Distributed and parallel computing

Fabien Labarbe-Desvignes
Nov 10 2019

n-body simulation

Summary

Overview of the project

The goal of this project's to create 3 programs (at least) that have for main goal to parallelize a 2D n-body simulation. Both programs are written in C++ (14), the firist one uses pthread parallelization and the second one use MPI (Message Passing Interface). Finally, in order to visualize the computations I used the graphic library Xlib.

What's a n-body simulation?

In physics and astronomy, an *N*-body simulation is a simulation of a dynamical system of particles, usually under the influence of physical forces, such as gravity (see *n*-body problem). *N*-body simulations are widely used tools in astrophysics, from investigating the dynamics of few-body systems like the Earth-Moon-Sun system to understanding the evolution of the large-scale structure of the universe.[1] In physical cosmology, *N*-body simulations are used to study processes of non-linear structure formation such as galaxy filaments and galaxy halos from the influence of dark matter. Direct *N*-body simulations are used to study the dynamical evolution of star clusters.

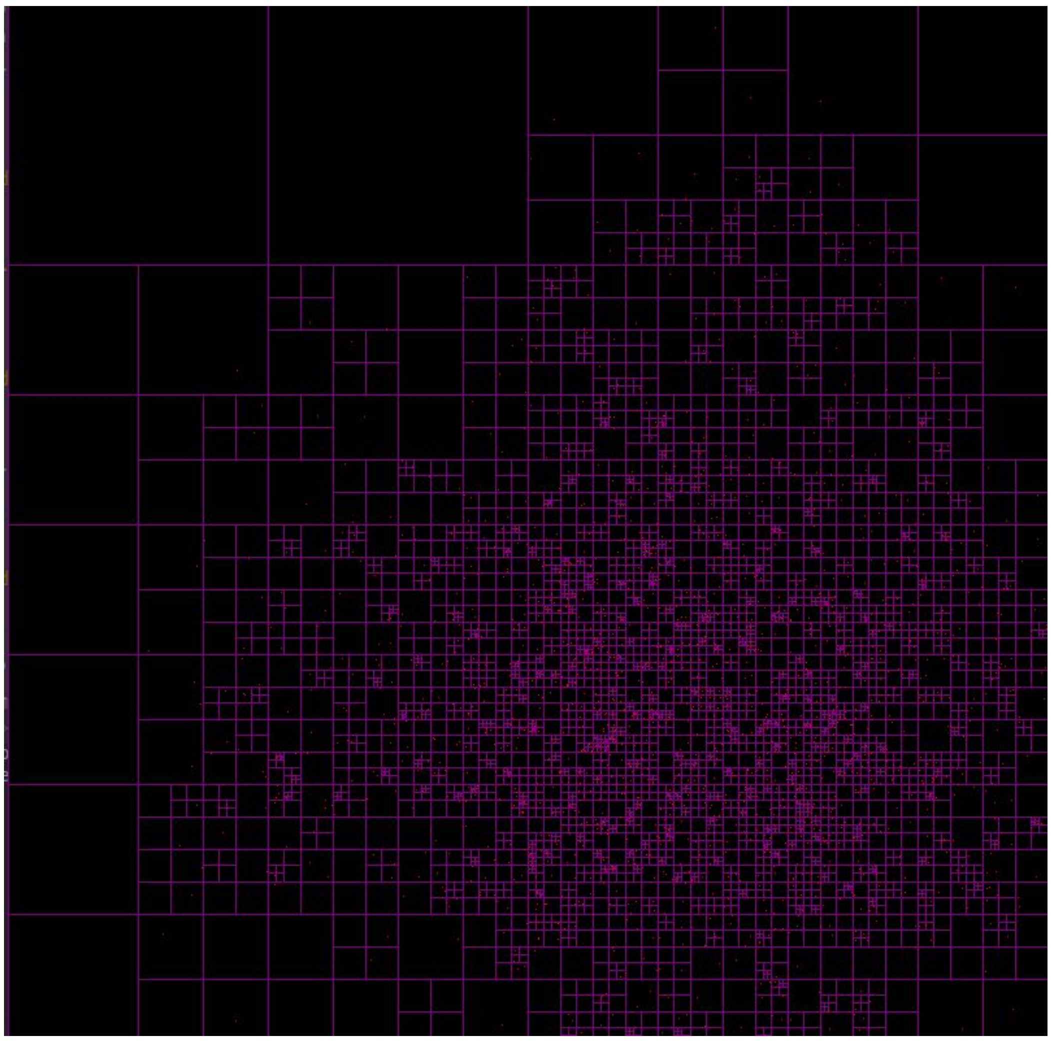


fig1: my n-body simulation using barnes-hut partitioning algorithm

Build & usage

Requirements:

cmake (for compilation), pthread, mpi

How to build?

. go to 119100002/programs pthread, mpi, openmp, mpi+openmp, barnes-hut

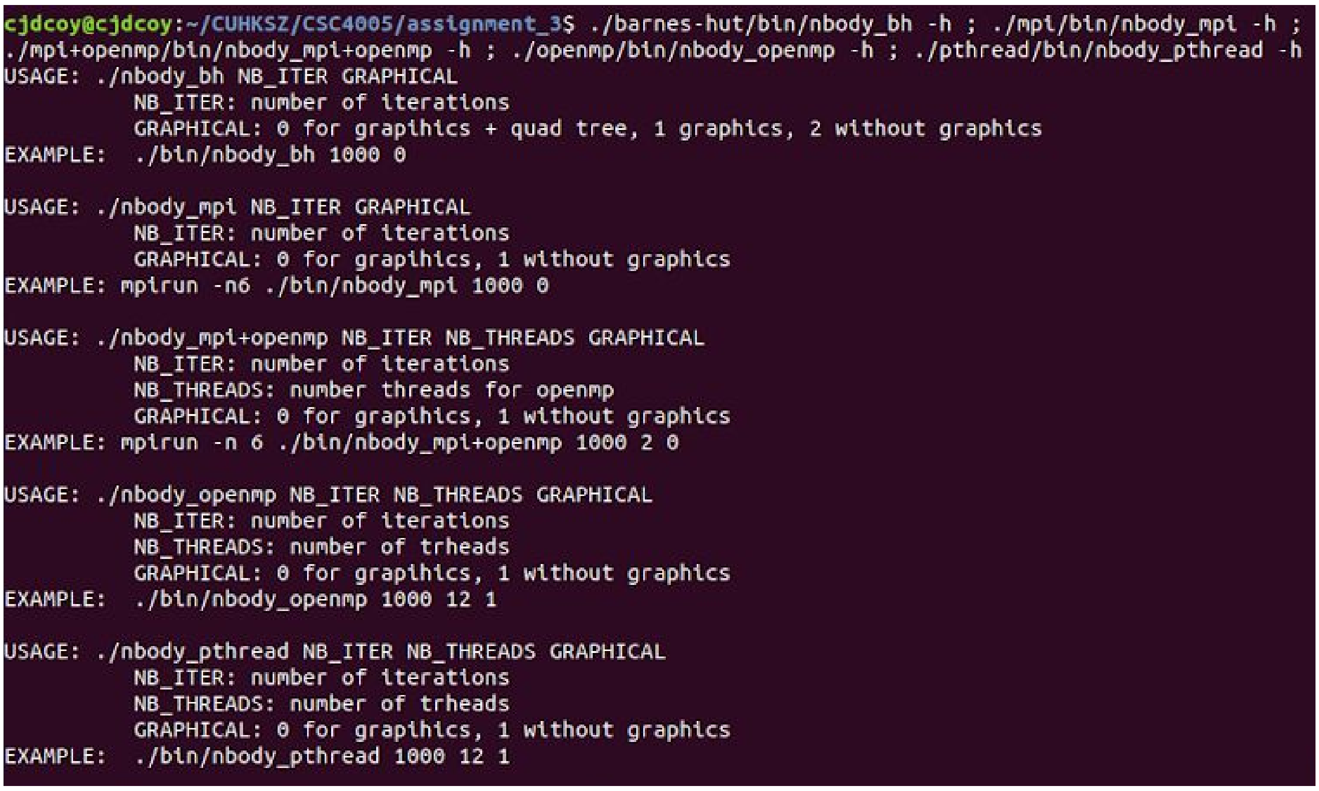
> cd pthread

. create a build folder, go in it,compile the cmake, make and come back

> mkdir build && cd build && cmake .. && make && cd ..

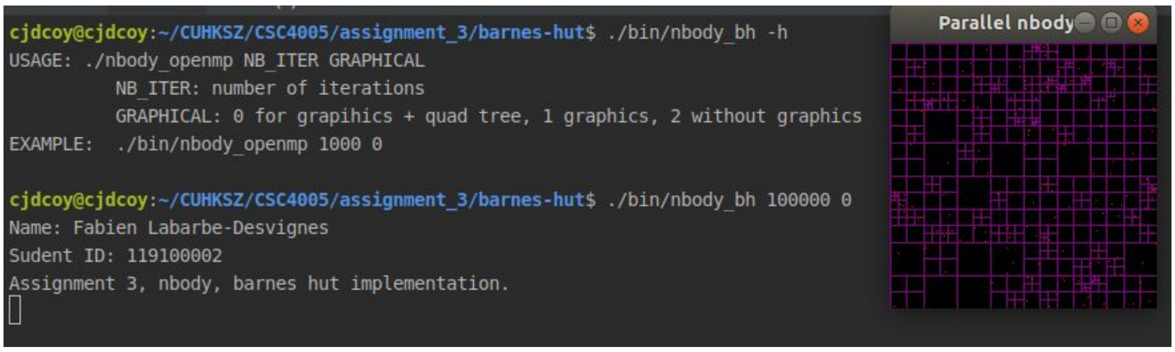
notes: the command line to build is the same for all versions.

Usages



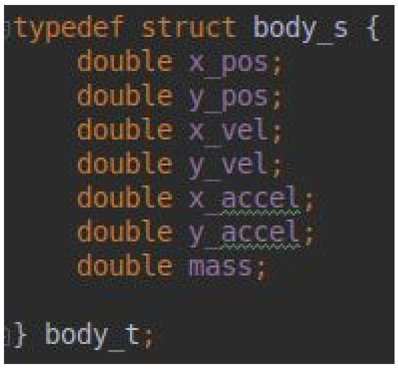
```
cjdcoy@cjdcoy:~/CUHKSZ/CSC4005/assignment_3$ ./barnes-hut/bin/nbody_bh -h ; ./mpi/bin/nbody_mpi -h ;
./mpi+openmp/bin/nbody_mpi+openmp -h ; ./openmp/bin/nbody_openmp -h ; ./pthread/bin/nbody_pthread -h
USAGE: ./nbody_bh NB_ITER GRAPHICAL
          NB_ITER: number of iterations
          GRAPHICAL: 0 for grapihics + quad tree, 1 graphics, 2 without graphics
EXAMPLE:  ./bin/nbody_bh 1000 0

USAGE: ./nbody_mpi NB_ITER GRAPHICAL
          NB_ITER: number of iterations
          GRAPHICAL: 0 for grapihics, 1 without graphics
EXAMPLE: mpirun -n6 ./bin/nbody_mpi 1000 0

USAGE: ./nbody_mpi+openmp NB_ITER NB_THREADS GRAPHICAL
          NB_ITER: number of iterations
          NB_THREADS: number threads for openmp
          GRAPHICAL: 0 for grapihics, 1 without graphics
EXAMPLE: mpirun -n 6 ./bin/nbody_mpi+openmp 1000 2 0

USAGE: ./nbody_openmp NB_ITER NB_THREADS GRAPHICAL
          NB_ITER: number of iterations
          NB_THREADS: number of threads
          GRAPHICAL: 0 for grapihics, 1 without graphics
EXAMPLE:  ./bin/nbody_openmp 1000 12 1

USAGE: ./nbody_pthread NB_ITER NB_THREADS GRAPHICAL
          NB_ITER: number of iterations
          NB_THREADS: number of threads
          GRAPHICAL: 0 for grapihics, 1 without graphics
EXAMPLE:  ./bin/nbody_pthread 1000 12 1
```

Example:



```
cjdcoy@cjdcoy:~/CUHKSZ/CSC4005/assignment_3/barnes-hut$ ./bin/nbody_bh -h
USAGE: ./nbody_openmp NB_ITER GRAPHICAL
          NB_ITER: number of iterations
          GRAPHICAL: 0 for grapihics + quad tree, 1 graphics, 2 without graphics
EXAMPLE:  ./bin/nbody_openmp 1000 0

cjdcoy@cjdcoy:~/CUHKSZ/CSC4005/assignment_3/barnes-hut$ ./bin/nbody_bh 100000 0
Name: Fabien Labarbe-Desvignes
Sudent ID: 119100002
Assignment 3, nbody, barnes hut implementation.
```

Bodies & Partitioning



```
typedef struct body_s {
    double x_pos;
    double y_pos;
    double x_vel;
    double y_vel;
    double x_accel;
    double y_accel;
    double mass;

} body_t;
```

fig2: body structure

Each body was generated with random values for its position and velocity and then added into a vector of bodies.

Since the generations are random I'm using the same partitioning in all 4 parallel methods. The partitioning method I use is pretty simple and straightforward:
I divide the total number of bodies by the number of processes / threads and then I assign a part of the vector of bodies to each thread.

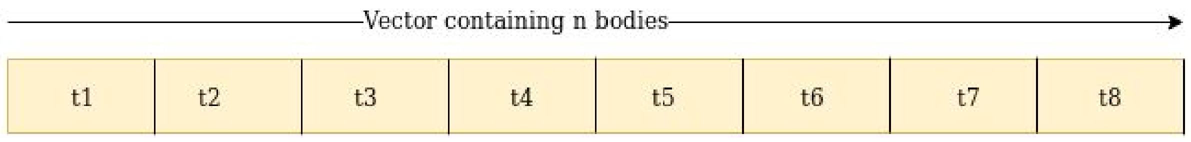


fig2: bodies distribution

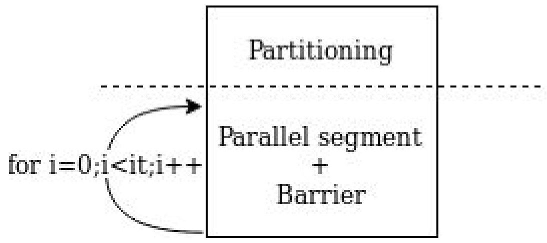
Algorithm design (parallel methods)

My algorithm can be divided in 2 main phases for the pthread and openMP program:

1. Partitioning
2. Processing + display

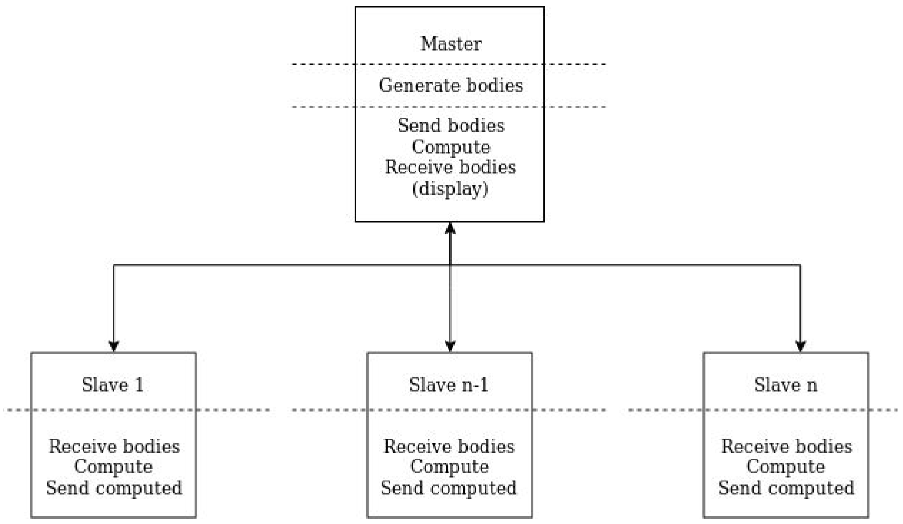
Phase 1:
Generate bodies with random values and add them to a vector in order to handle them more easily and separate the n vectors between the different processes/threads.

Phase 2 (pthread / openmp):
Each thread perform the calculation of its assigned bodies and then wait at a barrier so that the computation ain't rigged by faster threads. The main thread (id 0) display the bodies.



Phase 2 (mpi / mpi+openmp):

- master send bodies (with broadcast for perf) / slaves receive bodies
- master / slave compute new values for their assigned bodies (in the mpi + openmp version the computation is parallelized with n threads)
- mater receive the computed bodies / slaves send only the bodies they computed (for performance gain)

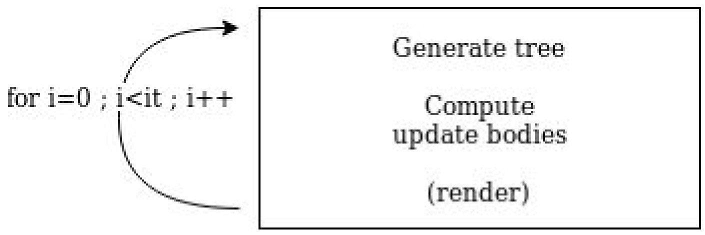


Algorithm design (barnes hut)

In the barnes hut version there's no "partitioning" since I didn't parallelize it.

The step that happen in the loop:

- quadratic tree generation
- computing of the bodies new force/positions
- update of the bodies on the map
- rendering



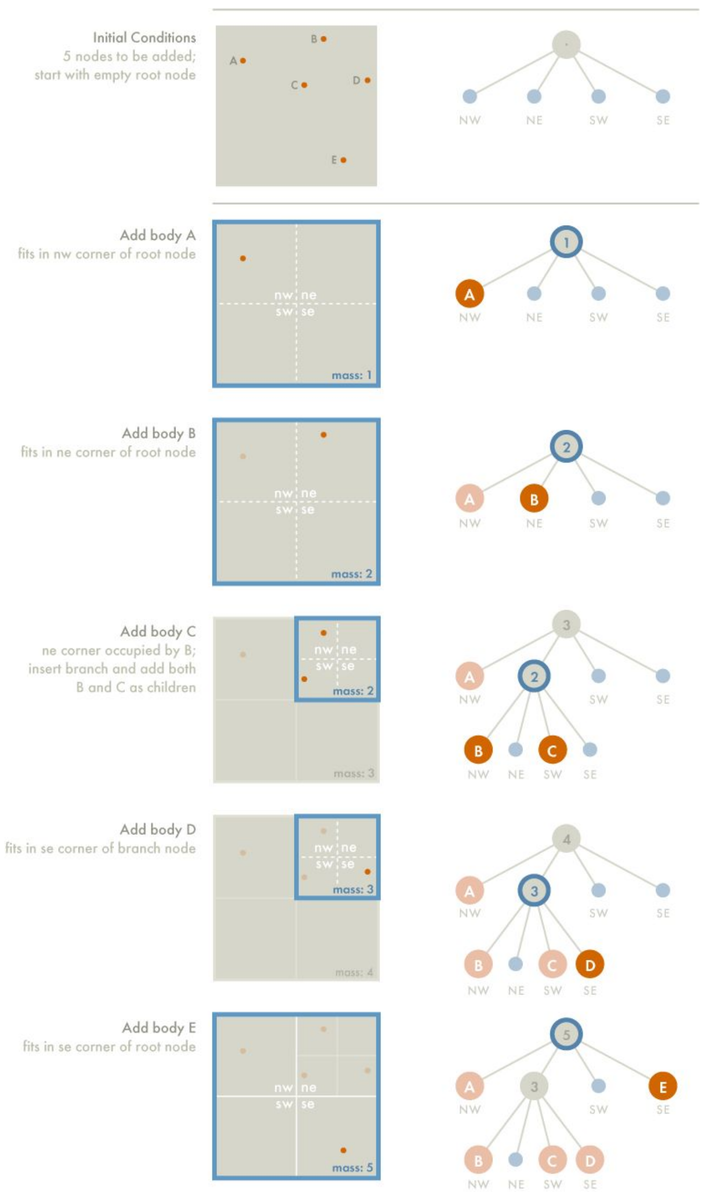


fig3: quadratic tree generation, source: http://arborjs.org/docs/barnes-hut

Obtained results (parallel methods)

Note**:** The following computation have been performed on a i7-9750H CPU (2.60GHz ×12) with an image of size 200*200. The display's not timed in those benchmarks. I consider the 1 thread/process version to be the same as the sequential version.

100 bodies:

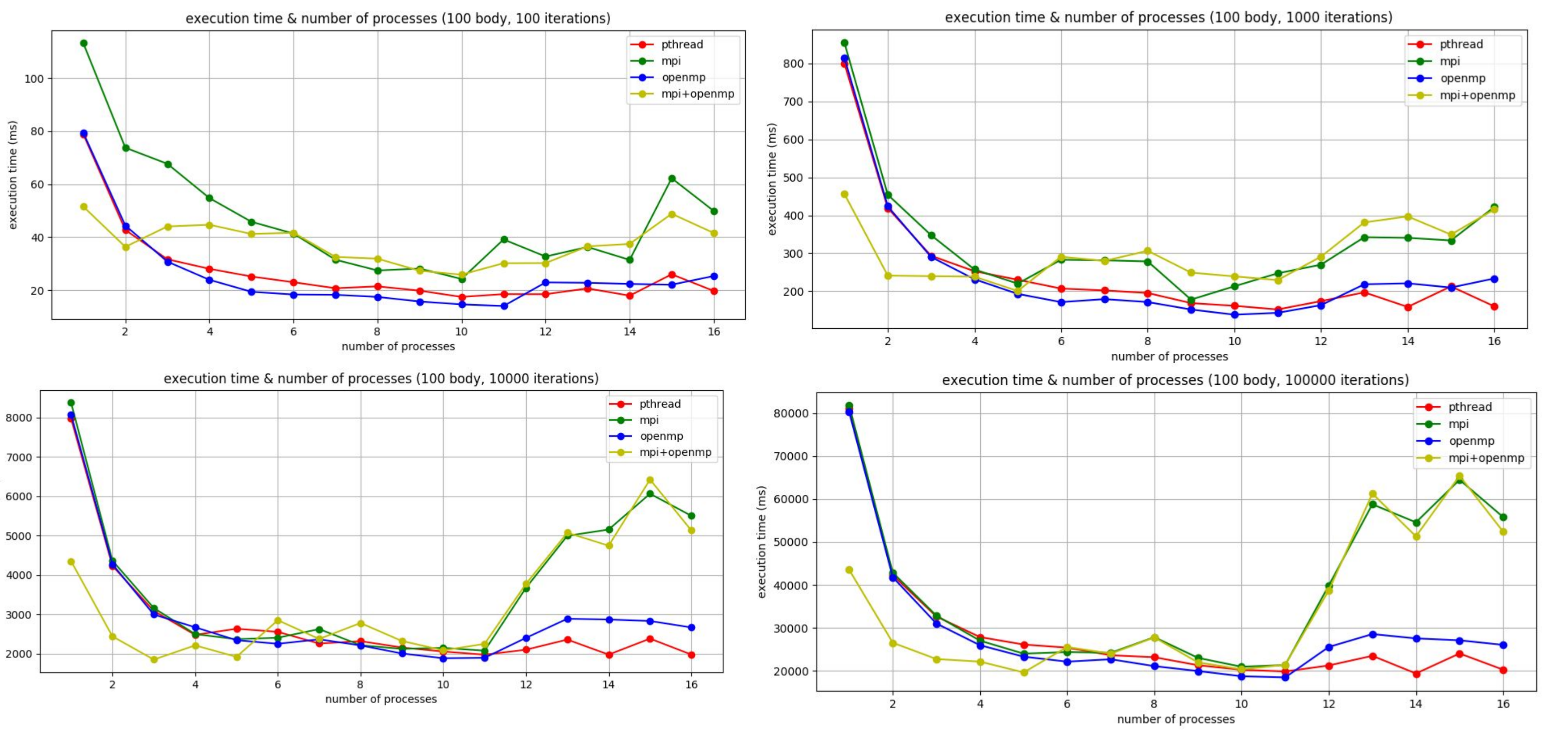


Figure 4 - 100 bodies comparaison

It's good to note that I didn't uploaded any benchmark below 100 iterations because my benchmarks with 10 iterations were sometimes taking below ONE millisecond (around 10ms) which I don't think is relevant for us.

On this version we can see that all the that all versions are faster than the sequential version of the nbody. MPI+OpenMP is faster in the begining because it start off right off the bat with 2 thread on the loop that updates bodies. Pthread and OpenMP are always faster than MPI versions because the more processes the more the MPI versions have to communicate. The slowdown caused by those communications can clearly be seen when the number of process is above 11. We can also see a deterioration in performance for Pthread and OpenMP after 11 threads because both versions implement a barrier which's taking too much time compared to the computation time with only 100 bodies.

We can see that the algorithm is pretty scalable, the runtime is proportional with the number of iteration to perform (800 ms > 8000 ms > 80000 ms) and we can assume that the barrier and communication will decrease in importance as the number of body grows.

1000 bodies:

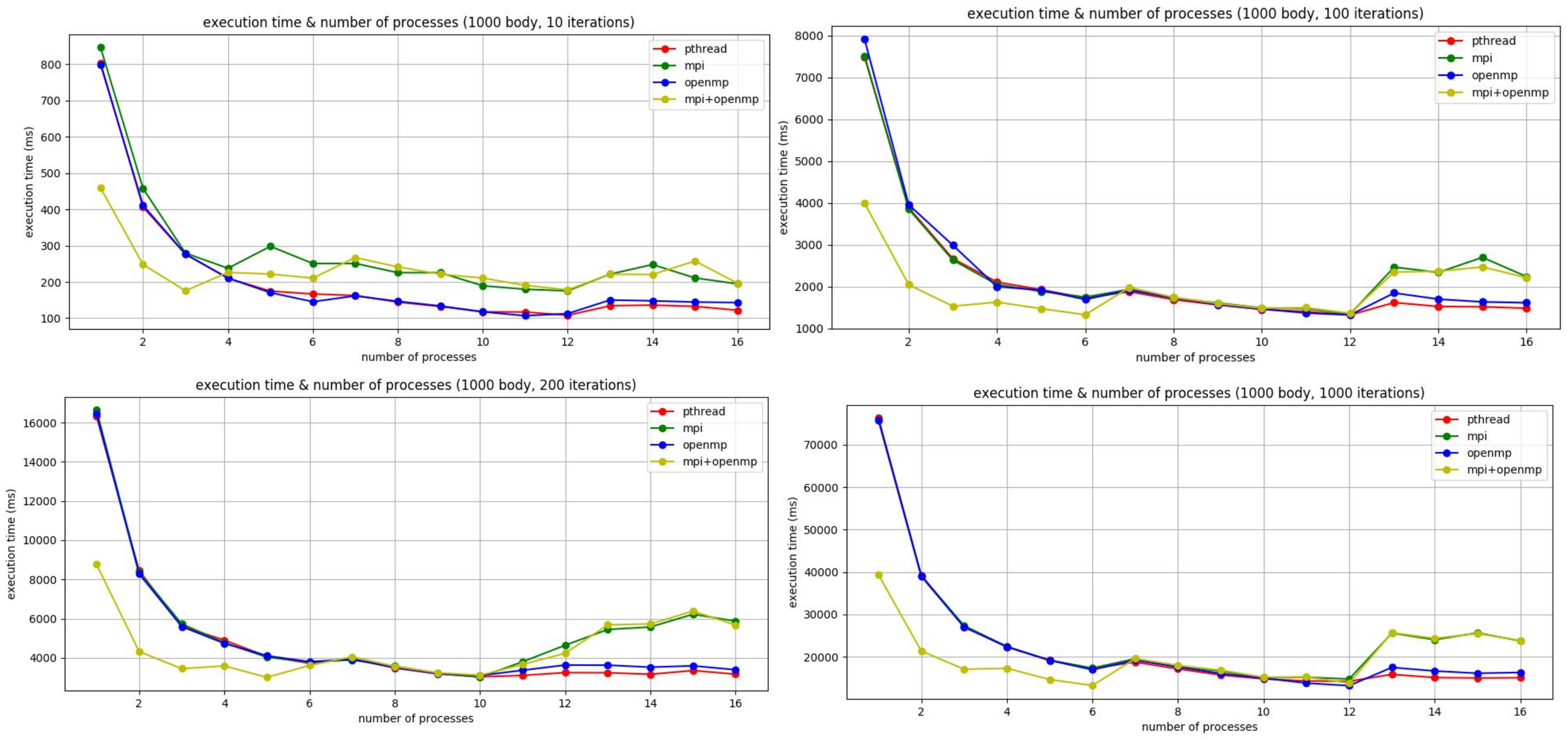


Figure 5- 1000 bodies comparaison

Those results validate the hypothesis I made earlier: the slowdowns provoked by the barrier and communication faded as the number of body grows. Here again, we can see that the program is scalable, the runtime and the number of iterations are proportional. When under 12 cores (my physical limit) the different parallelization methods are taking almost the exact same amount of time which's what I expected to happen.

Why are MPI versions Slower after 12 cores ?

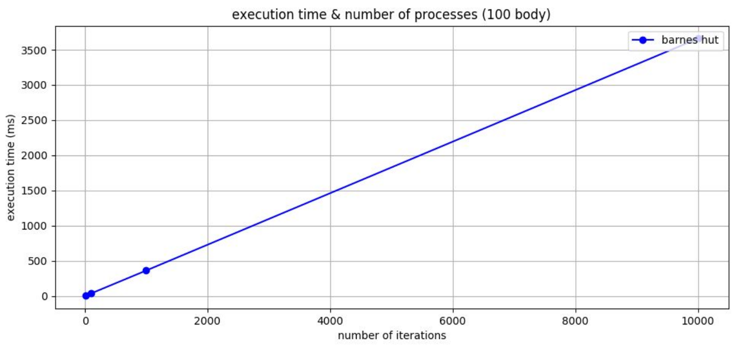
The different algorithms are pretty much the same except for the MPI implementations which are slower the number of threads grow over the number of cores on my machine (12 cores). That it's because MPI uses processes and not threads so when my computer does the load balancing between the processes it allow more resources to some processes than others meanwhile when threading the computing resources are distributed evenly.

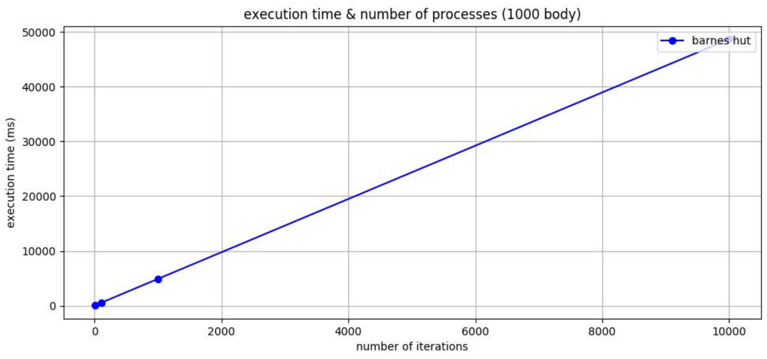
Why is MPI+OpenMP faster at first ?

MPI+OpenMP is always faster at first because it simulate a program running with N process and 2 threads. Of course in the beginning it has a huge lead on other programs as it has twice the number of threads working for it compared to the other programs but after 6 cores it gets limited by the performances of my computer. In an environment of 4 computers with 12 cores each available for example it would be much more efficient as I could push and use each pcs to their best.

Obtained results & analysis(barnes hut)

Note**:** The following computation have been performed on a i7-9750H CPU (2.60GHz ×12) with an image of size 200*200. The display's not timed in those benchmarks. I didn't implement parallelization for this method which means that it runs on a single core.



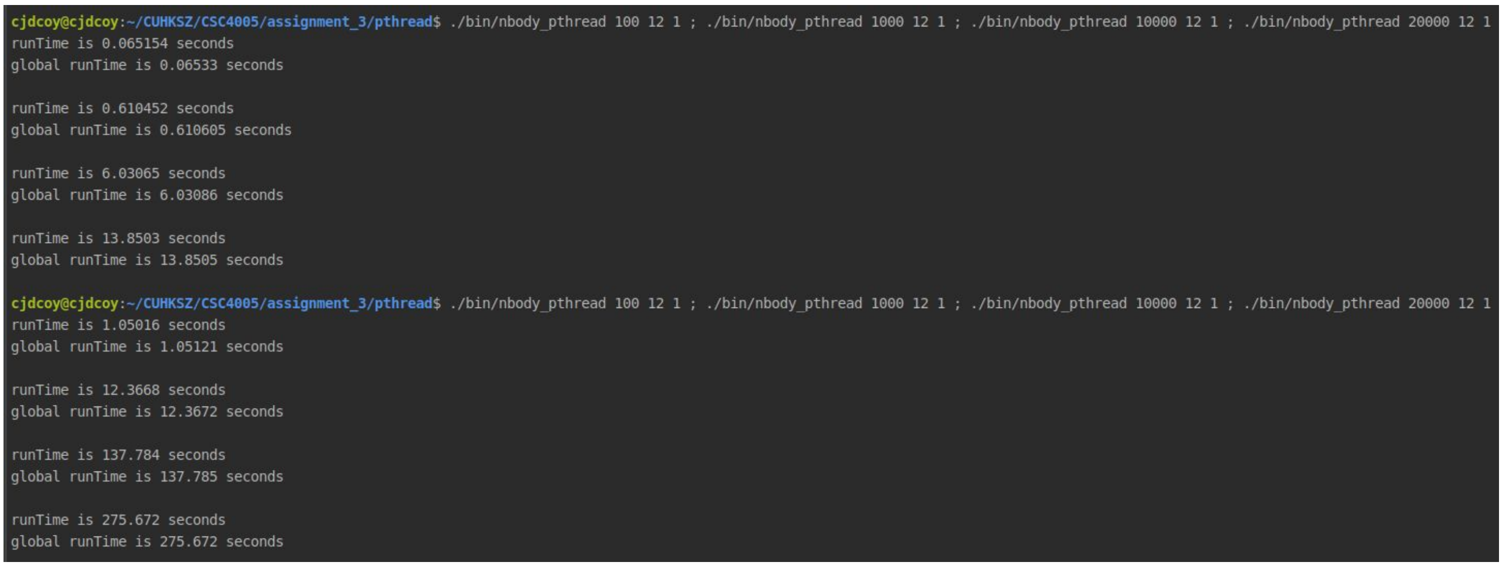


We can see that the barnes hut algorithm is much more faster than all of the different parallelization methods as It goes from O(n^2) to O(n Log n).

For 1000 bodies, 1000 iterations we're taking 6000 ms with barnes hut on 1 core meanwhile pthread was taking 17000ms with 16 cores.

This solution might appear like a 'cheat code' but the gain on time has a tradeoff: you lose in accuracy. Indeed, the less the quadratic tree is deep the less accurate the simulation is and likewise. If you want a very accurate simulation with this method you'll lose in computation time and have a result that look like the other parallelization methods.

nbody runtime vs global runtime



```
cjdcoy@cjdcoy:~/CUHKSZ/CSC4005/assignment_3/pthread$ ./bin/nbody_pthread 100 12 1 ; ./bin/nbody_pthread 1000 12 1 ; ./bin/nbody_pthread 10000 12 1 ; ./bin/nbody_pthread 20000 12 1
runTime is 0.065154 seconds
global runTime is 0.06533 seconds

runTime is 0.610452 seconds
global runTime is 0.610605 seconds

runTime is 6.03065 seconds
global runTime is 6.03086 seconds

runTime is 13.8503 seconds
global runTime is 13.8505 seconds

cjdcoy@cjdcoy:~/CUHKSZ/CSC4005/assignment_3/pthread$ ./bin/nbody_pthread 100 12 1 ; ./bin/nbody_pthread 1000 12 1 ; ./bin/nbody_pthread 10000 12 1 ; ./bin/nbody_pthread 20000 12 1
runTime is 1.05016 seconds
global runTime is 1.05121 seconds

runTime is 12.3668 seconds
global runTime is 12.3672 seconds

runTime is 137.784 seconds
global runTime is 137.785 seconds

runTime is 275.672 seconds
global runTime is 275.672 seconds
```

Figure 6 - 100 to 20 000 iterations, 200 bodies, 1000 bodies

I’m computing the global and nbody runtimes in order to have a better idea of the percentage of the non-parallelizable features in my program.

Since Pthread was the most consistent and fastest version overall I decided to measure the global runtime against the nbody runtime with it. It’s good to note that the initializations are pretty much all the same for the different programs, only the parallelized parts differs.

We can see that the non-parallelized part of the program takes about 100us whatever the number of bodies.

Performance analysis

I'm using the same speedup laws on all the algorithm version because the non-parallelized part is the same for all methods.

Using Amdahl's Law

we know that $Speedup = \frac{1}{(1-p)+p/N}$

We also know that the non scalable part is constant O(1) and takes around 100us so let's assume that the program is 99.9% parallelizable.
Maximum Speedup for the Pthread version on 12 CPUs:

$$Speedup = 1/((1-0.9998)+0.999/12)$$
$$Speedup = 11,983$$

Using Gustafson Law

we know that $S = N + (1-N)s$

$$Speedup = 12+(1-12)(0.001)$$
$$Speedup = 11,989$$

The results from both laws are definitely good and VERY close and that it is worth to parallelize the nbody simulation. Keep in mind that both laws only gives a very rough estimate. Indeed, those methods do ignore **parallelization overhead** and as you know there are barriers in the openmp / pthread version and message passing with the MPI versions.

Parallelization overhead is the amount of time required to coordinate parallel tasks, as opposed to do useful work (such as time to start a task, synchronization time).

Conclusion

During this project the nbody simulation was successfully implemented using MPI pthread, openmp and barnes hut. The performance analysis shows that all the version have decent performance upgrade on the sequential and that they also have a pretty similar runtime which is expected since the program's highly (99.9%) parallelized. It also shows off that you can sometimes decide to aggregate the data which leads to a loss in accuracy but a huge gain in performance.

I learnt a lot of things doing this project.
First, I learnt that if you want to really parallelize a program one of the best ways is to use MPI between different computers and parallelize the different sections on each computer to maximize the parallelization effects. Also, during the last project I didn't use openmp and I think that it's a good thing that I used it here because it allows to create a much more compact and simple code than thread while having the same performance effect in some cases. Finally, it made me discover the parallelization overhead problem that prevent Gustafson & amdahl's law to be accurate, I also read research papers talking about the Karp-Flatt Metric that allows to reduce those errors.