CS267 Assignment 2

Hussain Al Salem Jason Poulos Yang You

March 6, 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.

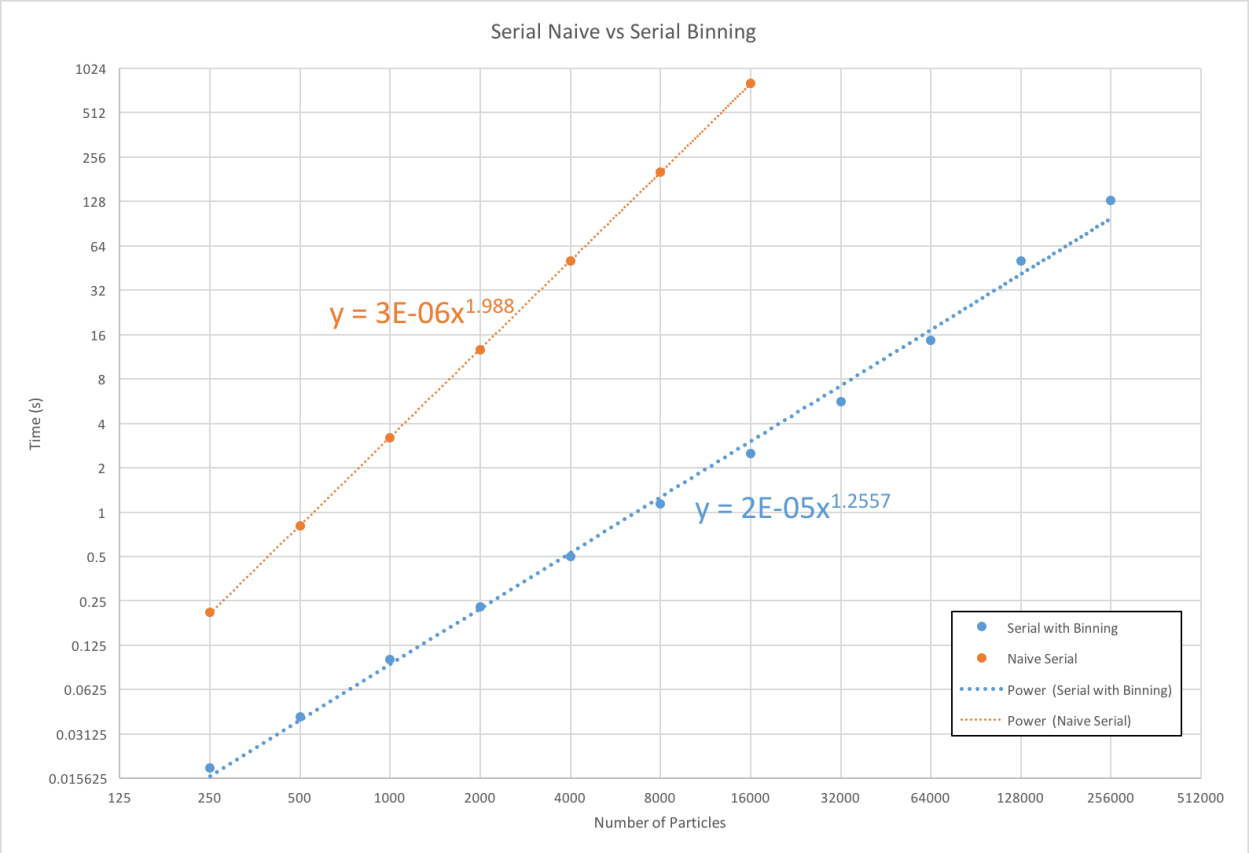

Figure 1: A plot in log-log scale that shows serial without binning runs on $O(n^2)$ time and serial with binning runs in $O(n)$ time.

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 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 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

A description of the communication you used in the distributed memory implementation.

4.2 Results

5 Comparison of distributed and shared implementations

5.1 Where does the time go?

Consider breaking down the runtime into computation time, synchronization time and/or communication time. How do they scale with p?

5.2 Discussion

A description of the design choices that you tried and how did they affect the performance.

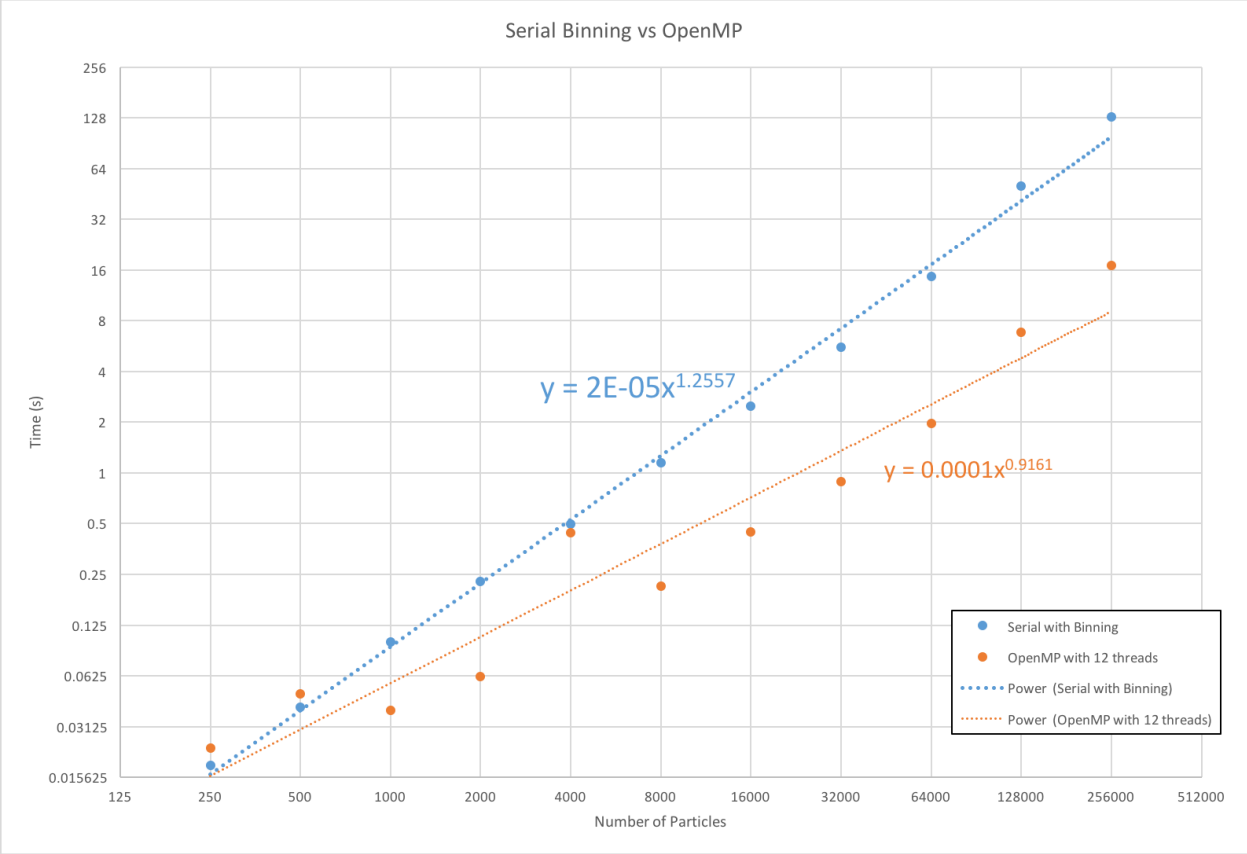

Figure 2: $O(n)$ scaling of the OpenMP implementation with 12 threads compared to the serial implementation.

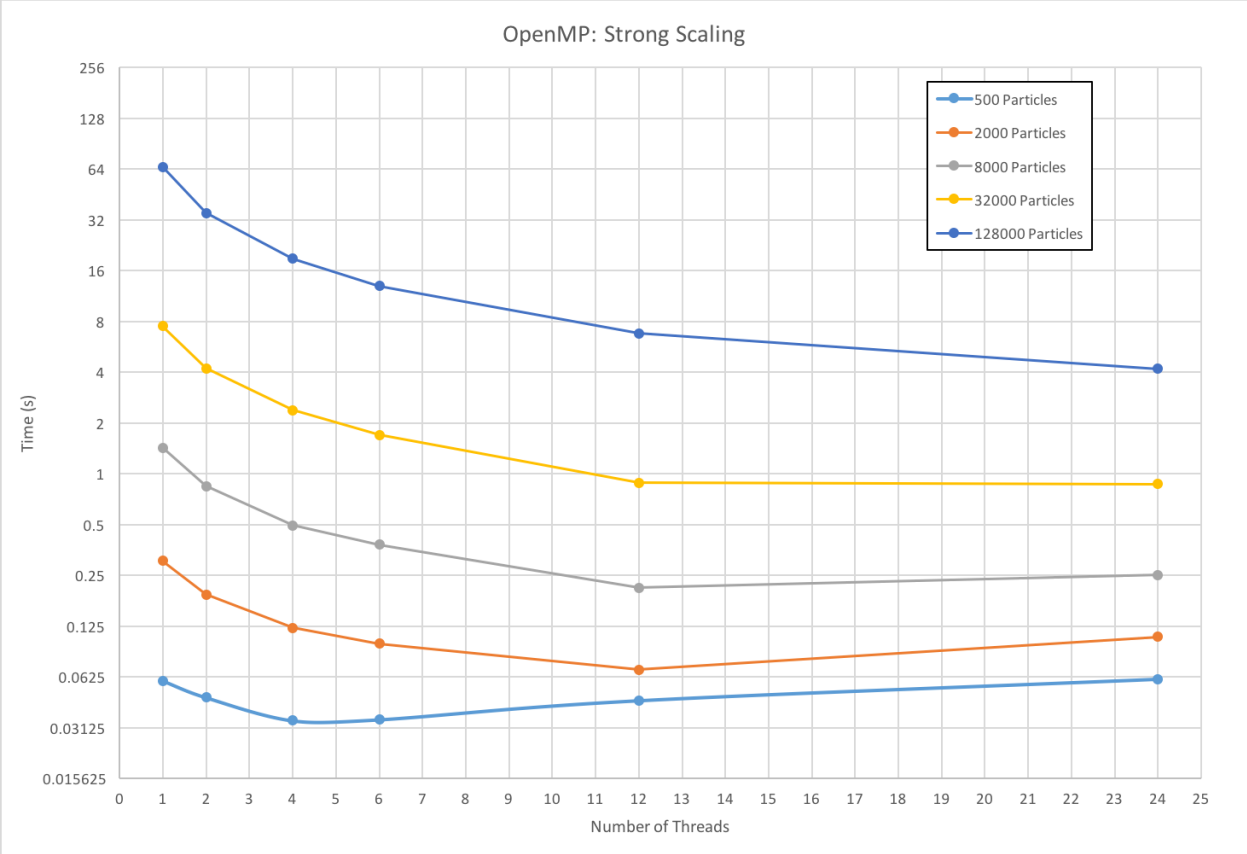

Figure 3: Strong scaling of the OpenMP implementation.

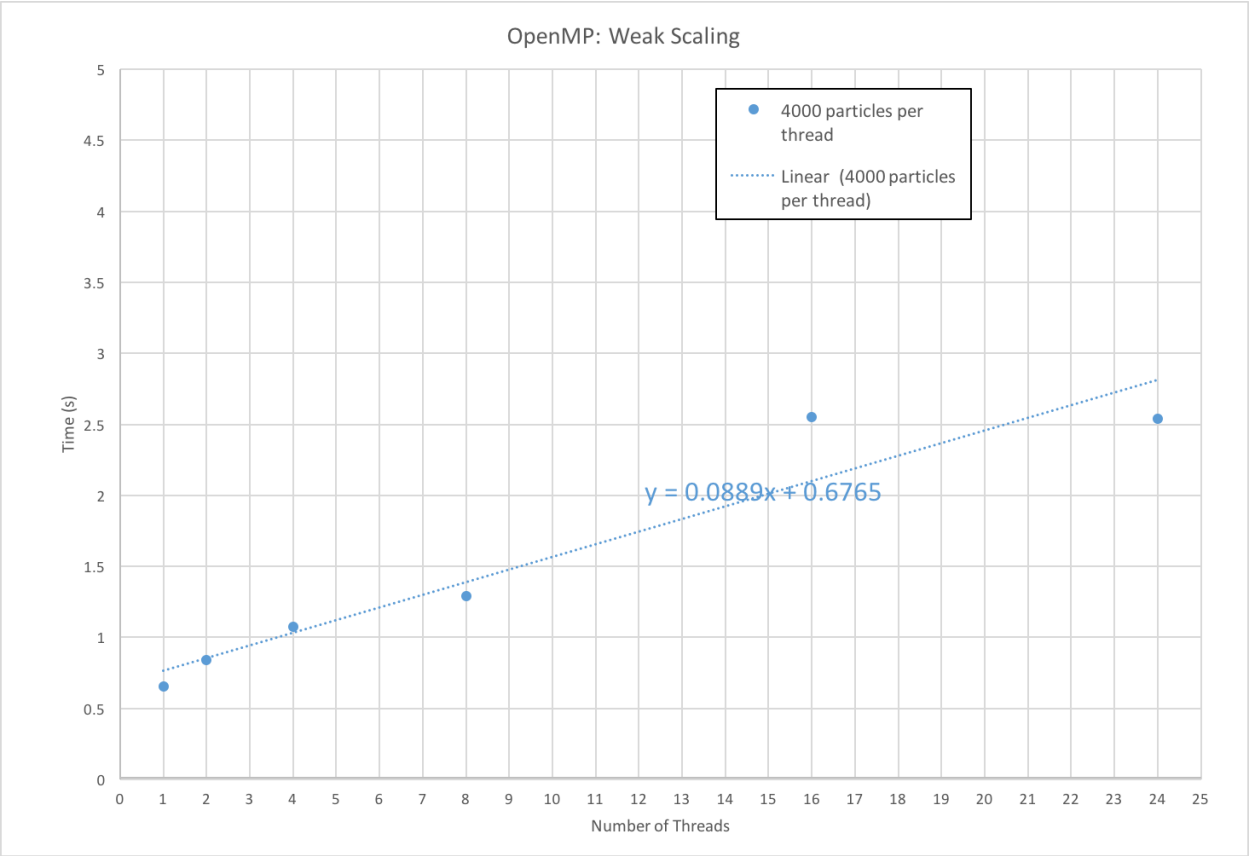

Figure 4: Weak scaling of the OpenMP implementation.

Figure 5: $O(n)$ scaling of the MPI implementation.

Figure 6: Weak scaling of the MPI implementation.

Figure 7: Strong scaling of the MPI implementation.

Figure 8: Speedup plots that show how closely parallel codes approach the idealized p-times speedup.

Figure 9: A plot of the speedup of the GPU code versus the serial, openmp, mpi runs on the CPU of the node

Figure 10: A plot in log-log scale that shows the performance of your code versus the naive GPU code.

A discussion on whether it is possible to do better

A discussion on using pthreads, OpenMP and MPI.

6 GPU Implementation

A description of any synchronization needed A description of any GPU-specific optimizations you tried

6.1 Results

6.2 Discussion

A discussion on the strengths and weaknesses of CUDA and the current GPU architecture