Lillie Hammer and Nick Silvestri

CSC 422: N-bodies project

Introduction

The program we developed was an n-body simulation. The n-body problem is important in physics because simulating the interactions between 2 or fewer bodies can be perfectly mathematically modelled to any given point in time in the future. However, once 3 or more bodies are part of the system, the system can very quickly become chaotic and impossible to predict, short of simulating the bodies moving for a certain (usually very small) amount of time until the given point in time desired. Even this approach has its problems, as eventually the small errors accumulated in the system over time can cause the system to deviate from reality. The best comparison of this problem is like that of the integral; finding the area under the curve can be approximated better and better with rectangles of a smaller and smaller base length, but in this case it is impossible to have infinitely small base length rectangles.

This report will show our implementation of the n-body problem in Java, and how we optimized our application to more quickly calculate iterations of the program.

Programs

Optimization can only go so far in this project. Forces on each body must be recalculated according to every other body, resulting in $O(n^2)$ time complexity. Collisions must also be checked against every other body, which again runs in $O(n^2)$ time.

With a sequential program, there is only one thread doing all of the work. Though this may still be done in seemingly little time for a certain number of bodies and timesteps, when the number of bodies goes up by a factor of two, the time complexity will increase by a factor of four. This is not very efficient for one thread to do all alone.
With multiple threads, we can divide up the work between them. There are smarter ways to do this, which we took advantage of. In calculating forces and checking for collisions, the first body must check against every other single body. However, the second body can ignore the first body, as the first body already did the checking for that pair. Therefore, the number of other bodies each one has to check against decreases. By having the for loop go in stripes, so that Process0 checks the first body, Process1 checks the second body, etc, we more evenly distribute the work that each process has to do.

To run our program, use the included Makefile command “make all”. Use the command “java ParallelCollisions <numThreads> <numBodies> <body radius> <numTimesteps>”. This will run our program with <numBodies> bodies placed in random positions, with the given radius and a mass of radius^3, and randomly-generated initial velocities. By default, a dissemination barrier is used and there is no border bounding the bodies. If the command “java ParallelCollisions <numThreads> <numBodies> <body radius> <numTimesteps> y” is used, prompts will follow asking if the user wants to use a dissemination or cyclic barrier; if the user wants to add a border (so bodies will be bounded, causing more collisions); and if the user wants the bodies to start with an initial velocity of zero.

Our program will print the settings that it is running with so there is no confusion. At the end, our program prints the total running time, collisions detected, and how long on average processes spent inside of barriers. Our program writes all of the bodies’ starting positions and velocities to a file called StartingBodies.txt, and writes the bodies’ ending positions and velocities to a file called FinalBodies.txt.

To run our program with graphics, use the files “Body.java”, “ParallelCollisionsGUI.java” and “SpaceThreadsGUI.java”. We used Eclipse to test the graphics, but were not able to run timing tests while ssh’ed into a department machine.

Verification

We were able to verify that our program was working correctly by including a graphical interface. If some interaction between any bodies was very wrong, it would be immediately apparent through the GUI. Smaller errors were checked manually with physics equations; particularly $F = Gm_1m_2/r^2$ and $v = \sqrt{Gm/r}$. With the first equation, we could determine whether the forces acting on each body was accurate with their distance, mass, and our gravitational constant (which was 6.67 x 10^{-2}, 10^9 times stronger than it is in real life). The second equation allowed us to verify the motions of the bodies. With a body of mass M, we could create a body of mass m<<M at a distance r from the larger body, give it a velocity v tangential to a circle centered around the larger body, and verify that its orbit was approximately circular (or elliptical with very low eccentricity). Furthermore, we had unit tests set up for several of our calculation methods, allowing us to verify that none of the modifications we made changed the accuracy of our calculations.

Timing Experiments

For our extra experiments, we tested how using different barriers affects run time, and how much overhead having more threads creates inside of which barrier we are using. We ran all of the following tests on Oxford, without graphics (as the visual GUI could not compile when ssh’ed into that machine), using a variable number of threads, 2000 bodies with a radius of 10, and 1000 timesteps. We also included a boundary border so that bodies could not escape past a certain point.
We found that as the number of threads increased, the overhead that having a barrier creates drastically increases as well. With 2-4 threads, each process was spending on average less than a second inside of barrier calls. At 30 threads, each process was spending on average about 20 seconds inside of barrier calls. This may have been caused by uneven distribution of work. Processes that got to a barrier first would need to wait until all of the other processes arrived as well; if one thread did significantly less work than the others, it would be waiting inside of the barrier for longer.

	Cyclic 1	Cyclic 2	Cyclic 3	Cyclic Avg	Dissem 1	Dissem 2	Dissem 3	Dissem Avg
2 threads	0.7079407	0.5173062	1.4854016	0.9035495	0.5910755	0.6193973	1.5816412	0.9307047
4 threads	0.5595718	1.3390636	0.9400718	0.9462357	0.5220787	0.5220787	1.0999223	0.7146932
8 threads	2.6136477	2.7095943	2.7924304	2.7052242	2.1575090	2.1928203	2.3306584	2.2269959
20 thread	25.1002314	25.0650224	20.1441873	23.4364804	22.0471982	22.0579330	19.8148931	21.3066748
30 thread	28.8872766	22.6092511	24.5665168	25.3543482	22.4699806	23.4547614	21.1718442	22.3655287

Fig 1A: Table comparison of time (seconds) spent in barriers

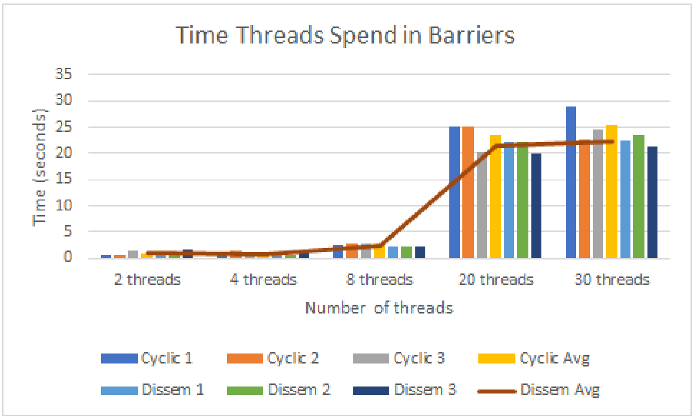

Fig 1B: Graph comparison of time spent in barriers

We did find that as the number of threads increased, the overall running time of our program improved. The best performance happened at around 8 threads, with an average run time of about 30 seconds.

	Cyclic 1	Cyclic 2	Cyclic 3	Cyclic Avg	Dissem 1	Dissem 2	Dissem 3	Dissem Avg
1 thread	147.0	147.0	148.0	147.3	146.0	147.0	147.0	146.7
2 threads	81.0	81.0	79.0	80.3	81.0	81.0	81.0	81.0
4 threads	45.0	44.0	43.0	44.0	44.0	45.0	43.0	44.0
8 threads	30.0	31.0	29.0	30.0	30.0	30.0	30.0	30.0
20 threads	40.0	31.0	35.0	35.3	36.0	36.0	31.0	34.3
30 threads	39.0	30.0	33.0	34.0	31.0	32.0	28.0	30.3

Fig 2A: Table comparison of overall run-time

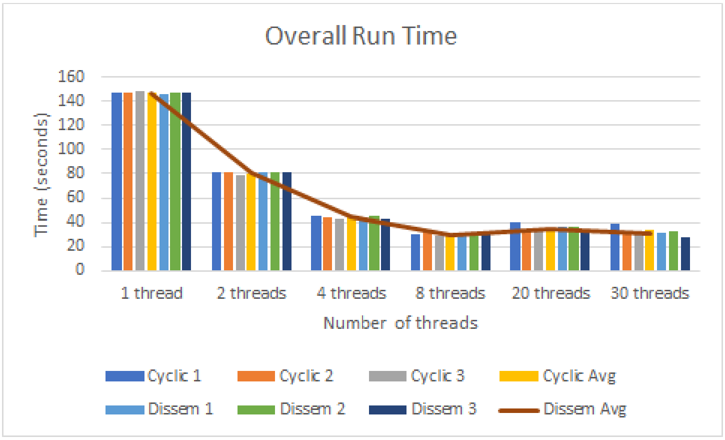

Fig 2B: Graph comparison of overall run time with cyclic vs dissemination barrier

Other Experiments

This project allowed us to examine how differing setups could affect performance. For example, we learned how dissemination barriers work using semaphores in class, but Java has a built-in CyclicBarrier class. Would changing the way we implement the barrier affect the overall time? After calculating the average time each process spends inside of the barrier method, we found that though there is in general a slight improvement using a dissemination barrier over a cyclic barrier, the improvement is not significant in overall run time. Therefore, if a dissemination barrier is difficult to implement in a program but a cyclic barrier is built-in, using the cyclic barrier instead would not hurt run-time.

As the number of processes increased, the average time spent inside of both barriers also increased (see Fig 1B). After 8 threads, this time drastically increased. This was likely the cause of the ~30-second overall run-time plateau that occurred after 8 threads (see Fig 2B). As the barrier is a necessary function call with multi-threading, there did not seem to be a workaround for this. There does seem to a be a point where increasing the number of threads does not significantly improve performance anymore, and at this point may be wasting resources. It would be beneficial to find this and stay below it, so that both time and memory are used most efficiently.

We also attempted to examine how adding visual graphics affects performance. Though the graphics were useful in confirming that our program worked as intended with a small number of bodies, we could not optimize it to be practical when working with a large number of them. We used a JavaFX canvas to draw each body, but only one thread could use the canvas, so it was in essence a sequential for loop. The canvas could only show so much of the area where bodies were in as well, so if there was no border and bodies were outside of the field of view, it made for an uninteresting sight. When we did add a border to contain the bodies inside the field of view, the additional collisions it caused increased the overall energy within the system, and it became impossible to keep track of the bodies.

Conclusion

We learned that though a barrier is necessary with multiple threads, using a certain type of barrier over another does not significantly improve performance. In fact, a barrier can eat much of the overall run time when the number of threads increases. We also learned that certain aspects of a project can be useful in the beginning, but may not necessarily be as useful near the end, as with having a graphical component.