HPPS 2025

Assignment Five

The N-body Problem

Due: 2026-01-08

Synopsis: Write a parallel implementation to the N-body problem that maintains a list of bodies close to the centre.

1 Introduction

This is the fifth of five assignments in the *High Performance Programming Systems* course. For this assignment you will be parallelising and investigating the N-body problem. Make sure to read the entire text before starting your work. There are useful hints at the end.

2 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 assignment we will be approximating both: simulating the behaviour of thousands of asteroids 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

$$\vec{a_i} = \sum_{j \neq i}^{n-1} \text{force}(\vec{p_i}, \vec{p_j}, \vec{m_j})$$

where

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

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.

2.1 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. See their source code or section 8 for usage examples.

2.2 Asteroids!

The N-body problem is often used to simulate the movement of stellar bodies. Our assignment program is going to aproximate this by pretending that the particles that it is modelling are actually asteroids surrounding a small space craft, conveniently located at the point (0,0,0). In order to alert the ships crew to danger, and asteroids that get too close should generate some sort of warning. For our simulation, this just means that any particles that get sufficently close to the centre should be added to a *warning list*, with this list then saved to a file at the conclusion of the simulation. The term *list* is here used in its informal sense; in fact the representation of the warning list is as an array where each element is of type `struct warning`, as defined in `utils.h`. Once written to a file, the file can been read using the `warnings2text` progam. See their source code or section 8 for usage examples.

2.3 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.

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.

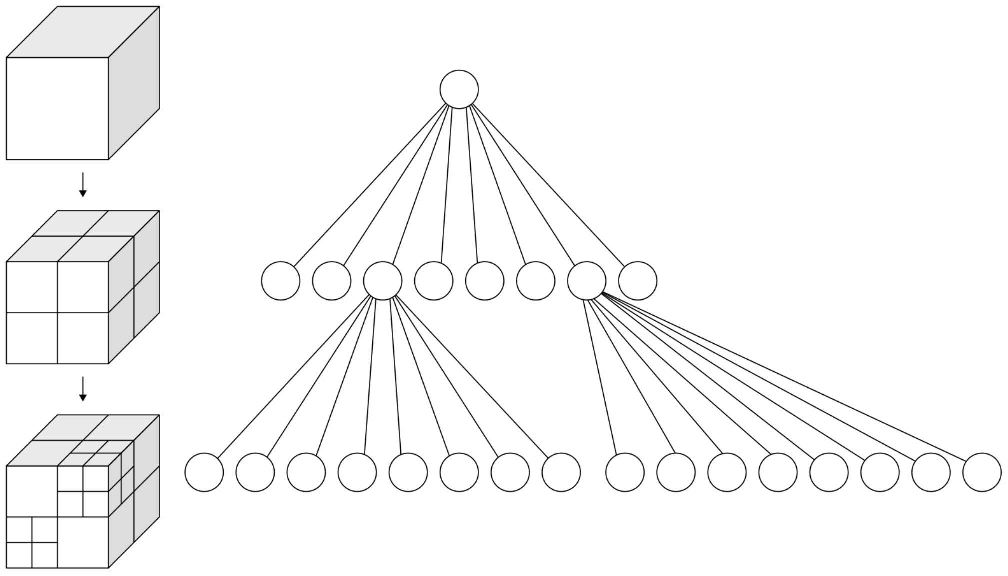
Figure 1: Visualisation of octree. Image by WhiteTimberwolf, original file at `https://commons.wikimedia.org/wiki/File:Octree2.svg`.

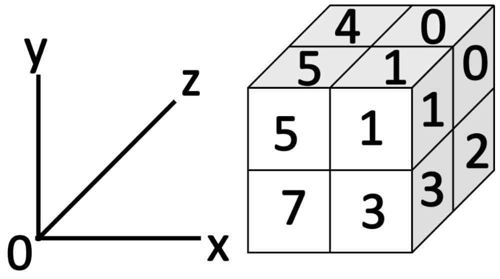

Figure 2: Numbering the children of an internal node from $0 - 7$. The child with the highest (x, y, z) coordinates is 0, and the one with the lowest coordinates is 7.

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.

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.

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

3 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(this is already part of the handout).

3.1 Task: Update warnings list

Modify the naive N-body simulation in `nbody.c` to update the warnings list as discussed in section 2.2. The distance-to-centre threshold is specified as a constant `WARNING_DISTANCE` in the handout code. Use the function `dist_centre` to find the distance. Since you do not know the final size of the warnings array in advance, you will need to expand it during execution - consider using `realloc()` for this purpose.

3.2 Task: Naive Simulation

Parallelise the implementation of the naive N-body simulation in `nbody.c` as well as you can with OpenMP. You should ensure that your solution is free of race conditions, as concerns both the final particle positions and the warning list.

3.3 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. Note that your implementation should still update the warning list with each instance of a particle getting within distance 0.01 of the centre in the same manner as the naive simulation.

4 Using MODI

When you are happy with your design, you can test it on a larger system. UCPH provides access to the MODI queue based system where each node has 256 GiB of memory and two 32-core CPUs. Using the MODI system, you will be able to simulate larger systems than what your laptop can handle.

Note: working with a queue based system means there will be a delay between the submission of your experiment and receiving the result. As we approach the hand-in date, the load on the system will likely grow, increasing the delay. We suggest you take this into account when planning when to run the experiments.

Since each MODI instance has two 32-core CPUs, it would make sense to choose a workload size that works well for one to 64 cores (i.e. if it takes 1 second on a single core, it will not take $\frac{1}{64}$s with 64 cores). Then submit jobs with different thread counts and plot the result times in a speedup graph, where the x-axis is the number of cores, and the y-axis is the time spent.

If you have time for it, you can consider scaling the workload and keeping the number of cores constant, similar to Gustafson-scaling.

Running on MODI

To help you run the code on MODI, the handout contains two helper files, `run_nbody.sh`, and `run_nbody_bh.sh`. These two files are meant to help you run your code on MODI, as it requires bash files to be submitted on slurm. The use of MODI and the helper files is described here:
`https://github.com/diku-dk/hpps-e2025-pub/blob/master/modi-guide/README.md`

And the specifications for MODI itself are here:
`https://erda.dk/public/MODI-user-guide.pdf`.

5 The structure of your report

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 `nbody.c`.
   a) Measure and show the weak and strong scaling of `nbody` as you vary the number of threads. Explain how you chose the datasets.
   b) Determine (in whatever way you find best) the performance overhead of updating the warning list.

c) Are there workloads for which updating the warning list can have a ruinous impact on parallel scalability, and why?

d) Do you have any memory leaks, and how can you know?

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

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

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

   c) Do you have any memory leaks, and how can you know?

It is up to you to decide on proper workloads (i.e. input sizes) that can provide good answers to these questions. Make sure to report the workloads you use.

All else being equal, **a short report is a good report**.

6 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, you should not need to modify this file.
- `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.
- `warnings2text.c`: A program for printing a warnings file in a human readable format.
- `nbody.c`: A complete implementation of naive N-body simulation, in need of some parallelisation.

- `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 8.

7 Advice

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.

You may wish to keep a backup of the sequential version in order to compare your parralel implementation against. You can compare the output 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, both in the particle positions and warnings (though they should be close-ish depending on how long you run for).

8 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 particles
```

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 warnings.out
0.001413
```

This writes the resulting particles to `particles.out` and the warning list to `warnings.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 warnings.out 10
0.009855s
```

The warnings (if any) can be listed with `warnings2text`:

```
$ ./warnings2text warnings.out
37 935
39 968
39 732
40 340
...
```

Each line lists the time step at which the warning occurred, followed by an integer identifying the particle.

Note that you will probably have to run at least 100 simulation steps in order for any particles to reach the centre (and thus produce warnings).

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 warnings.out 1 0.1
0.003960
```