CS 267: HOMEWORK #2
PARALLELIZE PARTICLE SIMULATION

Professor James Demmel
March 12, 2016

Sayna Ebrahimi
Fotis Iliopoulos
Pasin Manurangsi
University of California, Berkeley

Abstract

In this project we investigated the parallelization of particle simulation on distributed and shared memory CPU systems using MPI and OpenMP, respectively. We also implemented the simulation on GPU using CUDA. We have built our parallel algorithms based on our modified version of the sequential program for the same problem. Results for weak and strong scaling efficiency are compared against each other and scaling performances for each model are evaluated and discussed. We show that these techniques can effectively speed up the simulation. CUDA implementation is shown to be the most powerful tool for massively parallel problems.

1 Introduction

In many disciplines such as physics, astronomy and mechanics, researchers often study *dynamical systems* of particles. Each particle in such a system has a state at each time step and the system contains a set of equations that compute the state of each particle for the next time step. A natural question is to ask what the particles' states are after t time steps. It is known that such question is computationally hard [RT93]; this means that it is unlikely that there exists an efficient algorithm[1] that solves the problem. Thus, the widely used solution is to simply simulate the particles in each time step. This brings us to the central goal of this project: to simulate dynamical systems of particles as efficiently as possible.

In our dynamical system of interest, each particle's state consists of its current position and velocity, both in two dimensional space. For each time step, each particle's position is changed according to its velocity whereas the velocity is changed corresponding to forces exerted by other particles within a specified cutoff distance from the particle.

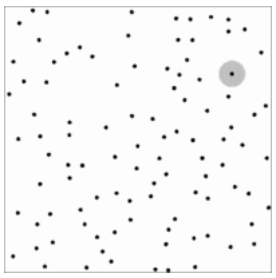

Figure 1: A snapshot of particles in a simulation. The shaded area corresponds to the interaction region; only particles inside the area interacts with the particle at the center of the region.

The trivial algorithm, in which one computes interactions between every pair particle in each time step, takes $O(n^2)$ time per time step where n is the number of particles. This, however, is not the best one can do with this particular system. More specifically, the nature of the system allows us to check for interactions only between each particle and particles that are close (within the cutoff distance) to it, That is, if we partition space into squares (or bins) each with height equal to the

[1] More specifically, if PSPACE != P, then there is no algorithm with running time that is polynomial in $\log t$.

cutoff distance, then it is enough for us to compute forces between each particle and the particles in its bin and the surrounding bins. When the density of the particle is constant, the expected number of particles in each grid is also constant, meaning that this "binning" algorithm takes only $O(n)$ time.

In addition to improvement in asymptotic running time from $O(n^2)$ to $O(n)$, we can use parallelization to speed up the simulation even further. We explore several models of parallelization in this project, including shared memory model and distributed memory model using OpenMP and MPI respectively. We also experiment with parallelization via a graphics processing unit (GPU) via NVIDIA CUDA.

Our report is arranged in the following fashion. In the next section, we explain our algorithms for each model along with the optimization techniques we tried and how effective they are. Then, in Section 3, we discuss the performances of our codes and whether they match our theoretical expectations. Finally, in Section 4, we suggest some ideas that may further improve performances of our codes.

2 Algorithms and Implementations

In this section, we describe our algorithms for the different models of parallel programming and how we implement them. This section is organized as follows. For each model, we have a subsection corresponding to it; we start by explaining the overall idea of the algorithm for the model. We then expand on implementation details and optimization techniques we tried and how well they worked.

2.1 Serial implementation

First, let us recall the given naive algorithm. As shown in Algorithm 1, at each time step, the force interactions between each pair of particles are calculated and stored as acceleration of each particle. Every particle is then moved and its velocity is subsequently updated accordingly to its acceleration.

Algorithm 1 Naive Serial Algorithm

```
1: for each time step t = 1 to T do
2:     for each particle i = 1 to n do
3:         for each particle j = 1 to n do
4:             apply force from particle j to particle i
5:     for each particle i = 1 to n do
6:         move particle i
```

By using the binning method (aka spatial partitioning), we can split our domain into small bins (or partitions). Each bin is simply a square of size being the cutoff distance. Each particle then merely interacts with its own and the surrounding bins. Therefore, to calculate the forces applied on each particle, we no longer need to iterate through all other particles and this reduces computational

time drastically. In fact, if the density of the particles[2] is constant, then the number of particles in each bin is constant in expectation. Hence, the running time is linear in the number of particles. The pseudo-code for the algorithm is shown in Algorithm 2.

Algorithm 2 $O(n)$ Serial Algorithm (Spatial Partitioning)

```
1: initialize bins
2: for each particle i = 1 to n do
3:     assign particle i to a bin according to its position
4: for each time step t = 1 to T do
5:     for each particle i = 1 to n do
6:         for each bin B nearby i do
7:             for each particle j in bin B do
8:                 apply force from particle j to particle i
9:     for each particle i = 1 to n do
10:        move particle i
11:        update particle i's bin according to its new position
```

Implementation details. We use vector in C++ Standard Library to implement our data structure for the algorithm. More specifically, we declare one vector for each bin; the vector contains the indices of all the particles in the bin. When we update a particle's bin, we just remove its index from the previous bin's vector and push it into the current bin's vector.

Below we list two optimization techniques we tried and why we did not keep them in the final code.

Iterate through bins instead of through particles. For the two loops that we iterate through particles (line 6 and 10 in the pseudo-code), we can instead iterate through bins first and, in that loop, iterate through particles in the bin. It may not be clear from the pseudo-code what the benefit of such alternate looping is since updating a particle's bin is not expanded out. In the actual implementation, we need to go through the whole vector corresponding to the old bin to find the particle and remove it from the vector. On the other hand, if we use the alternate looping, with can directly delete it from the vector. It seems like the latter should be better. However, based on our experiment, looping by particles outperforms looping by bins. As a result, we keep the former in our final code. A reasonable explanation is that the more layer of the loops may add too much of an overhead when we loop by bins, which prevents us from seeing any improvement.

Using smaller number of bins. While making each bin has side length the same as the cutoff distance seems like a good decision, the number of bins is $\frac{\text{size}^2}{\text{cutoff}^2}$, which is $5n$ in our case. While this

[2]The density of the particles in the number of particles per unit square of the plane.

is not too big, it seems that the memory usage overhead may worsen the performance of the code. We try changing decreasing the number of bins to smaller numbers. As we did so, we saw decline in performance so we kept the bin size to be the cutoff distance in the final code.

2.2 Shared memory implementation : OpenMP

OpenMP and Pthreads are two parallel progamming models for shared memory parallelization. For the purpose of this assignment, we implemented OpenMP only. The following pseudo-code shows how we parallelize our $O(n)$ serial code using OpenMP:

Algorithm 3 Shared Memory (OpenMP) Algorithm

```
1: initialize bins
2: initialize locks                                   ▷ We have one lock per bin
3: for each particle i = 1 to n do
4:     assign particle i to a bin according to its position
5: for each time step t = 1 to T do
6:     for each particle i = 1 to n do                ▷ This loop is executed in parallel
7:         for each bin B nearby i do
8:             for each particle j in bin B do
9:                 apply force from particle j to particle i
10:    for each particle i = 1 to n do                ▷ This loop is executed in parallel
11:        move particle i
12:        if particle i's move to a new bin then
13:            lock the old bin
14:            remove particle i's from the old bin
15:            unlock the old bin
16:            lock the new bin
17:            add particle i to the new bin
18:            unlock the new bin
```

Implementation details. Our shared-memory code is almost exactly the same as that for serial. The only difference is how we handle synchronization, which is described below.

Synchronization. *Race condition* is a common issue in shared memory parallelization when multiple threads are updating the same shared variable simultaneously. In our case, race condition may arise when two particles are removed from or added to the same bin when we are updating particles' bins. To prevent such race conditions, we have a lock (`omp_lock_t`) for each bin. Whenever the bin is modified, its corresponding lock is held by the thread that makes changes to the bin. While this may seem like excessive lockings on the surface, in practice, two threads rarely wants to modify the same bin due to the sparsity of the particles; in fact, as will be discussed below, this

yields a better performance than other approaches that use less locks.

Reducing synchronization used. We have tried several ways to reduce the number of lockings required for our code; we list a couple of interesting ones below. None of them give better performance that the simple version that is used in our final code.

- We can split the large loops into two smaller loops: the first loop handle moving particles and removing it from the old bin, and the second loop adds the particles into the new bins. If the first loop is iterating over bins, then we can remove particles without using locks. Hence, we only need locks for adding particles and, intuitively, we should get some improvement. However, we observe no improvement and conclude that the overhead of having two separate loops and having to iterate over bins instead of particles (see discussions in the serial section) weights in more than the improvement from less lockings.

- Follow up from the previous item, if the large loop is split into two smaller ones as stated above, then we can further reduce the number of locks required in adding particles to their new bins. Since most particles do not move too far from there previous bin, we can iterate through the bins, and, for each bin B, looks at the bins "close" to B to see whether there are any particles that should be moved into B. This steps does not need any locks. Of course, there can be particles that move far away so we need to have another loop that takes care of that. We experiment with various distance that for "closeness" but never get any improvement over the original code.

2.3 Distributed memory implementation : MPI

MPI is a parallel programming model for distributed memory parallelization. We also use binning for our MPI code. The central idea in our MPI implementation is that each processor is assigned with some bins that it is responsible for. At each time step, each processor computes the acceleration, velocity and displacement of the particles that belong to its bins and then sends the particles that leave its partitions to processors responsible for the destination partitions. Moreover, it sends to each processor the particles that are in bins adjacent to the bins responsible for this processor. These are needed to compute forces between particles correctly. Finally, it receives the particles that moved to its bins (or adjacent bins). The following pseudo-code shows the overall idea in our MPI algorithm. Since the actual code is fairly complicated, we leave out most of the details in the pseudo-code but we will discuss them below.

Algorithm 4 MPI ALGORITHM

```
1: initialize bins
2: compute which processor is correspond to each bin
3: for all time steps t = 1 to T do
4:     compute which processor(s) each particle this processor was responsible for in the previous step should be send to.
5:     send this processor's particles from previous step to other processors as necessary.
6:     receive particles from other processors.
7:     update this processor's bins.
8:     for each bin B that this processor is responsible do
9:         for each particle i = 1 in B do
10:            for each bin B' nearby B do
11:                for each particle j in bin B' do
12:                    apply force from particle j to particle i
13:    for each bin B that this processor is responsible do
14:        for each particle i = 1 in B do
15:            move particle i
```

Implementation details. We again have one vector for each bin. Each processor also records which bins it is corresponding to. As for communication, we use `Isend` when we send particles to other processors. For receiving, we use `Iprobe` to check whether there is any message ready to be received from any processor; if so, we receive the message using `Recv`. We use different tags for different time steps to ensure that we are not receiving particles for the future time steps. (More discussions about our choice of communication protocols can be found below.)

We have tried several optimization techniques, which we listed below.

Non-blocking vs blocking communication At first, we used the (sometimes) blocking `Send`. Our code was not only very slow but also prone to infinite waiting when the ordering of sends and receives are incorrect. We fix this by using `Isend` instead, which complicates our code a bit since the buffer cannot be reused immediately. Still, we experience a huge improvement by using `Isend`. Moreover, we first use `Recv` without `Iprobe` so we sometimes need to wait for messages from a processor even though there are messages from other processors ready for us to receive. `Iprobe` helps us solve this problem. However, we only experience little improvement (less than 5%) in the performance here.

Space partitioning. We tried to assign partitions to processors in a way that decreases the communication complexity of our algorithm. In particular, we want to minimize the number of particles that need to be send to multiple processors (because they are adjacent to partitions not

belong to the processor it is assigned to). This is the same as trying to minimize perimeter of the partitioning of the space. We ended up using vertical stripes as our partition; each processor is responsible to a stripe. In our experiment, this works much better than a random partitioning. We also tried grid partition when the number of processors is four by dividing the plane into 2×2 grid but we did not see any significant difference compared to the vertical stripes partitioning so we stick with the latter.

2.4 GPU: CUDA

CUDA is a programming interface used to write parallel code in heterogeneous architectures. The idea is that the user is in control of the CPU (host) and the GPU (device). Most of the computational effort is done in the GPU since it is more amenable to parallelization, as long as not too much communication effort comes with it. CUDA is drastically powerful for massive parallelization speedups. However, it may not be suitable for medium scale paralleization because it needs memory management for devices. Since the different threads in the GPU share memory, our algorithm for this case is basically the same as in the shared-memory case up to implementation details of data structures for our data. The pseudo-code for our CUDA implementation is shown below.

Algorithm 5 CUDA Algorithm

```
initialize bins
copy the particles from CPU to the GPU
for each time step t = 1 to T do
    assign particles to bins        ▷ Executed in GPU; inserting to a bin is atomically performed
    compute forces withing bins     ▷ This is executed in GPU
    moving particles                ▷ This is executed in GPU
copy the particles from GPU back to the CPU
```

Implementation details. Since STL vector does not work on GPU, we simply allocates memory for all the bins before the simulation begins. We allocate a memory for six particles for each bin. We then simply use this array in place of the vectors. The other main difference between our CUDA code and others is that we always recompute the bins every time step whereas, in other models, we only move each particle to its new bin if its bin changes. This involves both removing and adding particles to bins, which requires locks. On the other hand, recomputing bins does not involve removing particles from bins, which helps us avoid locking the bins; here we can just use `atomicAdd` to calculate the end index in the bin that the particle is moved to.

We note that the reason for the number six used above is that, in the given code, the correctness is checked by checking whether the square of the minimum distance between two particles are at least 0.4 times the square of the cutoff distance. From this, we can prove that, if an implementation passes such test, then, at any time step, each bin contains at most six particles as follows. We first divide each bin into six rectangles with height cutoff/2 and width cutoff/3. Each rectangle

cannot contain more than two particles because otherwise the square of the distance between the two particles is at most $(1/2)^2 + (1/3)^2 < 0.4$ times the square of the cutoff. As a result, there can be at most six particles in each bin, unless the checker given is incorrect.

We have tried the following optimization but decided to leave it from our final code.

Iterating by bin instead of particles. Similar to other implementations, we have a choice of looping over particles or looping over bins when we compute forces and move particles. We tried iterating over bins but its performance is slightly worse than iterating over particles so we use the latter in our final implementation.

3 Results

3.1 Speed up

Figure 2 shows a comparison between running time of the above algorithms for different number of particles (500-8000). For the naive code, slope of the data is 1.992 whereas for the optimized code it is 1.12 which matches our expectations.

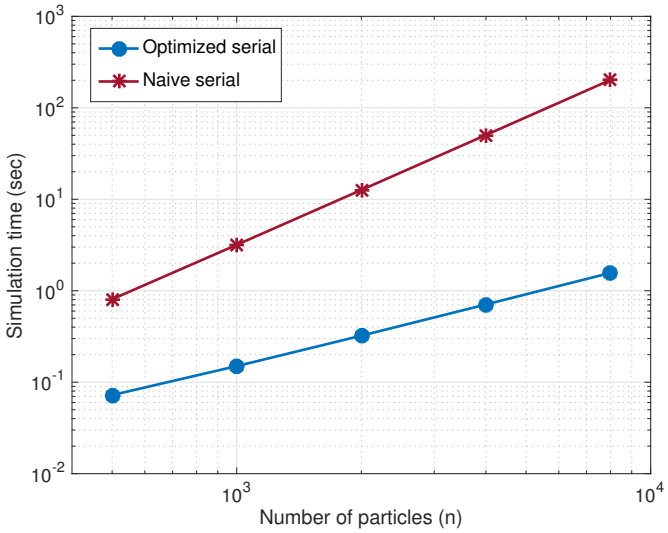

Figure 2: log-log plot of simulation time for different number of particles for the naive serial code and the modified one with binning method.

Figure 3 and 4 show the simulation time for different number of particles for fixed number of threads of 4 and 24, respectively. MPI jobs were performed on 3 machines and it is clear that an OpenMP implementation with 4 number of threads performs faster because it uses shared memory compared to MPI with overheads due to message-passing between 4 processors. However, MPI relatively scales better with 24 threads. The performance of the GPU code is also shown in this

figure for the sake of comparison despite the fact that they were executed on different machines. Clearly CUDA speeds up the performance by a large scale using many thousands of threads.

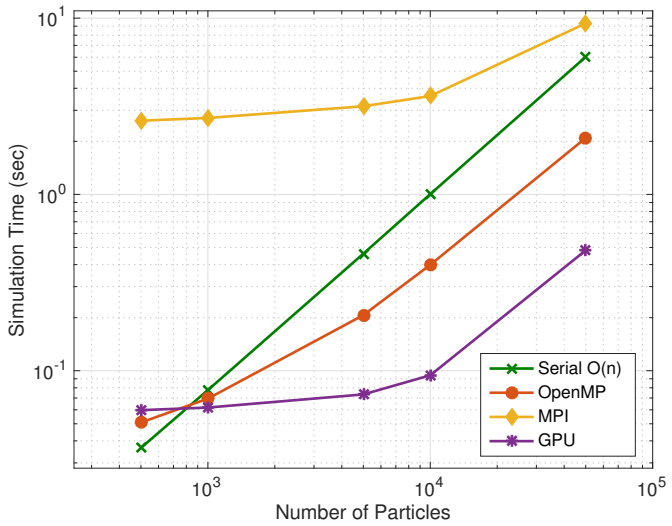

Figure 3: log-log plot of simulation time versus different number of particles for a fixed number of threads of 4

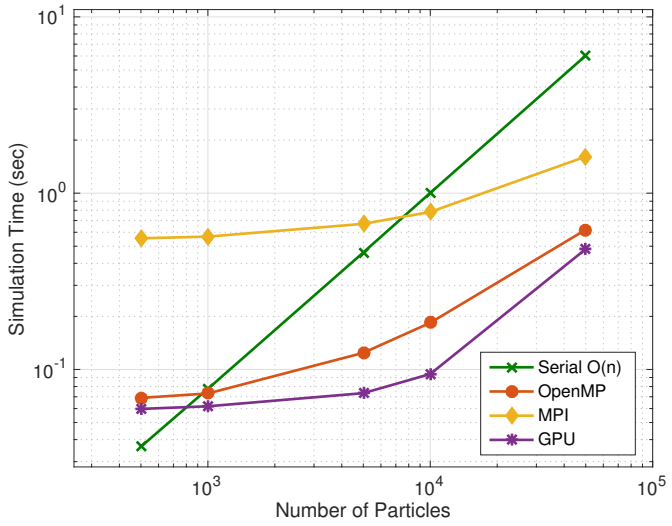

Figure 4: log-log plot of simulation time versus different number of particles for a fixed number of threads of 24

In Fig. 5 the number of flops in MPI and OpenMP is compared versus each other.

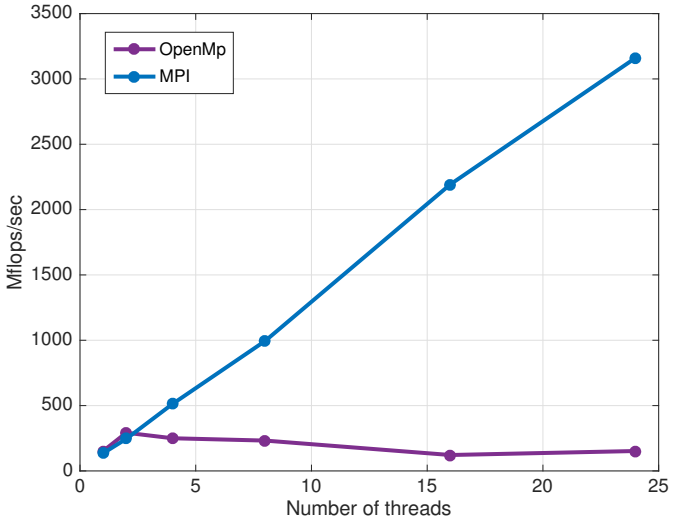

Figure 5: Comparison of Mflops/sec versus different number of threads for OpenMP and MPI

In Fig. 6 we have shown the performance of the optimized GPU code versus the naive CUDA code. As explained in section 2.4 partitioning the domain in parallel and assigning many threads to them efficiently speeds up the simulation.

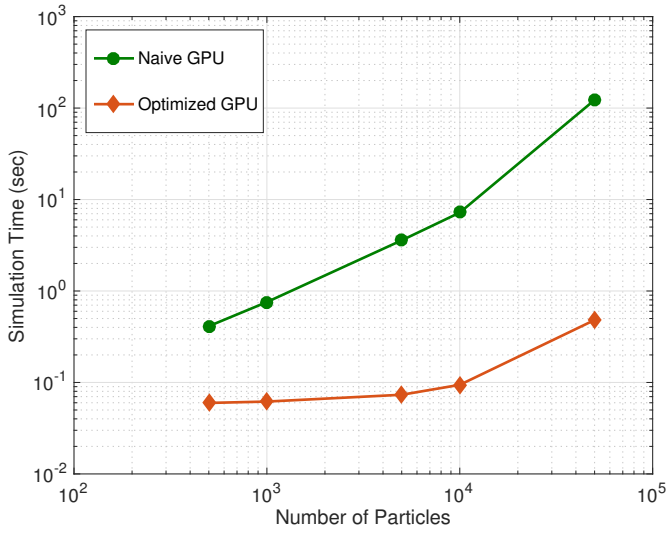

Figure 6: log-log plot of simulation time versus different number of particles for naive and optimized CUDA GPU

3.2 Scaling efficiency

In the context of high performance computing there are two common notions of scalability: The first is strong scaling, which is defined as how the solution time varies with the number of processors for a fixed total problem size. The second is weak scaling, which is defined as how the solution time varies with the number of processors for a fixed problem size per processor.

In this subsection we present our results regarding the scaling of our code for the cases of OpenMP and MPI. We also present the "slope estimate" for our optimized serial code. All the results were measured using the "autograder" code provided.

3.2.1 Optimized Serial

The *slope estimate* for line fit of our serial code was **1.10**. In Figure 7 we show the slopes estimates for different "ranges of number of Particles". That is, we show the slope estimate when we are increasing the number of particles from 500 to 1000, from 1000 to 2000, and so on and so forth.

This shows that our code is essentially of *linear* complexity.

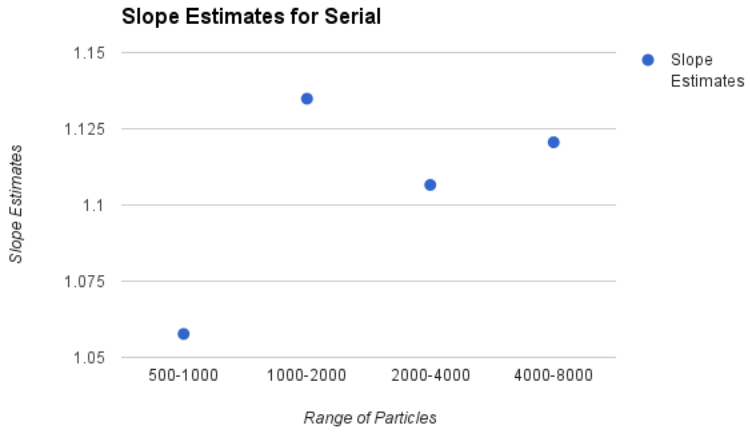

Figure 7: Slopes Estimates for Serial

3.2.2 OpenMP

For our OpenMP code, the average *strong scaling speed up* was **4.78**, while the average *strong scaling efficiency* was **0.66**. Finally, our average weak scaling efficiency was **41.59**.

In Figures 8 and 9 we present the average Strong Scaling Speed Up and Efficiency, respectively, versus the number of Threads/Processors. Figure 10 shows the weak Scaling Efficiency. As we expected, the OpenMP implementation doesn't scale in a great way as the number of processors is increased, due to the "shared memory" overhead.

Finally, we note that the "strong scaling" experiments were performed in instances of **50000 particles**.

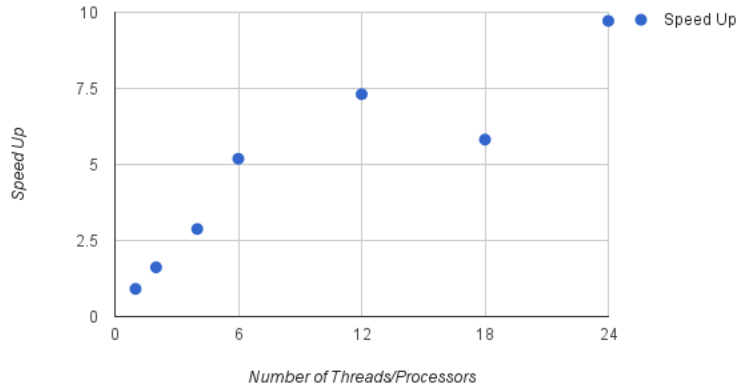

Figure 8: SS-OpenMP-Speed up

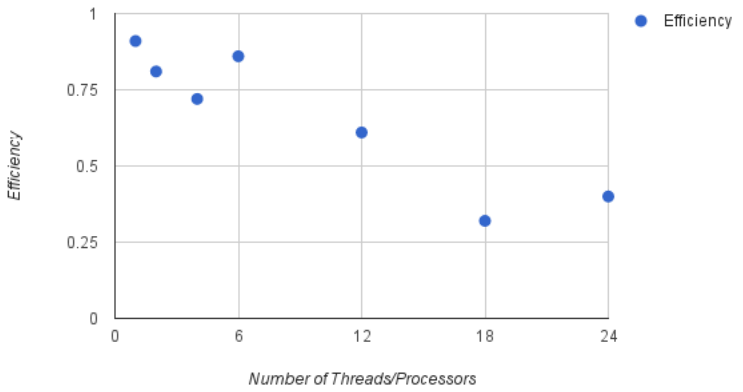

Figure 9: SS-OpenMP-Efficiency

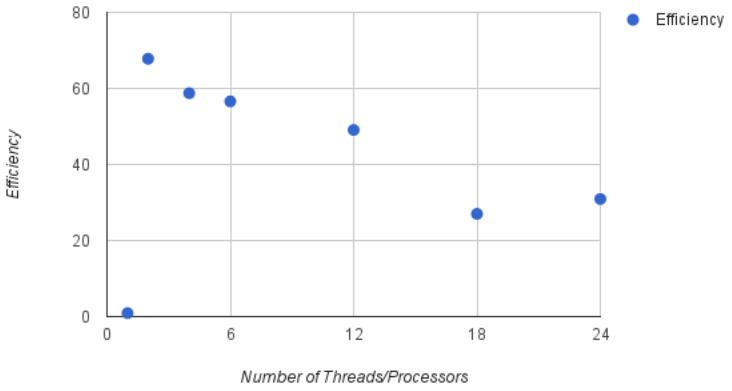

Figure 10: WS-OpenMP-Efficiency

3.2.3 MPI

In this section we present the scaling performance of our MPI code. Since MPI is a distributed memory model, we expect it to scale much better than OpenMP. This is indeed the case. However, turns out that this is not reflected to the results we got from the autograder, and there is *a good explanation*, as well as *a way to fix this*.

In particular, the (not so great) results we got from the autograder were the following (see also Figures 11, 12, 13):

- Average Strong Scaling Speed Up: **1.69**
- Average Strong Scaling Efficiency: **0.189**
- Average Weak Scaling Efficiency: **3.49**

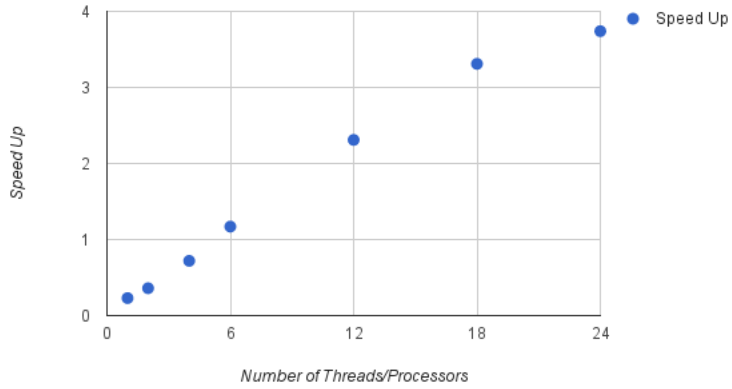

Figure 11: SS-MP-Speed up

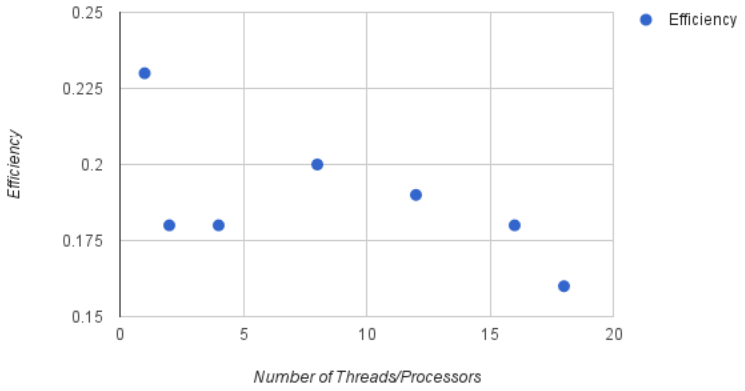

Figure 12: SS-MPI-Efficiency

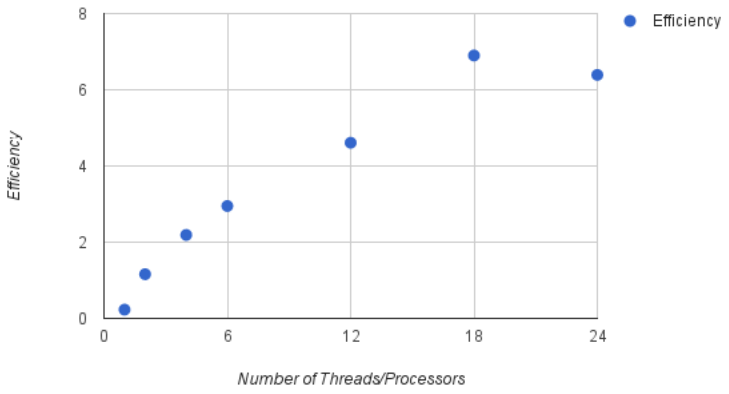

Figure 13: WS-MPI-Efficiency

The reason why the scaling results of the autograder were not great, while our code *does scale* indeed, is because the autograder compares the running time of the optimized serial implementation, with our MPI implementation for different number of processors, and not the running time of our MPI implementation running with 1 processor. The difference is actually significant, since the performance of the serial code is roughly **6** seconds, while the corresponding performance of our MPI code is **26** seconds (due to the overhead of communication). To make up for this fact we went ahead and did the "right comparison" (in the sense that it does reflect better the scaling of our code) and here are the results (see also Figures 14, 15, 16):

- *Actual* Average Strong Scaling Speed Up: **7.67**
- *Actual* Average Strong Scaling Efficiency: **0.8**

- *Actual* Average Weak Scaling Efficiency: **15.39**

Finally, we note that the "strong scaling" experiments were performed in instances of **50000 particles** and that we performed the experiments in **1** edison machine instead of **24** due to limited quota.

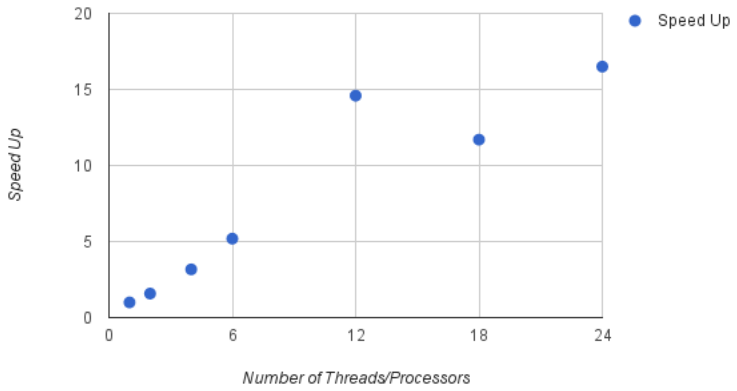

Figure 14: SS-MPI-(actual) Speed up

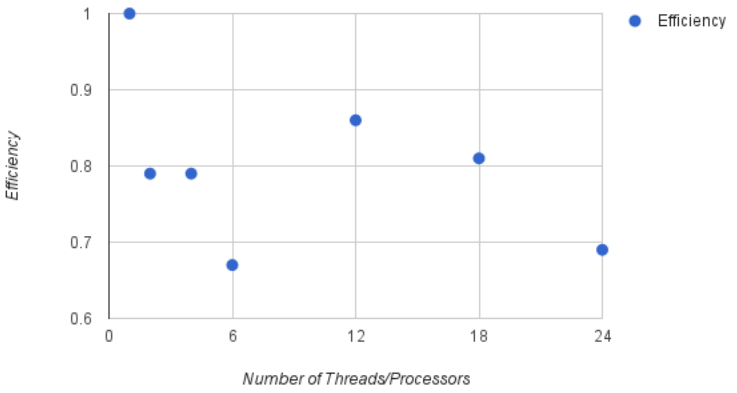

Figure 15: SS-MPI-(actual) Efficiency

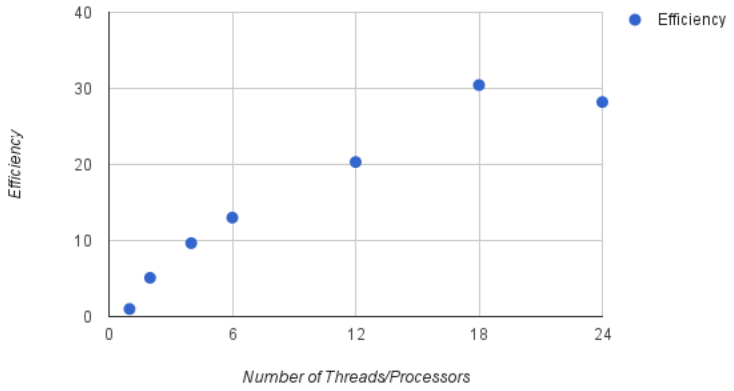

Figure 16: WS-MPI-(actual) Efficiency

4 Future Work

Due to the limited time and resource available for this project, there are a few potential improvements that we did not get to try out. We list them below and discuss why they may improve the performance.

- The given `common.c` is far from optimized. There are many places that can be improved. For example, the average and minimum distances should not be computed in `apply_forces` when `-no` flag is set. Moreover, a simple one-line change can make `apply_forces` updates the accelerations of both the particle and its neighbor. This means that we can reduce the number of multiplication and divisions by half, which should help speed up our code immensely. Note that, if we do so, we need to make sure that each pair of particles is called `apply_forces` only once but this is very simple in our serial and shared-memory implementations and is not complicated in our distributed-memory code either.

- As discussed above, our MPI code partition the plane into vertical stripes and assign each stripe to a processor. This, however, is not theoretically the most efficient way to partition the space. To minimize the number of communication, we want the total perimeter of the all the partitions to be minimized as the perimeter tells us, in expectation, how many particles need to be sent to more than one processors. For some number of processors, it is clear that there are better partitions; for example, if the number of processors is not a prime, then dividing the space both horizontally and vertically into grids yields a smaller perimeters. It is unclear to us, however, whether this is optimal. [Oud11] provides a partition that approximates the optimal perimeter. It is interesting whether such an optimal partition is known. If not, this can be a very good research question.

- We stated above that our CUDA code just uses simple arrays in contrast to vectors used by our serial, OpenMP and MPI codes. In almost all instances, arrays are faster than vectors, as vector

needs to dynamically allocate memories. It is interesting to see what kind of performance improvement we can get if we replace vectors by arrays in our serial, OpenMP and MPI implementations. Unfortunately, we were only able to prove that the number of particles in each bin is no more than six too close to the deadline and did not have enough time to convert our codes to use arrays. If we have more time, doing so should speed up our code even further.

References

[Oud11] Edouard Oudet. Approximation of partitions of least perimeter by Γ-convergence: Around Kelvin's conjecture. *Experimental Mathematics*, 20(3):260–270, 2011.

[RT93] John H. Reif and Stephen R. Tate. The complexity of n-body simulation. In *20TH ANNUAL COLLOQUIUM ON AUTOMATA, LANGUAGES AND PROGRAMMING (ICALP'93*, pages 162–176, 1993.