CS205: N-body Simulation

Danning Lai[1], Ana Vitoria Rodrigues Lima[1], Xu Tang[1], and Hanlin Zhu[1]

[1]Harvard University, IACS

July 26, 2023

Abstract

The N-body problem is a fundamental and challenging problem in various fields such as astrophysics, cosmology, and aerodynamics. This project aims to develop an efficient and accurate N-body simulation system using high-performance computing techniques, such as OpenMP, OpenMPI, ISPC, and data-oriented programming. The simulation system is based on the Cell Lists Algorithm, which is a popularly used method for molecular dynamics simulations. The first objective of this project is to develop a baseline code with a single process for the simulated N-body system, which can be parallelized using OpenMP and SIMD instructions to achieve single node best performance, serving as a benchmark for evaluation metrics. The main goal of the project is to parallelize the code using distributed computing techniques, making the simulation system more efficient and scalable for large-scale simulations on a computing cluster. The innovative aspect of this project is its ability to precisely pack an unfixed number of cells, each containing an varying number of particles, and transmit them to their corresponding neighbors. At the same time, the system is capable of accurately decoding and processing the dense information received from neighboring rank, which send and receive data with dynamic size.

1 Background and Significance

The problem we are solving is the gravitational interaction N-body simulation in 3D space. This is a complex problem that involves simulating the motion of a system of N particles under the influence of gravitational force. The gravitational interaction N-body problem is a classic problem in astrophysics, and it has many important applications in astronomy, cosmology, and aerodynamics.

The gravitational interaction N-body problem is important in astrophysics because it is essential to understanding the dynamics of celestial objects such as stars, planets, and galaxies. In astronomy, the simulation of N-body system can help explain the formation and evolution of galaxies and their components, such as globular clusters and galactic nuclei. Cosmological simulations, which use N-body techniques, are crucial to understanding the large-scale structure of the universe, including the distribution of dark matter.

In aerodynamics, N-body simulations can be used to model the flow of particles around an object, such as aircrafts or rockets, and to optimize their design for better performance. Additionally, the simulation of N-body systems is also relevant in fields such as materials science, biology, and chemistry, where it can help understand the behavior of molecules and atoms under different conditions.

Although the gravitational interaction N-body simulation is a fundamental topic in many fields, it poses significant challenges due to the lack of analytical solutions. This problem requires massive computational resources, particularly for large-scale systems, making it computationally intensive. Therefore, it is essential to have efficient and accurate methods to advance research in this field. Fortunately, recent

progress in machine learning and high-performance computing has made it possible to accelerate N-body simulations and reduce their computational cost while maintaining high precision. Specifically, techniques such as parallel computing and GPU acceleration can significantly speed up simulations.
In summary, the gravitational interaction N-body problem is a complex and important problem in astrophysics, astronomy, cosmology, aerodynamics, and other fields. Solving this problem efficiently and accurately can lead to significant advances in our understanding of the universe and its components, as well as in the design of new technologies.

2 Scientific Goals and Objectives

The scientific goal of this project is to develop an efficient and accurate N-body simulation system using high-performance computing. The objects of this project are:

- Develop an N-body simulation algorithm that can be parallelized efficiently to take advantage of the high-performance computing architecture's parallel processing capabilities.
- Optimize the parallelization of the numerical simulation algorithm to reduce communication overhead and load balancing issues.
- Implement the simulation algorithm on a high-performance computing architecture (cluster) to accelerate the simulation process and handle large-scale systems.
- Investigate the scalability of the developed algorithm by testing it on different sizes of N-body systems and assessing its performance with increasing numbers of processors.
- Evaluate the performance of the parallelized algorithm by analyzing its data movement and compute intensity, and comparing it with the theoretical limits of the roofline model.
- Identify and optimize the performance bottlenecks of the parallelized algorithm using techniques such as loop tiling, vectorization, and cache optimization to improve its computational efficiency.
- Compare the performance and efficiency of the parallelized algorithm with the non-parallelized version to evaluate the benefits of parallelization.

The ultimate goal of this project with a focus on parallelism is to develop a highly scalable and efficient N-body simulation system that can take advantage of the available parallel processing resources. This project aims to address the challenge of simulating the dynamics of a large number of interacting bodies, such as galaxies or star clusters, which require significant computational resources to accurately model their behavior.
By developing an N-body simulation algorithm that is highly parallelizable and optimized for high-performance computing architectures, this project aims to significantly reduce the time and resources required to simulate such systems. The parallelization of the algorithm will involve minimizing communication overhead and load balancing issues, which are common challenges when parallelizing computational tasks. By implementing the simulation algorithm on a high-performance computing architecture, such as a cluster, this project aims to accelerate the simulation process and handle large-scale systems that would be infeasible to simulate on a single processor.
HPC (High-Performance Computing) architecture is a specialized computing infrastructure that delivers high computational power, fast storage, and networking capabilities to manage large-scale scientific, engineering, and data-intensive applications. It is comprised of a cluster of interconnected computers or servers, each equipped with multiple processors, high-speed interconnects, and substantial amounts of

memory and storage.
In our project, the need for compute hours on an HPC architecture is warranted by the computational requirements of the N-body simulation problem. The gravitational interaction between N particles demands a large number of calculations, which increases exponentially with the number of particles. This computational complexity poses a significant challenge when simulating systems with a large number of particles using traditional numerical methods, which can be slow and require a vast amount of memory. In contrast, an HPC cluster can provide significant computing resources to expedite the simulation process and handle large-scale systems. The use of HPC can also allow for the exploration of a broader range of parameters, enabling a more comprehensive study of the N-body problem.

3 Algorithms and Code Parallelization

3.1 Cell List Algorithm

We develop our N-body simulation system based on the Cell List Algorithm, which is a computational technique aimed at reducing the computational complexity of pairwise interactions in N-body simulations. It partitions the simulation box into a grid of cubic cells, each with a fixed side length corresponding to the cutoff radius for pairwise interactions. Particles in the system are then assigned to individual cells based on their positions. It is worth noting that alternative algorithms such as the Barnes-Hut tree algorithm or Fast Multiple Method can also be used for N-body simulation. However, we chose the Cell List Algorithm due to its effectiveness in reducing the computational cost of pairwise force calculations. During the calculation of pairwise forces, the algorithm considers only particles within the cutoff radius and neighboring cells. This approach significantly reduces the computational cost of calculating pairwise forces compared to a brute-force approach. For the Cell List, we use periodic physical domain boundaries, meaning that a particle exiting the domain from one boundary immediately enters it again from the opposite boundary, as if the domain was wrapped around. The number of cells required for the simulation grid depends on the chosen cutoff radius for the simulation. In the following passage, we provide detailed implementation details of the Cell List Algorithm, including the determination of the number of cells required for the simulation grid.

3.2 Leapfrog Time Integration Algorithm

To propagate the positions and velocities of particles forward in time while accounting for the forces acting on them, we employ the leapfrog time integration algorithm. This algorithm, also referred to as the 'kick-drift-kick' method, ensures second-order accuracy by updating particle velocities halfway between position updates. At each time step, we update the positions and velocities of particles using the following operations:

1. "Kick" operation on $v^i_{t+1/2}$: We update the all particle velocities halfway between position updates, using the forces acting on them.

2. "Drift" operation on p^i_{t+1}: All particle positions are updated using the updated velocities from the previous kick operation.

3. "Kick" operation on v^i_{t+1}: All particle velocities are updated again, using the updated positions from the previous drift operation.

The sequence of operations is repeated for each time step in the simulation process.

3.3 Libraries and Parameters

The libraries we used have been: `cassert`, this one checks assumptions and conditions during program execution, and helps with debugging by aborting the program if the condition is false; `cmath`, this contains mathematical functions and performs mathematical operations on floating-point numbers; `iostream` provides standard input/output stream objects for reading from and writing to the console; `string` implements a string class; `vector` implements a vector template class; `random` offers random number generation facilities; `algorithm` contains a variety of algorithms for operations. Finally, notice that, for our simulation, we use non-dimensional parameters, i.e., unitless. This means that we're not using realistic numbers; we set the gravitational force G to be 1.

3.4 Baseline Code Implementation

Our program is composed of four header files: `Allocator.h`, `Cell.h`, `Nbody.h`, and `force_interaction.h`, and two cpp files: `force_interaction.cpp` and `main.cpp`.

The `Allocator.h` header defines an `aligned_allocator` struct that is utilized to allocate and deallocate aligned memory. Specifically, `posix_memalign` is used to allocate memory with a 32-byte alignment. This alignment is necessary for optimal performance when working with SIMD instructions, which is beneficial for ISPC and vectorizing force interaction computation. The customized allocator is intended to be used in conjunction with the vector data structure to ensure that data is properly aligned for efficient SIMD operations.

We implement the template class named `Cell` inside the `cell.h` header using the data-oriented programming paradigm to optimize performance. The class has private member variables for position, velocity, and acceleration, with each of these consisting of three vectors for the x, y, and z coordinates. Furthermore, the class is templated on capacity N, which specifies the maximum number of particles that can be stored in each cell, and this allows us to control the vector capacity. By doing so, we avoid memory re-allocation, which can introduce performance overhead due to the need to copy data from one memory block to another, and can also lead to memory leaks and pointer invalidation in our case.

`Nbody.h` is where we define all the necessary functions needed for updating the state of the simulation, including resetting temporary cells, updating ghosts (i.e. particles that will move the neighboring cells), updating boundaries, and updating particle positions, velocities, and accelerations. To develop a NUMA-aware program, we first allocate the array of cells on the heap inside the constructor, and then we use a for-loop to initialize the array. This ensures that we adhere to the NUMA first touch policy. We manage the cell objects by subsets using the `cell_subset` variable, which consists of 125 vectors of pointers to the cell objects. This approach facilitates the tracking of subsets of boundary cells and ghost cells.

Below is the detailed illustration of `Nbody.h`. (We are the main developer of the code.)

`class Cell`: We define our own class called Cell, which is templated on N and has 9 fields, p_x, p_y, p_z, v_x, v_y, v_z, a_x, a_y, and a_z, respectively the structure of array of the positions, velocity, and accelerations. When we call the constructor of this class Cell, we reserve a size of N for each of the fields to ensure the size of vectors doesn't change throughout the program. We include this class as a header in the main header file.

`find_n_cells()`: In this function, we aim to find the number of cells in the grid. We do so by dividing the side length by two until the cutoff radius is just above (greater than) the side length of a cell. The reason is that this would ensure only interactions and forces between the immediate neighbors will have effects. For example, in a 3-d case, one cell has 27 immediate neighbors, and we only consider interactions within this range in the algorithm. This is more efficient than algorithms that consider all pairwise forces.

`initialize()`: This function initializes all particles in the cell and stores pointers to subsets. In the first part, we generate the positions from a probability distribution, which is uniform distribution(0, 1) in our case. Moreover, the initial velocity and positions of all particles are 0. We want to simulate a "Big Bang" in the visualization. In the second part of initialize(), we further divide the cell list into subsets to facilitate the update procedure. The cell list comprises three distinct parts: the interior, boundary, and ghost cells. We note that particles in the boundary cells are prone to crossing over to neighboring cells belonging to different processes. To address this issue, ghost cells are introduced to temporarily store data for the boundary cells and aid with the update process. To account for the ghost cells, the original cell list with a side length of n_cell is increased to n_cell+2. The interior subset consists of cells whose force interactions do not require communication between processes. We assign indices of 2 and -2 to the ghost cells, 1 and -1 to the boundary cells, and (0, 0) to the interior. In three-dimensional space, there are five subsets on each side, i.e., -2, -1, 0, 1, and 2. Consequently, there are a total of 125 subsets (5*5*5) in the 3-dimensional cell list after this division. We use a variable called cell subset to store the 125 subsets, each containing pointers to the cells. This approach simplifies the tracking of boundary cells and ghost cells, making it easier to prepare for the upcoming exchange.

`dump()`: At specific timestep intervals, we output the coordinates of all particles of all processes into a file for visualization.

The following are important functions for the position and acceleration updates:

`update_boundaries()`: This function updates the boundary cells' information, including positions and velocities, using the information from the corresponding ghost cells on the opposite side. The first thing we do is to determine whether a cell is a ghost cell. The way we achieve this is through cell subset indexing. If the subset index of any dimension of a cell is -2, that means it is a beginning ghost cell, and we want to append the information to the corresponding boundary cell on the other side (which is indexed at 1). For a ghost cell indexed at 2, we append the information to the boundary cell at -1. This function is called at the end of update_p().

`update_ghosts()`: This function is the reverse of the update_boundaries() function above. All positions of particles in a boundary cell are given to the ghost cells. Determining a ghost cell and finding the boundary cell we want to send from are the same as before. The function iterates through all ghost cells, updating their properties to match their corresponding boundary cells and applying the necessary offset to their positions. However, in this case, we want to completely overwrite the ghost cell using information from the boundary cell instead of appending new information.

`update_p()`: This function computes the new positions of all particles. To prevent particles from being updated twice in case they move to another cell during the update, we use a function called reset_temp_cells(). This function creates a copy of the entire grid. Using a nested for-loop, we compute the new positions for all particles in each cell. Next, we verify whether a particle still belongs to its original cell and update its position in the corresponding temporary cell. Once all updates are complete, we swap the temporary cells with the actual cells. After updating the positions of all particles., we call update_boundaries() to update particles from the ghost cells to their corresponding boundary cells.

`update_v()`: This function calculates the velocities of particles. It iterates through all cells, and for each particle in each cell, computes the force between each pair of particles in the cell, updating their velocities accordingly.

`update_a()`: This function is responsible for updating the acceleration of particles within the simulation. It calls four other functions to do this: update_a_intra(), update_a_inner(), update_ghosts(), and update_a_boundary(). In terms of updating acceleration, our algorithm utilizes ISPC for vectorizing force interactions. Thanks to the previous data alignment, this process has become very efficient. The acceleration calculation is divided into three types: intra-cell, inter-cell, and boundary-cell interactions. First, we

calculate the interactions between all particles within each cell. Then, for all interior cells, we compute the interactions between each cell and its 27 neighboring cells since all their neighbors are in good standing. Next, we update the ghost cells, which involves moving particles' information in the boundary cells to their corresponding ghost cells. After this is finished, each boundary cell is surrounded by cells in good standing, allowing us to compute the force interactions between the boundary cells and their neighbors. Hence, we update the acceleration for inter-cell interactions, but specifically for boundary cells.

`update_a_intra()`: This function calculates the acceleration of particles due to gravitational forces within the same cell. It iterates through all cells and computes the gravitational force between each pair of particles in the cell, updating their accelerations accordingly. This does not involve interactions between cells or processes.

`update_a_inner()`: This function calculates the acceleration of particles due to gravitational forces between different cells, but only for the interior cells of the grid. It iterates through all cells, and for each cell, it iterates through neighboring cells to compute the gravitational force between particles in different cells. The accelerations of particles are updated accordingly.

`update_a_boundary()`: This function is similar to update_a_inner(), but it calculates the acceleration of particles due to gravitational forces between different cells for cells on the boundary of the grid. It iterates through all cells and checks if the cell is on the boundary. If it is, it iterates through neighboring cells and computes the gravitational force between particles in different cells, updating their accelerations accordingly.

3.5 Parallelization of the code

Our code employs hybrid MPI/OpenMP parallelism, as some parts require communications between processes while some do not. We will discuss our parallelization method as follows.

OpenMP parallelism: For the portion of code that does not have data dependencies, we employ the OpenMP parallelism using `#pragma omp parallel for schedule (dynamic,1)`. We have chosen dynamic scheduling for load-balancing purposes. As the number of particles in a cell can vary much, we do not want to waste resources on a cell with very few particles. Meanwhile, we choose a chunk size of one, because the outer loop is large enough.

MPI topology: A three-dimensional topology is created using the `MPI_Cart_create` function. A Cartesian MPI communicator `cart_comm_` with periodic boundaries is used. Then, we use `MPI_Cart_coords` to find the coordinates. We then find the 26 neighbors for each process. Finally, we call `MPI_Cart_rank` to convert the coordinates to ranks and store all ranks of neighbors.

Message packing and unpacking: `pack_boundary_subsets()` packs the positions of particles in the boundary subsets into message buffers, so that we can send the information to neighboring processes and store it in their ghost cells. We pack the 125 subsets into 27 buffers, one buffer for each neighbor. This function is used by `update_p()`. On the other hand, `pack_ghost_subsets()` packs the positions and velocities of particles in the ghost subsets into message buffers so that we can send the information to neighboring processes and unpack it in their boundary cells. This function is called by `update_a()`.`unpack_to_boundary_subsets()` and `unpack_to_ghost_subsets()` do the corresponding unpacking for the two functions above.

Position update using non-blocking communication: To let a process continue with its computation without waiting for the communication to complete, we choose to use non-blocking MPI communications such as `MPI_Isend`. In the position update part, after updating all particles, we reset a buffer and then pack the ghost subsets. Since a process may receive from others, we want to know the size of the incoming message so that we can resize the receive buffer. This is achieved with one `MPI_Probe` on the neighboring process. Moreover, we need two pairs of `MPI_Isend` and `MPI_Irecv`, one for sending the

size and the other for sending the actual message. We send the size because we have packed information of subsets into the message buffers, while each subset and cell contains different numbers of particles. We need to keep track of this information to make correct updates. Finally, we use `MPI_Waitall` with 52 requests (because sending two messages to each of the 26 neighbors) to wait for all communication operations to complete and the program can synchronize, after which we unpack to boundary subsets.

Asynchronous communication for acceleration update: We implement the acceleration update similar to the position update. Here, we also allow compute/communication overlap through non-blocking MPI communication. Since there are acceleration updates for forces interaction within the same cell or between different cells in the interior cells, we call `update_a_intra()` and `update_a_inner()` while we're doing the non-blocking communication. In this way, we make more efficient use of available resources and can help reduce overall program execution time.

File I/O with MPI: As explained in the previous section, we dump the coordinates information every certain timestep (to lower the frequency and avoid too much pressure on the File I/O); here, we choose the interval to be ten timesteps. We use `MPI_File_write_at_all` to let all processes simultaneously write the particles' data to the same file. To ensure they all write to the correct place in the file, `MPI_Exscan` is employed to find the corresponding byte offset for every process. We output the coordinates into a `.bin` file, and then read and visualize using a Python script.

ISPC: To write efficient parallel code for the `update_a_intra()` function, we utilized SIMD instructions with ISPC. We accomplished this by creating separate .cpp and header files for the function, and then writing the ISPC version of the function.

3.6 Validation, Verification: Relevant Literature

To reduce computational costs in our project, we use Cell List, a data structure commonly used in molecular dynamics simulations to identify particle pairs in short-ranged interactions. There is an established method called Barnes-Hut algorithm (Barnes and Hut (1986)) for the N-body simulation problem, however Dehnen (2002) points to how this might not scale as well for large N. Given this method has already been widely explored, we decided to implement the cell list algorithm under Professor Wermerlinger's suggestion.

Efficiently managing spatial information and reducing the complexity of pairwise particle interactions, cell list is particularly valuable in systems with large numbers of particles, where brute-force approaches to computing interactions between every particle pair can become computationally infeasible Allen and Tildesley (1989). Using a cell list is reasonable because, based on the physical principle inverse-square Law, the intensity of a physical quantity (such as gravitational force) decreases with the square of the distance from its source. Hence, in our project, as two particles move further away from each other, their force interaction becomes negligible. On the other hand, ghost cells facilitate efficient communication and data exchange. Therefore, ghost cells, in conjunction with cell lists, provide an effective means to manage spatial information and optimize the performance of parallelized N-body simulations.

Also, we use the Leapfrog Time Integration Algorithm, also known as the Verlet or Verlet-Störmer methods. It is a widely used numerical scheme for integrating Newton's equations of motion. Supportive related works include the book "Computer Simulation of Liquids" Allen and Tildesley (1989) and the article by Tuckerman et al. (1992) Tuckerman et al. (1992).

4 Performance Benchmarks and Scaling Analysis

In this section, the average wall time and Flops are obtained by averaging the results over total number of time steps after the system reached equilibrium post-Big Bang.

4.1 Roofline Analysis

Recall from lecture 10, the nominal peak arithmetic performance:

$$\pi = f \times n_c \times l_s \times \phi$$

where f is the base clock rate, n_c refers to the number of cores, w_s is the SIMD vector width, p is the precision (here we use double precision), $l_s = \frac{w_s}{p}$ represents the SIMD lanes, and ϕ denotes the total flop per cycle.

With Intel Xeon E5-2683v4 (Broadwell architecture), we can calculate the architecture peak performance as follows:

$$\begin{aligned}\pi &= 2.5 \text{ GHz} \times 32 \times \frac{256}{64} \times 4 \frac{\text{Flops}}{\text{cycle}} \\ &= 1280 \text{ Gflop/s}\end{aligned}$$

We conduct a roofline analysis for our system's most computationally expensive function, `update_a_intra`. To do this, we use a counter variable to calculate the number of force interactions in each time step after the system has stabilized post-Big Bang. We then divide this number by the total number of time steps to obtain the average number of force interactions. By analyzing the performance of `update_a_intra` using this method, we hope to better understand the performance characteristics and potential limitations of our system.
The average number of force interactions calculated is 431.1361149 ms. Inside the for-loop of `update_a_intra()`, there are 27 flops due to additions, subtractions, division, multiplications, and sqrt().
Therefore, the total FLOPs for one time step in average can be compute as: Flops that one force interaction needed × Average number of force interaction per iteration, which is:

$$\frac{27 \text{ Flops}}{10^9 \frac{\text{Flops}}{\text{GFlops}}} \times \frac{431.1361149 \text{ ms}}{10^6 \frac{\text{ms}}{\text{s}}} = 11.6406751 \text{ Gflop/s} \tag{1}$$

Now, we plot the performance as below. we see that the measured performance are way below the performance ceiling.

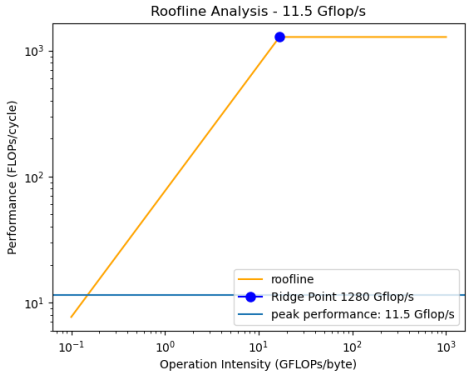

Figure 1: Roofline Model

The measured performance of the `update_a_intra` function is way below the peak performance, perhaps due to memory bound. A large number of total load and store operations are encountered when simulating 100000 particles. For example, at one time step, the result shows 10493488 load/store. Too

many memory accesses will slow down the whole process. Meanwhile, this particle count was chosen deliberately to saturate the computing power. However, the resulting memory-bound behavior leads to reduced efficiency as the processor is waiting for data to be fetched from memory.

4.2 Scaling Analysis - Weak Scaling

For weak scaling, the goal is to add more processors p and increase the problem size to get to the larger problem size *at the same time* of the problem size that is p times smaller, meaning $W_p = pW_s$. This can be indicated by:

$$S_p = \frac{W_p/T_p}{W_s/T_s} = p\frac{T_s}{T_p} = p\frac{T_s}{T_s + T_o}$$

In this case, we run the program with 1, $2^3 = 8$, $3^3 = 27$ processes, and obtain the following results:

	Test case A	Test case B	Test case C
Number of processes	1	8	27
Average wall clock time per time step [ms]	1217.07	746.75	6125.5
Number of particles	200	566	1040
Number of threads per time step	4	4	4

Table 1: Workflow parameters of the three test cases used during project development.

Notice that we determined the number of particles using the formula $\sqrt{200 \times 200 \times \text{number of processes}}$, under the assumption that the number of operations to be performed is proportional to N^2. However, we use a cell list which may affect the actual number of operations performed.
Based on the table, $T_s = 802.7$ ms, we can then compute the speedup S_p as following:

$$S_8 = 8 \times \frac{1217.07}{746.75} \approx 13.04$$
$$S_{27} = 27 \times \frac{1217.07}{6125.5} \approx 5.37$$

Then the efficiency can be obtain as below:

$$E_8 = 8.60 \div 8 \approx 1.63$$
$$E_{27} = 2.59 \div 27 \approx 0.20$$

Notice that the efficiency decreases as the number of processes increases. This is expected, since as we increase the number of processes, the amount of communication required between them also increases. In this case, each process needs to communicate with 26 neighboring processes, and as we add more processes, the amount of communication increases, leading to higher communication overheads and reduced efficiency. Furthermore, it is worth noting that the scaling analysis we conducted does not take into account any file I/O operations. We use file I/O mainly for visualization purposes, and our primary focus in this analysis is on the computational aspects of the program.

5 Resource Justification

The request of the annual amount of node hours is linked with the node hours used by the representative benchmarks. The number of node hours consumed by a simulation is computed by multiplying the number

of nodes by the wall time expressed in hours for a typical production run.
The optimal job size of the representative benchmark is using 8 processes, each with 4 threads. This is because, if we further increase the number of processes, the program becomes less efficient. Assuming the corresponding wall time for a production run is 746.75 ms, which is then equivalent to ~ 0.00663777777 node hours, as a result of the following product:

$$0.00663777777 \text{ node hours} = 32 \text{ nodes} \times \frac{0.74675s}{3600\frac{s}{\text{hour}}}$$

The benchmark is short and represents a small number of iterations and particles (cycles, time steps or an equivalent measure) in the `update_a_intra()` function.

6 Future Work

In future work, we plan to continue developing the N-body simulation system using high-performance computing techniques. One area we plan to focus on is SIMD vectorization to further optimize the performance of the code.Also, we encountered a high memory cost issue when using a templated class to avoid memory reallocation during the simulation process. This limits our ability to execute multiple processes with multiple threads. Therefore, we may need to explore other approaches to overcome this issue. Finally, as we move forward, we need to prioritize the gains we want to achieve while considering the tradeoffs that come with optimization.

References

M.P. Allen and D.J. Tildesley. *Computer Simulation of Liquids*. Computer Simulation of Liquids. Clarendon Press, 1989. ISBN 9780198556459. URL https://books.google.com/books?id=O32VXB9e5P4C.

Josh Barnes and Piet Hut. A hierarchical O(N log N) force-calculation algorithm. , 324(6096):446–449, December 1986. doi: 10.1038/324446a0.

Walter Dehnen. A hierarchical (n) force calculation algorithm. *Journal of Computational Physics*, 179(1): 27–42, jun 2002. doi: 10.1006/jcph.2002.7026. URL https://doi.org/10.1006%2Fjcph.2002.7026.

M. Tuckerman, B. J. Berne, and G. J. Martyna. Reversible multiple time scale molecular dynamics. *The Journal of Chemical Physics*, 97(3):1990–2001, 08 1992. ISSN 0021-9606. doi: 10.1063/1.463137. URL https://doi.org/10.1063/1.463137.