CSC 726: Parallel Algorithms
MPI Assignment Part 2

Irina Viviano & Esteban Murillo

Thursday 7th November, 2019

1 Serial Algorithm

1.1 Overview of Algorithm and Implementation

In this project, we simulate particle interaction, an application applicable to fields such as mechanics, biology, and astronomy. When particles reach a certain cutoff radius, they repel each other. A naïve implementation of the serial algorithm computes the forces between all pairs of particles, which, for n particles, is a $\mathcal{O}(n^2)$ algorithm.

In order to improve this algorithm, we consider the fact that not all particles necessarily interact within a given time step, that is, particles that are far enough apart will not interact. Thus, we only want to compute forces between particles that are close enough together. In our efficient implementation, we therefore divide the grid space in which the particles are located into bins, and only compute the force between particles in neighboring bins.

Our algorithm works as follows. First, we compute the number of bins. Observe that the grid size is always $\sqrt{\text{density} \times n}$, where the density is defined to be 0.0005 and n is the number of particles. Since the cutoff is the radius defining where particles interact, we set the length of the bins to be cutoff $\times$ 2, where the cutoff is defined to be 0.01. Thus, the number of bins is:

$$\left\lceil \frac{\sqrt{\text{density} \times n}}{\text{cutoff} \times 2} \right\rceil \tag{1}$$

Next, for each particle, we compute which bin a particle belongs to. This is done by first computing the x-offset and y-offset of a particle, which is simply the floor of the particle's x or y position divided by the length of a bin. Then, using column indexing, we are able to compute the index of the bin in which a particle belongs, and we store this information in a 2D vector. Observe that this can be done in $\mathcal{O}(1)$ time. Since we do this for all n particles, this computation is $\mathcal{O}(n)$.

We then loop through all the time steps. For each time step, we need to consider each particle so we can compute the force between itself and the particles neighboring it. For each particle, we compute an x-offset and y-offset so that we know which bin the current particle belongs to. The neighboring bins of this (x, y) position in the grid are therefore quite simple: just the bins with offsets $(x-1, y-1), (x-1, y), (x-1, y+1), (x, y-1), (x, y), (x, y+1), (x+1, y-1), (x+1, y)$, and $(x+1, y+1)$. We then must determine which grid values are valid since we do not want to attempt to access a neighbor that is out of the bounds of the grid. We can do this by recognizing that for the given x-offset and y-offset, the valid values for the (i, j)th bin are simply i from $\max(0, x-1)$ to $\min(x+1, \texttt{num\_bins})$, inclusive, and j from $\max(0, y-1)$ to $\min(y+1, \texttt{num\_bins})$, inclusive. Now that we know what the x-offset and y-offset is for each of our neighboring bins, we can compute the index of the bin using column indexing so that we can then index into

our data structure which keeps track of which particles are located in which bin. Finally, we loop through each of the neighboring bin indices (no more than 9 neighbors are possible) and compute the force between the current particle and the particles in the current neighboring bin.

Because the density of particles is maintained to be constant, there will be far fewer particles in a bin than n, and so the force computation is $\mathcal{O}(n)$. Finally, after the computations in each time step, we update the the 2D vector of bins to reflect which particles belong to which bin after they have all moved. This, again, is $\mathcal{O}(n)$. Thus, the overall complexity of our efficient serial algorithm is $\mathcal{O}(n)$, as desired.

1.2 Performance Results

Experimental results, shown in Figure 1, also support our analysis. We can see that the naive algorithm times follow a curve that is $\mathcal{O}(n^2)$, while our efficient algorithm times follow a curve that $\mathcal{O}(n)$. Indeed, the slope estimate for the naive algorithm was 2.016752, while the slope estimate for our improved algorithm was 1.117185. The raw data is summarized in Table 1.

Number of Particles	**Naïve Time (s)**	**Efficient Time (s)**
500	0.580876	0.076875
1000	2.29811	0.167468
2000	9.0795	0.364162
4000	37.4502	0.803102
8000	156.155	1.68615

Table 1: *Raw timing data for the naive and efficient algorithms.*

2 Parallel Algorithm using Shared Memory Model (OpenMP)

2.1 Overview of Algorithm and Implementation

We parallelize our efficient serial algorithm using OpenMP, which is based on a shared memory model. First, we compute the 2D vector which assigns particles to bins. We can do this in parallel using `#pragma omp parallel for`, making sure to share the bins data structure, which will divide the particles among the threads. The work to compute the bin that a particular particle belongs to is the same as in the serial case. However, when updating our 2D vector, we put a `#pragma omp critical` to ensure no data races and write conflicts ensue. Since this is a parallel `for`-loop ranging over all n particles and does $\mathcal{O}(1)$ work within, the cost is $\mathcal{O}(\log n)$. However, because of the critical section, each processor must perform this data write sequentially, causing the overall cost of this computation to be $\mathcal{O}(n)$. Thus, this computation is actually the bottleneck of our algorithm. (We attempted to use locks instead of a critical section in order to speed up the computation, but were unable to debug the deadlocks or segmentation fault on the lock that we were getting. We are still mystified as to why this was occurring, and it would be nice to take a look at the lock usage in the code together to see if we have some simple error in how we are using OpenMP).

We then enter the main parallel section. At this point, we ensure that the 2D vector containing all the bin information is a shared variable among all the threads. This ensures that each thread will be able to access the neighbors of its own particles in order to compute the forces. We also ensure that each thread has its own local copy of the variables used in the computation of a particle's bin.

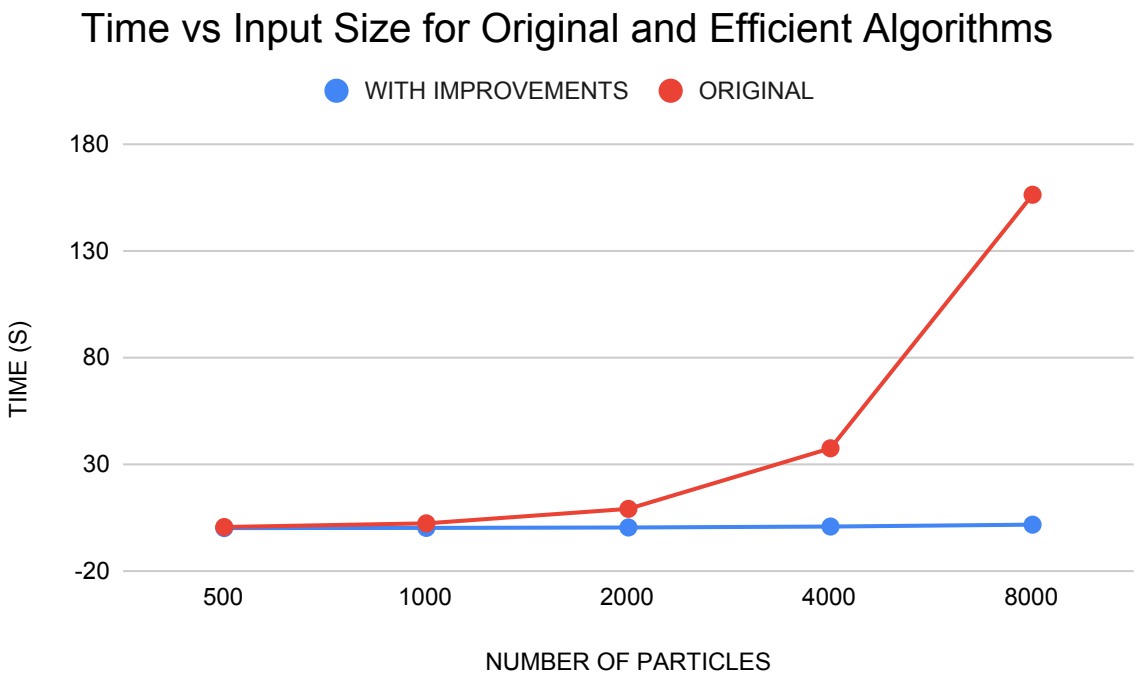

Figure 1: *Comparison of running time, in seconds, of original serial code and improved serial code. We can see the original code performs as expected with a $\mathcal{O}(n^2)$ curve, while the efficient algorithm is a $\mathcal{O}(n)$ curve.*

The next step is to loop through all the particles in parallel. Within this parallel `for`-loop, we have the same computations and loops for computing the force of the current particle with its neighbors as in the serial case. It is possible to put `#pragma omp parallel for`'s before our inner `for`-loops, but we chose not to do so. The reason for this is that it causes our parallel algorithm to slow down considerably, most likely due to the overhead of spawning threads, especially since the range of the loop indices is not very large so there is not a sufficient amount of work in the loops that requires parallelizing them explicitly. Note that we do not need to include a critical section here since we are only accessing shared memory locations, not writing to them. Given that there are at most 9 neighbors to consider and, because the density of the particles is constant, there are fewer than n particles in a bin, this means that the inner three loops cost $\mathcal{O}(9d)$, where d is the average number of particles in a bin. Since this occurs in a parallel `for`-loop, the overall cost is $\mathcal{O}(d \log n)$.

The final step, as in the serial case, is to update the 2D vector containing the bins each particle belongs to. We can do this with a `#pragma omp parallel for`, remembering to include a `#pragma omp critical` to ensure no data races and write conflicts ensue. Again, this is $\mathcal{O}(\log n)$ but because of the critical section, each processor must perform this data write sequentially, causing the overall cost of this computation to be $\mathcal{O}(n)$.

Therefore, the overall cost of our parallel algorithm is $\mathcal{O}(n)$, since we have the bottleneck of the critical section when computing the bins. In the case that the locks were working, the cost of our parallel algorithm would end up being $\mathcal{O}(\log n)$. Since our parallel algorithm has the same complexity as our serial algorithm, except it also has the additional overhead of the thread management, this explains why our algorithm runs faster in serial than in parallel.

2.2 Performance Results

We also perform both strong and weak scaling experiments of our efficient OpenMP parallel algorithm compared to the naïve parallel algorithm. The strong scaling timings are summarized in Figure 2 and the strong scaling speedups are summarized in Figure 3. We can see that our algorithm performs much faster than the naïve parallel algorithm and actually slows down instead of speeds up. Furthermore, the rate of the slowdown is worse than that of the naïve algorithm. The weak scaling timings are summarized in Figure 4 and the weak scaling speedups are summarized in Figure 5. For the weak scaling experiments, the amount of work was kept constant at $\frac{n}{p} = 500$ particles. As can be seen in Figure 4, our algorithm scales much better than the naïve algorithm. Our speed up, as shown in Figure 5, is actually a slow down, but we have better results than the naïve parallel algorithm. The raw timing and speedup data is summarized in Table 2.

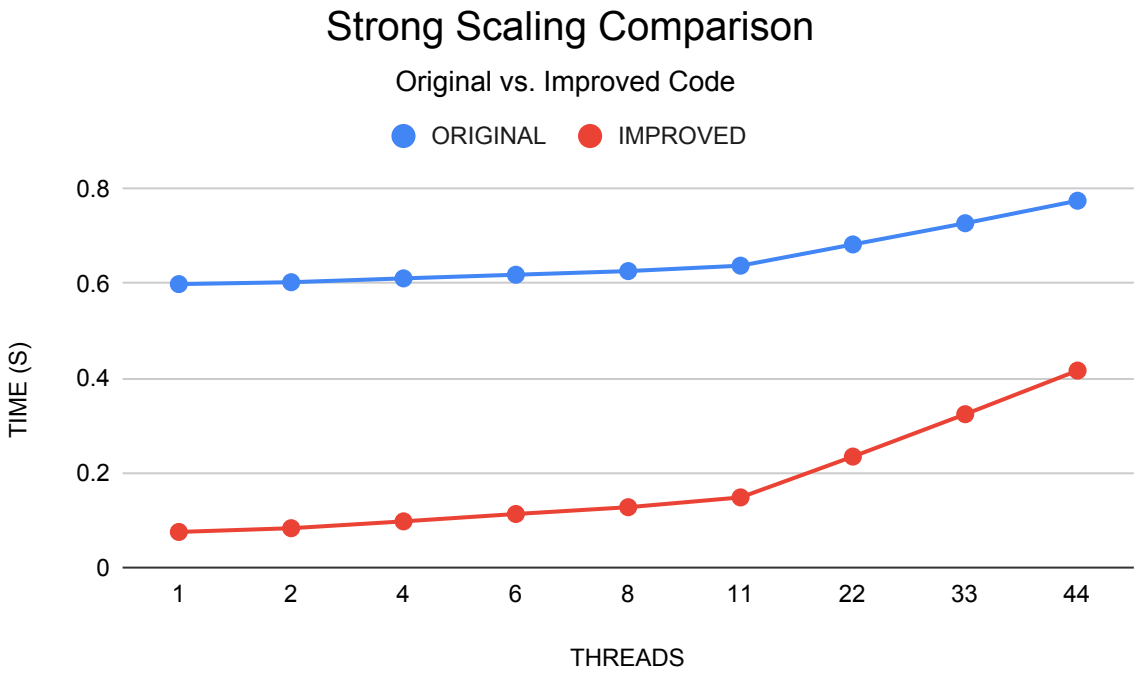

Figure 2: *Comparison of running time, in seconds, of original parallel code and improved parallel code for strong scaling experiments. We can see the improved code performs much faster.*

3 Parallel Implementation using Distributed Memory Model (MPI)

3.1 Overview of Algorithm and Implementation

We also parallelize our efficient serial algorithm using MPI, which is based on a distributed memory model. Now, all the particles are split up between the processors, with each processor only being able to "see" their chunk of the data. Therefore, when computing the 2D vector of bins, each processor only computes the bins of the particles they own. Now, since each processor

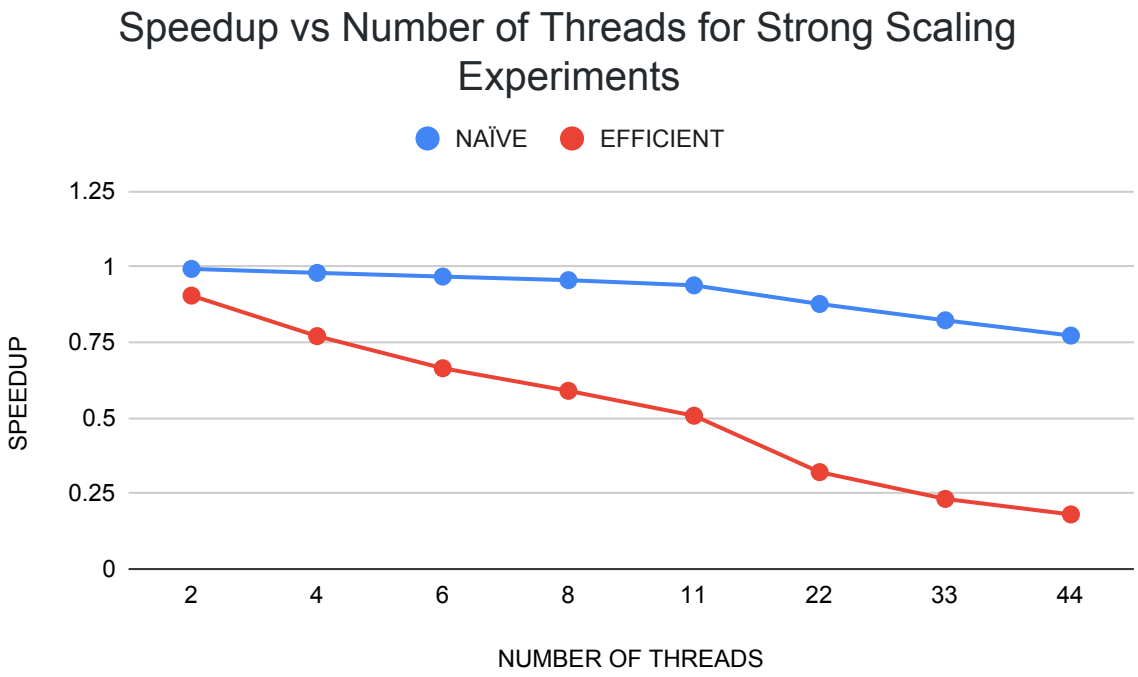

Figure 3: *Comparison of speedup of original parallel code and improved parallel code for strong scaling experiments. We can see that our improved algorithm slows down instead of speeding up.*

has their own 2D vector of bins, no data races occur, so this computation is indeed $\mathcal{O}(\log n)$. Since all the processors need to be able to know what particles are neighboring the location of their given particles, we have each processor send their computed bins to processor 0. Processor 0, in turn, collects this data in order to create the master 2D vector keeping track of which particles belong in which bin. Processor 0 then needs to broadcast this master 2D vector to all the other particles. Since this is all-to-all communication, the cost of this is likely expensive and the bottleneck of our algorithm.

The rest of the algorithm is much the same as our OpenMP code. An Allgather is used to distribute the particles to each processor (though this communication is once again quite expensive as it is all-to-all), and, with this, each processor will have the necessary information they need to work independently. Each processor loops through its local chunk of the particles, computes the neighboring bins, and computes the force between the current particle and all the neighboring particles. As in our OpenMP analysis, the cost of our inner `for`-loops is $\mathcal{O}(9d)$.

At the end of each time step, each processor must once again update their bins and send the updated bins to processor 0, who then compiles the master 2D vector, and broadcasts this information back to the processors so they can be prepared for the next iteration. Again, this is the bottleneck of our algorithm, and explains why the overall complexity of our algorithm is greater than $\mathcal{O}(\log n)$ and the run time in parallel is slower than the run time in serial.

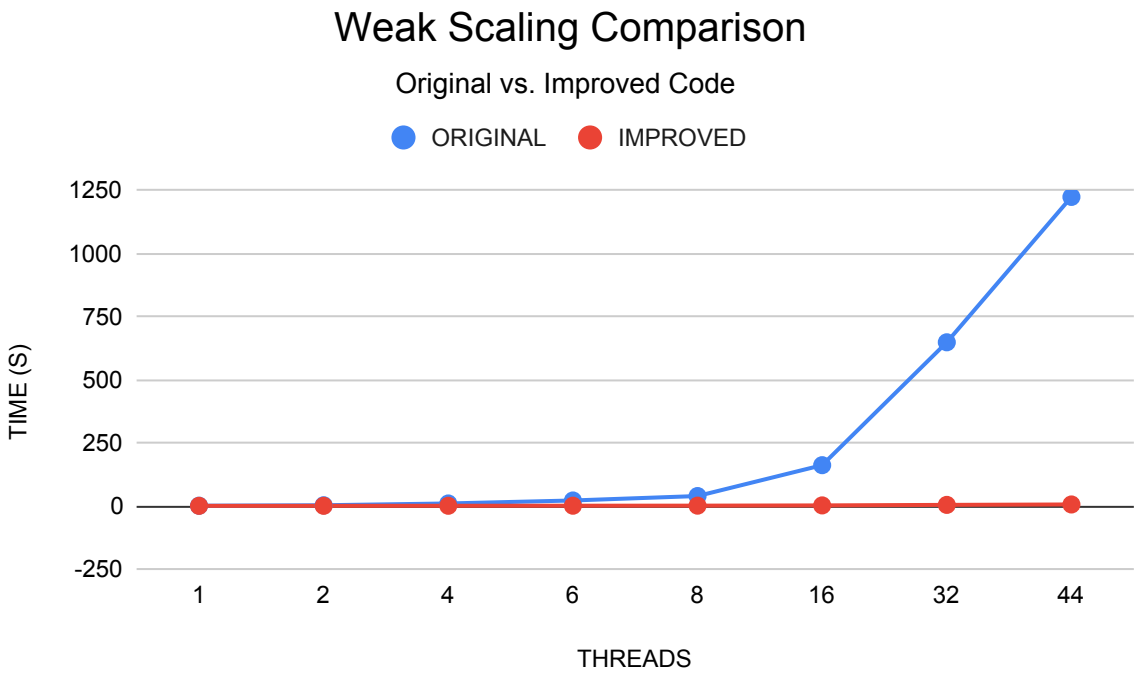

Figure 4: *Comparison of running time, in seconds, of original parallel code and improved parallel code for weak scaling experiments. The amount of work was fixed at* $\frac{n}{p} = 500$ *particles. Our algorithm scales much better than the naïve algorithm.*

3.2 Performance Results

We also perform strong scaling experiments of our efficient MPI parallel algorithm compared to the naïve parallel algorithm. The strong scaling timings are summarized in Figure 6 and the strong scaling speedups are summarized in Figure 7. We can see that both algorithms have identical performance. Despite the improvements to the algorithm itself, our implementation has a lot of overhead from processor communication which is likely causing the slowdown. The raw timing and speedup data is summarized in Table 3.

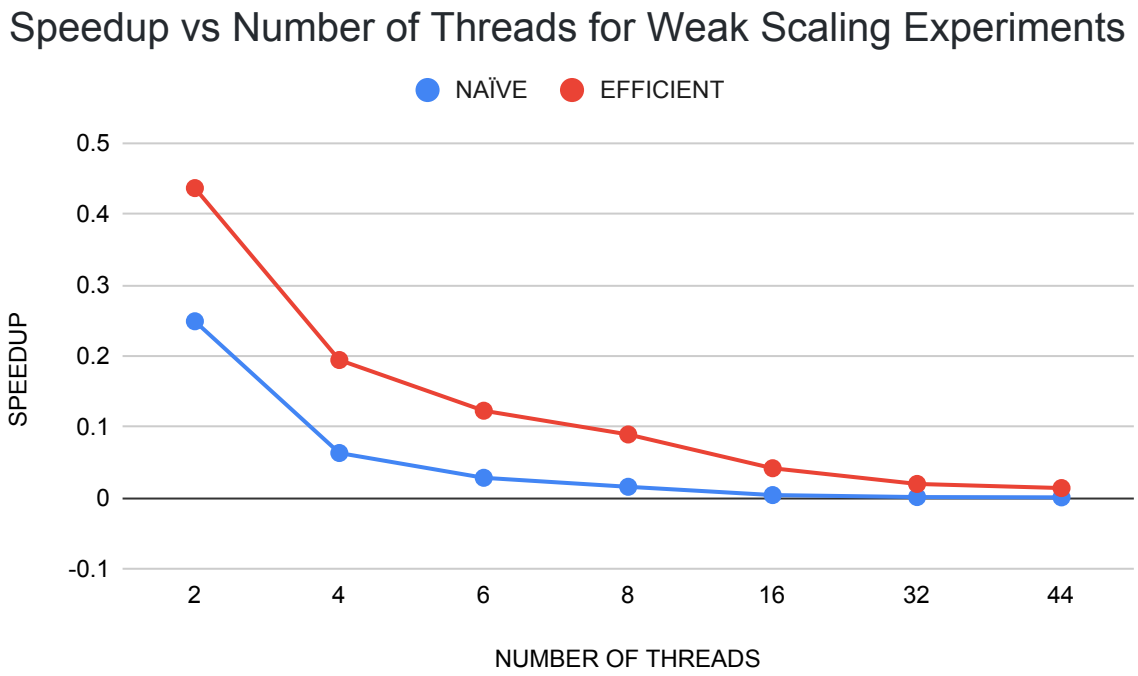

Figure 5: *Comparison of speedup of original parallel code and improved parallel code for weak scaling experiments. We slow down instead of speeding up, but we perform better than the naïve algorithm.*

Strong Scaling Results					
Particles	**Threads**	**Naïve Time (s)**	**Efficient Time(s)**	**Naïve Speedup**	**Efficient Speedup**
500	1	0.598843	0.075133	N/A	N/A
500	2	0.602856	0.083015	0.9933433523	0.9050533036
500	4	0.610955	0.097474	0.9801752993	0.7708004186
500	6	0.618638	0.113069	0.9680022889	0.664488056
500	8	0.626348	0.127398	0.9560867122	0.5897502316
500	11	0.637795	0.148107	0.9389270847	0.5072886494
500	22	0.68264	0.234398	0.8772456932	0.3205360114
500	33	0.727517	0.323938	0.8231326553	0.2319363582
500	44	0.775063	0.416147	0.7726378372	0.1805443749

Weak Scaling Results					
Particles	**Threads**	**Naïve Time (s)**	**Efficient Time (s)**	**Naïve Speedup**	**Efficient Speedup**
500	1	0.598843	0.075133	N/A	N/A
1000	2	2.40831	0.172185	0.248656942	0.436350437
2000	4	9.50596	0.387167	0.0629965832	0.1940583779
3000	6	21.3379	0.612818	0.02806475801	0.1226024692
4000	8	38.9223	0.84294	0.01538560157	0.08913208532
8000	16	161.202	1.80896	0.003714860858	0.04153380948
16000	32	648.117	3.87181	0.0009239736035	0.0194051361
22000	44	1224.09	5.5295	0.0004892148453	0.01358766615

Table 2: *Raw timing data for the OpenMP naïve and efficient algorithms.*

Strong Scaling Results					
Particles	**Threads**	**Naïve Time (s)**	**Efficient Time (s)**	**Naïve Speedup**	**Efficient Speedup**
2000	1	0.351557	0.307717	N/A	N/A
2000	2	1.22984	1.20006	0.2858558837	0.2564180124
2000	4	2.7264	2.76623	0.1289454959	0.1112405693
2000	8	6.07749	6.04545	0.0578457554	0.05090059466
2000	16	15.2751	15.6951	0.02301503754	0.01960592796
2000	32	37.6319	38.0153	0.009341994425	0.008094556665

Table 3: *Raw timing data for strong scaling results for the MPI naïve and efficient algorithms.*

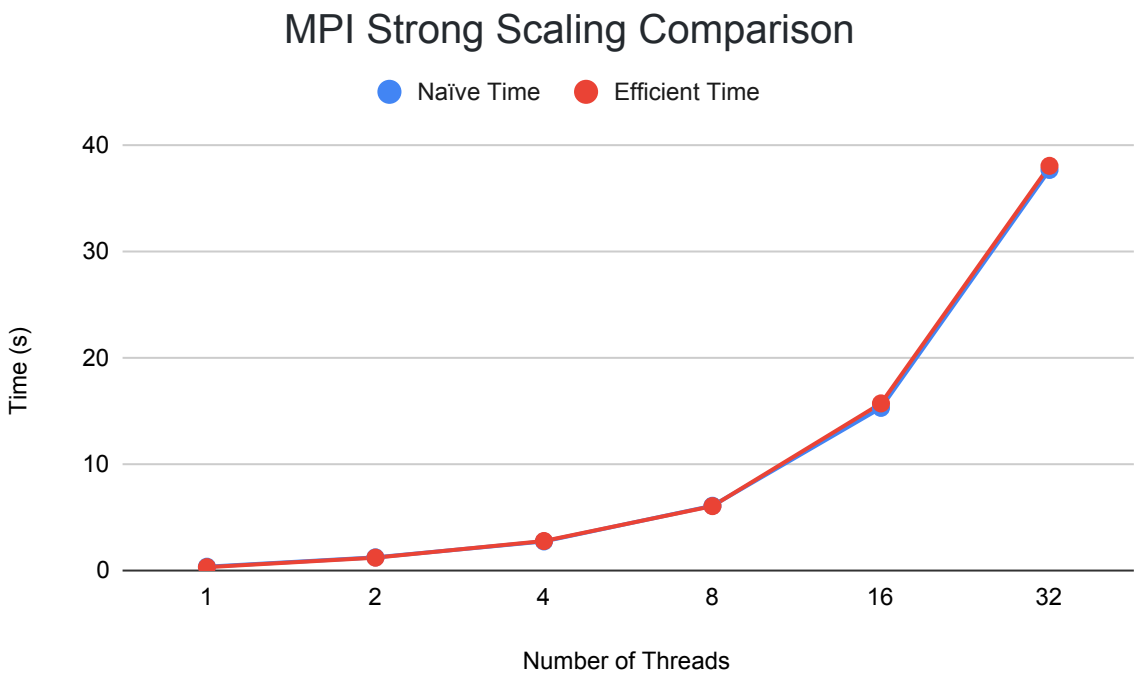

Figure 6: *Comparison of running time, in seconds, of original MPI parallel code and improved MPI parallel code for strong scaling experiments. Both algorithms have identical performance.*

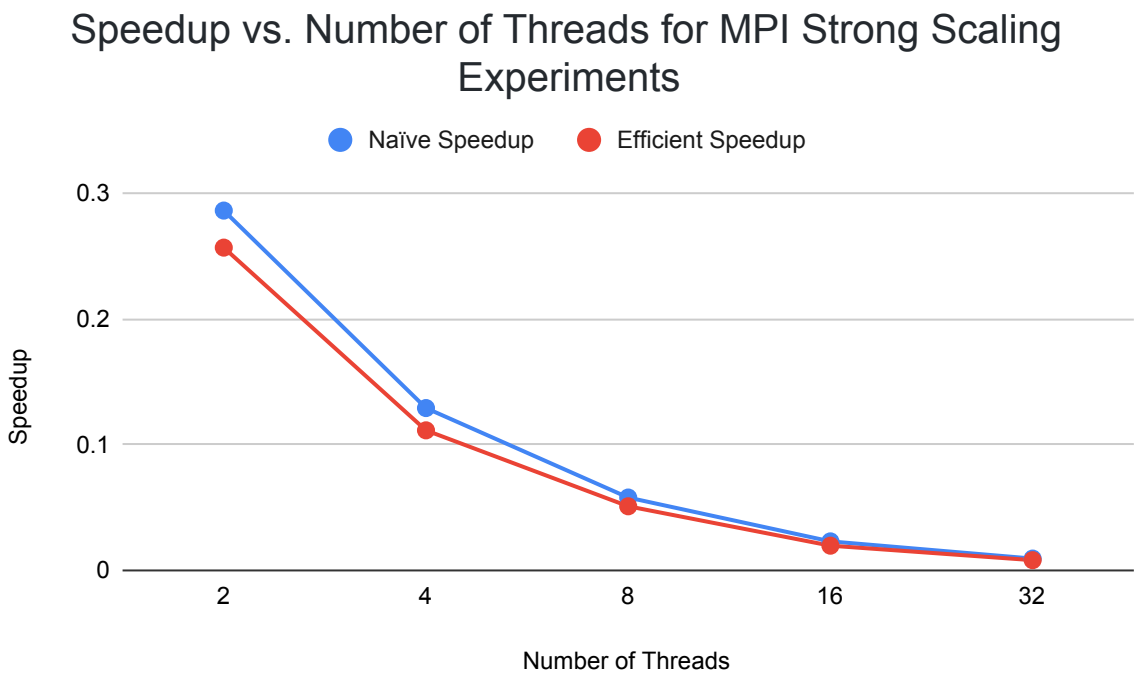

Figure 7: *Comparison of running time, in seconds, of original MPI parallel code and improved MPI parallel code for strong scaling experiments. Both algorithms have identical performance.*