factor when communication overhead is considered:

$$\begin{aligned} T_{\text{total}}(N) &= T_r(N) + T_p(N) + T_w(N) \\ &= 2N\alpha\tau_m + \tau_p \end{aligned}. \tag{1.16}$$

The speedup factor is given by

$$\begin{aligned} S(N) &= \frac{T_{\text{total}}(1)}{T_{\text{total}}(N)} \\ &= \frac{2\alpha N\tau_m + N\tau_p}{2N\alpha\tau_m + \tau_p}. \end{aligned} \tag{1.17}$$

Define the *memory mismatch ratio* (R) as

$$R = \frac{\tau_m}{\tau_p}, \tag{1.18}$$

which is the ratio of the delay for accessing one data block from the memory relative to the delay for processing one block of data. In that sense, τ_p is expected to be orders of magnitude smaller than τ_m depending on the granularity of the subtask being processed and the speed of the memory.

We can write Eq. 1.17 as a function of N and R in the form

$$S(N, R) = \frac{2\alpha RN + N}{2\alpha RN + 1}. \tag{1.19}$$

Figure 1.7 shows the effect of the two parameters, N and R, on the speedup when $\alpha = 1$. Numerical simulations indicated that changes in α are not as significant as the values of R and N. From the above equation, we get full speedup when the product $RN \ll 1$. This speedup is similar to Eq. 1.7 where communication overhead was neglected.

This situation occurs in the case of trivially parallel algorithms as will be discussed in Chapter 7.

Notice from the figure that speedup quickly decreases when $RN > 0.1$. When $R = 1$, we get a communication-bound problem and the benefits of parallelism quickly vanish. This reinforces the point that memory design and communication between processors or threads are very important factors. We will also see that multicore processors, discussed in Chapter 3, contain all the processors on the same chip. This has the advantage that communication occurs at a much higher speed compared with multiprocessors, where communication takes place across chips. Therefore, T_m is reduced by orders of magnitude for multicore systems, and this should give them the added advantage of small R values.

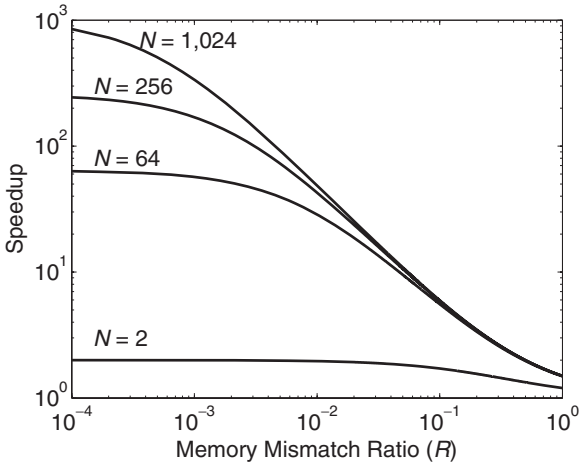

Figure 1.7 Effect of the two parameters, N and R, on the speedup when $\alpha = 1$.

The interprocessor communication overhead involves reading and writing data into memory:

$$T_c(N) = \beta N \tau_m, \tag{1.20}$$

where $\beta \geq 0$ and depends on the algorithm and how the memory is organized. $\beta = 0$ for a single processor, where there is no data exchange or when the processors in a multiprocessor system do not communicate while evaluating the algorithm. In other algorithms, β could be equal to $\log_2 N$ or even N. This could be the case when the parallel algorithm programmer or hardware designer did not consider fully the cost of interprocessor or interthread communications.

1.9 AMDAHL'S LAW FOR MULTIPROCESSOR SYSTEMS

Assume an algorithm or a task is composed of parallizable fraction f and a serial fraction $1 - f$. Assume the time needed to process this task on one single processor is given by

$$T_p(1) = N(1-f)\tau_p + Nf\tau_p = N\tau_p, \tag{1.21}$$

where the first term on the right-hand side (RHS) is the time the processor needs to process the serial part. The second term on RHS is the time the processor needs to process the parallel part. When this task is executed on N parallel processors, the time taken will be given by

$$T_p(N) = N(1-f)\tau_p + f\tau_p, \tag{1.22}$$

where the only speedup is because the parallel part now is distributed over N processors. Amdahl's law for speedup $S(N)$, achieved by using N processors, is given by

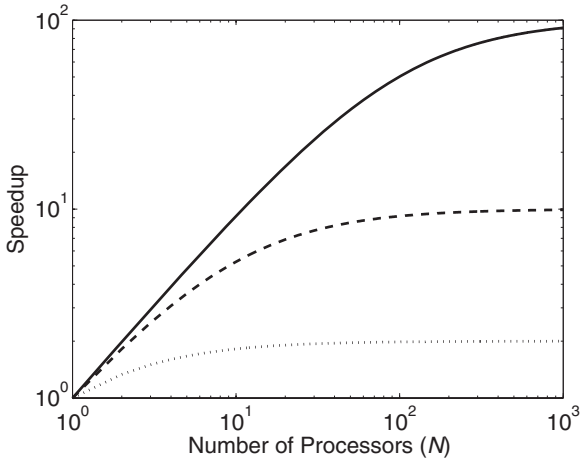

Figure 1.8 Speedup according to Amdahl's law. The solid line is for $f = 0.99$; the dashed line is for $f = 0.9$; and the dotted line is for $f = 0.5$.

$$\begin{aligned} S(N) &= \frac{T_p(1)}{T_p(N)} \\ &= \frac{N}{(1-f)N + f} \\ &= \frac{1}{(1-f) + f/N}. \end{aligned} \tag{1.23}$$

To get any speedup, we must have

$$1 - f << f/N. \tag{1.24}$$

This inequality dictates that the parallel portion f must be very close to unity especially when N is large.

Figure 1.8 shows the speedup versus f for different values of N. The solid line is for $f = 0.99$; the dashed line is for $f = 0.9$; and the dotted line is for $f = 0.5$. We note from the figure that speedup is affected by the value of f. As expected, larger f results in more speedup. However, note that the speedup is most pronounced when $f > 0.5$. Another observation is that speedup saturates to a given value when N becomes large.

For large values of N, the speedup in Eq. 1.23 is approximated by

$$S(N) \approx \frac{1}{1-f} \qquad \text{when } N \gg 1. \tag{1.25}$$

This result indicates that if we are using a system with more than 10 processors, then any speedup advantage is dictated mainly by how clever we are at discovering the parallel parts of the program and how much we are able to execute those parallel parts simultaneously. The figure confirms these expectations.

For extreme values of f, Eq. 1.23 becomes

$$S(N) = 1 \quad \text{when } f = 0 \quad \text{completely serial code} \tag{1.26}$$

$$S(N) = N \quad \text{when } f = 1 \quad \text{completely parallel code.} \tag{1.27}$$

The above equation is obvious. When the program is fully parallel, speedup will be equal to the number of parallel processors we use.

What do we conclude from this? Well, we must know or estimate the value of the fraction f for a given algorithm at the start. Knowing f will give us an idea on what system speedup could be expected on a multiprocessor system. This alone should enable us to judge how much effort to spend trying to improve speedup by mapping the algorithm to a multiprocessor system.

1.10 GUSTAFSON–BARSIS'S LAW

The predictions of speedup according to Amdahl's law are pessimistic. Gustafson [15] made the observation that parallelism increases in an application when the problem size increases. Remember that Amdahl's law assumed that the fraction of parallelizable code is fixed and does not depend on problem size.

To derive Gustafson–Barsis formula for speedup, we start with the N parallel processors first. The time taken to process the task on N processors is given by

$$T_p(N) = (1-f)\tau_p + f\tau_p = \tau_p. \tag{1.28}$$

When this task is executed on a single processor, the serial part is unchanged, but the parallel part will increase as given by

$$T_p(1) = (1-f)\tau_p + Nf\tau_p. \tag{1.29}$$

The speedup is given now by

$$\begin{aligned} S(N) &= \frac{T_p(1)}{T_p(N)} \\ &= (1-f) + Nf \\ &= 1 + (N-1)f. \end{aligned} \tag{1.30}$$

Figure 1.9 shows the speedup versus f for different values of N. The solid line is for $f = 0.99$; the dashed line is for $f = 0.9$; and the dotted line is for $f = 0.5$. Notice that there is speedup even for very small values of f and the situation improves as N gets larger.

To get any speedup, we must have

$$f(N-1) \gg 1. \tag{1.31}$$

Notice that we can get very decent speedup even for small values of f especially when N gets large. Compared with inequality 1.24, we note that the speedup constraints are very much relaxed according to Gustafson–Barsis's law.

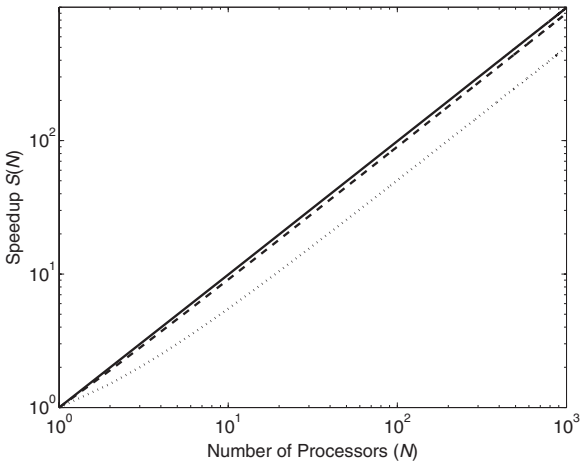

Figure 1.9 Speedup according to Gustafson–Barsis's law. The solid line is for $f = 0.99$; the dashed line is for $f = 0.9$; and the dotted line is for $f = 0.5$.

1.11 APPLICATIONS OF PARALLEL COMPUTING

The availability of inexpensive yet really powerful parallel computers is expected to make a hitherto unforeseeable impact on our lives. We are used now to parallel computers helping us access any information through web search engines. In fact, the search progresses as we are typing our search key words. However, there is room for improvement and, more importantly, for innovation, as the following sections illustrate.

1.11.1 Climate Modeling

Climate simulations are used for weather forecasting as well as for predicting global climate changes based on different phenomena or human activities. As Reference 1 points out, the resolution of today's climate models is 200 km. This is considered low resolution given the fact that some climate systems exist completely within such resolution scale.

Assume a high-resolution model for climate simulation partitions the globe using 3-D cells 1 km in size in each direction. Assume also that the total surface of the earth to be $510 \times 10^6 \text{km}^2$ and the thickness of the atmospheric layer to be approximately 1,000 km. Then, we need to simulate approximately 5×10^{11} weather cells. Assume further that each cell needs to do 200 floating point operations for each iteration of the simulation. Thus, we have to perform a total of 10^{14} floating point operations per iteration.

Let us now assume that we need to run the simulation 10^6 times to simulate the climate over some long duration of the weather cycle. Thus, we have the following performance requirements for our computing system:

Table 1.2 Parallel Multicore Computer Implementation Using Two Types of Microprocessors Needed to Perform 2.8×10^{15} FLOPS

Processor	Clock speed	GFLOPS/core	Cores needed	Power (MW)
AMD Opteron	2.8 GHz	5.6	4.9×10^5	52.0
Tensilica XTensa LX2	500.0 MHz	1.0	2.8×10^6	0.8

$$\begin{aligned}\text{Total number of operations} &= 10^{14} \text{ operations/iteration} \times 10^6 \text{ iterations} \\ &= 10^{20} \text{ floating point operations}\end{aligned} \quad (1.32)$$

A computer operating at a rate of 10^9 floating point operations per second (FLOPS) would complete the operations in 10^{11} seconds, which comes to about 31 centuries. Assuming that all these simulations should be completed in one workday, then our system should operate at a rate of approximately 2.8×10^{15} FLOPS. It is obvious that such performance cannot be attained by any single-processor computer. We must divide this computational task among many processors. Modeling the atmosphere using a mesh or a grid of nodes lends itself to computational parallelization since calculations performed by each node depend only on its immediate six neighboring nodes. Distributing the calculations among several processors is relatively simple, but care must be given to the exchange of data among the processors. Table 1.2 compares building a parallel processor system needed to give us a performance of 2.8×10^{15} FLOPS. We assume using desktop microprocessors versus using a simple embedded microprocessor [1].

The power advantage of using low-power, low-performance processors is obvious from the table. Of course, we need to figure out how to interconnect such a huge system irrespective of the type of processor used. The interconnection network becomes a major design issue here since it would be impossible to think of a system that uses buses and single global system clock.

1.11.2 CT

CT and magnetic resonance imaging (MRI) are techniques to obtain a high-resolution map of the internals of the body for medical diagnosis. Figure 1.10 shows a simplified view of a CT system. Figure 1.10a shows the placement of the patient on a gurney at the center of a very strong magnet and a strong X-ray source. The gurney is on a movable table in a direction perpendicular to the page. The X-ray source or emitter is placed at the top and emits a collimated beam that travels to the other side of the circle through the patient. An X-ray detector is placed diametrically opposite to where the X-ray source is. When the machine is in operation, the source/detector pair is rotated as shown in Fig. 1.10b. After completing a complete rotation and storing the detector samples, the table is moved and the process is repeated for a different section or slice of the body. The output of a certain detector at a given time

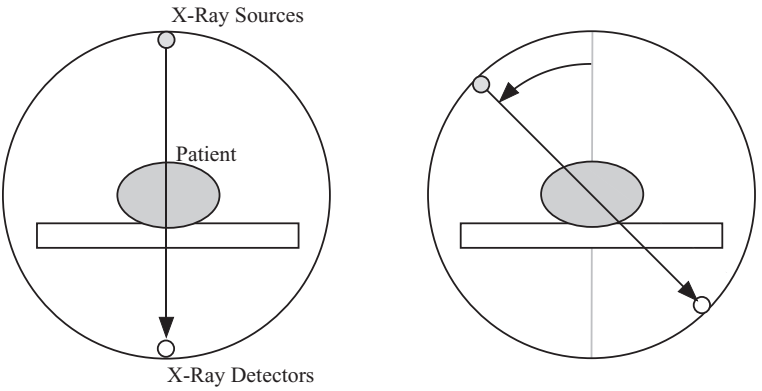

Figure 1.10 Computerized tomography (CT) system. (a) Setup of X-ray sources and detectors. (b) Schematic of the output of each sensor when a single X-ray source is active.

is affected by all the patient tissue that a certain X-ray beam encounters in its passage from the source to the detector. As things stand at the time of writing, the patient needs to be in this position for several minutes if not hours (personal experience).

Assume the image we are trying to generate is composed of $N \times N$ pixels, where N could be approximately equal to 4,000. Thus, we have approximately 10^7 pixels to generate per image, or slice, of the body scan. As the table moves, more slices should be generated. This allows for 3-D viewing of the body area of concern. For a system that generates $S = 1{,}000$ successive slices, $SN^2 = 10^{10}$ pixels will have to be processed. A slice will require approximately $N^2 (\log_2 N)^3$ calculations [16]. For our case, we need approximately

$$\begin{aligned}\text{Total number of operations} &= 10^{10} \text{ operations/slice} \times 10^3 \text{ slices} \\ &= 10^{13} \text{ floating point operations}\end{aligned} \tag{1.33}$$

Assume we need to generate these images in 1 second to allow for a real-time examination of the patient. In that case, the system should operate at a rate of approximately 10^{13} FLOPS. For an even more accurate medical diagnosis, high-resolution computerized tomography (HRCT) scans are required even at the nanoscale level where blood vessels need to be examined. Needless to say, parallel processing of massive data will be required for a timely patient treatment.

1.11.3 Computational Fluid Dynamics (CFD)

CFD is a field that is closely tied to parallel computers and parallel algorithms. It is viewed as a cost-effective way to investigate and design systems that involve flow of gas or fluids. Some examples of CFD are:

- ocean currents,
- our atmosphere and global weather,

- blood flow in the arteries,
- heart deformation during high-G maneuvers of a fighter jet,
- air flow in the lungs,
- design of airplane wings and winglets,
- seat ejection in a fighter jet,
- combustion of gases inside a car cylinder,
- jet engine air intake and combustion chamber,
- shape of a car body to reduce air drag, and
- spray from nozzles such as paint guns and rocket exhaust.

Typically, the region where the flow of interest is being studied is divided into a grid or mesh of points using the *finite element* method. The number of grid points depends on the size of the region or the desired resolution. A system of linear equations or a set differential equations is solved at each grid point for the problem unknowns. The number of unknown might be around 10^3, and each variable might require around 10^3 floating point operations at each grid point.

The targeted region of the CFD applications ranges from 10^{12} to 10^{18} FLOPS [17]. If the computer system operates at a speed of 10^9 (giga) FLOPS, then CFD applications would complete a simulation in the time period that ranges between 15 minutes and 30 years. On the other hand, a parallel computer system operating at 10^{12} (tera) FLOPS would complete the application in a time period between 1 second and 12 days. Currently, there are few supercomputer systems that operate at the rate of 10^{15} (peta) FLOPS. On such a system, the larger problem would take about 3 minutes to complete.

1.12 PROBLEMS

1.1. Assume you are given the task of adding eight numbers together. Draw the DG and the adjacency matrix for each of the following number adding algorithms:

(1) Add the numbers serially, which would take seven steps.

(2) Add the numbers in a binary fashion by adding each adjacent pair of numbers in parallel and then by adding pairs of the results in parallel, and continue this process.

1.2. Derive general expressions for the number of tasks required to do the number adding algorithms in Problem 1.1 when we have $N = 2^n$ numbers to be added. What conclusion do you make?

1.3. Now assume that you have a parallel computer that can add the numbers in Problem 1.1. The time required to add a pair of numbers is assumed 1. What would be the time required to perform the two algoritnms for the case $N = 2^n$? How much is the speedup?

1.4. Consider Problem 1.3. Now the parallel computers require a time C to obtain data from memory and to communicate the add results between the add stages. How much speedup is accomplished?

1.5. Which class of algorithms would the fast Fourier transform (FFT) algorithm belong to?

1.6. Which class of algorithms would the quicksort algorithm belong to?

1.7. The binary number multiplication problem in Chapter 2 could be considered as a RIA algorithm. Draw the dependence graph of such an algorithm.

1.8. The binary restoring division algorithm is based on the recurrence equation

$$r_{j+1} = 2r_j - q_{n-j-1}\, D \geq j < n,$$

where r_j is the partial remainder at the jth iteration; q_k is the kth quotient bit; and D is the denominator. It is assumed that the number of bits in the quotient is n and q_{n-1} is the quotient most significant bit (MSB). What type of algorithm is this division algorithm?

1.9. A processor has clock frequency f, and it requires c clock cycles to execute a single instruction. Assume a program contains I instructions. How long will the program take before it completes?

1.10. Repeat Problem 1.9 when a new processor is introduced whose clock frequency is $f' = 2f$ and $c' = 1.5c$.

1.11. Give some examples of serial algorithms.

1.12. Give some examples of parallel algorithms.

1.13. Consider the speedup factor for a fully parallel algorithm when communication overhead is assumed. Comment on speedup for possible values of α.

1.14. Consider the speedup factor for a fully parallel algorithm when communication overhead is assumed. Comment on speedup for possible values of R.

1.15. Write down the speedup formula when communication overhead is included and the algorithm requires interprocessor communications Assume that each task in the parallel algorithm requires communication between a pair of processors. Assume that the processors need to communicate with each other m times to complete the algorithm.

1.16. Consider an SPA with the following specifications:

Number of serial tasks per stage	N_s
Number of serial tasks per stage	N_p
Number of stages	n

Now assume that we have a single processor that requires τ to complete a task and it consumes W watts while in operation. We are also given $N = N_p$ parallel but very slow processors. Each processor requires $r\tau$ to complete a task and consumes W/r watts while in operation, where $r > 1$ is a performance derating factor.

(1) How long will the single processor need to finish the algorithm?

(2) How much energy will the single processor consume to finish the algorithm?

(3) How long will the multiprocessor need to finish the algorithm?

(4) How much energy will the multiprocessor system consume to finish the algorithm?

(5) Write down a formula for the speedup.

(6) Write down a formula for the energy ratio of the multiprocessor relative to the single processor.

1.17. The algorithm for floating point addition can be summarized as follows:

(1) Compare the exponents and choose the larger exponent.

(2) Right shift the mantissa of the number with the smaller exponent by the amount of exponent difference.

(3) Add the mantissas.

(4) Normalize the results.

Draw a dependence graph of the algorithm and state what type of algorithm this is.

1.18. The algorithm for floating point multiplication can be summarized as follows:

(1) Multiply the mantissas.

(2) Add the two exponents.

(3) Round the multiplication result.

(4) Normalize the result.

Draw a dependence graph of the algorithm and state what type of algorithm this is.

1.19. Discuss the algorithm for synthetic apperture radar (SAR).

1.20. Discuss the Radon transform algorithm in two dimensions.