# CSC726 Particle Simulation Lab Part 1

Dylan King and Jinku Cui

October 26, 2019

In this lab we were asked to first improve an algorithm for particle simulation, and then parallelize our improvements.

To improve the algorithm (provided with the assignment) from $O(n^2)$ to $O(n)$, we must partition the particles into local elements. In particular, since particles a distance more than the cutoff $r$ cannot interact, if we divide our $M \times M$ domain into square domains of size $r \times r$ (in practice since $M$ is not a multiple of $R$ we will have some smaller fringe rectangles) then particles in distinct boxes will be able to interact only if one box is in the 8 adjacent boxes of the first (otherwise, the distance is too far). It is Saturday evening, and I'm afraid that we don't quite have everything working as planned. I will summarize progress first and then provide details, timings, etc. First, we have successfully implemented the $O(n)$ algorithm for the serial program. We have also parallelized this program using OpenMP; the code runs correctly in parallel on the cluster. However, we have NOT seen any performance benefit from this parallelization. In fact, we seem to only observe a massive performance detriment; and after dutiful search I cannot find the source.

First, the $O(n)$ algorithm. Written in pseudocode it is quite simple. The only component of the original algorithm which is $O(n^2)$ is that which updates the acceleration of each particle, by checking the influence of all other particles. If we check the influence of a bounded number of particles (those nearby), we may reduce this to an $O(n)$ algorithm. So we replace

```
for each particle p1
    for each particle p2
        update the acceleration of p1 according to the influence of p2
```

by the algorithm which takes advantage of bins:

```
assign the particles to the N bins
for each particle p1
    for each particle p2 in one of the 8 bins neighboring the bin containing p1
        update the acceleration of p1 according to the influence of p2
```

Of course this pseudo-code omits a significant amount of detail. In practice, we have at least two options for the forward loop. The bins are stored as an array of linked lists, where each linked list hold the particles belonging to each

bin. We rebin during each timestep; it is just an $O(n)$ operation. Then we could either

1. loop over each bin, looping over all of the particles in that bin, computing interactions with those particles in neighboring bins

2. loop over each particle, and only consider interactions from particles neighboring the bin containing our primary particle

Option 2) requires another data structure; an array telling us which bin each particle is in (also $O(n)$). My current implementation uses Option 2).

Here are the performance results (collected using some scripts kindly provided by Dr. Ballard).

| # particles | runtime (s) |
|---|---|
| 500 | 0.037066 |
| 1000 | 0.08576 |
| 2000 | 0.207199 |
| 4000 | 0.459149 |
| 8000 | 0.960847 |
| 16000 2.0229 | |

Using the curve regression code give, we find an estimated exponent of 1.15; reasonably close to 1. Plotting, we find the figure below (Latex may be sliding it around).

Now we see want to parallelize this process using OpenMP. If we examine the pseudocode; our binning may be done in parallel (with some careful consideration so that two particles are not added to the same bin simultaneously) and we may loop over each particle in parallel. It is worth noting here that some of the hard work has been done for us. The function applyForce(p1,p2) updates the acceleration of particle p1 due to particle p2, but does not affect p2. This allows us to loop blindly in parallel over p1, because threads working on two distinct p1 will never reach a race condition. This was very thoughtful of the author of the template!

So it appears that the parallelization should be easy. And indeed, after learning some OpenMP, it isn't too hard to put together something that runs, and passes all of our tests for correctness. The problem is that it is actually *slower* than the serial version for more than 1 thread; and I cannot explain why. For more details see the code itself, where I provide comments on exactly what parallel pragmas we use where and why. I discussed options 1) and 2) above because I thought perhaps this was our problem, and that switching from 1) to 2) would solve the problem. But we did this, and got essentially no improvement, so I think I'm stuck. Here are the (depressing) numbers we get when we run some tests.

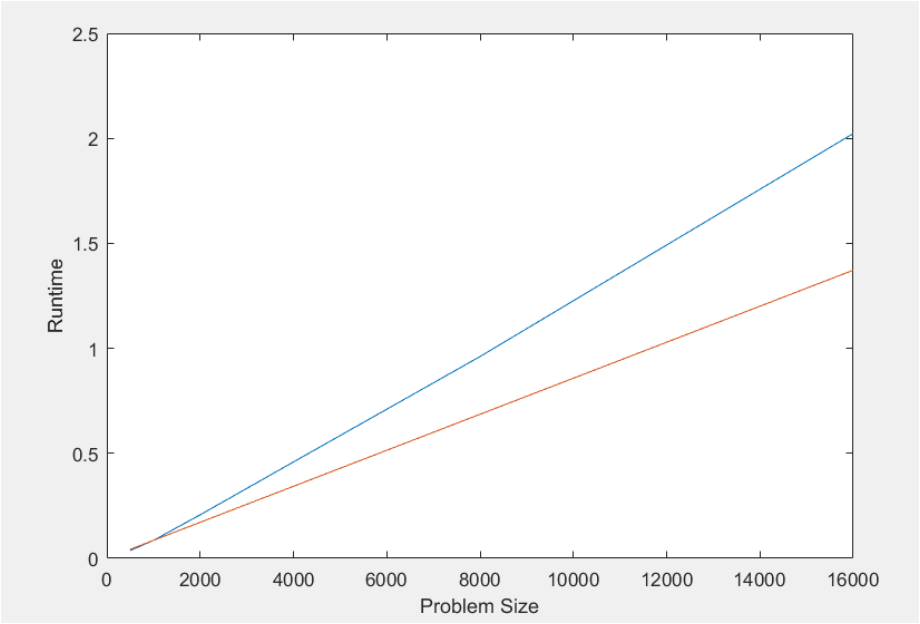


Figure 1: Orange: Linear Scaling Blue: Our Observed Results. All scales absolute

| # particles | # processors | runtime (s) |
|---|---|---|
| 500 | 1 | 0.051777 |
| 500 | 2 | 0.061207 |
| 500 | 4 | 0.074949 |
| 500 | 6 | 0.092151 |
| 500 | 8 | 0.107828 |
| 500 | 11 | 0.133219 |
| 500 | 22 | 0.223362 |
| 500 | 33 | 0.318162 |
| 500 | 44 | 0.416394 |
| 1000 | 2 | 0.122795 |
| 2000 | 4 | 0.294811 |
| 3000 | 6 | 0.469362 |
| 4000 | 8 | 0.64445 |
| 8000 | 16 | 1.35595 |
| 16000 | 32 | 2.83749 |
| 22000 | 44 | 4.02475 |

We immediately see the poor trends; more processors increases runtime!? Ridiculous. We still observe linear scaling given a fixed number of processors, but in general solving the problem size 16000 with 32 processors in parallel should not take longer than solving the same problem serially.