CHAPTER

Parallel program development

7

In the last four 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 implementing a sorting algorithm called sample sort. 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-memory, distributed-memory, and CUDA 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."

7.1 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-memory systems, distributed-memory systems, and GPUs.

7.1.1 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

An Introduction to Parallel Programming. https://doi.org/10.1016/B978-0-12-804605-0.00014-2

by particle k is given by

$$\mathbf{f}_{qk}(t) = -\frac{Gm_q m_k}{\left|\mathbf{s}_q(t) - \mathbf{s}_k(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_k(t)\right]. \tag{7.1}$$

Here, G is the gravitational constant (6.673×10^{-11} m^3/(kg $\cdot$ 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 (7.1) 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

$$\mathbf{F}_q(t) = \sum_{\substack{k=0\\k\neq q}}^{n-1} \mathbf{f}_{qk} = -Gm_q \sum_{\substack{k=0\\k\neq q}}^{n-1} \frac{m_k}{\left|\mathbf{s}_q(t) - \mathbf{s}_k(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_k(t)\right]. \tag{7.2}$$

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 (7.2) to find the acceleration of particle q:

$$\mathbf{s}_q''(t) = -G\sum_{\substack{j=0\\j\neq q}}^{n-1} \frac{m_j}{\left|\mathbf{s}_q(t) - \mathbf{s}_j(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_j(t)\right]. \tag{7.3}$$

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 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.

7.1.2 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)
4          Print positions and velocities of particles;
5       for each particle q
6          Compute total force on q;
7       for each particle q
8          Compute position and velocity of q;
9    }
10   Print positions and velocities of particles;
```

We can use our formula for the total force on a particle (Formula (7.2)) to refine our pseudocode for the computation of the forces in Lines 5–6:

```
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 7.1. To better understand this pseudocode, imagine the individual forces

```
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 7.1: A reduced algorithm for computing n-body 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` 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 into `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]`.

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.

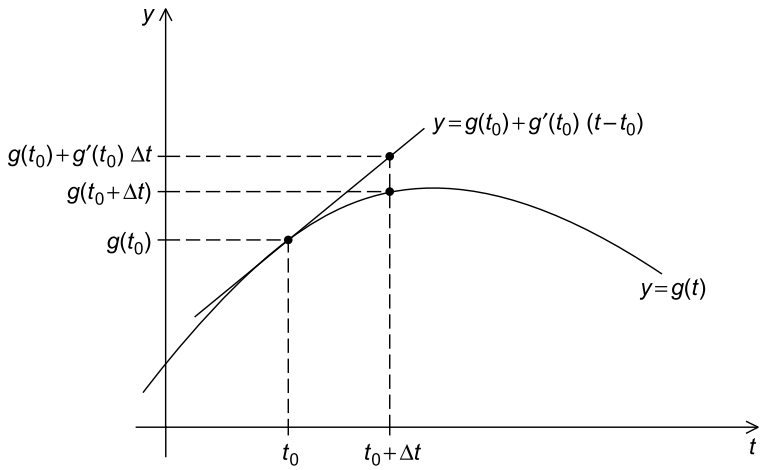

FIGURE 7.1

Using the tangent line to approximate a function.

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 (7.3). 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 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 Fig. 7.1 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

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

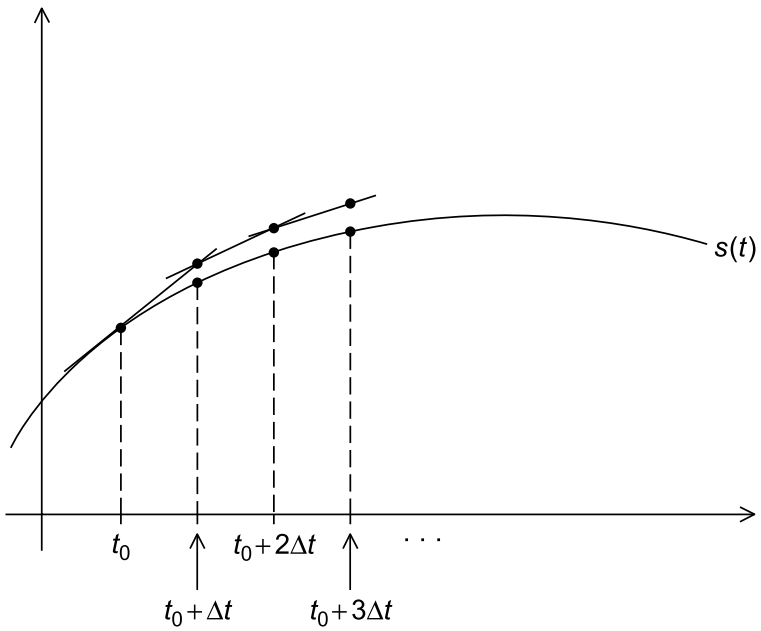

FIGURE 7.2

Euler's method.

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 $\mathbf{s}_q(\Delta t)$ 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 Fig. 7.2).

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.

7.1.3 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 (7.2), the 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 Fig. 7.3. 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 $\mathbf{s}_q(t)$, $\mathbf{v}_q(t)$,

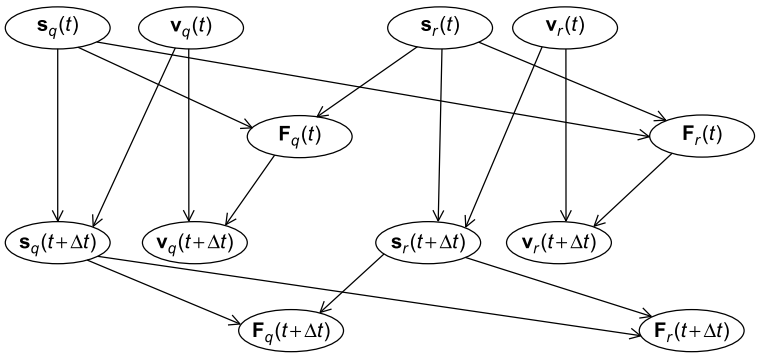

FIGURE 7.3

Communications among tasks in the basic n-body solver.

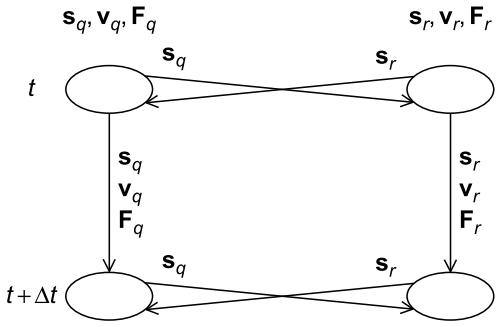

FIGURE 7.4

Communications among agglomerated tasks in the basic n-body solver.

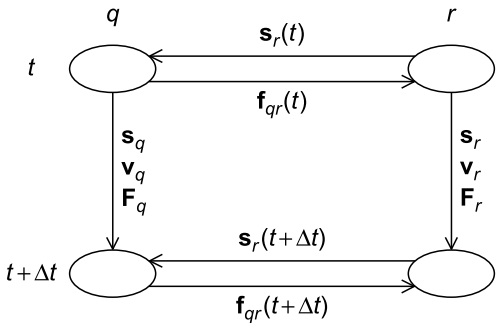

FIGURE 7.5

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

and $\mathbf{F}_q(t)$, our intertask communication is greatly simplified (see Fig. 7.4). Now the (agglomerated) 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.

Recall 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—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 Fig. 7.5.)

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 may 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. In principle, then, 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 partition 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 7.8 and 7.10).

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.

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

7.1.4 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.

7.1.5 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 to ensure that the iterations have a block partition:

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

7.1.6 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, since 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 /* Can be faster than memset */
   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 7.3.)

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

Table 7.1 First-phase computations for 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 7.2 First-phase computations for 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	$-\mathbf{f}_{05}-\mathbf{f}_{35}$	$-\mathbf{f}_{15}-\mathbf{f}_{45}$	$-\mathbf{f}_{25}$

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 7.4.)

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 7.1. 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 7.2 shows the same computations if we use a cyclic partition of the particles. 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.

Table 7.3 Run-times of the n-body solvers parallelized with OpenMP (times 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

7.1.7 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 7.11), 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 7.3 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 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.

7.1.8 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 **double**s, 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 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 versions, 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.

7.1.9 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, 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 MIMD 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 **double**s. 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 P should be stored in the appropriate block of each process' `pos` array:

```
P0: pos[0], pos[1], ... , pos[loc_n-1]
P1: pos[loc_n], pos[loc_n+1], ... , pos[loc_n + loc_n-1]
      ...
Pq: 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 7.2. As usual, process 0 will read and broadcast the command

```
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 7.2: Pseudocode for the MPI version of the basic n-body solver.

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 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 7.2. This ensures 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]
}
```

7.1.10 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. Fig. 7.6 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 surprisingly, the communications are even more complex when we use a cyclic distribution (see Exercise 7.14). 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 as being interconnected in a ring (see Fig. 7.7). 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.

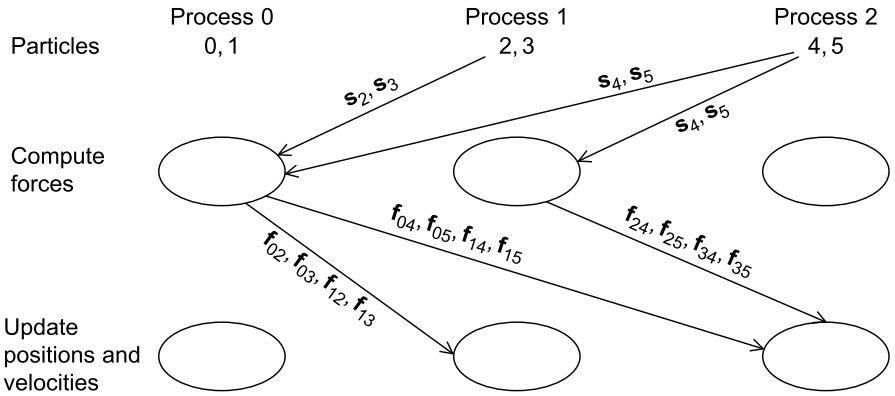

FIGURE 7.6

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

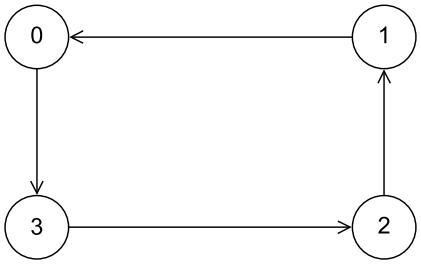

FIGURE 7.7

A ring of processes.

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 $(q + 1) \,\%\, \texttt{comm\_sz}$.

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 $\texttt{comm\_sz} - 1$ phases until each process has received the positions of all of the particles. Fig. 7.8 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 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

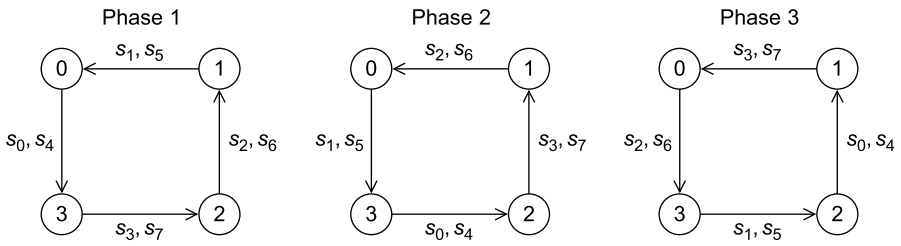

FIGURE 7.8

Ring pass of positions.

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, [17,37] 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 7.4). 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 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`.

Table 7.4 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	$0, -\mathbf{f}_{02} - \mathbf{f}_{12}$	$-\mathbf{f}_{01}, -\mathbf{f}_{03} - \mathbf{f}13 - \mathbf{f}_{23}$

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 7.3: 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 7.3.

Recall that using `MPI_Send` and `MPI_Recv` for the send-receive pairs in Lines 8–9 and 15–16 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 12–13 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 7.1), 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 7.16 and 7.17 for the other two functions.

Table 7.5 Performance of the MPI n-body solvers (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 7.6 Run-times for OpenMP and MPI n-body solvers (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

7.1.11 Performance of the MPI solvers

Table 7.5 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. 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 7.6). 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 **double**s per process for the masses. It also allocates $4n/p$ **double**s 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 $2pn + 2n$ **double**s for the forces and n **double**s 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 relative to p, 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.

7.1.12 Parallelizing the basic solver using CUDA

When we apply Foster's method to parallelizing the n-body solvers for GPUs, the first three steps are the same as they were for the general parallelization in Subsection 7.1.3. So our composite or agglomerated tasks correspond to the individual particles in the simulation. As before, this will greatly reduce the number of inter-task communications.

Now, however, the mapping stage is somewhat different. In the MIMD implementations of the n-body solvers, we expected the number of tasks or particles to be much greater than the number of available cores, and for a very large problem this will probably be the case for a GPU-solver as well. However, recall that for GPUs, we usually *want* many more threads than SIMD cores (or datapaths) so that we can hide the latency of slow operations, such as loads and stores. So when we implement the mapping phase for a CUDA solver, it may make sense to have just one task or particle per thread. Let's look at implementing this approach—one particle per thread—and

we'll take a look at multiple particles per thread in the Programming Assignments. (See Programming Assignment 7.5.)

With this assumption each thread will compute something like this:

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

However, there are a couple of race conditions in this pseudocode. Suppose threads A and B are executing the body of the loop for the same timestep. Also suppose that thread A updates the position of its particle *before* thread B starts the computation of the force on its particle. Then thread B will use the *wrong* value of the position of the particle assigned to thread A in the computation of the force on the particle assigned to B. So in any timestep no thread should proceed to the update of its particle's position and velocity until *after* all the threads have completed the updates to their particle's forces.

Now suppose that thread A has completed timestep t, and it has started computing the force on its particle in timestep $t+1$. Also suppose that thread B is still executing timestep t, and it hasn't yet updated its particle's position. Then thread A will use the wrong value for thread B's position in the computation of the force on its particle. So no thread should be able start timestep $t+1$ until all the threads have completed timestep t.

Both of these race conditions can be prevented by the use of barriers. If we have a barrier across all the threads after the computation of the forces, then the first race condition can't occur, since thread A won't be able to proceed to the update of its particle's position and velocity until all the threads—including B—have completed the computation of the total force on their particles. Similarly, if we have a barrier after the updates to the positions and velocities, then no thread can proceed to timestep $t+1$ until all the threads have completed their updates to the positions and velocities.

In Chapter 6 we saw that we could implement a barrier across *all* the threads by returning from one kernel and starting another. For example, suppose we wanted to place a barrier between calls to `Function_x` and `Function_y` in the following code:

```
/* Prototypes */
__device__ void Function_x(...);
__device__ void Function_y(...);
__global__ void My_kernel(...);
...
/* Host code */
...
My_kernel <<<blk_ct, th_per_blk>>> (...);

/* Device code */
__global__ void My_kernel(...) {
   Function_x(...);
   /* Want barrier here */
```

```
      Function_y(...);

      return;
   }
```

Then we can rewrite our code so that `Function_x` and `Function_y` are kernels:

```
   /* Prototypes */
   __global__ void Function_x(...);
   __global__ void Function_y(...);
   ...

   /* Host code */
   ...
   Function_x <<<blk_ct, th_per_blk>>> (...);
   Function_y <<<blk_ct, th_per_blk>>> (...);
   ...

   /* Device code */
   __global__ void Function_x(...) {
      ...
   }

   __global__ void Function_y(...) {
      ...
   }
```

In the second program the default behavior is for no thread to start executing `Function_y` until all the threads executing `Function_x` have returned.

Of course, in the general setting, there may be other code in `My_kernel`. In our case, the kernel with the race conditions might look something like this:

```
__global__ void Nbody_sim(vect_t forces[],
      struct particle_s curr[],
      double delta_t, int n, int n_steps) {
   int step;
   int my_particle = blkIdx.x*blkDim.x + threadIdx.x;

   for (step = 1; step <= n_steps; step++) {
      Compute_force(my_particle, forces, curr, n);
      Update_pos_vel(my_particle, forces, curr, n, delta_t);
   }
```

The only parts of the kernel that require the use of multiple threads are the calls to `Compute_force` and `Update_pos_vel`. So we can put the rest of the kernel in a *host* function[2]:

[2] In CUDA the `__host__` designation is the default. So we could omit it for this function.

```
__host__ void Nbody_sim(vect_t forces[],
      struct particle_s curr[],
      double delta_t, int n, int n_steps,
      int blk_ct, int th_per_blk) {
   int step;

   for (step = 1; step <= n_steps; step++) {
      Compute_force <<<blk_ct, th_per_blk>>> (
            forces, curr, n);
      Update_pos_vel <<<blk_ct, th_per_blk>>> (
            forces, curr, n, delta_t);
   }
   cudaDeviceSynchronize();
}
```

Now, instead of being device functions, `Compute_force` and `Update_pos_vel` are kernels, and instead of computing the particle indexes in `Nbody_sim`, we compute them in the new kernels.

In this code there is an implicit barrier before each call to `Update_pos_vel`. The call won't be started until all the threads executing `Compute_force` have returned. Also each call to `Compute_force` (except the first) will wait for the call to `Update_pos_vel` in the preceding iteration to complete.

The call to `cudaDeviceSynchronize` isn't needed, except as protection against accidentally executing host code that depends on the completion of the final call to `Update_pos_vel`. Recall that by default host code does *not* wait for the completion of a kernel.

7.1.13 A note on cooperative groups in CUDA

With the introduction of CUDA 9 and the Nvidia Pascal processor, it became possible to synchronize multiple thread blocks without starting a new kernel. The idea is to use a construct that is similar to an MPI communicator. In CUDA the new construct is called a **cooperative group**. A cooperative group consists of a collection of threads that can be synchronized with a function call that's made in a kernel or a device function.

To use cooperative groups, you must be running CUDA SDK 9 or later, and to use cooperative groups consisting of threads from more than one thread block, you must be using a Pascal processor or later.

Programming assignment 7.6 outlines how we might modify the basic CUDA solver so that it uses cooperative groups.

7.1.14 Performance of the basic CUDA n-body solver

Let's take a look at the performance of our basic CUDA n-body solver. We ran the CUDA program on an Nvidia-Pascal system (CUDA capability 6.1) with 20 SMs, 128 SPs per SM, and a 1.73 GHz clock. Table 7.7 shows the run-times. Every run was

Table 7.7 Run-times (in seconds) of basic CUDA n-body solver.

Blocks	Particles	Serial	CUDA
1	1024	7.01e-2	1.93e-3
2	2048	2.82e-1	2.89e-3
32	32,768	5.07e+1	7.34e-2
64	65,536	2.02e+2	2.94e-1
256	262,144	3.23e+3	2.24e+0
1024	1,048,576	5.20e+4	4.81e+1

with 1024 threads per block using **float** as the datatype. The serial times were taken on the host machine, which had an Intel Xeon Silver processor running at 2.1 GHz. Clearly, the CUDA implementation is *much* faster than the serial implementation. Indeed, the CUDA implementation is at least an order-of-magnitude faster, and it can be more than three orders of magnitude faster. Also note that the CUDA system is *much* more scalable than the serial implementation. As we increase the problem size, the run-times of the CUDA implementation usually increase much more slowly than the run-times of the serial implementation.

7.1.15 Improving the performance of the CUDA n-body solver

While the basic CUDA n-body solver is both faster and more scalable than the serial solver, its run-time *does* increase dramatically as the number of particles increases. For example, the time required for simulating 1,048,576 particles is nearly 25,000 times greater than the time required for 1024 particles. So we would like to further improve its performance.

If you've read the section on the OpenMP or Pthreads implementation, you'll recall that for these implementations, we used temporary storage for the calculations of the forces involving particles assigned to a particular thread. This allowed us to improve the overall performance by halving the number of calculations carried out by each thread. However, the implementations allocated storage for

$$2 \times \texttt{thread\_count} \times n$$

floating point values, and with n threads, this would require storage for $2n^2$ values. So if we were simulating a million particles, this would require storage for 2 *trillion* floating point values. If we use `floats`, this is 8 trillion bytes, and if we use `doubles`, it is 16 trillion bytes. We could probably reduce the amount of temporary storage by observing that we don't need to store n values for each particle (see Tables 7.1 and 7.2), but this observation reduces the temporary storage by less than a factor of 2. (See Exercise 7.9.) So we would still need storage for more than 1 trillion bytes. At the time of this writing (January 2020), our most powerful Nvidia processor (Turing) has about 25 gigabytes of global storage. So we'd need storage that's about 40 times larger than what's available to us now.

Another possibility is to use shared memory.

7.1.16 Using shared memory in the n-body solver

Recall that in CUDA *global* memory is accessible to all the SMs and hence all the threads. However, the time required to load data from or store data to global memory is quite large compared to operations on data in registers (e.g., arithmetic operations). To compute the force on particle q, the basic solver must access the positions and masses of *all* of the particles. In the worst case, this will require that we access global memory for all n particles. Nvidia processors do have an L2 cache that is shared among all the SMs and a per-SM L1 cache. So some of these accesses will not require loads from global memory, but we can do better.

Recall that each SM has a block of on-chip memory called *shared memory*. The storage in the shared memory is shared among the threads in a thread block that's running on that SM. Since shared memory is on the same chip as an SM, it is relatively small, but since it is on-chip, it is *much* faster than global memory. So if we can arrange that many of our loads are from shared memory instead of global memory, we may be able to provide significant performance improvements.

Since shared memory is *much* smaller than global memory, we can't hope to use shared memory to store a copy of *all* the data we need from global memory. An alternative is to partition the data we need from global memory into logical "tiles" that are small enough to fit into shared memory. Then each thread block can iterate through the tiles. After the first tile is loaded into shared memory, a thread block carries out all of its computations that make use of the data in the first tile. Then the second tile is loaded into shared memory, and the thread block carries out all of its computations that make use of the data in the second tile, etc. This gives us the following pseudocode:

```
1    for each tile {
2       Load tile into shared memory;
3       Barrier across the thread block;
4       Each thread carries out its computations
5          making use of the data in this tile;
6       Barrier across the thread block;
7    }
```

Effectively, the tiles are being used as cache lines, except that accessing them is managed by our program instead of the hardware. The first barrier ensures that we don't start the calculations in Lines 4–5 until all of the data from the tile has been loaded. The second barrier ensures that we don't start overwriting the data in the current tile until all the threads have finished using it.

To apply this pseudocode to the n-body problem, we load the masses and positions of a collection of m particles in line 2, and then in Lines 4–5 a thread block will compute the forces between *its* assigned particles and the m particles the block has just loaded. So we need to determine m.

If each thread loads the position and mass of a single particle, we can execute the loads without thread divergence. Furthermore, CUDA currently limits the number of threads in a thread block to 1024, and if we load the positions and masses of 1024

particles, each thread will load three `floats` or three `doubles`, and we'll need shared memory storage for at most

$$3 \times 8 \times 1024 = 24 \text{ Kbytes.}$$

Since current-generation Nvidia processors have at least 48 Kbytes of shared memory, we can easily load the positions and masses of $m =$ `th_per_blk` particles from global memory into shared memory. Finally, it's easy and natural to choose the particles in a tile to have contiguous indexes. So our tiles will be

- Tile 0: positions and masses of particles 0 through (`th_per_blk` $-$ 1)
- Tile 1: positions and masses of particles `th_per_blk` through (2 $\times$ `th_per_blk` $-$ 1)
- ...
- Tile k: positions and masses of particles ($k \times$ `th_per_blk`) through ((k + 1) $\times$ `th_per_blk` $-$ 1)
- ...
- Tile (`blk_ct` $-$ 1): positions and masses of particles ((`blk_ct` $-$ 1) $\times$ `th_per_blk`) through (`blk_ct` $\times$ `th_per_blk` $-$ 1)

In other words, we're using a *block* partition of the positions and masses to form the tiles.

This gives us the following pseudocode:

```
1   // Thread q computes total force on particle q
2   Load position and mass of particle q
3   F_q = 0;  /* Total force on particle q */
4   for each tile t = 0, 1, ..., blk_ct-1 {
5      // my_loc_rk = rank of thread in thread block
6      Load position and mass of particle
7         t*tile_sz + my_loc_rk into shared memory;
8      __syncthreads();
9
10     // All of the positions and masses for the particles in
11     //    tile t have been loaded into shared memory
12     for each particle k in tile t (except particle q) {
13        Load position and mass of k from shared memory;
14        Compute force f_qk of particle k on particle q;
15        F_q += f_qk;
16     }
17     // Don't start reading data from next tile until all
18     //   threads in this block are done with the current tile
19     __syncthreads();
20  }
21  Store F_q;
```

The code outside the **for** `each tile` loop is the same as the code outside the **for** `each particle` loop in the original basic solver: before the loop we load a position and a mass, and after the loop we store a force. The outer loop iterates through

the tiles, and, as we noted, each tile consists of `th_per_blk` particles. So each thread loads the position and mass of a single particle.

Before proceeding to the calculation of the forces, we call `__syncthreads()`. Recall that this is a barrier among the threads in a single thread block. So in any thread block, no thread will proceed to the computation of forces due to the particles in tile `t` until all of the data on these particles is in shared memory.

Once the data on the particles in tile `t` are loaded, each thread uses the inner **for** loop to compute the forces on its particle that are due to the particles in tile `t`. Note that if there are enough registers to store local variables, then all of these calculations will use data in registers or data in shared memory.

After each thread has computed the forces on its particle due to the particles in tile `t`, we have another call to `__syncthreads()`. This prevents any thread from continuing to the next iteration of the outer loop until all the threads in the thread block have completed the current iteration. This will ensure that the data on the particles in tile `t` isn't overwritten by data on particles in tile `t`+1 until all the threads in the block are done with the data on the particles in tile `t`.

Now let's look at how many words are loaded and stored by the original basic CUDA solver and the CUDA shared memory solver in the computation of the forces in a single timestep. Both solvers start by having each thread load the position and mass of a single particle, and both solvers complete the computation of the forces by storing the total force on each particle. So if there are n particles, both solvers will start by loading $3n$ words—two for each position, and one for each mass—and they'll store $2n$ words: two for each force.

Since these are the same for both solvers, we can just compare the number of words loaded and stored in the main loop of each of the solvers.

In the basic solver, each thread will iterate through *all* of the particles, except the thread's own particle. So each thread will load $3(n-1)$ words, and since we're assuming there are n threads, this will result in a total of $3n(n-1)$ words.

In the shared memory solver, each tile consists of `th_per_blk` particles, and there are `blk_ct` tiles. So each thread in each block will load 3 words in Lines 6–7. Thus each execution of these lines by each of the threads will result in loading $3n$ words. Since there are `blk_ct` tiles, we get a total of

$$3n \times \texttt{blk\_ct}$$

words loaded in the outer for loop. The loads in the inner for loop are from shared memory. So their cost is *much* less than the loads from global memory, and we have that the ratio of words loaded from global memory in the basic solver to the shared memory solver is

$$\frac{3n(n-1)}{3n \times \texttt{blk\_ct}} \approx \frac{n}{\texttt{blk\_ct}} = \text{th\_per\_blk}.$$

So, for example, if we have 512 threads per block, then the original, basic solver will load approximately 512 times as many words from global memory as the shared memory solver.