PROJECT PHPC-2016

A High Performance Implementation of the 2D N-Body Gravitational problem: Benchmark of the Barnes-Hut algorithm compared to the Brute-Force algorithm.

Principal investigator (PI)	Gael Lederrey
Institution	EPFL-CSE
Email	gael.lederrey@epfl.ch
Involved researchers	Only PI
Date of submission	June 10, 2016
Expected end of project	June 24, 2016
Target machine	Deneb
Proposed acronym	NBODY-BHVSBF

Abstract

The n-Body problem has been well studied by many scientists in many different fields such as atoms simulation or astrophysics. In this project, we present the implementation of the Barnes-Hut algorithm. A distributed memory version using the MPI library will also be presented. A benchmark comparison is done with the Brute-Force algorithm, serial and parallel versions. The whole project can be found at this url: https://github.com/glederrey/nBody-PHPC-2016-EPFL

1 Scientific Background

1.1 Gravitation

The n-Body problem covers the whole scale of science from the subatomic particles to the gigantic galaxies. In this project, we decided to take the Gravitional n-Body problem as a real case study. The force[1] of a body 1 on a body 2 is given by:

$$\vec{F}_{1\to 2} = G \cdot \frac{m_1 m_2 (\vec{x}_2 - \vec{x}_1)}{\|\vec{x}_2 - \vec{x}_1\|^3} \tag{1}$$

where G is the gravitational constant and is equal to $6.674 \cdot 10^{-11}$ $[Nm^2kg^{-2}]$, m_i are the masses of two bodies and $\vec{x}_i$ are their positions. As we can see with this equation, the forces decreases proportionally to $1/r^2$, r being the distance between two bodies. Therefore, the forces between two far bodies is small.

1.2 Algorithms

In order to solve the n-Body Gravitational problem, there exists a really simple algorithm called Brute-Force. It consists in looping over all bodies and inside this loop, a second traverses all bodies.

Algorithm 1 Brute-Force

```
n = Number of bodies
Initialize n bodies
while t < t_end do
    for i = 1 to n do
        for j = 1 to n do
            Apply forces from body j on body i
    for k = 1 to n do
        Update the body k
```

We can clearly see that the complexity to compute all forces is $\mathcal{O}(n^2)$. Updating the bodies will take only $\mathcal{O}(n)$ time. It just consists of looping on all the bodies and adding $v \cdot dt$ to the position and $a \cdot dt$ to the speed. But most of the time will be spent in the calculation of the forces. For a very large number of bodies this algorithm will be very slow. Therefore another algorithm can be used: Barnes-Hut[2].
The main idea of the Barnes-Hut algorithm is to approximate interactions between bodies which are far away from each other by grouping bodies together. They way to group bodies together is to use their center of mass. On way to implement this is to use a quadtree. Each leaves of the quadtree contains at max one body. Therefore, the height of the tree is approximately $\log(n)$, n is the number of bodies. All the parents leaves contain the information about the center of mass of the children nodes. With this trick, the algorithm will only compute the forces between a body and the center of mass of the far bodies. The aggregation is done using a precision parameter θ. The pseudo-code is given below:

Algorithm 2 Barnes-Hut

```
1: n = Number of bodies
2: Initialize n bodies
3: while t < t_end do
4:     for i = 1 to n do
5:         Add body i in the tree
6:     for j = 1 to n do
7:         Compute all the forces applied to body j.
8:     for k = 1 to n do
9:         Update the body k
```

Adding a body in the tree takes $\mathcal{O}(\log n)$ time (height of the tree). Therefore, adding all the bodies will take $\mathcal{O}(n \log n)$ time. The calculation of the forces will take $\mathcal{O}(n) \times \mathcal{O}(\log n) = \mathcal{O}(n \log n)$ time. And finally, updating the bodies will take $\mathcal{O}(n)$ time. Therefore, this algorithm will take $\mathcal{O}(n \log n)$ time which is much faster than the Brute-Force algorithm for a high number of bodies.

If we use a parallel version of these algorithms, we will have to send the bodies to the other process. Since each body needs to compute its forces with all the other bodies, the process will exchange all n bodies. This means that the communication takes $\mathcal{O}(n)$ time.

1.3 Too close and too far?

Two problems occur with the n-Body problem:

- What do we do when a body is moving too far form the other bodies?
- What do we do when two bodies are really close?

We decided to address these two problems in a very simple manner. First, we define a maximum and minimum distance. We use the maximum distance such that the root node of the tree has a height and a width of two times this maximum distance. If a body is outside of this root node, then we delete it. And if two bodies have a distance smaller than the minimum distance, we combine them using a perfect inelastic collision, *i.e.* the mass of the new body will me the sum of the two colliding bodies and the speed will be:

$$\vec{v} = \frac{m_1 \cdot \vec{v}_1 + m_2 \cdot \vec{v}_2}{m_1 + m_2}$$

2 Project Description

In this project we want to implement a high performance version of the Barnes-Hut algorithm in C++ [3]. A simple version of the Brute-Force algorithm will also be implemented to be used as a benchmark for the Barnes-Hut algorithm. At the end, four different codes will be produced:

- Serial version of the Brute-Force algorithm
- Distributed memory version, using MPI [4], of the Brute-Force algorithm
- Serial version of the Barnes-Hut algorithm
- Distributed memory version, using MPI, of the Barnes-Hut algorithm

The three first codes will be used as benchmark for the distributed memory version of the Barnes-Hut algorithm. Indeed, to prove that this version is the fastest, we need to make sure it surpasses the Brute-Force algorithm.

For the two distributed memory codes, we apply Amdahl's law. We will then compare the strong scaling and the weak scaling.

3 Implementations

These four applications are implemented in C++. As it was said in the previous section, the distributed memory versions use the MPI library. Everything has been compiled using `gcc` version 4.8.4. The codes have been debugged using the general debugger `gdb`, and `valgrind` has been used to remove all the memory leaks.

3.1 Optimization

For each of the codes, we compile it once without using any optimization flags and once with the following two flags: `-Ofast` and `-ftree-vectorize`.

3.1.1 Brute-Force

Not much efforts were put onto the optimization of the Brute-Force algorithm since it exists only for a benchmark purpose. For the MPI version of the code, the first (outer) loop is divided into the number of processes. The second (inner) loop is still on the n bodies. After updating the bodies, we use the function `Allgatherv` of MPI to distribute the new bodies to all the process.

3.1.2 Barnes-Hut

For the Barnes-Hut algorithm, we first assign each node to a process in order to keep a good load-balancing. This part is serial. Then, we can calculate the forces and update the bodies in the nodes belonging to its process. Then, we used the function `Allgatherv` of MPI to distribute the new bodies to all the process. Finally, we can reconstruct the tree. We did not implement a heuristic for the construction of the tree. But it can be a good idea for future improvement since this step can take some time.

4 Computational time

In this section, we want to show that the Brute-Force algorithm is in $\mathcal{O}(n^2)$ and that the Barnes-Hut algorithm is in $\mathcal{O}(n \log n)$. In Figure 1, the average iteration time taken by the two algorithms are shown for different number of bodies. Different values for the precision parameter θ were used. We can clearly see that the Brute-Force algorithm has the same slope as the red line. We can also see that the Barnes-Hut algorithm with $\theta = 0$ has the same slope. This is normal since it doesn't aggregate any bodies. But if $\theta > 0$, we can see that the slope becomes the same as the red dotted line for $\mathcal{O}(n \log n)$, especially with a large number of bodies. Therefore, the experimental computation times correspond to the theoretical ones.

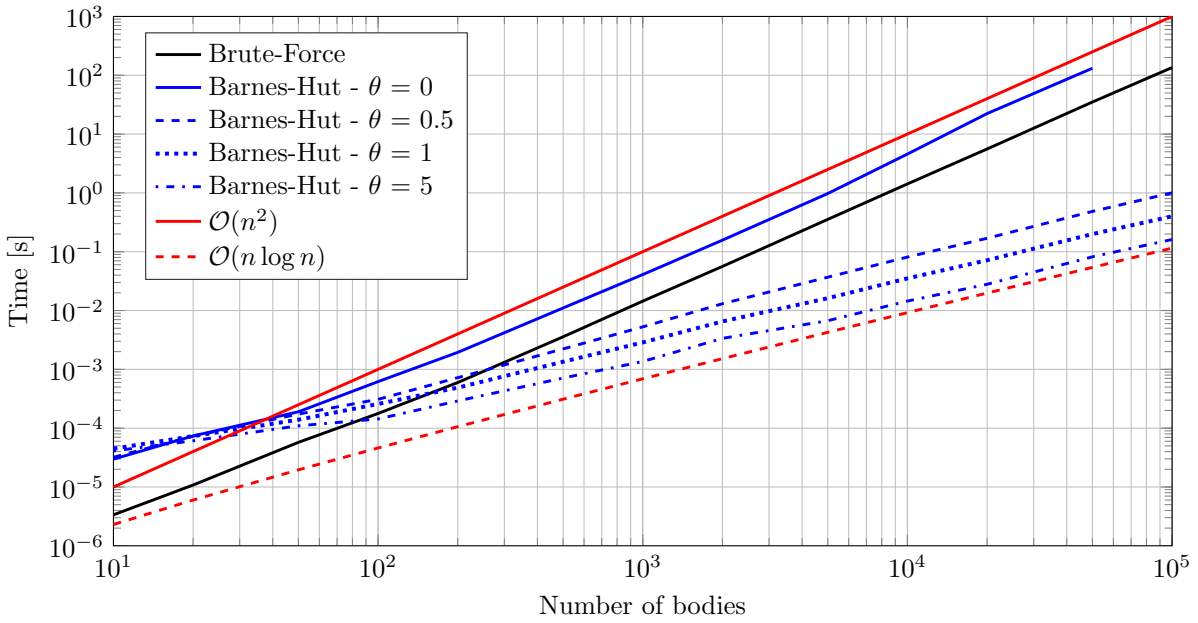

Figure 1: Average time of the main iteration for the Brute-Force and the Barnes-Hut algorithms. Different values for the precision, θ, have been used. The theoretical times are given in red.

An interesting fact is to see that for a small number of bodies, the Brute-Force algorithm is faster than the Barnes-Hut algorithm. This may be due to the optimization made by the compiler and the implementation of the Barnes-Hut algorithm. However, with 10^5 bodies, we see that the Barnes-Hut algorithm, $\theta = 1$, is more than 1000 times faster than the Brute-Force algorithm. For the rest of the project, we will always use $\theta = 1$.

5 Amdahl's law

Amdahl's law is given by:

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}} \tag{2}$$

where S_p is the theoretical speedup, $1 - \alpha$ is the parallelizable part of the code and p is the number of processors. Ideally, we should have $S_p = p$ but this does not happen in reality. To compute Amdahl's law, we don't take into account the time taken to load the initial data. It can take some time but it is done only once. On a long simulation, this time will be negligible.

5.1 Brute-Force

In the Brute-Force algorithm, everything can be parallelized except the communication between the process. Therefore we can expect a speedup very close to the perfect speedup, *i.e.* $S_p = p$.

5.2 Barnes-Hut

In the Barnes-Hut algorithm, the communication between the process is serial. The assignment of the nodes to the processes needs to be done in serial, so does the construction of the tree. All processes will do these two things. The construction of the tree can take a bit of time, therefore, we will except a smaller speedup.

5.3 Results

Table 1 presents different times taken by the code for the Brute-Force and the Barnes-Hut algorithms. The different times are: $\bar{t}_{it}$ for the average time of the iteration, $\bar{t}_c$ for the average time taken for the communication, $\bar{t}_a$ for the average time taken for the assignment of the nodes to the process and $\bar{t}_b$ for the average time taken for building the tree. The time $\bar{t}_{it}$ takes into account all other times and in addition the parallel times: computing the forces and updating the bodies. Therefore we can easily determine α, the percentage of serial part of the code, easily. It is given by:

$$\alpha_{\text{bf}} = \frac{\bar{t}_c}{\bar{t}_{it}} \qquad \text{or} \qquad \alpha_{\text{bh}} = \frac{\bar{t}_c + \bar{t}_a + \bar{t}_b}{\bar{t}_{it}}$$

Algorithm	n	$\bar{t}_{it}$ $[s]$	$\bar{t}_c$ $[s]$	$\bar{t}_a$ $[s]$	$\bar{t}_b$ $[s]$	α
Brute-Force	10^4	1.08	$2.87 \cdot 10^{-5}$	/	/	$2.66 \cdot 10^{-5}$
	10^5	$1.08 \cdot 10^2$	$1.52 \cdot 10^{-4}$	/	/	$1.41 \cdot 10^{-6}$
	10^6	$1.08 \cdot 10^4$	$4.07 \cdot 10^{-3}$	/	/	$3.77 \cdot 10^{-7}$
Barnes-Hut	10^4	$3.04 \cdot 10^{-2}$	$4.95 \cdot 10^{-5}$	$8.39 \cdot 10^{-7}$	$5.45 \cdot 10^{-3}$	$1.81 \cdot 10^{-1}$
	10^5	$3.70 \cdot 10^{-1}$	$6.49 \cdot 10^{-4}$	$2.64 \cdot 10^{-6}$	$7.24 \cdot 10^{-2}$	$1.97 \cdot 10^{-1}$
	10^6	4.39	$9.612 \cdot 10^{-3}$	$2.57 \cdot 10^{-6}$	$7.93 \cdot 10^{-1}$	$1.82 \cdot 10^{-1}$

Table 1: Times used to compute the serial part of the code for Amdahl's law.

Finally, given the serial fraction of the code α, we can plot Amdahl's law for different numbers of bodies. In Figure 2, we can clearly see that, as expected, the speedup for the Brute-Force algorithm is close to the ideal one, *i.e.* $S_p = p$. We can also see that the speedup for the Barnes-Hut algorithm is quite low.

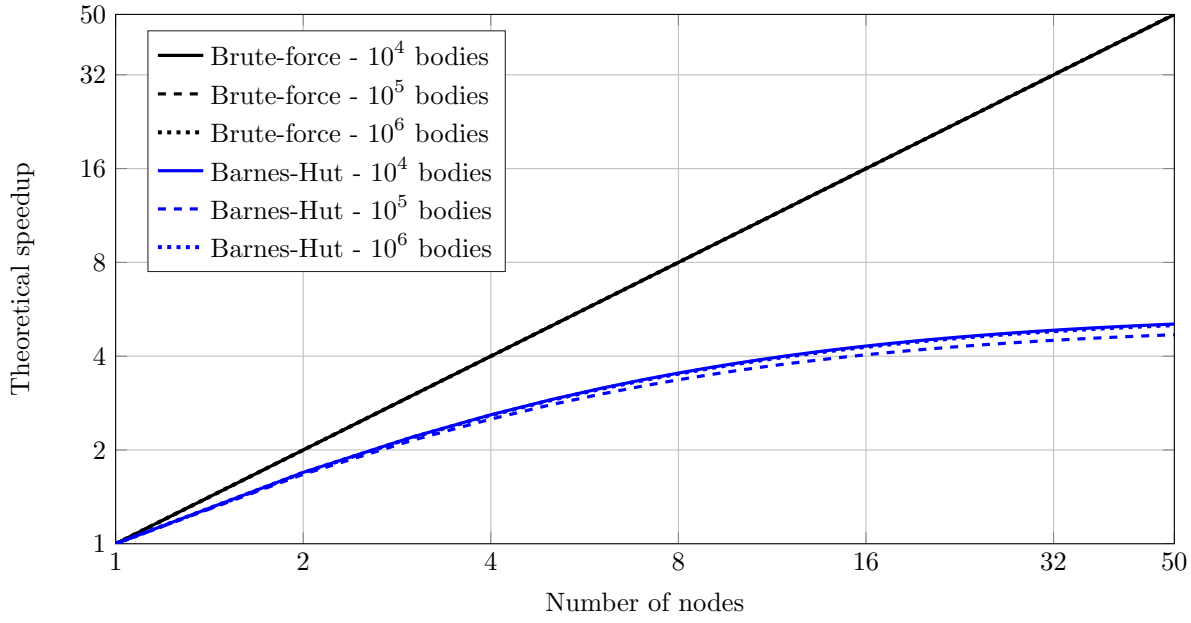

Figure 2: Result of the prediction from Amdahl's law of the theoretical speedup for the Brute-Force and the Barnes-Hut algorithm.

As we can see in Figure 2, the Brute-Force algorithms will scale almost perfectly. Therefore, we can imagine that with a very high number of processes, it will be able to surpass the Barnes-Hut algorithm. It is possible, but this will cost a lot of energy and it is expensive. So, it is better to use the Barnes-Hut algorithm. We will still have a look at the scaling of the Brute-Force algorithm to validate the theory.

6 Strong scaling

In this section, we want to have a look at the strong scaling of the Brute-Force and the Barnes-Hut algorithms. The strong scaling consists, in this case, of fixing the total number of bodies and running the code with a different number of processes. In order to get the speedup factor S_p, we divide the time T_p on p processes by the time T_1 on 1 process.
We ran the code on a private server[1] Here are the specificities of the server:

- OS: Ubuntu 14.04, 64- bit.
- Processor (2×): 2.6GHz Intel Xeon-Haswell (E5-2690-V3-DodecaCore)
- RAM: 8x16GB Micron 16GB DDR4 2Rx4
- Motherboard: SuperMicro X10DRU-i+

This server allows us to run up to 48 process[2] in parallel thanks to Intel's hyper-threading. We can therefore run the code with a different number of process using the same total number of bodies. Finally, we can compare it to the theoretical results, *i.e.* Amdahl's law.

6.1 Results

6.1.1 Brute-Force

The results of the strong scaling for the Brute-Force algorithm are given in Figure 3. We can see that the trend of the experience (blue) follows the theoretical one. For 10^4 and 10^5 bodies, the speedup is even better than the theoretical one. The curve for 10^6 bodies follows closely the theoretical one. We can see a strange behaviour after 18 cores. The reason is that there is always something running in the background on a private server, therefore using 24 cores at 100% is difficult. For a higher number of cores, the hyper-threading is not as good as real cores, therefore we can see some strange behaviour. However, we can see that the Brute-Force algorithm is highly scalable in terms of strong scaling.

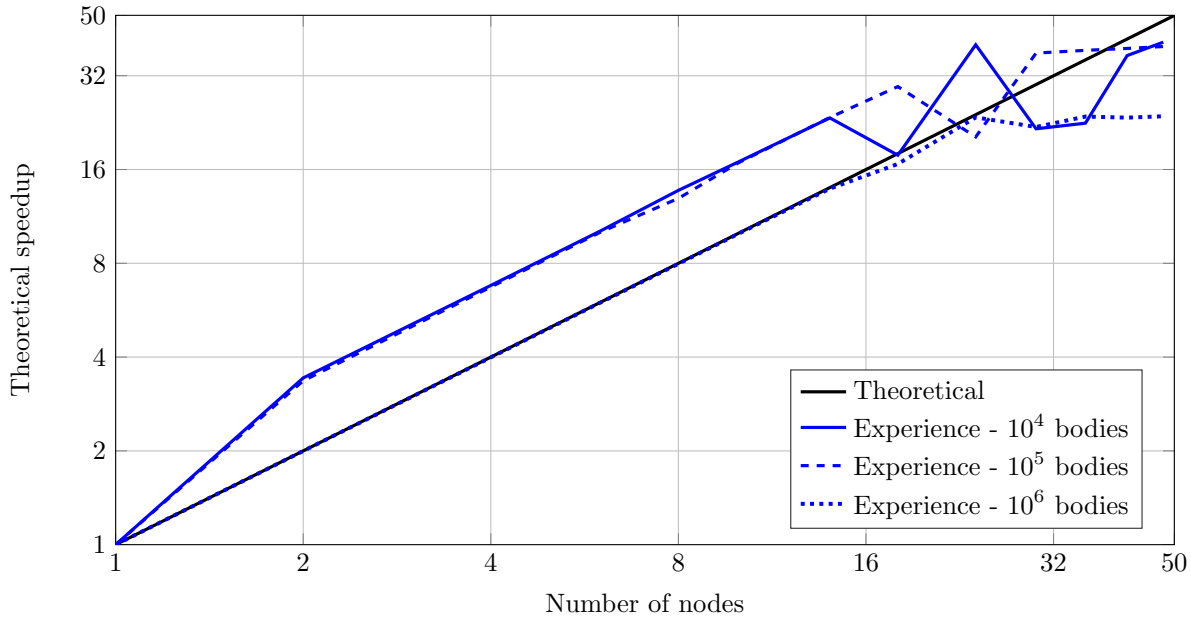

Figure 3: Results of the Strong scaling for the Brute-Force algorithm. We only plot here the theoretical curve for 10^5 bodies since the three theoretical curves are close to each other.

[1] The cluster Deneb was full. We decided to use a private bare metal server (provided by SoftLayer) for 24 hours.
[2] Normally, we should only use 24 cores. But in order to get more results, we decided to use the hyper-threading. Therefore, the results with more than 24 cores needs to be treated carefully.

6.1.2 Barnes-Hut

The results of the strong scaling for the Barnes-Hut algorithm are given in Figure 4. The experimental results follow the theoretical results. We can see that the speedup is even a bit better than the theoretical one. It comes from the fact that when computing the Amdahl's law for the Barnes-Hut algorithm, we computed some values for the average assignment and communication times. But these times can change while using a different number of processes. We can also see a strange behaviour with more than 18 cores. It is due to the same reasons as explained in the previous section. With these results, we confirm that the Barnes-Hut algorithm is not highly scalable in terms of strong scaling.

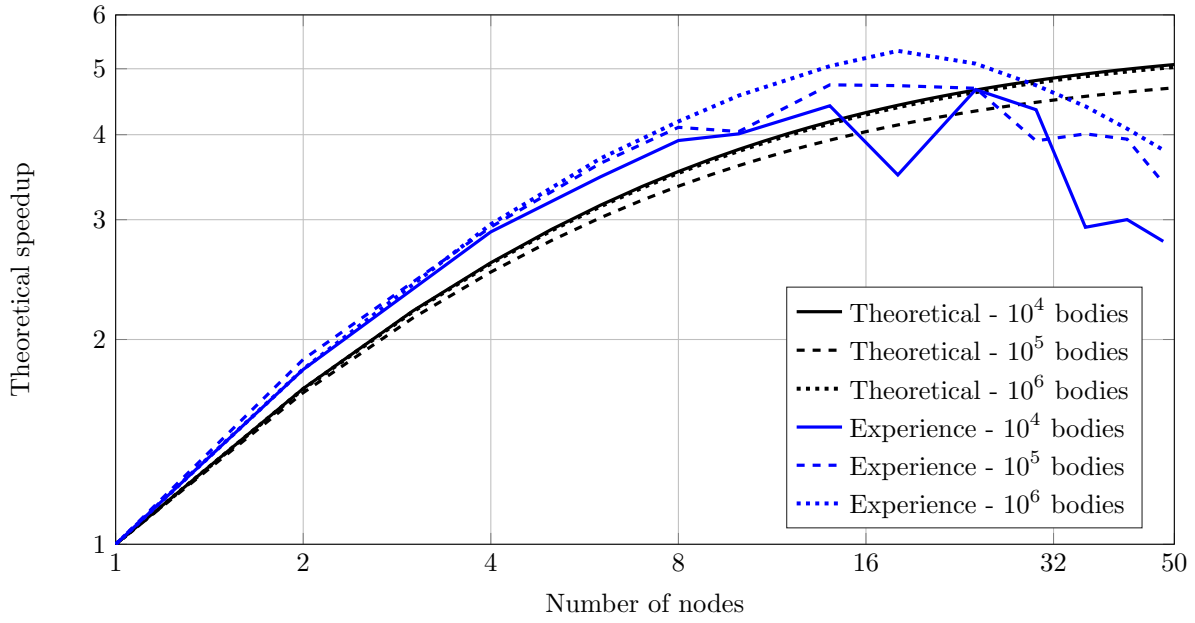

Figure 4: Results of the Strong scaling for the Barnes-Hut algorithm.

7 Weak scaling

In this section, we present the weak scaling. The weak scaling consists of fixing the number of bodies per process and changing the number of processes. Then, to obtain the efficiency E_p, we divide the time T_1 on 1 process by the time T_p on p process.
We ran the code on a private laptop[3] Here are the specificities of the laptop:

- Laptop: Dell XPS 15 L502X
- OS: Linux Mint 17.3 Cinnamon 64-bit.
- Processor: Intel Core i7-2760QM CPU @ 2.40GHz x 4
- RAM: 2x4GB Micron 4GB DDR3

This laptop is old and doesn't have a lot of cores (4 cores for 8 processes with Intel's hyper-threading). Nevertheless, we can grasp the concept of weak scaling. We must be careful while comparing the results with the previous section since we used a different machine.

[3]The cluster Deneb was still full at that time. The time limit for private server, described in the previous section, was over.

7.1 Results

7.1.1 Brute-Force

For the weak scaling, we expect a graph with a very shallow slope, *i.e.* the efficiency stays around 1. The results for the Brute-Force algorithm is given in Figure 5. In this case, we can see that the mean iteration time becomes bigger and bigger. Therefore, the efficiency drops down with a larger number of processes. This is quite surprising compared to the good strong scaling results of this algorithm. To analyse the the weak scaling, let us fix the bodies per process to N. If we have p processes, the complexity of the main loop becomes $\mathcal{O}(\frac{pN}{p} \times pN) = \mathcal{O}(p \times N^2)$. This is because we only divide the number of bodies in the first `for` loop and not in the second. With this complexity, as the number of processes increases, so does the computation time and, therefore, the efficiency decreases.

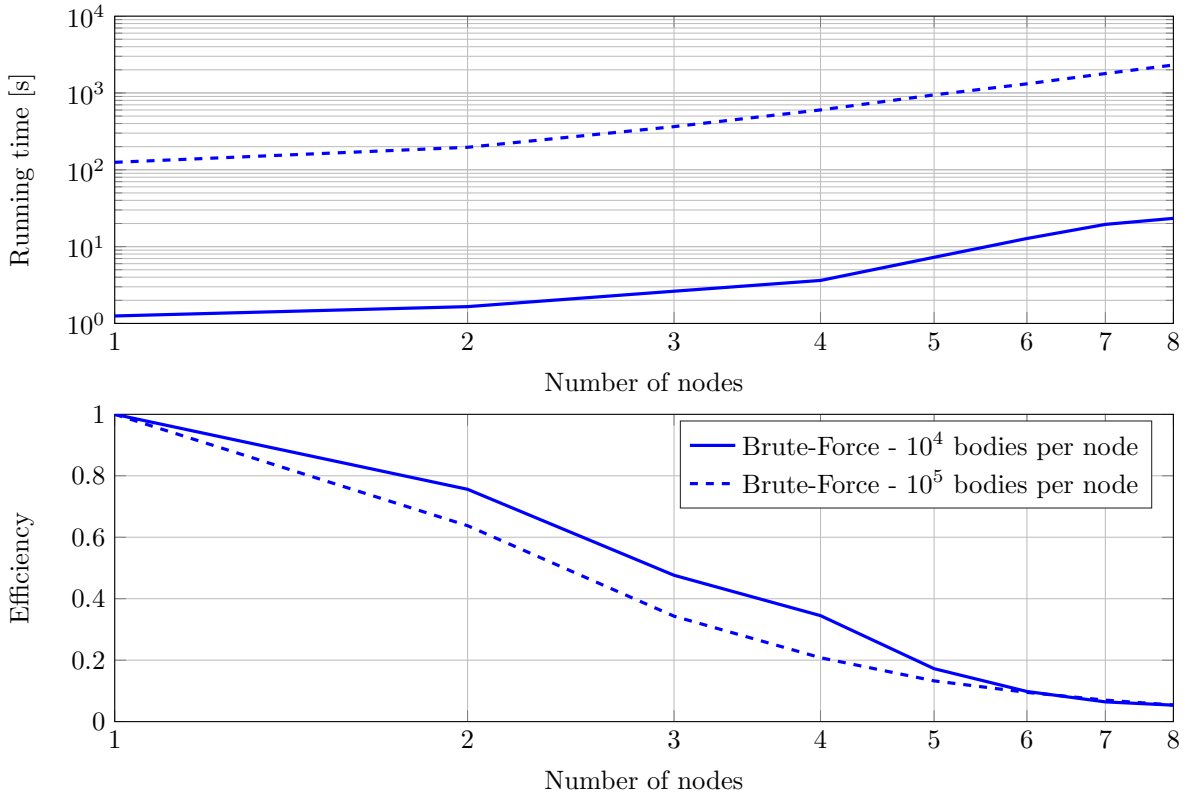

Figure 5: Results of the Weak scaling for the Brute-Force algorithm. The graph above is the mean iteration time in function of the number of process. The graph below is the efficiency in function of the number of process.

7.1.2 Barnes-Hut

The results for the Barnes-Hut algorithm are given in Figure 6. We can see that we have the same behaviour as for the Brute-Force algorithm. We can think in a similar way as for the Brute-Force algorithm. The complexity of the algorithm is $\mathcal{O}(n \log n)$. Therefore, if we have N bodies per process and p process, we will have a complexity of $\mathcal{O}(\frac{pN}{p} \times \log(pN)) = \mathcal{O}(N \log(pN))$. We can clearly see that the running time will increase as the number of process increase. However, we should have a smaller increase than the Brute-Force algorithm. But we would need to redo the experience on more nodes to make sure we can observe this behaviour.

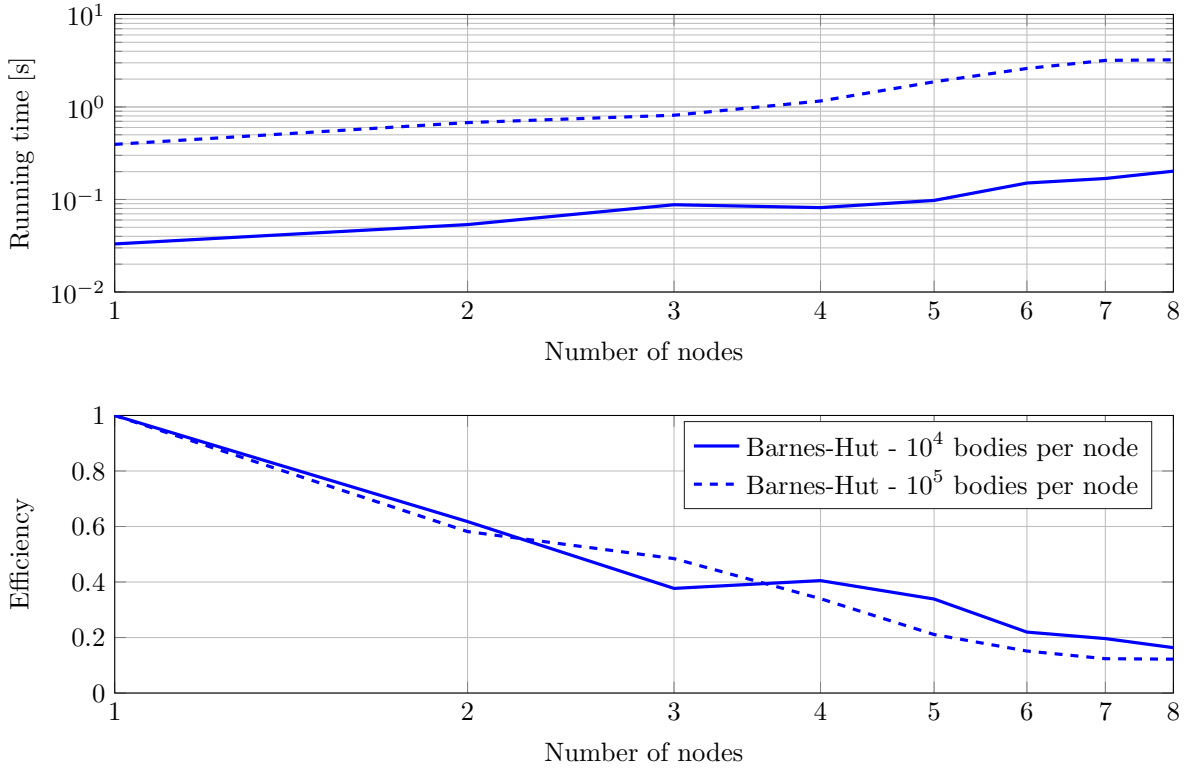

Figure 6: Results of the Weak scaling for the Barnes-Hut algorithm. The graph above is the mean iteration time in function of the number of process. The graph below is the efficiency as function of the number of processes.

8 Conclusion

In this project, we tested two algorithms on the n-Body Gravitational problem: the basic Brute-Force algorithm and the powerful Barnes-Hut algorithm. It was very interesting to compare these two algorithms because we could see that the slowest algorithm, Brute-Force, was more scalable than the fastest algorithm. The interesting fact is that even if we use a lot of process, this algorithm will never be as fast as the Barnes-Hut algorithm. This is a really interesting fact because it teaches us that sometimes developing a good algorithm is far more interesting than using an old and slow algorithm on a powerful cluster. It is even more ecological since we will use less nodes.

9 Budget

In this section, we want to present a resource budget for a huge experience. This simulation will use 10^8 bodies. We want to run it for 2000 time steps but we will only write the positions of the bodies every 5 time steps.

9.1 Computing power

As we saw, the Barnes-Hut algorithm has a complexity of $\mathcal{O}(n \log n)$. Using the times in Table 1 (and more experience not showed in this report) and assuming that the time of the iteration follows $\beta \times n \log n$, we can estimate $\beta \simeq 0.33 \cdot 10^{-6}$. Therefore, we can estimate the time required for one iteration with one process and 10^8 bodies: $t \simeq 600$ $[s]$. Hence, the total time will be around 14 days: $\frac{2000 \times 600}{3600 \times 24} \simeq 14$ [days].

We would like to reduce this to less than a week, therefore we would require a speed up of around three[4]. Figure 4 shows us that we would need 6 nodes.

9.2 Raw storage

Writing the data of 10^6 bodies for one time step takes 50 $[MB]$ on the hard drive. It is a linear scale with respect to the number of bodies. Therefore, writing the data of 10^8 bodies will take 5 $[GB]$. Since, we want to print the data every 5 time steps for 2000 time steps, this means that we will write 400 times. Therefore, we will need 2 $[TB]$ of space on the hard drive.
The RAM has also been tested. For 10^6 bodies, it takes around 600 $[MB]$. Therefore, for 10^8 bodies, we would need around 60 $[GB]$.

9.3 Grand Total

Total number of requested cores	6–10 [cores]
Minimum total memory	64 [GB]
Maximum total memory	128 [GB]
Temporary disk space for a single run	2 [TB]
Permanent disk space for the entire project	4 [TB]
Communications	Pure MPI
License	own code (BSD)
Code publicly available ?	Yes
Library requirements	None
Architectures where code ran	Intel 64

References

[1] Wikipédia - Newton's law of universal gravitation

[2] Berkley - CS267: Lecture 24, Apr 11 1996: Fast Hierarchical Methods for the N-body Problem, Part 1

[3] Stroustrup B.,*Programming – Principles and Practice Using C++*, Addison-Wesley, May 2014

[4] The MPI Forum, *MPI: A Message-Passing Interface Standard*, Technical Report, 1994

[4] Writing the bodies in a file will take a lot of time, therefore it is better to use a higher speedup.