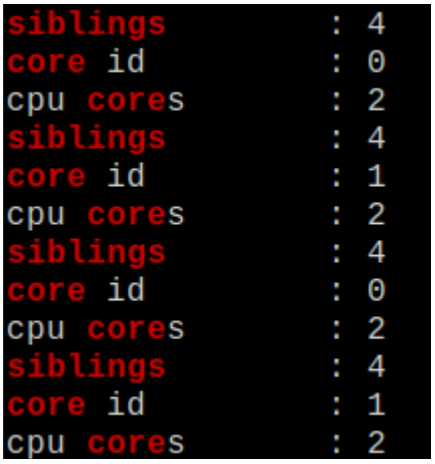
# Task 4
# Computer Architecture

3rd year Computer Science and Engineering
Escuela Politéctina Superior
Universidad Autónoma de Madrid

Pablo Ernesto Soëtard García

Exercise 0:
In one of our personal computers, the cpu configuration is as follows:




As we can see, we have 2 real cores and 4 virtual (siblings), that means that hyperthreading is active in our machine.

Note: not all the experiments have been executed in that computer.

Exercise 1:

1.1 Yes, you can execute as many threads as you want, but in a certain moment that will not decrease the execution time of your parallel program (because your CPU has a limited number of cores), in fact, it may increase it because threads are OS resources that have to be created and that takes time.

1.2 In the lab computers we may use 4 threads, in the cluster 16 threads in Intel machines and in our own computer 4 threads.

1.3 After modifying the requested program, we have seen that the priority order of the different ways of setting the number of threads is: OMP_NUM_THREADS, omp_set_num_threads(int num_threads); and #pragma omp parallel num_threads(numthr) being OMP_NUM_THREADS the one with less priority.

```
Inicio: a = 1,   b = 2,  c = 3
        &a = 0x7fffad92050c,x   &b = 0x7fffad920508,    &c = 0x7fffad920504

[Hilo 0]-1: a = 0,       b = 2,  c = 3
[Hilo 0]         &a = 0x7fffad9204c4,    &b = 0x7fffad920508,    &c = 0x7fffad9204c8
[Hilo 0]-2: a = 15,      b = 4,  c = 3
[Hilo 3]-1: a = 0,       b = 2,  c = 3
[Hilo 3]         &a = 0x7f25cf17fe04,    &b = 0x7fffad920508,    &c = 0x7f25cf17fe08
[Hilo 3]-2: a = 21,      b = 6,  c = 3
[Hilo 2]-1: a = 0,       b = 2,  c = 3
[Hilo 2]         &a = 0x7f25cfb80e04,    &b = 0x7fffad920508,    &c = 0x7f25cfb80e08
[Hilo 2]-2: a = 27,      b = 8,  c = 3
[Hilo 1]-1: a = 32549,   b = 2,  c = 3
[Hilo 1]         &a = 0x7f25d0581e04,    &b = 0x7fffad920508,    &c = 0x7f25d0581e08
[Hilo 1]-2: a = 1059437434,      b = 10,         c = 1059437404

Fin: a = 1,      b = 10,         c = 3
        &a = 0x7fffad92050c,    &b = 0x7fffad920508,    &c = 0x7fffad920504
```

1.4,5,6 When declaring a private variable OpenMP creates a copy of the original variable per thread, thus its address changes and its value is set to whatever data is stored in that new address. We can see that the modifications made to that private variable during the parallel part are not propagated to the main variable.

1.7 With public variables OpenMP does not create a copy of them, thus having all the variables of each thread the same memory reference, all changes made to that variable in the threads will persist in the main variable, as they are the same.

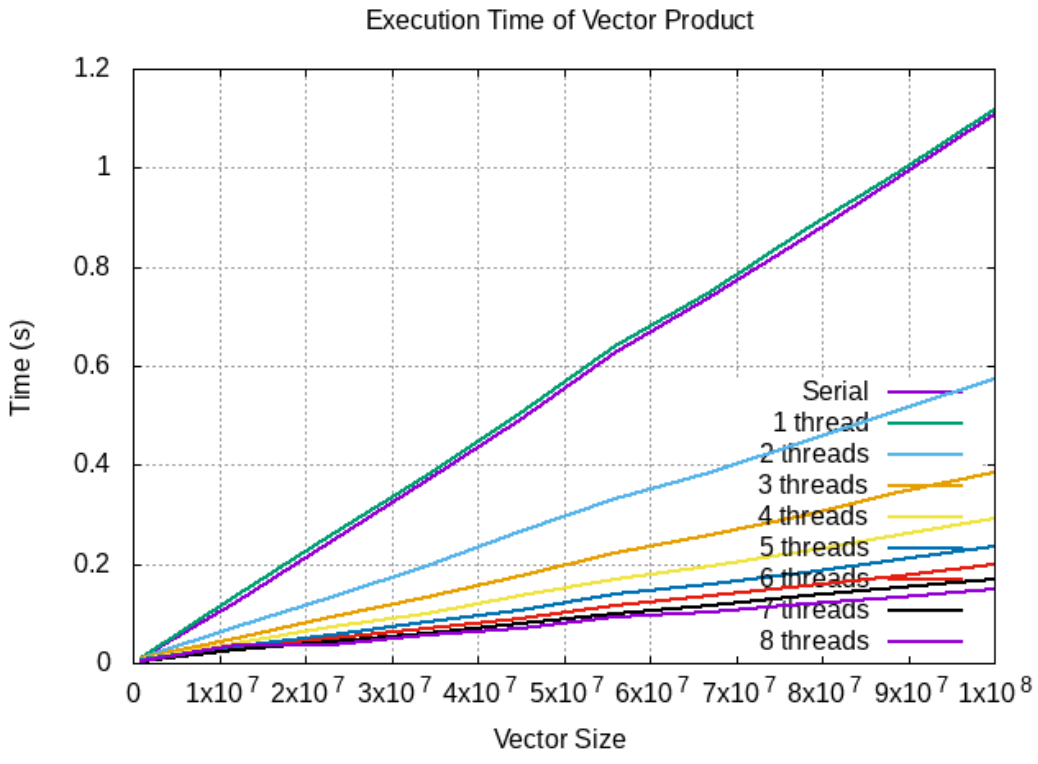
Exercise 2:

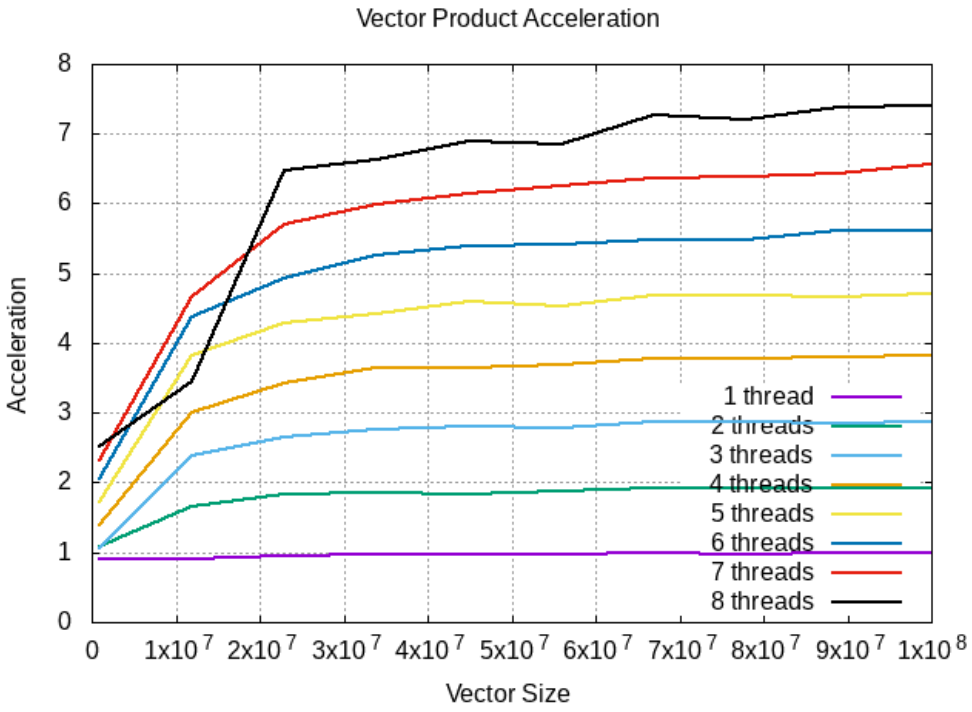
2.2 After executing several times both programs (the serial and parallel), we can see that the serial always outputs the same result, while the parallel outputs random results. The correct result is the one given by the serial program, as the parallel program has a race condition with the sum variable, and thus the result is very likely to be wrong having 16 threads reading and writing from the same variable without any order.

2.3 The synchronization problem can be solved with both directives (critical and atomic), we just need to put the directive above "sum += A[k]*B[k];" and surround that expression with brackets in the case of critical.
As we only have one statement that we want to protect it makes more sense to use atomic instead of critical.

2.4 We have substituted "#pragma omp parallel for" by "#pragma omp parallel for reduction(+:sum)", with this command we set the reduction operand to + and the variable to be reduced to sum. With these modification we have obtained much faster execution times.

2.5 We have executed 10 times, with 10 data points between vectors of size 1000000 and 100000000. We have chosen the pescalar_par3 as the best option, and we have executed it with 1 to 8 threads, as our computer has 4 cores. These are the results:






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

For small vector sizes the time required to create and execute each of the threads may be greater than the time taken to execute the program in serial.

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

If we look at the charts above it seems that the performance is always improved by increasing the number of threads, but this is not true, we know that in a certain moment, adding more threads will no longer improve the performance, as we will run out of resources, we are limited by the number of cores of our processor. Furthermore there may be a point in which the performance decreases due to the time taken by the threads to be created being too large.

- 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.

As we can see in the charts, as we increase the size of the vectors, the acceleration tends to converge to a certain value, we expect this behavior to persists for larger vector sizes, as both linear an parallel programs are O(N) in time as we can see in the time chart, so their acceleration may remain constant.
On the other side, for small sizes of vectors randomness and lucky may be a crucial factor, as perhaps the time taken to create the threads happens to be greater to the one to run the serial program, thus we would get accelerations smaller than 1.
We have set $M > 1000000$ as we think that for smaller sizes it is not worth, based on the data collected from the experiments, to execute the program in parallel.

Exercise 3:

10 seconds execution (1000x1000 matrix size):

Execution times (s)

| Version\ # threads | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Serial | 13.014265 | | | |
| Parallel – loop1 | 12.833610 | 8.363587 | 7.247190 | 6.314975 |
| Parallel – loop2 | 12.546344 | 6.100008 | 4.090761 | 3.087078 |
| Parallel – loop3 | 11.547909 | 5.945559 | 4.043026 | 3.010299 |

SpeedUp (relative to the serial version)

| Version\ # threads | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Serial | 1 | | | |
| Parallel – loop1 | 1.014076 | 1.556062 | 1.795767 | 2.060857 |
| Parallel – loop2 | 1.037295 | 2.133483 | 3.181379 | 4.215722 |
| Parallel – loop3 | 1.126980 | 2.188905 | 3.218941 | 4.323246 |

60 seconds execution (1750x1750 matrix size):

Execution times (s)

| Version\ # threads | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Serial | 70.951581 | | | |
| Parallel – loop1 | 67.960336 | 42.655093 | 31.200557 | 29.266182 |
| Parallel – loop2 | 66.162924 | 33.941543 | 22.450832 | 16.860906 |
| Parallel – loop3 | 62.287806 | 32.010189 | 21.466974 | 16.256644 |

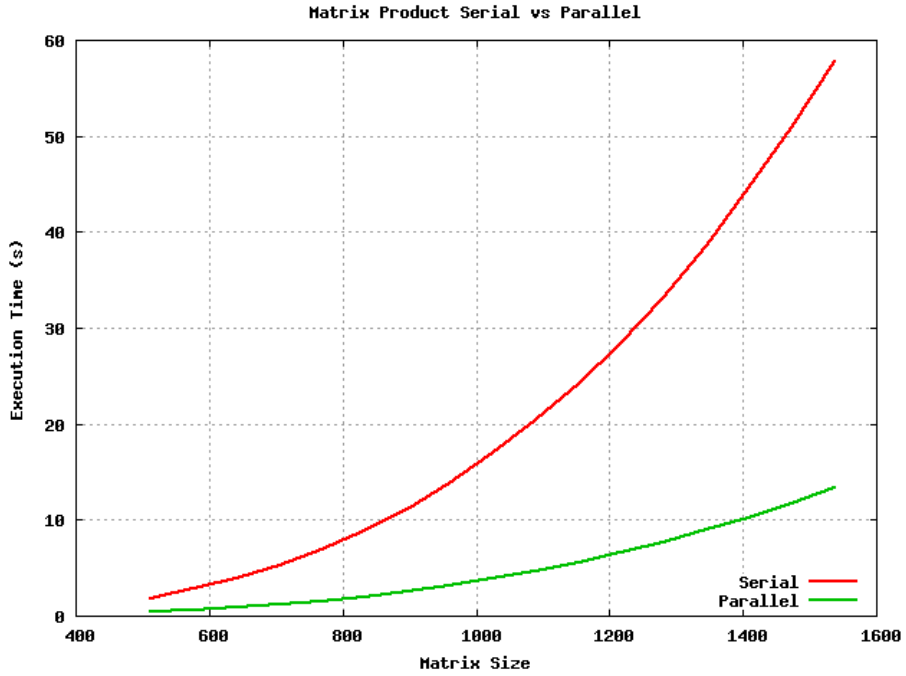
SpeedUp (relative to the serial version)

| Version\ # threads | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Serial | 1 | | | |
| Parallel – loop1 | 1.044014 | 1.663378 | 2.274048 | 2.424353 |
| Parallel – loop2 | 1.072376 | 2.090405 | 3.160309 | 4.208052 |
| Parallel – loop3 | 1.139092 | 2.216531 | 3.305150 | 4.364466 |

3.1 The worst version is the Parallel - loop1, in this version, we make the innermost loop parallel, thus we have to create and join the threads NxN times, being N the matrix size, and this takes a lot of time and resources. Instead, the most efficient version is the Parallel - loop3 that makes the outermost loop parallel, thus we only have to create and join the threads once, so the performance is way better.

3.2 Taking into account the results of the experiments and the conclusions stated in the previous point about them, it is reasonable to say that the coarse-grained parallelization achieves a better performance.

3.3 We have executed 5 times, with matrix sizes between 513 and 1537 with increments of 64. We have chosen the Parallel - loop3 as the best option, and we have executed it with 1 to 4 threads. These are the results:




We can see in the time chart that, as expected, the serial version takes more time to execute compared to the parallel version and that its growing rate is much faster.

In the acceleration chart, we can see that at the beginning we get a very high acceleration, reaching almost 4.43, but then, at around 800 (matrix size) we see that the acceleration decreases, getting a sort of stabilization around 4.34.
We were expecting that behavior because the acceleration tends to a certain value as matrix size gets larger and larger, as we explained in the vector's experiment, the reason why that happens is that both time functions, Serial and Parallel, are $O(N^3)$ with different growing factor rates, and thus its division (that is its acceleration) for large matrix sizes will result in the number which the acceleration will converge, in this cases around 4.34. For lower matrix size we may see that the acceleration decreases, as lucky and randomness take place, as we also explained in the vector's experiment.

Exercise 4:

4.1 In the program given, 100000000 rectangles are used.

4.2 The main difference is that in pi_par1 each of the threads adds its partial results to "sum[tid]" and in pi_par4 those partial results are added to a private variable "priv_sum" that at the end is assigned to "sum[tid]".

4.3 The final result is the same (3.141593) for both, pi_par1 and pi_par4, but the execution time is much faster in the pi_par4 program with 0.170392 seconds compared to 2.084247 seconds that the pi_part1 takes.
This is due to a false sharing issue, as the array sum is of size #threads it very likely fits in less than a cache block, thus in pi_par1, each time a thread modifies it, this forces all other threads to reload their shared sum array, and that takes time.
Instead, pi_par4 just modifies the sum array once, keeping its partial results in a private variable that must not be reloaded each time it is modified, because it is private.

4.4 The final result is the same (3.141593) for both, pi_par2 and pi_par3, but the execution time is much faster in the pi_par3 program with 0.175526 seconds compared to 3.141593 seconds that the pi_part2 takes.
This time the differences are that in pi_part2, sum is declared as firstprivate for each thread, thus it keeps its value, as sum is a pointer, keeping its original value means it points to the same memory space for every thread, so there is false sharing and due to the same reasons explained before, the execution time is similar to the one of pi_par1.
Instead in pi_par3, the trick that is performed is that now the sum array size is set in such a way that each thread addresses its "own" cache block, and thus, it is very unlikely that a cache block reload happens during the modifications of the sum array by the threads. This is why the execution time of pi_par3 is similar to the one of pi_par4.

4.5 For all configurations the result is the same (3.141593) but the execution time decrease as we increase the padding as it can be seen in the following times:

1: 2.607903
2: 0.956058
4: 0.518870
6: 0.442846
7: 0.432013
8: 0.218881
9: 0.181265
10: 0.185707
12: 0.181509

As we can see, as we increase the padding among the memory space that each thread accesses, we get lower times. This is true until size 8, where time decreasing starts to decelerate, this happens because we have reached the point were each thread addresses its own cache block (cache block of 64 bytes / 8 bytes per double => 8 units of padding) and thus there are no cache block reloading and we get similar execution times as in pi_par4.

4.6 The code placed under a critical directive can only be accessed by one thread at a time.
We have seen that the final result is somewhat random in the case of pi_par5 but always smaller than 3.141593, this happens because the iteration variable i is shared by all the thread, thus it is increased without any control and therefore it is very likely that some calculation loops are skipped, that is why we get a wrong result always smaller than pi. Also the pi variable is not initialized so it may also lead to adding a strange initial offset to the calculation.
Having said that, the execution time is also much slower in the case of pi_par5 (4.734990 seconds) compared to pi_par4 (0.171799 seconds), this may be caused by two factors, the first, a false sharing on the i variable that cause many reloads because other threads modify it without control, the other one is the critical region that acts like a funnel, slowing down the speed because it forces the threads to enter to that region of code one by one.

4.7 Once again for both programs we get the correct result (3.141593), but their difference is in the execution time. Program pi_par6 (2.153279 seconds) is much slower than pi_par7 (0.205631 seconds), pi_par6 uses the for directive to parallelize the for, but it uses the sum array as in pi_par1 causing false sharing and thus increasing the reloads and execution time.
In pi_par7 the for directive is used with the reduction(+:sum) directive, that organize all the threads in such a way that they perform their partial calculation in their own variable sum, and at the end all partial results are reduced by the operation given, in this case adding them. This time we have sum declared as a double and the directives for and reduction take care of the organization of the threads and the final result, thus we do not have false sharing and the execution time is similar to the best ones we have obtained in this exercise.

Exercise 5:

1 We know that the final goal of this application is to process live video streaming so parallelizing Loop 0 may be a good idea, as this will cause that each image will be processed in parallel and thus reduce the time, because you dont have to do it in a serial way as it is now. But if you just want to apply these algorithms to few images you may not need to parallelize the loop. If the number of arguments is lower than the number of cores of the machine, each argument image will be processed in parallel, otherwise all the argument images will be processed in different threads but you will not necessarily get the same performance as in the case before, because you will run out of physical resources.
If you were processing large images like the ones from a space telescope that occupy up to 6GB you will not get any performance enhancement by parallelizing the loop 0, as explained above, instead you will only get bigger threads that will be proportional in space size to the one of the input image, so if you introduce an image of 6GB or more you will very likely run out of main memory if you do not have a very powerful computer.

2 We have corrected several cache inefficient loops such as the one of "// RGB to grey scale" or the ones of the gaussian_kernel or PIXEL_GREY and PIXEL_EDGES. In all of this loops the image was accessed by two nested for loops, the outermost loop was looping through the width and the innermost through the height, thus traversing the image by columns and causing lots of cache misses. We have inverted the order of the loops in order to traverse the image by rows instead of by columns, reaching a much lower cache miss rate. With this approach we have reduced the time to process the 8k image 4 seconds on average.

3 From previous exercises we have seen that parallelizing outer loops is better than parallelizing inner ones, so that is what we did. We paralellized the outer loop of several nested loops that appear in the code, making sure that all the variables that were used inside them were properly declared as private, shared, firstprivate... And in some cases applying a reduction statement to the openmp directive to not cause false sharing among threads. After that we have seen that the better performance was achieved by parallelizing the "gaussian denoising" and the RGB to grey scale outer loops as well as the ones from the gaussian_kernel function. If you change the TYPE_FILTER to MEDIAN it may be a good idea to parallelize the "median denoising" outer loop, just uncomment the commented pragma.

4 These are the final results after performing the code modifications mentioned above:

| | Serial time (s) | Parallel time (s) | SpeedUp | FPS Serial | FPS Parallel |
|---|---|---|---|---|---|
| SD | 0.144031 | 0.084077 | 1,713084435 | 6,94 | 11,89 |
| HD | 0.614992 | 0.106775 | 5,75970030 | 1,62 | 9,36 |
| FHD | 1.399573 | 0.359995 | 3,887756774 | 0,71 | 2,77 |
| UHD-4K | 5.525943 | 0.926495 | 5,964352749 | 0,18 | 1,07 |
| UHD-8K | 24.793902 | 3.688512 | 6,721925264 | 0,04 | 0,27 |

5 After applying the -O3 compilation flag we have achieved the following results, as we can see the performance is much better now:

| | Serial time (s) | Parallel time (s) | SpeedUp | FPS Serial | FPS Parallel |
|---|---|---|---|---|---|
| SD | 0.044094 | 0.016386 | 2,690955694 | 22,67 | 61,02 |
| HD | 0.199952 | 0.040746 | 4,907279242 | 5,00 | 24,54 |
| FHD | 0.451088 | 0.090659 | 4,97565603 | 2,21 | 11,03 |
| UHD-4K | 1.803959 | 0.362109 | 4,981812106 | 0,55 | 2,76 |
| UHD-8K | 8.489441 | 1.406407 | 6,036261907 | 0,11 | 0,71 |

According to the gcc documentation, the O3 flag is the highest optimization level, that turns on all optimization options, including the ones of loop unrolling (-floop-unroll-and-jam) and loop vectorization (-ftree-loop-vectorize).