```
/* Synchronize all processes/threads */
Barrier();
my_start = Get_current_time();

/* Code that we want to time */
. . .

my_finish = Get_current_time();
my_elapsed = my_finish - my_start;

/* Find the max across all processes/threads */
global_elapsed = Global_max(my_elapsed);
if (my_rank == 0)
   printf("The elapsed time = %e seconds\n", global_elapsed);
```

Here, we first execute a **barrier** function that approximately synchronizes all of the processes/threads. We would like for all the processes/threads to return from the call simultaneously, but such a function usually can only guarantee that all the processes/threads have started the call when the first process/thread returns. We then execute the code as before and each process/thread finds the time it took. Then all the processes/threads call a global maximum function, which returns the largest of the elapsed times, and process/thread 0 prints it out.

We also need to be aware of the *variability* in timings. When we run a program several times, it's extremely likely that the elapsed time will be different for each run. This will be true even if each time we run the program we use the same input and the same systems. It might seem that the best way to deal with this would be to report either a mean or a median run-time. However, it's unlikely that some outside event could actually make our program run faster than its best possible run-time. So instead of reporting the mean or median time, we usually report the *minimum* time.

Running more than one thread per core can cause dramatic increases in the variability of timings. More importantly, if we run more than one thread per core, the system will have to take extra time to schedule and deschedule cores, and this will add to the overall run-time. Therefore, we rarely run more than one thread per core.

Finally, as a practical matter, since our programs won't be designed for high-performance I/O, we'll usually not include I/O in our reported run-times.

2.7 PARALLEL PROGRAM DESIGN

So we've got a serial program. How do we parallelize it? We know that in general we need to divide the work among the processes/threads so that each process gets roughly the same amount of work and communication is minimized. In most cases, we also need to arrange for the processes/threads to synchronize and communicate.

Unfortunately, there isn't some mechanical process we can follow; if there were, we could write a program that would convert any serial program into a parallel program, but, as we noted in Chapter 1, in spite of a tremendous amount of work and some progress, this seems to be a problem that has no universal solution.

However, Ian Foster provides an outline of steps in his online book *Designing and Building Parallel Programs* [19]:

1. *Partitioning*. Divide the computation to be performed and the data operated on by the computation into small tasks. The focus here should be on identifying tasks that can be executed in parallel.
2. *Communication*. Determine what communication needs to be carried out among the tasks identified in the previous step.
3. *Agglomeration or aggregation*. Combine tasks and communications identified in the first step into larger tasks. For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.
4. *Mapping*. Assign the composite tasks identified in the previous step to processes/threads. This should be done so that communication is minimized, and each process/thread gets roughly the same amount of work.

This is sometimes called **Foster's methodology**.

2.7.1 An example

Let's look at a small example. Suppose we have a program that generates large quantities of floating point data that it stores in an array. In order to get some feel for the distribution of the data, we can make a histogram of the data. Recall that to make a histogram, we simply divide the range of the data up into equal sized subintervals, or *bins*, determine the number of measurements in each bin, and plot a bar graph showing the relative sizes of the bins. As a very small example, suppose our data are

$$1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3, 4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9.$$

Then the data lie in the range 0–5, and if we choose to have five bins, the histogram might look something like Figure 2.20.

A serial program

It's pretty straightforward to write a serial program that generates a histogram. We need to decide what the bins are, determine the number of measurements in each bin, and print the bars of the histogram. Since we're not focusing on I/O, we'll limit ourselves to just the first two steps, so the input will be

1. the number of measurements, `data_count`;
2. an array of `data_count` floats, `data`;
3. the minimum value for the bin containing the smallest values, `min_meas`;
4. the maximum value for the bin containing the largest values, `max_meas`;
5. the number of bins, `bin_count`;

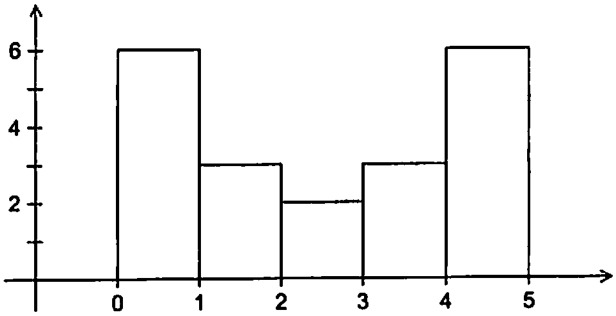

FIGURE 2.20

A histogram

The output will be an array containing the number of elements of data that lie in each bin. To make things precise, we'll make use of the following data structures:

- bin_maxes. An array of bin_count floats
- bin_counts. An array of bin_count ints

The array bin_maxes will store the upper bound for each bin, and bin_counts will store the number of data elements in each bin. To be explicit, we can define

```
      bin_width = (max_meas - min_meas)/bin_count
```

Then bin_maxes will be initialized by

```
   for (b = 0; b < bin_count; b++)
      bin_maxes[b] = min_meas + bin_width*(b+1);
```

We'll adopt the convention that bin b will be all the measurements in the range

```
bin_maxes[b-1] <= measurement < bin_maxes[b]
```

Of course, this doesn't make sense if $b = 0$, and in this case we'll use the rule that bin 0 will be the measurements in the range

```
min_meas <= measurement < bin_maxes[0]
```

This means we always need to treat bin 0 as a special case, but this isn't too onerous.

Once we've initialized bin_maxes and assigned 0 to all the elements of bin_counts, we can get the counts by using the following pseudo-code:

```
for (i = 0; i < data_count; i++) {
   bin = Find_bin(data[i], bin_maxes, bin_count, min_meas);
   bin_counts[bin]++;
}
```

The Find_bin function returns the bin that data[i] belongs to. This could be a simple linear search function: search through bin_maxes until you find a bin b that satisfies

```
   bin_maxes[b-1] <= data[i] < bin_maxes[b]
```

(Here we're thinking of bin_maxes[−1] as min_meas.) This will be fine if there aren't very many bins, but if there are a lot of bins, binary search will be much better.

Parallelizing the serial program

If we assume that data_count is much larger than bin_count, then even if we use binary search in the Find_bin function, the vast majority of the work in this code will be in the loop that determines the values in bin_counts. The focus of our parallelization should therefore be on this loop, and we'll apply Foster's methodology to it. The first thing to note is that the outcomes of the steps in Foster's methodology are by no means uniquely determined, so you shouldn't be surprised if at any stage you come up with something different.

For the first step we might identify two types of tasks: finding the bin to which an element of data belongs and incrementing the appropriate entry in bin_counts.

For the second step, there must be a communication between the computation of the appropriate bin and incrementing an element of bin_counts. If we represent our tasks with ovals and communications with arrows, we'll get a diagram that looks something like that shown in Figure 2.21. Here, the task labelled with "data[i]" is determining which bin the value data[i] belongs to, and the task labelled with "bin_counts[b]++" is incrementing bin_counts[b].

For any fixed element of data, the tasks "find the bin b for element of data" and "increment bin_counts[b]" can be aggregated, since the second can only happen once the first has been completed.

However, when we proceed to the final or mapping step, we see that if two processes or threads are assigned elements of data that belong to the same bin b, they'll both result in execution of the statement bin_counts[b]++. If bin_counts[b] is shared (e.g., the array bin_counts is stored in shared-memory), then this will result in a race condition. If bin_counts has been partitioned among the processes/threads, then updates to its elements will require communication. An alternative is to store multiple "local" copies of bin_counts and add the values in the local copies after all the calls to Find_bin.

If the number of bins, bin_count, isn't absolutely gigantic, there shouldn't be a problem with this. So let's pursue this alternative, since it is suitable for use on both shared- and distributed-memory systems.

In this setting, we need to update our diagram so that the second collection of tasks increments loc_bin_cts[b]. We also need to add a third collection of tasks, adding the various loc_bin_cts[b] to get bin_counts[b]. See Figure 2.22. Now we

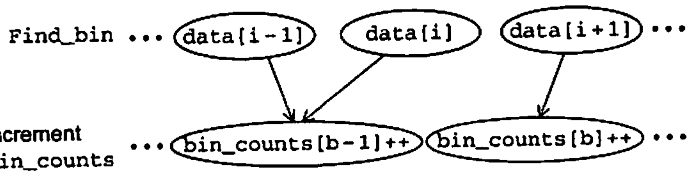

FIGURE 2.21

The first two stages of Foster's methodology

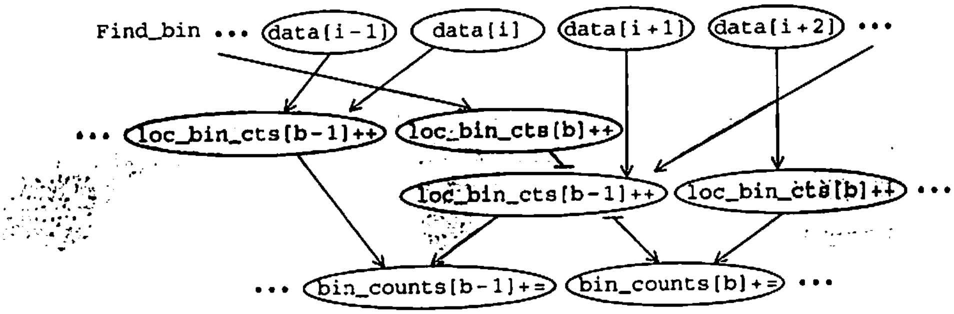

FIGURE 2.22

Alternative definition of tasks and communication

see that if we create an array `loc_bin_cts` for each process/thread, then we can map the tasks in the first two groups as follows:

1. Elements of `data` are assigned to the processes/threads so that each process/thread gets roughly the same number of elements.
2. Each process/thread is responsible for updating its `loc_bin_cts` array on the basis of its assigned elements.

To finish up, we need to add the elements `loc_bin_cts[b]` into `bin_counts[b]`. If both the number of processes/threads is small and the number of bins is small, all of the additions can be assigned to a single process/thread. If the number of bins is much larger than the number of processes/threads, we can divide the bins among the processes/threads in much the same way that we divided the elements of `data`. If the number of processes/threads is large, we can use a tree-structured global sum similar to the one we discussed in Chapter 1. The only difference is that now the sending process/threads are sending an array, and the receiving process/threads are receiving and adding an array. Figure 2.23 shows an example with eight processes/threads. Each

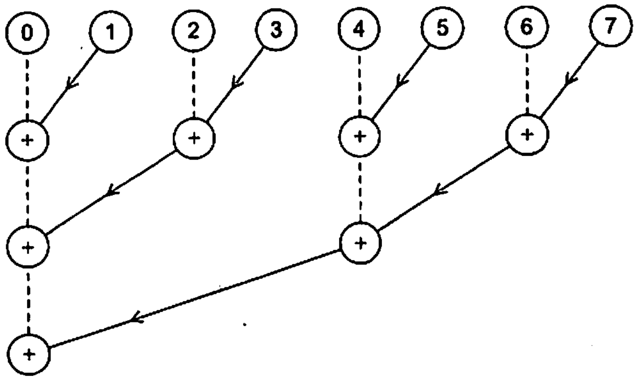

FIGURE 2.23

Adding the local arrays

circle in the top row corresponds to a process/thread. Between the first and the second rows, the odd-numbered processes/threads make their loc_bin_cts available to the even-numbered processes/threads. Then in the second row, the even-numbered processes/threads add the new counts to their existing counts. Between the second and third rows the process is repeated with the processes/threads whose ranks aren't divisible by four sending to those whose are. This process is repeated until process/thread 0 has computed bin_counts.

2.8 WRITING AND RUNNING PARALLEL PROGRAMS

In the past, virtually all parallel program development was done using a text editor such as vi or Emacs, and the program was either compiled and run from the command line or from within the editor. Debuggers were also typically started from the command line. Now there are also integrated development environments (IDEs) available from Microsoft, the Eclipse project, and others; see [16, 38].

In smaller shared-memory systems, there is a single running copy of the operating system, which ordinarily schedules the threads on the available cores. On these systems, shared-memory programs can usually be started using either an IDE or the command line. Once started, the program will typically use the console and the keyboard for input from stdin and output to stdout and stderr. On larger systems, there may be a batch scheduler, that is, a user requests a certain number of cores, and specifies the path to the executable and where input and output should go (typically to files in secondary storage).

In typical distributed-memory and hybrid systems, there is a host computer that is responsible for allocating nodes among the users. Some systems are purely *batch* systems, which are similar to shared-memory batch systems. Others allow users to check out nodes and run jobs interactively. Since job startup often involves communicating with remote systems, the actual startup is usually done with a script. For example, MPI programs are usually started with a script called mpirun or mpiexec.

As usual, RTFD, which is sometimes translated as "read the fine documentation."

2.9 ASSUMPTIONS

As we noted earlier, we'll be focusing on homogeneous MIMD systems—systems in which all of the nodes have the same architecture—and our programs will be SPMD. Thus, we'll write a single program that can use branching to have multiple different behaviors. We'll assume the cores are identical but that they operate asynchronously. We'll also assume that we always run at most one process or thread of our program on a single core, and we'll often use static processes or threads. In other words, we'll often start all of our processes or threads at more or less the same time, and when they're done executing, we'll terminate them at more or less the same time.