# Lab 5 – MPI Barnes-Hut

Jerry Reinoehl

CS380P Fall 2023

## 1 Approach

To implement the Barnes-Hut algorithm I first created a spatial partition tree (quadtree) and particle data structure. When a particle is inserted into the tree, the tree recursively divides regions into four subregions such that there is never more than one particle in the same subregion. After all particles have been inserted into the tree, the center of mass of each subregion is computed from the bottom node up. Next, for each particle, the force applied to the particle can be calculated by traversing the tree. If the particle is far enough away from a region, determined by the length of the region $l$, the distance to the region $d$, and the threshold $\theta$, then the force on the particle from that region can be calculated with the region's center of mass rather than each individual particle within the region, thus saving many computations and time. Finally, the calculated force is applied and the particle's position and velocity is updated.

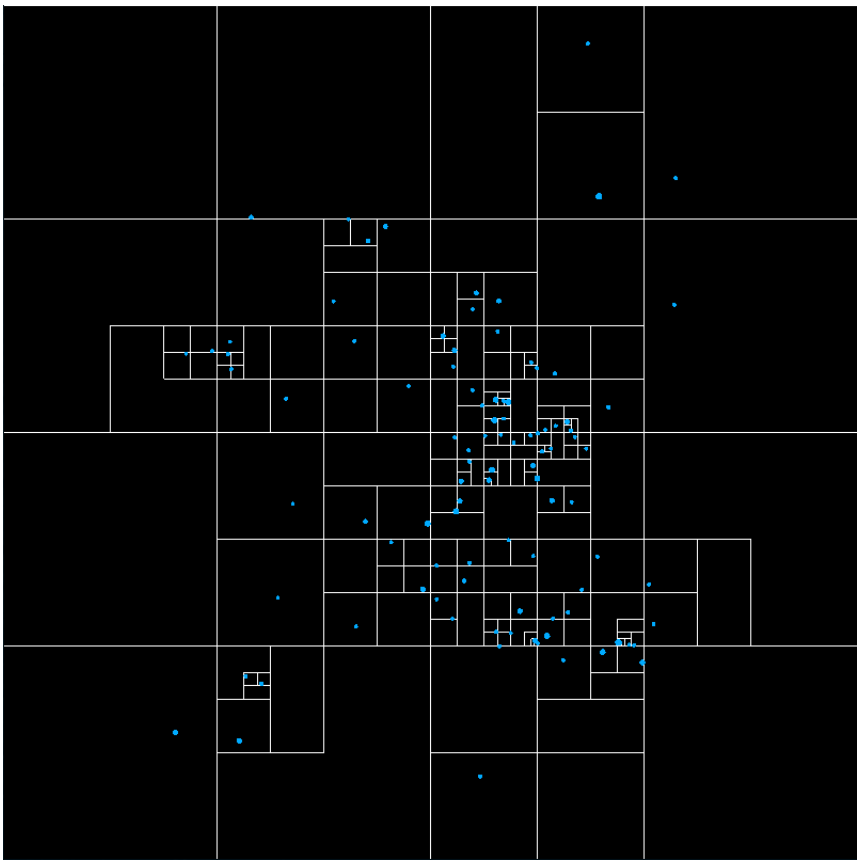
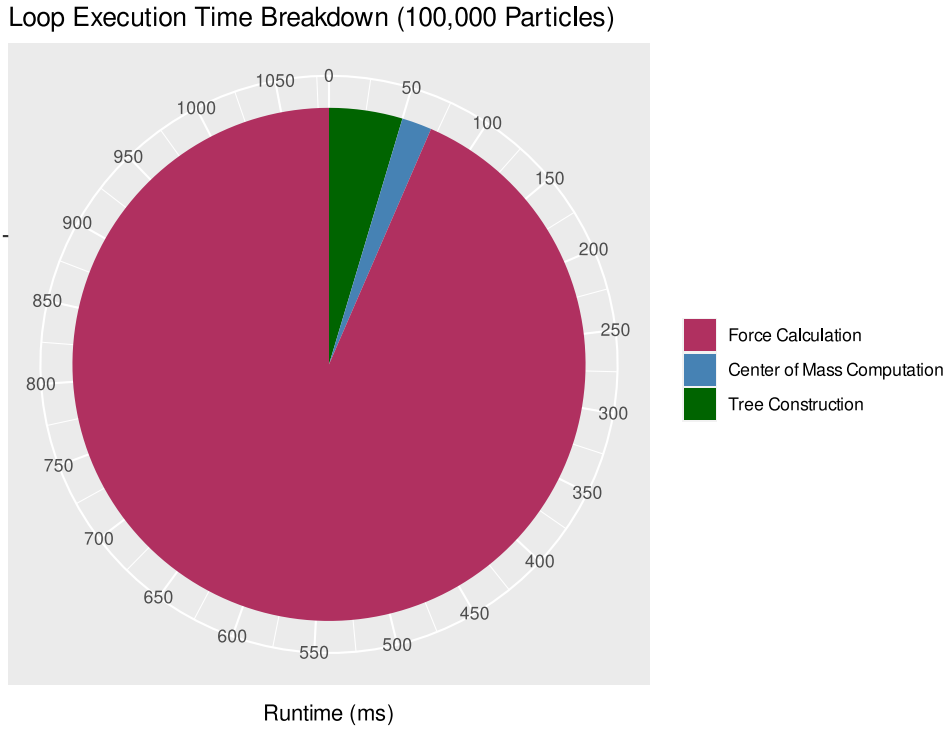


Figure 1: Visualization of Quadtree Space Partitioning

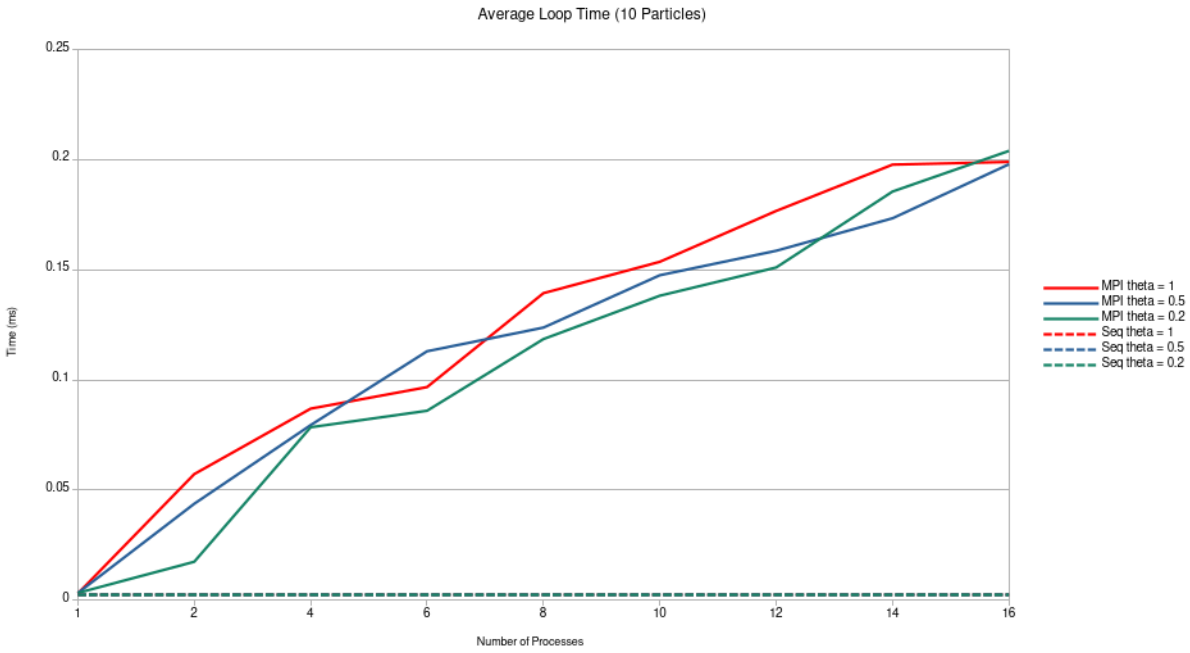
The graph below shows the amount of time spend in each phase of the sequential computation for 100,000 particles with a threshold of 0.5: tree construction, center of mass computation, and force calculation. The majority of the computation time (93.49%) comes from computing the force applied to each particle by traversing the quadtree. Therefore, this portion is the best candidate to parallelize with MPI to achieve a speedup.

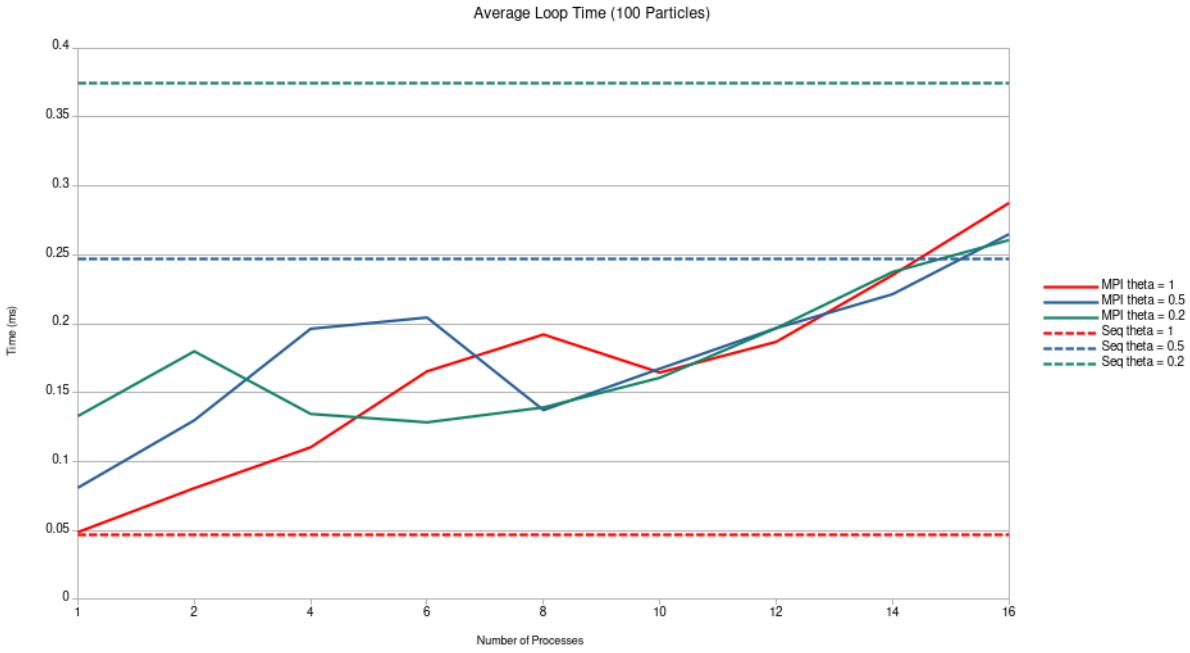



The first process (rank 0) reads in the particles from an input file. It then will broadcast the num of particles read to all other processes with an `MPI_Bcast`. All other processes will then allocate a buffer with enough space to hold the given number of particles. The first process will then broadcast the particles to the other processes. Each process then constructs a quadtree with all of the particles and computes the center of mass of each subregion. However, for the force computation and particle updates, the particles are evenly divided among the processes. Each process will concurrently traverse their tree to compute and update their set of particles. After each process computes and applies the force to each of its particles it will broadcast its set of particles with an `MPI_Ibcast` to all other processes and repeat.

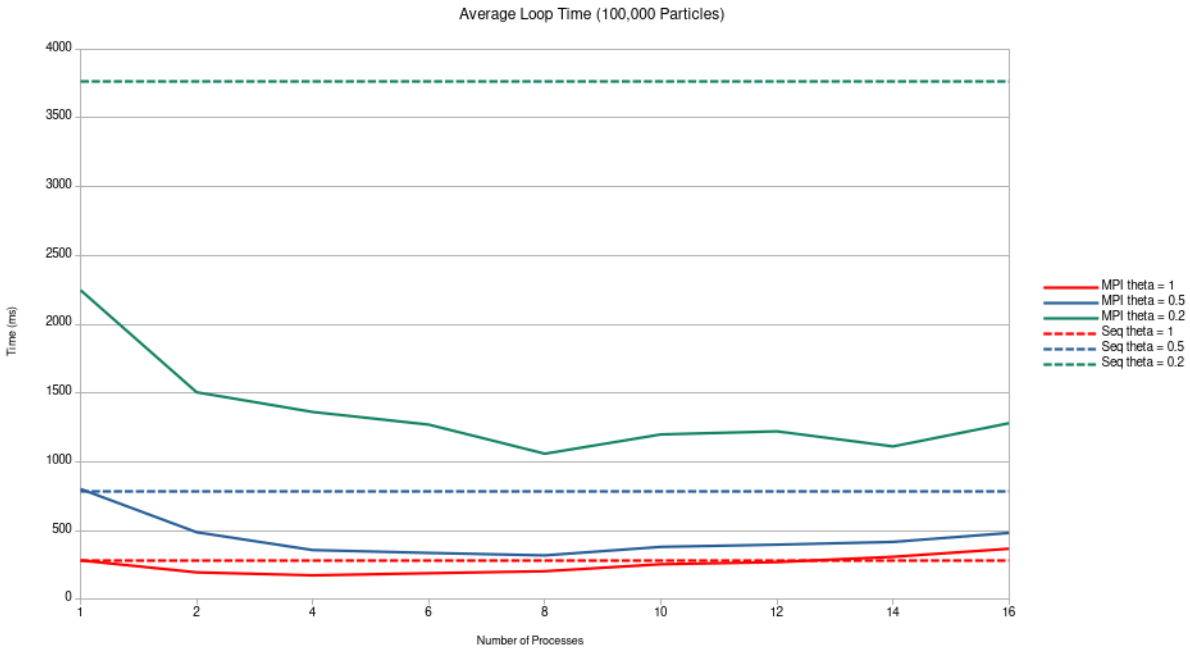
## 2 Performance

In the graphs below we compare the average loop time as we change the number of processes, the number of particles, and the threshold. We can see that when there are few force computations, such as for 100 particles with a threshold of 1 or with just 10 particles, the sequential implementation beats the MPI implementation. We also notice that as we increase the number of processes the execution time grows. This is because the overhead of communication outweights the time spend doing actual work. However, as we increase the amount of work done per loop, such as by decreasing the threshold or increasing the number of particles, we start to see some speedup.



We start to notice some speedup as we increase the number of particles to 100 and drop the threshold to 0.2. We notice much better scaling as we increase the number of particles to 100,000. The best speedup obtained was 3.55 for 100,000 particles with a threshold of 0.2 with 8 processes. This indicates that the parallel MPI solution should really only be used when simulating a large number of particles over many steps. For the MPI solution to be effective the amount of work per loop should take much more time than the broadcasting of particles between processes. The speedup of the MPI implementation improves up to about 8 processes, the number of cores on the test machine.




# 3 OS and Hardware Details

| OS Details | |
|---|---|
| OS | Arch Linux |
| Kernel | 6.6.2-arch1-1 |
| Architecture | x86_64 |
| Memory | 16 GiB |
| MPICH version | 4.1.2 |
| GCC version | 13.2.1 |

| CPU Hardware Details | |
|---|---|
| Name | AMD Ryzen 7 PRO 4750U Processor |
| Cores | 8 |
| Threads | 16 |
| Clock rate | 1.7 GHz |
| Cache L1 | 64 KiB (per core) |
| Cache L2 | 512 MiB (per core) |
| Cache L3 | 8 MiB (shared) |

# 4 Time Spend on Lab

| **Time Breakdown** | |
|---|---|
| Sequential Implementation | 10 hrs |
| OpenGL Visiualization | 3 hrs |
| MPI Parallel Implementation | 9 hrs |
| Report | 8 hrs |
| **Total** | **30 hrs** |