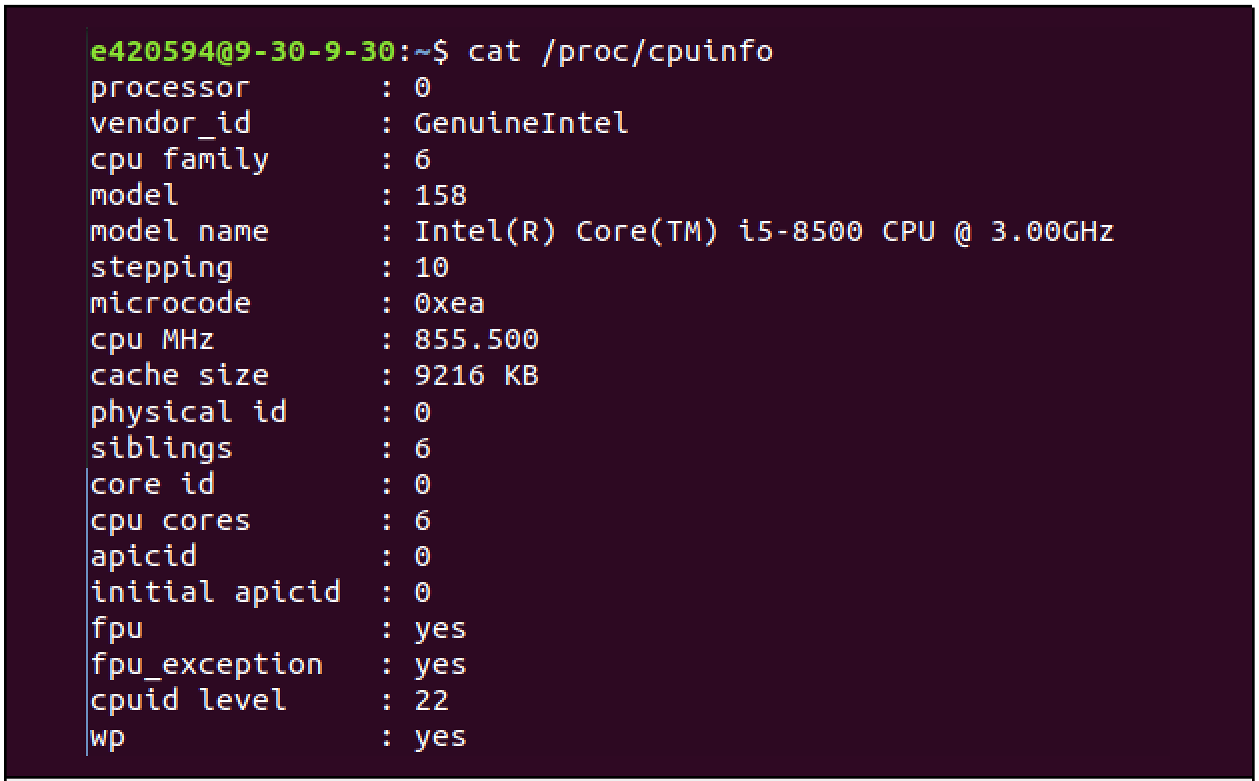
Arq0 Practice 4

Parallel processing in OpenMP

By Daniel Varela & Guillermo Martín-Coello

Experiment 0
Introduction to parallel programming

The first experiment is about finding the information about the system topology. To obtain the architecture of our machine we checked the linux /proc/cpuinfo file.



```
e420594@9-30-9-30:~$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 158
model name      : Intel(R) Core(TM) i5-8500 CPU @ 3.00GHz
stepping        : 10
microcode       : 0xea
cpu MHz         : 855.500
cache size      : 9216 KB
physical id     : 0
siblings        : 6
core id         : 0
cpu cores       : 6
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
```

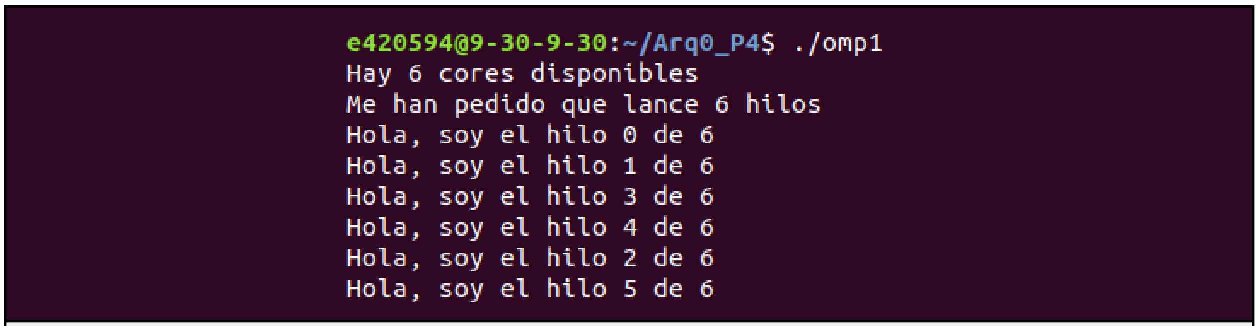
Image I: Terminal output

As we can see, our machine has 6 cpu cores without hyperthreading enabled. This is shown in the "siblings" parameter, where the amount of virtual CPUs is shown. When the number of cpu cores and virtual cores are the same, there is no hyperthreading. The machine only has one processor so the processor id shown is 0. The frequency of the processor is 855.500 MHz.

Experiment 1

Introduction to threading with OpenMP

The first experiment considers aOpenMP-based programs. This library provides tools to execute programs with multiple threads.



```
e420594@9-30-9-30:~/Arq0_P4$ ./omp1
Hay 6 cores disponibles
Me han pedido que lance 6 hilos
Hola, soy el hilo 0 de 6
Hola, soy el hilo 1 de 6
Hola, soy el hilo 3 de 6
Hola, soy el hilo 4 de 6
Hola, soy el hilo 2 de 6
Hola, soy el hilo 5 de 6
```

Image II: Terminal output II

1.1 Run more threads than cores.
It is possible to run more threads than cores there are in the system, nevertheless it does not provide any improvement. This is because OpenMP-based programs have the objective of parallel execution, providing parallel processing with all available cores, and the amount of threads that can be executed parallelly is equal to the amount of cores the computer has.

1.2 Number of threads.
According to the answers of questions 1.1 and question 0, we can determine that the amount of threads that should be used in the lab computers is 6 because they have a total of 6 cores.
In the cluster, to know the amount of threads that should be used, we launch the same command as we did in exercise 0 and we obtain that the main queue of the cluster has 16 cores. If the same process is followed for the intel cluster processors, we obtain that a total of 12 threads should be used, and for the AMD cluster processors, 16 threads.
To determine the amount of threads that should be used in our own computer, we check for the amount of cores it has and obtain cores, which should be the amount of threads we should utilize.

1.3 Priority between the number of threads.
We modified the program to use three different ways to assign the number of threads and tested the results to discover the priorities they have. The results we obtain were the following (in descending priority):

1	2	3

`//Option 3: parallel region`
`#pragma omp parallel num_threads(6)`
	`// Option 2: function`
	`omp_set_num_threads(arg);`
		`// Option 1: enviroment variable`
		`#define OMP_NUM_THREADS 8`

We tested this by executing the program omp1.c with the three types of assignments at the same time, to find out the highest priority assignment (parallel region), then we removed that one and obtained the second priority one (function), leaving the environment variable on the lowest priority assignment.

1.4 Private variables behavior
When a private variable is defined, OpenMP stores a copy of that variable on each thread so the threads can access that variable independently from each other.

1.5 Private variables behavior while parallel region execution
When the parallel region is executed and a private variable is declared, each thread creates a new private variable pointing to a different memory direction. When private is used, the variable will not be initialized, but when firstrivate is used, the variable will maintain the original value that it had previous to the parallelization.

1.6 Private variables behavior pursuing parallel region execution
Following the parallel region execution, private variables restore their original pointer and value.

1.7 Public variables behavior
Public variables maintain the memory direction and value from before the parallel region for all the threads, meaning that when a thread modifies the value and another thread wants to retrieve the value, it will get the one modified by the previous thread. These types of variables need to have concurrency locks so there isn't any issue while modifying or accessing the values at the same time.

Experiment 2

Threshold of parallelization

The second experiment covers the dot product. The materials for this experiment are two versions of the dot product algorithm, one serial and the other one parallelized with a team of threads. To distribute the work of a loop to the threads we use the clause:

```
#pragma omp parallel for
```

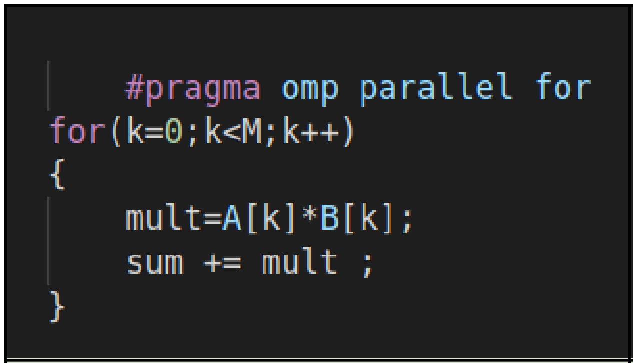
2.1 Serial version of scalar vector multiplication.
The program pescalar_serie.c generates two vectors of a given size and a value of 1 on each position and proceeds to calculate their scalar product. The results obtained show that the scalar product increases when increasing the size. The final scalar product is always the same as the vector length because all the values from both vectors are always one.

2.2 Parallelized scalar vector multiplication.

2.2.1 Parallelized scalar vector multiplication results.
The result we obtain is incorrect and it varies with each execution.

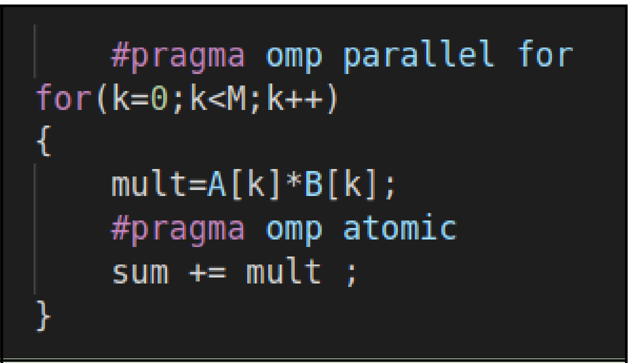
2.2.2 Parallelized scalar vector multiplication explanation.
The result in pescalar_par1.c is wrong because the program uses public variables declared in the main program and it doesn't protect them from concurrency. This way, since the variable that stores the result is accessed on assignment parallely, when different threads access the memory to retrieve it, they might be getting a wrong value for the variable because there might be another thread editing it.

2.3 New parallelized scalar vector multiplication.
The code was modified and tested with both the critical and atomic pragma directives.
2.3.1
Both pragma directives accomplish their function of isolating the operations so the concurrency does not affect the result. Nevertheless, the get to the same result by two different ways: While Critical works as a mutex not allowing the next operation until the one in progress is done, Atomic works at a deeper level obtaining the variable only before or after the operation is made, not while it is in progress. This makes atomic slightly more efficient in cases like the one we have. Another important difference is that atomic only works with a certain kind of operation (read, write, update…) while critical works with all of them.



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

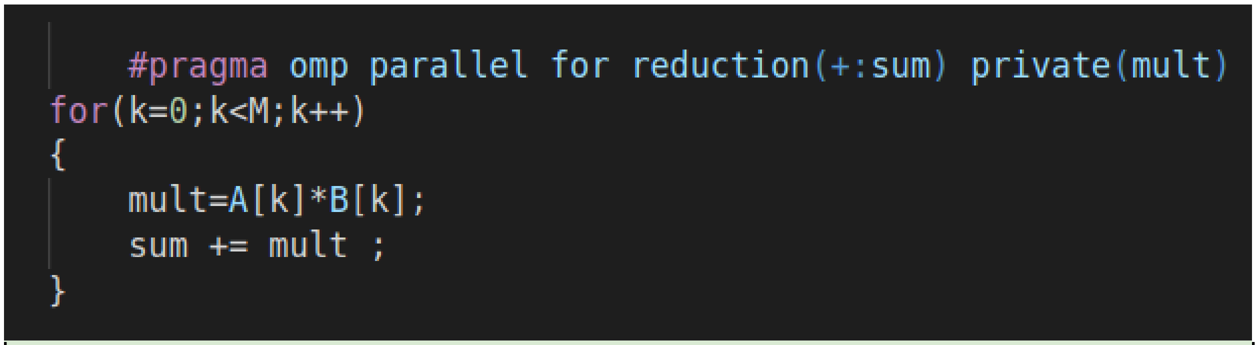
Image I: Parallel for code



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

Image II: Parallel for code with atomic statement

2.4 Parallel for reduction pragma.
Reduction is more efficient due to it allowing multiple threads to execute at once the code and it adds all the results of each thread to the variable sum afterwards. With the previous options, on the other hand we are slowing down the process by blocking threads from retrieving the variable while another thread is modifying it.

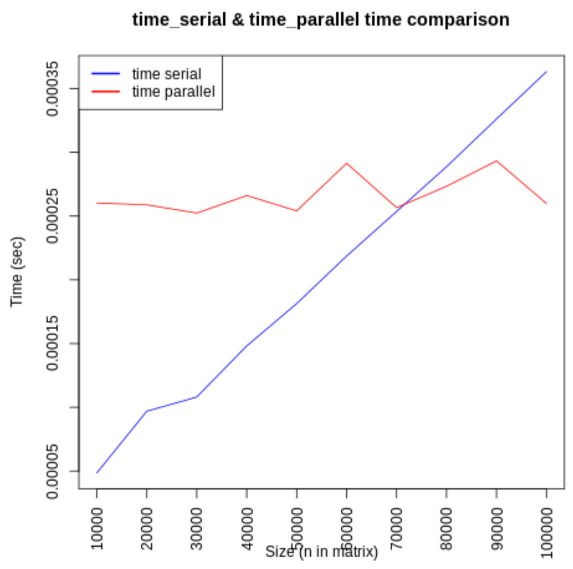
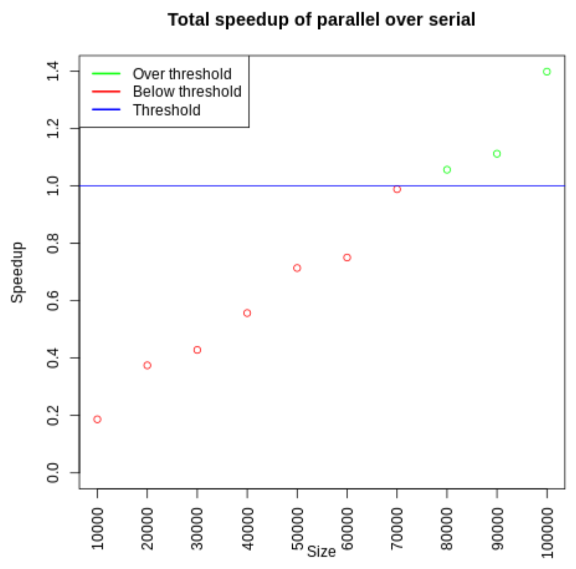


```
    #pragma omp parallel for reduction(+:sum) private(mult)
for(k=0;k<M;k++)
{
    mult=A[k]*B[k];
    sum += mult ;
}
```

Image III: Reduction parallelization

2.5 Runtime analysis.
The use of parallelized programs is not always the most efficient solution due to the fact that the overhead of creating threads might slow down the program more than speeding it up afterwards. This means there is a vector size from which it's worth to execute the program with various threads. This vector size is called threshold, to calculate it, we created a script that makes an average time execution for parallel and serial programs and it increases the vector size. When we plot the results, it's possible to see that the threshold is somewhere around the size 70000. It's also important to remark that the amount of threads that we throw affect also the result making it faster the greater number of threads we throw until we reach the number of threads that equals

the number of cores, where for reasons explained in exercise 1, the program speed does not improve but it gets worse because of the overhead cost of throwing unnecessary unused threads.

Graph I: Time serial and parallel over matrix size	Graph II: Parallel speedup and serial threshold
	

Experiment 3

Thread study

Execution times (s)

Threads	1	2	3	4
Serial	5.80289720	-	-	-
Parallel 1	7.67649870	5.83438100	5.63862540	5.69370480
Parallel 2	7.66472490	3.53041850	2.40966370	1.81598500
Parallel 3	7.59627830	4.15874440	2.83718030	2.06745520

Table I: Matrix multiplication results with different parallel structures and amount of threads. Size = 1000 x 1000

Speedup (with serial as reference)

Threads	1	2	3	4
Serial	1	-	-	-
Parallel 1	1.322873461	1.005425531	0.9716914165	0.9811831235
Parallel 2	1.320844508	0.6083889441	0.4152518332	0.3129445409
Parallel 3	1.309049262	0.7166669091	0.4889247909	0.3562798252

Table II: Matrix parallel multiplication results speed comparison with serial version. Size = 1000 x1000

Execution times (s)

Threads	1	2	3	4
Serial	71.17989200	-	-	-
Parallel 1	75.35346250	48.50690960	38.11987820	36.62936960
Parallel 2	77.89950130	40.18462520	26.33574250	19.90274940
Parallel 3	73.33786970	38.59424040	26.07667320	19.74612250

Table III: Matrix multiplication results with different parallel structures and amount of threads. Size = 2000 x 2000

Speedup (with serial as reference)

Threads	1	2	3	4
Serial	71,179892	-	-	-
Parallel 1	1.058634122	0.6814692779	0.5355427935	0.5146027701
Parallel 2	1.094403196	0.5645502412	0.3699885145	0.279611964
Parallel 3	1.030317238	0.542207066	0.3663488728	0.2774115266

Table IV: Matrix parallel multiplication results speed comparison with serial version. Size = 2000 x2000

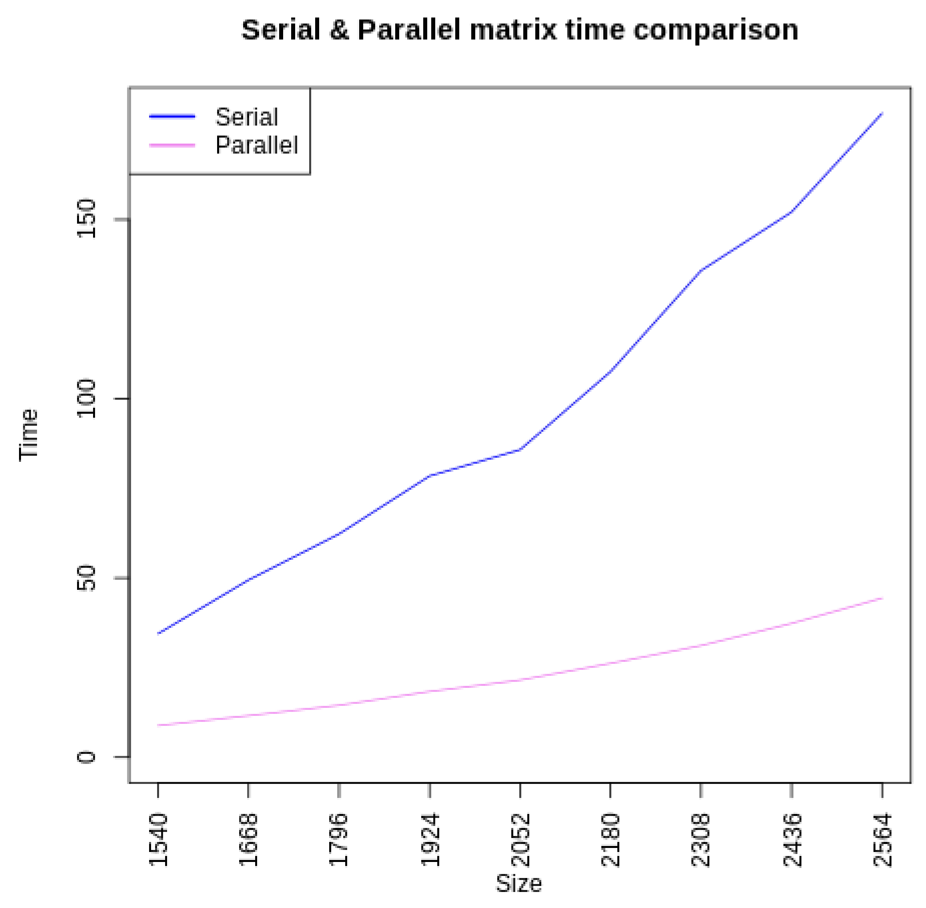
3.1 Loop performance

As the table shows, the best performance is obtained when paralleling the outermost loop while the worst one is obtained by paralleling the innermost one, leaving the middle loop in the middle. This is because the deeper in the loops the threads are thrown, the more threads the program throws. This means that while the outermost loop only throws the threads one time, the innermost one throws them NxN times making the cost of creating those threads greater and therefore most of the time not worth it.

3.2 Grain

As long as cases like this go, coarse-grained parallelization performs best due to the reasons explained in exercise 3.1 because it reduces the total cost of throwing the threads making the program more efficient, nevertheless there might be other cases where fine grained could be preferible by the programmer.

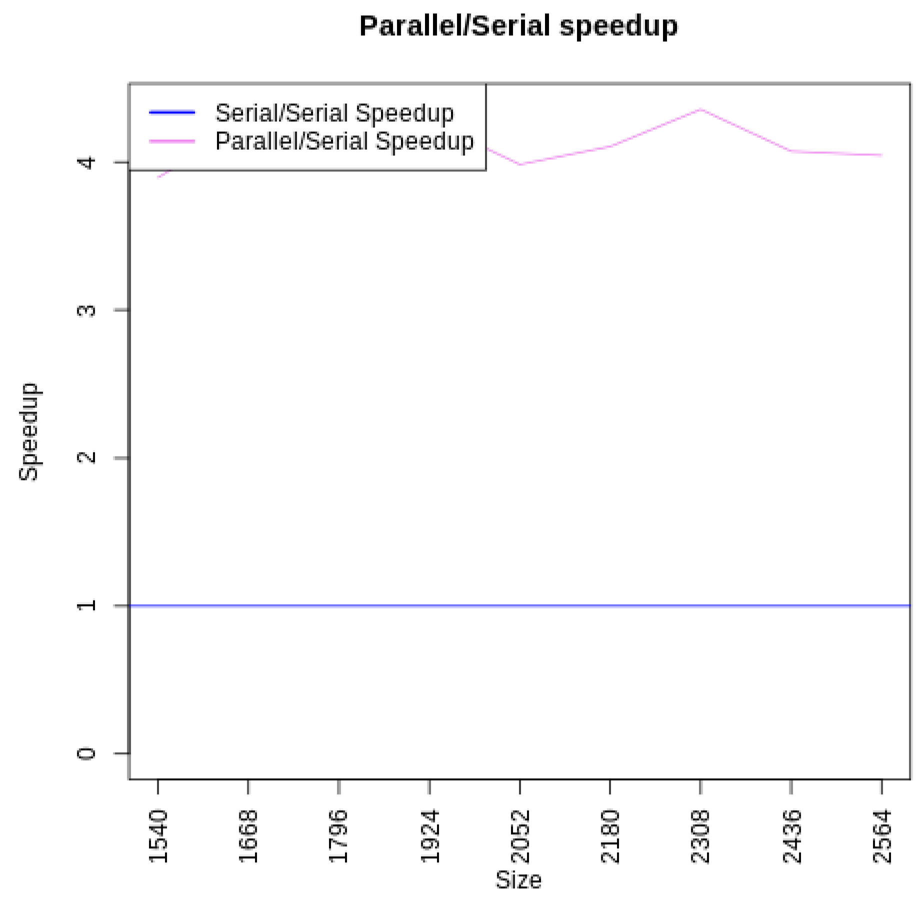
A graph with increasing matrix size is shown where we can observe how both serial and parallel programs relate to time:



Graph I: Serial & Parallel time comparison on different matrix sizes

3.3 Speedup Stabilization

As matrix size increases a change in trend should be shown where the speedup acceleration of the parallel program stabilizes. To verify this behaviour we constructed a new graph with higher matrix size values.



Graph II: Serial & Parallel speedup comparison on different matrix sizes

As the graph shows the value around the speedup stabilizes is around 4.

Experiment 4

The concept of False Sharing

4.1 # of rectangles and size.

The numerical integration is performed with n rectangles where n = 100000000. The length of each rectangle is h = 1/n (where n = 100000000), or 10^{-9}.

4.2 Performance and results.

All the versions of the program show a correct result of the π number. A table with the performance differences between the algorithms is shown below.

Pi algorithm times (s) and speedup (serial as reference)

	Time (s)	Speedup
Serial	1.3734978	1
Parallel 1	0.4648219	2.95489046449834
Parallel 2	0.4343466	3.16221607352285
Parallel 3	0.1873596	7.33081091121031
Parallel 4	0.1872983	7.33321017862949
Parallel 5	0.1872084	7.33673168511669
Parallel 6	0.4424168	3.10453355297538
Parallel 7	0.1874145	7.32866347054257

Table III: Pi algorithm serial & parallel multiplication results.

To process this data first we need the differences between every program:
Parallel 1: uses numThreads as a private variable .
Parallel 2: uses numThreads as a private variable,
uses sum as a first private variable.
Parallel 3: uses numThreads as a private variable,
uses padding.
Parallel 4: uses numThreads as a private variable,

uses a "private_sum" variable defined inside the parallel zone.
Parallel 5: uses x, sum, i as private variables
Uses numThreads as a variable defined inside the parallel zone.
Uses critical clause
Parallel 6: uses numThreads as a private variable,
uses a parallel for loop inside the already parallel zone.
Parallel 7: uses i, x as private variables,
uses parallel reduction for sum.

4.3 Private pointers.

It does not make sense to declare sum as a private variable. The "sum" variable is pointing to the elements of a shared array where each thread fills a different spot with the gathered information. This makes it being shared useless.
When you declare a pointer as private the address the pointer makes reference to doesn't change, only the address where the pointer is stored changes.

4.4 False sharing.

False sharing occurs when different threads use cache storing the same region so when the value of a thread is saved other threads caches are not valid anymore (this would produce a write failure followed by new cache read, increasing execution time). False sharing increases execution time but would never produce a wrong result.
The "pi_par1" program doesn't do anything to solve the false sharing problem so execution time is increased.
The "pi_par3" program gets the cache size to separate the values gathered by the threads on different caches so the false sharing problem won't happen. This decreases execution time, showing much faster results than "pi_par1".
The "pi_par5" makes the sums separately on each different thread and then sums the result using a critical clause. This reduces false sharing because it reduces memory writes to only the amount of threads the program is using. This improves performance significantly, showing a really fast result.

4.5 Critical clause

With a critical clause the work done by each thread to the result variable can be done inside each thread without needing to wait until all threads are done to start adding the numbers. This allows the program to be faster and to obtain a better performance result.

4.6 Work sharing

The program “pi_par6” uses an omp parallel for clause instead of dividing the work depending on the thread number. The performance is similar to the “pi_par1” program because it has the same false sharing problem.

4.7 Optimal version

The fastest program with a correct result is “pi_par7”. This program uses parallel reduction, which results in a very fast execution. Reduction adds the sum of private variables to then adding them up at the end (which reduces false sharing to only the amount of threads the program uses) with a very fast binary tree method.

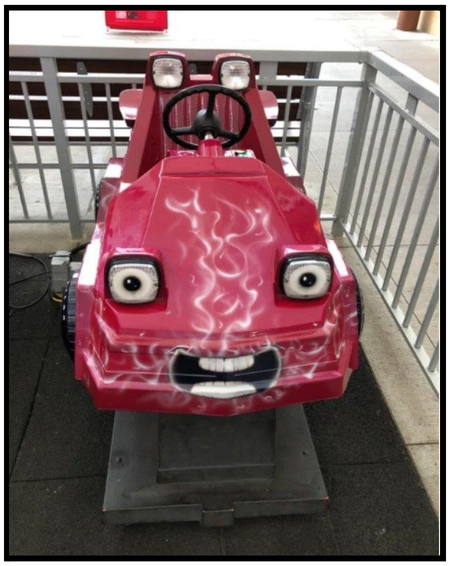
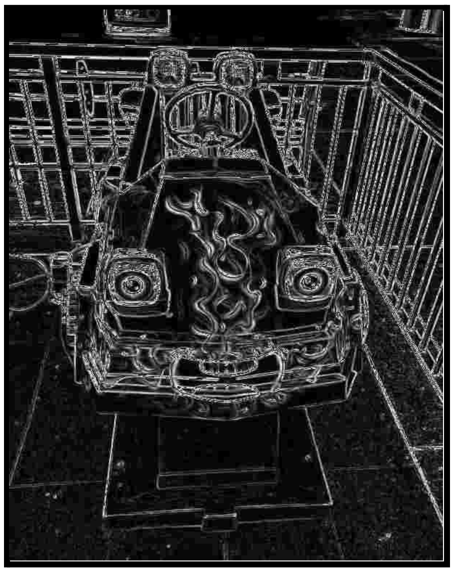
Experiment 5

Image processing

5.0 Program behavior

When the compiled program is executed three new images are created: image_grey, image_grad and image_grad_denoised. These images are versions of the original images with different effects.

Example of program edgeDetector execution with a beautiful picture

	
Original image	**Image grad**

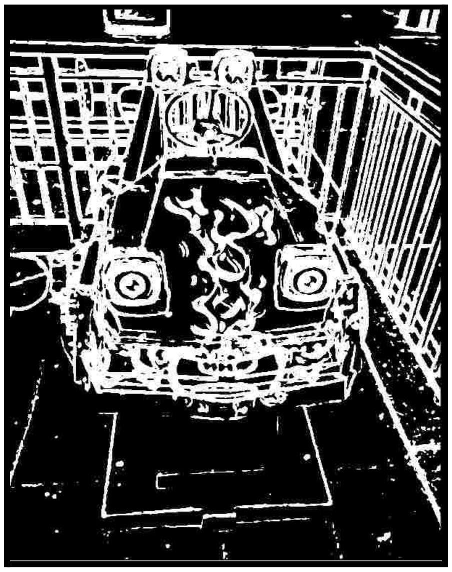
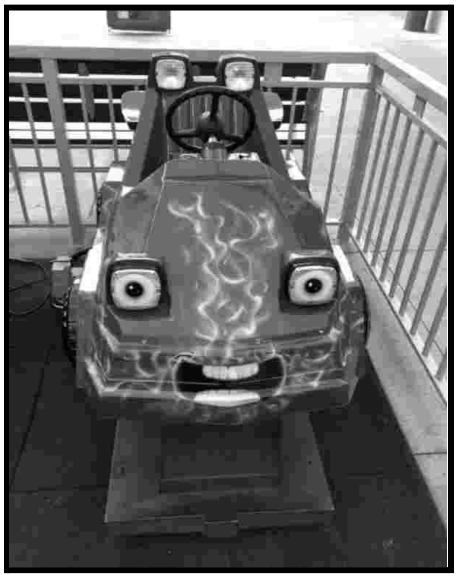

	

Image grad denoised	**Image gray**

The source code shows how it applies the different filters to the images by using an algorithm to detect edges.

5.1 Parallelization on loop 0

A.
This loop would create a new thread for each argument in the function. If the number of arguments are less than the number of arguments it would result in unused cores with an nonoptimal result.
B.
The picture size is 6 GB.
A pixel consist on 3 color channels (RGB) with 8 bits each, meaning 3 Bytes per pixel, resulting in a $6*10^9 / 3 = 2*10^9$ pixels image.
The program creates 3 new images of the same size as the original one. If the original image is 6GB, the new memory to be allocated would be 6*3=18GB of memory (for each thread). This is an amount that is certainly unfeasible.
Although loop 0 is the outertest and therefore taking into account the results obtained in previous experiments should be the best place to parallelize, it does not seem the best candidate due to these drawbacks.

5.2 Data access

Inside loop 0 of the program we can find three loops that pass through all the pixels of each image. These are nested loops (because the image is two dimensional) cycling first through the width and then through the height of the matrix (the image). As we prove in the previous practice this way of looping is not cache efficient and performance can be improved by substituting each one by the other (First looping through the y axis then the x axis).
After executing the improved version of the program we realize this improvement only modifies execution time lightly. The loop of the conversion algorithm to gray does experience a greater variation while the rest practically do not modify the result

5.3 Parallelized version

We realized in the beginning of this experiment that the 0 loop was not viable for parallelization. Following the conclusion obtained in experiment number 3 where the outermost loop (coarse-grained) was the best choice for parallelization in terms of execution time, so the next logical choices were the loops inside loop 0.

To manage the parallelization of these loops we decided to make loops parallelizations, declaring the inner variables as private. This would help increase the performance of the program.

5.4 Time and speedups of different images

To find the improvements in performance produced by the algorithm modifications and the parallelization we collected the execution time for images of different sizes.

IMAGEN	SERIAL	MODIFIED	PARALLEL	FPS
SD.jpg	.0436752	.0416254	.0133142	22.8962889694
HD.jpg	.1714012	.1685806	.0588500	5.8342648709
FHD.jpg	.3814420	.3743800	.0949124	2.6216305493
4K.jpg	1.5959300	1.4665302	.3102550	.6265938982
8K.jpg	6.5860204	6.0795832	1.2417804	.1518367601

The results show how the parallelization significantly improves performance time, while the algorithm improvement modifications for better cache usage are not that relevant.

5.5 Optimizations

We ran the program again using the -O3 gcc optimizations. The results are shown in the following table:

IMAGEN	SERIAL	MODIFIED	PARALLEL	FPS
SD.jpg	.0209576000	.0205136	.0043652	47.7153872580
HD.jpg	.0882872000	.0877604	.0192756	11.3266702307
FHD.jpg	.1982950000	.1971278	.0383568	5.0429915025
4K.jpg	.8245014000	.7880250	.1299180	1.2128542171
8K.jpg	3.3794654000	3.0902006	.4950542	.2959047901

The -O3 flag improves significantly performance on every program we execute. This happens due to the optimizations the compiler performs when this flag is active. This optimizations are dangerous because they can interfere with the parallelization we introduced on this programs, as it skips some steps to achieve a faster result.

Conclusions

The parallel processing and its consequences

The conclusion of the experiments done in this practice would be that parallel programming can be very useful and can reduce the time that a program takes to be executed. To get to this conclusion, tests and graphics have been carried out to check the impact of the different types of parallelization provided by OpenMP: the difference in performance in parallelizing nested loops, the difference between the use of different amounts of threads, the efficiency depending on the size to be parallelized, the concepts of false sharing and possible parallelization problems, and the possible optimizations of the code have been verified.

We would like to thank Daniel Perdices, our Computer Architecture professor for helping us complete this practice and teaching us a lot about this subject.