CHAPTER 1

Introduction

The need for load scheduling/sharing arises in many real-world situations. Common examples include the scheduling of processing loads in distributed computing systems, the scheduling of production loads in manufacturing systems, and the scheduling of available resources among many users. By and large, a scheduling problem addresses the following question: *What is the best possible way to organize a given work load so that it can be completed in the shortest possible time?* In a distributed computing system, processing loads arrive from many users at random time instants. A proper scheduling policy attempts to assign these loads to available processors so as to complete the processing of all loads in the shortest possible time.

In recent years, with the advent of sophisticated and complex parallel and distributed computing systems, research in the area of load scheduling has gained considerable momentum. This book is a contribution in one particular area in the broad field of load sharing in parallel and distributed computing systems. Specifically, it addresses issues associated with the distribution of *arbitrarily divisible loads* among processors in a distributed computing system subject to communication delays. Here we shall assume the words *sharing*, *scheduling*, and *distribution* to be synonymous, although in a strict sense this need not be true, and use them interchangeably throughout the book.

1.1 MULTIPROCESSOR AND MULTICOMPUTER SYSTEMS

Both multiprocessor and multicomputer systems are characterized by the presence of several processors in the system. Multiprocessor systems are usually considered

as computer systems that have several processors, but are serviced by a single set of peripherals. Hence, these processors are not autonomous and such systems are also known as *parallel processing systems* (PPS). In contrast, multicomputer systems consist of several autonomous processors connected via communication links. Such multicomputer systems are also known as *distributed computing systems* (DCS). These definitions are not rigid, and there exist a large number of systems that may not be easy to classify as one or the other. For example, loosely coupled multiprocessor systems (where each processor has a memory of its own, in contrast to tightly coupled multiprocessor systems where the processors share a common memory) mimic many of the characteristics of multicomputer systems and are also classified as distributed computing systems. A general distributed computing system normally has physically well-separated autonomous processors connected via communication links (Figure 1.1). In general, an entire parallel and distributed computing system, or a part of it, may have one of a number of known topologies, such as: bus, star, linear, tree, hypercube, ring, mesh, and so on.

Such systems may or may not be dedicated to a specific application. The main purpose of a DCS is to offer a variety of services to the users. In a DCS, the processing loads may arrive at many sites. A load submitted by a user at some site may be processed right there or at a different site, depending upon the availability of resources and the type of service demanded. When a job has to be processed at

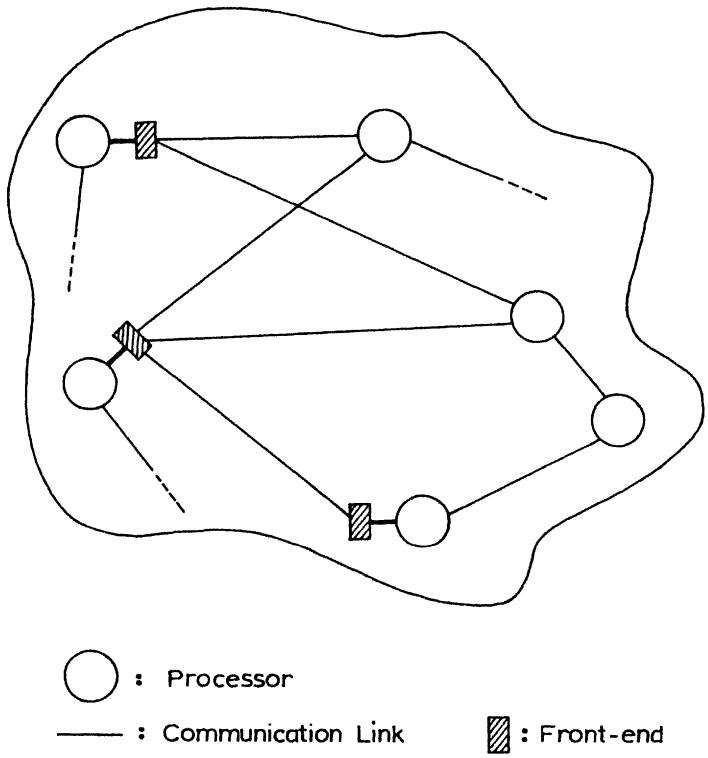

FIGURE 1.1 A Distributed Computing System (DCS)

a different site, it is transferred via communication links. Since the processors are physically well separated in a DCS, it takes a certain amount of time for the load to reach its destination; that is, load transfer from one site to another is subject to nonzero communication delays. Each processor in the network may be engaged in processing its own load, while simultaneously involved in communication with other processors. This is accomplished through dedicated communication coprocessors, often called *front ends*, attached to the processors. These front ends are shown as boxes in Figure 1.1. When a processor is not equipped with such front ends, it has to perform the tasks of both communication and computation.

A parallel processing system attempts to exploit the inherent parallelism in a problem by identifying those modules that can run concurrently so that the solution is obtained in the shortest possible time. Quite frequently, this may involve transfer and exchange of data from one processor to another. Communication overheads due to delays associated with these operations can become substantially high and degrade system performance unless an efficient scheduling policy is adopted.

In subsequent sections, we present a brief account of various aspects of load scheduling/sharing. This discussion is intended to give an overview of some of the important topics addressed in the literature on scheduling theory and is not meant to be exhaustive as these topics are not directly relevant to the specific problem—scheduling arbitrarily divisible loads—addressed in this book.

1.2 THE LOAD SCHEDULING/SHARING PROBLEM

In a broad sense, load scheduling/sharing problems can be classified in many ways. One of the possible classifications is

- Static
- Dynamic

It is also possible to classify them as

- Deterministic
- Stochastic

In *static scheduling*, the objective is to find an optimal schedule of a given number of loads to a set of processors. No dynamics of the system are taken into consideration. The scheduling of, for example, n loads to a set of m processors so that the time required to process all the loads is a minimum is one such problem. On the other hand, if the dynamics of the process are considered, then the scheduling is said to be *dynamic*. In the above example, if loads arrive at arbitrary time instants, and the scheduling policy at any instant in time depends on the current state of the system, then the scheduling is dynamic. It should be noted that the difference in the above two classifications lies in the consideration of the time factor.

In the case of *deterministic scheduling*, all the characteristics of the processing loads, such as their execution times and arrival times, are deterministic quantities and are known a priori. On the other hand, if the arrival of the loads is a random

process and/or the execution times are random variables with some known probability distribution, then the scheduling of loads is said to be *stochastic*.

Static scheduling can be either deterministic or stochastic depending on whether the execution times of the loads are known exactly or only in a stochastic sense. However, most static scheduling problems are considered to be deterministic and are usually solved using graph-theoretic techniques. Similarly, dynamic scheduling problems could also be deterministic or stochastic, though they have usually been considered in the stochastic framework and are solved using queueing-theoretic techniques.

Another related problem is called *load balancing*. It uses similar analytical and computational techniques, but its primary objective is to ensure that all the processors in the system are more or less equally loaded.

All the scheduling problems described above may be either preemptive or nonpreemptive. In the case of *preemptive scheduling*, interruption and subsequent resumption of execution of a load, either in the same processor or elsewhere, is permitted. In *nonpreemptive scheduling* the currently executing load is allowed to run till completion without any interruption.

In general, the formulation of a scheduling problem consists mainly of four steps.

(i) Modeling the system
(ii) Defining the type of processing load
(iii) Formulating an objective function (or cost function)
(iv) Specifying the constraints

The model describes the system, the type of network, the number of processors, whether they are equipped with front ends or not, the topology of the network, and so on. The type of processing load determines the scheduling algorithm. Different types of loads are defined in the next section. In general, the objective of the scheduling problem (the objective function) is as follows: *Given a set of loads and a system, what is the best possible mapping of these loads onto the processors such that the desired cost function is optimized*? As an example, the objective function could be the minimization of the following:

$$\text{Total cost} = \text{computation cost} + \text{communication cost} \tag{1.1}$$

Here the total cost refers to the processing time of a load or a set of loads. As can be seen from Equation (1.1), the total cost takes into account both the computation and the communication costs. The constraints for this problem may be the limitations on the availability of processors and communication channels in the system. Another objective could be to find the bounds on the number of schedules that satisfy certain constraints on communication only. There is another class of scheduling problems in which the objective is to optimally partition a multiprocessor system into smaller subsystems and to then schedule loads onto these subsystems such that the processing time is a minimum.

1.3 CLASSIFICATION OF LOADS

By and large, the scheduling problems discussed in the literature do not attempt to formulate scheduling policies based on the type of jobs submitted by a user, except

perhaps where resource constraints are involved. Usually, the stress has been on designing efficient parallel algorithms in place of conventional sequential algorithms. This requires identification of parallelism in an algorithm and is known as *function parallelism.* However, there is another kind of parallelism that occurs in the data and is called *data parallelism.* This is usually found in computational-intensive tasks with computational loads that consist of large numbers of data points that must be processed by programs, copies of which are resident in all the processors in the system (for example, the SIMD architecture). This adds a new dimension to the scheduling problem. Such loads can be split into several parts and assigned to many processors. But the manner in which this partitioning (or load division) can be done depends on the type of load. In this section we classify computational loads based on their *divisibility property.*

Scheduling of loads has also been categorized as either *job scheduling* or *task scheduling.* A job is defined to be composed of a number of tasks. If a job in its entirety is assigned to a processor, it is called job scheduling, as in distributed computing systems. If different tasks are assigned to different processors, it is called task scheduling, as in parallel processing systems. Thus, the kind of scheduling depends primarily on the type of load being processed. In Figure 1.2 we show a classification of loads based upon their divisibility property, that is, the property that determines whether a load can be decomposed into a set of smaller loads or not.

1.3.1 Indivisible Loads

These loads are independent, indivisible, and, in general, of different sizes. This means that a load cannot be further subdivided and has to be processed in its entirety in a single processor. These loads do not have any precedence relations and in the context of static/deterministic scheduling, these problems are considered to be

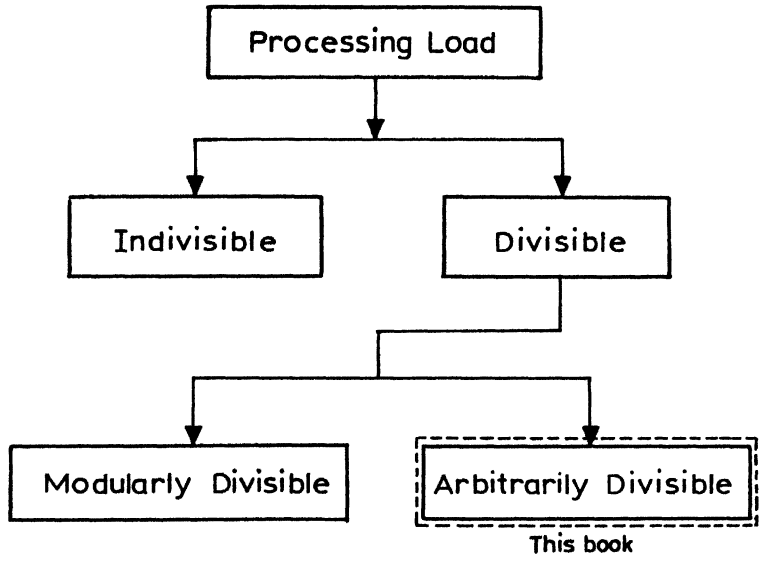

FIGURE 1.2 Classification of Processing Loads

analogous to *bin-packing* problems discussed in the literature. These problems are known to be NP-complete and hence only heuristic algorithms can be proposed to obtain suboptimal solutions in reasonable time. On the other hand, in the case of dynamic/stochastic scheduling schemes, these loads arrive at different time instants and the problem is to schedule them based on the availability and speed of the processors or only on the state of the system.

1.3.2 Modularly Divisible Loads

These loads are a priori subdivided into smaller loads or tasks based on some characteristics of the load or the system. These smaller loads are also called tasks/subtasks or modules, and hence the name. The processing of a load is said to be completed when all its modules are processed. Usually these loads are represented as graphs whose vertices correspond to the modules, and whose edges represent interaction between these modules. This modular representation of a load is known as *Task Interaction Graph* (TIG) in the literature. If these modules are subject to precedence relations, then a directed graph is used. On the other hand, if the graph is not directed, though the modules may exchange information, then it is assumed that they can be executed in any order. They can also be totally independent, in which case they may be modeled as indivisible loads, each consisting of a single module.

1.3.3 Arbitrarily Divisible Loads

This kind of load has the property that all elements in the load demand an identical type of processing. These loads can be arbitrarily partitioned into any number of load fractions. These load fractions may or may not have precedence relations. For example, in the case of Kalman filtering applications, the data is arbitrarily divisible but precedence relations exist among these data segments or load fractions. On the other hand, if the load fractions do not have precedence relations, then each load fraction can be independently processed. This book addresses the problem of scheduling arbitrarily divisible loads, which do not have precedence relations, among several processors. Such loads are encountered in many applications. We describe some of them below.

1.4 DIVISIBLE LOADS: APPLICATIONS

Feature extraction and edge detection in image processing. In computer vision systems, image feature extraction is an extremely important function. This basically consists of two levels of processing, namely, a local computation followed by a nonlocal interprocessor communication and computation. The first level of computation partitions the given image into many segments. Each of these segments is processed locally and independently on different processors. This is done to extract local features of the image from different segments. In the second level of computation, these local features from different processors are exchanged and processed to extract the desired feature. It is at the first level of computation that the load can be considered to be arbitrarily divisible without any precedence relations. Similarly,

edge detection is a very well-known problem in image processing. Here the objective is to detect the edge or boundary of an image. As before, the given image can be arbitrarily partitioned into several subframes of varying sizes (that is, each may contain a different number of pixels) and each of these subframes can be processed independently.

A practical situation in which processing of such data may frequently be necessary involves the space shuttle orbiter, which collects massive volume of image data that has to be communicated to the earth station for processing (by a parallel or distributed computing system). This kind of data also has the potential of arbitrary divisibility. The data can be partitioned and sent directly for processing to a number of processors situated at various geographical points on the surface of the earth, in which case they incur considerable communication delay. Depending on the location of the processing units the communication delays will be different.

Signal processing. A simple application involves the problem of recovering a signal buried in zero-mean noise. The raw data consists of a large number of measurements that can be arbitrarily partitioned and shared among several processors. Another application involves passing a very long linear data file through a digital filter. This might be for frequency shaping purposes (that is, passing the data through a low pass filter) or for pattern matching (that is, passing the data through a matched filter designed to find a particular pattern). In either case the data file may be partitioned among a number of processors. Each processor runs the same filter on its segment of the data. Some care must be taken at the partition boundaries (overlapping the segments slightly is one possibility) when the results are reported back to the originating processor. For the frequency shaping case the output is a filtered data record while for the pattern matching case the results are the location(s) in the data file where the desired pattern was found.

Here we will briefly describe a feature extraction problem in which the arbitrary divisibility property of the image data is exploited to expedite processing. Consider an image in the form of a cluster of pixels that may be a subset of the original image array. The primary task of image feature extraction is to process this data to generate a representation that facilitates higher level symbolic manipulations. It is possible to exploit data parallelism at this stage of processing by assigning different portions of the image array to each of the processors in a parallel or distributed processing system. There could be several subtasks that must be executed to achieve this goal. For example, computation of Hough transforms and region moments are two universally recognized tasks that have to be performed in a majority of situations. In addition, these tasks allow us to exploit data parallelism to the full extent.

To illustrate the above point, consider the Hough transform of straight lines in an image. It is given as an array $B(\rho, \theta)$, each element of which represents the number of pixels whose spatial coordinates (x, y) in the given image array satisfy the equation

$$\rho = x \cos\theta + y \sin\theta \tag{1.2}$$

For each pair (x, y), the value of ρ is computed for a set of discrete values of θ. Thus, each point in the (x, y) plane generates a curve in the (ρ, θ) plane. Based on the nature of these curves and their relative positions one can identify $B(\rho, \theta)$ and obtain information about the features in the given image array. Note that the computation

of Hough transform for each point in the image array is done independent of any of the other points. This aspect makes the data (image array) arbitrarily divisible.

Similarly, the $(k + l)$th order region moment of a cluster of image data is computed as

$$m_{kl} = \sum_x \sum_y x^k y^l I(x, y) \tag{1.3}$$

where $I(x, y)$ is the image intensity of the pixel (x, y). The complete set of moments of order n consists of all moments of order less than or equal to n. Here, too, it is apparent that the data can be arbitrarily divided among the processors to carry out the required computations.

As an illustrative example, let us assume that the data to be processed in the above manner is stored in a (512×512) image array. Let the computations done on a single pixel take 1 unit of time in any of the processors in a network consisting of four identical processors (p_0, p_1, p_2, p_3) connected through a bus and having separate local memories (shown in Figure 1.3). The data to be processed is resident in p_0, which can communicate segments of the data, one at a time, to the other processors. If the communication delay in sending the data is negligible then it is wise to distribute the data in four equal parts. For example, each processor can be assigned 128 rows (see Figure 1.4, left side), thus incurring a processing time of $(1 \times 128 \times 512)$ time units. This strategy is normally recommended in the literature. However, when the communication delay is not negligible, as when the processors are well separated, then this strategy is no longer optimal. Suppose the time delay for communicating one pixel from one processor to another is 10 percent of the computation time per pixel. Then the times taken by each processor to complete its computation is $(1 \times 128 \times 512)$ time units for p_0, $(1.1 \times 128 \times 512)$ time units for p_1, $(1.2 \times 128 \times 512)$ time units for p_2, and $(1.3 \times 128 \times 512)$ time units for p_3. Thus, the processing of the complete data is over only after processor p_3 completes its computation. Hence, the presence of communication delay has increased the processing time by 30 percent. But it is obvious that we can exploit the arbitrary divisibility property of the data to improve performance. For example, let us allocate 147 rows to p_0, 134 rows to p_1, 121 rows to p_2, and 110 rows to p_3 (see Figure 1.4, right side). Then the processing time for each processor is $(1 \times 147 \times 512)$ time units for p_0, $(1.1 \times 134 \times 512)$ time units

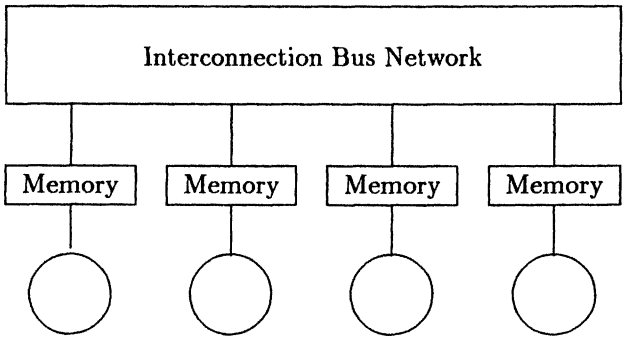

FIGURE 1.3 Bus Network with Four Processors

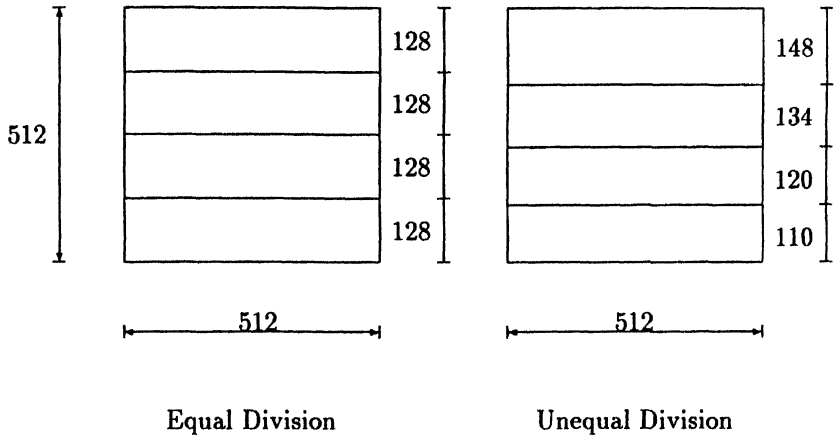

FIGURE 1.4 Partitioning of Data for Image Feature Extraction

for p_1, $(0.1 \times 134 \times 512 + 1.1 \times 121 \times 512)$ time units for p_2, and $(0.1 \times 134 \times 512 + 0.1 \times 121 \times 512 + 1.1 \times 110 \times 512)$ time units for p_3. From the above, we find that p_1 takes the maximum time to complete its computation. For comparison, we can rewrite this time as $(1.152 \times 128 \times 512)$ time units and note that this strategy has produced a 15 percent reduction over the naive equal division strategy.

The above example demonstrates how the arbitrary divisibility property of the data can be exploited to enhance the performance of a real-world image feature extraction algorithm. However, note that since we have allocated data in terms of rows, the data is not arbitrarily divisible in the true sense, but may be considered to be so for large volumes of data. We shall clarify this point in detail in Chapter 2.

Although we use the words scheduling, sharing, and distribution interchangeably in the context of an arbitrarily divisible load, which is shared or distributed among several processors, the phrases *load sharing* and *load distribution* appear to be more appropriate.

1.5 COMMUNICATION DELAY

As demonstrated in the example given in the above section, a crucial aspect in the modeling of a parallel and distributed computing system is the communication delay incurred during transfer of load through the links. These delays, in general, are due to communication processing time, queueing time, transmission time, and propagation time. In order to evaluate the performance of a system, the mathematical model must take into account all these delays. However, for the ease of analysis, approximate models can be employed.

When a load, or a part of it, is communicated to other processors via communication links, the delay (the time it takes to reach the destination) incurred is reflected in the objective function as communication costs, as shown in Equation (1.1). A good scheduling policy must take into account these communication delays. In the case of scheduling modularly divisible loads, it is assumed that the communication costs between two interacting modules are known a priori. In the literature, many

computational-intensive loads have been considered arbitrarily divisible, and are usually partitioned and distributed equally among several identical processors. As shown above in the discussion of Figure (1.4), this equal partitioning is optimal if one considers a system where communication delays are negligible. However, if the same application were to be carried out in a system where communication delays were significant, the load distribution strategy would have to take this delay into account.

1.6 DIVISIBLE LOAD THEORY

Most of the parallel processing literature concentrates on identifying and exploiting inherent parallelisms in sequential programs and on producing parallel programs that can run on multiprocessor systems. However, there is another kind of parallelism that can be exploited. This is the parallelism inherent in large *computational loads*. There exists a large class of loads that involve very large data files that must be processed by programs, copies of which are resident in all the processors in the system. In general, such a load (data file) can be one of the types mentioned in Section 1.3 or a mix of some or all of them; that is, some segments of the load can belong to one category while the others belong to a different category. Identification of these segments in a given load will be the first step toward exploiting parallelism in the data. This knowledge, coupled with system-dependent constraints, like communication delays and processor characteristics, can then be used to partition and share the load optimally among the processors in the network. There is no well-established theory in the literature that helps a user to accomplish this goal. But, judging from its applicability from the viewpoint of scheduling computational-intensive loads, such a theory will indeed be useful. We will call this theory, which identifies and exploits data parallelism in a computational load, *Divisible Load Theory* (DLT). We wish to add a word of caution that the theory cannot yet be considered complete, but is rather at an incipient stage. However, many key concepts that form the basis of DLT have been studied extensively in the literature on scheduling and load balancing.

It is not our intention to cover divisible load theory in this book, but rather to apply some of its fundamental concepts to the problem of scheduling arbitrarily divisible loads in which each data point receives independent processing. Though there has been a considerable body of literature dealing with scheduling/sharing of indivisible and modularly divisible loads, until recently arbitrarily divisible loads and loads of the mixed type have not received much attention. Only very recently has there been interest in the scheduling of arbitrarily divisible loads. Studies in this area address the following question:

Given an arbitrarily divisible load without precedence relations and a multiprocessor/multicomputer system subject to communication delays, in what proportion should the processing load be partitioned and distributed among the processors so that the entire load is processed in the shortest possible time?

One of the major issues is that of computation-communication trade-off relationships. The answer to the above question depends entirely on this issue and we will devote considerable attention to it here. Since this is the first attempt in this direction, we adopt a simplified linear system model for study. This yields a continuous mathematical formulation and provides a flexible analytical tool. Like previous

linear models in other areas (for example, electric circuit theory and queueing theory) this leads to tractable analysis and a rich set of results.

Apart from the applications specified in Section 1.3.3 for arbitrarily divisible loads, there are situations in which one may need to process a large volume of data in almost real time. Such applications include target identification, problems in search theory, and processing of data in distributed sensor networks. In these applications, processing the load in the shortest possible time is a most crucial requirement, and these are precisely the situations in which divisible load theory becomes even more important. However, not all such loads need be arbitrarily divisible. In general, these loads are of a mixed kind.

Finally, we stress that this book presents important theoretical developments concerning scheduling strategies for arbitrarily divisible loads, but does not cover issues related to implementation of these strategies on any specific parallel or distributed computing system. Implementation issues, which are a subject of study in themselves, are beyond the scope of this book.

BIBLIOGRAPHIC NOTES

Section 1.1 The taxonomy of multiprocessor/multicomputer systems can be found in Hwang and Briggs (1989). An excellent overview of parallel and distributed systems and their many salient features is available in Bertsekas and Tsitsiklis (1989), Bokhari (1987), and Lewis et al. (1992). A specific instance of a distributed system made up of several high performance work stations interconnected by a high speed network, for solving computationally-intensive tasks, is described in Atallah et al. (1992). Parallel and distributed systems, their mapping into various kinds of architectures, and their salient features are dealt with in many books [Hwang and Briggs (1989), Siegel (1990), DeCegama (1989)]. In addition, Coulouris and Dollimore (1988) present a comprehensive description of several distributed architectures, their communication networks and protocols, and provide case studies on many real-world architectures. Sloman and Kramer (1987) also address many of the above aspects of distributed systems but with primary emphasis on communications issues. Leighton (1992) describes in detail techniques to implement various mathematical algorithms in linear arrays, trees, and hypercube architectures.

Section 1.2 One of the first papers to provide a classification of scheduling methods in distributed computing systems is by Casavant and Kuhl (1988). Tzafestas and Triantafyllakis (1993) provide an extensive survey of deterministic scheduling policies. Stankovic et al. (1995) also provide a classification of scheduling policies. In addition, they present an excellent account of the applicability of classical scheduling theory to real-time systems. Some recent representative papers that model the scheduling problems in several different ways are Anger et al. (1990), Fernandez-Baca (1989), Krishnamurti and Ma (1992), Lee et al. (1992), Nation et al. (1993), Price and Salama (1990), Shin and Chen (1990), and Veltman et al. (1990). A comprehensive discussion on load sharing policies in multicomputer systems is available in Shivaratri et al. (1992). An excellent collection of papers providing a broad coverage of topics related to scheduling and load balancing in parallel and distributed computing systems is available in Shirazi et al. (1995). The papers in this collection are grouped into chapters based on important issues such as static scheduling; task granularity and partitioning (though in a different context than that addressed in the present book); scheduling tools for parallel processing; load balancing in distributed systems; task migration; and load indices. The introductory notes provided by the editors, prefacing each group of papers, are especially enlightening. Another source of information on current work in this area is the special issue

on scheduling and load balancing published by the *J. of Parallel and Distributed Computing* [Shirazi and Hurson (1992)].

Section 1.3 The concept of data parallelism and function parallelism, in the context of a SIMD architecture, is given in Kim et al. (1991). Job scheduling and task scheduling are defined explicitly in Tantawi and Towsley (1985). The formal classification of loads from the viewpoint of divisibility is of recent origin [Bharadwaj (1994)]. Static scheduling of indivisible loads has been addressed by Coffman (1976) and Coffman et al. (1978), among many others. Stankovic (1985) and Weber (1993) discuss certain problems associated with dynamic/stochastic scheduling policies for such loads. Norman and Thanisch (1993), in a survey paper, discuss scheduling of modularly divisible loads. Task interaction graphs have been used extensively in Lee et al. (1992). Jobs with precedence relationships have been discussed by Murthy and Selvakumar (1993), Peng and Shin (1993), Price and Salama (1990), and Xu (1993), while those without precedence relationships have been considered in a paper by Stone (1977) and later extended by Lee et al. (1992). Scheduling of a collection of independent tasks in a multiprocessor system, and some related issues, are discussed in Swami et al. (1992). The literature on these topics is vast and we have cited only a few representative references. The problem of scheduling arbitrarily divisible loads was first considered by Cheng and Robertazzi (1988).

Section 1.4 Edge detection and feature extraction problems in image processing that expressly use divisible loads have been described in Carlson et al. (1994), Choudhary and Ponnusamy (1991), and Gerogiannis and Orphanoudakis (1993), among others. The data processing requirements for the space shuttle orbiter is a recent area of research and development and a brief account of this work can be found in Binder (1994). The image feature extraction problem given here for illustration is taken from Gerogiannis and Orphanoudakis (1993), where the arbitrary divisibility property of the data has been used to implement a load balancing algorithm, based on an efficient redistribution scheme, on an iPSC/2 hypercube (a MIMD machine) and the CM-2 (a SIMD machine) to obtain significant improvements in performance. However, the communication delays are not considered in any of these studies.

Section 1.5 A general model of communication delay is given in Bertsekas and Tsitsiklis (1989). However, many approximations depending on the application have been discussed in the literature, for example, Stone (1977), Lee et al. (1992), Fernandez-Baca (1989), Norman and Thanisch (1993), and Reed and Grunwald (1987).

Section 1.6 The first paper that uses divisible load theory is by Cheng and Robertazzi (1988) and was motivated by data fusion application in distributed sensor networks [Tenney and Sandell (1981)]. However, the definition of divisible load theory given here is the first attempt to provide a functional, but not a complete, description of the theory.