CS267 Assignment 2

Hussain Al Salem Jason Poulos Yang You

March 10, 2016

1 Introduction

In this report, we describe several parallel implementations of a 2D particle simulator and report their performance. The goal is to parallelize code that runs in time $T = O(n)$ on a single processor to run in time T/p when using p processors by taking advantage of shared and distributed memory models. Specifically, we try three implementations: serial code that runs in $O(n)$ time; a MPI distributed memory implementation that runs in $O(n)$ time and $O(n/p)$ scaling; and a OpenMP shared memory implementation. For Part 2 of the assignment, we implement in GPU.

2 Serial Implementation

2.1 Data structures

Our serial implementation uses a binning structure to execute the interaction step in $O(n)$ time. The 2D grid is divided into small squared bins where each side has the size of cutoff. The size of cutoff was used since it is the maximum distance of interaction. Thus, each particle can only interact with particles in its own square and its 8 neighboring squares on a single iteration. Furthermore, bins that are attached to the edges will have less neighbors and thus they were fixed accordingly. The small bins are stored in a data structure type where each bin has a unique consecutive index. Each bin points to the particles indices contained inside the bin. After each iteration, we empty the bins and update them according to the new particle locations. In the interaction step, each particle only need to check for particles within its bin and neighboring bins. Since the number of particles per square is bound (and equal to 1 on average), the interaction step takes $O(n)$ time.

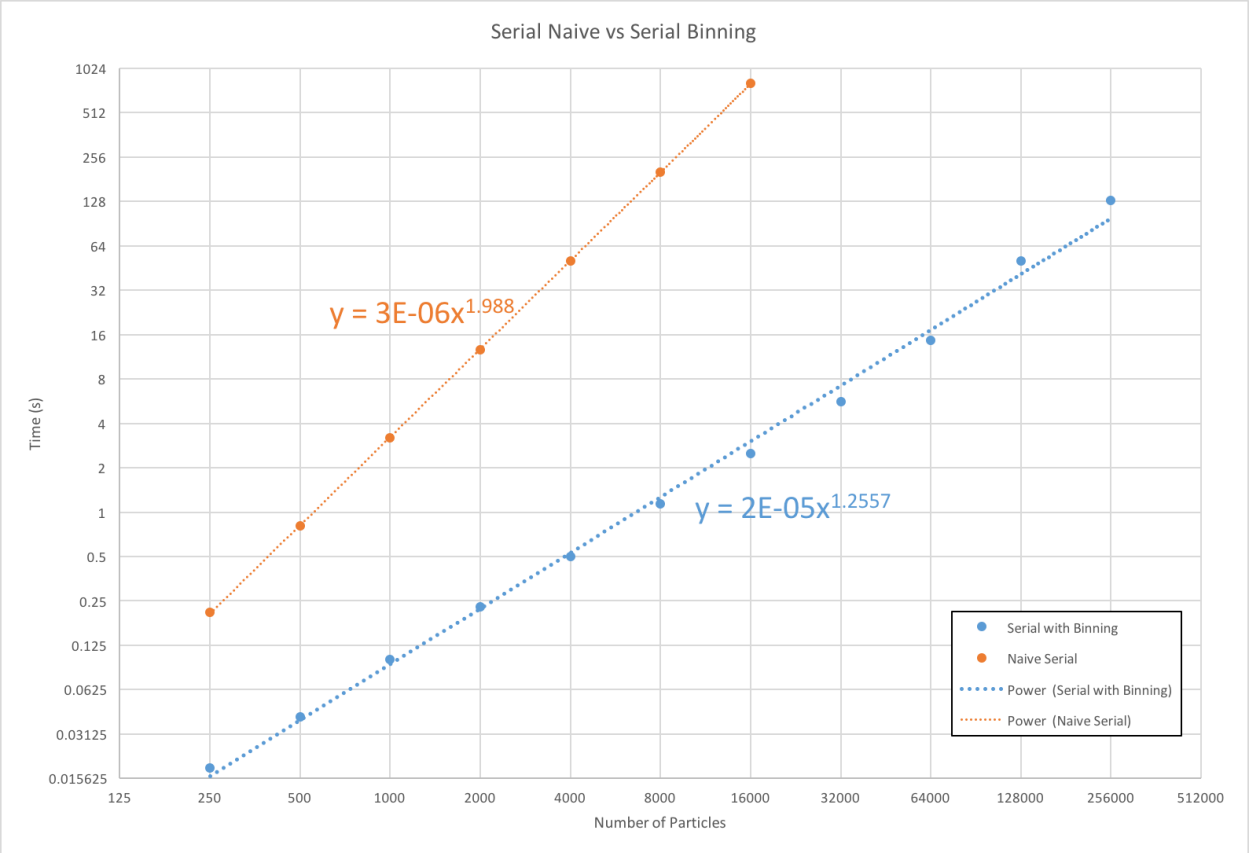

Figure 1: A plot in log-log scale that shows serial without binning runs on $O(n^2)$ time and serial with binning runs in $O(n)$ time.

2.2 Results

Figure 1 shows that our serial binning algorithm follows linear scaling ($O(n)$). For comparison, it shows the performance of the naive $O(n^2)$ algorithm on smaller problems.

3 OpenMP implementation

3.1 Synchronization

We implement OpenMP for shared memory parallelization. Synchronization is used after each parallel region in order to prevent race conditions. First, we assign each particle to bins in parallel. Next, we assign each bin to a lock (using `omp_lock_t`). Bin sizes are about equal to the given cutoff size for particle interaction. Thus, the number of bins is generally higher than the number of particles. When we compute forces in each bin in parallel, there will be little contention in most bins because they are empty or relatively sparse. Lastly, we

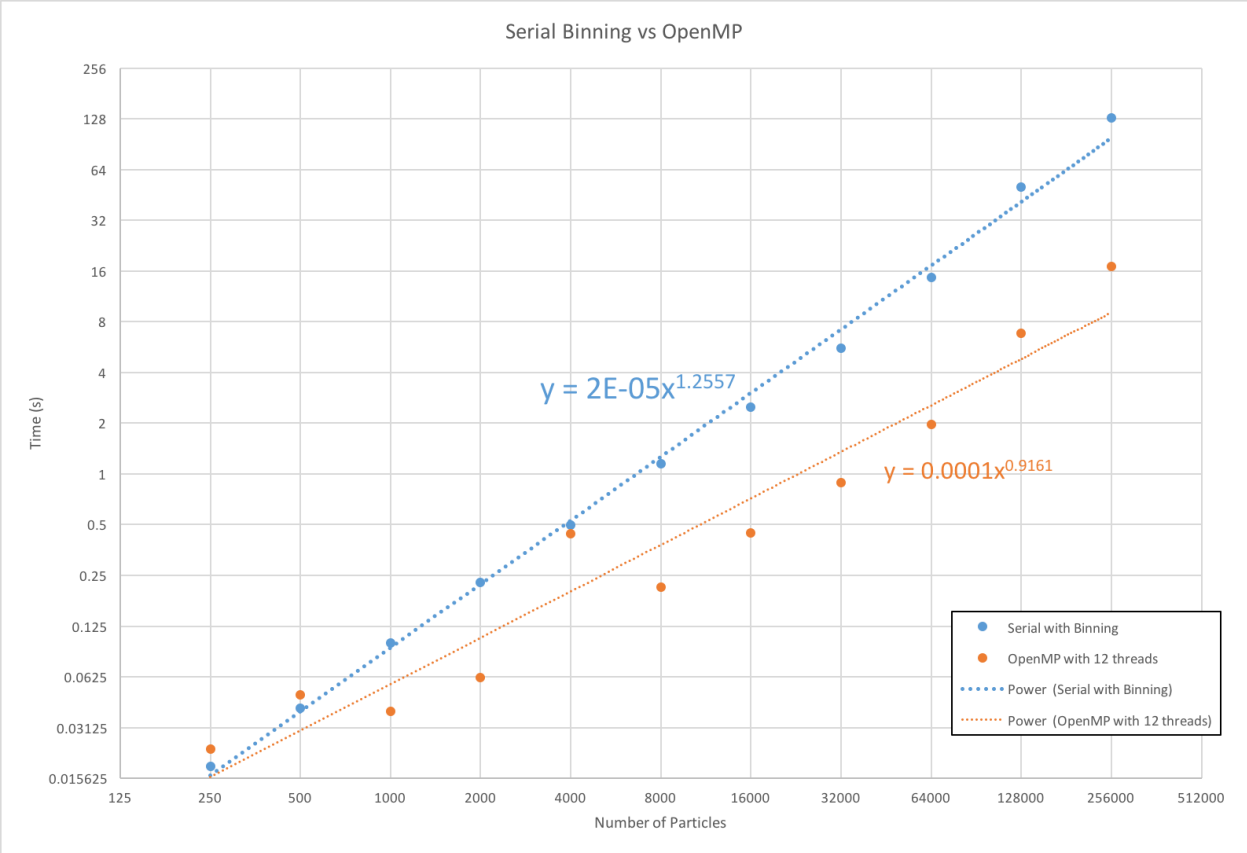

Figure 2: $O(n)$ scaling of the OpenMP implementation with 12 threads compared to the serial implementation.

move each particle and clear each bin in parallel for the next timestep.

3.2 Results

Figure 2 shows considerable speedup of our OpenMP implementation compared to serial binning using 12 threads. We compare based on 12 threads based on the strong scaling measurements in Figure 3. On average, the maximum speed is achieved with 12 threads for number of particles between 500 to 128,000. Four threads is faster below this range, and 24 threads is faster above this range. The simulation time for OpenMP is $O(n^{0.916})$, while the serial is $O(n^{1.256})$.

Figure 3 shows the strong scaling of our OpenMP implementation, or how the simulation time varies with the number of threads for a fixed total problem size. OpenMP scales as the number of particles exceeds 32,000. Figure 4 shows the weak scaling of OpenMP, or how the computation time varies with the number of threads for a fixed problem size per thread. The slope of the linear scaling at 4,000 particles per thread is 0.088, whereas a slope of 0 would

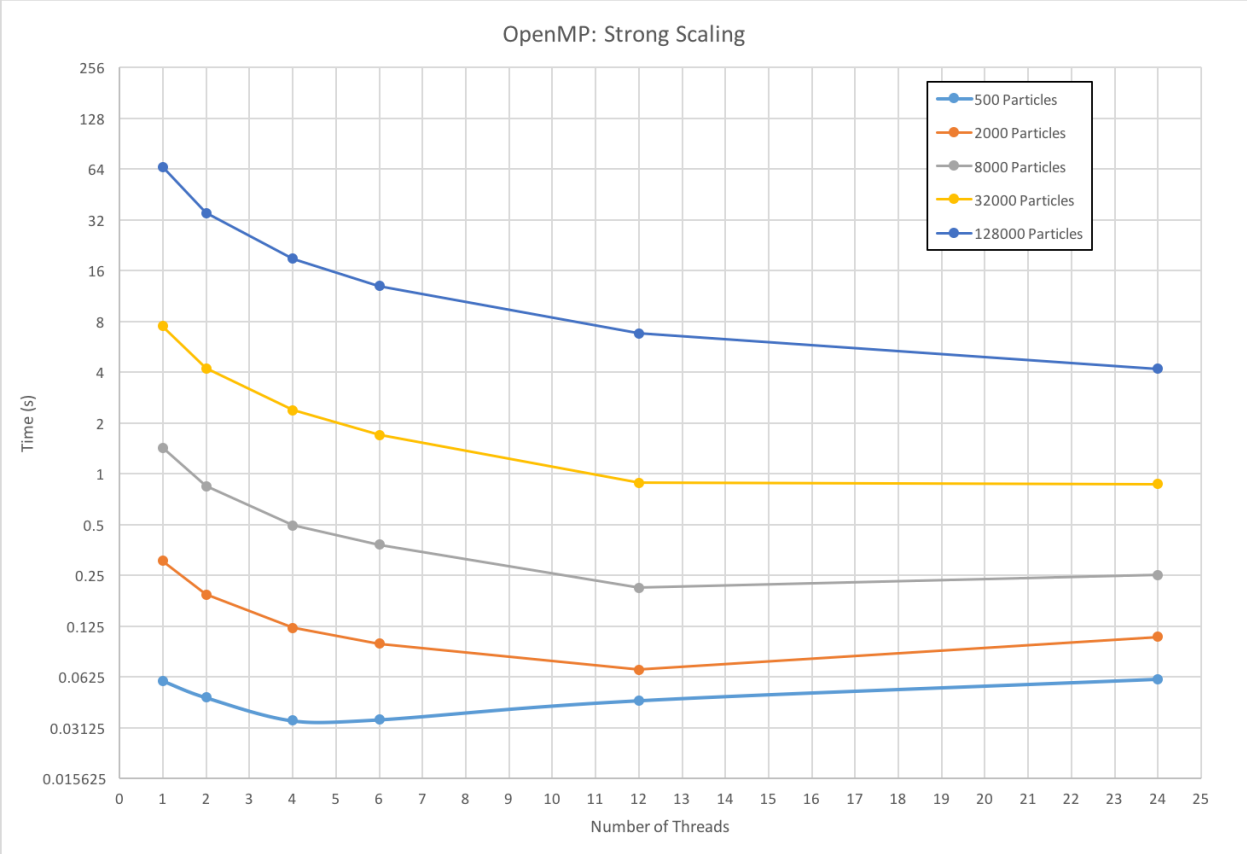

Figure 3: Strong scaling of the OpenMP implementation.

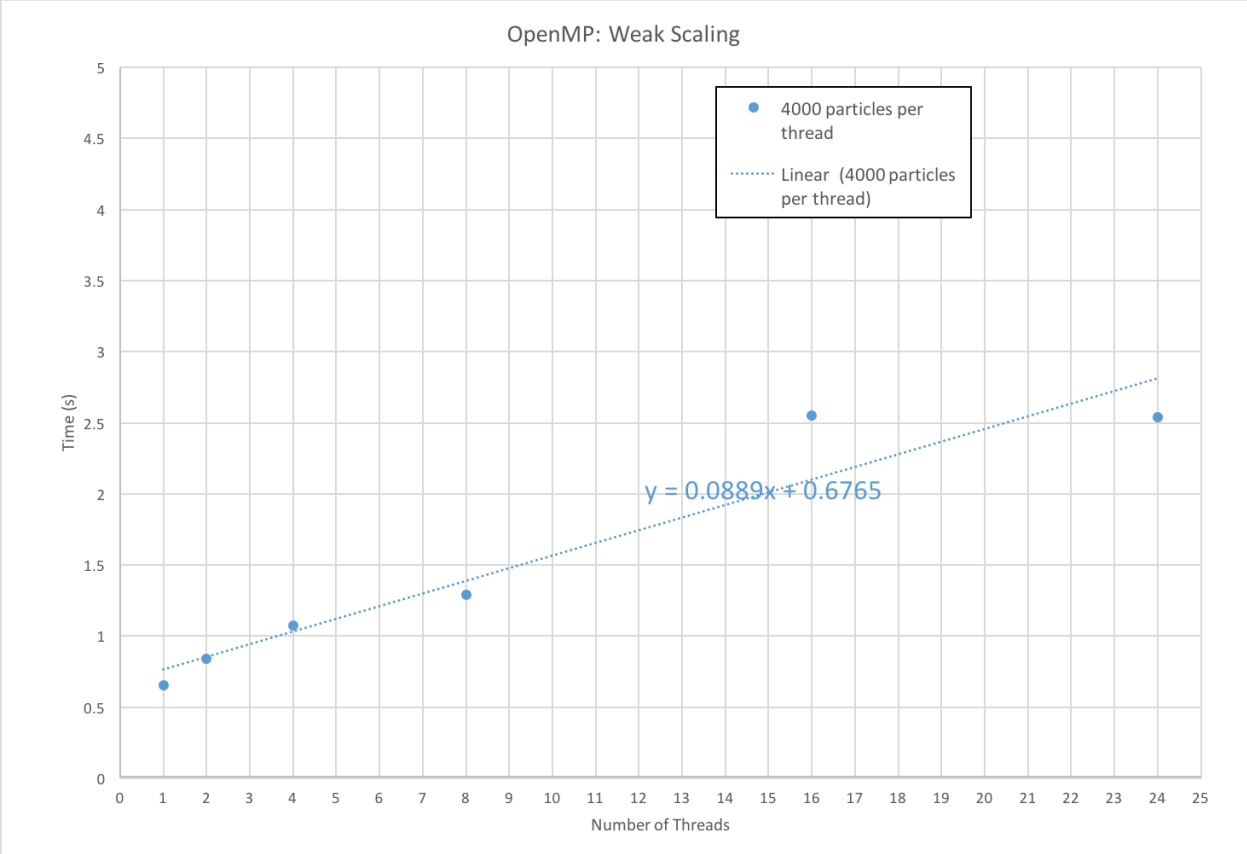

Figure 4: Weak scaling of the OpenMP implementation.

indicate that our code is flawless. There is more communication as the number of threads increases, which explains why the slope is not exactly 0.

4 MPI Implementation

4.1 Communication between nodes

Let us define B as the number of bins, and $b = \sqrt{B}$. We deploy the bins as b $\times$ b distribution, which is just like a grid. Then we divide them by row. Suppose we have p processors, each processor will have $\frac{b^2}{p} = \frac{B}{p}$ bins. We use the root node (rank is zero) to initiate the particles and then use MPI_Scatterv to scatter the particles to all the nodes. Concretely, if a certain particle (pt) belongs to a bin (bi), and bi is owned by a certain node (nd), then pt will only be sent to nd. In each step, we need to make the bins in boundary exchange some particles with the bins in neighbor processor's boundary. We use MPI_Ibsend to conduct this kind of communication. For more details, please check exchange_neighbors. Because we have got the boundary information from the neighboring node, we can conduct force computation on each local node. Each node will get its own $davg$, $navg$, $dmin$ variables. Then we will use MPI_Reduce to get the global variables $rdavg$, $rnavg$, $rdmin$ by MPI_SUM, MPI_SUM, and MPI_MIN respectively.

4.2 Results

The speedup MPI implementation over serial version can be found in Fig. 5. The weak scaling results of MPI implementation can be found in Fig. 6. The strong scaling results of MPI implementation can be found in Fig. 7. The performance comparison among serial version, OpenMP version, and MPI version can be found in Fig. 8. The performance comparison among all the implementations can be found in Fig. 11. The experimental setting is the same with the OpenMP version. On average, the maximum speed is achieved with 24 processes for number of particles between 500 to 128,000. The simulation time for MPI is $O(n^{0.749})$, while OpenMP is $O(n^{0.916})$ and the serial is $O(n^{1.256})$.

4.3 Discussion

From Fig. 8 and Fig. 11 we can observe that MPI achieves better scaling efficiency than OpenMP and GPUs. This means explicit communication behavior does help us to design more scalable software. Note that our implementation is a 1-D partition version, the 2-D partition version may have better scaling efficiency.

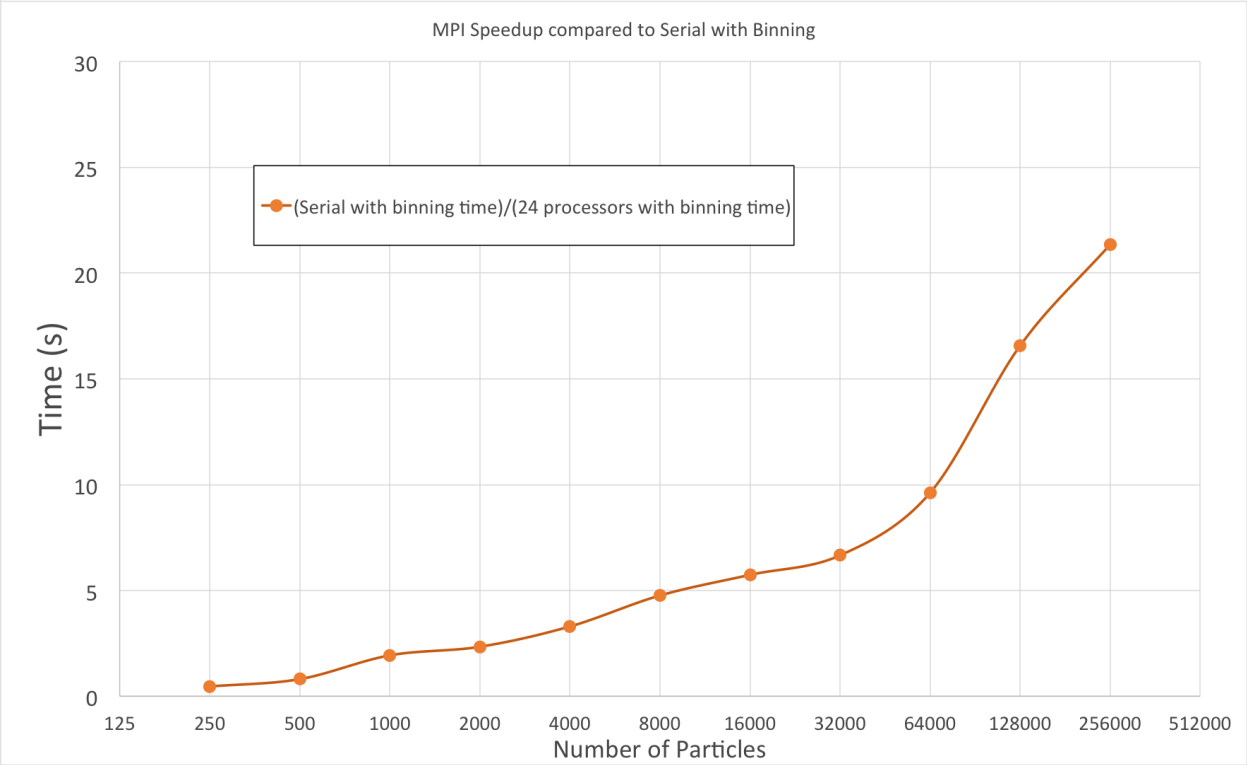

Figure 5: Speedup of MPI implementation.

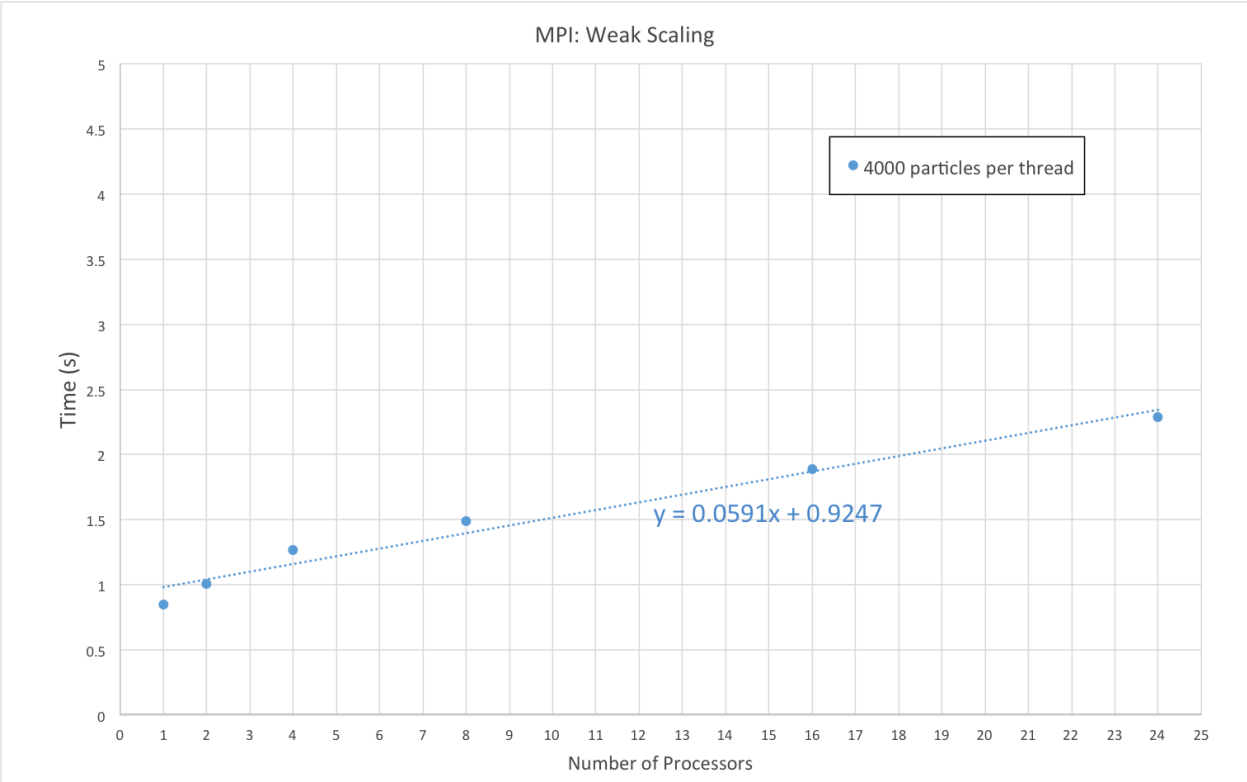

Figure 6: Weak scaling of the MPI implementation.

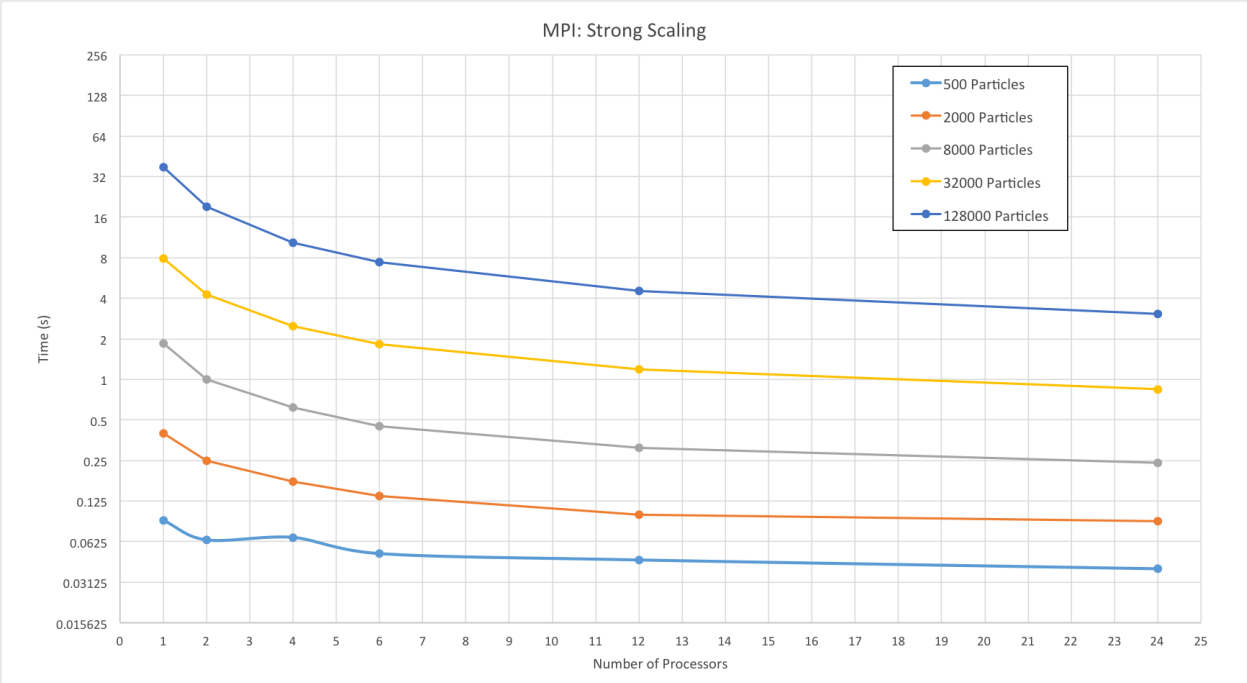

Figure 7: Strong scaling of the MPI implementation.

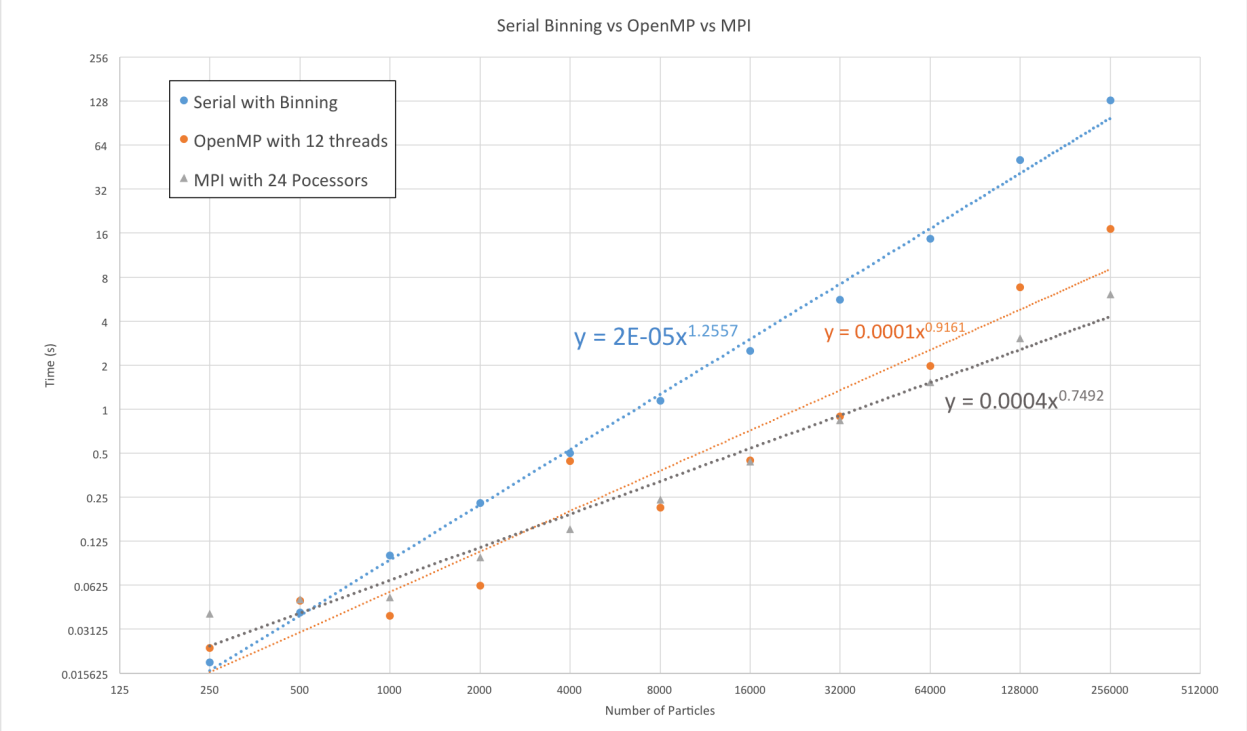

Figure 8: Serial vs OpenMP vs MPI.

The major challenges we faced when writing the MPI code is the communication part, which took us a lot of efforts to manage the sender and receiver in the right way. We use MPI_Ibsend to conduct a nonblocking buffered sending communication, which is critical for our implementation. In order to design the efficient communication model, we need to re-design some data structures. On the other hand, we use the c++ STL's builtin vector to manage the particle, which is very efficient because we can not predict the number of particles each processor own every step so that we need to dynamically update the array at runtime . Also, debugging MPI code is still non-trivial and tedious. The NERSC tools are useful but most of time we use the printf to get the runtime information.

5 GPU Implementation

In order to reduce simulation time from $O(n^2)$ to $O(n)$, the GPU implementation must support binning. In serial binning, resizable vectors were used to store particles in each bin, but resizing is not supported on the GPU. Therefore, a fixed sized memory allocation was used to store the particles. For this implementation, we chose 32 particles per bin. We also maintained two extra 32-element arrays with the `bin_t` structure: `staying` and `leaving`. These are used for rebinning when moving particles.

Initially, each particle is assigned to a bin on the driver. Computing the forces occurs in parallel on the GPU, with one thread per bin. Each thread computes the forces between all particles in its bin and each adjacent bin.

The next step is moving particles. This occurs in two steps both of which occur in parallel with one thread per bin. First, each thread moves each particle in its bin by calling `move_particle_gpu`. It then checks with the particle remains in the bin. If it does, it is added to the bin's `staying` array; otherwise it goes to the `leaving` array. Second, each thread consolidates the remaining particles in it bin and the outgoing particles destined to the current bin from neighboring bins. This is done by looping over all particles in the `leaving` arrays.

This method led to adding an extra global synchronization step due to splitting the move operation into two kernels, but it avoids the need for locking since there are no read-write conflicts on the same data. In the first step, each thread only reads and writes its bin, while in the second step, each thread reads neighbors' particle lists but only writes to its bin's `staying` and `leaving` arrays, leaving the main particle list uncontended.

Before that, we tried avoiding the need to maintain bins by removing the explicit bins entirely and instead computing the bin of each particle on the fly based on its position. This simplified the move step, since it avoided explicit rebinning. However, it required each

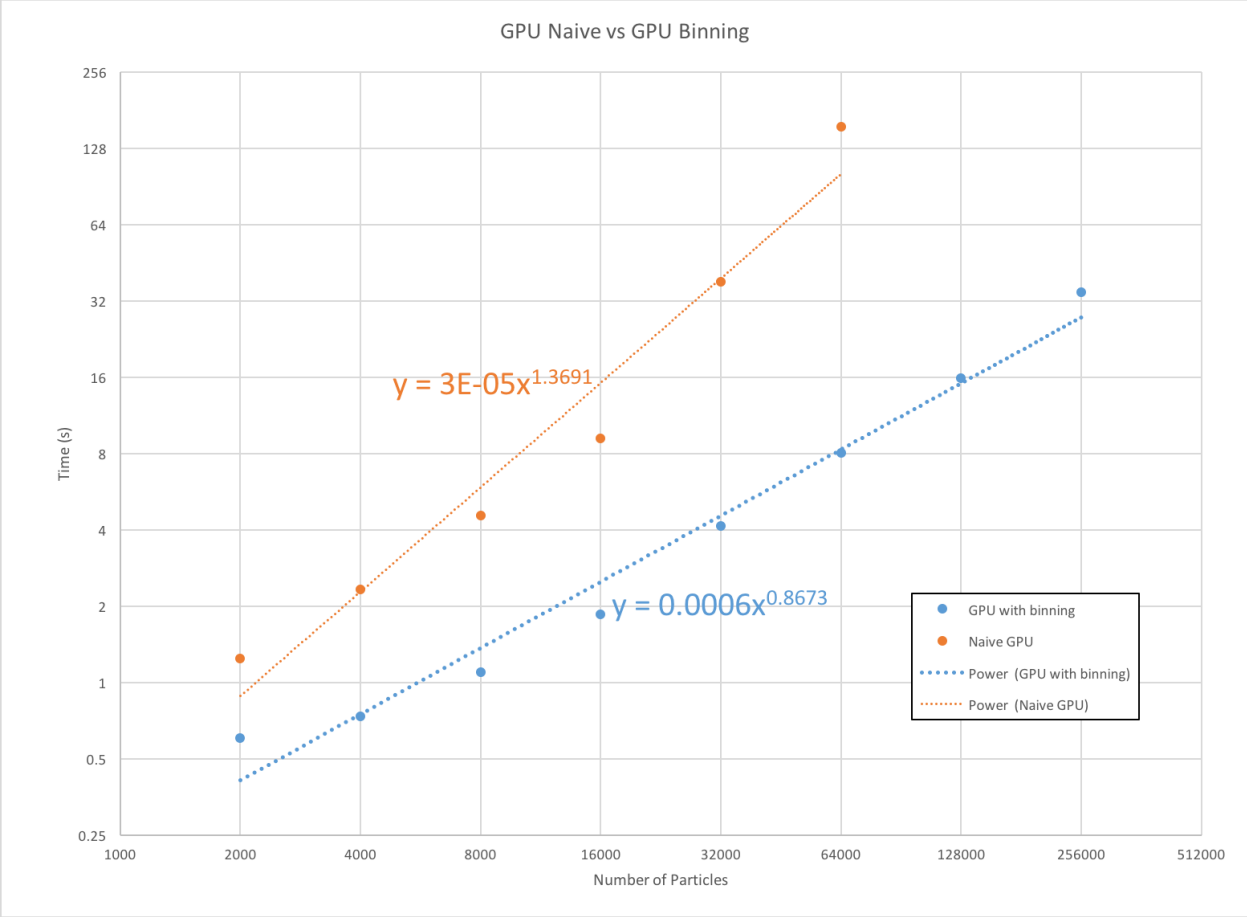

Figure 9: Log-log scale plot that shows GPU with binning versus GPU with binning performance.

thread to examine every particle during the force computation step, which turned out to be orders of magnitude slower than examining only neighboring bins.

5.1 Results

We benchmarked the the GPU Naive Code and our GPU binning implementation on the Stampede GPU architecture. Figure 9 shows that our implementation is $O(n^{0.8673})$ fast compared to the naive GPU implementation which is $O(n^{1.3691})$ fast. Figure 10 shows that the GPU code has slightly better performance than the OpenMP. The GPU code is $O(n^{0.8673})$ fast while the OpenMP code is $O(n^{0.9161})$ fast.

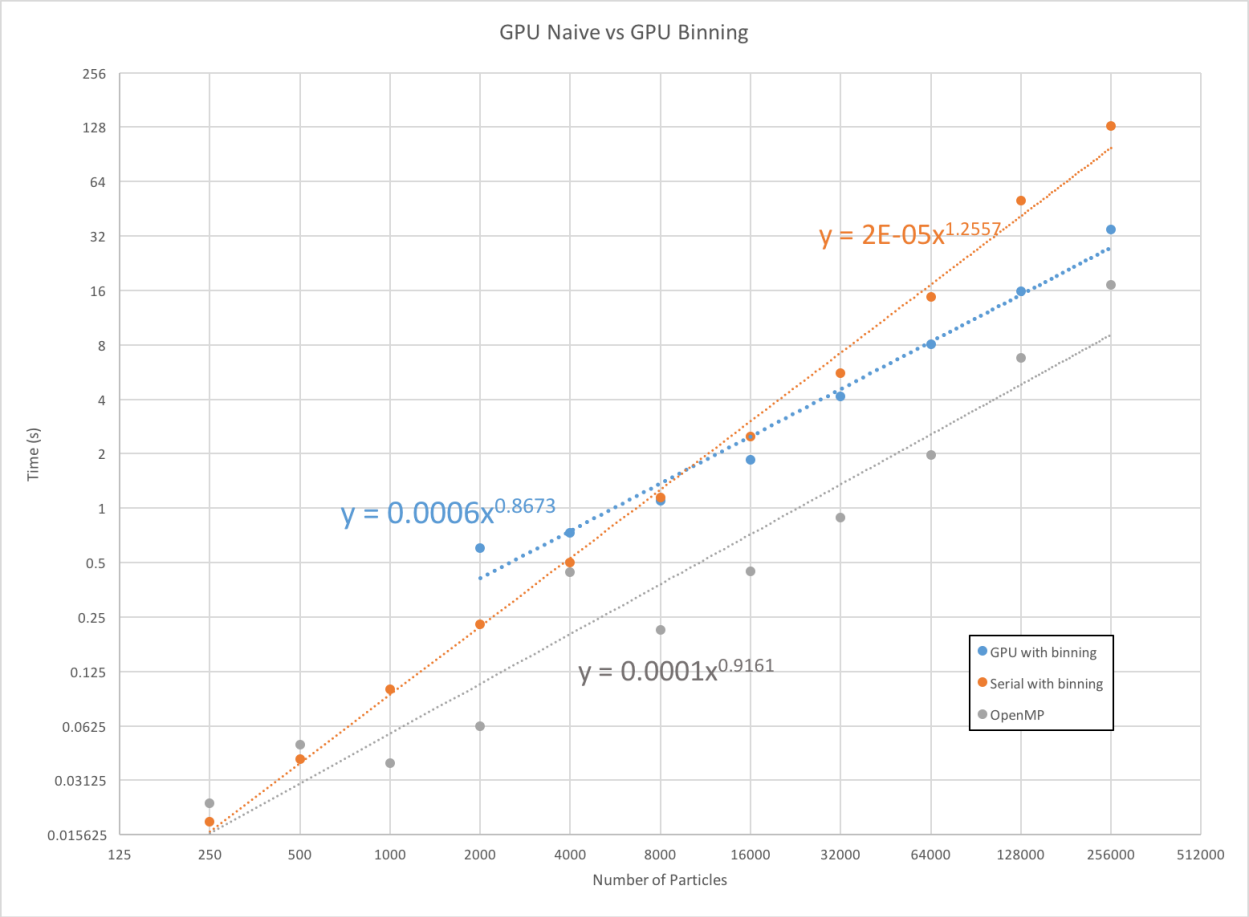

Figure 10: Log-log scale plot that shows Serial with binning, GPU with binning and OpenMP performance.

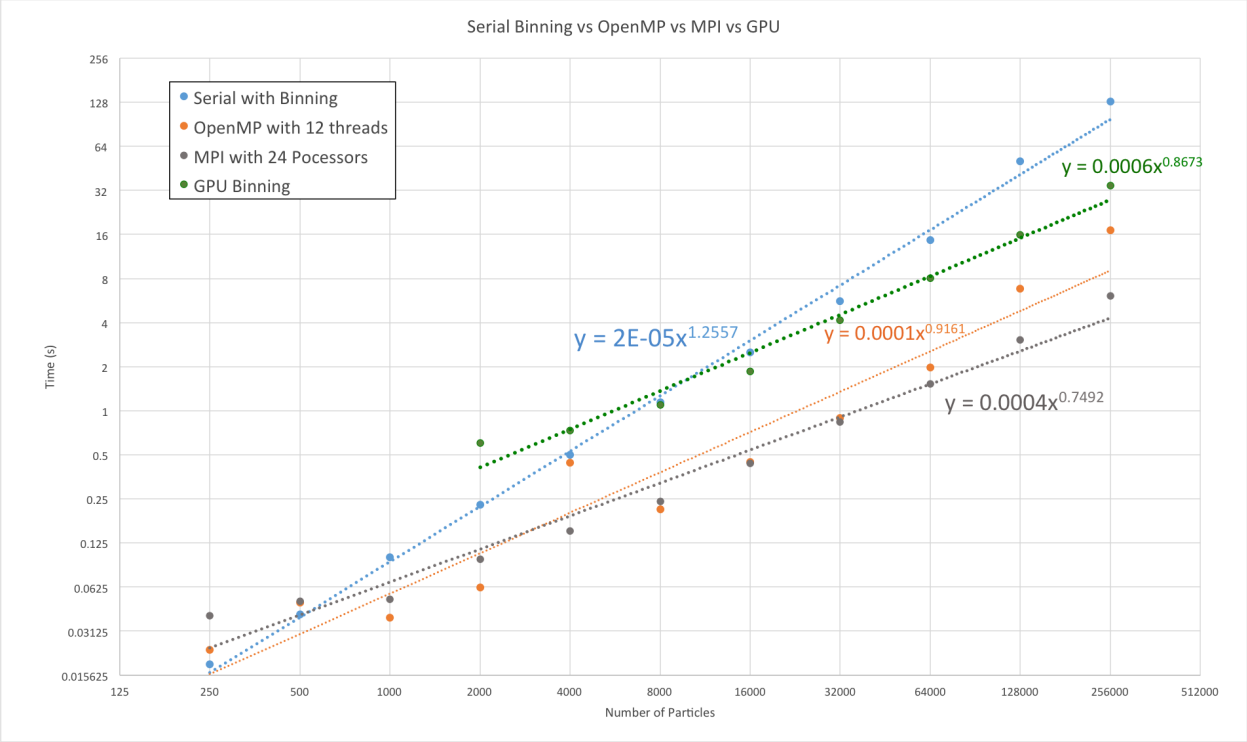

Figure 11: Comparison among all the implementations (Serial vs OpenMP vs MPI vs GPU).

5.2 Discussion

Though CUDA allows writing GPU code in a subset of C, its libraries provide much less functionality than STL. In particular, there is no resizable, concurrent vector data structure, which would have made the particle simulation code much simpler to write, though slower than the lock-free approach we ultimately settled on.

Additionally, the GPU architecture is well-suited for high-intensity applications, but binning substantially increased the memory bandwidth needs of our particle simulator, lowering its intensity and reducing the speedup due to the GPU.

We therefore conclude that, for applications like particle simulation, the performance gains of utilizing the GPU are not worth the substantially increased programming effort compared to OpenMP, which required only a few lines of modifications to the serial code.