University of Lleida

Master's Degree in Informatics Engineering

Higher Polythecnic School

OpenMP Implementation

High Performance Computing

Francesc Contreras

Albert Pérez

May 17, 2021

Table of contents

1 Introduction **1**

2 Purpose & Objectives **1**

3 Environment **1**

4 Description of the problem **2**

5 Resources **4**

6 Statement **4**

7 Program study **5**

8 Parallelization **7**

8.1 Overview parallelization 7
8.2 Convolve 2D 8
8.3 Scheduling 9

9 Results **12**

9.1 Execution time results 12
9.2 Speedup and Efficiency 13
9.3 Scability of the program 14

10 Conclusions **15**

11 Code repository **16**

List of Tables

1 Scheduling comparison 11
2 Serial time results 12
3 Parallel time results 12

List of Figures

1 Convolution process 2
2 Emboss effect 2
3 Convolution kernel 2
4 Convolution process 3
5 Convolution operations with different kernel matrix 3
6 Serial program flow 6
7 Example of time spent by each task 7
8 Scheduling comparison 10

1 Introduction

This document is the report consisting of a High Performance Computing project part 1, which is based on solving a specific problem by means of *OpenMp* (Open Multi-Processing) programming model.

Regarding the structure of the document, first the purpose and objectives are detailed and then the environment for the execution of the solutions.

In addition, further on there is a description of the problem to be parallelized as well as the resources provided to carry out the parallelization.

Then, the requirements and the problem statement are described, which specify the tasks to be performed.

And finally, are specified the sections related to the design decisions of the parallel solution, the analysis and conclusions of the results obtained with the execution of different serial and parallel programming approaches, as well as different configurations and problem sizes when executing them.

2 Purpose & Objectives

Specifically, the main aim is to parallelize the given problem and analyse the scalability of the proposed solutions and the effects of design decisions.

Moreover, draw conclusions over the behaviour analysis done to get deeper in learning the parallelization concepts to be applied in high computing environments.

3 Environment

The cluster used to run the tests was provided by *Universidad de Lleida* (UDL), and is remotely accessible, for instance, with the execution of an *ssh* command at the server *moore.udl.cat.*

The so-called *Moore cluster* [3] is a homogeneous cluster with a total of 32 processing units and 32GB of main memory, distributed over 8 nodes with an Intel Core i5 processors with 4 cores each. Each node works at 3.1Ghz rate and has 4GB of main memory. In order to avoid memory issues and never-ending jobs, all jobs are launched through an *Sun Grid Engine* (SGE) queue system and are limited to 1GB of main memory and a maximum of 3 hours of execution time.

4 Description of the problem

Convolution is a mathematical operation on two functions f and g used to produce a third function that is interpreted as a modified version of one of the original functions. Basically, it gives the area overlapped between both functions as a function of the amount that one of the original functions is translated, as it is shown in Figure 1.

In order to clarify the objectives of the convolution process, it will be presented applied in the field of image processing. In this field, the convolution process is widely used to change the intensity of each individual pixel to reflect the intensities of the surrounding area.

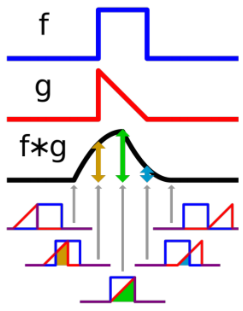


Figure 1: Convolution process

A common use of convolution is to create image filters as could be blur, sharpen, and edge detection. Figure 2 shows the results after applying the emboss filter.

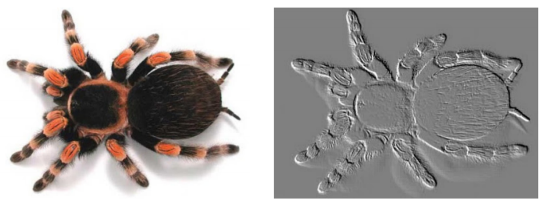
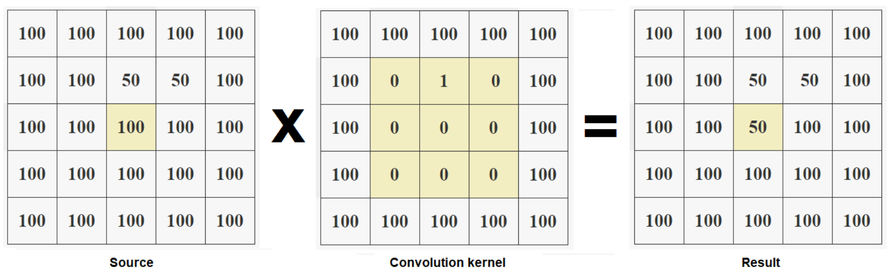


Figure 2: Emboss effect

The convolution process in the field of image processing is based on applying a matrix named as “convolution kernel”. This matrix consists of an n×n matrix identifying the surrounding area of the evaluated pixel. Each cell of the kernel contains a pixel weight in respect of the pixel.



100	100	100	100	100
100	100	50	50	100
100	100	100	100	100
100	100	100	100	100
100	100	100	100	100

Source

X

100	100	100	100	100
100	0	1	0	100
100	0	0	0	100
100	0	0	0	100
100	100	100	100	100

Convolution kernel

=

100	100	100	100	100
100	100	50	50	100
100	100	50	100	100
100	100	100	100	100
100	100	100	100	100

Result

Figure 3: Convolution kernel

In the example shown in Figure 3, the convolution process corresponds to the individual multiplication of every paired cells/pixels, from the source and the kernel.

Finally, it obtains the sum of all of them, which corresponds to the resulting pixel value, as it is presented in the Figure 4.

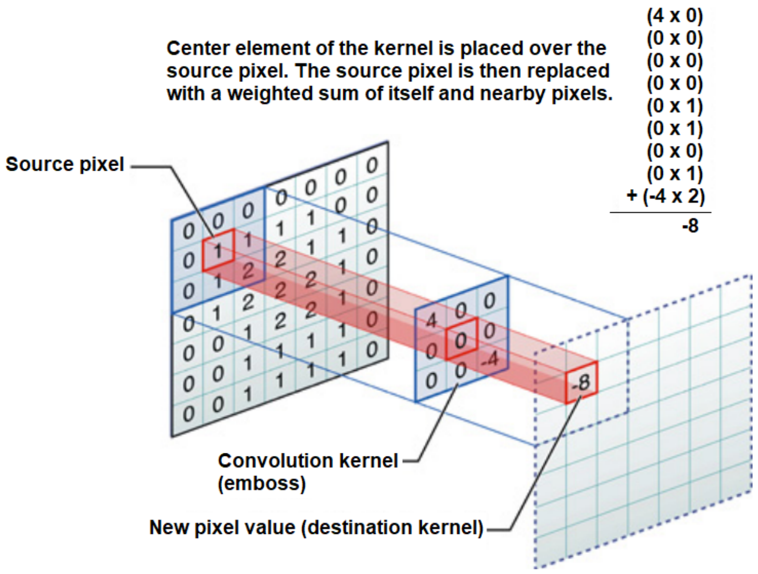

Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination kernel)

Figure 4: Convolution process

Varying the composition of the kernel, it is possible to obtain different results. Figure 7 shows different image convolution results.

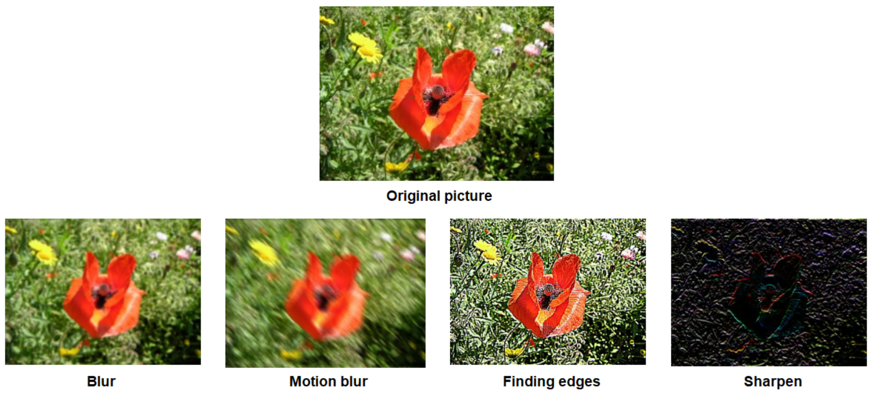


Figure 5: Convolution operations with different kernel matrix

The convolution process complexity depends on the original matrix size, the kernel dimension and also on the mathematical function applied to the cells. By this reason, its parallelization is an important technique to take into account in further implementations.

5 Resources

To develop this project, you can create your own test-bed. However, in order to compare the performance and the results obtained from different groups solutions, we will provide the test-bed that you have to pass for providing your results and elaborate the reports. The material you have for this project is as follows:

- **Serial code**: Source code for a serial implementation. This serial implementation allows you to process small and large images. For large images (¿500MB) you will have problems allocating enough memory for processing convolution. If you want to test for images bigger than the images provided for us, then you would need to split the original images into partitions.
- **PPM images**: In folder "/share/apps/files/convolution/images" you can find the set of images to test. Notice that some of these images are so large.
- Kernel matrix: In folder "/share/apps/files/convolution/kernel" you can find the set of kernel matrices. Kernel matrices are described using text files. Some of the kernel provided (3x3 Edge, 5x5 Sharpen) are very useful in image processing area; the big ones (25x25, 49x49, 99x99) are randomly generated for testing the scalability of your proposals.
- The serial code generates a text file with the time spent in different execution phases and a PPM file with the resulting image.

6 Statement

In order to obtain the best performance of an application implemented with a parallel programming model, firstly, we should start by optimizing the application at node-level. According to this, the first section will focus on studying the operation of the serial version, analysing the pieces of code liable to be parallelized following a work and data decomposition model. Next, we will apply the OpenMP directives to parallelize the corresponding code according to the hardware and the suitable work decomposition model at node level.

In this section you should elaborate a report discussing about the following points:

- Justify the strategy used to parallelize the serial code.
- Check the performance obtained using different thread scheduling policies (static, dynamic, guided) and chunk sizes and discuss about the results. It is not needed to test for all the test-bed of images.
- Analyze scalability and speedup in relation to the number of threads and size problem. Consider on your discussion the effects of the dependences between iterations, in case it was needed, in your solution.

7 Program study

The first step must be followed in developing a parallel software is to understand the problem to solve in parallel. As in this case, we got a serial program, this necessitates understanding the existing code before spending time in attempt to develop a parallel solution, because should be determined whether or not the problem is one that can actually be parallelized.

As mentioned in previous sections, the main purpose of the program is based on image convolution. But initially, we will focus on analysing the whole program structure. And then, obviously we will get deeper into analysing how the convolution procedure works.

So, mainly the parallel analysis, either of the entire program or only the convolution compute work, will be focused on identifying the following points:

- **Hotspots**: Know where most of the real work is being done. Focus on parallelizing the hotspots and ignore those sections of the program that account for little CPU usage.
- **Bottlenecks**: Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred? For exemple, I/O is usually a sign of program slow down.
- **Inhibitors to parallelism**: One common class of inhibitor is *data dependence* as program sections might be waiting for some values.

Furthermore, there are some factors that must also be taken into account such as the communication between tasks as it will depend directly on the program or problem in question; and also how to break the problem into discrete "chunks" of work to be able to distribute them, knowing this concept as decomposition or partitioning which might follow two ways:

- **Data decomposition**: each parallel task will work on a data portion.
- **Domain decomposition**: split the problem according to the work that must be done rather than on the data manipulated by the computation.

First of all, when analysing the serial program itself, can be seen that it has 5 high-level tasks which are the following: read image, duplicate the image data, read kernel, compute the data (convolution) and finally save the data computed as a file.

All these tasks are used in the main program to apply the convolution to the image. So, basically are considered three main groups of task, which are the reading ones, followed by the convolution computation and finally by the output writing.

In the Figure 7 you may consult the program high-level task diagram where all previous points are detailed. Moreover, must be said that the duplicate image data task is specified as a different group as might be parallelized over the read kernel because they don't have any kind of data dependence.

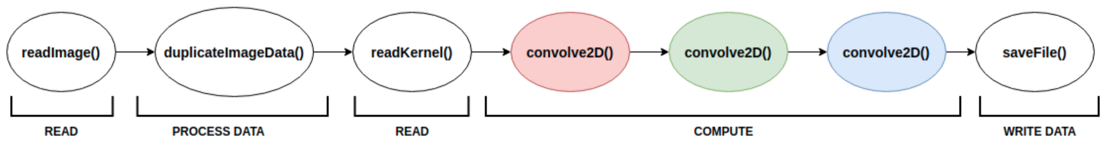


Figure 6: Serial program flow

In addition, exist data dependence from the read tasks over the compute as this last one needs the image source and output, and the kernel; to compute the result. So, this means that cannot be parallelized the reading over the compute, but as well as the compute over the writing where it exist another data dependence regards the output image.

Then, after having analysed the source code, we can see how the computation task, which splits into three *convolve2D* parts, is the unique hot-spot in this code, because observing the duration times it takes so much longer over the rest tasks as may be predict is the main purpose/calculation of the program. Therefore, in order to perform this task, we will have to wait for source, duplicated image and kernel to be generated. This data will be generated by the data processing and reading tasks, as commented before as a data dependence.

On the other hand, we can see how the tasks of reading, data processing and writing are bottlenecked, as it is very important to preserve the order in which the data is received and the only way to do this is sequentially. Also parallelism these parts, regarding its inner code, is absurd as the files used are not very large and doesn't take so much time which sincerely is ridiculous over the computation one.

Up to this point we have made a small study and introduction to how parallelize the program and the problem in question. In the next section the emphasis on parallelization is made explicit.

8 Parallelization

8.1 Overview parallelization

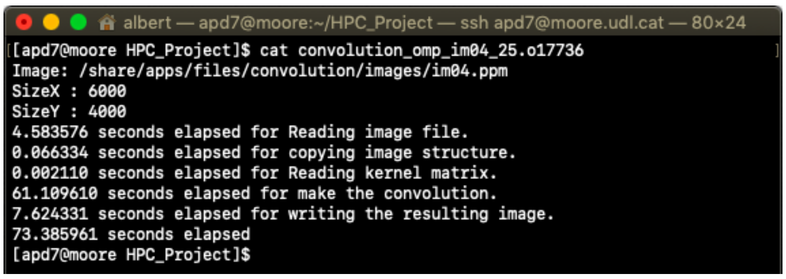
To parallelise this sequential code we will divide the tasks according to their function: reading, processing, computing and saving data.

On one side we have the read tasks *readImage(...)* and *readKernel(...)*, data writing *saveFile(...)* and data processing *duplicateImageData(...)*, which we will treat in the same group because they don't affect directly the amount of time spent by the program.

Besides that, we will have the three tasks *convolve2D(...)* which will be performed by the three primary colours (Red, Green and Blue) where the convultion must be applied in order to obtain the output image.

The tasks in the first group will not be stopped, as the files used are not very large and therefore improving this part of the code will be negligible, because analysing its inner code as when reading is needed file descriptors even though getting those functions into a parallelizadtion mode will be necessarily a sequential reading over all. But, as mentioned before the amount of time spent with such these input size files is not enough to consider its parallelization a good point.

Moreover, also depends on the order in which the data is read and saved to be able to perform the calculations, so will be very important to maintain this order and to do so, we will have to do it in a sequential way. However, if in the future the files to be processed were very large, you could always divide the file into fragments, where each fragment could execute as a task and finally join them together, but currently this is not the case based on the test results obtained.


albert — apd7@moore:~/HPC_Project — ssh apd7@moore.udl.cat — 80×24

```
[apd7@moore HPC_Project]$ cat convolution_omp_im04_25.o17736
Image: /share/apps/files/convolution/images/im04.ppm
SizeX : 6000
SizeY : 4000
4.583576 seconds elapsed for Reading image file.
0.066334 seconds elapsed for copying image structure.
0.002110 seconds elapsed for Reading kernel matrix.
61.109610 seconds elapsed for make the convolution.
7.624331 seconds elapsed for writing the resulting image.
73.385961 seconds elapsed
[apd7@moore HPC_Project]$
```

Figure 7: Example of time spent by each task

On the other hand, we can parallelize how the *convolve2D(...)* tasks are executed for each colour, since there are no dependencies between these tasks, so we can execute these three in parallel, improving the execution time of the program/problem main part.

So if we can run the *convolve2D(...)* tasks in parallel, we can say that these functions are part of a section.

```
                                    ....
#pragma omp parallel sections num_threads(MAX_THREADS) default(none)
    shared(source,output,kern)
{

   #pragma omp section
   convolve2D(source->R, output->R, source->width, source->height, kern->vkern,
       kern->kernelX, kern->kernelY);

   #pragma omp section
   convolve2D(source->G, output->G, source->width, source->height, kern->vkern,
       kern->kernelX, kern->kernelY);

   #pragma omp section
   convolve2D(source->B, output->B, source->width, source->height, kern->vkern,
       kern->kernelX, kern->kernelY);
}
                                    ....
```

8.2 Convolve 2D

Having parallelised the tasks as a set, we set out to parallelise the *convolve2D(...)* function. In order to do so, we analysed the entire function, and realised:

- We can use the *parallel* directive that defines a parallel region, which is code that will be executed by multiple threads in parallel, also the *for* directive that causes the work done in a for loop inside a parallel region to be divided among threads.
- For general attributes we can use the clause *schedule* that applies to the *for* directive and *num_threads* to sets the number of threads in a thread team.
- For data-sharing we can use the clause *private* to specifies that each thread should have its own instance of a variable and *firstprivate* that specifies that each thread should have its own instance of a variable, and that the variable should be initialized with the value of the variable, because it exists before the parallel construct.

```
                                    ....
   // start convolution
   #pragma omp parallel for schedule(dynamic) num_threads(MAX_THREADS) private(sum, i,
       rowMax, rowMin, j, m, n, colMax, colMin) firstprivate(kPtr, inPtr, inPtr2,
       outPtr)
       for(i= 0; i < dataSizeY; ++i)              // number of rows
       {
           // compute the range of convolution, the current row of kernel should be
               between these
           rowMax = i + kCenterY;
           rowMin = i - dataSizeY + kCenterY;
                                    ....
```

8.3 Scheduling

First of all, the configuration for this analysis has been *4 threads*, the image *im03.ppm* and the *kernel99x99_random.txt* in order to get a good example to test having a mid size image with a bit much longer kernel.

The main purpose of this analisys, is to get deeper in understanding how the *convolve2D(...)* inner loop better adapts its execution depending on what shceduling policy is applied, and thus get the best one in this case.

8.3.1 STATIC

The **schedule(static, chunk-size)** clause of the loop construct specifies that the for loop has the static scheduling type. OpenMP divides the iterations into chunks of size chunk-size and it distributes the chunks to threads in a circular order.

When no chunk-size is specified, OpenMP divides iterations into chunks that are approximately equal in size and it distributes at most one chunk to each thread.

The static scheduling type is appropriate when all iterations have the same computational cost.

8.3.2 DYNAMIC

The **schedule(dynamic, chunk-size)** clause of the loop construct specifies that the for loop has the dynamic scheduling type. OpenMP divides the iterations into chunks of size chunk-size. Each thread executes a chunk of iterations and then requests another chunk until there are no more chunks available.

There is no particular order in which the chunks are distributed to the threads. The order changes each time when we execute the for loop.

If we do not specify chunk-size, it defaults to one.

The dynamic scheduling type is appropriate when the iterations require different computational costs. This means that the iterations are poorly balanced between each other.

8.3.3 GUIDED

The guided scheduling type is similar to the dynamic scheduling type. OpenMP again divides the iterations into chunks. Each thread executes a chunk of iterations and then requests another chunk until there are no more chunks available.

The difference with the dynamic scheduling type is in the size of chunks. The size of a chunk is proportional to the number of unassigned iterations divided by the number of the threads. Therefore the size of the chunks decreases.

If we do not specify chunk-size, it defaults to one.

The guided scheduling type is appropriate when the iterations are poorly balanced between each other and it is especially appropriate when poor load balancing occurs toward the end of the computation.

8.3.4 AUTO

The auto scheduling type delegates the decision of the scheduling to the compiler and/or runtime system.

8.3.5 RUNTIME

The runtime scheduling type defers the decision about the scheduling until the runtime. We already described different ways of specifying the scheduling type in this case.

8.3.6 Results & Conclusion

The results shown below represent the different computation times specifically in the *convolve2D(...)* task. As we can see in the chart the best results are given when we use the schedule as dynamic, this is because the iterations are poorly balanced between each other.

As a general conclusion regarding the chunks, we can observe that there is a moment when the more chunks the computation time decreases, but at the same time if we increase this effect is negative as the load balancing is not correct and causes an imbalance between the tasks.

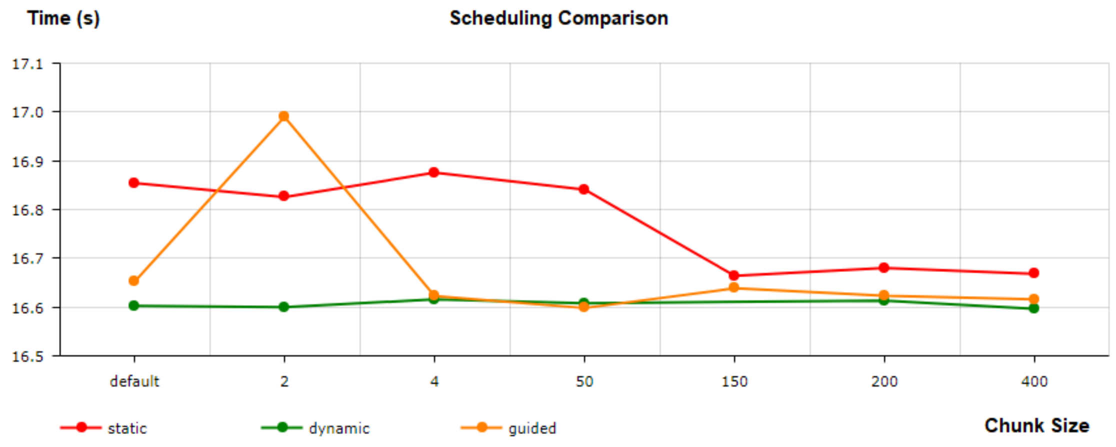


Figure 8: Scheduling comparison

Schedule Clause	**Chunk Size**	**Convolution Time (s)**
static	default	16.8544
	2	16.8255
	4	16.8754
	50	16.8509
	150	16.6641
	200	16.6801
	400	16.6679
dynamic	default	16.6025
	2	16.5997
	4	16.6156
	50	16.6077
	150	16.6025
	200	16.6134
	400	16.5961
guided	default	16.6518
	2	16.9900
	4	16.6228
	50	16.5981
	150	16.6384
	200	16.6230
	400	16.6157
runtime		16.6279
auto		16.8164

Table 1: Scheduling comparison

9 Results

9.1 Execution time results

	Img Resolution	Kernel Size	Time (s)
im01	292x320	5x5	0.1351
		25x25	0.6881
		49x49	2.3516
im03	800x600	5x5	0.4494
		25x25	3.3352
		49x49	12.2802
im04	6000x4000	5x5	18.3815
		25x25	163.0034
		49x49	636.6958

Table 2: Serial time results

		Runtime (seconds)		
	Kernel	**2 Threads**	**4 Threads**	**8 Threads**
im01	5x5	0.1386	0.1830	0.1381
	25x25	0.5873	0.4316	0.3682
	49x49	1.7539	1.0389	1.080546
im03	5x5	0.4116	0.3847	0.3763
	25x25	2.8248	1.5166	1.5584
	49x49	8.9337	4.6070	4.7323
im04	5x5	16.7009	13.9729	14.0195
	25x25	132.4713	73.4487	72.9765
	49x49	445.63	232.2603	232.3589

Table 3: Parallel time results

9.2 Speedup and Efficiency

	Kernel	Threads	Speedup	Efficiency	Efficiency (%)
im01	5x5	2	0.9747	0.4873	48.73
		4	0.7382	0.1845	18.45
		8	0.9782	0.1222	12.22
	25x25	2	1.1716	0.5858	58.58
		4	1.5943	0.3985	39.85
		8	1.8688	0.2336	23.36
	49x49	2	1.3407	0.6703	67.03
		4	2.2635	0.5658	56.58
		8	2.1763	0.2720	27.20
im03	5x5	2	1.0918	0.5459	54.59
		4	1.1681	0.2920	29.20
		8	1.1942	0.1492	14.92
	25x25	2	1.1806	0.5903	59.03
		4	2.1991	0.5497	54.97
		8	2.1401	0.2675	26.75
	49x49	2	1.3745	0.6872	68.72
		4	2.6655	0.6663	66.63
		8	2.5949	0.3243	32.43
im04	5x5	2	1.1006	0.5503	55.03
		4	1.3155	0.3288	32.88
		8	1.3111	0.1638	16.38
	25x25	2	1.2304	0.6152	61.52
		4	2.2192	0.5548	55.48
		8	2.2336	0.2792	27.92
	49x49	2	1.4237	0.71185	71.18
		4	2.7413	0.6853	68.53
		8	2.7401	0.3425	34.25

9.3 Scability of the program

The scalability of the application can be measured by the execution time compared regarding the image input size and the effectively utilize an increasing number of processors.

So, to test the scalability we shall look at the efficiency values obtained in the results. As we look at the results we can see that the efficiency is higher when we use a larger image and kernel.

Therefore we can say that this problem, using parallelization, does not have to pose any scalability problem, so it is scalable.

Mainly, because it will be more efficient and will turn more embarrassingly parallel when larger images and kernel are used as may be seen in the results obtained like a progression directly proportional in mathematics terms.

10 Conclusions

Based on the knowledge acquired at the time in this course, and of course by the results obtained after executing the parallel application several times, we conclude that these results are satisfactory when executed in parallel despite of the efficiency where sometimes can be rough.

But there for sure, the 4 threads is the best option when talking into higher image sizes as it responds good regarding the time spent and how the program efficiently is parallelized.

Furthermore, when the image size is lower the higher num of threads are not well received as we got lower efficiency on the execution test. But, besides all these points, the program is well-parallelized as we got better times and depending the case an efficiency over the 70%.

Finally, we know the may be possible to get into the parallelization of the convolution procedure by divinding the data or the domain required, but since OpenMp does not provide communication, we should attemt to get the expected results in the next deliveries, using MPI (Message Passing Interface).

To conclude, was a pleasure to learn more about how to well parallelize a program as well as the OpenMP programming interface which gets so useful nowadays and more when new technologies are raised.

References

[1] Distributed Computing Group — Polytechnic School — University of Lleida

[2] Introduction to OpenMP — Francesc Giné, Josep L. Lérida (Group of Distributed Computing), University of Lleida

[3] Moore Cluster Home Webpage

[4] HPC Project: Convolution Operation with OpenMP

[5] OpenMP Microsoft Summary

[6] OpenMP Scheduling comparison

11 Code repository

HPC Project: Convolution Github Repository