HPP Group Assignment

Olof Björck, Gunnlaugur Geirsson

February 22, 2019

Assignment 4

The problem

The evolution of a number of particles in space can be simulated using Newtonian mechanics. This can be implemented numerically using Plummer spheres and symplectic Euler time integration. A straightforward implementation requires $O(N^2)$ computations for N particles but this can be improved using the Barnes-Hut algorithm. In this report, we implement the Barnes-Hut algorithm for simulation of the evolution of a number of particles in space in two dimensions. We will investigate the performance and complexity of our Barnes-Hut implementation to see if it indeed is a favorable implementation of the N-body problem to a naive implementation.

The solution

Our solution algorithm can be divided into the following steps:

- Read input data from the command line;
- Read input data about particles from input file;
- Simulate with the Barnes-Hut algorithm; and finally
- Write particles updated data to output file.

Particle structure

The simulation is done by updating the state of the particles by a timestep several times, with the timestep size, and the number of timesteps to simulate, given as input. The input variables are set as constants (keyword `const`) and every particle is represented by separate arrays of information, i.e. position, mass, and velocity. The two straightforward ways to represent the particles in C is either as an array of particle structs, or as a "particles" struct of arrays. We tried both methods and found no real speedup difference with either approach. When implementing Barnes-Hut, we decided to use a single struct of array pointers, as this can supposedly give better performance in some cases.

Quadtree

The Barnes-Hut algorithm includes building a quadtree. In order to do this, we create a quadtree node structure consisting of pointers to children nodes, the total node mass and center of mass, and the node side length and center position of the node. There is no reason to store four separate pointers to each child, if all four children are allocated contiguously. Only a pointer to the first child is required, which can then be incremented to reach the other children. The tool `valgrind` was of great use here, to find illegal memory reads and writes and other memory leaks.

With this tree-node structure, we also update the compound node mass and center of mass for each node traversed when inserting a new node into the quadtree. That is, we did not need a separate function to calculate the node masses and center of masses after the quadtree had been built. This eliminates the need

to traverse the quadtree more times than necessary. Much time was taken to order the conditionals when inserting a new particle, so likely conditions would occur first.

To improve performance we perform no bounds checking when inserting a new node. This removes then need for a conditional when looping through the particles, as the cost of undefined behaviour if particles start leaving the playing field.

Simulation

When calculating particle interactions, it is apparent that computing the forces between particles is redundant. Instead of calculating forces, we calculate the resulting accelerations of the particles to avoid first multiplying by mass and then dividing by mass later. The particle forces (and then the particle velocity and position) are calculated by recursively traversing the quadtree, checking if each node traversed satisfies the theta condition. If the condition is met, then the node is treated as a particle and the acceleration computed, otherwise the tree traversal continues. This computation is entirely parallel for each particle within a single timestep, and is parallelized using pthreads. Due to the parallelization and tree structure there is unfortunately no way to utilize Newton's third law however, unlike the $\mathcal{O}(N^2)$ approach.

The brightness information about a particle is not used in any calculation. We chose to store brightness in a separate array used only for writing to output file.

Performance and discussion

The Barnes-Hut algorithm is not trivial to parallelize into separate tasks; hence, we worked together most of the time when writing and debugging the code, and when writing the report. There was no clear division of work.

The program was compiled and run on an Intel(R) Core(TM) i5-5257U CPU @ 2.70GHz processor in macOS Mojave version 10.14.3 (18D109). The compiler used was Apple LLVM version 10.0.0 (clang-1000.11.45.5), with thread model posix.

Optimization techniques attempted

We used some general optimization techniques throughout the code:

- Strength reduction by adding instead of multiplying by a known small constant;
- Evaluating cheap conditionals before evaluating more expensive ones;
- Tried to do as little work in general as possible;
- Tried to improve branch prediction by doing as little work as possible inside loops specifically;
- Usage of keywords: Declaring `const` and `__restrict` was done whenever possible. Keyword `__attribute__((pure))` was utilized for one function, used for finding the correct quadrant to place a new node in, but it did not seem appropriate for most other functions;
- Padding and packing: Variables in structs were ordered largest first, so as to minimize padding. Use of keyword `__attribute__((packed))` decreased performance in all cases and so was not used.

The code was compiled with flag `-O3` (`-O3` performed better than any other `-O` flag) and using `-march=native`. In addition, the following compiler flags and general optimizations were used when compiling, and throughout the code, due to all of them increasing speedup, even if just slightly.

- `-ffunroll-loops`: It is often necessary to loop through all four children of a quadtree node, which are unrolled with this compiler flag instead of manually. The same is true for any loops which make threads. Due to the program relying more on recursion than loops, there was no major speedup when using compiler unrolling, nor any obvious cases of manual unrolling in the code.
- `-ffast-math`: Using this flag provided more speedup, at next to no increase in error.

- `-ftree-vectorize`: Due to the inherently recursive nature of the code, there are not many operations which are trivial to vectorize. Using this keyword, only the function call to insert a node is vectorized according to -fopt-info-vec, which seems rather reductive. Either way, it appears to give a slight speedup.

Best results

The best time found for 5000 particles (from input file `ellipse_N_05000.gal`) with 100 timesteps and step size 0.00001 was 1.753 seconds, timed with the built-in timing command `time` in the macOS terminal (bash). All timing results presented are the real-time (wall time) runtimes of the program.

The space complexity of Barnes-Hut compared to the $O(N^2)$ method is shown in figure 1. The Barnes-Hut clearly has a much better performance for large N, with an order of around $O(N^{1.2})$ for large N. Due to the tree-search method involved and the slightly superlinear performance, it seems reasonable to assume that Barnes-Hut is $O(N \log N)$.

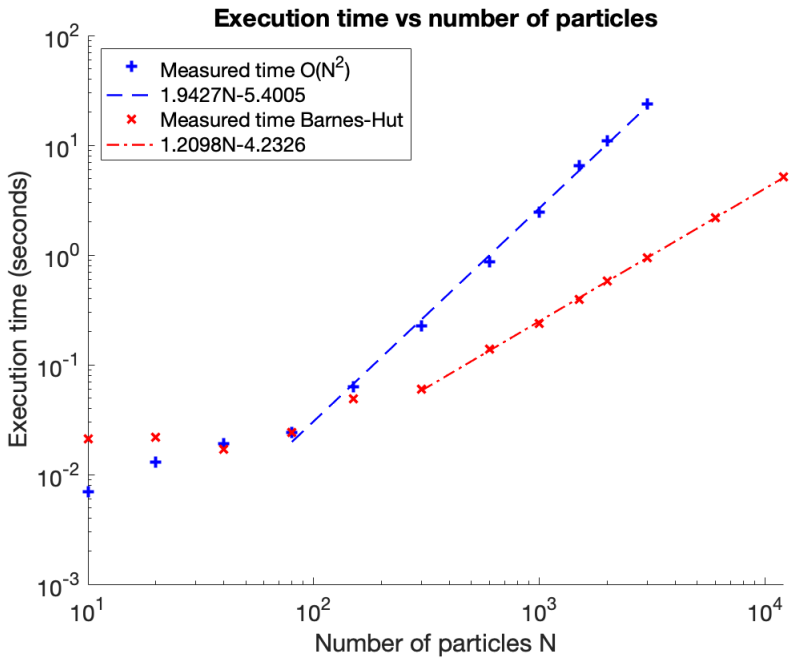

Figure 1: The execution time plotted against number of particles included in the simulation for a naive implementation and a Barnes-Hut implementation of the N-body problem. The $O(N^2)$ complexity is shown by the straight line $1.9427N - 5.4005$ with slope approximately 2 and the Barnes-Hut complexity is shown by the straight line $1.2098N - 4.2326$. Both algorithms were run with 100 time steps, with theta chosen to ensure the relative error is below 0.001.

Bottlenecks

Using `cachegrind` it seems that the greatest code bottleneck is during simulation, when recursing the tree. Around 20% of branch predictions fail when checking theta. There is no obvious solution however, as the check must be made to decide whether to continue traversing the tree. Unfortunately we could not utilize `gprof` to study the bottlenecks further, due to problems running the program on our computers.