

Arquitectura de ordenadores. Practice 4.

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

December 15, 2021

Contents

0 Exercise 0: Information about the system topology	3
1 Exercise 1: Basic OpenMP programs	3
1.1 Is it possible run more threads than cores on the system? Does it make sense to do it?	3
1.2 How many threads should you use on the computers in the lab? And in the cluster? And on your own team?	3
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	3
1.4 How does OpenMP behave when we declare a private variable?	3
1.5 What happens to the value of a private variable when the parallel region starts executing?	3
1.6 What happens to the value of a private variable at the end of the parallel region?	3
1.7 Does the same happen with public variables?	3
2 Exercise 2: Parallelize the dot product	4
2.1 Run the serial version and understand what the result should be for different vector sizes.	4
2.2 Run the parallelized code with the openmp pragma and answer the following questions in the document. Is the result correct? What is happening?	4
2.3 Modify the code and name the program pescalar_par2. This version should give the correct result using the appropriate pragma:	4
2.4 Modify the code and name the resulting program pescalar_par3. This version should give the correct result using the appropriate pragma. Comparing with the previous point, which option will be chosen and why?	4
2.5 Run time analysis	5
2.6 (Optional) Exhaustive analysis	5
3 Exercise 3: Parallel matrix multiplication	7
3.1 Which of the three versions performs the worst? Why? Which of the three versions performs better? Why?	7
3.2 Based on the results, do you think fine-grained (innermost loop) or coarse-grained (outermost loop) parallelization is preferable in other algorithms?	8
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.	8
4 Exercise 4: Example of numerical integration	9
4.1 How many rectangles are used in the program? Which value does h take?	9
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	10
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	10
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?	10
4.5 What is the effect of using the pragma critical? How is the performance? Why does this happen?	11
4.6 Regarding pi_par6, how is the performance compared to other versions? Why does this effect happen?	11
4.7 Which version is optimal and why?	11
5 Exercise 5: Optimization of calculation programs	12
5.0 Compile and run the program using some images as arguments. Examine the results that were generated and analyze briefly the provided program.	12
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?	13
(5.1.a) What happens if fewer arguments are passed than number of cores?	13
(5.1.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.	14
5.2 During the previous task (task 3), we observed that the order of access to the data	

0 Exercise 0: Information about the system topology

We have used the lab computers, which have 6 cores, with hyperthreading not activated. (This is because in the file `/proc/cpuinfo` the number of physical cores is the same as the number of virtual cores, which is 6).

For the bigger tests, we used the *MV* machine from the cluster; it has 8 cores, we already mentioned this in last practice (3), exercise 0.

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 parallelize 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. Private variables 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); therefore, 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. Is the result correct? What is happening?

The parallel version is not working due to the fact that variable `sum` is being shared (default mode). So we have several threads writing in the same variable at the same time which leads to 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 a *pragma clause* just before the critical operation (the sum). Both of the mentioned *pragmas* solve the data race. Nonetheless, the way they achieved so is different. The *critical* 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.

These are the two modifications:

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

```
#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. Comparing with the previous point, which option will be chosen and why?

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.

The optimal version:

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

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 our own computers, which uses 4 threads) is very inconsistent. However, the first value that meets the requirements is $T = 1200000$:

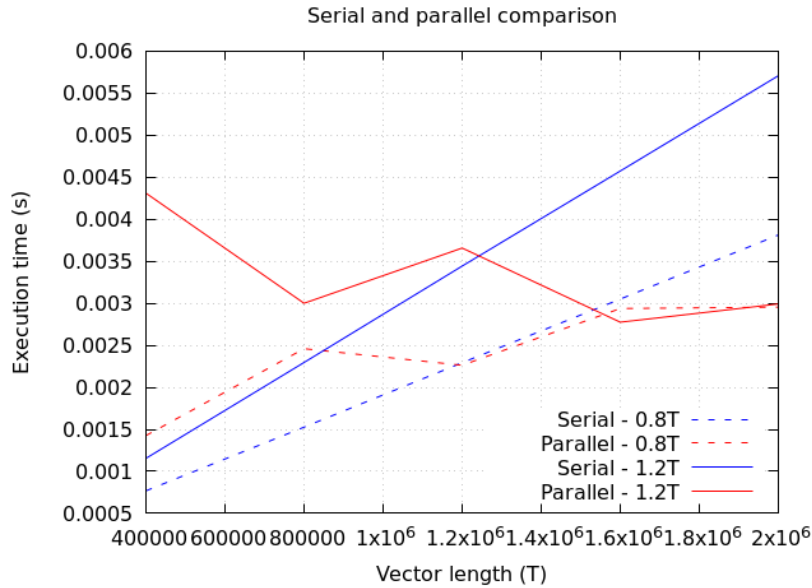


Figure 1: Search for threshold in the lab computers

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, approximately, $T = 81000$, as can be seen in the following figure (2):

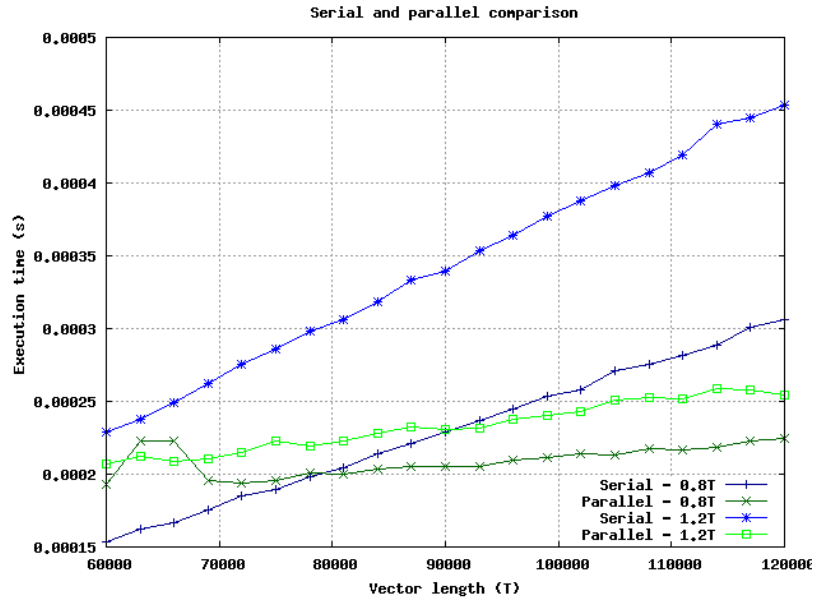


Figure 2: Search for threshold in the cluster: 4 cores

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.

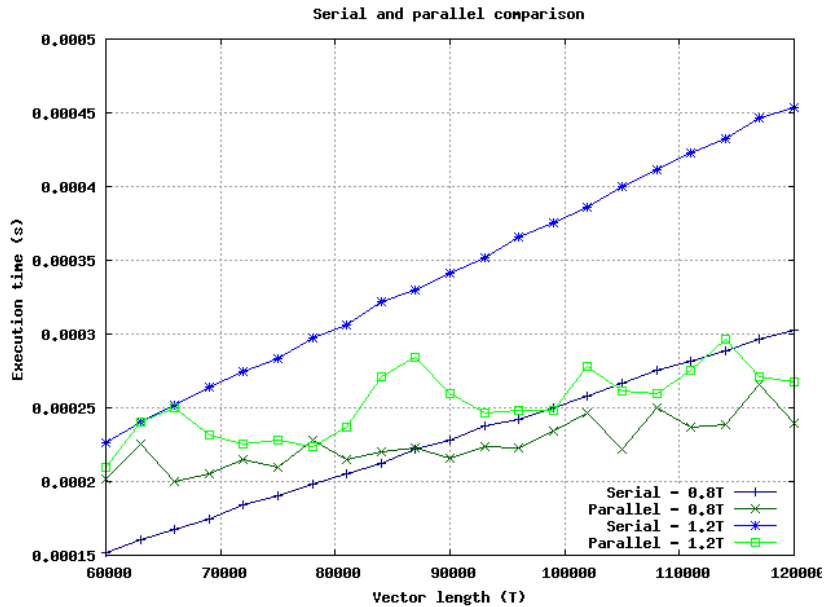


Figure 3: Search for threshold in the cluster: 8 cores

3 Exercise 3: Parallel matrix multiplication

Here are the requested tables for $N = 1000$, which do not have big enough times to see a significant difference in some cases. In fact, we can see the *loop_2* version being very similar to the *loop_3* one; however, that is not the case asymptotically (as can be seen in tables 3 and 4).

threads	1	2	3	4
serial	5.83421	5.83421	5.83421	5.83421
loop1	6.63794	6.45245	6.45245	5.8327
loop2	4.23795	2.901	2.901	2.24439
loop3	4.19559	2.89626	2.89626	2.23571

Table 1: Times (seconds) for $N = 1000$

threads	1	2	3	4
serial	1	1	1	1
loop1	0.878919	0.904185	0.904185	1.00026
loop2	1.37666	2.0111	2.0111	2.59946
loop3	1.39056	2.01439	2.01439	2.60956

Table 2: Ratios for $N = 1000$

For matrices of size $N = 5000$, the programs take high enough times to make a difference between the different implementations. Here are the results:

threads	1	2	3	4
serial	953.668	953.668	953.668	953.668
loop1	1004.25	656.298	467.513	384.674
loop2	1081.64	617.473	409.260	312.718
loop3	1002.03	501.907	349.435	268.127

Table 3: Times (seconds) for the requested tests ($N = 5000$)

threads	1	2	3	4
serial	1	1	1	1
loop1	0.949636	1.45310	2.03988	2.47916
loop2	0.881690	1.54447	2.33022	3.04961
loop3	0.951736	1.90009	2.72917	3.55678

Table 4: Ratios for the requested tests ($N = 5000$)

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. (However, *multiplication_loop2.c* “forks” N times).

In consequence, the best performing version is better than the others because it avoids the time cost caused by creating and destroying threads, as it only creates and destroys threads once. In all of the other aspects, they all perform the same operations.

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 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:

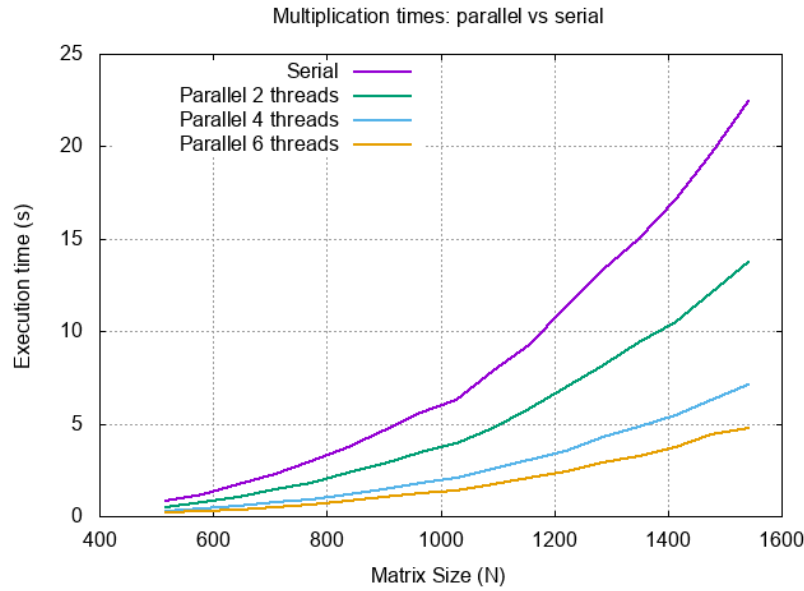


Figure 4: Time in seconds.

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 .

As commented before, for the sizes that we have tested, the asymptotic behaviour is already present (as N is big enough). The four time functions 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 5, 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:

As we can see, with increasing number of threads, the stabilizing 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 $N = 450$.

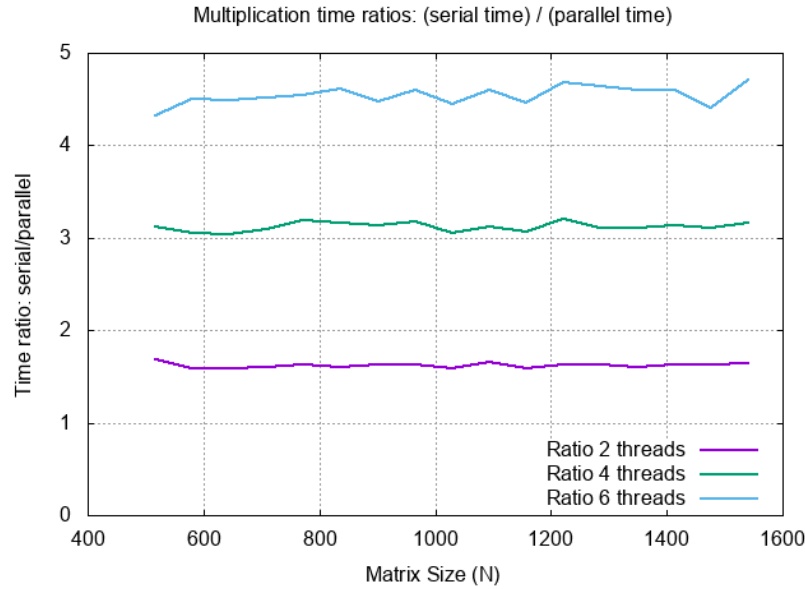


Figure 5: Time ratios.

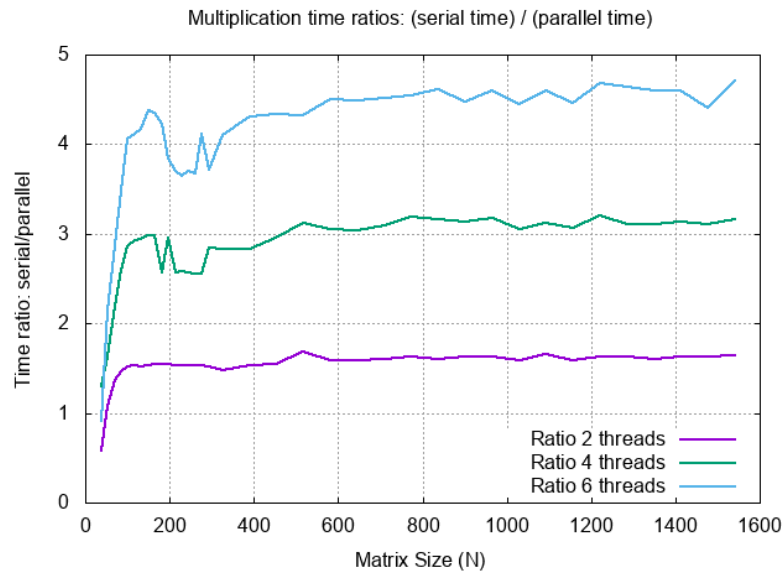


Figure 6: Time ratios extending the plot with lower values.

4 Exercise 4: Example of numerical integration

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

The number 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 7: 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 precision 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 a *segmentation fault* as the pointer in each thread would point to an “unknown” position outside the process 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 writing 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 therefore 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 position 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). Finally, version *pi_par5* uses a private variable for storing the partial sums (*sum*) and after the calculation 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* obtains 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 position modified by thread 1 will be placed in a different block than 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 a specific part of the code (in *pi_par5*). This clause is used just before the threads end. By using this clause, false sharing is avoided and we get a speedup of around 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 clause is used: *for*. This clause divides the iterations of the loop among the threads, declares the index variables as private... but inside the loop, once again each thread modifies a position of the array *sum* so it becomes invalid for the rest of the threads.

4.7 Which version is optimal and why?

According 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 discussed before, this version is allocating more memory than needed. So overall, one could say that the optimal version is either the fourth, 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 differentiated 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 applies the following conversion: $\text{grey value} = 0.2989R + 0.5870G + 0.1140B$ - where R,G,B are the values for each primary colour of the original image.



Figure 8: (a) RGB image (b) Greyscale image

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

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

in every pixel (except those on the borders) and, for each matrix. computing 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 3×3 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 considered 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 figure 9)

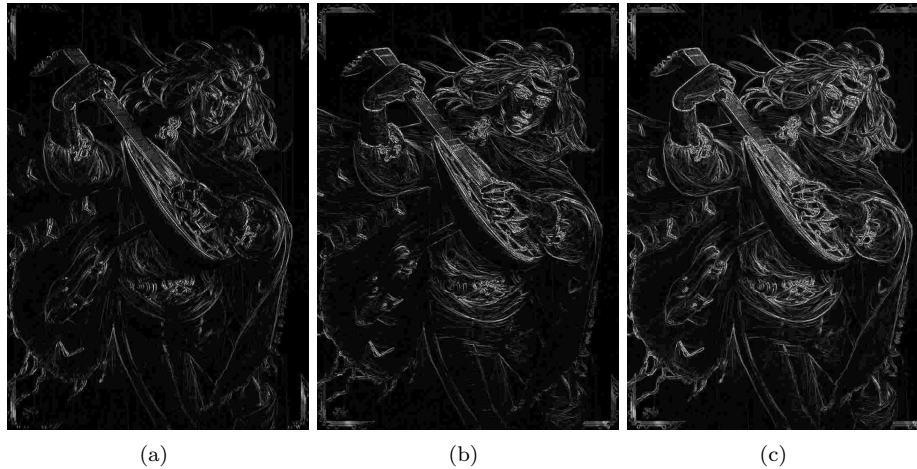


Figure 9: (a) Sobel in x direction (b) Sobel in y direction (c) Global Sobel



Figure 10: 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 could help finish the task. There are other drawbacks, as explained in the next questions.

(5.1.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.

(5.1.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.

Each thread would need to allocate memory for the RGB image and for the three new images created during the algorithm. In an RGB image, each pixel is represented with 3 bytes, one for each colour. So, if the images occupy up to 6GB, the corresponding number of pixels per image is given by: $6\text{GB} * \frac{2^{30}\text{B}}{1\text{GB}} * \frac{1\text{pixel}}{3\text{B}} = 2^{31}$ pixels. (The program also accepts RGBA images which have an extra byte per pixel for the alpha channel).

Then, each thread would need to allocate memory for the three new images it creates (grey, edges and edges.denoised version). All of them consist of an array of bytes, one for each pixel. So it would sum up to 3 grey images $* \frac{2^{31}(1\text{ channel})\text{ pixel}}{1\text{ grey image}} * \frac{1\text{B}}{(1\text{ channel})\text{ pixel}} = 6\text{GB}$ of memory needed (apart from loading the original image of 6GB). There are other memory allocations but are insignificant when compared to the ones just commented. To sum up, every thread would need the double of memory of the image it is processing. Furthermore, this memory is freed at the end of the algorithm instead of just after where it has been used, so the total consumption of memory of all the threads together would be even greater.

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.

(5.2.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.

(5.2.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 . Therefore, the innermost 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 instructions. Again, there is no alteration in the result if we change the order of this loop.

Lastly, in the *denoising* filters, there 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. In the case of the gaussian filter, the loops on indices p_1 and p_2 are used to perform an addition over the accumulator *sum*; because the sum is commutative, there is nothing to worry about.

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

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

(5.3.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.*

We have done 3 different versions. All modifications are made to the version that uses the optimal ordering of indexes in the loops (which is written in *edgeDetector.optLoops.c*).

The first modification, *par0* is such that each operation: rgb to gray, sobel transformation and gaussian denoising, has a parallelization of the outermost loop.

In the second version, *par1* we have done it such that each operation has a parallelization of the outermost loop, which is forced to create 4 threads (using `num_threads()`); and the second loop has a parallelization which is forced to create 2 threads. In total, this version creates the same number of threads (8) for each operation as *par0*, because we are executing with the MV machine, using 8 cores.

The last version, *par2*, is similar to *par0*, parallelizing the outermost loops. This time, however, we use the clause `collapse(2)`, which has been implemented specifically to parallelize 2 nested loops of this kind: “square loops”.

(5.3.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.*

Of the first 2 versions, it is expected that *par0* will be the best one. This is because parallelizing the outermost loop has the best results, as we have seen in exercise 3. On the other hand, there should be no difference between *par2* and *par0*.

The reason why using `collapse` makes no difference relies on the number of iterations of the nested loops, in other words, the size of the image. `collapse` makes sense when the outer loop has fewer iterations than threads available (so it prevents some thread(s) from being idle) or when the workload is uneven among iterations. But in this case, where all threads are going to be busy at all times because the number of iterations of the first loop is much higher than the number of threads, *par2* and *par0* are going to have similar performances.

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*

For obtaining the results of this exercise, we have run *scr5.sh*. In order to illustrate the improvement of optimizing the index order in the nested loops, we have included the times of the original version (*edgeDetector*). However, the speedups are calculated using the one with optimized loops (because we are parallelizing optimized loops).

Program	SD	HD	FHD	UHD	UHD-4k	UHD-8k
edgeDetector	0.0773424	0.17391	0.396718	1.7948	2.31059	6.93214
optLoops	0.0743518	0.165544	0.37326	1.50866	1.61268	6.00717
par0	0.0193453	0.0308818	0.0781132	0.285718	0.300107	1.11034
par1	0.058066	0.0861351	0.158138	0.562464	0.60438	1.98864
par2	0.0146477	0.0308326	0.0704847	0.280482	0.297517	1.10992

Table 5: Execution time (seconds)

Program	SD	HD	FHD	UHD	UHD-4k	UHD-8k
edgeDetector	0.961332	0.951894	0.940869	0.840572	0.697951	0.866567
optLoops	1	1	1	1	1	1
par0	3.8434	5.36057	4.77845	5.28024	5.37368	5.41021
par1	1.28047	1.92191	2.36034	2.68223	2.66832	3.02074
par2	5.07601	5.36912	5.29562	5.37881	5.42046	5.41225

Table 6: Speedups

Note: in the table, speedups are calculated by columns, comparing the computations of an image with the serial version of the same image.

The speedup of *par1* clearly grows from 1.28 to 3.02 as the image size increases.

Speedups of *par2* are quite stable around 5.0-5.4. The speedups of *par0* are similar, but for images SD and FHD they fall to 3.84 and 4.78, respectively. In both of these versions, we cannot clearly observe a higher speedup for bigger images.

Finally, here (table 7) we have the fps of each program, for each image size.

$$\text{fps} = (\text{time in seconds per image})^{-1}$$

Program	SD	HD	FHD	UHD	UHD-4k	UHD-8k
edgeDetector	12.9295	5.7501	2.52068	0.557165	0.43279	0.144256
optLoops	13.4496	6.04069	2.6791	0.66284	0.620086	0.166468
par0	51.6921	32.3815	12.8019	3.49995	3.33214	0.900625
par1	17.2218	11.6097	6.32359	1.77789	1.65459	0.502856
par2	68.2701	32.4332	14.1875	3.56529	3.36115	0.900966

Table 7: Frames per second

As expected, *par0* is much better than *par1* (we commented this in question (5.3.b)). As we also commented, *par2* and *par0* perform similarly. In each of the images, the results obtained by *par2* are slightly better; the difference being very small for big image sizes (UHD, 4k, 8k), and higher for smaller sized images: with a difference of a 24.3% in the case of SD quality.

The best version, *par2*, could process images from a video stream in real time for videos of SD or HD definitions. If we increased the definitions of the frames, our version could not process the rate of 30 fps. In fact, for the largest resolutions, our programs cannot even process one image per second.

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 compiler 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.*

When the flag -O3 is included for the compilation, many optimizations are carried out; however, loop unrolling is not one of those as shown in the next image (result of adding -Q -help=optimizers in the Makefile to see the enabled ones). If we want the compiler to unroll loops we have to add one of these flags: -funroll-loops or -funroll-all-loops.

```
$ make -f Makefile2 | grep unroll
-fsplit-ivs-in-unroller      [enabled]
-funroll-all-loops          [disabled]
-funroll-loops               [disabled]
-fvariable-expansion-in-unroller [disabled]
```

On the other hand, the optimizations related to vectorization are the following:


```
$ make -f Makefile2 | grep vector
-ftree-loop-vectorize      [enabled]
-ftree-slp-vectorize       [enabled]
-ftree-vectorize           [disabled]
Specifies the cost model for vectorization. -fvect-cost-model=[unlimited|dynamic|cheap]
dynamic
```

Which means that some kind of vectorization is implemented. However, loop unrolling is disabled.

In any case, the resulting images that are output by the compiled with `-O3` programs are the same as the ones output by the normally compiled programs.

If we compare these results with the ones from the previous section, we notice that the average execution time has decreased by a factor of (roughly) 3 for all the versions and image resolutions. This behaviour is better observed in the fps tables: table 7 and table 10.

Program	SD	HD	FHD	UHD	UHD-4k	UHD-8k
optLoops	0.0384636	0.0841784	0.188973	0.762204	0.808152	3.05029
par0	0.0071536	0.0148413	0.0260301	0.14612	0.125468	0.437312
par1	0.033679	0.0386352	0.0736955	0.225493	0.243872	0.769885
par2	0.00591487	0.0122253	0.0262656	0.165792	0.129966	0.422036

Table 8: Execution time (seconds) (compiling with `-O3`)

Program	SD	HD	FHD	UHD	UHD-4k	UHD-8k
optLoops	1	1	1	1	1	1
par0	5.37682	5.6719	7.25979	5.21629	6.4411	6.97509
par1	1.14206	2.1788	2.56424	3.38017	3.31384	3.96201
par2	6.50286	6.88559	7.1947	4.59735	6.21818	7.22756

Table 9: Speedups (compiling with `-O3`)

This time, we can see a higher variance in the speedups of both *par0* and *par2*. However, *par1* increases the speedup with the image size, just like in the previous question (5.4).

It is also worth noting that there are images in which *par0* outperforms *par2*.

Program	SD	HD	FHD	UHD	UHD-4k	UHD-8k
optLoops	25.9986	11.8795	5.29176	1.31198	1.23739	0.327838
par0	139.79	67.3795	38.4171	6.84369	7.97016	2.2867
par1	29.6921	25.8831	13.5693	4.43473	4.10051	1.2989
par2	169.065	81.7976	38.0726	6.03165	7.69432	2.36947

Table 10: Frames per second (compiling with `-O3`)

Once again, if we only focus on the performance of the best versions; comparing this to the table from last section, we see that the fps rate has increased by a factor of 3, which would allow us to process videos in real time (30 fps) for resolutions SD, HD and FHD. However, we are still unable to process video for higher image definitions. In order to achieve so, further and more extensive parallelization would be needed as well as hardware of higher features; for instance, a larger number of cores.