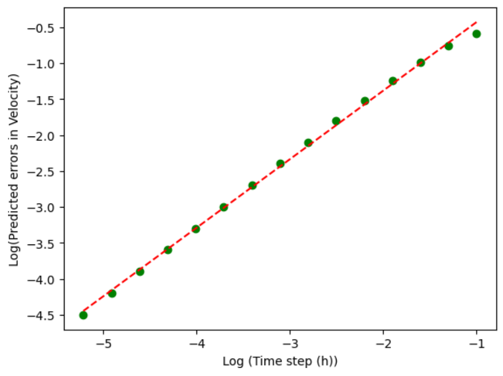
Parallel Scientific Computing 1; N-body solver with Euler and Linked cell algorithms

STEP 1: Gravitational forces

The implementation is of time complexity $O(N^2)$. To consider the interactions between all particles, we calculate the forces between each particle. For this reason, we iterate through each particle twice, making the complexity $O(N^2)$. The memory complexity still stands at $O(N)$, as we maintain 3 arrays of length N.

We use the symmetry principle of newtons 3rd law to reduce the number of force calculations we need. This means each particle exerts and experiences an equal and opposite force. Hence, cutting down our force calculations by a factor of 0.5. This is done by implementing a sliding window approach, where we only iterate through unique pairs of particles, and add the equal and opposite values for each particle (upper triangle of particle matrix).

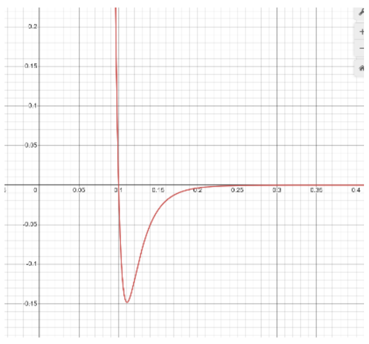
To reduce runtime complexity, we can also determine which forces to calculate. The forces are inversely proportional to the square of the Euclidean distance between two particles. This means that at very large distances the force exerted due to mass would be negligible, hence we can reduce those calculations. To do this we can implement a linked cell algorithm and determine a cut-off radius dependent on the mass of the objects, such that any particles further than that cut-off radius would not have the forces calculated.



We use the explicit Euler method as a time stepping scheme, which has convergence of 1. To study the convergence of the implementation, I tracked the maximum velocity between two particles in a system where 2 particles start from positions (0,0,0) and (0,30,0) with the same velocities (1,0,0) and masses of 50. This is a two-particle setup, where two particles in parallel curve towards each other and collide at time 39s. I track the maximum velocity at 39s, right before collision, and estimate the error as the difference between timestep(h) and its successive half(h/2). In the graph the y-axis we plot the log of the difference between the estimated max velocity at t=39s, by h and h/2. The reason
for choosing high masses and the timestep before collision, is that the gradient is at its maximum, hence the possibility of error in calculation of velocity would be the highest. We find the convergence order of our implementation to be 0.95. This is slightly lower than the expected order 1 of explicit Euler because of the round off error especially at lower timesteps.

The convergence order remains the same for collisions as well. This is because the mechanics are not being changed. Thus, we observe the same convergence order of 0.95.

STEP 2: Molecular forces



To choose the cut-off radius, we plot the force calculation as a graph. Here we notice that the molecular forces have asymptotic behavior at x =0, as its attractive between ranges [-0,1,0.1] and attractive at distances greater than 0.1. After 0.22 the force magnitude decreases exponentially and is almost insignificant. To choose the cut-off radius we plot the graph of molecular forces as distance varies. We notice asymptotic behavior at distance greater than 0.1, as the force decreases significantly after d=0.22 to almost negligible. Hence, we choose cut-off radius as 0.22, hence any particles with distance greater than 0.22 between each other we don't calculate the forces between as it would not make a significant difference for their velocity calculations.

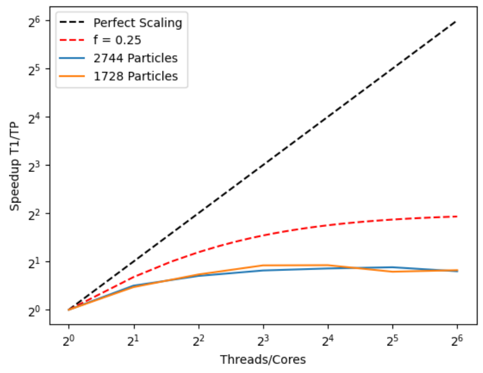
To implement this, I used a multidimensional vector for the cell data structure. With a combination of 3 indices, I can refer to all cells which exist in our constrained space. The reason for doing this is that it makes it easier to find the neighboring cells. Each cell refers to another vector storing the indices of particles that exist within that cell. Using these indices, we then access the mass, velocity and force arrays for each particle and update them.

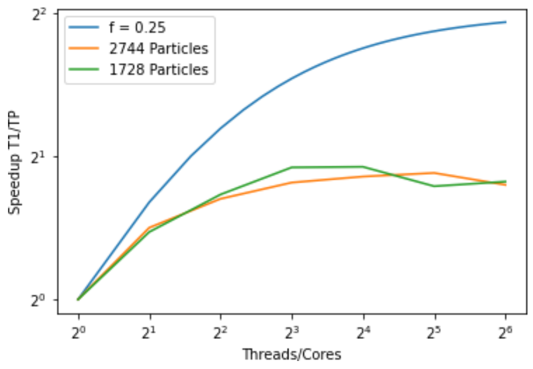
In the first implementation, we stored the indices of the particles that are inside the cell. In memory those particles may not be close to each other. While calculating the potential, we access particles inside a cell one at a time. Hence, our access to memory is not continuous. As the particles only vibrate, this means that the relative positions and velocities of the particles don't change often. Therefore, particles would often exist in the same cell. Therefore, instead of storing the index of particles in the cell, we store the particle information itself resulting in contiguous blocks of memory and quicker lookups. This is more relevant to our problem as it a static data structure would be more useful in this case.

STEP 4: Instruction level parallelization

All our tests are performed on Hamilton 8, where each consists of 2 sockets of AMD EPYC 7702 64-Core Processors with 128 total cores. During parallelization, one thread runs on one core, hence there is no hyperthreading. All code is compiled using g++ compiler with flags -fopenmp -O3 -march=native --std=c++0x -fno-math-errno, and intel compiler with flags -qopenmp -O3 -xHost --std=c++0x. Here, the march=native means that the code is tuned specific to the micro architecture of the host. O3 is used to auto vectorize the code. The fopenmp flag is to compile using open MP.

Strong Scaling





To test the scalability of my code, I undertook a strong scaling study which measures scalability given a fixed problem size, in our case measuring runtime between 1,2,4…,64 cores. For each result, we ran our program 5 times and took the average to account for variability. We disabled any printing during our runs, and the setup time for our particles was considered. I calculated the proportional of parallel code in my program, by timing the length of time spent in the parallel regions of the code under 1 thread. We saw consistently for all varying problem sizes that f = 0.25 (proportion of sequential code). We would expect this to reduce with more particles, since the amount of work done within our parallel region should proportionally increase to any setup times, however our collision (sequential) calculations also share the same time complexity and hence trading off our force calculations. I plot Amdahl's law given the ratio of sequential component of our code; we can therefore theories an upper bound for speedup as we increase the number of cores. Speedup is defined by the amount of time taken for 1 core to run the program divided by the time taken by N cores (N cores should ideally return an N speedup). From our plots we can see our program does not scale effectively since its performance stagnates with more cores quickly with respect to our Amdahl plot. This could be due to the overhead caused by the number of threads we spawned, and latency due to memory bottlenecks