Exam in HPPS

January 15-17, 2024

Preamble

This document consists of 12 pages including this preamble; make sure you have them all. Your solution is expected to consist of a *short* report in PDF format, as well as a `.zip` or `.tar.gz` archive containing your source code, such that it can be compiled immediately with `make` (i.e. include the non-modified parts of the handout as well).

You are expected to upload *two* files in total for the entire exam. The report must have the specific structure outlined in section 3.

- The exam is *strictly individual*. You are not allowed to communicate with others about the exam in any way.
- If you believe there is an ambiguity or error in the exam text, *contact one of the teachers*. If you are right, we will announce a correction.
- Your final grade is based on an overall evaluation of your solution, but *as a guiding principle* each task and question is weighted equally.

Make sure to read the entire text before starting your work. There are useful hints at the end.

1 The problem: N-body simulation

N-body simulations are used in physics and related fields to simulate how particles interact through physical forces. One of the traditional applications is simulating the gravitational interaction of stellar bodies. In some problems only a few bodies are simulated (for example the planets in the solar system), but in general we can be simulating an enormous number of particles. In this exam we will be focusing on the latter case: simulating the behaviour of thousands of particles in a three-dimensional space over time.

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 write

$$||\vec{p_i} - \vec{p_j}|| = \sqrt{(p_i.x - p_j.x)^2 + (p_i.y - p_j.y)^2 + (p_i.z - p_j.z)^2}$$

for the Euclidean distance between two points $\vec{p_i}, \vec{p_j}$, where $p_i.x$ is the x component of the point p_i.

We can compute the *acceleration* $\vec{a_i}$ of particle i as affected by all other particles with the formula

$$\vec{a_i} = \sum_{j \neq i}^{N} \text{force}(\vec{p_i}, \vec{p_j}, \vec{m_j})$$

where

$$\text{force}(\vec{p_j}, \vec{p_j}, \vec{m_j}) = \frac{m_j(\vec{p_j} - \vec{p_i})}{(||\vec{p_j} - \vec{p_i}||^2 + \epsilon^2)^{3/2}}$$

and ϵ is a softening constant used to avoid excessive interactions between particles that are extremely close.

After computing the acceleration, the velocity v_i can be updated with

$$\vec{v_i} \leftarrow \vec{v_i} + \vec{a_i}$$

and then finally, once all velocities have been computed,

$$\vec{p_i} \leftarrow \vec{p_i} + \vec{v_i}$$

This is repeated for some number of steps to simulate the progress of time. As computing the acceleration for a single particle involves looking at all n particles, the total number of interactions computed is $O(n^2)$. We call this the *naive* algorithm.

1.1 Particle files

The state of an N-body simulation is represented by the positions, masses, and velocities of its particles. We store this as a binary data file that has the following format.

- First, a 32-bit integer n indicating the number of particles in the file.
- Then n particles, each represented as 7 double precision numbers (8 bytes each) in this order:
  - The mass.
  - The x, y, z coordinates.
  - The x, y, z velocities.

Each particle thus takes up 56 bytes, and a particle file storing n particles takes up $4 + 56n$ bytes.

You may assume that all numbers are stored in native byte order (little endian, on all machines you realistically have access to).

The code handout contains a program `genparticles` that can generate particle files, and `particles2text` that can show particle files as text. These will work once you have implemented Task 1. See their source code or section 6 for usage examples.

1.2 The Barnes-Hut Algorithm

The quadratic complexity of the naive N-body algorithm makes it unsuitable for simulating systems with many particles. A Barnes-Hut simulation is an *approximate* algorithm that addresses this issue by organising the space in an *octree*, a spatial tree structure. When we then compute the gravitational influence on a given particle i, we only look at individual particles that are *close* to i (determined by a parameter θ, see below), and treat groups of distant particles as a single aggregate particle located at their centre of mass. This means that a Barnes-Hut simulation does not arrive at the exact same result as a naive simulation, but in practice it is close. For sensible definitions of "close", Barnes-Hut requires only $O(n \log n)$ particle interactions.

The main challenge in understanding (and implementing) Barnes-Hut is grasping the octree data structure.

1.2.1 Octree

An octree is similar to a k-d tree in that it describes a recursive partitioning of a space. It is different in that it is specialised to just three dimensions[1], and that whenever we split the space, we split it evenly along all three dimensions, rather than along a point. In a k-d-tree, each node stores a point, but in an octree, only (some) leaf nodes do. Each node of an octree always represents a cube-shaped area of space, with none of the cubes overlapping.

An octree is a tree where each *internal node* (non-leaf) has exactly eight children, each covering an eight of the space of their parent, and all with identical volume. Such subdivisions are called *octants*. Figure 1 shows an octree with three levels. The first level contains an internal node that represents the entire space. On the next level, each of its eight children, represent an eight of the total space. Six of the nodes on the second level are *external nodes*, and have no further children. The remaining two nodes are internal nodes, and thus have eight children each. All of the nodes on the third level are external nodes.

When using an octree to represent particles in a space, we maintain the invariant that external nodes contain *at most* one particle (but may contain zero). Assuming no two particles occupy the *exact* same position, this can always be done by making the octree sufficiently fine grained, by splitting nodes as needed.

We number the children of an internal node as shown on fig. 2.

Each node in the octree contains a *corner coordinate* that indicates the position of the lowest corner, as well as the *edge length*. Returning to fig. 1, suppose the root node has a corner coordinate of $(0, 0, 0)$ and a length 1. This means the root node represents a unit cube centered at $(0.5, 0.5, 0.5)$[2]. It eight children would then all have length 0.5, and with the following corners:

[1]The two-dimensional version is called a *quadtree*.

[2]In some treatments, the root node of an octree is *unbounded*, but for simplicity we treat it like any other node.

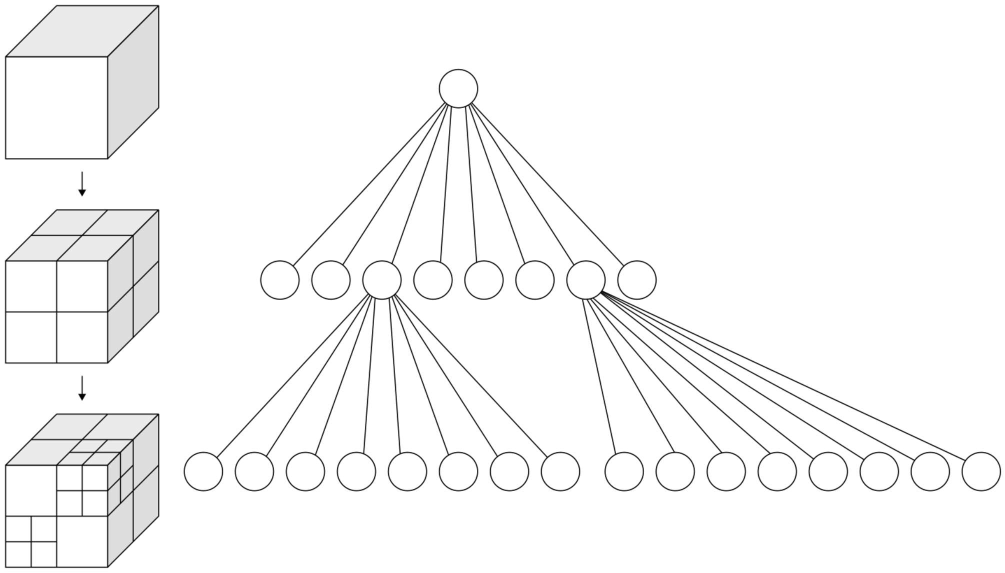

Figure 1: Visualisation of octree. Image by WhiteTimberfolf, original file at `https://commons.wikimedia.org/wiki/File:Octree2.svg`.

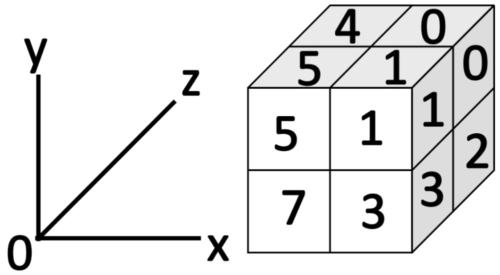

Figure 2: Numbering the children of an internal node from $0-7$. The child with the highest (x, y, z) coordinates is 0, and the one with the lowest coordinates is 7.

0: $(0.5, 0.5, 0.5)$	4: $(0.0, 0.5, 0.5)$
1: $(0.5, 0.5, 0.0)$	5: $(0.0, 0.5, 0.0)$
2: $(0.5, 0.0, 0.5)$	6: $(0.0, 0.0, 0.5)$
3: $(0.5, 0.0, 0.0)$	7: $(0.0, 0.0, 0.0)$

An external node x contains a field *x.particle* indicating the particle (if any) stored in that node.

An internal node x contains a vector field *x.com* indicating its centre of mass, and a field *x.mass* storing the sum mass of all particles in the children of the node.

1.2.2 Inserting a particle

```
Procedure insert(node, p)
if node is internal then
    child ← child of node that p belongs in;
    insert (child, p);
    Update centre of mass for node;
else if node contains no particle then
    node.particle ← p
else
    cur ← node.particle;
    Convert node to internal node;
    insert(node, cur);
    insert(node, p);
```

For two particles i, j with masses m_i, m_j and positions $\vec{p_i}, \vec{p_j}$, their total mass is

$$m_i + m_j$$

and their combined centre of mass is

$$\frac{(\vec{p_i} \times m_i) + (\vec{p_j} \times m_j)}{m_i + m_j}$$

You can use this to update the centre of mass of a node, which can be considered as a single large particle.

1.2.3 Computing accelerations

The procedure below computes the gravitational acceleration acting on particle i. It assumes an implicit parameter $\theta \in [0, 1]$ that controls when we stop recursing down the tree. For $\theta = 0$, Barnes-Hut is equivalent to the naive algorithm. The acceleration is accumulated in a global variable $\vec{a}$. In your actual implementation, $\vec{a}$ is a pointer passed to the function.

```
Procedure accel(node, i)
if node is internal then
    d ← $||\mathsf{node}.com - p_\mathsf{i}||$;
    if node.$l$/d $< \theta$ then
        $\vec{a} \leftarrow \vec{a} + \mathrm{force}(\vec{p_\mathsf{i}}, \overrightarrow{\mathsf{node}.com}, \mathsf{node}.mass)$;
    else
        foreach child of node do
            accel(child, i);
else if node contains a particle that is different from i then
    $\vec{a} \leftarrow \vec{a} + \mathrm{force}(\vec{p_\mathsf{i}}, \overrightarrow{p_{\mathsf{node}.p}}, m_{\mathsf{node}.p})$;
```

2 Your tasks

You are given a code handout containing incomplete programs. The programming tasks involve finishing these. Each task lists which files need to be modified. You are not allowed to modify `util.h`. Do *not* print any extra text to standard output. Feel free to add debugging prints during development, but when you hand in your solution, the *only* thing that gets printed to standard output must be the total runtime (which is already part of the handout).

2.1 Task: Particle files

Implement the following functions in `util.c`:

- `read_particles()`
- `write_particles()`

Once this is done, the programs `genparticles` and `particles2text` should work.

2.2 Task: Naive Simulation

Finish the implementation of the naive N-body simulation in `nbody.c`. Parallelise it as well as you can with OpenMP, and make any other optimisations you think are interesting. You may use the helper functions `dist()` and `force()` from `util.c`, but you are not required to do so.

2.3 Task: Barnes-Hut Simulation

Finish the implementation of the Barnes-Hut N-body simulation in `nbody-bh.c`. The program already contains a substantial amount of skeleton code that you can use as a starting point. Make sure to read the embedded comments. Parallelise it as well as you can with OpenMP, and make any other optimisations you think are interesting. You may use the helper function `dist()` from `util.c`, but you are not required to do so.

3 The structure of your report

Do not put your name in your report. The exam is supposed to be anonymous. Your report must be structured exactly as follows.

Introduction.
Briefly mention general concerns, your own estimation of the quality of your solution, whether it is fully functional, which programs you have written or modified in order to answer the questions, and possibly how to run your tests. Make sure to report the computer you are benchmarking on (in particular the core count).

Sections answering the following specific questions.

1. The following questions pertain to `util.c`.
   a) How can `read_particles()` fail?
   b) How can `write_particles()` fail?

c) How much memory is taken up by the result of your implementation of `read_particles()` when it has read a particle file with n particles, and what is the peak size of the memory you have allocated while `read_particles()` is executing?

2. The following questions pertain to `nbody.c`.

   a) Does the work done by the `nbody()` function (including functions it calls) exhibit good temporal and/or spatial locality? Why or why not?

   b) For *every* loop in `nbody()` (you probably have four of them), explain whether it could potentially be parallelised or not, and then explain whether you decided to parallelise it in your implementation, and why.

   c) Measure and show the weak and strong scaling of `nbody` as you vary the number of threads. Explain how you chose the datasets.

3. The following questions pertain to `nbody-bh.c`.

   a) Does the work done by the `nbody()` function (including functions it calls) exhibit good temporal and/or spatial locality? Why or why not?

   b) How did you parallelise your implementation?

   c) Show the speedup of `nbody-bh` relative to `nbody` as you vary the number of particles.

   d) Measure and show the strong scaling of `nbody-bh` as you vary the number of threads. Explain how you chose the datasets.

   e) What makes it challenging to measure the weak scaling of `nbody-bh`?

   f) Find a workload (changing either the number of particles or θ or both) where `nbody` is faster than `nbody-bh`. Explain why.

4. **Bonus questions.** You are not required to answer these in order to get the top grade.

   a) Did you attempt any code optimisations, and did they work out?

b) How could the `struct bh` type be represented more compactly?

c) For n particles, how many nodes does the octree need in the worst case? What use can we make of that?

4 The code handout

The code handout contains the following.

- `util.h`: Prototypes for utility functions. **Do not modify this file.**
- `util.c`: Definitions of utility functions, some of which you have to implement yourself.
- `genparticles.c`: A program for generating random particle files of a given size.
- `particles2text.c`: A program for printing a particle file in a human readable format.
- `nbody.c`: An incomplete implementation of naive N-body simulation.
- `nbody-bh.c`: An incomplete implementation of Barnes-Hut N-body simulation.
- `Makefile`: You may want to modify the `CFLAGS` for debugging and benchmarking. You are also free to add targets for new programs if you wish, or to modify the existing ones.
- `plot.gnu`: Gnuplot script used by `video.sh`.
- `video.sh`: Repeatedly runs `nbody` (or `nbody-bh`) to perform N-body simulation steps and produces a video of the changes in particle positions. Requires that you have installed `gnuplot` and `ffmpeg`.

See also section 6.

5 Advice

Take note of the difference in the point field ordering in particle files compared to the struct.

Since you are not allowed to modify `util.h`, any code you want to share between `nbody.c` and `nbody-bh.c` will unfortunately have to be duplicated.

Start by getting a sequential version to work. Then perform parallelisation. Use your known-working sequential version as a reference. You can compare files with `cmp` (although depending on how you parallelise, you might not get exactly the same results).

For the Barnes-Hut simulation, you may find it useful to draw on a piece of paper how the tree changes when a small number of particles are inserted.

Remember that it is expected that the naive and Barnes-Hut simulations will produce different results.

Start answering questions as soon as you have finished a programming task. Do not wait until you have implemented all tasks.

Even if you don't get your simulations working correctly, as long as your code does not crash, you can still perform the performance analysis requested in section 3.

Speedup graphs are preferable, but if you find it too time consuming to produce graphs, then tables of numbers are sufficient.

6 Usage examples

None of the material in this section is normative with respect to the exam and you do not need to read it, but it may be useful.

You can generate a file `particles` containing 1000 particles like so:

```
$ ./genparticles 1000 particle
```

Note that this may or may not be an appropriate size for testing or benchmarking.

We can print the particles like so:

```
$ ./particles2text particles
0.000008 -0.260472 0.042248 -0.193347 -0.018488 0.009923 -0.014604
0.000008 -0.407767 0.572616 0.425586 0.018990 -0.013190 -0.011699
```

```
0.000006 -0.126822 0.086158 -0.086469 -0.009806 0.001401 0.000567
0.000009 -0.080574 -0.072165 -0.138754 0.019611 -0.000959 -0.021063
...
```

We can perform a single step of the N-body simulation like so:

```
$ $ ./nbody particles particles.out
0.001413
```

This writes the resulting particles to `particles.out`. The terminal output is the simulation runtime in seconds (excluding reading and writing files).

We can perform ten simulation steps like so:

```
$ ./nbody particles particles.out 10
0.009855s
```

If `ffmpeg` and `gnuplot` are installed, we can produce a video visualisation of all the intermediate steps like so:

```
$ sh video.sh particles
```

If you wish, you can modify the `video.sh` script to change the framerate and such. You are not required to make use of this script, so do not spend any time getting it working.

By default, `nbody-bh` uses $\theta = 0.5$, We can pass a different θ as the last option of the command line (note that we must explicitly pass the number of simulation steps first):

```
$ ./nbody-bh particles particles.out 1 0.1
0.003960
```