Written Exam in HPPS—Practical Part

January 28—30, 2021

Preamble

This is the problem text practical part of the HPPS exam. This document consists of iii pages excluding this preamble; make sure you have them all. See the preamble of the theory part for general information. Your solution of this practical part is expected to consist of a *short* report in PDF format, *separate* from your solution of the theory part, as well as a `.zip` or `.tar.gz` archive containing your source code. That is, you are expected to upload *three* files in total for the entire exam. The report is not expected to be as polished as in the assignments. Focus on concrete details and numeric results.

1 Background

An N-body simulation is a common scientific model, where we simulate the interactions between N particles in some space. A particularly common case, and the one you will be working with for the exam, is simulating the gravitational interactions between particles moving around in three-dimensional space. This is used in astronomy to simulate e.g. galaxies. We treat each particle as a *point mass*, which means that collisions are impossible, but the distance between particles affect the strength of their mutual attraction.

Mathematically, we are given n particles, each identified by its position $\vec{p_i}$ velocity $\vec{v_i}$ (both 3D vectors), and mass m_i (a scalar). We can compute the *acceleration* $\vec{a_i}$ of particle i as affected by all other particles with the formula

$$\vec{a_i} = \sum_{j}^{N} \frac{m_j(\vec{p_j} - \vec{p_i})}{(|\vec{p_j} - \vec{p_i}|^2 + \epsilon^2)^{3/2}}$$

where ϵ is a softening constant used to avoid excessive interactions between particles that are extremely close (and also to avoid division by zero for $i = j$).

After computing the accelerations, the velocity v_i of each particle can then be updated, and then finally the positions p_i. In a real simulation, this is repeated for some number of steps to simulate the progress of time. You will only be working with a single step. As computing the acceleration for a single particle involves looking at all N particles, the asymptotic complexity is $O(N^2)$.

2 Your task

You are given a working Python implementation of N-body simulation, `nbody.py`. It is functionally correct, but very slow. Your task is to write a corresponding C program, and to optimise it as best you can. The program reads and writes textual data files where each line corresponds to a particle in the following format:

```
mass px py pz vx vy vz
```

See for example `5_bodies.txt` in the handout. Code for reading and writing these files is part of the handout. We can run the Python N-body simulation as follows:

```
$ ./nbody.py 5_bodies.txt 5_bodies_out.txt
```

This produces a file `5_bodies_out.txt`. When you write your C implementation you should make sure they produce the same results as the Python implementation.

The handout has a third optional parameter that defines how many iterations the program should run. This allows you to keep data loaded into memory and repeat the computational part. To run the example with 100 iterations you can add the argument as follows:

```
$ ./nbody.py 5_bodies.txt 5_bodies_out.txt 100
```

The handout Python and skeleton C programs are constructed to treat commandline arguments the same, so you can easily switch between the two.

3 Task specifics

You are expected to complete the following tasks, documented in a short report.

3.1 Implement N-body in C

The code handout contains a skeleton implementation `nbody.c`. It can read and write data files, but the simulation function itself is incomplete. The implementation makes use of the file `util.h`, which you do *not* need to modify. Among other things, it defines a type `struct particle` for representing a single particle.

Your tasks for 3.1 are:

- Finish the C-based implementation.
- Measure the performance of your implementation for various N.
- Measure the performance of the Python implementation for the same N values.
- Include the results in your hand-in.

3.2 Estimate scalability

Estimate the parallel fraction p of the N-body simulation (either the handed out Python code or your own C version).

Your tasks for 3.2 are:

- Provide your estimate of the parallel fraction of the N-body simulation.
- Provide a brief description of how you arrived at this fraction.

3.3 Parallelise and optimise your N-body implementation

Use what you have learned in the course to make your implementation run faster. Make sure to submit both your original implementation as well as any optimised ones. Here are two possible improvements:

1. Parallelise the simulation with OpenMP. Compare your scaling results with what might have been expected from Amdahl's and/or Gustafson's laws.
2. In the `nbody` function, instead of traversing arrays of `struct particle`, first extract all positions into three arrays `xs`,`ys`,`zs` of `n doubles` each, such that e.g. `xs[i]` contains the x part of the position of particle `i`. Do the same thing for masses and velocities, giving seven arrays in total. Then in the main loops, read from these arrays instead of the `struct particle` array. Consider why this is faster. Hint: the elements of a `struct` are laid out in memory next to each other.

Implementing these two improvements properly is enough to get full marks in the practical part (as long as the other requirements are also fulfilled), but you can also try to optimise your implementation in other ways, if you can think of any. Make sure to present *speedup* of your optimised implementation(s) versus the un-optimised one.

Your tasks for 3.3 are:

- Implement one or more optimizations, either your own idea or based on the suggestions above.
- Compute the speedup of your optimized program, compared to your original C-based implementation.

4 Optional visualization

Note: the visualization is entirely optional and can be ignored.

The movements of particles makes for nice visualizations, and looks more impressive the more particles you can simulate. For this reason, the code handout contains three files for creating animations: `animate.sh`, `animate.py`, and `animate.html`.

The start script is named `animate.sh` and is set up to invoke `nbody.py` 100 times, creating data files for 100 frames. After the runs are completed, it will invoke `animate.py` to generate an animated GIF named `animated.gif`. If you do not have a viewer that supports displaying animated GIFs, you can open `animate.html` and view the animation in your browser.

When your code is implemented in C, and can run much faster, you can edit `animate.sh` and make it call your optimized binary by editing the `COMMAND=` line.

You need to install the Python package `array2gif` for the animation to work. This done with the command `pip3 install array2gif`.

5 The code handout

`nbody.py`: A functionally correct implementation of N-body simulation in Python.

`nbody.c`: The start of an N-body simulation in C. Same interface as `nbody.py`, only the input/output parts are implemented.

`util.h`: Various utility definitions used by `nbody.c`. You do not need to modify this file, but you are allowed to.

`genparticles.py`: A Python program that generates random data files. You don't need to modify or understand this program.

`5_bodies.txt`: A data file with 5 lines.

`Makefile`: Contains two useful targets:

`make`
: Compiles `nbody`. Add more programs to the `EXECUTABLES` variable in `Makefile` to compile these as well.

`make` n`_bodies.txt`
: Produce a random data file containing n lines, each representing a body. This is how you generate random test data for your programs.

`animate.sh`: An optional helper script for generating animations.

`animate.py`: An optional helper script for creating animated GIFs from datafiles.

`animate.html`: An optional helper webpage for viewing the animated GIF.