Arquitectura de ordenadores. Practice 4.

Pablo Cuesta Sierra, Álvaro Zamanillo Sáez
(group 1392, team 04)

December 12, 2021

Contents

1 Exercise 1: Basic OpenMP programs

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

It is possible to do so: if we first execute omp1 without arguments to see the available cores and then we execute it again with a number of threads higher than the amount of cores, we will see that the execution is carried out without problems. Using more threads than cores may make no sense when the goal is simply to parrallelize a single task because the threads will interrupt each other. However, if there are several tasks to be run at the same time and any of them may provoke the thread to go asleep, then using more threads could help to take advantage of the time that particular thread is asleep.

1.2 *How many threads should you use on the computers in the lab? And in the cluster? And on your own team?*

For a general situation, the number of threads to be used is the one determined by the function *omp_get_max_threads()*. Indeed this value used by default.

1.3 *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 priority order is: OMP_NUM_THREADS < `omp_set_num_threads()` < #pragma omp parallel num_threads(numthr)

1.4 *How does OpenMP behave when we declare a private variable?*

Each thread created will create its own (new) variable.

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

The variable is not necessarily initialized when allocated in each thread stack, although when executing `omp2` in the lab computers, it is initialized as if it were a *firstprivate* variable (with the value of the variable previous to the parallel section). Furthermore, it is private to each thread so its value will only change because of instructions performed by that specific thread.

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

After the parallel region, the master threads resumes execution and the value of the variable is the one from before the parallel execution. Privates variable are local to the threads and the parallel region, so they are deleted when the threads terminate.

1.7 *Does the same happen with public variables?*

Public variables are shared by all threads (no copies of the variable are created for each thread); threfore, after the parallel region, the value will be determined by the code executed in the parallel region (and, perhaps, by the order of execution of the several threads).

2 Exercise 2: Parallelize the dot product

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

As we are calculating the dot product of two vectors of size M where all the components are 1, the expected output is M.

2.2 *Run the parallelized code with the openmp pragma and answer the following questions in the document*

The parallel version is not working due to the fact that variable sum is being shared (default mode). So we have severals threads writing in the same variable at the same time which produces a data race.

2.3 *Modify the code and name the program pescalar_par2. This version should give the correct result using the appropriate pragma:*

The modification needed is adding the pragma clause just before the calculation. Both *pragmas* solve the data race. Nonetheless, the way they achieved so is different. *Critial* pragma implies the use of a mutex

so only one thread can access the block at a time (which is quite expensive in execution time) whereas *atomic* ensures that the whole calculation (line 42) is executed as a single operation. The result is exactly the same as we are enclosing just a sentence of code, however, the second option is considerably faster.

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

2.4 *Modify the code and name the resulting program pescalar_par3. This version should give the correct result using the appropriate pragma:*

If we use *reduction* we get better results than the ones obtained with *atomic*. This is the appropriate solution in this case, because it is the optimized version meant for this specific task.

2.5 *Run time analysis*

We have written a script to test this (*scr2_threshold.sh*). As can be seen in the following figure (fig. 1), the parallel version (in the lab computers, that use 4 threads) is very inconsistent. However, the first value that meets the requirements is $T = 1200000$:

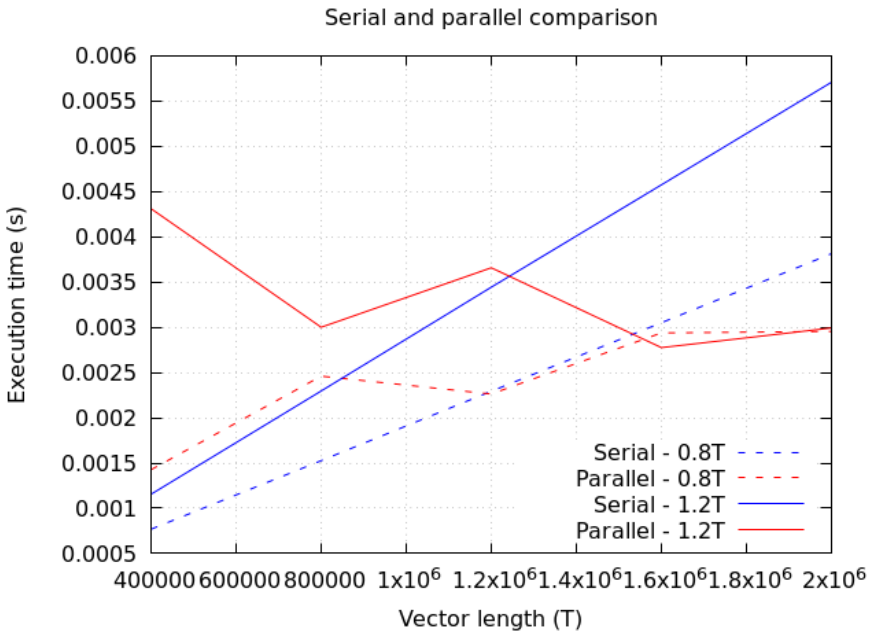

Figure 1: Search for threshold in the lab computers

2.6 *(Optional) Exhaustive analysis*

Executing the same tests in the cluster, using also 4 cores, we have reached the value that meets the requirements at, aproximately, $T = 81000$, as can be seen in the following figure (2):

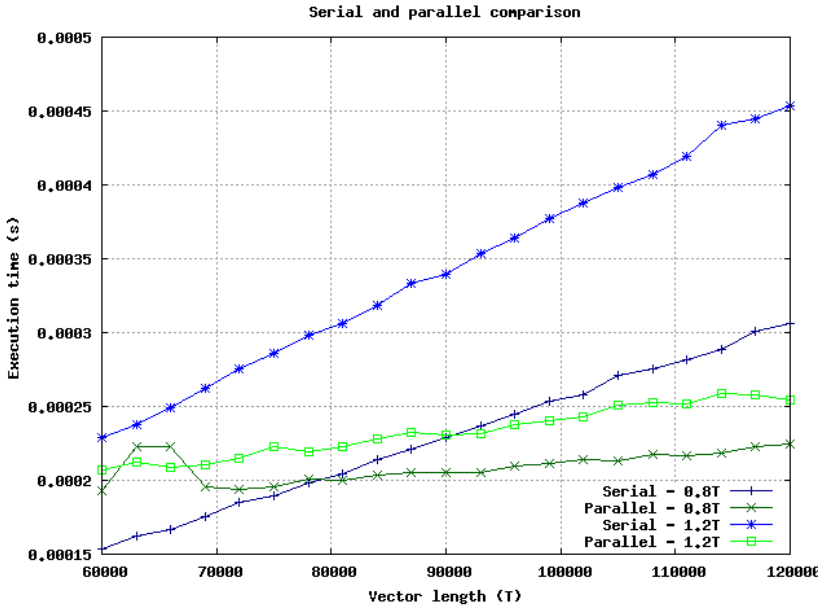

Figure 2: Search for threshold in the cluster: 4 cores

Changing the number of threads to 8, the number increases slightly, the threshold would be around $T = 90000$. This makes sense, because, although using 8 threads would yield a much better performance for very large matrix sizes, the threshold for which creating a higher number of threads is worth the time spent creating them is also a bit higher.

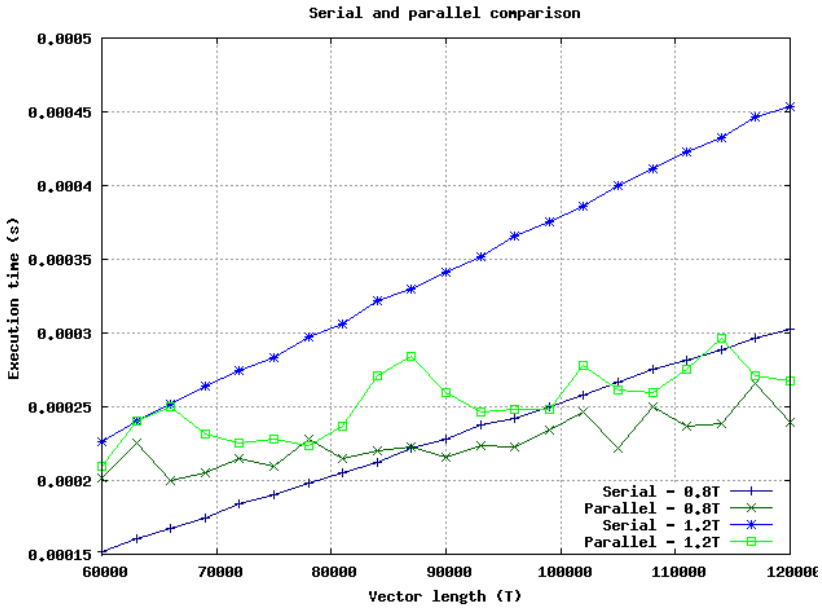

Figure 3: Search for threshold in the cluster: 8 cores

3 Exercise 3: Parallel matrix multiplication

We have sampled times for the different programs using matrices of size 2000, which yield high enough times to make a difference between the different implementations. Here are the results:

threads	1	2	3	4
serial	53.7103	54.5405	55.4747	56.9697
loop1	56.1735	29.408	27.5136	28.2914
loop2	54.843	27.9134	18.6343	14.2978
loop3	53.3989	27.7954	18.5844	14.0748

Figure 4: Times (seconds) for the requested tests

threads	1	2	3	4
serial	1	1.01546	1.03285	1.06068
loop1	1.04586	0.54753	0.512259	0.526741
loop2	1.02109	0.519703	0.34694	0.266203
loop3	0.994202	0.517506	0.346011	0.262049

Figure 5: Ratios for the requested tests

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

The worst performing version is the one that parallelizes the innermost loop, because it is forced to "fork" itself N^2 times, which causes the execution to be a lot slower. The best performing one, as could be expected, is the one that parallelizes the outermost loop, it has to separate into different threads only once (the *multiplication_loop2.c* one "forks" N times).

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

Based on these results, coarse-grained parallelization is generally preferable, as it avoids slowing the execution by killing and creating threads multiple times.

Taking the best parallel version (parallelizing the outermost loop) alongside the serial version, we have tested for N between $512 + P$ and $512 + 1024 + P$ with increments of 64. And with the parallel version, we have tested for 2, 4 and 6 threads. These are the results:

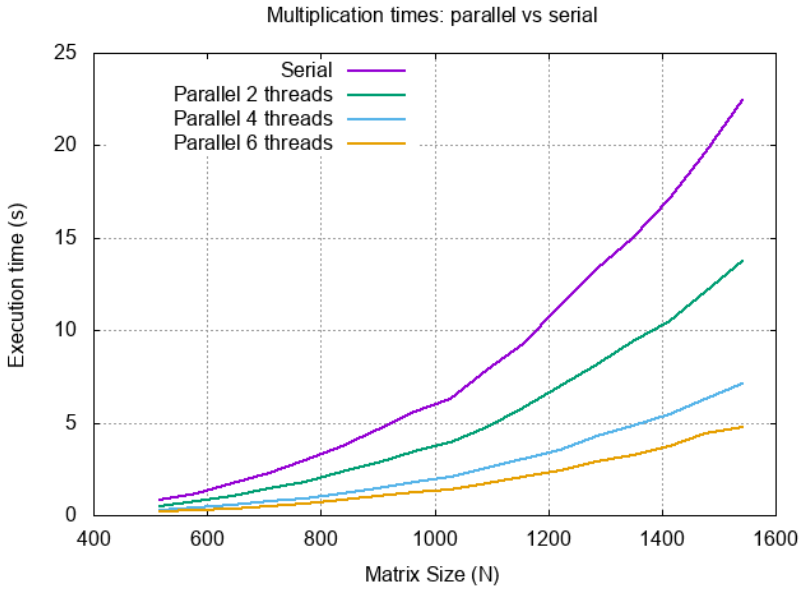

Figure 6: Time in seconds.

As expected, the plots have a cubic behaviour. Comparing the different plots, as the values of N are sufficiently large to take advantage of the parallelization, the higher the number of threads, the less time it takes to compute the multiplication. The serial version is the slowest for all of these values of N.

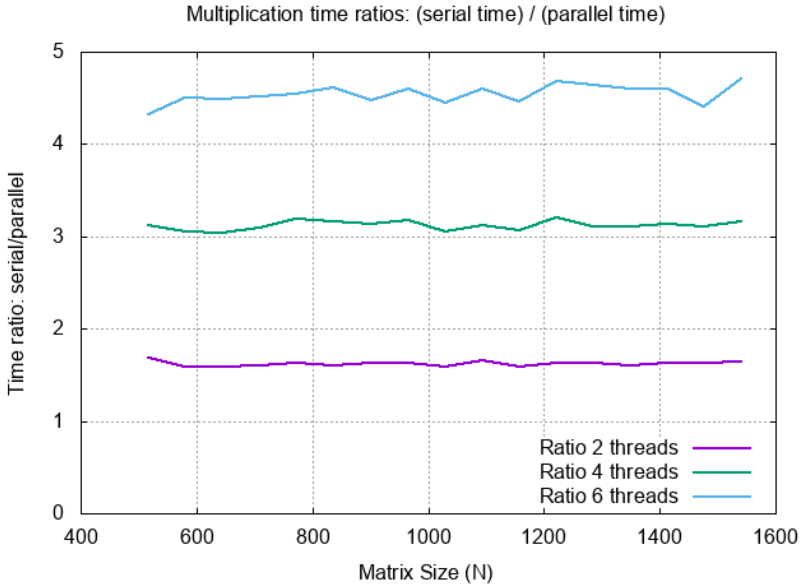

Figure 7: Time ratios.

As commented before, for the sizes that we have tested, the asymptotic behaviour is already present (as N is big enough). The four time fuctions have a complexity of $\mathcal{O}(N^3)$, which means that if we calculate the speedup for each parallel version (with different number of threads), it will stabilize around a constant in each case. This constant behaviour of the ratios is what can be seen in this plot. Again, as lower number of threads is worse than higher number of threads for big enough values of N, the constant in question is higher when the number of threads is higher.

3.3 *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.*

The values that we got already stabilize the speedup; as can be seen in figure 7, the values stabilize: for 2 threads, around 2.6; for 4 threads, around 3.1; and 6 threads, 4.6.

In order to check the values of N that stabilize the speedup, we have to check lower values:

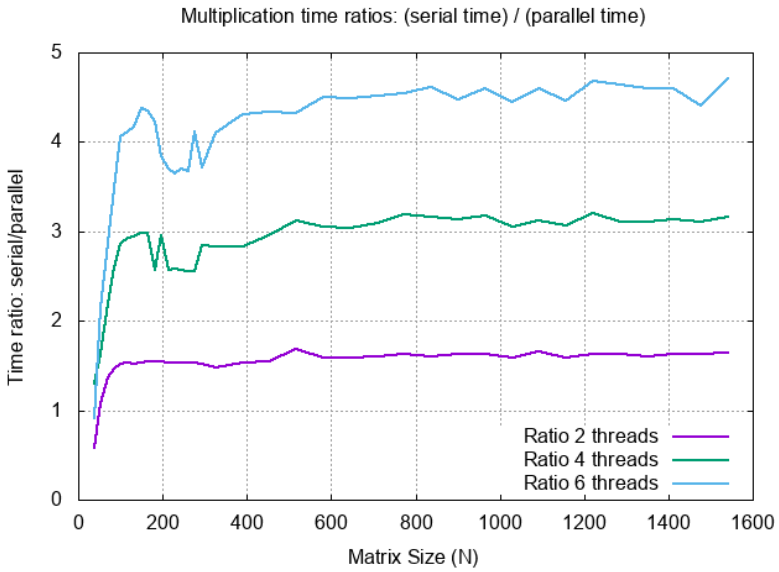

Figure 8: Time ratios extending the plot with lower values.

As we can see, with increasing number of threads, the sabilizing value increases slightly. For 2 threads, the value is around $N = 100$; for 4 threads, between $N = 130$ and $N = 400$, and for 6 threads, between $N = 160$ and $N = 450$.

4 Exercise 4: Example of numerical integration

4.1 *How many rectangles are used in the program? Which value does h take?*

The numer of rectangles is determined by the variable n, whose value is 10^8. The variable h takes the length of the segments in which the interval $[0, 1]$ is divided (the base of the rectangles), so it is $h = 1/n$. In this case, $h = 10^{-8}$.

4.2 *Run all versions of the program. Analyze the performance (mean execution time and speedup) and assessed whether the result the program yields is correct or not. Display all this information in a table and add brief explanation of why some programs are not giving the correct result*

The tests have been made using the *scr4_test.sh* script. We have submitted them to the cluster, in order to make a sufficiently large number of repetitions (50) and be able to get good results by making the means.

Program	Average exec. time (s)	Speedup	Result
pi_serie	1.36902	1	3.141593
pi_par1	0.581416	2.35463	3.141593
pi_par2	0.57015	2.40116	3.141593
pi_par3	0.183284	7.46939	3.141593
pi_par4	0.183334	7.46735	3.141593
pi_par5	0.183621	7.45568	3.141593
pi_par6	0.571115	2.3971	3.141593
pi_par7	0.185861	7.358725	3.141593

Figure 9: Times (seconds) and result of all the programs

As we can see, the results of the π approximation are the same in all the programs. (And it is the best approximation that can be expressed with the precission that these programs have by default, just 6 decimal places).

The best performance is given by versions *pi_par3*, *pi_par4*, pi_par5 and *pi_par7*, which have a speedup (with respect to *pi_serie*) higher than 7.On the other hand, the worst are *pi_par1*, *pi_par2* and *pi_par6*, with a speedup of just over 2.3.

4.3 *Regarding pi_par2, does it make sense to declare sum as private variable? What does it happen when you declare a pointer as private*

In pi_par2, the variable sum (a pointer) is marked with the clause *firstprivate*. As a result, each thread creates a copy of the pointer and the value its initialized with the one before the parallel region. So even though each thread has its own pointer, all of them point to the same memory position. Indeed, if the clause *private* is used instead of *firstprivate*, we would possibly get segmentation fault as the pointer in each thread would point to an "unknown" position outside the proccess memory region.

Regarding how this change affects the performance, as we expect, there is no difference between *pi_par1* and *pi_par2*; aside from *pi_par2* being slightly slower (by about 10 ms). Although we could think this higher execution time is caused because *pi_par2* has to do more operations when creating the threads, because of the allocation of this new variable for each thread, this is insignificant and the slightly higher execution time that we get may be just a coincidence.

4.4 *What are the differences between pi_par5, pi_par3 and pi_par1? Explain the concept of false sharing. Why does pi_par3 obtain the linesize of the cache?*

False sharing occurs when even though different threads are wiriting in different memory addresses, all of these addresses are contained in the same cache block. Every time one thread modifies its part of the block, the block becomes invalid for the rest of the threads, and threfore they need to replace the block in their cache.

Version *pi_par1* (and *pi_par2*) is a clear example of false sharing: the k-th thread modifies the posisiton k of the array *sum* provoking that the block is no longer valid for the rest of threads. This occurs for

every iteration of the loop and as result the execution time is higher than the one obtained with the serial version. At first glance, one could say that version pi_par4 is another example of false sharing as different threads modify the same block of an array. However, this time, this only happens when the value of *priv_sum* is copied to the array *sum*; that is to say, once for every thread. After that, threads do not access any data of that array again. So we can conclude that is not the case of false sharing. On the other hand, *pi_par3* still uses an array for storing the partial sums of each thread but avoids false sharing (as discussed in the next paragraph). Finaly, version *pi_par5* uses a private variable for storing the partial sums (*sum*) and after the calcultion each threads add the result to the shared variable *pi*. To avoid data races, that part of the code is encapsulated with the clause *critical* which ensures mutual exclusion between the threads.

The version *pi_par3* obstains the linesize of the cache to ensure that the positions of the array *sum* that are modified by each thread are located in different blocks. That is to say, the postion modifies by thread 1 will be placed in a different block that the position modified by thread 2 and so on. The small drawback is that we are allocating more memory than used: in each block (64 bytes), we are only using 8 bytes.

4.5 *What is the effect of using the pragma critical? How is the performance? Why does this happen?*

As mentioned before, this clause ensures mutual exclusion for an specific part of the code. This clause is used just before the threads end. By using this clause, false sharing is avoided and we get a speedup of 7.4.

4.6 *Regarding pi_par6, how is the performance compared to other versions? Why does this effect happen?*

The execution time for version pi_par6 is almost the same as the one for the first version. Indeed, both programs are an example of false sharing. In version 6, a new caluse is used: *for*. This clause divides the iterations of the loop among the threads, declare the index varibales as private... but inside the loop, once again each thread modifies a position of the array *sum* so it becomes unvalid for the rest of the threads.

4.7 *Which version is optimal and why?*

Accoriding to the results obtained, the best version in terms of execution time is the third one (by a slight difference of hundreds of seconds). However, as discused before, this version is allocating more memory than needed. So overall, one could say that the optimal version is either the forth, the fifth or the seventh.

5 Exercise 5: Optimization of calculation programs

5.0 *Compile and run the program using some images as arguments. Examine the results that were generated and analyze briefly the provided program.*

After executing the program, we see that we get three new images. A grey scale version, another in grey scale but with the edges significantly diferenciated and finally, a black and white image where the edges are clear.

The first step is to convert a RGB image into a greyscale one. To do so, the program iterates through all the pixels in the image and apply the following conversion: 0.2989R + 0.5870G + 0.1140B - where R,G,B are the values for each primary colour of the original image.

(a) (b)

Figure 10: (a) RGB image (b) Greyscale image

Then it applies the Sobel operator. This consist of placing two matrices

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

in every pixel (except those on the borders) and, for each matrix, we compute the sum (weighted by the matrix) of its neighbours elements. By applying each matrix, we get a measure of the variation of intensity near a pixel in the x and y direction. Then we calculate the norm of the 2 dimesion vector whose components are the numbers just obtained before; that is to say, we calculate the norm of the "intensity variation" near each pixel. With this procedure, we get an image in greyscale where the edges are clearly marked. However, because we are using 3x3 integers matrices as the convolute operator, it is likely to happen that some lines are shown as edges when they are not.

To deal with this problem, denoising is carried out. For this part of the program, a gaussian kernel is used to ponderate the results obtained with the Sobel operator for each pixel with its local neighbours results. If a pixel mets a threshold to be considerated part of an edge, it is assigned color white and, if not, black. With this procedure, we get rid of the noise (the light blurred inside the shapes in 11)

(a) (b) (c)

Figure 11: (a) Sobel in x direction (b) Sobel in y direction (c) Global Sobel

Figure 12: Imaged denoised

5.1 *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 may make sense to think that this loop could be parallelized when the task we want to perform is processing a high number of images. This way, each thread would process a different image at a time.

However, this may not be the optimal approach. For example, if the images vary in size, some thread would have a lot more work to do than others, wasting resources that coul help finish the task. There are other drawbacks, as explained in the next questions.

(a) *What happens if fewer arguments are passed than number of cores?*
The remaining cores will be idle, while the others work. This is a pretty inefficient way to distribute the tasks.

(b) *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.

5.2 *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.*

(a) *It is imperative that the program continue to perform the same algorithm, so only changes should be made to the program that do not change the output.*

(b) *Explain why the order is not correct if you change it.*

All nested loops in *main()* access data in the suboptimal order. For instance, accessing:

```
grey_image[j * width + i]
```
(1)

This is equivalent to accessing column i, row j. Therfore, the inermost loop should iterate on i, accessing contiguous data in each iteration of this loop. However, the loops in the provided code do exactly the opposite, which is why we have changed them.

In each of the nested loops inside *Loop 0* we can change the order, because the instructions performed inside the innermost loop are independent from one another, which means that the order in which they are performed does not alter the result. This is similar to what we did when changing the order in which the elements were added when calculating the sum of the elements of a matrix.

In the first loop (RGB to grey scale), an element of an array is accessed in each iteration (using 1). So there is no problem.

The next loop (Sobel edge detection) does something similar, writing in array *edges*. It also reads from array *grey_image*, but reading does not matter when we worry about changing order of insstructions. Again, there is no alteration in the result if we change the order of this loop.

The last loop is a nested loop on indices i and j, which contains another nested loop on indices p_1 and p_2. The loops on i and j, apart from iterating on p_1 and p_2, only write to an array: *edges_denoised*, in different positions each time. The p_1 and p_2 loops write into an array in order. This is something to take into account, as changing order in the indices would alter the result of this array; however, this array is later sorted, so the order in which it was filled is not relevant. At last, this loop can also be optimized without changing the result of the computation.

This improvement has been implemented in file *edgeDetector_optLoops.c*, and in all of the parallel improvements.

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

(a) *It is imperative that the program continue to perform the same algorithm, so only changes should be made to the program that do not change the output.*

(b) *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.*

5.4 *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*

5.5 *Something that we have left aside is to use compiler optimizations. Repeat the previous section adding the -O3 flag to the gcc command. Obtain information about the optimizations the compiler implements (and how they might affect our parallelization). Does the compile implement loop unrolling? Is this option activated? Does the compiler implement some kind of vectorization? Note: the -O3 option can generate warnings when compiled. Check that the output remains the same in any case.*