Nathan Love, Kelly Woicik
njlove, kwoicik
May 5th, 2022

Parallel N-bodies Simulator

I. Summary

We implemented an n-bodies algorithm utilizing Barnes-Hut trees and cost-zone thread partitions using OpenMP, and compared the performance to a sequential implementation on the GHC cluster machines.

II. Background

We chose to parallelize an n-bodies algorithm with the following structure:

> For every iteration, create a parallel for loop indexing over number of threads. Within each thread:
>
> 1. Decide on a partition of the tree to claim.
> 2. Calculate the forces on each particle in its partition.
> 3. Using the force on this particle and the "time" the iteration takes, calculate a new position.
> 4. Add the updated particle position to a new tree for next iteration.
>
> So for every iteration, we construct a new tree out of the updated particle positions, and we are only READING the old positions.

Our key data structure here is our barnes-hut tree as this is where all of our body and cost information is being stored. Key operations on the tree involve being able to traverse it in order to calculate forces on all of our particles as well as being able to reconstruct and insert bodies into our updated tree at the end of every iteration.

Our simulator takes in a text file in the following format:

> {grid dimension x} {grid dimension y}
>
> {number of bodies}
>
> {body 1 x} {body 1 y} {body 1 velocity x} {body 1 velocity y} {body 1 weight}
>
> {body 2 x} {body 2 y} {body 2 velocity x} {body 2 velocity y} {body 2 weight}
>
> …
>
> {body N x} {body N y} {body N velocity x} {body N velocity y} {body N weight}

After computation, it outputs a text file in the same format with updated parameters after convergence has been reached as well as the updated parameters at the end of each loop. The general format is shown below:

{grid dimension x} {grid dimension y}

{number of bodies}

{body 1 x0} {body 1 y0}

{body 2 x0} {body 2 y0}

…

{body N x0} {body N y0}

{body 1 x1} {body 1 y1}

…

{body N xM} {body N yM}

Where the number at the end denotes the iteration number, up to iteration M when the system has converged to a solution.

The computationally expensive part of our algorithm lies in calculating the force on one particle based on all other particles that affect it. This portion of the computation is generally O(Nlog(N)) when using a Barnes-Hut-esque algorithm, depending on the cutoff ratio used to determine whether a subtree should be treated as one body or not. However, since the new positions of the particles depend only on the previous iteration, this computation should be easily parallelizable leading to a runtime of O(Nlog(N) / P) where P is the number of threads working on it in parallel. However, distributing the work evenly among the threads like that may not always occur if there are greater numbers of particles in different areas of the simulation. To fix that, we can make use of ”cost zones” to split the total amount of work approximately evenly across multiple threads.

Additionally, it is in our best interest to prune our tree as high up as we can without exceeding our per-thread workload threshold; this allows for better locality as barnes-hut trees explicitly place nodes based on proximity (to a certain extent) and our cluster of nodes will mainly be affected by forces from other nodes in the same cluster, leading to high temporal locality. As we only need to read from our current tree during each iteration, all threads are able to compute the new positions fully in parallel. Our algorithm, as it currently stands, is not amenable to SIMD execution as SIMD tends to shine when we have very vectorized code that is for the most part, static; however, our algorithm relies on constantly changing nodes that differ in the forces currently acting on them with each given iteration.

One area where dependencies *do* exist in the computation, however, is at the re-insertion of the new particles into the new tree for the next iteration. Races can exist if we attempt to insert two particles to the same spot in the tree at the same time, so some synchronization is required at this step to ensure that no races occur.

III. Approach

As stated earlier, we made use of OpenMP and tested our algorithm on the GHC cluster machines. To see how our threads are mapped more precisely, we'll revisit our algorithm outline above with a bit more information on cost zones:

> For every iteration, create a parallel for loop indexing over number of threads. Within each thread:
>
> 1. Decide on a partition of the tree to claim that is less than or equal to TC/num_threads[1]. Assign the leftover partitions of the tree to thread 0.
> 2. Calculate the forces on each particle in its partition.
> 3. Using the force on this particle and the time the iteration takes, calculate a new position. Track computational cost[2] while doing this calculation.
> 4. Add the updated particle position and cost to a new tree for the next iteration.

Due to the nature of OpenMP and the ease of pragma statements, we didn't have to modify much of our original serial algorithm in order to map to our parallel machines. However, the final parallel algorithm updates particles as it encounters them in the old tree while the sequential one updates them in order of their indices. Due to this, we made a quick optimization to our sequential version where we added an array of *copies* of the bodies so it could look up a body by index instantly instead of parsing through the tree each time.

After going parallel, we tested both a static division of the work based on equal numbers of particles and dynamic divisions based on cost zones. We determined the computational cost for one body by summing up the total number of nodes we needed to visit in order to calculate its new position. This tally propagates up all the way to the root of our tree, where we store our total cost for one iteration of our algorithm. Thus, when we go to partition our tree into threads at the start of an iteration, we perform a DFS until we find a subsection of our tree less than or equal to TC/num_threads and add it to our partition, then repeat this until the partition's cost is as close to TC/num_threads as possible. We assign all leftover subsections to thread 0 for simplicity.

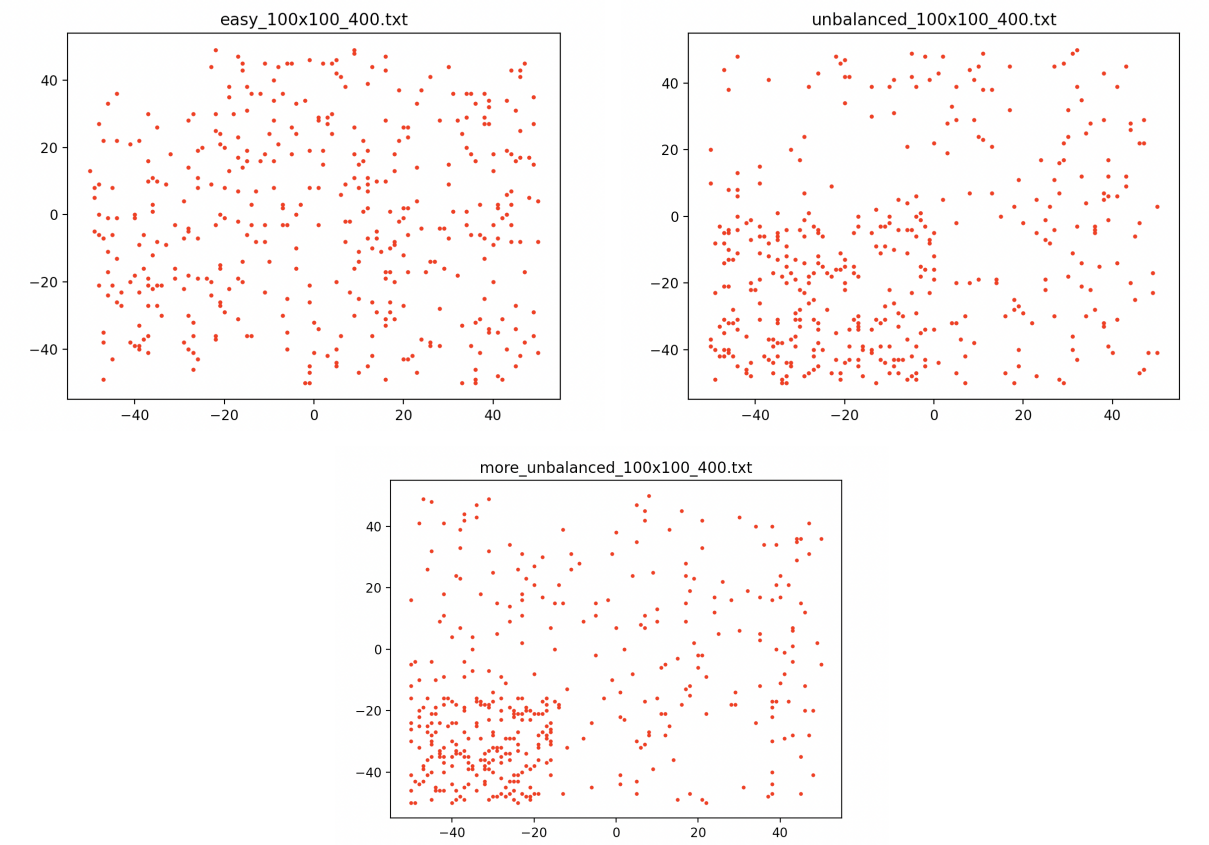
---

[1] Here, TC denotes the total cost of the entire tree and num_threads denotes the number of threads we're operating on.

[2] We define the cost here as how many other nodes we need to visit in order to calculate the new position of our current body. This idea was gathered from the reference paper mentioned below.

IV. Results

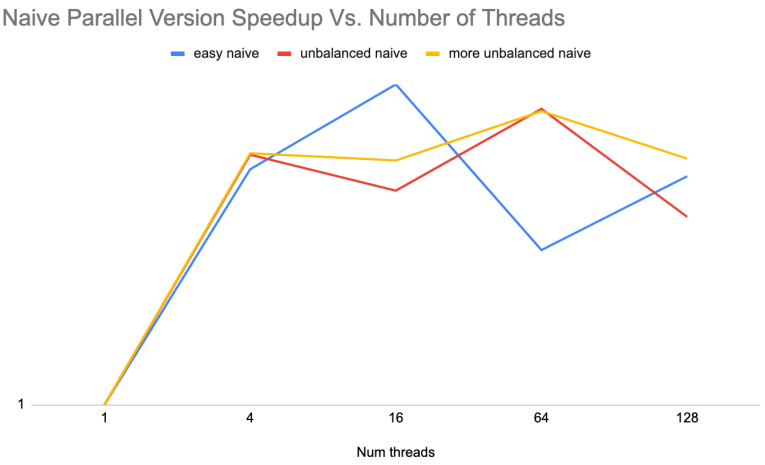
For reference, the following graphs represent the input datasets used in our program:



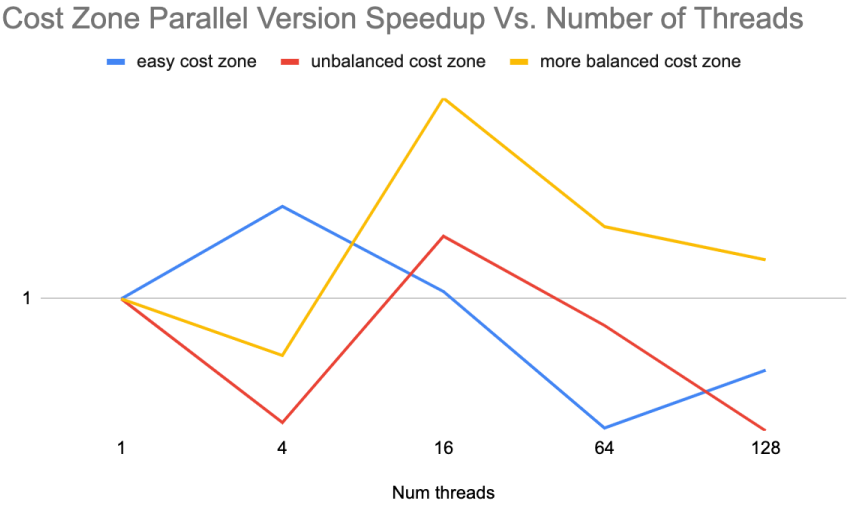
As we focused our algorithm on cost zones for skewed trees, a skewed input distribution was most important for our results.

We had two main metrics for the performance of our program: total speedup and workload distribution. To gather total runtime information we ran the program with a particular set of parameters 11 times, only timing the last 10 and then taking the average of those 10 runs. We compared the total runtime of the simulation using different parameters to determine the total speedup.

Our other metric was the workload distribution. Since parsing the input file and writing the output file were not parallelized, to measure the workload distribution we tracked only the time actually spent in computation by each OpenMP thread and compared how long different threads spent on computation.



Our naive parallel implementation (shown above) displays fairly typical speedup behavior; the speedup increases almost proportionally for smaller numbers of threads, but eventually for larger numbers the overhead of dividing the workload and scheduling the different threads begins to outweigh the time saved by the parallel computation, so speedup falls off. Of course, there is also a significant amount of work done in initialization and output that was not parallelized, so it can be said that the speedup is also eventually limited by a lack of parallelism once the computation happens quickly enough.



The more sophisticated parallel implementation (shown above) had *worse* speedup than the naive one, even when the particle positions were more unbalanced (which should have led to greater workload imbalance between threads, and increased benefit from the dynamic balancing). Worse total speedup is not necessarily surprising, since the workload balancing adds additional overhead, but what *is* surprising is that the workload distribution between threads *also* appears to be worse, even though workload imbalance is exactly what this algorithm was meant to correct.

```
bash-4.2$ ./barnesHut -i unbalanced_100x100_400.txt -v 2 -t 16
Elapsed time for thread 0: 0.531393
Elapsed time for thread 1: 0.545151
Elapsed time for thread 2: 0.547357
Elapsed time for thread 3: 0.517476
Elapsed time for thread 4: 0.541906
Elapsed time for thread 5: 0.524903
Elapsed time for thread 6: 0.549319
Elapsed time for thread 7: 0.505262
Elapsed time for thread 8: 0.507895
Elapsed time for thread 9: 0.522771
Elapsed time for thread 10: 0.517836
Elapsed time for thread 11: 0.518389
Elapsed time for thread 12: 0.531738
Elapsed time for thread 13: 0.544889
Elapsed time for thread 14: 0.521329
Elapsed time for thread 15: 0.523057
```

^ Time spent on computation in different threads running in the static workload version on a moderately unbalanced initial distribution of particles.

```
bash-4.2$ ./barnesHut -i unbalanced_100x100_400.txt -v 3 -t 16
Elapsed time for thread 0: 2.243255
Elapsed time for thread 1: 1.138679
Elapsed time for thread 2: 0.500910
Elapsed time for thread 3: 0.422705
Elapsed time for thread 4: 0.337943
Elapsed time for thread 5: 0.328521
Elapsed time for thread 6: 0.325953
Elapsed time for thread 7: 0.303403
Elapsed time for thread 8: 0.210324
Elapsed time for thread 9: 0.231697
Elapsed time for thread 10: 0.225324
Elapsed time for thread 11: 0.202317
Elapsed time for thread 12: 0.239482
Elapsed time for thread 13: 0.232739
Elapsed time for thread 14: 0.217538
Elapsed time for thread 15: 0.249076
```

^ Time spent on computation in different threads running in the dynamic workload version on a moderately unbalanced initial distribution of particles.

```
bash-4.2$ ./barnesHut -i more_unbalanced_100x100_400.txt -v 2 -t 16
Elapsed time for thread 0: 0.572831
Elapsed time for thread 1: 0.524413
Elapsed time for thread 2: 0.536499
Elapsed time for thread 3: 0.536018
Elapsed time for thread 4: 0.562530
Elapsed time for thread 5: 0.512973
Elapsed time for thread 6: 0.531101
Elapsed time for thread 7: 0.561973
Elapsed time for thread 8: 0.520239
Elapsed time for thread 9: 0.557945
Elapsed time for thread 10: 0.528419
Elapsed time for thread 11: 0.521553
Elapsed time for thread 12: 0.541828
Elapsed time for thread 13: 0.542039
Elapsed time for thread 14: 0.501542
Elapsed time for thread 15: 0.512767
```

^ Time spent on computation in different threads running in the static workload version on a severely unbalanced initial distribution of particles.

```
bash-4.2$ ./barnesHut -i more_unbalanced_100x100_400.txt -v 3 -t 16
Elapsed time for thread 0: 2.178373
Elapsed time for thread 1: 1.086069
Elapsed time for thread 2: 0.633380
Elapsed time for thread 3: 0.440872
Elapsed time for thread 4: 0.370976
Elapsed time for thread 5: 0.334276
Elapsed time for thread 6: 0.323980
Elapsed time for thread 7: 0.257914
Elapsed time for thread 8: 0.231987
Elapsed time for thread 9: 0.145138
Elapsed time for thread 10: 0.337600
Elapsed time for thread 11: 0.285180
Elapsed time for thread 12: 0.331122
Elapsed time for thread 13: 0.293641
Elapsed time for thread 14: 0.285704
Elapsed time for thread 15: 0.257132
```

^ Time spent on computation in different threads running in the dynamic workload version on a severely unbalanced initial distribution of particles.

A few additional lines of code to separate our measure of workload imbalance into time spent claiming a partition and time spent computing shows that the imbalance exists almost entirely in the actual computation time. Fairly consistently, thread 0 spends much more time on its computation than all of the others (this is seen in the 4 figures above). This seems to suggest that

the partitioning algorithm itself is not very good. Perhaps the simple number of nodes traversed during calculation is not a good estimate of the total work required to update a particle, and it would be more appropriate to count the number of nodes actually interacted with. Or maybe threads that claim a partition later tend to claim a smaller partition due to the costs of remaining subtrees being less likely to sum up cleanly to the maximum allowed cost per thread (total cost / # of threads). In any case, we found this algorithm for partitioning the workload among threads to be unsuited for this problem.

V. References

[1] Ananth Y. Grama, Vipin Kumar, and Ahmed Sameh, “Scalable Parallel Formulations of the Barnes-Hut Method for n-Body Simulations” 1994

VI. List of Work and Distribution of Total Credit

We feel that the total credit should be distributed 50-50. Kelly worked primarily on the visualizing, input and output files, report, and data generation while Nathan did all the behind the scenes physics calculation methodology and implemented the algorithm in code.