N-Body Problem with MPI

Parallel Programming Essay

Sara Longhi

2025

Introduction

The N-body problem is a classic issue in physics and numerical simulation, which involves calculating the motion of N bodies that interact with each other through forces, typically gravitational ones. It is a generalization of the well-known two-body problem. While the latter is relatively straightforward to solve analytically, the three-body and, more generally, the N-body problems lack closed-form solutions. In other words, there is no single universal equation capable of predicting their future trajectories, which leads to highly intricate and often chaotic dynamics. The three-body problem is one of the most famous challenges in physics. Whittaker famously described it as "the most celebrated of all dynamical problems". It does not admit a unique, general solution and obtaining highly accurate results is notoriously difficult. Only a handful of exact solutions are known, such as those discovered by Euler and Lagrange. As mentioned earlier, there is no analytical solution in the general case, as the system is subject to deterministic chaos. This means that even tiny variations in initial conditions can produce vastly different trajectories, an effect often referred to as the butterfly effect or an avalanche phenomenon.
The primary objective of this work is to develop a simplified simulation of the N-body problem through a parallel implementation in C, based on the teachings from the Parallel Programming course by Andrea Polini at the University of Camerino [1]. In order to make the problem computationally tractable and focus on the core aspects of gravitational dynamics, several simplifications were introduced. The bodies are modeled as point masses, thus ignoring their physical size, rotational dynamics, and possible collisions. The motion is restricted to two dimensions, which reduces computational overhead while still capturing the essential behavior of gravitational interaction. Additionally, certain physical effects—such as the strict conservation of angular momentum, the precise conic trajectories of isolated bodies (elliptic, hyperbolic, or parabolic orbits), and the instantaneous variability of acceleration—are deliberately neglected. These approximations enable a clearer focus on the numerical and parallelization strategies rather than on modeling all the subtleties of celestial mechanics. The parallelization is implemented using the MPI (Message Passing Interface) library, which provides a standardized and efficient framework for communication between processes. This approach allows the computational workload to be distributed across multiple processors, significantly reducing execution time and enabling the simulation of systems with a larger number of bodies compared to a sequential approach.

MPI and its functions

MPI can be defined as a standard for parallel programming based on message passing, where multiple processes, each with its own local memory, cooperate to solve a computational problem. It is a single program, multiple data, or SPMD which means that it requires just a program to handle different tasks preformed by multiple processes.Unlike parallel programming with shared

memory (such as OpenMP), MPI uses a distributed memory model, in which processes do not share variables but must explicitly exchange information through messages. In message-passing programs, a program running on one core-memory pair is usually called a process, and two processes can communicate by calling functions: one process calls a send function and the other calls a receive function. This paragraph provides a brief introduction to the good practices and the main functions for writing an MPI Program, as outlined by Pacheco in his book [2].
Every MPI program begins with the `MPI_Init` function and ends with the `MPI_Finalize` function. Between these two calls, other MPI functions can be executed.It is good practice not to call MPI library functions after invoking `MPI_Finalize`, because all MPI-related states are cleaned up at that point.
Another key concept in MPI is the communicator, which is a collection of processes that can communicate with each other. When `MPI_Init` is called, a constant called `MPI_COMM_WORLD` is created. This constant belongs to the `MPI_Comm` enumeration and represents a communicator that includes all the processes involved in the program execution. From this communicator, it is possible to obtain:

- the number of processes using `MPI_Comm_size`
- the rank (unique identifier) of the process invoking the function using `MPI_Comm_rank`

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {

   MPI_Init(&argc, &argv);

    int comm_size, my_rank;

    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    //Some communication and computation

   MPI_Finalize();
}
```

When using MPI to enable communication between distinct processes, the most fundamental mechanism is message passing, typically implemented with the functions {`MPI_Send` for sending and `MPI_Send` for receiving. These functions operate within a distributed memory model, where each process has its own local memory, and the only way to share information is by explicitly sending it as messages. A call to `MPI_Send` requires the programmer to specify the buffer containing the data to be sent, the number of elements, the data type (e.g., integers, characters, floating-point values), the rank of the destination process, a tag to distinguish different types of messages, and a communicator that defines the scope of communication (commonly `MPI_COMM_WORLD`). On the receiving side, `MPI_Send` obtains the message by specifying a buffer to store the received data, the maximum receive size, the expected data type, the source (or `MPI_ANY_SOURCE` to accept from any sender), the tag (or `MPI_ANY_TAG`), the same communicator, and a status structure that provides information about the reception. The messages sent and received must be compatible: they must refer to the same communicator, have matching tags, and compatible data sizes. The implementation of `MPI_Send` and `MPI_Recv` provides two different behaviors: when the size of a single message is small, an internal buffer is used (the eager protocol), whereas for larger messages the buffer cannot be used, and a blocking behavior (the rendezvous protocol) is applied. If two processes establish mutual communication using `MPI_Send` and `MPI_Recv` , the result is that neither process will be able to activate the `MPI_Recv` call, causing the program to deadlock. A program that relies on an internal buffer created by MPI is said to be unsafe because it may work correctly under certain conditions but can fail or hang when the dataset changes. The standard version of `MPI_Send` is typically asynchronous from a logical point of view: the send operation can be considered complete even if the message has not yet been received, as long as it has been copied into an internal buffer provided by the MPI implementation.

```c
int MPI_Send(
    void *msg_buf,
    int msg_size,
    MPI_Datatype msg_type,
    int dest,
    int tag,
    MPI_Comm communicator);
```

```c
int MPI_Recv(
    void *buf,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm comm,
    MPI_Status * status);
```

As previously mentioned, a common problem in MPI programs is the potential for deadlock when two processes attempt to send data to each other simultaneously using `MPI_Send`, both waiting for the other to receive. To avoid this, MPI provides the `MPI_Sendrecv` function, which combines a send operation and a receive operation in a single call. A typical example is the exchange of data between adjacent processes in a ring or grid, where each process sends data to one neighbor and receives different data from another. `MPI_Sendrecv` ensures that the send and receive operations are internally coordinated, eliminating the risk of deadlock. `MPI_Sendrecv` is blocking: the function only returns when both the send and receive operations have completed. The send portion inside `MPI_Sendrecv` is asynchronous and uses temporary buffers when needed, but the overall operation is synchronized to prevent deadlocks.MPI also provides synchronous variants such as `MPI_Ssend`, where the send operation completes only when the receiver has actually started to receive the message, offering more predictable behavior in scenarios that require strict synchronization. Moreover, MPI offers non-blocking variants `MPI_Isend`, `MPI_Irecv`, which return immediately with a handle (called a request). The actual completion of the communication must then be checked with calls such as `MPI_Wait` or `MPI_Test`. Non-blocking communications are essential for overlapping computation and communication, reducing idle times and improving efficiency in large-scale parallel programs.

```
Initialize MPI

MPI_Request request;
MPI_Status status;

if (rank == source) {
    MPI_Isend(buffer, count, datatype, destination, tag,
        communicator, &request);
// ..Perform computations while sending..
    MPI_Wait(&request, &status);
}
else if (rank == destination) {
    MPI_Irecv(buffer, count, datatype, source, tag, communicator,
        &request);
// ..Perform computations while receiving..
    MPI_Wait(&request, &status);
}

Finalize MPI
```

Another way to communicate between processes is by using `MPI_Bcast`. While `MPI_Send`, `MPI_Recv`, and their variations are used for point-to-point communication, `MPI_Bcast` allows a single sender (usually the root process) to broadcast data to all other processes within a communicator. This is a blocking operation, meaning that execution is halted until all processes have received the data. `MPI_Ibcast`, the non-blocking version of `MPI_Bcast`, allows the broadcast operation to be initiated without waiting for its completion. Control is returned immediately to the calling process, which can perform other tasks while the broadcast progresses in the background. However, it is still necessary to call `MPI_Wait`(or a similar function) to ensure that the broadcast completes before accessing the communicated data.

```
MPI_Bcast(
    void *buffer,
    int count,
    MPI_Datatype datatype,
    int root,
    MPI_Comm comm);
```

```
MPI_Ibcast(
    void *buffer,
    int count,
    MPI_Datatype datatype,
    int root,
    MPI_Comm comm,
    MPI_Request *request);
```

Another important pair of functions provided by MPI are MPI_Scatter and its more flexible counterpart MPI_Scatterv. These functions are used when the root process needs to distribute portions of a dataset to all other processes within a communicator. The MPI_Scatter function is suitable when the data can be divided equally among all processes. In this case, each process receives the same number of elements, and the distribution is straightforward and uniform. It is often used in situations where the workload is homogeneous and can be easily split into equal parts. However, in real-world scenarios, it is common to encounter situations where the data cannot be evenly divided. This may happen, for example, when the total number of elements is not a multiple of the number of processes, or when certain processes need to handle more elements due to application-specific logic. In such cases, MPI_Scatterv becomes particularly useful. Unlike MPI_Scatter, MPI_Scatterv allows the root process to specify exactly how many elements to send to each process (sendcounts array), and from which position in the data buffer each process's portion should begin (displs array). This provides a high level of flexibility, enabling unbalanced but precise data distribution. Each process then receives its corresponding portion of data in a local buffer, whose size must match the amount of data it is supposed to receive. The parameters of the function include both the datatype of the elements and the rank of the root process, making it compatible with a wide variety of use cases and data structures.

```c
int MPI_Scatter(
    const void *sendbuf,
    int sendcount,
    MPI_Datatype sendtype,
    void *recvbuf,
    int recvcount,
    MPI_Datatype recvtype,
    int root,
    MPI_Comm comm
);
```

```c
int MPI_Scatterv(
    const void *sendbuf,
    const int *sendcounts,
    const int *displs,
    MPI_Datatype sendtype,
    void *recvbuf,
    int recvcount,
    MPI_Datatype recvtype,
    int root,
    MPI_Comm comm
);
```

`MPI_Gather` Another core function provided by MPI for collective communication is `MPI_Gather`. This function allows data to be collected from all processes in a communicator and assembled on a single designated process, known as the root. It is typically used at the end of a parallel computation when each process has produced a local result that needs to be combined or analyzed collectively. In a typical usage of `MPI_Gather`, each process provides a buffer containing a fixed number of elements (all processes must send the same amount), and the root process collects these elements into a single larger buffer. The data from each process is stored in the receive buffer on the root process in rank order. Another fundamental function in the MPI communication model is `MPI_Gatherv`, which is the more flexible version of `MPI_Gather`. The latter is used to collect equally sized data portions from all processes into a single buffer on the root process, `MPI_Gatherv` is designed to handle situations where the number of elements contributed by each process is not the same. This means that each process can send a different amount of data to the root, and the root process can gather this variable-sized input into a contiguous buffer. To do this, `MPI_Gatherv` requires two additional arrays on the root: recvcounts and displs. The former specifies how many elements each process will send, and the latter determines where in the final receive buffer each process's data should be placed.

```c
int MPI_Gather(
    const void *sendbuf,
    int sendcount,
    MPI_Datatype sendtype,
    void *recvbuf,
    int recvcount,
    MPI_Datatype recvtype,
    int root,
    MPI_Comm comm
);
```

```c
int MPI_Gatherv(
    const void *sendbuf,
    int sendcount,
    MPI_Datatype sendtype,
    void *recvbuf,
    const int *recvcounts,
    const int *displs,
    MPI_Datatype recvtype,
    int root,
    MPI_Comm comm
);
```

Implementation

This chapter discusses the implementation choices made in developing a solution to the N-body problem. Although the physical aspects of the problem are important, the primary focus here is on the parallelization approach adopted to improve computational efficiency. In particular, this chapter examines how the Message Passing Interface [3] standard was used to implement

the distribution of the computational workload across multiple processes. Various strategies were explored to optimize communication and data distribution between processes. Attention is given to the methods used to divide the problem domain among processes, the communication patterns adopted (including point-to-point and collective operations), and the reasoning behind the choice of specific MPI routines.
Before delving into the actual implementation of the program, we begin by describing the proposed solution to the *n-body problem*.
The *n-body problem* is a classical challenge in computational physics that involves determining the position and velocity of a set of n bodies interacting with each other through forces, typically gravitational. Each body exerts a force on all the others, and the sum of these forces determines each body's acceleration according to Newton's second law. This results in a computational complexity of $O(n^2)$, as every body must interact with every other body.
As the number of bodies increases, the computational cost grows significantly, making the problem well-suited for parallelization. For this reason, the goal of this project was to implement a parallel version of the problem using the MPI library, allowing the computational workload to be distributed among multiple processes to reduce execution time.
The simulation of the n-body problem is based on an iterative approach in which, at each time step, the position and velocity of each body are updated according to the gravitational interactions with all other bodies in the system.
Initially, for each pair of bodies A and B, the gravitational acceleration is computed according to Newton's law:

$$a = \frac{Gm_B}{r_{AB}^2}$$

where G is the gravitational constant and r_{AB} is the distance between the two bodies. Next, the magnitude of the displacement d experienced by body A during the time interval t is computed, considering its initial velocity and the gravitational acceleration produced by body B. The displacement is determined from the kinematic relation that includes both the contribution of the initial velocity and the acceleration:

$$d = \sqrt{v_{0X_A}^2 + v_{0Y_A}^2} + \frac{1}{2}a_A t^2$$

The displacement vector is then decomposed into its Cartesian components:

$$d_{XA} = d\frac{x_B - x_A}{r_{AB}}$$

$$d_{YA} = d\frac{y_B - y_A}{r_{AB}}$$

With this information, the velocity of body A is updated along both axes. The velocity components are updated by taking into account the acceleration due to the gravitational force:

$$v_{X_A} = v_{0X_A} + (\sqrt{v_{0X_A}^2 + v_{0Y_A}^2} + a_A t)\frac{x_B - x_A}{r_{AB}}$$

$$v_{Y_A} = v_{0Y_A} + (\sqrt{v_{0X_A}^2 + v_{0Y_A}^2} + a_A t)\frac{y_B - y_A}{r_{AB}}$$

Finally, the new position of body A is obtained by updating its coordinates x_A and y_A .

$$x_A = x_{0A} + d_{XA}$$

$$y_A = y_{0A} + d_{YA}$$

Two short programs were developed in C: one using blocking communication `nBody.c` [4] and the other using non-blocking communication `nBodyNB.c` [5] , in order to analyze how these choices affect the speed and efficiency of the application. The structure of both programs and the computations performed are identical. By keeping all other aspects constant and changing only the communication methods, it becomes possible to draw meaningful conclusions about the behavior and performance of these different MPI communication strategies.
The first step involved defining a structure to represent a body, characterized by its basic physical properties: mass, position coordinates, and velocity coordinates. Since, as explained in the introduction, the system is considered to be two-dimensional, the coordinates refer specifically to the x and y axes. A custom MPI datatype, named `MPI_BODY`, was created to allow structured communication of these elements. To define this custom datatype correctly, it is essential to consider concepts such as offset and padding. Each data member, depending on its type, must be properly aligned in memory (data cannot be stored arbitrarily). For instance, a `double` typically needs to be stored at a memory address that is a multiple of 8 bytes. The offset, in this context, refers to the byte distance of a member from the beginning of the struct. Proper alignment and offset calculation are crucial to ensure the correctness and portability of the MPI data structure. After that, the bodies are initialized either through random generation of their fields or, alternatively, by reading data from a file (these behavior can be choose through a command provided by the user). Once initialized, the bodies are stored in an array called `bodies`. At this stage, the number of bodies to be distributed to each process is computed, along with the corresponding displacement array Fig.1. Each process is assigned a specific portion of the bodies array. The displacement is stored in an array of size p (where p is the number of processes), where each element represents the index of the first body that a given process is responsible for. This way, each process knows exactly how many bodies it needs to handle, as well as the exact starting point within the array from which to begin processing. This computation, the `bodiesPerProcess` and `displ` are used in the `MPI_Scatterv` function.

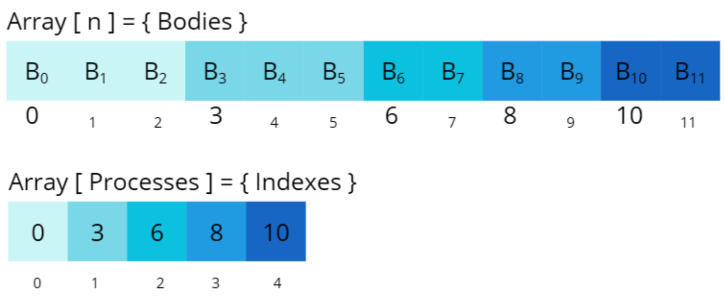

Array [n] = { Bodies }
B_0 B_1 B_2 B_3 B_4 B_5 B_6 B_7 B_8 B_9 B_{10} B_{11}
0 1 2 3 4 5 6 7 8 9 10 11
Array [Processes] = { Indexes }
0 3 6 8 10
0 1 2 3 4

Figure 1: Visualization of how the displacement array maps data to MPI processes.

Using the `MPI_Bcast` function, the root process broadcasts both the displacement array and the number of bodies assigned to each process. Then, with `MPI_Scatterv`, the root distributes the corresponding portions of the bodies array to all processes, while each non-root process receives exactly the number of bodies assigned to it. During each iteration, the new coordinates and velocities of the bodies are computed, assuming a constant acceleration. Before any data exchange occurs, each process performs a partial computation of the new positions and velocities using only the subset of bodies it holds locally. The idea is to organize the processes in a ring topology, where each process is responsible for updating only the bodies assigned to it. However, to perform this update correctly, each process must have knowledge of all other bodies in the

system. To achieve this efficiently, each process forwards the subset of bodies it received to the next process in the ring. This data passing operation is repeated p times (where p is the total number of processes). With each reception of a new subset of bodies, the partial calculations for position and velocity are updated. After the final step, all required interactions have been computed, and the final state of each body is updated accordingly.
In the blocking communication program, the `MPI_Sendrecv` method was used. This function belongs to the MPI-1 standard and was designed as a highly optimized and less problematic alternative to the blocking methods `MPI_Send` and `MPI_Recv`, which, as mentioned in the previous chapter, can cause situations of stalling or deadlock if not handled properly. With `MPI_Sendrecv`, which is more performant, the send and receive actions are executed within a single call.
The implementation using `MPI_Sendrecv` is much simpler and more straightforward compared to the non-blocking version used in `nBodyNB.c`.

```
MPI_Sendrecv(
     sendBuffer, sendCount, MPI_BODY, next, 0,
     recvBuffer, recvCount, MPI_BODY, prev, 0,
     MPI_COMM_WORLD, MPI_STATUS_IGNORE
 );
```

It is important to note that in the non-blocking version with `MPI_Isend` and `MPI_Irecv`, the deadlock problem is not completely resolved and therefore must be managed carefully. Another challenge arising in the implementation of `nBodyNB.c` is the management of buffer sizes. Two receive buffers are declared: one buffer is used in the `MPI_Irecv` call from the previous node `prev`, and the other is the buffer received in the previous step and sent using `MPI_Isend` to the next node, `next`.

```
MPI_Isend(recvBuffer[1 - current], bodiesPerProcess[(senderRank + 1) %
   numberOfTasks], MPI_BODY, next, 0, MPI_COMM_WORLD, &sendReq);
```

```
MPI_Irecv(recvBuffer[current], recvCount, MPI_BODY, prev, 0,
   MPI_COMM_WORLD, &recvReq);
```

Finally, the updated bodies with their new attributes are written to the text file `bodies.txt`, while the execution times are both printed to the console and saved to a separate text file `nBodyExecutionTime.txt`.

Results

A comparison was made between the two programs in terms of execution time, with particular focus on speed and efficiency. The goal of this implementation was to test certain hypotheses, for example, how execution time would vary based on specific parameters such as input size, the number of processors used in parallelization, and code optimization.
Verifying the first two assumptions (execution time variation based on input size and number of processors) was relatively straightforward. In both programs, the expected proportional relationship was observed: as the size of the input increased, so did the time required to compute the new state of the body (position and velocity in the plane). When the input was kept constant and the number of processors was increased, an inversely proportional relationship was expected, and was indeed confirmed.

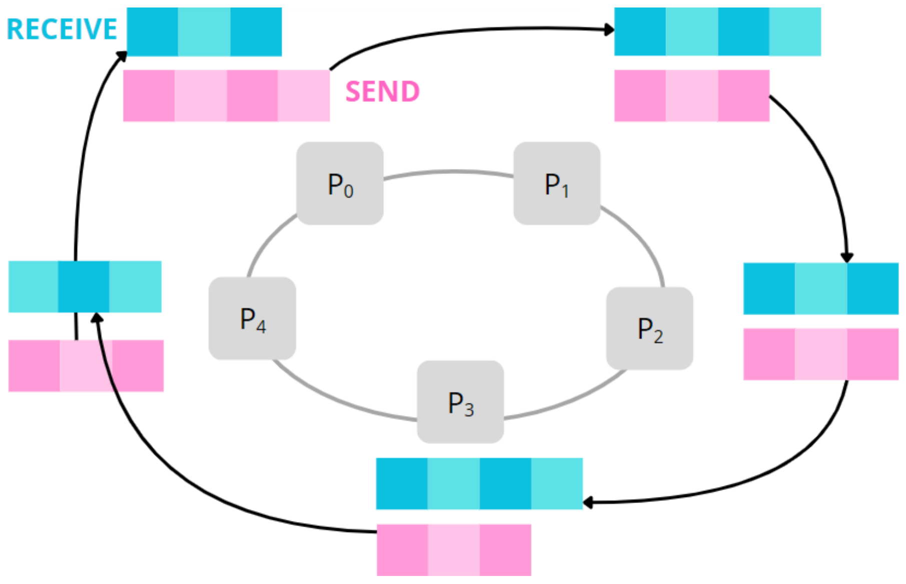


Figure 2: In this image is clarified how the recvBuffer[0] and recvBuffer[1] are used by processes and their different size

It is important to note that, during this testing phase, the input data was always read from the file data.txt rather than being generated randomly. This choice was made to ensure consistent computational complexity across all executions. Using randomly generated data would have introduced variability in complexity, which could have significantly impacted execution times. By consistently reading from the same dataset, the complexity of the calculations was kept constant, ensuring fair and reliable comparisons.
The selected dataset contains 10,000 bodies, and it was decided to perform only two iterations. Since subsequent iterations depend on the results obtained in the previous ones, the number of iterations was limited to clearly highlight the difference between execution with and without parallelization.
Initially, a significant difference in execution times between the two programs was expected, especially in scenarios where resource demands increase rapidly. The analysis of the speed-up and efficiency graphs 3 for the two programs, nBody and nBodyNB, provides insight into their behavior as the number of processors increases. In both cases, speed-up improves with more processors, confirming that parallelization does reduce execution time. However, this improvement is not linear. Up to 4 processors, nBodyNB performs slightly better, likely due to lower overhead in managing parallelism. Beyond that point, nBody shows better scalability, reaching a maximum speed-up of approximately 4.2× with 12 processors, compared to 3.8× for nBodyNB.

As for efficiency, both programs show a decreasing trend, as expected from Amdahl's Law [6]: as the number of processors increases, the marginal gain in performance decreases due to non-parallelizable sections of the code and communication overhead. Efficiency remains above 0.5 up to 4–6 processors, but drops below that threshold at 8 and 12 processors, indicating a progressively less effective use of computational resources.
A key difference between the two programs lies in their MPI communication model. nBody uses blocking communication with MPI_Sendrecv, whereas nBodyNB implements non-blocking communication using MPI_ISend and MPI_Irecv. In theory, the non-blocking model allows for overlapping communication and computation, which can improve overall efficiency by reducing idle time.

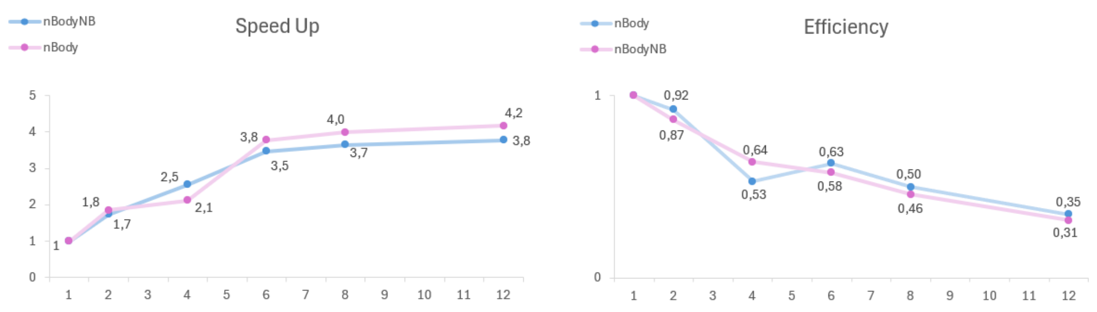
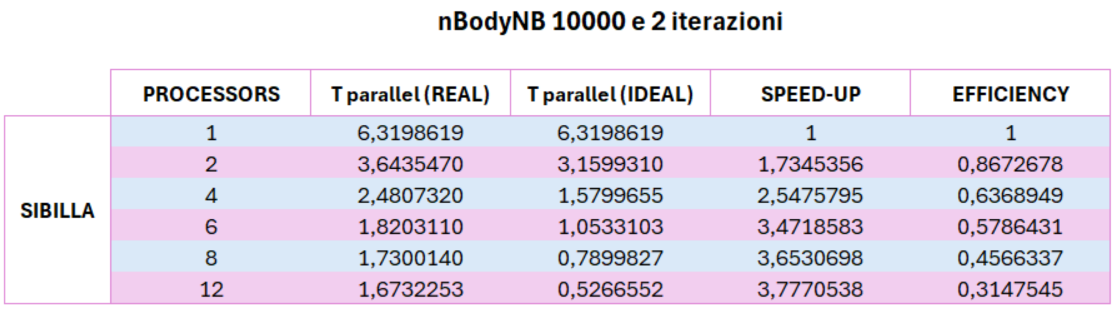


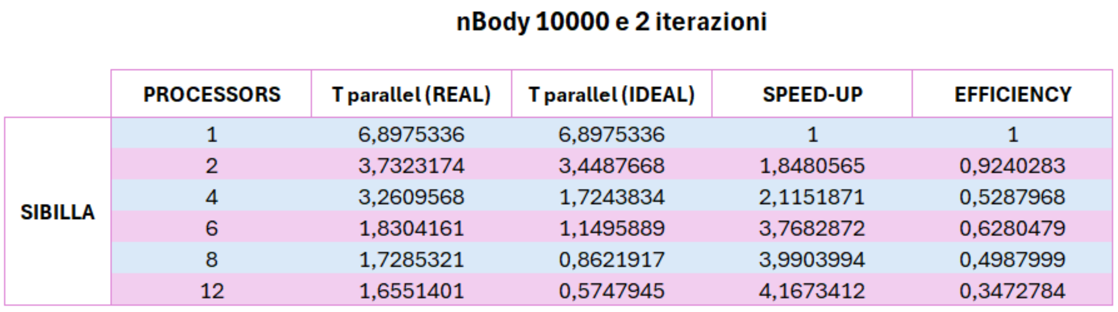
Figure 3: Comparison of speedup and efficiency between the nBody and nBodyNB algorithms across varying processor counts.



nBodyNB 10000 e 2 iterazioni

	PROCESSORS	T parallel (REAL)	T parallel (IDEAL)	SPEED-UP	EFFICIENCY
SIBILLA	1	6,3198619	6,3198619	1	1
	2	3,6435470	3,1599310	1,7345356	0,8672678
	4	2,4807320	1,5799655	2,5475795	0,6368949
	6	1,8203110	1,0533103	3,4718583	0,5786431
	8	1,7300140	0,7899827	3,6530698	0,4566337
	12	1,6732253	0,5266552	3,7770538	0,3147545

Figure 4: The image presents performance data for the nBody algorithms with 10000 particles over 2 iterations, showing real and ideal parallel execution times, speed-up, and efficiency across different processor counts.



nBody 10000 e 2 iterazioni

	PROCESSORS	T parallel (REAL)	T parallel (IDEAL)	SPEED-UP	EFFICIENCY
SIBILLA	1	6,8975336	6,8975336	1	1
	2	3,7323174	3,4487668	1,8480565	0,9240283
	4	3,2609568	1,7243834	2,1151871	0,5287968
	6	1,8304161	1,1495889	3,7682872	0,6280479
	8	1,7285321	0,8621917	3,9903994	0,4987999
	12	1,6551401	0,5747945	4,1673412	0,3472784

Figure 5: The image presents performance data for the nBodyNB algorithms with 10000 particles over 2 iterations, showing real and ideal parallel execution times, speed-up, and efficiency across different processor counts.

Initially, it was expected that nBodyNB would outperform nBody, precisely because non-blocking communication should enable better performance by hiding communication latency. However, in the specific context of this simulation, that advantage does not clearly emerge, especially at higher processor counts. A likely explanation is that the computational time per iteration is not large enough to allow meaningful overlap between computation and communication. When the computation phase is too short, the non-blocking communication cannot complete in parallel and becomes effectively synchronous, negating its intended benefit.
When there is a low workload of computation, the benefits of non-blocking communication may not be evident, or it could even be disadvantageous. The overhead of managing non-blocking communication can outweigh the benefits of parallelization, as the computations do not take enough time to justify the time spent on managing the communication.
In conclusion, although nBodyNB is based on a more advanced and theoretically more efficient communication model, the characteristics of this specific problem—particularly its low computational intensity—make it difficult to leverage the potential of non-blocking communication. As a result, nBody, with its simpler and more linear blocking communication model, ends up delivering slightly better performance in highly parallel configurations.

References

[1] A. Polini. *Lecture notes of the Parallel and Distributed Programming course at the university of Camerino 2024/2025* `http://didattica.cs.unicam.it/doku.php?id=didattica:ay2425:pdp:main`

[2] Pacheco, Peter. *An Introduction to Parallel Programming.* 1st ed., Morgan Kaufmann Publishers Inc., 2011.

[3] Dongarra, Jack J., Hempel, Rolf, Hey, Anthony J.G., and Walker, David W. *MPI: A Message Passing Interface.* In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (Supercomputing '95)*, pp. 878–883. IEEE Computer Society Press, Los Alamitos, CA, USA, December 1995. DOI: `https://doi.org/10.1109/SUPERC.1995.246134`

[4] S.Longhi. *nBody.c source code*, 2025. Available at: `https://github.com/saralonghi/nBodyProblem_MPI/blob/master/nBody.c`

[5] S.Longhi. *nBodyNB.c source code*, 2025. Available at: `https://github.com/saralonghi/nBodyProblem_MPI/blob/master/nBodyNB.c`

[6] Amdahl, Gene M. *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities.* Reprinted from the AFIPS Conference Proceedings, Vol. 30 (Atlantic City, N.J., Apr. 18–20), AFIPS Press, Reston, Va., 1967, pp. 483–485. Reprinted in *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 3, 2007, pp. 19–20. DOI: `https://doi.org/10.1109/N-SSC.2007.4785615`