N-Body Algorithms on Shared Memory

Malo Drougard

December 19, 2014

Contents

Chapter 1

Introduction

The goal of this project is twice, in one hand we want to simulate the gravity on a big number of particles and in the other hand we want to study the effect of multi-threading. So, we have see the theoretical aspect of the problem, then we have develop implementations. All the codes are available on the following github repository: https://github.com/MaloDrougard/NBody. After that, we have allow this implementation to run on multiple threads and take measures. Finally, we have analyze this measures. Thus, we have compare and illustrate some aspects of the parallelization and develop a quite complete view of the NBody problem. We invite you to read and enjoy this document.

Chapter 2

NBody problem

In this chapter, we will describe the NBody problem. NBody stay for a simulation of N particles or bodies under some constraints. Let's focus on the gravity phenomena. We want to model the attraction between a set of particle during a laps of time. Assume that a particle is a point of mass and, we are in Newtonian physic. The force acting on particle x_i at a given time t is:

$$f(x_i) = \sum_{j=0, i \neq j}^{n} G * m_i * m_j \frac{r_{ij}}{\|r_{ij}\|^3} \tag{2.1}$$

where:
$r_{ij} = x_j - x_i$,
x_i: position of the particle i,
m: mass of the particle

We do this for all particles $i = 1, ..., n$.
See that the additive property of forces and the superposition property of gravity (a particle don't affect the force of attraction generate by a other particle) have permit that this summation is valid in our model.

Since we have done these calculations, we have the force acting on each particle at given time. But what happens next? The particles move and their forces change. This is very difficult to calculate the real value of this function f. So, we make an approximation and we consider that for a short time, call $delta - t$, the force are not changing and the particles are moving in a constant acceleration motion. A new position for every particles can be calculate in this way. We can, then, starting again to calculate the force for every particles and do this how many times we want. We call one iteration of calculating the force and the new position for every particles a slot. Thus, we have a simulation for our n bodies under gravity. Remark that we make abstraction of collision. The next chapter will look at algorithms that perform the simulation.

Chapter 3

Algorithms

Theres exist a infinite number of algorithm to resolve the NBody problem. We are going to explore two of them, the Brute Force algorithm and the Barnes-Hut algorithm. Both of them have some advantage and inconvenience. For each we first describe them using pseudo code, see appendix to have more information about notation. Then, we make a theoretical time analysis. All time-complexity are on the number of particle N. At last, we look if there is a way to parallelize them.
Note about parallelism: we consider that we are in a shared memory context. The different parallel section can have access on a commune memory and thus have access on the same object.

3.1 Brute Force

3.1.1 Description

The brute force is very intuitive, we do exactly what we see in the last chapter.

Algorithm 1: NBody

Input: $ParticlesSet$, $\#Slots$
Output: -

```
1  while #Slots ≠ 0 do
2      foreach Particle_i ∈ ParticlesSet do
3          foreach Particle_i ∈ ParticlesSet do
4              F = 0;
5              if Particle_j ≠ Particle_j then
6                  F = F+ force of Particle_j;
7              end
8              update the acceleration of Particle_i with F;
9          end
10     end
11     foreach Particle_i ∈ ParticlesSet do
12         update the position;
13     end
14     #Slots ← #Slots − 1 ;
15 end
```

3.1.2 Time Complexity Analysis

We see that there is a nested loop for in a other loop for. Since each one is running N times we have a complexity of $O(N^2)$.

3.1.3 Available parallelism

One way to parallelize this is to simply split the set in equal part and then, give one part to each thread. There is one thing we must take care of. This is that each thread computes the force with the wanted position and not a mess of update one and old one. We can manage this by keep a reference set that contains old positions and we compute all the value from this. At the end of the slot, when all the threads have update his part we update the reference set with all the part of the different thread.

3.2 Barnes-Hut

3.2.1 Description

The Barnes-hut is based the following approximation. When a group of particle is far enough of a other particle. The attractive force generate from this group of particle on the individual particle is approximately equal to the attractive force of the center of mass of this group. We can imagine take advantage of this approximation by calculating once the center of mass of a group of particle and use it to calculate the effect of this group on multiple particles. Let's look at an algorithm's that do this using a tree. The root of the tree represent the whole area where the particles are. He have four child that divide the area in four, each child represent a quarter of the area. If there is only one particle one of this area, the child will keep the particle information and have no child. If there are more than one particle the child will contains the total mass of this particles and the center of mass. He will also have four child for each quarter of his area. All the tree has these properties. The Barnes-hut algorithm will

compute the force of one particle by looking at the tree. It will start one the root and see if the mass center is far enough, if yes compute the force using this, otherwise go down in the tree. We will see the trick to avoid the computation with a mass center that contains it self.

Algorithm 2: *Banes − Hut*

Input: *ParticlesSet*, *#Slots*
Output: -

```
1  begin
2      while #Slots ≠ 0 do
3          build the tree calling RecursiveQuadtreeBuild;
4          foreach Particle ∈ ParticlesSet do
5              calculate the total force calling TotalForce;
6          end
7          update position of ParticlesSet ;
8          #Slots ← #Slots − 1 ;
9      end
10 end
```

Algorithm 3: *RecursiveQuadtreeBuild*

```
/* This algorithm give us a recursive way to build a tree from a given
   particles set.                                                    */
```

Input: *ParticlesSet*, *ParentNode*
Output: *Tree*

```
1  begin
2      Node ← new node with ParentNode as parent;
3      if ParticlesSet is empty then
4          Node ← Nil;
5      end
6      else if ParticlesSet have one particle Particle then
7          Node ← new Leaf containing Particle ;
8          Node.Mass ← Particle.Mass;
9          Node.MassCenter ← Particle.Position;
10     end
11     else
12         divide ParticlesSet in four sub-sets, SubSet1, SubSet2, SubSet3, SubSet4 each
           sub-set cover a quarter of the area;
13         Node.Children1 ← RecursiveQuadtreeBuild(SubSet1, Node);
14         Node.Children2 ← RecursiveQuadtreeBuild(SubSet2, Node);
15         Node.Children3 ← RecursiveQuadtreeBuild(SubSet3, Node);
16         Node.Children4 ← RecursiveQuadtreeBuild(SubSet4, Node);
17         Node.Mass ← ∑ Children.Mass;
18         Node.MassCenter ← ∑ Children.Mass ∗ Children.MassCenter;
19     end
20     return Node
21 end
```

Algorithm 4: *TotalForce*

```
/* This algorithm calculate the attraction that are acting on a particle
   given a tree that represent a set of particle                          */
Input: Tree, Leaf, 0 < θ < 1
Output: -
1  begin
2      F ← 0;
3      if Tree is NIL then
4          F ← 0
5      end
6      else if Tree is a Leaf then
           /* if Tree is the same Leaf as Leaf then the following calculus
              should return 0                                              */
7          F ← force between these two leafs;
8      end
9      else if |√2*Tree.Length / (Tree.MassCenter−Leaf.MassCenter)| < θ then
10         F ← force between Leaf and the center of mass of Tree ;
11     end
12     else
13         forall the Children do
14             F = F + TotalForce(Children, Leaf);
15         end
16     end
17     return F
18 end
```

Remark that there is a parameter θ that allow us to set accuracy we want. In the other hand, if the leaf (that represent a particle) is in the tree, then distance between the mass center and particle can not be bigger than the diagonal. Since the accuracy can not be bigger then one, we will never take 9 if the mass center contain the particle. It seems now logic, that 0 is the best accuracy and 1 the worst.

3.2.2 Time Complexity Analysis

We will first calculate the time complexity of each sub routines call in Barnes-Hut, then calculate the complexity of Barnes-Hut.

Time-complexity of *RecursiveQuadtreeBuild*:
The difficulty here is that we don't know when the recursive call split the set. We can also, imagine that the split forever if the particle are infinitely close. Let's do a trick to bound the number of operation between to split. We, first, assume that the particle can not be infinitely close. Imagine that the position of particles are represented in a fixed length format as Int, Float or Double. So, the closed way to have two particles is to have this two particles separate represented number. Imagine the case that the tree is composed by two particles close enough. The algorithm go recursively until the set must split. Doing this he call each time 4 RecursiveQuadtreeBuild, but 3 return immediately because they are empty. So we have a cost in the worst case until the split directly related by the machine precision. This is a big number but a constant, let's call this number c. If we take c as cost of one iteration we can assume that at

each iteration the algorithm split. Let's analyze this algorithm using this trick.

Now, at each iteration the subset must be split in minimum 2 sub-sets so we have:

$$F(N) = F(c + 2 * F(N/2)) = F(c + 2 * (c + 2 * F(N/4)) = ... \tag{3.1}$$
$$and\ F(1) = c \tag{3.2}$$

F is a function that give us the number of elementary operations done to compute this algorithm on input of size N. c is the constant seen before.

We observe that F can be rewrite to the form:

$$F(N) = c + 2c + 2^2c + 2^3c + ... + 2^x + c \tag{3.3}$$

The x that make F ending:

$$N/2^x = 1 \tag{3.4}$$
$$N = x^2 \tag{3.5}$$
$$x = \log_2 N \tag{3.6}$$

We see that we have a geometrical suite up to $\log_2 N$. Finally we have:

$$F(N) = c * (1 - 2^{\log_2(N)+1})/(1 - 2) + c \tag{3.7}$$
$$= c * (2N - 1) + c \tag{3.8}$$
$$= 2cN + c \tag{3.9}$$

So this algorithm runs in big-O(N) time.

Time-complexity of $TotalForce$:

Again we the trick we assume that the tree is a complete binary tree. We start traversing the tree from the root. But the accuracy factor bound the depth of the exploration. So at each node we can only explore a constant number of new node. If the tree is wheel balanced, remark this assumption because we will seen them once more, we reach the Leaf in $O(logN)$. Thus the algorithm runs in $O(logN)$

Finaly let's look at the time complexity of $Banes - Hut$:

At each iteration we have a call of $RecursiveQuadtreeBuild$ and N call of $TotalForce$:

$$O(N) + N * O(\log N) => O(N \log N) \tag{3.10}$$

So, we have a time complexity in $O(N \log N)$.

3.2.3 Available parallelism

We can do a very similar decomposition as in the Brute-Force algorithm. We build the tree from the set. We split the set in equally part. We keep tree as reference, to compute the forces. We give at each thread a subset to process, and at the end when all thread terminate, we rebuild a new set from the update subsets, and starting again the procedure.

See that we don't parallelize the tree build. We will see later the impact of this.

Chapter 4

Implementation

Now we have a good theoretical base to implement a solution for our problem. We have implement both algorithms. We use the C++ language with the OpenMP API [Bla]. The code is implement in a oriented object style. For all the different variante of the code the most class are the same. We only change the main and the needed functions. We develop a special object, TimeAnalyzer, dedicate to retrieve execution time information using OpenMP procedures. We also develop some GUI only for windows. There is quite basic but give us a idea of how the particle moves, remember that we don't take care of collision.
The console software can be launch from a bash console as follow:

> *NBody.exe infile outfile threads slots delta-t acc max-particles*

He basically take a file, from where he generate the set, and give out the result in the outfile. The outfile contains the position of the particles after the computation with the respect of parameters, *threads* for the number threads, *slots* for the number of slots, and *delta-t* the delta time, but also execution time information.

We don't get more in the implementation in detail here, you can access the source code on GitHub repository https://github.com/MaloDrougard/NBody. The Github contains more information about how to use the code and how it's works. We highly suggest you to take a round on it.

Chapter 5

Measures

5.1 Data

The input data are taken from http://bima.astro.umd.edu/nemo/archive/ (Plummer). This data have a well distribution. We consider two sets one with 8096 particles and one with 65536 particles. The chose was motivate be the fact that they was well distribute and that the size scale. So, we can except a $N \log N$ time for the Barnes-Hut and we can take measure without waiting for days. The sequential brute force algorithm take on our hardware about 8 seconds to calculate a slot with 8096 particles.

5.2 Hardware

The measure was take on a server AMD Opertron processor 6272, it have 64 cores. So we can expect run efficiently up to 64 threads or at least around 62-63, because the system also use some core.

5.3 Tools

The TimeAnalyzer object of our implementation retrieves information during the execution and give us in the result file. To be more precise, it gives for each thread and each slot the following time:
For Brute Force:

sequential start	sequential end	parallel start	parallel end

For Barnes Hut:

seq1 start	seq1 end	para start	para end	seq2 start	seq2 end

There is two sequential sections in Barnes Hut, one for building the tree, one for generate the new set. Remark, that the sequential sections are executed from one thread only. So, there is only one thread per slot that don't have a not null value in the sequential table.

We have launch the program from console with a bash script with different arguments. Then we use the R software to paste together the information of the different run and analyze the tables. In fact, R script take in input a text file containing the file names we want to process and then fetch tables and plot graphs. The bash and R script are also available on the git repository.

5.4 Measures

We want to verify two things, the first is the time complexity of our algorithms. To do this we have run our algorithm with one thread on different number of particles. We start with 253 particles and then we increase the number of particles of 253 at each new run. Each run calculate 30 slots, to decrease the impact of random variable and processor optimization.
Secondly we want to see the impact of using multiple threads. To do this, we have run the 8096 particle set with 1 to 64 threads on each implemented algorithm. We have made 100 iterations for each of the this run to minimize the random values and other reasons that would become clear later.

Chapter 6

Results

In this chapter we will explore the measures and see what they tell us. There is two big section, the first one will take care off the time complexity and the second one of the parallelism.

6.1 Time complexity

We want to know if our implementation is consistent with the time analyze done in the previous chapter. In particular we want to determine from how many particle the Barnes-Hut algorithm should be really prefer form the Brute-Force one.

Let's look on graphs who plot the time against the number of particle:

Figure 6.1: Brute-Force: time in function of particles numbers

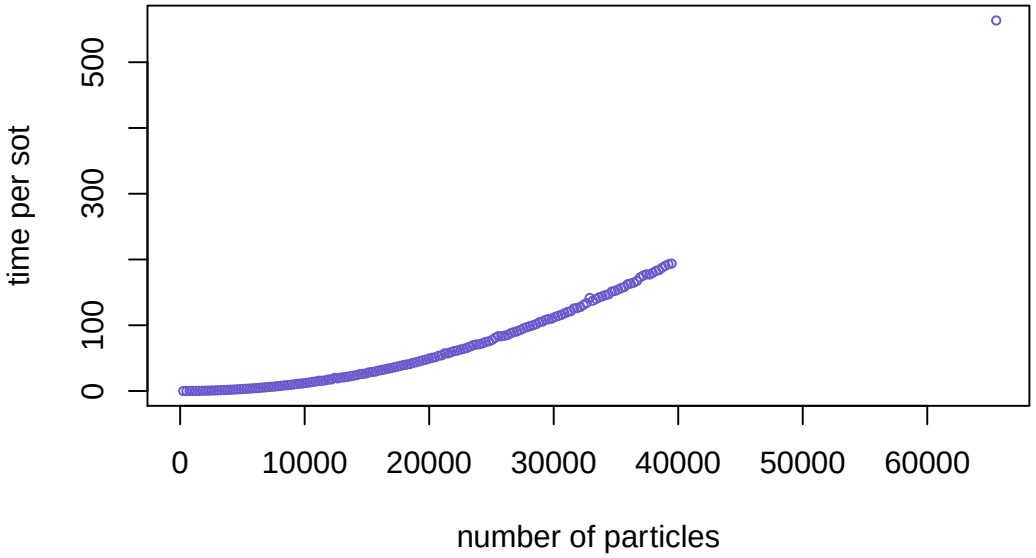


Figure 6.2: Barnes-Hut: time in function of particles numbers

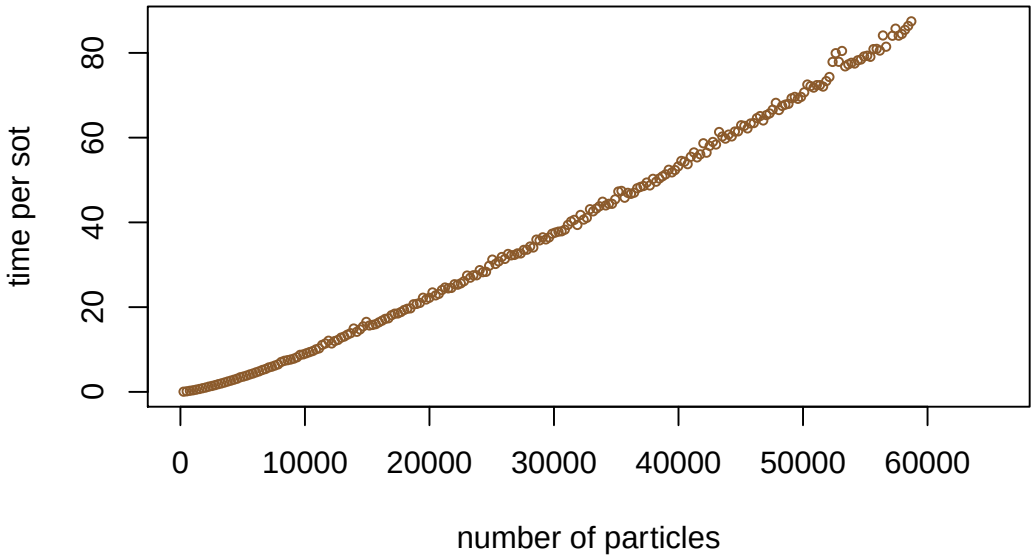
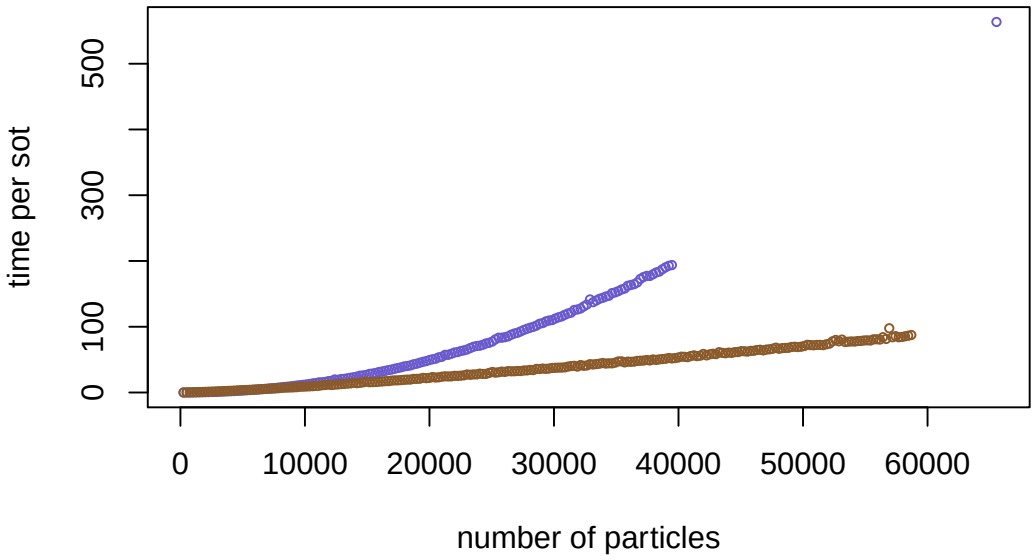


Figure 6.3: time in function of particles numbers



We see that both have the expected form. We don't compute all values for the brute-force algorithm because the program take already tree days to compute this values. We also see that

the value start diverging from 20'000 bodies. This is a quite big number but after the cap the execution time of brute-force increase quickly. So, a good thing is to evaluate the number of particles before starting a execution. Remember that Barnes-Hut us one more approximation. The first approximation is that the time is discrete as seen in the description of the problem.

6.2 Parallelism

We can imagine that if we perfectly parallelize our program, the time to compute a slot is divide by the number of thread. But, you can also imagine that in practice this is no so perfect. This can be mathematized with the notion of speedup. So, we will take a look on this notion and the Amdahl's law prediction. Then, we will look at both algorithms separately.

6.2.1 Amdahl's law

recall the speedup:

$S_p = t_s/t_p$

S_p : speedup
t_s : serial execution time
t_p : parallel execution time

recall Amdahl's law:

$$S \leq \frac{1}{f + \frac{1-f}{P}}$$

f: fraction of the program that can not be parallelized

P: number of processing unit

Taking P tends to infinity, we get the maximal achievable speedup.

$$S_{max} \leq \frac{1}{f}$$

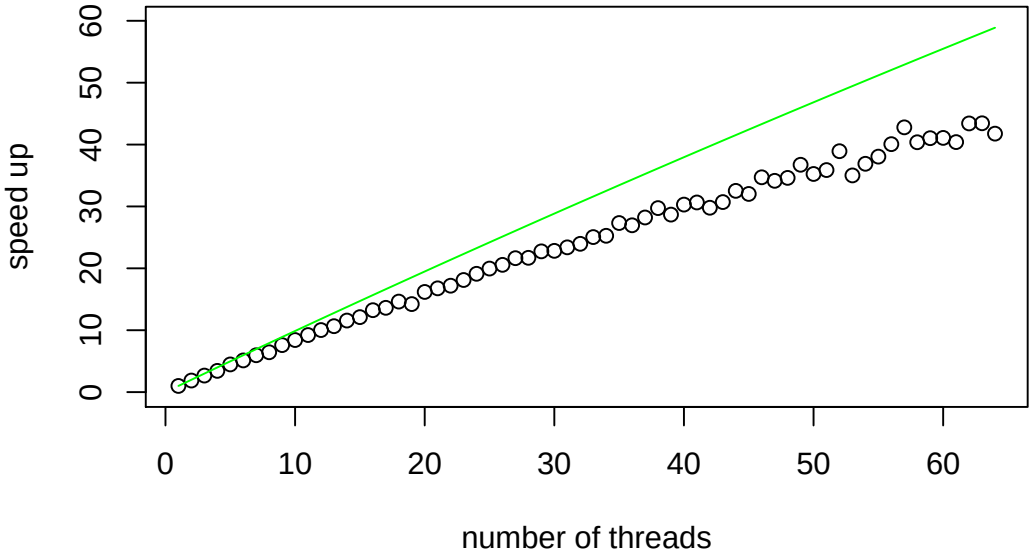
So we can predict the speedup when we increase the number of threads. We see that only the sequential time was take in account. We have assume that there is no waiting dependence between the thread and that the sequential part is constant in time. We will see if the assumptions are true and if we can improve the the prediction.

6.2.2 Brute-Force

We have a fraction $f = \frac{0.01}{7.85} = 0.0013$.(We take the mean of the sequential time over all the different run for the nominator.) f is very close to 0. So, the Amdahl's prediction seems to be linear. For these graphs and the rest of the report, the green curve will represent the the Amdahl's prediction.

Let's took a look on the measured speedup for each number of threads:

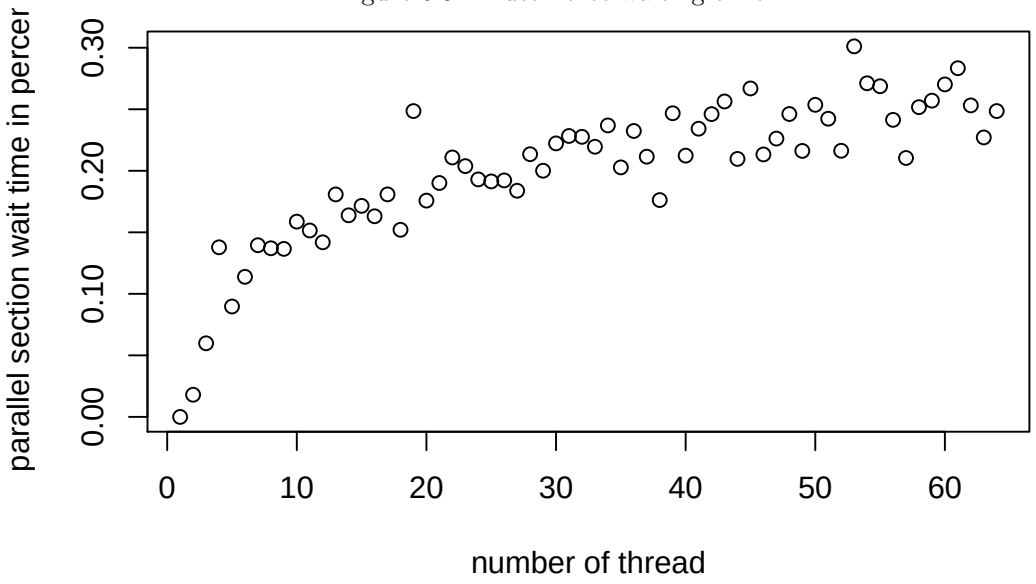
Figure 6.4: Brute-Force speedup



We observe that the seems to be linear but diverge from the prediction. Maybe, because one of the assumption is violate. First let's look if the sequential time is constant. We can imagine that the fact to recompose a set from multiple sub-sets take longer if the there is more sub-sets. But all the sequential time are very little. The maximal sequential time represent a fraction of 0.0015 of the total execution time and the minimal sequential time represent a fraction of 0.0007. The distance between the both is small and can not explain the divergence.

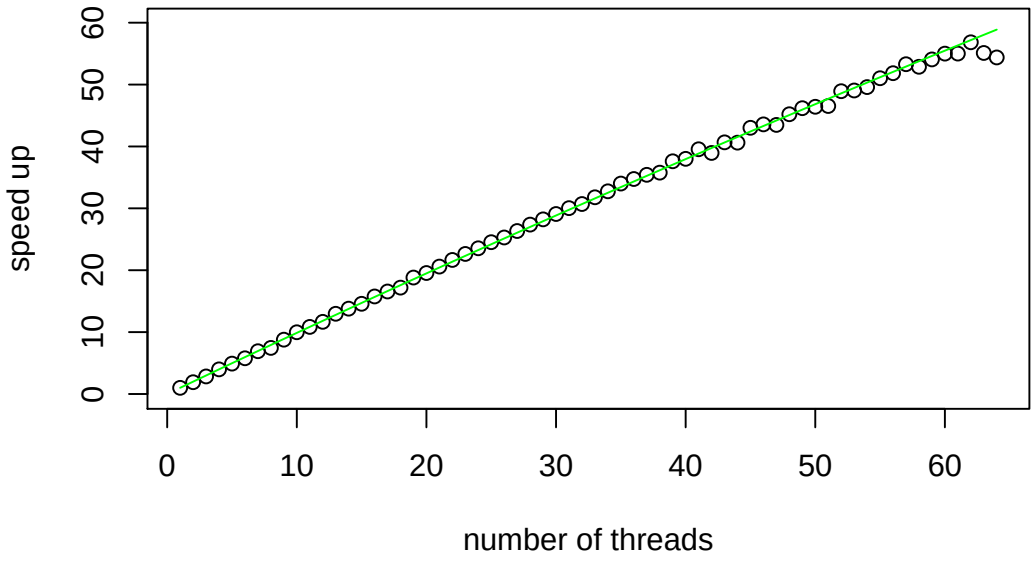
Now lets take a look on how many time on average the faster thread wait on the slowest thread in the parallel section.

Figure 6.5: Brute-Force waiting time



The relative waiting time seems really increase with the number of threads. This can explain the divergence. Let's subtract the waiting of the total execution time and see what would be the speedup.

Figure 6.6: Brute-Force adapte speedup



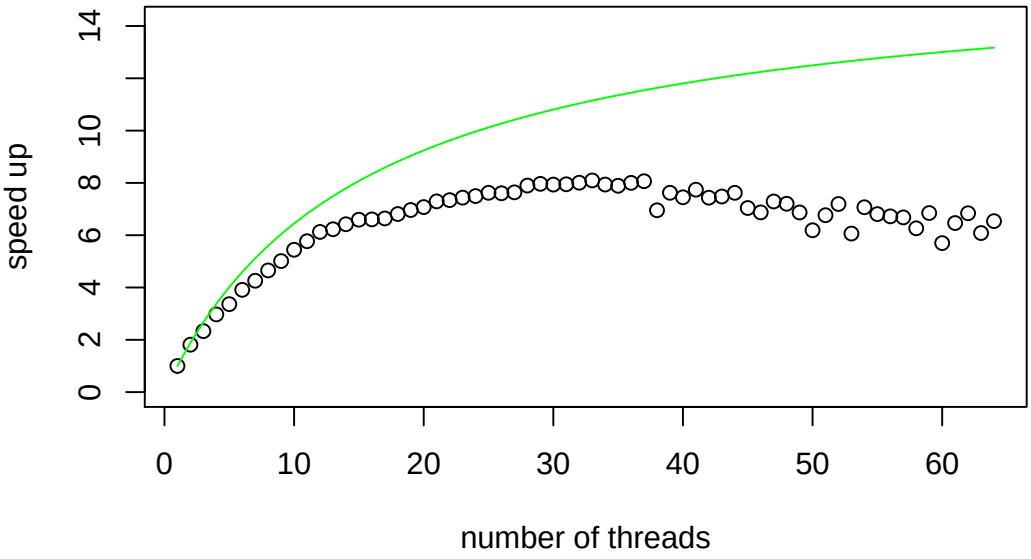
Whouaa! We see that the waiting time explain the divergence. The next step would be understanding why the wait time increase and see if we can do something. This waiting factors seem to come from outside our implementation, since all threads execute exactly the same number of operation (all the subset have exactly the same number of particles at one near). Maybe the system requires always some constant time to synchronize threads and this time have more and more impact, since the total time becomes smaller and smaller.

6.2.3 Barnes-Hut

The same reasoning as in previous section hold.

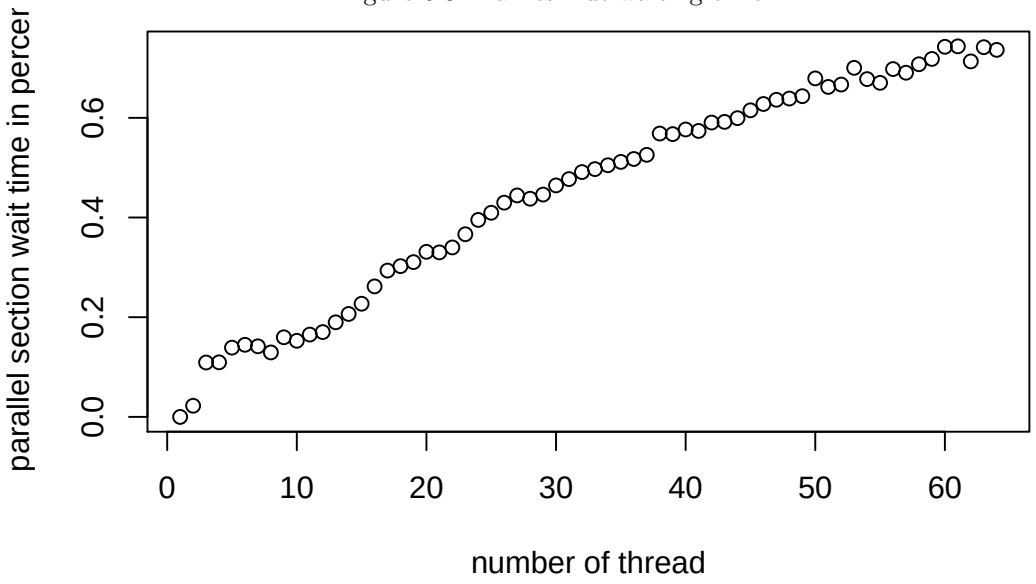
Let's take a look on the speedup. Here the fraction is $f = \frac{0.4}{6.59} = 0.06$. This mean that 6 percent of the code is serial and we can expect a speedup of 17 if we have a infinite number of processor.

Figure 6.7: Barnes-Hut speedup



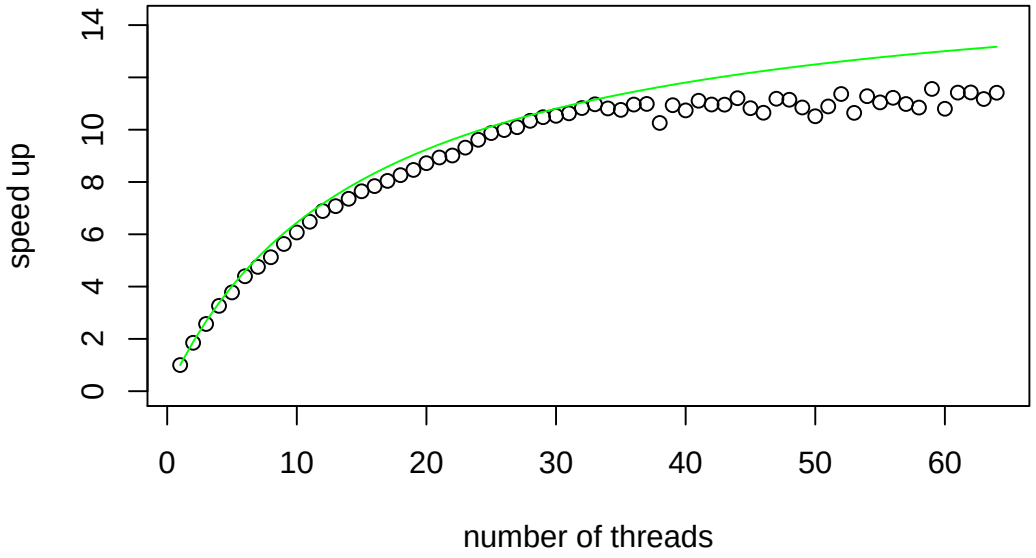
There is a very dramatic divergence. We lost speed but we add more resources. Again the sequential time can not explain the divergence. Let's look at the wainting time in parallel section.

Figure 6.8: Barnes-Hut waiting time



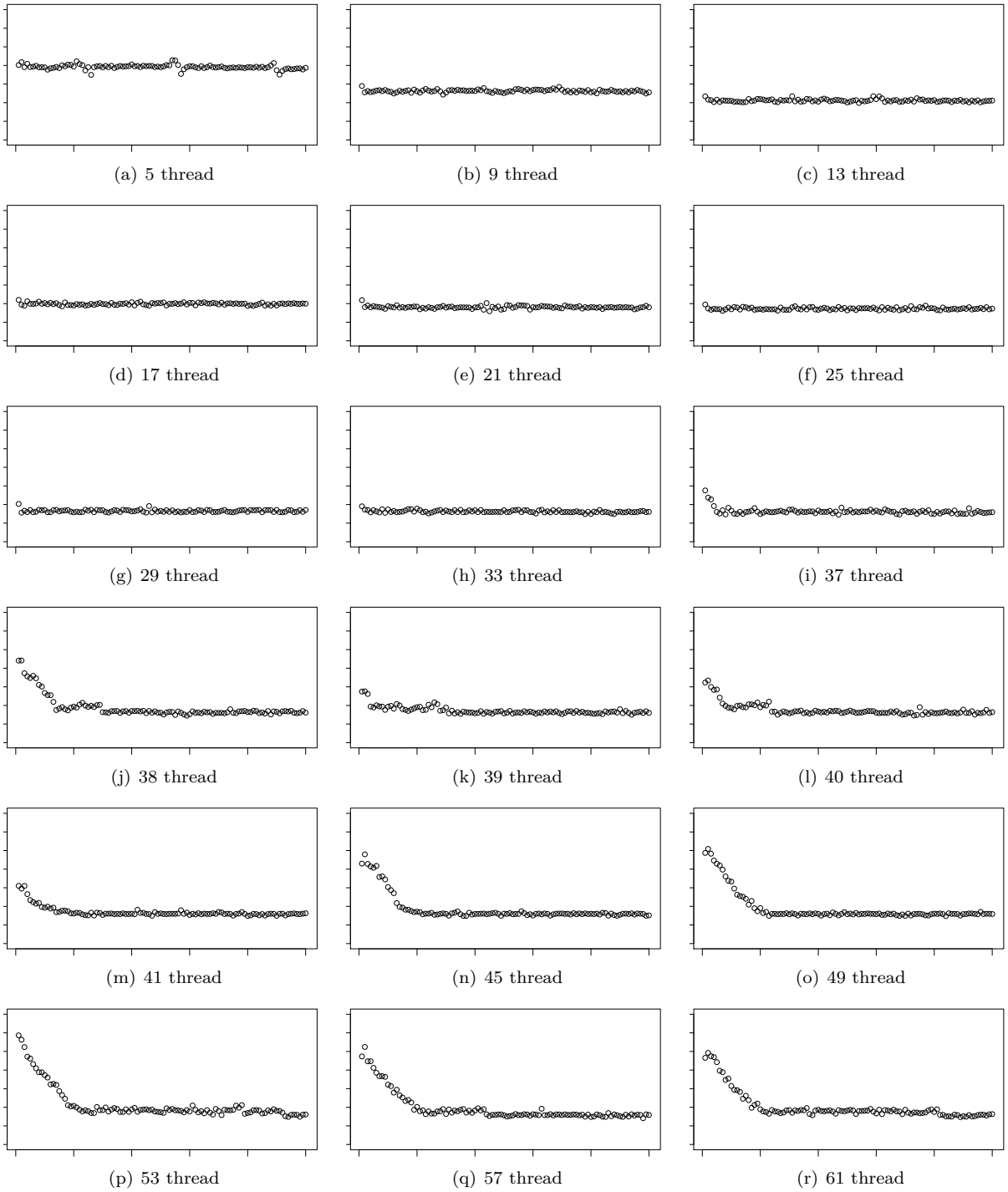
The waiting time seems to grows. Let's do the same trick as before.

Figure 6.9: Barnes-Hut adapte speedup



The is much closer of the expectation, but there seem always to have a divergence starting

between the run with 30 threads and 40 threads. Are assumption are verify so what can cause this divergence. After some research we find a strange behavior, the time per slot seem to decrease during the same run. Fot the next plots, the y axis is the execution time and the y axis is the slot number. Remark that we plot every runs just for the transition passage.



(a) 5 thread (b) 9 thread (c) 13 thread

(d) 17 thread (e) 21 thread (f) 25 thread

(g) 29 thread (h) 33 thread (i) 37 thread

(j) 38 thread (k) 39 thread (l) 40 thread

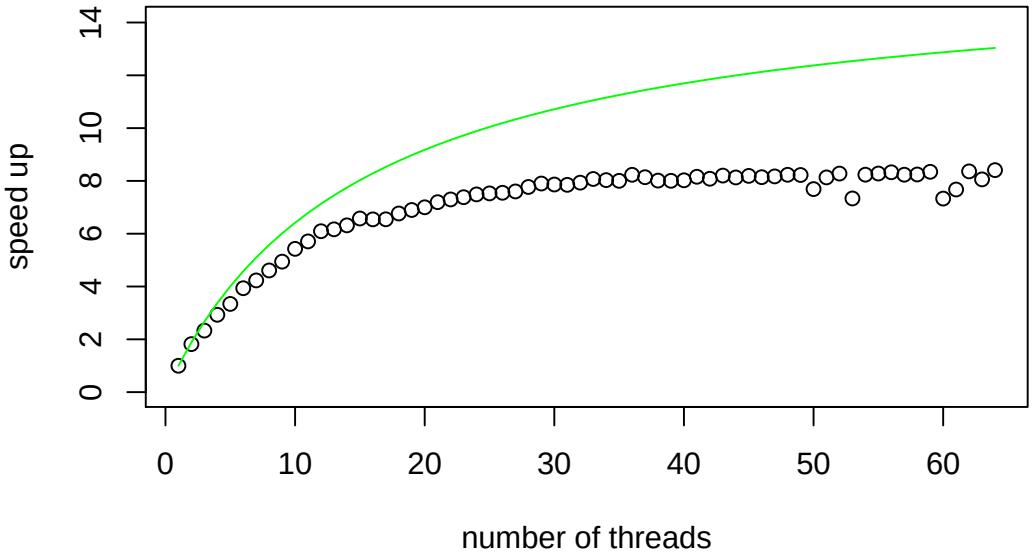
(m) 41 thread (n) 45 thread (o) 49 thread

(p) 53 thread (q) 57 thread (r) 61 thread

Figure 6.10: Barnes-Hut slot time for each number of thread

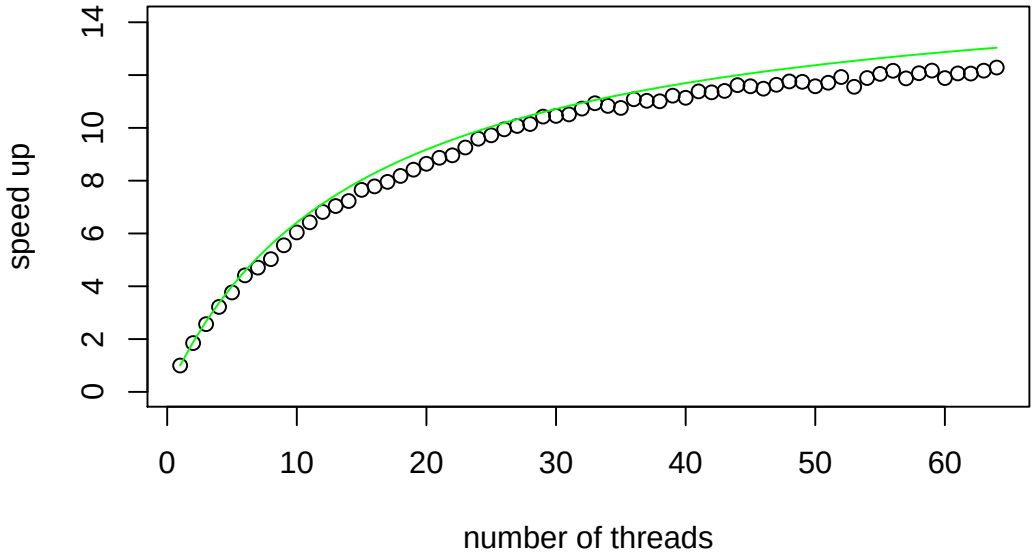
Well, it's seem that the firsts slots are slower starting from 30 threads in parallel. Today we don't find out why. Maybe, there is some cache optimization that can be no more easily done. But, we also see that the value get stabilized. So, we can imagine that the effect is not so dramatic if we really want to use the program, because we often want to simulate for a big number of slot (think about the display of a simulation). Let's try to compute the speedup with only the last 50 slots and see what happen.

Figure 6.11: Barnes-Hut speedup for the last slots



We see that the dramatic behavior is remove, we always get a little bit speedup when we allow more resources. Now, let's simulate that there is no waiting time.

Figure 6.12: Barnes-Hut adapte speedup for last slots



We explain the major part of our results and we can enjoy the fact that we understanding better multi-thread program.

We can ask our self what would be the better way to optimize this implementation now. We can imagine that some particles need more operations that the others because less approximation can be use. For example she is in a dense area of particles. This can lead to have sub-set that take a bigger execution time that the average (the waiting time can be explain in part from this fact). One way to go forward in optimization would be to resolve this. We don't enter in detail here but we can imagine take the execution time for each thread and give some particles of slower thread to the faster one. So we don't have equally sub-set in numbers of particle but in time. With this optimization we could expect smaller waiting time but the Amdhal's prediction would be the same. An other approach would be to try to reduce the fraction f of serial code. This fraction is dominate by the building of the tree. Here we can imagine two ways to handle that. We can avoid the fact of building each time by just updating the tree or try to parallelize the construction. This approach have the advantage to change f and thus change the prediction. We should maybe consider first this approach special if a huge number of cores are available.

Chapter 7

Conclusion

We have see that our practical result is consistent with the theoretical one. So, we can not contest the well foundation of theory. More, seriously, it seems that our code have behavior in accord with the theory.

A interesting fact that we can remark is that often the mind is more powerfull that a lot of machines. We first see that if run the Brute for algorithm on 65536 particles we must have 100 core to compete with the Barnes-Hut on one processor. It's also important to remark that is important to optimize the sequential code before asking more resource. For the little story, we have first not really take care of optimization and parallelize the program. Then we see some optimization and we optimize. We get a speedup of 300 without any added processors. So multi-threading work for increasing the speed of a program but we maybe first consider other solution.

Chapter 8

Context an challenges

This part of the document is more personal and relative to the academical context of the project. This project was presented in EPFL as a semester bachelor project in 2014. I will first thanks, Roger Hersch who have do this possible by supervising the project and lend the sever, and Sergiu Gaman who give me useful advices.

For me, the most challenging part but also the most enjoyable part was to develop the project by my self. I divide the time and the step of the project as I like. I was, also, quite free for the implementation design, the algorithm, and the tools to use. For example, I have decide by my self to make a special object from time measure and bound them with the R software. I also learn the C++ language, trusted name for his speed. I think that I have now, I better idea of how and the time needed to develop a programs.

Bibliography

[Bla] Blaise Barney, Lawrence Livermore National Laboratory. https://computing.llnl.gov/tutorials/openmp/.

[las99] Tieteellinen laskenta. N-body algorithms. Master's thesis, Tancred Lindholm, 1999.

List of Figures

Appendix A

Notations

We will see several algorithm. All of this are object oriented and we have *this* color for object and *that* color for procedure.

We also will use a tree structure and we will use this following object as tree, remark that this tree is dedicated to this particular application:

```
absctract class Tree {
  parent
  Mass
  MassCenter
  center
  length
}

class Leaf extends Tree{
  particle
  Mass = particle.mass
  MassCenter = particle.position
}

class Node extends Tree{
  children1
  children2
  children3
  children4
}

class Nil extends Tree{
  Mass = 0;
  MassCenter = 0;
  center = NULL;
  length = 0;
}
```

We see that a tree is neither a leaf, a node or nil. If it is nil the tree is empty. If it is a leaf,

it have one element. If it is a node it have some child. So We can define a tree from a root node with parent set to nil, then is child is some leaf or node that define a sub-tree. This is write in pseudo code but I hope that you understand the structure.