2.19 Continue the illustration of a directory-based cache coherence protocol begun in Figure 2.16. Assume the following five operations now occur in the order listed: CPU 2 reads X, CPU 2 write 5 to X, CPU 1 reads X, CPU 0 reads X, CPU 1 writes 9 to X. Show the states of the directories, caches, and memories after each of these operations.

2.20 Do some research and find, for each category in Flynn's taxonomy, at least one commercial computer fitting that category. (It is OK to name a computer that is no longer available, but you may not name a computer mentioned in this book.)

2.21 Continue the example of the operation of a systolic priority queue begun in Figure 2.22 by illustrating the states it would pass through as it processed these five requests: Insert 4, Extract Minimum, Insert 11, Insert 9, Extract Minimum.

2.22 Explain why contemporary supercomputers are invariably multicomputers.

CHAPTER 3

Parallel Algorithm Design

From the highest to the humblest tasks,
all are of equal honor; all have their part to play.
Winston Churchill

3.1 INTRODUCTION

It's time to start designing parallel algorithms! Our methodology is based on the task/channel model described by Ian Foster [31]. This model facilitates the development of efficient parallel programs, particularly those running on distributed-memory parallel computers.

The first two sections of this chapter describe the task/channel model and the fundamental steps of designing parallel algorithms based on this model. We then study a few simple problems. For each of these problems we design a task/channel parallel algorithm and derive an expression for its expected execution time. In the process our execution time model becomes increasingly sophisticated.

3.2 THE TASK/CHANNEL MODEL

The task/channel model represents a parallel computation as a set of tasks that may interact with each other by sending messages through channels (Figure 3.1). A **task** is a program, its local memory, and a collection of I/O ports. The local memory contains the program's instructions and its private data. A task can send local data values to other tasks via output ports. Conversely, a task can receive data values from other tasks via input ports.

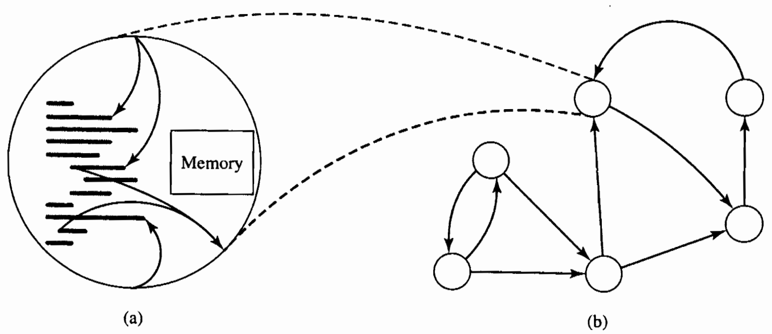

Figure 3.1 The task/channel programming model. (a) A task consists of a program, local memory, and a collection of I/O ports. (b) A parallel computation can be viewed as a directed graph in which vertices represent tasks and directed edges represent communication channels.

A **channel** is a message queue that connects one task's output port with another task's input port. Data values appear at the input port in the same order in which they were placed in the output port at the other end of the channel.

Obviously, a task cannot receive a data value until the task at the other end of the channel has sent it. If a task tries to receive a value at an input port and no value is available, the task must wait until the value appears, and we say the receiving task has **blocked.** In contrast, a process sending a message never blocks, even if previous messages it has sent along the same channel have not yet been received. Put another way, in the task/channel model receiving is a **synchronous** operation, while sending is an **asynchronous** operation.

In the task/channel model local accesses of private data are easily distinguished from nonlocal data accesses that occur over channels. This is good, because we should think of local accesses as being much faster than nonlocal data accesses.

When we talk about the execution time of a parallel algorithm, we are referring to the period of time during which any task is active. The starting time is when all tasks simultaneously begin executing. The finishing time is when the last task has stopped executing.

3.3 FOSTER'S DESIGN METHODOLOGY

Ian Foster has proposed a four-step process for designing parallel algorithms [31]. It encourages the development of scalable parallel algorithms by delaying machine-dependent considerations to the later steps. We'll use Foster's design methodology in this chapter and throughout the rest of the book to develop parallel algorithms for a wide variety of applications.

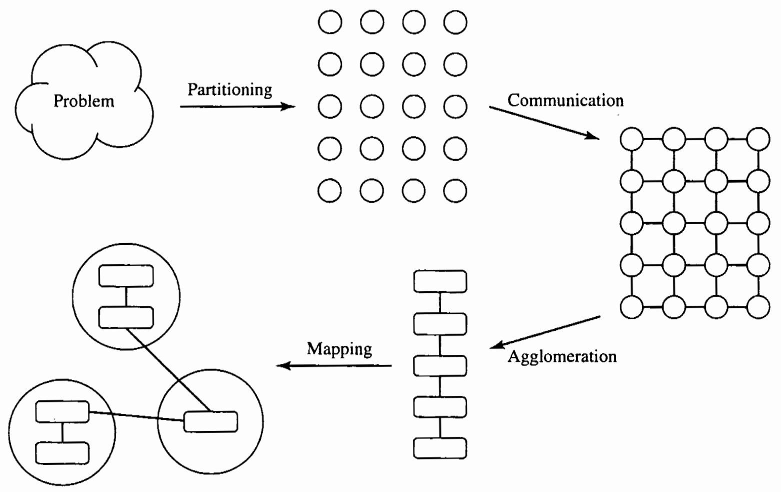

Figure 3.2 Foster's parallel algorithm design methodology.

The four design steps are called partitioning, communication, agglomeration, and mapping (Figure 3.2). In this section we explain each of these steps and provide a checklist that can help you determine if you're producing a good design. While the explanations at this point are rather theoretical, we'll spend the rest of the chapter grounding the theory by working through several practical examples.

3.3.1 Partitioning

When we begin the design of a parallel algorithm, we typically try to discover as much parallelism as possible. **Partitioning** is the process of dividing the computation and the data into pieces. A good partitioning splits both the computation and the data into many small pieces. To do this, we can either take a data-centric approach or a computation-centric approach.

Domain decomposition is the parallel algorithm design approach in which we first divide the data into pieces and then determine how to associate computations with the data. Typically our focus is on the largest and/or most frequently accessed data structure in the program.

Consider the example illustrated in Figure 3.3. Here a three-dimensional matrix is the largest and most frequently accessed data structure. We could partition the matrix into a collection of two-dimensional slices, resulting in a one-dimensional collection of primitive tasks. Alternatively, we could partition the matrix into a collection of one-dimensional slices, resulting in a two-dimensional collection of primitive tasks. Finally, we could consider each matrix element individually, producing a three-dimensional collection of primitive tasks. At this

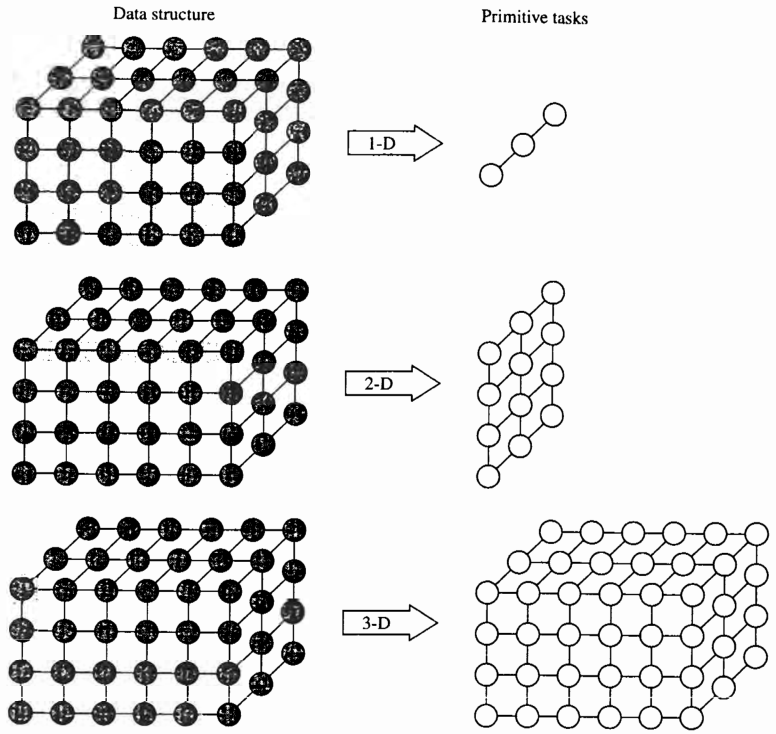

Figure 3.3 Three domain decompositions of a three-dimensional matrix, resulting in markedly different collections of primitive tasks.

point in the design process it is usually best to maximize the number of primitive tasks. Hence the three-dimensional partitioning is preferred.

Functional decomposition is the complementary strategy in which we first divide the computation into pieces and then determine how to associate data items with the individual computations. Often functional decompositions yield collections of tasks that achieve concurrency through pipelining.

For example, consider a high-performance system supporting interactive image-guided brain surgery (Figure 3.4) [37]. Before the surgery begins, the system inputs a set of CT scans of a patient's brain and registers these images, constructing a three-dimensional model. During surgery, the system tracks the position of the surgical instruments, converts them from physical coordinates to image coordinates, and displays on a monitor the position of the instruments amid the surrounding tissue. The system has inherent concurrency. While one task is converting an image from physical coordinates to image coordinates, a

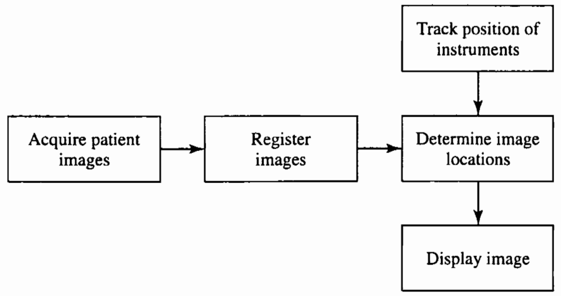

Figure 3.4 Functional decomposition of a system supporting interactive image-guided surgery.

second task can be displaying the previous image, and a third task can be tracking instrument positions for the next image.

Whichever decomposition we choose, we call each of these pieces a **primitive task.** Our goal is to identify as many primitive tasks as possible, because the number of primitive tasks is an upper bound on the parallelism we can exploit.

We can use the following checklist to evaluate the quality of a partitioning. The best designs satisfy all of these attributes (Foster [31]).

- There are at least an order of magnitude more primitive tasks than processors in the target parallel computer. (If this condition is not satisfied, later design options may be too constrained.)
- Redundant computations and redundant data structure storage are minimized. (If this condition is not satisfied, the design may not work well when the size of the problem increases.)
- Primitive tasks are roughly the same size. (If not, it may be hard to balance work among the processors.)
- The number of tasks is an increasing function of the problem size. (If not, it may be impossible to use more processors to solve larger problem instances.)

3.3.2 Communication

After we have identified the primitive tasks, the next step is to determine the communication pattern between them. Parallel algorithms have two kinds of communication patterns: local and global. When a task needs values from a small number of other tasks in order to perform a computation, we create channels from the tasks supplying the data to the task consuming the data. This is an example of a **local communication.**

In contrast, a **global communication** exists when a significant number of the primitive tasks must contribute data in order to perform a computation. An example of a global communication is computing the sum of values held by the primitive processes. While it is important to note when global communications are needed, it is generally not helpful to draw communication channels for them at this stage of the algorithm's design.

We call communication among tasks part of the overhead of a parallel algorithm, because it is something the sequential algorithm does not need to do. Minimizing parallel overhead is an important goal of parallel algorithm design. Keeping this in mind, we can use Foster's checklist to help us evaluate the communication structure of our parallel algorithm.

- The communication operations are balanced among the tasks.
- Each task communicates with only a small number of neighbors.
- Tasks can perform their communications concurrently.
- Tasks can perform their computations concurrently.

3.3.3 Agglomeration

During the first two steps of the parallel algorithm design process, our focus was on identifying as much parallelism as possible. At this point we most likely do not have a design that would execute efficiently on a real parallel computer. For example, if the number of tasks exceeds the number of processors by several orders of magnitude, simply creating these tasks would be a source of significant overhead. In the final two steps of the design process we have a target architecture in mind (e.g., centralized multiprocessor or multicomputer). We consider how to combine primitive tasks into larger tasks and map them onto physical processors to reduce the amount of parallel overhead.

Agglomeration is the process of grouping tasks into larger tasks in order to improve performance or simplify programming. Sometimes we want the number of consolidated tasks to be greater than the number of processors on which our parallel algorithm will execute. Often, however, when developing MPI programs, we leave the agglomeration step with one task per processor. In this case, the mapping of tasks to processors is trivial.

One of the goals of agglomeration is to lower communication overhead. If we agglomerate primitive tasks that communicate with each other, then the communication is completely eliminated, because the data values controlled by the primitive tasks are now in the memory of the consolidated task (Figure 3.5a). We call this **increasing the locality** of the parallel algorithm. If the tasks cannot perform their computations concurrently, because later tasks are waiting for data provided by earlier tasks, then it's usually a good idea to agglomerate the tasks.

Another way to lower communication overhead is to combine groups of sending and receiving tasks, reducing the number of messages being sent (Figure 3.5b). Sending fewer, longer messages takes less time than sending more, shorter messages with the same total length because there is a message startup cost (called the message latency) incurred every time a message is sent, and this time is independent of the length of the message.

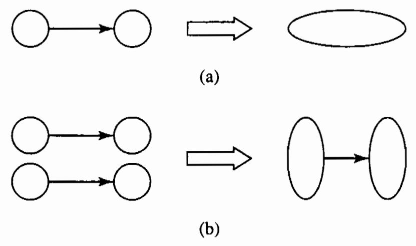

Figure 3.5 Agglomerating tasks can eliminate communications or at least reduce their overhead. (a) Combining tasks that are connected by a channel eliminates that communication, increasing the locality of the parallel algorithm. (b) Combining sending and receiving tasks reduces the number of message transmissions.

A second goal of agglomeration is to maintain the scalability of the parallel design. We want to ensure that we have not combined so many tasks that we will not be able to port our program at some point in the future to a computer with more processors. For example, suppose we are developing a parallel program that manipulates a three-dimensional matrix of size $8 \times 128 \times 256$. We plan to execute our program on a centralized multiprocessor with four CPUs. If we design the parallel algorithm so that the second and third dimensions are agglomerated, we could certainly execute the resulting program on four CPUs. Each task would be responsible for a $2 \times 128 \times 256$ submatrix. Without changing the design, we could even execute on a system with eight CPUs. Each task would be responsible for a $1 \times 128 \times 256$ submatrix. However, we could not port the program to a parallel computer with more than eight CPUs without changing the design, which would probably result in massive changes to the parallel code. Hence the decision to agglomerate the second and third dimensions of the matrix could turn out to be a shortsighted one.

A third goal of agglomeration is to reduce software engineering costs. If we are parallelizing a sequential program, one agglomeration may allow us to make greater use of the existing sequential code, reducing the time and expense of developing the parallel program.

We can use Foster's checklist to evaluate the quality of an agglomeration:

- The agglomeration has increased the locality of the parallel algorithm.
- Replicated computations take less time than the communications they replace.
- The amount of replicated data is small enough to allow the algorithm to scale.

- Agglomerated tasks have similar computational and communications costs.
- The number of tasks is an increasing function of the problem size.
- The number of tasks is as small as possible, yet at least as great as the number of processors in the likely target computers.
- The trade-off between the chosen agglomeration and the cost of modifications to existing sequential code is reasonable.

3.3.4 Mapping

Mapping is the process of assigning tasks to processors. If we are executing our program on a centralized multiprocessor, the operating system automatically maps processes to processors. Hence our discussion assumes the target system is a distributed-memory parallel computer.

The goals of mapping are to maximize processor utilization and minimize interprocessor communication. **Processor utilization** is the average percentage of time the system's processors are actively executing tasks necessary for the solution of the problem. Processor utilization is maximized when the computation is balanced evenly, allowing all processors to begin and end execution at the same time. (Conversely, processor utilization drops when one or more processors are idle while the remainder of the processors are still busy.)

Interprocessor communication increases when two tasks connected by a channel are mapped to different processors. Interprocessor communication decreases when two tasks connected by a channel are mapped to the same processor.

For example, consider the mapping of Figure 3.6. Eight tasks are mapped onto three processors. The left and right processors are responsible for two tasks, while the middle processor is responsible for four tasks. If all processors have the same speed and every task requires the same amount of time to be performed, then

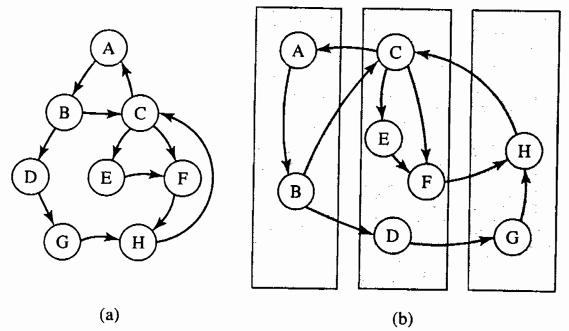

Figure 3.6 The mapping process. (a) A task/channel graph. (b) Mapping of tasks to three processors. Some channels now represent intraprocessor communications, while others represent interprocessor communications.

the middle processor will spend twice as much time executing tasks as the other two processors. If every channel communicates the same amount of data, then the middle processor will also be responsible for twice as many interprocessor communications as the other two processors.

Increasing processor utilization and minimizing interprocessor communication are often conflicting goals.

For example, suppose there are p processors available. Mapping every task to the same processor reduces interprocessor communication to zero, but reduces utilization to $1/p$. Our goal, then, is to choose a mapping that represents a reasonable middle point between maximizing utilization and minimizing communication.

Unfortunately, finding an optimal solution to the mapping problem is NP-hard [38], meaning there are no known polynomial-time algorithms to map tasks to processors to minimize the execution time. Hence we must rely on heuristics that can do a reasonably good job of mapping.

When a problem is partitioned using domain decomposition, the tasks remaining after the agglomeration step often have very similar size, meaning the computational loads are balanced among the tasks. If the communication pattern among the tasks is regular, a good strategy is to create p agglomerated tasks that minimize communication and map each of these tasks to its own processor.

Sometimes the number of tasks is fixed and the communication pattern among them is regular, but the time required to perform each task has significant variability. If nearby tasks tend to have similar computational requirements, then a cyclic (or interleaved) mapping of tasks to processors can result in a balanced computational load, at the expense of higher communications costs.

Some problems yield an unstructured communication pattern among the tasks. In this case it is important to map tasks to processors to minimize the communication overhead of the parallel program. A static load-balancing algorithm, executed before the program begins running, can determine the mapping strategy.

To this point, we have focused on designs utilizing a fixed number of tasks. Dynamic load-balancing algorithms are needed when tasks are created and destroyed at run-time or the communication or computational requirements of tasks vary widely. A dynamic load-balancing algorithm is invoked occasionally during the execution of the parallel program. It analyzes the current tasks and produces a new mapping of tasks to processors.

Finally, some parallel designs rely upon the creation of short-lived tasks to perform particular functions. Tasks do not communicate with each other. Instead, each task is given a subproblem to solve and returns with the solution to that subproblem. Task-scheduling algorithms can be centralized or distributed.

In a centralized task-scheduling algorithm, the pool of processors is divided into one manager and many workers. The manager processor maintains a list of tasks to be assigned. When a worker processor has nothing to do, it requests a task from the manager. The manager replies with a task. The worker completes the task, returns the solution, and requests another task. A potential problem with manager/worker–style task scheduling is that the manager can become a bottleneck. To some extent this problem can be ameliorated by allocating multiple tasks at a time or allowing workers to prefetch tasks while they are working on earlier tasks.

In a distributed task-scheduling algorithm, each processor maintains its own list of available tasks. A mechanism is needed to spread the available tasks among the processors. Some algorithms rely on a "push" strategy. Processors with too many available tasks send some of them to neighboring processors. Other algorithms rely on a "pull" strategy. Processors with no work to do ask neighboring processors for work. A challenge with distributed task-scheduling algorithms is determining the termination condition. The uncompleted tasks are spread among the processors, and it is difficult for any process to know when all of them have been completed. In contrast, the manager process in a manager/worker–style algorithm always knows exactly how many uncompleted tasks remain.

Other task-scheduling algorithms represent a compromise between the centralized and decentralized algorithms we have described. For example, a two-level hierarchical manager/worker strategy has two levels of managers. The higher-level manager supervises a group of managers. Each lower-level manager allocates tasks to its own group of workers. Periodically the managers communicate with each other to balance the number of unassigned tasks held by each low-level manager.

Figure 3.7 summarizes how different characteristics of the parallel algorithm lead to different mapping strategies. Because the mapping strategy depends on

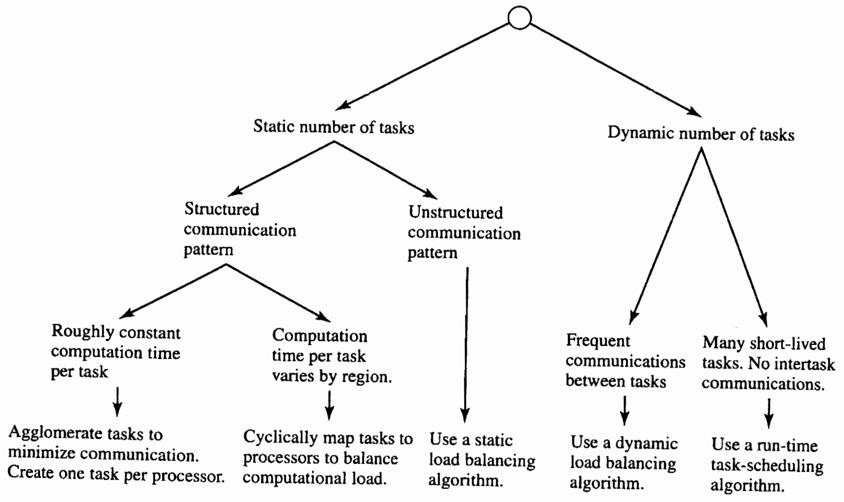
Static number of tasks
Dynamic number of tasks
Structured communication pattern
Unstructured communication pattern
Roughly constant computation time per task
Computation time per task varies by region.
Frequent communications between tasks
Many short-lived tasks. No intertask communications.
Agglomerate tasks to minimize communication. Create one task per processor.
Cyclically map tasks to processors to balance computational load.
Use a static load balancing algorithm.
Use a dynamic load balancing algorithm.
Use a run-time task-scheduling algorithm.

Figure 3.7 A decision tree to choose a mapping strategy. The best strategy depends on characteristics of the tasks produced as a result of the partitioning, communication, and agglomeration steps.

decisions made earlier in the parallel algorithm design process, it is important to keep an open mind during the design process. The following checklist (from Foster [31]) can help you decide if you've done a good job of considering design alternatives:

- Designs based on one task per processor and multiple tasks per processor have been considered.
- Both static and dynamic allocation of tasks to processors have been evaluated.
- If a dynamic allocation of tasks to processors has been chosen, the manager (task allocator) is not a bottleneck to performance.
- If a static allocation of tasks to processors has been chosen, the ratio of tasks to processors is at least 10:1.

3.4 BOUNDARY VALUE PROBLEM

3.4.1 Introduction

Let's apply our parallel algorithm design methodology to a simple, yet realistic, problem. See Figure 3.8. A thin rod made of uniform material is surrounded by a blanket of insulation so that temperature changes along the length of the rod are a result of heat transfer at the ends of the rod and heat conduction along the length of the rod. The rod has length 1. Both ends of the rod are exposed to an ice bath having temperature 0°C, while the initial temperature at distance x from the end of the rod is $100\sin(\pi x)$.

Over time, the rod gradually cools. A partial differential equation models the temperature at any point of the rod at any point in time. The finite difference method is one way to solve a partial differential equation on a computer. Figure 3.9 shows a finite difference approximation to the rod-cooling problem. Each curve represents the temperature distribution of the rod at some point in time. The curves drop as time increases. If you look carefully, you can see that each "curve" is actually composed of 10 line segments. In reality, the temperature distributions

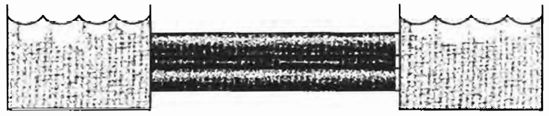

Figure 3.8 A thin rod (dark gray) is suspended between two ice baths. The ends of the rod are in contact with the icewater. The rod is surrounded by a thick blanket of insulation. We can use a partial differential equation to model the temperature at any point on the rod as a function of time.