N-Body parallelization with MPI
DRAFT

Nicholas Molyneaux

January 9, 2016

1 N-Body

The n-body simulations are relevant for many fields of science, from planet and galaxy interactions all the way down to molecular dynamics. The computational challenge comes from the fact that all bodies have some effect on all the others. The effect comes from the gravitational pull each body has on the others, the force one body of mass m_1 has on another of mass m_2 depends on the distance between each body r and the gravitational constant G. The formula is the following:

$$\|\boldsymbol{F}\| = G\frac{m_1 m_2}{r^2}$$

where $G = 6.674 \times 10^{-11}\,\mathrm{Nm^2kg^{-2}}$. The direction of the force is given by the center of masses of the objects in the following way, where p_1 and p_2 are the positions of both masses.

$$F_x = \|\boldsymbol{F}\|\frac{p_{2,x} - p_{1,x}}{r}$$
$$F_y = \|\boldsymbol{F}\|\frac{p_{2,y} - p_{1,y}}{r}$$

1.1 Complexity

A classical brute-force approach to this problem has a complexity of $\boldsymbol{\mathcal{O}(n^2)}$, where n is the number of bodies in the system. This will rapidly become unbearable, since for 1000 bodies the calculations take the order of one second on a laptop. Since this problem is time dependant, the calculations must be performed at each time step, hence the importance of having a parallel application to perform the body interactions is obvious. The copmutation of the interactions at each step is straight forward, using two loops the force on each body induced by the other bodies is calculated.
One well-known solution to this problem is the Barnes-Hut algorithm. This method calculates the forces acting between one body, and a group of bodies represented by the average of the group's masses and positions. With this algorithm, the complexity is reduced to $\boldsymbol{\mathcal{O}(nlogn)}$. In this case, the world (i.e. collection of bodies) is stored in a structure called "quad-tree", where each node stores the average of the bodie's positions and masses. At the leaves there is either a body if it fits into the quadrant, or nothing if no body is present. To calculate the interactions between all bodies, we traverse all bodies, and based on a distance criteria we either calculate the interaction with the exact body or the interaction with an "averaged body".

1.2 Computations vs communications

For a simple brut-force approach, the computations follow $\mathcal{O}(n^2)$ at each time step. To guarantee correct results, the acceleration, velocity and position of all bodies must be updated in

a synchronized manner, hence one cannot move forward in time until all forces have been calculated. Once the new positions are calculated for all bodies on each node, these updated positions must be broadcast to all other nodes so they can continue the compuations at the next time step. Therefore the communication complexity is $\mathcal{O}(n)$, since the positions of the bodies must be passed around between nodes. The ratio of computations to communications can be approximated as $\mathcal{O}(n)$ for the parallel brute-force approach.
For Barnes-Hut's algorithm, this ratio remains the same. The choice is made to rebuild the quad-tree from scratch at each time iteration, this makes the implementation much easier and the time taken is not significant compared to the computation time. With this in mind, the communications are the same as for the brute-force approach, since at each time step the positions and velocities of each body must be passed to all compute nodes again. Therefore the calculations are in $\mathcal{O}(nlogn)$ and the communications are $\mathcal{O}(n)$.

1.3 Amdahl's Law

Since the goal is the analyse the parallel aspects of the problem, the time taken to write the results to a file at each time step are not taken into account. This part is significant and limits the speedup values to very poor ones. In both cases, the initial data is stored in a file which must be read to get the initial positions and velocities.

Brute-Force For the brut force approach, since the data is stored in arrays, very little preprocessing is required. The fraction of non-parallelizable code is close to nothing, as the initial time during which the data is loaded from a file is neglected. Table 1 contains the values required to use Amdahl's law for predicting the speedup. Since there is no calculation performed serially, only the communication time is relevant. In Figure 1, the speedups are shown for two different problem sizes, but it is apparent that such large speedups for large numbers of nodes is not feasible.

Quad-Tree For a quad-tree method, the serial part is still only reading the initial data. Since each node builds its own quad-tree, the serial part remains only reading the initial data. Hence the initial communication will be the distribution of the initial positions and velocities. The measurements which are performed for the brute-force approach are still valid, as the difference happens during the time iterations. Except that now the computation time is significantly faster, hence the fraction of serial code is larger. With this regard, the optmistic upperbound will not be as high as previously since the computations are in $\mathcal{O}(nlogn)$ (and not $\mathcal{O}(n^2)$) , but the communications are the same. Here the limitation of Amdahl's law is evident. Since each compute-node builds the tree, this time is not counted in Amdahl's law but it is the limiting factor when the number of compute-nodes becomes large.

Algorithm	# bodies	Serial total time [s]	Serial cal. [s]	Initial comm. [s]	Fraction
Brute-force	10^5	450	-	0.006	$1.33 \cdot 10^{-5}$
	10^6	45'000	-	0.06	$1.33 \cdot 10^{-6}$
Barnes-Hut	10^5	8.5	-	0.006	$7.06 \cdot 10^{-4}$
	10^6	100	-	0.06	$6.00 \cdot 10^{-4}$

Table 1: Measurements for Amdahl's estimation of the speedups. Since the time taken to read the initial data from a file is not considered, there is not any serial calculation. Only the time taken for the initial communication is relevant.

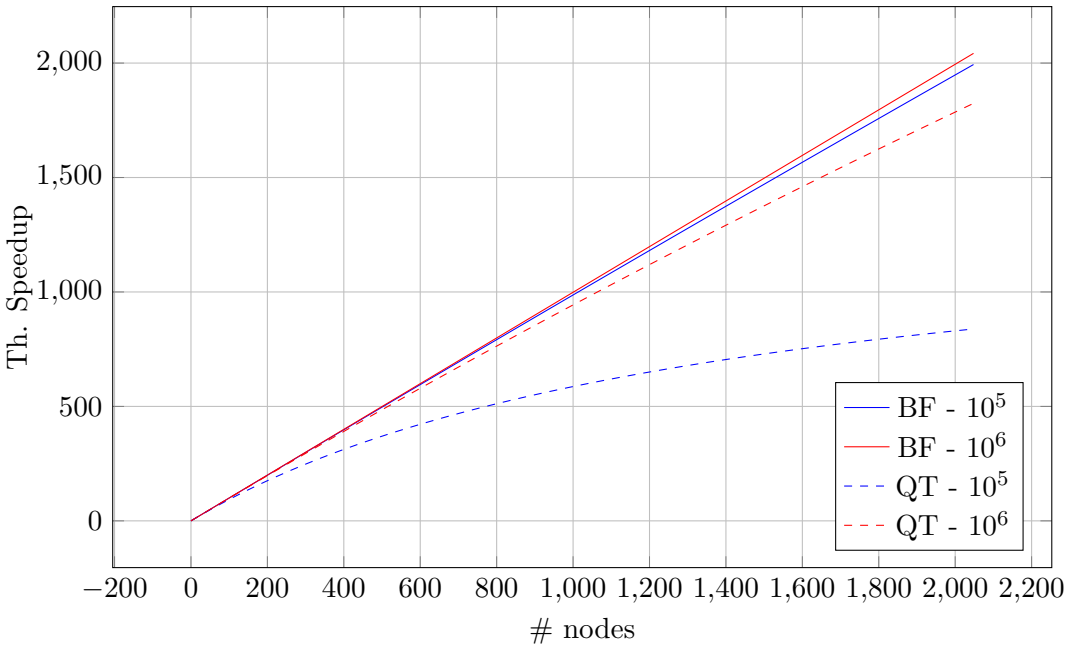
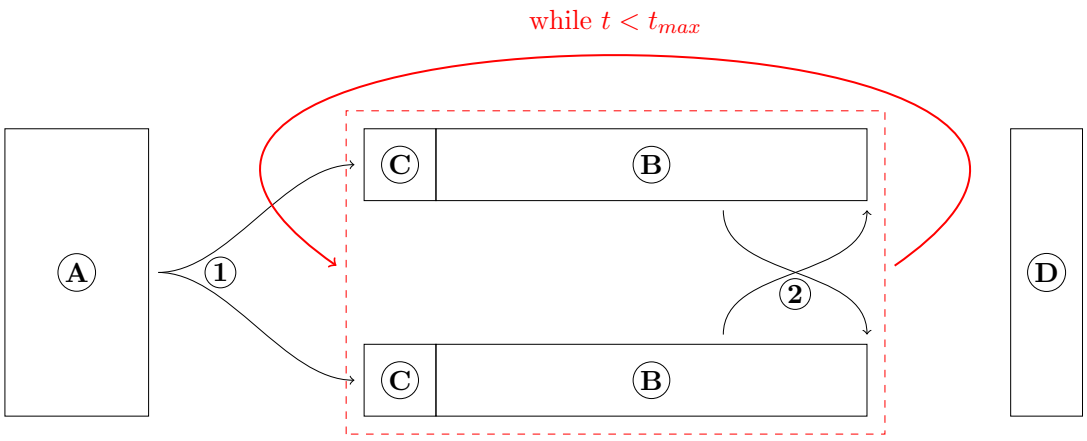


Figure 1: Amdahl's estimation for the upper bound of the speedup, for both the brut-force approach and the quad-tree method. These are estimations based on some simple timings on a serial implementation. It is already apparent that these values are too high. The fact that, with the Barnes-Hut algorithm the speedup is still quasi-linear using 2'000 nodes emphasizes that these values are not realistic.

2 Parallelization strategy

The choice for MPI was justified by the need for industry and the universal aspect of the message passing interface paradigm. For the n-body problem, the messages to be passed are mainly the updated positions and velocities of the bodies which were calculated on the other nodes. An efficient way of storing the data must be conceived to minimize the transfer times and allow the Barnes-Hut algorithm to run. As mentionned previously, two strategies are implemented. A brute-force approach and the Barnes-Hut algorithm.
The flow graps and timing diagrams are very similar for both appoaches. Once the initial data has been loaded by the main node, this data is sent to all the workers for them to perform the calculations. As Figure 2 shows, the data is initially sent using MPI's broadcast and scatter functionalities. Both solutions contain the computation step, but the Barnes-hut algorithm has an extra one, building the quad-tree. At the end of each time step, the new positions and velocities are synchronized across all nodes.



(A) - Load data, splits the data
(B) - Computes the displacement for bodies on node
(C) - Only QT: build quad-tree
(D) - Finalization
(1) - Sends the initial data to all nodes using MPI_Bcast and MPI_Scatter
(2) - Sends updated positions to all nodes using MPI_Allgather

Figure 2: Flowchart for the brut-force parallel implementation. Once the initial data is sent to all nodes, they update the positions of all bodies which are assigned to them, then send to all of the other nodes the updates positions. The step where the quad-tree is built (C) is only present for Barnes-Hut algorithm, in the brute-force approach this step does no exist.

2.1 Load-balancing

As the brute-force strategy is load-balanced by construction, nothing needs to be done to keep each node doing the same work. On the other hand, the quad-tree approach needs to be carefully load-balanced to keep each noe doing the same amout of work. Since the quad-tree stores the information based on spacial coordinates, simply splitting the tree at the same depth by distributing to each node a branch is not acceptable. By doing this, the case would often occur that the most of the bodies are found in one quadrant, whilst the other quadrants are empty.
To solve this problem, the tree is distributed by performing a tree walk. Starting from the root of the tree, following a depth-first search, each branch tries to be assigned to a node. For this to happen, the compute-nodes can be imaged as bins with a capacity. As the tree is explored, the branch are either assigned to a compute-node if the capacity allows it, or the branch is split into its sub-branches and the process is started again. The code shown in Figure 3 presents this idea.

```
void Quadtree::assignNode(int *bodies_per_node,
                          std::vector<std::vector<Node*> > &node_assignment,
                          Node &node)
{
    max_bodies_per_node = int(1.01 * root.nb_bodies/(nb_proc));

    if (node.nb_bodies > max_bodies_per_node && !node.is_leaf)
    {
        assignNode(bodies_per_node, node_assignment, *node.tr);
        assignNode(bodies_per_node, node_assignment, *node.tl);
        assignNode(bodies_per_node, node_assignment, *node.bl);
        assignNode(bodies_per_node, node_assignment, *node.br);
    }
    else if (!node.is_leaf)
    {
        int i = 0;
        while( ( (node.nb_bodies + *(bodies_per_node+i)) > max_bodies_per_node) && i < (nb_proc) )
            i++;
        if (i == nb_proc)
        {
            assignNode(bodies_per_node, node_assignment, *node.tr);
            assignNode(bodies_per_node, node_assignment, *node.tl);
            assignNode(bodies_per_node, node_assignment, *node.bl);
            assignNode(bodies_per_node, node_assignment, *node.br);
        }
        else
        {
            node_assignment[i].push_back(&node);
            *(bodies_per_node+i) += node.nb_bodies;
        }
    }
    else if(node.is_leaf && node.contains_body)
    {
        node_assignment[nb_proc-1].push_back(&node);
        *(bodies_per_node+nb_proc-1) += 1;;
    }
}
```

Figure 3: Load balancing code. This is a tree-walk and each branch is either assigned to a node, or split into the sub-branches and then the process is repeated. This is a recursive function which allows the tree-walk to be performed easily.

2.2 Theoretical speedup

The timing diagram (Figure 4) is also nearly the same for both approaches. The only significant differences concerns the presence/absence of the quad-tree construction. The other differences which cannot be observed on the diagram, is the relative length of each step. For the brute-force approach, the computation step is multiple longer than any other step, whereas for the quad-tree approach the difference in duration is not as large. The critical path is show through the diagram as the black arrow. Since the synchronization between nodes is a blocking process, after each computation stage each node must wait for all the other to finish there calculations, this also emphasizes how import the load balancing is (see section 2.1). The timing diagram assumes all nodes are load balanced, the critical path after the initial calculation step can be any one; in practice, it will be the last node which finishes its own calculations.
From Figure 4, the theoretical speedup can be calculated. Since each node builds the same tree, this step can never be accelerated by parallelizing the code, as well as loading the data from a file. The communications must also be taken into account when estimating the theoretical speedup. Hence only the computation step can be accelerated, until it becomes insignificant compared to

the other steps. The theoretical speedup can hence be derived knowing the following quantities:

- t_{comm} as the time to send the positions, velocities and masses from master to all nodes
- $t_{building}$ as the time to build the quad-tree and the time to calculate the body assignment (see load balancing at section 2.1)
- t_{comp} as the time to compute forces and update positions for all bodies on one node
- t_{sync} as the time to synchronize positions after one time step (this should be similar to t_{comm} except that the function is an MPI_Allgatherv instead of a MPI_Bcast hence the times will be different)

Based on the measurements which are shown in Tables 5 and 5, a constant time is assumed for both t_{comm} and t_{sync}. For a few nodes, the computation time is significantly larger than the communication time, where as for large numbers of nodes the times become closer to constants. For the timing diagram drawn in Figure 4, hence 4 nodes and only one time step, the theoretical speedup formula is as follows:

$$t_{serial} = t_{building} + t_{comp} \tag{1}$$

$$t_{par} = t_{comm} + t_{building} + \frac{t_{comp}}{4} + t_{sync} \tag{2}$$

$$S_{th} = \frac{t_{serial}}{t_{par}} = \frac{t_{building} + t_{comp}}{t_{comm} + t_{building} + \frac{t_{comp}}{4} + t_{sync}}$$

And this can be generalized to k nodes:

$$S_{th,k} = \frac{t_{building} + t_{comp}}{t_{comm} + t_{building} + \frac{t_{comp}}{k} + t_{sync}}$$

This equation is valid under the assumption that the computation and quad-tree building times are significantly larger than the communication times, and that the nodes are perfectly load balanced. For the brute-force approach the same methodology is applied, except that the building time is not present (one can simply set $t_{building} = 0$ in the formula for $S_{th,k}$).
The values which are used for the calculation of the theoretical speedups are based on Tables 3, 4 and 5. A summary of these values is given in Table 2.

Algorithm	# bodies	t_{serial}	t_{comm}	$t_{building}$	t_{comp}	t_{sync}
Brute-force	10^5	453.94	0.0035	-	453.81	0.0015
	10^6	44'875	0.035	-	44'874	0.025
Barnes-Hut	10^5	8.24	0.0038	0.2	8.23	0.003
	10^6	100	0.045	2.7	97.8	0.03

Table 2: Values used for the calculation of the theoretical speedups.

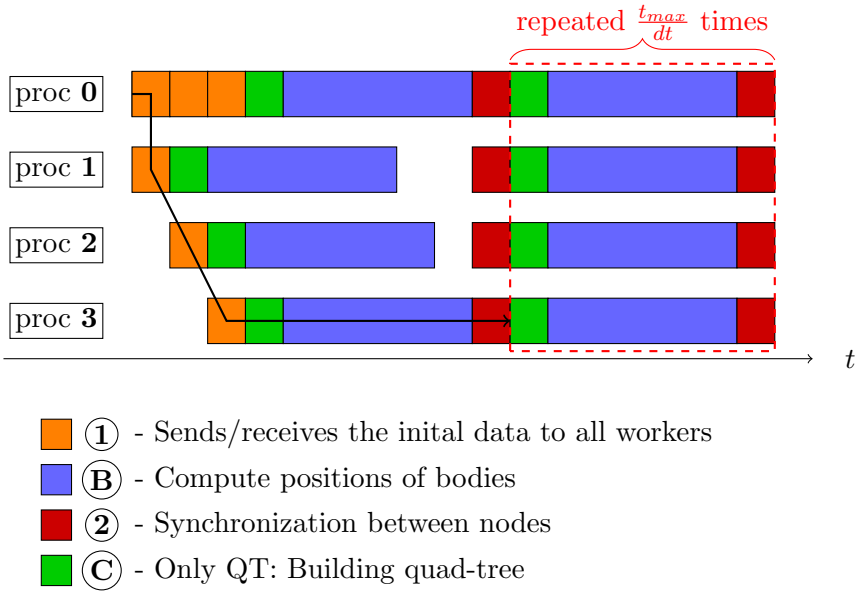


Figure 4: Timing diagram for both solutions. The block of computations is repeated until the final time is reached. The circled characters next to the coloured boxes refer to Figure 2, to emphasize that they are the same steps. The green step (C) is only present for Barnes-Hut algorithm, in the brute-force approach this step does no exist.

3 Performance discussion

All the calculations and measurements where performed on Deneb. In order to observe the effects of distributed memory nodes (and not shared memory nodes), the computations where performed using 1 process per node. The CPU which are used are either: 2.6GHz with 64GB of DDR3 RAM, or 2.5 GHz with 64GB of DDR4 RAM. The assignment to each node is unknown, hence the computations can be done on either of the two different CPUs, or a combination of both CPU types for multiple nodes. The speedups are all calculated based on the time required to compute one time iteration for the N-body problem, hence it is the time taken to finish the first synchronization step shown in Figure 4.

3.1 Brute-force performance

As the brute-force approach is very expensive in computations, the speeds are very good. In Figure 5 the theoretical speedup is shown along with the measured ones. Up to 64 nodes, the theoretical speedup is still linear, and the measured speedup follows this path closely. When the number of nodes gets larger than 20, there is slight jump downwards in the speedup values for the simulation with 10^6 bodies. This comes from the different computation times, observable in Table 3. Since two different CPUs can be used, with different clock rates and RAM access speeds, the same computation will take slightly different times. This is not observable for the simulation with fewer bodies, but becomes significant for large numbers of bodies. In Figure 6 the parallel compute times normalized by the serial compute times are shown. Two groups of compute times stand out, most certainly based on the two different CPUs.

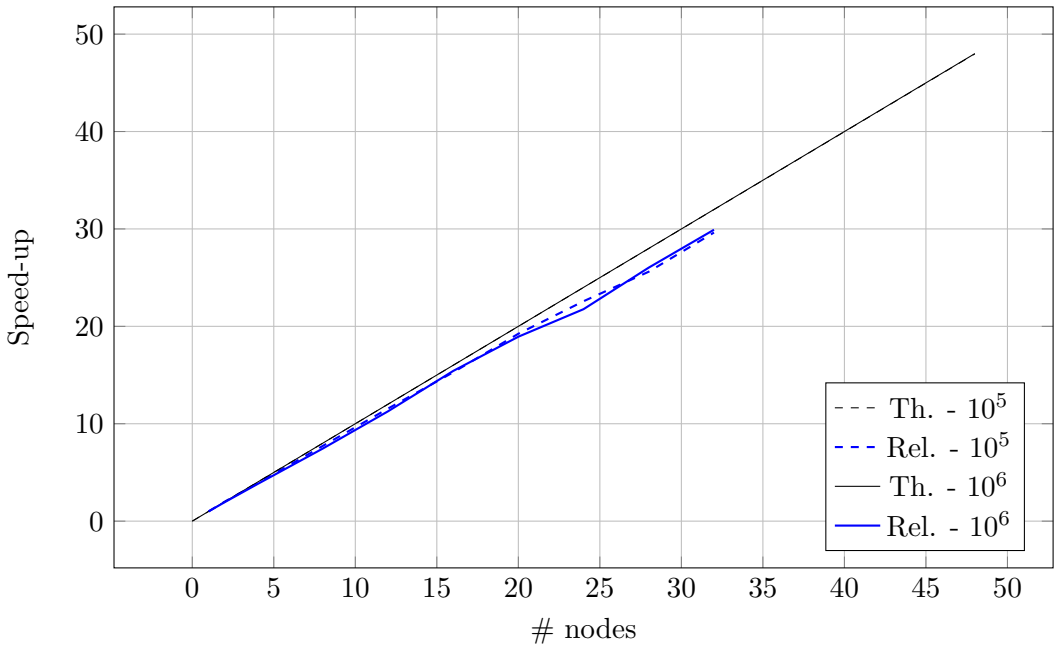


Figure 5: Speedup chart for the brute-force approach.

# nodes	1	2	4	8	12	16	20	24	28	32
init. Bcast [s]	0.008	0.022	0.028	0.027	0.0367	0.028	0.034	0.034	0.03	0.059
comput. [s]	44875	22945	11784	5626	2982	2826	1795	1490	1619	1117
	-	-	-	6026	3539.5	-	2359	1960	-	1480
sync. [s]	0.0032	0.0041	0.0049	0.0056	0.0072	0.0062	0.0083	0.0085	0.0076	0.0079

Table 3: Execution times for the brute-force approach, with 10^6 bodies. The second line of computation times is for the simulations where two distinct times are observed, see Figure 6.

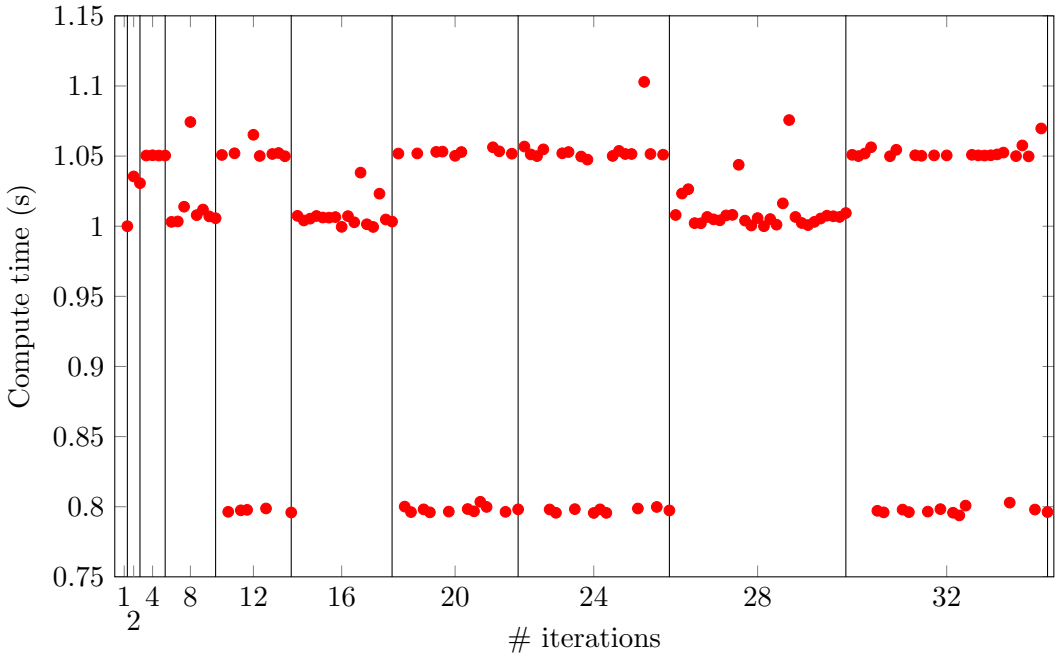


Figure 6: Compute times for the brute-force approach using 10^6 bodies, normalized by the number of bodies and number of nodes. There are two cases, either the compute times all have approximately the same value, or there are two groups of points. The different CPUs are suspected to be the origin for this differentiation.

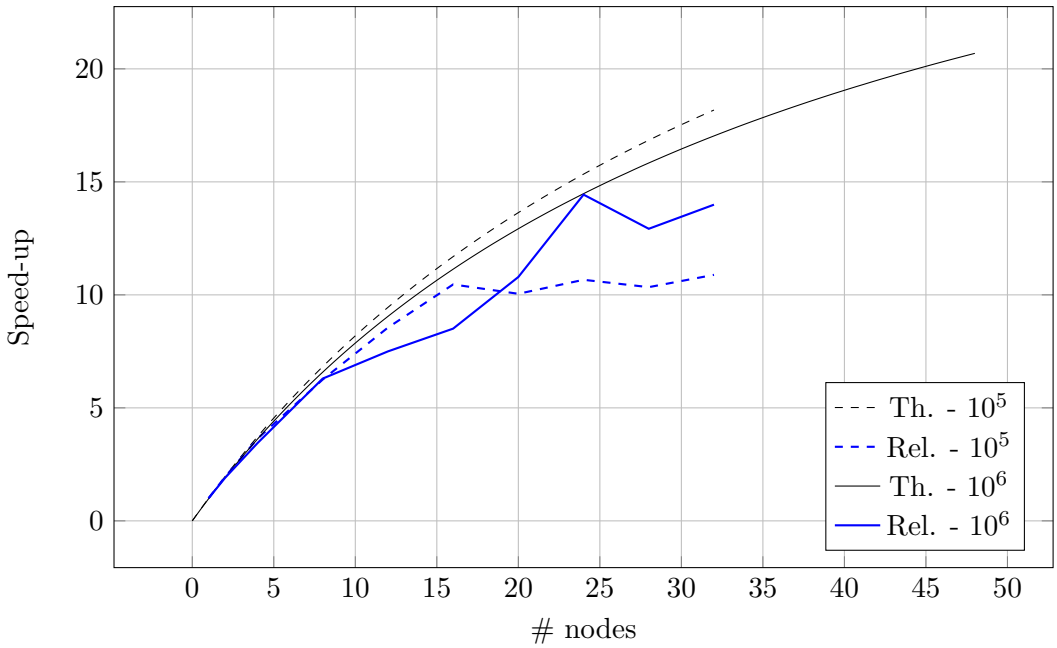


Figure 7: Speedup chart for the quad-tree approach.

# nodes	1	2	4	8	12	16	20	24	28	32
init. Bcast [s]	$1.9 \cdot 10^{-6}$	0.0036	0.0048	0.0031	0.0039	0.0036	0.0037	0.0059	0.0066	0.0077
qt build [s]	0.202	0.201	0.202	0.205	0.204	0.203	0.207	0.201	0.200	0.261
comput. [s]	8.24	4.12	2.06	1.03	0.68	0.517	0.420	0.315	0.262	0.310
	-	±0.0055	±0.008	±0.07	±0.04	±0.04	±0.04	±0.03	±0.03	±0.04
sync. [s]	$1.4 \cdot 10^{-3}$	0.0017	0.0025	0.0014	0.0026	0.0036	0.0034	0.0039	0.0041	0.0036

Table 4: For 10^5 bodies, measurements of each operation on Deneb using the Barnes-Hut algorithm.

3.2 Barnes-Hut performance

The speedup measurements for the Barnes-Hut algorithm follow the theoretical speedup, as long as the problem size is larger enough for the number of nodes. For 10^6 bodies, 32 nodes is still interesting, but the measured speedup starts to leave the theoretical estimation. There is an odd bump in the speedup measurements for 12, 16 and 20 nodes. This could be due to the different CPUs as mentioned previously, or the load balancing is not sufficiently reliable. The second aspect is discussed in the following section. For the simulations with 10^5, from 16 nodes it is clear that the problem is too small for the number of nodes which is used. The speedup does not go pas the value of 10, and would certainly even decrease for even larger numbers of nodes.
The issue with the Barnes-Hut algorithm in parallel is the load balancing of each node. In Tables 4 and 5, the standard deviations of the computation times are reported next to the mean values. For both problem sizes, the variations stabilize for the larger number of nodes. This means that as the computation time gets reduced, the idle time of the nodes waiting for the slowest node to finish will become significant.

# nodes	1	2	4	8	12	16	20	24	28	32
init. Bcast [s]	$2 \cdot 10^{-6}$	0.029	0.041	0.041	0.047	0.043	0.053	0.048	0.031	0.044
qt build. [s]	2.37	2.32	3.05	3.03	3.04	3.03	3.03	2.33	2.72	2.68
comput. [s]	97.8	48.7	28.64	11.77	9.89	7.45	6.012	3.83	3.9	3.43
	-	±0.56	±0.71	±0.82	±0.72	±1.05	±0.43	±0.25	±0.48	±0.50
sync. [s]	0.015	0.015	0.019	0.025	0.026	0.026	0.031	0.029	0.025	0.024

Table 5: For 10^6 bodies, measurements of each operation on Deneb using the Barnes-Hut algorithm.

3.3 Load balancing performance

In Figure 8, the computation time for each time step, for each node is shown. The node assignment is performed according to the algorithm presented previously. The setup for this simulation is two masses of 1500 bodies each, which are destined to collide at some point. For the first phase, while the two masses are compact and the bodies are attracted to the respective centers, the computation increases on all nodes since the bodies are getting closer and close together. Once the bodies start to move away from the centers, some will leave the gravitational field of the local system due to numerical integration errors over time, whilst the others will start gravitating around the heavy body located in the respective centers. Once the bodies are spread out in space randomly, the load balancing becomes critical.
The second node ($node_1$), has more a longer computation time than the other nodes until iteration # 3800. This is the case since it has been assigned approximately 750 nodes which are much closer together than the bodies assigned to the other nodes. Indeed, if a group of 100 bodies is close together, than it will take more computations to calculate their interactions than 100 bodies which are far apart. The same applies to one group of 100 bodies, or 10 groups of 10 bodies. Therefore the actual node assignment is acceptable, but could be improved by taking into account the density of bodies as well.

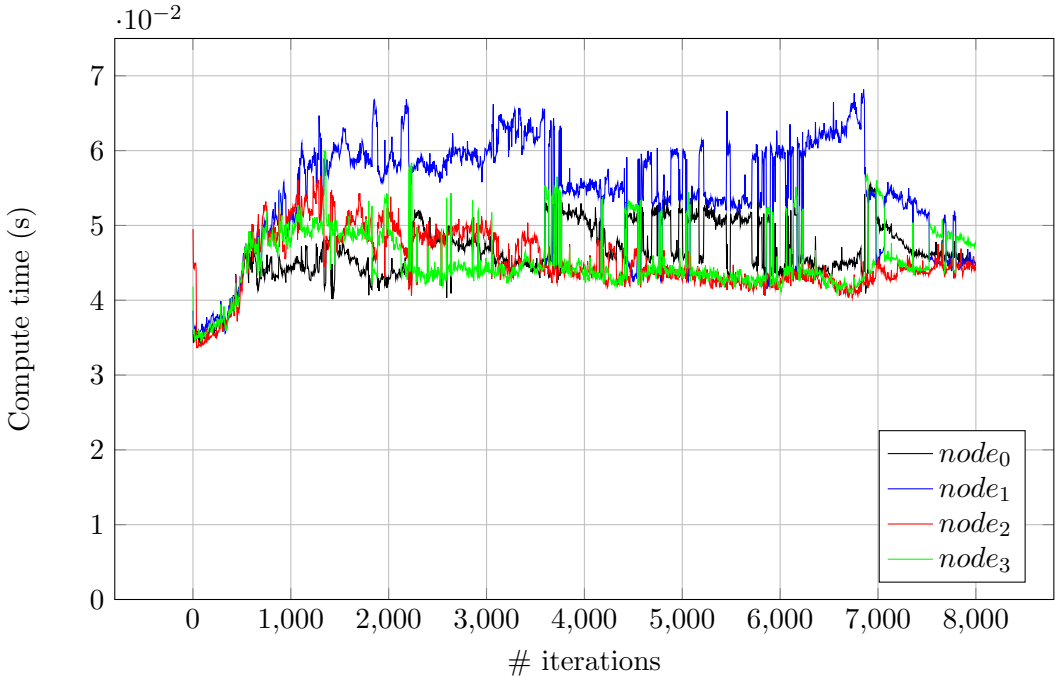


Figure 8: Compute time for the quad tree approach, using four nodes and 3000 bodies.

When considering nodes 0 and 1 (black and blue lines), between iterations #4500 and #6000, the effect of assign a group of bodies can be observed, see Figure 9. As a group of bodies which are compact are assigned either nodes 0 or 1, the compute time jumps. This happens since this groups of bodies is time consuming to calculate, and will create an in balance in the nodes if poorly assigned. The limits of assigning the bodies only by space position, and not density, stand out here. To solve this issue, one could implement a body assignment strategy which takes into account the number of bodies which must be assigned and also the density of bodies. Each node could be given a capacity expressed in $nbBodies \cdot density$ to solve this issue.

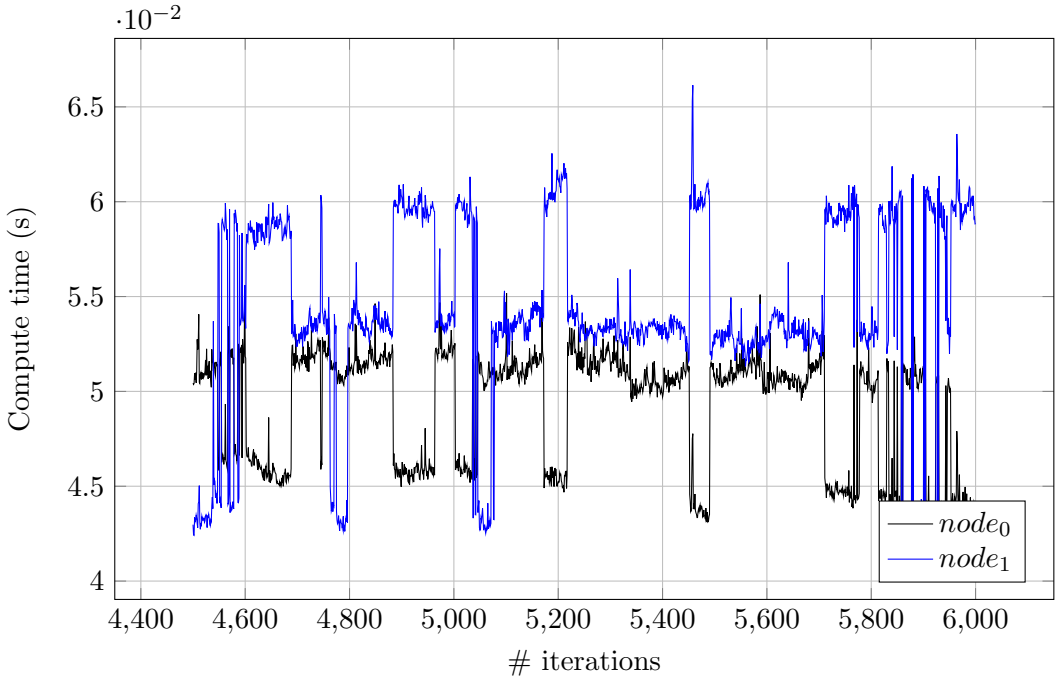


Figure 9: Compute time for the quad tree approach, using four nodes and 3000 bodies.