HPP Group Assignment

Olof Björck, Gunnlaugur Geirsson

March 2, 2019

Assignment 5

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 a multi-threaded version of 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 determine its scalability and performance for different number of threads.

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 each given timestep, 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 do 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. Due to the tree structure there is unfortunately no simple way to utilize Newton's third law with this approach, 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.

Parallelization

Most of the computation time in each timestep is spend updating the particles, not building the tree, see table 1. It is thus logical to focus any efforts of parallelizing the code on the particle update algorithm.

The computations to update particle positions are entirely parallel for each particle within a single timestep (i.e. the particle positions can be updated without affecting one another, in any order) as the non-updated quadtree stores all the necessary information. A positive and very welcome result of the information necessary for the computations being separate from the particle information that is updated is that there is no data dependency between the particles being updated; the computations are thus easily parallelized.

The parallelization is done by splitting the serial updating of particles into parallel tasks of equal workload for each thread, with the last thread making sure that all particles are updated even if the number of particles is not evenly divisible by the number of threads. This approach causes the last thread to perform more work than the other threads, causing a load imbalance and a negative effect on synchronization. A solution is to perform a second parallelization of the particles that did not fit in the first parallelization. However, this did not lead to any speedup, likely due to the increased overhead cost of a second parallelization, and that the leftover particles are very few unless using many cores, e.g. a maximum of three extra particles when using four cores. Three extra particles is negligible when performing larger simulations where every thread works on several hundreds of particles. In a situation where the number of cores used are comparable to the number of particles, this might yield an improvement, but that is not the case here.

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 serial 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, and declaring loop conditions to be known at compile time whenever possible.
- 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. Keyword inline can be declared for multiple functions (barring recursive ones), but seems to offer no speedup in any case;
- 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`: Included in -O3. Due to the inherently recursive nature of the code, there are not many operations which are trivial to vectorize. 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. Explicit vectorization was not done in the code.

Best results

The best time found for 5000 particles (from input file `ellipse_N_05000.gal`) with 100 timesteps, step size 0.00001, and `theta_max` = 0.12 was 1.723 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 $O(N \log N)$ for high enough `theta_max`. We used `theta_max` = 0.12 which we found to result in an acceptably low error.

Bottlenecks

Using `cachegrind` it seems that the greatest code bottleneck is during simulation, when checking for `theta_max`, with around 20% of branch predictions failing on this condition. There is no clear solution to this, as the check must be made to decide whether to continue traversing the tree. Unfortunately we are unable to utilize `gprof` to study the bottlenecks further, due to problems running gprof on both our computers, and the university server vitsippa.

Parallelization

As we could not utilize gprof to profile the execution time of the program, we have to rely on software timers, and use clock() instead. It is apparent that the earlier assumption, regardig the computation time required to create the tree being marginal compared to the time spent updating the particle positions, is very true for large N. This can be seen in table 1. Hence our decision to focus on parallelizing the simulation part of the code is justified.

The speedup by using several threads in a simulation of $N = 10000$ particles is shown in figure 2. In figure 2, it is obvious that using threads entails a significant speedup. The execution time when using 4 threads is approximately half of the execution time when using 1 thread. However, increasing the number of threads beyond 4 does not seem to entail any speedup, but rather decrease performance as the thread

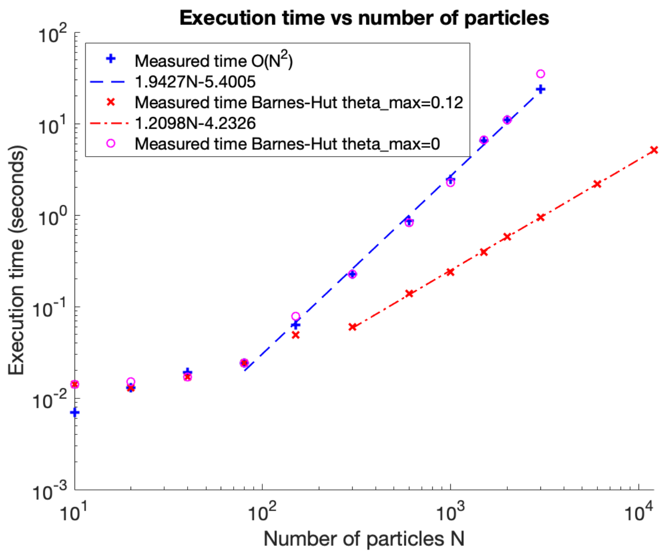

Figure 1: The execution time plotted against number of particles included in the simulation for a naive implementation and a single-thread Barnes-Hut implementation of the N-body problem. The $O(N^2)$ complexity is shown by the straight line $1.9427N - 5.4005$ with slope approximately 2 and the Barnes-Hut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

Table 1: Total serial execution time of different parts of code, simulating an elliptical galaxy of size N=10000 for 100 timesteps, with `theta_max` $= 0.2$

Computation	Time in seconds [s]
Building and freeing quadtrees	0.1832
Simulating particles	12.2821

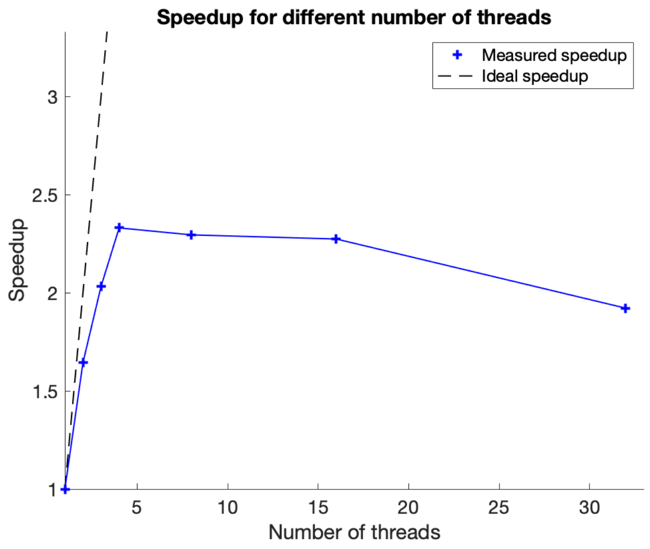

Figure 2: The execution time plotted against number of threads used in the simulation. The simulation were of 10000 particles for 100 time steps, with `theta_max` = 0.12. The computer used has 4 cores.

overhead increases. This is expected behaviour, as the processor used has 4 cores. The speedup by using several threads in a smaller simulation of $N = 3000$ particles is shown in figure 3. The speedup is not as large as with $N = 10000$ which can be explained by the smaller simulation not making as much use of the threads every iteration.

We also investigated the speedup using several threads on the university server vitsippa which has 32 cores (AMD Opteron(tm) Processor 6282 SE). This is shown in figure 4. From figure 4, we can discern that the speedup was the greatest using 32 threads which is expected from the concordant number of cores.

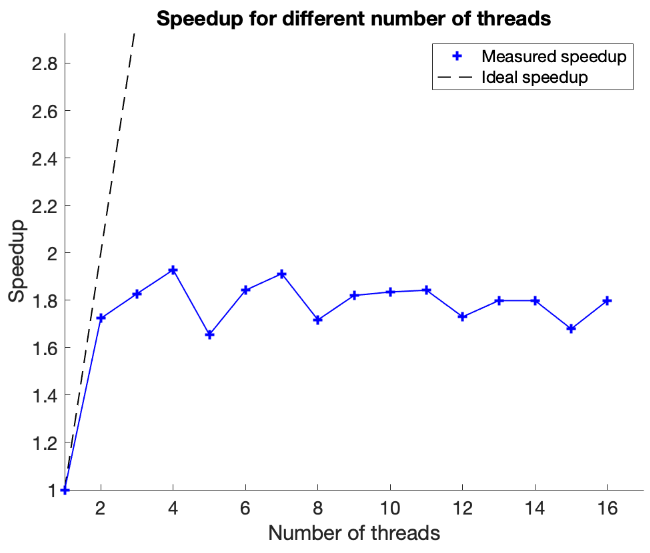

Figure 3: The execution time plotted against number of threads used in the simulation. The simulation were of 3000 particles for 100 time steps, with `theta_max` = 0.12. The computer used has 4 cores.

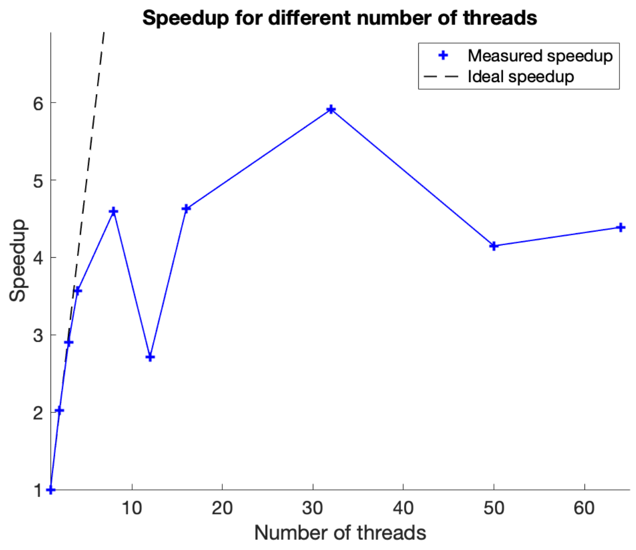

Figure 4: The execution time plotted against number of threads used in the simulation. The simulation were of 3000 particles for 100 time steps, with `theta_max` = 0.12. The computer used has 32 cores.