Abraham Zamora

AZ284

Lab 5: MPI-Barnes-Hut

Approaching the project

The first approach was to understand how the N-Body Barnes-Hut algorithm worked. Luckily, there are plenty of sources. Tom Ventimiglia and Kevin Wayne provide a good overview of the problem[1]. This also helped to understand the type of data structure that would be needed for the lab. The Quadtree provided the best data structure for the project, as it can represent a 2D space from 0 to 4 within the X and Y axis. My implementation was adapted from two sources, cited in the code but provided here as well.[2][3] In addition, I needed to better understand the MPI standard in order to parallelize the process. I found the docs and a few other sources useful in this regard.[4][5]

Before any optimizations were added (these are discussed later in the report), let's discuss the decision making that went into the program. The io and argparse files are adapted largely from starter code provided in the first lab. In the parameters, I omitted the optional visualization, so running with -v will not do anything. I made this decision to not add visualization given the fact that I was doing this on Codio. I have experience with OpenGL, so it would have been interesting to do the visualizations. But the limitations of the system and the time constraints in trying to implement the visualizations after working through the rest. The io and argparse files help in parsing command line arguments, reading the files, and outputting the result files. The helpers.cpp file contains the distance calculation. The bodies file contains the Leapfrog-Verlet integration, and the header file defines the struct that contains the body in question (id, x position, y pos, velocities, and mass). The Quadtree file contains the quadtree class, and the associated functions, including inserting new bodies into the quadtree, updating the center of mass, subdividing the tree, and calculating the net force. The net force calculations are provided in the lab instructions and adapted here. The main file contains the parallelization. The overall work flow is as follows.

1. Root process initializes and broadcasts to all processes.
2. Each process computes forces and updates the assigned particles.
3. Updated data is gathered and synchronized at the root process.
4. Workload is redistributed dynamically during the simulation as needed.[6]

---

[1] Ventimiglia, T., & Wayne, K. (n.d.). *Barnes-Hut algorithm overview*. Arbor.js. Retrieved November 25, 2024, from http://arborjs.org/docs/barnes-hut

[2] GeeksforGeeks. (n.d.). *Quad tree*. Retrieved from https://www.geeksforgeeks.org/quad-tree

[3] FreeCodeCamp. (n.d.). *Quadtrees explained*. YouTube. Retrieved from https://www.youtube.com/watch?v=-OLQlDHCMgM

[4] Norvig, P. (n.d.). *Using MPI with C*. Retrieved from https://www.paulnorvig.com/guides/using-mpi-with-c.html

[5] Curc. (n.d.). *Programming with MPI in C*. Retrieved from https://curc.readthedocs.io/en/latest/programming/MPI-C.html

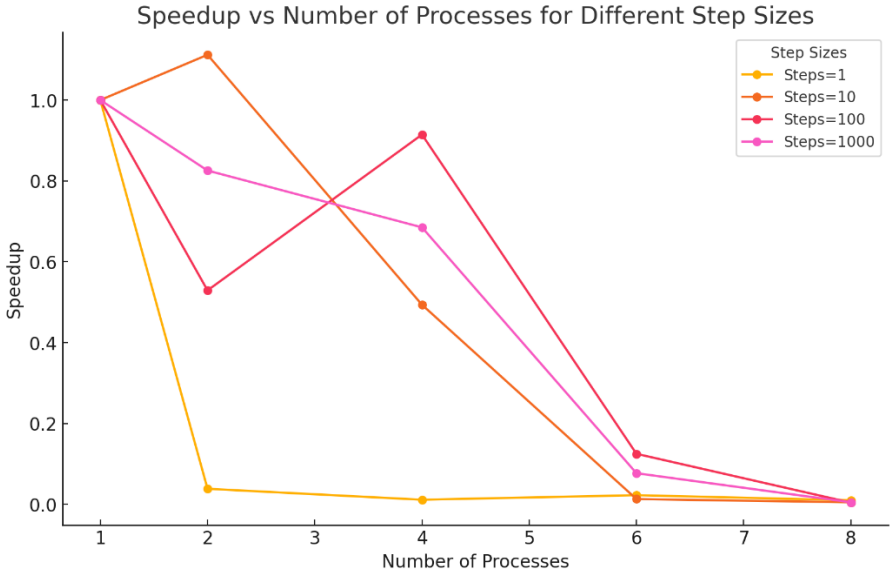
[6] This was introduced later. The optimization section has more information. Originally, this was much simpler and not necessarily a dynamic redistribution.

5. Repeat 2-4 for all of the steps.

The Force Calculation was the main portion that was parallelized. Here is where we reset the forces of all the particles owned by each process. This uses the quadtree structure to calculate the net force acting on each particle, and then the position of the particle is updated using the Leapfrog Verlet integration. Communication happens independently for each process, thus using parallelization here. Once the main implementation was complete, a tested a few runs and compared the results with other students to make sure that the final numbers were not too far off to ensure I was on the right track. For full transparency, I will provide the link to the google folder with the final outputs that other students provided. These were shared in Ed Discussion, but I also want to be fully transparent.[7]

Measurements

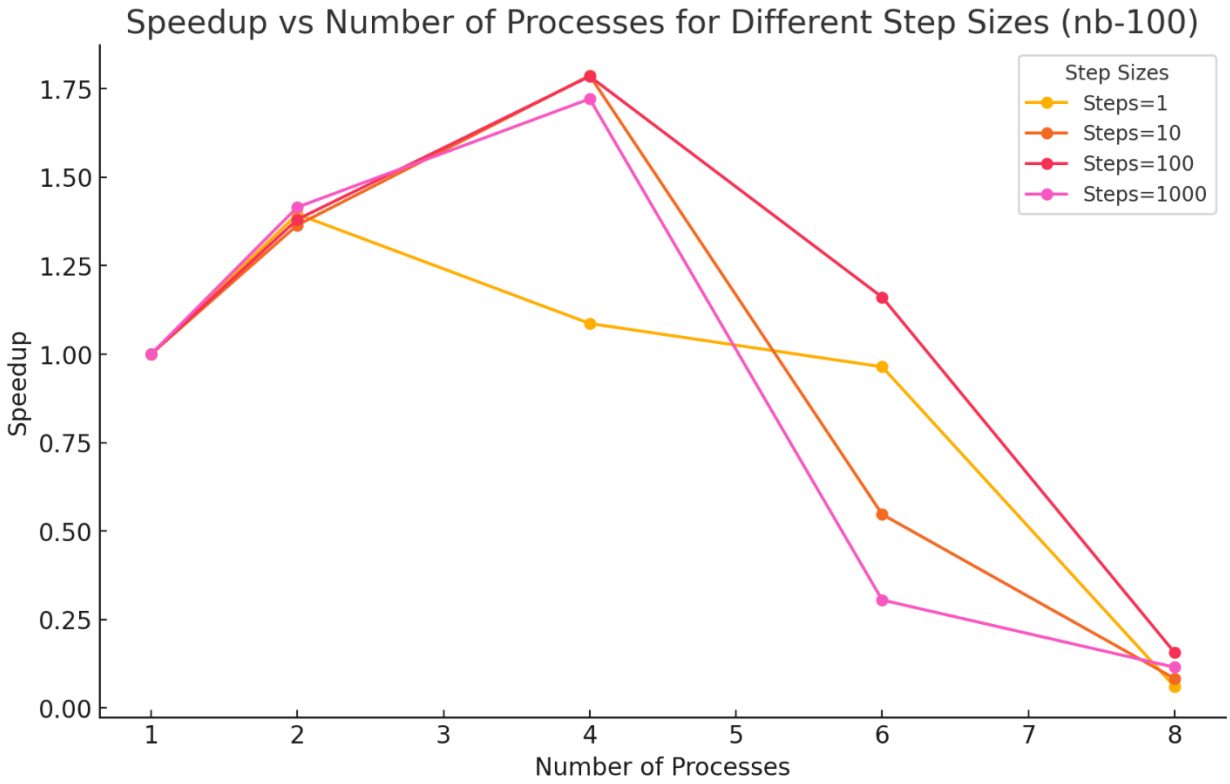
With the parallelization in place, it was time to test performance. Parallel speedups were calculated the same as the previous labs, with T_Seq divided by T_Parr. I will note that the 10 bodies file was not useful in this endeavor. This file was largely used for testing the implementation. With so few bodies, I noticed no speedup advantage running this on Codio regardless of the number of steps (up to 1000). The following graph demonstrates this. The only noticeable (and it was very small) speedup came when I ran the file with two processes and ten steps.



This intuitively makes sense given all the previous labs. When the number of data points are small, introducing parallelization does not necessarily provide any benefits. In fact, it might even hurt performance, as the overhead required to set up the processes (whether it be threads, go processes, or MPI processes) far outweigh their benefits. Could adding more steps provided a better performance? It is possible, but I will discuss why I did not graph the performance over 1000 steps shortly.

[7] https://drive.google.com/drive/u/0/folders/14pYj3v5mOGDE83lvrZhr-jKKnv_QuOST. Link to other students output files to compare values

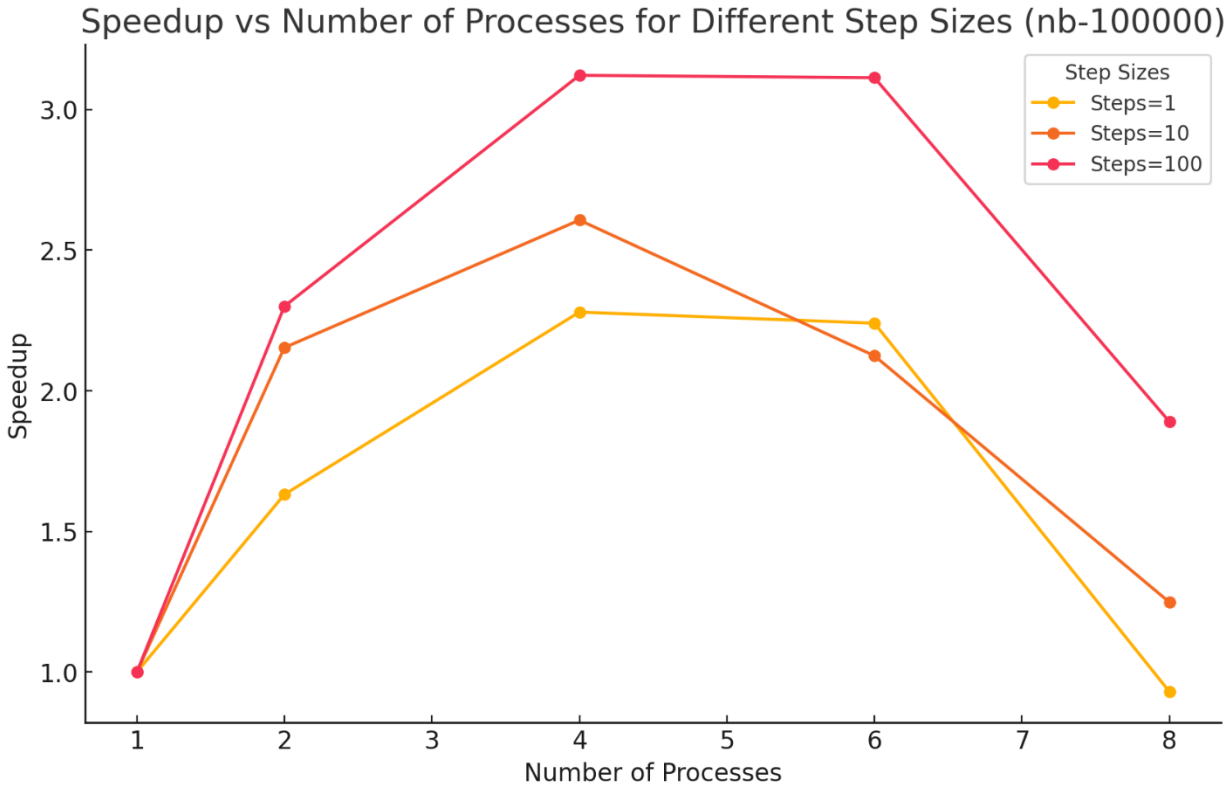
Let's examine the performance for 100 bodies. With more data points, now we begin to see the benefit of parallelization, but only to a point. The following image shows the outcomes of the tests ran.



Even with only running 1 step, there is a benefit for running between 2-4 processes. As more steps are introduced, the speedup reaches 1.75 times the sequential implementation. However, after 4 processes, the benefits degrade. This could be due to the limited capacities in using Codio. There is still a slight benefit with 6 processes at 100 steps, but this does not hold for 1000 steps.

Before moving on the 100,000 bodies, it is important to discuss why I limited to 1000 steps when running the 10 and 100 bodies. I noticed that, around 5000-7000 steps, there started to be issues with the output files. The x and y positions and velocities seemed off for every number of processes. These inconsistencies were immediately noticeable, and I tried to debug the issue. It seems a few things are happening, and it is possible that this is why the instructions for the lab mentioned to limit the code to at most 8 processors, 1000 steps and 5 average iterations. Indeed, running more seemed to start producing sluggish behavior and incorrect calculations that did not appear when running under 1000 steps, regardless of the number of data points in the file. As these performance issues mount, floating point calculations (that already depend on the computer) begin to compound and produce inconsistent results. Finally, when running the 100,000 bodies file, even running 1000 steps was practically impossible. It would run, but it was definitely taxing the system. One test ran at 1000 steps ran for about 90 minutes. So, this was not worth trying to explore even if we cut the time in half by increasing the processes. It is too time consuming. While it might be fun to explore on our own or with a more powerful machine, the limitations of Codio and the time constraints of the lab require me to make a choice.

With this out of the way, let's look at the 100,000 bodies file and see if there is a noticeable speedup.



We finally get to see the benefits of parallelization in full. With the large number of data points that need to be processed, we see speedup regardless of the steps and processes used. As the number of steps increase, we see the greatest benefit between 4-6 processes, as it is 3 times faster than the sequential implementation. This is a huge improvement. As in the previous 100 bodies test, we begin to see performance begin to dip again at 8, which might be due to the Codio limitations. But, this is also something we observed back in Lab 1 as well. More threads did not necessarily yield better results due to the overhead created by each running thread and the capacities of the machine. Similarly, we are seeing this again in this lab.

Optimizations Added

I considered a few options to optimize the program. Using CUDA was an option, but the one thing I was concerned about was the amount of time it would take to fully implement a CUDA option. Given the time limitations, I decided this was not an option for now. I might try to do this during my free time since I have a computer with a GPU. There was two things that I found might be helpful to help optimize the program. The first was to separate some of the private data elements in the quadtree class into a cache-aligned struct. In theory, this should help to minimize pointer chasing whenever possible. When testing, this did not seem to provide any speedup. However, by minimizing pointer chasing, we can prevent memory issues and have more efficient memory access patterns. This improvement was a much simpler approach. It required creating the cache aligned struct in the header file for quadtree, and then changing the data variables in the functions of the quadtree implementation in the quadtree.cpp file.

The second optimization was introducing dynamic load balancing. In theory, this would help distribute the work based on active particles to improve process utilization. The work distribution is based on particle density. This also did not much in terms of a noticeable speedup, but it helps to handle a few things. It better handles the particle loss scenario. Resources are more evenly utilized, and some idle time is reduced. This also helps to adapt the changing particle distributions. The implementation of this in the loadbalancing.cpp and loadbalancing.h file, two files that were not in the original implementation. In the earliest implementation, the program would not adapt to the changing particle distribution. Instead, it would just divide the bodies based on diving the size of the bodies by the number of processes. A more dynamic approach should theoretically provide a benefit at larger bodies sizes with large number of iterations. But, due to the limits of the Codio environment, it is difficult to notice too much of an improvement that is interesting to not by graphing the results.

While neither optimization produced the kind of noticeable benefits that a CUDA implementation would, by improving how we use memory when dealing with the quadtrees and better load balancing of the work needed, we introduce concepts that could prove more beneficial in larger simulations, in systems with higher memory latency, and when there are longer running calculations. Given how big the n-Body problem can be when simulating orbiting stars and planetary systems, these changes would be very beneficial for those larger simulations even if they yield minimal benefits with the limited steps and computing power within Codio.

Conclusion

Parallelization provided the biggest benefit with the highest number of data points. This was largely something that had already been observed from prior labs. It would have been interesting to try to run this program through a CUDA, particularly if using more than 1000 steps and 100,000 bodies. This program was entirely done within Codio. The limitations of this have been discussed throughout the rest of this report. In total, this project took about 40 hours.

The folder contains a Makefile that can be run with the “make all” command to compile. All tests were run with a theta value of .5, DT timestep of .005. I ran a few tests with different values, but decided that I would keep everything consistent and just use the .5 value for theta and .005 for DT timestep.