CHAPTER

Parallel Program Development

6

In the last three chapters we haven't just learned about parallel APIs, we've also developed a number of small parallel programs, and each of these programs has involved the implementation of a parallel algorithm. In this chapter, we'll look at a couple of larger examples: solving *n*-body problems and solving the traveling salesperson problem. For each problem, we'll start by looking at a serial solution and examining modifications to the serial solution. As we apply Foster's methodology, we'll see that there are some striking similarities between developing shared- and distributed-memory programs. We'll also see that in parallel programming there are problems that we need to solve for which there is no serial analog. We'll see that there are instances in which, as parallel programmers, we'll have to start "from scratch."

6.1 TWO *n*-BODY SOLVERS

In an *n*-body problem, we need to find the positions and velocities of a collection of interacting particles over a period of time. For example, an astrophysicist might want to know the positions and velocities of a collection of stars, while a chemist might want to know the positions and velocities of a collection of molecules or atoms. An *n*-body solver is a program that finds the solution to an *n*-body problem by simulating the behavior of the particles. The input to the problem is the mass, position, and velocity of each particle at the start of the simulation, and the output is typically the position and velocity of each particle at a sequence of user-specified times, or simply the position and velocity of each particle at the end of a user-specified time period.

Let's first develop a serial *n*-body solver. Then we'll try to parallelize it for both shared- and distributed-memory systems.

6.1.1 The problem

For the sake of explicitness, let's write an *n*-body solver that simulates the motions of planets or stars. We'll use Newton's second law of motion and his law of universal gravitation to determine the positions and velocities. Thus, if particle q has position $\mathbf{s}_q(t)$ at time t, and particle k has position $\mathbf{s}_k(t)$, then the force on particle q exerted by

An Introduction to Parallel Programming. DOI: 10.1016/B978-0-12-374260-5.00006-3

particle k is given by

$$\mathbf{f}_{qk}(t) = -\frac{Gm_q m_k}{\left|\mathbf{s}_q(t) - \mathbf{s}_k(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_k(t)\right]. \tag{6.1}$$

Here, G is the gravitational constant ($6.673 \times 10^{-11}\text{m}^3/(\text{kg} \cdot \text{s}^2)$), and m_q and m_k are the masses of particles q and k, respectively. Also, the notation $\left|\mathbf{s}_q(t) - \mathbf{s}_k(t)\right|$ represents the distance from particle k to particle q. Note that in general the positions, the velocities, the accelerations, and the forces are vectors, so we're using boldface to represent these variables. We'll use an italic font to represent the other, scalar, variables, such as the time t and the gravitational constant G.

We can use Formula 6.1 to find the total force on any particle by adding the forces due to all the particles. If our n particles are numbered $0, 1, 2, \ldots, n-1$, then the total force on particle q is given by

$$\mathbf{F}_q(t) = \sum_{\substack{k=0 \\ k \neq q}}^{n-1} \mathbf{f}_{qk} = -Gm_q \sum_{\substack{k=0 \\ k \neq q}}^{n-1} \frac{m_k}{\left|\mathbf{s}_q(t) - \mathbf{s}_k(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_k(t)\right]. \tag{6.2}$$

Recall that the acceleration of an object is given by the second derivative of its position and that Newton's second law of motion states that the force on an object is given by its mass multiplied by its acceleration, so if the acceleration of particle q is $\mathbf{a}_q(t)$, then $\mathbf{F}_q(t) = m_q\mathbf{a}_q(t) = m_q\mathbf{s}_q''(t)$, where $\mathbf{s}_q''(t)$ is the second derivative of the position $\mathbf{s}_q(t)$. Thus, we can use Formula 6.2 to find the acceleration of particle q:

$$\mathbf{s}_q''(t) = -G\sum_{\substack{j=0 \\ j \neq q}}^{n-1} \frac{m_j}{\left|\mathbf{s}_q(t) - \mathbf{s}_j(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_j(t)\right]. \tag{6.3}$$

Thus Newton's laws give us a system of *differential* equations—equations involving derivatives—and our job is to find at each time t of interest the position $\mathbf{s}_q(t)$ and velocity $\mathbf{v}_q(t) = \mathbf{s}_q'(t)$.

We'll suppose that we either want to find the positions and velocities at the times

$$t = 0, \Delta t, 2\Delta t, \ldots, T\Delta t,$$

or, more often, simply the positions and velocities at the final time $T\Delta t$. Here, Δt and T are specified by the user, so the input to the program will be n, the number of particles, Δt, T, and, for each particle, its mass, its initial position, and its initial velocity. In a fully general solver, the positions and velocities would be three-dimensional vectors, but in order to keep things simple, we'll assume that the particles will move in a plane, and we'll use two-dimensional vectors instead.

The output of the program will be the positions and velocities of the n particles at the timesteps $0, \Delta t, 2\Delta t, \ldots$, or just the positions and velocities at $T\Delta t$. To get the output at only the final time, we can add an input option in which the user specifies whether she only wants the final positions and velocities.

6.1.2 Two serial programs

In outline, a serial n-body solver can be based on the following pseudocode:

```
1     Get input data;
2     for each timestep {
3        if (timestep output) Print positions and velocities of
             particles;
4        for each particle q
5           Compute total force on q;
6        for each particle q
7           Compute position and velocity of q;
8     }
9     Print positions and velocities of particles;
```

We can use our formula for the total force on a particle (Formula 6.2) to refine our pseudocode for the computation of the forces in Lines 4–5:

```
for each particle q {
   for each particle k != q {
      x_diff = pos[q][X] - pos[k][X];
      y_diff = pos[q][Y] - pos[k][Y];
      dist = sqrt(x_diff*x_diff + y_diff*y_diff);
      dist_cubed = dist*dist*dist;
      forces[q][X] -= G*masses[q]*masses[k]/dist_cubed * x_diff;
      forces[q][Y] -= G*masses[q]*masses[k]/dist_cubed * y_diff;
   }
}
```

Here, we're assuming that the forces and the positions of the particles are stored as two-dimensional arrays, `forces` and `pos`, respectively. We're also assuming we've defined constants `X = 0` and `Y = 1`. So the x-component of the force on particle q is `forces[q][X]` and the y-component is `forces[q][Y]`. Similarly, the components of the position are `pos[q][X]` and `pos[q][Y]`. (We'll take a closer look at data structures shortly.)

We can use Newton's third law of motion, that is, for every action there is an equal and opposite reaction, to halve the total number of calculations required for the forces. If the force on particle q due to particle k is $\mathbf{f}_{qk}$, then the force on k due to q is $-\mathbf{f}_{qk}$. Using this simplification we can modify our code to compute forces, as shown in Program 6.1. To better understand this pseudocode, imagine the individual forces as a two-dimensional array:

$$\begin{bmatrix} 0 & \mathbf{f}_{01} & \mathbf{f}_{02} & \cdots & \mathbf{f}_{0,n-1} \\ -\mathbf{f}_{01} & 0 & \mathbf{f}_{12} & \cdots & \mathbf{f}_{1,n-1} \\ -\mathbf{f}_{02} & -\mathbf{f}_{12} & 0 & \cdots & \mathbf{f}_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\mathbf{f}_{0,n-1} & -\mathbf{f}_{1,n-1} & -\mathbf{f}_{2,n-1} & \cdots & 0 \end{bmatrix}.$$

(Why are the diagonal entries 0?) Our original solver simply adds all of the entries in row q to get `forces[q]`. In our modified solver, when $q = 0$, the body of the loop

```
for each particle q
   forces[q] = 0;
for each particle q {
   for each particle k > q {
      x_diff = pos[q][X] - pos[k][X];
      y_diff = pos[q][Y] - pos[k][Y];
      dist = sqrt(x_diff*x_diff + y_diff*y_diff);
      dist_cubed = dist*dist*dist;
      force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
      force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff

      forces[q][X] += force_qk[X];
      forces[q][Y] += force_qk[Y];
      forces[k][X] -= force_qk[X];
      forces[k][Y] -= force_qk[Y];
   }
}
```

Program 6.1: A reduced algorithm for computing n-body forces

`for each particle q` will add the entries in row 0 into `forces[0]`. It will also add the kth entry in column 0 into `forces[k]` for $k = 1, 2, \ldots, n-1$. In general, the qth iteration will add the entries to the right of the diagonal (that is, to the right of the 0) in row q into `forces[q]`, and the entries below the diagonal in column q will be added into their respective forces, that is, the kth entry will be added in to `forces[k]`.

Note that in using this modified solver, it's necessary to initialize the `forces` array in a separate loop, since the qth iteration of the loop that calculates the forces will, in general, add the values it computes into `forces[k]` for $k = q+1, q+2, \ldots, n-1$, not just `forces[q]`.

In order to distinguish between the two algorithms, we'll call the n-body solver with the original force calculation, the *basic* algorithm, and the solver with the number of calculations reduced, the *reduced* algorithm.

The position and the velocity remain to be found. We know that the acceleration of particle q is given by

$$\mathbf{a}_q(t) = \mathbf{s}_q''(t) = \mathbf{F}_q(t)/m_q,$$

where $\mathbf{s}_q''(t)$ is the second derivative of the position $\mathbf{s}_q(t)$ and $\mathbf{F}_q(t)$ is the force on particle q. We also know that the velocity $\mathbf{v}_q(t)$ is the first derivative of the position $\mathbf{s}_q'(t)$, so we need to integrate the acceleration to get the velocity, and we need to integrate the velocity to get the position.

We might at first think that we can simply find an antiderivative of the function in Formula 6.3. However, a second look shows us that this approach has problems: the right-hand side contains unknown functions $\mathbf{s}_q$ and $\mathbf{s}_k$—not just the variable t—so we'll instead use a **numerical** method for *estimating* the position and the velocity. This means that rather than trying to find simple closed formulas, we'll approximate

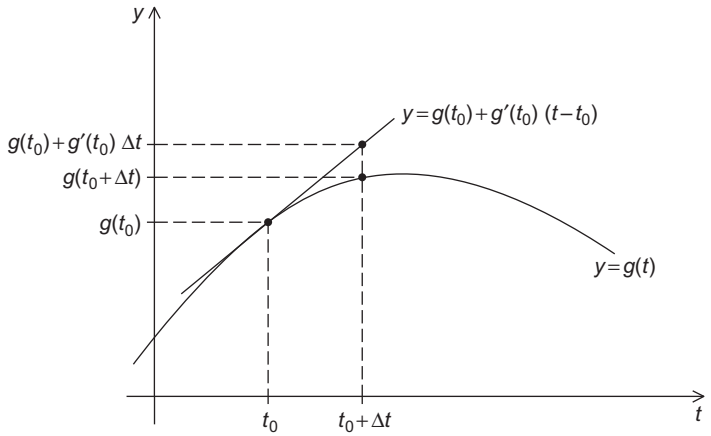

FIGURE 6.1

Using the tangent line to approximate a function

the values of the position and velocity at the times of interest. There are *many* possible choices for numerical methods, but we'll use the simplest one: Euler's method, which is named after the famous Swiss mathematician Leonhard Euler (1707–1783). In Euler's method, we use the tangent line to approximate a function. The basic idea is that if we know the value of a function $g(t_0)$ at time t_0 and we also know its derivative $g'(t_0)$ at time t_0, then we can approximate its value at time $t_0 + \Delta t$ by using the tangent line to the graph of $g(t_0)$. See Figure 6.1 for an example. Now if we know a point $(t_0, g(t_0))$ on a line, and we know the slope of the line $g'(t_0)$, then an equation for the line is given by

$$y = g(t_0) + g'(t_0)(t - t_0).$$

Since we're interested in the time $t = t_0 + \Delta t$, we get

$$g(t + \Delta t) \approx g(t_0) + g'(t_0)(t + \Delta t - t) = g(t_0) + \Delta t g'(t_0).$$

Note that this formula will work even when $g(t)$ and y are vectors: when this is the case, $g'(t)$ is also a vector and the formula just adds a vector to a vector multiplied by a scalar, Δt.

Now we know the value of $\mathbf{s}_q(t)$ and $\mathbf{s}'_q(t)$ at time 0, so we can use the tangent line and our formula for the acceleration to compute $\mathbf{s}_q(\Delta t)$ and $\mathbf{v}_q(\Delta t)$:

$$\mathbf{s}_q(\Delta t) \approx \mathbf{s}_q(0) + \Delta t \mathbf{s}'_q(0) = \mathbf{s}_q(0) + \Delta t \mathbf{v}_q(0),$$

$$\mathbf{v}_q(\Delta t) \approx \mathbf{v}_q(0) + \Delta t \mathbf{v}'_q(0) = \mathbf{v}_q(0) + \Delta t \mathbf{a}_q(0) = \mathbf{v}_q(0) + \Delta t \frac{1}{m_q} \mathbf{F}_q(0).$$

When we try to extend this approach to the computation of $\mathbf{s}_q(2\Delta t)$ and $\mathbf{s}'_q(2\Delta t)$, we see that things are a little bit different, since we don't know the exact value of $\mathbf{s}_q(\Delta t)$

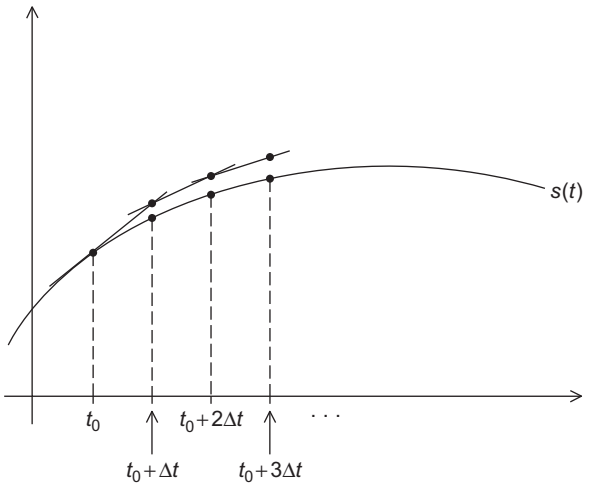

FIGURE 6.2

Euler's method

and $\mathbf{s}'_q(\Delta t)$. However, if our approximations to $\mathbf{s}_q(\Delta t)$ and $\mathbf{s}'_q(\Delta t)$ are good, then we should be able to get a reasonably good approximation to $\mathbf{s}_q(2\Delta t)$ and $\mathbf{s}'_q(2\Delta t)$ using the same idea. This is what Euler's method does (see Figure 6.2).

Now we can complete our pseudocode for the two n-body solvers by adding in the code for computing position and velocity:

```
pos[q][X] += delta_t*vel[q][X];
pos[q][Y] += delta_t*vel[q][Y];
vel[q][X] += delta_t/masses[q]*forces[q][X];
vel[q][Y] += delta_t/masses[q]*forces[q][Y];
```

Here, we're using `pos[q]`, `vel[q]`, and `forces[q]` to store the position, the velocity, and the force, respectively, of particle q.

Before moving on to parallelizing our serial program, let's take a moment to look at data structures. We've been using an array type to store our vectors:

```
#define DIM 2

typedef double vect_t[DIM];
```

A struct is also an option. However, if we're using arrays and we decide to change our program so that it solves three-dimensional problems, in principle, we only need to change the macro `DIM`. If we try to do this with structs, we'll need to rewrite the code that accesses individual components of the vector.

For each particle, we need to know the values of

- its mass,
- its position,

- its velocity,
- its acceleration, and
- the total force acting on it.

Since we're using Newtonian physics, the mass of each particle is constant, but the other values will, in general, change as the program proceeds. If we examine our code, we'll see that once we've computed a new value for one of these variables for a given timestep, we never need the old value again. For example, we don't need to do anything like this

```
new_pos_q = f(old_pos_q);
new_vel_q = g(old_pos_q, new_pos_q);
```

Also, the acceleration is only used to compute the velocity, and its value can be computed in one arithmetic operation from the total force, so we only need to use a local, temporary variable for the acceleration.

For each particle it suffices to store its mass and the current value of its position, velocity, and force. We could store these four variables as a struct and use an array of structs to store the data for all the particles. Of course, there's no reason that all of the variables associated with a particle need to be grouped together in a struct. We can split the data into separate arrays in a variety of different ways. We've chosen to group the mass, position, and velocity into a single struct and store the forces in a separate array. With the forces stored in contiguous memory, we can use a fast function such as `memset` to quickly assign zeroes to all of the elements at the beginning of each iteration:

```
#include <string.h>  /* For memset */
. . .
vect_t* forces = malloc(n*sizeof(vect_t));
. . .
for (step = 1; step <= n_steps; step++) {
   . . .
   /* Assign 0 to each element of the forces array */
   forces = memset(forces, 0, n*sizeof(vect_t));
   for (part = 0; part < n-1; part++)
      Compute_force(part, forces, . . .)
   . . .
}
```

If the force on each particle were a member of a struct, the force members wouldn't occupy contiguous memory in an array of structs, and we'd have to use a relatively slow `for` loop to assign zero to each element.

6.1.3 Parallelizing the n-body solvers

Let's try to apply Foster's methodology to the n-body solver. Since we initially want *lots* of tasks, we can start by making our tasks the computations of the positions, the velocities, and the total forces at each timestep. In the basic algorithm, the algorithm in which the total force on each particle is calculated directly from Formula 6.2, the

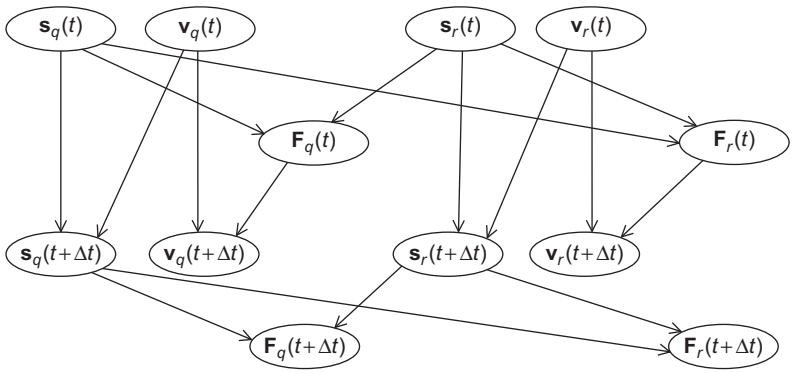

FIGURE 6.3

Communications among tasks in the basic n-body solver

computation of $\mathbf{F}_q(t)$, the total force on particle q at time t, requires the positions of each of the particles $\mathbf{s}_r(t)$, for each r. The computation of $\mathbf{v}_q(t+\Delta t)$ requires the velocity at the previous timestep, $\mathbf{v}_q(t)$, and the force, $\mathbf{F}_q(t)$, at the previous timestep. Finally, the computation of $\mathbf{s}_q(t+\Delta t)$ requires $\mathbf{s}_q(t)$ and $\mathbf{v}_q(t)$. The communications among the tasks can be illustrated as shown in Figure 6.3. The figure makes it clear that most of the communication among the tasks occurs among the tasks associated with an individual particle, so if we agglomerate the computations of $\mathbf{s}_q(t)$, $\mathbf{v}_q(t)$, and $\mathbf{F}_q(t)$, our intertask communication is greatly simplified (see Figure 6.4). Now the tasks correspond to the particles and, in the figure, we've labeled the communications with the data that's being communicated. For example, the arrow from particle q at timestep t to particle r at timestep t is labeled with $\mathbf{s}_q$, the position of particle q.

For the reduced algorithm, the "intra-particle" communications are the same. That is, to compute $\mathbf{s}_q(t+\Delta t)$ we'll need $\mathbf{s}_q(t)$ and $\mathbf{v}_q(t)$, and to compute $\mathbf{v}_q(t+\Delta t)$, we'll need $\mathbf{v}_q(t)$ and $\mathbf{F}_q(t)$. Therefore, once again it makes sense to agglomerate the computations associated with a single particle into a composite task.

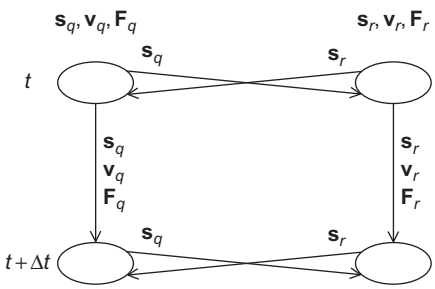

FIGURE 6.4

Communications among agglomerated tasks in the basic n-body solver

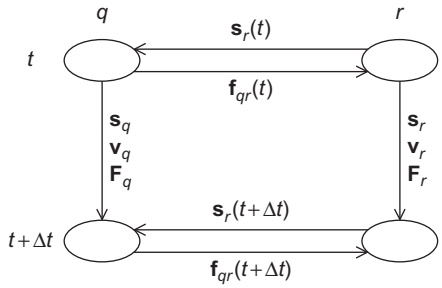

FIGURE 6.5

Communications among agglomerated tasks in the reduced n-body solver ($q < r$)

Recollect that in the reduced algorithm, we make use of the fact that the force $\mathbf{f}_{rq} = -\mathbf{f}_{qr}$. So if $q < r$, then the communication *from* task r *to* task q is the same as in the basic algorithm—in order to compute $\mathbf{F}_q(t)$, task/particle q will need $\mathbf{s}_r(t)$ from task/particle r. However, the communication from task q to task r is no longer $s_q(t)$, it's the force on particle q due to particle r, that is, $\mathbf{f}_{qr}(t)$. See Figure 6.5.

The final stage in Foster's methodology is mapping. If we have n particles and T timesteps, then there will be nT tasks in both the basic and the reduced algorithm. Astrophysical n-body problems typically involve thousands or even millions of particles, so n is likely to be several orders of magnitude greater than the number of available cores. However, T may also be much larger than the number of available cores. So, in principle, we have two "dimensions" to work with when we map tasks to cores. However, if we consider the nature of Euler's method, we'll see that attempting to assign tasks associated with a single particle at different timesteps to different cores won't work very well. Before estimating $\mathbf{s}_q(t + \Delta t)$ and $\mathbf{v}_q(t + \Delta t)$, Euler's method must "know" $\mathbf{s}_q(t)$, $\mathbf{v}_q(t)$, and $\mathbf{a}_q(t)$. Thus, if we assign particle q at time t to core c_0, and we assign particle q at time $t + \Delta t$ to core $c_1 \neq c_0$, then we'll have to communicate $\mathbf{s}_q(t), \mathbf{v}_q(t)$, and $\mathbf{F}_q(t)$ from c_0 to c_1. Of course, if particle q at time t and particle q at time $t + \Delta t$ are mapped to the same core, this communication won't be necessary, so once we've mapped the task consisting of the calculations for particle q at the first timestep to core c_0, we may as well map the subsequent computations for particle q to the same cores, since we can't simultaneously execute the computations for particle q at two different timesteps. Thus, mapping tasks to cores will, in effect, be an assignment of particles to cores.

At first glance, it might seem that any assignment of particles to cores that assigns roughly n/`thread_count` particles to each core will do a good job of balancing the workload among the cores, and for the basic algorithm this is the case. In the basic algorithm the work required to compute the position, velocity, and force is the same for every particle. However, in the reduced algorithm the work required in the forces computation loop is much greater for lower-numbered iterations than the work required for higher-numbered iterations. To see this, recall the pseudocode that computes the total force on particle q in the reduced algorithm:

```
for each particle k > q {
   x_diff = pos[q][X] - pos[k][X];
   y_diff = pos[q][Y] - pos[k][Y];
   dist = sqrt(x_diff*x_diff + y_diff*y_diff);
   dist_cubed = dist*dist*dist;
   force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
   force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff;

   forces[q][X] += force_qk[X];
   forces[q][Y] += force_qk[Y];
   forces[k][X] -= force_qk[X];
   forces[k][Y] -= force_qk[Y];
}
```

Then, for example, when $q = 0$, we'll make $n - 1$ passes through the `for each particle k > q` loop, while when $q = n - 1$, we won't make any passes through the loop. Thus, for the reduced algorithm we would expect that a cyclic partition of the particles would do a better job than a block partitition of evenly distributing the *computation*.

However, in a shared-memory setting, a cyclic partition of the particles among the cores is almost certain to result in a much higher number of cache misses than a block partition, and in a distributed-memory setting, the overhead involved in communicating data that has a cyclic distribution will probably be greater than the overhead involved in communicating data that has a block distribution (see Exercises 6.8 and 6.9).

Therefore with a composite task consisting of all of the computations associated with a single particle throughout the simulation, we conclude the following:

1. A block distribution will give the best performance for the basic n-body solver.
2. For the reduced n-body solver, a cyclic distribution will best distribute the workload in the computation of the forces. However, this improved performance *may* be offset by the cost of reduced cache performance in a shared-memory setting and additional communication overhead in a distributed-memory setting.

In order to make a final determination of the optimal mapping of tasks to cores, we'll need to do some experimentation.

6.1.4 A word about I/O

You may have noticed that our discussion of parallelizing the n-body solver hasn't touched on the issue of I/O, even though I/O can figure prominently in both of our serial algorithms. We've discussed the problem of I/O several times in earlier chapters. Recall that different parallel systems vary widely in their I/O capabilities, and with the very basic I/O that is commonly available it is very difficult to obtain high performance. This basic I/O was designed for use by single-process, single-threaded programs, and when multiple processes or multiple threads attempt to access the I/O buffers, the system makes no attempt to schedule their access. For example, if multiple threads attempt to execute

```
printf("Hello from thread %d of %d\n", my_rank, thread_count);
```

more or less simultaneously, the order in which the output appears will be unpredictable. Even worse, one thread's output may not even appear as a single line. It can happen that the output from one thread appears as multiple segments, and the individual segments are separated by output from other threads.

Thus, as we've noted earlier, except for debug output, we generally assume that one process/thread does all the I/O, and when we're timing program execution, we'll use the option to only print output for the final timestep. Furthermore, we won't include this output in the reported run-times.

Of course, even if we're ignoring the cost of I/O, we can't ignore its existence. We'll briefly discuss its implementation when we discuss the details of our parallel implementations.

6.1.5 Parallelizing the basic solver using OpenMP

How can we use OpenMP to map tasks/particles to cores in the basic version of our n-body solver? Let's take a look at the pseudocode for the serial program:

```
for each timestep {
  if (timestep output) Print positions and velocities of particles;
  for each particle q
     Compute total force on q;
  for each particle q
     Compute position and velocity of q;
}
```

The two inner loops are both iterating over particles. So, in principle, parallelizing the two inner `for` loops will map tasks/particles to cores, and we might try something like this:

```
    for each timestep {
       if (timestep output) Print positions and velocities of
           particles;
#      pragma omp parallel for
       for each particle q
          Compute total force on q;
#      pragma omp parallel for
       for each particle q
          Compute position and velocity of q;
    }
```

We may not like the fact that this code could do a lot of forking and joining of threads, but before dealing with that, let's take a look at the loops themselves: we need to see if there are any race conditions caused by loop-carried dependences.

In the basic version the first loop has the following form:

```
#  pragma omp parallel for
   for each particle q {
      forces[q][X] = forces[q][Y] = 0;
      for each particle k != q {
         x_diff = pos[q][X] - pos[k][X];
         y_diff = pos[q][Y] - pos[k][Y];
```

```
            dist = sqrt(x_diff*x_diff + y_diff*y_diff);
            dist_cubed = dist*dist*dist;
            forces[q][X] -= G*masses[q]*masses[k]/dist_cubed * x_diff;
            forces[q][Y] -= G*masses[q]*masses[k]/dist_cubed * y_diff;
         }
      }
```

Since the iterations of the `for each particle q` loop are partitioned among the threads, only one thread will access `forces[q]` for any q. Different threads do access the same elements of the `pos` array and the `masses` array. However, these arrays are only *read* in the loop. The remaining variables are used for temporary storage in a single iteration of the inner loop, and they can be private. Thus, the parallelization of the first loop in the basic algorithm won't introduce any race conditions.

The second loop has the form:

```
#  pragma omp parallel for
   for each particle q {
      pos[q][X] += delta_t*vel[q][X];
      pos[q][Y] += delta_t*vel[q][Y];
      vel[q][X] += delta_t/masses[q]*forces[q][X];
      vel[q][Y] += delta_t/masses[q]*forces[q][Y];
   }
```

Here, a single thread accesses `pos[q]`, `vel[q]`, `masses[q]`, and `forces[q]` for any particle q, and the scalar variables are only read, so parallelizing this loop also won't introduce any race conditions.

Let's return to the issue of repeated forking and joining of threads. In our pseudocode, we have

```
   for each timestep {
      if (timestep output) Print positions and velocities of
          particles;
#     pragma omp parallel for
      for each particle q
         Compute total force on q;
#     pragma omp parallel for
      for each particle q
         Compute position and velocity of q;
   }
```

We encountered a similar issue when we parallelized odd-even transposition sort (see Section 5.6.2). In that case, we put a `parallel` directive before the outermost loop and used OpenMP `for` directives for the inner loops. Will a similar strategy work here? That is, can we do something like this?

```
#  pragma omp parallel
   for each timestep {
      if (timestep output) Print positions and velocities of
          particles;
#     pragma omp for
      for each particle q
```

```
         Compute total force on q;
#     pragma omp for
      for each particle q
         Compute position and velocity of q;
   }
```

This will have the desired effect on the two `for each particle` loops: the same team of threads will be used in both loops and for every iteration of the outer loop. However, we have a clear problem with the output statement. As it stands now, every thread will print all the positions and velocities, and we only want one thread to do the I/O. However, OpenMP provides the `single` directive for exactly this situation: we have a team of threads executing a block of code, but a part of the code should only be executed by one of the threads. Adding the `single` directive gives us the following pseudocode:

```
#  pragma omp parallel
   for each timestep {
      if (timestep output) {
#        pragma omp single
         Print positions and velocities of particles;
      }
#     pragma omp for
      for each particle q
         Compute total force on q;
#     pragma omp for
      for each particle q
         Compute position and velocity of q;
   }
```

There are still a few issues that we need to address. The most important has to do with possible race conditions introduced in the transition from one statement to another. For example, suppose thread 0 completes the first `for each particle` loop before thread 1, and it then starts updating the positions and velocities of its assigned particles in the second `for each particle` loop. Clearly, this could cause thread 1 to use an updated position in the first `for each particle` loop. However, recall that there is an implicit barrier at the end of each structured block that has been parallelized with a `for` directive. So, if thread 0 finishes the first inner loop before thread 1, it will block until thread 1 (and any other threads) finish the first inner loop, and it won't start the second inner loop until all the threads have finished the first. This will also prevent the possibility that a thread might rush ahead and print positions and velocities before they've all been updated by the second loop.

There's also an implicit barrier after the `single` directive, although in this program the barrier isn't necessary. Since the output statement won't update any memory locations, it's OK for some threads to go ahead and start executing the next iteration before output has been completed. Furthermore, the first inner `for` loop in the next iteration only updates the `forces` array, so it can't cause a thread executing the output statement to print incorrect values, and because of the barrier at the end of the first inner loop, no thread can race ahead and start updating positions and velocities in

the second inner loop before the output has been completed. Thus, we could modify the `single` directive with a `nowait` clause. If the OpenMP implementation supports it, this simply eliminates the implied barrier associated with the `single` directive. It can also be used with `for`, `parallel for`, and `parallel` directives. Note that in this case, addition of the `nowait` clause is unlikely to have much effect on performance, since the two `for each particle` loops have implied barriers that will prevent any one thread from getting more than a few statements ahead of any other.

Finally, we may want to add a schedule clause to each of the `for` directives in order to insure that the iterations have a block partition:

```
#     pragma omp for schedule(static, n/thread_count)
```

6.1.6 Parallelizing the reduced solver using OpenMP

The reduced solver has an additional inner loop: the initialization of the `forces` array to 0. If we try to use the same parallelization for the reduced solver, we should also parallelize this loop with a `for` directive. What happens if we try this? That is, what happens if we try to parallelize the reduced solver with the following pseudocode?

```
#  pragma omp parallel
   for each timestep {
      if (timestep output) {
#        pragma omp single
         Print positions and velocities of particles;
      }
#     pragma omp for
      for each particle q
         forces[q] = 0.0;
#     pragma omp for
      for each particle q
         Compute total force on q;
#     pragma omp for
      for each particle q
         Compute position and velocity of q;
   }
```

Parallelization of the initialization of the forces should be fine, as there's no dependence among the iterations. The updating of the positions and velocities is the same in both the basic and reduced solvers, so if the computation of the forces is OK, then this should also be OK.

How does parallelization affect the correctness of the loop for computing the forces? Recall that in the reduced version, this loop has the following form:

```
#  pragma omp for
   for each particle q {
      force_qk[X] = force_qk[Y] = 0;
      for each particle k > q {
```

```
            x_diff = pos[q][X] - pos[k][X];
            y_diff = pos[q][Y] - pos[k][Y];
            dist = sqrt(x_diff*x_diff + y_diff*y_diff);
            dist_cubed = dist*dist*dist;
            force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
            force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff;

            forces[q][X] += force_qk[X];
            forces[q][Y] += force_qk[Y];
            forces[k][X] -= force_qk[X];
            forces[k][Y] -= force_qk[Y];
         }
      }
```

As before, the variables of interest are `pos`, `masses`, and `forces`, since the values in the remaining variables are only used in a single iteration, and hence, can be private. Also, as before, elements of the `pos` and `masses` arrays are only read, not updated. We therefore need to look at the elements of the `forces` array. In this version, unlike the basic version, a thread *may* update elements of the `forces` array other than those corresponding to its assigned particles. For example, suppose we have two threads and four particles and we're using a block partition of the particles. Then the total force on particle 3 is given by

$$\mathbf{F}_3 = -\mathbf{f}_{03} - \mathbf{f}_{13} - \mathbf{f}_{23}.$$

Furthermore, thread 0 will compute $\mathbf{f}_{03}$ and $\mathbf{f}_{13}$, while thread 1 will compute $\mathbf{f}_{23}$. Thus, the updates to `forces[3]` *do* create a race condition. In general, then, the updates to the elements of the `forces` array introduce race conditions into the code.

A seemingly obvious solution to this problem is to use a `critical` directive to limit access to the elements of the `forces` array. There are at least a couple of ways to do this. The simplest is to put a `critical` directive before all the updates to `forces`

```
#  pragma omp critical
   {
      forces[q][X] += force_qk[X];
      forces[q][Y] += force_qk[Y];
      forces[k][X] -= force_qk[X];
      forces[k][Y] -= force_qk[Y];
   }
```

However, with this approach access to the elements of the `forces` array will be effectively serialized. Only one element of `forces` can be updated at a time, and contention for access to the critical section is actually likely to seriously degrade the performance of the program. See Exercise 6.3.

An alternative would be to have one critical section for each particle. However, as we've seen, OpenMP doesn't readily support varying numbers of critical sections, so we would need to use one lock for each particle instead and our updates would

look something like this:

```
omp_set_lock(&locks[q]);
forces[q][X] += force_qk[X];
forces[q][Y] += force_qk[Y];
omp_unset_lock(&locks[q]);

omp_set_lock(&locks[k]);
forces[k][X] -= force_qk[X];
forces[k][Y] -= force_qk[Y];
omp_unset_lock(&locks[k]);
```

This assumes that the master thread will create a shared array of locks, one for each particle, and when we update an element of the `forces` array, we first set the lock corresponding to that particle. Although this approach performs much better than the single critical section, it still isn't competitive with the serial code. See Exercise 6.4.

Another possible solution is to carry out the computation of the forces in two phases. In the first phase, each thread carries out exactly the same calculations it carried out in the erroneous parallelization. However, now the calculations are stored in its *own* array of forces. Then, in the second phase, the thread that has been assigned particle q will add the contributions that have been computed by the different threads. In our example above, thread 0 would compute $-\mathbf{f}_{03} - \mathbf{f}_{13}$, while thread 1 would compute $-\mathbf{f}_{23}$. After each thread was done computing its contributions to the forces, thread 1, which has been assigned particle 3, would find the total force on particle 3 by adding these two values.

Let's look at a slightly larger example. Suppose we have three threads and six particles. If we're using a block partition of the particles, then the computations in the first phase are shown in Table 6.1. The last three columns of the table show each thread's contribution to the computation of the total forces. In phase 2 of the computation, the thread specified in the first column of the table will add the contents of each of its assigned rows—that is, each of its assigned particles.

Note that there's nothing special about using a block partition of the particles. Table 6.2 shows the same computations if we use a cyclic partition of the particles.

Table 6.1 First-Phase Computations for a Reduced Algorithm with Block Partition

		Thread		
Thread	**Particle**	**0**	**1**	**2**
0	0	$\mathbf{f}_{01}+\mathbf{f}_{02}+\mathbf{f}_{03}+\mathbf{f}_{04}+\mathbf{f}_{05}$	0	0
	1	$-\mathbf{f}_{01}+\mathbf{f}_{12}+\mathbf{f}_{13}+\mathbf{f}_{14}+\mathbf{f}_{15}$	0	0
1	2	$-\mathbf{f}_{02}-\mathbf{f}_{12}$	$\mathbf{f}_{23}+\mathbf{f}_{24}+\mathbf{f}_{25}$	0
	3	$-\mathbf{f}_{03}-\mathbf{f}_{13}$	$-\mathbf{f}_{23}+\mathbf{f}_{34}+\mathbf{f}_{35}$	0
2	4	$-\mathbf{f}_{04}-\mathbf{f}_{14}$	$-\mathbf{f}_{24}-\mathbf{f}_{34}$	$\mathbf{f}_{45}$
	5	$-\mathbf{f}_{05}-\mathbf{f}_{15}$	$-\mathbf{f}_{25}-\mathbf{f}_{35}$	$-\mathbf{f}_{45}$

Table 6.2 First-Phase Computations for a Reduced Algorithm with Cyclic Partition

		Thread		
Thread	**Particle**	**0**	**1**	**2**
0	0	$\mathbf{f}_{01}+\mathbf{f}_{02}+\mathbf{f}_{03}+\mathbf{f}_{04}+\mathbf{f}_{05}$	0	0
1	1	$-\mathbf{f}_{01}$	$\mathbf{f}_{12}+\mathbf{f}_{13}+\mathbf{f}_{14}+\mathbf{f}_{15}$	0
2	2	$-\mathbf{f}_{02}$	$-\mathbf{f}_{12}$	$\mathbf{f}_{23}+\mathbf{f}_{24}+\mathbf{f}_{25}$
0	3	$-\mathbf{f}_{03}+\mathbf{f}_{34}+\mathbf{f}_{35}$	$-\mathbf{f}_{13}$	$-\mathbf{f}_{23}$
1	4	$-\mathbf{f}_{04}-\mathbf{f}_{34}$	$-\mathbf{f}_{14}+\mathbf{f}_{45}$	$-\mathbf{f}_{24}$
2	5	$-\mathbf{f}_{05}-\mathbf{f}_{35}$	$-\mathbf{f}_{15}-\mathbf{f}_{45}$	$-\mathbf{f}_{25}$

Note that if we compare this table with the table that shows the block partition, it's clear that the cyclic partition does a better job of balancing the load.

To implement this, during the first phase our revised algorithm proceeds as before, except that each thread adds the forces it computes into its own subarray of `loc_forces`:

```
#  pragma omp for
   for each particle q {
      force_qk[X] = force_qk[Y] = 0;
      for each particle k > q {
         x_diff = pos[q][X] - pos[k][X];
         y_diff = pos[q][Y] - pos[k][Y];
         dist = sqrt(x_diff*x_diff + y_diff*y_diff);
         dist_cubed = dist*dist*dist;
         force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
         force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff;

         loc_forces[my_rank][q][X] += force_qk[X];
         loc_forces[my_rank][q][Y] += force_qk[Y];
         loc_forces[my_rank][k][X] -= force_qk[X];
         loc_forces[my_rank][k][Y] -= force_qk[Y];
      }
   }
```

During the second phase, each thread adds the forces computed by all the threads for its assigned particles:

```
#  pragma omp for
   for (q = 0; q < n; q++) {
      forces[q][X] = forces[q][Y] = 0;
      for (thread = 0; thread < thread_count; thread++) {
         forces[q][X] += loc_forces[thread][q][X];
         forces[q][Y] += loc_forces[thread][q][Y];
      }
   }
```

Before moving on, we should make sure that we haven't inadvertently introduced any new race conditions. During the first phase, since each thread writes to its own

subarray, there isn't a race condition in the updates to `loc_forces`. Also, during the second phase, only the "owner" of thread q writes to `forces[q]`, so there are no race conditions in the second phase. Finally, since there is an implied barrier after each of the parallelized `for` loops, we don't need to worry that some thread is going to race ahead and make use of a variable that hasn't been properly initialized, or that some slow thread is going to make use of a variable that has had its value changed by another thread.

6.1.7 Evaluating the OpenMP codes

Before we can compare the basic and the reduced codes, we need to decide how to schedule the parallelized `for` loops. For the basic code, we've seen that any schedule that divides the iterations equally among the threads should do a good job of balancing the computational load. (As usual, we're assuming no more than one thread/core.) We also observed that a block partitioning of the iterations would result in fewer cache misses than a cyclic partition. Thus, we would expect that a block schedule would be the best option for the basic version.

In the reduced code, the amount of work done in the first phase of the computation of the forces decreases as the `for` loop proceeds. We've seen that a cyclic schedule should do a better job of assigning more or less equal amounts of work to each thread. In the remaining parallel `for` loops—the initialization of the `loc_forces` array, the second phase of the computation of the forces, and the updating of the positions and velocities—the work required is roughly the same for all the iterations. Therefore, *taken out of context* each of these loops will probably perform best with a block schedule. However, the schedule of one loop can affect the performance of another (see Exercise 6.10), so it may be that choosing a cyclic schedule for one loop and block schedules for the others will degrade performance.

With these choices, Table 6.3 shows the performance of the n-body solvers when they're run on one of our systems with no I/O. The solver used 400 particles for 1000 timesteps. The column labeled "Default Sched" gives times for the OpenMP reduced solver when all of the inner loops use the default schedule, which, on our system, is a block schedule. The column labeled "Forces Cyclic" gives times when the first phase of the forces computation uses a cyclic schedule and the other inner loops use the default schedule. The last column, labeled "All Cyclic," gives times when all of

Table 6.3 Run-Times of the n-Body Solvers Parallelized with OpenMP (times are in seconds)

Threads	Basic	Reduced Default Sched	Reduced Forces Cyclic	Reduced All Cyclic
1	7.71	3.90	3.90	3.90
2	3.87	2.94	1.98	2.01
4	1.95	1.73	1.01	1.08
8	0.99	0.95	0.54	0.61

the inner loops use a cyclic schedule. The run-times of the serial solvers differ from those of the single-threaded solvers by less than 1%, so we've omitted them from the table.

Notice that with more than one thread the reduced solver, using all default schedules, takes anywhere from 50 to 75% longer than the reduced solver with the cyclic forces computation. Using the cyclic schedule is clearly superior to the default schedule in this case, and any loss in time resulting from cache issues is more than made up for by the improved load balance for the computations.

For only two threads there is very little difference between the performance of the reduced solver with only the first forces loop cyclic and the reduced solver with all loops cyclic. However, as we increase the number of threads, the performance of the reduced solver that uses a cyclic schedule for all of the loops does start to degrade. In this particular case, when there are more threads, it appears that the overhead involved in changing distributions is less than the overhead incurred from false sharing.

Finally, notice that the basic solver takes about twice as long as the reduced solver with the cyclic scheduling of the forces computation. So if the extra memory is available, the reduced solver is clearly superior. However, the reduced solver increases the memory requirement for the storage of the forces by a factor of `thread_count`, so for very large numbers of particles, it may be impossible to use the reduced solver.

6.1.8 Parallelizing the solvers using pthreads

Parallelizing the two n-body solvers using Pthreads is very similar to parallelizing them using OpenMP. The differences are only in implementation details, so rather than repeating the discussion, we will point out some of the principal differences between the Pthreads and the OpenMP implementations. We will also note some of the more important similarities.

- By default local variables in Pthreads are private, so all shared variables are global in the Pthreads version.
- The principal data structures in the Pthreads version are identical to those in the OpenMP version: vectors are two-dimensional arrays of `doubles`, and the mass, position, and velocity of a single particle are stored in a struct. The forces are stored in an array of vectors.
- Startup for Pthreads is basically the same as the startup for OpenMP: the main thread gets the command-line arguments, and allocates and initializes the principal data structures.
- The main difference between the Pthreads and the OpenMP implementations is in the details of parallelizing the inner loops. Since Pthreads has nothing analogous to a `parallel for` directive, we must explicitly determine which values of the loop variables correspond to each thread's calculations. To facilitate this, we've written a function `Loop_schedule`, which determines
  - the initial value of the loop variable,
  - the final value of the loop variable, and
  - the increment for the loop variable.

The input to the function is
- the calling thread's rank,
- the number of threads,
- the total number of iterations, and
- an argument indicating whether the partitioning should be block or cyclic.

- Another difference between the Pthreads and the OpenMP versions has to do with barriers. Recall that the end of a `parallel for` directive in OpenMP has an implied barrier. As we've seen, this is important. For example, we don't want a thread to start updating its positions until all the forces have been calculated, because it could use an out-of-date force and another thread could use an out-of-date position. If we simply partition the loop iterations among the threads in the Pthreads version, there won't be a barrier at the end of an inner `for` loop and we'll have a race condition. Thus, we need to add explicit barriers after the inner loops when a race condition can arise. The Pthreads standard includes a barrier. However, some systems don't implement it, so we've defined a function that uses a Pthreads condition variable to implement a barrier. See Subsection 4.8.3 for details.

6.1.9 Parallelizing the basic solver using MPI

With our composite tasks corresponding to the individual particles, it's fairly straightforward to parallelize the basic algorithm using MPI. The only communication among the tasks occurs when we're computing the forces, and, in order to compute the forces, each task/particle needs the position and mass of every other particle. `MPI_Allgather` is expressly designed for this situation, since it collects on each process the same information from every other process. We've already noted that a block distribution will probably have the best performance, so we should use a block mapping of the particles to the processes.

In the shared-memory implementations, we collected most of the data associated with a single particle (mass, position, and velocity) into a single struct. However, if we use this data structure in the MPI implementation, we'll need to use a derived datatype in the call to `MPI_Allgather`, and communications with derived datatypes tend to be slower than communications with basic MPI types. Thus, it will make more sense to use individual arrays for the masses, positions, and velocities. We'll also need an array for storing the positions of all the particles. If each process has sufficient memory, then each of these can be a separate array. In fact, if memory isn't a problem, each process can store the entire array of masses, since these will never be updated and their values only need to be communicated during the initial setup.

On the other hand, if memory is short, there is an "in-place" option that can be used with some MPI collective communications. For our situation, suppose that the array `pos` can store the positions of all n particles. Further suppose that `vect_mpi_t` is an MPI datatype that stores two contiguous `doubles`. Also suppose that n is evenly divisible by `comm_sz` and `loc_n = n/comm_sz`. Then, if we store the local positions in a separate array, `loc_pos`, we can use the following call to collect all of the positions

on each process:

```
MPI_Allgather(loc_pos, loc_n, vect_mpi_t,
      pos, loc_n, vect_mpi_t, comm);
```

If we can't afford the extra storage for `loc_pos`, then we can have each process q store its local positions in the qth block of `pos`. That is, the local positions of each process should be stored in the appropriate block of each process' `pos` array:

```
Process 0:  pos[0], pos[1], . . . , pos[loc_n−1]
Process 1:  pos[loc_n], pos[loc_n+1], . . . , pos[loc_n + loc_n−1]
   . . .
Process q:  pos[q*loc_n], pos[q*loc_n+1], . . . , pos[q*loc_n +
     loc_n−1]
   . . .
```

With the `pos` array initialized this way on each process, we can use the following call to `MPI_Allgather`:

```
MPI_Allgather(MPI_IN_PLACE, loc_n, vect_mpi_t,
  pos, loc_n, vect_mpi_t, comm);
```

In this call, the first `loc_n` and `vect_mpi_t` arguments are ignored. However, it's not a bad idea to use arguments whose values correspond to the values that will be used, just to increase the readability of the program.

In the program we've written, we made the following choices with respect to the data structures:

- Each process stores the entire global array of particle masses.
- Each process only uses a single n-element array for the positions.
- Each process uses a pointer `loc_pos` that refers to the start of its block of `pos`. Thus, on process, 0 `local_pos = pos`, on process 1 `local_pos = pos + loc_n`, and, so on.

With these choices, we can implement the basic algorithm with the pseudocode shown in Program 6.2. Process 0 will read and broadcast the command line arguments. It will also read the input and print the results. In Line 1, it will need to distribute the input data. Therefore, `Get input data` might be implemented as follows:

```
if (my_rank == 0) {
   for each particle
      Read masses[particle], pos[particle], vel[particle];
}
MPI_Bcast(masses, n, MPI_DOUBLE, 0, comm);
MPI_Bcast(pos, n, vect_mpi_t, 0, comm);
MPI_Scatter(vel, loc_n, vect_mpi_t, loc_vel, loc_n, vect_mpi_t, 0,
    comm);
```

So process 0 reads all the initial conditions into three n-element arrays. Since we're storing all the masses on each process, we broadcast `masses`. Also, since each process

```
1  Get input data;
2  for each timestep {
3     if (timestep output)
4        Print positions and velocities of particles;
5     for each local particle loc_q
6        Compute total force on loc_q;
7     for each local particle loc_q
8        Compute position and velocity of loc_q;
9     Allgather local positions into global pos array;
10 }
11 Print positions and velocities of particles;
```

Program 6.2: Pseudocode for the MPI version of the basic *n*-body solver

will need the global array of positions for the first computation of forces in the main `for` loop, we just broadcast `pos`. However, velocities are only used locally for the updates to positions and velocities, so we scatter `vel`.

Notice that we gather the updated positions in Line 9 at the end of the body of the outer `for` loop of Program 6.2. This insures that the positions will be available for output in both Line 4 and Line 11. If we're printing the results for each timestep, this placement allows us to eliminate an expensive collective communication call: if we simply gathered the positions onto process 0 before output, we'd have to call `MPI_Allgather` before the computation of the forces. With this organization of the body of the outer `for` loop, we can implement the output with the following pseudocode:

```
Gather velocities onto process 0;
if (my_rank == 0) {
   Print timestep;
   for each particle
      Print pos[particle] and vel[particle]
}
```

6.1.10 Parallelizing the reduced solver using MPI

The "obvious" implementation of the reduced algorithm is likely to be extremely complicated. Before computing the forces, each process will need to gather a subset of the positions, and after the computation of the forces, each process will need to scatter some of the individual forces it has computed and add the forces it receives. Figure 6.6 shows the communications that would take place if we had three processes, six particles, and used a block partitioning of the particles among the processes. Not suprisingly, the communications are even more complex when we use a cyclic distribution (see Exercise 6.13). Certainly it would be possible to implement these communications. However, unless the implementation were *very* carefully done, it would probably be *very* slow.

Fortunately, there's a much simpler alternative that uses a communication structure that is sometimes called a **ring pass**. In a ring pass, we imagine the processes

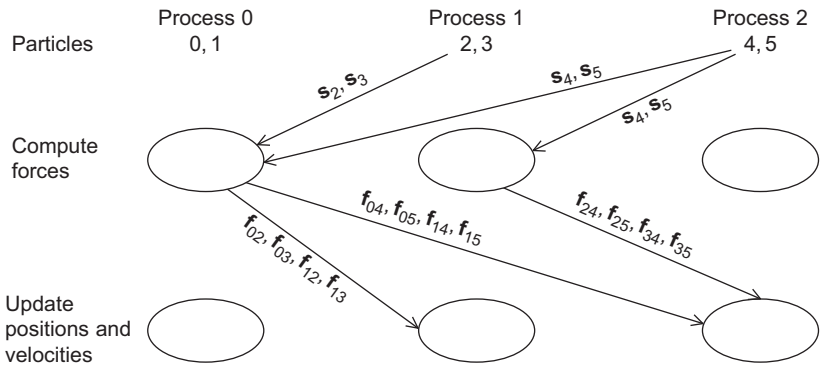

FIGURE 6.6

Communication in a possible MPI implementation of the reduced n-body solver

as being interconnected in a ring (see Figure 6.7). Process 0 communicates directly with processes 1 and `comm_sz` $-$ 1, process 1 communicates with processes 0 and 2, and so on. The communication in a ring pass takes place in phases, and during each phase each process sends data to its "lower-ranked" neighbor, and receives data from its "higher-ranked" neighbor. Thus, 0 will send to `comm_sz` $-$ 1 and receive from 1. 1 will send to 0 and receive from 2, and so on. In general, process q will send to process $(q - 1 + \texttt{comm\_sz}) \,\%\, \texttt{comm\_sz}$ and receive from process $(q + 1) \,\%\, \texttt{comm\_sz}$.

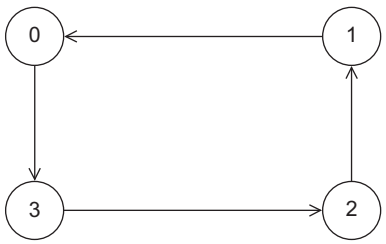

FIGURE 6.7

A ring of processes

By repeatedly sending and receiving data using this ring structure, we can arrange that each process has access to the positions of all the particles. During the first phase, each process will send the positions of its assigned particles to its "lower-ranked" neighbor and receive the positions of the particles assigned to its higher-ranked neighbor. During the next phase, each process will forward the positions it received in the first phase. This process continues through `comm_sz` $-$ 1 phases until each process has received the positions of all of the particles. Figure 6.8 shows the three phases if there are four processes and eight particles that have been cyclically distributed.

Of course, the virtue of the reduced algorithm is that we don't need to compute all of the inter-particle forces since $\mathbf{f}_{kq} = -\mathbf{f}_{qk}$, for every pair of particles q and k. To see

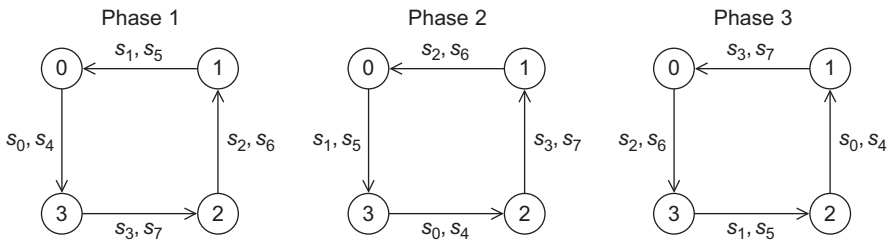

FIGURE 6.8

Ring pass of positions

how to exploit this, first observe that using the reduced algorithm, the interparticle forces can be divided into those that are *added* into and those that are subtracted from the total forces on the particle. For example, if we have six particles, then the reduced algorithm will compute the force on particle 3 as

$$\mathbf{F}_3 = -\mathbf{f}_{03} - \mathbf{f}_{13} - \mathbf{f}_{23} + \mathbf{f}_{34} + \mathbf{f}_{35}.$$

The key to understanding the ring pass computation of the forces is to observe that the interparticle forces that are *subtracted* are computed by another task/particle, while the forces that are *added* are computed by the owning task/particle. Thus, the computations of the interparticle forces on particle 3 are assigned as follows:

Force	$\mathbf{f}_{03}$	$\mathbf{f}_{13}$	$\mathbf{f}_{23}$	$\mathbf{f}_{34}$	$\mathbf{f}_{35}$
Task/Particle	0	1	2	3	3

So, suppose that for our ring pass, instead of simply passing `loc_n` = n/`comm_sz` positions, we also pass `loc_n` forces. Then in each phase, a process can

1. compute interparticle forces resulting from interaction between its assigned particles and the particles whose positions it has received, and
2. once an interparticle force has been computed, the process can add the force into a local array of forces corresponding to its particles, *and* it can subtract the interparticle force from the received array of forces.

See, for example, [15, 34] for further details and alternatives.

Let's take a look at how the computation would proceed when we have four particles, two processes, and we're using a cyclic distribution of the particles among the processes (see Table 6.4). We're calling the arrays that store the local positions and local forces `loc_pos` and `loc_forces`, respectively. These are not communicated among the processes. The arrays that are communicated among the processes are `tmp_pos` and `tmp_forces`.

Before the ring pass can begin, both arrays storing positions are initialized with the positions of the local particles, and the arrays storing the forces are set to 0. Before the ring pass begins, each process computes those forces that are due to interaction

Table 6.4 Computation of Forces in Ring Pass

Time	Variable	Process 0	Process 1
Start	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	0,0	0,0
	tmp_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	tmp_forces	0,0	0,0
After Comp of Forces	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{02}$, 0	$\mathbf{f}_{13}$, 0
	tmp_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	tmp_forces	$0, -\mathbf{f}_{02}$	$0, -\mathbf{f}_{13}$
After First Comm	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{02}$, 0	$\mathbf{f}_{13}$, 0
	tmp_pos	$\mathbf{s}_1, \mathbf{s}_3$	$\mathbf{s}_0, \mathbf{s}_2$
	tmp_forces	$0, -\mathbf{f}_{13}$	$0, -\mathbf{f}_{02}$
After Comp of Forces	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{01} + \mathbf{f}_{02} + \mathbf{f}_{03}, \mathbf{f}_{23}$	$\mathbf{f}_{12} + \mathbf{f}_{13}, 0$
	tmp_pos	$\mathbf{s}_1, \mathbf{s}_3$	$\mathbf{s}_0, \mathbf{s}_2$
	tmp_forces	$-\mathbf{f}_{01}, -\mathbf{f}_{03} - \mathbf{f}_{13} - \mathbf{f}_{23}$	$0, -\mathbf{f}_{02} - \mathbf{f}_{12}$
After Second Comm	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{01} + \mathbf{f}_{02} + \mathbf{f}_{03}, \mathbf{f}_{23}$	$\mathbf{f}_{12} + \mathbf{f}_{13}, 0$
	tmp_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	tmp_forces	$0, -\mathbf{f}_{02} - \mathbf{f}_{12}$	$-\mathbf{f}_{01}, -\mathbf{f}_{03} - \mathbf{f}_{13} - \mathbf{f}_{23}$
After Comp of Forces	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{01} + \mathbf{f}_{02} + \mathbf{f}_{03}, -\mathbf{f}_{02} - \mathbf{f}_{12} + \mathbf{f}_{23}$	$-\mathbf{f}_{01} + \mathbf{f}_{12} + \mathbf{f}_{13}, -\mathbf{f}_{03} - \mathbf{f}_{13} - \mathbf{f}_{23}$
	tmp_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	tmp_forces	$0, -\mathbf{f}_{02} - \mathbf{f}_{12}$	$-\mathbf{f}_{01}, -\mathbf{f}_{03} - \mathbf{f}_{13} - \mathbf{f}_{23}$

among its assigned particles. Process 0 computes $\mathbf{f}_{02}$ and process 1 computes $\mathbf{f}_{13}$. These values are added into the appropriate locations in `loc_forces` and subtracted from the appropriate locations in `tmp_forces`.

Now, the two processes exchange `tmp_pos` and `tmp_forces` and compute the forces due to interaction among their local particles and the received particles. In the reduced algorithm, the lower ranked task/particle carries out the computation. Process 0 computes $\mathbf{f}_{01}, \mathbf{f}_{03}$, and $\mathbf{f}_{23}$, while process 1 computes $\mathbf{f}_{12}$. As before, the newly computed forces are added into the appropriate locations in `loc_forces` and subtracted from the appropriate locations in `tmp_forces`.

To complete the algorithm, we need to exchange the `tmp` arrays one final time.[1] Once each process has received the updated `tmp_forces`, it can carry out a simple vector sum

```
loc_forces += tmp_forces
```

to complete the algorithm.

[1] Actually, we only need to exchange `tmp_forces` for the final communication.

```
1  source = (my_rank + 1) % comm_sz;
2  dest = (my_rank - 1 + comm_sz) % comm_sz;
3  Copy loc_pos into tmp_pos;
4  loc_forces = tmp_forces = 0;
5
6  Compute forces due to interactions among local particles;
7  for (phase = 1; phase < comm_sz; phase++) {
8     Send current tmp_pos and tmp_forces to dest;
9     Receive new tmp_pos and tmp_forces from source;
10    /* Owner of the positions and forces we're receiving */
11    owner = (my_rank + phase) % comm_sz;
12    Compute forces due to interactions among my particles
13       and owner's particles;
14 }
15 Send current tmp_pos and tmp_forces to dest;
16 Receive new tmp_pos and tmp_forces from source;
```

Program 6.3: Pseudocode for the MPI implementation of the reduced n-body solver

Thus, we can implement the computation of the forces in the reduced algorithm using a ring pass with the pseudocode shown in Program 6.3. Recall that using `MPI_Send` and `MPI_Recv` for the send-receive pairs in Lines 8 and 9 and 15 and 16 is *unsafe* in MPI parlance, since they can hang if the system doesn't provide sufficient buffering. In this setting, recall that MPI provides `MPI_Sendrecv` and `MPI_Sendrecv_replace`. Since we're using the same memory for both the outgoing and the incoming data, we can use `MPI_Sendrecv_replace`.

Also recall that the time it takes to start up a message is substantial. We can probably reduce the cost of the communication by using a single array to store both `tmp_pos` and `tmp_forces`. For example, we could allocate storage for an array `tmp_data` that can store $2 \times$ `loc_n` objects with type `vect_t` and use the first `loc_n` for `tmp_pos` and the last `loc_n` for `tmp_forces`. We can continue to use `tmp_pos` and `tmp_forces` by making these pointers to `tmp_data[0]` and `tmp_data[loc_n]`, respectively.

The principal difficulty in implementing the actual computation of the forces in Lines 12 and 13 lies in determining whether the current process should compute the force resulting from the interaction of a particle q assigned to it and a particle r whose position it has received. If we recall the reduced algorithm (Program 6.1), we see that task/particle q is responsible for computing $\mathbf{f}_{qr}$ if and only if $q < r$. However, the arrays `loc_pos` and `tmp_pos` (or a larger array containing `tmp_pos` and `tmp_forces`) use *local* subscripts, not global subscripts. That is, when we access an element of (say) `loc_pos`, the subscript we use will lie in the range $0, 1, \ldots,$ `loc_n` $- 1$, not $0, 1, \ldots, n-1$; so, if we try to implement the force interaction with the following pseudocode, we'll run into (at least) a couple of problems:

```
for (loc_part1 = 0; loc_part1 < loc_n-1; loc_part1++)
   for (loc_part2 = loc_part1+1; loc_part2 < loc_n; loc_part2++)
```

```
Compute_force(loc_pos[loc_part1], masses[loc_part1],
      tmp_pos[loc_part2], masses[loc_part2],
      loc_forces[loc_part1], tmp_forces[loc_part2]);
```

The first, and most obvious, is that `masses` is a global array and we're using local subscripts to access its elements. The second is that the relative sizes of `loc_part1` and `loc_part2` don't tell us whether we should compute the force due to their interaction. We need to use global subscripts to determine this. For example, if we have four particles and two processes, and the preceding code is being run by process 0, then when `loc_part1 = 0`, the inner loop will skip `loc_part2 = 0` and start with `loc_part2 = 1`; however, if we're using a cyclic distribution, `loc_part1 = 0` corresponds to global particle 0 and `loc_part2 = 0` corresponds to global particle 1, and we *should* compute the force resulting from interaction between these two particles.

Clearly, the problem is that we shouldn't be using local particle indexes, but rather we should be using *global* particle indexes. Thus, using a cyclic distribution of the particles, we could modify our code so that the loops also iterate through global particle indexes:

```
for (loc_part1 = 0, glb_part1 = my_rank;
     loc_part1 < loc_n-1;
     loc_part1++, glb_part1 += comm_sz)
  for (glb_part2 = First_index(glb_part1, my_rank, owner, comm_sz),
       loc_part2 = Global_to_local(glb_part2, owner, loc_n);
       loc_part2 < loc_n;
       loc_part2++, glb_part2 += comm_sz)
       Compute_force(loc_pos[loc_part1], masses[glb_part1],
             tmp_pos[loc_part2], masses[glb_part2],
             loc_forces[loc_part1], tmp_forces[loc_part2]);
```

The function `First_index` should determine a global index `glb_part2` with the following properties:

1. The particle `glb_part2` is assigned to the process with rank `owner`.
2. `glb_part1 < glb_part2 < glb_part1 + comm_sz`.

The function `Global_to_local` should convert a global particle index into a local particle index, and the function `Compute_force` should compute the force resulting from the interaction of two particles. We already know how to implement `Compute_force`. See Exercises 6.15 and 6.16 for the other two functions.

6.1.11 Performance of the MPI solvers

Table 6.5 shows the run-times of the two n-body solvers when they're run with 800 particles for 1000 timesteps on an Infiniband-connected cluster. All the timings were taken with one process per cluster node. The run-times of the serial solvers differed from the single-process MPI solvers by less than 1%, so we haven't included them.

Clearly, the performance of the reduced solver is much superior to the performance of the basic solver, although the basic solver achieves higher efficiencies.

Table 6.5 Performance of the MPI *n*-Body Solvers (times in seconds)

Processes	Basic	Reduced
1	17.30	8.68
2	8.65	4.45
4	4.35	2.30
8	2.20	1.26
16	1.13	0.78

Table 6.6 Run-Times for OpenMP and MPI *n*-Body Solvers (times in seconds)

Processes/ Threads	OpenMP		MPI	
	Basic	Reduced	Basic	Reduced
1	15.13	8.77	17.30	8.68
2	7.62	4.42	8.65	4.45
4	3.85	2.26	4.35	2.30

For example, the efficiency of the basic solver on 16 nodes is about 0.95, while the efficiency of the reduced solver on 16 nodes is only about 0.70.

A point to stress here is that the reduced MPI solver makes much more efficient use of memory than the basic MPI solver; the basic solver must provide storage for all n positions on each process, while the reduced solver only needs extra storage for $n/$`comm_sz` positions and $n/$`comm_sz` forces. Thus, the extra storage needed on each process for the basic solver is nearly `comm_sz`$/2$ times greater than the storage needed for the reduced solver. When n and `comm_sz` are very large, this factor can easily make the difference between being able to run a simulation only using the process' main memory and having to use secondary storage.

The nodes of the cluster on which we took the timings have four cores, so we can compare the performance of the OpenMP implementations with the performance of the MPI implementations (see Table 6.6). We see that the basic OpenMP solver is a good deal faster than the basic MPI solver. This isn't surprising since `MPI_Allgather` is such an expensive operation. Perhaps surprisingly, though, the reduced MPI solver is quite competitive with the reduced OpenMP solver.

Let's take a brief look at the amount of memory required by the MPI and OpenMP reduced solvers. Say that there are n particles and p threads or processes. Then each solver will allocate the same amount of storage for the local velocities and the local positions. The MPI solver allocates n `doubles` per process for the masses. It also allocates $4n/p$ `doubles` for the `tmp_pos` and `tmp_forces` arrays, so in addition to the

local velocities and positions, the MPI solver stores

$$n + 4n/p$$

`doubles` per process. The OpenMP solver allocates a total of $2pn + 2n$ `doubles` for the forces and n `doubles` for the masses, so in addition to the local velocities and positions, the OpenMP solver stores

$$3n/p + 2n$$

`doubles` per thread. Thus, the difference in the local storage required for the OpenMP version and the MPI version is

$$n - n/p$$

`doubles`. In other words, if n is large, the local storage required for the MPI version is substantially less than the local storage required for the OpenMP version. So, for a fixed number of processes or threads, we should be able to run much larger simulations with the MPI version than the OpenMP version. Of course, because of hardware considerations, we're likely to be able to use many more MPI processes than OpenMP threads, so the size of the largest possible MPI simulations should be *much* greater than the size of the largest possible OpenMP simulations. The MPI version of the reduced solver is much more scalable than any of the other versions, and the "ring pass" algorithm provides a genuine breakthrough in the design of n-body solvers.

6.2 TREE SEARCH

Many problems can be solved using a tree search. As a simple example, consider the traveling salesperson problem, or TSP. In TSP, a salesperson is given a list of cities she needs to visit and a cost for traveling between each pair of cities. Her problem is to visit each city once, returning to her hometown, and she must do this with the least possible cost. A route that starts in her hometown, visits each city once and returns to her hometown is called a *tour*; thus, the TSP is to find a minimum-cost tour.

Unfortunately, TSP is what's known as an **NP-complete** problem. From a practical standpoint, this means that there is no algorithm known for solving it that, in all cases, is significantly better than exhaustive search. Exhaustive search means examining all possible solutions to the problem and choosing the best. The number of possible solutions to TSP grows exponentially as the number of cities is increased. For example, if we add one additional city to an n-city problem, we'll increase the number of possible solutions by a factor of $n - 1$. Thus, although there are only six possible solutions to a four-city problem, there are $4 \times 6 = 24$ to a five-city problem, $5 \times 24 = 120$ to a six-city problem, $6 \times 120 = 720$ to a seven-city problem, and so on. In fact, a 100-city problem has far more possible solutions than the number of atoms in the universe!

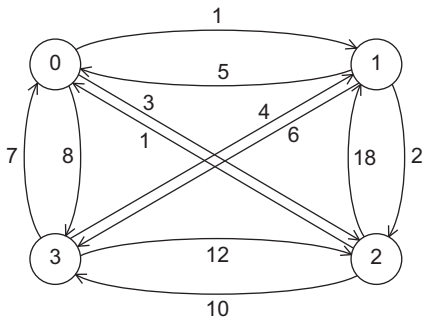

FIGURE 6.9

A four-city TSP

Furthermore, if we could find a solution to TSP that's significantly better in all cases than exhaustive search, then there are literally hundreds of other very hard problems for which we could find fast solutions. Not only is there no known solution to TSP that is better in all cases than exhaustive search, it's very unlikely that we'll find one.

So how can we solve TSP? There are a number of clever solutions. However, let's take a look at an especially simple one. It's a very simple form of tree search. The idea is that in searching for solutions, we build a *tree.* The leaves of the tree correspond to tours, and the other tree nodes correspond to "partial" tours—routes that have visited some, but not all, of the cities.

Each node of the tree has an associated cost, that is, the cost of the partial tour. We can use this to eliminate some nodes of the tree. Thus, we want to keep track of the cost of the best tour so far, and, if we find a partial tour or node of the tree that couldn't possibly lead to a less expensive complete tour, we shouldn't bother searching the children of that node (see Figures 6.9 and 6.10).

In Figure 6.9 we've represented a four-city TSP as a labeled, directed graph. A **graph** (not to be confused with a graph in calculus) is a collection of vertices and edges or line segments joining pairs of vertices. In a **directed graph** or **digraph**, the edges are oriented—one end of each edge is the tail, and the other is the head. A graph or digraph is **labeled** if the vertices and/or edges have labels. In our example, the vertices of the digraph correspond to the cities in an instance of the TSP, the edges correspond to routes between the cities, and the labels on the edges correspond to the costs of the routes. For example, there's a cost of 1 to go from city 0 to city 1 and a cost of 5 to go from city 1 to city 0.

If we choose vertex 0 as the salesperson's home city, then the initial partial tour consists only of vertex 0, and since we've gone nowhere, it's cost is 0. Thus, the root of the tree in Figure 6.10 has the partial tour consisting only of the vertex 0 with cost 0. From 0 we can first visit 1, 2, or 3, giving us three two-city partial tours with costs 1, 3, and 8, respectively. In Figure 6.10 this gives us three children of the root. Continuing, we get six three-city partial tours, and since there are

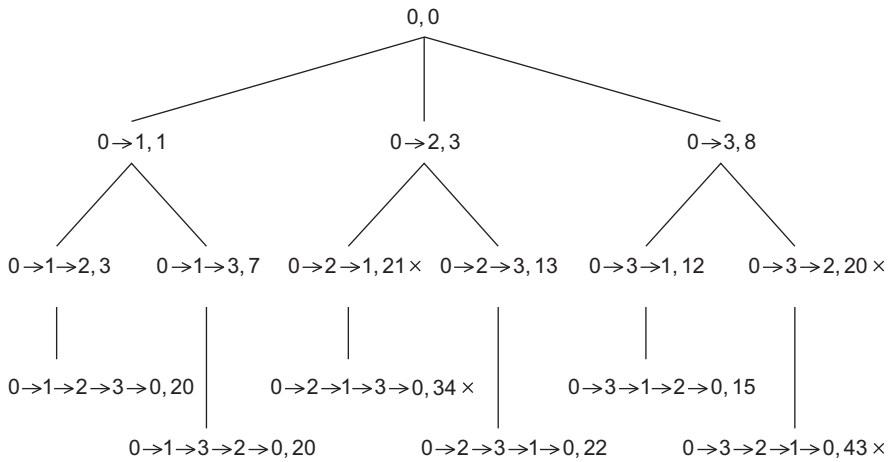

FIGURE 6.10

Search tree for four-city TSP

only four cities, once we've chosen three of the cities, we know what the complete tour is.

Now, to find a least-cost tour, we should search the tree. There are many ways to do this, but one of the most commonly used is called **depth-first search**. In depth-first search, we probe as deeply as we can into the tree. After we've either reached a leaf or found a tree node that can't possibly lead to a least-cost tour, we back up to the deepest "ancestor" tree node with unvisited children, and probe one of its children as deeply as possible.

In our example, we'll start at the root, and branch left until we reach the leaf labeled

$$0 \to 1 \to 2 \to 3 \to 0, \text{ Cost } 20.$$

Then we back up to the tree node labeled $0 \to 1$, since it is the deepest ancestor node with unvisited children, and we'll branch down to get to the leaf labeled

$$0 \to 1 \to 3 \to 2 \to 0, \text{ Cost } 20.$$

Continuing, we'll back up to the root and branch down to the node labeled $0 \to 2$. When we visit its child, labeled

$$0 \to 2 \to 1, \text{ Cost } 21,$$

we'll go no further in this subtree, since we've already found a complete tour with cost less than 21. We'll back up to $0 \to 2$ and branch down to its remaining unvisited child. Continuing in this fashion, we eventually find the least-cost tour

$$0 \to 3 \to 1 \to 2 \to 0, \text{ Cost } 15.$$

6.2.1 Recursive depth-first search

Using depth-first search we can systematically visit each node of the tree that could possibly lead to a least-cost solution. The simplest formulation of depth-first search uses recursion (see Program 6.4). Later on it will be useful to have a definite order in which the cities are visited in the `for` loop in Lines 8 to 13, so we'll assume that the cities are visited in order of increasing index, from city 1 to city $n - 1$.

The algorithm makes use of several global variables:

- `n`: the total number of cities in the problem
- `digraph`: a data structure representing the input digraph
- `hometown`: a data structure representing vertex or city 0, the salesperson's hometown
- `best_tour`: a data structure representing the best tour so far

The function `City_count` examines the partial tour `tour` to see if there are n cities on the partial tour. If there are, we know that we simply need to return to the hometown to complete the tour, and we can check to see if the complete tour has a lower cost than the current "best tour" by calling `Best_tour`. If it does, we can replace the current best tour with this tour by calling the function `Update_best_tour`. Note that before the first call to `Depth_first_search`, the `best_tour` variable should be initialized so that its cost is greater than the cost of any possible least-cost tour.

If the partial tour `tour` hasn't visited n cities, we can continue branching down in the tree by "expanding the current node," in other words, by trying to visit other cities from the city last visited in the partial tour. To do this we simply loop through the cities. The function `Feasible` checks to see if the city or vertex has already been visited, and, if not, whether it can possibly lead to a least-cost tour. If the city is feasible, we add it to the tour, and recursively call `Depth_first_search`. When

```
1  void Depth_first_search(tour_t tour) {
2     city_t city;
3
4     if (City_count(tour) == n) {
5        if (Best_tour(tour))
6           Update_best_tour(tour);
7     } else {
8        for each neighboring city
9           if (Feasible(tour, city)) {
10             Add_city(tour, city);
11             Depth_first_search(tour);
12             Remove_last_city(tour);
13          }
14    }
15 }  /* Depth_first_search */
```

Program 6.4: Pseudocode for a recursive solution to TSP using depth-first search

we return from `Depth_first_search`, we remove the city from the tour, since it shouldn't be included in the tour used in subsequent recursive calls.

6.2.2 Nonrecursive depth-first search

Since function calls are expensive, recursion can be slow. It also has the disadvantage that at any given instant of time only the current tree node is accessible. This could be a problem when we try to parallelize tree search by dividing tree nodes among the threads or processes.

It is possible to write a nonrecursive depth-first search. The basic idea is modeled on recursive implementation. Recall that recursive function calls can be implemented by pushing the current state of the recursive function onto the run-time stack. Thus, we can try to eliminate recursion by pushing necessary data on our own stack before branching deeper into the tree, and when we need to go back up the tree—either because we've reached a leaf or because we've found a node that can't lead to a better solution—we can pop the stack.

This outline leads to the implementation of iterative depth-first search shown in Program 6.5. In this version, a stack record consists of a single city, the city that will be added to the tour when its record is popped. In the recursive version we continue to make recursive calls until we've visited every node of the tree that corresponds to a feasible partial tour. At this point, the stack won't have any more activation records for calls to `Depth_first_search`, and we'll return to the function that made the

```
1  for (city = n-1; city >= 1; city--)
2     Push(stack, city);
3  while (!Empty(stack)) {
4     city = Pop(stack);
5     if (city == NO_CITY) // End of child list, back up
6        Remove_last_city(curr_tour);
7     else {
8        Add_city(curr_tour, city);
9        if (City_count(curr_tour) == n) {
10          if (Best_tour(curr_tour))
11             Update_best_tour(curr_tour);
12          Remove_last_city(curr_tour);
13       } else {
14          Push(stack, NO_CITY);
15          for (nbr = n-1; nbr >= 1; nbr--)
16             if (Feasible(curr_tour, nbr))
17                Push(stack, nbr);
18       }
19    }  /* if Feasible */
20 }  /* while !Empty */
```

Program 6.5: Pseudocode for an implementation of a depth-first solution to TSP that doesn't use recursion

original call to `Depth_first_search`. The main control structure in our iterative version is the `while` loop extending from Line 3 to Line 20, and the loop termination condition is that our stack is empty. As long as the search needs to continue, we need to make sure the stack is nonempty, and, in the first two lines, we add each of the non-hometown cities. Note that this loop visits the cities in decreasing order, from $n - 1$ down to 1. This is because of the order created by the stack, whereby the stack pops the top cities first. By reversing the order, we can insure that the cities are visited in the same order as the recursive function.

Also notice that in Line 5 we check whether the city we've popped is the constant `NO_CITY`. This constant is used so that we can tell when we've visited all of the children of a tree node; if we didn't use it, we wouldn't be able to tell when to back up in the tree. Thus, before pushing all of the children of a node (Lines 15–17), we push the `NO_CITY` marker.

An alternative to this iterative version uses partial tours as stack records (see Program 6.6). This gives code that is closer to the recursive function. However, it also results in a slower version, since it's necessary for the function that pushes onto the stack to create a copy of the tour before actually pushing it on to the stack. To emphasize this point, we've called the function `Push_copy`. (What happens if we simply push a pointer to the current tour onto the stack?) The extra memory required will probably not be a problem. However, allocating storage for a new tour and copying the existing tour is time-consuming. To some degree we can mitigate these costs by saving freed tours in our own data structure, and when a freed tour is available we can use it in the `Push_copy` function instead of calling `malloc`.

On the other hand, this version has the virtue that the stack is more or less independent of the other data structures. Since entire tours are stored, multiple threads or processes can "help themselves" to tours, and, if this is done reasonably carefully,

```
1  Push_copy(stack, tour);  // Tour that visits only the hometown
2  while (!Empty(stack)) {
3     curr_tour = Pop(stack);
4     if (City_count(curr_tour) == n) {
5        if (Best_tour(curr_tour))
6           Update_best_tour(curr_tour);
7     } else {
8        for (nbr = n-1; nbr >= 1; nbr--)
9           if (Feasible(curr_tour, nbr)) {
10             Add_city(curr_tour, nbr);
11             Push_copy(stack, curr_tour);
12             Remove_last_city(curr_tour);
13          }
14    }
15    Free_tour(curr_tour);
16 }
```

Program 6.6: Pseudocode for a second solution to TSP that doesn't use recursion

it won't destroy the correctness of the program. With the original iterative version, a stack record is just a city and it doesn't provide enough information by itself to show where we are in the tree.

6.2.3 Data structures for the serial implementations

Our principal data structures are the tour, the digraph, and, in the iterative implementations, the stack. The tour and the stack are essentially list structures. In problems that we're likely to be able to tackle, the number of cities is going to be small—certainly less than 100—so there's no great advantage to using a linked list to represent the tours and we've used an array that can store $n+1$ cities. We repeatedly need both the number of cities in the partial tour and the cost of the partial tour. Therefore, rather than just using an array for the tour data structure and recomputing these values, we use a struct with three members: the array storing the cities, the number of cities, and the cost of the partial tour.

To improve the readability and the performance of the code, we can use preprocessor macros to access the members of the struct. However, since macros can be a nightmare to debug, it's a good idea to write "accessor" functions for use during initial development. When the program with accessor functions is working, they can be replaced with macros. As an example, we might start with the function

```
/* Find the ith city on the partial tour */
int Tour_city(tour_t tour, int i) {
   return tour->cities[i];
}  /* Tour_city */
```

When the program is working, we could replace this with the macro

```
/* Find the ith city on the partial tour */
#define Tour_city(tour, i) (tour->cities[i])
```

The stack in the original iterative version is just a list of cities or `int`s. Furthermore, since there can't be more than $n^2/2$ records on the stack (see Exercise 6.17) at any one time, and n is likely to be small, we can just use an array, and like the tour data structure, we can store the number of elements on the stack. Thus, for example, `Push` can be implemented with

```
void Push(my_stack_t stack, int city) {
   int loc = stack->list_sz;
   stack->list[loc] = city;
   stack->list_sz++;
}  /* Push */
```

In the second iterative version, the version that stores entire tours in the stack, we can probably still use an array to store the tours on the stack. Now the push function will look something like this:

```
void Push_copy(my_stack_t stack, tour_t tour) {
   int loc = stack->list_sz;
```

```
    tour_t tmp = Alloc_tour();
    Copy_tour(tour, tmp);
    stack->list[loc] = tmp;
    stack->list_sz++;
}  /* Push */
```

Once again, element access for the stack can be implemented with macros.

There are many possible representations for digraphs. When the digraph has relatively few edges, list representations are preferred. However, in our setting, if vertex i is different from vertex j, there are directed, weighted edges from i to j and from j to i, so we need to store a weight for each ordered pair of distinct vertices i and j. Thus, in our setting, an **adjacency matrix** is almost certainly preferable to a list structure. This is an $n \times n$ matrix, in which the weight of the edge from vertex i to vertex j can be the entry in the ith row and jth column of the matrix. We can access this weight directly, without having to traverse a list. The diagonal elements (row i and column i) aren't used, and we'll set them to 0.

6.2.4 Performance of the serial implementations

The run-times of the three serial implementations are shown in Table 6.7. The input digraph contained 15 vertices (including the hometown), and all three algorithms visited approximately 95,000,000 tree nodes. The first iterative version is less than 5% faster than the recursive version, and the second iterative version is about 8% slower than the recursive version. As expected, the first iterative solution eliminates some of the overhead due to repeated function calls, while the second iterative solution is slower because of the repeated copying of tour data structures. However, as we'll see, the second iterative solution is relatively easy to parallelize, so we'll be using it as the basis for the parallel versions of tree search.

6.2.5 Parallelizing tree search

Let's take a look at parallelizing tree search. The tree structure suggests that we identify tasks with tree nodes. If we do this, the tasks will communicate down the tree edges: a parent will communicate a new partial tour to a child, but a child, except for terminating, doesn't communicate directly with a parent.

We also need to take into consideration the updating and use of the best tour. Each task examines the best tour to determine whether the current partial tour is feasible or the current complete tour has lower cost. If a leaf task determines its tour is a better tour, then it will also update the best tour. Although all of the actual computation can

Table 6.7 Run-Times of the Three Serial Implementations of Tree Search (times in seconds)

Recursive	First Iterative	Second Iterative
30.5	29.2	32.9

be considered to be carried out by the tree node tasks, we need to keep in mind that the best tour data structure requires additional communication that is not explicit in the tree edges. Thus, it's convenient to add an additional task that corresponds to the best tour. It "sends" data to every tree node task, and receives data from some of the leaves. This latter view is convenient for shared-memory, but not so convenient for distributed-memory.

A natural way to agglomerate and map the tasks is to assign a subtree to each thread or process, and have each thread/process carry out all the tasks in its subtree. For example, if we have three threads or processes, as shown earlier in Figure 6.10, we might map the subtree rooted at $0 \rightarrow 1$ to thread/process 0, the subtree rooted at $0 \rightarrow 2$ to thread/process 1, and the subtree rooted at $0 \rightarrow 3$ to thread/process 2.

Mapping details

There are many possible algorithms for identifying which subtrees we assign to the processes or threads. For example, one thread or process could run the last version of serial depth-first search until the stack stores one partial tour for each thread or process. Then it could assign one tour to each thread or process. The problem with depth-first search is that we expect a subtree whose root is deeper in the tree to require less work than a subtree whose root is higher up in the tree, so we would probably get better load balance if we used something like **breadth-first search** to identify the subtrees.

As the name suggests, breadth-first search searches as widely as possible in the tree before going deeper. So if, for example, we carry out a breadth-first search until we reach a level of the tree that has at least `thread_count` or `comm_sz` nodes, we can then divide the nodes at this level among the threads or processes. See Exercise 6.18 for implementation details.

The best tour data structure

On a shared-memory system, the best tour data structure can be shared. In this setting, the `Feasible` function can simply examine the data structure. However, updates to the best tour will cause a race condition, and we'll need some sort of locking to prevent errors. We'll discuss this in more detail when we implement the parallel version.

In the case of a distributed-memory system, there are a couple of choices that we need to make about the best tour. The simplest option would be to have the processes operate independently of each other until they have completed searching their subtrees. In this setting, each process would store its own *local* best tour. This local best tour would be used by the process in `Feasible` and updated by the process each time it calls `Update_best_tour`. When all the processes have finished searching, they can perform a global reduction to find the tour with the *global* least cost.

This approach has the virtue of simplicity, but it also suffers from the problem that it's entirely possible for a process to spend most or all of its time searching through partial tours that couldn't possibly lead to a global best tour. Thus, we should

probably try using an approach that makes the current global best tour available to all the processes. We'll take a look at details when we discuss the MPI implementation.

Dynamic mapping of tasks

A second issue we should consider is the problem of load imbalance. Although the use of breadth-first search ensures that all of our subtrees have approximately the same number of nodes, there is no guarantee that they all have the same amount of work. It's entirely possible that one process or thread will have a subtree consisting of very expensive tours, and, as a consequence, it won't need to search very deeply into its assigned subtree. However, with our current, *static* mapping of tasks to threads/processes, this one thread or process will simply have to wait until the other threads/processes are done.

An alternative is to implement a **dynamic** mapping scheme. In a dynamic scheme, if one thread/process runs out of useful work, it can obtain additional work from another thread/process. In our final implementation of serial depth-first search, each stack record contains a partial tour. With this data structure a thread or process can give additional work to another thread/process by dividing the contents of its stack. This might at first seem to have the potential for causing problems with the program's correctness, since if we give part of one thread's or one process' stack to another, there's a good chance that the order in which the tree nodes will be visited will be changed.

However, we're already going to do this; when we assign different subtrees to different threads/processes, the order in which the tree nodes are visited is no longer the serial depth-first ordering. In fact, in principle, there's no reason to visit any node before any other node as long as we make sure we visit "ancestors" before "descendants." But this isn't a problem since a partial tour isn't added to the stack until after all its ancestors have been visited. For example, in Figure 6.10 the node consisting of the tour $0 \rightarrow 2 \rightarrow 1$ will be pushed onto the stack when the node consisting of the tour $0 \rightarrow 2$ is the currently active node, and consequently the two nodes won't be on the stack simultaneously. Similarly, the parent of $0 \rightarrow 2$, the root of the tree, 0, is no longer on the stack when $0 \rightarrow 2$ is visited.

A second alternative for dynamic load balancing—at least in the case of shared memory—would be to have a shared stack. However, we couldn't simply dispense with the local stacks. If a thread needed to access the shared stack every time it pushed or popped, there would be a tremendous amount of contention for the shared stack and the performance of the program would probably be worse than a serial program. This is exactly what happened when we parallelized the reduced n-body solver with mutexes/locks protecting the calculations of the total forces on the various particles. If every call to `Push` or `Pop` formed a critical section, our program would grind to nearly a complete halt. Thus, we would want to retain local stacks for each thread, with only occasional accesses to the shared stack. We won't pursue this alternative. See Programming Assignment 6.7 for further details.

6.2.6 A static parallelization of tree search using pthreads

In our static parallelization, a single thread uses breadth-first search to generate enough partial tours so that each thread gets at least one partial tour. Then each thread takes its partial tours and runs iterative tree search on them. We can use the pseudocode shown in Program 6.7 on each thread. Note that most of the function calls—for example, `Best_tour`, `Feasible`, `Add_city`—need to access the adjacency matrix representing the digraph, so all the threads will need to access the digraph. However, since these are only *read* accesses, this won't result in a race condition or contention among the threads.

There are only four potential differences between this pseudocode and the pseudocode we used for the second iterative serial implementation:

- The use of `my_stack` instead of `stack`; since each thread has its own, private stack, we use `my_stack` as the identifier for the stack object instead of `stack`.
- Initialization of the stack.
- Implementation of the `Best_tour` function.
- Implementation of the `Update_best_tour` function.

In the serial implementation, the stack is initialized by pushing the partial tour consisting only of the hometown onto the stack. In the parallel version we need to generate at least `thread_count` partial tours to distribute among the threads. As we discussed earlier, we can use breadth-first search to generate a list of at least `thread_count` tours by having a single thread search the tree until it reaches a level with at least `thread_count` tours. (Note that this implies that the number of threads should be less than $(n-1)!$, which shouldn't be a problem). Then the threads can

```
Partition_tree(my_rank, my_stack);

while (!Empty(my_stack)) {
   curr_tour = Pop(my_stack);
   if (City_count(curr_tour) == n) {
      if (Best_tour(curr_tour)) Update_best_tour(curr_tour);
   } else {
      for (city = n-1; city >= 1; city--)
         if (Feasible(curr_tour, city)) {
            Add_city(curr_tour, city);
            Push_copy(my_stack, curr_tour);
            Remove_last_city(curr_tour)
         }
   }
   Free_tour(curr_tour);
}
```

Program 6.7: Pseudocode for a Pthreads implementation of a statically parallelized solution to TSP

use a block partition to divide these tours among themselves and push them onto their private stacks. Exercise 6.18 looks into the details.

To implement the `Best_tour` function, a thread should compare the cost of its current tour with the cost of the global best tour. Since multiple threads may be simultaneously accessing the global best cost, it might at first seem that there will be a race condition. However, the `Best_tour` function only *reads* the global best cost, so there won't be any conflict with threads that are also checking the best cost. If a thread is updating the global best cost, then a thread that is just checking it will either read the old value or the new, updated value. While we would prefer that it get the new value, we can't insure this without using some very costly locking strategy. For example, threads wanting to execute `Best_tour` or `Update_best_tour` could wait on a single mutex. This would insure that no thread is updating while another thread is only checking, but would have the unfortunate side effect that only one thread could check the best cost at a time. We could improve on this by using a read-write lock, but this would have the side effect that the readers—the threads calling `Best_tour`—would all block while a thread updated the best tour. In principle, this doesn't sound too bad, but recall that in practice read-write locks can be quite slow. So it seems pretty clear that the "no contention" solution of possibly getting a best tour cost that's out-of-date is probably better, as the next time the thread calls `Best_tour`, it will get the updated value of the best tour cost.

On the other hand, we call `Update_best_tour` with the intention of *writing* to the best tour structure, and this clearly can cause a race condition if two threads call it simultaneously. To avoid this problem, we can protect the body of the `Update_best_tour` function with a mutex. This isn't enough, however; between the time a thread completes the test in `Best_tour` and the time it obtains the lock in `Update_best_tour`, another thread may have obtained the lock and updated the best tour cost, which now may be less than the best tour cost that the first thread found in `Best_tour`. Thus, correct pseudocode for `Update_best_tour` should look something like this:

```
pthread_mutex_lock(best_tour_mutex);
/* We've already checked Best_tour, but we need to check it
    again */
if (Best_tour(tour))
   Replace old best tour with tour;
pthread_mutex_unlock(best_tour_mutex).
```

This may seem wasteful, but if updates to the best tour are infrequent, then most of the time `Best_tour` will return `false` and it will only be rarely necessary to make the "double" call.

6.2.7 A dynamic parallelization of tree search using pthreads

If the initial distribution of subtrees doesn't do a good job of distributing the work among the threads, the static parallelization provides no means of redistributing work. The threads with "small" subtrees will finish early, while the threads with large subtrees will continue to work. It's not difficult to imagine that one thread gets the lion's

share of the work because the edges in its initial tours are very cheap, while the edges in the other threads' initial tours are very expensive. To address this issue, we can try to dynamically redistribute the work as the computation proceeds.

To do this, we can replace the test `!Empty(my_stack)` controlling execution of the `while` loop with more complex code. The basic idea is that when a thread runs out of work—that is, `!Empty(my_stack)` becomes false—instead of immediately exiting the `while` loop, the thread waits to see if another thread can provide more work. On the other hand, if a thread that still has work in its stack finds that there is at least one thread without work, and its stack has at least two tours, it can "split" its stack and provide work for one of the threads.

Pthreads condition variables provide a natural way to implement this. When a thread runs out of work it can call `pthread_cond_wait` and go to sleep. When a thread with work finds that there is at least one thread waiting for work, after splitting its stack, it can call `pthread_cond_signal`. When a thread is awakened it can take one of the halves of the split stack and return to work.

This idea can be extended to handle termination. If we maintain a count of the number of threads that are in `pthread_cond_wait`, then when a thread whose stack is empty finds that `thread_count` − 1 threads are already waiting, it can call `pthread_cond_broadcast` and as the threads awaken, they'll see that all the threads have run out of work and quit.

Termination

Thus, we can use the pseudocode shown in Program 6.8 for a `Terminated` function that would be used instead of `Empty` for the `while` loop implementing tree search.

There are several details that we should look at more closely. Notice that the code executed by a thread before it splits its stack is fairly complicated. In Lines 1–2 the thread

- checks that it has at least two tours in its stack,
- checks that there are threads waiting, and
- checks whether the `new_stack` variable is `NULL`.

The reason for the check that the thread has enough work should be clear: if there are fewer than two records on the thread's stack, "splitting" the stack will either do nothing or result in the active thread's trading places with one of the waiting threads.

It should also be clear that there's no point in splitting the stack if there aren't any threads waiting for work. Finally, if some thread has already split its stack, but a waiting thread hasn't retrieved the new stack, that is, `new_stack != NULL`, then it would be disastrous to split a stack and overwrite the existing new stack. Note that this makes it essential that after a thread retrieves `new_stack` by, say, copying `new_stack` into its private `my_stack` variable, the thread must set `new_stack` to `NULL`.

If all three of these conditions hold, then we can try splitting our stack. We can acquire the mutex that protects access to the objects controlling termination (`threads_in_cond_wait`, `new_stack`, and the condition variable). However, the condition

```
threads_in_cond_wait > 0 && new_stack == NULL
```

```
1  if (my_stack_size >= 2 && threads_in_cond_wait > 0 &&
2        new_stack == NULL) {
3     lock term_mutex;
4     if (threads_in_cond_wait > 0 && new_stack == NULL) {
5        Split my_stack creating new_stack;
6        pthread_cond_signal(&term_cond_var);
7     }
8     unlock term_mutex;
9     return 0;  /* Terminated = false; don't quit */
10 } else if (!Empty(my_stack)) /* Keep working */
11    return 0;  /* Terminated = false; don't quit */
12 } else { /* My stack is empty */
13    lock term_mutex;
14    if (threads_in_cond_wait == thread_count-1)
15                                        /* Last thread running */
16       threads_in_cond_wait++;
17       pthread_cond_broadcast(&term_cond_var);
18       unlock term_mutex;
19       return 1;  /* Terminated = true; quit */
20    } else { /* Other threads still working, wait for work */
21       threads_in_cond_wait++;
22       while (pthread_cond_wait(&term_cond_var, &term_mutex) != 0);
23       /* We've been awakened */
24       if (threads_in_cond_wait < thread_count) { /* We got work */
25          my_stack = new_stack;
26          new_stack = NULL;
27          threads_in_cond_wait--;
28          unlock term_mutex;
29          return 0;  /* Terminated = false */
30       } else {  /* All threads done */
31          unlock term_mutex;
32          return 1;  /* Terminated = true; quit */
33       }
34    }  /* else wait for work */
35 }  /* else my_stack is empty */
```

Program 6.8: Pseudocode for Pthreads `Terminated` function

can change between the time we start waiting for the mutex and the time we actually acquire it, so as with `Update_best_tour`, we need to confirm that this condition is still true after acquiring the mutex (Line 4). Once we've verified that these conditions still hold, we can split the stack, awaken one of the waiting threads, unlock the mutex, and return to work.

If the test in Lines 1 and 2 is false, we can check to see if we have any work at all—that is, our stack is nonempty. If it is, we return to work. If it isn't, we'll start the termination sequence by waiting for and acquiring the termination mutex in Line 13. Once we've acquired the mutex, there are two possibilities:

- We're the last thread to enter the termination sequence, that is, `threads_in_cond_wait == thread_count-1`.
- Other threads are still working.

In the first case, we know that since all the other threads have run out of work, and we have also run out of work, the tree search should terminate. We therefore signal all the other threads by calling `pthread_cond_broadcast` and returning true. Before executing the broadcast, we increment `threads_in_cond_wait`, even though the broadcast is telling all the threads to return from the condition wait. The reason is that `threads_in_cond_wait` is serving a dual purpose: When it's less than `thread_count`, it tells us how many threads are waiting. However, when it's equal to `thread_count`, it tells us that all the threads are out of work, and it's time to quit.

In the second case—other threads are still working—we call `pthread_cond_wait` (Line 22) and wait to be awakened. Recall that it's possible that a thread could be awakened by some event other than a call to `pthread_cond_signal` or `pthread_cond_broadcast`. So, as usual, we put the call to `pthread_cond_wait` in a `while` loop, which will immediately call `pthread_cond_wait` again if some other event (return value not 0) awakens the thread.

Once we've been awakened, there are also two cases to consider:

- `threads_in_cond_wait < thread_count`
- `threads_in_cond_wait == thread_count`

In the first case, we know that some other thread has split its stack and created more work. We therefore copy the newly created stack into our private stack, set the `new_stack` variable to `NULL`, and decrement `threads_in_cond_wait` (i.e., Lines 25–27). Recall that when a thread returns from a condition wait, it obtains the mutex associated with the condition variable, so before returning, we also unlock the mutex (i.e., Line 28). In the second case, there's no work left, so we unlock the mutex and return true.

In the actual code, we found it convenient to group the termination variables together into a single struct. Thus, we defined something like

```
typedef struct {
   my_stack_t new_stack;
   int threads_in_cond_wait;
   pthread_cond_t term_cond_var;
   pthread_mutex_t term_mutex;
} term_struct;
typedef term_struct* term_t;

term_t term;  // global variable
```

and we defined a couple of functions, one for initializing the `term` variable and one for destroying/freeing the variable and its members.

Before discussing the function that splits the stack, note that it's possible that a thread with work can spend a lot of time waiting for `term_mutex` before being able

to split its stack. Other threads may be either trying to split their stacks, or preparing for the condition wait. If we suspect that this is a problem, Pthreads provides a nonblocking alternative to `pthread_mutex_lock` called `pthread_mutex_trylock`:

```
int pthread_mutex_trylock(
          pthread_mutex_t*   mutex_p   /* in/out */);
```

This function attempts to acquire `mutex_p`. However, if it's locked, instead of waiting, it returns immediately. The return value will be zero if the calling thread has successfully acquired the mutex, and nonzero if it hasn't. As an alternative to waiting on the mutex before splitting its stack, a thread can call `pthread_mutex_trylock`. If it acquires `term_mutex`, it can proceed as before. If not, it can just return. Presumably on a subsequent call it can successfully acquire the mutex.

Splitting the stack

Since our goal is to balance the load among the threads, we would like to insure that the amount of work in the new stack is roughly the same as the amount remaining in the original stack. We have no way of knowing in advance of searching the subtree rooted at a partial tour how much work is actually associated with the partial tour, so we'll never be able to guarantee an equal division of work. However, we can use the same strategy that we used in our original assignment of subtrees to threads: that the subtrees rooted at two partial tours with the same number of cities have identical structures. Since on average two partial tours with the same number of cities are equally likely to lead to a "good" tour (and hence more work), we can try splitting the stack by assigning the tours on the stack on the basis of their numbers of edges. The tour with the least number of edges remains on the original stack, the tour with the next to the least number of edges goes to the new stack, the tour with the next number of edges remains on the original, and so on.

This is fairly simple to implement, since the tours on the stack have an increasing number of edges. That is, as we proceed from the bottom of the stack to the top of the stack, the number of edges in the tours increases. This is because when we push a new partial tour with k edges onto the stack, the tour that's immediately "beneath" it on the stack either has k edges or $k - 1$ edges. We can implement the split by starting at the bottom of the stack, and alternately leaving partial tours on the old stack and pushing partial tours onto the new stack, so tour 0 will stay on the old stack, tour 1 will go to the new stack, tour 2 will stay on the old stack, and so on. If the stack is implemented as an array of tours, this scheme will require that the old stack be "compressed" so that the gaps left by removing alternate tours are eliminated. If the stack is implemented as a linked list of tours, compression won't be necessary.

This scheme can be further refined by observing that partial tours with lots of cities won't provide much work, since the subtrees that are rooted at these trees are very small. We could add a "cutoff size" and not reassign a tour unless its number of cities was less than the cutoff. In a shared-memory setting with an array-based stack, reassigning a tour when a stack is split won't increase the cost of the split, since the tour (which is a pointer) will either have to be copied to the new stack or a new

Table 6.8 Run-Times of Pthreads Tree-Search Programs (times in seconds)

	First Problem				Second Problem			
Threads	*Serial*	*Static*	*Dynamic*		*Serial*	*Static*	*Dynamic*	
1	32.9	32.7	34.7	(0)	26.0	25.8	27.5	(0)
2		27.9	28.9	(7)		25.8	19.2	(6)
4		25.7	25.9	(47)		25.8	9.3	(49)
8		23.8	22.4	(180)		24.0	5.7	(256)

location in the old stack. We'll defer exploration of this alternative to Programming Assignment 6.6.

6.2.8 Evaluating the Pthreads tree-search programs

Table 6.8 shows the performance of the two Pthreads programs on two fifteen-city problems. The "Serial" column gives the run-time of the second iterative solution—the solution that pushes a copy of each new tour onto the stack. For reference, the first problem in Table 6.8 is the same as the problem the three serial solutions were tested with in Table 6.7, and both the Pthreads and serial implementations were tested on the same system. Run-times are in seconds, and the numbers in parentheses next to the run-times of the program that uses dynamic partitioning give the total number of times the stacks were split.

From these numbers, it's apparent that different problems can result in radically different behaviors. For example, the program that uses static partitioning generally performs better on the first problem than the program that uses dynamic partitioning. However, on the second problem, the performance of the static program is essentially independent of the number of threads, while the dynamic program obtains excellent performance. In general, it appears that the dynamic program is more scalable than the static program.

As we increase the number of threads, we would expect that the size of the local stacks will decrease, and hence threads will run out of work more often. When threads are waiting, other threads will split their stacks, so as the number of threads is increased, the total number of stack splits should increase. Both problems confirm this prediction.

It should be noted that if the input problem has more than one possible solution—that is, different tours with the same minimum cost—then the results of both of the programs are nondeterministic. In the static program, the sequence of best tours depends on the speed of the threads, and this sequence determines which tree nodes are examined. In the dynamic program, we also have nondeterminism because different runs may result in different places where a thread splits its stack and variation in which thread receives the new work. This can also result in run-times, especially dynamic run-times, that are *highly* variable.

6.2.9 Parallelizing the tree-search programs using OpenMP

The issues involved in implementing the static and dynamic parallel tree-search programs using OpenMP are the same as the issues involved in implementing the programs using Pthreads.

There are almost no substantive differences between a static implementation that uses OpenMP and one that uses Pthreads. However, a couple of points should be mentioned:

1. When a single thread executes some code in the Pthreads version, the test

   ```
   if (my_rank == whatever)
   ```

   can be replaced by the OpenMP directive

   ```
   #  pragma omp single
   ```

   This will insure that the following structured block of code will be executed by one thread in the team, and the other threads in the team will wait in an implicit barrier at the end of the block until the executing thread is finished.

   When `whatever` is 0 (as it is in each test in the Pthreads program), the test can also be replaced by the OpenMP directive

   ```
   #  pragma omp master
   ```

   This will insure that thread 0 executes the following structured block of code. However, the `master` directive doesn't put an implicit barrier at the end of the block, so it may be necessary to also add a `barrier` directive after a structured block that has been modified by a `master` directive.
2. The Pthreads mutex that protects the best tour can be replaced by a single `critical` directive placed either inside the `Update_best_tour` function or immediately before the call to `Update_best_tour`. This is the only potential source of a race condition after the distribution of the initial tours, so the simple `critical` directive won't cause a thread to block unnecessarily.

The dynamically load-balanced Pthreads implementation depends heavily on Pthreads condition variables, and OpenMP doesn't provide a comparable object. The rest of the Pthreads code can be easily converted to OpenMP. In fact, OpenMP even provides a nonblocking version of `omp_set_lock`. Recall that OpenMP provides a lock object `omp_lock_t` and the following functions for acquiring and relinquishing the lock, respectively:

```
void omp_set_lock(omp_lock_t*    lock_p   /* in/out */);
void omp_unset_lock(omp_lock_t*  lock_p   /* in/out */);
```

It also provides the function

```
int omp_test_lock(omp_lock_t*    lock_p   /* in/out */);
```

which is analogous to `pthread_mutex_trylock`; it attempts to acquire the lock `*lock_p`, and if it succeeds it returns true (or nonzero). If the lock is being used by some other thread, it returns immediately with return value false (or zero).

If we examine the pseudocode for the Pthreads `Terminated` function in Program 6.8, we see that in order to adapt the Pthreads version to OpenMP, we need to emulate the functionality of the Pthreads function calls

```
pthread_cond_signal(&term_cond_var);
pthread_cond_broadcast(&term_cond_var);
pthread_cond_wait(&term_cond_var, &term_mutex);
```

in Lines 6, 17, and 22, respectively.

Recall that a thread that has entered the condition wait by calling

```
pthread_cond_wait(&term_cond_var, &term_mutex);
```

is waiting for either of two events:

- Another thread has split its stack and created work for the waiting thread.
- All of the threads have run out of work.

Perhaps the simplest solution to emulating a condition wait in OpenMP is to use busy-waiting. Since there are two conditions a waiting thread should test for, we can use two different variables in the busy-wait loop:

```
/* Global variables */
int awakened_thread = -1;
int work_remains = 1;  /* true */
. . .
while (awakened_thread != my_rank && work_remains);
```

Initialization of the two variables is crucial: If `awakened_thread` has the value of some thread's rank, that thread will exit immediately from the `while`, but there may be no work available. Similarly, if `work_remains` is initialized to 0, all the threads will exit the `while` loop immediately and quit.

Now recall that when a thread enters a Pthreads condition wait, it relinquishes the mutex associated with the condition variable so that another thread can also enter the condition wait or signal the waiting thread. Thus, we should relinquish the lock used in the `Terminated` function before starting the `while` loop.

Also recall that when a thread returns from a Pthreads condition wait, it reacquires the mutex associated with the condition variable. This is especially important in this setting since if the awakened thread has received work, it will need to access the shared data structures storing the new stack. Thus, our complete emulated condition wait should look something like this:

```
/* Global vars */
int awakened_thread = -1;
work_remains = 1;  /* true */
. . .
```

```
omp_unset_lock(&term_lock);
while (awakened_thread != my_rank && work_remains);
omp_set_lock(&term_lock);
```

If you recall the discussion of busy-waiting in Section 4.5 and Exercise 4.3 of Chapter 4, you may be concerned about the possibility that the compiler might reorder the code around the busy-wait loop. The compiler should not reorder across calls to `omp_set_lock` or `omp_unset_lock`. However, the updates to the variables *could* be reordered, so if we're going to be using compiler optimization, we should declare both with the `volatile` keyword.

Emulating the condition broadcast is straightforward: When a thread determines that there's no work left (Line 14 in Program 6.8), then the condition broadcast (Line 17) can be replaced with the assignment

```
work_remains = 0;  /* Assign false to work_remains */
```

The "awakened" threads can check if they were awakened by some thread's setting `work_remains` to false, and, if they were, return from `Terminated` with the value true.

Emulating the condition signal requires a little more work. The thread that has split its stack needs to choose one of the sleeping threads and set the variable `awakened_thread` to the chosen thread's rank. Thus, at a minimum, we need to keep a list of the ranks of the sleeping threads. A simple way to do this is to use a shared queue of thread ranks. When a thread runs out of work, it enqueues its rank before entering the busy-wait loop. When a thread splits its stack, it can choose the thread to awaken by dequeuing the queue of waiting threads:

```
got_lock = omp_test_lock(&term_lock);
if (got_lock != 0) {
   if (waiting_threads > 0 && new_stack == NULL) {
      Split my_stack creating new_stack;
      awakened_thread = Dequeue(term_queue);
   }
   omp_unset_lock(&term_lock);
}
```

The awakened thread needs to reset `awakened_thread` to -1 before it returns from its call to the `Terminated` function.

Note that there is no danger that some other thread will be awakened before the awakened thread reacquires the lock. As long as `new_stack` is not `NULL`, no thread will attempt to split its stack, and hence no thread will try to awaken another thread. So if several threads call `Terminated` before the awakened thread reacquires the lock, they'll either return if their stacks are nonempty, or they'll enter the wait if their stacks are empty.

6.2.10 Performance of the OpenMP implementations

Table 6.9 shows run-times of the two OpenMP implementations on the same two fifteen-city problems that we used to test the Pthreads implementations. The programs

Table 6.9 Performance of OpenMP and Pthreads Implementations of Tree Search (times in seconds)

	First Problem						Second Problem					
	Static		*Dynamic*				*Static*		*Dynamic*			
Th	*OMP*	*Pth*	*OMP*		*Pth*		*OMP*	*Pth*	*OMP*		*Pth*	
1	32.5	32.7	33.7	(0)	34.7	(0)	25.6	25.8	26.6	(0)	27.5	(0)
2	27.7	27.9	28.0	(6)	28.9	(7)	25.6	25.8	18.8	(9)	19.2	(6)
4	25.4	25.7	33.1	(75)	25.9	(47)	25.6	25.8	9.8	(52)	9.3	(49)
8	28.0	23.8	19.2	(134)	22.4	(180)	23.8	24.0	6.3	(163)	5.7	(256)

were also run on the same system we used for the Pthreads and serial tests. For ease of comparison, we also show the Pthreads run-times. Run-times are in seconds and the numbers in parentheses show the total number of times stacks were split in the dynamic implementations.

For the most part, the OpenMP implementations are comparable to the Pthreads implementations. This isn't surprising since the system on which the programs were run has eight cores, and we wouldn't expect busy-waiting to degrade overall performance unless we were using more threads than cores.

There are two notable exceptions for the first problem. The performance of the static OpenMP implementation with eight threads is much worse than the Pthreads implementation, and the dynamic implementation with four threads is much worse than the Pthreads implementation. This could be a result of the nondeterminism of the programs, but more detailed profiling will be necessary to determine the cause with any certainty.

6.2.11 Implementation of tree search using MPI and static partitioning

The vast majority of the code used in the static parallelizations of tree search using Pthreads and OpenMP is taken straight from the second implementation of serial, iterative tree search. In fact, the only differences are in starting the threads, the initial partitioning of the tree, and the `Update_best_tour` function. We might therefore expect that an MPI implementation would also require relatively few changes to the serial code, and this is, in fact, the case.

There is the usual problem of distributing the input data and collecting the results. In order to construct a complete tour, a process will need to choose an edge into each vertex and out of each vertex. Thus, each tour will require an entry from each row and each column for each city that's added to the tour, so it would clearly be advantageous for each process to have access to the entire adjacency matrix. Note that the adjacency matrix is going to be relatively small. For example, even if we have 100 cities, it's unlikely that the matrix will require more than 80,000 bytes of

storage, so it makes sense to simply read in the matrix on process 0 and broadcast it to all the processes.

Once the processes have copies of the adjacency matrix, the bulk of the tree search can proceed as it did in the Pthreads and OpenMP implementations. The principal differences lie in

- partitioning the tree,
- checking and updating the best tour, and
- after the search has terminated, making sure that process 0 has a copy of the best tour for output.

We'll discuss each of these in turn.

Partitioning the tree

In the Pthreads and OpenMP implementations, thread 0 uses breadth-first search to search the tree until there are at least `thread_count` partial tours. Each thread then determines which of these initial partial tours it should get and pushes its tours onto its local stack. Certainly MPI process 0 can also generate a list of `comm_sz` partial tours. However, since memory isn't shared, it will need to send the initial partial tours to the appropriate process. We could do this using a loop of sends, but distributing the initial partial tours looks an awful lot like a call to `MPI_Scatter`. In fact, the only reason we can't use `MPI_Scatter` is that the number of initial partial tours may not be evenly divisible by `comm_sz`. When this happens, process 0 won't be sending the same number of tours to each process, and `MPI_Scatter` requires that the source of the scatter send the same number of objects to each process in the communicator.

Fortunately, there is a variant of `MPI_Scatter`, `MPI_Scatterv`, which *can* be used to send different numbers of objects to different processes. First recall the syntax of `MPI_Scatter`:

```
int MPI_Scatter(
      void          sendbuf     /* in  */,
      int           sendcount   /* in  */,
      MPI_Datatype  sendtype    /* in  */,
      void*         recvbuf     /* out */,
      int           recvcount   /* in  */,
      MPI_Datatype  recvtype    /* in  */,
      int           root        /* in  */,
      MPI_Comm      comm        /* in  */);
```

Process `root` sends `sendcount` objects of type `sendtype` from `sendbuf` to each process in `comm`. Each process in `comm` receives `recvcount` objects of type `recvtype` into `recvbuf`. Most of the time, `sendtype` and `recvtype` are the same and `sendcount` and `recvcount` are also the same. In any case, it's clear that the `root` process must send the same number of objects to each process.

`MPI_Scatterv`, on the other hand, has syntax

```
int MPI_Scatterv(
      void*        sendbuf          /* in  */,
```

```
      int*          sendcounts      /* in  */,
      int*          displacements   /* in  */,
      MPI_Datatype  sendtype        /* in  */,
      void*         recvbuf         /* out */,
      int           recvcount       /* in  */,
      MPI_Datatype  recvtype        /* in  */,
      int           root            /* in  */,
      MPI_Comm      comm            /* in  */);
```

The single `sendcount` argument in a call to `MPI_Scatter` is replaced by two array arguments: `sendcounts` and `displacements`. Both of these arrays contain `comm_sz` elements: `sendcounts[q]` is the number of objects of type `sendtype` being sent to process q. Furthermore, `displacements[q]` specifies the start of the block that is being sent to process q. The displacement is calculated in units of type `sendtype`. So, for example, if `sendtype` is `MPI_INT`, and `sendbuf` has type `int*`, then the data that is sent to process q will begin in location

```
sendbuf + displacements[q]
```

In general, `displacements[q]` specifies the offset into `sendbuf` of the data that will go to process q. The "units" are measured in blocks with extent equal to the extent of `sendtype`.

Similarly, `MPI_Gatherv` generalizes `MPI_Gather`:

```
int MPI_Gatherv(
      void*         sendbuf          /* in  */,
      int           sendcount        /* in  */,
      MPI_Datatype  sendtype         /* in  */,
      void*         recvbuf          /* out */,
      int*          recvcounts       /* in  */,
      int*          displacements    /* in  */,
      MPI_Datatype  recvtype         /* in  */,
      int           root             /* in  */,
      MPI_Comm      comm             /* in  */);
```

Maintaining the best tour

As we observed in our earlier discussion of parallelizing tree search, having each process use its own best tour is likely to result in a lot of wasted computation since the best tour on one process may be much more costly than most of the tours on another process (see Exercise 6.21). Therefore, when a process finds a new best tour, it should send it to the other processes.

First note that when a process finds a new best tour, it really only needs to send its *cost* to the other processes. Each process only makes use of the cost of the current best tour when it calls `Best_tour`. Also, when a process updates the best tour, it doesn't care what the actual cities on the former best tour were; it only cares that the cost of the former best tour is greater than the cost of the new best tour.

During the tree search, when one process wants to communicate a new best cost to the other processes, it's important to recognize that we can't use `MPI_Bcast`,

for recall that MPI_Bcast is blocking and every process in the communicator must call MPI_Bcast. However, in parallel tree search the only process that will know that a broadcast should be executed is the process that has found a new best cost. If it tries to use MPI_Bcast, it will probably block in the call and never return, since it will be the only process that calls it. We therefore need to arrange that the new tour is sent in such a way that the sending process won't block indefinitely.

MPI provides several options. The simplest is to have the process that finds a new best cost use MPI_Send to send it to all the other processes:

```
for (dest = 0; dest < comm_sz; dest++)
   if (dest != my_rank)
      MPI_Send(&new_best_cost, 1, MPI_INT, dest, NEW_COST_TAG,
          comm);
```

Here, we're using a special tag defined in our program, NEW_COST_TAG. This will tell the receiving process that the message is a new cost–as opposed to some other type of message–for example, a tour.

The destination processes can periodically check for the arrival of new best tour costs. We can't use MPI_Recv to check for messages since it's blocking; if a process calls

```
MPI_Recv(&received_cost, 1, MPI_INT, MPI_ANY_SOURCE, NEW_COST_TAG,
      comm, &status);
```

the process will block until a matching message arrives. If no message arrives—for example, if no process finds a new best cost—the process will hang. Fortunately, MPI provides a function that only *checks* to see if a message is available; it doesn't actually try to receive a message. It's called MPI_Iprobe, and its syntax is

```
int MPI_Iprobe(
            int          source        /* in  */,
            int          tag           /* in  */,
            MPI_Comm     comm          /* in  */,
            int*         msg_avail_p   /* out */,
            MPI_Status*  status_p      /* out */);
```

It checks to see if a message from process rank source in communicator comm and with tag tag is available. If such a message is available, *msg_avail_p will be assigned the value true and the members of *status_p will be assigned the appropriate values. For example, status_p->MPI_SOURCE will be assigned the rank of the source of the message that's been received. If no message is available, *msg_avail_p will be assigned the value false. The source and tag arguments can be the wildcards MPI_ANY_SOURCE and MPI_ANY_TAG, respectively. So, to check for a message with a new cost from any process, we can call

```
MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm, &msg_avail, &status);
```

```
MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm, &msg_avail,
      &status);
while (msg_avail) {
   MPI_Recv(&received_cost, 1, MPI_INT, status.MPI_SOURCE,
         NEW_COST_TAG, comm, MPI_STATUS_IGNORE);
   if (received_cost < best_tour_cost)
      best_tour_cost = received_cost;
   MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm, &msg_avail,
     &status);
}  /* while */
```

Program 6.9: MPI code to check for new best tour costs

If `msg_avail` is true, then we can receive the new cost with a call to `MPI_Recv`:

```
MPI_Recv(&received_cost, 1, MPI_INT, status.MPI_SOURCE,
      NEW_COST_TAG, comm, MPI_STATUS_IGNORE);
```

A natural place to do this is in the `Best_tour` function. Before checking whether our new tour is the best tour, we can check for new tour costs from other processes with the code in Program 6.9.

This code will continue to receive messages with new costs as long as they're available. Each time a new cost is received that's better than the current best cost, the variable `best_tour_cost` will be updated.

Did you spot the potential problem with this scheme? If there is no buffering available for the sender, then the loop of calls to `MPI_Send` can cause the sending process to block until a matching receive is posted. If all the other processes have completed their searches, the sending process will hang. The loop of calls to `MPI_Send` is therefore unsafe.

There are a couple of alternatives provided by MPI: **buffered sends** and **nonblocking sends**. We'll discuss buffered sends here. See Exercise 6.22 for a discussion of nonblocking operations in MPI.

Modes and Buffered Sends

MPI provides four **modes** for sends: **standard**, **synchronous**, **ready**, and **buffered**. The various modes specify different semantics for the sending functions. The send that we first learned about, `MPI_Send`, is the standard mode send. With it, the MPI implementation can decide whether to copy the contents of the message into its own storage or to block until a matching receive is posted. Recall that in synchronous mode, the send will block until a matching receive is posted. In ready mode, the send is erroneous unless a matching receive is posted *before* the send is started. In buffered mode, the MPI implementation must copy the message into local temporary storage if a matching receive hasn't been posted. The local temporary storage must be provided by the user program, not the MPI implementation.

Each mode has a different function: MPI_Send, MPI_Ssend, MPI_Rsend, and MPI_Bsend, respectively, but the argument lists are identical to the argument lists for MPI_Send:

```
int MPI_Xsend(
      void*         message       /* in */,
      int           message_size  /* in */,
      MPI_Datatype  message_type  /* in */,
      int           dest          /* in */,
      int           tag           /* in */,
      MPI_Comm      comm          /* in */);
```

The buffer that's used by MPI_Bsend must be turned over to the MPI implementation with a call to MPI_Buffer_attach:

```
int MPI_Buffer_attach(
      void*  buffer       /* in */,
      int    buffer_size  /* in */);
```

The buffer argument is a pointer to a block of memory allocated by the user program and buffer_size is its size in bytes. A previously "attached" buffer can be reclaimed by the program with a call to

```
int MPI_Buffer_detach(
      void*   buf_p       /* out */,
      int*    buf_size_p  /* out */);
```

The *buf_p argument returns the address of the block of memory that was previously attached, and *buf_size_p gives its size in bytes. A call to MPI_Buffer_detach will block until all messages that have been stored in the buffer are transmitted. Note that since buf_p is an output argument, it should probably be passed in with the ampersand operator. For example:

```
char buffer[1000];
char* buf;
int buf_size;
. . .
MPI_Buffer_attach(buffer, 1000);
. . .
/* Calls to MPI_Bsend */
. . .
MPI_Buffer_detach(&buf, &buf_size);
```

At any point in the program only one user-provided buffer can be attached, so if there may be multiple buffered sends that haven't been completed, we need to estimate the amount of data that will be buffered. Of course, we can't know this with any certainty, but we do know that in any "broadcast" of a best tour, the process doing the broadcast will make comm_sz − 1 calls to MPI_Bsend, and each of these calls will send a single int. We can thus determine the size of the buffer needed for a single broadcast. The amount of storage that's needed for the *data* that's transmitted can be determined with a call to MPI_Pack_size:

```
int MPI_Pack_size(
      int           count     /* in  */,
      MPI_Datatype  datatype  /* in  */,
      MPI_Comm      comm      /* in  */,
      int*          size_p    /* out */);
```

The output argument gives an upper bound on the number of bytes needed to store the data in a message. This won't be enough, however. Recall that in addition to the data, a message stores information such as the destination, the tag, and the communicator, so for each message there is some additional overhead. An upper bound on this additional overhead is given by the MPI constant `MPI_BSEND_OVERHEAD`. For a single broadcast, the following code determines the amount of storage needed:

```
int data_size;
int message_size;
int bcast_buf_size;

MPI_Pack_size(1, MPI_INT, comm, &data_size);
message_size = data_size + MPI_BSEND_OVERHEAD;
bcast_buf_size = (comm_sz - 1)*message_size;
```

We should guess a generous upper bound on the number of broadcasts and multiply that by `bcast_buf_size` to get the size of the buffer to attach.

Printing the best tour

When the program finishes, we'll want to print out the actual tour as well as its cost, so we do need to get the tour to process 0. It might at first seem that we could arrange this by having each process store its local best tour—the best tour that it finds—and when the tree search has completed, each process can check its local best tour cost and compare it to the global best tour cost. If they're the same, the process could send its local best tour to process 0. There are, however, several problems with this. First, it's entirely possible that there are multiple "best" tours in the TSP digraph, tours that all have the same cost, and different processes may find these different tours. If this happens, multiple processes will try to send their best tours to process 0, and all but one of the threads could hang in a call to `MPI_Send`. A second problem is that it's possible that one or more processes never received the best tour cost, and they may try to send a tour that isn't optimal.

We can avoid these problems by having each process store its local best tour, but after all the processes have completed their searches, they can all call `MPI_Allreduce` and the process with the global best tour can then send it to process 0 for output. The following pseudocode provides some details:

```
struct {
   int cost;
   int rank;
} loc_data, global_data;

loc_data.cost = Tour_cost(loc_best_tour);
loc_data.rank = my_rank;
```

```
MPI_Allreduce(&loc_data, &global_data, 1, MPI_2INT, MPI_MINLOC,
      comm);
if (global_data.rank == 0) return;
      /* 0 already has the best tour */
if (my_rank == 0)
   Receive best tour from process global_data.rank;
else if (my_rank == global_data.rank)
   Send best tour to process 0;
```

The key here is the operation we use in the call to `MPI_Allreduce`. If we just used `MPI_MIN`, we would know what the cost of the global best tour was, but we wouldn't know who owned it. However, MPI provides a predefined operator, `MPI_MINLOC`, which operates on pairs of values. The first value is the value to be minimized—in our setting, the cost of the tour—and the second value is the *location* of the minimum—in our setting, the rank of the process that actually owns the best tour. If more than one process owns a tour with minimum cost, the location will be the lowest of the ranks of the processes that own a minimum cost tour. The input and the output buffers in the call to `MPI_Allreduce` are two-member structs. Since both the cost and the rank are `int`s, both members are `int`s. Note that MPI also provides a predefined type `MPI_2INT` for this type. When the call to `MPI_Allreduce` returns, we have two alternatives:

- If process 0 already has the best tour, we simply return.
- Otherwise, the process owning the best tour sends it to process 0.

Unreceived messages

As we noted in the preceding discussion, it is possible that some messages won't be received during the execution of the parallel tree search. A process may finish searching its subtree before some other process has found a best tour. This won't cause the program to print an incorrect result; the call to `MPI_Allreduce` that finds the process with the best tour won't return until every process has called it, and some process will have the best tour. Thus, it will return with the correct least-cost tour, and process 0 will receive this tour.

However, unreceived messages can cause problems with the call to `MPI_Buffer_detach` or the call to `MPI_Finalize`. A process can hang in one of these calls if it is storing buffered messages that were never received, so before we attempt to shut down MPI, we can try to receive any outstanding messages by using `MPI_Iprobe`. The code is very similar to the code we used to check for new best tour costs. See Program 6.9. In fact, the only messages that are not sent in collectives are the "best tour" message sent to process 0, and the best tour cost broadcasts. The MPI collectives will hang if some process doesn't participate, so we only need to look for unreceived best tours.

In the dynamically load-balanced code (which we'll discuss shortly) there are other messages, including some that are potentially quite large. To handle this situation, we can use the `status` argument returned by `MPI_Iprobe` to determine the size of the message and allocate additional storage as necessary (see Exercise 6.23).

6.2.12 Implementation of tree search using MPI and dynamic partitioning

In an MPI program that dynamically partitions the search tree, we can try to emulate the dynamic partitioning that we used in the Pthreads and OpenMP programs. Recall that in those programs, before each pass through the main `while` loop in the search function, a thread called a boolean-valued function called `Terminated`. When a thread ran out of work—that is, its stack was empty—it went into a condition wait (Pthreads) or a busy-wait (OpenMP) until it either received additional work or it was notified that there was no more work. In the first case, it returned to searching for a best tour. In the second case, it quit. A thread that had at least two records on its stack would give half of its stack to one of the waiting threads.

Much of this can be emulated in a distributed-memory setting. When a process runs out of work, there's no condition wait, but it can enter a busy-wait, in which it waits to either receive more work or notification that the program is terminating. Similarly, a process with work can split its stack and send work to an idle process.

The key difference is that there is no central repository of information on which processes are waiting for work, so a process that splits its stack can't just dequeue a queue of waiting processes or call a function such as `pthread_cond_signal`. It needs to "know" a process that's waiting for work so it can send the waiting process more work. Thus, rather than simply going into a busy-wait for additional work or termination, a process that has run out of work should send a request for work to another process. If it does this, then, when a process enters the `Terminated` function, it can check to see if there's a request for work from some other process. If there is, and the process that has just entered `Terminated` has work, it can send part of its stack to the requesting process. If there is a request, and the process has no work available, it can send a rejection. Thus, when we have distributed-memory, pseudocode for our `Terminated` function can look something like the pseudocode shown in Program 6.10.

`Terminated` begins by checking on the number of tours that the process has in its stack (Line 1); if it has at least two that are "worth sending," it calls `Fulfill_request` (Line 2). `Fulfill_request` checks to see if the process has received a request for work. If it has, it splits its stack and sends work to the requesting process. If it hasn't received a request, it just returns. In either case, when it returns from `Fulfill_request` it returns from `Terminated` and continues searching.

If the calling process doesn't have at least two tours worth sending, `Terminated` calls `Send_rejects` (Line 5), which checks for any work requests from other processes and sends a "no work" reply to each requesting process. After this, `Terminated` checks to see if the calling process has any work at all. If it does—that is, if its stack isn't empty—it returns and continues searching.

Things get interesting when the calling process has no work left (Line 9). If there's only one process in the communicator (`comm_sz` = 1), then the process returns from `Terminated` and quits. If there's more than one process, then the process "announces" that it's out of work in Line 11. This is part of the implementation

```
1  if (My_avail_tour_count(my_stack) >= 2) {
2     Fulfill_request(my_stack);
3     return false;  /* Still more work */
4  } else { /* At most 1 available tour */
5     Send_rejects();  /* Tell everyone who's requested */
6                      /* work that I have none         */
7     if (!Empty_stack(my_stack)) {
8        return false;  /* Still more work */
9     } else {  /* Empty stack */
10       if (comm_sz == 1) return true;
11       Out_of_work();
12       work_request_sent = false;
13       while (1) {
14          Clear_msgs();  /* Msgs unrelated to work, termination */
15          if (No_work_left()) {
16             return true;  /* No work left.  Quit */
17          } else if (!work_request_sent) {
18             Send_work_request();  /* Request work from someone */
19             work_request_sent = true;
20          } else {
21             Check_for_work(&work_request_sent, &work_avail);
22             if (work_avail) {
23                Receive_work(my_stack);
24                return false;
25             }
26          }
27       }  /* while */
28    } /* Empty stack */
29 } /* At most 1 available tour */
```

Program 6.10: `Terminated` function for a dynamically partitioned TSP solver that uses MPI

of a "distributed termination detection algorithm," which we'll discuss shortly. For now, let's just note that the termination detection algorithm that we used with shared-memory may not work, since it's impossible to guarantee that a variable storing the number of processes that have run out of work is up to date.

Before entering the apparently infinite `while` loop (Line 13), we set the variable `work_request_sent` to false (Line 12). As its name suggests, this variable tells us whether we've sent a request for work to another process; if we have, we know that we should wait for work or a message saying "no work available" from that process before sending out a request to another process.

The `while(1)` loop is the distributed-memory version of the OpenMP busy-wait loop. We are essentially waiting until we either receive work from another process or we receive word that the search has been completed.

When we enter the `while(1)` loop, we deal with any outstanding messages in Line 14. We may have received updates to the best tour cost and we may have received requests for work. It's essential that we tell processes that have requested

work that we have none, so that they don't wait forever when there's no work available. It's also a good idea to deal with updates to the best tour cost, since this will free up space in the sending process' message buffer.

After clearing out outstanding messages, we iterate through the possibilities:

- The search has been completed, in which case we quit (Lines 15–16).
- We don't have an outstanding request for work, so we choose a process and send it a request (Lines 17–19). We'll take a closer look at the problem of which process should be sent a request shortly.
- We do have an outstanding request for work (Lines 21–25). So we check whether the request has been fulfilled or rejected. If it has been fulfilled, we receive the new work and return to searching. If we received a rejection, we set `work_request_sent` to false and continue in the loop. If the request was neither fulfilled nor rejected, we also continue in the `while(1)` loop.

Let's take a closer look at some of these functions.

`My_avail_tour_count`

The function `My_avail_tour_count` can simply return the size of the process' stack. It can also make use of a "cutoff length." When a partial tour has already visited most of the cities, there will be very little work associated with the subtree rooted at the partial tour. Since sending a partial tour is likely to be a relatively expensive operation, it may make sense to only send partial tours with fewer than some cutoff number of edges. In Exercise 6.24 we take a look at how such a cutoff affects the overall run-time of the program.

`Fulfill_request`

If a process has enough work so that it can usefully split its stack, it calls `Fulfill_request` (Line 2). `Fulfill_request` uses `MPI_Iprobe` to check for a request for work from another process. If there is a request, it receives it, splits its stack, and sends work to the requesting process. If there isn't a request for work, the process just returns.

Splitting the stack

A `Split_stack` function is called by `Fulfill_request`. It uses the same basic algorithm as the Pthreads and OpenMP functions, that is, alternate partial tours with fewer than `split_cutoff` cities are collected for sending to the process that has requested work. However, in the shared-memory programs, we simply copy the tours (which are pointers) from the original stack to a new stack. Unfortunately, because of the pointers involved in the new stack, such a data structure cannot be simply sent to another process (see Exercise 6.25). Thus, the MPI version of `Split_stack` *packs* the contents of the new stack into contiguous memory and sends the block of contiguous memory, which is *unpacked* by the receiver into a new stack.

MPI provides a function, `MPI_Pack`, for packing data into a buffer of contiguous memory. It also provides a function, `MPI_Unpack`, for unpacking data from a buffer

of contiguous memory. We took a brief look at them in Exercise 6.20 of Chapter 3. Recall that their syntax is

```
int MPI_Pack(
      void*         data_to_be_packed    /* in     */,
      int           to_be_packed_count   /* in     */,
      MPI_Datatype  datatype             /* in     */,
      void*         contig_buf           /* out    */,
      int           contig_buf_size      /* in     */,
      int*          position_p           /* in/out */,
      MPI_Comm      comm                 /* in     */);

int MPI_Unpack(
      void*         contig_buf           /* in     */,
      int           contig_buf_size      /* in     */,
      int*          position_p           /* in/out */,
      void*         unpacked_data        /* out    */,
      int           unpack_count         /* in     */,
      MPI_Datatype  datatype             /* in     */,
      MPI_Comm      comm                 /* in     */);
```

`MPI_Pack` takes the data in `data_to_be_packed` and packs it into `contig_buf`. The `*position_p` argument keeps track of where we are in `contig_buf`. When the function is called, it should refer to the first available location in `contig_buf` before `data_to_be_packed` is added. When the function returns, it should refer to the first available location in `contig_buf` after `data_to_be_packed` has been added.

`MPI_Unpack` reverses the process. It takes the data in `contig_buf` and unpacks it into `unpacked_data`. When the function is called, `*position_p` should refer to the first location in `contig_buf` that hasn't been unpacked. When it returns, `*position_p` should refer to the next location in `contig_buf` after the data that was just unpacked.

As an example, suppose that a program contains the following definitions:

```
typedef struct {
   int* cities; /* Cities in partial tour           */
   int count;   /* Number of cities in partial tour */
   int cost;    /* Cost of partial tour             */
} tour_struct;
typedef tour_struct* tour_t;
```

Then we can send a variable with type `tour_t` using the following code:

```
void Send_tour(tour_t tour, int dest) {
   int position = 0;

   MPI_Pack(tour->cities, n+1, MPI_INT, contig_buf, LARGE,
         &position, comm);
   MPI_Pack(&tour->count, 1, MPI_INT, contig_buf, LARGE,
         &position, comm);
   MPI_Pack(&tour->cost, 1, MPI_INT, contig_buf, LARGE,
```

```
            &position, comm);
      MPI_Send(contig_buf, position, MPI_PACKED, dest, 0, comm);
   }  /* Send_tour */
```

Similarly, we can receive a variable of type `tour_t` using the following code:

```
void Receive_tour(tour_t tour, int src) {
   int position = 0;

   MPI_Recv(contig_buf, LARGE, MPI_PACKED, src, 0, comm,
         MPI_STATUS_IGNORE);
   MPI_Unpack(contig_buf, LARGE, &position, tour->cities, n+1,
         MPI_INT, comm);
   MPI_Unpack(contig_buf, LARGE, &position, &tour->count, 1,
         MPI_INT, comm);
   MPI_Unpack(contig_buf, LARGE, &position, &tour->cost, 1,
         MPI_INT, comm);
}  /* Receive_tour */
```

Note that the MPI datatype that we use for sending and receiving packed buffers is `MPI_PACKED`.

Send_rejects

The `Send_rejects` function (Line 5) is similar to the function that looks for new best tours. It uses `MPI_Iprobe` to search for messages that have requested work. Such messages can be identified by a special tag value, for example, `WORK_REQ_TAG`. When such a message is found, it's received, and a reply is sent indicating that there is no work available. Note that both the request for work and the reply indicating there is no work can be messages with zero elements, since the tag alone informs the receiver of the message's purpose. Even though such messages have no content outside of the envelope, the envelope does take space and they need to be received.

Distributed termination detection

The functions `Out_of_work` and `No_work_left` (Lines 11 and 15) implement the termination detection algorithm. As we noted earlier, an algorithm that's modeled on the termination detection algorithm we used in the shared-memory programs will have problems. To see this, suppose each process stores a variable `oow`, which stores the number of processes that are out of work. The variable is set to 0 when the program starts. Each time a process runs out of work, it sends a message to all the other processes saying it's out of work so that all the processes will increment their copies of `oow`. Similarly, when a process receives work from another process, it sends a message to every process informing them of this, and each process will decrement its copy of `oow`. Now suppose we have three process, and process 2 has work but processes 0 and 1 have run out of work. Consider the sequence of events shown in Table 6.10.

The error here is that the work sent from process 1 to process 0 is lost. The reason is that process 0 receives the notification that process 2 is out of work before it receives the notification that process 1 has received work. This may seem improbable,

Table 6.10 Termination Events that Result in an Error

Time	Process 0	Process 1	Process 2
0	Out of Work Notify 1, 2 oow = 1	Out of Work Notify 0, 2 oow = 1	Working oow = 0
1	Send request to 1 oow = 1	Send Request to 2 oow = 1	Recv notify fr 1 oow = 1
2	oow = 1	Recv notify fr 0 oow = 2	Recv request fr 1 oow = 1
3	oow = 1	oow = 2	Send work to 1 oow = 0
4	oow = 1	Recv work fr 2 oow = 1	Recv notify fr 0 oow = 1
5	oow = 1	Notify 0 oow = 1	Working oow = 1
6	oow = 1	Recv request fr 0 oow = 1	Out of work Notify 0, 1 oow = 2
7	Recv notify fr 2 oow = 2	Send work to 0 oow = 0	Send request to 1 oow = 2
8	Recv 1st notify fr 1 oow = 3	Recv notify fr 2 oow = 1	oow = 2
9	Quit	Recv request fr 2 oow = 1	oow = 2

but it's not improbable that process 1 was, for example, interrupted by the operating system and its message wasn't transmitted until after the message from process 2 was transmitted.

Although MPI guarantees that two messages sent from process A to process B will, in general, be received in the order in which they were sent, it makes no guarantee about the order in which messages will be received if they were sent by different processes. This is perfectly reasonable in light of the fact that different processes will, for various reasons, work at different speeds.

Distributed termination detection is a challenging problem, and much work has gone into developing algorithms that are guaranteed to correctly detect it. Conceptually, the simplest of these algorithms relies on keeping track of a quantity that is conserved and can be measured precisely. Let's call it *energy,* since, of course, energy is conserved. At the start of the program, each process has 1 unit of energy. When a process runs out of work, it sends its energy to process 0. When a process fulfills a request for work, it divides its energy in half, keeping half for itself, and sending half to the process that's receiving the work. Since energy is conserved and since the program started with `comm_sz` units, the program should terminate when process 0 finds that it has received a total of `comm_sz` units.

The `Out_of_work` function when executed by a process other than 0 sends its energy to process 0. Process 0 can just add its energy to a `received_energy` variable. The `No_work_left` function also depends on whether process 0 or some other process is calling. If process 0 is calling, it can receive any outstanding messages sent by `Out_of_work` and add the energy into `received_energy`. If `received_energy` equals `comm_sz`, process 0 can send a termination message (with a special tag) to every process. On the other hand, a process other than 0 can just check to see if there's a message with the termination tag.

The tricky part here is making sure that no energy is inadvertently lost; if we try to use `floats` or `doubles`, we'll almost certainly run into trouble since at some point dividing by two will result in underflow. Since the amount of energy in exact arithmetic can be represented by a common fraction, we can represent the amount of energy on each process exactly by a pair of fixed-point numbers. The denominator will always be a power of two, so we can represent it by its base-two logarithm. For a large problem it is possible that the numerator could overflow. However, if this becomes a problem, there are libraries that provide arbitrary precision rational numbers (e.g, GMP [21]). An alternate solution is explored in Exercise 6.26.

Sending requests for work

Once we've decided on which process we plan to send a request to, we can just send a zero-length message with a "request for work" tag. However, there are many possibilities for choosing a destination:

1. Loop through the processes in round-robin fashion. Start with `(my_rank + 1) % comm_sz` and increment this destination (modulo `comm_sz`) each time a new request is made. A potential problem here is that two processes can get "in synch" and request work from the same destination repeatedly.
2. Keep a global destination for requests on process 0. When a process runs out of work, it first requests the current value of the global destination from 0. Process 0 can increment this value (modulo `comm_sz`) each time there's a request. This avoids the issue of multiple processes requesting work from the same destination, but clearly process 0 can become a bottleneck.
3. Each process uses a random number generator to generate destinations. While it can still happen that several processes may simultaneously request work from the same process, the random choice of successive process ranks should reduce the chance that several processes will make repeated requests to the same process.

These are three possible options. We'll explore these options in Exercise 6.29. Also see [22] for an analysis of the options.

Checking for and receiving work

Once a request is sent for work, it's critical that the sending process repeatedly check for a response from the destination. In fact, a subtle point here is that it's critical that the sending process check for a message from the destination process with a "work available tag" or a "no work available tag." If the sending process simply checks

for a message from the destination, it may be "distracted" by other messages from the destination and never receive work that's been sent. For example, there might be a message from the destination requesting work that would mask the presence of a message containing work.

The `Check_for_work` function should therefore first probe for a message from the destination indicating work is available, and, if there isn't such a message, it should probe for a message from the destination saying there's no work available. If there is work available, the `Receive_work` function can receive the message with work and unpack the contents of the message buffer into the process' stack. Note also that it needs to unpack the energy sent by the destination process.

Performance of the MPI programs

Table 6.11 shows the performance of the two MPI programs on the same two fifteen-city problems on which we tested the Pthreads and the OpenMP implementations. Run-times are in seconds and the numbers in parentheses show the total number of times stacks were split in the dynamic implementations. These results were obtained on a different system from the system on which we obtained the Pthreads results. We've also included the Pthreads results for this system, so that the two sets of results can be compared. The nodes of this system only have four cores, so the Pthreads results don't include times for 8 or 16 threads. The cutoff number of cities for the MPI runs was 12.

The nodes of this system are small shared-memory systems, so communication through shared variables should be much faster than distributed-memory communication, and it's not surprising that in every instance the Pthreads implementation outperforms the MPI implementation.

The cost of stack splitting in the MPI implementation is quite high; in addition to the cost of the communication, the packing and unpacking is very time-consuming. It's also therefore not surprising that for relatively small problems with few processes, the static MPI parallelization outperforms the dynamic parallelization. However, the

Table 6.11 Performance of MPI and Pthreads Implementations of Tree Search (times in seconds)

	First Problem						Second Problem					
	Static		*Dynamic*				*Static*		*Dynamic*			
Th/Pr	*Pth*	*MPI*	*Pth*		*MPI*		*Pth*	*MPI*	*Pth*		*MPI*	
1	35.8	40.9	41.9	(0)	56.5	(0)	27.4	31.5	32.3	(0)	43.8	(0)
2	29.9	34.9	34.3	(9)	55.6	(5)	27.4	31.5	22.0	(8)	37.4	(9)
4	27.2	31.7	30.2	(55)	52.6	(85)	27.4	31.5	10.7	(44)	21.8	(76)
8		35.7			45.5	(165)		35.7			16.5	(161)
16		20.1			10.5	(441)		17.8			0.1	(173)

8- and 16-process results suggest that if a problem is large enough to warrant the use of many processes, the dynamic MPI program is much more scalable, and it can provide far superior performance. This is borne out by examination of a 17-city problem run with 16 processes: the dynamic MPI implementation has a run-time of 296 seconds, while the static implementation has a run-time of 601 seconds.

Note that times such as 0.1 second for the second problem running with 16 processes don't really show superlinear speedup. Rather, the initial distribution of work has allowed one of the processes to find the best tour much faster than the initial distributions with fewer processes, and the dynamic partitioning has allowed the processes to do a much better job of load balancing.

6.3 A WORD OF CAUTION

In developing our solutions to the *n*-body problem and TSP, we chose our serial algorithms because they were easy to understand and their parallelization was relatively straightforward. In no case did we choose a serial algorithm because it was the fastest or because it could solve the largest problem. Thus, it should not be assumed that either the serial or the parallel solutions are the best available. For information on "state-of-the-art" algorithms, see the bibliography, especially [12] for the *n*-body problem and [22] for parallel tree search.

6.4 WHICH API?

How can we decide which API, MPI, Pthreads, or OpenMP is best for our application? In general, there are many factors to consider, and the answer may not be at all clear cut. However, here are a few points to consider.

As a first step, decide whether to use distributed-memory, or shared-memory. In order to do this, first consider the amount of memory the application will need. In general, distributed-memory systems can provide considerably more main memory than shared-memory systems, so if the memory requirements are very large, you may need to write the application using MPI.

If the problem will fit into the main memory of your shared-memory system, you may still want to consider using MPI. Since the total available cache on a distributed-memory system will probably be much greater than that available on a shared-memory system, it's conceivable that a problem that requires lots of main memory accesses on a shared-memory system will mostly access cache on a distributed-memory system, and, consequently, have much better overall performance.

However, even if you'll get a big performance improvement from the large aggregate cache on a distributed-memory system, if you already have a large and complex serial program, it often makes sense to write a shared-memory program. It's often

possible to reuse considerably more serial code in a shared-memory program than a distributed-memory program. It's more likely that the serial data structures can be easily adapted to a shared-memory system. If this is the case, the development effort for the shared-memory program will probably be much less. This is especially true for OpenMP programs, since some serial programs can be parallelized by simply inserting some OpenMP directives.

Another consideration is the communication requirements of the parallel algorithm. If the processes/threads do little communication, an MPI program should be fairly easy to develop, and very scalable. At the other extreme, if the processes/threads need to be very closely coordinated, a distributed-memory program will probably have problems scaling to large numbers of processes, and the performance of a shared-memory program should be better.

If you decided that shared-memory is preferable, you will need to think about the details of parallelizing the program. As we noted earlier, if you already have a large, complex serial program, you should see if it lends itself to OpenMP. For example, if large parts of the program can be parallelized with `parallel for` directives, OpenMP will be much easier to use than Pthreads. On the other hand, if the program involves complex synchronization among the threads—for example, read-write locks or threads waiting on signals from other threads—then Pthreads will be much easier to use.

6.5 SUMMARY

In this chapter, we've looked at serial and parallel solutions to two very different problems: the *n*-body problem and solving the traveling salesperson problem using tree search. In each case we began by studying the problem and looking at serial algorithms for solving the problem. We continued by using Foster's methodology for devising a parallel solution, and then, using the designs developed with Foster's methodology, we implemented parallel solutions using Pthreads, OpenMP, and MPI. In developing the reduced MPI solution to the *n*-body problem, we determined that the "obvious" solution would be extremely difficult to implement correctly and would require a huge amount of communication. We therefore turned to an alternative "ring pass" algorithm, which proved to be much easier to implement and is probably more scalable.

In the dynamically partitioned solutions for parallel tree search, we used different methods for the three APIs. With Pthreads, we used a condition variable both for communicating new work among the threads and for termination. OpenMP doesn't provide an analog to Pthreads condition variables, so we used busy-waiting instead. In MPI, since all data is local, we needed to use a more complicated scheme to redistribute work, in which a process that runs out of work chooses a destination process and requests work from that process. To implement this correctly, a process that runs out of work enters a busy-wait loop in which it requests work, looks for a response to the work request, and looks for a termination message.

We saw that in a distributed-memory environment in which processes send each other work, determining when to terminate is a nontrivial problem. We also looked at a relatively straightforward solution to the problem of distributed termination detection, in which there is a fixed amount of "energy" throughout the execution of the program. When processes run out of work they send their energy to process 0, and when processes send work to other processes, they also send half of their current energy. Thus, when process 0 finds that it has all the energy, there is no more work, and it can send a termination message.

In closing, we looked briefly at the problem of deciding which API to use. The first consideration is whether to use shared-memory or distributed-memory. To decide this, we should look at the memory requirements of the application and the amount of communication among the processes/threads. If the memory requirements are great or the distributed-memory version can work mainly with cache, then a distributed-memory program is likely to be much faster. On the other hand, if there is considerable communication, a shared-memory program will probably be faster.

In choosing between OpenMP and Pthreads, if there's an existing serial program and it can be parallelized by the insertion of OpenMP directives, then OpenMP is probably the clear choice. However, if complex thread synchronization is needed—for example, read-write locks or thread signaling—then Pthreads will be easier to use. In the course of developing these programs, we also learned some more about Pthreads, OpenMP, and MPI.

6.5.1 Pthreads and OpenMP

In tree search, we need to check the cost of the current best tour before updating the best tour. In the Pthreads and OpenMP implementations of parallel tree search, updating the best tour introduces a race condition. A thread that wants to update the best tour must therefore first acquire a lock. The combination of "test lock condition" and "update lock condition" can cause a problem: the lock condition (e.g. the cost of the best tour) can change between the time of the first test and the time that the lock is acquired. Thus, the threads also need to check the lock condition *after* they acquire the lock, so pseudocode for updating the best tour should look something like this:

```
if (new_tour_cost < best_tour_cost) {
   Acquire lock protecting best tour;
   if (new_tour_cost < best_tour_cost)
      Update best tour;
   Relinquish lock;
}
```

Remember that we have also learned that Pthreads has a *nonblocking* version of `pthreads_mutex_lock` called `pthread_mutex_trylock`. This function checks to see if the mutex is available. If it is, it acquires the mutex and returns the value 0. If the mutex isn't available, instead of waiting for it to become available, it will return a nonzero value.

The analog of `pthread_mutex_trylock` in OpenMP is `omp_test_lock`. However, its return values are the opposite of those for `pthread_mutex_trylock`: it returns a nonzero value if the lock is acquired and a zero value if the lock is not acquired.

When a single thread should execute a structured block, OpenMP provides a couple of alternatives to the test, and:

```
if (my_rank == special_rank) {
   Execute action;
}
```

With the `single` directive

```
#     pragma omp single
      Execute action;

      Next action;
```

the run-time system will choose a single thread to execute the action. The other threads will wait in an implicit barrier before proceeding to `Next action`. With the `master` directive

```
#     pragma omp master
      Execute action;

      Next action;
```

the master thread (thread 0) will execute the action. However, unlike the `single` directive, there is no implicit barrier after the block `Execute action`, and the other threads in the team will proceed immediately to execute `Next action`. Of course, if we need a barrier before proceeding, we can add an explicit barrier after completing the structured block `Execute action`. In Exercise 6.6 we see that OpenMP provides a `nowait` clause which can modify a `single` directive:

```
#     pragma omp single nowait
      Execute action;

      Next action;
```

When this clause is added, the thread selected by the run-time system to execute the action will execute it as before. However, the other threads in the team won't wait, they'll proceed immediately to execute `Next action`. The `nowait` clause can also be used to modify `parallel for` and `for` directives.

6.5.2 MPI

We learned quite a bit more about MPI. We saw that in some of the collective communication functions that use an input and an output buffer, we can use the argument `MPI_IN_PLACE` so that the input and output buffers are the same. This can save on memory and the implementation may be able to avoid copying from the input buffer to the output buffer.

The functions MPI_Scatter and MPI_Gather can be use to split an array of data among processes and collect distributed data into a single array, respectively. However, they can only be used when the amount of data going to or coming from each process is the same for each process. If we need to assign different amounts of data to each process, or to collect different amounts of data from each process, we can use MPI_Scatterv and MPI_Gatherv, respectively:

```
int MPI_Scatterv(
         void*        sendbuf          /* in  */,
         int*         sendcounts       /* in  */,
         int*         displacements    /* in  */,
         MPI_Datatype sendtype         /* in  */,
         void*        recvbuf          /* out */,
         int          recvcount        /* in  */,
         MPI_Datatype recvtype         /* in  */,
         int          root             /* in  */,
         MPI_Comm     comm             /* in  */);

int MPI_Gatherv(
         void*        sendbuf          /* in  */,
         int          sendcount        /* in  */,
         MPI_Datatype sendtype         /* in  */,
         void*        recvbuf          /* out */,
         int*         recvcounts       /* in  */,
         int*         displacements    /* in  */,
         MPI_Datatype recvtype         /* in  */,
         int          root             /* in  */,
         MPI_Comm     comm             /* in  */);
```

The arguments sendcounts for MPI_Scatterv and recvcounts for MPI_Gatherv are arrays with comm_sz elements. They specify the amount of data (in units of sendtype/recvtype) going to or coming from each process. The displacements arguments are also arrays with comm_sz elements. They specify the offsets (in units of sendtype/recvtype) of the data going to or coming from each process.

We saw that there is a special operator, MPI_MIN_LOC, that can be used in calls to MPI_Reduce and MPI_Allreduce. It operates on pairs of values and returns a pair of values. If the pairs are

$$(a_0, b_0), (a_1, b_1), \ldots, (a_{\texttt{comm\_sz}-1}, b_{\texttt{comm\_sz}-1}),$$

suppose that a is the minimum of the a_i's and q is the smallest process rank at which a occurs. Then the MPI_MIN_LOC operator will return the pair (a_q, b_q). We used this to find not only the cost of the minimum-cost tour, but by making the b_i's the process ranks, we determined which process owned the minimum-cost tour.

In our development of two MPI implementations of parallel tree search, we made repeated use of MPI_Iprobe:

```
int MPI_Iprobe(
           int         source        /* in  */,
           int         tag           /* in  */,
```

```
MPI_Comm    comm          /* in  */,
int*        msg_avail_p   /* out */,
MPI_Status  status_p      /* out */);
```

It checks to see if there is a message from `source` with tag `tag` available to be received. If such a message is available, `msg_avail_p` will be given the value true. Note that `MPI_Iprobe` doesn't actually receive the message, but if such a message is available, a call to `MPI_Recv` will receive it. Both the `source` and `tag` arguments can be the wildcards `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, respectively. For example, we often wanted to check whether any process had sent a message with a new best cost. We checked for the arrival of such a message with the call

```
MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm, &msg_avail,
      &status);
```

If such a message is available, its source will be returned in the `*status_p` argument. Thus, `status.MPI_SOURCE` can be used to receive the message:

```
MPI_Recv(&new_cost, 1, MPI_INT, status.MPI_SOURCE, NEW_COST_TAG,
      comm, MPI_STATUS_IGNORE);
```

There were several occasions when we wanted a send function to return immediately, regardless of whether the message had actually been sent. One way to arrange this in MPI is to use **buffered send mode**. In a buffered send, the user program provides storage for messages with a call to `MPI_Buffer_attach`. Then when the program sends the message with `MPI_Bsend`, the message is either transmitted immediately or copied to the user-program-provided buffer. In either case the call returns without blocking. When the program no longer needs to use buffered send mode, the buffer can be recovered with a call to `MPI_Buffer_detach`.

We also saw that MPI provides three other modes for sending: **synchronous**, **standard**, and **ready**. Synchronous sends won't buffer the data; a call to the synchronous send function `MPI_Ssend` won't return until the receiver has begun receiving the data. Ready sends (`MPI_Rsend`) are erroneous unless the matching receive has already been started when `MPI_Rsend` is called. The ordinary send `MPI_Send` is called the standard mode send.

In Exercise 6.22 we explore an alternative to buffered mode: nonblocking sends. As the name suggests, a nonblocking send returns regardless of whether the message has been transmitted. However, the send must be completed by calling one of several functions that *wait* for completion of the nonblocking operation. There is also a nonblocking receive function.

Since addresses on one system will, in general, have no relation to addresses on another system, pointers should not be sent in MPI messages. If you're using data structures that have embedded pointers, MPI provides the function `MPI_Pack` for storing a data structure in a single, contiguous buffer before sending. Similarly, the function `MPI_Unpack` can be used to take data that's been received into a single contiguous buffer and unpack it into a local data structure. Their syntax is

```
int MPI_Pack(
      void*         data_to_be_packed    /* in     */,
      int           to_be_packed_count   /* in     */,
      MPI_Datatype  datatype             /* in     */,
      void*         contig_buf           /* out    */,
      int           contig_buf_size      /* in     */,
      int*          position_p           /* in/out */,
      MPI_Comm      comm                 /* in     */);

int MPI_Unpack(
      void*         contig_buf           /* in     */,
      int           contig_buf_size      /* in     */,
      int*          position_p           /* in/out */,
      void*         unpacked_data        /* out    */,
      int           unpack_count         /* in     */,
      MPI_Datatype  datatype             /* in     */,
      MPI_Comm      comm                 /* in     */);
```

The key to their use is the `position_p` argument. When `MPI_Pack` is called, it should reference the first available location in `contig_buf`. So, for example, when we start packing the data `*position_p` should be set to 0. When `MPI_Pack` returns, `*position_p` will refer to the first available location following the data that was just packed. Thus, successive elements of a data structure can be packed into a single buffer by repeated calls to `MPI_Pack`. When a packed buffer is received, the data can be unpacked in a completely analogous fashion. Note that when a buffer packed with `MPI_Pack` is sent, the datatype for both the send and the receive should be `MPI_PACKED`.

6.6 EXERCISES

6.1. In each iteration of the serial *n*-body solver, we first compute the total force on each particle, and then we compute the position and velocity of each particle. Would it be possible to reorganize the calculations so that in each iteration we did all of the calculations for each particle before proceeding to the next particle? That is, could we use the following pseudocode?

```
for each timestep
   for each particle {
      Compute total force on particle;
      Find position and velocity of particle;
      Print position and velocity of particle;
   }
```

If so, what other modifications would we need to make to the solver? If not, why not?

6.2. Run the basic serial *n*-body solver for 1000 timesteps with a stepsize of 0.05, no output, and internally generated initial conditions. Let the number

of particles range from 500 to 2000. How does the run-time change as the number of particles increases? Can you extrapolate and predict how many particles the solver could handle if it ran for 24 hours?

6.3. Parallelize the reduced version of the *n*-body solver with OpenMP or Pthreads and a single `critical` directive (OpenMP) or a single mutex (Pthreads) to protect access to the `forces` array. Parallelize the rest of the solver by parallelizing the inner `for` loops. How does the performance of this code compare with the performance of the serial solver? Explain your answer.

6.4. Parallelize the reduced version of the *n*-body solver with OpenMP or Pthreads and a lock/mutex for each particle. The locks/mutexes should be used to protect access to updates to the `forces` array. Parallelize the rest of the solver by parallelizing the inner `for` loops. How does the performance compare with the performance of the serial solver? Explain your answer.

6.5. In the shared-memory reduced *n*-body solver, if we use a block partition in both phases of the calculation of the forces, the loop in the second phase can be changed so that the `for thread` loop only goes up to `my_rank` instead of `thread_count`. That is, the code

```
#       pragma omp for
        for (part = 0; part < n; part++) {
           forces[part][X] = forces[part][Y] = 0.0;
           for (thread = 0; thread < thread_count; thread++) {
              forces[part][X] += loc_forces[thread][part][X];
              forces[part][Y] += loc_forces[thread][part][Y];
           }
        }
```

can be changed to

```
#       pragma omp for
        for (part = 0; part < n; part++) {
           forces[part][X] = forces[part][Y] = 0.0;
           for (thread = 0; thread < my_rank; thread++) {
              forces[part][X] += loc_forces[thread][part][X];
              forces[part][Y] += loc_forces[thread][part][Y];
           }
        }
```

Explain why this change is OK. Run the program with this modification and compare its performance with the original code with block partitioning and the code with a cyclic partitioning of the first phase of the forces calculation. What conclusions can you draw?

6.6. In our discussion of the OpenMP implementation of the basic *n*-body solver, we observed that the implied barrier after the output statement wasn't necessary. We could therefore modify the `single` directive with a `nowait` clause. It's possible to also eliminate the implied barriers at the ends of the two `for`

each particle q loops by modifying for directives with nowait clauses. Would doing this cause any problems? Explain your answer.

6.7. For the shared-memory implementation of the reduced n-body solver, we saw that a cyclic schedule for the computation of the forces outperformed a block schedule, in spite of the reduced cache performance. By experimenting with the OpenMP or the Pthreads implementation, determine the performance of various block-cyclic schedules. Is there an optimal block size for your system?

6.8. If $\mathbf{x}$ and $\mathbf{y}$ are double-precision n-dimensional vectors and α is a double-precision scalar, the assignment

$$\mathbf{y} \longleftarrow \alpha\mathbf{x} + \mathbf{y}$$

is called a DAXPY. DAXPY is an abbreviation of "Double precision Alpha times **X** Plus **Y**." Write a Pthreads or OpenMP program in which the master thread generates two large, random n-dimensional arrays and a random scalar, all of which are doubles. The threads should then carry out a DAXPY on the randomly generated values. For large values of n and various numbers of threads compare the performance of the program with a block partition and a cyclic partition of the arrays. Which partitioning scheme performs better? Why?

6.9. Write an MPI program in which each process generates a large, initialized, m-dimensional array of doubles. Your program should then repeatedly call MPI_Allgather on the m-dimensional arrays. Compare the performance of the calls to MPI_Allgather when the global array (the array that's created by the call to MPI_Allgather) has

a. a block distribution, and

b. a cyclic distribution.

To use a cyclic distribution, download the code cyclic_derived.c from the book's web site, and use the MPI datatype created by this code for the *destination* in the calls to MPI_Allgather. For example, we might call

```
MPI_Allgather(sendbuf, m, MPI_DOUBLE, recvbuf, 1, cyclic_mpi_t,
      comm);
```

if the new MPI datatype were called cyclic_mpi_t.

Which distribution performs better? Why? Don't include the overhead involved in building the derived datatype.

6.10. Consider the following code:

```
int n, thread_count, i, chunksize;
double x[n], y[n], a;
    . . .
#   pragma omp parallel num_threads(thread_count) \
       default(none) private(i) \
       shared(x, y, a, n, thread_count, chunksize)
```

```
    {
#     pragma omp for schedule(static, n/thread_count)
      for (i = 0; i < n; i++) {
         x[i] = f(i);  /* f is a function */
         y[i] = g(i);  /* g is a function */
      }
#     pragma omp for schedule(static, chunksize)
      for (i = 0; i < n; i++)
         y[i] += a*x[i];
   }  /*  omp parallel */
```

Suppose `n = 64`, `thread_count = 2`, the cache-line size is 8 `doubles`, and each core has an L2 cache that can store 131,072 `doubles`. If `chunksize = n/thread_count`, how many L2 cache misses do you expect in the second loop? If `chunksize = 8`, how many L2 misses do you expect in the second loop? You can assume that both `x` and `y` are aligned on a cache-line boundary. That is, both `x[0]` and `y[0]` are the first elements in their respective cache lines.

6.11. Write an MPI program that compares the performance of `MPI_Allgather` using `MPI_IN_PLACE` with the performance of `MPI_Allgather` when each process uses separate send and receive buffers. Which call to `MPI_Allgather` is faster when run with a single process? What if you use multiple processes?

6.12. **a.** Modify the basic MPI implementation of the *n*-body solver so that it uses a separate array for the local positions. How does its performance compare with the performance of the original *n*-body solver? (Look at performance with I/O turned off.)

b. Modify the basic MPI implementation of the *n*-body solver so that it distributes the masses. What changes need to be made to the communications in the program? How does the performance compare with the original solver?

6.13. Using Figure 6.6 as a guide, sketch the communications that would be needed in an "obvious" MPI implementation of the reduced *n*-body solver if there were three processes, six particles, and the solver used a cyclic distribution of the particles.

6.14. Modify the MPI version of the reduced *n*-body solver so that it uses two calls to `MPI_Sendrecv_replace` for each phase of the ring pass. How does the performance of this implementation compare to the implementation that uses a single call to `MPI_Sendrecv_replace`?

6.15. A common problem in MPI programs is converting global array indexes to local array indexes and vice-versa.

a. Find a formula for determining a global index from a local index if the array has a block distribution.

b. Find a formula for determining a local index from a global index if the array has a block distribution.

c. Find a formula for determining a global index from a local index if the array has a cyclic distribution.
d. Find a formula for determining a local index from a global index if the array has cyclic distribution.

You can assume that the number of processes evenly divides the number of elements in the global array. Your solutions should only use basic arithmetic operators ($+, -, *, /, \%$). They shouldn't use any loops or branches.

6.16. In our implementation of the reduced n-body solver, we make use of a function `First_index` which, given a global index of a particle assigned to one process, determines the "next higher" global index of a particle assigned to another process. The input arguments to the function are the following:

a. The global index of the particle assigned to the first process
b. The rank of the first process
c. The rank of the second process
d. The number of processes

The return value is the global index of the second particle. The function assumes that the particles have a cyclic distribution among the processes. Write C-code for `First_index`. (*Hint*: Consider two cases: the rank of the first process is less than the rank of the second, and the rank of the first is greater than or equal to the rank of the second).

6.17. **a.** Use Figure 6.10 to determine the maximum number of records that would be on the stack at any one time in solving a four-city TSP. (*Hint*: Look at the stack after branching as far as possible to the left).
b. Draw the tree structure that would be generated in solving a five-city TSP.
c. Determine the maximum number of records that would be on the stack at any one time during a search of this tree.
d. Use your answers to the preceding parts to determine a formula for the maximum number of records that would be on the stack at any one time in solving an n-city TSP.

6.18. Breadth-first search can be implemented as an iterative algorithm using a **queue**. Recall that a queue is a "first-in first-out" list data structure, in which objects are removed, or *dequeued*, in the same order in which they're added, or *enqueued.* We can use a queue to solve TSP and implement breadth-first search as follows:

```
queue = Init_queue();  /* Create empty queue */
tour = Init_tour();    /* Create partial tour that visits
    hometown */
Enqueue(queue, tour);
while (!Empty(queue)) {
   tour = Dequeue(queue);
   if (City_count(tour) == n) {
      if (Best_tour(tour))
         Update_best_tour(tour);
   } else {
```

```
         for each neighboring city
            if (Feasible(tour, city)) {
               Add_city(tour, city);
               Enqueue(tour);
               Remove_last_city(tour);
            }
      }
      Free_tour(tour);
   }  /* while !Empty */
```

This algorithm, although correct, will have serious difficulty if it's used on a problem with more than 10 or so cities. Why?

In the shared-memory implementations of our solutions to TSP, we use breadth-first search to create an initial list of tours that can be divided among the threads.

a. Modify this code so that it can be used by thread 0 to generate a queue of at least `thread_count` tours.

b. Once the queue has been generated by thread 0, write pseudocode that shows how the threads can initialize their stacks with a block of tours stored in the queue.

6.19. Modify the Pthreads implementation of static tree search so that it uses a read-write lock to protect the examination of the best tour. Read-lock the best tour when calling `Best_tour`, and write-lock it when calling `Update_best_tour`. Run the modified program with several input sets. How does the change affect the overall run-time?

6.20. Suppose the stack on process/thread A contains k tours.

a. Perhaps the simplest strategy for implementing stack splitting in TSP is to pop $k/2$ tours from A's existing stack and push them onto the new stack. Explain why this is unlikely to be a good strategy.

b. Another simple strategy is to split the stack on the basis of the cost of the partial tours on the stack. The least-cost partial tour goes to A. The second cheapest tour goes to `new_stack`. The third cheapest goes to A, and so on. Is this likely to be a good strategy? Explain your answer.

c. A variation on the strategy outlined in the preceding problem is to use average cost per edge. In average cost per edge, the partial tours on A's stack are ordered according to their cost divided by the number of edges in the partial tour. Then the tours are assigned in round-robin fashion to the stacks, that is, the cheapest cost per edge to A, the next cheapest cost per edge to `new_stack`, and so on. Is this likely to be a good strategy? Explain your answer.

Implement the three strategies outlined here in one of the dynamic load-balancing codes. How do these strategies compare to each other and the strategy outlined in the text? How did you collect your data?

6.21. **a.** Modify the static MPI TSP program so that each process uses a local best tour data structure until it has finished searching. When all the processes

have finished executing, the processes should execute a global reduction to find the least-cost tour. How does the performance of this implementation compare to the static implementation? Can you find input problems for which its performance is competitive with the original static implementation?

b. Create a TSP digraph in which the initial tours assigned to processes $1, 2, \ldots,$ `comm_sz` $- 1$ all have an edge that has a cost that is much greater than the total cost of any complete tour that will be examined by process 0. How do the various implementations perform on this problem when `comm_sz` processes are used?

6.22. `MPI_Recv` and each of the sends we've studied is blocking. `MPI_Recv` won't return until the message is received, and the various sends won't return until the message is sent or buffered. Thus, when one of these operations returns, you know the status of the message buffer argument. For `MPI_Recv`, the message buffer contains the received message—at least if there's been no error—and for the send operations, the message buffer can be reused. MPI also provides a **nonblocking** version of each of these functions, that is, they return as soon as the MPI run-time system has registered the operation. Furthermore, when they return, the message buffer argument cannot be accessed by the user program: the MPI run-time system can use the actual user message buffer to store the message. This has the virtue that the message doesn't have to be copied into or from an MPI-supplied storage location.

When the user program wants to reuse the message buffer, she can force the operation to complete by calling one of several possible MPI functions. Thus, the nonblocking operations split a communication into two phases:

- Begin the communication by calling one of the nonblocking functions
- Complete the communication by calling one of the completion functions

Each of the nonblocking send initiation functions has the same syntax as the blocking function, except that there is a final *request* argument. For example,

```
int MPI_Isend(
      void*          msg         /* in  */,
      int            count       /* in  */,
      MPI_Datatype   datatype    /* in  */,
      int            dest        /* in  */,
      int            tag         /* in  */,
      MPI_Comm       comm        /* in  */,
      MPI_Request*   request_p   /* out */);
```

The nonblocking receive replaces the status argument with a request argument. The request arguments identify the operation to the run-time system, so that when a program wishes to complete the operation, the completion function takes a request argument.

The simplest completion function is MPI_Wait:

```
int MPI_Wait(
      MPI_Request*  request_p  /* in/out */
      MPI_Status*   status_p   /* out    */);
```

When this returns, the operation that created *request_p will have completed. In our setting, *request_p will be set to MPI_REQUEST_NULL, and *status_p will store information on the completed operation.

Note that nonblocking receives can be matched to blocking sends and nonblocking sends can be matched to blocking receives.

We can use nonblocking sends to implement our broadcast of the best tour. The basic idea is that we create a couple of arrays containing comm_sz elements. The first stores the cost of the new best tour, the second stores the requests, so the basic broadcast looks something like this:

```
int costs[comm_sz];
MPI_Request requests[comm_sz];

for (dest = 0; dest < comm_sz; dest++)
   if (my_rank != dest) {
      costs[dest] = new_best_tour_cost;
      MPI_Isend(&costs[dest], 1, MPI_INT, dest, NEW_COST_TAG,
            comm, &requests[dest]);
   }
requests[my_rank] = MPI_REQUEST_NULL;
```

When this loop is completed, the sends will have been started, and they can be matched by ordinary calls to MPI_Recv.

There are a variety of ways to deal with subsequent broadcasts. Perhaps the simplest is to wait on all the previous nonblocking sends with the function MPI_Waitall:

```
int MPI_Waitall(
      int          count       /* in     */,
      MPI_Request  requests[]  /* in/out */,
      MPI_Status   statuses[]  /* out    */);
```

When this returns, all of the operations will have completed (assuming there are no errors). Note that it's OK to call MPI_Wait and MPI_Waitall if a request has the value MPI_REQUEST_NULL.

Use nonblocking sends to implement a broadcast of best tour costs in the static MPI implementation of the TSP program. How does its performance compare to the performance of the implementation that uses buffered sends?

6.23. Recall that an MPI_Status object is a struct with members for the source, the tag, and any error code for the associated message. It also stores information

on the size of the message. However, this isn't directly accessible as a member, it is only accessible through the MPI function `MPI_Get_count`:

```
int MPI_Get_count(
      MPI_Status*   status_p   /* in  */,
      MPI_Datatype  datatype   /* in  */,
      int*          count_p    /* out */);
```

When `MPI_Get_count` is passed the status of a message and a datatype, it returns the number of objects of the given datatype in the message. Thus, `MPI_Iprobe` and `MPI_Get_count` can be used to determine the size of an incoming message before the message is received. Use these to write a `Cleanup_messages` function that can be called before an MPI program quits. The purpose of the function is to receive any unreceived messages so that functions such as `MPI_Buffer_detach` won't hang.

6.24. The program `mpi_tsp_dyn.c` takes a command-line argument `split_cutoff`. If a partial tour has visited `split_cutoff` or more cities, it's not considered a candidate for sending to another process. The `Fulfill_request` function will therefore only send partial tours with fewer than `split_cutoff` cities. How does `split_cutoff` affect the overall run-time of the program? Can you find a reasonably good rule of thumb for deciding on the `split_cutoff`? How does changing the number of processes (that is, changing `comm_sz`) affect the best value for `split_cutoff`?

6.25. Pointers cannot be sent by MPI programs since an address that is valid on the sending process may cause a segmentation violation on the receiving process, or, perhaps worse, refer to memory that's already being used by the receiving process. There are a couple of alternatives that can be used to address this problem:

a. The object that uses pointers can be packed into contiguous memory by the sender and unpacked by the receiver.

b. The sender and the receiver can build MPI derived datatypes that map the memory used by the sender and valid memory on the receiver.

Write two `Send_linked_list` functions and two matching `Recv_linked_list` functions. The first pair of send-receive functions should use `MPI_Pack` and `MPI_Unpack`. The second pair should use derived datatypes. Note that the second pair may need to send two messages: the first will tell the receiver how many nodes are in the linked list, and the second will send the actual list. How does the performance of the two pairs of functions compare? How does their performance compare to the cost of sending a block of contiguous memory of the same size as the packed list?

6.26. The dynamically partitioned MPI implementation of the TSP solver uses a termination detection algorithm that may require the use of very high-precision

rational arithmetic (that is, common fractions with very large numerators and/or denominators).

a. If the total amount of energy is `comm_sz`, explain why the amount of energy stored by any process other than zero will have the form $1/2^k$ for some nonnegative integer k. Thus, the amount of energy stored by any process other than zero can be represented by k, an unsigned integer.

b. Explain why the representation in the first part is extremely unlikely to overflow or underflow.

c. Process 0, on the other hand, will need to store fractions with a numerator other than one. Explain how to implement such a fraction using an unsigned integer for the denominator and a bit array for the numerator. How can this implementation deal with overflow of the numerator?

6.27. If there are many processes and many redistributions of work in the dynamic MPI implementation of the TSP solver, process 0 could become a bottleneck for energy returns. Explain how one could use a spanning tree of processes in which a child sends energy to its parent rather than process 0.

6.28. Modify the implementation of the TSP solver that uses MPI and dynamic partitioning of the search tree so that each process reports the number of times it sends an "out of work" message to process 0. Speculate about how receiving and handling the out of work messages affects the overall run-time for process 0.

6.29. The C source file `mpi_tsp_dyn.c` contains the implementation of the MPI TSP solver that uses dynamic partitioning of the search tree. The online version uses the first of the three methods outlined in Section 6.2.12 for determining to which process a request for work should be sent. Implement the other two methods and compare the performance of the three. Does one method consistently outperform the other two?

6.30. Determine which of the three APIs is preferable for the n-body solvers and solving TSP.

a. How much memory is required for each of the serial programs? When the parallel programs solve large problems, will they fit into the memory of your shared-memory system? What about your distributed-memory system?

b. How much communication is required by each of the parallel algorithms?

c. Can the serial programs be easily parallelized by the use of OpenMP directives? Do they need synchronization constructs such as condition variables or read-write locks?

Compare your decisions with the actual performance of the programs. Did you make the right decisions?

6.7 PROGRAMMING ASSIGNMENTS

6.1. Look up the classical fourth-order Runge Kutta method for solving an ordinary differential equation. Use this method instead of Euler's method to estimate

the values of $\mathbf{s}_q(t)$ and $\mathbf{s}'_q(t)$. Modify the reduced versions of the serial n-body solver, either the Pthreads or the OpenMP n-body solver, and the MPI n-body solver. How does the output compare to the output using Euler's method? How does the performance of the two methods compare?

6.2. Modify the basic MPI n-body solver so that it uses a ring pass the instead of a call to `MPI_Allgather`. When a process receives the positions of particles assigned to another process, it computes *all* the forces resulting from interactions between its assigned particles and the received particles. After receiving `comm_sz` $-$ 1 sets of positions, each process should be able to compute the total force on each of its particles. How does the performance of this solver compare with the original basic MPI solver? How does its performance compare with the reduced MPI solver?

6.3. We can simulate a ring pass using shared-memory:

```
Compute loc_forces and tmp_forces due to my particle
    interactions;
Notify dest that tmp_forces are available;
for (phase = 1; phase < thread_count; phase++) {
   Wait for source to notify me that tmp_forces are available;
   Compute forces due to my particle interactions with
         ''received'' particles;
   Notify dest that tmp_forces are available;
}
Add my tmp_forces into my loc_forces;
```

To implement this, the main thread can allocate n storage locations for the total forces and n locations for the "temp" forces. Each thread will operate on the appropriate subset of locations in the two arrays. It's easiest to implement "notify" and "wait" using semaphores. The main thread can allocate a semaphore for each source-dest pair and initialize each semaphore to 0 (or "locked"). After a thread has computed the forces, it can call `sem_post` to notify the dest thread, and a thread can block in a call to `sem_wait` to wait for the availability of the next set of forces. Implement this scheme in Pthreads. How does its performance compare with the performance of the original reduced OpenMP/Pthreads solver? How does its performance compare with the reduced MPI solver? How does its memory usage compare with the reduced OpenMP/Pthreads solver? The reduced MPI solver?

6.4. The storage used in the reduced MPI n-body solver can be further reduced by having each process store only its n/`comm_sz` masses and communicating masses as well as positions and forces. This can be implemented by adding storage for an additional n/`comm_sz` `doubles` to the `tmp_data` array. How does this change affect the performance of the solver? How does the memory required by this program compare with the memory required for the original MPI n-body solver? How does its memory usage compare with the reduced OpenMP solver?

6.5. The `Terminated` function in the OpenMP dynamic implementation of tree search uses busy-waiting, which can be very wasteful of system resources. Ask a system guru if your Pthreads and OpenMP implementations can be used together in a single program. If so, modify the solution in the OpenMP dynamic implementation so that it uses Pthreads and condition variables for work redistribution and termination. How does the performance of this implementation compare with the performance of the original implementation?

6.6. The implementations of iterative tree search that we discussed used an array-based stack. Modify the implementation of either the Pthreads or OpenMP dynamic tree search program so that it uses a linked-list-based stack. How does the use of the linked list affect the performance?

Add a command-line argument for the "cutoff size" that was discussed briefly in the text. How does the use of a cutoff size affect the performance?

6.7. Use Pthreads or OpenMP to implement tree search in which there's a shared stack. As we discussed in the text, it would be very inefficient to have all calls to `Push` and `Pop` access a shared stack, so the program should also use a local stack for each thread. However, the `Push` function can occasionally push partial tours onto the shared stack, and the `Pop` function can pop several tours from the shared stack and push them onto the local stack, if the calling thread has run out of work. Thus, the program will need some additional input arguments:

- **a.** The frequency with which tours are pushed onto the shared stack. This can be an `int`. For example, if every 10th tour generated by a thread should be pushed onto the shared stack, then the command-line argument would be 10.
- **b.** A blocksize for pushing. There may be less contention if, rather than pushing a single tour onto the shared stack, a block of several tours is pushed.
- **c.** A blocksize for popping. If we pop a single tour from the shared stack when we run out of work, we may get too much contention for the shared stack.

How can a thread determine whether the program has terminated?

Implement this design and your termination detection with Pthreads or OpenMP. How do the various input arguments affect its performance? How does the optimal performance of this program compare with the optimal performance of the dynamically load-balanced code we discussed in the text?