UNIVERSITÀ
DI PAVIA

Advanced Computer Architecture

Scalability and Communication Overhead in Distributed N-Body Simulation using MPI on GCP

Claudio Guarrasi

Department of Electrical, Computer and Biomedical Engineering
University of Pavia

Email: claudio.guarrasi01@universitadipavia.it
GitHub: https://github.com/PapiDrago/n-body-problem

August 20, 2025

Abstract

This work presents the design, implementation, and performance evaluation of a parallel *N-Body simulation* using the *Message Passing Interface* (MPI). The direct method formulation of the N-Body problem, characterized by its quadratic $O(N^2)$ computational complexity, was adopted as the baseline to preserve exact particle interactions. The parallel implementation distributes bodies across processes and employs collective communication—specifically `MPI_Allgatherv`—to ensure that each process has a complete global view of particle positions required for force computation. Execution and communication times were measured using `MPI_Wtime()`.

The application was deployed on multiple Google Cloud Platform (GCP) configurations, ranging from a single multi-core instance to clusters of virtual machines distributed across different geographical regions. Performance was analyzed in terms of execution time, speedup, and scalability, with particular attention to the fraction of time spent in communication.

Experimental results show that strong scalability is achieved when processes remain within a single machine or intra-regional clusters, where communication overhead is negligible. However, performance deteriorates significantly in inter-regional and multi-region deployments, where network latency and synchronization dominate the runtime. Weak scalability analysis further confirms the inherent limitations of the direct method, as the per-process workload increases with the number of particles. Alternative approaches that reduce computational complexity are briefly discussed as possible directions for improving scalability in large-scale simulations.

Contents

1 Introduction

The N-body problem is a well-known problem in physics and it has several applications ranging from modeling the gravitational interactions in galaxies and solar systems to simulating charged particle dynamics in plasmas and atoms in molecular systems.

Depending on the goal of the analysis, various assumptions can be relaxed or adjusted. In the experiments conducted, it was considered a closed system of N point masses interacting solely through gravitational forces, while neglecting possible collisions. This assumption is quite reasonable when modeling astronomical systems, especially when the goal is to compute the trajectories of celestial bodies within them. In such systems, the gravitational force dominates due to the extremely large masses involved, allowing other forces to be safely neglected.

Numerous algorithmic solutions have been proposed to address the N-body problem, many of which have been implemented in software to automate the computation of relevant physical quantities.

This work does not aim to propose a new or improved algorithmic solution. Instead, given an existing approach, the focus is on developing a parallel implementation that highlights the complexities involved in distributing the computation and evaluates its performance in a distributed environment.

Essentially the objectives of this work were:

1. to analyze a possible serial algorithm for solving the N-body problem;
2. to perform an *a priori* study of the available parallelism using Amdahl's Law;
3. to develop a parallel implementation using the Message Passing Interface (MPI);
4. to evaluate performance and scalability through experiments conducted on Google Cloud Platform (GCP).

2 Physical Model

In order to understand the algorithm it is important to recall the physics laws on which the model is based.

- Newton's law of universal gravitation:

$$\vec{F}_{i,j} = G\frac{m_i m_j}{\|\vec{r}_i - \vec{r}_j\|^3}(\vec{r}_j - \vec{r}_i) \tag{1}$$

  G Gravitational constant.

  m_i, m_j Masses of bodies i and j.

  $\vec{r}_i, \vec{r}_j$ Position vectors of bodies i and j.

  $\vec{F}_{i,j}$ Force acting on body i due to body j.

  Please note that $\vec{F}_{i,j}$ is an attractive force that has the same direction of the displacement vector $(\vec{r}_j - \vec{r}_i)$.

- Newton's second law of motion:

$$\Sigma\vec{F} = m\vec{a} \tag{2}$$

  It states that the sum of all the forces $\Sigma\vec{F}$ acting on a particular point mass is equal to the product between its mass m and its acceleration $\vec{a}$.

In our scenario we can combine (1) and (2) to obtain, at a certain time instant, the acceleration $\vec{a}_i$ of point mass i implied by all the gravitational forces acting on that point mass. Analytically we can write:

$$\vec{a}_i = \sum_{\substack{j=0 \\ j\neq i}}^{N-1}\left[G\frac{m_j}{\|\vec{r}_i - \vec{r}_j\|^3}(\vec{r}_j - \vec{r}_i)\right] \tag{3}$$

Since we are ultimately interested in the trajectory of each body, from $\vec{a}_i$ we can compute the velocity $\vec{v}_i$ and the position $\vec{r}_i$ of mass i by simply integrating $\vec{a}_i$ with respect to time.

$$\vec{v}_i(t_k) = \vec{v}_i(t_{k-1}) + \int_{t_{k-1}}^{t_k}\vec{a}_i(\tau)\,d\tau \tag{4}$$

$$\vec{r}_i(t_k) = \vec{r}_i(t_{k-1}) + \int_{t_{k-1}}^{t_k}\vec{v}_i(\tau)\,d\tau \tag{5}$$

t_{k-1} Time instant defining the initial condition for the current time step.

Although the notation t_k and t_{k-1} may initially appear cumbersome, it will prove useful later when expressing the iterative numerical formulation of the problem.

2.1 Iterative Nature of the N-body Solution

It is important to note that for systems involving more than two bodies ($N > 2$), the N-body problem does not admit a general closed-form solution. This means that it is not possible to determine the complete trajectories of all bodies analytically from the initial conditions alone.

In the special case $N = 2$, the problem can be reduced to a one-body system by moving to the center-of-mass reference frame. In this frame, the motion can be fully described using relative coordinates, and a closed-form analytical solution can be derived for the position and velocity of each body. However, as soon as a third body is introduced, the system becomes non-linear and highly sensitive to initial conditions, making analytical solutions intractable in the general case [1].

For this reason to compute the position of one particular mass at time k it is necessary to iteratively solve the equations of motion ((3),(4),(5)) for all the N bodies at each time step $t \leq k$. This is required because, as shown in (3), the acceleration at time k depends on the positions of all the bodies at that time. Just from this consideration we can already anticipate the high computational cost.

2.2 Numerical Integration

At each simulation step, we must numerically integrate the equations of motion in order to update the velocity and position of all bodies. In the continuous-time domain, this corresponds to evaluating (4) and (5).

However, in practice a digital computer can only work with discrete time instants due to its inherent limitations: it operates at a finite clock frequency, stores only a finite number of samples in memory, and uses finite-precision arithmetic. As a result, we reformulate the problem in the discrete-time domain.

For convenience, the vector notation is omitted here. The time-discretized velocity update equation can be expressed as:

$$\begin{cases} \dfrac{v_{k+1} - v_k}{\Delta t} = a_k(r_k, k) \\ v_k = v(k\Delta t) \end{cases} \tag{6}$$

v_k Velocity at time t_k, with $t_k = k\,\Delta t$.

$a_k(r_k, k)$ Acceleration at step k, computed from (3).

Δt Step size between two consecutive time instants.

With some little algebraic manipulations we can obtain:

$$v_{k+1} = v_k + a_k \Delta t \tag{7}$$

Similarly, the position update is:

$$r_{k+1} = r_k + v_k \Delta t \tag{8}$$

Incidentally, we expressed position and velocity using the explicit Euler method [2].

However, not all numerical integration methods are equally suitable when simulating physical systems on a computer. This can be seen by writing the exact velocity at time step $k+1$ using the Taylor expansion with Lagrange remainder:

$$v(t_k + \Delta t) = v(t_k) + a(t_k)\,\Delta t + \frac{\Delta t^2}{2}\frac{d^2 v}{dt^2}(\xi), \quad \xi \in [t_k,\, t_k + \Delta t] \tag{9}$$

Neglecting the second-order term yields (7). This highlights that any numerical integration method produces an approximation, and in the case of explicit Euler, the local truncation error is proportional to Δt^2.

Choosing a very small Δt reduces the error but increases the computational cost. For example, if $\Delta t = 0.01\,\mathrm{s}$, simulating one second of motion requires 100 iterations of the solution algorithm, whereas with $\Delta t = 0.1\,\mathrm{s}$, only 10 iterations are needed.

It also crucial to address the fact that, iteration after iteration, the numerical error accumulates, bringing the computed velocity to deviate more and more from the true value, which in turn causes the total energy of the simulated system to drift, violating the thermodynamical principle of energy conservation. For the N-Body system described in section 1, this can be seen by considering the total kinetic energy K_{TOT} of the system at time step k:

$$K_{TOT}(k) = \frac{1}{2}\sum_{i=0}^{N-1} m_i v_i^2(k) \tag{10}$$

Since the system is closed, this quantity should remain constant during the simulation. However, due to the impossibility of computing the exact velocities, this condition cannot be perfectly satisfied. What can be done instead is to bound the energy oscillations around the true value. It can be proven that a *symplectic* numerical methods allows us to achieve this [3].

It is for this reason that I chose to pick for my project the *semi-implicit Euler* integration method, which is symplectic [4], and very similar to explicit Euler method. The only difference is that to compute the position at time step $k+1$ we use the updated velocity v_{k+1} and not the velocity at the previous time step k as seen in 7:

$$r_{k+1} = r_k + v_{k+1}\,\Delta t \tag{11}$$

3 Serial Algorithm

In the literature [1], several approaches have been proposed to address the N-body problem, each with different computational complexities and levels of approximation. In this project, the *direct method* was chosen, as it most closely reflects the conceptual formulation of the problem without making further approximations.

Algorithm 1 shows the basic structure of the direct method: initially all the N bodies are initialized. Each of them has mass, a starting position and a starting velocity. Then, for each simulation step, the kinematic quantities are updated.

Algorithm 1 Serial N-body algorithm (direct method)

```
1: Initialize positions, velocities, and masses
2: for each time step do
3:     Compute accelerations (Algorithm 2)
4:     Update velocities
5:     Update positions
6: end for
```

3.1 Complexity Analysis

3.1.1 Time Complexity

The overall time complexity is dominated by the *Compute accelerations* algorithm 2, since the other components of the serial algorithm involve only a single loop over the N bodies (N is the problem size). In contrast, the *Compute accelerations* step contains a nested loop that also iterates over all N bodies (in practice $N-1$, which is asymptotically equivalent to N).

More precisely, the actual computational cost also depends on the number of time steps T in the simulation, leading to a total complexity of $O(TN^2)$.

Since in the experiments conducted $N \gg T$, the factor T will be omitted from the complexity expressions in the remainder of this work.

Algorithm 2 Inner loop: compute accelerations (direct method)

```
Require: Positions $\{\vec{r}_j\}_{j=0}^{N-1}$, masses $\{m_j\}_{j=0}^{N-1}$
Ensure: Accelerations $\{\vec{a}_i\}_{i=0}^{N-1}$
 1: for $i \leftarrow 0$ to $N-1$ do
 2:     $\vec{a}_i \leftarrow (0,0,0)$
 3:     for $j \leftarrow 0$ to $N-1$ do
 4:         if $j \neq i$ then
 5:             $\vec{d} \leftarrow \vec{r}_j - \vec{r}_i$
 6:             $R^3 \leftarrow \|\vec{d}\|^3$
 7:             $invR^3 \leftarrow 1/R^3$
 8:             $\vec{a}_i \leftarrow \vec{a}_i + G\, m_j\, invR^3\, \vec{d}$
 9:         end if
10:     end for
11: end for
    Time complexity: $O(N^2)$
```

3.1.2 Space Complexity

The serial algorithm requires storing the masses, the positions, the velocities and the accelerations of the N bodies. Since the kinematic quantities are 3D vectors, each elements corresponds to 3 floating-point values. This means the total storage is:

$$N \text{ masses} + 3N \text{ positions} + 3N \text{ velocities} \\ + 3N \text{ accelerations} = 10N \text{ real values}$$

Asymptotically, this results in a space complexity of $O(N)$.

On modern general-purpose computers, this is not a limiting factor because several gigabytes of RAM are available. For example, if double precision is used (8 bytes per floating-point number), one million bodies would require:

$$8 \times 10 \times 10^6 = 80 \text{ MB}$$

which easily fits in memory.

3.2 Implementation in C

The conceptual serial algorithm described in the previous subsection was implemented in the C programming language. This choice was primarily motivated by the need to perform detailed profiling of the code. The complete source code is publicly available on the GitHub repository for this project[1](file *serial.c*).

The implementation follows closely the structure of Algorithm 1, with each major step of the simulation (initialization, acceleration computation, velocity update, and position update) mapped to dedicated functions. This modular structure facilitates both readability and profiling, enabling the identification of computational bottlenecks.

It is worth noticing that to represent the physical vector has been used a struct called *vector* which has three fields, each one is of type *double* and corresponds to one of the three vector components. In addition the serial application, at the end of each simulation step, prints the positions of all the bodies on a text file. This has been done in order to check qualitatively the results and to compare them to those of the MPI parallel application.

3.2.1 Testing

Testing was performed qualitatively, leveraging the physical laws embedded in the program's functions and the results discussed in Subsection 2.2 regarding symplectic numerical integrators, which ensure long-term stability of the simulated system.

For this purpose, a slightly modified version of the serial application (*serial_testing.c*), adapted from [5], was used. This variant allows the user to configure simulation parameters, including the number of bodies, the time-step size, and all initial quantities for each body (mass, position, and velocity), via a plain-text input file. The exact file format is documented in Appendix A.

The application produces an output text file containing the positions of all bodies at each time step. This output was then processed by a Python script (*animate_nbody_2d.py*) to generate a 2D animation, enabling a qualitative verification of the trajectories and overall behavior of the simulated system.

Two representative qualitative test cases were examined. In the first case, shown in Figure 1, the system consisted of a very massive body placed at the origin and a much lighter body with an initial velocity orthogonal to its initial position vector. The simulation confirmed the expected physical behavior:

- the massive body remained essentially stationary due to its large inertia,
- the lighter body followed a closed orbit (elliptical in this specific setup),
- the lighter body's speed increased when it was closer to the massive body, in agreement with Equation (1).

[1]`https://github.com/PapiDrago/n-body-problem`

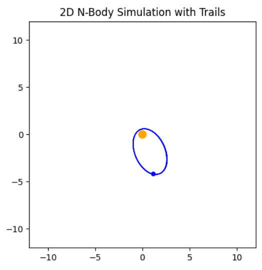

Figure 1: Two-body test: a stationary massive body (orange) and a lighter body (blue) in orbit.

The plot in Figure 1 was obtained by setting the configuation file read by *serial_testing.c*, as follows:

- Body 0 (massive): $m = 1.0$, initial position $(0, 0, 0)$, initial velocity $(0, 0, 0)$;
- Body 1 (light): $m = 3 \times 10^{-6}$, initial position $(-1, -1, 0)$, initial velocity $(0, 6, 0)$.

The time step was set to $\Delta t = 0.01$, and the simulation was run for $T = 2000$ steps.

A second test was performed using a larger set of bodies, inspired by a simplified solar system model. Figure 2 shows the result of a longer simulation run with $T = 2000$ time steps.

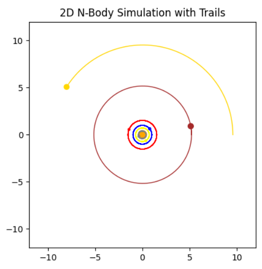

Figure 2: Solar system test: all the planets revolves steadily around the central star.

Throughout the simulation, no planet escaped its orbit, indicating that the numerical integration preserved stability over time. Moreover, bodies at different distances from the central mass exhibited the expected orbital periods, and gravitational interactions between planets did not cause unphysical divergence.

These qualitative observations confirm that the combination of the direct gravitational computation and the chosen symplectic integrator (§2.2) yields a physically plausible evolution, even over longer durations.

The parameters used to produce Figure 2 are reported in Appendix A.

3.2.2 Random Initialization

The standard version of the serial application initialized the simulated system randomly. This makes simpler doing experiments with a large quantities of bodies: the program receives the number of bodies from the user through the command-line. It is worth noticing the use into the inizialization routine of the function *rand_uniform*.

```
double rand_uniform(unsigned int *seed,
    double min, double max) {
    return min + (max - min) * ((double)
        rand_r(seed) / RAND_MAX);
}
```

Using a seed allows to have random reproducible inizializations since a seed determines the beginning of the pseudo-random sequence of values. Calling of `rand_r(seed)` ensures that each different process who may running the application uses the same seed because it is a thread-safe operation. This will be crucial when comparing the results with the parallel application. Please also notice that in `rand_uniform` function, the value returned by `rand_r(seed)` is normalized and bounded to an arbitrary range, and that RNG values are drawn from an uniform distribution in order to avoid directional or positional biases.

In our experiments the interval $(-1.0, 1.0)$ has been forced to focus on the gravitational interaction.

3.2.3 Performance Analysis

In subsection 3.1.1, it was shown that the time complexity of the serial algorithm is $O(N^2)$. To evaluate whether this theoretical estimate holds in practice, the C implementation was executed eight times with the number of bodies N increasing from 100 to 12 800. For all runs, the number of simulation steps was fixed at $T = 100$, the time step size at $\Delta t = 0.01$, and the gravitational constant at $G = 39.47$.

Execution time was measured using the *clock* function from the C standard library (`time.h`).

The tests were conducted on a machine whose detailed specifications are reported in Appendix B.

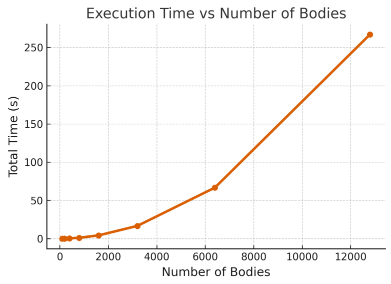

Figure 3: Execution time as a function of the number of bodies N.

As expected the running time has a non-linear relationship with the number of bodies as can be seen in Figure 3.

The log-log plot in Figure 4 reveals an approximately straight line, indicating a power-law relationship, i.e., a quantity varies with the power of another, suggesting a time complexity around $O(N^2)$.

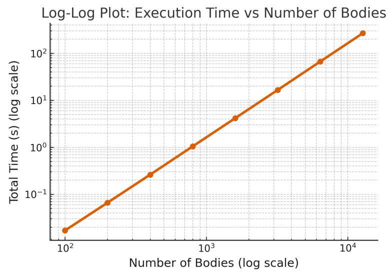

Figure 4: Log-log plot of execution time vs. number of bodies, revealing a power-law relationship.

The raw measurements used to generate Figures 3 and 4 are reported in Appendix C.

Since the empirical results confirmed that the time complexity of the C program is $O(N^2)$, it is worth recalling the reasoning presented in subsection 3.1.1: the overall time complexity is dominated by the *Compute accelerations* algorithm 2, as the other components of the serial algorithm involve only a single loop over the N bodies (N being the problem size). This implies that the execution time is mainly due to the `computeAccelerations()` function.

To verify this, the C code was compiled with profiling enabled, and the *gprof* profiler was used to produce a performance report. The main results are presented in Appendix D.1. As predicted, `computeAccelerations()` dominates the runtime (98.49%), confirming it is the primary bottleneck of the serial implementation.

It is also insightful to analyze performance from a hardware perspective. The built-in Linux profiler *perf* was used for this purpose. The results of the *perf* profiler are reported in Appendix D.2.

Since `computeAccelerations()` keeps the CPU busy for the vast majority of execution time, these measurements can be safely attributed to this function.

The very low branch-miss rate (0.06%) indicates that the control flow within the inner loop is highly predictable. Furthermore, the low percentage of frontend idle cycles (0.14%) confirms that the workload is highly CPU-bound. The measured instructions-per-cycle (IPC) of 2.43 corresponds to a cycles-per-instruction (CPI) of approximately 0.41, meaning that on average each instruction is retired in less than one clock cycle. This is likely due to the microarchitecture of the test machine, which can exploit multiple execution pipelines, out-of-order execution, minimal memory stalls and compiler optimizations.

3.2.4 A-priori Study of Available Parallelism

In Subsection 3.2.3 it is highlighted how, although the `computeAccelerations()` is the bottleneck of the serial algorithm, its CPI is (0.41), which is lower than 1. This efficiency is due both to the microarchitecture of the testing machine (see Appendix B) and to the limited data dependencies in `computeAccelerations()`, namely that to compute the accelerations, only the positions need to be known in advance.

On the other hand is crucial to note that the inner loop (Algorithm 2) iterations are independent from one another, i.e., they could be overlapped. This means, of course, that loop unrolling can be applied, but more importantly that the loop iterations can be distributed across processes.

This suggests that it would be effective to build a parallel application. By using *Amdahl's law* it is possible to know a priory the theoretical speedup of the parallel application, i.e., how much faster the parallel program would be compared to the serial algorithm.

$$\begin{aligned} Speedup(N) &= \frac{time_{serial}}{time_{parallel}(N)} = \frac{(S+P)\times time_{serial}}{(S\times t_{serial} + \frac{P\times t_{serial}}{N})} \\ &= \frac{S+P}{S+\frac{P}{N}} = \frac{1}{S+\frac{P}{N}} \end{aligned} \tag{12}$$

time_serial Execution time of the serial application.

time_parallel Execution time of the parallel application.

N The number of CPUs.

S The fraction of the code that cannot be parallelized (inherently sequential part).

P The fraction of the code that can be parallelized.

$S + P = 1$.

In this case all the code can be parallelized since there are no loop-carried dependencies. Nevertheless also the other pieces of the program can be parallelized as well. Then applying (12), with $S = 0$ and $P = 1$, gives:

$$Speedup(N) = N \tag{13}$$

This means that a priori a linear speedup is expected: doubling the number of CPUs N corresponds to halve the execution time.

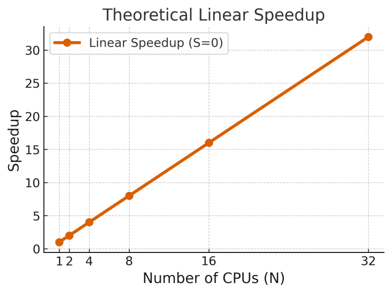

Figure 5: Speedup as a function of the number of CPUs.

However, this scenario is too optimistic to be true — and indeed it would only occur if communication among the processes were instantaneous. To be clear if we assign to each CPU (process) a single mass, in order for each process to compute the acceleration for its own mass it is necessary to know where the other masses are located. Hence each process has to communicate to the others the position, and this takes time.

4 Parallel Algorithm

4.1 Conceptual Design

The chosen approach to parallelize the N-body problem algorithm (Algorithm 1) is as follows. Given N bodies and P processes, each process is assigned $\frac{N}{P}$ bodies. This means it is responsible for initializing these bodies and updating their kinematic quantities during the simulation.

However, to compute the accelerations for its local bodies, a process requires the positions of *all* bodies in the system. For this reason, after the initialization phase and at the end of every simulation step, all processes must exchange the positions of their bodies. During the initialization phase, the masses must also be broadcast to all processes, since they are required in the acceleration computation. It is important to note that no process can start computing accelerations for its local bodies until it has received the position data from all other processes. This flow is shown in Algorithm 3.

Algorithm 3 Parallel N-body (single-process view)

```
1: Initialize local positions, local velocities, and local masses
2: Send local positions and local masses
3: Wait for global positions and global masses
4: for each time step do
5:     Compute local accelerations (Algorithm 4)
6:     Update local velocities
7:     Update local positions
8:     Send local positions
9:     Wait for global positions
10: end for
```

As discussed in Subsections 3.2.3 and 3.2.4, the main motivation for building a parallel algorithm was to distribute the workload of the inner loop of the serial algorithm (Algorithm 2).

Conceptually, the structure of the computation remains the same, with the only difference that the outermost loop now iterates over $\frac{N}{P}$ local bodies, as shown in Algorithm 4.

Algorithm 4 Parallel inner loop: compute local accelerations

```
Require: Global positions $\{\vec{r}_j\}_{j=0}^{N-1}$, global masses $\{m_j\}_{j=0}^{N-1}$
Ensure: Local accelerations $\{\vec{a}_i\}_{i=0}^{(N/P)-1}$
1: for each local body $i$ do
2:     $\vec{a}_i \leftarrow (0,0,0)$
3:     for $j \leftarrow 0$ to $N-1$ do
4:         if $j$ is not the same as global index of $i$ then
5:             $\vec{d} \leftarrow \vec{r}_j - \vec{r}_i$
6:             $R^3 \leftarrow \|\vec{d}\|^3$
7:             $invR^3 \leftarrow 1/R^3$
8:             $\vec{a}_i \leftarrow \vec{a}_i + G\, m_j\, invR^3\, \vec{d}$
9:         end if
10:    end for
11: end for
    Time complexity (per process): $O\left(\frac{N}{P}\cdot N\right) = O\left(\frac{N^2}{P}\right)$
```

It follows that the total computational work remains $O(N^2)$, but the per-process workload is reduced to $O\left(\frac{N^2}{P}\right)$, assuming perfect load balancing and negligible communication overhead.

4.2 Theoretical Scalability

Scalability measures how much faster a parallel algorithm runs using P processes compared to running on a single process:

$$Scalability(P) = \frac{t_{parallel}(1)}{t_{parallel}(P)} \tag{14}$$

where $t_{parallel}(P)$ is the execution time of the parallel application using P processes.

Assuming no communication overhead and ignoring hardware-related effects, in the problem considered in this report, with $P = 1$, the running time can be expressed as

$$t_{parallel}(1) = N_1^2\,\bar{t} \tag{15}$$

where $\bar{t}$ is the time per iteration of the inner loop (Algorithm 2) and N_1 is the number of bodies in the parallel application with 1 process.

This implies that with $P > 1$ the running time is

$$t_{parallel}(P) = \frac{N_p^2}{P}\,\bar{t} \tag{16}$$

where N_p is the number of bodies in the parallel application with P processes.

Then it is possible to write:

$$Scalability(P) = \frac{N_1^2 \bar{t}}{\frac{N_p^2}{P} \bar{t}} = \frac{N_1^2}{N_p^2} P \tag{17}$$

Depending on whether the problem size, the total number of bodies N, is fixed or increases with P, two kinds of scalability measures are typically considered: *strong scalability* and *weak scalability*, discussed below.

Strong Scalability. Strong scalability measures how the execution time decreases when the number of processes P increases, while keeping the total problem size N fixed. In this case, $N_p = N_1 = N$ and the scalability formula reduces to

$$Scalability_{\text{strong}}(P) = P$$

Theoretically, the strong scalability of the parallel N-Body algorithm is ideal, since the same workload can be completed P times faster. The last equation highlights the fact in this case (direct-method N-body implementation) since there is no inherently sequential part as concluded in Subsection 3.2.4, analyzing strong scalability is essentially equivalent to analyzing speedup.

Weak Scalability. Weak scalability measures how the execution time changes when the number of processes P increases while keeping the *per-process data workload* fixed. In this case the number of bodies assigned to each process remains constant when P increases, thus it is possible to write $\frac{N_p}{P} = N_1$. Rearranging the scalability expression gives

$$Scalability_{\text{weak}}(P) = \frac{N_1^2}{(PN_1)^2} P = \frac{1}{P}.$$

This shows that the presented parallel N-Body algorithm exhibits poor weak scalability. Ideally, increasing both the number of bodies N and the number of processes P in the same ratio should maintain the execution time constant, i.e. $Scalability_{\text{weak}}(P) = 1$. However, this is not the case here: if both N and P are doubled, each process is still responsible for the same number of bodies as before, but the inner loop of the `computeAccelerations()` routine must iterate over all N bodies. This can also be seen directly from the per-process computational complexity $O\left(\frac{N^2}{P}\right)$: doubling both N and P results in a doubling of the per-process execution time.

As a consequence, although the *data workload* per process is constant, the *computational workload* grows with P, and the overall execution time of the parallel algorithm increases rather than staying constant. This is an inherent limitation of the direct method formulation in the N-Body problem.

4.3 MPI Implementation

The *Message Passing Interface* (MPI) is a standardized specification for developing parallel programs that execute across multiple processes, potentially on different machines [6]. MPI provides mechanisms for processes to communicate by sending and receiving messages, as well as for synchronizing their execution. For this project, version 4.1.1 of the *OpenMPI* implementation was used, targeting the C/C++ programming languages [7].

The full MPI implementation is provided in the file `parallel.c`, available in the remote repository associated to the project (see footnote 1).

4.3.1 Communication Strategy

With reference to Algorithm 4, each process requires access to the positions (and initially also the masses) of all bodies in the system in order to compute local accelerations. This necessitates both communication and synchronization between processes at each iteration.

In this work, the `MPI_Allgatherv` function was chosen for this purpose, since it simultaneously performs the required *sending*, *receiving*, and implicit *waiting*. The main advantages are:

- **Blocking collective communication:** every process broadcasts its local data to all other processes in the MPI communicator, ensuring that all participants have a consistent global view. Since the operation is blocking, no process can proceed until the data exchange of all processes has completed;
- **Consistent ordering:** the received items are gathered into arrays, with the same element order from the perspective of each process;
- **Flexibility:** unlike `MPI_Allgather`, the `MPI_Allgatherv` variant allows each process to contribute a different number of items, which makes it more general and adaptable.

The `MPI_Allgatherv` function is invoked as follows.

```
MPI_Allgatherv(local_positions, b,
    MPI_VECTOR, global_positions,
    recvcounts, displs, MPI_VECTOR,
    MPI_COMM_WORLD);
```

The address of the first element of the sending buffer is represented by *local_positions*. This buffer has b elements where b is the number of bodies each process has to manage. These elements are of type *MPI_VECTOR*, a properly defined MPI data type to wrap the struct *vect* already mentioned in Subsection 3.2.

```
void createVectorType() {
    int count = 3; // Custom MPI_Datatype
        will have 3 fields (x, y, z)
    int blocklengths[3] = {1, 1, 1}; //
        Each field is just one element
    MPI_Aint offsets[3]; // How elements in
         the struct are stored.
    offsets[0] = offsetof(vector, x); // 0
        byte offset from struct base as the
         'vector' struct
    offsets[1] = offsetof(vector, y); // 8
        byte
    offsets[2] = offsetof(vector, z); // 16
         byte
    MPI_Datatype types[3] = {MPI_DOUBLE,
        MPI_DOUBLE, MPI_DOUBLE}; // Each
        element of custom MPI_Datatype is
        an MPI_DOUBLE

    MPI_Type_create_struct(count,
        blocklengths, offsets, types, &
        MPI_VECTOR);
    MPI_Type_commit(&MPI_VECTOR); // MPI
        finalizes the definition and use it
}
```

This allows the positions (represented as 3D vectors) to be handled and sent in a clear and structured manner.

The array *recvcounts* contains as many entries as the number of MPI processes, since it tells the `MPI_Allgatherv` function how many elements each process contributes. When the number of bodies cannot be evenly divided among processes, the remainder r is distributed in a round-robin fashion: the first r processes each receive one additional body. This works because MPI assigns each process in the communicator a unique *rank*, from 0 to $P-1$ (with P the total number of processes). Consequently, the first r elements of the *recvcounts* array will be incremented by one compared to the others.

The array *displs* is used to ensure that the receiving buffer *global_positions* is properly organized, i.e., to specify at which offset the data received from each process should be stored. More precisely, the value at index i indicates the starting position in the receiving buffer where the block of elements broadcast by the process of rank i will be placed. The displacement depends on both the number of elements contributed by each process and the rank order. The following code shows how both *recvcounts* and *displs* are populated:

```
/**q = bodies / size, r = bodies % size;**/
void computeAllGathervParams(int*
    recvcounts, int* displs, int size, int
    q, int r) {
  for (int i = 0; i < size; i++) {
      if (i < r) {
         recvcounts[i] = q + 1;
         displs[i] = i * (q + 1);
     } else {
        recvcounts[i] = q;
        displs[i] = r * (q + 1) + (i - r) *
             q;
    }
  }
}
```

Memory remark. This communication strategy requires each process to store not only its local data ($3\frac{N}{P}$ positions, $3\frac{N}{P}$ velocities, $3\frac{N}{P}$ accelerations, and $\frac{N}{P}$ masses) but also the global arrays ($3N$ positions and N masses). This results in a per-process memory usage of

$$M(P) = 10\frac{N}{P} + 4N,$$

which asymptotically remains $O(N)$, dominated by the replicated global arrays. Consequently, memory requirements do not scale down with the number of processes.

4.3.2 Initialization and Testing

The initialization phase in the parallel application closely follows what was discussed for the serial application in Subsection 3.2.2, with the key difference that the seed of the random number generator is not determined solely by the local loop index i. Instead, it also depends on the global index of each body. As a consequence, the initial configuration of the parallel application can be made identical to that of the serial application because the evaluation of the global index reflects the ordered distribution of bodies across processes, mirroring the indexing scheme of the serial application.

This reproducibility allows a direct comparison of the outputs, which is essential for validating the correctness of the parallel algorithm.

The global index of the first body managed by each process is computed as follows:

```
unsigned int global_start_index(int r, int
    bodies, int rank){
     unsigned int start_index;
     if (rank < r) {
       return rank*bodies;
     } else {
       return r*(bodies+1) + (rank - r)*
           bodies;
       }
}
```

Then, during the initialization loop, the seed used for the local body i is set as:

```
unsigned int seed = start_index + i;
```

To practically compare the results a simple C program (`tester.c`) has been used. When it runs it opens both the output files of the serial and the parallel applications and checks value by value if they are equal within a tolerance introduced to allow the comparison between machines with different floating-point precisions.

4.4 Experimental Evaluation on GCP

In order to analyze the performance of the MPI implementation, it was necessary not only to rely on a multi-core architecture capable of running multiple processes concurrently, but also to distribute these processes across different machines in order to assess the impact of inter-process communication through MPI functions—in this case, `MPI_Allgatherv`.

To meet both requirements, multiple virtual instances were deployed on the Google Cloud Platform (GCP), using the Compute Engine service [8], which allows the creation and execution of virtual machines on Google's infrastructure.

These virtual instances were organized into experimental clusters, based on their hardware configuration and geographic location:

- **Single fat instance:** 1 machine with 16 vCPUs and 32 GB RAM, located in a single region (us-central1-a).
- **Inter-regional fat instances:** 2 machines with 8 vCPUs and 16 GB RAM each (16 vCPUs total), distributed across two regions: us-central1-a and europe-west9-a.
- **Intra-regional thin instances:** 4 machines with 3 vCPUs and 3 GB RAM each (12 vCPUs total), located in the same region (us-central1-a).
- **Inter-regional thin instances:** 4 machines with 3 vCPUs and 3 GB RAM each (12 vCPUs total), distributed across two regions: us-central1 and europe-west1.
- **Four-region thin instances:** 4 machines with 3 vCPUs and 3 GB RAM each (12 vCPUs total), distributed across four regions: us-central1, europe-west1, asia-northeast1, and me-central1.

All experiments carried out on GCP were constrained by the fact that the Google account used had a limited quota of 32 vCPUs across all regions. The raw data of the GCP experiments can be consulted in Appendix E where it is also presented the methodology used to obtain them.

Remark. All the above machines provide two hardware threads per physical core. Therefore, in principle, each configuration could run with up to twice as many MPI processes (e.g., 32 processes on the single fat instance instead of 16). However, in this work only one MPI process per physical core was used in order to focus on the impact of *network communication overhead.* Exploiting both hardware threads would introduce additional effects related to context switching, shared cache contention, and memory bandwidth limitations, which would make it harder to isolate the contribution of inter-process communication.

4.4.1 Strong Scalability Analysis

As discussed in Subsection 4.2, in this project strong scalability is effectively equivalent to speedup, since the parallel application with a single process can be regarded as the serial application and no inherently sequential part exists in the algorithm. Therefore, in the following analysis the terms *speedup* and *strong scalability* will be used interchangeably. In Subsection 3.2.4, the theoretical speedup was formulated using Amdahl's law, and it was concluded to be ideal since it scales linearly with the number of processes P (Equation (13)). However, it was also anticipated that the overhead introduced by communication through the network could significantly reduce the practical performance.

When increasing the number of processes while remaining on the same machine, the experimental results empirically confirm the theoretical analysis, as shown in Figure 6 for the single fat instance. In this configuration, the execution time of the parallel application halves when the number of processes is doubled (Figure 7).

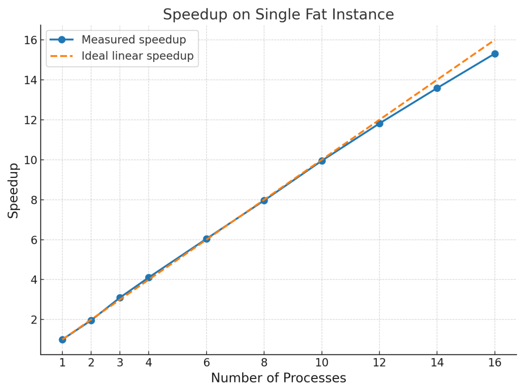

Figure 6: The single fat instance exhibits good strong scalability.

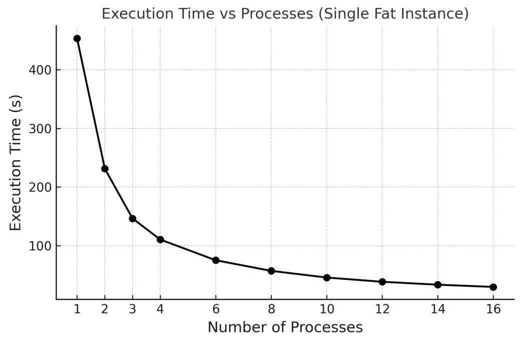

Figure 7: On the single fat instance, the running time halves each time the number of processes is doubled.

The raw measurements used to generate Figures 7 and 6 are reported in Appendix E. These data also show that the total time spent in communication, i.e. the cumulative time of `MPI_Allgatherv`, remains relatively stable across all executions and is negligible compared to the total running time in this configuration, as illustrated in Figure 8. Consequently, the achievable speedup of these configurations worsens significantly, as clearly observed in the execution times of the four-region cluster (Figure 9). This degradation can be explained by the fact that `MPI_Allgatherv` is a blocking collective operation: all participating processes must wait until every process has both sent and received its portion of data. When processes are distributed across geographically distant regions, the overall execution time becomes constrained by the slowest communication path. Thus, even if most processes complete their transfers quickly, a single high-latency or congested link can stall the entire computation, amplifying the communication overhead. Furthermore, it is important to point out that delays due to network latency and throughput are inherently unpredictable. For this reason, Figure 9 should not be interpreted as a precise measure of running time, but rather as evidence that—even when increasing the number of processes—it cannot be guaranteed that the execution time will decrease.

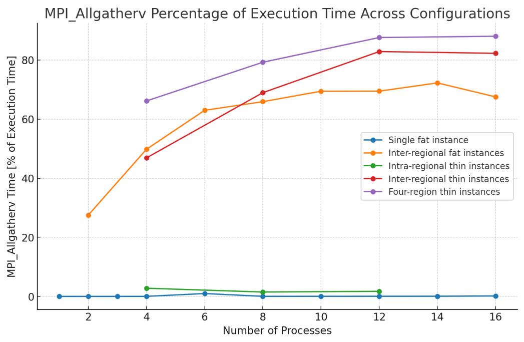

Figure 8: Percentage of execution time spent in `MPI_Allgatherv` across all GCP configurations.

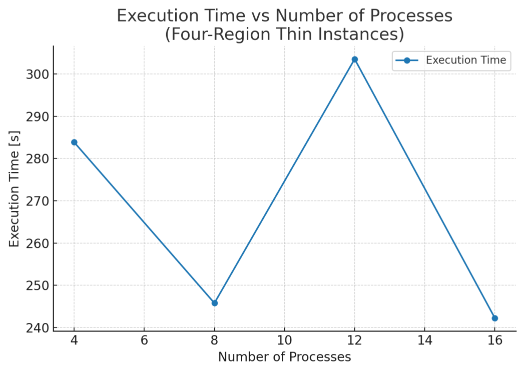

Figure 9: Execution times of the four-region cluster: communication overhead dominates, leading to poor strong scalability.

4.4.2 Weak Scalability Analysis

As proved in Subsection 4.2, the computational workload grows with P, thus the overall execution time of the parallel application increases rather than staying constant. Rearraging equation 14 we obtain:

$$t_{parallel}(P) = P \times t_{parallel}(1) \tag{18}$$

Hence, when doubling both the number of processes and the number of bodies, the running time is expected to double as well, which is exactly what is observed in Figure 10. Further consideration of other configurations beyond the single fat instance is unnecessary, because the weak scalability performance can only worsen due to the communication overhead.

Unfortunately, this is an inherent limitation of the direct method formulation in the N-Body problem. The simplest way to improve weak scalability is to relax the physics by neglecting the influence of distant (non-local) bodies. This reduces the inner loop (Algorithm 4) complexity to $O((\frac{N}{P})^2)$ which would make constant the per-process computational workload.

More scalable approaches involve switching to approximate or hierarchical algorithms such as Barnes–Hut or Particle Mesh, which reduce the computational complexity from $O(N^2)$ to $O(N \log N)$ or even $O(N)$, while maintaining acceptable physical accuracy. For a practical comparison of these techniques, see Blelloch and Narlikar [9].

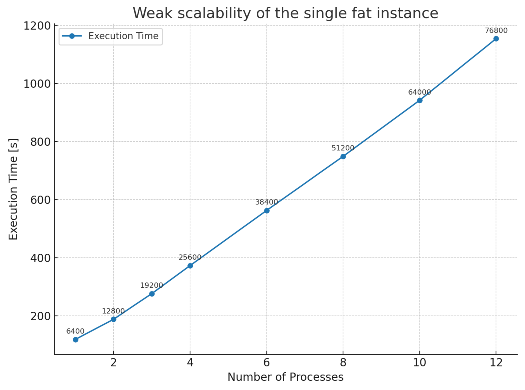

Figure 10: The parallel application exhibits a very poor weak scalability.

A Simulation Parameters for the Solar System Test

The following configuration was used for the simulation shown in Figure 2:

```
39.4784176 10 20000 0.001
1.0
0 0 0
```

```
0 0 0
3.00e-6
0.39 0 0
0 10.2 0
3.26e-7
0.72 0 0
0 7.4 0
3.00e-6
1.0 0 0
0 6.28 0
3.32e-7
1.52 0 0
0 5.1 0
9.50e-4
5.2 0 0
0 2.75 0
2.86e-4
9.58 0 0
0 2.03 0
4.36e-5
19.2 0 0
0 1.43 0
5.15e-5
30.1 0 0
0 1.14 0
1.02e-8
39.5 0 0
0 1.0 0
```

Where the first line specifies:

- Gravitational constant G
- Number of bodies N
- Number of steps T
- Time step Δt

and each body is defined by: mass, initial (x, y, z) position, and initial (v_x, v_y, v_z) velocity.

B Machine Specifications

Architecture	x86_64
CPU Model	AMD Ryzen 5 5625U with Radeon Graphics (12 threads, 6 cores)
Base/Max Clock	2.3 GHz / 4.388 GHz
L1 Cache	192 KiB (6 instances)
L2 Cache	3 MiB (6 instances)
L3 Cache	16 MiB (1 instance)
Memory	8 GB DDR4
OS	Ubuntu 22.04.5 LTS, 64-bit
Compiler	gcc 11.4.0 with -O2 optimization

C Serial Implementation Timing Data

The following table reports the execution times (in seconds) measured for the performance analysis in Subsection 2.2.

N (bodies)	Execution time (s)
100	0.012
200	0.049
400	0.198
800	0.796
1600	3.187
3200	12.732
6400	50.902
12800	203.671

Table 1: Execution times used to generate Figures 3 and 4.

D Profiling Results

This appendix reports profiling data for the serial C implementation. Two profiling tools were used:

- GNU *gprof*, to obtain a function-level breakdown of execution time.
- Linux *perf*, to collect low-level CPU performance metrics.

D.1 gprof Flat Profile (Summary)

The program was compiled with the `-pg` flag and executed under the same settings used in the performance analysis (§2.2) with the exception of $N = 10000$. The resulting `gmon.out` file was processed with:

```
gprof serial gmon.out > profiling_report.txt
```

Function	% Time	Self (s)	Calls
`computeAccelerations`	98.49	150.54	100
`_init`	1.49	2.28	—
`computeVelocities`	0.01	0.02	100
`logPositions`	0.01	0.01	101
`rand_uniform`	0.00	0.00	60000
`computePositions`	0.00	0.00	100
`simulate`	0.00	0.00	100
`initiateSystem`	0.00	0.00	1

Table 2: Distribution of execution time by function (gprof flat profile).

Please note that the `_init` function is not part of the N-body program; it is automatically inserted by the C runtime environment during the executable's startup sequence to perform system-level initializations before `main(` is called.

D.2 perf Results

The profiling command used was:

```
perf stat ./serial 12800
```

Running this command executes the serial version of the program with $N = 12\,800$ bodies and produces a performance summary directly on the terminal. The main statistics collected by `perf stat` are reported below.

```
Task-clock (ms)            268.400

CPU Utilization            1.000 CPUs

Context-switches           933     (3.48/s)

CPU Migrations             194     (0.72/s)

Page-faults                316     (1.18/s)

Cycles                     1.165e9 (4.341 GHz)

Stalled-cycles-frontend    1.590e6
                           (0.14% frontend idle)

Instructions               2.837e9 (2.43 IPC)

Branches                   1.152e8

Branch-misses              7.05e7
                         (0.06% of all branches)

Elapsed time (s)           268.410

User time (s)              268.398

System time (s)            0.002
```

E GCP Experiments Data

E.1 Measurement Methodology

The execution and communication times reported in this appendix and used to generate the figures in Subsection 4.4 were measured using the `MPI_Wtime()` function, which returns the elapsed wall-clock time in seconds. Specifically, timestamps were recorded immediately before and after the code sections of interest, and their difference was accumulated across iterations. For instance, the cumulative time spent in the `MPIAllgatherv` routine was measured as follows:

```
t7 = MPI_Wtime();
MPI_Allgatherv(local_positions, b,
    MPI_VECTOR, global_positions,
    recvcounts, displs,MPI_VECTOR,
    MPI_COMM_WORLD);
t8 = MPI_Wtime();
cumul_time_allgatherv3 += t8 - t7;
```

E.2 Single Fat Instance

#Proc	ExecTime[s]	CommTime[s]
1	453.49	0.003
2	231.26	0.010
3	146.31	0.012
4	110.36	0.021
6	75.06	0.730
8	56.97	0.017
10	45.58	0.018
12	38.38	0.018
14	33.38	0.020
16	29.62	0.045

E.3 Inter-regional Fat Instances

#Proc	ExecTime[s]	CommTime[s]
2	258.13	71.01
4	190.75	94.97
6	174.07	109.63
8	146.50	96.53
10	131.86	91.55
12	111.25	77.27
14	108.22	78.18
16	81.08	54.74

E.4 Intra-regional Thin Instances

#Proc	ExecTime[s]	CommTime[s]
4	96.00	2.66
8	48.58	0.72
12	32.55	0.56

E.5 Inter-regional Thin Instances

#Proc	ExecTime[s]	CommTime[s]
4	179.09	83.94
8	163.20	112.52
12	213.73	177.08
16	157.28	129.38

E.6 Four-region Thin Instances

#Proc	ExecTime[s]	CommTime[s]
4	283.83	187.73
8	245.79	194.76
12	303.51	265.88
16	242.24	213.26

E.7 Weak Scalability (Single Fat Instance)

#Proc	#Bodies	ExecTime[s]
1	6400	118.98
2	12800	188.44
3	19200	276.88
4	25600	373.24
6	38400	562.96
8	51200	749.22
10	64000	942.11
12	76800	1154.50

References

[1] D. C. Heggie, "The classical gravitational n-body problem," *arXiv preprint astro-ph/0503600*, 2005. [Online]. Available: https://arxiv.org/abs/astro-ph/0503600

[2] K. E. Atkinson, *An Introduction to Numerical Analysis*. John Wiley & Sons, 1989.

[3] R. D. Engle, R. D. Skeel, and M. Drees, "Monitoring energy drift with shadow hamiltonians," *Journal of Computational Physics*, vol. 206, no. 2, pp. 432–452, 2005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0021999104005212

[4] T. Cioaca and H. Caramizaru, "On the impact of explicit or semi-implicit integration methods over the stability of real-time numerical simulations," 2013. [Online]. Available: https://arxiv.org/abs/1311.5018

[5] R. Code, "N-body problem — rosetta code," 2025, [Online; accessed 11-August-2025]. [Online]. Available: https://rosettacode.org/wiki/N-body_problem?oldid=379856

[6] *MPI: A Message-Passing Interface Standard Version 4.1*, Message Passing Interface Forum, 2021. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf

[7] *OpenMPI Documentation*, OpenMPI Project, 2024. [Online]. Available: https://www-lb.open-mpi.org/doc/v4.1

[8] "Google cloud compute engine documentation," https://cloud.google.com/compute/docs, accessed: 2025-08-18.

[9] G. Blelloch and G. Narlikar, "A practical comparison of n-body algorithms," *Parallel Algorithms*, vol. 30, pp. 81–96, 1997.