HPCA2

Arthur Picault & Brieuc Horard

20th April 2020

GitHub: https://github.com/AeroPyk/HPCA2

1 Exercise 1: Hello World

1.1 Output

```
Amount of thread at THIS time: 1
Amount of thread NOW: 4
hello(2) world(2)
Amount of thread NOW: 4
hello(3) world(3)
Amount of thread NOW: 4
hello(1) world(1)
Amount of thread NOW: 4
hello(0) world(0)
Amount of thread at the end: 1
```

I just added an instruction to display how many thread were active at the time. We can check that in the parallel section we have 4 (my laptop having 2 cores but 4 logical processors) threads and only one outside.

1.2 Compilation

On my own laptop I use CLion wich use CMake files to run. However this is pretty much the same. As I stubborn enough to stick to windows, I installed tdm64-gcc-9.2.0 compiler specifying openMP installation. Then in my IDE I only had to put it as default compiler and add the *-fopenmp* flag in the CMake file. It looks like:

```
cmake_minimum_required(VERSION 3.15)
project(A2 C)

set(CMAKE_C_FLAGS -fopenmp)
set(CMAKE_C_STANDARD 99)

add_executable(A2 main.c)
```

1.3 On Beskow

Depending on the copiler we want to use (specially between gnu and Cray) we have, we use the flag *-fopenmp* with gcc and *-openmp* with Cray.

1.4 Change the number of thread

There are 2 ways to do so. The first one that comes to mind is to use the library and call the function:

omp_set_num_threads(**Number of thread**)

The second method is to use the environmental variable:

OMP_NUM_THREADS **Number of thread**

1.5 Without any openMP flags

If we dare omitting the flag, the compiler yell at us a nice error mostly saying "undefined reference to omp_get_num_threads".

2 Exercise 2: STREAM

2.1 Plots

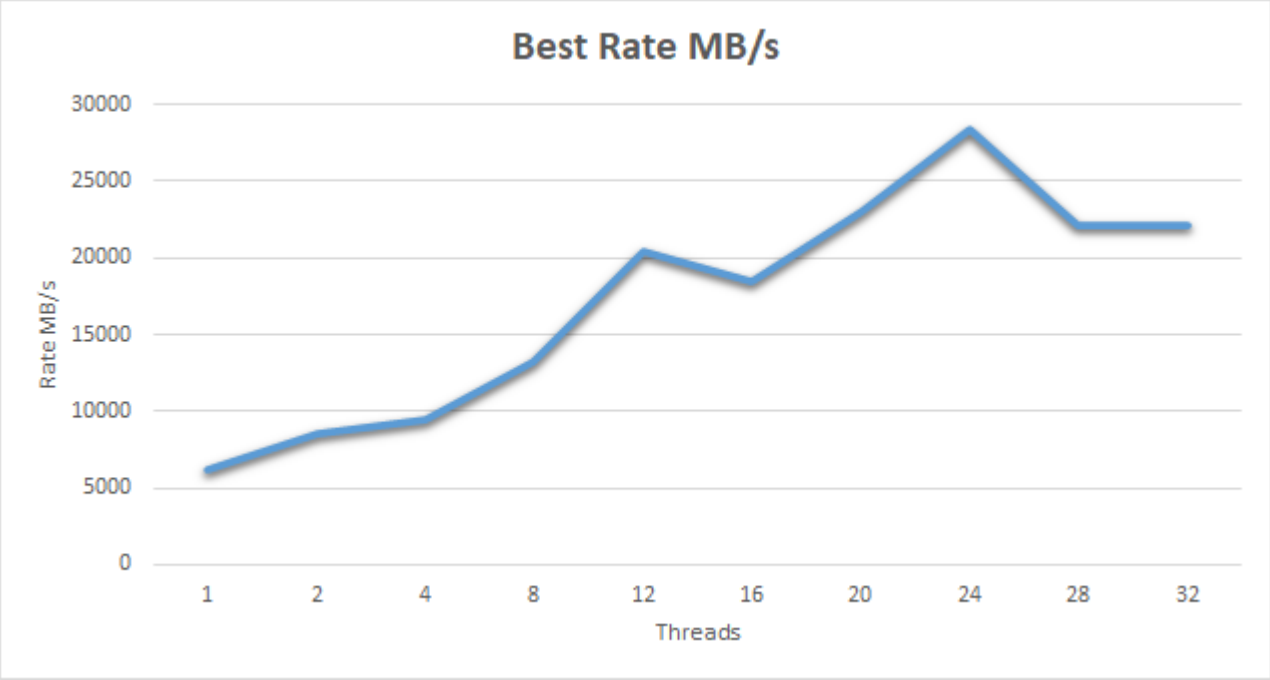

Figure 1: Best rates

Figure 2: Time transfer

2.2 Bandwidth regarding the number of thread

We can appreciate these curves especially on the graph showing the time transfer as it steadily decrease but ending in a form of a tray. Particularly on the Min time curve we confirm the theory of the hardware limitation that we reach. As more threads are used, the total bandwidth is used and we couldn't perform better even with a thousand thread.

2.3 Schedules

I am convinced that despite the overhead, the dynamic schedule is the fastest since it allows the exploitation of all threads at best.

To set the schedule to something else than static (default) we need to add the information through *schedule(kind[, chunk-size])* in a pragma statement. Possible choices are *static, dynamic, guided*.

In stream.c file: ctrl+f and "schedule". Line 314 we can change the schedule.

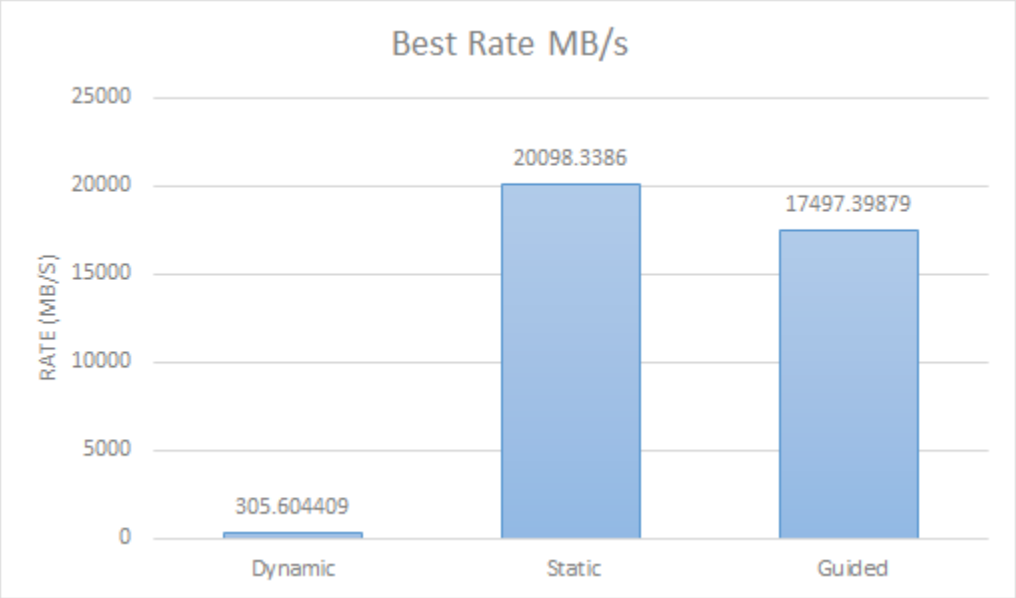

Figure 3: Best rates

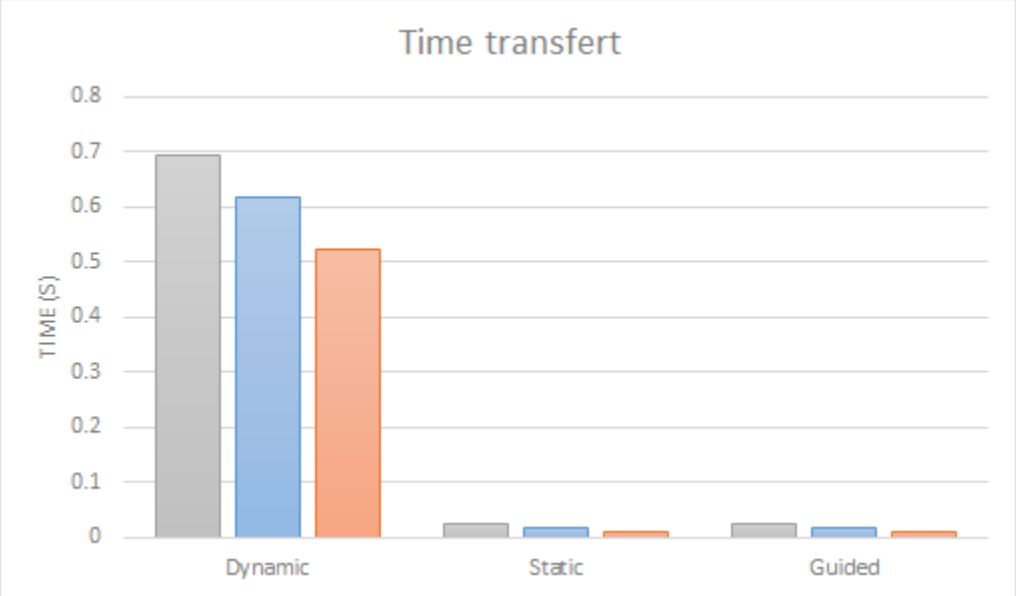

Figure 4: Time transfer

So my expectations were proved wrong since the dynamic version is by far the worse. But after a second thought, makes sense because the over head with 32 threads is very high.

3 Exercise 3: MaxLoc etc...

The program written is such as it test all the methods with X threads once an array has been generated, this mainly to check if the method ends on a wrong result compare to the serial method. Several pass are performed to get average and standard deviation.

3.1 Serial Code

To record the performance we just loop over the initialization of and the search for the maximum. We get:

Avg (s)	Min (s)	Max (s)
0.002088	0.002026	0.002107

3.2 OMP Parallel

We expect this to mess things up since the threads could write the same variable at the same time and no atomic action are used. However on a small amount of threads it seems that it's unlikely that it happens.

3.3 OMP Critical

An improvement for the issue of this one is to test the condition if the value we test is higher or not than the maximum, before trying the critical section. This way we save a lot of time since if the value is lower than the maximum even if it is updated it will never make the current value higher than the maximum.

3.3.1 Classic

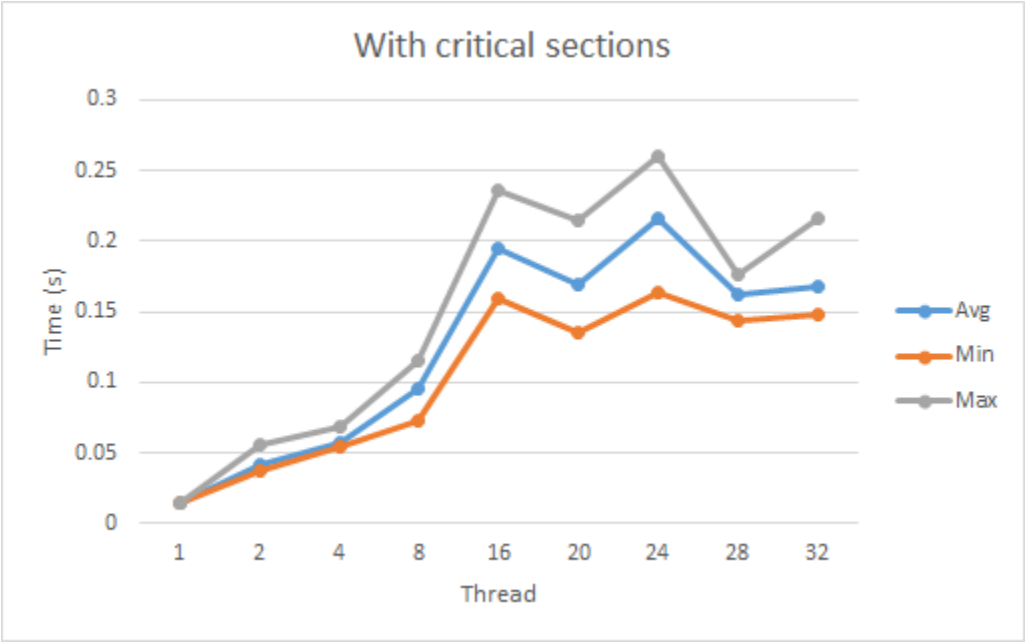

Figure 5: Critical section improved

On this version we can clearly see that critical sections are time consuming because they block all the threads (+ overhead) and slow down the whole search.

3.3.2 Improved

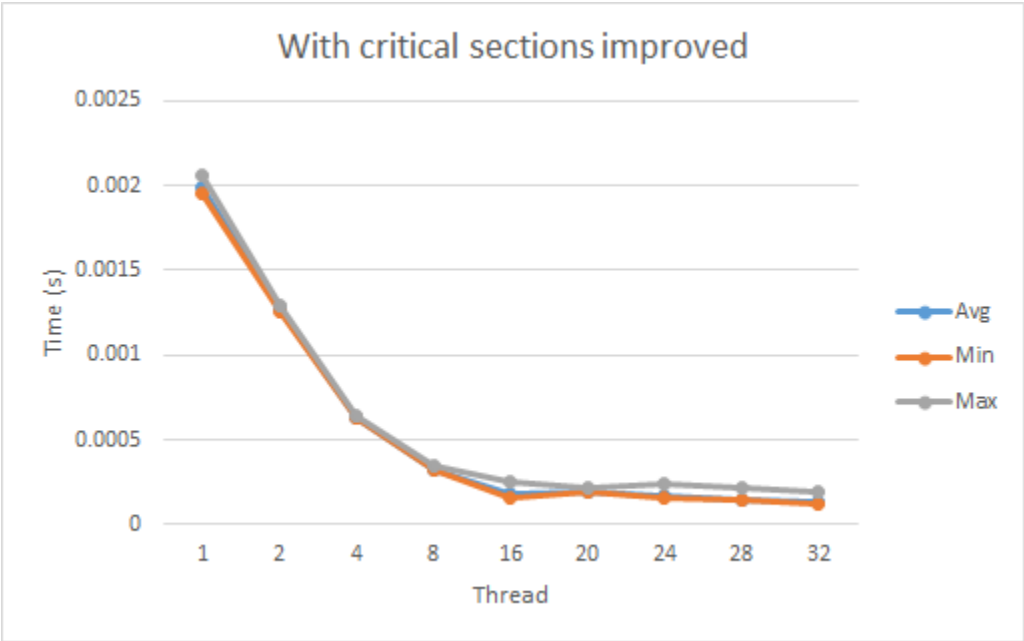

Figure 6: Critical section improved

With such thing we get close to the next version where we avoid the critical sections.

3.4 Avoid critical

The main idea here to use a variable for each thread gathered at first in an array. A thread being the only one writing in "its" variable shouldn't get into any troubles since there will be one writer at the time.

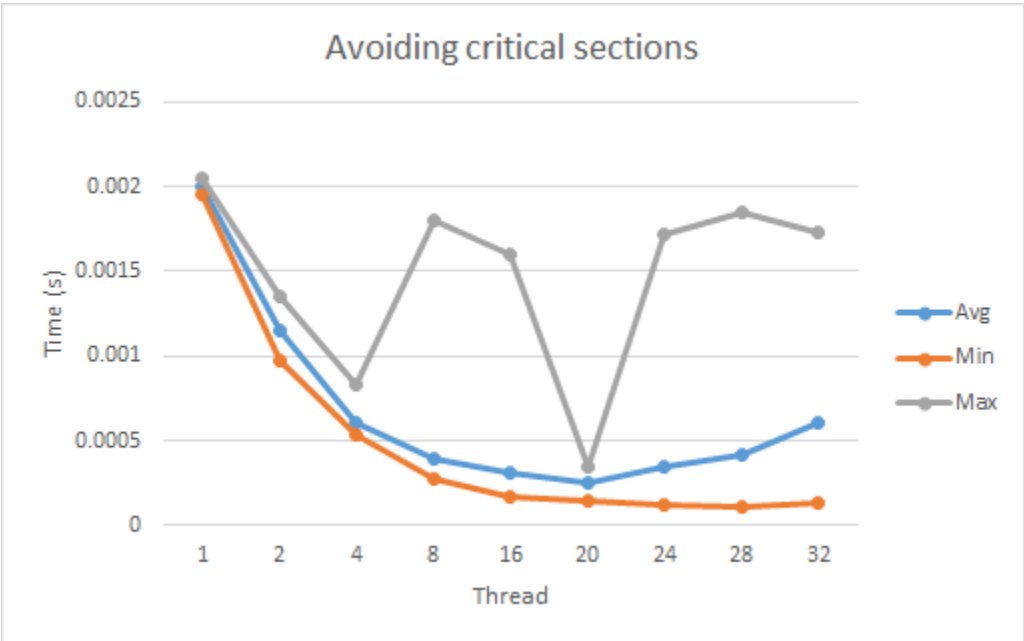

Figure 7: Avoiding critical sections

3.5 Removing false sharing

We can use either padding or local storage to avoid false sharing. We can pad either with a structure as shown in the lecture or use bigger array and say storing values every 100 indices instead of consecutive ones.

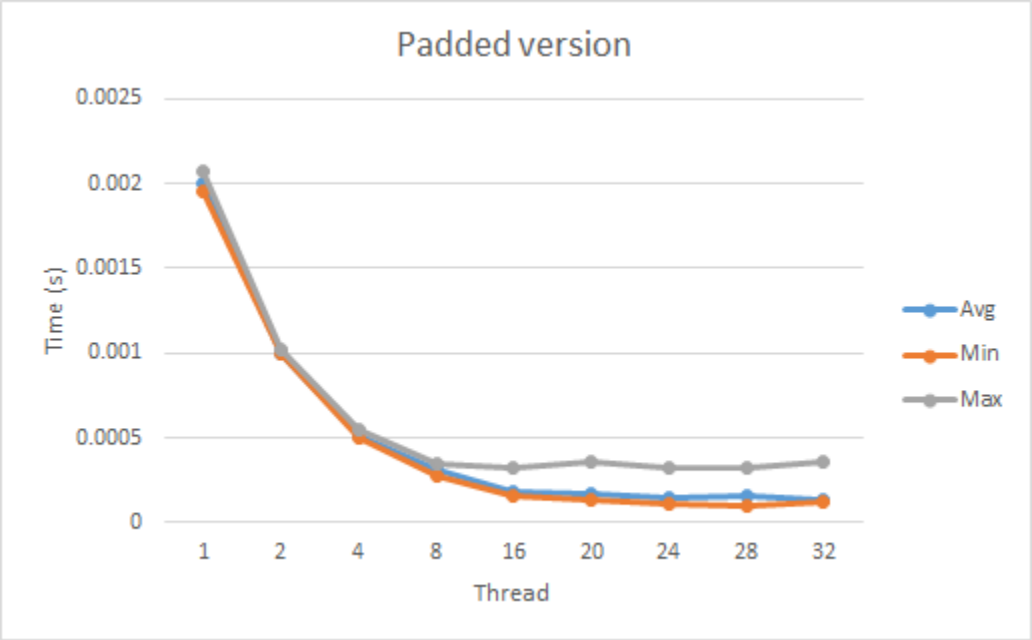

Figure 8: Using padding to avoid false sharing

Though we cannot perfectly see the improvement, there is one. Because of the fact that the padding prevents from cache misses, we get better performances.

4 Exercise 4: FFTW

In order to parallelize the DFTW with OpenMP we put *#pragma omp parallel for* before the first for loop. We also tried to put the #pragma statement before the second for loop but the result is roughly the same, so for readability reasons we put it before the first one.

Just this statement were enough to improve the program and divided the time execution by 10. The code is still correct since the error could occur if the variable used for the result (X?_o[k]) changed of thread during computation, creating a collision. But since, at this part, the inner loop write at k^{th} value of a table and none of previous values are used to calculated the new one. In other words, all calculations are independent which let us perform them in separate threads.

We found a way of improving the program by saving the value of cos and sin instead of compute them each time we need them.

For the serial time the average is 9,0503189 seconds with a standard deviation of 0,109119736.

For the run time with 32 threads, with improvement, the average is 0,37648914 seconds with a standard deviation of 0,000805193.

An other improvement we thought about was to switch the loops to improve the use of

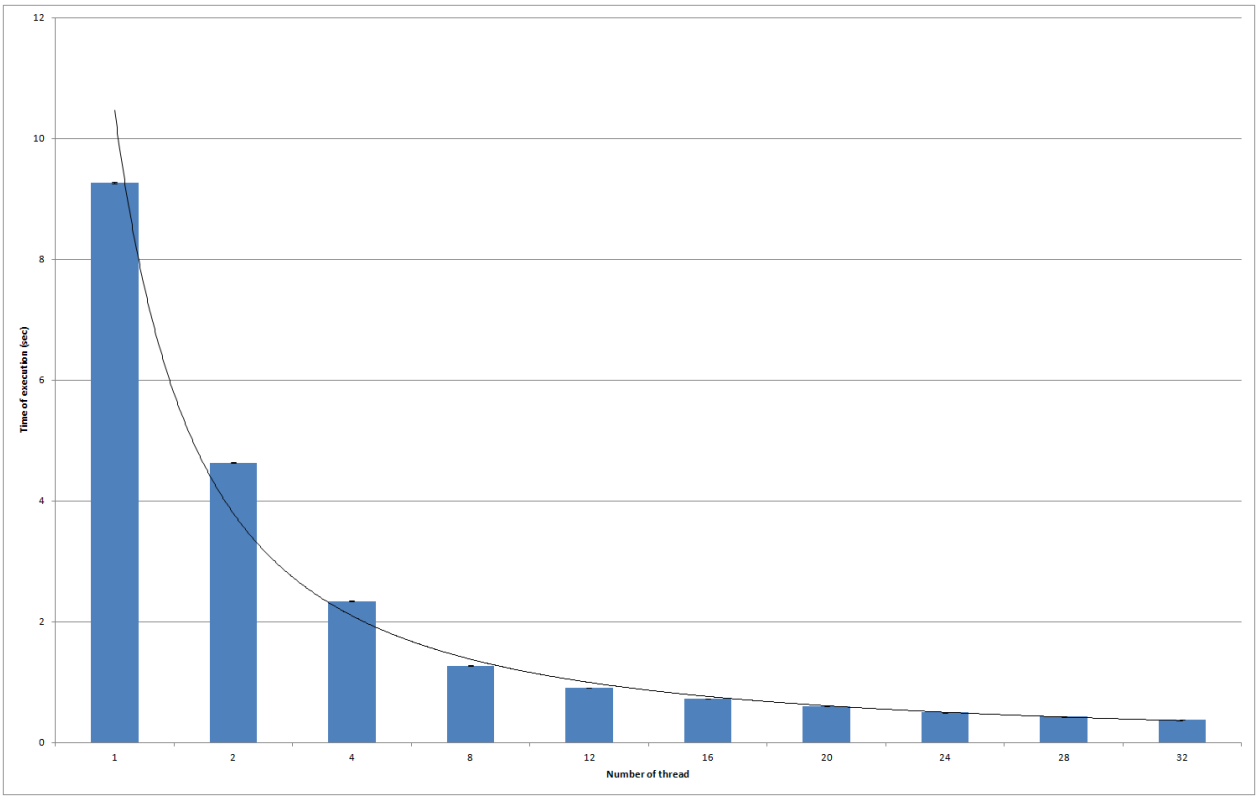

Figure 9: Measure of time execution depending on the number of threads without omp critial

the cache. But by doing so, we will have errors in the results, so the technique must be improved before real implementation.

5 Exercise 5: N-body simulator

5.1 Serial

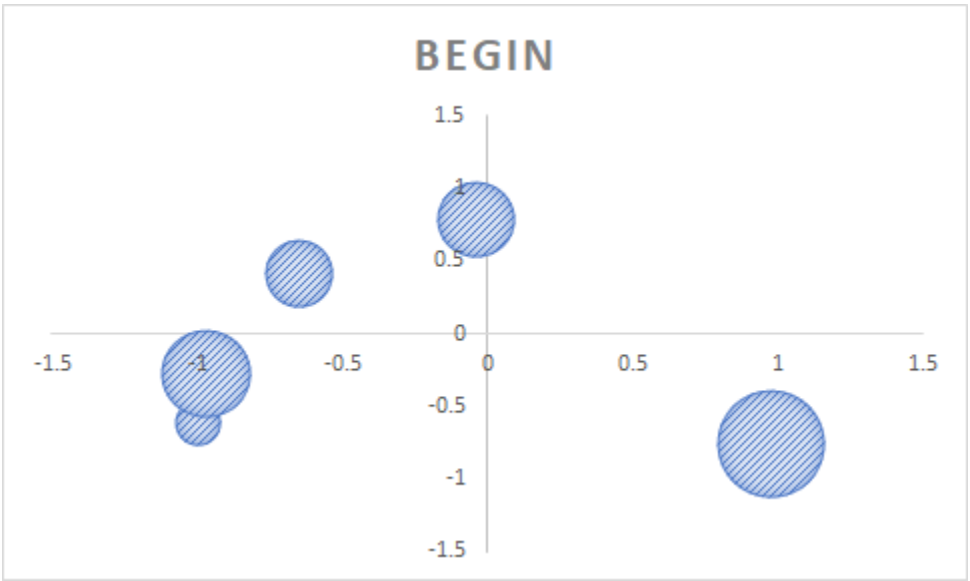

Figure 10: 5 bodies in initial states

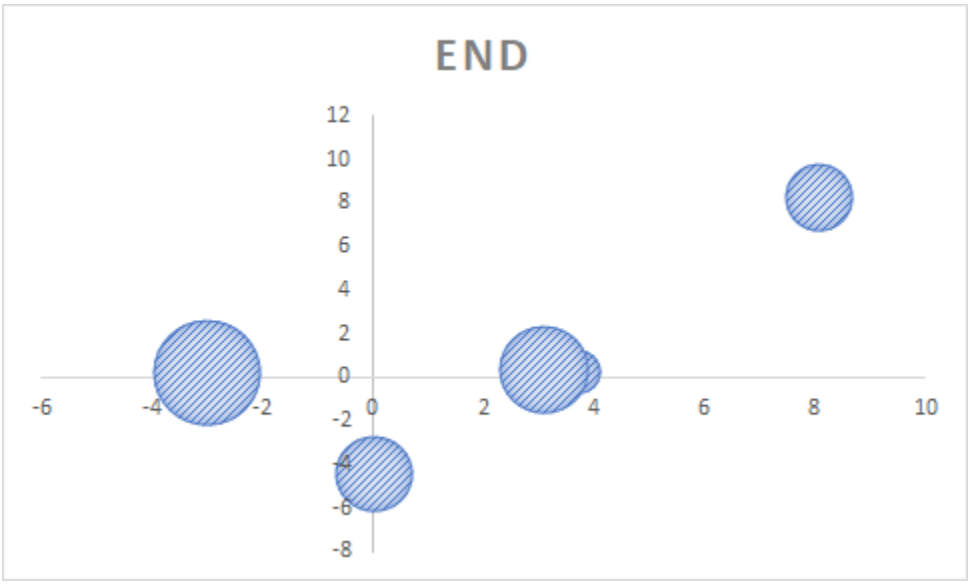

Figure 11: 5 bodies after a few cycles

To implement the problem, first of all G has been modified to a higher value. It was either that or having way bigger masses. This because otherwise we would a crazy amount of cycle to get to see moving the bodies. Here we are concerned by the optimization so that's not big deal to get another value of G.

Second of all, I wanted to plot as well in 3D with velocity vector and force vector. However my knowledge on gnu-plot isn't sufficient and the main problem here is, once again, optimization. We will have to trust the calculations.

On my laptop I have the following results (2D):

Particles	500	1000	2000
Simple	3.302s	10.518s	35.71s
Reduced	1.514s	5.7s	19.576s

The reduced algorithm is the fastest, indeed because for the calculation of one Force it reuse the value already in cache to compute a second force.

On Beskow:

Particles	500	1000	2000
Simple	1.493991s	5.862827s	23.360460s
Reduced	0.759668s	3.031051s	12.087909s

Beskow is globally twice as much faster than my compute (I would expect nothing less).

5.2 Parallel

First of all, we can't do the same as the previous exercise and say calculations are independent since it is actually not. Indeed, each body is moving influenced by others. Therefore we can't just divide the calculations according to batch of bodies. We have to be finer than that.

Simple first thing, we can parallelize the initializations at the beginning of both algorithm since no calculation are related. This on my computer with N=500, made me save a solid 0.8s and 0.4s on each version (simple and reduced), 4 threads used (with 4 logical processors). Quick calcul: each thread saved respectively 0.2 and 0.1s.

From there we're going toward different optimization for both function since one is writing 2 different spaces in forces which could lead to interference between threads.

5.2.1 Simple Algorithm

Here this algorithm have a big advantage of not accessing twice the forces array. Therefore we can simply parallelize the simpleAlgo function in function sA, parallelizing the loop over q.

Because inside the loop only forces[q][d] is accessed, we can safely separate the task in different threads which will care of distinct value of q.

However this would be a really bad idea to directly parallelize the inner loop over k since it could let 2 threads write at the same time the same forces[q][d].

5.2.2 Reduced Algorithm

The strength of this algorithm makes it weak. Indeed, we wanted to gain time by using a computation for 2 results however this implies to write 2 location in forces array and therefore prevent from threading this part because the forces[k][d] could be also accessed in another thread.

However we might be able to compute forces[k][d] locally and sum up every part at the end!

After some experimenting, 2 local array were necessary. Comment in the code, but in short: I thought only the forces[k][d] was a threat (therefore in critical section) however, it happens that a thread is slower enough to still be on computation of forces[q][d] and then conflict with another who would be in the critical region.

To solve that it's enough to use 2 local array and then copy everything in the critical region.

5.2.3 Proof of correctness

The program runs both version of the algorithm and at the end the *doubleCheck()* function compare both final position and print an error message if a position is different from another version.

Moreover in a test period, we compared serial and parallel program giving the same result for the same initialization (same srand()).

Many comments toward the argument of the optimization have also been added in the code.

One more thing, the value of G has been increased from 6.6742e-11 to 6.6742e-5 because otherwise it wouldn't had any effect on the simulation. However a too high value would lead to slight divergence between both models.

5.2.4 On my laptop

Particles	500	1000	2000
Simple	1.236s	5.067s	17.093s
Reduced	0.787s	3.401s	11.893s

With my laptop I can already see improvements by a factor of 2 (4 threads).

5.2.5 On Beskow

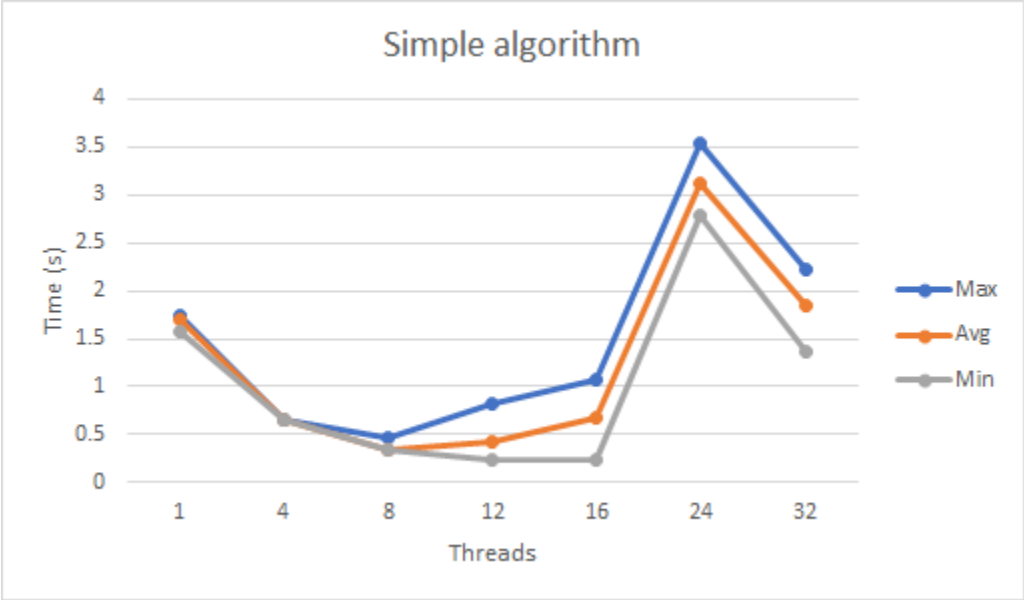

Figure 12: Algorithm simple

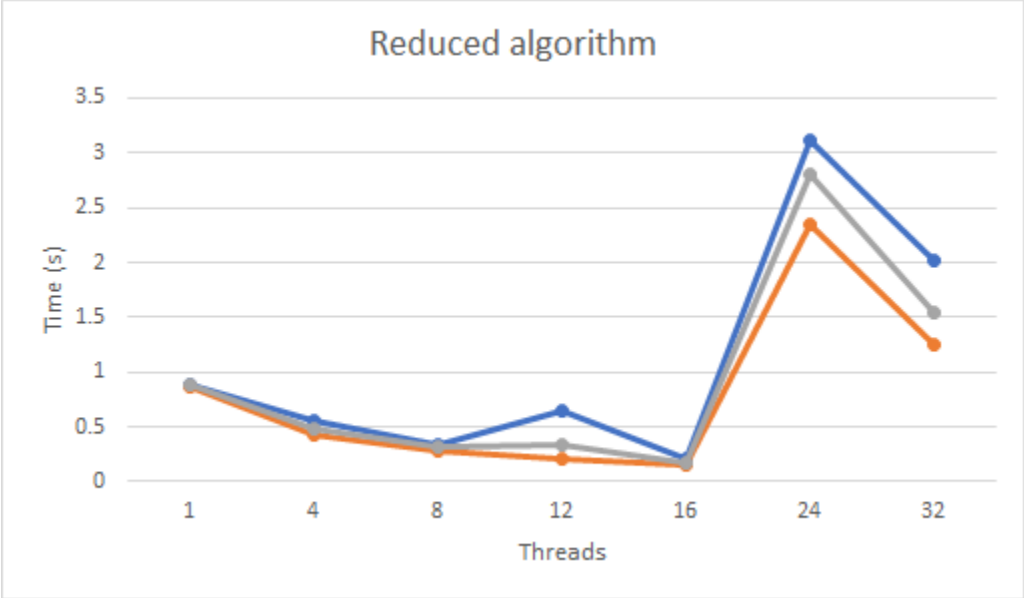

Figure 13: Algorithm Reduced

For N=500 performances are even better. However with high number of threads we can see that it struggles getting better performances and instead takes a while more probably due to an overhead from the high number of thread.

5.3 Schedule

Could we have some details about this last question ? All I can recall about schedule are static, guided and dynamic (seen in exercise 2). But then it deals with chunks.

I believe it is a way to correct the behaviour we can see on the previous figure on the high number of cores.