



University of Lleida

Master's Degree in Informatics Engineering

Higher Polytechnic School

Hybrid Implementation

High Performance Computing

Francesc Contreras

Albert Pérez

June 11, 2021

Table of contents

1	Introduction	1
2	Purpose & Objectives	1
3	Environment	1
4	Hybrid Solution	2
5	Results	5
5.1	Execution time results	5
5.2	Speedup and Efficiency	6
5.3	Scability of the program	6
6	Comparison	7
6.1	OpenMP vs Hybrid	7
6.2	MPI vs Hybrid	9
7	Conclusions	11
8	Code repository	12
	Appendices	13
A	OpenMP Results	13
B	MPI Results	14

List of Tables

1	Serial runtime	5
2	Hybrid runtime	5
3	Hybrid Speedup and Efficiency	6
4	OpenMP runtime	13
5	OpenMP Speedup and Efficiency	13
6	MPI runtime	14
7	MPI Speedup and Efficiency	14

List of Figures

1	OpenMP Speedup	7
2	Hybrid Speedup	7
3	OpenMP Efficiency	8
4	Hybrid Efficiency	8
5	MPI Speedup	9
6	Hybrid Speedup	9
7	MPI Efficiency	10
8	Hybrid Efficiency	10

1 Introduction

This document is the report consisting of a High-Performance computing project part 3, which is based on solving a specific image convolution problem, now, by combining into a hybrid solution the two previous approaches which correspond on one side to the OpenMP¹ implementation and on the other, the MPI² solution.

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

Finally, are specified the sections related to the Hybrid solution, the performance and scalability analysis, the comparison between the different solutions implemented in this project for different images and processes sizes, and the overall conclusions.

2 Purpose & Objectives

Specifically, the main aim is to reach an hybrid solution, where joining the two previous programming models result an implementation with a higher performance as the objective regards on taking the best strengths of both models.

Therefore, once the hybrid solution is designed and implemented we would able to make a performance and scalability comparison to analyze the effects of design decisions and which solution fits better in different cases.

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 *Universitat 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.

¹Open Multi-Processing

²Message Passing Interface

4 Hybrid Solution

As commented, there are two previous parts of this project. The first one corresponds to the OpenMP solution focused on creating an explicit parallelization, and the second one corresponds to the MPI centered on the communication between nodes/cores in charge of carrying out the tasks. But now, the objective rises on designing a hybrid solution using both programming models' strengths, achieving, on one hand, good inter-node communication and an explicit thread parallelization of the task work amount.

In short, as argued in their respective individual solutions, the MPI should be used to split large problems into small parts between all the available computational resources to diminish the execution time, meaning, to distribute among all the resources the amount of work to do. And, the OpenMP parallelize in different thread execution the work node must do.

With that in mind, will be performed a static image division regarding the computational resources available becoming the data image portions that will be computed by each node, following the MPI design implementation explained in a previous assignment.

Once the MPI environment interface is initialized, is able to perform the distribution regarding the amount of work of the specific problem, in our case the convolution over an image, among the total number (size) of computational nodes identified by their process ID (rank) over the world communicator.

You may see in the next MPI flow code extraction, all these previous points commented which stand as the MPI sections over the program, distributing the work and gathering afterward the results to be joined into the resulting convolved image.

```
....  
  
// Initialize the MPI environment  
MPI_Init(&argc,&argv);  
MPI_Status status;  
  
int rank;  
MPI_Comm_rank(MPI_COMM_WORLD,&rank);  
  
int size;  
MPI_Comm_size(MPI_COMM_WORLD,&size);  
  
// Load Balancing  
int *sendcounts = malloc(sizeof(int)*size);  
int *displacement = malloc(sizeof(int)*size);  
  
int sizePerCore = (source->width*source->height)/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;

// arrays memory allocation
int *receiveR = malloc(sizeof(int)*(sizePerCore*2));
....

int *outputR = malloc(sizeof(int)*(sizePerCore*20));
....

// Scatter to distribute the task
MPI_Scatterv(source->R,sendcounts,displacement,MPI_INT,receiverR,sendcounts[rank],
    MPI_INT,0,MPI_COMM_WORLD);
....

// Task work convolve computation
convolve2D(receiverR, outputR, source->width, height, kern->vkern, kern->kernelX,
    kern->kernelY);
....

// Gathering all partial results
MPI_Gatherv(outputR,sendcounts[rank],MPI_INT,output->R, sendcounts,displacement,
    MPI_INT,0,MPI_COMM_WORLD);
....

```

Then, once each node has its amount of work, it is where OpenMP enters. So, in order to to the convolution, as also detailed in its previous assignment, the convolve computation will be parallelized using a scheduled dynamic strategy meaning each thread will execute a chunk of iterations and then requests another chunk until there are no more chunks available.

The dynamic scheduling type is appropriate when the iterations require different computational costs as regards this kind of problem as there are nested loops in the *convolve2D* function.

Further, regards the first OpenMP implementation where there were sections to execute in parallel the three *convolve2D* tasks one for each image color RGB channel since there are no dependencies between them, actually, in this hybrid implementation as we are using MPI for distribution these higher level parallelization is implicit done by the MPI programming model well-explained before respecting the image data.

Below, there is the corresponding OpenMp flow code extraction where you may observe how internally the convolve computation is done.

```

// OpenMP
int convolve2D(int* in, int* out, int dataSizeX, int dataSizeY, float* kernel,
int kernelSizeX, int kernelSizeY)
{
....
// OpenMP convolve FOR parallelization
#pragma OpenMP 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;
....
}
....
}

```

In addition, we recognize using all the threads that are available for each node, regarding the environment, 4 threads.

The reason is mere, just because in the first assignment we realized that the most efficient scenario when employing OpenMP results with the total available threads, particularly if the convolution calculation has a high cost.

Therefore, based on those statistics, the hybrid solution was tested granting this constraint, and all the next sections results rest on the previous premises.

5 Results

5.1 Execution time results

	Img Resolution	Kernel Size	Runtime (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 runtime

		Runtime (s)		
	Kernel Size	2 Cores	4 Cores	8 Cores
Im01	5x5	0.1074	0.1975	0.1991
	25x25	0.3544	0.3066	0.2272
	49x49	1.3020	0.6715	0.4504
Im03	5x5	0.3508	0.4937	0.5289
	25x25	1.7526	1.3240	0.9094
	49x49	6.71160	3.5310	1.8876
Im04	5x5	15.2716	15.4064	17.4811
	25x25	91.5990	60.4009	39.1814
	49x49	340.6425	184.5493	99.6861

Table 2: Hybrid runtime

5.2 Speedup and Efficiency

	Kernel Size	Speedup			Efficiency (%)		
		2 Cores	4 Cores	8 Cores	2 Cores	4 Cores	8 Cores
Im01	5x5	1.2583	0.6842	0.6786	62.9174	17.1042	8.4827
	25x25	1.9416	2.2441	3,0292	97.0787	56.1020	37.8648
	49x49	1.8061	3.5022	5,2211	90,3051	87.5538	65.2641
Im03	5x5	1.2811	0.9102	0.8497	64.0547	22.7562	10.6217
	25x25	1.9030	2.5191	3.6673	95.1507	62.9770	45.8413
	49x49	1.8285	3.4779	6.5057	91.4252	86.9464	81.3210
Im04	5x5	1.2036	1.1931	1.0515	60.1820	29.8276	13.1438
	25x25	1.7795	2.6987	4.1602	88.9766	67.4673	52.0028
	49x49	1.8691	3.4500	6.3870	93.4551	86.2501	79.8376

Table 3: Hybrid Speedup and Efficiency

5.3 Scability of the program

As seen in Table 3, the efficiency of the Hybrid solution achieves its maximum rate when the scenario uses 2 cores/nodes, because the tasks are further well-balanced than using more. Additionally, employing 4 or 8 cores the efficiency is progressively reduced, however, the best runtime performances result with all the "power" (max. cores and threads) of the nodes applied.

Therefore, those premises assure a non-scalability issue, regarding the results reference got. Further, much of this conclusion appears as there are no direct dependencies between tasks, thanks, to the kind of problem, and the decomposition carried out.

In terms of results, the efficiency gets higher as much computational resources are applied, also getting better times than previous standalone implementations. 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 direct proportional progression in mathematics terms.

In Figure 2 on page 7 may be seen the progression mentioned for the hybrid solution, because when image gets larger and more computational resources are used, the efficiency tends at its maximum.

6 Comparison

This section is focused on comparing the Hybrid solution performance in terms of Speedup and Efficiency with the previous implementations using separately, the OpenMP and MPI programming models. Thus, some figures are detailing graphically the comparison as is better than using numbers, even though you will find the sources as two appendices, one for each model.

6.1 OpenMP vs Hybrid

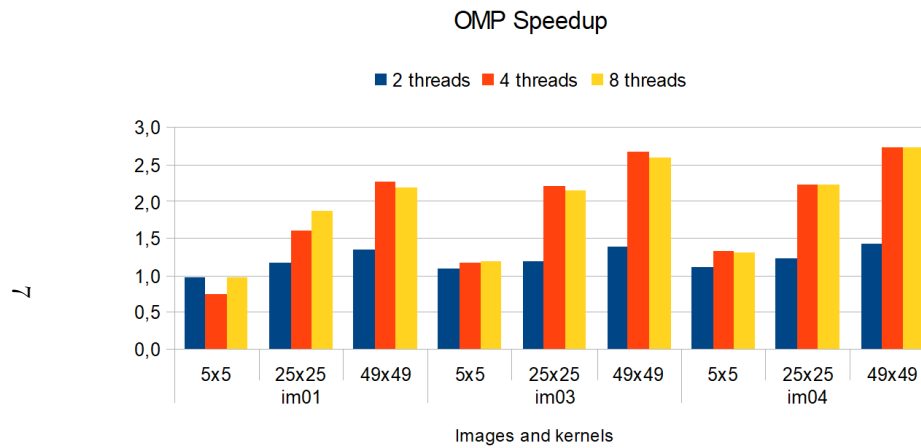


Figure 1: OpenMP Speedup

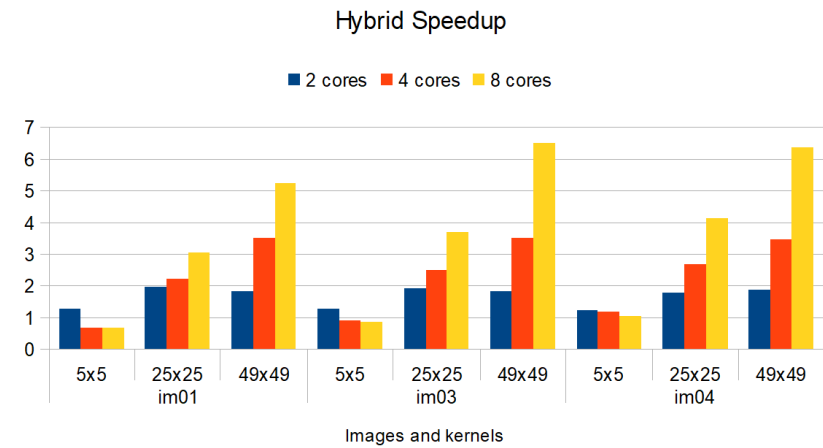


Figure 2: Hybrid Speedup

First of all, we will make a comparison between OpenMP vs Hybrid. Regards the values, they reflect the Speedup is more constant when using OpenMP which means that the distribution of the workload between the threads is more stable compared to the distribution per core and threads. Nonetheless, the hybrid implementation stands out in Speedup, when more cores per thread run.

In addition, there is a correlation over the 4 threads and 8 threads on the OpenMP implementation as the cluster environment only it is up to 4 per core. And obviously, for the hybrid implementation as based on the results obtained and also as commented previously, we chose 4 threads to run the solution.

Moreover, regards the hybrid speedup compared to the OpenMP implementation, we may observe how the difference is bigger when 8 cores with 4 threads are used, thus meaning the tasks are well distributed and the nodes used all computation resources available to run their own part.

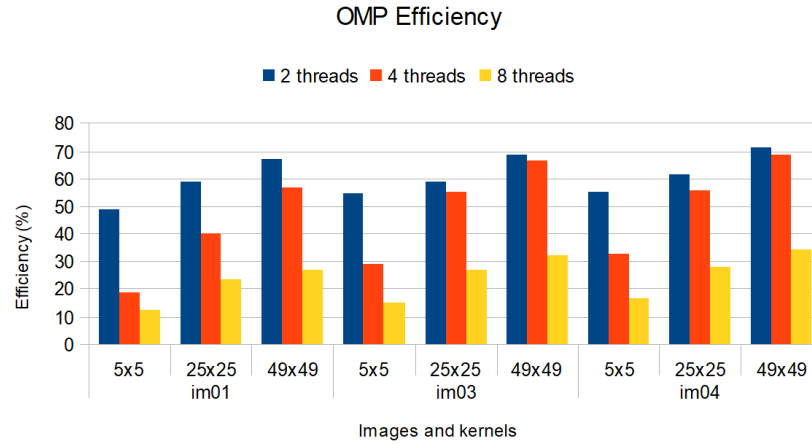


Figure 3: OpenMP Efficiency

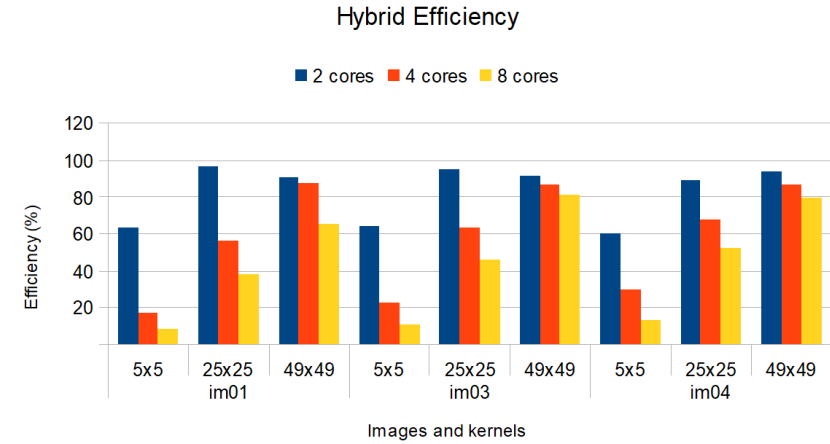


Figure 4: Hybrid Efficiency

Regarding the efficiency, we can say that it is almost equal in both parts, but OpenMP stands out in some points and Hybrid in others.

We must pay attention when there exit a lower image size and are used a lot of computational resources, thus, as expected, the efficiency in both cases is horrible. That's the reason why in lower sizes the 2 threads/cores highlights, but when getting bigger they work better.

And, if we join both results, the Speedup and Efficiency, we conclude that even in the efficiency graphs the 4 thread/cores are equally categorized, the speedup sets up the big difference when greater the image becomes.

Also, must be said, that the OpenMP Efficiency 8 threads results are not real, just the same commented on the Speedup. But it is enlightening knowing the computational runtime is affected when trying to use more resources not available via software.

6.2 MPI vs Hybrid

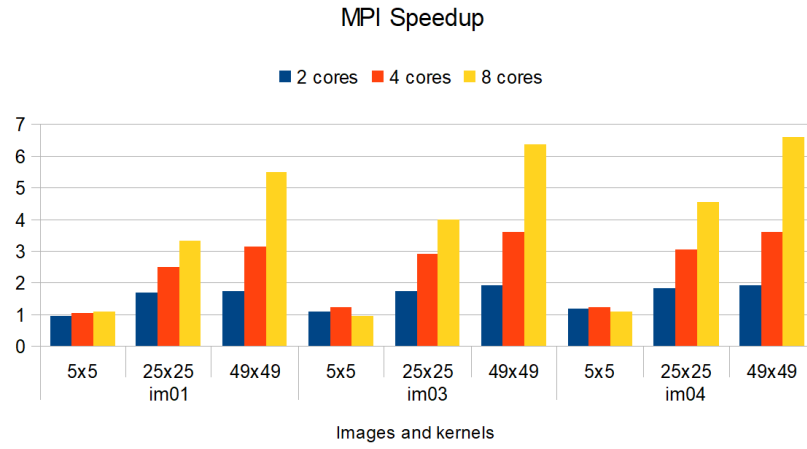


Figure 5: MPI Speedup

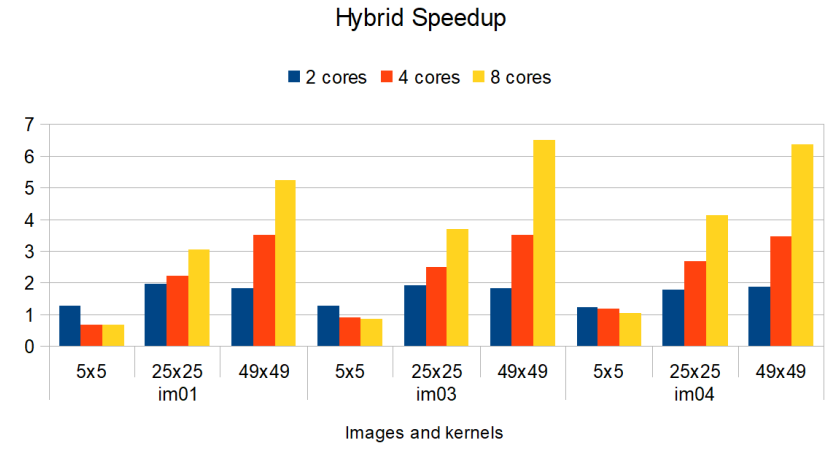


Figure 6: Hybrid Speedup

On the other hand, the implementation between hybrid and MPI does not have much difference, in general terms both have a good speedup, although it is true that there are small improvements when using 2 and 4 cores with MPI.

This equality presented over the results, It has sense because the main code of hybrid solution use MPI and only one piece of code use OMP.

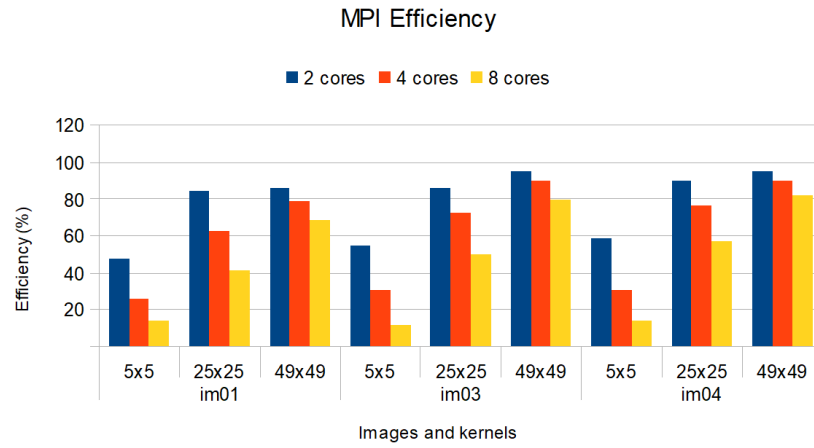


Figure 7: MPI Efficiency

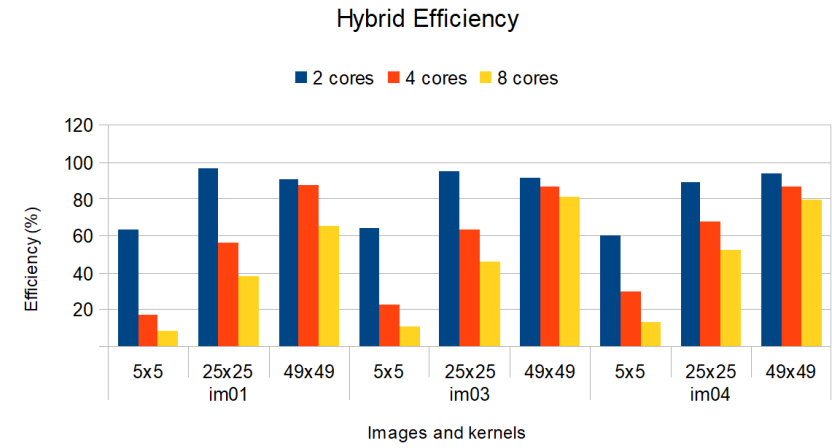


Figure 8: Hybrid Efficiency

10

We can also see how their efficiency is almost identical in all scenarios, however, when we use only MPI, the efficiency is more stable.

Accordingly, we can conclude that the results are good, meaning that this problem can have benefits being parallelized and, depending on the available resources, be split between different computers.

7 Conclusions

We were very pleased to be able to put our knowledge acquired in class to the test by developing these projects. Based on that knowledge and the results obtained after executing the hybrid application with various entries, we have found that the results are satisfactory.

Our initial thoughts were that the Hybrid implementation would greatly outperform the OpenMP and MPI, however, due to our very high Idle Time indexes, the hybrid implementation ended up with worse average performances than the other implementations. Also, the execution could improve, but this is due to a better partitioning of the problem, which indicates a poor judgment on the project. We think that the project can also benefit from a performance improvement if the joining of the pixels and the generation of the image were made in parallel.

References

- [1] [Distributed Computing Group — Polytechnic School — University of Lleida](#)
- [2] [Introduction to OpenMP — Francesc Giné, Josep L. Lèrida \(Group of Distributed Computing\), University of Lleida](#)
- [3] [Moore Cluster Home Webpage](#)
- [4] [HPC Project: Convolution Operation with OpenMP](#)
- [5] [OpenMP Microsoft Summary](#)
- [6] [OpenMP Scheduling comparison](#)
- [7] [MPI Tutorial](#)
- [8] [MPI Scatter and Gather](#)
- [9] [Parallel Programming on Distributed Memory environments - Chapter 4 - Presentation](#)
- [10] [Parallel Hardware Architectures & Parallel Programming Models - Chapter 4 - Preliminaries](#)
- [11] [MPI - Message Passing Interface - Chapter 4 - Part 1](#)
- [12] [MPI - Message Passing Interface - Chapter 4 - Part 2](#)
- [13] [MPI - Message Passing Interface - Chapter 4 - Part 3](#)
- [14] [MPI - Message Passing Interface - Chapter 4 - Part 4](#)

8 Code repository

[HPC Project: Convolution Github Repository](#)

Appendices

A OpenMP Results

		Runtime (s)		
	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 4: OpenMP runtime

		Speedup			Efficiency (%)		
	Kernel Size	2 Threads	4 Threads	8 Threads	2 Threads	4 Threads	8 Threads
Im01	5x5	0.9747	0.7382	0.9782	48.73	18.45	12.22
	25x25	1.1716	1.5943	1.8688	58.58	39.85	23.36
	49x49	1.3407	2.2635	2.1763	67.03	56.58	27.20
Im03	5x5	1.0918	1.1681	1.1942	54.59	29.20	14.92
	25x25	1.1806	2.1991	2.1401	59.03	54.97	26.75
	49x49	1.3745	2.6655	2.5949	68.72	66.63	32.43
Im04	5x5	1.1006	1.3155	1.3111	55.03	32.88	16.38
	25x25	1.2304	2.2192	2.2336	61.52	55.48	27.92
	49x49	1.4237	2.7413	2.7401	71.18	68.53	34.25

Table 5: OpenMP Speedup and Efficiency

B MPI 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 6: MPI runtime

		Speedup			Efficiency (%)		
	Kernel Size	2 Cores	4 Cores	8 Cores	2 Cores	4 Cores	8 Cores
Im01	5x5	0,9528	1.0204	1,0984	47.64	25.51	13.73
	25x25	1,6919	2.4967	3,3225	84.60	62.42	41.53
	49x49	1,7161	3.1392	5,5215	85.81	78.48	69.02
Im03	5x5	1,0961	1.1981	0,9449	54.80	29.95	11.81
	25x25	1,7128	2.9073	4,0029	85.64	72.68	50.04
	49x49	1,9078	3.5885	6,3747	95.39	89.71	79.68
Im04	5x5	1,1727	1.2070	1,0825	58.64	30.17	13.53
	25x25	1,7967	3.0536	4,5463	89.84	76.34	56.83
	49x49	1,9093	3.5820	6,5820	95.46	89.55	82.28

Table 7: MPI Speedup and Efficiency