Particle Simulation with OpenMP

Lawton Manning, Sarah Parsons - CSC 726

November 7, 2019

1 Introduction

In this assignment we worked to improve a sequential algorithm for the simulation of particle forces, and to parallelize the algorithm using OpenMP and MPI.

2 Sequential Algorithm Improvements

The configuration of the original sequential algorithm was to compute the forces of one particle on all particles per each time step, thus requiring $O(n^2)$ time. We have worked to linearize this solution, to reduce the complexity to $O(n)$. In order to achieve this goal, we first approached the problem by considering the grid space and worked to divide the space into a finite number of bins, each with a size of $(totalgriddimension/cutoff)$. Upon splitting the grid into bins, our algorithm traverses each bin in row-major order and computes the forces on all particles in the bin from its neighbor bins. By reducing the size of the computation space to one in which only the neighbor points within the cutoff range are accounted for, the time complexity reduces to $O(n)$ (as opposed to computing the potential force of all points on a single point which costs $O(n^2)$). Consider the following:

c = cutoff
A = area of computation space = $9c^2$, where 9 is the number of bins in the space
d = density of computation space = $|P|/A$ for P points

Therefore, $A = |P|/d = 9c^2$ and so $|P| = 9c^2d$, which represents the number of comparisons per point. In order to linearize the time complexity, our goal is to achieve $9c^2dn$ total comparisons for n points. Thus, so long as $9c^2d < n$, this goal will be met. Given that the density of the grid is evenly distributed on average, we would not expect the density of the computation space to ever reach size n.

Figure 1: Mock grid with bin indexing

	0.4	0.8	1.2	1.6	2
0.4	0	1	2	3	4
0.8	5	6	7	8	9
1.2	10	11	12	13	14
1.6	15	16	17	18	19
2	20	21	22	23	24

Figure 1 represents a mock-up of the particle space. Highlighted in yellow is an example of a computation space of $9c^2$ where c is the cutoff (e.g. 0.4). For a point at (1,1), all of its potential neighbors exist in the

highlighted area. In order to account for all areas in the grid, the total number of bins = $(1 + n/c)^2$, where n is size of grid, c is cutoff, and $1 + n/c$ is number of columns. We included one extra column to account for edges in areas where the cutoff does not perfectly divide the grid space. For example, the index of the bin at point (1,1) would be $(x/\text{cutoff}) * \#\text{columns} + (y/\text{cutoff}) = int(1/0.4) * 5 + int(1/0.4) = 12$, as shown in Figure 1. By indexing each point with linear bin notation, we can define the computation space for each point to consist of only the 8 surrounding boxes and its own box, rather than all boxes in the grid space. After binning each block, we created a linked list of bin addresses. Knowing that for i in [-1,1] and j in [-1,1], $z + (i * \ \#\text{columns}) + j$ represents the index of all 9 bins, or neighbors (e.g. bins 6-18 in Figure 1), in a computation space relative to a bin with index z, we defined the computation space as such, and computed 'apply_force' for each point with all points in this space.

For the cache complexity, we leveraged the fact that our bins are ordered by row, and so by accessing and storing data using row-major, we could optimize our cache using spatial locality. To build our bins, we used a linked list of particles ordered by bin index. In our linked list, we defined all points in the computation space for the corresponding bin and set all other nodes to be null. Our algorithm then traverses the linked list and computes the force of all points on all other points in the list while ignoring nodes that are null. After computing the applied force for a 9-bin computation space, we then unbin the particles and repeat the process for each time step. By doing so, we reuse our memory allocation in our linked list with each time step.

As shown in our results below, we were able to achieve a time complexity of $O(n)$, and sometimes even better.

3 Parallel Algorithm - OpenMP

To parallelize the particle simulation algorithm, we used OpenMP pragmas for the for loops that traverse the bins in the grid and compute the forces between neighbors. Since the underlying data structure uses linked lists, the binning process must be done more or less sequentially as adding to any linked list or vector requires mutually exclusive writes. However, we leveraged omp_set_lock and omp_unset_lock to reduce the sequential costs, and by doing so, were able to improve our speedup.

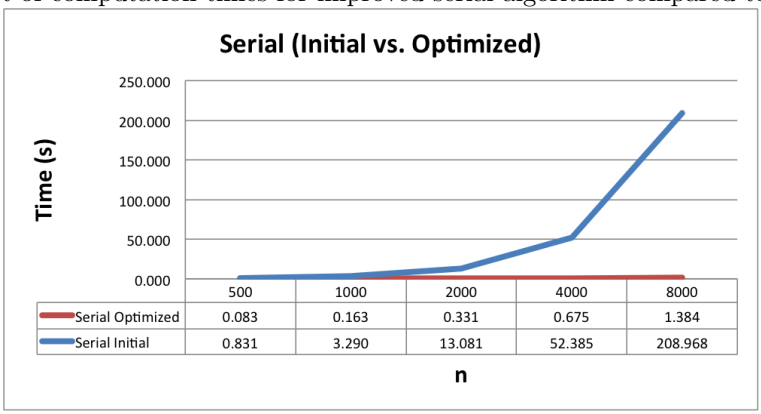
4 Parallel Algorithm - MPI

First, we approached our problem by considering the distribution of bins in 1D, with a particle space divided into columns. As each processor is responsible for a set of columns (evenly dividing, if possible, the total number of columns in the grid), given the barrier between each of the processor's territories, our algorithm ensures that particles within the cutoff range of a barrier is shared with the neighboring processor for accurate force calculations. By sending the bin index, the number of affected particles, and the actual particles to the neighboring processor, each processor can compute the force of all particles within its purview, but avoid excess computations of irrelevant particles. Using vectors for our particle counts, indices of particles, and particle entities, we use MPI_Isend and MPI_Recv to distribute our particles from rank 0 processor to the other processors. When handling border cases, we built a function to determine the particles on the border of each processor's territory (within the cutoff range of the border). We then used our $dist_particles$ function with MPI_Isend and MPI_Irecv to send the count of border particles and particles to their respective neighbors. Once forces have been computed for each processor's respective particles, then each processor sends its count of particles and particles to the rank 0 processor using MPI_Isend and rank 0 receives the particles with an MPI_Recv.

5 Results

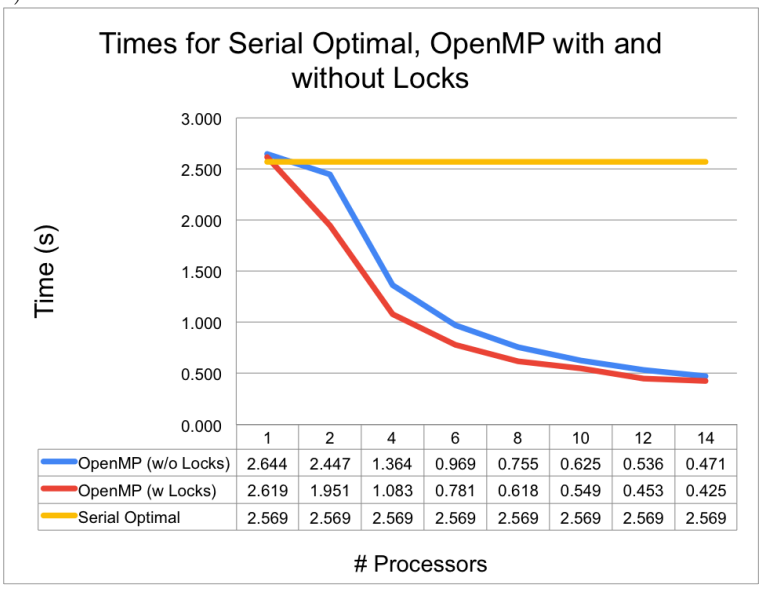
By incorporating binning, bin force computations, and row-major traversal, we were able to reduce the time complexity from $O(n^2)$ to $O(n)$, as shown in Figure 3. Compared to the slope estimate for the initial line fit of 1.994, the slope estimate for our improved serial algorithm is 1.018.

Figure 2: Plot of computation times for improved serial algorithm compared to initial serial



	500	1000	2000	4000	8000
Serial Optimized	0.083	0.163	0.331	0.675	1.384
Serial Initial	0.831	3.290	13.081	52.385	208.968

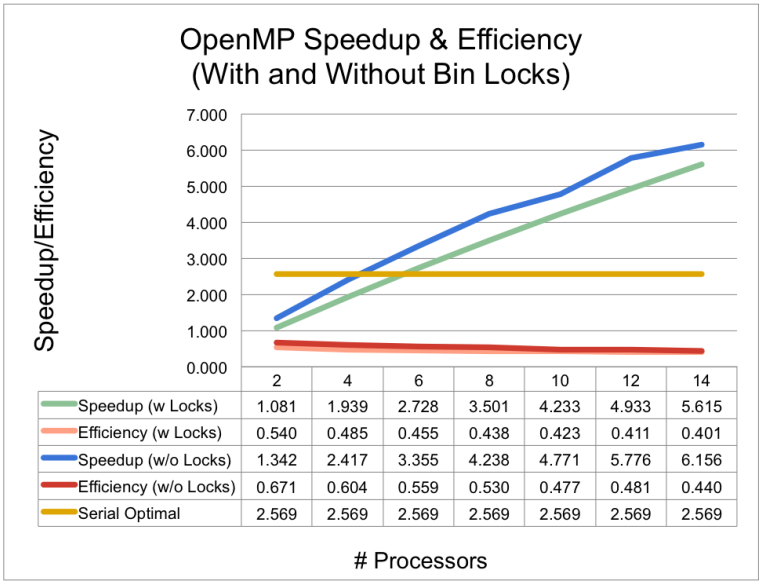
Figure 3: Plot of computation times for improved serial algorithm and OpenMP algorithm with and without locks (15,000 particles)



	1	2	4	6	8	10	12	14
OpenMP (w/o Locks)	2.644	2.447	1.364	0.969	0.755	0.625	0.536	0.471
OpenMP (w Locks)	2.619	1.951	1.083	0.781	0.618	0.549	0.453	0.425
Serial Optimal	2.569	2.569	2.569	2.569	2.569	2.569	2.569	2.569

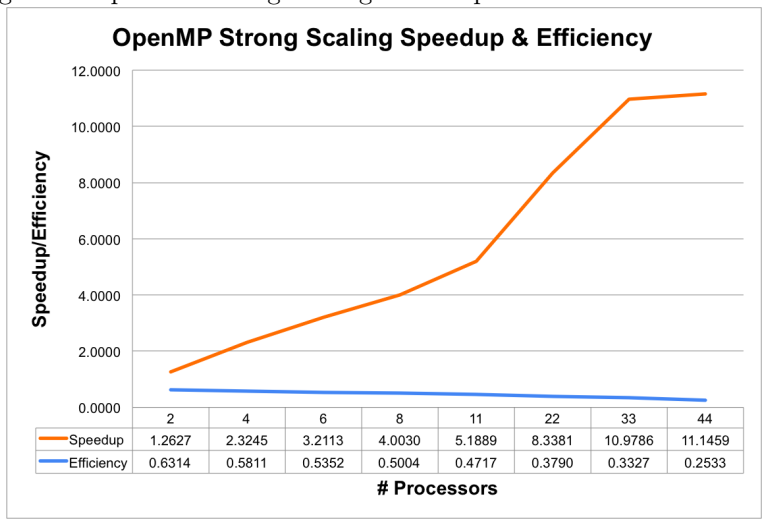
For the OpenMP parallel optimization, we achieved speedup when testing with 15,000 particles. As the number of processors increased, the speedup increased. However, the efficiency slowly began to decline with more processors. To optimize our algorithm, we chose to remove locks when binning the particles, which reduced the sequential cost of binning. Consequently, we improved our speedup and efficiency, as shown in Figure 4. In Table 1, we see that upon removing the locks, the percentage of time spent on binning the particles was reduced to approximately 1/3 of the initial allocation with locks. Although the percentage of time spent applying the force to particles increased, the overall time decreased when removing the locks (see Figure 3). As shown, using OpenMP and unlocking the bins, we were able to reduce the total time by 80 percent with 14 processors for 15,000 particles.

Figure 4: Parallel Speedup and Efficiency for OpenMP with and without Bin Locks (15,000 particles)



	2	4	6	8	10	12	14
Speedup (w Locks)	1.081	1.939	2.728	3.501	4.233	4.933	5.615
Efficiency (w Locks)	0.540	0.485	0.455	0.438	0.423	0.411	0.401
Speedup (w/o Locks)	1.342	2.417	3.355	4.238	4.771	5.776	6.156
Efficiency (w/o Locks)	0.671	0.604	0.559	0.530	0.477	0.481	0.440
Serial Optimal	2.569	2.569	2.569	2.569	2.569	2.569	2.569

Figure 5: OpenMP Strong Scaling with Improved Bin Forces Function



	2	4	6	8	11	22	33	44
Speedup	1.2627	2.3245	3.2113	4.0030	5.1889	8.3381	10.9786	11.1459
Efficiency	0.6314	0.5811	0.5352	0.5004	0.4717	0.3790	0.3327	0.2533

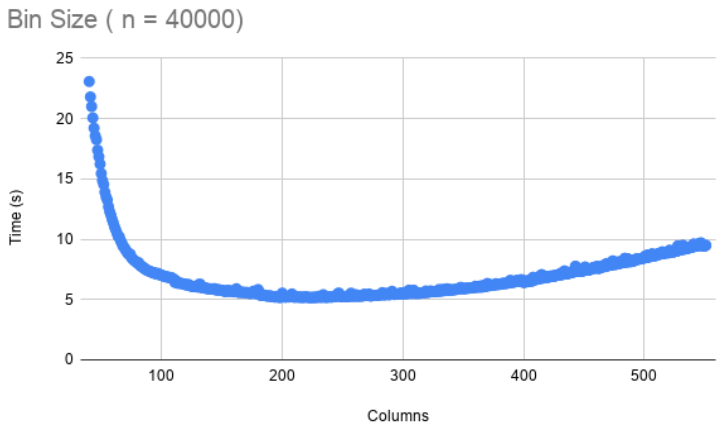
Table 1: OpenMP Program Component Allocation of Time with and without Bin Locks (15,000 particles)

With Bin Locks							
Num_Particles	Processors	Total	Bin	Force	UnBin	Reset	Move
15000	1	2.64	14%	79%	1%	4%	3%
15000	2	2.45	30%	64%	2%	2%	2%
15000	4	1.36	31%	63%	3%	2%	2%
15000	6	0.97	31%	62%	3%	2%	2%
15000	8	0.76	31%	62%	3%	2%	2%
15000	10	0.62	31%	62%	3%	3%	2%
15000	12	0.54	30%	61%	4%	3%	2%
15000	14	0.47	30%	61%	4%	3%	2%
Without Bin Locks							
Num_Particles	Processors	Total	Bin	Force	UnBin	Reset	Move
15000	1	2.62	5%	85%	3%	4%	3%
15000	2	1.95	8%	81%	3%	3%	2%
15000	4	1.08	8%	80%	6%	3%	2%
15000	6	0.78	8%	78%	7%	3%	2%
15000	8	0.62	8%	77%	8%	4%	2%
15000	10	0.55	8%	77%	8%	4%	3%
15000	12	0.45	9%	76%	8%	4%	3%
15000	14	0.43	9%	75%	8%	4%	3%

To investigate the parallel slowdown, we ran some experiments with varying bin size. We found that the bin size that corresponded to the best performance of the OpenMP algorithm corresponded to about twice as long as the cutoff (225 instead of 450). With further experimentation with other varying parameters, we can find a correlation that causes this curve and replicate it explicitly in the algorithm.

To find where the algorithm experiences the most latency, we measured a breakdown of the timings in 1. We noticed that the most latency occurs with applying the force, which is to be expected as it is the $O(C * n)$ complexity task. However, the binning also contributed a fair amount in OpenMP. This seems to be due to the overhead of locks and unlocks to avoid race conditions while binning to the same bins.

Figure 6: Bin Size Experiments



Despite our efforts to parallelize our algorithm using MPI, our results were lackluster. As we increased the number of particles and increased the number of processors, we expected the speedup to improve given the reduced ratio of overhead to parallelized work across processors. At a point, we thought we would surpass an overhead limit and our algorithm would dominate. However, even when testing with 100,000 particles, we saw negligible speedup (see Figure 6). As shown, we see that regardless of changing the number of particles, the speedups and efficiency did not change when increasing then number of processors. We predict this is because of the overhead from the computation of forces, which we did not optimize (see Figure 7). As shown, the speedup for binning, distributing, and moving the particles was decent, given its improved values across processors. On the contrary, the speedup for computing the force was negligible. Our binning method was quite impressive, as we reached a speedup of approximately 27 with 32 processors.

While finishing the write-up, we made a slight adjustment to our bin_forces algorithm, which encapsulates the apply_force subroutine by iterating over the neighbors of a particular bin and applying forces to the

particles in that bin. We decided to switch the ordering of the loops such that the same particle is updated multiple times consecutively, hopefully improving the cache complexity of this operation. This improved the timings greatly which obviously improved our MPI speedup as well as the OpenMP implementation which shares the same function.

Figure 7: MPI Speedups and Efficiencies for 100k particles

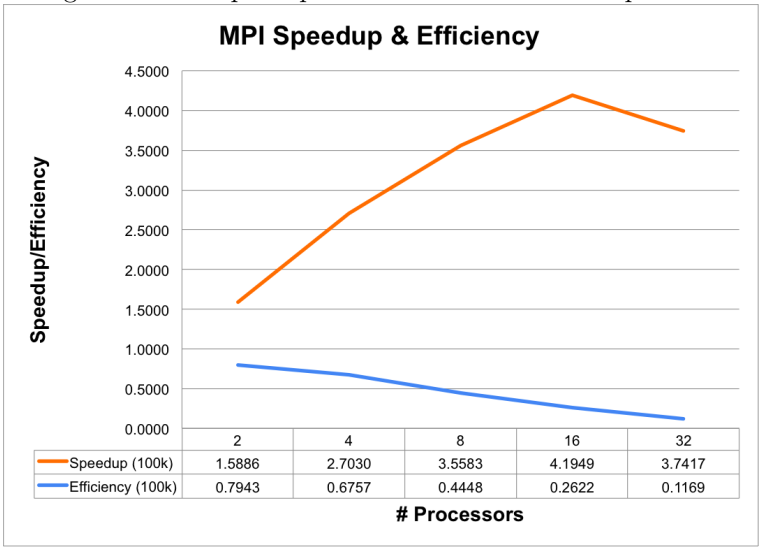


Figure 8: MPI Strong Scaling Speedups for Component Analysis

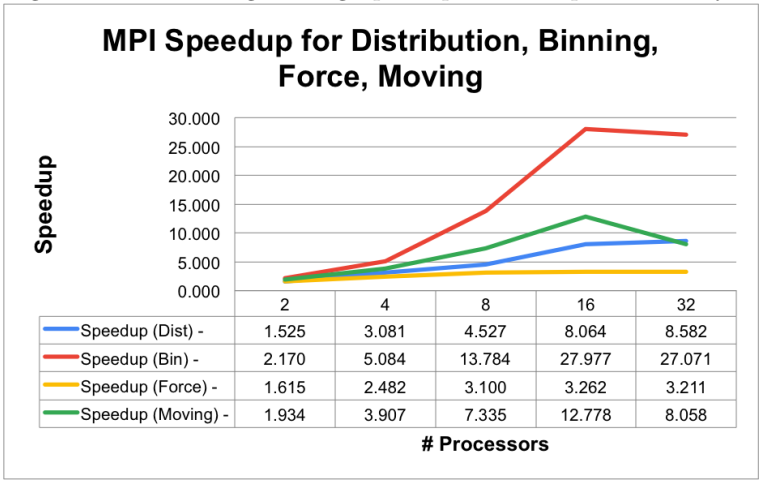
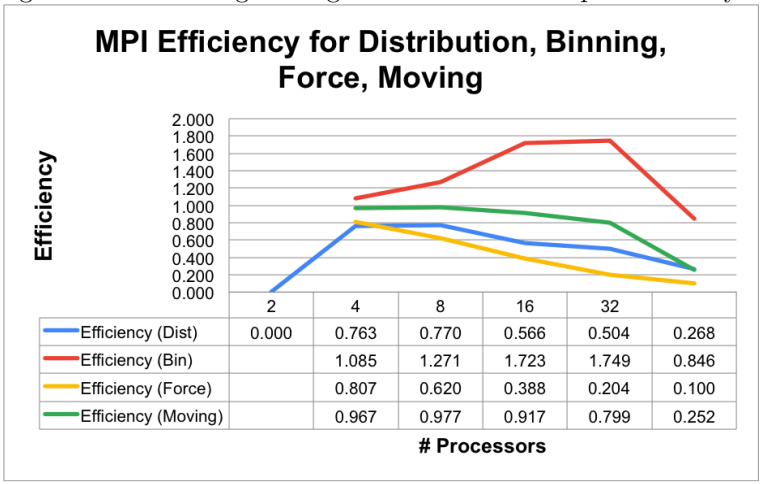
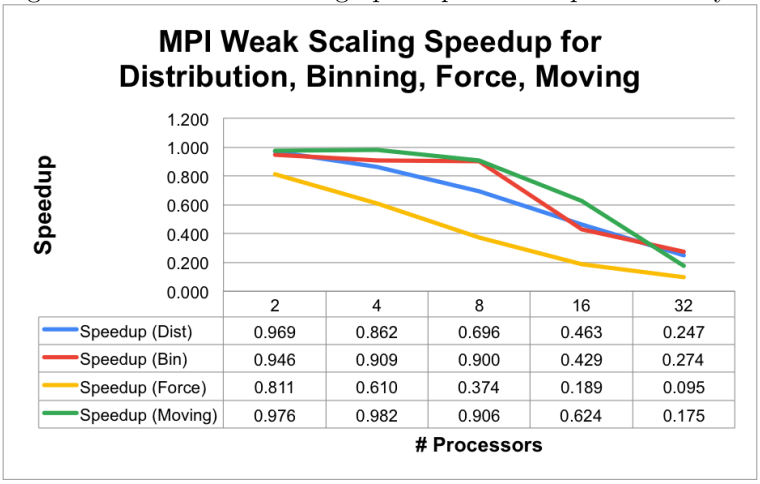


Figure 9: MPI Strong Scaling Efficiencies for Component Analysis



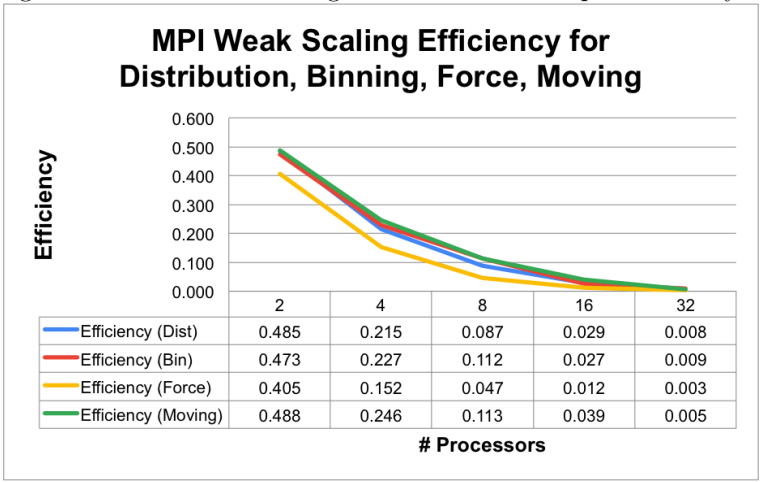
	2	4	8	16	32	
Efficiency (Dist)	0.000	0.763	0.770	0.566	0.504	0.268
Efficiency (Bin)		1.085	1.271	1.723	1.749	0.846
Efficiency (Force)		0.807	0.620	0.388	0.204	0.100
Efficiency (Moving)		0.967	0.977	0.917	0.799	0.252

Figure 10: MPI Weak Scaling Speedups for Component Analysis



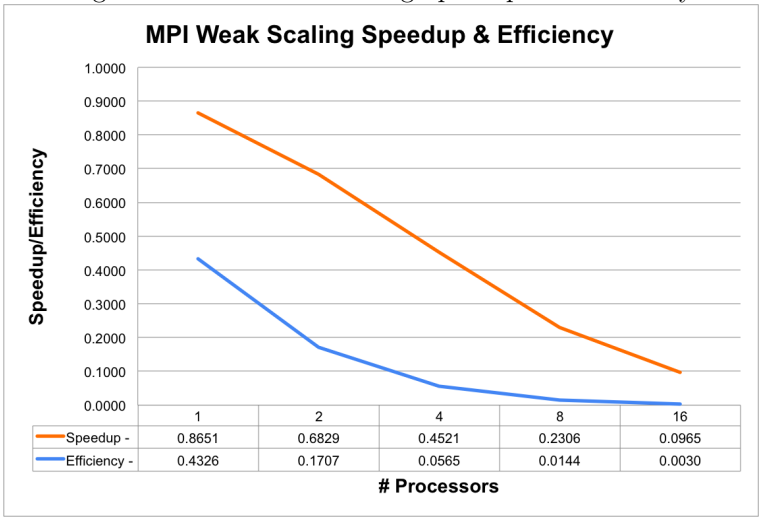
	2	4	8	16	32
Speedup (Dist)	0.969	0.862	0.696	0.463	0.247
Speedup (Bin)	0.946	0.909	0.900	0.429	0.274
Speedup (Force)	0.811	0.610	0.374	0.189	0.095
Speedup (Moving)	0.976	0.982	0.906	0.624	0.175

Figure 11: MPI Weak Scaling Efficiencies for Component Analysis



	2	4	8	16	32
Efficiency (Dist)	0.485	0.215	0.087	0.029	0.008
Efficiency (Bin)	0.473	0.227	0.112	0.027	0.009
Efficiency (Force)	0.405	0.152	0.047	0.012	0.003
Efficiency (Moving)	0.488	0.246	0.113	0.039	0.005

Figure 12: MPI Weak Scaling Speedup and Efficiency



6 Conclusion

In conclusion, using a binning strategy and optimizing spatial locality, we achieved linear time complexity for the serial algorithm. As we parallelized the particle simulation using OpenMP, we achieved positive speedups and were able to optimize our algorithm further by removing bin locks and by increasing the number of particles to surpass the overhead limit. Finally, using MPI, we were able to gain slight speedup, but nothing noteworthy. Although our binning strategy was the most impactful optimization of our algorithm, yielding surprising results, it was not enough to overtake the poor performance of the $apply_forces$ function.