# Task 4: Exploiting the potential of modern architectures

**NOTE: Read carefully the statement of each exercise before proceeding to do it.**

The objective of this practice is to experiment and learn how to take advantage of *multicore* (or *manycore*) architectures through the OpenMP programming f*ramework* [1].

All the work associated with this practice should be done in a Linux environment. The student can use the computers in the laboratories, or the ARQO cluster. It will also be possible to use your own computer. In any of the cases, it is necessary to state in which equipment/s this task has been executed. If it is not being said, any reasonable thread number can be used. Testing different configurations will be positively evaluated.

## MATERIAL

Along with the task statement, the file *"MaterialP4.zip"* is delivered, which contains the following files that will be referenced throughout the statement:

- `omp1.c` – file with the example code of a program that creates threads using OpenMP. The number of threads created is passed as an argument to the program.
- `omp2.c` – file with the example code of a program that shows the differences in the privacy declaration of variables in OpenMP.
- `pescalar_serie.c` – file with a serial code that executes the dot product of two vectors.
- `pescalar_par1.c` – file with the sample code of program that performs loop parallelization with OpenMP.
- `Makefile` – makefile used to compile all source code files. It must be modified so that the requested programs are also compiled.
- `pi_serie.c` – file with a serial code that performs the calculation of pi by numerical integration.
- `pi_par1.c` a `pi_par7.c` – files with different implementations in OpenMP to calculate pi by numerical integration.
- `edgeDetector.c` – program that applies an image processing algorithm to detect edges.

## IMPORTANT

In this document we refer to a P number that will affect the experiments and results that will be obtained. The value of P to be used must be equal to the number of the team modulo 8 plus 1 (that is, it will be a number between 1 and 8).

## Exercise 0: Information about the system topology

First of all, we should be able to obtain the information related to the architecture of the machine we are working on. This is easy on a Linux system and we can do it by executing any of the following commands:

```
> cat /proc/cpuinfo

> dmidecode
```

**NOTE:** the last command requires superuser permissions, so we will NOT be able to use it in the lab's installation. You can use the lstopo command instead. In the cluster it is installed in /share/apps/tools/hwloc/bin/lstopo.

If we look at the information that the `cpuinfo` file gives us, we can deduce how many *cores* our computer has and whether or not the *hyperthreading* option is enabled (the `cpu_cores` parameter indicates how many physical CPUs there are in the computer in total, and the `siblings` parameter refers to the number of virtual CPUs). In a computer with more than one processor, the `physical_id` parameter differentiates the processor where each core is.

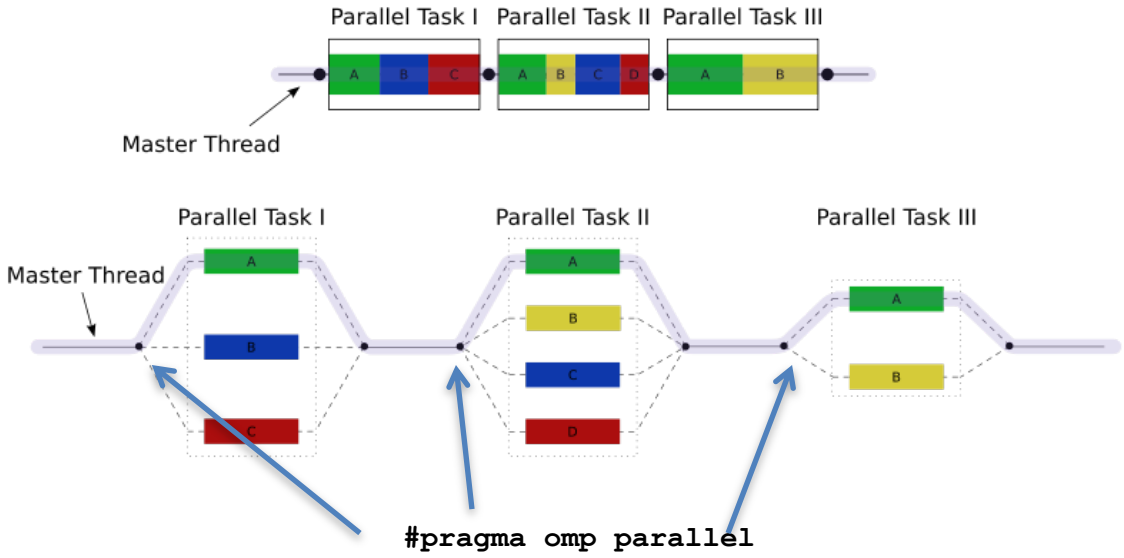
Based on the information obtained, indicate in the document associated with this task, the information related to the number and frequency of the processors available in the equipment used, and whether or not the *hyperthreading* option is active.

**Note:** you can switch to different cluster nodes (from the frontend) with `ssh compute-x-y`, to check their architecture. There are differentiated nodes for Intel (compute-0-0) and AMD nodes (compute-0-7) with different numbers of cores.

## Exercise 1: Basic OpenMP programs (1 p)

In this exercise, we are going to make a first contact with OpenMP-based programs. To compile these programs, we will use `gcc` as a compiler, adding the `-lgomp` and `-fopenmp` flags to indicate that we want to use OpenMP libraries and extensions.

OpenMP allows us to launch in a very simple way a number of threads (or threads) to execute tasks in parallel (each thread executing on its corresponding CPU). The creation of a thread team is done each time the `#pragma omp parallel` directive is included in the program, and the synchronization and termination of the threads is done when the parallel region indicated with the `parallel` ends.




In the material, you are asked to compile and run the `omp1.c` program to check how the creation and execution of thread teams is done, and to become familiar with the `omp_get_num_threads ()` and `omp_get_thread_num ()` functions that allow us to identify each of the active threads.

1.1 Is it possible run more *threads* than *cores* on the system? Does it make sense to do it?
1.2 How many threads should you use on the computers in the lab? And in the cluster? And on your own team?

The number of threads that are created when opening a parallel region can be selected in the following ways:

With an environment variable:

```
OMP_NUM_THREADS
```

with a function:

```
omp_set_num_threads(int num_threads);
```

with a clause inside the parallel región:

```
#pragma omp parallel num_threads(numthr)
```

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.

A fundamental characteristic when programming using OpenMP is to correctly define which variables must be *public* (accessible to all threads, it is the default value) or *private* (each thread accesses its own copy). An example of using these features can be found in the `omp2.c` program.

It is requested to execute the `omp2` program and include the output in the document of the task. Based on the output obtained (and after reading and understanding the source code that generates it), answer the following questions:

1.4 How does OpenMP behave when we declare a private variable?
1.5 What happens to the value of a private variable when the parallel region starts executing?
1.6 What happens to the value of a private variable at the end of the parallel region?
1.7 Does the same happen with public variables?

## Exercise 2: Parallelize the dot product (2 p)

We will take as a reference code the scalar product whose serial version and an attempted parallelized code are delivered as material for the task. OpenMP is specially designed to parallelize the work of a *for* loop between a team of threads. When you want to distribute the work of a loop in this way, the clause `#pragma omp parallel for` is used. By indicating this clause, OpenMP already does work for us, such as declaring the variable used as the loop index as private.

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

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

- Is the result correct?
- What is it happening?

Before trying to decrease the execution time, it is important that the result is correct. To do this, you must guarantee that the accumulation used in this code does not have problems due to data races.

2.3 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.
- What is the chosen option and why?

2.4 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
```

- Comparing with the previous point, which option will be chosen and why?

**2.5 Run time analysis.**

It is quite frequent that when developing an algorithm, we want it to have the best performance no matter which the inputs are. Thus, parallelization must be limited to situations where the overhead of launching threads is depictable with respect to the performance gain. Luckily, some of the pragmas in OpenMP allow us to specify a conditional parallelization. For instance, the example below starts a parallel region only if the condition is met.

```
#pragma omp parallel if (M>threshold)
```

Specifically, we want to estimate this quantity threshold assuming M is the size of the vectors. For that purpose, an iterative method is proposed where students should increase or decrease the input size (an argument of the program) until the point we want is found.

An initial value will be proposed by the students, large enough so that time results are significant. Depending the result of the next two conditions, students will raise or reduce the size until finding an optimal value. Since some numerical instability could happen, a threshold $T$ will be assessed as correct if:

1. The mean execution time of serial version is less than the mean execution time of the parallel version for vectors of size $ceil(0.8T)$.
2. The mean execution time of serial version is greater than the mean execution time of the parallel version for vectors of size $ceil(1.2T)$.

Students are required to provide three different values of T: the correct one, one below the optimal threshold and one above the optimal threshold. For each of them, conditions 1 and 2 should be evaluated.

**2.7 (Optional) Exhaustive analysis (Up to 1 extra point)**

Previous exercise is expected to be performed manually with few iterations. However, any kind of automatization of this process or using more exhaustive algorithms such as grid exploration (testing a grid of values) will be assessed. Please also use any plot to support your arguments.

Is it the threshold for your computer and the cluster the same? Does it depend on the number of threads?

## Exercise 3: Parallel Matrix multiplication (3 p)

For this exercise, the starting code for matrix multiplication is the one developed in task 3. In this case, the program consists of three nested loops, so when implementing the parallelization of the program it is reasonable to ask: which of the loops has to be parallelized?

It is requested to implement and submit (in addition to the serial version of task 3) three parallel versions of the matrix multiplication paralleling the three existing loops. All versions must work correctly.

It is requested to fill in the following tables, performing the execution for matrices of size large enough so that the execution time values allow to see differences between the different cases studied (for example, start evaluating a size of 1000x1000 that takes about 10 seconds and try at least one other size that takes 60 seconds):

**Execution time (s)**

| Version \ # threads | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Serie | | | | |
| Parallel – loop1 | | | | |
| Parallel – loop2 | | | | |
| Parallel – loop3 | | | | |

***Speedup*** **(taking as reference the serial version)**

| Version \ # threads | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Serie | 1 | | | |
| Parallel – loop1 | | | | |
| Parallel – loop2 | | | | |
| Parallel – loop3 | | | | |

**NOTE:** Loop 1 is considered the innermost, Loop 2 is the middle loop, and Loop 3 is the outermost loop. In the serial case, complete only the column for one thread.

Considering the results obtained, answer the following questions:

3.1 Which of the three versions performs the worst? Why? Which of the three versions performs better? Why?
3.2 Based on the results, do you think fine-grained (innermost loop) or coarse-grained (outermost loop) parallelization is preferable in other algorithms?

Taking as reference the execution times of the serial version and the best parallel version obtained previously (the best combination of the three code versions, and the C possibilities for the number of parallel threads), take times in a file and make a chart of the evolution of the execution time and the speedup (parallel version vs serie) as the size of the NxN matrices for N varies between 512 + P and 1024 + 512 + P (with increments of 64). After including these two charts in the document, include a paragraph describing and justifying the behavior observed in them.

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.

## Exercise 4: Example of numerical integration (2 p)

We are going to work throughout this session with an algorithm to obtain an approximation of the value of the number π through numerical integration.

## Method summary

In particular, we want to compute the area below function $f$ in the interval $[0, 1]$ approximating it by rectangles. As an example, next equation computes the estimate for first rectangle:

$$\int_0^h f(x)dx \approx h \cdot f\left(0 + \frac{h}{2}\right)$$

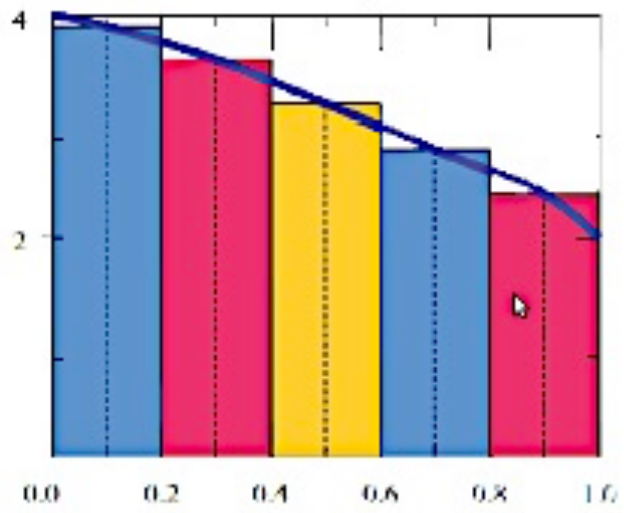
Repeating this for all rectangles, we arrive to:

$$\int_0^1 f(x)dx = \sum_i \int_{h \cdot i}^{h \cdot (i+1)} f(x)dx \approx \sum_i h \cdot f\left(h \cdot i + \frac{h}{2}\right)$$

If we choose $f(x) = \frac{4}{1+x^2}$, it can be shown that its integral is π:

$$\int_0^1 \frac{4}{1+x^2}dx = 4\int_0^1 \frac{1}{1+x^2}dx = 4[\arctan(1) - arctan(0)] = 4\left[\frac{\pi}{4} - 0\right] = \pi$$

Next figure represents the algorithm for $h = 0.2$.




The narrower these rectangles are, the greater the approximation to the real result, and therefore to the number π. An example of a program that performs this task can be found in the file `pi_serie.c` that can be found in the material folder for this task.

## Loss of performance due to the effect of *false sharing* in OpenMP

**Question 4.1:** How many rectangles are used in the program to perform the numerical integration? Which value does $h$ take?

**Questions 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-

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

**Questions 4.4**: What are the differences between pi_par5, pi_par3 and pi_par1? Explain the concept of false sharing and indicate all the versions that might be affected by it (and to what extent). Why does pi_par3 obtain the linesize of the cache?

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

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

**Questions 4.7**: Which version is optimal and why?

## Exercise 5: Optimization of calculation programs (4 points)

One of the advantages of OpenMP is that it is extremely easy to apply to computationally intensive programs without requiring a complete change in architecture or design. In this last exercise, it is proposed to apply all the knowledge obtained during the task to optimize and parallelize a calculation program.

The file `edgeDetector.c` includes a program that applies an image processing algorithm to detect edges. This algorithm has been provided to us, as a proof of concept, with the idea that it finally processes images from a video stream in real time. To do this, students are reminded that a video is usually composed of approximately 30 fps (frames per second). That is, the program would have to process 30 images in one second.

Answer the questions reasonably:

0. Compile and run the program using some images as arguments. Examine the results that were generated and analyze briefly the provided program.
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?
    a. What happens if fewer arguments are passed than number of cores?
    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.
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.
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.
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. 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.

## MATERIAL TO SUBMIT

A document with the answers to the questions throughout this statement.
In independent folders, the files with the data obtained in the experiments of the exercises that indicate it, and the scripts used to obtain these data (if any have been used).
Source codes implemented or modified to solve the exercises.

The submission will be made through Moodle, on the date indicated for this task.

## REFERENCES

[1] **OpenMP: Tutorials and technical articles** http://openmp.org/wp/resources/#Tutorials