Assignment 4:

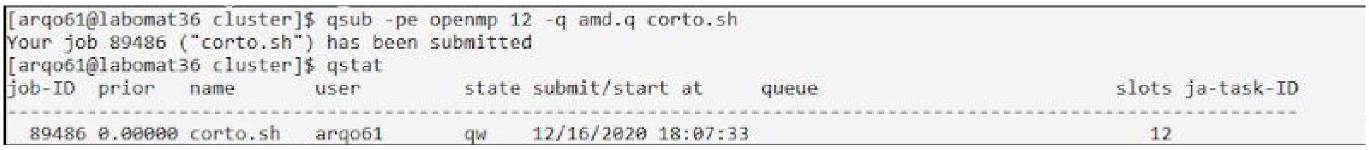
Exploiting the potential of modern architectures

Authors:
Kevin de la Coba Malam
Miguel Herrera Martínez

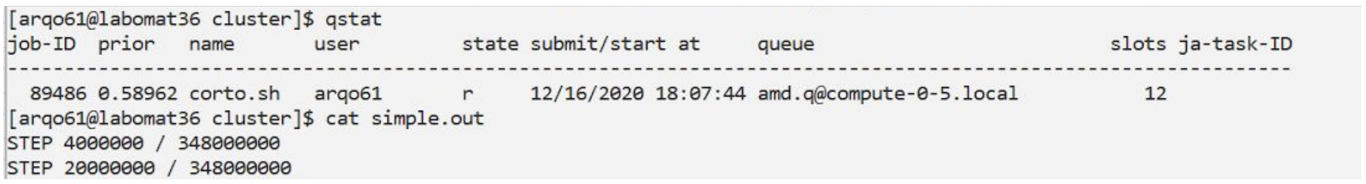
Group 1391
Pair 2

1. Introduction

In this assignment we learned how to parallelize programs using **OpenMP**. After parallelizing these programs we used the cluster in order to have a better parallel performance.
This is an example of how the program was executed by the cluster. We decided to take an AMD processor with 12 cores.



```
[arqo61@labomat36 cluster]$ qsub -pe openmp 12 -q amd.q corto.sh
Your job 89486 ("corto.sh") has been submitted
[arqo61@labomat36 cluster]$ qstat
job-ID  prior   name       user         state submit/start at     queue                          slots ja-task-ID
-----------------------------------------------------------------------------------------------------------------
  89486 0.00000 corto.sh   arqo61       qw    12/16/2020 18:07:33                                   12
```



```
[arqo61@labomat36 cluster]$ qstat
job-ID  prior   name       user         state submit/start at     queue                          slots ja-task-ID
-----------------------------------------------------------------------------------------------------------------
  89486 0.58962 corto.sh   arqo61       r     12/16/2020 18:07:44 amd.q@compute-0-5.local           12
[arqo61@labomat36 cluster]$ cat simple.out
STEP 4000000 / 348000000
STEP 20000000 / 348000000
```

2. Exercises

a. Information about the system topology

In order to see the information related to the computer architecture, we execute the following command:

cat /proc/cpuinfo

This will give us the information of the terminal between the cluster and us. So the number of cores of the cpu we used was 12.

b. Basic OpenMP programs

Is it possible to run more threads than cores on the system? Does it make sense to do it?

Yes it is possible, and it makes sense because you can be able to have multiple programs running at the same time (more programs than number of cores).

How many threads should you use in the cluster? And on your own team?

In the cluster we should use 12 or 16 threads whether they are the amd or intel processors as they have 12 or 16 cores respectively. On our own computer we use 4 threads because our personal computer has 4 cores.

Modify the omp1.c program to use the three ways to choose the number of threads and try to guess the priority among them.

The one with the highest priority is *#pragma omp parallel* ***num_threads(numthr)***, as the number of threads is set just before the creation of the threads. Then we have the function ***omp_set_num_threads(int num_threads)***, which is called in the program before #pragma... The call of this function overwrites the value in OMP_NUM_THREADS. Finally, we have the last which is the *environment* variable.

In order to guess this priority, we create a program which uses ‘x’ number of threads and then we change the environment variable to ‘y’. We can observe that the program won't change, because the number of threads has already been set (even when you change the value before the pragma creates the threads), it is using ‘x’ threads.

How does OpenMP behave when we declare a private variable?

It depends on which type of "privacy". If it is just *private*, each thread creates a copy of the variable but it is initialized to 0, while if you use *firstprivate* each thread has a copy of the variable but it also has the same value of the original one.

What happens to the value of a private variable when the parallel region starts executing?

OMP creates a copy and initialises the variable to 0 if you use the normal private, but if you use the firstprivate the value is maintained.

What happens to the value of a private variable at the end of the parallel region?

The variable is destroyed so the value disappears.

Does the same happen with public variables?

No, a copy is not created, the threads use the original variable, the last value after any modification will be there.

c. Parallelize the dot product

Run the serial version and understand what the result should be for different vector sizes.

The bigger the size, the bigger the time the function spends. It has one loop, so the time to sum is linear (O(n) being n the size), and depends on the size.

Run the parallelized code with the openmp pragma. Is the result correct? What is it happening?

There's a problem with concurrence, the variable sum is read and written by all the threads, so the result is wrong.

Modify the code and name the program pescalar_par2. This version should give the correct result using the appropriate pragma:
#pragma omp critical
#pragma omp atomic

- *Can it be solved with both directives? Indicate the modifications made in each case.*

  Yes, it can be solved with both directives. Here is our piece of code with the corresponding modifications:

  ```
  #pragma omp parallel for
  for(k=0;k<M;k++) {
          #pragma omp atomic
           sum = sum + A[k]*B[k];
   }

  #pragma omp parallel for
  for(k=0;k<M;k++) {
          #pragma omp critical
          {
                  sum = sum + A[k]*B[k];
          }
  }
  ```

- *What is the chosen option and why?*

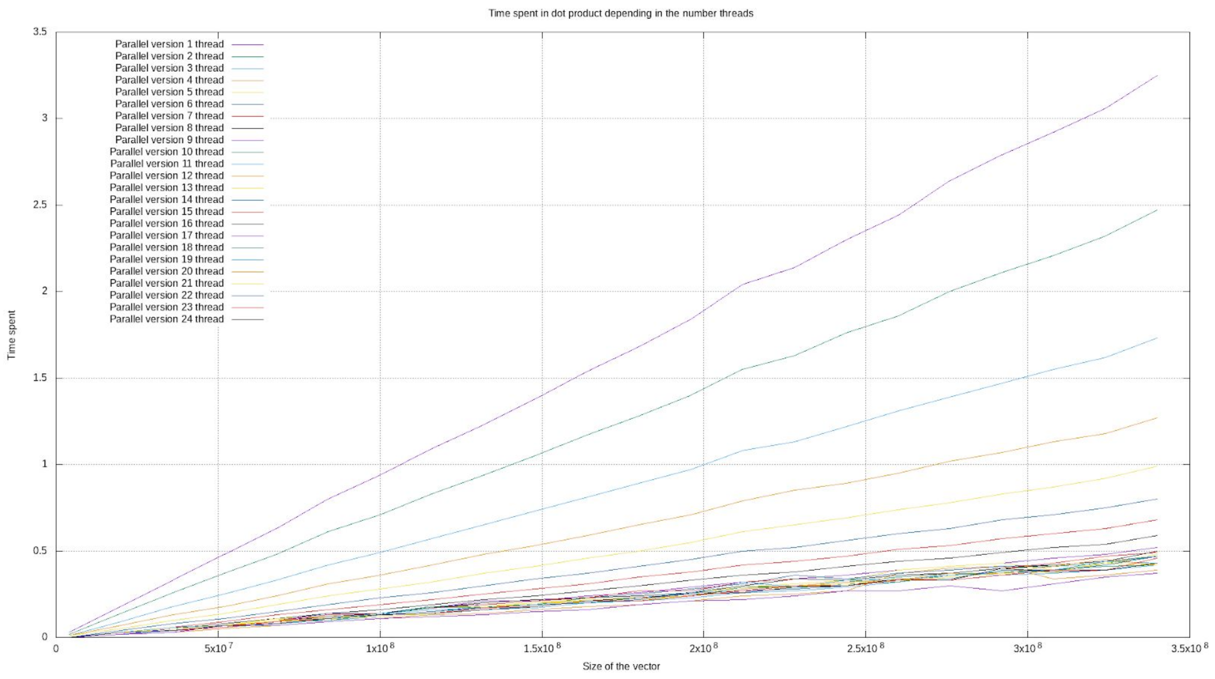
  As we are only executing one operation, making it atomic is our chosen option. We made this choice because critical is for more than one operation, atomic is just for one operation.

Modify the code and name the resulting program pescalar_par3. This version should give the correct result using the appropriate pragma:
#pragma omp parallel for reduction

We will choose the reduction option, because it is the fastest, and it is the fastest because with the previous options we were preventing from concurrence problems, the entire operation (including + and *) but we only need to protect from concurrence the addition of the variable sum.

Perform runtime analysis, by choosing a vector size that allows you to differentiate the behavior of parallelization. Taking now only the execution of the serial version and the parallel version that it considers to be the best option:

- *An analysis of the performance obtained in the scalar product problem is requested for different number of threads: from 1 to 2 * C, where C is the number of cores of the equipment used.*



As we can see in the chart, having 1-5 threads is the less efficient chose in order to run the program fast, we can see that having 6-8 is not that bad but is still on top on 9-24, in this range we can see that the performance is very similar, although it looks like having 17 threads for this program is the best choice.
To sum up, having 1-8 threads is the less efficient choice, and from 9-24 the performance is similar but the best choice will be 14-18.

- *Analyze how the acceleration obtained varies for vectors of different sizes.*

If we check the performance of the serial version with a size of 340.000.000 we can see that the performance is 3.25 seconds, having 2 threads makes the performance be 2.47. There's an acceleration of:
3.25/2.47 = 1.31, there is an acceleration of 31%.
4 threads - 3.25/1.27 = 2.56, 156% of improvement.
8 threads - 3.25/0.59 = 5.51, 451% of improvement.
12 threads - 3.25/0.39 = 8.33, 733% of improvement.
16 threads - 3.25/0.5 = 6.5, 550% of improvement.
20 threads - 3.25/0.42 = 7.73, 673% of improvement.
24 threads - 3.25/0.43 = 7.55, 655% of improvement.

Considering the results obtained, it is asked to answer the following questions:

- *In terms of the size of the vectors, is it always worth throwing threads to do the work in parallel, or are there cases where it is not?*

No, it is not always worth, for example, having a size of 40.000 and 2 threads makes the computation slower than computing without parallelism. Basically if we have small sizes it is not worth it.

- *If it does not always compensate, in which cases does it not compensate and why?*

  For sizes smaller than 40.000 having parallelism means that there are "too many people" working on the task. OpenMP has to manage all the threads and that requires resources.

- *Is performance always improved by increasing the number of threads to work with? If not, why is this effect?*

  No it is not, depending on the size of the task we may need a number ‘x’ of threads to make the performance optimal, otherwise, the computation will be worse.

- *Evaluate if there is any size of the vector from which the behavior of the acceleration will be very different from that obtained in the chart.*

  Around 40.000.

d. Parallel Matrix multiplication

Execution time (s) - size 1000

Version \ # threads	1	2	3	4
Serie	15.437755			
Parallel - loop1	16.069800	8.041054	6.175389	5.839255
Parallel - loop2	14.726471	6.074125	3.837393	3.035796
Parallel - loop3	11.842573	5.656065	2.778428	2.364950

Execution time (s) - size 2000

Version \ # threads	1	2	3	4
Serie	130.094616			
Parallel - loop1	130.862567	84.587017	60.805688	49.218190
Parallel - loop2	120.777636	72.260574	49.018565	36.806443

Parallel - loop3	122.531393	73.48178	48.041028	36.824258

Speedup (taking as reference the serial version) - size 1000

Version \ # threads	1	2	3	4
Serie	1			
Parallel - loop1	0.96	1.92	2.50	2.64
Parallel - loop2	1.04	2.54	4.02	5.095
Parallel - loop3	1.30	2.73	5.55	6.54

Speedup (taking as reference the serial version) - size 2000

Version \ # threads	1	2	3	4
Serie	1			
Parallel - loop1	0.99	1.53	2.14	2.64
Parallel - loop2	1.07	1.80	2.65	3.53
Parallel - loop3	1.06	1.77	2.70	3.53

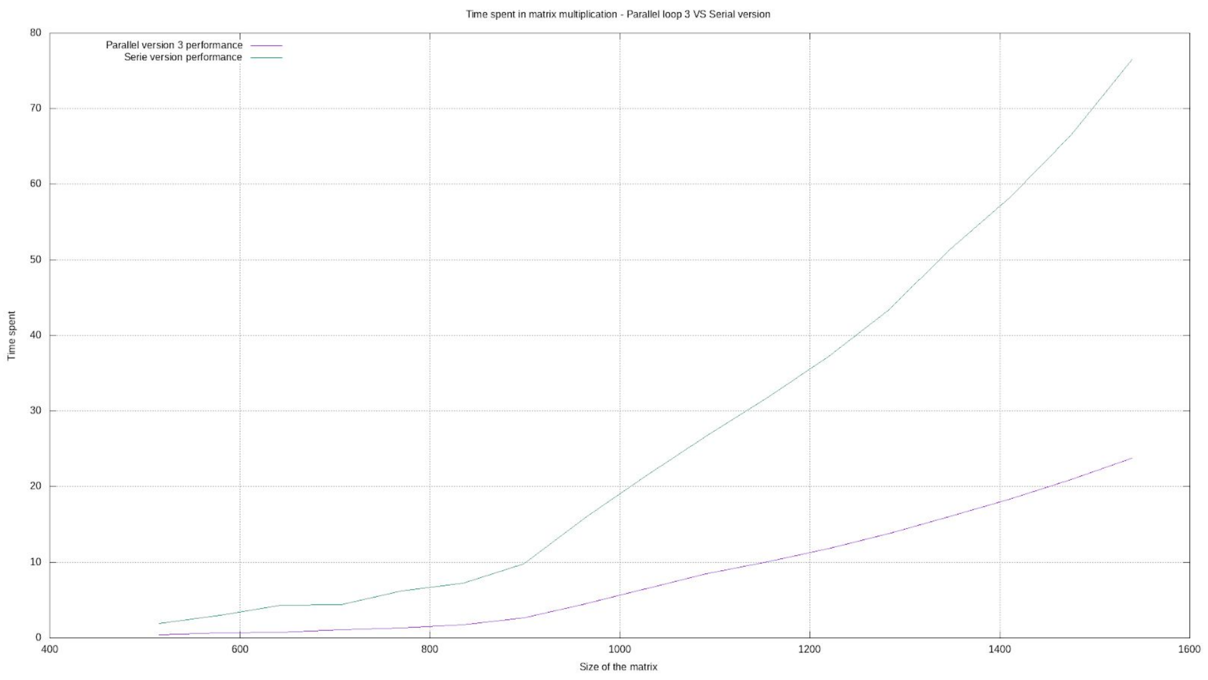
Which of the three versions performs the worst? Why? Which of the three versions performs better? Why?

The worst is the innermost loop, because if we parallelize just that loop we are parallelizing less code. Also, we are going to create threads in every iteration and this process requires time.

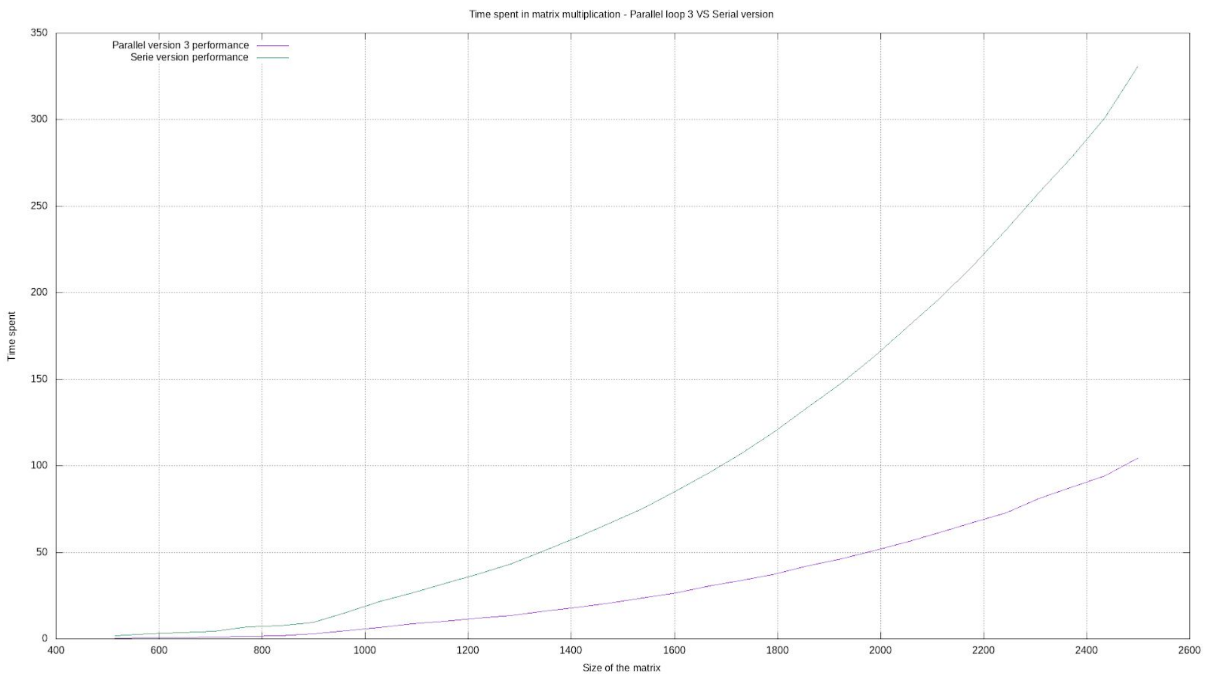
The outer loop parallelization is the best choice because we create the threads once and all the loops inside are parallelized as well distributing more of the work.

Based on the results, do you think fine-grained (innermost loop) or coarse-grained (outermost loop) parallelization is preferable in other algorithms?

This depends on the algorithm, generally we would prefer the coarse-grained but depending on the loop the fine-grained may be better, but generally the coarse-grained is better.



We can see that the parallelization in the outermost loop is way more efficient than the serial version.
We can also see that somewhere between 800 and 1000 the growth becomes linear (not exactly linear but it looks like), and for the serial case the same but, between 1200 and 1400 it changes a little bit and it looks that it is becoming linear.



We continued the execution until 2500 and here we can see in a better way the growth of both cases, we can see that is quadratic because the complexity of the algorithm is O(N^2).

If in the previous chart you did not obtain a behavior of the acceleration as a function of N that stabilizes or decreases with increasing the size of the matrix, continue increasing the value of N until you get a chart with this behavior and indicate for which value of N you begin to see the change in trend.

From somewhere between 800 and 1000, the growth becomes linear.

e. Example of numerical integration

How many rectangles are used in the program to perform the numerical integration?

100 million rectangles are used in the program to perform the numerical integration.

What differences do you “see” between these two versions?

In pi_par4 there is a variable called "priv_sum", this is a local static variable created inside the pragma zone which is used in the loop and then the value is stored in the allocated memory.

In pi_par1 we use the variable which was allocated before creating the paralel zone, so managing concurrence makes the version 1 slower.
Although we allocate memory for the number of threads, the variable is considered unique, therefore it cannot be accessed at the same time.

Run the two versions. Are there differences in the result obtained? And in performance? If the answer is affirmative, could you justify why this effect is due?

Yes there are differences, the version 4 is faster than the version 1. This is due to the way the summation is done. In version 1 we use a common variable for all threads in each iteration of the loop, but in the version 4 we use a variable created inside the pragma zone, version only updates the value of that variable when the loop is finished (once per thread).

Run parallel versions 2 and 3 of the program. What happens to the result and the performance obtained? Did you expect these results?

The performance of the version 3 is faster. Version 3 checks the size of each block of the cache and then allocates memory for a variable, so all the threads access at one block of the cache.

Open the file pi_par3.c and modify line 32 of the file to take the fixed values 1, 2, 4, 6, 7, 8, 9, 10 and 12. Run this program for each of these values. What happens to the performance that is observed?

Value	Performance (s)
1	0.395503
2	0.343166
3	0.307214
4	0.324928
5	0.327565
6	0.349168
7	0.242553
8	0.222530
9	0.271752
10	0.192711
12	0.283687

It looks like the performance is better when the numbers are between 7-10 (both values included).

Run versions 4 and 5 of the program. Explain the effect of using the critical directive. What differences in performance do you see? Why this effect?

In version 5 the variable used for the summations is a copy of the one created by the caller thread (the father), in the version 4 each thread creates their unique static variable for the summation and then updates their part of the variable (with part we mean sum[0...]).

In the version 5 the critical zone is in the update part of the variable, this makes that this part cannot be done at the same time by different threads, but in the version 4 the variable is divided in the number of threads used, so each thread access to his part and updates the variable, without needing to check concurrence.

Run versions 6 and 7 of the program. Explain the effect of using the directives used. What differences in performance do you see? Why this effect?

The version 7 is the less efficient because, first it parallelizes the algorithm and creates some variables, after this it parallelizes the for loop so threads are able to work without concurrence problems. In the version 6 with the "reduction" we parallelize just the summation of the loop, making the parallelization way more efficient.

f. Optimization of calculation programs

The program includes an outermost loop that iterates over the arguments applying the algorithms to each of the arguments (indicated as Loop 0). Is this loop optimal to be parallelized?

It depends on how many images we sent, if there is just 1 image, it is not worth it.

- *What happens if fewer arguments are passed than number of cores?*

  That the parallelization makes the performance worse.

- *Suppose you are processing images from a space telescope that occupy up to 6GB each, is this the right option? Comment how much memory each thread consumes based on the size in pixels of the input image.*

  If the images sent to parallelize are only 1, this is still inefficient, we need to parallelize the inner loops, but if we send more images it will be better to parallelize it because each image takes a lot of time.

During the previous task (task 3), we observed that the order of access to the data is important. Are there any loops that are accessing the data in a suboptimal order? Please correct it in such case.

Yes there are, the ones which access the images pixels.

- *Explain why the order is not correct if you change it.*

  We swapped the loops, instead of going width and then height (columns) we go through height and then width (rows). The order is not correct for the same reason it was in the previous assignment, accessing the elements by columns means that the element we are looking for is not in any of the cache blocks.

Bypassing Loop 0, test different parallelizations with OpenMP explaining which ones should get better performance.

- *It is not necessary to fully explore all the possible parallels. It is necessary to use the knowledge obtained in this task to define which would be the best solutions. Explain the reasons in the document.*

  We just parallelized the loops which traverse through the image. In these cases we just did *#pragma omp parallel for private(j)* in the outer loop. If in the loop there was an operation we just did *#pragma omp parallel for private(i) reduction(+:sum)*, so in this case we performed a reduction in the variable which is used.

Also at the beginning of the code we write *#pragma omp parallel for private(i, j) if (nargs > 3)*, with the ‘if’ we are just saying that if we have to parallelize more than 1 image, then we will parallelize the main loop, so we compute the images at the same time.

Fill in a table with time and speedup results compared to the serial version for images of different resolutions (SD, HD, FHD, UHD-4k, UHD-8k). You must include a column with the fps at which the program would process.

Seconds per execution

Version - Threads	8K	4K	FHD	HD	SD
Serie	21.401123	5.034060	1.265117	0.557221	0.131499
Parallel - 1 Thread	23.428791	5.849365	1.463171	0.653356	0.164293
Parallel - 2 Thread	16.960269	4.089475	1.018536	0.504539	0.113385
Parallel - 4 Thread	9.627730	3.872561	0.887876	0.392000	0.066402
Parallel - 8 Thread	11.761535	2.885491	0.723117	0.319663	0.080014

As we can see the performance is improved when we use more than one thread. If we check the 8K image’s time, we can see that the best performance is the 4 threads case, but if we check the other resolutions we can see that 8 threads is the best. As we just measured these times once, the time of the 4 threads could be an outlier, because in almost all the resolutions 8 threads is the winner.

FPS

Version - Threads	8K	4K	FHD	HD	SD
Serie	0.04673	0.1986	0.79044	1.79462	7.6046
Parallel - 1 Thread	0.0427	0.17096	0.6834	1.53056	6.0867
Parallel - 2 Thread	0.0589	0.2445	0.9818	1.982	8.8195
Parallel - 4 Thread	0.10387	0.25823	1.1262	2.5510	15.059
Parallel - 8 Thread	0.085	0.34656	1.3829	3.12829	12.4978

If we check the FPS we can see that having 4 or 8 threads makes the execution of the SD images fast, it can execute 15 images per second.
The other good thing of the parallelization is that we can have the same times if we add more images (obviously not exactly the same, it depends on our hardware). If we try to compute several images and we set more threads than images, each image will have the same computation time as the ones we have now (depends on the hardware).