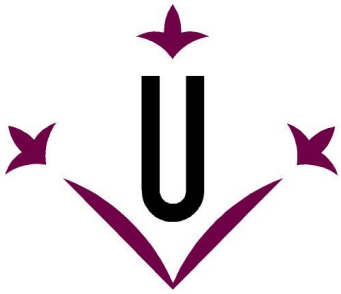

University of Lleida

Master's Degree in Informatics Engineering

Higher Polythecnic School

MPI Implementation

High Performance Computing

Francesc Contreras

Albert Pérez

June 6, 2021

Table of contents

1 Introduction **1**

2 Purpose & Objectives **1**

3 Environment **1**

4 MPI (Message Passing Interface) **2**

5 Strategy **3**

6 MPI Solution **5**

7 Results **8**
7.1 Execution time results 8
7.2 Scability of the program 9

8 Conclusions **10**

9 Code repository **11**

List of Tables

1 Serial time results 8
2 Parallel time results 8
3 Speedup and Efficiency 9

List of Figures

1 Distributed memory 2
2 Hybrid distributed memory 2
3 Communication 2
4 Communicator 2
5 Problem partitioning 3
6 MPI_Scatterv 6
7 MPI_Gatherv 7

1 Introduction

This document is the report consisting of a High Performance Computing project part 2, which is based on solving a specific problem through the *MPI* (Message Passing Interface) 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.

Finally, are specified the sections related to the design and strategic decisions taken for the parallel solution, the analysis, and the conclusions based on 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 analyze the scalability of the proposed solution and the effects of design decisions as there are basically two ways on focus the paralleization, mapping statically the task or using a dynamic master-worker paradigm, which are detailed in the following sections.

Moreover, draw conclusions over the behavior 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* via the student account.

The so-called *Moore cluster* 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 MPI (Message Passing Interface)

MPI was born to provide a widely used standard for writing message-passing programs, designed specifically for distributed memory architectures which were becoming increasingly popular around the 1980s - early 1990s.

As architecture trends changed, shared memory the SMPs[1] architecture, thanks to technological progress, were combined over networks creating hybrid distributed memory / shared memory systems.

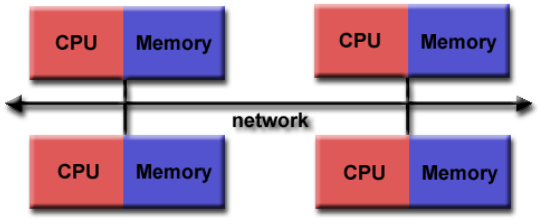

Figure 1: Distributed memory

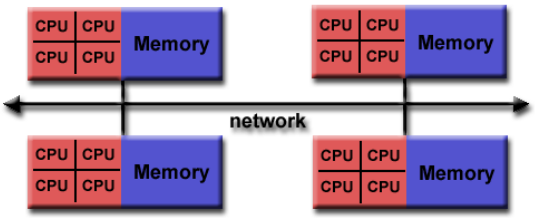

Figure 2: Hybrid distributed memory

As consequence, MPI typically implement SPMD (Single Program Multiple Data) model, meaning all processors execute the same program but with different data, however, also allows MPMD (Multiple Program Multiple Data).

Written in a conventional sequential language, e.g., C or Fortran gets its identity on providing a message interface over processes where the term of communicator rises. A communicator defines a group of processes that have the ability to communicate with one another where each one is identified by its assigned rank.

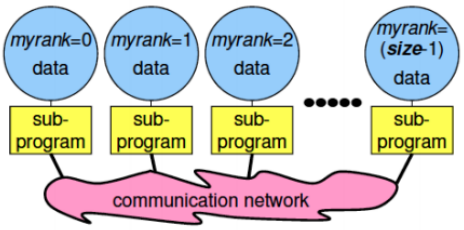

Figure 3: Communication

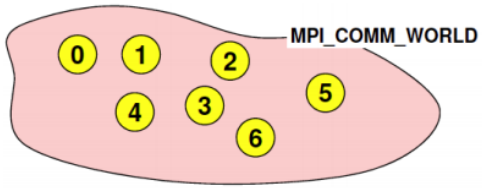

Figure 4: Communicator

The communication is the base of MPI as a programming model, built upon send and receive operation among processes. Therefore, a process may send a message to another process by providing the rank and a unique tag to identify the message. This kind of communication is known

[1] Shared Memory Parallelism

as point-to-point because involve a sender and a receiver, but there are many parallel scenarios where processes may need to communicate with everyone else. That's why MPI can handle a wide variety of collective communications types involving all processes or the ones fitting into the communicator specified, but also are included synchronous/asynchronous communications.

To end this brief introduction since the aim was to establish a starting point, MPI is characterized by getting efficient implementations and being source-code portable.

5 Strategy

There are mainly two strategies regarding parallelization problems, well-called mapping strategies as referring to how the tasks of the problem decomposition are assigned into different processors/cores. And the key is analyzing accurately the scenario, meaning the problem and the kind of environment where to apply, since the aim is getting the best possible performance.

a. **Static mapping of tasks:** the region is divided into a fixed number of fragments. The allocation of the work is done at the beginning of the execution of the program and cannot be modified.

b. **Dynamic mapping of tasks** the fragments will be mapped to different processors / cores as soon as they finish with the previous calculations.

The point is to define a strategy, first, based on the problem decomposition, and secondly, taking into account the environment, in this case, the *Moore cluster*.

In this way, the convolution problem does an image treatment by using a kernel. But the fundamental is the image data, which in the program is represented as a *type-def structure* containing the principal RGB[2] channels, those representing the image, set as memory allocated *integer* arrays. According to this, as the problem consists on apply the kernel convolution over this values, the problem may be divided regarding the image into quadrants, rows, columns, etc.

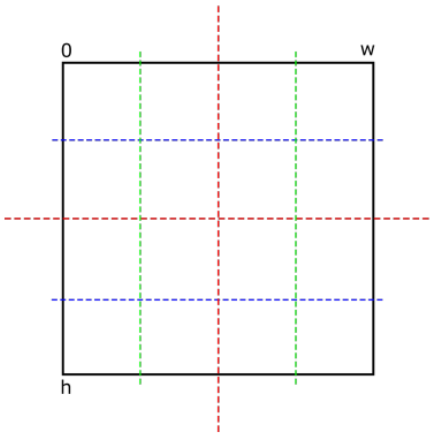

Figure 5: Problem partitioning

[2]Red, Green and Blue: color composition

Thus, this division becomes the tasks that should be done and distribute over the processes. So, the idea is to divide the image, each channel array, to get a set of tasks (portion of the image domain) to be computed in parallel by the processes which will perform the convolution.

Hence, was chosen to apply a horizontal division where the channel rows will be divided into sets. Furthermore, this kind of problem is composed of homogeneous tasks because each one will perform the same convolution algorithm, implying a no communication needed scenario as the hand of the computation doesn't require any additional data computed at the same time by the other processes. Just its image portion and the kernel.

At this point when we know or at least have an idea of the problem nature and how to parallelize it over several processes, also must pay attention to the environment available where the execution will be performed. Because, depending on the underlying physical architecture of the machine and the kind of tasks to perform, the programmer should build one or another MPI parallel algorithm implementation.

Then, regarding the *Moore cluster*, it has the same hardware for each node resulting in a homogeneous cluster (all features detailed in the environment section on page 1). By definition, all nodes have the same computation resources/power. This leads to, considering the alike tasks because of the same computation, a homogeneous-homogeneous pair scenario.

Therefore, we consider a best practice to implement the static strategy instead of the master-slave paradigm, which fits better when the tasks work amount varies, appearing the processes will spend different times, or when the nodes are heterogeneous being ones faster than others. In both last cases, a dynamic mapping strategy is better because the amount of work may be distributed equitation to get the outshine performance.

To summarize, we shall apply a static mapping strategy in order to assign the tasks to the processes because their amount of work is equal and the cluster is homogeneous meaning that by dividing the image data domain by channel rows getting several row sets and distributing them equally. Because doing the contrary, applying a dynamic strategy, first, it makes no sense in terms of amount of computation distribution, but also the program might suffer communication overhead as the master will need to attend the workers requests.

6 MPI Solution

As previously said, the problem decomposition consists of splitting the image into a fixed number of chunks to become processed in parallel by different processors/cores. Accordingly, we break down a heavy calculation into smaller independent parts.

First of all, we concluded every core/node will take all the input data, the kernel and the image. We considered that as the input data is relatively small in our case, nonetheless, if it grows in the future assigning the task of reading to one core or a couple of them combining the data afterward might be a good practice.

Then, to divide the image we must focus on its structure which contains three attributes representing each of the RGB channels as memory allocated integer arrays corresponding to the pixels' colors. Likewise, there are extra attributes such as width and height that store relative values from which we may determine the division over the image core data.

With that in mind, the RGB channels should be distributed betwixt the cores, and attributes such as width and height will provide the math calculations solving for the scattering realization.

```
                                    ....
// MPI
// Load Balancing
int sizePerCore = (source->width*source->height)/size;
int *sendcounts = malloc(sizeof(int)*size); // memory allocation
int *displacement = malloc(sizeof(int)*size);
int height = sizePerCore/source->width;

for (int i = 0; i < size-1; i++)
{
    sendcounts[i] = sizePerCore;
    displacement[i] = i * sizePerCore;
}

sendcounts[size-1] = (source->width*source->height) % size == 0 ? sizePerCore :
     sizePerCore + 1;
displacement[size-1] = (size-1) * sizePerCore;

                                    ....
```

On one side, we must know how much data image corresponds to each core, so basically, from the total image pixels along with the involved cores (size) results the division. In turn, at this first step should be set how many data will receive each core into the *sendcounts* allocated array, but also since the aim is perform a distribution shall be assigned the displacement values.

Furthermore, we need to manage the odd cases where the image can't be split equally into different fragments. Therefore, we take into account if the image odd length or not. Nevertheless, the possible task overload produces no impact on the efficiency, because the difference is insignificant.

Subsequently, we planned to use the *MPI_Scatterv* procedure to send RGB channels arrays chunks to different processes, but being conscious of the image odd size cases when may be necessary different distribution sizes (*sendcounts*).

Essentialy, it takes an array of elements and distributes them among the process rank copying the appropriate chunk into the process receiving buffer.

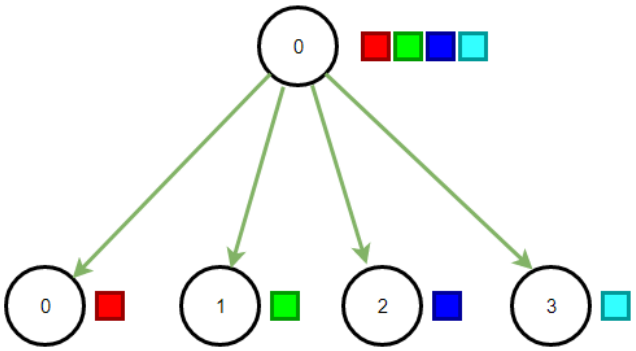

Figure 6: MPI_Scatterv

```
                                ....
// Scatterv
int *receiveR = malloc(sizeof(int)*(sizePerCore*2)); // received task
int *receiveB = malloc(sizeof(int)*(sizePerCore*2));
int *receiveG = malloc(sizeof(int)*(sizePerCore*2));

int *outputR = malloc(sizeof(int)*(sizePerCore*20)); //intermediate convolve output
int *outputG = malloc(sizeof(int)*(sizePerCore*20));
int *outputB = malloc(sizeof(int)*(sizePerCore*20));

MPI_Scatterv(source->R,sendcounts,displacement,MPI_INT,receiveR,sendcounts[rank],
MPI_INT,0,MPI_COMM_WORLD);

MPI_Scatterv(source->G,sendcounts,displacement,MPI_INT,receiveG,sendcounts[rank],
MPI_INT,0,MPI_COMM_WORLD);

MPI_Scatterv(source->B,sendcounts,displacement,MPI_INT,receiveB,sendcounts[rank],
MPI_INT,0,MPI_COMM_WORLD);

// Convolve for each channel
convolve2D(receiveR, outputR, source->width, height, kern->vkern, kern->kernelX,
    kern->kernelY);
convolve2D(receiveG, outputG, source->width, height, kern->vkern, kern->kernelX,
    kern->kernelY);
convolve2D(receiveB, outputB, source->width, height, kern->vkern, kern->kernelX,
    kern->kernelY);
                                ....
```

After sent the chunks to the different processes and calculated the convolution2D, we use the **MPI_Gatherv** to take elements from many processes and gathers them to one single/root process. Also, the the elements are ordered by the rank of the process from which they were received.

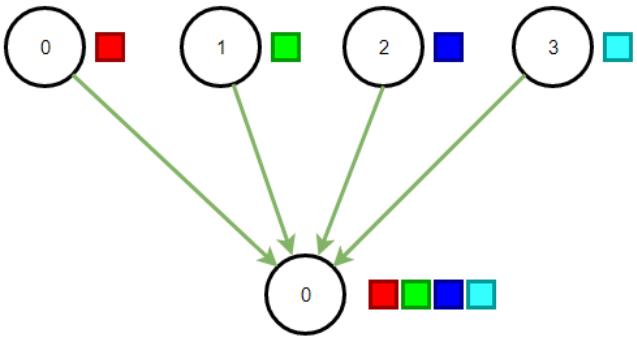

Figure 7: MPI_Gatherv

```
                                ....
// Gatherv
MPI_Gatherv(outputR, sendcounts[rank], MPI_INT, output->R, sendcounts, displacement,
    MPI_INT, 0, MPI_COMM_WORLD);
MPI_Gatherv(outputG, sendcounts[rank], MPI_INT, output->G, sendcounts, displacement,
    MPI_INT, 0, MPI_COMM_WORLD);
MPI_Gatherv(outputB, sendcounts[rank], MPI_INT, output->B, sendcounts, displacement,
    MPI_INT, 0, MPI_COMM_WORLD);
                                ....
```

Finally we use the rank 0, to join all the work realized by the different processors and generate the final output.

```
                                ....
  if(rank==0){
                                ....
       // Image writing
       if (saveFile(output, argv[3])!=0) {
           printf("Error saving the image\n");
           free(source);
           free(output);
           return -1;
                                ....
   }
                                ....
```

7 Results

7.1 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 1: Serial time results

		Runtime (s)		
	Kernel Size	**2 Cores**	**4 Cores**	**8 Cores**
Im01	5x5	0.1418	0.1324	0.1230
	25x25	0.4067	0.2756	0.2071
	49x49	1.3703	0.7491	0.4259
Im03	5x5	0.4410	0.3751	0.4756
	25x25	1.9472	1.1472	0.8332
	49x49	6.4370	3.4221	1.9264
Im04	5x5	15.8674	15.2293	16.9809
	25x25	90.7215	53.3810	35.8538
	49x49	333.4752	177.74	96.7325

Table 2: Parallel time results

		Speedup			Efficiency		
	Kernel Size	**2 Cores**	**4 Cores**	**8 Cores**	**2 Cores**	**4 Cores**	**8 Cores**
Im01	5x5	0,9528	1.0204	1,0984	0,4764	0.2551	0,1373
	25x25	1,6919	2.4967	3,3225	0,8460	0.6242	0,4153
	49x49	1,7161	3.1392	5,5215	0,8581	0.7848	0,6902
Im03	5x5	1,0961	1.1981	0,9449	0,5480	0.2995	0,1181
	25x25	1,7128	2.9073	4,0029	0,8564	0.7268	0,5004
	49x49	1,9078	3.5885	6,3747	0,9539	0.8971	0,7968
Im04	5x5	1,1727	1.2070	1,0825	0,5864	0.3017	0,1353
	25x25	1,7967	3.0536	4,5463	0,8984	0.7634	0,5683
	49x49	1,9093	3.5820	6,5820	0,9546	0.8955	0,8228

Table 3: Speedup and Efficiency

7.2 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.

As we can observe in Table 3, the benchmarks showed a great speedup when the computation is high, which means we have excellent results when the image and kernel has a large size or computation cost. The same happens with the efficiency, when we use the minimum size such as im01 with the kernel 5x5, the efficiency is reduced considerably, but in the opposite case, when we use a high image and complex kernel, the efficiency increases a lot and it has at least 80% of efficiency.

Therefore we can say that this problem, using parallelization, does not have to pose any scalability problem, so it is highly scalable. Also, the strategy followed, we can conclude that the load for each node is very well balanced, because every core has the same size of task. 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. However, if the input data grows in the future, assign the task of reading the inputs to one core and share them afterwards between the rest of the cores might be a good practice.

According to the overhead of parallelization, do not exist dependencies between the different channels or even in the same task for the computations, for this reason, we can say that we do not have any additional overhead.

8 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 8 nodes 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 number of cores 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 80%.

Finally, using MPI we were able to divide the data and domain of the problem as in the previous OpenMP version was not possible because does not provide communicattion. Concluding in better results as a consequence of a deeper parallelization over the convolution computation.

In addition, we expect combine these two technologies into an Hybrid program, since we have never worked on one, and would be great to test its performance and see how to integrate the parallel code all together.

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

References

[1] MPI Tutorial

[2] MPI Scatter and Gather

[3] Parallel Programming on Distributed Memory environments - Chapter 4 - Presentation

[4] Parallel Hardware Architectures & Parallel Programming Models - Chapter 4 - Preliminaries

[5] MPI - Message Passing Interface - Chapter 4 - Part 1

[6] MPI - Message Passing Interface - Chapter 4 - Part 2

[7] MPI - Message Passing Interface - Chapter 4 - Part 3

[8] MPI - Message Passing Interface - Chapter 4 - Part 4

9 Code repository

HPC Project: Convolution Github Repository