# Parallel Programming Practical
MPI Assignment - N-body Simulation

Laurens Verspeek
VUnetID: 2562508

February 3, 2016

## 1 Introduction

This report discusses an implementation of a parallel N-body algorithm using C++ and the MPI (Message Passing Interface) communication library and compares it with the sequential version of the N-body simumlation. The N-body problem is the problem of predicting the individual motions of a group of celestial objects interacting with each other gravitationally.
This report considers a 2D (rather than 3D) space and Newton's law of gravity. This law states that each pair of two bodies exert a certain force on each other.

The setup and design overview of the implementation is discussed in section *Implementation*. The speedup performance compared to the sequential algorithm is discussed in section *Results*.

## 2 Implementation

This report describes the parallelisation of the sequential algorithm with MPI of the N-body problem. However to understand the parallel version, the sequential algorithm is discussed shortly. After that the parallel implementation will be discussed.

### 2.1 Sequential algorithm

The N-body problem is solved by creating a discrete time, so that the algorithm can compute the forces and positions and velocoties for each new timestep for every body. The force between each pair of bodies have to be computed, so this problem has $O(N^2)$ complexity. In algorithm 1 the simplified pseudocode of the sequential algorithm is displayed.

**Algorithm 1** Basic N-body algorithm

```
1: for all timestep do
2:     Compute forces between all bodies
3:     Compute new positions and velocities
```

### 2.2 Parallel algortihm

This algorithm can be quite easily parallized by distributing the N bodies equally among all participating machines. After each step, all machines need to gather all the updated info from the other machines in order to calculate the correct velocities and positions for the next timestep. The parallel algorithm is implemented with the MPI in C++.

At first we initialize MPI with *MPI_Init* and retrieve the number of participating machines with *MPI_Comm_size* and our own identifier with *MPI_Comm_rank*. We then calculate how many bodies each machines gets (and if we have a remainder, the last

machine will do that).

Next we determine which machine is responsible for which bodies. The 'offset' is calculated by multiplying the number of bodies per machine with the machines local identifier. This can be done, because the identifiers start with 0 and gets incremented by 1 for each new machine. Each machines needs all the bodies, so the root machine (with id 0) calculates the pseudorandom start values for the bodies and broadcasts all this information to all the machines with *MPI_Bcast*.

Before we enter the main loop, we start the timer with *MPI_Wtime*. Then for each step in the main loop, all machines calculate the positions, velocities and forces for their bodies (by looking at their offset). Then the sum of the forces is send to all machines with the MPI operation *MPI_Allreduce* that used the operator *MPI_SUM*. All updated positions are send to each machines with the powerful *MPI_Allgather* function. Figure 1 shows how data is distributed after a call to *MPI_Allgather*.

When the main loop is finished, the MPI timers on each machine are stopped (differ-

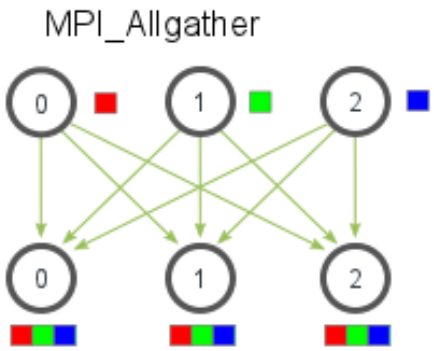


Figure 1: Visualize how data is distributed after a call to *MPI_Allgather*

ence between start time and stop time). The root machine collects all the bodies from all the machines with *MPI_Gather*. With *MPI_Reduce* and the use of the *MPI_MAX* operator, the root node gets the time of the machine that took the longest to finish. That maximum time is used in the next section to measure the performance.

# 3 Results

The sequential and the parallel algorithms were tested on the DAS-4 system. All the bodies are generated with the same seed, so we can see if the output of the sequential algorithm is the same as the parallel algorithm. The parallel algorithm was tested with 1, 2, 4, 8 and 16 nodes. As said in the previous section, the MPI timers were used to measure the execution time of the main loop. The benchmarks are run with different number of bodies and different number of iterations:

- 100.000 iterations (more communication)
  - 64 bodies (less computation)
  - 128 bodies
  - 256 bodies
- 100 iterations (less communication)
  - 100 bodies
  - 1000 bodies
  - 10000 bodies (more computation)

| # of bodies | Sequential | 1 node | 2 nodes | 4 nodes | 8 nodes | 16 nodes |
|---|---|---|---|---|---|---|
| **64** | 30.313s | 30.584s | 21.267s | 12.118s | 7.974s | 6.716s |
| **128** | 179.990s | 180.425s | 91.669s | 47.822s | 23.655s | 16.152s |
| **256** | 654.156s | 579.845s | 315.933s | 183.167s | 85.151s | 50.920s |

Table 1: Run times of the N-body problem with different number of bodies and 100.000 iterations

| # of bodies | Sequential | 1 node | 2 nodes | 4 nodes | 8 nodes | 16 nodes |
|---|---|---|---|---|---|---|
| **100** | 0.085s | 0.099s | 0.051s | 0.0403s | 0.041s | 0.068s |
| **1000** | 8.310s | 8.505s | 4.051s | 2.502s | 1.413s | 0.770s |
| **10000** | 749.044s | 750.585s | 382.786s | 193.160s | 130.034s | 62.552s |

Table 2: Run times of the N-body problem with different number of bodies and 100 iterations

All the results can be found in tables 2 and 1. Note that all the results in that table are the average of 3 measurements.

To make these results more visual, the speedups are displayed in figures 3 and 2. As we can see in the speedup graphs, the more bodies there are, the better the speedup is. This is because the computation increases exponentially (as this is a $O(N^2)$ complexity problem) when there are more bodies. There needs to be communication for each iteration between all the machines, because the forces and positions needs to be available on all nodes for each new iteration as explained in the previous section. Because of this, the speedup is not completely linear. The speedup for 100 bodies and 100 iterations is the worst, because there communication overhead is bigger than the computations that needs to be done. However, the speedup graphs also shows that algorithm is very scalable, because if we increase the number of bodies, the speedup gets closer to linear.

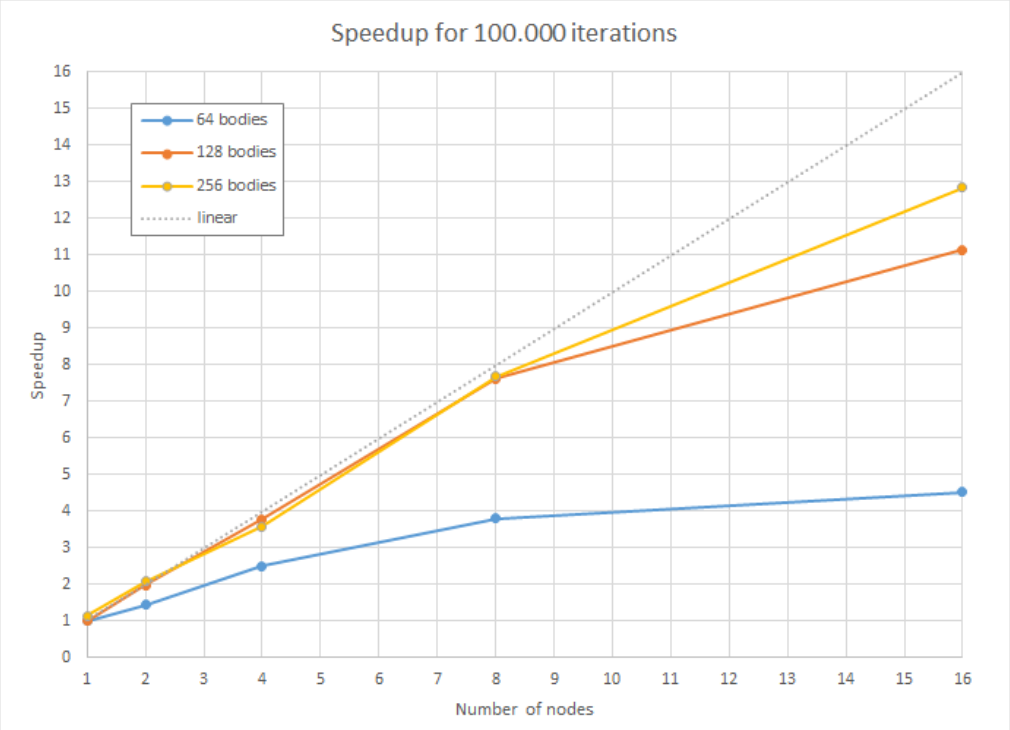


Figure 2: Speedup of the parallel implementation with different number of nodes compared to the sequential algorithm for 100.000 iterations

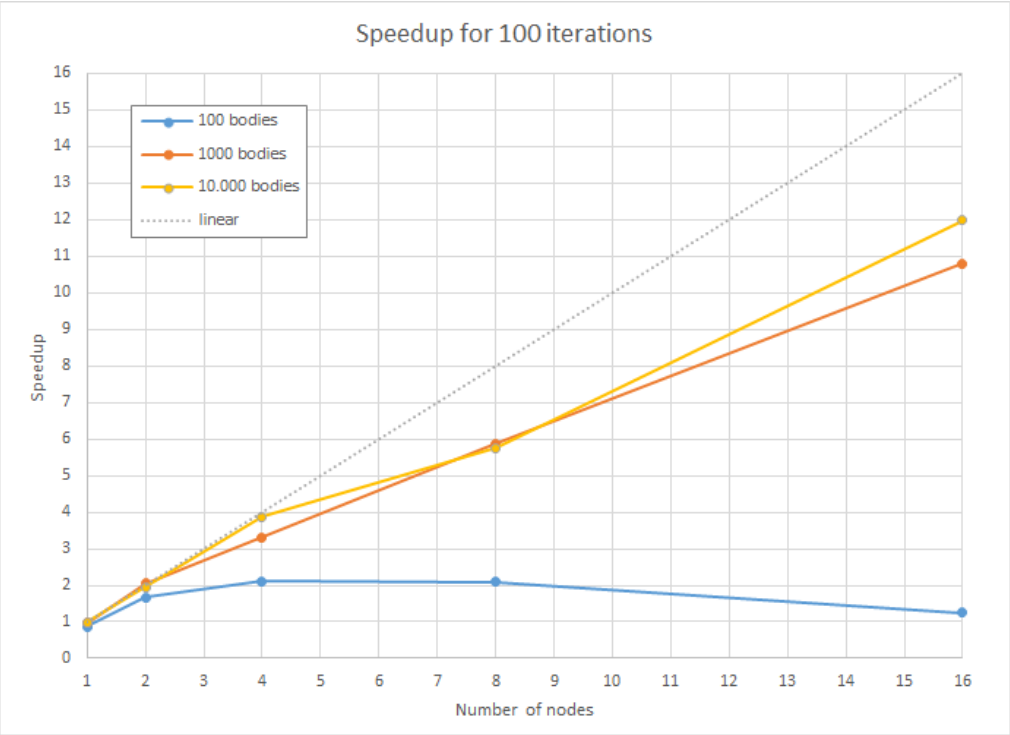


Figure 3: Speedup of the parallel implementation with different number of nodes compared to the sequential algorithm for 100 iterations