2 Designing Parallel Algorithms

Now that we have discussed what parallel algorithms look like, we are ready to examine how they can be designed. In this chapter, we show how a problem specification is translated into an algorithm that displays concurrency, scalability, and locality. Issues relating to modularity are discussed in Chapter 4.

Parallel algorithm design is not easily reduced to simple recipes. Rather, it requires the sort of integrative thought that is commonly referred to as "creativity." However, it *can* benefit from a methodical approach that maximizes the range of options considered, that provides mechanisms for evaluating alternatives, and that reduces the cost of backtracking from bad choices. We describe such an approach and illustrate its application to a range of problems. Our goal is to suggest a framework within which parallel algorithm design can be explored. In the process, we hope you will develop intuition as to what constitutes a good parallel algorithm.

After studying this chapter, you should be able to design simple parallel algorithms in a methodical fashion and recognize design flaws that compromise efficiency or scalability. You should be able to partition computations, using both domain and functional decomposition techniques, and know how to recognize and implement both local and global, static and dynamic, structured and unstructured, and synchronous and asynchronous communication structures. You should also be able to use agglomeration as a means of reducing communication and implementation costs and should be familiar with a range of load-balancing strategies.

2.1 Methodical Design

Most programming problems have several parallel solutions. The best solution may differ from that suggested by existing sequential algorithms. The design methodology that we describe is intended to foster an exploratory approach to design in which machine-independent issues such as concurrency are considered early and machine-specific aspects of design are delayed until late in the design process. This methodology structures the design process as four distinct stages: partitioning, communication, agglomeration, and mapping. (The acronym PCAM may serve as a useful reminder of this structure.) In the first two stages, we focus on concurrency and scalability and seek to discover algorithms with these qualities. In the third and fourth stages, attention shifts to locality and other

performance-related issues. The four stages are illustrated in Figure 2.1 and can be summarized as follows:

1. *Partitioning.* The computation that is to be performed and the data operated on by this computation are decomposed into small tasks. Practical issues such as the number of processors in the target computer are ignored, and attention is focused on recognizing opportunities for parallel execution.
2. *Communication.* The communication required to coordinate task execution is determined, and appropriate communication structures and algorithms are defined.
3. *Agglomeration.* The task and communication structures defined in the first two stages of a design are evaluated with respect to performance requirements and implementation costs. If necessary, tasks are combined into larger tasks to improve performance or to reduce development costs.
4. *Mapping.* Each task is assigned to a processor in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs. Mapping can be specified statically or determined at runtime by load-balancing algorithms.

The outcome of this design process can be a program that creates and destroys tasks dynamically, using load-balancing techniques to control the mapping of tasks to processors. Alternatively, it can be an SPMD program that creates exactly one task per processor. The same process of algorithm discovery applies in both cases, although if the goal is to produce an SPMD program, issues associated with mapping are subsumed into the agglomeration phase of the design.

Algorithm design is presented here as a sequential activity. In practice, however, it is a highly parallel process, with many concerns being considered simultaneously. Also, although we seek to avoid backtracking, evaluation of a partial or complete design may require changes to design decisions made in previous steps.

The following sections provide a detailed examination of the four stages of the design process. We present basic principles, use examples to illustrate the application of these principles, and include design checklists that can be used to evaluate designs as they are developed. In the final sections of this chapter, we use three case studies to illustrate the application of these design techniques to realistic problems.

2.2 Partitioning

The partitioning stage of a design is intended to expose opportunities for parallel execution. Hence, the focus is on defining a large number of small tasks in order to yield what is termed a *fine-grained* decomposition of a problem. Just as fine sand is more easily poured than a pile of bricks, a fine-grained decomposition provides the greatest flexibility in terms of potential parallel algorithms. In later design stages, evaluation of communication requirements, the target architecture, or software engineering issues may lead us to forego opportunities for parallel execution identified at this stage. We then revisit the original par-

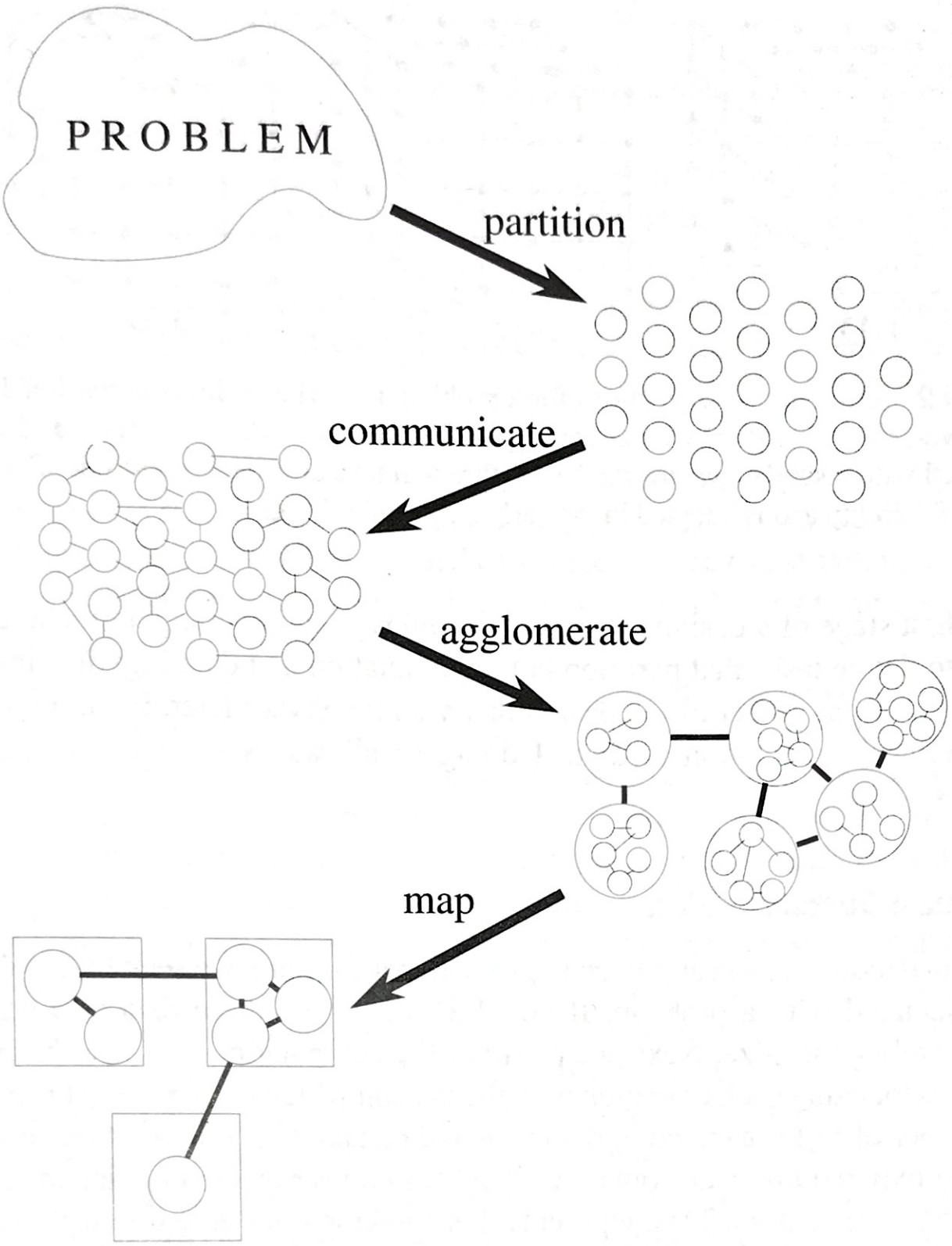

Figure 2.1 PCAM: a design methodology for parallel programs. Starting with a problem specification, we develop a partition, determine communication requirements, agglomerate tasks, and finally map tasks to processors.

tition and agglomerate tasks to increase their size, or granularity. However, in this first stage we wish to avoid prejudging alternative partitioning strategies.

A good partition divides into small pieces both the *computation* associated with a problem and the *data* on which this computation operates. When designing a partition, programmers most commonly first focus on the data associated with a problem, then determine an appropriate partition for the data, and finally work out how to associate computation with data. This partitioning technique is termed *domain decomposition*. The alternative approach—first decomposing the computation to be performed and then dealing with the data—is termed *functional decomposition*. These are complementary techniques which may be applied to different components of a single problem or even applied to the same problem to obtain alternative parallel algorithms.

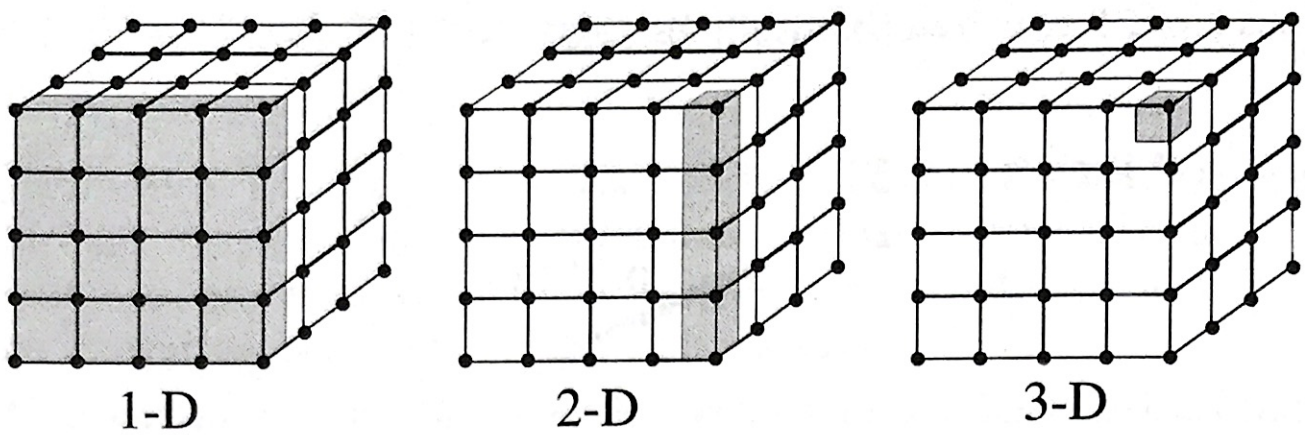

Figure 2.2 Domain decompositions for a problem involving a three-dimensional grid. One-, two-, and three-dimensional decompositions are possible; in each case, data associated with a single task are shaded. A three-dimensional decomposition offers the greatest flexibility and is adopted in the early stages of a design.

In this first stage of a design, we seek to avoid replicating computation and data; that is, we seek to define tasks that partition both computation and data into disjoint sets. Like granularity, this is an aspect of the design that we may revisit later. It can be worthwhile replicating either computation or data if doing so allows us to reduce communication requirements.

2.2.1 Domain Decomposition

In the domain decomposition approach to problem partitioning, we seek first to decompose the data associated with a problem. If possible, we divide these data into small pieces of approximately equal size. Next, we partition the computation that is to be performed, typically by associating each operation with the data on which it operates. This partitioning yields a number of tasks, each comprising some data and a set of operations on that data. An operation may require data from several tasks. In this case, communication is required to move data between tasks. This requirement is addressed in the next phase of the design process.

The data that are decomposed may be the input to the program, the output computed by the program, or intermediate values maintained by the program. Different partitions may be possible, based on different data structures. Good rules of thumb are to focus first on the largest data structure or on the data structure that is accessed most frequently. Different phases of the computation may operate on different data structures or demand different decompositions for the same data structures. In this case, we treat each phase separately and then determine how the decompositions and parallel algorithms developed for each phase fit together. The issues that arise in this situation are discussed in Chapter 4.

Figure 2.2 illustrates domain decomposition in a simple problem involving a three-dimensional grid. (This grid could represent the state of the atmosphere in a weather model, or a three-dimensional space in an image-processing problem.) Computation is performed repeatedly on each grid point. Decompositions in the x, y, and/or z dimensions are possible. In the early stages of a design, we favor the most aggressive decomposition possible, which in this case defines one task for each grid point. Each task maintains as its state the various values associated with its grid point and is responsible for the computation required to update that state.

2.2.2 Functional Decomposition

Functional decomposition represents a different and complementary way of thinking about problems. In this approach, the initial focus is on the computation that is to be performed rather than on the data manipulated by the computation. If we are successful in dividing this computation into disjoint tasks, we proceed to examine the data requirements of these tasks. These data requirements may be disjoint, in which case the partition is complete. Alternatively, they may overlap significantly, in which case considerable communication will be required to avoid replication of data. This is often a sign that a domain decomposition approach should be considered instead.

While domain decomposition forms the foundation for most parallel algorithms, functional decomposition is valuable as a different way of thinking about problems. For this reason alone, it should be considered when exploring possible parallel algorithms. A focus on the computations that are to be performed can sometimes reveal structure in a problem, and hence opportunities for optimization, that would not be obvious from a study of data alone.

As an example of a problem for which functional decomposition is most appropriate, consider Algorithm 1.1. This explores a search tree looking for nodes that correspond to "solutions." The algorithm does not have any obvious data structure that can be decomposed. However, a fine-grained partition can be obtained as described in Section 1.4.3. Initially, a single task is created for the root of the tree. A task evaluates its node and then, if that node is not a leaf, creates a new task for each `search` call (subtree). As illustrated in Figure 1.13, new tasks are created in a wavefront as the search tree is expanded.

Functional decomposition also has an important role to play as a program structuring technique. A functional decomposition that partitions not only the computation that is to be performed but also the code that performs that computation is likely to reduce the complexity of the overall design. This is often the case in computer models of complex systems, which may be structured as collections of simpler models connected via interfaces. For example, a simulation of the earth's climate may comprise components representing the atmosphere, ocean, hydrology, ice, carbon dioxide sources, and so on. While each component may be most naturally parallelized using domain decomposition techniques, the parallel algorithm as a whole is simpler if the system is first decomposed using functional decomposition techniques, even though this process does not yield a large number of tasks (Figure 2.3). This issue is explored in Chapter 4.

2.2.3 Partitioning Design Checklist

The partitioning phase of a design should produce one or more possible decompositions of a problem. Before proceeding to evaluate communication requirements, we use the following checklist to ensure that the design has no obvious flaws. Generally, all these questions should be answered in the affirmative.

1. Does your partition define at least an order of magnitude more tasks than there are processors in your target computer? If not, you have little flexibility in subsequent design stages.
2. Does your partition avoid redundant computation and storage requirements? If not, the resulting algorithm may not be scalable to deal with large problems.

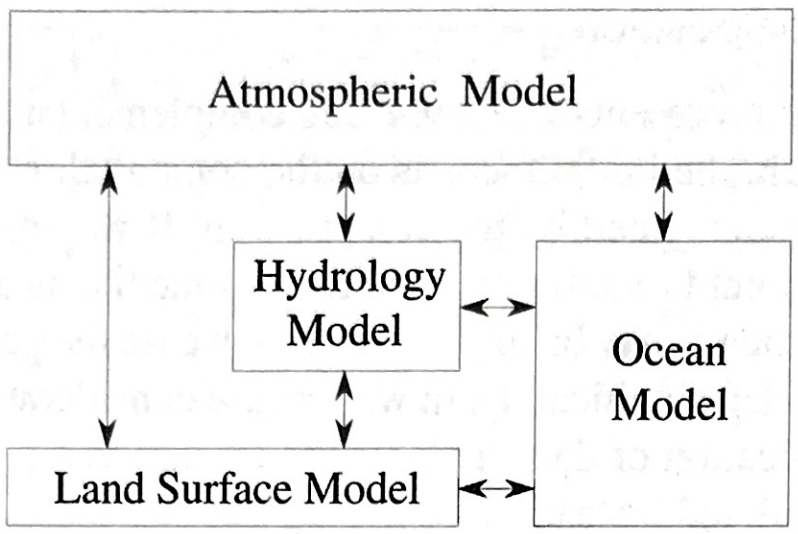

Figure 2.3 Functional decomposition in a computer model of climate. Each model component can be thought of as a separate task, to be parallelized by domain decomposition. Arrows represent exchanges of data between components during computation: the atmosphere model generates wind velocity data that are used by the ocean model, the ocean model generates sea surface temperature data that are used by the atmosphere model, and so on.

3. Are tasks of comparable size? If not, it may be hard to allocate each processor equal amounts of work.
4. Does the number of tasks scale with problem size? Ideally, an increase in problem size should increase the number of tasks rather than the size of individual tasks. If this is not the case, your parallel algorithm may not be able to solve larger problems when more processors are available.
5. Have you identified several alternative partitions? You can maximize flexibility in subsequent design stages by considering alternatives now. Remember to investigate both domain and functional decompositions.

Answers to these questions may suggest that, despite careful thought in this and subsequent design stages, we have a "bad" design. In this situation it is risky simply to push ahead with implementation. We should use the performance evaluation techniques described in Chapter 3 to determine whether the design meets our performance goals despite its apparent deficiencies. We may also wish to revisit the problem specification. Particularly in science and engineering applications, where the problem to be solved may involve a simulation of a complex physical process, the approximations and numerical techniques used to develop the simulation can strongly influence the ease of parallel implementation. In some cases, optimal sequential and parallel solutions to the same problem may use quite different solution techniques. While detailed discussion of these issues is beyond the scope of this book, we present several illustrative examples of them later in this chapter.

2.3 Communication

The tasks generated by a partition are intended to execute concurrently but cannot, in general, execute independently. The computation to be performed in one task will typically require data associated with another task. Data must then be transferred between tasks so