MPI Assignment: N-body Simulations

Stefano Sandonà

Vrije Universiteit Amsterdam, Holland

1 The N-body problem

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

2 The sequential algorithm

Simplify structure:

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

As first thing, we have to discretize the time, so the application knows what to do at certain time steps. Than for each step, the algorithm computes all the forces between all the bodies. With the resulting forces, the new velocity and positions of all the bodies are updated.

2.1 Force computation

This step of the algorithm is the most expensive and determine its complexity. The forces that bodies exert on each other has to be calculated once per pair of bodies.

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

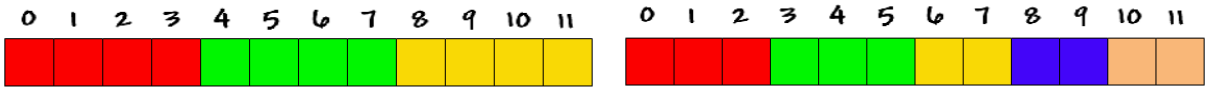
That means with N bodies, we have (N-1)+(N-2)+(N-3)+...+1 pairs, so O(N^2) force computations for every time step.

3 Parallel N-body algorithm

An efficient parallelization of this algorithm is not trivial because for every step we need the updated information from all bodies that are part of our system in order to calculate the correct velocities and positions.

3.1 Bodies distribution

One of the major problems of the parallel algorithms is the load imbalance. If we don't distribute the work fairly between the machines involved in the computation, we'll have some machines idle while others will still work. This problem affects a lot the performance of the application because the overall execution time depends on the last machine that terminate the computation. As each body has to interact with all the others, we have the same amount of work per body, so we just have to find a way to distribute equally the N bodies among all machines. Using a simple *for* construct we can calculate the chunks of bodies to assign to each computational node (number of bodies per node and the boundaries of the chunks). In Figure 1 we can see two examples of repartition.



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

Figure 1: Repartition of 12 bodies

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

3.2 The Work repartition

3.2.1 Naive repartition

This is the most critical part of the implementation, from this depends the efficiency of the whole algorithm. Using a naive approach for every step we could simply use the MPI_Allgatherv operation, to send to all other nodes our updated chunk of bodies and receive all the other chunks from other nodes. Than compute the forces that all other bodies exert on the actual chunk and update the actual chunk bodies positions and speeds.

A problem of this approach is the big amount of data transferred in each iteration (N bodiesType structures) and the fact that we are computing the same forces between some pairs of bodies in more than one machine. In fact, computing the force that the body A exert on the body B, implicitly we are computing also the force that the body B exert on the body A (force of B on A is negative of A on B). Given a situation like the one described on Figure 1a, we can see on Tables 1 and 2 the forces computed by the first two nodes. Forces between the pairs highlighted in red are calculated by both Node 0 and Node 1. For each node we calculate the forces for 38 pairs, that means 114 pairs (38 x 3) in total. The number of possible pairs is 66, so we are doing about 1.7 times the needed work.

An approximation of the computational complexity is shown on the Equation 1). The first O represents the computation of the forces between the assigned chunk's bodies and all the other bodies, while the second O represents the computation of the forces among the assigned chunk's bodies.

$$\begin{aligned} Computation &= O(\lceil\frac{N}{P}\rceil * \lceil\frac{N}{P}\rceil * (P-1)) + O(\sum_{i=1}^{\lceil\frac{N}{P}\rceil}(\lceil\frac{N}{P}\rceil - i)) \\ &= O(\lceil\frac{N^2}{P}\rceil) \end{aligned} \tag{1}$$

Table 1: Node 0 - Calculated forces

bodies[0]	0-1	0-2	0-3	0-4	0-5	0-6	0-7	0-8	0-9	0-10	0-11
bodies[1]	1-2	1-3	1-4	1-5	1-6	1-7	1-8	1-9	1-10	1-11	
bodies[2]	2-3	2-4	2-5	2-6	2-7	2-8	2-9	2-10	2-11		
bodies[3]	3-4	3-5	3-6	3-7	3-8	3-9	3-10	3-11			

Table 2: Node 1 - Calculated forces

bodies[4]	4-0	4-1	4-2	4-3	4-5	4-6	4-7	4-8	4-9	4-10	4-11
bodies[5]	5-0	5-1	5-2	5-3	5-6	5-7	5-8	5-9	5-10	5-11	
bodies[6]	6-0	6-1	6-2	6-3	6-7	6-8	6-9	6-10	6-11		
bodies[7]	7-0	7-1	7-2	7-3	7-8	7-9	7-10	7-11			

3.2.2 Optimized repartition

The aim of a good repartition is to not repeat the same work on different nodes. Following this objective the idea is, given a pair of chunks, half of the forces that two bodies from different chunks exert on each other are calculated by one node, and half by the other. The ripartition is clarify on the Figures 2, 3 and 4.

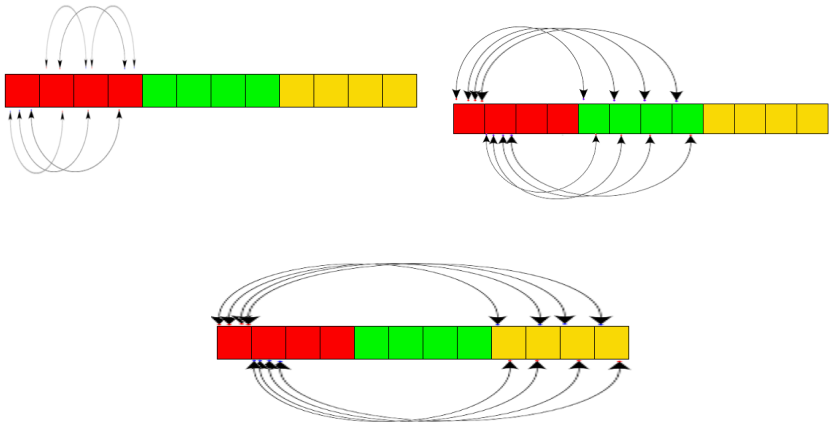


Figure 2: Forces computed by Node 0 (6+8+8=22)

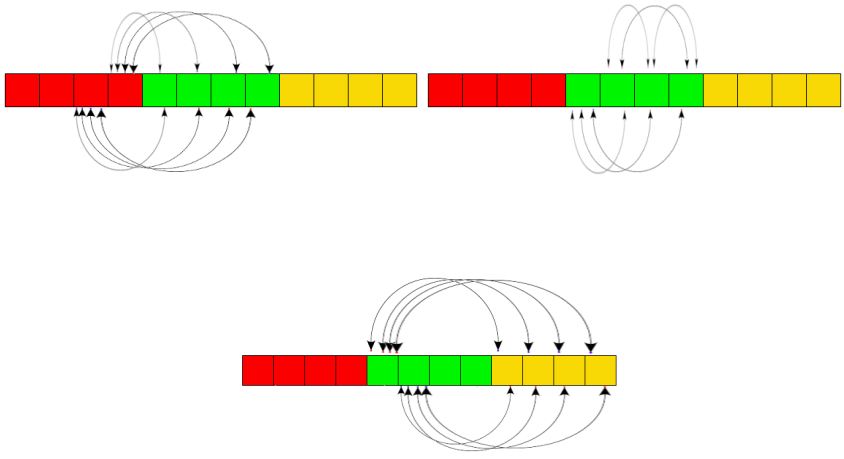


Figure 3: Forces computed by Node 1 (8+6+8=22)

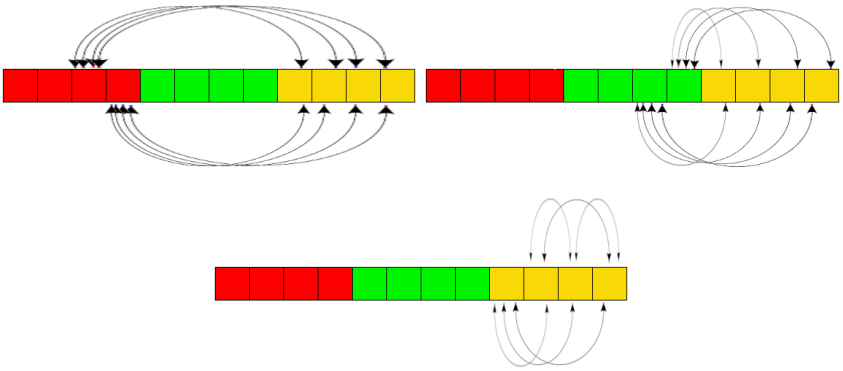


Figure 4: Forces computed by Node 2 (8+8+6=22)

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. We have 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 2. 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.

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

3.3 The final solution

3.3.1 Communication VS Computation

The communication is a common bottleneck on 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 we need the updated information (positions, velocities) of all other bodies of the system in order to correctly compute the forces of the assigned chunk. That means an exchange of data for every step. To gain performances we have to find a good compromise between the data exchange and the work to do on each node. In particular, we have to maximize the ratio *computation/communication*.

3.3.2 Approach one

Due to the fact that the force computation is the most expensive part of the algorithm and that we want to exchange as less data as possible, a good compromise is the following. At the beginning, we broadcast to all the nodes the bodies (MPI_Bcast Figure 5). Than for each step, each node calculates the forces for the assigned pairs (Figure 6), after that with an MPI_Allreduce operation, we sum all the calculated forces by all the nodes (Figure 7) and as the last thing we compute the velocities and positions for all the bodies (Figure 8). Proceeding in this way, we exchange with the other nodes only an array of N elements containing the forces calculated for the assigned pairs. The major problem with this approach, is that we calculate the positions and velocities of all the bodies in every node, that represent a duplication of work. An approximation of the whole complexity and of the total amount of data exchanged per step is presented in Equations 3 and 4.

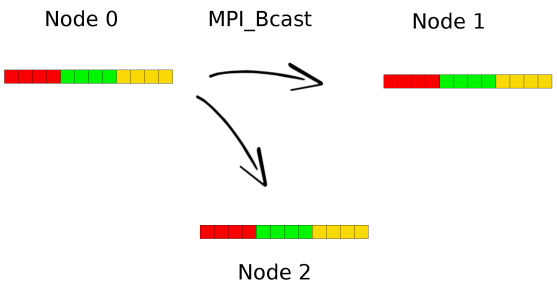


Figure 5: A - Broadcast the bodies to all the nodes

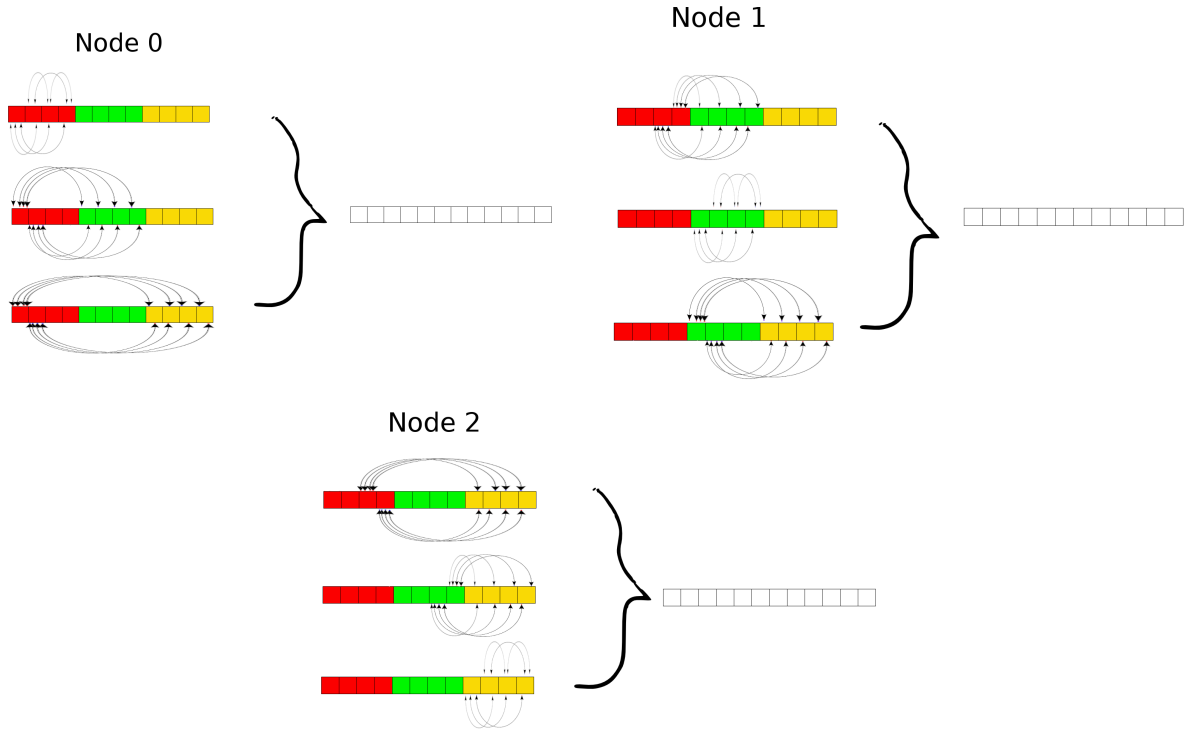


Figure 6: B - Forces computation, the results are put in a vector

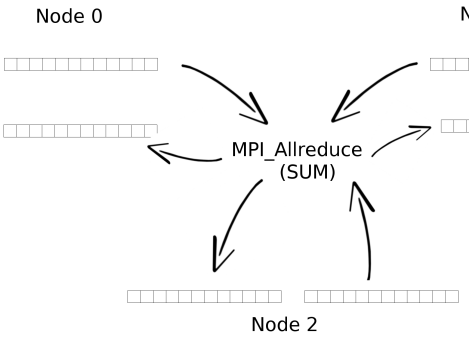


Figure 7: C- Forces reduction

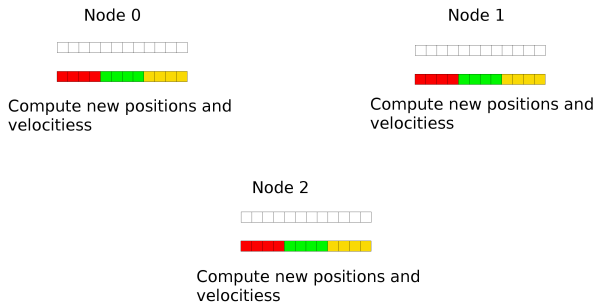


Figure 8: D - Update positions and velocities

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

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

3.3.3 Approach two

An alternative to approach one, could be to calculate in each node only the velocities and the positions of the bodies of the assigned chunk, but doing this we have to share with the other nodes also this information. We have first to scatter the bodies between the nodes (Figure 9), then for each step collect the updated bodies from other nodes (Figure 10), calculate the forces for the assigned pairs (Figure 11), after that with an MPI_Allreduce operation, we sum all the calculated forces by all the nodes (Figure 12) and as the last thing we compute the velocities and positions of the assigned bodies (Figure 13). An approximation of the whole complexity and of the total amount of data exchanged per step is presented in Equations 5 and 6.

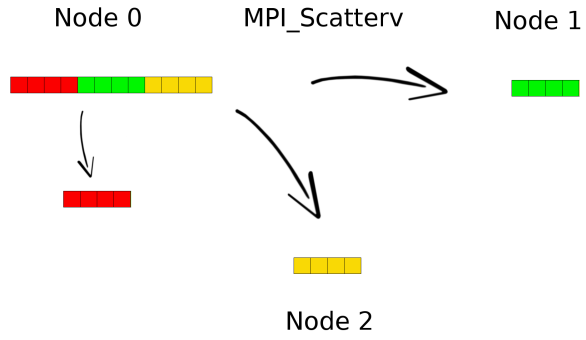


Figure 9: A - Scatter the bodies to the nodes

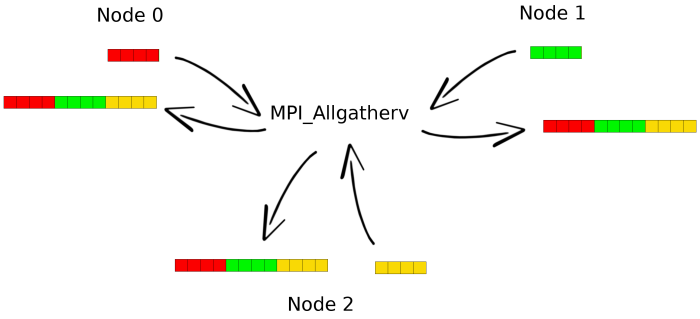


Figure 10: B - Collect updated bodies from other nodes

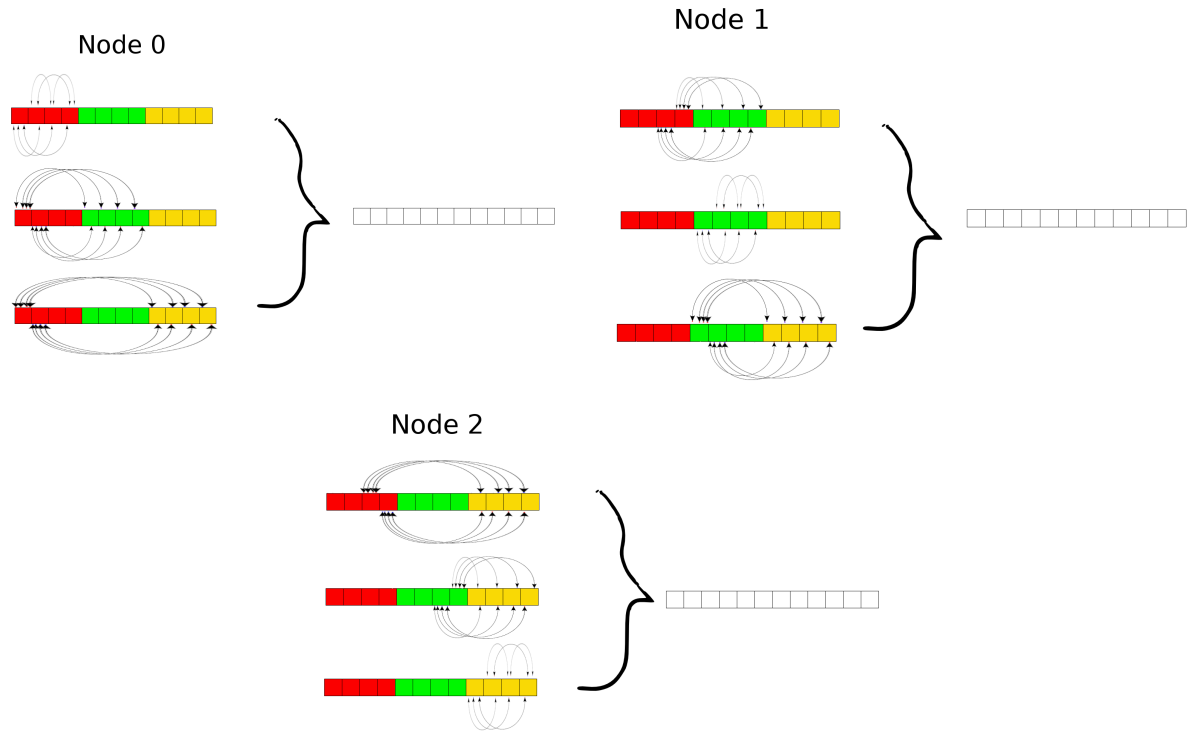


Figure 11: C - Forces computation, the results are put in a vector

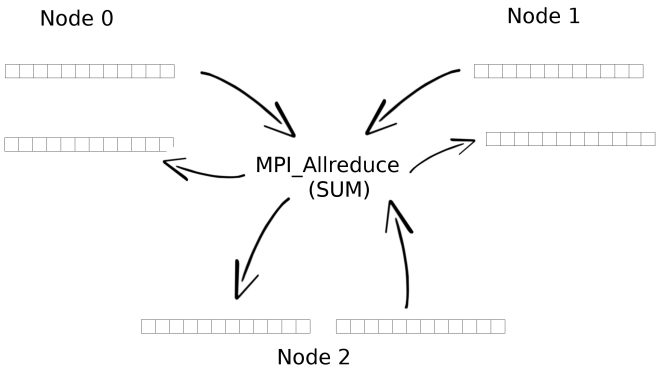


Figure 12: D - Forces reduction

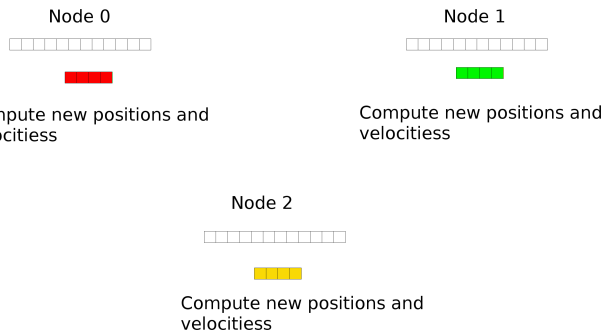


Figure 13: E - Update positions and velocities

$$
\begin{aligned}
Complexity(per step) &= O(\lceil \frac{N^2}{2P} \rceil)) + O(\lceil \frac{N}{P} \rceil) + O(\lceil \frac{N}{P} \rceil) \\
&= O(\lceil \frac{N^2}{2P} \rceil)
\end{aligned}
\tag{5}
$$

$$
Communication(per step) = O(N) * sizeof(bodyType) \tag{6}
$$

3.3.4 Solutions comparison

The decision at this point, is between computing the N velocities and positions in all the nodes or exchange more data with the other nodes. 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 3 and 4 (corresponding graphs in Figure 14) and in Tables 5 and 6 (corresponding graphs in Figure 15) 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 3: 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 4: 128 bodies and 100000 iterations - Speedups

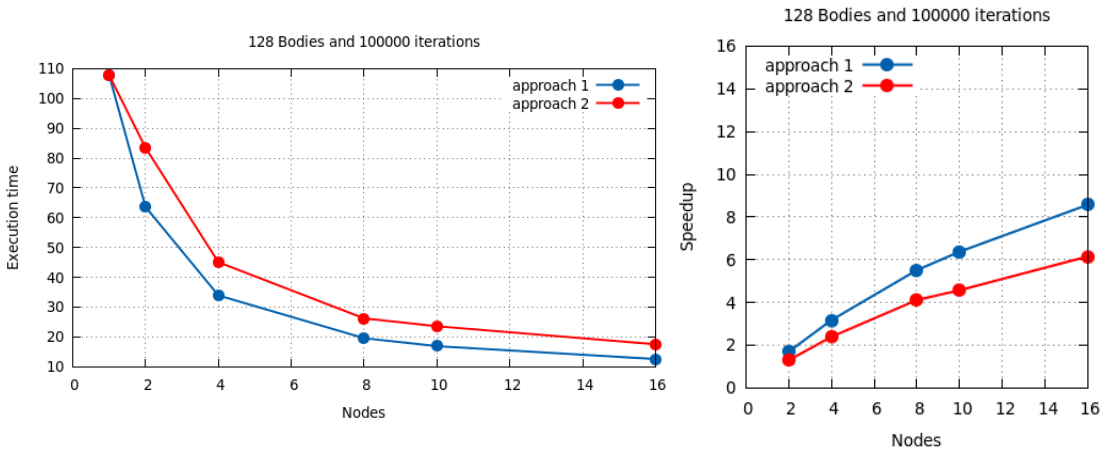


Figure 14: Execution Time and Speedups of the two approaches with 128 bodies and 100000 iterations

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 5: 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 6: 10000 bodies and 100 iterations - Speedups

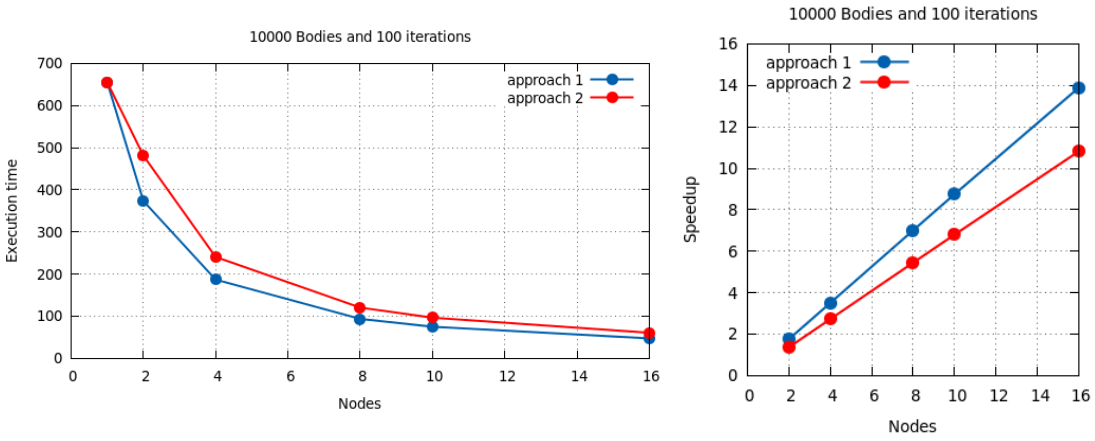


Figure 15: Execution Time and Speedups of the two approaches with 10000 bodies and 100 iterations

3.3.5 Results

In this section the most interesting results are reported. In Figure 16 and Tables 7, 8 we have an example with a big number of iterations (fixed) and different numbers of bodies (small). In Figure 17 and Tables 9, 10 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.

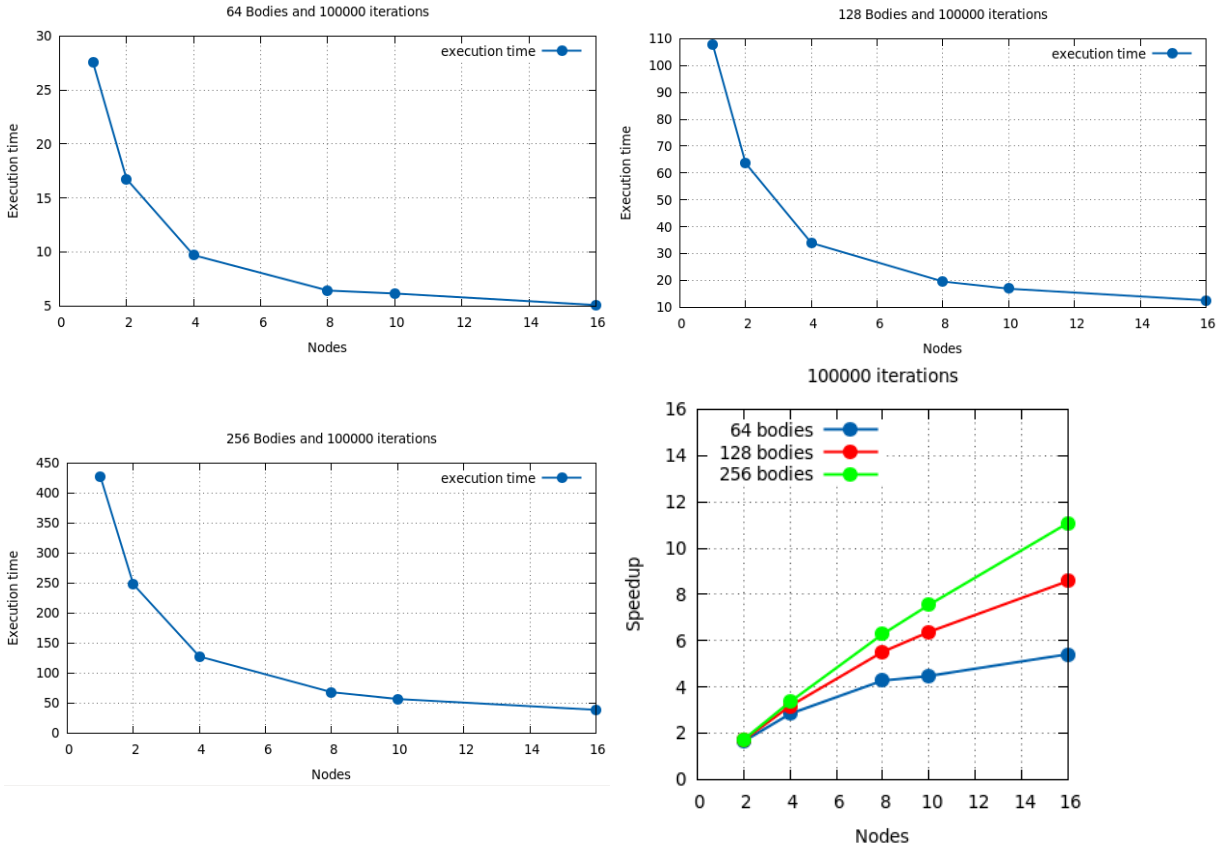


Figure 16: Execution Time and Speedups, 100000 iterations

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 7: 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 8: 100000 iterations - Speedups

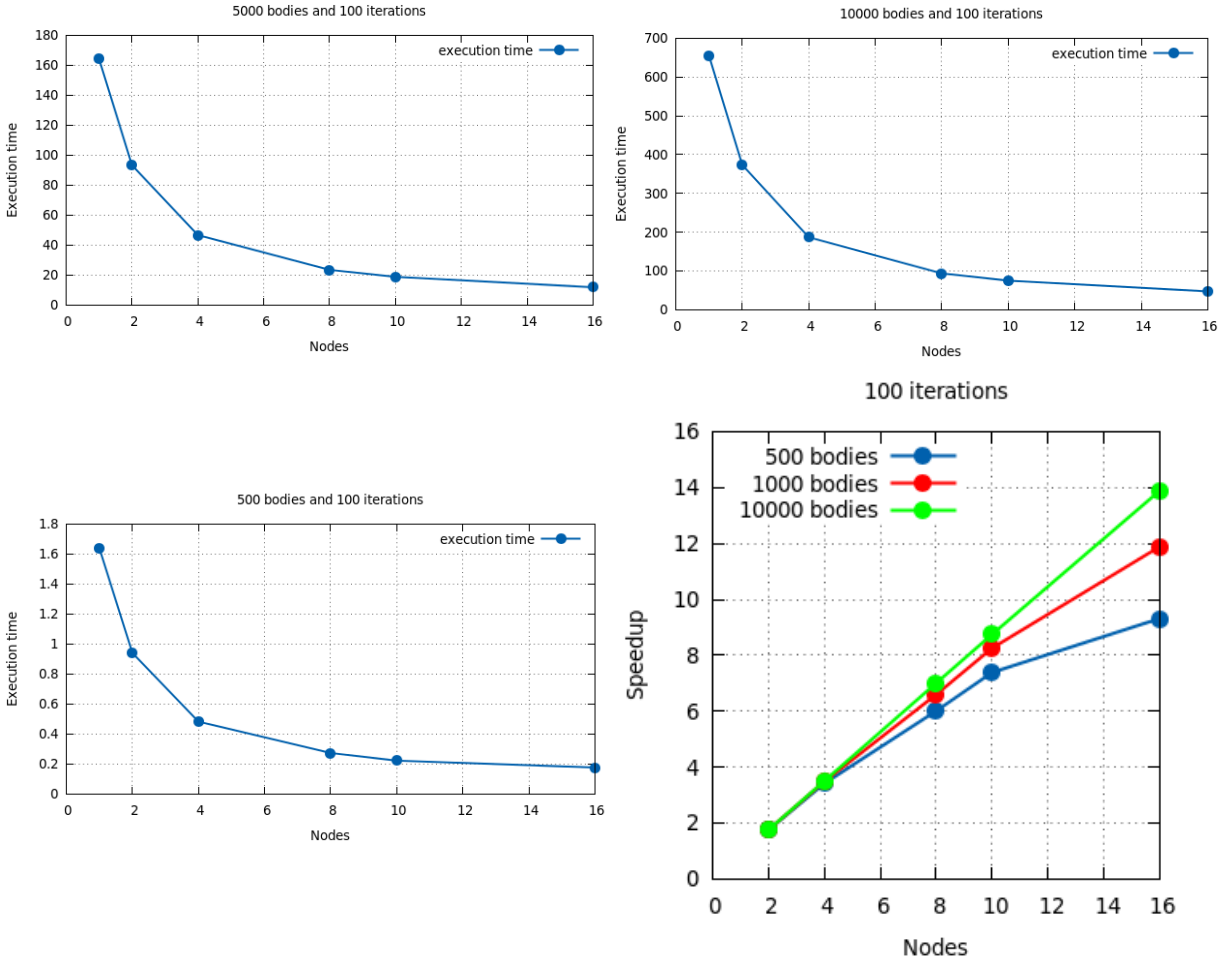


Figure 17: Execution Time and Speedups, 100 iterations

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 9: 100000 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 10: 100000 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 18 and Tables 11, 12). This is due to the fact that we have to repeat the same work but for different times.

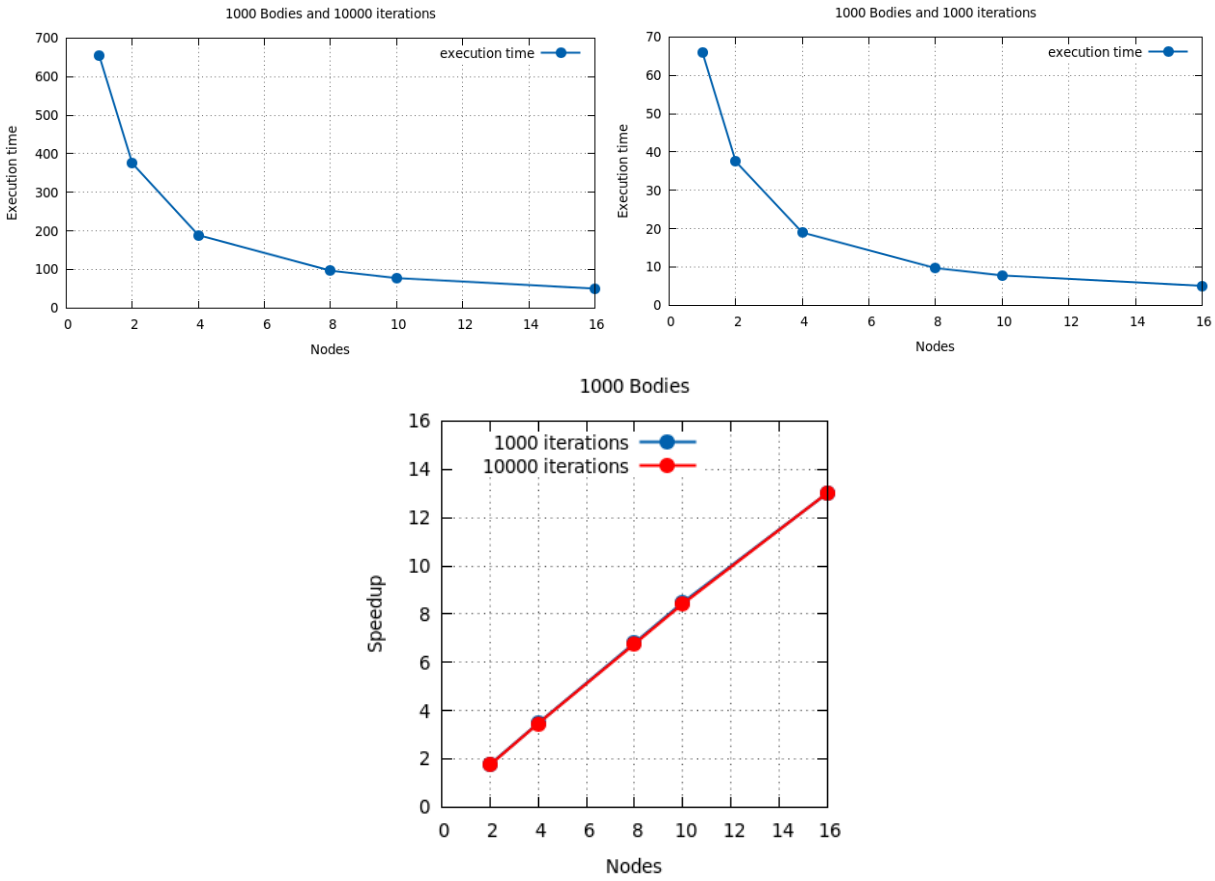


Figure 18: Execution Time and Speedups, 1000 bodies

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 11: 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 12: 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.