CS 267 Parallel Computing Homework 2

Parts I, II and III(Serial, OpenMP, MPI and GPU)

Chiyu "Max" Jiang, Eleanor Cawthon, Madeleine Traverse

May 17, 2017

1 Introduction

The primary objective of this project is to implement an $\mathcal{O}(n)$ complexity algorithm to simulate particle dynamics solely dependent on short-range forces, and extend the serial implementation to a shared-memory and a distributed memory implementation. We begin in Section 2 by describing our changes to the provided implementations, and analyzing their theoretical and experimental complexity. We then examine analyze the performance of our code using VTune in Section 3, and discuss potential further improvements. We conclude with an evaluation of the scalability of our two parallel approaches in Section 4.

2 Implementation Notes and Complexity

2.1 Getting Serial to $\mathcal{O}(n)$

Before parllelizing our code, we first needed to improve the reference serial implementation from which we would build our parallel versions and to which we would be comparing them. We implemented two different improved serial algorithms.

2.1.1 Barnes-Hut Tree

The first algorithm we implemented utilizes a data-structure which is called a Barnes-Hut Tree [1]. Barnes-Hut Tree algorithm is essentially an order $\mathcal{O}(n \log n)$ algorithm. However since this algorithm is more generally extensible to particle simulations that take into consideration the long-range force interaction and provides an efficient framework for simplifying it, it is still of interest to implement and compare this data structure with the binning method. The Barnes-Hut tree data structure is based on the quad tree data structure. Initializing the data structure involves breaking the simulation domain into finer and finer quadrants until each quadrant contains at most one particle. Organized in this way, clusters of particles can be represented by the center of mass of the cluster. Under certain threshold conditions, this can be a good enough approximation. Since the depth of a balanced quad tree is $\mathcal{O}(\log n)$, the complexity of this algorithm is $\mathcal{O}(n \log n)$. This is not ideal in terms of efficiency compared to the binning method, but it is significantly more efficient that the naive implementation which is order $\mathcal{O}(n^2)$.

2.1.2 Binning Method

We used a binning method to implement the Serial code. By dividing the space of the simulation into bins, we were able to do far fewer computations as compared with the original code. For each particle, the `apply_forces()` function was only applied to the particles in that particle's bin and in the bins that neighbored it, so for a given particle, the runtime would be $\mathcal{O}(9bn)$ instead of $\mathcal{O}(n^2)$.

We accomplished the binning by creating a matrix of C++ vectors, with each position in the matrix representing a single bin. We recreated this matrix at the start of each time step to adjust for the movement

of the particles (which may have switched bins). An improvement for the serial code would be to edit the existing matrix instead of creating a new one, but since creating the matrix is only an $\mathcal{O}(n)$ operation, we did not think the improvement would be appreciable.

To find the optimal number of bins for the simulation, we ran the test script multiple times with different numbers of bins, using roughly a binary search to pick the tested bin counts. We found 16 bins to be optimal for $n < 5000$, while 32 (for serial and OpenMP; 36 for MPI since our implementation requires the number of bins to be a square) was optimal for $5000 \leq n < 20,000$. We found 64 bins led to better performance than 81 or 128 for all $n > 20,000$ tested.

2.1.3 Complexity test

It is apparent from the small error margins with respect to the log-log linear best fit lines in Figure 1 that both functions run in polynomial time, that is, $\mathcal{O}(n^k)$ for some constant k. Interestingly, although Barnes-Hut's theoretical runtime is $\mathcal{O}(n \log n)$ rather than $\mathcal{O}(n)$, empirically for the values we tested, we found that Barnes-Hut conformed quite closely to the idealized $\mathcal{O}(n)$ performance.

The overall slope of the line of best fit is an average of piecewise slopes. Our binning code achieves much lower slopes at lower values of n, with slopes ranging from 0.407 to 1.372 for an average slope of 1.040 ± 0.593. Although there is a range, the overall performance of our serial code is thus very close to $\mathcal{O}(n)$.

In contrast, Barnes-Hut performs worse in absolute terms for most tested values of n. However, it scales more consistently, with a slope of 1.270, with a range of slopes varying just $\pm$ 0.07. This suggests that the low density of our simulation allows the n term to dominate the $\log(n)$ term in the Barnes-Hut theoretical runtime — in other words, our tree is shallow.

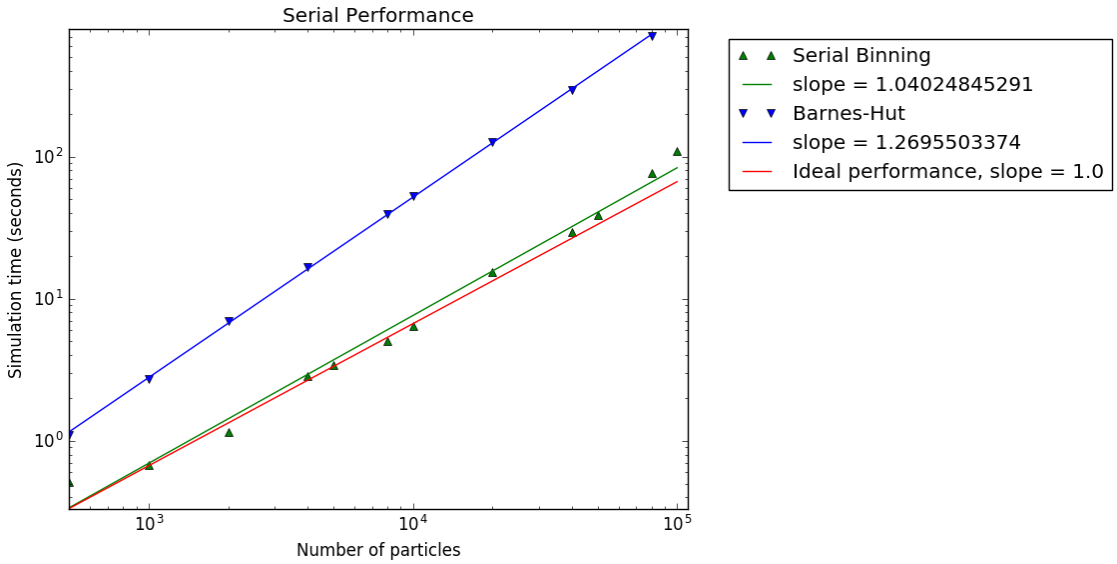

Figure 1: log-log plot of number of particles versus simulation time for performance analysis on the two serial methods. Note: The "Ideal performance" line is meant to illustrate the ideal slope. The y-intercept was chosen for visual clarity and is not analytically significant.

2.2 Shared Memory Implementation with OpenMP

2.2.1 Binning Method

To create the OpenMP code, the serial code with binning was reorganized to allow for OpenMP Pragmas. The comparing of the particles in the bin matrix was parallelized by collapsing the matrix for loops into

one long process that could be divided between threads. The scheduler was set to dynamic, however no statistically significant difference was observed between static and dynamic allocation. We also attempted to parallelize the creation of the bin matrix at the beginning of the timestep, but constant segfaults suggested that the code was not compatible with the pragmas used as the threads were attempting to create infinite bins.

2.2.2 Barnes-Hut Tree

In our first serial implementation of the Barnes-Hut Tree approach, we re-created the entire particle tree every time the simulation moved to a new step. Although this did not affect the Big-O runtime, this involved double the amount of work per step, which is significant even though it is a constant factor. It also made the code difficult to parallelize because the initial construction of the tree requires synchronization, introducing a serialized bottleneck. For these reasons, we decided it was important to enable our implementation to make updates to the already-constructed tree, allowing sub-trees to be updated in parallel. Unfortunately, implementing this functionality proved substantially more difficult than expected, and we ultimately abandoned this variant due to time constraints. As such, we did not parallelize the Barnes-Hut variant.

2.3 Distributed Memory Implementation with MPI

We explored two basic approaches to converting our binning algorithm to use distributed memory. We conceptualized our algorithm as requiring two basic steps: dividing the particles into bins, and distributing particles to the processors. As such, we had a choice between having the master thread divide particles into bins and then sending a bin and its neighbors to each process, or to first distribute particles to processors and then have each process perform the binning on its subset of the particles. Our preliminary results led us to adopt the second approach, based on a combination of expected programmer efficiency and compute efficiency.

2.4 Implementation with GPU

We tried several different approaches towards GPU implementation. We wrote a basic CPU based binning code that performs binning using CPU and calculating forces using GPUs. We also attempted two other approaches including binning using GPU as well as a batching method to take advantage of more threads available in the GPU. However the overhead of the latter two methods have proven to be immense and its performance gain did not overweight the cost. The methods will be further discussed below.

2.4.1 CPU-based Binning

The particle simulation for the GPU has been optimized with the binning method as described above. The bins are calculated serially in the CPU. For each particle, the number of the particle's index in the particle array, 0 through $(n-1)$ is stored in the bin array to indicate their bin location. Then the bin array and the particle array are copied to GPU storage. Each GPU thread takes its own number and is assigned the particle with the corresponding index. It performs applyforces() for the particles in the nearby bins and then updates the particle array.

2.4.2 GPU-based Binning

This code utilizes the thrust library in tandem with CUDA library for parallel implementation. To begin with, GPU computes the bin number for all particles in parallel. The the `sort_by_key` function in thrust sorts the particle indices based on the bin number as key so as to collect the particles that below to the same bin. Then `count` is performed to calculate the offset for the start of each bin. Thrust uses GPU to perform the sorting and counting algorithm in parallel. Then GPU applies forces using the binning information. However results based on $n = 2000$ particles show the overhead of binning to be 5 times that of particle operations.

2.4.3 Batching Method

Batching method essentially does not reduce the complexity of the overall algorithm, therefore it does not reduce an $\mathcal{O}(n^2)$ algorithm to $\mathcal{O}(n)$. However it theoretically cuts the cost by a constant factor. Here's a description of the algorithm: launch `NUM_BATCHES` $\times n$ threads. Also initialize `NUM_BATCHES` copies of struct for the workers of different batches to record the added acceleration. Then as a last step, sum over the different batches to get the overall acceleration on every particle. Denote n as the number of particles, n_b as the number of batches, the number of operations (assuming all required threads can be simultaneously launched and enough memory can be allocated to record summed acceleration) to calculate the total acceleration on all particles is:

$$n_{op} = \frac{n}{n_b} + n_b$$

which can be minimized by taking $n_b = \sqrt{n}$ with rounding to the nearest integer. However in practice this method proves to be slow, possibly because of cache miss caused by extra allocation of memory to store batch sums of particle acceleration.

2.4.4 Comments on GPU and CUDA

Synchronization
Locks are not needed for the GPU code because there is no data at risk of being overwritten. There is implicit synchronization between the frames and time-step that keep all of the data at the same step in the program. Furthermore, when threads are checking the location of other threads in order to calculate applied-forces(), the locations are not overridden until the next frame during moveparcicles(), so there is no risk of reading a particle's location incorrectly based on whether or not its thread has updated it. Synchronization is embedded in the thread protocol to keep all threads at same frame until the time-step moves forward, such as the wait that happens between the appliedforces() call and the moveparticle() call. It is important to note that there is a performance cost in this scenario as some threads may finish appliedforces() early and must remain inactive until moveparticle() is called. This is a necessary trade off to ensure correctness.

Strength and weakness of CUDA and current GPU
GPU in general is extremely good at performing high throughput computation that does not require a lot of communication. Compared to OpenMP and MPI, it is especially difficult to communicate data between different threads. CUDA especially does not have functions calls that explicitly allow common communication operations such as reductions. Implementations have been made in auxiliary libraries such as thrust and ArrayFire, but in experience the overhead is still extremely costly, at least compared to CPU execution on the same task (binning).

3 Performance Analysis

In order to facilitate a rapid development cycle, we used VTune to test performance with small values of n in order to aid in developing code that would scale. Specifically, all VTune testing was done with small enough n to fit in 16 bins. This introduces the potential error of overstating the overhead of steps that do not scale with n.

3.1 Serial

The serial code used the binning method to allow for fewer particle comparisons. This was very successful. In the Naive code, `apply_force` took 1.60e12. In our code, it took 4.83e10 clock ticks. This is clearly a significant improvement.

According to the VTune report, the serial code was mostly Back-End bound, taking up around 31% of the performance time. This can often be due to data-cache misses or stalls. Of the Back-End use, approximately 28% was Core bound. This metric is sometimes due to dependencies in program data. Further analysis of

the code breakdown pinpointed the slow down as an issue with the dynamically allocated vectors used in the matrix map. The performance problem with creating the vectors is primarily an issue due to the way C++ creates vectors. A small amount of space is malloced and then, whenever the vector exceeds that space, another larger space is malloced and everything in the first space must be copied into the second. Since this was happening multiple times for each time step, the performance was negatively affected. The choice of the pointer particles to be stored as vectors was made to allow the matrix to be edited each time-step instead of requiring it to be rebuilt. The time to build the matrix, however, was $\mathcal{O}(n)$ time, and is in fact probably mostly negligible, so a far better storage for the particles would be an array.

The serial code performance could also be improved by using a quadtree structure for the bins instead of a simple matrix. Even fewer `apply_force` function would need to be called, which would reduce the overall runtime.

3.2 OpenMP

The openMP code used the binning method in serial, but distributed the bins to different threads. In the Naive code for an n of size 12,000, `apply_force` took 1.33e12 clock ticks. Our code for the same n took 3.24e8 clock ticks, a significant improvement.

According to the VTune report, the code was heavily Back-End bound, taking up around 70% of the time. Of the Back-End time, approximately 43% was Memory Bound. When this metric is high, it can be due to stalls waiting for memory loads and stores. This was again discovered to be a consequence of using the dynamically allocated vectors and could be fixed by using arrays. Other improvements relate to improving the serial version of the code, as mentioned with the quad tree bins.

3.3 MPI

The MPI code was designed to take advantage of the relatively low latency involved in communication between nodes. This allows for memory to be distributed, which can be important with very large data sets. Consequently, the largest performance cost is in this communication. According to the Craypat Profile, the `MPI_Alltoallv` call takes up between 19-21% of the sampling and performance time. The other three MPI calls combined take up approximately 10% of the performance time. A solution to decrease the time spent on the `MPI_Alltoallv` would be to require one processor to hold all of the data, instead of distributing it. This "master" would then only send the required bins to each processor and they would not need to communicated with each other. If the data is too large not to distribute it, a more complex model with multiple "masters" could be implemented where they communicate with each other to keep the data up-to-date.

The performance slowdown of the dynamic vectors from the original serial code is present in the MPI implementation as well and could be fixed by switching the data structure to arrays.

A strong benefit was seen again in the MPI code versus Naive due to the binning. Where the Naive code spent approximately 67% of the performance time on `apply_force`, our code spent only approximately 37%, a major improvement. This is due to the fact that binning greatly cuts down on the number of times the function needs to be called. It could be reduced even further by implementing a quad-tree structure for the bins so that less particles need to be compared.

4 Results At Scale

Perhaps due to inherited inefficiencies in our serial code, we encountered difficulty in scaling to the target $n = 50,000$ particles per processor. We include data from the only tests that terminated at this scale: our serial codes, our OpenMP code with 50,000 particles per processor at up to 18 processors, MPI with 50,000 particles on up to 12 processors, At larger values of n, our code did not terminate within the allotted time. Accordingly, we also tested with $n = 5,000$ for strong scaling and $n = 5,000 \times p$ for weak scaling to p processors, so that we would have more data to compare.

4.1 Serial vs. OpenMP vs. MPI: Weak Scaling

In Figure 2, we compare the execution time of each parallel run to the execution time for our serial binning code to run a given number of particles-per-processor on a single processor. Steeper (more negative) slopes correspond to more degraded performance with increased n, and therefore worse scalability. The y-axis shows a scalar between 0 and 1, where 1 would represent perfect scaling where p processors can simulate $n \times p$ particles in the same time it takes one processor to simulate n particles.

The line with slope = 0 is the theoretical ideal performance with perfect parallelism. However, calling this performance ideal is misleading, as it does not take into account Amdahl's Law. A more representative comparison is the comparison of the slopes of the two serial methods with the slopes of the parallel methods. The two serial methods have the same slope as exhibited in Figure 1, but negative.[1]

At similar scale ($4000 < n < 120000$), OpenMP exhibits better scalability (less negative slope) and better absolute performance (higher efficiency scores) than our MPI implementation. For our MPI implementation, with a slope of approximately -1, each time the number of processors is doubled, the extent of the speedup is halved. In the smaller range of n values, our OpenMP code degrades by only 38%. The best fit line for MPI fits the data more closely, suggesting more consistent overhead in that implementation. However, the absolute performance of MPI is substantially worse than that of OpenMP.

For values of n below 90,000, the slope of our OpenMP curve is much gentler, with performance significantly degraded in the jump from 60,000 particles on 12 processors to 90,000 on 18. This is the first value of n where we use 64 bins instead of 32. We also tested with 32 and 36 bins at these n-values, but the performance was significantly worse. It is possible that there is a better number of bins between 36 and 64, but the degraded performance with higher n continues to be apparent for the 50,000 per processor curve, and so this may simply be an example of limited scalability.

Our limited results for $n = 50,000$ show that at least for for OpenMP, the amount of overhead per particle per processor continues to increase as the problem size grows. Similar to our results for $90,000$, we see a significant dropoff at $n = 600,000$. We again experimented with larger numbers of bins, but again this did not improve performance.

We note that the performance of OpenMP with one thread is almost identical to the performance of our serial code, even at $n = 50,000$. This speaks to the usability of OpenMP as a library — it was relatively easy to write code that, without modification, reduces to the serial runtime without incurring overhead from the additional code required for OpenMP. In contrast, our MPI code on a single processor performs significantly worse than our serial code even at $n = 5,000$.

At $n = 50,000$ on one processor, the MPI performance appears to be better than the performance predicted by the best fit line for $50,000$ spread across 10 processors. This further indicates room for improvement in our scaling in our MPI code.

4.2 OpenMP vs. MPI: Strong Scaling

In Figure 3, we vary the number of processors while keeping n constant. In this graph, the slope represents the amount of speedup per processor, with ideal speedup represented by a slope of -1.0 corresponding to runtime of time T/p, where T is the runtime on a single processor.

In our OpenMP code with $n = 5000$, the overhead of adding processors obviates the performance gains at $p > 12$. The data is very close to linear (in log-log scale) for $p \leq 12$. For larger problems, our OpenMP code continues to scale well for all p through 12, after which there is a similar dropoff. We note that this trend is also present in our weak scaling results, especially for 5,000 particles per processor. Since Edison has 12 cores per CPU, this is the point at which work begins being shared across both CPUs-per-node, and our results indicate that this incurs significant overhead particularly in the shared memory design of OpenMP. We were unable to test the MPI code with more than 12 nodes because of compute constraints, but we hypothesize that MPI would not exhibit this particular dropoff.

[1]The numbers are slightly different because only results with $n \geq 4000$ are included in Figure 2, whereas Figure 1 includes results with smaller n, on which our binning algorithm performs better.

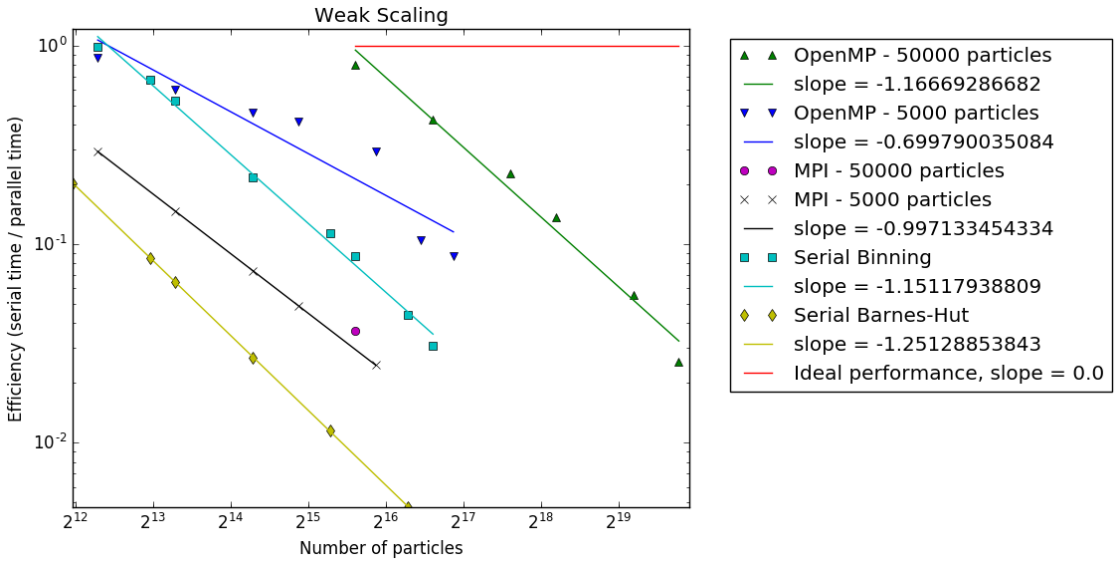

Figure 2: Weak scaling results, OpenMP vs. MPI vs. Binning vs. Barnes-Hut. OpenMP particle numbers are per processor. MPI was tested with 5,000 particles per processor. The two serial methods were tested on a single processor with varying numbers of particles.

As with weak scaling, the absolute performance of our OpenMP code is significantly better than that of our MPI code for equivalent problems. We attribute this to the increased complexity of debugging and profiling distributed memory as compared with shared memory. We note however that the results for MPI with 50,000 particles come the closest to the idealized slope of -1.0, with a slope of -0.96 and low error, suggesting the slowness of our MPI code only increases linearly with scale instead of super-linearly.

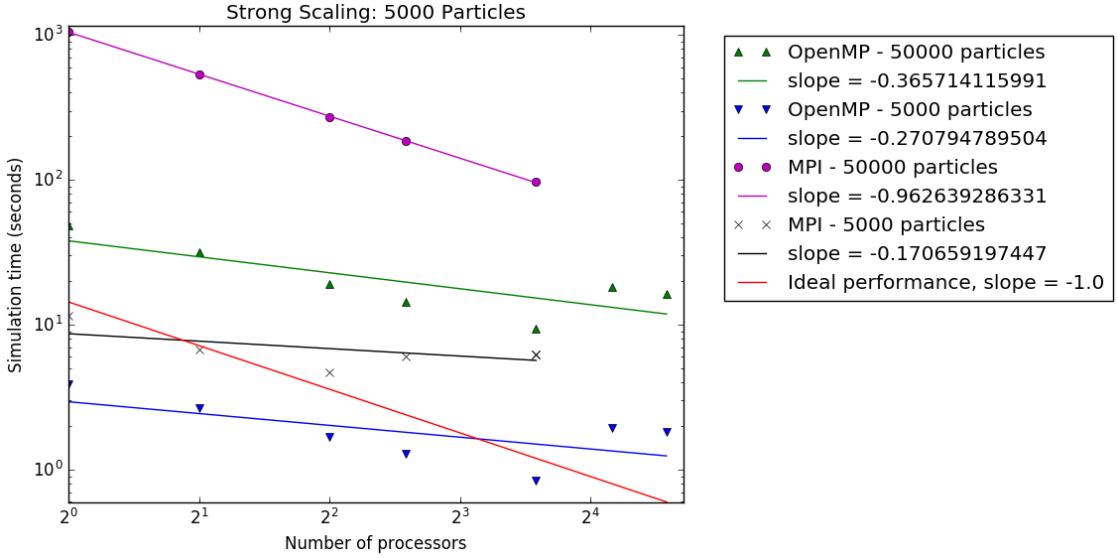

Figure 3: Strong scaling results, OpenMP vs. MPI. The y-intercept of the ideal performance line is the average of three runs of our binning code with 5000 particles.

5 Conclusion

By using a binning method, we achieved serial performance very close to $\mathcal{O}(n)$. When we parallelized this algorithm, we achieved better scalability overall with our shared memory implementation using OpenMP, but we encountered degraded performance when using more than 12 processes (the number of cores per CPU on Edison). Our distributed memory code using MPI exhibited worse overall performance but more consistent scalability for very large numbers of particles.

References

[1] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. *nature*, 324(6096):446–449, 1986.