Parallel computing course assignment solution: Parallel N-Body simulation in C++ with MPI

Marko Grönroos

November 12, 2025

1 Introduction

I present here parallel (in addition to non-parallel) solution to the N-body simulation problem. The basic Newton's motion equations describing N-body system are:

$$\vec{F}_a = m_a G \sum_{b=0, b\neq a}^{N-1} m_b \left(\frac{\vec{x}_b - \vec{x}_a}{||\vec{x}_b - \vec{x}_a||^3} \right) \tag{1}$$

$$\vec{a}_a = \frac{\vec{F}_a}{m_a} \tag{2}$$

$$\vec{v}_a(t+h) = \vec{v}_a(t) + h \cdot \vec{a}_a$$

$$\vec{x}_a(t+h) = \vec{x}_a(t) + h \cdot \vec{v}_a(t+h) \tag{3}$$

where $\vec{F}$ is a force vector that describes all the forces affecting a body, m is a mass (in kg) of a body, G is the gravitational constant $G = 6.67259 \cdot 10^{-11}$, $\vec{x}$ is a position vector of a body, $||\vec{x}_b - \vec{x}_a||$ is the euclidean distance between two bodies, $\vec{a}$ is the acceleration vector of a body, t is time, and h is the time step of the iteration.

The source code and documentation are available from `http://iki.fi/magi/opinnot/mpi/`.

2 Object oriented design

The N-body system is abstracted in the class *NBody* which is a system containing bodies of class *Body*. The basic class relationships are shown in Figure 1, with some of the central member attributes and methods of the classes.

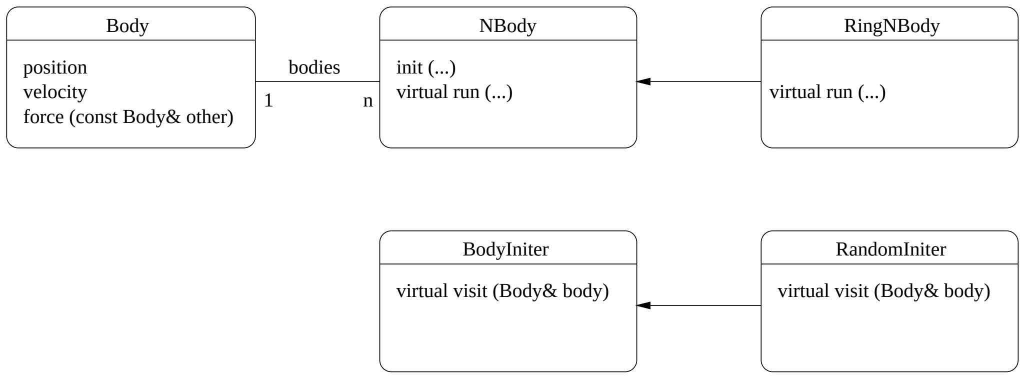

Figure 1: Class hierarchy.

2.1 Class *Body*

The abstraction for bodies is class *Body*. The attributes of the bodies are position and velocity, represented in two-dimensional vectors. For vectors, I use the *Coord2D* class from MagiClib. Using three-dimensional coordinates would be straight-forward, by replacing the *Coord2D* with *Coord3D*, which is an identical class with additional dimension.

2.2 Class *NBody*

The *NBody* class presents a non-parallel solution to the N-body problem. Parallelization of the N-body problem is an algorithmic specialization, and thus implemented by inheritance with the *RingNBody* class, as explained below.

2.3 Initialization framework

Initialization of the system is done by initializer objects that visit bodies to initialize them. The initializers inherit the baseclass *BodyIniter* and implement its *visit()* method. The initializer can use the body index number parameter if it wants to.

I implemented two initializers, the *RandomIniter* and *PresetIniter*.

RandomIniter initializes the bodies with evenly distributed random values in domain given in parameters.

PresetIniter initializes the bodies with parameters given in the parameter file for the application.

2.4 Graphics

The N-body system is visualized using MPE graphics window. The diameter of each body is calculated from its mass M, according to the density parameter ρ, as follows:

$$V = \frac{1000M}{\rho} = \frac{4}{3}\pi r^3 \Leftrightarrow r = \left(\frac{3000M}{4\rho\pi}\right)^{\frac{1}{3}},$$

where V is the volume of the body, in m^3. Default density is $\rho = 5.0 \cdot 10^3 kgm^{-3}$, roughly the density of Earth and most other solid planets.

2.5 Simulation features

When two bodies go very near each other, the iterative nature of the algorithm causes large errors in the computation. Typically a body jumps very near another body, and causes a very strong force between the bodies. This causes the both bodies to accelerate to very high speeds. We reduce this problem by ignoring the gravitational force if the distance between the bodies is very small.

Random numbers can not be initialized by using the *time()* function, because if two or more processes start during the same second, their random value generators will be initialized identically. The simulator calculates the random generator seed from current time and process id.

3 Parallelization

I use the parallel ring algorithm, as described in the lectures[1]. The parallel algorithm is implemented by inheriting the *NBody* class and redefining the *run()* method.

[1] I read only the lecture notes, and didn't fully understand the presented description of the ring algorithm. I hope that I understood the basic idea correctly.

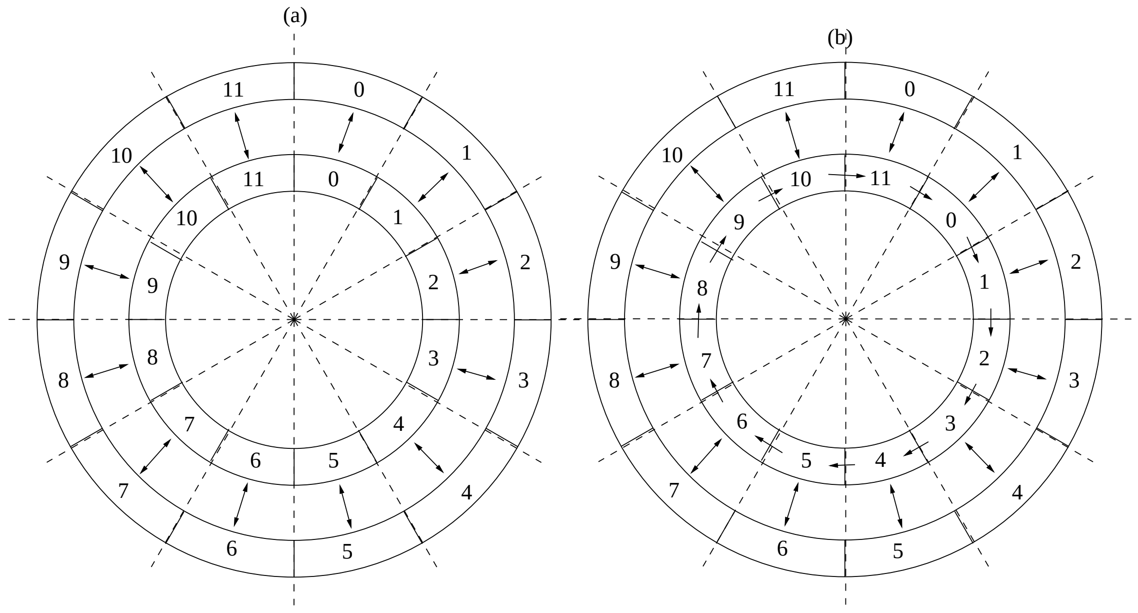

Figure 2: Shifting data in the process rings.

The basic idea is that each process has a set of *local* bodies, owned by that process. The local bodies of each process are duplicated to a set of *circulating* bodies. The circulating bodies are transferred through the ring and interacted with the *resident* (local) bodies at each process. After the circulating bodies have travelled through all processes and returned back to the original process, the symmetric forces accumulated in the circulating copies are added to the forces accumulated in the resident bodies. The resident and circulating sets of body objects thus form two interacting rings that are rotated one full revolution during each iteration step. These rings are illustrated in Figure 2, where the outer ring represents the resident bodies and the inner ring the circulating bodies. The physical processes are represented as sectors separated by dotted lines.

In the beginning of an iteration, after we have copied the local bodies to the resident bodies, we first calculate forces between these two sets, as shown in Figure 2a . We add the force vector affecting each body to the data of that body, both in the resident and circulating bodies.

Then we rotate the inner ring by transmitting the circulating bodies data to the next process in the ring, as shown in Figure 2a, and then compute the forces between the resident and received circulating bodies. We continue this until the circulating set has travelled full circle around the ring and returned back to its original process. We can then update the velocities of the local bodies (both resident and circulating), and proceed to the next iteration cycle.

This basic idea can be seen most clearly in the main loop of the *run()*-method of the class *RingNBody*:

```
// Create a double-buffered container for circulating bodies
Array<PackArray<Body> > circulatingBodies;
circulatingBodies.add (new PackArray<Body> (mBodies.size)); // Create buffer #0
circulatingBodies.add (new PackArray<Body> (mBodies.size)); // Create buffer #1

int ringSize = mrMPI.world().size();
for (int iter=0; iter<iters; iter++) {
    // Update positions of the bodies. Use ½h on the first
    // iteration, to implement leapfrog method.
    updatePositions (iter? h:h/2, !(iter%updateFreq), plotCoords);

    resetForces ();

    // Copy the local bodies into circulating bodies.
```

```
        circulatingBodies[0].shallowCopy (mBodies);

        for (int i=0; i<ringSize; i++) {
            // Calculate forces diagonally between resident and
            // circulating bodies. On the step=0, the circulating
            // bodies are local.
            calculateForces (mBodies, circulatingBodies[i%2], true);

            // Send circulating bodies forward in the ring
            mrMPI.world().nbSend (circulatingBodies[i%2].data, 1, mpBodyVectorType->getType(), mNext);

            // Receive circulating bodies from previous node in the
            // ring. In the last step, we receive back the local
            // circulating bodies, which we sent out in the step=0.
            mrMPI.world().recv (circulatingBodies[(i+1)%2].data, 1, mpBodyVectorType->getType(), mPrev);
        }

        // Add the forces from the circulated local bodies to resident
        // bodies
        for (int i=0; i<mBodies.size; i++)
            mBodies[i].addForce (circulatingBodies[ringSize%2][i].totalForce());

        // Update velocities of the bodies
        updateVelocities (h);
    }
```

The algorithm uses double-buffering of the circulating bodies, because it uses non-blocking send method to shift the bodies forward in the process ring.

We use the symmetric calculation of forces to avoid unnecessary duplication of calculations. The symmetric calculation between two sets of bodies is done as follows:

```
    void NBody::calculateForces (PackArray<Body>& a, PackArray<Body>& b, bool diagonal) {
        float r;
        for (int i=0; i<a.size; i++) {
            // Iterate only half of the a*b matrix
            for (int j=diagonal? i+1:0; j<b.size; j++) {
                // Compute force
                Coord force = a[i].force (b[j], r);

                // Forces apply only if the bodies are further than the
                // minimum distance
                if (r>mMinR) {
                    a[i].addForce (force);
                    b[j].addForce (-force); // Symmetric force
                }
            }
        }
    }
```

Here the set of bodies *a* is a resident, and *b* is circulating. If the parameter `diagonal` is true, the inner loop is done from *i+1*. The symmetric calculations are illustrated in Figure 3.

The algorithm does not transmit all the data of the *Body* objects, only the positions, forces, and masses. This requires a bit dirty tweaking of the vectors with the MPI data types, and relies strongly on the order of the attributes within the *Body* class, and that the class nor its attributes have a virtual table. The solution is definitely not very clean object-oriented programming, but transmitting the object data otherwise would be much less efficient. Using a more clean solution might have required giving up the object-oriented approach altogether.

4 Experiments

I tested the program with three N-body systems. Two tests were with real planetary bodies, and one with totally random bodies. The simulations with planetary bodies could only be done with

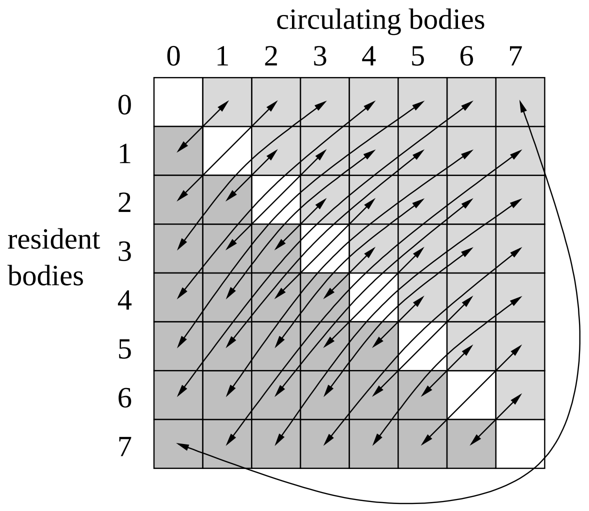

Figure 3: Symmetric forces between resident and circulating bodies

the non-parallel program, because the number of bodies was so small, 2 and 5. The real-world tests were good for verifying the correctness of the physics calculations.

4.1 Earth-Moon system

For experiments with Earth and Moon, I used the average distance of the Moon from Earth, $r = 384400km$. An approximate initial velocity can be calculated from $v = 2\pi r/d/24/3600$, where d is the length of month, about $d \approx 30$. Thus, $v = 931m/s$. It should be obvious that these are *very* approximate values. The experiment-specific parts of `nbody.cfg` file are as follows:

```
[]
initer=PresetIniter
viewCenter.r   =400E6
[NBody]
n              =2
[PresetIniter]
body0          =0,0,0,-11.45,5.97E24
body1          =384400000,0,0,931,7.348E22
```

I gave Earth a small velocity upwards ($11.45m/s$), because Moon's initial velocity causes a slow drift downwards also for earth, which would cause the system to travel off the screen.

Figure 4 shows a simulation of the the Earth-Moon-system, with $h = 60 * 60 * 24 = 86400s$ (one day). The picture on the left shows about one month, while the picture on the right shows a period of roughly 1000 years. Earth (really about three pixels in diameter) wobbles very little in the center. The orbital ellipse clearly rotates over time, which should not be possible with non-relativistic two-body system. Thus we can conclude the rotation to be a result of calculation errors. A smaller time step would reduce the error.

4.2 Inner planets

I made also tests with some planets, as their approximate parameters are easily available. The application parameters for these tests are as follows:

```
[]
initer         =PresetIniter
viewCenter.r   =200E9
[NBody]
```

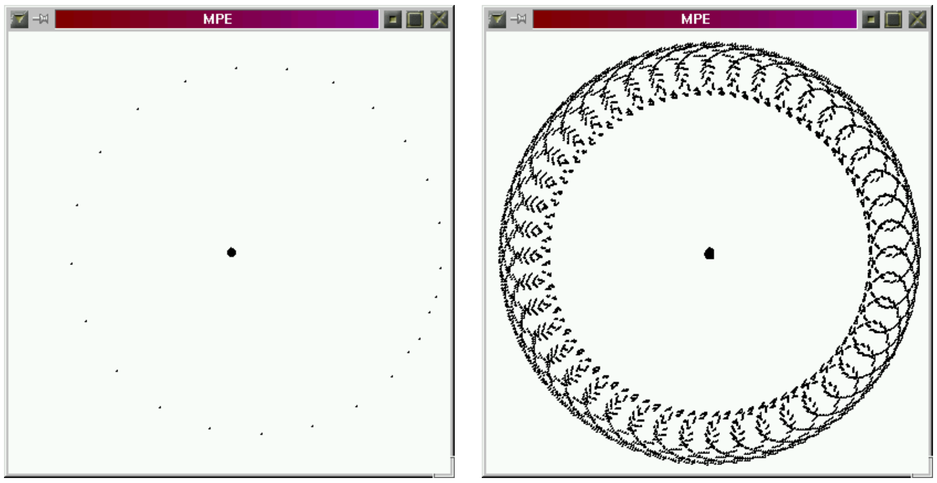

Figure 4: Earth and Moon

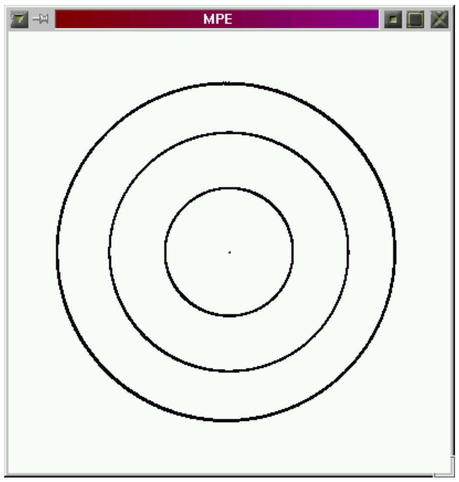

Figure 5: Orbits of Mercurius, Venus and Earth around the Sun.

```
n               =5
[PresetIniter]
body0           =0,0,0,0,1.989E30
body1           =57.9E9,0,0,47846,3.3E23
body2           =108E9,0,0,35061,4.87E24
body3           =150000000000,0,0,30000,5.97E24
body4           =150384400000,0,0,30931,7.348E22
```

Figure5 shows a test run done with a setting that does not undraw the positions of the bodies. Time step is one day, and the system was simulated over several years. As can be seen, the orbits are as they should be. Moon's orbit is not visible in the picture, as one pixel in the figure corresponds to million kilometers, and the radius of Moon's orbit is less than half of that.

More realistic initial parameters for the planets could be easily calculated, but the calculations could not in any way compete with more advanced planetary theories.

Notes For some reason, when using very high masses (around $10^{30}kg$), the velocity values sometimes go outside the value range of *float* and *double* values. This is somewhat strange, since their value range should be well larger than $\pm10^{100}$, and even multiplying $10^{30} \cdot 10^{30}$ (in force calculation) gives only 10^{60}. I did not make tests with *long double*.

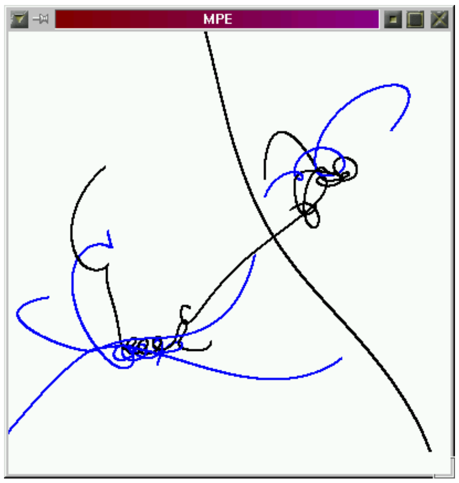

Figure 6: Random orbits with parallel solution.

4.3 Random system

The parallel solution was tested with a random system with two processes and four bodies in each process. The value range of the random initial positions was $[(-1,-1),(1,1)]$ (a square of 2x2 meters), initial velocities were $0.1mm/s$ in random direction, minimum gravitation effect radius was $1cm$. Mass of all the bodies was $100kg$.

One test run is shown in Figure 6. Bodies handled by process 0 are shown in blue (or gray), bodies by process 1 in black. The bodies escaped rather quickly from the view. I can not find any other reason for this behaviour but the accumulation of energy from calculation errors of the iterative simulation.

5 Summary

The parallel solution with ring algorithm worked fine. I did not make any measurements about the efficiency of the parallelization, because I used only one physical processor to make the tests.