CSC726 Particle Simulation Lab Parts 1 2

Dylan King and Jinku Cui

November 8, 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

# particle	runtime (s)
500	0.042857
1000	0.089421
2000	0.202524
4000	0.433961
8000	0.892421
16000	1.8285

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).

Using the curve regression code give, we find an estimated exponent of 1.09; reasonably close to 1. Plotting, we find the figure below (Latex may be sliding it around).

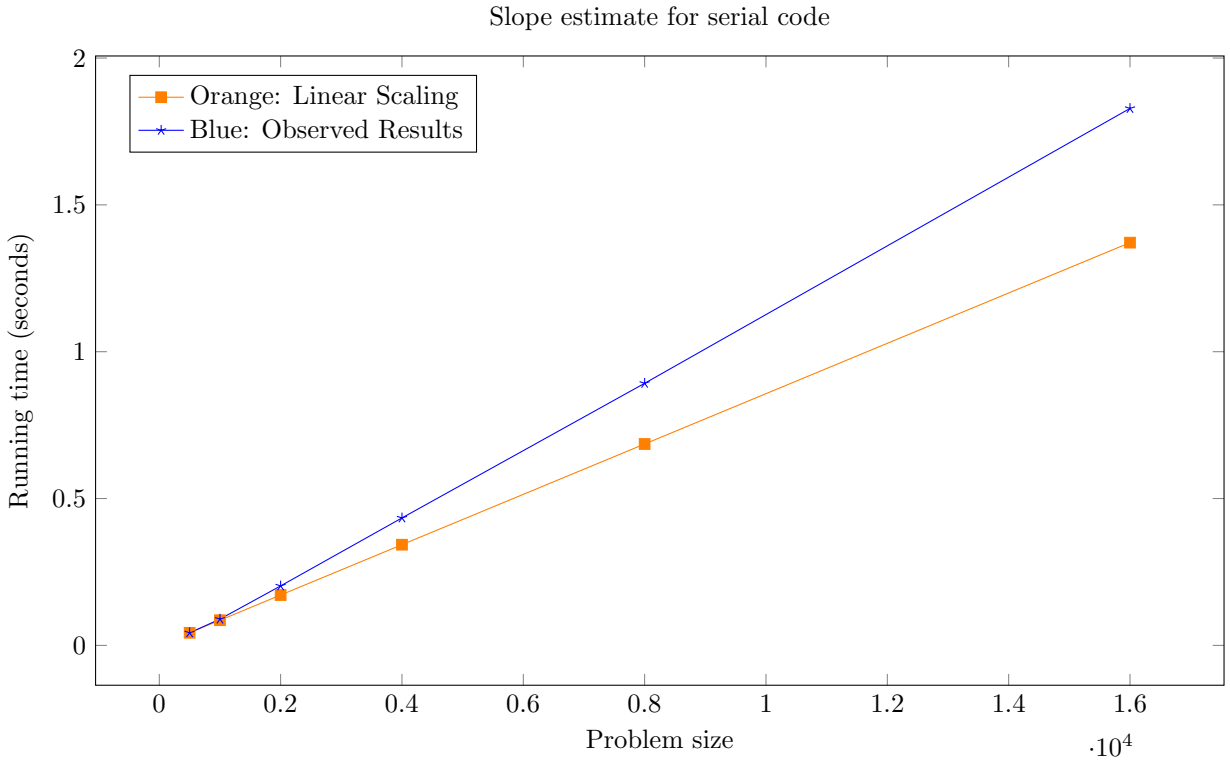


Figure 1: The slope of orange line is 1. According to the result of *auto-deac-serial*, the slope estimate for blue line fit is 1.089472.

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. OpenMP also has the traditional mechanism of a lock. A lock is somewhat similar to a critical section: it guarantees that some instructions can only be performed by one process at a time. However, a critical section is indeed about code; a lock is about data. With a lock we make sure that each of the bins can only be touched by one process at a time. Below is the results for the parallelized version.

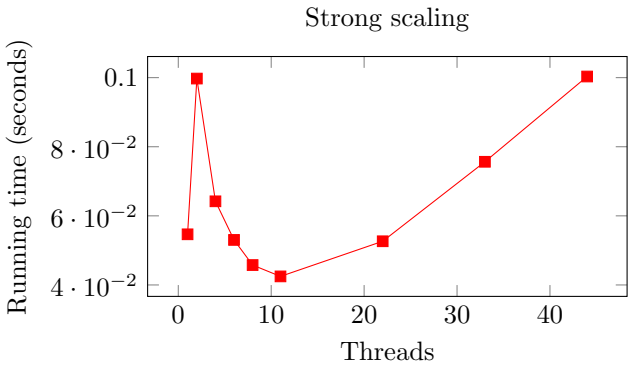
# particles	# processors	runtime (s)
500	1	0.05466
500	2	0.099734
500	4	0.064209
500	6	0.053032
500	8	0.04576
500	11	0.042475
500	22	0.052644
500	33	0.075649
500	44	0.10033

(a) Strong scaling

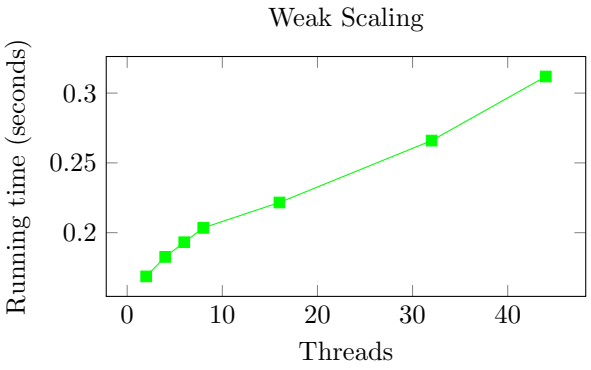
# particles	# processors	runtime (s)
1000	2	0.168685
2000	4	0.182679
3000	6	0.193156
4000	8	0.203453
8000	16	0.221634
16000	32	0.265991
22000	44	0.311882

(b) Weak scaling

The corresponding figures for strong scaling and weak scaling are plotted below.



(a) Average strong scaling efficiency is ?



(b) Average weak scaling effiency: ?

Update on 11/08/2019: We have successfully written code parallelizing this process using MPI, but it is not *quite* optimal and we are still tinkering. The cluster has been locked up for a decent portion of this evening, so I will need to gather some data soon over the next few days. Until then, here I will outline ideas, what we tried, etc.

We have p processors; a natural idea is to assign a group of particles to each, and let those p processors work on just those, and communicating all proc. locations at each step (this was currently implemented). This was great, except this communication is quite expensive, and seems superfluous.

A better idea is to use our bucket idea from above; so the particles are clumped into buckets, based on relative spacing. Then each processor can tackle a group of buckets. Then we could share all particles between each timestep, but it turns out that we only need the buckets adjacent to those used by our processor; a much smaller computation.

We have worked to implement this, and found medium success. A truly optimal scheme, I *think*, does a few rounds where each box passes to its upper, left, lower, and below - neighbors, and then recieved from them symmetrically, etc.

I was not sure this was a reasonable implementation goal, so I have programmed the relaxed (but still quite better than naive) communication of sending every border bucket to all other processors; really they are only needed by 4, but doing borders lets us use broadcast. This performs OK; in particular, if the number of buckets covered by each processor is quite large, then we will find a large speedup here.