Parallel Scientific Computing

Piotr Borowiecki

1 N-body simulator

In step 1 an n-body simulator was implemented using the naive approach of directly calculating interactions between all pairs of bodies. As a time-stepping scheme, the Velocity Störmer Verlet Method method was used, which is a second-order method.

1.1 Accuracy

Accuracy of the time-stepping method was verified experimentally by considering a system o two bodies, A and B, orbiting the center of mass. Initial positions, velocities and masses were set to

$$x_A = (1, 0, 0), v_A = (0, 0.5, 0), m_A = 1.0$$
$$x_B = (-1, 0, 0), v_B = (0, -0.5, 0), m_B = 1.0$$

In such a system, body A satisfies $x_A(t) = \big(cos(\frac{t}{2}), sin(\frac{t}{2}), 0\big)$, and this was compared to the outcome of the simulation at time $t = 1$ as the time-step δt varied. The error was calculated as the distance between predicted position and the outcome of the simulation, and plotted against δt as shown in Figure 1. The resulting plot is quadratic as expected.

1.2 Complexity

Since at every step all the interactions are directly calculated, the algorithm used has order of complexity $O(N^2)$. This could be reduced to $O(N \log N)$ or even $O(N)$ for the Fast Multipole Method. However, although the asymptotic complexity of these two methods is superior, there is an overhead of constructing an octree at every step, which could offset any potential time savings for small enough N.

1.3 Collisions

A fixed time-step can lead to some collisions going "unnoticed" as these are checked with no interpolation. This can be mitigated by adjusting time-step, so that the leaps at each step are bounded and the proximity condition for collision can be reliably

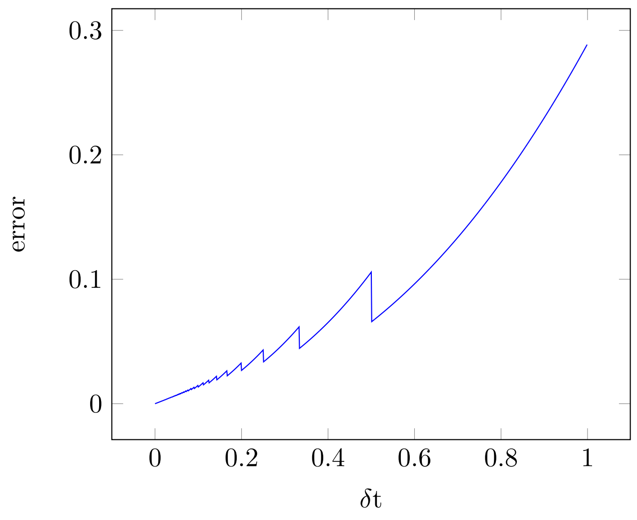

Figure 1: Accuracy of the time-stepping scheme (Velocity Störmer Verlet)

checked at each step. By bounding the time-step from above, we can ensure that the convergence order remains the same, but the computation times become unpredictable.

2 Molecular Forces

In this section the cell-list algorithm was used to achieve order of complexity $O(N)$. The 3D-space is partitioned into cells of side length equal to twice the cut-off radius, and each cell contains the list of particles within it bounds, which is maintained on-the-fly throughout the simulation. Each particle interacts only with particles inside the same or neighbouring cells.

2.1 Data structure

The grid of cells is represented by an STL container `std::unordered_map` which allows for memory-efficient storage of cells with average look-up times of order $O(1)$, and the cells themselves are represented by `std::unordered_set`. Using these containers allows for, on the one hand, unbounded simulation box and on the other, keeping in memory only the cells which are currently occupied by particles. Therefore they are good both for fast-moving and for vibrating particles.

3 Vectorisation

Both `gcc` and `icpc` successfully vectorise the inner loop in the most computationally expensive step of calculating gravitational forces between pairs of particles, resulting in significant performance gains. For example, computation time for a simulation with parameters $t_{end} = 10, \delta t = 0.1$ for $N = 10000$ bodies decreased from 43 to 10.9 seconds (on an AMD EPYC 7B12 using `avx2` instruction set), an impressive, almost 400% performance boost.

4 Instruction-level parallelism

The vectorised code from previous step was parallelised using OpenMP's `#pragma` clauses. Due to data race issues the symmetry of gravitational force is no longer exploited, which doubles the number of interaction to compute at each step. This can be mitigated by, for example, having each thread work on its own, local copy of an array for storing contributions to acceleration of each particle, and then accumulating this data into the global array at the end, but this would require extra memory to store $3N\times$ (the number of threads) numbers, which could become problematic for large, massively parallel simulations.

4.1 Scalability

Since I never succeeded logging in to Hamilton, the scalability tests were performed using resources provided by Intel Developer Cloud, namely a dual 6-core Intel Xeon Gold 6128 CPU, giving 24 threads (2 threads per core) in total. As evidenced by the results, OpenMP does an excellent job parallelising the code with very little effort from the programmer. Tests were performed using parameter $t_{end} = 1.0, \delta t = 0.01$ with N ranging from 1000 to 10000, with particles being randomly, uniformly distributed in a unit cube centered at the origin, each of mass 1. The results are shown on Figure 2.

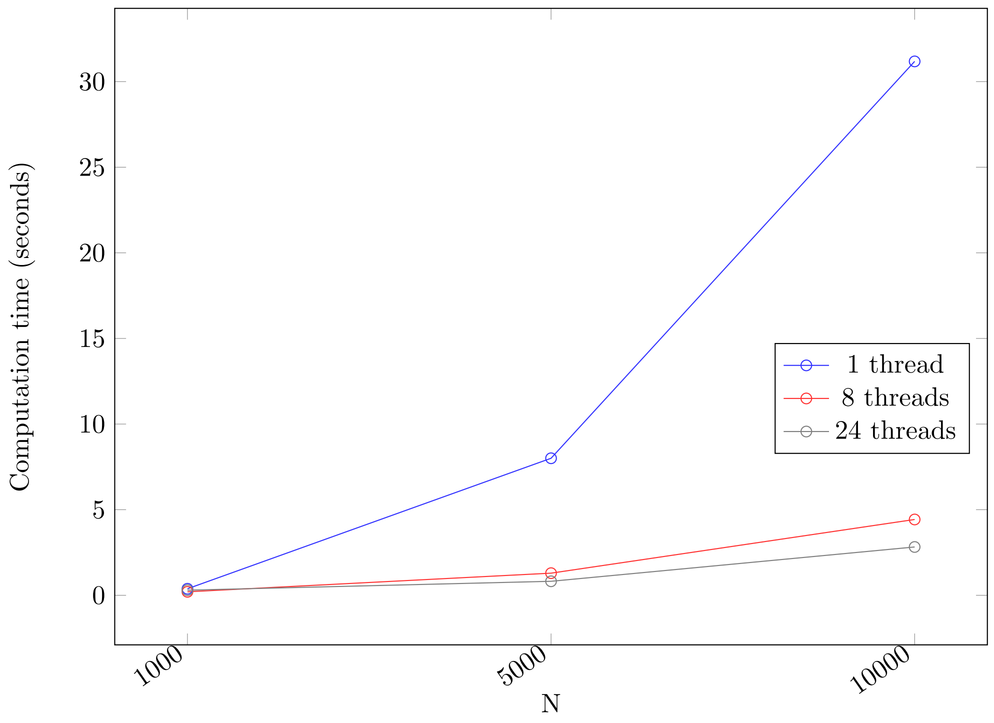

Figure 2: Scalability