N-body Simulation: A Dwarf of Parallel Computing

Richard Kirchofer

April 29, 2015

1 N-body Simulation

An n body simulation is a dwarf of computing where every item in the problem must be compared against every other item. n*(n-1)/2 total comparisons must be made. This is why the problem is described as having an n squared runtime.

2 Motivation

I began exploring this topic because it seemed interesting. I had written a small simulation of our solar system in Python but it was extremely slow. I wanted to expand upon that with the performance of C++. C++ is already a faster language and the parallelism takes it even further. Python is slow in part because it is an interpreted language but also it is not possible to multi-thread python applications because only one instance of the interpreter may run at a time on a system.

3 Expectation

I didn't have many expectations for this project. I knew that I would learn things along the way and I expected that I would find my focus as I worked on the project.

4 Methods

4.1 MPI

I began with a simple implementation using MPI since I had done the most work with MPI and felt most comfortable with it. I worked on this code for over a week, modifying the interactions of the particles. I learned the graphics library SFML which gave me instant feedback on what my simulation did. I could change the properties of the simulation and see the effect right away. Much of the work I did with the MPI code taught me how to create and maintain a graphics render window. I also defined the particle interactions. A difficulty that I faced was the unnecessary overhead of the MPI calls. It would be much simpler and efficient if I didn't have to send the data around. This is also unnecessary since I was just running the code on my own personal computer. If I wanted my code to run on multiple independent nodes, then I can spawn a process on each of the nodes and use MPI to coordinate the multiple processes. MPI allows me to pass data among the processes as well as synchronize the processes.

4.2 OpenMP

OpenMP, in contrast to MPI, allows a program to run in multiple threads. This speeds up computation without the overhead of MPI. It avoids the overhead of MPI message passing by using shared memory. Because all of the threads of execution are on the same machine then the program can execute without duplicating any of the memory. Each thread may operate on the same shared data. It was very easy to substitute OpenMP into the code. OpenMP allows me to parallelize certain portions of the code while other portions remain serial.

4.3 Barnes-Hut

The Barnes-Hut model is an alternate way to arrange the data to allow for effective approximation. This sacrifices the accuracy of the simulation for a faster runtime. It uses quad-trees to break up the work. The basic rule to follow when dividing up the data is that no two points may sit in the same quadrant. The data structure used normally consists of internal nodes, external nodes, and points. The points have their location, velocity,

and acceleration. The internal nodes hold at most four items where the items consist of some combination of points and external nodes. The external nodes are just placeholder nodes. They are on the edge of the graph because they hold no nodes themselves. An internal node represents an empty quadrant. In my implementation, there is no such thing as an external node, the internal nodes simply have null pointers.

The Barnes-Hut model is able to reduce the number of calculations because it does not have to inspect all of the points. The internal nodes of the graph keep track of the aggregate mass of the points within it as well as the coordinates of the center of mass. If one point is far enough away from another node then the simulation will just calculate the attraction of the point to the node instead of the point to all of the other points that the node represents.

5 Observation

In order to keep the particles on the screen so that I could observe them, I first just displayed them at the modulus of their actual position. This let me see the particles and watch their interactions without actually altering the simulation. It became difficult to monitor them because I couldn't tell whether the particles were actually close to each other in space or if they were just being drawn next to each other on the display. A simple solution to this would be to set bounds on the space. The best way to do this would be to connect the top of the space to the bottom of the space and also to connect the left side of the space to the right side of the space. This would create a torus and I could watch the simulation play out in the space of my screen.

Before I created a torus, I added a quick fix. If the particle wandered off the bottom of the screen then I would move it to the top of the screen. It's almost the same as a torus except that while the particles may travel across the boundary, their attractions do not. I noticed some interesting behavior in this simulation.

If the simulation space was infinite or if it was a true torus then the temperature of the system could only decrease. The collisions are less than elastic so with every collision they lose momentum. By not allowing the particle attractions to warp around the edges of the bounding box, I violate conservation of momentum. If there is a dense cluster of particles near the

right edge of the screen then it is very easy for orbiting particles to fall off of the edge. When they move across the boundary, they appear again on the left side of the screen without losing their momentum. In this way, they only ever gain speed as they are attracted in only one direction.

This creates a thick belt of fast moving particles. They are all attracted to the same dense cluster of particles sitting on the right side of the screen. This dense cluster only has particles to its left so it is attracted in that direction. It slowly migrates towards the left putting more space between it and the right boundary of the simulation. Eventually the fast moving particles stop accelerating. With the dense cluster in the middle of the screen, they are accelerating for as much time as they are decelerating. If the simulation has any friction then they will all slow down and reach equilibrium.

This behavior revealed another flaw in the simulation. The fast moving particles would rarely collide with the dense cluster and instead skip over it. This is because I move them by adding their velocities to the current position of the particles. If they are moving fast then they skip over a large region of space.

6 Improvements

Not all collisions are recorded in this simulation. If the particles are moving fast enough then they may skip past each other. I can modify the simulation to remedy this situation. If I save the locations of the particles at the previous time step then I can identify all of the potential missed collisions. For each of the places where the paths of two particles crossed, I check to see if they were in the same place at the same time and then know if the two particles should have collided.

I also plan to replace SFML with SDL, add SIMD intrinsics, and implement the project in CUDA. SDL is a more polished library with a larger community, a larger development team, and more features. SDL is also cross-platform and has broader applications. SFML introduced me to graphics but I can take it further by learning SDL. SIMD intrinsics definitely belong in this code. The coordinate locations can be substituted with a SIMD vector and SIMD addition will speed up the simulation. I would also like to learn CUDA and this is a suitable application for manycore GPUs.

7 Benchmark

I benchmarked the program using valgrind by Julian Seward. Valgrind with the tool callgrind will run the program in a virtual environment and log all of the program's calls. The step that currently takes the most time is the move function call. Updating the data of all of the particles took over 99% of the time. This seems difficult to believe so I plan to test this further before I claim it as a conclusive result.

The program demonstrates moderate scaling in both strong and weak scaling tests.

7.1 Weak Scaling

number of particle - number of processors - time
500 - 1 - 0.386048
1000 - 2 - 0.774975
1500 - 3 - 1.21484
2000 - 4 - 1.62167

7.2 Strong Scaling

number of particle - number of processors - time
2000 - 1 - 5.74465
2000 - 2 - 3.02172
2000 - 3 - 2.13861
2000 - 4 - 2.09139

8 Conclusion

I still have a great deal of work to do on this project.

The github repository with the current code.

`https://github.com/rgkirch/N-Body-Particle-Simulation`

A video of 10k particles through 1k steps.

`https://youtu.be/IxI9vtj0Aw8`