Carlos Moustafa
aem278
aem278

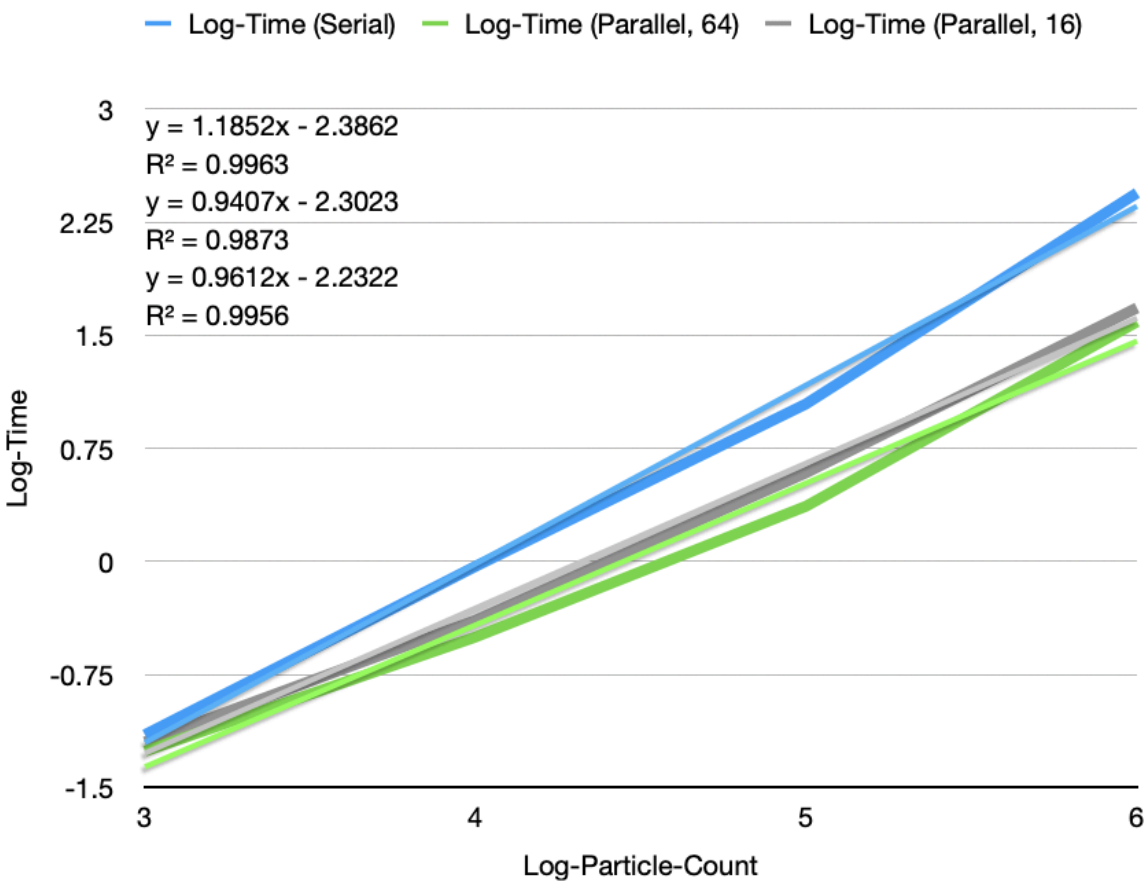
Serial Method

Design Choices

I decided to split the [0, size] x [0, size] world into a grid of mxm cells. I let the size of each cell be equal to the cutoff distance for particle forces. I chose this cell size so that I could minimize the size of each cell while ensuring correctness.

For representing the grid, I chose a 3D vector of particle_t pointers. I chose the vector due to its dynamic allocation of memory, since the number of particles in each cell changes each simulation step.

The log-log plot below shows that the serial and parallel codes run close to O(n).



Parallel Method

Design Choices

The first parallel attempt I made was to only parallelize the movement of the particles across the loops. Since this task is embarrassingly parallel (after the forces have previously been computed), parallelizing this is easy. However, I noticed a 2x slowdown from 0.114187 to 0.178474 seconds after making this change. I expect this is due to false sharing of the particles array. Since this array is owned by main.cpp, I couldn't think of an easy way to add padding to get around this false sharing.

Trying a critical section using omp criticl on binning the particles, I get 295.162 seconds of runtime for 16 threads on 1e6 particles, which was undesirable. I replaced the omp critical section with managing my own vector of locks, one lock for each grid cell. This gave a big performance improvement, bringing my solution to 86.018 seconds for 16 threads on 1e6 particles.

Additionally, I decided to use a collapse(2) on computing the forces for each particle. This is so each thread gets assigned a single cell to compute the force of particles in.

Shared Memory Synchronization

The only shared memory across threads is the grid of cells and the corresponding storage of locks for each cell. While computing the force to apply to the particles in a processor's bin, it's important not to update the position yet since this may cause a race condition with the neighboring processor's reading from that bin. Instead of updating the particles position, I store the next position of the particle in its acceleration field (since this field is unused in a timestep after applying all the forces). Then, after a sync barrier, the threads can update the position of the particles with the previously-computed values.

After this, and another barrier, I have processors rebin particles concurrently. To avoid race conditions on inserting to the shared grid, a processor must obtain the lock for the grid they will insert the particle to before doing so. This add some contention, but significantly less than using a single critical section for the entire grid.

Discussion

The figure below shows the time taken to simulate 1e6 particles, against the number of processors dedicated to that task. As we can see, it is far from the ideal p-times speedup expectation. Based on my experiments with modifying the code and observing the results, I noticed that most of the time taken in the parallel setting was due to synchronization time and barriers. I can imagine this being slightly better if I updated the parallel code to use some methods discussed in class, such as octa-tree for optimal load-balancing and actually updating the position of the particles while neighbor cells use an unmodified copy for neighbor cells to compute against.

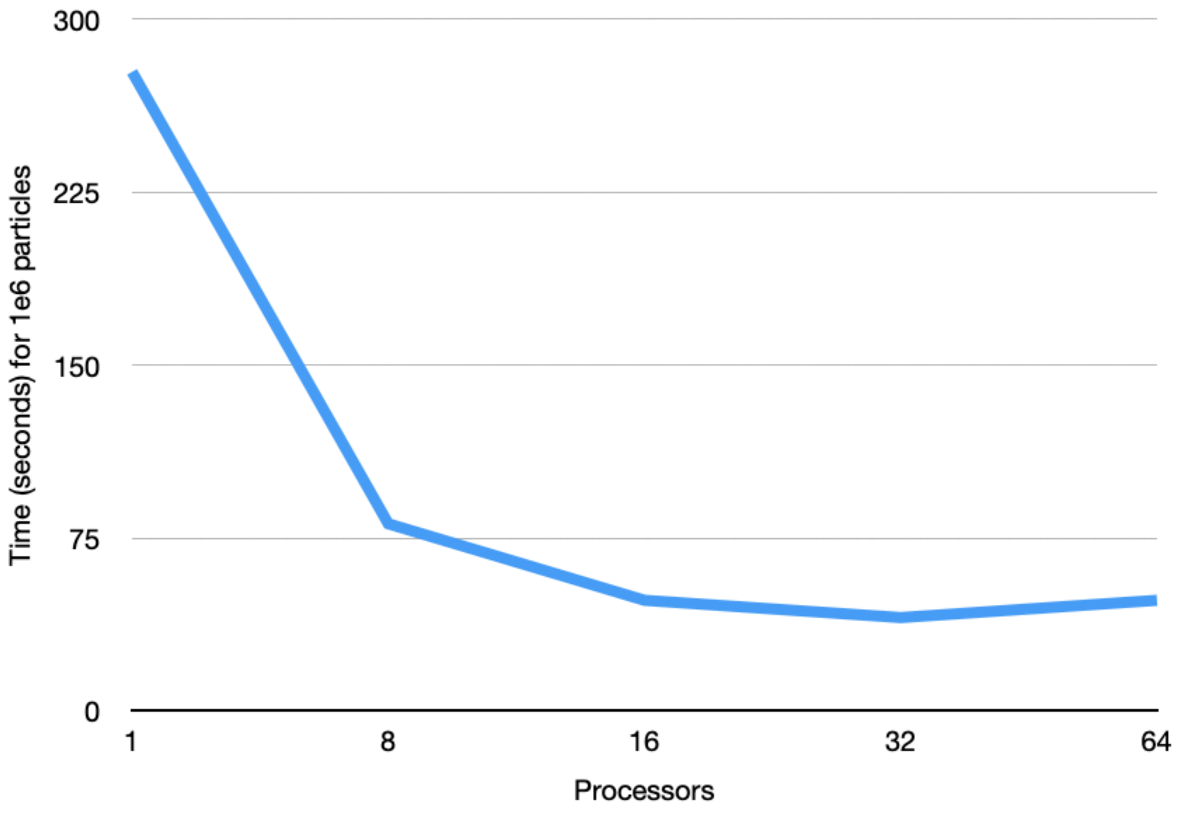
However, I'd expect that, even with these optimizations, we would not get significantly close to the ideal p-times speedup trend. I predict this using Amdahl's law. There are parts of the simulation that are inherently serial, and increasing the number of threads won't optimize this fact away. Some necessarily serial codes include:

- All particle forces must be moved *before* any particle can be rebinned
- All particles must be rebinned from the previous timestep's updates *before* any particle can begin computing local forces for the current timestep.
- During rebinning, only a single bin can be updated at once, to prevent race-conditions.

With these inherent serial properties of the task, adding more processors eventually stops improving performance.

In the figure below, it actually seems that 32 processors performed slightly better than 64 for simulating 1e6 particles. This may be due to added overhead of synchronizing more threads outweighing the benefits of adding such threads.

If I had more time, I would have experimented with modifying the grid cell sizes so that there is exactly one cell per thread. I expect this would prevent threads' average idle times, and balance load better on average across the threads. It would also have the nice property that all cells of the grid are being worked on concurrently, and no cell must wait to be processed.


300
225
150
75
0
Time (seconds) for 1e6 particles
1
8
16
32
64
Processors