James Ni

CSC357: Parallel and Concurrent Programming

Final Project

December 15, 2016

Parallel N-Body Simulations using OpenMPI and the Barnes-Hut Algorithm

I aimed to implement the Barnes-Hut algorithm for an N-Body simulation using Java and MPI for my final project. N-Body simulations are computationally intensive, as the direct and accurate brute force method of calculating the sum of the forces between one particle to every other particle obviously has a runtime Θ(N^2), where N is the number of bodies in the simulation. The large runtime complexity of this simple brute force method, along with the fact that each body is separate from one another outside of their force interacts, strongly suggested to me that N-Body simulations are problems that naturally tends towards parallelization.

Of course, it is important to consider that other algorithms exist for N-Body simulation computation in a sequentially manner that is more efficient than the All-pairs brute force implementation. There are several such algorithms, but the one I choose to implement and parallelize is the Barnes-Hut Algorithm, an algorithm for N-Body force calculations which has an O(N log[N]) runtime in general. The Barnes-Hut can also degenerate into the All-pairs algorithm, so implementing it will also naturally and easily allow me to test the All-pairs algorithm upon successful implementation.

While there are other methods that can be implemented for the N-Body simulation, Blelloch and Narlikar performed a comparison of some N-Body algorithms.[1] It is noted that the other algorithms analyzed are significantly more complicated in implementation. Most notably,

[1] Girija Narlikar and Guy Blelloch, "Parallel N-Body Simulations," http://www.cs.cmu.edu/~scandal/alg/nbody.html. accessed December 15, 2016.

Greengard's Fast Multipole Method performs better than the Barnes-Hut implementation except for low-accuracy modelling (Root mean square error on the order of 10^{-3}). The Barnes-Hut algorithm, however, performs the best for low-accuracy gravitational force simulations with values of N larger than 10^{8} and is the simplest implementation. As I am focusing on the gravitational N-Body simulation first, it is thus most feasible for me to begin by implementing simpler algorithms such as the Barnes-Hut algorithm.

The Barnes-Hut algorithm partitions the problem into a hierarchical tree structure for lower level calculations, progressing up to higher level calculations. It compares each particle to another cluster of particles, progressively decreasing the scope of the size of the cluster. Clusters are determined with a square, or cubic in three-dimensions, size of an area, recursively made into four smaller squares until the resulting cell contains only one mass. While this method lowers the accuracy of the simulation, the implementation can be modified for higher and lower levels of accuracy in exchange for runtime complexity. The value which modifies this is a threshold ratio of the size of the space which the area entails divided by the distance to that space. In my implementation, this is called the threshold in the main NBodyBH.java file, and theta in the Barnes Hut tree BHTree.java file, which I inputted as a Parameter to my Barnes-Hut Tree (line 98). I initialized the threshold at line 40, defaulting to .5. It is named Theta in the BHTree implementation to maintain some integrity and similarity to an implementation of the Two-Dimensional Barnes-Hut Tree with four quadrants by chindesaurus, whose base sequential implementation I borrowed and extended. If this threshold value is set to 0, the algorithm will degenerate into the All-pairs algorithm.[2]

[2] Chindesaurus, "An O(N log N) N-Body simulation using the Barnes-Hut algorithm," https://github.com/chindesaurus/BarnesHut-N-Body, accessed December 15, 2016.

For the parallelization of this algorithm, I focused on the issue of where the memory requires for a large number of masses may not fit onto one machine's processor or memory. This will be reflected in some design choices detailed later. I choose to utilize the Message Passing Interface (MPI) standard OpenMPI in my implementation, with my decision lying in familiarity from the previous assignment and being a sufficiently good fit for the task desired. Another parallel programming API I have some familiarity with is Apache's Spark, but the fine-grained computation needed would be difficult to handle with Spark. With further investigations onto MPI's API, I found that it was sufficiently comprehensive to handle all the inter-machine or inter-processor communication, with some compromises in the Java implementation.

As mentioned, the base sequential implementation of the Barnes-Hut algorithm was provided by chindesaurus from github. This thus provided a visualization method in the StdDraw.java, a Quad.java class for the Quadrants, a BHTree.java class for the Barnes-Hut Tree implementation, a Body,java class for celestial bodies, and a main class in the NBodyBH.java class. I extended not just the main class, but also some of the supplementary classes to fit my implementation.[3]

The Body class implements an object which holds a point's position, velocity, mass, forces acting on it, and a color for visualization. It provides a way to update its position and velocity as well as calculate resulting forces. There is also a method to sum up bodies, to calculate center of mass and total mass for the Barnes-Hut Tree in the plus(Body) method. I extended this class slightly to include the addForce(double, double) method, allowing me to sum resulting forces x and y-axis forces calculated on other machines. I also included field accessor

[3] Ibid.

methods to be able to transmit position and mass, as well as force when calculated on another machine.

The Quad class implements a quadrant system of splitting and defines four different regions of space for the Barnes-Hut Tree. It contains methods to find each Quadrant, using cardinal directions North-West, North-East, South-West, and South-East to designate each subregion. It contains methods to check if certain coordinates exist within the range of the Quad’s sectioned space.

The BHTree class implements the Barnes-Hut tree. As mentioned described, the Barnes-Hut tree successively places all masses into the tree, and calculates the resulting center of mass and total mass for objects within the space enclosed a quadrant. Each successive level of the Barnes-Hut Tree represents a smaller space than the previous, with the area enclosed lowering by a factor of ¼. The Barnes-Hut algorithm creates this tree at each time step stage of the simulation, which is then used to calculate the resulting force of an area on a particle. I modified this class to accept a ratio as a parameter upon creation, so different threshold values can be tested on the algorithm, including the All-pairs threshold value of 0. I also modified its force calculation method to stop calculations upon hitting a certain number of recursive reads. Here, it is hard coded as having a maximum of 10 levels. This has the possibility to further lower the time complexity of very large datasets, and 10 levels would thus rescale the area by a factor of $(2^{10})^2 = 2^{20}$, or on the order of a millionth of the size of the total area, which should still provide accurate simulations.

Finally, the parallelization occurs in the main client class, NBodyBH.java. Originally, the implementation read data points from a file. For the purpose of focusing on the parallelism of this assignment, I simplified this to simply generating data points for simulation in each

Quadrant. I also initialize the data points that exists on a machine's memory to only exist in one quadrant initially, further simplifying the base problem. This is seen in my initialization stage, corresponding to lines of code from line 32 to line 78, or optionally, line 90. I initialize a very large mass at the center of the system to easily visualize that each processor is actively exchanging data and doing proper calculations. I also include an alternative, commented out, for initializing a large mass at the center of each quadrant.

With all points successfully initialized, the simulation functions per time-step by creating Barnes-Hut Trees and placing all masses still within the scope of the visualized screen into the trees, then recalculating the force interactions per particle with all masses within its memory. Once it finishes calculating all forces within its local memory, it sends the current position of its masses forward onto successive processes based on rank, and receives the current position of masses in other machine's memory from previous processes in a cyclic manner. This allows every process to continually calculating force, efficiently utilizing computational resources and inter-machine bandwidth, as no process is left waiting for information from another process. I choose this method of message in the MPI command sendRecvReplace() as I was concerned about the space complexity with which receiving all the data points all once would result in due to MPI calls on functions such as Broadcast or AllToAll. sendRecvReplace() is a blocking transfer function, but the alternative of using a non-blocking send and a blocking receive would have functioned in exactly the same manner. This allows me to limit how the space requirements of my program to scales a function of two times the number of masses allocated to each machine process, rather than the total number of masses in all processes. Each process calculates the force resulting from the Barnes-Hut algorithm of masses within its memory after receiving the position and mass information from another process, then sends it back and receives the force calculated from another process. The received forced are summed onto the local masses', until all machine's masses have been accounted for. Finally, the resulting force is then used to update the velocity and then position of the masses.

In the implementation, I also replace the resulting masses into a buffer, and then call an allGather() so all points are visible on all processes. This is only for visualization purposes, and thus I was not concerned with the time or space complexity of the algorithm in this portion of the code. This portion can easily be omitted, and some other form of output, such as writing to a file, can be done instead if desired. As my test simulation had a large mass in the center, I also colored that mass different for my visualization.

Due to the large space and time complexity of communication of the visualization portion, it is difficult to fully analyze the contribution of the Barnes-Hut algorithm in comparison to the All-pairs algorithms if I set the threshold ratio to 0. For 4 processes, the simulation runs relatively smoothly. As the number of processes increases, the simulation slows down significantly, but it seems likely by my analysis that this is much more due to the increased communication of the visualization step rather than the computation step.

An odd result that I received that changed based on the ordering of masses I receive and MPI communication call was that with this more efficient cyclic pattern, the central heavy mass occasionally flies off at rapid speeds. By my analysis, this is not likely due to be a communication error between the points, as it functions well before and after, but instead due to many masses approaching very closely to the central mass and applying a very large force to the central mass, forcing it to speed out of the frame. If this is not the cause, there may be some communication error in MPI due to the calculated force or mass of surrounding particles. Oddly enough, this did not occur when I had a less efficient, more sequential message passing pattern;

however, I do not see a flaw in the logic which would cause this conditionally. It also does not occur on run of the simulation.

From this project, I was able to delve further into the MPI API and successfully program a parallel N-Body simulation with the Barnes-Hut algorithm in Java with OpenMPI. The parallel implement when used on my relatively small randomly generated data set is satisfactorily efficient in its runtime outside of the space and runtime complexity used in this version of the visualization. This parallel algorithm I developed is also more intended for larger datasets where memory allocation onto one computer can be an issue, such that the message passing cost is less significant to the per-step computational costs. I invite anyone who wishes to further this parallel algorithm to utilize the code with inputted data from a file and thus utilizing it on very large data sets over multiple cores or machines as designed and intended for, as well as to find and use alternative visualizations such as outputting per time-step positions onto a file and using an alternative program to read the file and produce the visualization.

Works Cited

Narlikar, Girija and Guy Blelloch. “Parallel N-Body Simulations.”

http://www.cs.cmu.edu/~scandal/alg/nbody.html. Accessed December 15, 2016.

chindesaurus. “An O(N log N) N-Body simulation using the Barnes-Hut algorithm.”

https://github.com/chindesaurus/BarnesHut-N-Body. Accessed December 15, 2016.