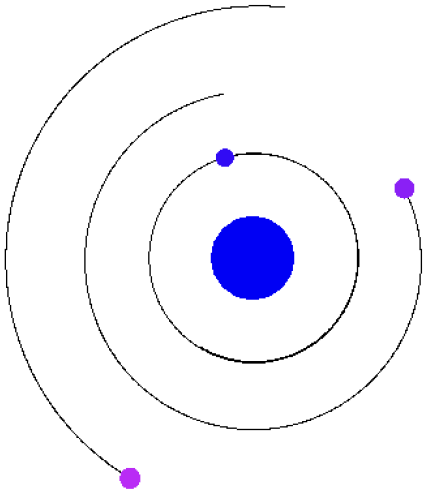
POLITECNICO
MILANO 1863

A Highly Scalable and Parallel N-Body Simulation

Advanced Methods for Scientific Computing



Malak Matar, Stanislaw Ostyk-Narbutt, Aldas Lenkšas

High Performance Computing
January 2026

Abstract

The N-body problem arises in physics and scientific computing studying the interaction of a large number of bodies through coupled forces. This project presents an implementation for a scalable N-body simulation with strong focus on parallelism and efficiency. Multiple solvers are implemented including a serial solver acting like a baseline to compare with the MPI and OpenMP parallel algorithms. In addition, same implementations but with reduced formulation that exploit force symmetry were also added to lower the computational cost, and the state-of-the-art Barnes-Hut simulation. Both gravitational and Coulomb forces are supported with a generic force function allowing even further extensions. The implementation also allows interchangeable numerical integrators including Euler, sympathetic Euler, Velocity Verlet, and fourth-order Runge-Kutta. This allows the study of stability and performance of each. Both fixed and adaptive time stepping were tested. For adaptive, the timestep adjusts dynamically based on the particle's motion to improve stability. A robust OpenGL visualization tool which allows for full interactivity, including movement, zoom, and simulation control is provided. It supports 3D rendering of even a million particles using highly-parallelized GPU instancing and other parallelization optimizations.

Contents

1 Introduction

The N-body problem refers to the task of predicting the time evolution of a system of N interacting bodies subject to physical forces such as gravitation or coulomb forces. Unlike the two-body problem, which admits closed-form analytical solutions, the general N-body problem exhibits complex and often chaotic dynamics due to the simultaneous interaction of each body with all others. As a result, numerical simulation is the primary tool for studying such systems.

This project implements and compares several approaches to the N-body problem with a large level of abstraction in the choice of algorithm, integration scheme, and generic force. Moreover, it includes sequential algorithms and parallel algorithms based on the Message Passing Interface (MPI). Both basic and reduced formulations of force computation are considered. In addition, a robust visualization tool based on OpenGL and GLFW is provided to display simulation outputs, tailored to high performance workloads such as rendering over 100,000 particles simultaneously. The visualization component operates on generated data files, enabling post-processing and potential offloading of simulations to cluster environments.

2 Problem Description

Formally, the problem consists of n interacting bodies, each having its mass m_i, initial position $\vec{s_i}(0)$ and initial velocity $\vec{v_i}(0)$. The task is to simulate how these bodies develop over time, i.e. to find their new positions $\vec{s_i}(t)$ and velocities $\vec{v_i}(t)$ at certain times. An example of such is the movement of stars and planets in a galaxy, or interaction and movement of the atoms in a small-scale system.

The whole simulation depends on the interaction between the bodies which can be described by the forces $\vec{f_{i,j}}(t)$, which denotes the force exerted on i-th body by j-th body. In application such as planet movement simulation the force used would be the one given by Newton's law of universal gravitation:

$$\vec{f_{i,j}}(t) = -G\frac{m_i m_j}{|\vec{s_i}(t) - \vec{s_j}(t)|^3}\left(\vec{s_i}(t) - \vec{s_j}(t)\right)$$

The total force on any body is then given by a sum of such forces:

$$\vec{F_i}(t) = \sum_{j=1, j\neq i}^{n} \vec{f_{i,j}}(t)$$

Newton's second law of motion provides differential equations which can be used to get new positions and velocities:

$$\begin{cases} \vec{F_i}(t) = m_i \vec{a_i}(t) \\ \vec{a_i}(t) = \frac{d}{dt}\vec{v_i}(t) \\ \vec{v_i}(t) = \frac{d}{dt}\vec{s_i}(t) \end{cases}$$

Mathematically this would be enough. However, note that a numerical solver for N-Body problem advances in time using small timesteps Δt, and finds positions and velocities for times $t = \Delta t, 2\Delta t, ..., S\Delta t$, where S is the number of steps for the solver to take. Alternatively, the solver can also advance in time with steps of Δt, that can be dynamically adjusted to improve the accuracy of the solver, until a certain final time T is reached.

3 Abstraction-Centered Software Design

A series of algorithms are developed (each discussed in Chapter 4, with both basic and reduced version in serial and parallel form. Crucially, all algorithms abstract away the integrator being used to advanced velocity and position, and abstract the force. This allows seamless switching between using gravitational forces to Coulomb forces, and testing each with different integrators for advancing the time step discussed in Chapter 5.

Underlying all algorithms is the general model where each body i is characterized by its mass m_i, position vector $\mathbf{s}_i(t) \in \mathbb{R}^d$, and velocity vector $\mathbf{v}_i(t) \in \mathbb{R}^d$ where d denotes the dimension. The system was built to be dimension-independent, controllable via the `dim` constant on which the solver is templated.

3.1 Force Abstraction

The force abstraction allows for different types of forces to be calculated independent of the algorithm by simply extending the generic force and implementing a new calculation that can depend on various attributes, as shown in the code snippet below.

```
1 template <int DIM, typename Attributes>
2 struct force {
3 public:
4   virtual Vec<DIM> operator()(const body<DIM, Attributes>
    ↪  &subjectBody,
5                               const body<DIM, Attributes>
                                ↪  &exertingBody) const = 0;
6 };
```

For example, rather than gravitation, charged particles can be considered under forces governed by the Coulomb law. The change is a simple extension of the force and demonstrates the high level of abstraction and flexibility of the force function:

```
struct charge {
  double charge;
};

template <int DIM> struct coulomb : public force<DIM, charge> {

public:
  static constexpr double k = 8.987551785972e9;

  Vec<DIM> operator()(const body<DIM, charge> &subjectBody,
                      const body<DIM, charge> &exertingBody) const
                       ↪ override {
    ...
    double coeff = -k * subjectBody.attributes().charge *
                  exertingBody.attributes().charge /
                  (dist2 * std::sqrt(dist2));
    ...
  }
};
```

This can be extended to virtually any type of force, making this simulation tool highly versatile.

3.2 Integrator abstraction

The integrators are meant for time stepping one step in time, in particular a small Δt, or in code marked as `dt`. The abstraction is shown in the code snippet below.

```
template <int DIM, typename Attributes>
class Integrator {
public:
  virtual void step(Bodies<DIM, Attributes> &bodies, double dt) =
   ↪ 0;
};
```

This can be extended easily to implement any type of integrators which are discussed in Chapter 5. As an example the Euler method is presented in the code snippet below.

```
template <int DIM, typename Attributes>
class Euler : public Integrator<DIM, Attributes> {
private:
```

```
4     AccelerationAccumulator<DIM, Attributes>&
      ↪ accelerationAccumulator;
5
6   public:
7     Euler(AccelerationAccumulator<DIM, Attributes>&
      ↪ accelerationAccumulator)
8         : accelerationAccumulator(accelerationAccumulator) {}
9
10    void step(Bodies<DIM, Attributes> &bodies, double dt) override {
11
12      accelerationAccumulator.compute(bodies);
13      size_t n = bodies.localSize();
14
15      OMP_STATIC_LOOP
16      for (size_t i = 0; i < n; ++i) {
17        bodies.local(i).pos() += bodies.local(i).vel() * dt;
18      }
19
20      OMP_STATIC_LOOP
21      for (size_t i = 0; i < n; ++i) {
22        bodies.local(i).vel() += accelerationAccumulator.accel(i) *
          ↪ dt;
23      }
24    }
25  };
```

Any of the other integrators can be implemented in a similar manner, making this type of abstraction highly flexible and easy-to-use.

3.3 Solver abstraction

Our project implements and analyzes several different solvers for the N-Body Simulation. It is therefore highly important to have an abstraction for the solver, which can be extended to implement specific solvers. The simplified code is presented in the snippet below.

```
1   template <int DIM, typename Attributes>
2   class NBodySolver {
3   public:
4       virtual void init(int argc, char **argv, const std::string
        ↪ &filename) {...}
5
6       virtual void finalize() {}
```

```
7
8       virtual void runSimulation(...) {
9           Timer t;
10          t.start();
11
12          for (int step = 0; step < steps; step++) {
13              integrator.step(bodies, dt);
14              oneStep();
15              // ...
16          }
17
18          t.end();
19          t.print();
20
21          std::string outputFinalFilename = outputFilePref +
            ↪  ".finalTimestep.out";
22          saveOutput(outputFinalFilename);
23
24          finalize();
25      }
26  };
```

Using such an abstraction, solvers discussed in Chapter 4 have been implemented. The functions such as `NBodySolver::init` and `NBodySolver::finalize` serve as a way to initialize and finalize required variables for the simulation. In particular, in the MPI solvers, MPI initialization and finalization are called during these functions, as well as initial required communication is performed.

3.4 Build System

The project is organized into two main components: simulation executables and an optional visualization executable. The build system is based on CMake and exposes two configuration options. The first option enables or disables the compilation of simulation targets, while the second controls the compilation of visualization targets. As the solvers have OpenMP parallelization implemented, it is possible to turn it on or off by using the USE_OPENMP flag when running CMake script.

The visualization component depends on external libraries that are included as git submodules. These dependencies must be initialized prior to building the visualization executable. Since OpenGL is not supported in the laboratory container environment, visualization is intended to be built and executed into a different build folder outside of the container used for the course environment.

4 N-Body Solver Algorithms

Several different solvers have been implemented: Serial (both Basic and Reduced versions), OpenMP based (both Basic and Reduced versions), MPI based (both Basic and Reduced versions), and Barnes Hut. This chapter describes each of these solvers.

The solver logic is separated from the time advancing logic, which is handled by *integrators* (see Chapter 5). This means that the role of an N-Body solver is to basically calculate forces (or, accelerations, as this is equivalent) for each particle at the current step.

4.1 Serial Solvers

The *Basic Serial Solver* is the most simplest approach to the N-Body problem. The total force for each body is calculated by iterating over each other body and summing up all such contributions. A simple code is presented in the code snippet below.

```
for (size_t i = 0; i < n; i++) {
    for (size_t j = 0; j < n; j++) {
        if (i == j) continue;
        forces[i] += force(bodies[i], bodies[j]);
    }
}
```

To improve this algorithm one can observe that $\vec{f}_{i,j}(t) = -\vec{f}_{j,i}(t)$. This means that each pairwise body interaction needs to be considered only once. Such an algorithm is called *Reduced Serial Solver*, and is implemented as in the code snippet below.

```
for (size_t i = 0; i < n; i++) {
    for (size_t j = i + 1; j < n; j++) {
        Vec<DIM> f = force(bodies[i], bodies[j]);
        forces[i] += force;
        forces[j] -= force;
    }
```

```
}
```

Both Basic and Reduced Serial Solvers have $O(n^2)$ per each simulation step, however Basic Serial Solver computes force $n(n-1) \approx n^2$ times while Reduced Serial Solver only $\frac{n(n-1)}{2} \approx \frac{n^2}{2}$ times, reducing the overall computation time by a factor of 2.

4.2 OpenMP-Based Parallel Implementations

For both Serial Solvers, it is easy to extend them to OpenMP-Based Solvers, as the only implementation step required is to parallelize the outter for loop, as presented in the code snippet below.

```
#pragma omp parallel for schedule(static)
for (size_t i = 0; i < n; i++) {
    for (size_t j = 0; j < n; j++) {
        if (i == j) continue;
        forces[i] += force(bodies[i], bodies[j]);
    }
}
```

This can be done due to the fact that the total force of each particle can be calculated independently. As each inner loop does the same amount of work, static thread schedule is used. Parallelizing the inner loop is not efficient due to the thread creation and deletion overhead.

To parallelize the Reduced Solver, the same parallelization strategy can be applied with the only difference of using dynamic thread schedule instead of static, as the inner for loop is no longer balanced across each outer loop iteration.

4.3 MPI-Based Parallel Implementations

MPI-Based Solvers have been implemented for both Basic and Reduced algorithms using MPI, following the methodology described in Chapter 7 of the Pacheco textbook [3].

4.3.1 Basic MPI-Based Solver

The Basic MPI-Based Solver firstly distributes the particles among MPI nodes using a blocked decomposition. The initial information about bodies is broadcast to all processes using `MPI_Bcast`. Then at each simulation step, the processes gather all positions

and velocities using `MPI_Allgatherv`, but only count total forces for the locally distributed bodies.

Compared to the Basic Serial Solver, the MPI-Based version works P times faster (P is the number of MPI nodes) since the force calculation is divided into P nodes. In particular, each node holds $\frac{n}{P}$ bodies, and at each step calculate the pairwise forces between local bodies and all the global bodies, ending up in complexity of $O\left(\frac{n^2}{P}\right)$ per simulation step.

4.3.2 Reduced MPI-Based Solver

To parallelize Reduced Solver using MPI, a ring-based communication strategy must be utilized. Each MPI node holds data only for its local particles. During each simulation step, the algorithm performs $P - 1$ phases (P is the number of MPI nodes). During each phase, processes exchange particle data with neighboring ranks in a logical ring. Forces are computed incrementally, and local force accumulators are updated accordingly. This approach avoids global all-to-all communication at every step, resulting in improved performance.

Similarly to Basic MPI-Based Solver, the Reduced MPI-Based Solver is expected to be P times faster than the Serial one, as the force calculation is distributed among P nodes. To be more precise, each node stores $\frac{n}{P}$ bodies and during each ring pass phase calculates the interaction between local bodies and the other $\frac{n}{P}$ bodies that were passed from the neighboring process. This leads to $O\left(\frac{n^2}{P^2}\right)$ per each ring pass phase, or in total $O\left(\frac{n^2}{P}\right)$ per simulation step.

4.3.3 Utilizing OpenMP parallelization

Similarly to how OpenMP was utilized for parallelizing Serial solvers, it can be used for MPI-Based Solver as well. In the implementation of the MPI-Based Solver, the force calculation loops can be parallelized. Note that such an implementation is an approach interesting to try, but it does not necessarily benefit from OpenMP parallelization, as performance time can be dominated by MPI communication.

4.4 Barnes-Hut algorithm

The key idea to optimize the $O(n^2)$ force calculation to $O(n \log n)$ is to observe that force calculation can be written as matrix-vector multiplication, and afterwards approximated by low-rank matrices [2]. Barnes-Hut algorithm firstly constructs a tree, and then calculates total force for each body by traversing the tree.

4.4.1 Solver dimensionality

We want to highlight that, similarly to all other N-Body solvers implemented, our implemented Barnes-Hut Solver for N-Body simulation is independent of the dimensionality of the problem. If the simulation needs to be run in 2D universe, the Solver constructs Quadtree, while for 3D an Octree is constructed. Although less practical, 1D (binary tree), 4D, or other dimensions are possible.

4.4.2 Tree construction

The tree (Quadtree for 2D and Octree for 3D) is constructing by inserting bodies one by one. The leaves of the tree are the nodes that have at most one body stored. Moreover, each tree node stores the position of the center of mass of all bodies that are inside the bounds of the node. This center of mass is later needed for approximating forces. Initially, the tree is only the root node containing the maximum bounds for the simulation.

When a body is inserted into the tree, there are three possible cases:

- The current node is not a leaf. This means there are other bodies stored in this subtree, and the current node stores their center of mass. Update center of mass with new node, and traverse recursively into the child node which corresponds to the bounds that would store the new node.
- The current node is a leaf, and contains no nodes. Just insert new node here.
- The current node is a leaf, and already contains a body. In this case, update center of mass in this node, and move both the new body and the old body of the node down the tree.

Each node insertion takes $O(\log n)$ time, resulting in the total complexity of $O(n \log n)$.

Since in the implementation the nodes are created dynamically and only when a new node needs to be added, the second case never happens. This gives some performance speedup. However, the downside of the Barnes-Hut algorithm is that the tree must be reconstructed at every step.

4.4.3 Force calculation

To calculate the total force for each body, the tree is traversed for each body, taking $O(\log n)$ steps. The algorithm is based on the idea that the interaction between near bodies should be calculated pairwise, while the interaction between a body and a cluster of far bodies can be calculated by approximating far bodies by one single body.

When the total force on some body needs to be calculated, the tree is traversed starting from the root in the following fashion:

- In the leaf node, just calculate the force between the required body and the body in the leaf.
- If $\frac{l}{d} < \theta$, the force between the required body and the bodies in the node's subtree is approximated by taking the center of mass in the current node.
- Otherwise, recursively iterate to all children of current node.

Here we used l to denote the length of the Barnes-Hut tree node, and d the distance between the required body and the center of mass in the current node. θ is the parameter to be chosen. $\theta = 0$ means we end up in $O(n^2)$ algorithm, while $\theta \in [0.5, 1.0]$ gives a significant speedup. However, higher theta values give poorer accuracy.

The force calculation per each body is approximately $O(\log n)$, resulting in $O(n \log n)$ overall complexity per each simulation step.

4.5 Utilizing OpenMP parallelization

Only the force calculation part can benefit from parallelization, as the current tree construction approach is not parallelizable. Forces can be calculated for each body independently. However, as the tree construction has to be done at every timestep and in practice takes most of the performance time, the parallelization with the current implementation does not have much effect.

5 Numerical Integrators

Several time integration schemes are implemented to advance particle positions and velocities based on computed accelerations. For each body i, with position $\mathbf{s}_i$ and velocity $\mathbf{v}_i$,

$$\frac{d\mathbf{s}_i}{dt} = \mathbf{v}_i, \qquad \frac{d\mathbf{v}_i}{dt} = \mathbf{a}_i(\mathbf{s}_1, \ldots, \mathbf{s}_N), \tag{5.1}$$

where $\mathbf{a}_i$ is the acceleration induced by the forces from all the other bodies. Each integrator of the ones below advances the state from time t_n to $t_{n+1} = t_n + \Delta t$ using the current $(\mathbf{s}, \mathbf{v})$ and one or more evaluations of $\mathbf{a}$. The main differences between the integrators is in the update order and how many times per step they compute acceleration. For direct $O(N^2)$ force evaluation, the dominating cost is computing the acceleration. Hence, the number of acceleration evaluations per step is useful for runtime. The four integrators were implemented for the purpose of testing and comparing their efficiency.

5.1 Euler Integrator

The explicit Euler method is the simplest as it updates positions using the *old* velocity and then updates velocities using the accelerations evaluated at the beginning of the step:

$$\mathbf{s}^{n+1} = \mathbf{s}^n + \mathbf{v}^n \Delta t, \qquad \mathbf{v}^{n+1} = \mathbf{v}^n + \mathbf{a}^n \Delta t \tag{5.2}$$

In practice, it computes acceleration for all bodies, then loops over the bodies and updates the position and loops again and updates the velocity. Due to its simplicity, it is useful for debugging and as a baseline. However for long simulations it can cause energy drifts.

5.2 Symplectic Euler Integrator

It is very similat to explicit euler but with a change in update order which gives a major improvement. It updates velocity first and then uses the *new* velocity to update positions:

$$\mathbf{v}^{n+1} = \mathbf{v}^n + \mathbf{a}^n \Delta t, \qquad \mathbf{s}^{n+1} = \mathbf{s}^n + \mathbf{v}^{n+1} \Delta t \tag{5.3}$$

It has the same computational cost as Euler but offers more stable energy and keeps orbits bounded for longer.

5.3 Velocity Verlet Integrator

The Velocity Verlet scheme is a second-order, symplectic integrator that splits the velocity update into two half steps and updates position in betweeen.

$$\mathbf{v}^{n+\frac{1}{2}} = \mathbf{v}^n + \frac{1}{2}\,\mathbf{a}^n\,\Delta t, \tag{5.4}$$

$$\mathbf{s}^{n+1} = \mathbf{s}^n + \mathbf{v}^{n+\frac{1}{2}}\,\Delta t, \tag{5.5}$$

$$\mathbf{a}^{n+1} = \mathbf{a}\left(\mathbf{s}^{n+1}\right), \tag{5.6}$$

$$\mathbf{v}^{n+1} = \mathbf{v}^{n+\frac{1}{2}} + \frac{1}{2}\,\mathbf{a}^{n+1}\,\Delta t \tag{5.7}$$

The key improvment is that acceleration changes when positions change by recomputing the acceleration after the position update. The tradeoff is the cost since it requires two acceleration evaluations per time step. But it provides a favorable balance between accuracy, stability, and computational cost and is commonly used as a default integrator in N-body simulations.

5.4 Runge Kutta Fourth Order Integrator

The fourth-order Runge Kutta method (RK4) achieves high accuracy by getting the derivatives at four points within the interval. Let $(\mathbf{s}^n, \mathbf{v}^n)$ be the current state, RK4 computes:

$$k_1^s = \mathbf{v}^n, \qquad k_1^v = \mathbf{a}(\mathbf{s}^n), \tag{5.8}$$

$$k_2^s = \mathbf{v}^n + \frac{\Delta t}{2} k_1^v, \qquad k_2^v = \mathbf{a}\left(\mathbf{s}^n + \frac{\Delta t}{2} k_1^s\right), \tag{5.9}$$

$$k_3^s = \mathbf{v}^n + \frac{\Delta t}{2} k_2^v, \qquad k_3^v = \mathbf{a}\left(\mathbf{s}^n + \frac{\Delta t}{2} k_2^s\right), \tag{5.10}$$

$$k_4^s = \mathbf{v}^n + \Delta t\, k_3^v, \qquad k_4^v = \mathbf{a}\left(\mathbf{s}^n + \Delta t\, k_3^s\right). \tag{5.11}$$

The final update becomes

$$\mathbf{s}^{n+1} = \mathbf{s}^n + \frac{\Delta t}{6}(k_1^6 + 2k_2^s + 2k_3^s + k_4^s), \qquad \mathbf{v}^{n+1} = \mathbf{v}^n + \frac{\Delta t}{6}(k_1^v + 2k_2^v + 2k_3^v + k_4^v) \quad (5.12)$$

The final update is given by

$$\mathbf{x}^{n+1} = \mathbf{x}^n + \frac{\Delta t}{6}\left(\mathbf{k}_1^x + 2\mathbf{k}_2^x + 2\mathbf{k}_3^x + \mathbf{k}_4^x\right),$$

$$\mathbf{v}^{n+1} = \mathbf{v}^n + \frac{\Delta t}{6}\left(\mathbf{k}_1^v + 2\mathbf{k}_2^v + 2\mathbf{k}_3^v + \mathbf{k}_4^v\right).$$

In our implementation, RK4 is written using temproary copies of the body array to build intermediate states. Although it performs well for short simulations and validation tasks, it is not symplectic and can exhibit slow energy drift. It is also the most computationally expensive requireing for acceleration evaluation per step. Consequently, it is not ideal for large N.

5.5 Time Stepping Strategy

While the numerical integrators describe how the particle's position and velocity vectors are updated within a single time step, the choice of that timestep size Δt has a big effect on stability and accuracy. Two different time step strategies were used.

5.5.1 Fixed Time Step

In the initial implementation, the simulation advances using a fixed timestep that is specified once at the beginning of the run. The simulation loop executes a predetermined number of iterations where at each, the same Δt is passed to the selected numerical integrator. Although this is a simple approach, it requires the timestep to be chosen wisely as it must be small enough to remain stable during close body encounters which can lead to a high computational cost even if it can be avoided when the system is evolving smoothly and the accelerations are small.

5.5.2 Adaptive Time Step

To improve numerical stability and efficiency, an adaptive timestep strategy was implemented. It dynamically adjusts the timestep based on the current stat of the system. This approach can be very computationally beneficial because when the velocities of the particles are small, it's more acceptable to take bigger steps since it won't affect the energy drift much.

At each iteration, a candidate timestep is computed for every particle using its current velocity and acceleration by the following:

$$\Delta t_i = \varepsilon_v \frac{\max(\|\mathbf{v}_i\|, v_{\min})}{\max(\|\mathbf{a}_i\|, a_{\min})}$$

Global Synchronization

All Implementations use a single global timestep per iteration. For the MPI implementation, each process selects the smallest variant among its local particles from their candidate Δt_i. A global timestep is then obtained using `MPI_Allreduce`

$$\Delta t_{\text{global}} = \min_{\text{ranks}} \Delta t_{\text{local}}, \tag{5.13}$$

with the `MPI_MIN` which computes the minimum timestep across all the ranks and makes it available for everyone. In the serial case the local timestep is directly used as the global timestep. This ensures that all processes advance with the same timestep determined by the most restrictive particle in the whole system. To stabilize the change and choose how large or small the change can be, the computed timestep is bounded by:

- $Deltat_{\min}$ which prevents it from getting excessively small,
- $\Delta t_{\max|}$, preventing overly large time advances
- a growth limiter that restricts how much a timestep may increase in one iteration

The growth limiter is defined by

$$\Delta t \leq \alpha \, \Delta t_{\text{prev}}$$

where $\alpha > 1$ is a predefined factor. This prevents sudden jumps in size. The simulation advances until a predefined final time $T_{textend}$ is reached. If the computed timestep would cause the simulation time to exceed this value, the timestep is reduced accordingly:

$$\Delta t = \min(\Delta t, \, T_{\text{end}} - t)$$

Which guarantees it will terminate at the right time without overshooting.

The MPI reduced implementation follows the same principle but modifies the timestep limiting strategy as it has a dynamic upper bound

Algorithm Overview

Algorithm 1 Adaptive Global Timestep N-Body Integration

Initialize $t \leftarrow 0$, $\Delta t_{\text{prev}} \leftarrow \Delta t_0$
Set $T_{\text{end}} = \Delta t_0 \cdot N_{\text{steps}}$
Compute initial accelerations
while $t < T_{\text{end}}$ **do**
    for each local particle i **do**
        Compute $\Delta t_i = \varepsilon_v \frac{\max(\|\mathbf{v}_i\|, v_{\text{floor}})}{\max(\|\mathbf{a}_i\|, a_{\text{floor}})}$
    end for
    $\Delta t_{\text{local}} \leftarrow \min_i \Delta t_i$
    if MPI enabled **then**
        $\Delta t_{\text{global}} \leftarrow \text{AllreduceMin}(\Delta t_{\text{local}})$
    else
        $\Delta t_{\text{global}} \leftarrow \Delta t_{\text{local}}$
    end if
    Apply bounds and growth limiter to Δt_{global}
    $\Delta t_{\text{step}} \leftarrow \min(\Delta t_{\text{global}}, T_{\text{end}} - t)$
    Integrate system for Δt_{step}
    Synchronize particle data (MPI)
    Recompute accelerations
    $t \leftarrow t + \Delta t_{\text{step}}$
    $\Delta t_{\text{prev}} \leftarrow \Delta t_{\text{step}}$
end while

6 Visualization Tool

The visualization tool is developed as a detached system from the main simulation solver to allow for headless running on an offloaded cluster, while syncing files allows for external visualization. This also allows the solver to be unimpeded by visualization workloads, allowing the scalability of the solver to be tested in detail with minimal overhead.

As this simulation tool is designed for high performance computing workloads, the visualization must be able to handle rendering even millions of particle at a time. As such, OpenGL was chosen for its low-level optimization capabilities where GPU parallelization techniques can be exploited for the best performance.

To minimize performance overhead while ensuring maximum flexibility and user control over the visualization, Dear ImGUI is used to control the timesteps, play the simulation loop, and control the rendering of the particles, including their size and trace.

Moreover, the visualization is **fully interactive**. Using 'W', 'A', 'S', and 'D' keys the camera rotates around in spherical coordinates of radius defined by the 'Zoom' slider, allowing the user to zoom in and navigate across particles while the simulation is either playing or paused.

6.1 Design

The design is centered around OOP principles and scalability. This is the outcome of several iterations in which the visualization tool was expanded. The main components discussed below are the camera and the render buffer objects (particle and its trace).

6.1.1 Camera and Movement

To allow for full interactivity with movement around the particle swarm, spherical coordinates were chosen for easiest movement around the particles.

This requires positioning the camera on the surface of a sphere and on every movement update its direction to point towards the center of the particle swarm at the origin of the scene. A diagram of this design is shown in Figure 6.1.

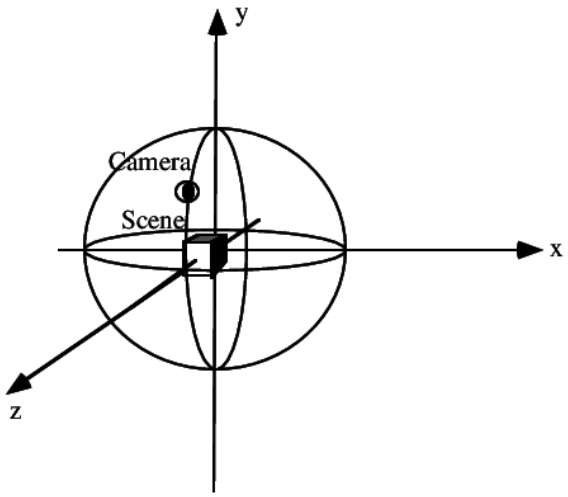


Figure 6.1: Camera on the Surface of a Sphere [1]

The transformations to the position and direction of the camera are based on the angles determining its coordinates using pitch and yaw calculations. The below code snippet demonstrates the abstraction in terms of the pitch and yaw angles and the radius of the sphere, adjusted via the `zoom` slider in the GUI.

```
class Camera {
    public:
        Camera(float radius, Shader& shader, int windowWidth, int
        ↪  windowHeight);

        void rotateCamera(float pitch, float yaw);
        void zoomCamera(float offset);
        ...
};
```

Figure 6.2 demonstrates an example of what this look like if the camera is rotated to look downwards at an angle at the swarm of particles. Next, Figure 6.3 demonstrates the sphere with a decreased radius, thus zoomed into the particle swarm.

6.1.2 Render Buffer Objects

Next, a crucial component of OpenGL rendering is the concept of buffer objects, specifically the VBO, EBO, VAO objects. These in essence store different information that the GPU needs to draw each unique object onto the screen. A naive implementation would be to use an isolated buffer object per particle. This would result in the GPU quickly running out of memory to store each buffer, and greatly reduces performance.

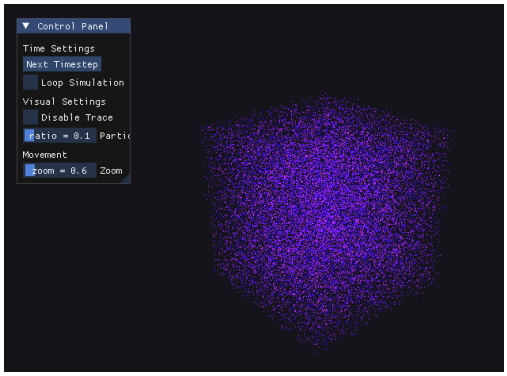


Figure 6.2: Example of Camera's Perspective in Spherical Coordinates

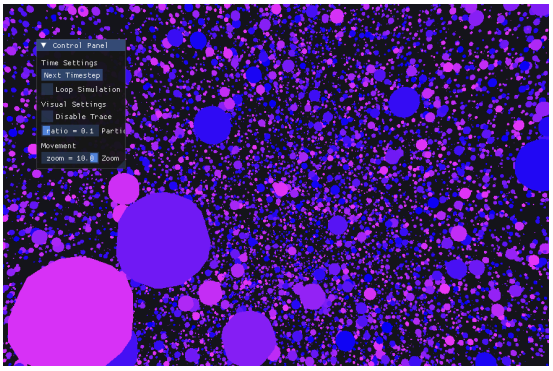


Figure 6.3: Zoomed in Camera on Particle Swarm

Therefore, all particles share a single buffer object, and instead get drawn at different scales (size) and colors using the shader with an array of sizes and colors available but sharing all the same vertices for the objects. So, from one object, all particles get draw.

This was further abstracted into a `RenderBufferObject` base class from which either `ParticleBufferObject` or `TraceBufferObject` classes can be extended. This stems from the fact that the trace also needed only a single buffer object.

```
1 class RenderBufferObject
2 {
3 public:
4     virtual void initialize(int n) = 0;
5     virtual void draw(int n) = 0;
6 };
```

The trace could be further optimized, as all traces were point particles of the same shape and color following the particles. Therefore, rather than using advanced vertex object, the most basic and embarrassingly-parallel GL_POINTS object was chosen which simple draws a pixel of no width or shape onto the screen. This way, even one million traces objects can be rendered onto the screen without causing performance degradation. This is especially important considering that there are typically 10× the number of trace particles as n-body particles, as they form a trail of length 10 behind each particle. Using GL_POINTS, the trace scales extremely well, though visually does not offer much insight with 1000 particles due to the sheer quantity of objects, therefore it can be disabled. Figure 6.4 demonstrates the smooth rendering of 1000 particles with $10,000$ traces particles in total with low gravitational constant, while Figure 6.5 shows four highly-interactive bodies and their long, beautifuly chaotic paths.

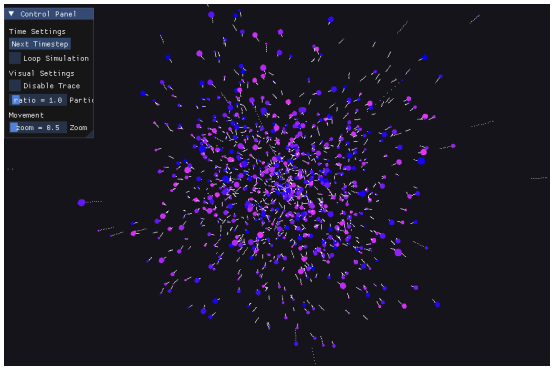


Figure 6.4: 1000 Particles with Trace Enabled

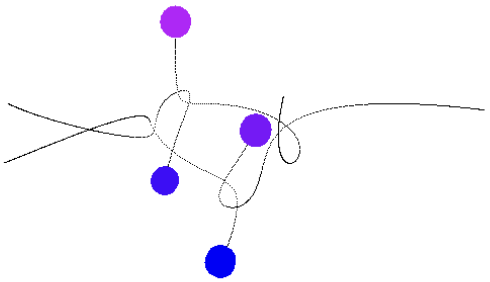


Figure 6.5: Four Bodies With Long Traces

6.2 GPU Instancing

The crucial performance upgrade which allowed rendering millions of objects simultaneously onto the screen was using instancing from the advanced OpenGL tutorial series.

This combats the last crucial perforamnce bottleneck: even if the same buffer objects are used for all particles, the program because bound by the number of CPU calls to the GPU requesting a `draw`. For 100,000 particles each with a trace of 10 points, that is over **one million** draw calls to the GPU **per frame**. This can be reduced to two calls using instancing: one for all particles, one for all traces.

Instancing essentially requires modifying the shader to draw all particles in one call using buffer data stored on the GPU. The code snippet below demonstrates how loading new instance data which has the sizes and colors of each particle is loaded onto the GPU from which a modified shader can draw all particles in a single call.

```
1  struct ParticleInstanceData {
2      glm::mat4 model;
3      glm::vec3 color;
4  };
5
6  void
   ↪  ParticleBufferObject::loadNewData(std::vector<ParticleInstanceData>&
   ↪  instances) {
7      glBindBuffer(GL_ARRAY_BUFFER, instanceVBO);
8      glBufferSubData(GL_ARRAY_BUFFER, 0,
9          instances.size() * sizeof(ParticleInstanceData),
10         instances.data());
11 }
```

Using instancing, the visualization tool is able to handle extremely large inputs, perfect for the HPC nature of the solver. For example, one million particles are able to be rendered on a mid-range Integrated Radeon Graphics card, though this does push the boundaries of the visualization tool. The resulting cubic structure of the initial state of the simulation is shown in Figure 6.6 and in a different simulation the final state is shown in 6.7.

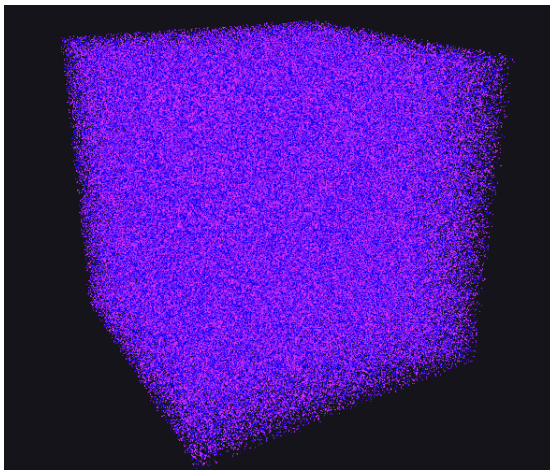


Figure 6.6: Example of One Million Particles in Initial State

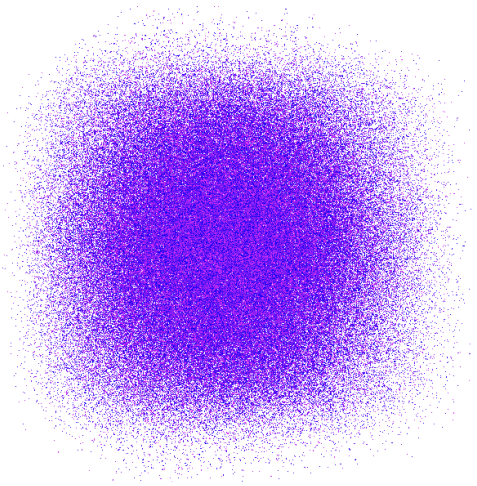


Figure 6.7: One Million Particles in Final State

7 Results

7.1 Solver Performance Evaluation

To evaluate the performance of different solvers, an extensive analysis was performed. The execution time is compared between different solvers. The OpenMP parallelization impact was analyzed, as well as dependence of dimensionality in Barnes-Hut Solver.

7.1.1 Comparison of different solvers

Solvers have been tested in order to compare their performance. The results are presented in Figure 7.1.1. Reduced Serial performs the worst, while Basic MPI and Reduced with OpenMP are doing better. Reduced MPI-Based Solver shows a significant improvement in execution time, while Barnes-Hut performs the best. The plot also clearly shows that even for 10^6 particles Barnes-Hut Solver performs better than other solvers for 10^5 particles. This is expected as its (theoretical) time complexity $O(n \log n)$ is better than $O(n^2)$ or $O\left(\frac{n^2}{P}\right)$ that other solvers have. This makes Barnes-Hut method more suitable for N-Body simulations with large number of bodies.

When comparing only the Serial, Reduced and their variants with OpenMP, one can observe an interesting pattern. Even though Reduced version is expected to be faster than Basic version, when using OpenMP parallelization, the Reduced with OpenMP becomes faster for 10^5 particles, while Basic with OpenMP is faster with less particles. A highly possible explanation is the thread scheduling and its overhead. Basic version only requires static thread scheduling, while the Reduced one needs dynamic, which only becomes more impactful when there is a higher number of particles.

7.1.2 OpenMP parallelization impact

All solvers have been implemented with OpenMP parallelization option. The results of the comparison between using OpenMP and turning it off are displayed in Table 7.1.2.

From Table 7.1.2, we can conclude that for our current solver implementations, MPI-Based and Barnes-Hut based solvers do not benefit at all from the OpenMP paralleliza-

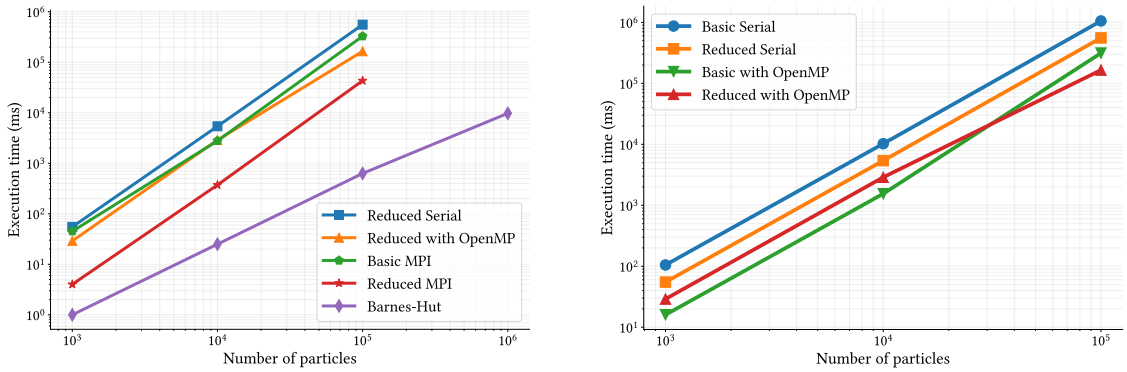


Figure 7.1: Execution time comparison of different solvers. The simulation was run in 2D setting. For Solvers that used OpenMP, the number of threads was set to 4. For MPI-based solvers, the number of MPI nodes was set to 4. The Barnes-Hut threshold value was set to $\theta = 0.5$.

Solver	Without OpenMP (seconds)	With OpenMP (seconds)
Basic Serial	274.760	69.117
Reduced Serial	144.415	31.906
Basic MPI-Based (4 nodes)	75.318	74.621
Reduced MPI-Based (4 nodes)	10.551	9.826
Barnes-Hut	30.243	31.220

Table 7.1: Execution time comparison for different solvers. Number of OpenMP threads was set to 2 for MPI-Based solvers, and 8 to other. The simulation was run in 2D setting with 3 000 particles and 3 000 steps using Sympletic integrator.

tion. This suggests that a different implementation must be considered to fully utilize the possibility of parallelizing using MPI and OpenMP together. Moreover, the test should be repeated on a proper HPC cluster.

The possible reasons for OpenMP parallelization not showing improvements include the communication between nodes overhead. For Barnes-Hut solver, the current execution time is mostly dominated (>90% of time) by the tree construction which clearly shows that parallelization will not be benefitial in the current implementation of the tree. Additionally, parallel traversal through the tree is not cache-friendly which makes the parallelization even slower than a serial Barnes-Hut Solver.

7.1.3 Barnes-Hut Solver efficiency based on the dimensionality of the problem

As the current implementation supports any kind of dimension of the problem (in practical applications, both 2D and 3D cases are supported), it is beneficial to look into the impact of the dimension of the problem for the execution time of the Barnes-Hut Solver.

Figure 7.2 reveals that Barnes-Hut in 2D has a better execution time, but with larger number of particles the difference becomes negligible. From this we can conclude that dimensionality does not impact the performance of Barnes-Hut tree and the number of particles is way more impactful.

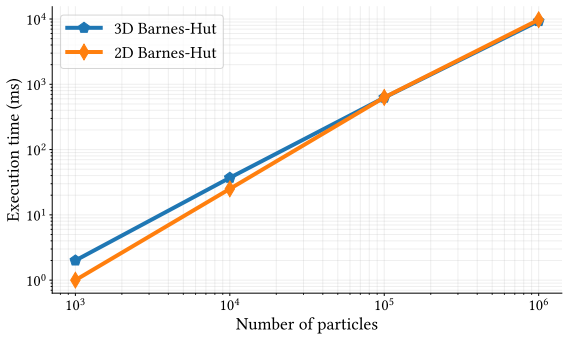


Figure 7.2: Comparison of the execution time when using Barnes-Hut solver in 2D case and 3D case. Sympletic integrator was used in both cases. Barnes-Hut threshold value was set to $\theta = 0.5$.

7.2 Solar System Case Study

Taking a step back from simulating millions of particles, a solar system case study was performed to see orbitals. Due to the chaotic nature of n-body systems, achieving a "stable" orbital is nearly impossible, as the slightest variations offset the body from its orbital into either a downwards spiral colliding into the another body, or bodies often tend to shoot out, escaping the pull of another body, essentially breaking out of orbit. A two-body system proved much simpler to achieve harmonic patterns, as shown in Figure 7.3, which illustrates the complexity of an N-body system.

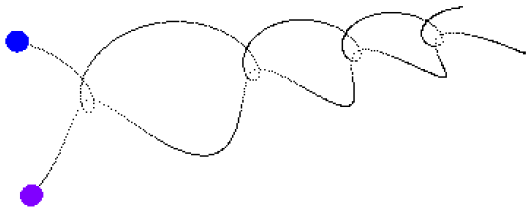


Figure 7.3: Two Body System in Predictable Pattern

Firstly, several simplifications were made. The gravitational constant was taken to be 1 to speed up the process by increasing the force, which is 11 orders of magnitude greater

than its real value. Secondly, planet mass and coordinates were scaled down. Lastly, only a small subset of orbiting planets were considered for visualization purposes. The radius of each body was scaled as cubic root of the mass to allow us to see the planets despite their relatively significantly smaller mass compared to the central star.

The Verlet scheme was used as the integrator as other discretization methods failed to produce accurate results under a large enough time step. Moreover, a timestep of $\Delta t = 0.01$ was the largest that would allow for stable orbits, as numerical errors in the computation would immediately offset the trajectory into spirals rather than orbitals. Finally, calculations for the masses were made to ensure that the central "sun" would be immobile while other planets would be perfectly in orbit, making use of Kepler's Laws of Planetary Motion for the values.

Two figures are shown: Figure 7.4 demonstrates the predictable, stable orbitals achieved with three planets. To demonstrate the beauty of 3-dimensional orbitals, five planets were set off with various tilts off the central XY plane into orbitals in Figure 7.5.

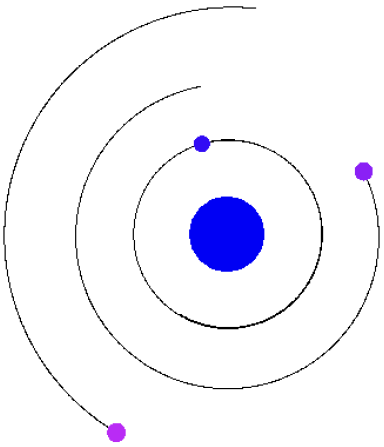


Figure 7.4: 2D Orbitals of Three Planets Around a Star

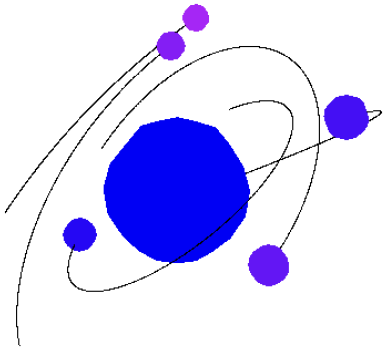


Figure 7.5: 3D Orbitals of Five Planets Around a Star

8 Conclusion

The N-Body simulation in its finalized state provides a powerful, highly-scalable solver and visualization tool tailored for high-performance workloads. At every step of the simulation pipeline, attention was put into ensuring a high level of software quality to allow for generalization and tailoring to specific needs and workflows: four integration schemes (Euler, Symplectic, Verlet, RK4), two timestep techniques (static and adaptive), three algorithms (basic, reduced, and Barnes-Hut), and two parallelization techniques (MPI and OpenMP). The flexibility of this solver makes it extremely useful for analyzing the dynamics of large particle systems.

Ultimately the performance indicated that that for large workflows, Barnes-Hut provided the best performance which allows for extremely large particle swarms with complex mechanics to be simulated in comparably little time.

While the solver was mostly tested on gravitational forces, the structure remains generic and extensible, including an already made implementation of Coloumb's Law for charged particles.

Lastly, the visualization tool allows for all the results, from small-scale stable orbitals as in the Solar System case study to HPC workloads of one million particles and their traces, while allowing the user to navigate through the chaotic dynamics of these large systems.

Bibliography

[1] Pierre Barral, Guillaume Dorme, and Dimitri Plemenos. Visual understanding of a scene by automatic movement of a camera. 01 1999.

[2] Long Chen. Introduction to fast multipole methods. Lecture notes.

[3] Peter Pacheco. Chapter 7: Distributed-memory programming with mpi. In *An Introduction to Parallel Programming*, chapter 7. Morgan Kaufmann, 2011.