epcc


# Parallel Barnes-Hut N-body Toy Galaxy Simulation for Outreach

Xinyi Ding

August 16, 2024

MSc in High Performance Computing
The University of Edinburgh
2024

## Abstract

The field of computational astrophysics increasingly relies on simulations to understand the formation and evolution of galaxies. However, these simulations often require significant computational resources, making them inaccessible to many educational institutions and smaller research facilities. This project aims to address this limitation by developing a galaxy simulation model that can run efficiently on resource-constrained hardware, specifically for a WeeArchie cluster consisting of Raspberry Pi nodes.

The main goal of this research was to implement a parallelized Barnes-Hut N-body galaxy simulation model that is required to be able to perform complex astrophysical computations on resource-constrained hardware. The project involved developing a serial version of the galaxy simulation and then parallelizing it using a hybrid MPI + OpenMP framework. As WeeArchie was not available, initial testing and performance evaluation was performed on the Cirrus computing cluster. Various configurations of MPI processes and threads were explored to optimize runtime performance. The project has evaluated the scalability of the simulation using three different galaxy sizes, specifically $10^4$, $10^5$, and $10^6$ particles. The results of the study show that the current parallel simulation model can achieve significant performance improvements and scalability on multiple nodes, with up to eight nodes used in the test. The best performance configuration was found to be using two MPI processes per node, each with 18 threads, which maximizes the utilization of each node's resources.

This project aims to overcome the limitations of running complex simulations on resource-constrained hardware by developing a galaxy simulation model designed for a WeeArchie cluster (composed of Raspberry Pi nodes). The model is intended for outreach purposes, with the goal of sparking interest among students in how supercomputers work.

# Contents

# List of Tables

# List of Figures

# Acknowledgements

I would like to express my sincere gratitude to everyone at the EPCC for their invaluable teaching and guidance over the past year. The experience has been truly rewarding and has greatly enriched my knowledge and skills.

I am especially grateful to my supervisors Darren and Laura for their excellent support and guidance throughout the project. Their insights and expertise were instrumental in my research.

A special thanks to Daniyal for his patience and assistance in handling my numerous requests for the Cirrus CPU core budget.

Lastly, I would really like to thank my family for helping me get into such a reputable university and program, as well as for motivating me to finish my master's degree.

# Chapter 1

# Introduction

In the past few decades, simulations have become a crucial tool in astrophysics, significantly enhancing our knowledge of the cosmos. By simulating galaxies, researchers can study the evolution of their structures, behavioral patterns, and interaction dynamics. Significant insights have been gained by modeling these intricate celestial systems more thoroughly and extensively thanks to developments in computing technology. However, the simulation of the complicated galaxy presents challenges, particularly in terms of computational resource usage and efficiency.

As shown in my feasibility study [1], many studies use expensive and efficient GPU hardware to improve computational efficiency in order to explore large galaxies, and many studies use advanced high-performance computing clusters to achieve the same purpose. However, there is a dearth of research on galaxy simulations' computational efficiency and scalability on resource-constrained systems. State-of-the-art simulation methods, while powerful, often require large amounts of computational resources, beyond the capabilities of educational institutions or small research organizations. Accessibility to advanced computing tools is limited by financial and resource constraints, posing challenges to memory efficiency, computational scalability, and real-time data processing. These issues hinder the wider application and educational use of these simulations.

This project aims to address these challenges by developing a customized Barnes-Hut N-body galaxy simulation model tailored for WeeArchie, a Raspberry Pi cluster. The project focuses on creating a model that simulates the evolution of disk galaxies, using a hybrid MPI + OpenMP framework to improve computational efficiency and memory usage. This approach makes detailed and dynamic simulations more accessible and practical in resource-constrained environments. The project has broader implications beyond its immediate technical goals. It can improve educational activities by democratizing access to galaxy simulations, which allow educators and students to interact hands-on with complicated astrophysical ideas.

This dissertation is organized into several chapters. It begins with a targeted study of related work in the field of galaxy simulations, followed by a detailed exploration of the code implementation process and the methodological framework adopted. Subsequent chapters analyze the results, highlighting the effectiveness and limitations of the approach. The report concludes with a discussion of potential future work, exploring ways to further enhance the model and its applications. This project seeks to further the more general objective of increasing the usability and accessibility of galactic simulations for a diverse set of users by tackling the difficulties associated with resource-intensive simulations and enhancing scalability. Through this work, we hope to inspire further innovation and exploration in the field of computational astrophysics.

# Chapter 2

# Background

This chapter discusses the two main algorithms used in this project: the N-body method and the Barnes-Hut algorithm. These methods form the foundation of our galaxy simulation and are crucial for understanding the dynamics of complex astrophysical systems.

## 2.1 The N-body Method

The N-body method is a computational approach used to simulate the behavior of physical systems by approximating particle movements. It models the dynamics of a physical system using a collection of discrete particles and the interactions among them [2]. These particles can represent various entities, such as atoms in a molecule, planets in a solar system, or stars in a galaxy. The forces between particles may include gravitational or electrostatic interactions. The simulation proceeds by updating the properties of the particles at each time step, typically including force, acceleration, velocity, and position. This iterative process enables researchers to study the temporal and spatial evolution of physical systems.

Since Aarseth [3] pioneered the development of direct N-body codes for star cluster simulations in the 1960s, the N-body method has been widely developed and applied in astrophysics. In recent years, numerous studies have focused on expanding the simulation scale and introducing innovative techniques. The Outer Rim Simulation [4] was introduced in 2019 by the upgraded Hardware/Hybrid Accelerated Cosmology Code (HACC) [5], which became one of the largest high-resolution N-body simulations at the timefeaturing 3.6 trillion ($3.6 \times 10^{12}$) particles. In 2020, Neureiter et al. [6] presented the SMART model, which combines Schwarzschild 3D implementation with a three-axis N-body simulation. The model demonstrates the ability to accurately recover

galaxy properties, further improving the accuracy and applicability of simulations. In the following years, Ishiyama et al. [7] proposed the Uchuu simulation, which used the GreeM code [8] to track the evolution of dark matter halos and subhalos of different scales. Garrison et al. [9] developed ABACUS, a fast and accurate cosmological N-body code based on static multipole grid gravitational potential calculations, and it was successfully used in the ABACUSSUMMIT simulation involving $6 \times 10^{13}$ particles.

Furthermore, the susceptibility of thin stellar disks in intermediate- and low-mass interactions was investigated by Anta et al. [10], in 2023, using high-resolution N-body simulations of the FCC 170 model. The model represents an almost edge-on galaxy in the Fornax cluster with multiple components, including a nuclear stellar disk (NSD), and the N-body realization of the model is believed to be useful as a realistic description of real galaxies in merger simulations. In the same year, critical processes including mass segregation and stellar capture have been captured in studies of the co-evolution of stars and supermassive black holes during galaxy mergers using direct N-body simulation approaches [11].

## 2.2 Barnes-Hut (BH) Algorithm

The Barnes-Hut algorithm, introduced by Barnes and Hut [12], reducing the complexity of the N-body method from $O(n^2)$ to $O(n \log n)$ by grouping nearby objects and treating them as a single entity for force calculations, using their center of mass. This method divides the region containing particles into multiple sub-regions (four in two-dimensional space and eight in three-dimensional space), which significantly reduces the time complexity of N-body simulation, making it highly effective for handling large-scale galaxy simulations. The Barnes-Hut algorithm is one of the most widely used hierarchical methods for solving N-body problems [13].

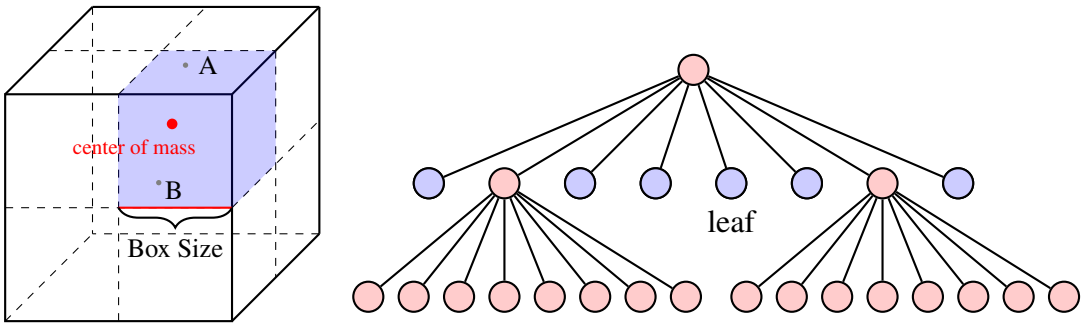



Figure 2.1: Illustration of the Barnes-Hut algorithm showing (left) subdivision of space into octants and (right) the corresponding octree structure.

### 2.2.1 Octree Construction of BH

Figure 2.1 shows the mechanism of the use of the BH algorithm in three-dimensional space. Initially, the space is considered a single cubic region. The algorithm then divides this cubic space into eight sub-cubic regions, each treated as a `node`. The highlighted purple region in the left figure represents one such node. When a node contains more than one particle, it is further subdivided into eight smaller cubes, continuing recursively. This process builds the octree shown in the Figure 2.1 on the right. Each point in the octree represents a subspace (i.e., a node). When a node cannot be further subdivided and contains only one particle, it is considered a `leaf`.

### 2.2.2 Force Calculation

The tree is traversed from the root to the tip to compute the gravitational force acting on a particular particle. If a node is far enough away from the particle under consideration, the particles within that node are treated as a single entity, with their combined mass and center of mass representing the entire group. The Figure 2.1 (left) shows an example of how center of mass is defined, if a node contains only particles `A` and `B`, the center of mass will be the midpoint between `A` and `B`. This approximation can significantly reduce the number of direct computations required. If a node is too close, the algorithm does not use the center of mass approximation but rather traverses deeper into the node's children. This continues until the node is subdivided into leaf nodes or until the subdivisions are considered sufficiently far to use their center of mass for calculations. This decision is typically based on the ratio of the size of the region represented by the node (marked as the `box size` in the left of Figure 2.1) to the distance between the particle and the node's center of mass. This method drastically reduces the number of interactions that need to be computed explicitly, from $O(n^2)$ in a direct-sum approach to $O(n \log n)$ in the Barnes-Hut method, enhancing computational efficiency especially in large-scale simulations.

### 2.2.3 Application and Parallelization

The Barnes-Hut algorithm is widely used in various astrophysical N-body simulations, including well-known examples such as GADGET-2 [14], Arepo [15] and PKDGRAV3 [16].

Considerable work has been done in both GPU and CPU parallelization to enhance the performance of the Barnes-Hut algorithm. Burtscher et al. [17] presented the first efficient CUDA implementation of the tree-based Barnes-Hut N-body algorithm that was entirely GPU-implemented. Hamada et al. [18] also developed a new tree-based code for Barnes-Hut that can perform N-body cosmological simulations on PCs equipped

with regular GPUs, achieving significant performance improvements over standard PC clusters connected via ethernet switches.

There has been some dedicated research focused on the parallelization of the Barnes-Hut tree code to enhance its efficiency on modern computing platforms. One of the earliest efforts was in 1992 when Warren and Salmon [19] used orthogonal recursive bisection (ORB) to dynamically distribute the workload across processors, achieving effective parallelization of the tree code. The following year, they introduced a hashed octree method [20] that used domain decomposition, employing hash tables to facilitate global data access and enable distributed memory operations on parallel systems by requesting keys for non-local data access. In 1994, Grama et al. [21] developed two parallel formulations of the Barnes-Hut algorithm for N-body simulations, which are particularly suitable for simulations with irregular particle density. One approach involves static domain decomposition, implicit load balancing, and collective communication. Another approach combines static domain decomposition with dynamic load balancing using Morton sorting. Both methods showed acceptable load balancing and high performance, but the study focused on two-dimensional astrophysical simulations, and further research is needed to extend the results to three-dimensional simulations and other scientific areas.

A new version of the Barnes-Hut tree code (PEPC) is presented by Speck et al. [22]. It implements a Parallel Full Approximation Scheme in Space and Time (PFASST) that outperforms conventional spatial parallel techniques by combining temporal parallelism with spatial decomposition (MPI/PThreads). Gangavarapu et al. [23] also developed an efficient, high-performance N-body simulation method using the Barnes-Hut algorithm, OpenMP, and CUDA, focusing on 2D implementations and single-node performance.These techniques have the potential to significantly advance the discipline if they can be expanded upon and used to tackle a wide range of scientific and industrial issues in large-scale N-body simulations, such as fluid dynamics, astrophysics, and plasma simulations.

In this project, we implemented the Barnes-Hut algorithm using a hybrid MPI + OpenMP approach, focusing on CPU parallelization. Although there are notable performance benefits to GPU implementations, as demonstrated by Burtscher et al. [17] and Hamada et al. [18], the primary goal of this work was to guarantee compatibility with the WeeArchie cluster, which is made up of Raspberry Pi nodes with limited resources. Given the hardware limitations, particularly the lack of dedicated GPU resources on WeeArchie, we chose to prioritize a CPU-based parallelization strategy. Additionally, considering the complexity of implementing domain decomposition as proposed by Warren and Salmon [19], and due to time constraints, we opted for a simpler data decomposition approach. This also allowed us to distribute the computational workload across multiple nodes using MPI, while maximizing performance within each node through OpenMP.

# Chapter 3

# Implementation

This chapter outlines the detailed implementation of the N-body galaxy simulation, including the serial code version, MPI version, and the hybrid MPI + OpenMP version. The serial and MPI versions of the code are adaptations of an open-source project [1] by Justin Rokisky [24]. Building on the successfully implemented MPI version, we integrated OpenMP to exploit shared memory parallelism, creating a hybrid parallel version. By combining the advantages of shared and distributed memory systems, this hybrid version can theoretically improve program scalability and computational efficiency.

## 3.1 Serial Code

### 3.1.1 Initialization

**Supermassive black hole (SMBH)**

In most large galaxies, a supermassive black hole resides at the center [25], with a mass ranging from 100,000 to several billion solar masses [26]. In our galaxy simulation, the central supermassive black hole is modeled as a single particle with a mass of $10^5$ for simplified calculations. This particle remains stationary at the origin (initial position $(0, 0)$ with an initial velocity of $0$), serving as the gravitational center for the whole galaxy. In addition, for simplicity in computation, all other particles are assigned a mass of $1$.

[1] Barnes-Hut Simulation using MPI, available on GitHub.

### Particle Positions

The Barnes-Hut Simulation from Rokisky [24] implemented a basic, randomly distributed gravitational N-body simulation. However, this project extends that model by simulating a galaxy-like structure. The primary modification involves generating particles in a way that mirrors a galaxy's distribution. Instead of placing particles within a random range, we aim to approximate an elliptical galaxy model by restricting their initial positions to a circular pattern. Specifically, we confine the particles to a circular disk with slight thickness. To achieve this, we set a radius range to control the galaxy's initial diameter. For each particle, apart from the central particle representing the supermassive black hole, a random $\theta$ angle value is assigned, and its position $(x, y, z)$ is calculated using Cartesian coordinates as follows:

$$\begin{aligned} x &= r \cdot \cos(\theta), \\ y &= r \cdot \sin(\theta), \\ z &\in \left[-\frac{H}{2}, \frac{H}{2}\right], \end{aligned} \tag{3.1}$$

Here, $r$ is a random value for the radial distance within the preset range, for example, $[1, 1000]$, and $\theta$ is a random value between $[0, 2\pi]$.The radius $r$ can be adjusted as the simulation scales, allowing for the representation of different galaxy sizes and structures. This method ensures the particles are distributed in a manner that resembles a galaxy's circular structure. The $z$-coordinate introduces a controlled thickness, represented by $H$, giving the galaxy a three-dimensional appearance. The chosen values for galaxy thickness aim to balance realism with computational efficiency, allowing for the simulation of disk-like structures observed in many galaxies. For instance, a thickness range such as $[10, 10]$ might be chosen to create a subtle yet noticeable thickness. This value is selected to reflect the typical thinness of elliptical galaxies while still ensuring that particles do not overlap excessively, thus maintaining simulation stability and realism.

### Particle Velocities

During the initial stages of development, we observed that without assigned initial velocities, the particles in our simulation tended to collapse toward the center. [2] This behavior does not accurately reflect the dynamics of most disk-like galaxies, where stars exhibit rotational motion that maintains the galaxy's stable structure. In reality, this rotational behavior emerges naturally during the galaxy formation process. To enhance the realism of the simulation, we assign appropriate initial velocities to all

[2] A demonstration of this behavior can be viewed in a video available on Gitlab.

particles except the SMBH. Drawing inspiration from the galaxy simulation code (CUDA version) [3] by Sandham [27], each particle is given a velocity that simulates rotational motion around the central SMBH, thereby maintaining the galaxy's stable structure. The velocity components are calculated using the formula for circular orbital velocity[28], ensuring that the particles follow circular orbits.

$$v = \sqrt{\frac{GM}{r}} = \sqrt{\frac{100000.0}{r}} \tag{3.2}$$

In the Formula 3.2, $G$ is the gravitational constant, which is set to 1 in the simulation to simplify calculations. $M$ represents the combined mass of the two orbiting bodies. In this context, $M$ refers to the mass of the central gravitational body (the SMBH) and the mass of the orbiting particle itself. Since each particle is assigned a mass of 1 for computational convenience, its contribution to $M$ is negligible. Therefore, $M$ is effectively equal to the mass of the SMBH, which is 100,000.

The velocity components in Cartesian coordinates are calculated as follows:

$$\begin{aligned} v_x &= R \cdot v \cdot \sin(\theta) \\ v_y &= -R \cdot v \cdot \cos(\theta) \\ v_z &= 0 \end{aligned} \tag{3.3}$$

Where $R$ represents the direction of motion, with $R = 1$ for clockwise motion. Setting $R$ to 1 for clockwise motion guarantees that particles move in stable orbits around the central mass, effectively simulating the rotational dynamics of a galaxy. This approach ensures that our simulations can convincingly and realistically reproduce galactic behavior.

### 3.1.2 Barnes-Hut Implementation

#### Tree Construction

An `octree` structure is required in order to use the Barnes-Hut approach to build 3D galaxies. A number of important components are defined for each `octree node struct` in the implementation, including an array of eight child nodes, a pointer to a particle (which is set to NULL unless the node is a leaf containing a particle), the number of leaf nodes within the node's region, the coordinates of the node's center of

[3] The project available on Bitbucket.

mass, the length of the cube's edges representing the node's spatial region, the total mass of all particles within the node's space (used for gravitational calculations), and the relative coordinates of the node within the entire cubic space (i.e., the entire galaxy being simulated).

Using this node structure, we construct the octree by inserting particles into their appropriate positions within the tree. As particles are inserted, the program updates the center of mass and the total mass of each node they pass through. If a node already contains a particle, the node will be further subdivided until each node contains at most one particle. This process ensures that the octree accurately represents the spatial distribution and mass properties of the particles within the simulated galaxy, facilitating efficient force calculations.

### Force Calculation

The force computation in our simulation is crucial for accurately modeling the gravitational interactions between particles. Each particle's force is computed based on the gravitational attraction from all other particles.

The force calculation uses Newton's law of gravitation. For a given particle $P_1$, the force components are updated by considering the influence of another particle $P_2$. The formula used is as follows:

$$F = G\frac{m_1 m_2}{r^2} = \frac{1}{r^2} \tag{3.4}$$

Where:

- $F$ is the gravitational force.
- $G$ is the gravitational constant, which is set to 1 in our simulation. This simplification is made to normalize the forces and simplify the calculations, making the simulation more straightforward to implement and analyze.
- $m_1$ and $m_2$ are the masses of the two particles. In our model, to reduce the computational burden and simplify the interactions, we assign a mass of 1 to all particles, with the exception of the SMBH.
- $r$ is the distance between the two particles.

### Threshold

There is a threshold parameter `THETA` in the Barnes-Hut algorithm, which is crucial to balance accuracy and performance in simulation results. For each octree node, if

the ratio of the edge length of the node (or box size, as marked in Figure 2.1) to the distance from the center of mass to the target particle is less than `THETA`, the center of mass of the node can represent all particles in the area for force calculation. Otherwise, the algorithm will recursively explore the child nodes of the node for more detailed calculations.

A smaller `THETA` value improves accuracy because it requires a more detailed computation within the child nodes, which leads to higher computational cost and a complexity close to $O(n^2)$, resulting in longer simulation times. Conversely, a larger `THETA` value leads to an algorithm implementation closer to $O(n \log n)$ complexity, which improves performance, but may lead to less stable results since the approximation may ignore local interaction details.

### Updating Particle Position and Velocity

To update the positions and velocities of particles in the simulation, we first define a time step $dt$, which determines the time increment for each update. In this implementation, $dt$ is set to 0.01. This small time step ensures the simulation remains stable and that changes in position and velocity are gradual. If $dt$ is too large (e.g., 0.1), especially in a small simulation volume (e.g., $10^4$ particles), it may cause particles to be ejected outward and dispersed due to excessive changes in velocity, potentially leading to numerical instability. Conversely, if $dt$ is too small (e.g., 0.001), each simulation step will only produce a negligible change, requiring more iterations and longer computational time to achieve meaningful results. The choice of $dt$ depends on the number of particles in the simulation. When simulating a larger number of particles (e.g. $10^6$ particles), a slightly larger $dt$ value (e.g., 0.1) can be used without compromising stability. We determine the appropriate $dt$ value for different size of galaxies by running a few test iterations to observe whether the particles move correctly. This trial-and-error approach helps ensure that the chosen $dt$ value maintains the balance between stability and computational efficiency.

Next, we calculate the acceleration ($a$) for each particle. The acceleration in the x, y, and z directions is determined by dividing the force in each corresponding direction ($F_x$, $F_y$, $F_z$) acting on each particle by its mass ($m$). The formula for calculating acceleration in each direction is the same:

$$a = \frac{F}{m} \tag{3.5}$$

The next step is to calculate the particle velocity at the midpoint of the time step, using a method similar to the Verlet integration of the velocity [29] to achieve better numerical accuracy. The midpoint velocity ($v_{\text{mid}}$) is calculated by adding the product of half the time step $dt$ and the acceleration $a$ to the current velocity ($v_{\text{old}}$):

$$v_{\text{mid}} = v_{\text{old}} + (0.5 \times dt \times a) \tag{3.6}$$

Using the midpoint velocity, we then update the particle's position. This is achieved by adding the product of the midpoint velocity ($v_{\text{mid}}$) and the time step ($dt$) to the current position ($p_{\text{old}}$):

$$p_{\text{new}} = p_{\text{old}} + (dt \times v_{\text{mid}}) \tag{3.7}$$

Finally, we update the particle's velocity ($v_{\text{new}}$) to reflect the effect of the computed acceleration over the entire time step. This is done by adding another half-step of acceleration to the midpoint velocity:

$$v_{\text{new}} = v_{\text{mid}} + (0.5 \times dt \times a) \tag{3.8}$$

These calculations ensure that particle motion is both accurate and stable throughout the simulation, correctly handling changes within each time step of the update.

## 3.2 Implementation of MPI

In this section, we analyze the version of MPI developed by Rokisky [24] that is specifically designed to run in parallel with 8 MPI processes.

### 3.2.1 Rokisky version

#### Particle Generation

The root process (rank 0) generates a set of random particles using the function `generate_random_particles`. Once generated, all particles are broadcast to the other processes using `MPI_Bcast`. Subsequently, each sub-process independently constructs a complete octree for all the generated particles.

#### Data Decomposition

Particles are partitioned according to their index and each process is responsible for a subset of particles determined by its rank. Each process will then traverse the octree to calculate the forces acting on its assigned particles and updates their velocities and positions accordingly.

### Synchronization

After updating their assigned particles, each process broadcasts its updated particles to all other processes using `MPI_Bcast`. Other processes update their own octrees with the content of the broadcast that was not sent by them. This step ensures that all processes have the latest data for all particles before proceeding to the next time step. An `MPI_Barrier` is used at the end of each time step to ensure that all processes have completed their calculations before continuing. This synchronization step is crucial for maintaining data consistency across processes and ensuring accurate simulation results.

### Limitations

Rokisky's parallel implementation uses a simple approach where each process builds a complete copy of the octree and independently computes the forces on its assigned block of particles, updating their positions and velocities. While simple and easy to implement, this approach has several obvious limitations:

- **Memory Usage**: Each process creates a complete octree for all particles, which means that as the number of particles increases, the memory requirements of each process also grow linearly. This may become a bottleneck on systems with limited memory. This concern is particularly relevant for this project, as the original goal is to develop a toy model specifically designed for Raspberry Pi based cluster WeeArchie. Raspberry Pis are characterized by their low cost and limited memory capacity. Efficient memory usage is crucial to ensure the model runs effectively on such constrained systems.
- **Static Particle Allocation**: Particle allocation is static and does not account for load balancing during runtime. As the simulation progresses, particles can become more clustered and less evenly distributed, leading to regions with high particle density. This uneven distribution can cause load imbalances and negatively impact parallel efficiency.
- **Communication Overhead**: After each time step, the process broadcasts the updated particle information to all other processes. However, as the number of processes increases, this all-to-all communication will significantly increase the communication overhead.
- **Cross-node Communication**: In a distributed system, cross-node communication is much slower than communication within a single node. Therefore, this all-to-all communication method will be even less applicable.
- **Computational Redundancy**: Each process has to update particles to its octree in turn after receiving the broadcast information, which costs extra time, especially in large scale simulations.

These drawbacks demonstrate the need for more advanced techniques to control memory consumption, lower communication overhead, and dynamically distribute the computational burden in order to enhance MPI-based Barnes-Hut N-body simulation performance.

**Simple Modifications**

Due to the challenges of domain decomposition for 3D models, we continue to use simple data decomposition to distribute tasks to sub-processes. We still use `MPI_Bcast` to broadcast particle initialization information, with each process creating a complete octree. This modified version primarily optimizes inter-process and cross-node communication.

Unlike Rokisky's version, which broadcasts updated particles directly and requires each process to locally rewrite the updated data, this version collects updated particle data from all processes to the root process using `MPI_Gather`. The root process then broadcasts the updated particle array back to all processes using `MPI_Bcast`.

This approach reduces the communication overhead by centralizing the update process, so that the performance of distributed systems can be improved by minimizing the frequency and amount of data exchanged between nodes.

## 3.3 Implementation of MPI+OpenMP

In this program, OpenMP takes advantage of the capabilities of multi-core CPU architectures to enhance parallelism within each MPI process. Specifically, OpenMP is applied to two key parts of the code: octree insertion and force calculation.

### 3.3.1 Octree Insertion

In the octree insertion phase, OpenMP is used to parallelize the loop responsible for inserting particles into the octree structure. The insertion for loop can be processed concurrently with particles using the `pragma omp parallel for` directive. Noting that the octree is a shared data structure, the `pragma omp critical` section is used to ensure that only one thread at a time performs the insertion operation, thus preventing race conditions and preserving data integrity.

### 3.3.2 Force Calculation and Particle Update

OpenMP is used once more to parallelize the loop that calculates the force acting on each particle and updates its location and velocity. Different particles can be processed individually by several threads in parallel using the `pragma omp parallel for` directive. By parallelizing these computationally intensive tasks, OpenMP can

accelerate the execution of these operations, enhancing the overall performance of the simulation.

### 3.3.3 Benefits and Limitations

By using OpenMP, the application may more efficiently utilize the processing capacity of contemporary multicore platforms, leading to increased computational efficiency in each MPI process. It retains the flexibility of the hybrid parallelism paradigm by fusing the shared memory parallelism of OpenMP with the distributed memory parallelism of MPI. This integration improves performance and speeds up computation time by efficiently scaling to multiple nodes and optimally utilizing the resources of each node.

While the final hybrid parallel version improves upon some limitations of Rokisky's implementation, it still has certain constraints that can be addressed in future work. The introduction of threads within nodes can lead to contention for shared resources, such as cache and memory bandwidth, potentially affecting performance. Proper tuning and balancing of MPI processes and OpenMP threads on each node are crucial and may vary depending on the system architecture and specific workload. Managing thread parallelism also introduces additional overhead. If each thread's workload is too small, the benefits of parallelization may be negated by this overhead. Additionally, the use of `pragma omp critical` sections can limit scalability within each process if octree insertion becomes a bottleneck, particularly when the number of threads is high relative to the operation's complexity.

# Chapter 4

# Methodology

## 4.1 Testing

The original plan for the project was to develop a galaxy simulation model tailored specifically for WeeArchie, a smaller-scale demonstration of how supercomputers work together to solve complex problems. However, WeeArchie was temporarily unavailable for testing, so an alternative approach was needed to continue developing and testing the simulation model. Initially, the backup plan considered replicating the WeeArchie setup using a virtual machine in the Edinburgh International Data Facility (EIDF). However, this approach involved complex configuration and could significantly delay the project's progress due to the complexity of replicating the WeeArchie environment on a virtual machine. It was ultimately decided to use the Cirrus supercomputer as an alternative to test the simulation model. Cirrus offers high-performance computing resources, including powerful CPUs, suitable for computationally intensive tasks such as galaxy simulations. This makes Cirrus a viable platform for testing and development without WeeArchie. The project was able to proceed with development and testing by using Cirrus, which made it possible for the project to go forward and for the simulation model to be tuned and verified on time.

### 4.1.1 Testing Parameters

The testing phase of the simulation involves adjusting several key parameters to optimize performance. These parameters are introduced in the script as special comments prefixed with `#SBATCH`, which provide information to the queue system for resource allocation and job execution. These parameters include the number of nodes, the number of MPI processes, the total number of tasks, and the number of threads per task. Each parameter plays a crucial role in determining the efficiency and scalability of the simulation.

- **Number of nodes** (`-nodes`): Indicates the total number of computing nodes used in the simulation, which has an impact on the capacity for distributed computing.
- **MPI per node** (`-tasks-per-node`): This defines the number of MPI processes running on each node. Adjusting this parameter affects the parallel processing capabilities, influencing how effectively the workload is distributed across the one node.
- **Total MPI processes** (`-ntasks`): The total number of MPI processes used by one execution is the product of the number of nodes and the number of MPI processes, that is `ntasks` = `nodes` × `tasks-per-node`.
- **Number of threads** (`-cpus-per-task`): This parameter indicates the number of threads assigned to each MPI process, impacting multi-threading efficiency. Another key consideration is that when setting `#SBATCH -cpus-per-task=x`, the environment variable `OMP_NUM_THREADS` must be configured to the same value using `export OMP_NUM_THREADS=x`

### 4.1.2 Performance Metrics

The performance of the simulations was mainly evaluated using the following metrics:

- **Runtime**: The primary metric for evaluating the performance of the galaxy simulations is the runtime. This is measured as the total time taken for the simulation to complete. By comparing runtimes across different configurations, we can assess the efficiency of the simulation and the impact of various computational parameters.
- **Speedup**: the speedup is also used to measure the improvement in simulation performance when using parallel computing resources. It is defined as the ratio of the runtime of the best-known serial execution to the runtime of the parallel execution. Mathematically, it is expressed as:

$$\text{Speedup} = \frac{\text{Runtime of serial simulation}}{\text{Runtime of parallel simulation}} \tag{4.1}$$

  Speedup offers insight into the effectiveness of parallelization and assists in identifying scenarios in which the increased use of computer resources results in decreasing returns.

### 4.1.3 Testing Initialization

To balance simulation efficiency and the realism of galaxy models, we chose to simulate a galaxy with $10^5$ particles during the testing phase. Additionally, several initial parameters, including the threshold (`THETA`) and other galaxy-specific data, are configured to ensure the uniformity of the simulation tests. Table 4.1 lists these initial setup parameters. Unless specified otherwise, all tests in this experiment will use this configuration as the initial setup.

| **Variables** | **Value** |
|---|---|
| Initial Radius of Simulated Galaxy ($r$) | [1, 1000] |
| Number of Particles ($N$) | $10^5$ |
| Time Step ($dt$) | 0.1 |
| Number of Iteration Steps | 50 |
| Mass of Central Particle | $10^5$ |
| Threshold (`THETA`) | 1.0 |

Table 4.1: Initial setup parameters for all tests.

### 4.1.4 Testing Cases

**Test 1: Serial Baseline**

Serial simulations will be tested with default parameters to establish baseline performance metrics. This test will provide a standard against which parallel simulation results can be compared, allowing us to quantify performance improvements.

**Test 2: MPI Processes Variation**

This test evaluates the performance impact of varying the number of processes while maintaining a single thread per process on a single node. This test was first performed to verify whether the simulated MPI parallel implementation can achieve consistent reductions in runtime as the number of processes increases. This verification is critical to determine whether each node can be fully utilized in subsequent performance tests using different numbers of nodes. Given that the maximum number of usable processes per node on the Cirrus system is 36, the test variables are detailed in the Table 4.2 below:

| **Num of Nodes** (–nodes) | 1 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **MPI Processes *per node*** (–tasks-per-node) | 1 | 2 | 4 | 8 | 12 | 16 | 24 | 32 | 36 |
| **Num of Threads** (–cpus-per-task) | 1 | | | | | | | | |
| **Total MPI Processes** (–ntasks) | 1 | 2 | 4 | 8 | 12 | 16 | 24 | 32 | 36 |

Table 4.2: Test configurations for processes variation in the parallelized simulation.

### Test 3: Nodes Variation

This test evaluates the performance impact of varying the number of nodes while maintaining a single thread per process with fixed number of processes. The number of processes will be determined with reference to the Test 2 result. The test variables are detailed in the Table 4.3 below:

| **Num of Nodes** (–nodes) | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| **MPI Processes *per node*** (–tasks-per-node) | 36 | | | | |
| **Num of Threads** (–cpus-per-task) | 1 | | | | |
| **Total MPI Processes** (–ntasks) | 36 | 72 | 144 | 288 | 576 |

Table 4.3: Test configurations for nodes variation in the parallelized simulation.

### Test 4: Threads Variation

This test determines the effect on performance of changing the number of threads per process while maintaining a constant number of nodes and processes. The number of nodes used in this test is based on the results from Test 3, selecting configurations that generally yield good performance without excessive resource usage. Additionally, this test sets the number of processes to 1 in order to facilitate testing of as many threads as possible while guaranteeing that processes and threads do not place more than one node in the shared memory segment, that is, the product of the number of processes per node and the number of threads per node does not exceed 36. Details about the configuration are shown in the Table 4.4 below:

| **Num of Nodes** (–nodes) | 8 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **MPI Processes *per node*** (–tasks-per-node) | 1 | | | | | | | | |
| **Num of Threads** (–cpus-per-task) | 1 | 2 | 4 | 8 | 12 | 16 | 24 | 32 | 36 |
| **Total MPI Processes** (–ntasks) | $8 \times 1 = 8$ | | | | | | | | |

Table 4.4: Test configurations for threads variation in the parallelized simulation.

### Test 5: Processes and Threads Combination

This test is designed to determine the optimal combination of the number of processes and threads that yields the best performance. The number of nodes used will be based on the results of Test 2. The detailed configurations are outlined in the Table 4.5 below:

### Test 6: Strong Scaling [30]

After determining the optimal combination of MPI processes and threads through previous tests, we performed strong scaling tests for three different problem sizes using this configuration. The purpose of these tests is to evaluate how simulation performance

| **Num of Nodes** (–nodes) | 8 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **MPI Processes *per node*** (–tasks-per-node) | 2 | 3 | 4 | 6 | 9 | 12 | 18 | 36 |
| **Num of Threads** (–cpus-per-task) | 18 | 12 | 9 | 6 | 4 | 3 | 2 | 1 |
| **Total MPI Processes** (–ntasks) | 16 | 24 | 32 | 48 | 72 | 86 | 144 | 288 |

Table 4.5: Test configurations for varying numbers of processes and threads.

improves as computing resources increase while keeping the problem size constant. By testing multiple problem sizes, we can understand how scaling efficiency varies with the complexity and size of the simulation, providing insight into the limits of scalability and the effectiveness of parallelization methods across different workloads. Given the limited budget for Cirrus core hours and the desire to achieve results on a large scale, we selected three different galaxy sizes, with $10^4$, $10^5$, and $10^6$ particles, respectively, to see the variation of our toy model. The simulation for $10^6$ particles remains within acceptable resource consumption limits, with each test consuming no more than 100 Corehs. For each scale, the simulation is run on 1, 2, 4, and 8 nodes separately. This approach allows us to assess the efficiency and scalability of the simulation across different problem sizes and node counts.

## 4.2 Data Collection

### 4.2.1 Simulation Data

Data collection for the simulation result is conducted after each simulation iteration. The primary output of each iteration is the three-dimensional position data of the particles, which is recorded in separate text files in the `pos` folder to ensure orderly storage and facilitate analysis. Each file is named with a suffix corresponding to the iteration number, enabling tracking of changes between iterations and allowing subsequent analysis of the evolution of particle positions throughout the simulation.

### 4.2.2 Performance Data

The data collection process for the galaxy simulation performance tests is automated using a shell script designed to run on the Cirrus HPC cluster. The process begins with configuring the SLURM job scheduler to allocate the necessary computational resources. For different tests, adjustments are made to parameters such as the number of nodes, processes, and threads. Each test case is repeated three times to ensure the consistency and reliability of the results, minimizing the impact of outliers or transient fluctuations during runtime. The time taken for each simulation is also recorded, and the average of the three runs is calculated and written to the corresponding CSV file.

## 4.3 Data Analysis

The data analysis is conducted using Python libraries, primarily `matplotlib` and `numpy`, within Jupyter notebooks to visualize the results.

### 4.3.1 Simulation Results Visualization

The visualization process begins by loading position data from multiple files in the `pos` folder, ensuring that these files are numerically ordered for coherent animation sequencing. In order to visually represent the outcomes of the 3D simulation, a figure including two subplots organized horizontally is created. One subplot displays the (x, y) coordinates, offering a top-down view. This perspective helps illustrate the overall trajectory and rotational dynamics of the galaxy as seen from above. The second subplot presents the (x, z) coordinates, which provides a side view of the galaxy. Since the (y, z) view does present a similar perspective, only the (x, z) plane is plotted to avoid redundancy. During animation updates, each subplot is refreshed with the corresponding coordinate pairs from the current frame's data file, providing dynamic visualization over time. Finally, the entire animation is compiled and saved as an MP4 file, offering a more convenient way to view results at different time points compared to GIF animations.

### 4.3.2 Performance Results Analysis

For performance data, we also use the Python `numpy` library to read data stored in CSV files and `matplotlib` to visualize them as line graphs for easy analysis. In addition to plotting the average values, we also include standard deviation as error bars on the graphs. This approach provides a clear representation of the variability in the results and allows for more informed interpretation of the simulation's performance across different configurations.

# Chapter 5

# Results and Findings

## 5.1 Simulation Results

This section presents the results of the galaxy simulation involving $10^5$ particles at the 1100th iteration, using the initial settings specified in Section 4.1.3. The visualizations offer distinct perspectives of the galaxy's structure and dynamics. The appendix contains a series of visualizations that visually track the movement and trajectory changes of particles throughout the simulation. For a comprehensive view of the entire simulation process, a complete video showcasing all 2000 iterations is available through our project repository on GitLab.

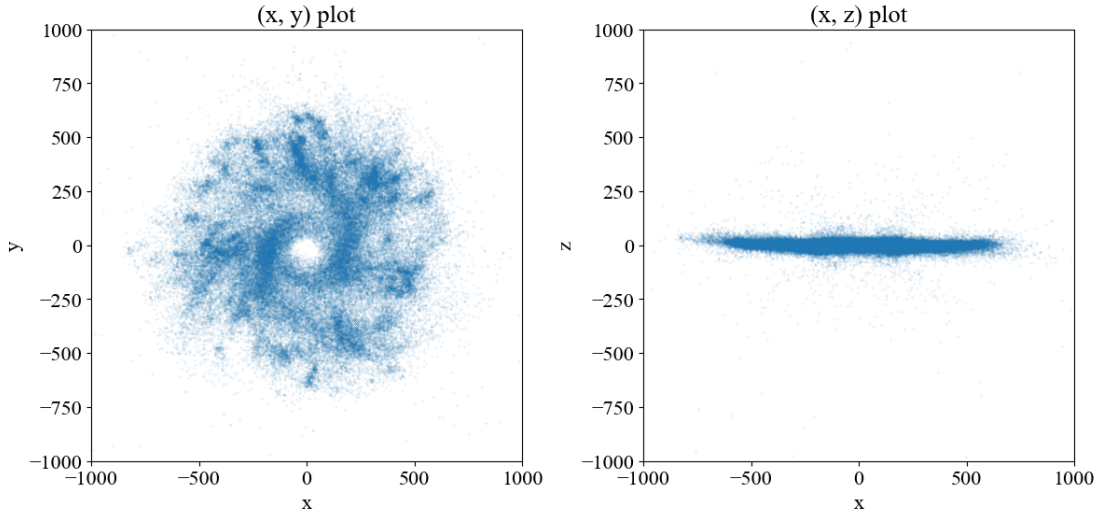



Figure 5.1: Galaxy simulation of $10^5$ particles, showing (x, y) and (x, z) views at the 1100th iteration.

The left image in Figure 5.1 displays a top-down view of the galaxy with (x, y) coordinates. This visualization reveals the evolution of a disk-like galaxy, highlighting the process where particles are drawn toward the center by the gravitational pull of the black hole. The distribution of particles shows a distinct spiral pattern, suggesting dynamic rotational motion around the center. The right image in Figure 5.1 provides a side view of the galaxy using (x, z) coordinates. From this perspective, the galaxy appears significantly flatter, highlighting the thinness of the galactic disk. The concentration of particles near the central plane emphasizes the galaxy's disk structure, with minimal vertical dispersion.

## 5.2 Performance Results

### 5.2.1 Serial Performance (Test 1)

Following the configuration declared in Test 1, we measured the runtime for a simulation containing $10^5$ particles to be approximately 468 seconds in total to complete 50 iterations. The runtime for every iteration is depicted in the Figure 5.2 below. It shows that the time required to complete each iteration gradually increases as the iterations proceed. In particular, it usually takes longer to get to the latter iterations. The first iteration was the fastest, taking around 5.9 seconds, as indicated by the green arrow in Figure 5.2. In contrast, the 41st iteration had the longest duration, at roughly 10.7 seconds, marked with a red arrow in the same figure. On average, each iteration took about 9.4 seconds. This trend is to be expected in a serial galaxy simulation. The variation in run time is due to the uneven distribution of forces between particles that need to be calculated in each iteration. As the simulation progresses, particles are attracted by the gravity of the central black hole and continue to gather towards the center, causing them to clump towards the center. While the `THETA` value remains fixed, the number of particle interactions that can be approximated decreases, and the number of interactions that require precise calculation increases, leading to longer running times.

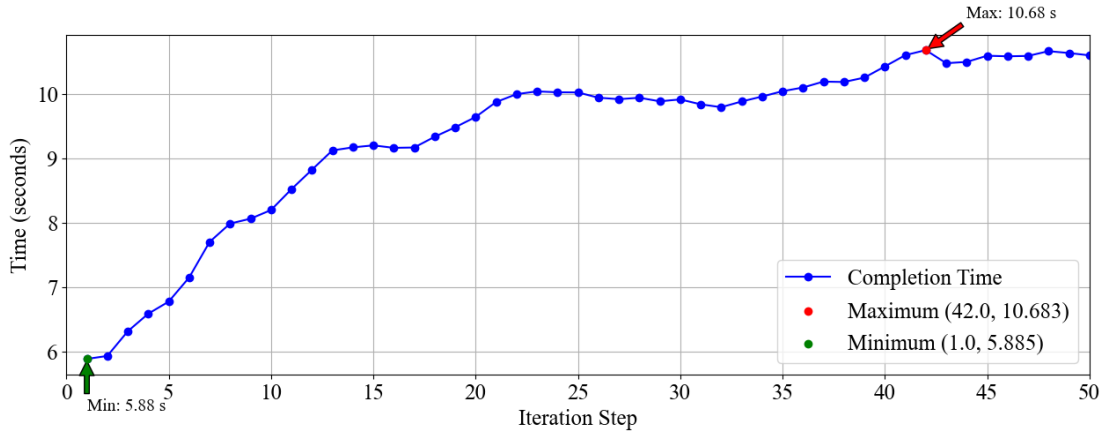
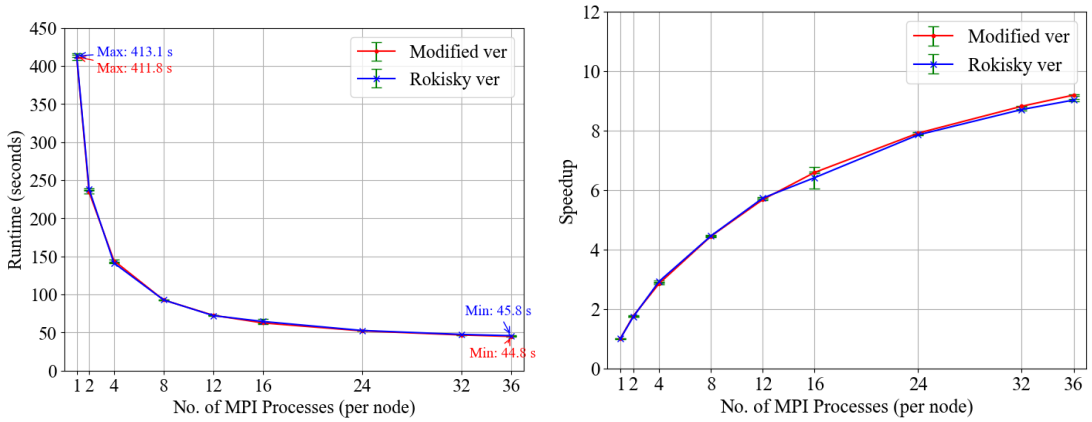



Figure 5.2: Runtime per iteration in the serial simulation of a galaxy with $10^5$ particles.

### 5.2.2 1 Node, Multiple Processes, 1 Thread (Test 2)

In Test 2, we evaluated two versions of the MPI implementation: the `Rokisky ver` and the `Modified ver`. Figure 5.3a illustrates the runtime of both versions when executed with varying numbers of processes on a single node with one thread. As the number of threads increases, the runtime for completing 50 iterations for both versions decreases significantly, dropping from over 400 seconds to under 50 seconds. The `Modified ver` achieves the lowest runtime of 44.8 seconds when using 36 threads. Additionally, we observe that the running timelines of the two versions in the left Figure 5.3a are basically overlapped, which indicates that the parallel effects on a single node in the two versions are generally the same.

In order to better compare the performance of the two versions, Figure 5.3b illustrates their speedup ratios. The two lines in the figure are basically coincident, but the speedup ratio of `Modified ver` (red line) is slightly higher. Overall, the speedup of both versions continues to increase as the number of processes increases from 1 to 36. This indicates that adding more processes generally results in better performance for both versions. As the number of processes approaches 36, both lines increase slightly. This behavior is common in parallel computing, where the benefits of adding more processes decrease due to overhead costs such as communication and synchronization delays between processes. In addition, the presence of error bars at various points (especially when processes number is 16) means that there is variability in the speedup measurements for these configurations. This variability may be caused by factors such as uneven work distribution, communication overhead, or fluctuations in the runtime environment, which are acceptable because they do not affect the overall trend.




(a) Runtime comparison of MPI versions

(b) Speedup comparison of MPI versions

Figure 5.3: Performance plots of MPI implementations on a single node.

### 5.2.3 Multiple Nodes, Fixed Processes, 1 Thread (Test 3)

To validate the limitations of the `Rokisky version` assessed in the Implementation chapter—specifically, that its performance in distributed systems (multiple nodes) is hindered by increased communication overhead—we tested this version using a fixed number of MPI processes per node (36) across multiple nodes. The test involved simulating a galaxy with $10^5$ particles, running a single iteration, and comparing the runtime to that of the `Modified version`. The Figure 5.4 below presents the results of the tests conducted with different numbers of nodes.

Compared to single-node execution times, the `Rokisky version` experiences a significant increase in runtime in a multi-node environment, with execution times growing as the number of nodes increases. This indicates insufficient handling of cross-node communication in the `Rokisky version`. In contrast, the `Modified version` optimizes distributed communication, maintaining strong performance across multiple nodes, and the running time is gradually shortened as the number of nodes increases.

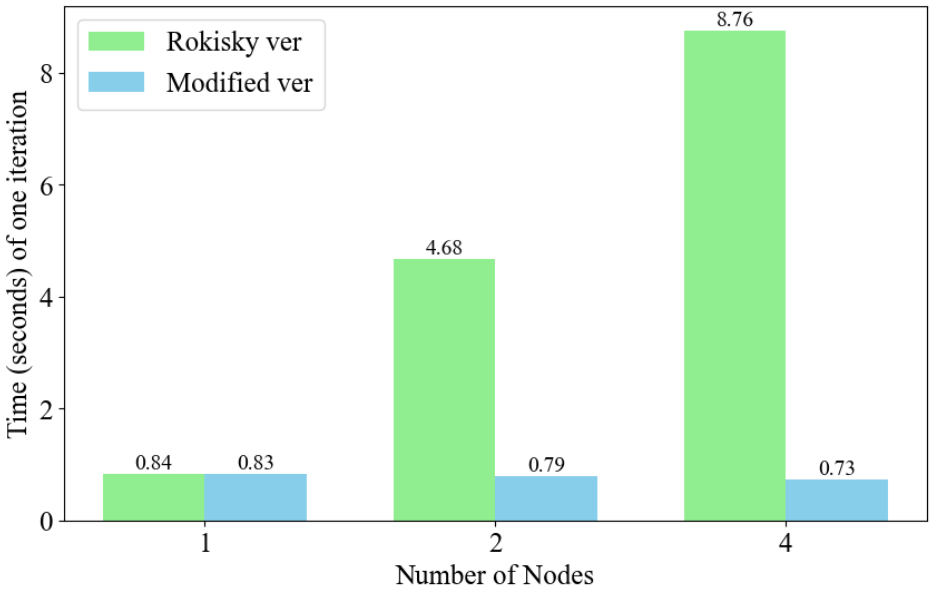



Figure 5.4: Comparison of execution time of one iteration for Rokisky and Modified versions across multiple Nodes.

#### Detailed Analysis of the Modified Version

Next is the result of Test 3 on the modified version. According to the result of Test 2, the performance is best when using 36 processes, so we test the operation of the modified version under 1-16 nodes (each node uses 36 processes). The specific runtime

performance is shown in Figure 5.5. It can be noted that the running time decreases with increasing node count, suggesting that the new MPI version performs well on distributed memory systems. The shortest time for simulating 50 iterations in the test is between 32.8-32.9s. Figure A.1 in appendix displays the speedup achieved with increasing node counts. The speedup is most noticeable between 1 and 4 nodes. Between 4 and 8 nodes, the speedup gain drops to around 0.06, and further reduces to 0.01 with 12 nodes. More concerning, the runtime slightly increases rather than decreases when 16 nodes are used. This slight increase in runtime when using 16 nodes could be due to increased communication overhead and load imbalance as the number of nodes grows.

The error bars in the charts reflect the variability in runtime measurements. Notably, larger error bars are observed at lower node counts (1 and 2 nodes), which could be due to external factors such as network latency and load variations. In general, the galaxy simulation operating on 8 nodes seems to be the most optimal design, balancing performance and resource use, taking into account the stability and performance outcomes. Moreover, compared to configurations with similar performance using 12 or 16 nodes, the 8-node setup reduces the demand on node resources.

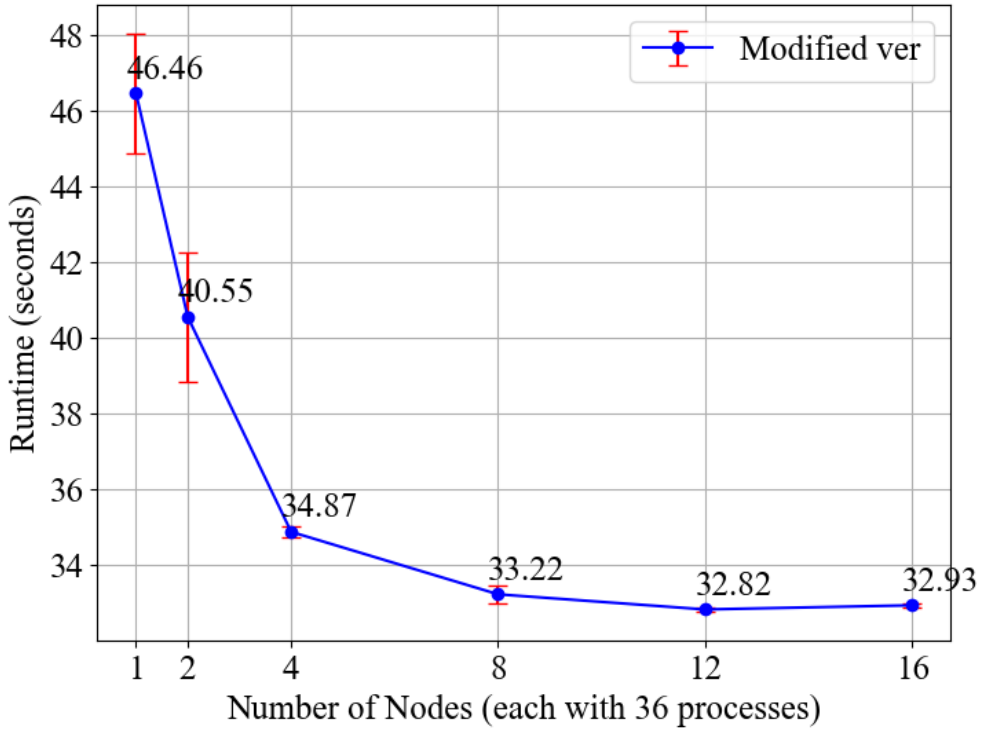



Figure 5.5: Runtime of the modified version across multiple nodes with 36 processes per node.

### 5.2.4 Fixed Nodes, 1 Process, Multiple Thread (Test 4)

After examining how the number of processes and nodes influences the parallel performance of the galaxy simulation, we conducted Test 4 to evaluate the impact of OpenMP parallelism on simulation performance. Based on the results from the previous test, we fixed the number of nodes at 8, with each node running 1 process. The runtime results are plotted in Figure 5.6. Initially, the runtime decreases sharply from 70.34 seconds with one thread to 42.16 seconds with four threads. The runtime decrease is less noticeable as the number of threads rises beyond four. The minimum runtime of 36.93 seconds is achieved with 24 threads. Beyond this point, the runtime with 32 threads is similar to or slightly higher than that with 16 threads, indicating that parallelization efficiency begins to reach its limit around 16 threads. The overhead of managing additional threads may start to offset the benefits of parallel processing, as evidenced by the increased runtime to approximately 40 seconds with 36 threads.

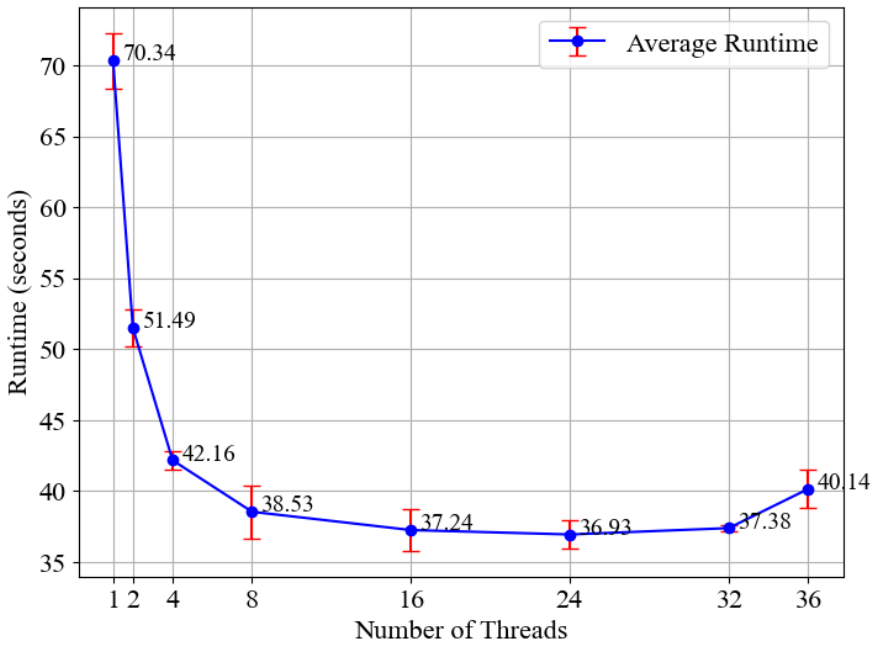



Figure 5.6: Runtime of the simulation across a range of thread counts.

### 5.2.5 Threads $\times$ Processes $=$ 36, 8 Nodes (Test 5)

In this test, we examined the impact of different configurations of threads and processes on parallel performance when using a total of 36 CPUs per node and run with 8 nodes. Figure 5.7 plots the runtime for each configuration, with the number of processes

increasing from left to right. The data points on the chart represent the average runtime from three test times.

The red bars in the figure highlight the two configurations with the shortest runtimes. The optimal configuration is using 2 processes, each with 18 threads, resulting in a runtime of 31.87 seconds. The second-best configuration involves using the full 36 processes, with each process having a single thread, which had an average runtime of 32.33 seconds.

However, as the number of processes increases and the number of threads per process decreases, it can be observed that the simulation time gradually increases. The configuration of 2 processes with 18 threads per process likely represents a more balanced optimization between threads and processes, allowing for more effective parallel processing while minimizing the overhead for each thread. This configuration perfectly aligns with the NUMA architecture of Cirrus nodes [31]. Each process is confined to a single NUMA region, ensuring that all memory accesses by threads within the process stay within that region. This minimizes costly memory access latencies that can occur when a thread in one NUMA region attempts to access memory in another NUMA region. As a result, this configuration makes more efficient use of the memory hierarchy and minimizes inter-process communication overhead.

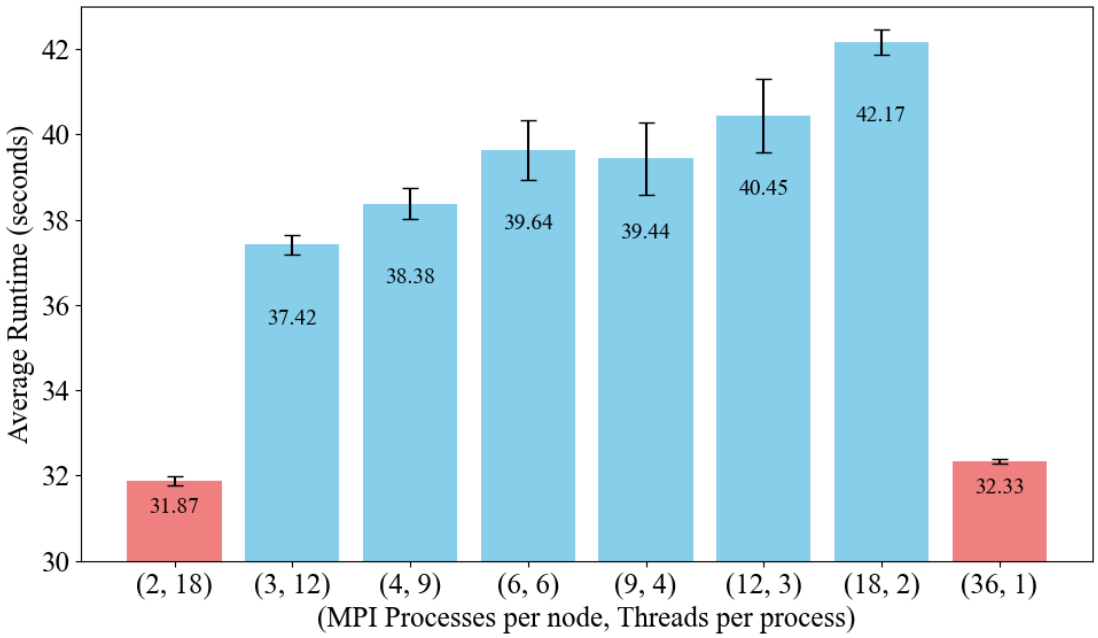



Figure 5.7: Average runtime for different configurations of threads and processes with 36 CPUs per node.

### 5.2.6 Strong Scaling (Test 6)

Based on the results from Test 6, we determined that using 2 MPI processes, each with 18 threads, allows for optimal utilization of a node, providing the best performance to simulate $10^5$ particles. Therefore, we conducted a strong scaling test with this configuration for each node. The speedup plot is shown in the Figure 5.8, with each line representing a different number of particles being simulated. The overall trend in speedup increases with the number of nodes, reaffirming the effectiveness of parallelization. However, the speedup gains are not linear, indicating the existence of parallelization overheads, such as communication and synchronization between nodes. As the number of nodes increases, the rate of speedup gain slightly decreases, which is typical due to the increased complexity of managing multiple processes and the inherent inefficiencies of network communication.

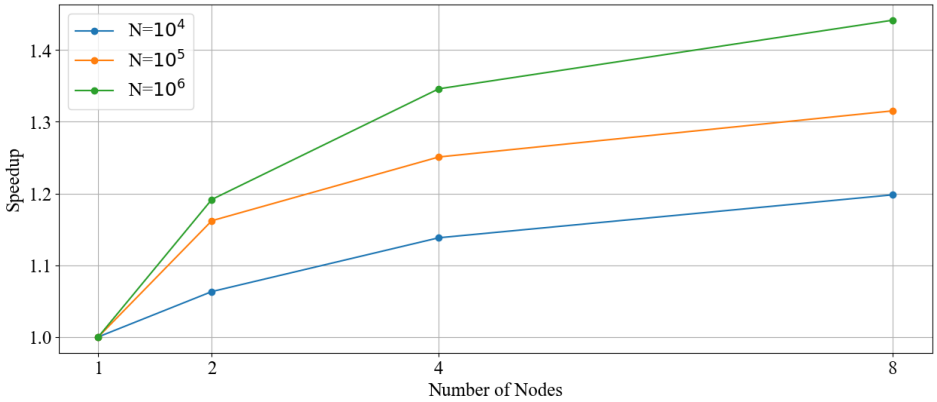



Figure 5.8: Speedup plot for strong scaling test with different particle counts.

It is noteworthy that simulations with more particles achieve higher speedups, as also shown in Table 5.1. The more particles there are, the greater the reduction in runtime when using more nodes, indicating that larger problem sizes benefit more from parallelization due to a more favorable computation to communication ratio. This suggests that for computationally intensive tasks involving a large number of particles, the communication overhead is relatively small compared to the total computation time. This finding highlights the potential of the project in practical applications, as galaxies typically contain a large number of particles, approximately $10^8$ [32].

| Number of Particles | Nodes = 1 (Average Time) | Nodes = 8 (Average Time) |
|---|---|---|
| $10^4$ | 3.87 s | 3.23 s |
| $10^5$ | 42.18 s | 32.07 s |
| $10^6$ | 517.29 s | 358.81 s |

Table 5.1: Comparison of average runtime for different particle sizes on 1 node and 8 nodes.

# Chapter 6

# Conclusion

In general, we successfully created a hybrid parallelized galaxy simulation model, produced a basic visualization of the simulation results, and evaluated performance by using alternative hardware (Cirrus), almost in accordance with the success criteria specified in the feasibility report [1].

The project first completed the development of the serial galaxy simulation project. After studying different projects, including Rokisky's Barnes-Hut simulation implementation and Sandham galaxy simulation initialization parameter settings, a simulation of the disk galaxy formation process was realized. Figure 6.1 depicts the evolution of the galaxy simulation over 1600 iterations using the model we developed. In developing the parallel version of the simulation, we encountered challenges due to the unavailability of the key hardware, WeeArchie. It was challenging to duplicate WeeArchie's setup using an EIDF virtual machine (VM) as a backup. Consequently, we used the Cirrus system for code debugging and parallel performance testing. Although this approach deviated from the project's original goal of targeting simulations on devices with limited memory and computational resources, we accomplished the fundamental task of building a parallelized galaxy simulation model. The performance testing explored the impact of different combinations of nodes, MPI processes, and threads on simulation performance. We determined that the optimal configuration was two MPI processes per node, with each process utilizing 18 threads. This setup provided the best runtime performance, and performance increased with the number of nodes. For larger problem sizes, using more nodes (up to eight in our tests) improved performance significantly. These results demonstrate the scalability of our parallel galaxy simulation program across eight nodes (288 cores) and show performance improvements even with two nodes (72 cores) compared to the serial version.

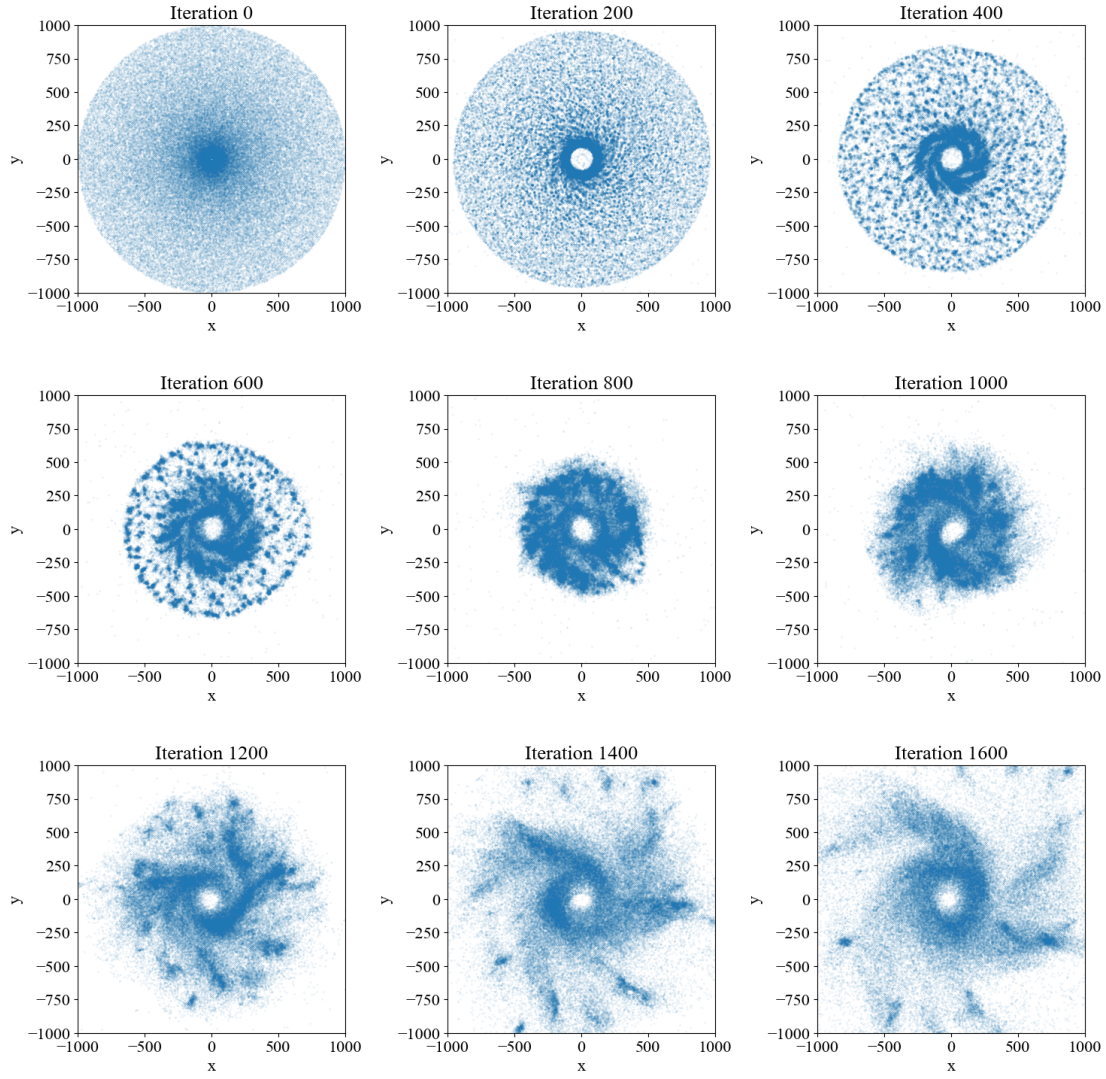



Figure 6.1: Visualization of the galaxy simulation process at different iterations, starting from an initial disk-like distribution (Iteration 0) to a more evolved spiral structure (Iteration 1600). The figure illustrates how the particles gradually cluster towards the center and form spiral arms as the simulation progresses.

The final goal of this project is to make the program run on WeeArchie. However, WeeArchie has only 16 compute nodes, each with 4 cores and 1 GB of memory per node, which is significantly smaller than Cirrus, which offers 256 GB of memory per node. As a result, further optimization may be needed to adapt the project for WeeArchie's limited resources. Due to time constraints and the unavailability of WeeArchie, we have not yet fully achieved the project's objectives. With more time and access to WeeArchie, this project could be further optimized to meet our original goals.

This project has laid the groundwork for developing a parallelized, executable program for galaxy simulations in resource-constrained environments. Once adapted to WeeArchie, it has the potential to demonstrate how complex simulations can be effectively performed using low-cost hardware configurations, making fascinating galaxy parallel simulation models accessible for educational purposes. During the initial phase of building the serial galaxy simulation, more time was spent than anticipated. This was due to the omission of the supermassive black hole (SMBH) in the initial galaxy construction and some errors made during the initialization of particle velocities. If I were to redo this project, I would conduct more detailed early research on galaxy simulations to avoid missing critical elements like the SMBH and to prevent the missteps made in initialization. Additionally, securing early access to WeeArchie would be crucial to allow for more extensive testing and optimization tailored specifically to its hardware constraints. Alternatively, identifying smaller devices that align more closely with the project's scope would also be important. This would prevent us from becoming constrained during the development of the parallel version due to a lack of testing equipment.

# Chapter 7

# Future Work

The primary focus of future work will be to adapt the galaxy simulation model to run efficiently on WeeArchie. This step is crucial for demonstrating the model's capability to perform complex simulations on low-cost hardware configurations. Given WeeArchie's constraints of 1 GB of memory per node, optimization of the MPI parallelization is essential to ensure efficient memory usage. In order to run WeeArchie on its constrained storage capacity, memory needs must be lowered, which could be achieved by implementing a more complex domain decomposition mechanism.

Another area of focus will be optimizing parallel communication. The current implementation uses `MPI_Bcast` and `MPI_Gather` for communication between nodes. Future work could explore alternative communication strategies, such as using `MPI_Reduce` for more efficient data aggregation or exploring non-blocking communication methods like `MPI_Isend` and `MPI_Irecv` to overlap computation and communication, thereby reducing latency and improving overall performance.

The project implementation also mentioned the importance of the threshold parameter (`THETA`) used in the Barnes-Hut algorithm. This parameter can significantly affect the accuracy of the simulation, as larger theta values lead to more force approximations and thus may lead to less precise results. Therefore, a thorough investigation of how different theta values affect the simulation results would be of great benefit to future work. This research may involve systematic testing to determine the optimal theta value that balances accuracy and computational efficiency.

Lastly, developing a user-friendly interface for the simulation program could make it more accessible to educators and researchers who may not have extensive programming experience. This could involve creating a graphical user interface (GUI) that allows users to set simulation parameters, run simulations, and visualize results without needing to interact with the code directly.

# Appendix A

# Test 4 Speedup Plot

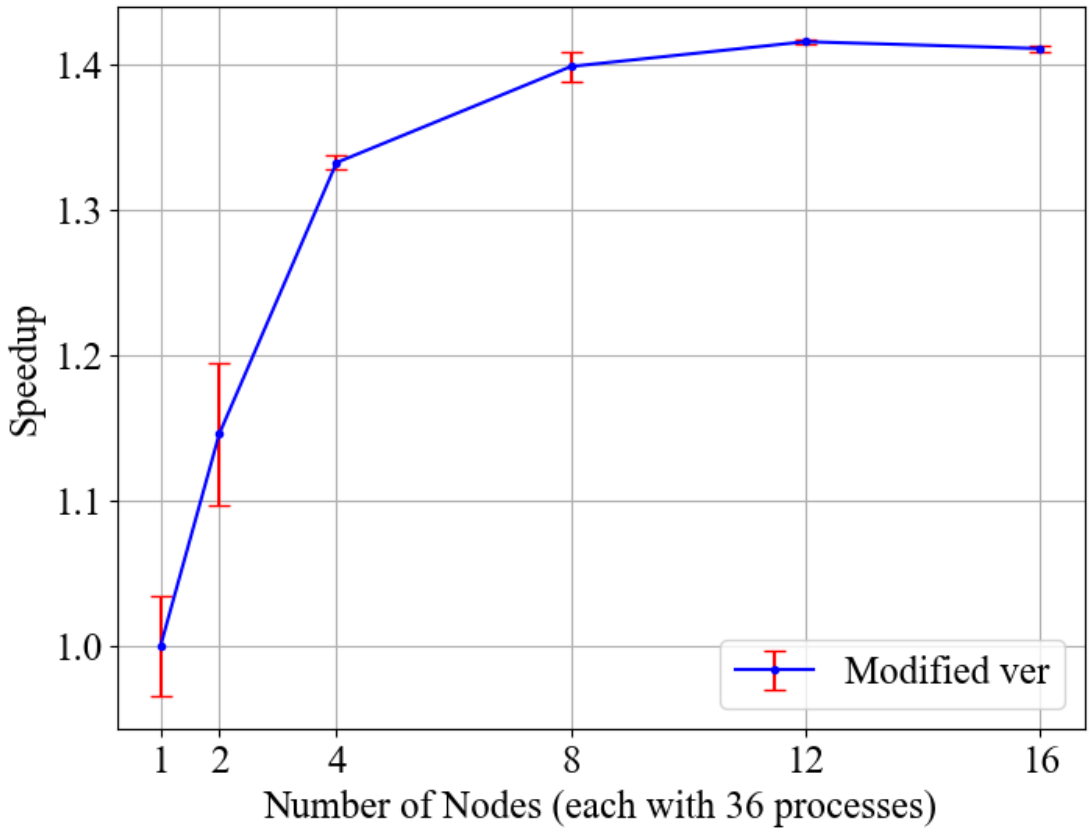



Figure A.1: Speedup of the modified version across multiple nodes with 36 processes per node.

# Bibliography


[1] X. Ding, "Galaxy exploration: Accessible galaxy simulations in education and outreach with wee archie," 2024.

[2] A. Y. Grama, J. Fogarty, H. Aktulga, and S. Pandit, *N-Body Computational Methods*, pp. 1259–1268. Boston, MA: Springer US, 2011.

[3] S. J. Aarseth, "Dynamical evolution of clusters of galaxies, I," , vol. 126, p. 223, Jan. 1963.

[4] K. Heitmann, H. Finkel, A. Pope, V. Morozov, N. Frontiere, S. Habib, E. Rangel, T. Uram, D. Korytov, H. Child, *et al.*, "The outer rim simulation: A path to many-core supercomputers," *The Astrophysical Journal Supplement Series*, vol. 245, no. 1, p. 16, 2019.

[5] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, and K. Heitmann, "Hacc: extreme scaling and performance across diverse architectures," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), Association for Computing Machinery, 2013.

[6] B. Neureiter, J. Thomas, R. Saglia, R. Bender, F. Finozzi, A. Krukau, T. Naab, A. Rantala, and M. Frigo, "SMART: a new implementation of Schwarzschild's Orbit Superposition technique for triaxial galaxies and its application to an N-body merger simulation," *Monthly Notices of the Royal Astronomical Society*, vol. 500, pp. 1437–1465, 10 2020.

[7] T. Ishiyama, F. Prada, A. A. Klypin, M. Sinha, R. B. Metcalf, E. Jullo, B. Altieri, S. A. Cora, D. Croton, S. de la Torre, D. E. Millán-Calero, T. Oogi, J. Ruedas, and C. A. Vega-Martínez, "The Uchuu simulations: Data Release 1 and dark matter halo concentrations," *Monthly Notices of the Royal Astronomical Society*, vol. 506, pp. 4210–4231, 06 2021.

[8] T. Ishiyama, T. Fukushige, and J. Makino, "GreeM: Massively Parallel TreePM Code for Large Cosmological N-body Simulations," *Publications of the Astronomical Society of Japan*, vol. 61, pp. 1319–1330, 12 2009.

[9] L. H. Garrison, D. J. Eisenstein, D. Ferrer, N. A. Maksimova, and P. A. Pinto, "The abacus cosmological N-body code," *Monthly Notices of the Royal Astronomical Society*, vol. 508, pp. 575–596, 09 2021.

[10] P. M. Galán-de Anta, E. Vasiliev, M. Sarzi, M. Dotti, P. R. Capelo, A. Incatasciato, L. Posti, L. Morelli, and E. M. Corsini, "The fragility of thin discs in galaxies – I. Building tailored N-body galaxy models," *Monthly Notices of the Royal Astronomical Society*, vol. 520, pp. 4490–4501, 02 2023.

[11] S. Li, S. Zhong, P. Berczik, R. Spurzem, X. Chen, and F. Liu, "Tracing the evolution of smbhs and stellar objects in galaxy mergers: A multi-mass direct n-body model," *The Astrophysical Journal*, vol. 944, no. 1, p. 109, 2023.

[12] J. Barnes and P. Hut, "A hierarchical O(N log N) force-calculation algorithm," , vol. 324, pp. 446–449, Dec. 1986.

[13] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy, "Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity," *Journal of Parallel and Distributed Computing*, vol. 27, no. 2, pp. 118–141, 1995.

[14] V. Springel, "The cosmological simulation code gadget-2," *Monthly Notices of the Royal Astronomical Society*, vol. 364, p. 1105–1134, Dec. 2005.

[15] V. Springel, "E pur si muove:galilean-invariant cosmological hydrodynamical simulations on a moving mesh," *Monthly Notices of the Royal Astronomical Society*, vol. 401, p. 791–851, Jan. 2010.

[16] D. Potter, J. Stadel, and R. Teyssier, "Pkdgrav3: Beyond trillion particle cosmological simulations for the next era of galaxy surveys," 2016.

[17] M. Burtscher and K. Pingali, "Chapter 6 - an efficient cuda implementation of the tree-based barnes hut n-body algorithm," in *GPU Computing Gems Emerald Edition* (W. mei W. Hwu, ed.), Applications of GPU Computing Series, pp. 75–92, Boston: Morgan Kaufmann, 2011.

[18] T. Hamada, K. Nitadori, K. Benkrid, Y. Ohno, G. Morimoto, T. Masada, Y. Shibata, K. Oguri, and M. Taiji, "A novel multiple-walk parallel algorithm for the barnes–hut treecode on gpus–towards cost effective, high performance n-body simulation," *Computer science-research and development*, vol. 24, no. 1, pp. 21–31, 2009.

[19] M. S. Warren and J. K. Salmon, "Astrophysical n-body simulations using hierarchical tree data structures," *SC*, vol. 92, pp. 570–576, 1992.

[20] M. S. Warren and J. K. Salmon, "A parallel hashed oct-tree n-body algorithm," in *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pp. 12–21, 1993.

[21] A. Y. Grama, V. Kumar, and A. Sameh, "Scalable parallel formulations of the barnes-hut method for n-body simulations," in *Supercomputing'94: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, pp. 439–448, IEEE, 1994.

[22] R. Speck, D. Ruprecht, R. Krause, M. Emmett, M. Minion, M. Winkel, and P. Gibbon, "A massively space-time parallel n-body solver," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, 2012.

[23] T. Gangavarapu, H. Pal, P. Prakash, S. Hegde, and V. Geetha, "Parallel openmp and cuda implementations of the n-body problem," in *Computational Science and Its Applications–ICCSA 2019: 19th International Conference, Saint Petersburg, Russia, July 1–4, 2019, Proceedings, Part I 19*, pp. 193–208, Springer, 2019.

[24] J. Rokisky, "Github - jrokisky/mpi-barnes-hut: a serial and mpi implementation of the barnes-hut simulation." https://github.com/Jrokisky/MPI-Barnes-hut/tree/master. (Accessed on 07/26/2024).

[25] "Galaxies - nasa science." https://science.nasa.gov/universe/galaxies/. (Accessed on 08/09/2024).

[26] "Supermassive black hole - wikipedia." https://en.wikipedia.org/wiki/Supermassive_black_hole. (Accessed on 08/09/2024).

[27] J. Sandham, "jsandham / nbodycuda — bitbucket." https://bitbucket.org/jsandham/nbodycuda/src/master/. (Accessed on 07/26/2024).

[28] "Circular orbit - wikipedia." https://en.wikipedia.org/wiki/Circular_orbit. (Accessed on 08/09/2024).

[29] "Verlet integration - wikipedia." https://en.wikipedia.org/wiki/Verlet_integration#Velocity_Verlet. (Accessed on 08/10/2024).

[30] "Scaling - hpc wiki." https://hpc-wiki.info/hpc/Scaling#Strong_Scaling. (Accessed on 08/08/2024).

[31] "Cirrus hardware." https://www.cirrus.ac.uk/about/hardware.html. (Accessed on 08/03/2024).

[32] "Galaxy - wikipedia." https://en.wikipedia.org/wiki/Galaxy. (Accessed on 08/08/2024).