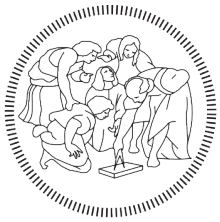

POLITECNICO
MILANO 1863

NBody Problem

By
Bice Marzagora
Mara Tortorella
Samuele Vignolo
Professor: Luca Formaggia

Exam Project

High Performance Computing Engineering

January 2024

Abstract

This project focuses on the development and implementation of two solvers for the N-body problem, aiming to provide efficient simulations of interactions between particles.
It adopts a comprehensive approach, addressing both serial and parallel algorithms to tackle the computational challenges associated with large-scale N-body simulations.

Key features of the implementation include the utilization of templates for code scalability, allowing for the addition of dimensions to the simulation. The incorporation of inheritance enables the simulation of various forces without necessitating extensive code modifications.

The project underscores the importance of computational efficiency, scalability, and adaptability in problems of increasing dimensions.

Contents

Chapter 1

Introduction

1.1 The NBody problem

The n-body problem is a classical problem in physics that involves predicting the motion of a system of bodies interacting with each other through forces.
For semplicity, we will talk about the gravitational case becasue this problem is commonly associated to celestial mechanics, since it is intensively studied in relation with the gravitational forces, but it can also be employed in terms of the electrical forces (Coulomb forces) and nuclear forces.
In simpler terms, it deals with understanding and calculating the positions and velocities of multiple celestial objects, such as planets, stars, or galaxies, over time.

The fundamental challenge in the n-body problem arises from the fact that each body in the system influences every other body through gravitational attraction, leading to a complex and intertwined set of equations of motion. The gravitational force between two bodies is proportional to the product of their masses and inversely proportional to the square of the distance between them, according to Newton's law of universal gravitation.

For a system of n bodies, the n-body problem involves solving a set of second-order differential equations (Newton's equations of motion) for each body simultaneously. Analytical solutions are often difficult or impossible to obtain for systems with more than two bodies, except for specific cases (such as the three-body problem or restricted three-body problem) where some simplifications or special conditions apply.

Numerical methods are commonly employed to approximate the solutions to the

n-body problem. In this paper we will discuss a standard approach, based on Newton's third law of dynamics, and the more advanced algorithm developed by John Barnes and Piet Hut.

Chapter 2

Standard Algorithm

2.1 Introduction

As we said before, the simulation of gravitational interactions between celestial bodies is a common problem in computational physics.
Newton's laws of gravity provide a robust framework for modeling these interactions that, combined with his laws on dynamics, make a functional algorithm for the Nbody problem. In this chapter, we will delve into this standard algorithm, which enables the simulation of the gravitational dynamics of a system with multiple interacting bodies

2.2 Simmetry in force calculations

This algorithm simulates the motions of planets or stars: through Newton's second law of motion and his law of universal gravitation, we determine the positions and velocities in our grid at each iteration.
In particular, let us consider two particles, q and k, where the force exerted on q due to k is denoted as f_{qk}.
According to Newton's third law, the force on k due to q is simply the negative of f_{qk}, i.e., $-f_{qk}$.This symmetry implies that the forces between particles are reciprocal, and we can exploit this property to cut down on redundant calculations.

[1] In the serial algorithm, this symmetry significantly reduces the computational burden. Specifically, the i-th particle only needs to calculate the forces exerted on the successive ni particles.
The forces on the preceding particles have already been computed and can be reused with a simple sign change. This optimization halves the number of force calculations in the serial algorithm, resulting in a notable time-saving during the computation.

The benefits of exploiting Newton's third law extend seamlessly to parallel algorithms: in parallel programming, dynamic scheduling is often employed to distribute the workload among multiple threads efficiently.
In this way, we can assign work to different threads in a way that ensures each thread computes forces for an equal number of particles.
This balanced workload distribution is crucial for avoiding performance bottlenecks and achieving optimal parallelization.
Without Newton's third law optimization, threads computing forces for particles early in the sequence might end up with significantly more work than those handling particles later in the sequence.
By utilizing symmetry, we ensure that the workload is evenly distributed among threads, preventing unbalanced work and maximizing parallel efficiency.

The incorporation of Newton's third law into our N-body solver not only aligns with the fundamental principles of classical mechanics but also serves as a powerful tool for optimizing computational efficiency.
Whether in a serial or parallel context, the reduction in redundant force calculations leads to a more streamlined and performant simulation, making our project more robust and scalable for a wide range of applications in astrodynamics and computational physics.

2.3 Scalability

In conjunction with the advantages of Newton's third law of dynamics, we have strategically employed specific data structures to fortify the scalability of our code. The following design choices contribute to the extensibility and adaptability of our simulation

- **Template**: we have incorporated a template approach to enhance the code's scalability, facilitating a generalized structure. This template not only accommodates the current dimensions of our simulation but is crafted in order to allow integration of additional dimensions in the future. This design principle ensures that the code can easily adapt to the evolving needs of the simulation without compromising its efficiency.

- **Use of inheritance**: the core of our design involves the use of inheritance within the Force class. The parent instance of the Force class introduces an abstract method responsible for force computation. The beauty of this approach lies in its

extensibility. Child classes inherit from the parent and can effortlessly override this method to implement the specific computations for different forces.
This design not only streamlines the current implementation for gravitational and Coulomb forces but also lays the groundwork for incorporating a diverse range of forces in the future. The extensibility to nuclear and repulsive forces, for instance, is seamlessly accommodated without necessitating substantial modifications to the existing codebase.
This modular and hierarchical design is concieved to allow addition of new forces without inducing a ripple effect across the entirety of the code. This adaptability is crucial for the dynamic nature of simulations, where the incorporation of diverse forces is not just a possibility but an inevitability.
The use of inheritance, in this context promotes maintainability and extensibility over the lifecycle of the project.

Chapter 3

Parallel version of standard algorithm

3.1 Introduction

In this chapter, we present the parallel N-Body simulation algorithm implemented in the given C++ code. The algorithm utilizes OpenMP directives to parallelize the simulation loop, allowing for concurrent computation of forces and updates to particle positions. The simulation is designed for a specified number of iterations and is responsible for managing particle interactions, collisions, and updating particle positions.

3.2 Algorithm Overview

The parallel N-Body simulation algorithm consists of the following key steps:

1. **Initialization**:
    - Initialize necessary data structures, including local force vectors and coordinate files.
2. **Simulation Loop**:
    - Assign blocks of particles to threads dynamically.
    - Handle collisions among particles or between a particle and the boundary concurrently.
    - Compute forces between particles and update local force vectors concurrently.
    - Sum up local force vectors to calculate the total force acting on each particle.

- Update particle positions and reset local force vectors.
- Periodically write updated particle positions to coordinate files.

3. **File Handling**:
   - Initialize and open coordinate files for writing particle positions.
   - Close coordinate files after the simulation loop is completed.

3.3 Parallelization Using OpenMP

The algorithm harnesses the power of OpenMP parallelization to distribute the computational workload efficiently across multiple threads. Two key OpenMP directives, `#pragma omp parallel` and `#pragma omp for`, are pivotal in achieving parallel execution of the simulation loop, force calculations, and coordinate file writing.

An important aspect of the parallelization strategy lies in the manual management of the scheduling for the `for` loops. Specifically, the algorithm employs a dynamic scheduling strategy (`schedule(dynamic, 1)`) in the first loop where forces are computed. This choice is driven by the varying computational loads among particles, attributed to the third dynamic law. In this scenario, the last particles need to perform significantly less work compared to their predecessors. In contrast, for other loops where the workload is more uniform across particles, a static scheduling strategy is adopted.

This fine-tuned scheduling approach ensures optimal utilization of computational resources, addressing the varying workloads and enhancing the overall efficiency of the parallel N-Body simulation algorithm.

3.4 Parameters and Data Structures

The algorithm relies on several parameters and data structures:

- `Dimension`: Number of dimensions of the simulation (2D or 3D).
- `it`: Number of iterations for the simulation.
- `particles`: Reference to the vector of particles.
- `dim`: Number of dimensions of the simulation.
- `softening`: Overhead to avoid particle overlapping and fusion.
- `delta_t`: Time step for updating the simulation.

- `f`: Reference to the `Force` object responsible for calculating particle interactions.
- `u`: Parameter controlling the frequency of coordinate file updates.
- `numFilesAndThreads`: Number of coordinate files and threads used in the simulation. The number of threads for the parallel section is set previously by taking the minimum between the size of the particle vector and the maximum available threads. This precautionary measure is implemented to prevent potential issues in scenarios where the number of threads exceeds the quantity of particles in the simulation.
- `localForces`: A three-dimensional vector crucial for avoiding concurrent write. Its first dimension corresponds to the number of threads, the second to the number of particles, and the third to the dimension of the simulation (2D or 3D). This data structure plays a critical role in the initial loop to ensure parallelized computation without encountering race conditions.

3.5 Efficiency Considerations

While the algorithm's underlying complexity remains $O(n^2)$, the introduction of parallelization significantly enhances its efficiency. By effectively distributing the computational workload among threads, the algorithm achieves a nearly linear speedup proportional to the number of threads used, denoted as m. Consequently, the parallel implementation can perform almost m times faster than its serial counterpart. This efficiency gain is particularly pronounced when dealing with large-scale simulations, where the benefits of parallel processing are maximized.

Chapter 4

Barnes-Hut algorithm

4.1 Introduction

The Barnes-Hut algorithm stands as a significant advancement in computational algorithms, particularly in addressing the challenges of N-body simulations where the interactions between numerous entities need to be computed. Unlike standard algorithms that necessitate $O(n^2)$ computations to evaluate pairwise interactions—quickly becoming infeasible as the number of entities, n, grows—the Barnes-Hut technique streamlines this process to an $O(n \log n)$ complexity. It cleverly approximates the cumulative effect of these distant entities, thereby optimizing computational efforts and resource utilization,

The core idea behind the Barnes-Hut algorithm is to group distant particles together and treat them as a single entity when calculating gravitational forces on a target particle. This is achieved by partitioning the simulation space into hierarchical quadrants, using a particular data structure called quadtree.

4.2 Quadtree Data Structure Implementation

In the implementation of the N-body simulation, a crucial component is the quadtree data structure, as defined in the `quadtreeNode.hpp` file.

The quadtree data structure is a hierarchical tree where each node represents a quadrant of the simulation space. Each non-leaf node can have up to four children, corresponding to the four subdivisions of the space it represents. This structure allows for a recursive partitioning of the space, where each level of the tree provides a finer granularity of the simulation area.

A noteworthy aspect of the quadtree implementation is the use of smart pointers,

specifically `std::unique_ptr`, in managing the quadtree nodes. Smart pointers are advanced constructs in C++ that provide automatic memory management, ensuring that resources are properly released when they are no longer needed, thus preventing memory leaks.

The use of `std::unique_ptr` in the quadtree nodes offers several advantages:

- **Ownership Semantics**: `std::unique_ptr` encapsulates the notion of exclusive ownership of the dynamic memory it points to. In the context of the quadtree, this means that each node has exclusive ownership of its children. This clear ownership model simplifies memory management and ensures that each node is responsible for deallocating its children.

- **Automatic Resource Management**: When a `std::unique_ptr` goes out of scope, its destructor is called, and the memory it owns is automatically deallocated. This feature is particularly beneficial in the recursive structure of the quadtree, where nodes can be dynamically created and destroyed based on the simulation's needs. Automatic resource management reduces the risk of memory leaks, a common issue in dynamic tree structures.

- **Safety Features**: `std::unique_ptr` prevents accidental copying of the pointer, which could lead to double-free errors. It enforces a move-only semantics, which is appropriate for the exclusive ownership model of the quadtree nodes. This safety feature aligns with the design principle of preventing unsafe operations that could compromise the integrity of the data structure.

4.3 Generation of the Tree Structure

The construction process begins by establishing the root node, representing the entire simulation space. This space, typically a square in 2D or a cube in 3D, must encompass all particles in the simulation.

The tree is constructed through recursive subdivision, where each node is divided into four (in 2D) or eight (in 3D) child nodes until each node satisfies certain criteria:

1. **Determine the Subdivision Point:** The subdivision point, generally the geometric center, is used to divide the node into smaller regions.

2. **Assign Particles to Subnodes:** Particles are assigned to the appropriate subnode based on their positions relative to the subdivision point.

3. **Recursive Process:** This subdivision process continues recursively for each non-empty subnode until each node contains at most one particle or a predefined maximum depth is reached.

In addition to this conventional approach, a parallelized version of the tree generation process was developed to enhance performance. This parallel implementation allows for simultaneous subdivision and particle assignment across multiple branches of the tree, leveraging multi-threading or distributed computing resources to expedite the construction phase.

For a detailed exploration of the parallel tree generation methodology, including its implementation and performance benefits, refer to section 5.3.
As the tree is built, each node, internal or leaf, calculates and stores vital properties for the force calculation phase:

- **Center of Mass:** Computed as the weighted average of the positions of all particles within the node, with weights being their masses.
- **Total Mass:** The sum of the masses of all particles within the node.
- **Size and Boundaries:** Each node is associated with size and boundary information that defines the region of space it represents.

4.4 Implementation Details

The serialBarnesHut function in main.cpp orchestrates the simulation:

1. **Building the Quadtree:** The first step in each iteration of the simulation is to build the quadtree. Particles are inserted into the quadtree, and as the tree is constructed, each non-leaf node calculates the aggregated mass and center of mass of the particles it represents.
2. **Force Calculation:** Once the quadtree is built, the algorithm traverses the tree to calculate the gravitational forces acting on each particle. If a group of particles (represented by a non-leaf node) is sufficiently far away from a target particle, the group is treated as a single entity with their aggregated mass located at their center of mass. This approximation significantly reduces the number of pairwise force calculations.

   The algorithm determines whether the node is sufficiently far from the particle in question. This determination is based on a predefined threshold parameter,θ,

which compares the width of the region represented by the node to its distance from the particle. If the node is "far enough" (determined by), the particles in that region are approximated as a single point mass located at the center of mass of the node. This approximation reduces the number of force calculations, as it eliminates the need to individually consider every particle in distant nodes.

3. **Updating Particle Positions:** After calculating the forces, the algorithm updates the positions of the particles based on the computed forces and the time step of the simulation.

4.5 Efficiency Considerations

The efficiency of the Barnes-Hut algorithm, particularly the quadtree construction phase, is characterized by its $O(n \log n)$ computational complexity. This efficiency arises from the hierarchical nature of the quadtree structure and how it scales with the number of particles, n, in the simulation.

Logarithmic Tree Depth

The depth of the quadtree increases logarithmically with the number of particles. This means that even as the number of particles grows large, the increase in the number of levels in the tree (and consequently, the number of computations required to navigate and update the tree) does not grow proportionally to n, but rather as a function of $\log n$. This logarithmic relationship is key to the algorithm's scalability and performance.

Balanced Division of Space

Each level of the quadtree divides the simulation space into four (in 2D) or eight (in 3D) equal parts, with each subsequent level focusing on a progressively smaller area. This division allows the algorithm to distribute particles among a manageable number of nodes at each level, reducing the potential computational load at any single node or level of the tree.

Dynamic Adaptation to Particle Distribution

The quadtree's structure dynamically adapts to the distribution of particles within the simulation space. In regions of high particle density, the tree will have more levels (i.e., greater depth), allowing for more precise calculations where they are most needed. In sparser regions, the tree remains shallower, avoiding unnecessary computations. This

adaptive behavior contributes to the overall efficiency, ensuring that computational resources are focused where they are most beneficial.

Efficient Force Calculations

The primary purpose of the quadtree in the Barnes-Hut algorithm is to facilitate efficient calculation of forces between particles. By grouping distant particles and treating them as a single entity, the algorithm significantly reduces the number of pairwise force calculations required. The hierarchical structure of the quadtree enables this approximation to be done systematically and efficiently, with the depth of the tree playing a crucial role in determining the level of approximation.

In summary, the $O(n \log n)$ time complexity of the quadtree construction in the Barnes-Hut algorithm reflects a sophisticated balance between the need for detailed force calculations and the computational constraints of handling large numbers of particles. The logarithmic growth of the tree's depth with the number of particles ensures that the algorithm remains efficient and scalable, even for simulations involving millions of particles.

Chapter 5

Parallel Barnes-Hut Algorithm

5.1 Introduction

This chapter introduces the parallel implementation of the N-Body Barnes-Hut simulation algorithm, as presented in the provided C++ code. Following the footsteps of the parallel standard algorithm outlined in Chapter 3, this method capitalizes on OpenMP directives to parallelize the simulation loop. This allows for concurrent computation of forces and simultaneous updates to particle positions. The algorithm adeptly manages complex tasks, including particle interactions, collision resolution, and precise updates to particle positions.

5.2 Algorithm Workflow

The parallel Barnes-Hut N-Body simulation algorithm unfolds through a series of key steps:

1. **Initialization**:
    - Establish critical data structures, including coordinate files.
2. **Simulation Loop**:
    - For each iteration, create the QuadTree structure in parallel, as detailed in Section 5.3.
    - Concurrently manage collisions among particles or with the boundary.
    - Calculate forces acting between particles using a methodology equal to the serial Barnes-Hut implementation, elaborated in Section 4.4.
    - Update particle positions, velocity, and reset local force vectors.

- Periodically record updated particle positions in coordinate files.

3. **File Handling**:
    - Initialize and open coordinate files to document particle positions.
    - Close coordinate files upon completion of the simulation loop.

5.3 Parallel tree creation

The `generateTreeParallel` function within `main.cpp` serves as a template function specifically designed for the parallel construction of quadtrees, a pivotal component for optimizing the efficiency of the Barnes-Hut algorithm in N-dimensional particle simulations.
This function takes two parameters as input: a pointer to a vector of `Particle` objects, which represents the set of particles in the simulation, and a double `dimSimulationArea`, specifying the dimension of the simulation area.
The parallel generation of the tree follows these steps:

1. **Determine the number of available threads and subdivision of the simulation area:** The execution of the function begins by determining the optimal number of threads (`num_threads`), using the `omp_get_max_threads` function. Then, it calculates the number of quadrants (`num_quad`) required to effectively partition the simulation space, ensuring that each thread can work on a separate portion of the space. This ensures that each thread has a distinct portion of the simulation space. Typically, e the number of subareas is approximated to the next multiple of a power of 4 to ensure a balanced division.

2. **Generation of sub-trees:** Each thread is responsible for generating the subtree corresponding to its subarea. This step occurs independently for each thread, enabling efficient parallel execution.

3. **Merging phase:** After generating sub-trees, the merging phase begins. Results generated by each thread, represented by the sub-trees, are combined to form the final quadtree. This merging process can be performed in parallel, using a Divide and Conquer Approach: the set of sub-trees are divided into smaller groups of four subtree, which are assigned to a thread for parallel merging. This ensures that the merging workload is distributed evenly across available threads.

4. **Final tree:** At the end of the process, a complete quadtree is obtained, reflecting the optimized subdivision of the simulation space. This quadtree is ready to be used in the Barnes-Hut algorithm during the N-body simulation.

In summary, by dividing the simulation space into segments and constructing the quadtree in parallel, the function significantly enhances the efficiency of the Barnes-Hut algorithm, making it feasible to simulate large systems with numerous particles. This approach not only optimizes computational resources but also underscores the importance of parallel algorithms in tackling complex computational physics problems.

5.4 Function Parameters and Data Structures

The function `parallelBarnesHut` takes several parameters and utilizes key data structures in its implementation:

- `iterationNumber`: The number of iterations for the simulation loop.
- `particles`: A pointer to the vector of `Particle<Dimension>` objects representing the particles in the simulation.
- `dimSimulationArea`: An integer representing the dimension of the simulation area.
- `softening`: A double value representing the overhead to avoid particles overlapping and fusing together.
- `delta_t`: A double value representing the time step after which the simulation is updated.
- `f`: A reference to the `Force<Dimension>` object responsible for calculating particle interactions.
- `speedUp`: An integer specifying the frequency of updating particle positions for visualization.
- `numFilesAndThreads`: A size_t value representing the number of coordinate files and threads used.

5.5 Parallelization considerations

The adoption of OpenMP for parallelization in this algorithm remains consistent, chosen for its simplicity and high-performance outcomes. In contrast to previous instances, the third law of dynamics was not employed. The decision to omit it in this implementation was driven by a careful consideration of performance trade-offs. The added complexity associated with considering the third law did not yield substantial performance gains, and its exclusion significantly streamlined the algorithm. As a result, each particle can now compute its force concurrently with others, contributing to overall computational efficiency.

5.6 Efficiency considerations

Despite maintaining a fundamental time complexity of O($nlog(n)$) like the serial implementation, where n represents the number of particles, the introduction of parallelization has a transformative impact on the algorithm's efficiency.

By adeptly distributing the computational workload among multiple threads, the algorithm attains a remarkable boost in performance. Like the parallel version of the standard algorithm, the speedup achieved is nearly linear, directly proportional to the number of threads used, denoted as m. Consequently, the parallel implementation demonstrates an impressive m-fold increase in speed compared to its serial counterpart. This pronounced efficiency gain becomes particularly evident and impactful in scenarios involving large-scale simulations, where the advantages of parallel processing are maximized.

Chapter 6

Numerical Integration Methods in N-body Simulation

In the simulation of N-body problems, numerical integration methods play a crucial role in updating the positions and velocities of particles over time. This chapter focuses on two numerical integration methods implemented in your project: Euler integration and velocity Verlet integration.

6.1 Euler Integration

Euler integration is a straightforward numerical method widely used for solving ordinary differential equations, including those encountered in N-body simulations. The method is named after the Swiss mathematician Leonhard Euler, and it is characterized by its simplicity and computational efficiency.

6.1.1 Basic Principle

The fundamental principle behind Euler integration is to update the state of a system at discrete time steps. In the context of the N-body problem, the state of a particle is described by its position $\mathbf{r}$ and velocity $\mathbf{v}$. The update equations for Euler integration are given by:

$$\mathbf{r}_{\text{new}} = \mathbf{r}_{\text{old}} + \mathbf{v}_{\text{old}} \cdot \Delta t \tag{6.1}$$

$$\mathbf{v}_{\text{new}} = \mathbf{v}_{\text{old}} + \frac{\mathbf{F}}{m} \cdot \Delta t \tag{6.2}$$

Here, $\mathbf{F}$ represents the force acting on the particle, m is the particle's mass, and Δt is the time step.

6.1.2 Algorithmic Implementation

The algorithmic implementation of Euler integration involves updating the particle's position and velocity sequentially.

```
void update(const double delta_t) {
            for(size_t i = 0; i < Dimension; ++i)
            {
                pos[i] += vel[i] * delta_t;
                vel[i] += (force[i] / ((property<0)?
                    -property:property)) * delta_t;
            }
        }
```

Control over the `property` attribute of the particle is exercised for simplicity, wherein we assume the mass of the particle is equivalent to the magnitude of its charge, stored in the `property` variable.

6.1.3 Limitations

While Euler integration is computationally efficient, it has certain limitations. One notable drawback is its tendency to accumulate errors over time, especially when dealing with systems exhibiting rapid changes. The method assumes a constant force during each time step, which might not accurately represent complex physical scenarios.

In the context of N-body simulations, where interactions between multiple particles are involved, Euler integration may lead to inaccuracies in predicting long-term trajectories. Researchers often opt for more sophisticated integration schemes, such as the velocity Verlet method, to improve accuracy in such scenarios.

Despite its limitations, Euler integration remains a valuable tool for quick simulations and initial explorations due to its simplicity and ease of implementation.

6.1.4 Conclusion

In summary, Euler integration provides a basic yet effective approach to numerically solving differential equations governing the motion of particles in an N-body system. Its simplicity makes it a popular choice for initial implementations and quick simulations, but users should be aware of its limitations, particularly when high accuracy is required over extended periods of simulation time.

6.2 Velocity Verlet Integration

Velocity Verlet integration is a numerical method commonly used in molecular dynamics simulations and N-body problems. It is a symplectic integration scheme, preserving certain properties of the system, such as energy conservation, over extended periods. The method is especially suitable for systems where the conservation of total energy is crucial.

6.2.1 Basic Principles

The Velocity Verlet integration algorithm eliminates the half-step velocity, resulting in a simplified form:

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \mathbf{v}(t)\Delta t + \frac{1}{2}\mathbf{a}(t)\Delta t^2 \tag{6.3}$$

Here, $\mathbf{x}(t)$ represents the position, $\mathbf{v}(t)$ is the velocity, $\mathbf{a}(t)$ is the acceleration, Δt is the time step, and t denotes the current time.

The acceleration at the next time step ($\mathbf{a}(t + \Delta t)$) is derived from the interaction potential using the updated position $\mathbf{x}(t + \Delta t)$.

Finally, the velocity is updated using both the current and next accelerations:

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \frac{1}{2}(\mathbf{a}(t) + \mathbf{a}(t + \Delta t))\Delta t \tag{6.4}$$

It is important to note that this algorithm assumes that the acceleration $\mathbf{a}(t + \Delta t)$ only depends on the position $\mathbf{x}(t + \Delta t)$ and does not depend on the velocity $\mathbf{v}(t + \Delta t)$.

6.2.2 Advantages of Velocity Verlet Integration

- **Symplectic Nature:** Velocity Verlet is symplectic, preserving phase-space volume and ensuring long-term stability in conservative systems.
- **Energy Conservation:** The method conserves energy well, making it suitable for simulations where maintaining total energy is critical.
- **Second-Order Accuracy:** Velocity Verlet has second-order accuracy, providing better convergence properties compared to first-order methods like Euler integration.

6.2.3 Implementation in Code

The implementation of Velocity Verlet integration in code is straightforward.

```
void velocityVerletUpdate(const double delta_t) {
            std::array<double, Dimension> prevAccel;
            for(size_t i=0; i<Dimension; ++i)
            {
                prevAccel[i] = accel[i];
                accel[i] = (force[i] / ((property<0)? -property:property));
                pos[i] += vel[i] * delta_t
                    + 0.5 * prevAccel[i] * delta_t * delta_t;
                vel[i] += 0.5 * (prevAccel[i] + accel[i]) * delta_t;
            }
        }
}
```

Chapter 7

File Handling

7.1 Introduction

In this chapter, we explore the file handling procedures implemented in our N-Body simulation project. The writing to files is a crucial aspect, occurring in nearly every iteration of the algorithms, contingent on the `speedup` parameter. The subsequent visualization of the simulation, as explained in Chapter 9, relies on reading these files using a Python script.

7.2 Info.txt File

Regardless of the type of simulation being executed, an `Info.txt` file is created. This file contains essential information, including:

- Number of particles used in the simulation
- Dimension of the simulation area
- Number of lines the program will write for each particle (calculated as the division between `it` and `speedUp` parameters)
- Number of files that will be created to store the positions of the particles
- Radius of each particle (for graphical purposes)
- Id and initial position of particles

The `Info.txt` file serves as a comprehensive summary of the simulation parameters.

7.3 File Creation and Avoidance of Collisions

After the creation of the `Info.txt` file, the simulation commences, generating multiple files (`Coordinates_i.txt`) where i represents the id of each thread. The number of these files (m) is determined by the value stored in the `numFilesAndThreads` variable. This approach ensures that each thread writes to its designated file, preventing collisions among threads.

For parallel algorithms, this strategy guarantees that different threads do not interfere with each other, contributing to a collision-free file writing process. In the case of a serial algorithm, only two files are created: `Info.txt` and `Coordinates_0.txt`.

7.3.1 The `speedUp` Parameter

In the course of the simulation, file writes are not executed in every iteration but rather once every `speedUp` iterations. This design choice is driven by two key considerations:

- **Enhanced Algorithm Efficiency:** The decision to write to the file less frequently is motivated by the goal of optimizing algorithm performance. By reducing the frequency of write operations, we achieve faster execution without compromising precision. This approach not only speeds up the algorithm but also helps maintain a lighter file size.
- **Dynamic Adjustment for Graphics Simulation:** In the context of the Python graphical simulation, where the frame frequency remains constant, the choice of the `speedUp` parameter becomes crucial. Adjusting the `deltaT` parameter directly impacts the simulation speed. Decreasing the `deltaT` parameter while increasing the number of iterations and the `speedUp` parameter ensures a consistent simulation outcome. This strategy results in improved accuracy and efficiency, creating a more refined and faster simulation.

7.4 Open and Close Operations

Before and after the simulation, open and close operations are performed on the files, respectively. These operations are crucial to ensure proper handling of file resources, avoiding potential issues related to file access and modification during the simulation.

Efficient file handling is integral to the successful execution of our N-Body simulation. The systematic creation, organization, and management of files facilitate both the simulation process and subsequent data visualization, enhancing the overall robustness and reliability of our implementation.

Chapter 8

Comparative Analysis of the algorithms

8.1 Performance Comparison

Performance Comparison of N-body Algorithms

In this section, we provide a performance comparison of the four implemented algorithms for the N-body problem: the brute-force algorithm, the parallelized brute-force algorithm with N threads, the serial Barnes-Hut algorithm, and the parallel Barnes-Hut algorithm.

1. **Brute-Force Algorithm:** The brute-force algorithm has order $O(n^2)$, where n represents the number of particles, since it loops over all the particles of the simulation and, for each particle, calculates the force exerted on the other particles. The brute-force approach is inefficient for large scale problems.

2. **Parallelized Brute-Force Algorithm with n Threads:** While the algorithm's inherent complexity remains $O(n^2)$, adding parallelization amplifies its efficiency. By addistributing the computational workload across threads, the algorithm attains nearly linear speedup, directly proportional to the number of threads employed and denoted as m. As a result, the parallel implementation can achieve an almost m-fold acceleration compared to its serial counterpart. This efficiency gain becomes more important in the context of large-scale simulations, where the advantages of parallel processing are most evident.

3. **Serial Barnes-Hut Algorithm:** This algorithm has order $O(nlog(n))$, this complexity arises from the hierarchical decomposition of the simulation space. In

each level of the tree, the algorithm makes decisions about whether to treat a group of particles as a single, distant mass. This process occurs recursively down the tree. The logarithmic term $O(log(n))$ accounts for the depth of the tree, reflecting the reduction in the number of computations as the algorithm focuses on larger groups of particles in distant regions.

4. **Parallel Barnes-Hut Algorithm:** In this method, despite maintaining a fundamental time complexity of $O(nlog(n))$ like the serial implementation, where n represents the number of particles, the introduction of parallelization has a transformative impact on the algorithm's efficiency. By adeptly distributing the computational workload among multiple threads, the algorithm attains a remarkable boost in performance. Like the parallel version of the standard algorithm, the speedup achieved is nearly linear, directly proportional to the number of threads used, denoted as m. Consequently, the parallel implementation demonstrates an impressive m-fold increase in speed compared to its serial counter- part. This pronounced efficiency gain becomes particularly evident and impactful in scenarios involving large-scale simulations, where the advantages of parallel processing are maximized.

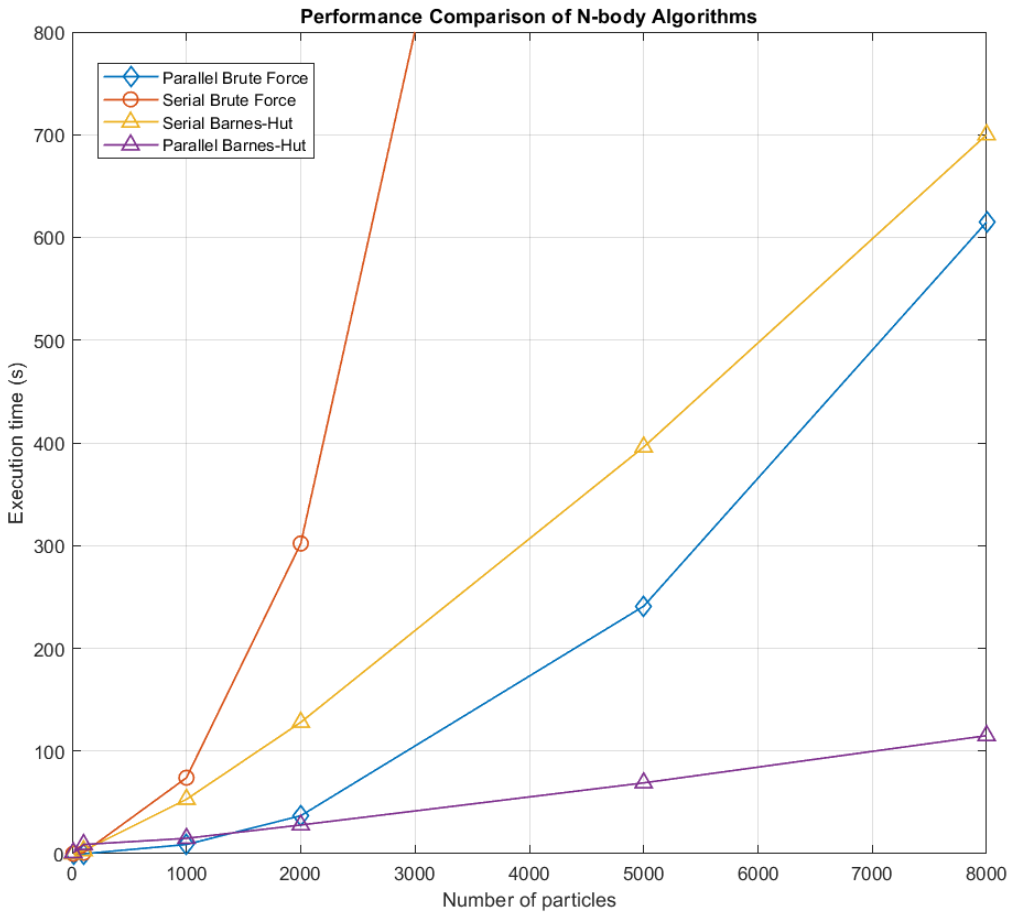

Figure 8.1: Simulations using 16 threads in the parallel algorithms

Chapter 9

Animation of the Nbody simulation

A Python script was designed in order to visualize the results of the N-body simulation. The results are stored in a series of files, the Python script reads these files, processes the data, and graphically simulates the motion of particles in a 2D or 3D space using the Matplotlib library.

9.1 Python Code Overview

The script is structured to perform several key operations: setting up the graphical environment, reading and processing input files, initializing graphical representations of particles, and animating their motion based on the simulation data.

9.1.1 Data Processing

The script reads the simulation parameters and particles' coordinates from files generated by the C++ program. This is done by opening `Info.txt` for simulation parameters and a series of `Coordinates_i.txt` files for particle positions, as discussed in chapter 7.

Depending on the dimensionality of the simulation (2D or 3D), the script uses different approaches to visualize the particles:

- In 2D, particles are represented as circles with their positions updated in each frame of the animation.
- In 3D, particles are visualized as spheres using Matplotlib's 3D plotting capabilities, with the script dynamically updating their positions.

The script features buttons to control the animation, allowing the user to start, stop, and reset the simulation. This interactivity is facilitated by Matplotlib's 'Button' widget.

Bibliography

[1] M. M. Peter S. Pacheco, *An Introduction to Parallel Programming*. Morgan Kaufmann-Elsevier, 2022.