MPI Assignment: N-body Simulations

Stefano Sandonà

Vrije Universiteit Amsterdam, Holland

1 The N-body problem

The N-body problem consists of the prediction of the individual motions of N celestial objects (bodies) with some properties (mass, initial velocity, radius, ...). This is done by measuring the force that they exert on each other (Coulomb gravity,...) and as a result, a simulation of the behavior of the system over time is obtained.

2 The sequential algorithm

The Listing 1 shows the simplified structure of the sequential algorithm.

```
for each timestep do
        Compute forces between all bodies
        Compute new positions and velocities
```

Listing 1: general sequential algorithm

As first thing, the time is discretized, so that the application knows what to do at certain time steps. After that, for each step, the algorithm computes the force between the bodies. With the result, the new velocity and position of all the bodies are updated.

2.1 Force computation

This step of the algorithm is the most expensive and defines its complexity. The force that bodies exert on each other has to be calculated once per pair of bodies (Listing 2).

```
for (b=0; b<bodyCt; ++b) {
                for (c=b+1; c<bodyCt; ++c) {
                        ...
```

Listing 2: loop for the force calculation

With N bodies, there are *(N-1)+(N-2)+(N-3)+...+1* pairs, so that $O(N^2)$ force computations for every time step.

3 Parallel N-body algorithm

An efficient parallelization of this algorithm is not trivial because for every step the updated information from all the bodies that are part of our system is needed in order to calculate the correct velocities and positions. The next paragraph was introduced to explain the technology used to develop the parallel implementation.

3.1 MPI

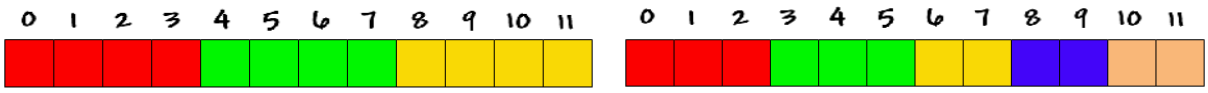
For this project was used the Message Passing Interface (MPI), a message passing library that is developed and maintained by a consortium of academic, research, and industry partners, that can be added to sequential languages (C, Fortran) to program multiple nodes. Using this library, to set up a point-to-point or collective communication is simplified. In addition to High performance, network and process fault tolerance is provided, making this library optimal for the parallel programming.

3.2 Work distribution

One of the major problems of the parallel algorithms is the load imbalance. If the work is not fairly distributed among the machines involved in the computation, there will be some machines idle while the others will still work. This problem affects a lot the performance of an application because the overall execution time depends on the last machine that terminate the computation. Looking at this specific case, there were two possible ways to follow: a fair distribution of the bodies or a fair distribution of the forces to compute. The major differences between this 2 approaches were encountered when the amount of work to distribute was not perfectly divisible by the number of machines. Using a simple *for* construct (Listing 3), it is possible to calculate the chunks of forces/bodies to assign to each computational node. The Figure 1 shows two examples of repartition.

```
int sum = 0;
int rem = workAmount % numProcs;
for (i = 0; i < numprocs; i++) {
    work_per_proc[i] = workAmount / numProcs;
    if (rem > 0) {
        work_per_proc[i]++;
        rem--;
    }
    displs[i] = sum;
    sum += work_per_proc[i];
}
```

Listing 3: fair distribution of work



(a) 3 computational nodes (b) 5 computational nodes

Figure 1: Repartition of 12 forces/bodies

A comparison of the two approaches was reported on the next paragraphs in order to clarify the made decisions.

3.2.1 Approach 1 - Bodies repartition

The total amount of bodies (known at the beginning) is distributed as fairly as possible among the MPI processes as shown in the Listing 3. Then, to avoid a duplication of work, given a pair of bodies chunks, half of the forces are calculated by one MPI node, and half by the other. The repartition is clarified in the Figures 2, 3 and 4.

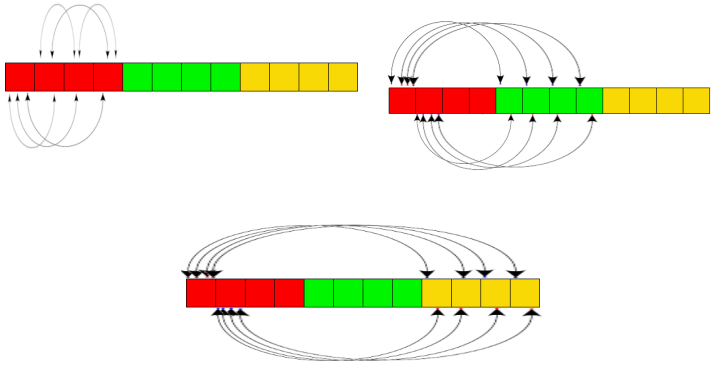


Figure 2: Forces computed by Node 0 (6+8+8=22)

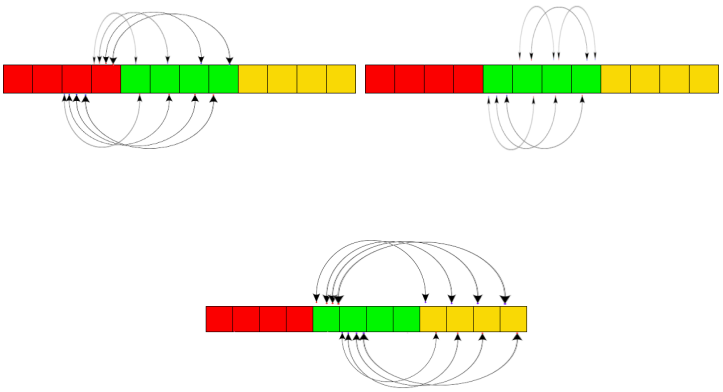


Figure 3: Forces computed by Node 1 (8+6+8=22)

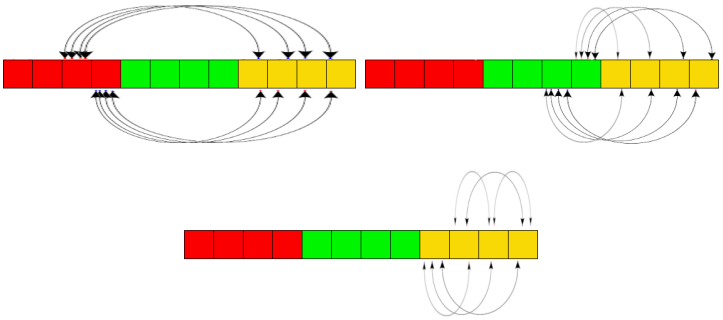


Figure 4: Forces computed by Node 2 (8+8+6=22)

Proceeding in this way, the forces between all pairs of bodies are calculated only one time per step. Each node computes the forces of 22 pairs. There are 3 nodes, so 3 times 22 is 66 that correspond to the number of possible pairs. No useful work is done. The approximated computational time is shown in the Equation 1. The first O represents the computation of the forces between the assigned chunk's bodies and the bodies of other chunks, while the second O represents the computation of the forces among the assigned chunk's bodies.

$$Complexity = O(\lceil \frac{N}{P} \rceil * (\lceil \frac{N}{P} \rceil * \frac{1}{2}) * (P-1)) + O(\sum_{i=1}^{\lceil \frac{N}{P} \rceil} (\lceil \frac{N}{P} \rceil - i)) \tag{1}$$

3.2.2 Approach 2 - Forces repartition

With this approach, the bodies are not taken into consideration, but only the forces to compute, resulting in a more simple and intuitive way to proceed. After calculating the total amount of forces to compute per step (Listing 4), these can be simply distributed as shown in Listing 3. The approximated computational time is shown in the Equation 2.

```
int forceCt = 0;
    for(i = 0; i < bodyCt; i++) {
        forceCt += i;
    }
```

Listing 4: total amount of forces

$$Complexity = O(\lceil \frac{N^2}{P} \rceil) \tag{2}$$

3.2.3 Load imbalance

As mentioned before, the load imbalance is a painful aspect of the parallel programming. If the total amount of work is perfectly divisible by the number of MPI processes, the two approaches are not so different, in fact, the number of forces calculated per machine is the same. The problems come out, when the work (bodies or forces) is not equally distributed. With the second approach, the inequity is simply reduced at the minimum. The maximum work difference between 2 nodes is of one force, that means less than 1% of load imbalance if the assigned forces per machine are more than 100. To reach this bound with 16 MPI nodes, for example, 58 bodies (1653 forces) are sufficient. With the First approach is more difficult to balance the work, because the assigned number of bodies is different, and from one body the forces that exerts on all the others has to be calculated, that means a load imbalance of *O(N)*. For this reason the second approach appeared more simple and efficient, so that it is adopted.

3.3 The final solution

3.3.1 Communication VS Computation

The communication is a common bottleneck of lots of parallel implementations. The data exchange between nodes should be reduced at the minimum in order to obtain good performances. The problem with the N body algorithm is that at every step the updated information (positions, velocities) of all other bodies of the system is needed in order to correctly compute the forces of the assigned chunk.. To gain performances, a good compromise between the data exchange and the work to do on each node has to be found. In particular, the ratio *computation/communication* should be maximized.

3.3.2 Approach one

Due to the fact that the force computation is the most expensive part of the algorithm and that as less data as possible should be exchanged, a good compromise is the following. The force (x, y) applied to a certain body is separated from the structure Body type, so at the beginning there are an array of N bodies and an array of N forces. As first thing, the bodies are broadcasted to all the nodes (*MPI_ Bcast* Figure 5). After that, at each step, each node calculates the assigned forces, then with an *MPI_ Allreduce* operation, all calculated arrays of N forces are summed (Figure 6) and as the last thing the velocities and positions for all the bodies are computed. Proceeding in this way, only an array of N elements containing the calculated forces is exchanged with the other nodes. The major problem with this approach, is that the positions and velocities of all the bodies are calculated in every node, that represents a duplication of work. An approximation of the complexity of a loop iteration and of the total amount of data exchanged per step is presented in Equations 3 and 4. In the first equation, the first O represents the force computation step, the second the velocities computation and the third the positions computation. In the second equation, the *2* represents the (x, y) values of the force.

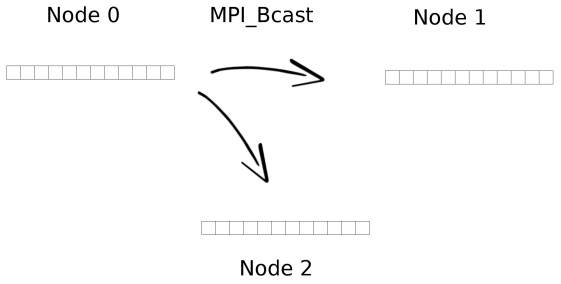


Figure 5: Broadcast the N bodies to all the nodes

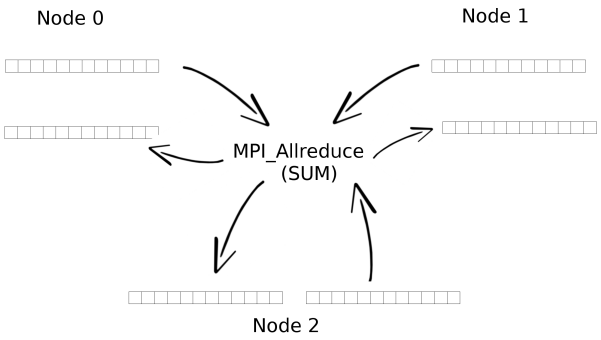


Figure 6: The N forces reduction

$$Complexity(perstep) = O(\lceil \frac{N^2}{P} \rceil)) + O(N) + O(N) \tag{3}$$

$$Communication(perstep) = O(2 * N) * sizeof(double) \tag{4}$$

3.3.3 Approach two

The aim of this approach, is to avoid the work duplication during the positions and velocities computation step. To calculate the force that two bodies exert on each other, at each step their mass, radius and position is needed. As first thing, the position (x, y) and the force of each body are separated from the Body type, so at the beginning there are an array of N bodies, an array of N positions and an array of N forces. After that, the bodies and the positions are broadcasted to all the nodes and each node calculates its range of assigned bodies and positions (with the method of Listing 3). The force calculation step is the same as the one of the previous approach (calculation and MPI_Allreduce), afterwars the assigned positions and bodies are updated. Before start another iteration, with an MPI_Allgatherv operation (Figure 7) the positions are gathered from all the nodes to all the nodes in order to have all the needed updated information for the next step. At the end, with an MPI_Allgatherv operation the final bodies are gathered from all the nodes to the "master" node that prints the results. An approximation of the complexity of a loop iteration and of the total amount of data exchanged per step is presented in Equations 5 and 6. In the first equation, the first O represents the force computation step, the second the velocity computation and the third the position computation. In the second equation, the 4 represents the (x, y) values of the force and the (x, y) values of the position.

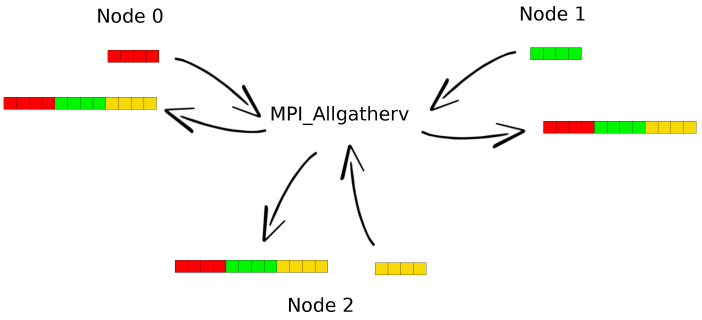


Figure 7: B - Collect updated positions from other nodes

$$Complexity(perstep) = O(\lceil \frac{N^2}{P} \rceil)) + O(\lceil \frac{N}{P} \rceil) + O(\lceil \frac{N}{P} \rceil) \tag{5}$$

$$Communication(perstep) = O(4 * N) * sizeof(double) \tag{6}$$

3.3.4 Solutions comparison

After a comparison of the two solutions for different size of inputs (steps and number of bodies), it's clear that the amount of data exchanged is the key point. In Tables 1 and 2 (corresponding graphs in Figure 8) and in Tables 3 and 4 (corresponding graphs in Figure 9) we can see execution times and speedups of the 2 approaches for different problem sizes. It's easy to understand that the first approach present lower execution times than the other and so better speedups. That means that is more convenient to duplicate a bit the work instead of exchange more data with the other nodes. The first approach so is better.

nodes\approach	1	2
1	107.856	107.856
2	63.612	83.450
4	33.939	45.068
8	19.566	26.229
10	16.916	23.593
16	12.584	17.523

Table 1: 128 bodies and 100000 iterations - Execution times

nodes\approach	1	2
2	1.69	1.29
4	3.17	2.39
8	5.51	4.11
10	6.37	4.57
16	8.57	6.15

Table 2: 128 bodies and 100000 iterations - Speedups

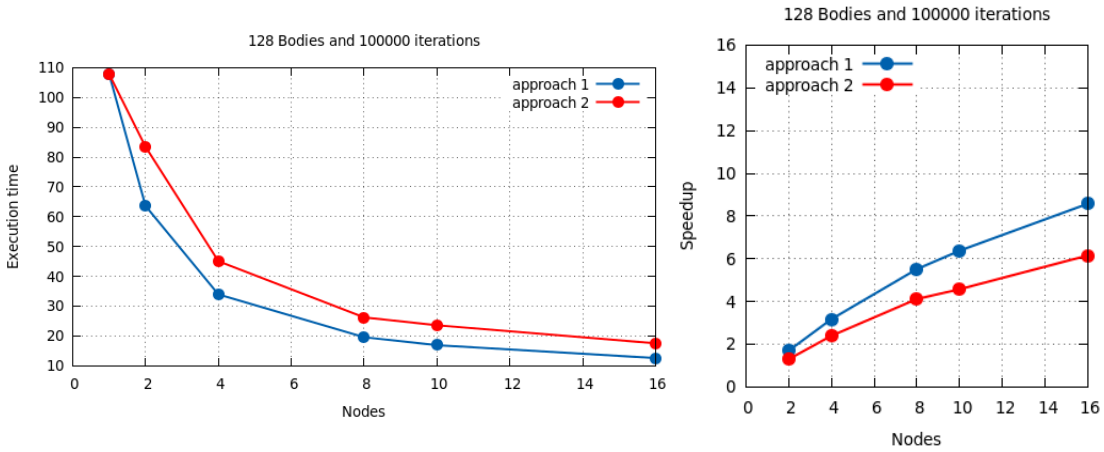


Figure 8: Execution Time and Speedups of the two approaches with 128 bodies and 100000 iterations

nodes\approach	1	2
1	655.162	655.162
2	373.328	481.161
4	186.764	240.538
8	93.483	120.394
10	74.908	96.431
16	47.192	60.563

Table 3: 10000 bodies and 100 iterations - Execution times

nodes\approach	1	2
2	1.75	1.36
4	3.5	2.72
8	7	5.44
10	8.74	6.79
16	13.88	10.81

Table 4: 10000 bodies and 100 iterations - Speedups

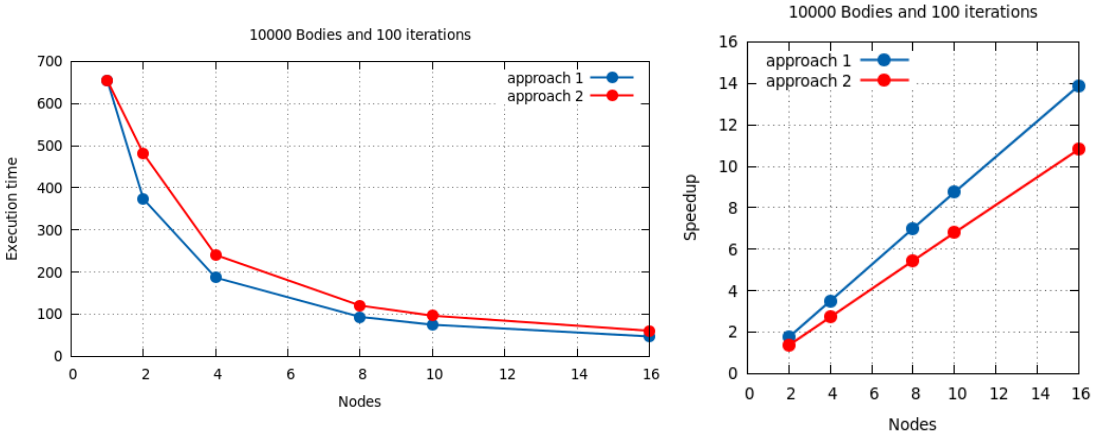


Figure 9: Execution Time and Speedups of the two approaches with 10000 bodies and 100 iterations

3.3.5 Results

In this section the most interesting results are reported. In Figure 10 and Tables 5, 6 we have an example with a big number of iterations (fixed) and different numbers of bodies (small). In Figure 11 and Tables 7, 8 we have an example with a small number of iterations (fixed) and different numbers of bodies (big). As we can see, the more we increase the size of the problem (number of bodies) and the more the implementation obtains good speedups. The aim of parallel programming is to deal with big size problems, so this implementation is good due to the fact that the more we increase the number of bodies, the more the speedup is close to be linear.

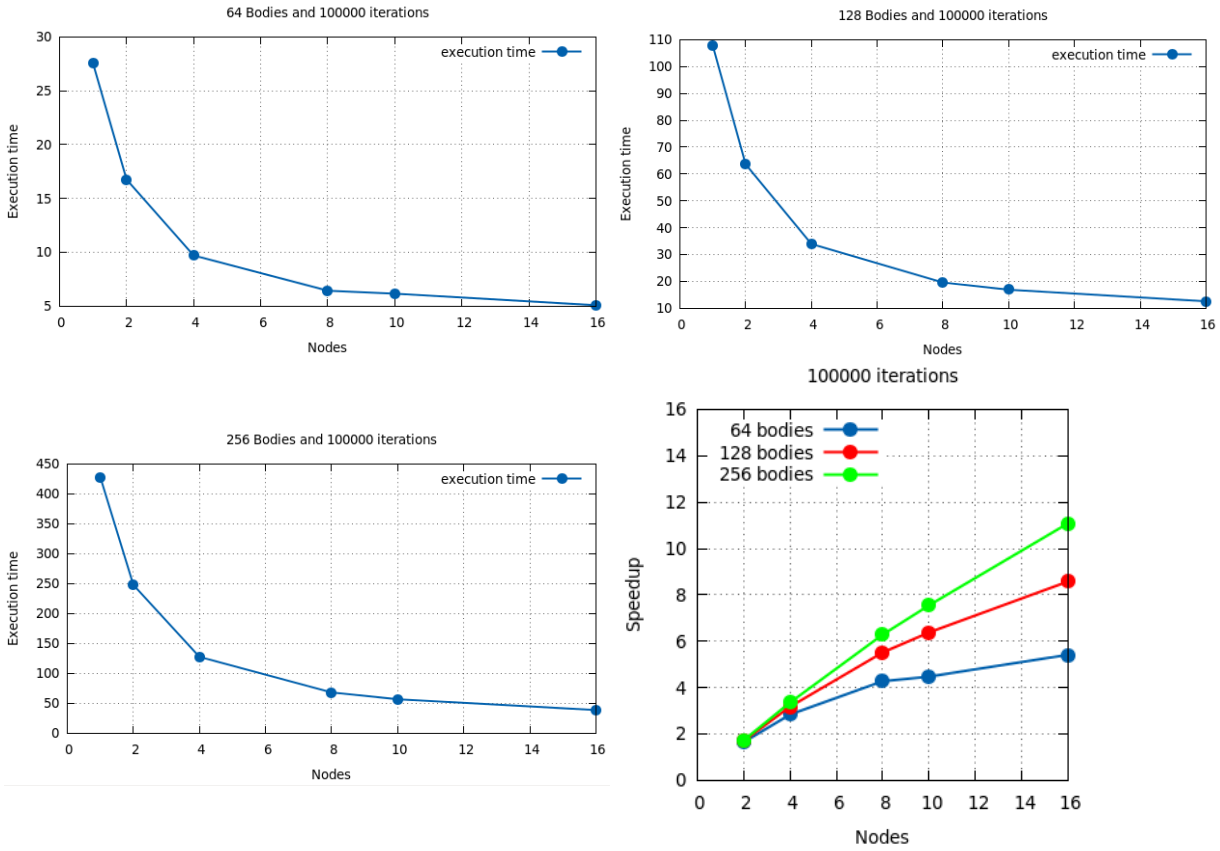


Figure 10: Execution Time and Speedups, 100000 iterations

nodes\bodies	64	128	256
1	30.414	120.347	474.974
2	16.752	63.612	247.778
4	9.706	33.939	127.529
8	6.435	19.566	68.082
10	6.155	16.916	56.687
16	5.081	12.584	38.576

Table 5: 100000 iterations - Exec. times

nodes\bodies	64	128	256
2	1.81	1.89	1.91
4	3.13	3.54	3.72
8	4.72	6.15	6.97
16	5.98	9.56	12.31

Table 6: 100000 iterations - Speedups

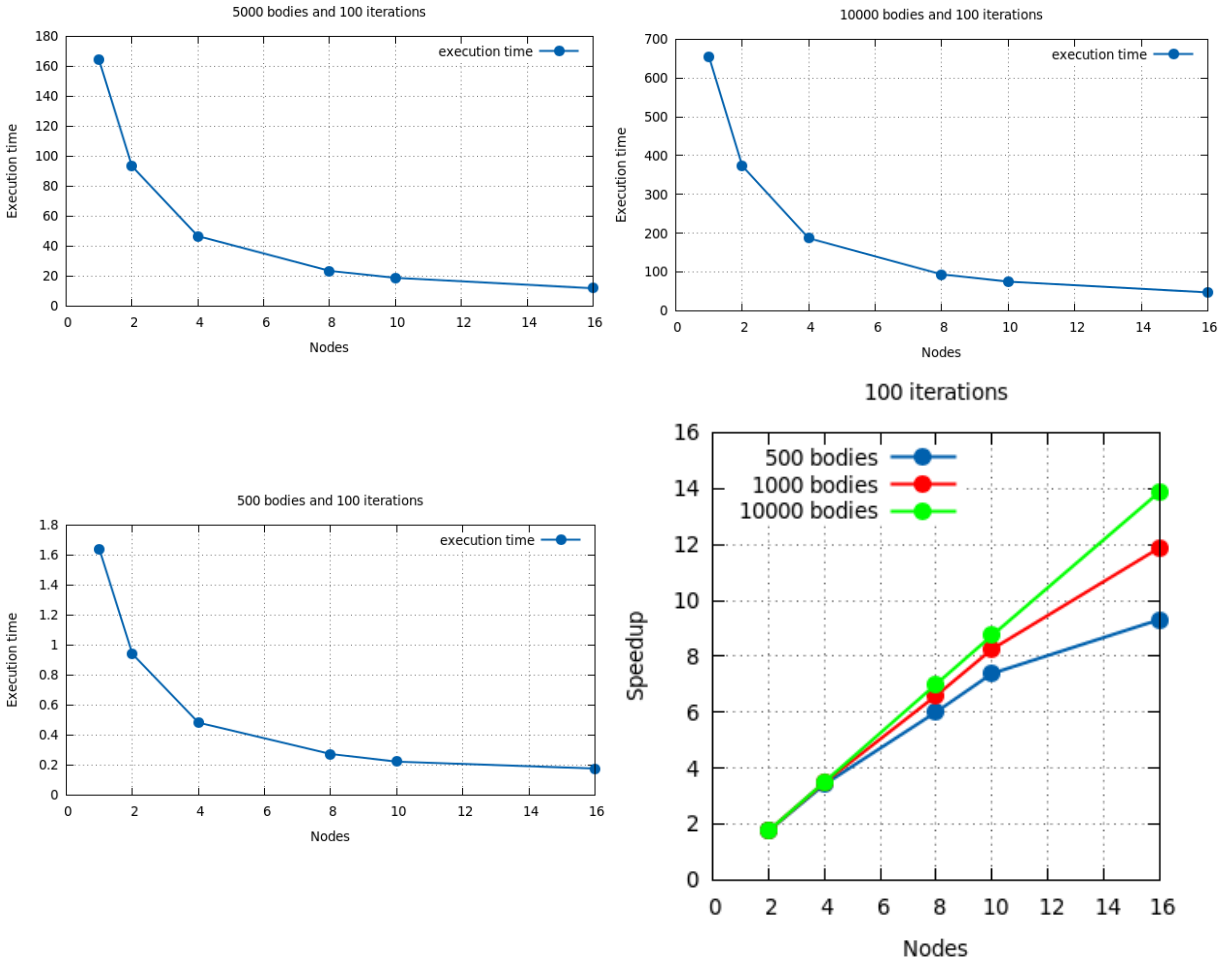


Figure 11: Execution Time and Speedups, 100 iterations

nodes\bodies	500	1000	10000
1	1.818	7.275	725.597
2	0.940	3.749	373.328
4	0.482	1.892	186.764
8	0.273	1.001	93.483
10	0.222	0.799	74.908
16	0.176	0.555	47.192

Table 7: 100 iterations - Exec. times

nodes\bodies	500	1000	10000
2	1.93	1.94	1.94
4	3.77	3.84	3.88
8	6.65	7.26	7.76
16	10.32	13.10	15.37

Table 8: 100 iterations - Speedups

As expected, if we fix the number of bodies and we change the number of iterations, we obtain the same speedup results (Figure 12 and Tables 9, 10). This is due to the fact that we have to repeat the same work but for different times.

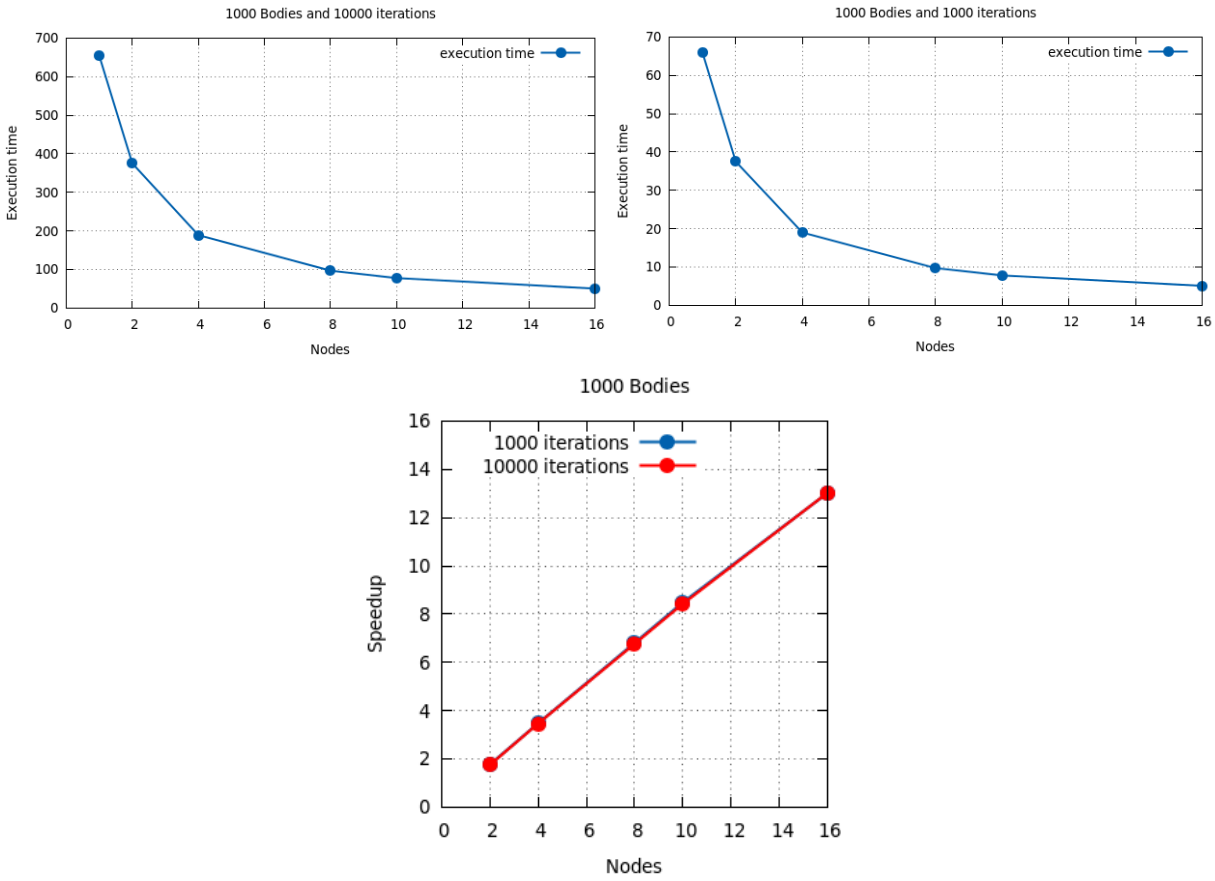


Figure 12: Execution Time and Speedups, 1000 bodies

nodes\iterations	1000	10000
1	65.991	654.008
2	37.488	374.935
4	18.989	188.824
8	9.702	96.724
10	7.778	77.597
16	5.070	50.201

Table 9: 1000 bodies - Execution times

nodes\iterations	1000	10000
2	1.76	1.74
4	3.47	3.46
8	6.8	6.76
10	8.48	8.42
16	13.01	13.02

Table 10: 1000 bodies - Speedups

All the tests were performed on the Leiden University cluster (fs1.das4.liacs.nl) and the execution times were measured using the "wall clock time". Data distribution and data gathering (as being part of the application's initialisation and de-initialisation) are not considered on the measured performance.

4 Conclusions and Personal considerations

With the realized implementation, the speedups are close to be linear for big size problems and that means that is a good implementation. This was the first time for me to use the MPI programming model and the hardest part was to understand how the MPI operations work and use them to compute various tests (e.g. MPI_Allgatherv, MPI_Scatterv). From my work I understood the powerful of this programming model and that is interesting to have more than one implementation in order to compare the results and outline any weaknesses.