CS 140 Assignment 3: Simulating the N-body Problem

Assigned January 19, 2010

Due by 11:59 pm Monday, January 31

This assignment is to write an MPI program to simulate a large number of astronomical bodies (stars and planets) moving under the influence of gravity. You will write a parallel routine to generate the initial data in place, already distributed across the processors. For grading purposes, both correctness and performance (speed and scaling) will count.

1 Background

Kepler and Newton showed that two bodies orbit around each other in elliptical paths, in the absence of any other influence. For systems with three or more bodies, however, there is in general no closed-form equation for the paths they follow. Therefore, cosmologists need to use simulation to understand such phenomena as galactic evolution. (It's hard to set up a lab experiment with actual galaxies!)

We start with n bodies whose masses, positions, and velocities we specify. The basic n-body computation is to compute the force that each body exerts on each other body (n^2 forces in all); then compute how the sum of the forces on each body accelerates it (that is, changes its velocity); then compute, for each body, where it will move during a single "time step" at its new velocity. We repeat this computation over and over again, simulating one time step per iteration. The simulation isn't perfect, because we pretend that the velocities don't change in the middle of a time step; but we can make it more accurate by taking smaller and smaller time steps.

Here are the mathematical details. Suppose the n bodies have masses m_1, $m_2, \ldots, m_n$ (in kilograms); the initial position of the i-th body in 3-dimensional space is (x_i, y_i, z_i) (in meters); and that the initial velocity vector of the i-th body is $\vec{v_i} = (vx_i, vy_i, vz_i)$ (in meters/second). The force between bodies i and j comes from the gravitational law,

$$f = Gm_i m_j / r^2,$$

where r is the distance between them (in meters) and G is the gravitational constant (which is 6.67×10^{-11} in the units we're using). We convert this scalar force into a vector of forces in the (x, y, z) directions by

$$\vec{f_{ij}} = f \cdot ((x_i, y_i, z_i) - (x_j, y_j, z_j))/r.$$

Then, we compute the acceleration of body i due to body j from the vector of forces by using Newton's law $f = ma$. We add up all the accelerations on body i in this way,

$$\vec{a_i} \mathrel{-}= \vec{f_{ij}}/m_i,$$

to get the total acceleration on each body at this time step. Finally, for each body we use the acceleration to update the velocity for the time step (whose length is dt),

$$\vec{v}_i \mathrel{+}= dt \cdot \vec{a}_i,$$

and then use the new velocity to update the position for the time step,

$$xyz_i \mathrel{+}= dt \cdot \vec{v}_i.$$

There's a Matlab code that implements this linked to the course web site. You should try running the code in Matlab (`nbody(0)` simulates the solar system, `nbody(100)` simulates 100 random bodies) to see what it does—the Matlab code draws a picture at each time step. You should also read the Matlab code to see how the actual implementation of the steps above works.

It's actually hard to get a realistic solar system or galaxy simulation. If you take small enough time steps to follow Mercury accurately, you never see Pluto move, for example. Try changing the time step in the Matlab code from 1 week to 10 weeks just to see what happens—the inner planets get pretty wild. Also, if you want, you can try to find a more realistic initial set of random bodies than the ones the Matlab code generates.

2 What to implement

You will write a C / MPI code to do what the Matlab code does (minus the pictures, unless you want to get really fancy). You should write both `nbody()` and the data-generation routine `gennbody()`. You can assume that the number of bodies n is divisible by the number of processors p.

To debug your code, I suggest that you run it for only one or two time steps, and compare its output with the Matlab code. As always, first get it working on 1, then 2, then 4 processors, with a very small n. (I suggest that you modify the solar-system generator to omit 2 planets, and use the resulting 8-body problem as a debugging test on 1, 2, and 4 processors.)

3 Where's the data (and how does it move)?

You may assume that n, the number of bodies, is divisible by p, the number of processors. Each processor is responsible for n/p of the bodies; your `gennbody` should generate the data already distributed across the processors, and your `finalxyz` should end up distributed across the processors in the same way.

The data will move between processors in a pattern very similar to the merry-go-round 1-dimensional matrix multiplication algorithm presented in class (and on the class slides) on January 19. (Of course, the individual processor's computation is completely different.) You will probably want two complete copies of the data: One copy stays on its own processor, and the other copy moves around the other processors in a round-robin fashion, stepping from processor k to processor $k+1$, then $k+2$, and so on. You should use `MPI_Send` and `MPI_Recv` to move the data; you will want to move a whole block of n/p data points (where each data point is the 7 values representing one body) at once.

The parallel code will alternate between a communication phase, in which every processor sends a block of bodies to the next processor, and a computation phase, in which every processor computes the forces and updates the accelerations on its own bodies from the bodies in the block it's just received. Here's how one time step of the simulation goes: There are p computation phases (so every body sees every other body), interleaved with $p-1$ communication phases (in which

the round-robin data merry-go-round moves on to the next processor); then, once every body has accumulated acceleration from every other body, each processor updates the velocity and then the position of all of its own bodies, to finish off the time step. This whole thing is inside the loop over time steps.

4 What experiments to do

First, get your code thoroughly debugged, using the test inputs provided on the website along with the input/output specification. A large part of your grade will be based on the correctness of the output of your code.

Second, run timing and scaling experiments on randomly generated n-body problems. You do not need to implement the solar system. You should do timings of your code on 1, 2, and 4 processors for a range of values of n from, say, 20 up to the largest number you can run within a couple of minutes of wall clock time. If possible, you should also do a scaling study with a single large value of n and a range of larger numbers of processors p.

You should turn in plots of running time versus n and versus p, and also a plot that shows the parallel efficiency t_1/pt_p as a function of p for your large scaling study.

You can debug your code on any machine you like (for example, you can use CSIL for debugging with $p = 1$ and $p = 2$), but the results you turn in must come from runs on Triton.

Your grade will depend on correctness, on absolute performance on a large problem, and on how well your program scales.

5 What to turn in

Use `ssh` or `scp` to copy your files from Triton to CSIL, and then use `turnin hw3@cs140` from CSIL to turn in the following:

- Your source code.
- A `Makefile` that compiles your code.
- A report (as a text file or PDF, named `report.txt` or `report.pdf`, respectively) containing instructions for compiling and running your code, plus tables of your run time results, plus your plots, plus a description and interpretation of your results and any conclusions you draw from them.
- Your `turnin` command should look like:

  ```
  turnin hw3@cs140 Makefile report.pdf nbody.c ...
  ```

  and so on with whatever files you need.
- Don't turn in any executable or .o files.

You should do this assignment in a team of two people. Be sure to put the names of both members on your report.