Parallel Computing - MPI

The n-body problem

Author

EL AMRANI Wadie

Professor

DA COSTA PINTO CAMPOS Georges

Toulouse, 20 octobre 2023

Contents

1 Introduction

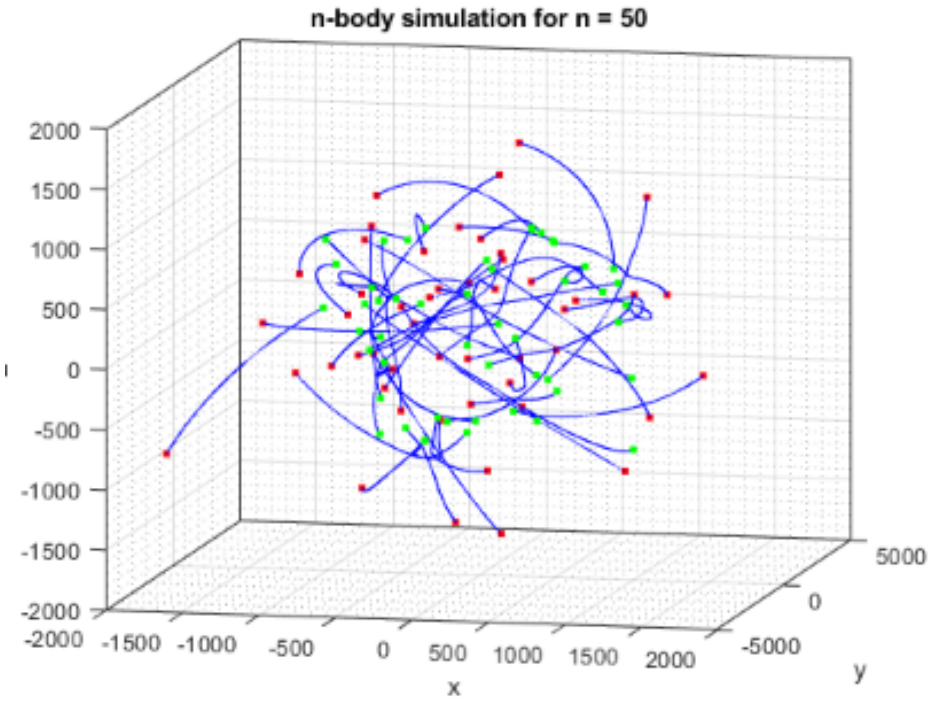


Figure 1: n-body simulation for n = 50

The n-body problem is one of the most famous problems in mathematical physics, with its first complete mathematical formulation dating back to Newton's Principia. Classically, it refers to the problem of predicting the motion of n celestial bodies that interact gravitationally. Nowadays, other problems, such as those from molecular dynamics, are also often referred to as n- body problems.

For n = 2, the problem was completely solved by Johann Bernoulli.

For n = 3, solutions exist in special cases. In general, numerical methods must be used to simulate such systems.

We will specialize the discussion in terms of the gravitational problem, so we treat as inputs the mass, position, and velocity of each particle. The output is typically the positions and velocities of all the particles at some final time or sequence of times. We assume Newtonian physics to describe the motion.

2 Problem formulation

2.1 The 2-body case

Consider two particles of masses m_1 and m_2, separated by a distance r. According to Newton's law of universal gravitation, the force of attraction F between the two masses is given by

$$F = G\frac{m_1 m_2}{r^2} \tag{1}$$

where G is the gravitational constant. This force dictates the motion of the two bodies and can be used to derive their future positions and velocities, assuming no other forces act upon them.

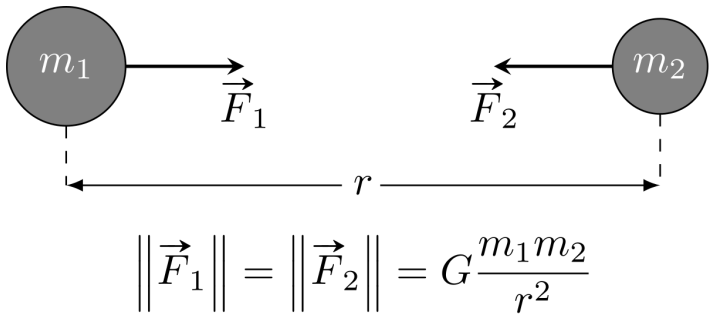


Figure 2: The gravitational force between two particles

2.2 The N-body case

Suppose we have n particles with masses m_i and positions $\mathbf{r}_i(t)$. Then particle i has a (gravitational) force exerted on it by particle j given by

$$\mathbf{f}_{i,j}(t) = -\frac{Gm_i m_j}{\|\mathbf{r}_i(t) - \mathbf{r}_j(t)\|^3}(\mathbf{r}_i(t) - \mathbf{r}_j(t)), \tag{2}$$

where $G = 6.673 \times 10^{-11}$ m/(kg $\cdot$ s^2).

The total force on particle i is thus given by

$$\mathbf{F}_i(t) = \sum_{\substack{j=0 \\ j\neq i}}^{n-1} \mathbf{f}_{i,j}(t) = -Gm_i \sum_{\substack{j=0 \\ j\neq i}}^{n-1} \frac{m_j}{\|\mathbf{r}_i(t) - \mathbf{r}_j(t)\|^3}(\mathbf{r}_i(t) - \mathbf{r}_j(t)). \tag{3}$$

Now using Newton's second law $\mathbf{F} = m\mathbf{a}$ translated to our notation, we have

$$\mathbf{F}_i(t) = m_i \ddot{\mathbf{r}}_i(t), \tag{4}$$

for $i = 0, 1, \ldots, n-1$.

From this, we obtain a system of second-order ordinary differential equations (ODEs)

$$\ddot{\mathbf{r}}_i(t) = -G\sum_{\substack{j=0 \\ j\neq i}}^{n-1} \frac{m_j}{\|\mathbf{r}_i(t) - \mathbf{r}_j(t)\|^3}(\mathbf{r}_i(t) - \mathbf{r}_j(t)), \tag{5}$$

for $i = 0, 1, \ldots, n-1$.

To make life slightly simpler, we assume $\mathbf{r}_i(t) \in \mathbb{R}^2$. We also assume that we will integrate the ODEs with an explicit numerical method and constant step size Δt. (More details on exactly what this means in a moment.) The output is to be given at a future time $T = N\Delta t$.

3 Data structures

In our n-body simulation, two primary data structures, `data` and `force`, play crucial roles.

3.1 Data

The `data` array is a crucial data structure in our n-body simulation, representing the state of each celestial body. It is structured as a two-dimensional array with 'N' rows and 6 columns, where 'N' is the number of bodies. Each row in this array corresponds to a specific celestial body, and the columns represent the body's unique identifier (ID), position in the 2D space (POSX, POSY), velocity (SPEEDX, SPEEDY), and mass (WEIGHT). The structure of the `data` array can be visualized as follows:

$$\text{data} = \begin{bmatrix} \text{ID}_1 & \text{POSX}_1 & \text{POSY}_1 & \text{SPEEDX}_1 & \text{SPEEDY}_1 & \text{WEIGHT}_1 \\ \text{ID}_2 & \text{POSX}_2 & \text{POSY}_2 & \text{SPEEDX}_2 & \text{SPEEDY}_2 & \text{WEIGHT}_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \text{ID}_N & \text{POSX}_N & \text{POSY}_N & \text{SPEEDX}_N & \text{SPEEDY}_N & \text{WEIGHT}_N \end{bmatrix}$$

Each row in this matrix represents the state of a celestial body at a given time step in the simulation, allowing for efficient tracking and updating of each body's position and velocity.

Specifically, `data[i]` is a 6-element vector comprising floating-point numbers that encapsulate the state of the i-th body. These elements include the body's unique identifier (ID), its position in the 2D space (POSX, POSY), its velocity (SPEEDX, SPEEDY), and its mass (WEIGHT). The structure of this vector can be visualized as follows:

$$\text{data}[i]^T = \begin{bmatrix} \text{ID} \\ \text{POSX} \\ \text{POSY} \\ \text{SPEEDX} \\ \text{SPEEDY} \\ \text{WEIGHT} \end{bmatrix}$$

3.2 Force

The `force` array is a crucial data structure in the n-body simulation, designed to store the net gravitational forces acting on each celestial body. It is initialized as a two-dimensional array with dimensions $(N, 2)$, where N represents the number of bodies. Each row in this array corresponds to a specific body, and the two elements in each row represent the X and Y components of the net force vector acting on that body. This can be visualized as:

$$\text{force} = \begin{bmatrix} F_{x1} & F_{y1} \\ F_{x2} & F_{y2} \\ \vdots & \vdots \\ F_{xN} & F_{yN} \end{bmatrix}$$

In this matrix, F_{xi} and F_{yi} denote the X and Y components of the force acting on the i-th body. This structure allows for efficient calculation and updating of forces during each step of the simulation.

3.3 Python Implementation

In the implementation of our n-body simulation, we leverage the NumPy library, a fundamental package for scientific computing in Python. NumPy provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays. This makes it particularly well-suited for efficiently handling the computations required in our simulation.

3.3.1 NumPy Library

NumPy is utilized due to its high performance and ease of use when dealing with large arrays of numerical data. It offers efficient storage and manipulation of arrays, which is critical in simulations that involve complex calculations and large datasets.

```
1 import numpy as np
```

3.3.2 Data Initialization

The `data` array, representing the state of each celestial body, is initialized as a NumPy array of floating-point numbers. Each body's state is stored as a vector within this array.

```
1 data = np.array([id, positionx, positiony, speedx, speedy,
  weight], dtype='f')
```

3.3.3 Force Initialization

The `force` array, used to store the net gravitational forces, is initialized as a two-dimensional NumPy array. This array is reset to zero at the beginning of each simulation step to accurately calculate the forces in the current state.

```
1 force = np.zeros((n_bodies, 2))
```

Through the use of NumPy, the simulation achieves high computational efficiency, which is essential for handling the complex calculations involved in the n-body problem.

4 Sequential version

4.1 Sequential algorithm

At each time step, we calculate the forces that apply to each star, then we update the data concerning it (position, velocity, and acceleration).

The basic solution consists at each iteration and for each star in:

1. calculating the forces exerted by every other star,
2. computing the resultant of forces by summing up all these forces,
3. updating the position, velocity, and acceleration of the star based on the resultant of forces.

The sequential algorithm for a 2 dimension problem is as follows:

High-Level Algorithm

Algorithm 1 Serial N-Body Solver - High Level

Require: Input data
1: **for** each (constant) timestep **do**
2: **for** each particle i **do**
3: Compute $F_i(t)$.
4: Update $r_i(t)$ (and $\dot{r}_i(t) := v_i(t)$).
5: **return** $r_i(T)$ for each particle i.

Detailed Implementation

Algorithm 2 Serial N-Body Solver - Detailed Implementation

1: **for** $t = 0$ **to** NB_STEPS **do**
2: force = np.zeros($(N, 2)$) ▷ force[i] is a 2D vector
3: **for** $i = 0$ **to** N **do**
4: **for** $j = 0$ **to** N **do**
5: **if** $i \neq j$ **then**
6: force[i] = force[i] + interaction(data[i], data[j])
7: **for** $i = 0$ **to** N **do**
8: data[i] = update(data[i], force[i])

update()

update(d, f): This function takes a body d and a force f as parameters and updates the position and velocity of the body d based on the force f.

The update function calculates the new velocities and positions as follows:

$$v_x = v_{x,old} + \Delta t \cdot \frac{F_x}{m} \tag{6}$$

$$v_y = v_{y,old} + \Delta t \cdot \frac{F_y}{m} \tag{7}$$

$$p_x = p_{x,old} + \Delta t \cdot v_x \tag{8}$$

$$p_y = p_{y,old} + \Delta t \cdot v_y \tag{9}$$

Here, v_x and v_y are the new velocities, p_x and p_y are the new positions, F_x and F_y are the force components, m is the mass of the body, and Δt is the time step.

```
def update(d, f):
    dt = 1e11
    vx = d[SPEEDX] + dt * f[0]/d[WEIGHT]
    vy = d[SPEEDY] + dt * f[1]/d[WEIGHT]
    px = d[POSX] + dt * vx
    py = d[POSY] + dt * vy
    return create_item(d[ID], positionx=px, positiony=py,
                       speedx=vx, speedy=vy, weight=d[WEIGHT])
```

Interaction()

interaction(body1, body2): this function returns a list with two elements representing the force on each of the X and Y axes, resulting from the interaction of *body2* on *body1*

The interaction function is defined as:

$$F = \frac{G \cdot m_i \cdot m_j}{dist^2 + softening^2} \cdot \frac{\vec{r}_{ij}}{dist} \tag{10}$$

Where F is the force vector, G is the gravitational constant, m_i and m_j are the masses of bodies i and j, $\vec{r}_{ij}$ is the position vector from body i to body j, $dist$ is the distance between the bodies, and *softening* is a small constant to prevent division by zero in the case of overlapping positions.

```
def interaction(i, j):
    dist = math.sqrt( (j[POSX]-i[POSX])*(j[POSX]-i[POSX]) +
                      (j[POSY]-i[POSY])*(j[POSY]-i[POSY]) )
    if dist == 0:
        return np.zeros(2)
    g = 6.673e-11
    factor = g * i[WEIGHT] * j[WEIGHT] / (dist*dist+3e4*3e4)
    return np.array([factor*(j[POSX]-i[POSX])/dist,
                     factor*(j[POSY]-i[POSY])/dist])
```

4.2 Implementation

The following Python code represents a sequential implementation of an n-body simulation. The simulation calculates the forces between bodies according to the laws of gravity, updates their positions and velocities, and outputs a signature value representing the state of the system.

```
import numpy as np
import sys
import math
import random
import matplotlib.pyplot as plt
import time
from n_bodies_base import init_world, interaction, update, signature

# The number of bodies and number of steps are passed as command-line arguments
nbbodies = int(sys.argv[1])
NBSTEPS = int(sys.argv[2])

# Initialize the world with n bodies
data = init_world(nbbodies)

# Simulation loop
for t in range(NBSTEPS):
    force = np.zeros((nbbodies, 2))  # Initialize the force array
    for i in range(nbbodies):
        for j in range(nbbodies):
            if i != j:  # Skip self-interaction
                force[i] += interaction(data[i], data[j])  # Calculate force

    for i in range(nbbodies):
        data[i] = update(data[i], force[i])  # Update the body's state

# Output the signature of the world
print(signature(data))
```

4.3 Demonstration and Visualization of N-Body Simulation Progress

In the provided figure, we have implemented a visualization for an n-body simulation, specifically tailored to display the dynamic evolution of a system comprising 20 bodies over 90 steps. The simulation is designed to calculate gravitational interactions, update positions and velocities, and graphically represent the state of the system at each step.

The central body, which could be likened to a star in an astrophysical model, is distinctly marked in blue to differentiate it from the other bodies, which are depicted in red. This color distinction helps in quickly identifying the central mass and observing its influence on the surrounding bodies.

The visualization is structured to capture snapshots every 10 steps.

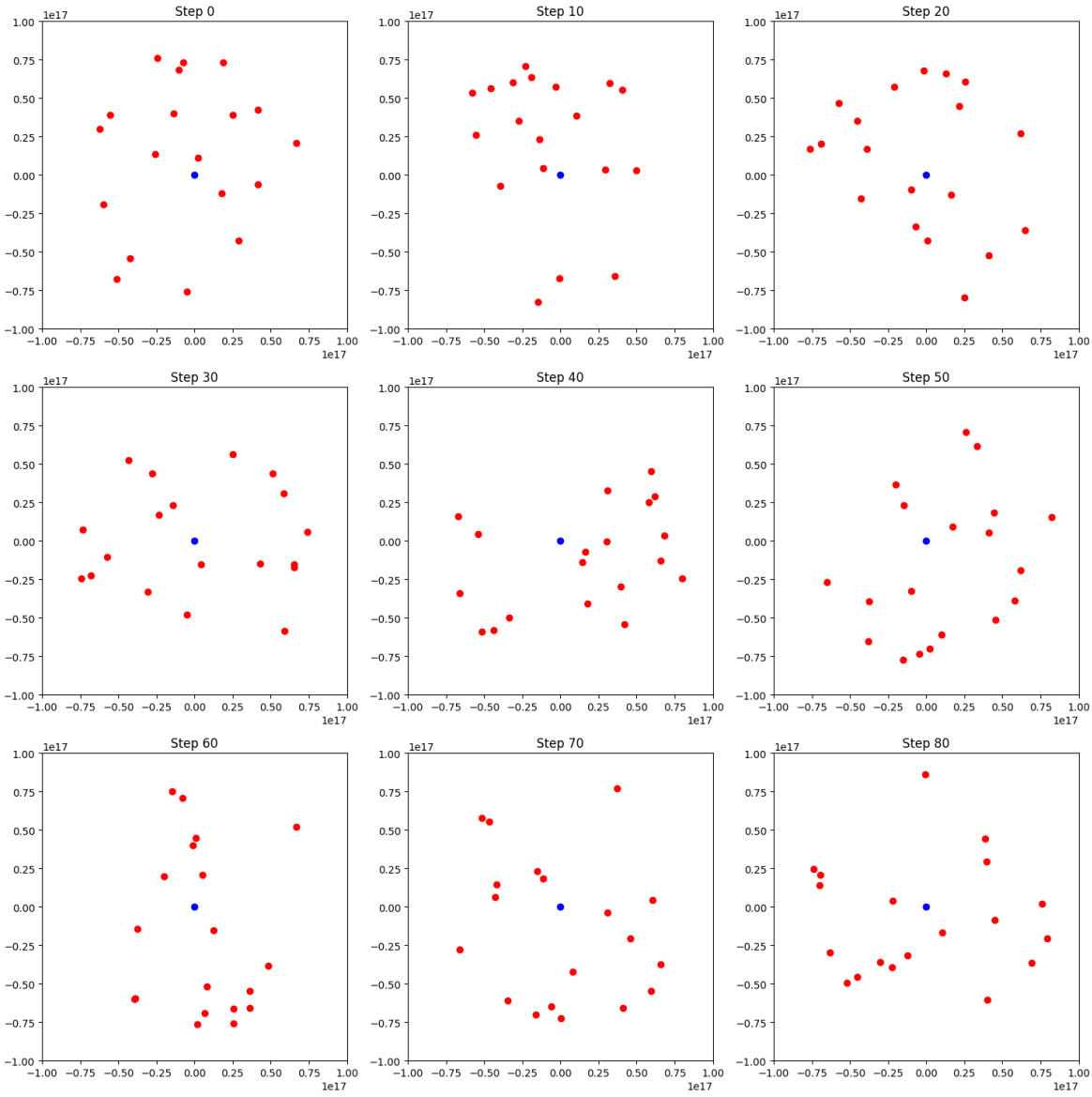


Figure 3: Visual Evaluation of 20-Body Simulation Steps

4.4 Evaluation

The theoretical complexity of the sequential implementation of the n-bodies simulation can be established based on the nested loop structure used for calculating the gravitational interactions between bodies. Given a fixed number of steps, the algorithm iterates over each pair of bodies to compute the forces, resulting in a complexity of $O(n^2)$. This quadratic complexity arises because each body is compared with every other body exactly once per step.

To empirically evaluate the performance of our implementation, we introduce the `measure_performance` function. This function measures the execution time for a fixed number of steps, allowing us to observe the practical computational demands as the number of bodies increases. The function is defined as follows:

```
def measure_performance(n_bodies, n_steps=10):
    data = init_world(n_bodies)
    start_time = time.time()

    for t in range(n_steps):
        force = np.zeros((n_bodies, 2))
        for i in range(n_bodies):
            for j in range(n_bodies):
                if i != j:
                    force[i] += interaction(data[i], data[j])
        for i in range(n_bodies):
            data[i] = update(data[i], force[i])

    end_time = time.time()
    return end_time - start_time
```

Results :

Figure 4: Measure Execution Time

The results from the `measure_performance` function indicate that the execution time increases quadratically with the number of bodies, which is consistent with the theoretical complexity of $O(n^2)$. This quadratic increase in execution time is expected due to the double loop over all pairs of bodies. As the number of bodies grows, the number of pairwise force calculations grows as the square of the number of bodies, leading to a significant increase in computational workload. This behavior underscores the need for optimized or parallel algorithms for large-scale simulations.

5 Parallel version MPI

5.1 Introduction to MPI and its Application in the N-body Problem

MPI (Message Passing Interface) is a standardized and highly efficient method for communication in parallel computing environments. It is widely used in high-performance computing for solving complex computational problems by dividing tasks across multiple processors.

In the context of the N-body problem, MPI offers a robust framework for handling the computational and data distribution challenges inherent in simulating interactions among a large number of bodies. By leveraging MPI's capabilities, we can significantly improve the performance and scalability of N-body simulations.

5.1.1 Overview of MPI in the N-body Problem

MPI facilitates the division of the N-body problem into smaller, manageable subproblems that can be processed in parallel. This approach not only accelerates computation but also allows for handling larger systems that would be infeasible with a sequential approach.

5.1.2 Plan for MPI Implementation

In the upcoming sections, we will delve into:

- The architecture of MPI and its suitability for the N-body problem.
- The implementation strategy for dividing the N-body problem into parallel tasks.
- Data distribution and communication patterns among processes using MPI.
- Performance evaluation and comparison between the sequential and parallel versions of the N-body simulation.

This discussion aims to provide a comprehensive understanding of how MPI can be effectively utilized to address the computational challenges of the N-body problem in a parallel computing environment.

5.2 The Parallel Algorithm

Algorithm 3 High-Level Overview of Parallel N-Body Solver Using MPI

1: **Initialize MPI Environment**: Establish communication context.
2: **Determine Parallel Environment Parameters**: Identify the number of processes and the rank of each process.
3: **Read Simulation Parameters**: Obtain the number of bodies and simulation steps from command-line arguments.
4: **Divide Simulation Data**: Partition the initial data among processes, ensuring each process handles a subset of bodies.
5: **Scatter Initial Data**: Distribute initial body data from the root process to all other processes.
6: **for** `each time step in the simulation` **do**
7: **Synchronize Data**: Share updated body positions and velocities among all processes.
8: **Calculate Forces**: Each process computes forces acting on its subset of bodies.
9: **Update Body States**: Update positions and velocities of bodies based on calculated forces.
10: **Synchronize Updated Data**: Ensure all processes have the latest state of all bodies.
11: **Gather Final Data**: Collect the final state of all bodies at the root process.
12: **Output Results**: (Optional) Root process outputs the final state of the simulation.

5.3 Implementation

```
# Initialize MPI
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

# Command-line arguments for number of bodies and steps
n_bodies = int(sys.argv[1])
n_steps = int(sys.argv[2])

# Determine the size of each local chunk
local_n = n_bodies // size
all_data = np.empty((n_bodies, 6), dtype='f')

# Prepare a local array to receive the scattered data
local_data = np.empty((local_n, 6), dtype='f')

# Scatter the data from the root process to all processes
if rank == 0:
    init_data = init_world(n_bodies)
else:
    init_data = None
comm.Scatter(init_data, local_data, root=0)

for t in range(n_steps):
```

```
    # Synchronize all_data before force calculation
    comm.Allgather(local_data, all_data)

    # Step 3: Force Calculation
    local_force = np.zeros((local_n, 2))
    for i in range(local_n):
        for j in range(n_bodies):
            if rank * local_n + i != j:  # Avoid self-interaction
                local_force[i] += interaction(local_data[i], all_data[j])

    # Step 4: Data Update
    for i in range(local_n):
        local_data[i] = update(local_data[i], local_force[i])

    # Synchronize updated local_data across all processes
    comm.Allgather(local_data, all_data)

# Step 6: Output and Iteration
# Gather data at root for final output or further processing
final_data = np.empty((n_bodies, 6), dtype='f') if rank == 0 else None
comm.Gather(local_data, final_data, root=0)

if rank == 0:
    print(signature(final_data))
```

5.4 Results

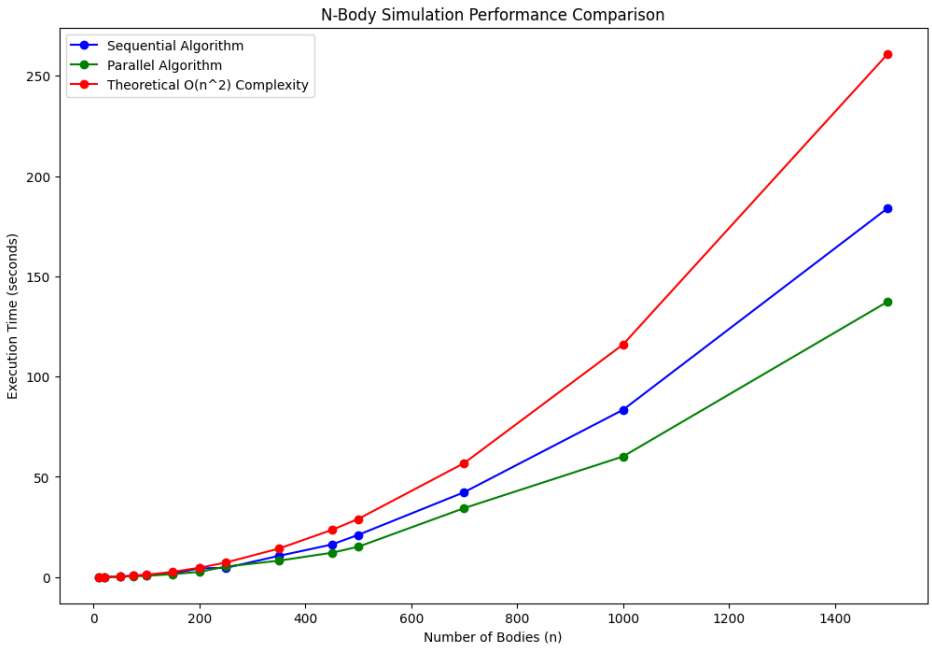


Figure 5: Enter Caption

Observations

- The time plot illustrates the effectiveness of parallel computing in handling large-scale n-body simulations. The distributed computation allows for a more scalable solution.
- The improvement in execution time underscores the advantage of MPI's data distribution and task parallelism, effectively reducing the computational load on individual processors.

5.5 Exploiting Symmetry in N-Body Simulations

The efficiency of the simulation can be greatly enhanced by exploiting the symmetry inherent in Newton's third law of motion. This law states that the force exerted by one body on another is equal in magnitude and opposite in direction to the force exerted by the second body on the first. This symmetry implies that for every pair of bodies, the force calculation needs to be performed only once, reducing the computational load significantly.

5.5.1 Algorithm

The sequential algorithm, leveraging this symmetry, is outlined as follows:

Algorithm 4 Sequential N-Body Simulation Exploiting Symmetry

for $t = 0$ **to** NB_STEPS **do**
  force $= [[0, 0]$ for $_{\text{in range}}(N)]$ ▷ Initialize force for each body
  for $i = 0$ **to** N **do**
    for $j = 0$ **to** i **do**
      force_j_on_i $=$ interaction(data$[i]$, data$[j]$)
      force$[i]$ $=$ force$[i]$ $+$ force_j_on_i
      force$[j]$ $=$ force$[j]$ $-$ force_j_on_i
  for $i = 0$ **to** N **do**
    data$[i]$ $=$ update(data$[i]$, force$[i]$) ▷ Update each body's state

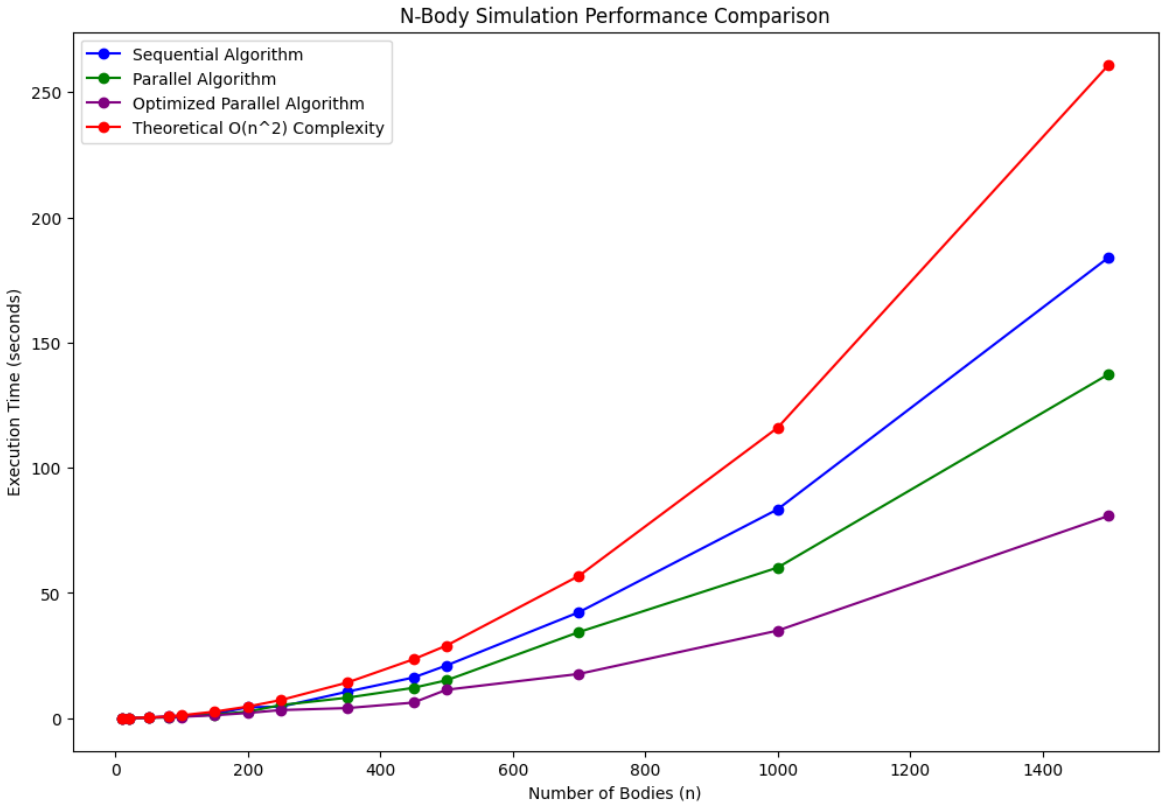
5.5.2 Python Implementation

The corresponding Python implementation of the above algorithm is provided below.

```
# Each process will work on a portion of the bodies
local_n = n_bodies // size
start_index = rank * local_n
end_index = start_index + local_n

# Initialize the world
if rank == 0 :
    all_data = init_world(n_bodies)
else:
    all_data = np.empty((n_bodies, 6), dtype='f')

# Broadcast the initialized world to all processes
comm.Bcast(all_data, root=0)

for t in range(n_steps):
    # Each process calculates a portion of the forces
    local_force = np.zeros((n_bodies, 2))
    for i in range(rank, n_bodies, size):
        for j in range(i):
            force_j_on_i = interaction(all_data[i], all_data[j
])
            local_force[i] += force_j_on_i
            local_force[j] -= force_j_on_i
```

```
23
24      # Sum up the force contributions from all processes
25      total_force = np.zeros((n_bodies, 2))
26      comm.Allreduce(local_force, total_force, op=MPI.SUM)
27
28      # Update the data
29      if rank == 0:
30          for i in range(n_bodies):
31              all_data[i] = update(all_data[i], total_force[i])
32
33      # Broadcast the updated data to all processes
34      comm.Bcast(all_data, root=0)
35
36  # Only the root process prints the signature
37  if rank == 0:
38      print(signature(all_data))
```

Results



- The time plot provides compelling evidence of the optimization's impact. By exploiting symmetry in force calculations, the algorithm significantly reduces the number of necessary computations.
- This approach not only enhances performance but also underscores the importance of algorithmic optimizations in parallel computing environments.