Lab 1b: Many-Body Simulation

In this lab you will learn to use Cilkview and Cilkscreen to debug your program (whether it's a "performance bug" or race). You will also see how a program can be parallelized differently, and depending on how the parallelism is organized, one way may be more efficient than the other.

Again, you should spend 4 hours or less on this lab, and you should try to do most of the work minus write-up by class on next Tuesday. Lab 1c will be handed out on next Tuesday, and the write-up is due on next Friday, which will require more of your time.

Although you should do all the steps of this lab, you need not submit documentation demonstrating that you did every step. Since the focus of the class is on independent discovery, you should feel free to investigate what you see as the interesting issues and document those. You should not wander too far from the topic of the lab, however. For example, for this part of the lab, it may be interesting to explore different ways to parallelize the N-Bodies simulation or dividing up the iteration space differently (than what's suggested in 1.3). Remember, however, this may end up costing you more than 4 hours. If you choose to do so, you should be conscious about how much time you are spending on this part of the lab.

Like lab 1a, we will upload this lab handout to NB, and we would like you to make at least one comment about some aspect of this lab using NB.

Getting Started

We have given you a git repository with the code for this portion of the lab. You can obtain this code using the following command:

```
git clone /afs/csail.mit.edu/proj/courses/6.884/spring10/labs/lab1b/ lab1b
```

Once this repository has been cloned, you should see one subdirectory within the new directory `lab1b`, called `nbodies`. The source for this portion of the lab is in this subdirectory.

1 N-Bodies Simulation

In this part, you will modify a program that simulates a set of planetary bodies drifting in space in the neighborhood of a single, massive "sun." Each body b_i (including the sun) has a mass m_i and an initial velocity v_i, and is attracted to another body b_j of mass m_j by a force f_{ij}, which obeys the formula for gravitational attraction,

$$f_{ij} = G\frac{m_i m_j}{r^2} ,$$

where G is the gravitational constant, and r is the distance between the bodies. The force on b_i is directional and pulls b_i towards b_j. The total force f_i on body b_i is the vector sum of the forces f_{ij} for all $j \neq i$.

The N-bodies simulation program begins by creating N bodies (the number N is specified on the command line) with randomly-selected masses in a random distribution around the sun. All of the bodies are given initial velocities such that the entire system appears to have a clockwise spin. A two-dimensional coordinate system is used for simplicity.

The simulation progresses by computing successive positions of each of the N bodies for successive moments in time using a two-pass algorithm. The first pass computes the force on each body as the sum of the forces exerted on it by all of the other bodies in the system. The second pass adjusts the velocity of each

body according to the force computed in the first pass and moves the body's position according to its average velocity during that quantum of time. Every few time quanta, a snapshot of the entire system is rendered as a picture in `png` format. You can view the result as a short movie using the provided JavaScript-powered web page. After running the `nbodies` binary, type `make publish` to copy the `nbodies.html` file and your PNG outputs to your CSAIL webpage. Then visit `http://people.csail.mit.edu/<username>/nbodies/` to view your movie.

Note that the execution times for these programs can be fairly long. Since Cilkview runs the program at least 5 times, expect to wait several minutes for a Cilkview run to complete. You may wish to reduce the number of images produced to save time. If you choose to do so, do not reduce the number of bodies in the simulation, or you will reduce the total parallelism in your program.

When running Cilkscreen to detect races, you can and should reduce the number of bodies and images produced to a bare minimum (e.g. 10 bodies and 2 images). Note that if you produce fewer images the visualization will be truncated, and the web page may display some parts of the previous run if there were leftover PNG files in your directory.

1.1 Burdened parallelism in the N-bodies simulation

The file `nbodies_loops.cilk` implements the n-body simulation. The `calculate_forces` function uses a pair of nested loops to compute the forces for every pair of bodies. The `update_positions` function uses a single loop to update the position of each body.

The `nbodies` binary accepts two arguments: the first is the number of bodies in the simulation (defaults to 300), and the second is the number of PNG frames to produce (defaults to 100). The number of simulation steps computed between frames is 40. All of these defaults and constants may be changed by modifying the `#defines` at the top of the file.

(a) Compile `nbodies_loops` using make, and run it with the default number of bodies (300). View the simulation in a web browser by running `make publish` and visiting:

`http://people.csail.mit.edu/<username>/nbodies/`.

Click the "Start" button to view the movie. (JavaScript must be enabled.) Parallelize the program by changing the loop in `update_positions()` and the inner loop in `calculate_forces()` to `cilk_for`. Run the program in Cilkview and report the results.

(b) Unlike in the matrix-multiplication example, you cannot parallelize the outer loop in `calculate_forces`, because doing so would cause multiple iterations to race on updating a single body's forces. (Try it in CilkScreen if you want.) However, you can invert the inner and outer loops of your current code so that the `cilk_for` loop over *i* is on the outside. Try it, and report your results from Cilkview. Which version is faster and why?

By default, the Cilk++ runtime divides the `cilk_for` loop into chunks containing one or more iterations. Each chunk is spawned, and code in a chunk is executed serially. The number of loop iterations in a chunk is called the ***grain size***. One can manually set the grain size by placing `cilk_grainsize` pragma just before the `cilk_for`:

```
#pragma cilk_grainsize = expression
```

If the grain size is not set, the Cilk++ runtimec chooses some sensible default value, based on the number of workers. You may try to use this pragma to set the grain size, and run the code

through CilkScreen. Depending on what you set the grain size to, you may see some difference in the amount of parallelism in the code.

1.2 Resolving races with locks

The formula for computing the gravitational force between two bodies b_i and b_j is symmetrical such that $f_{ji} = -f_{ij}$ (i.e., the magnitude of the force on both bodies is the same, but the direction is reversed.) Our current implementation of the N-bodies simulation, however, computes each force twice: once when computing the force that b_j applies to b_i, and again when computing the force that b_i applies to b_j. We can halve the total number of iterations in `calculate_forces` if we take advantage of this fact and compute the force only once for each pair of bodies.

In the file `nbodies_symmetric.cilk`, we have modified the inner loop in `calculate_forces` to avoid calculating forces that have already been computed in an earlier iteration (according to the serial ordering). When computing a force f_{ij} and adding it to f_i, we also compute the inverse force f_{ji} and add it to f_j. Unfortunately, this simple optimization has its problems, as we shall see.

(c) In `nbodies_symmetric`, parallelize `update_positions` and the outer (i) loop of `calculate_forces`. Compile `nbodies_symmetric` and run it with a command-line argument of 300. Did we see the expected speedup of 2 versus the (parallel) version of `nbodies_loops`? Run the program again in the CilkScreen race detector, but shorten the run time by using a command-line argument of 10 instead of 300. Where did the races come from, and why weren't they visible in the initial run?

(d) One way to fix the race is to use a *mutex* (mutual exclusion) lock to mediate concurrent access to each object. Add a member `mtx`, of type `cilk::mutex` to the `Body` struct. In `add_forces`, insert the statement "`b->mtx.lock();`" before updating `b->xf` and `b->yf` and insert "`b->mtx.unlock();`" after updating them. Run the program with Cilkview (using the original command-line argument of 300) and report the theoretical and actual speedup.

1.3 Solving races without locks

For sufficiently large data sets, the previous solution should produce little lock contention. Nevertheless, both locks and atomic instructions are expensive, even in the absence of contention, because they interrupt the CPU's pipeline and serialize operations that the CPU would have internally performed in parallel. It would be ideal if we could parallelize the N-bodies problem without introducing data races at all, thus eliminating the need for locks or atomic instructions.

Matteo Frigo, one of the authors of Cilk++, came up with one such solution that uses divide-and-conquer parallelism in a way that ensures that no two parallel strands will attempt to modify the same body. His algorithm, with the Cilk keywords removed, is implemented in the file `nbodies_nolocks.cilk`.

Figure 1 shows the core of the program. The lines labeled $[A]$ (Lines 44–45) can be executed in parallel with each other. Similarly, the lines labeled $[B]$ (lines 16–17) can be executed in parallel with each other, and the lines labeled $[C]$ (lines 18–19) can be executed in parallel with each other. This program is not meant to be obvious. Let's explore what it does.

The serial program is equivalent to calling `add_force(&bodies[i], fx, fy);` and `add_force(&bodies[j], fx, fy);` for all $0 \le i \le j < N$. Another way to look at it is that a plot of the points (i, j) such that $0 \le i \le j < N$ comprise the shaded area shown in Figure 2.

```
1  // update the force vectors on bi and bj exerted on each by the
       other.
2  void add_force(Body* b, double fx, double fy)
3  {
4      b->xf += fx;
5      b->yf += fy;
6  }
7
8  /* traverse the rectangle i0 <= i < i1,  j0 <= j < j1 */
9  void rect(int i0, int i1, int j0, int j1, Body *bodies)
10 {
11     int di = i1 - i0, dj = j1 - j0;
12     const int THRESHOLD = 16;
13     if (di > THRESHOLD && dj > THRESHOLD) {
14         int im = i0 + di / 2;
15         int jm = j0 + dj / 2;
16         rect(i0, im, j0, jm, bodies);  // [B]
17         rect(im, i1, jm, j1, bodies);  // [B]
18         rect(i0, im, jm, j1, bodies);  // [C]
19         rect(im, i1, j0, jm, bodies);  // [C]
20     }
21     else {
22         for (int i = i0; i < i1; ++i) {
23             for (int j = j0; j < j1; ++j) {
24                 // update the force vector on bodies[i] exerted
                       by bodies[j]
25                 // and, symmetrically, the force vector on
                       bodies[j] exerted
26                 // by bodies[i].
27                 if (i == j) continue;
28
29                 double fx, fy;
30                 calculate_force(&fx, &fy, bodies[i], bodies[j]);
31                 add_force(&bodies[i], fx, fy);
32                 add_force(&bodies[j], -fx, -fy);
33             }
34         }
35     }
36 }
37
38 // traverse the triangle n0 <= i <= j < n1
39 void triangle(int n0, int n1, Body *bodies)
40 {
41     int dn = n1 - n0;
42     if (dn > 1) {
43         int nm = n0 + dn / 2;
44         triangle(n0, nm, bodies);  // [A]
45         triangle(nm, n1, bodies);  // [A]
46         rect(n0, nm, nm, n1, bodies);
47     }
48     else if (dn == 1) {
49         // Do nothing.  A single body has no interaction with
               itself.
50     }
51 }
52
53 void calculate_forces(int nbodies, Body *bodies) {
54     triangle(0, nbodies, bodies);
55 }
```

Figure 1: A lock-free code for N-bodies.

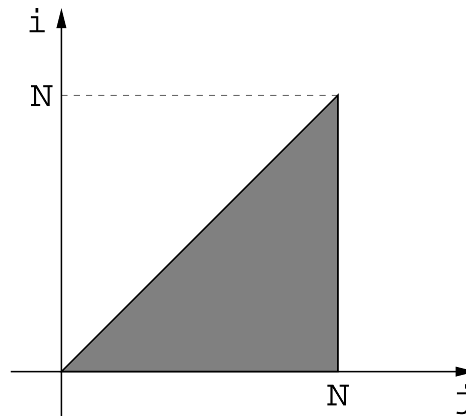

Figure 2: Traversing the space, $0 \leq i \leq j < N$

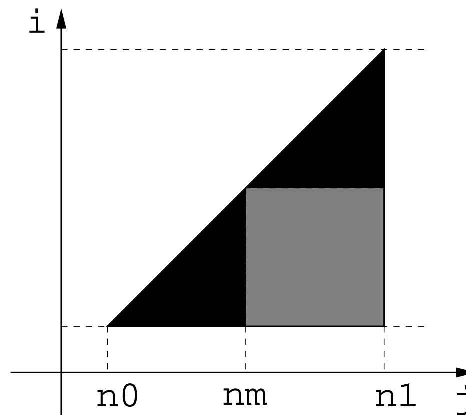

Figure 3: Cutting a triangle into two smaller triangles and a rectangle

Procedure `triangle` traverses this triangle in parallel, and in fact it is a little bit more general, because it traverses any triangle of the form $n0 \leq i \leq j < n1$. Initially, we set $n0 = 0$ and $n1 = N$ in `cilk_main`.

Procedure `triangle` works by recursively partitioning the triangle. If the triangle consists of only one point, then it visits the point $(n0, n0)$ directly. Otherwise, the procedure cuts the triangle into one rectangle and two triangles, as shown in Figure 3.

The two smaller triangles can be executed in parallel, because one consists only of points (i, j) such that $i < nm$ and $j < nm$, and the other consists only of points (i, j) such that $i \geq nm$ and $j \geq nm$. Thus, the two triangles update nonoverlapping regions of the `force` array, and thus they do not race with each other. However, the rectangle races with both triangles, and thus we need a `cilk_sync` statement before processing the rectangle.

To traverse a rectangle we use procedure `rect`, which also works recursively. Specifically, if the rectangle is large enough, the procedure cuts the rectangle $i_0 \leq i < i_1$, $j_0 \leq j < j_1$, into four smaller subrectangles, as shown in Figure 4.

The amazing thing is that the two black subrectangles can be traversed in parallel with each other without races. Similarly, the two gray subrectangles can be traversed in parallel with each other without races.

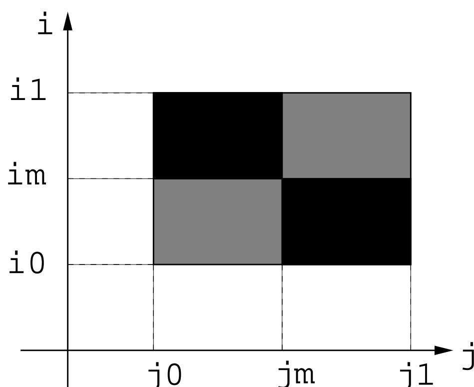

Figure 4: Dividing a rectangle into four

However, the black subrectangles race with the gray, so we must use a `cilk_sync` statement after processing the first pair of subrectangles.

To see why there are no races between the two black subrectangles (the same argument applies to the gray) observe that the i-ranges of the two subrectangles do not overlap, because one is smaller than i_m and the other is larger. For the same reason, the j-ranges do not overlap either. In order for races not to occur, however, we must also prove that the i-range of one subrectangle does not overlap with the j-range of the other, because we are updating both `bi` and `bj`. This property holds because when `triangle` calls `rect` initially, the i-range is $n0 \le i < nm$, whereas the j-range is $nm \le j < n1$, so the two ranges never overlap.

This algorithm partitions the original data into smaller and smaller subsets. Thus, in addition to avoiding races and locks, the algorithm exploits cache locality in a way similar to that of the recursive matrix-multiplication example.

(e) The file `nbodies_nolock.cilk` implements Matteo's algorithm as a serial program. Add `cilk_spawn` and `cilk_sync` statements to take advantage of the parallelism described in the previous section. Briefly describe the changes you made. Also add a `cilk_for` to parallelize the `update_positions` function.

(f) Confirm that the program is race free by running it in the race detector (use only 10 bodies to avoid long run times). Run the program in Cilkview and report the results. Compare the performance to runs of earlier versions of the program.