CSC 726: Parallel Algorithms
MPI Assignment Part 1

Irina Viviano & Esteban Murillo

Sunday 27th October, 2019

Serial Algorithm

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. First, we again compute which bin the current particle belongs to, which can be done in $\mathcal{O}(1)$ time. We then compute the valid bins neighboring the current particle's bin, call it b. Essentially, we use several `if-else` statements to determine if b is in the top row (when the floor b divided by the number of bins equals 0), in the bottom row (when the floor of b divided by the number of bins equals one less than the number of bins), in the top row (when b mod the number of bins is equivalent to 0), in the bottom row (when b mod the number of bins is equivalent to one less than the number of bins), and the corner cases can be computed by a combination of these four cases. This computation can be done in $\mathcal{O}(1)$ time. We then 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.

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 1.066136. The raw data is summarized in Table 1.

Number of Particles	**Naïve Time (s)**	**Efficient Time (s)**
500	0.580876	0.14909
1000	2.29811	0.310214
2000	9.0795	0.648145
4000	37.4502	1.36569
8000	156.155	2.85955

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

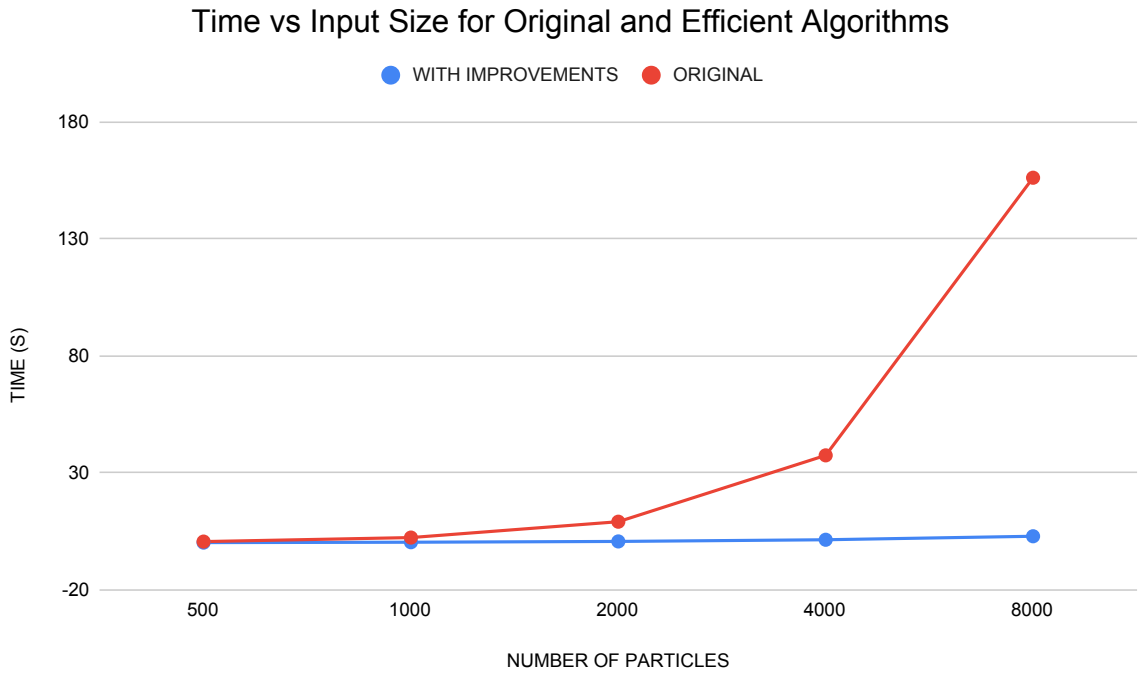

Figure 1: *Comparison of running time, in seconds, of original serial code and improved serial code. We can see the original code performs as expected with a $\mathcal{O}(n^2)$ curve, while the efficient algorithm is a $\mathcal{O}(n)$ curve.*

Parallel Algorithm using Shared Memory Model (OpenMP)

We parallelize our efficient serial algorithm using OpenMP, which is based on a shared memory model. First, we compute the 2D vector of which assigns particles to bins. We can do this in parallel using `#pragma omp parallel for`, 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 greater, causing this to actually be the bottleneck of our algorithm.

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 certain variables (vector of neighbors of the current particle and variables used in the computation of a particle's bin).

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 both our `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. 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 two 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 greater, causing this to actually be the bottleneck of our algorithm.

We also perform both strong and weak scaling experiments of our efficient 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 slightly faster and has considerable gains in speedup. 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, depending on the number of threads used, a slowdown may occur. We believe the reason for this is that spawning threads causes a considerable amount of overhead. More specifically, this happens when the number of threads is four. After this point, we continue to see a speedup, meaning that the overhead of spawning threads does not overpower the cost of the computations. The raw timing and speedup data is summarized in Table 2.

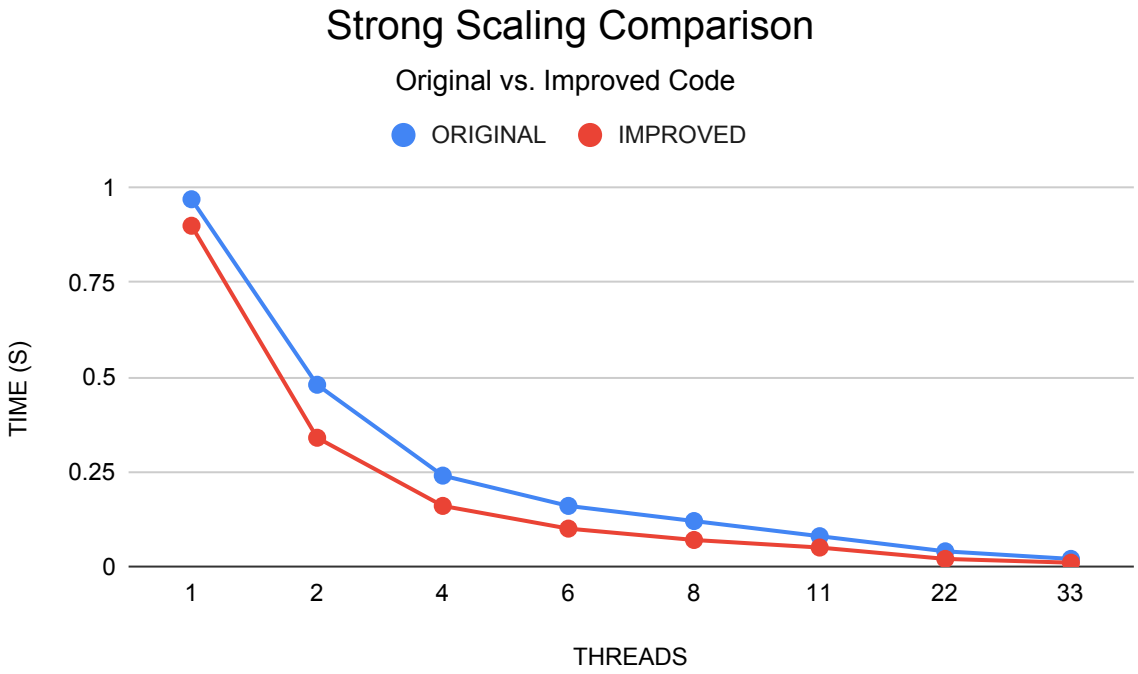

Figure 2: *Comparison of running time, in seconds, of original parallel code and improved parallel code for strong scaling experiments. We can see the improved code performs slightly faster.*

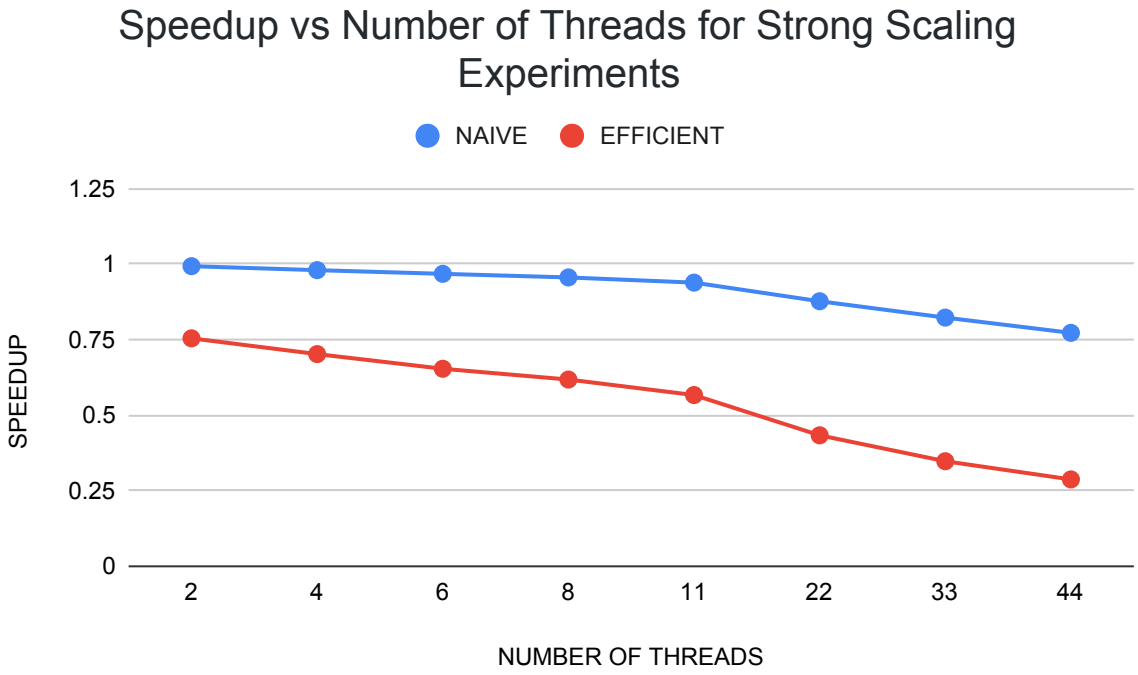

Figure 3: *Comparison of speedup of original parallel code and improved parallel code for strong scaling experiments. We can see that our improved algorithm makes significant gains in speedup.*

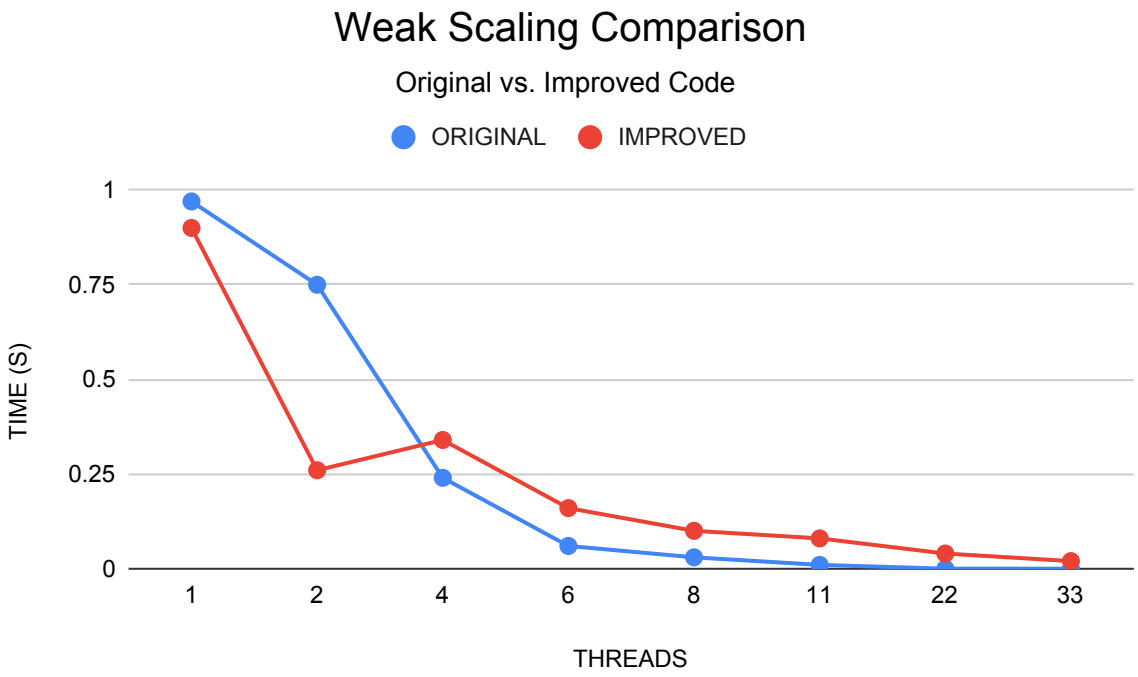

Figure 4: *Comparison of running time, in seconds, of original parallel code and improved parallel code for weak scaling experiments. The amount of work was fixed at* $\frac{n}{p} = 500$ *particles.*

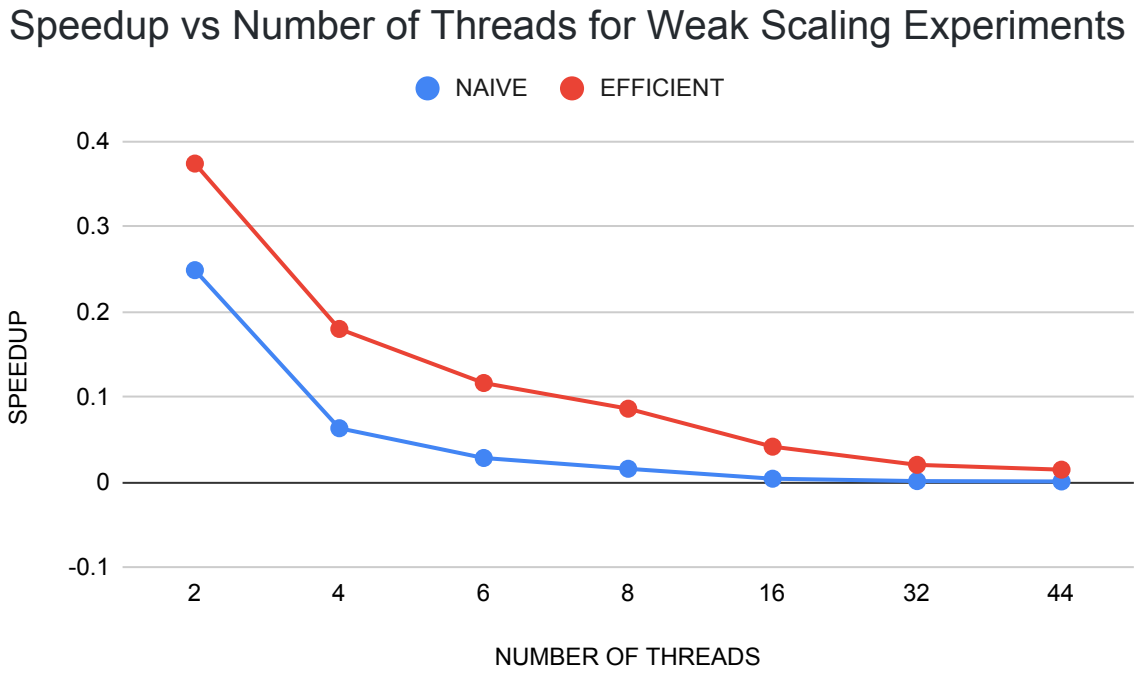

Figure 5: *Comparison of speedup of original parallel code and improved parallel code for weak scaling experiments.*

Strong Scaling Results					
Particles	**Threads**	**Naïve Time (s)**	**Efficient Time(s)**	**Naïve Speedup**	**Efficient Speedup**
500	1	0.598843	0.161574	N / A	N / A
500	2	0.602856	0.214159	0.9933433523	0.7544581362
500	4	0.610955	0.230069	0.9801752993	0.7022849667
500	6	0.618638	0.24715	0.9680022889	0.6537487356
500	8	0.626348	0.261424	0.9560867122	0.6180534304
500	11	0.637795	0.285099	0.9389270847	0.5667294519
500	22	0.68264	0.372758	0.8772456932	0.4334554859
500	33	0.727517	0.46486	0.8231326553	0.3475756142
500	44	0.775063	0.562292	0.7726378372	0.2873489219

Weak Scaling Results					
Particles	**Threads**	**Naïve Time (s)**	**Efficient Time (s)**	**Naïve Speedup**	**Efficient Speedup**
500	1	0.598843	0.161574	N / A	N / A
1000	2	2.40831	0.432209	0.248656942	0.3738330299
2000	4	9.50596	0.899539	0.0629965832	0.1796186713
3000	6	21.3379	1.39144	0.02806475801	0.1161199908
4000	8	38.9223	1.88023	0.01538560157	0.08593310393
8000	16	161.202	3.90746	0.003714860858	0.04135013538
16000	32	648.117	8.07563	0.0009239736035	0.02000760312
22000	44	1224.09	11.27	0.0004892148453	0.01433664596

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