Higher Polytechnic School

Master's Degree of Computer Engineering

High Performance Computing

Universitat de Lleida

Hybrid OpenMP+MPI Implementation
HPC Project

Members:
Roger Truchero Visa
Gerard Donaire Alós

Date: June 8, 2020

Contents

Introduction **3**

Document Structure **4**

1 Experimentation configurations **4**

2 Analysis and Results Static mapping of tasks **5**
2.1 Speedup and efficiency obtained for different number of cores and problem size 6
2.2 Load balancing 8

3 Analysis and Results Dynamic mapping of tasks **9**
3.1 Speedup obtained for different number of cores and problem size 10
3.2 Load balancing 12

4 Strengths and Weaknesses of each implementation **13**
4.1 Static 13
4.2 Dynamic 13

5 Conclusions **14**

6 Bibliography **15**

List of Figures

1 Static problem partitioning . . . 3
2 Dynamic problem partitioning . . . 3
3 plots of speedup and efficiency with 2 processes . . . 6
4 plots of speedup and efficiency with 4 processes . . . 6
5 plots of speedup and efficiency with 8 processes . . . 7
6 plots of speedup and efficiency with 16 processes . . . 7
7 plots of speedup and efficiency with 2 processes . . . 10
8 plots of speedup and efficiency with 4 processes . . . 10
9 plots of speedup and efficiency with 8 processes . . . 11
10 plots of speedup and efficiency with 16 processes . . . 11

List of Tables

1 Execution table for each features with Static mapping of tasks and 2 processes. 5
2 Execution table for each features with Static mapping of tasks and 4 processes. 5
3 Execution table for each features with Static mapping of tasks and 8 processes. 5
4 Execution table for each features with Static mapping of tasks and 16 processes. . . . 5
5 Execution table for each features with Dynamic mapping of tasks and 2 processes. . . 9
6 Execution table for each features with Dynamic mapping of tasks and 4 processes. . . 9
7 Execution table for each features with Dynamic mapping of tasks and 8 processes. . . 9
8 Execution table for each features with Dynamic mapping of tasks and 16 processes. . . 9

Introduction

In this activity we are instructed to make an implementation combining both programming models, MPI and OpenMP. We will compare the performance of this hybrid solution in relation to the previous delivery for different problem and processes sizes.

MPI implementation is providing us the possibility to apply a data decomposition model. For this end, we can consider different options to implement the task mapping:

- **Static mapping of tasks:** It consists in dividing the region into a fixed number of fragments, which are processed in parallel by different processors/cores. Note that the allocation is done at the beginning of the execution of the program and cannot be modified.Can be classified into quadrants, rows, columns, etc. Figure 2 shows an example.

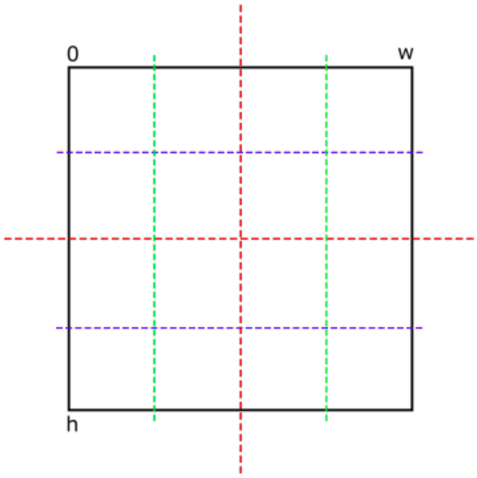


Figure 1: Static problem partitioning

- **Dynamic mapping of tasks:** In this case, the fragments will be mapped to different processors/cores as soon as they finish with the previous calculations.

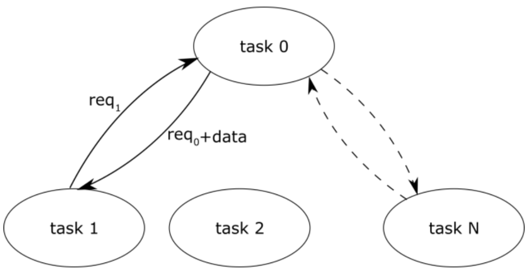


Figure 2: Dynamic problem partitioning

Document Structure

This document will be based on the following sections:

- **Analysis Static mapping of tasks:** In this section we will analyse the speedup obtained for different number of cores and problem size using a static mapping of tasks. We will analyze the load balancing too.
- **Analysis Dynamic mapping of tasks:** In this section we will analyse the speedup obtained for different number of cores and problem size using a dynamic mapping of tasks. We will analyze the load balancing too.
- **Strengths and Weaknesses of each implementation:** In this section we will explain the strengths and weaknesses of each implementation and we discuss the scenarios where we think it would be more convenient to use each one and why. We will consider on the discussion the effects of heterogeneity on the execution nodes.
- **Experimentation configurations:** In this section we will explain what is the configuration that we used in order to analyse the performance and scalability of our solutions.

1 Experimentation configurations

In order to analyse the performance and scalability of our solutions we experiment the configurations that is shown below:

- **Number of processes:** 1, 2, 4, 8, 16, 32.
- **Number of threads:** 1, 2, 4, 8, 16.
- **Number of iterations:** 10^4, 10^5, 10^6.
- **Image size:** 600x400, 3000x2000, 6000x4000, 30000x20000.

We also have to say that for the realization of all the tests **we have used only the dynamic schedule to parallelize the loop with openmp**, since in the previous practice it was with the one that we obtained better results over the guided and the static.

2 Analysis and Results Static mapping of tasks

In the following tables we are going to observe the results obtained with the configurations that are shown in **section 1**. We will discuss about this results in order to compare if the expectations we had hoped for have been fulfilled or not.

		ITERATIONS											
		10^4				10^5				10^6			
2 PROCESSES		**600x400**	**3000x2000**	**6000x4000**	**30000x20000**	**600x400**	**3000x2000**	**6000x4000**	**30000x20000**	**600x400**	**3000x2000**	**6000x4000**	**30000x20000**
THREADS	**Serial**	4,58	113,51	453,55	11337,24	44,88	1122,9	4487,22	112290	448,55	11207,4	44855	1120743
	1	2,430397	59,7075	373,097	5907,75	23,3358	578,9386	2333,51	57892,6	232,524	6088,44	23252,6	608844
	2	2,287066	44,778	255,097	4477,2	12,0482	336,427	1204,45	33644,7	120,471	2978,49	12047,4	297849
	4	1,698835	36,8622	165,273	3688,24	12,0279	299,974	1202,12	29998,3	120,235	2969,01	12023,5	296901
	8	1,348815	35,7772	133,658	3579,13	11,9465	298,684	1194,96	29869,1	119,657	2985,21	11966,1	298523
	16	1,283182	32,2973	127,334	3229,29	11,9048	298,408	1191,06	29841,8	118,43	2984,98	11842	298499

Table 1: Execution table for each features with Static mapping of tasks and 2 processes.

		ITERATIONS											
		10^4				10^5				10^6			
4 PROCESSES		**600x400**	**3000x2000**	**6000x4000**	**30000x20000**	**600x400**	**3000x2000**	**6000x4000**	**30000x20000**	**600x400**	**3000x2000**	**6000x4000**	**30000x20000**
THREADS	**Serial**	4,58	113,51	453,55	11337,24	44,88	1122,9	4487,22	112290	448,55	11207,4	44855	1120743
	1	2,066892	50,9633	203,902	5096,34	19,85	509,66	1984,99	51000,2	197,771	5096,66	19777,1	509677
	2	1,778512	41,7844	165,611	4178,43	19,3156	418,022	1932,02	41802,3	193,709	4180,02	19370,9	418002
	4	1,550491	36,0626	144,867	3606, 27	13,9985	360,664	1400,88	36068,1	138,125	3606,55	13812,6	360658
	8	1,381072	33,8238	134,84	3382,87	12,5413	338,328	1255,16	33832,9	102,74	3383,75	10274	338576
	16	1,347229	32,7275	130,26	3272,88	12,3421	327,371	858,88	32737,8	62,4522	3273,97	6246,11	327299

Table 2: Execution table for each features with Static mapping of tasks and 4 processes.

		ITERATIONS											
		10^4				10^5				10^6			
8 PROCESSES		**600x400**	**3000x2000**	**6000x4000**	**30000x20000**	**600x400**	**3000x2000**	**6000x4000**	**30000x20000**	**600x400**	**3000x2000**	**6000x4000**	**30000x20000**
THREADS	**Serial**	4,58	113,51	453,55	11337,24	44,88	1122,9	4487,22	112290	448,55	11207,4	44855	1120743
	1	1,939824	31,3729	116,712	3137,3	11,6692	600,511	1167,01	60052,1	116,71	6006,23	11670,2	600687
	2	1,787932	26,4547	104,832	2645,47	10,4941	225,427	1050,34	22542,1	105,039	2254,3	10505,4	225432
	4	1,015711	20,2067	101,331	2020,71	10,1329	186,998	1013,29	18699,7	101,399	1869,99	10139,3	186998
	8	0,792221	18,199	101,238	1820,03	10,1229	202,605	1012,28	20261	101,222	2026,2	10122,9	202622
	16	0,770729	17,3165	85,1023	1731,65	8,588008	158,299	655,951	15830,4	85,89	1583	8589,08	158300

Table 3: Execution table for each features with Static mapping of tasks and 8 processes.

		ITERATIONS											
		10^4				10^5				10^6			
16 PROCESSES		**600x400**	**3000x2000**	**6000x4000**	**30000x20000**	**600x400**	**3000x2000**	**6000x4000**	**30000x20000**	**600x400**	**3000x2000**	**6000x4000**	**30000x20000**
THREADS	**Serial**	4,58	113,51	453,55	11337,24	44,88	1122,9	4487,22	112290	448,55	11207,4	44855	1120743
	1	0,831546	17,1538	69,1487	1715,46	6,568574	161,963	657,288	16197,3	62,393	1960,14	6239,56	196014
	2	0,741991	15,3458	61,7272	1535,02	6,307763	158,691	630,676	15868,1	55,3441	1414,73	5535,12	141473
	4	0,808847	14,7522	58,8352	1475,62	5,503276	133,104	550,322	13310	52,1339	1404,73	5213,99	140474
	8	0,650405	14,2855	56,7931	1428,66	5,144717	128,77	516,431	12877,9	50,9223	1274,37	5092,45	127439
	16	0,634062	13,9743	55,8556	1397,44	5,128598	126,294	512,989	12629,9	49,2689	1252,91	4926,78	125292

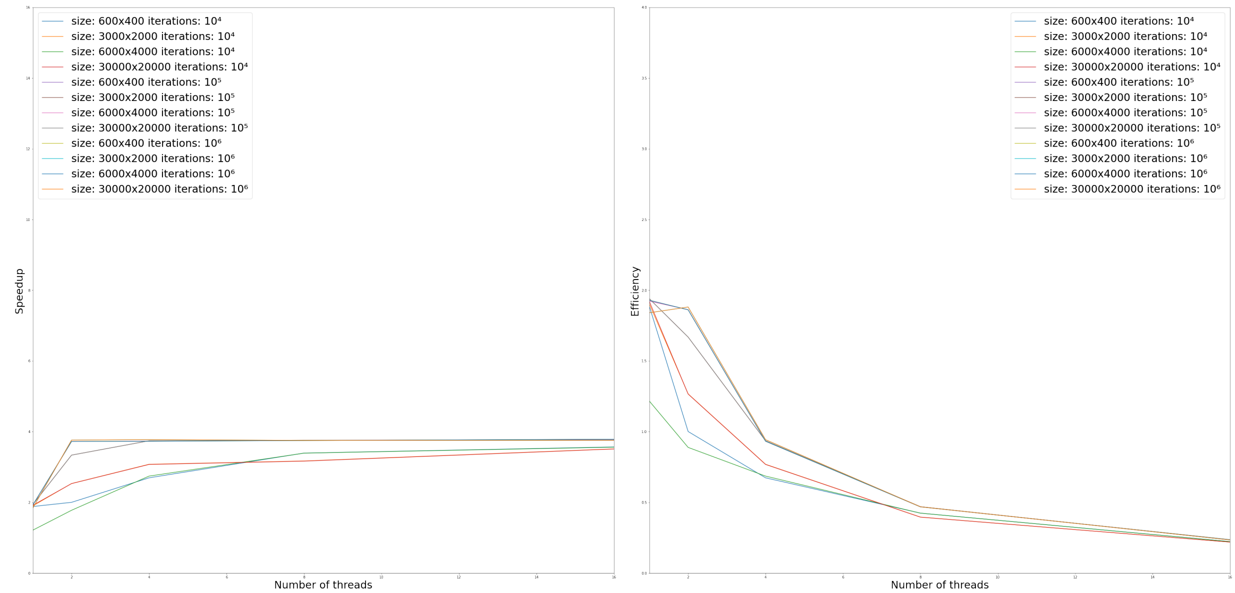
Table 4: Execution table for each features with Static mapping of tasks and 16 processes.

Once we have carried out all the executions in the cluster we can observe the following results. These show a clear decrease of the execution time as we increase the number of processes. The progression is not 100% linear, since each node of the cluster has 4 processes, when we are using different nodes, these have to lose time in their communication, being thus affected the performance.

We can also observe how as we increase the number of threads, the execution time also improves enormously, decreasing it up to 5% in some cases. As in the previous purely openmp version, we can see the same progression by increasing the size of the image and the number of iterations, so it seems that this relationship has not changed.

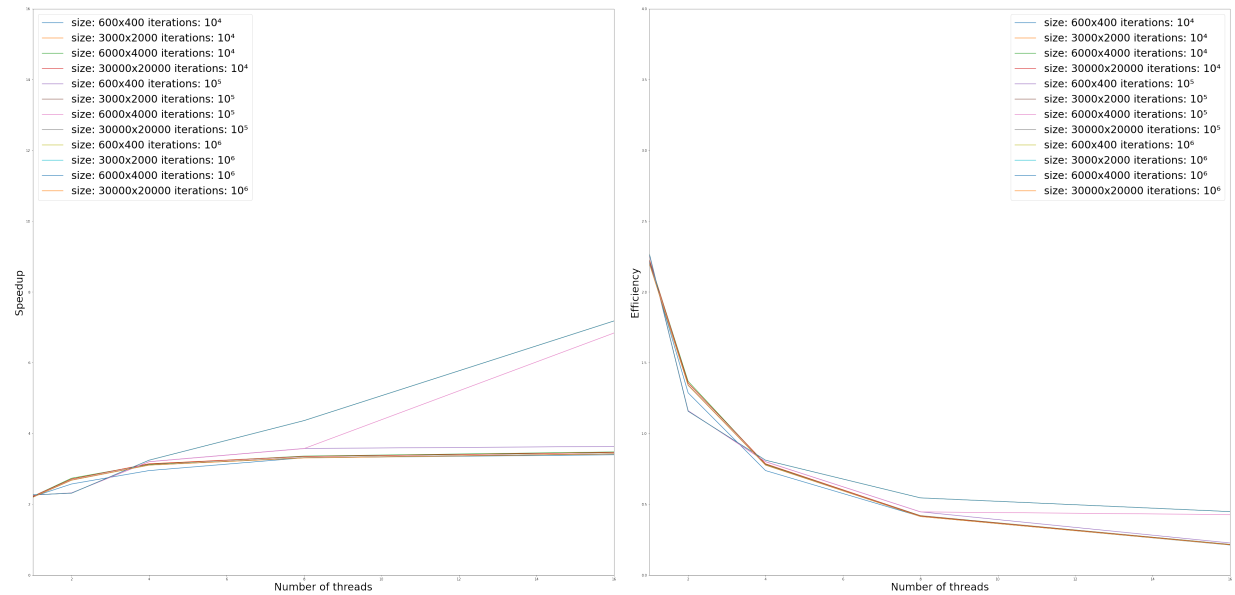
2.1 Speedup and efficiency obtained for different number of cores and problem size

In the following figures we can see the **static mapping speedup** and **efficiency** obtained using **2, 4, 8, 16 processes** and applying the experimentation configurations that we mention in **section 1**:



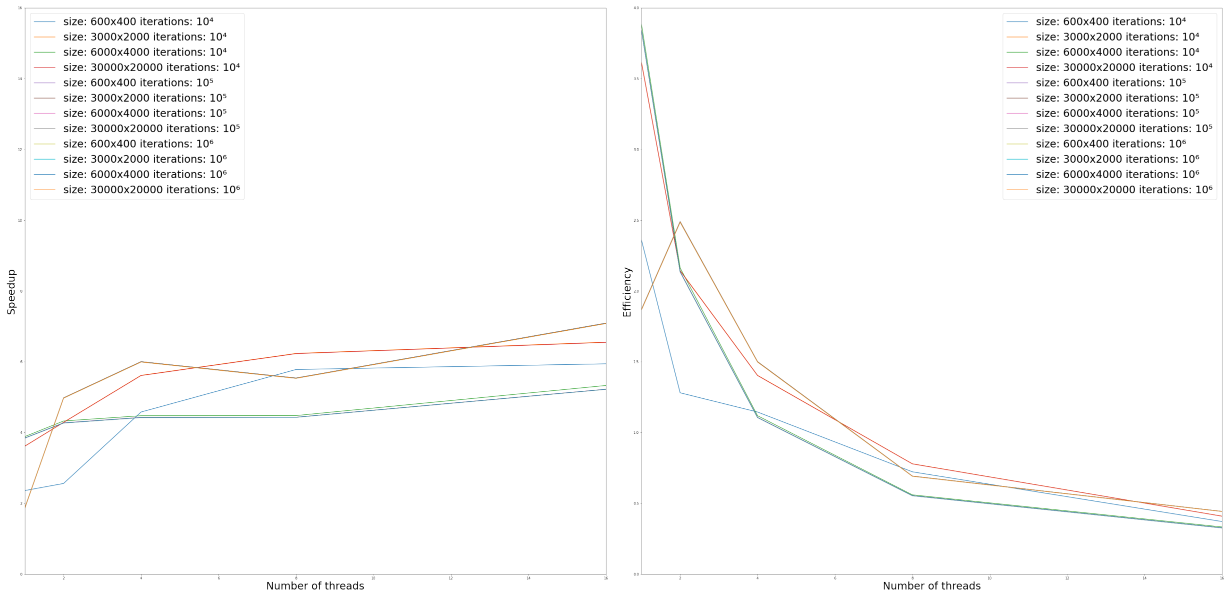
(a) Static speedup (b) Static efficiency

Figure 3: plots of speedup and efficiency with 2 processes



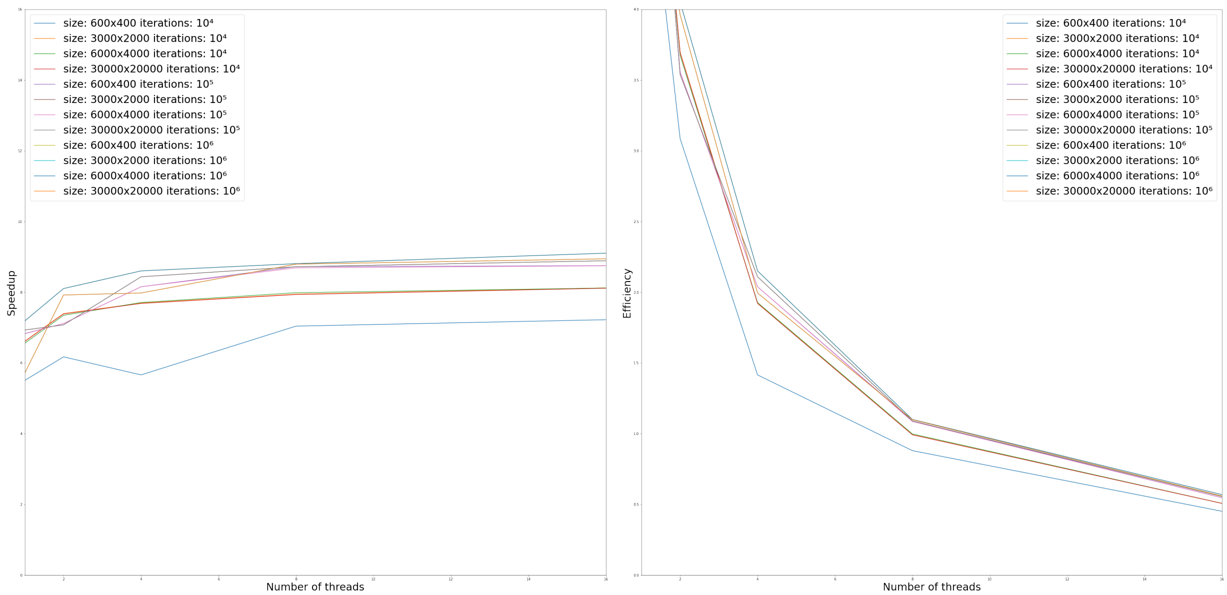
(a) Static speedup (b) Static efficiency

Figure 4: plots of speedup and efficiency with 4 processes



(a) Static speedup (b) Static efficiency

Figure 5: plots of speedup and efficiency with 8 processes



(a) Static speedup (b) Static efficiency

Figure 6: plots of speedup and efficiency with 16 processes

2.2 Load balancing

To perform the computations and decomposition in the static method, what we have done is to assign an equal load volume for the first N-1 processes, and to distribute the remaining load for the last process, which always absorbs most of the computation when the task distribution is not symmetrical.

The **decomposition of the work has been done by blocks of rows**, in this way each process calculates the assigned rows and generates its piece of the final image. When all the processes have finished the process with range 0 (master) is in charge of concatenating all the .ppm files and converting them into an image with the global result of all the computations made by all the processes, yours included making use of the following command:

```
convert
mandelbrot_hybrid_static_1.ppm
mandelbrot_hybrid_static_2.ppm
...
mandelbrot_hybrid_static_{n}.ppm
-append mandelbrot_hybrid_static.png
```

To **control when all the processes have finished**, we use the function MPI_Barrier(MPI_COMM_WORLD), which blocks the step so that until all the processes are finished all the information cannot be gathered, since we will not have it in any other case.

Using this type of strategy allows us to avoid over-communication between processes and therefore focus on purely arithmetic operations by increasing the performance of our code.

3 Analysis and Results Dynamic mapping of tasks

In the following tables we are going to observe the results obtained with the configurations that are shown in **section 1**. We will discuss about this results in order to compare if the expectations we had hoped for have been fulfilled or not.

		ITERATIONS											
		10^4				10^5				10^6			
2 PROCESSES		**600x400**	**3000x2000**	**6000x4000**	**30000x20000**	**600x400**	**3000x2000**	**6000x4000**	**30000x20000**	**600x400**	**3000x2000**	**6000x4000**	**30000x20000**
THREADS	**Serial**	4,58	113,51	453,55	11337,24	44,88	1122,9	4487,22	112290	448,55	11207,4	44855	1120743
	1	4,706125	119,937	475,623	11993,8	46,028027	1756,25	4602,852	175623	549,203	17554,4	54920,9	1755544
	2	4,570051	200,31	490,494	20032,2	59,533632	1677,06	5954,231	167717	503,526	16770,9	50356,2	1677702
	4	6,427663	143,51	585,712	14353,9	51,257675	1566,23	5126,964	156627	463,635	15659,7	46369	1565997
	8	5,077286	154,127	550,655	15412,7	49,257833	1308,42	4925,842	130851	600,147	13085,6	60011,2	1308567
	16	5,423158	125,722	535,664	12569,9	50,978249	1349,98	5098,241	135100	503,214	13510	50333,3	1351000

Table 5: Execution table for each features with Dynamic mapping of tasks and 2 processes.

		ITERATIONS											
		10^4				10^5				10^6			
4 PROCESSES		**600x400**	**3000x2000**	**6000x4000**	**30000x20000**	**600x400**	**3000x2000**	**6000x4000**	**30000x20000**	**600x400**	**3000x2000**	**6000x4000**	**30000x20000**
THREADS	**Serial**	4,58	113,51	453,55	11337,24	44,88	1122,9	4487,22	112290	448,55	11207,4	44855	1120743
	1	1,628936	40,3468	219,677	4035,45	15,830185	395,34	1584,236	39534,1	164,984	3955,66	16643,4	399043
	2	2,326891	51,4586	278,258	5144,56	20,165345	447,829	2016,954	44682,9	199,985	4487,21	20174,2	452665
	4	3,650229	62,3084	232,519	6230,88	22,634003	512,578	2265,126	51258,5	174,078	5150,98	17560,8	519625
	8	3,441584	63,9953	241,161	6398,9	21,897097	473,036	2189,692	47302,1	203,635	4728,09	20542,1	476964
	16	3,744399	77,1329	199,315	7713,43	25,246738	514,593	2526,235	51462,1	182,115	5149,92	18371,5	519518

Table 6: Execution table for each features with Dynamic mapping of tasks and 4 processes.

		ITERATIONS											
		10^4				10^5				10^6			
8 PROCESSES		**600x400**	**3000x2000**	**6000x4000**	**30000x20000**	**600x400**	**3000x2000**	**6000x4000**	**30000x20000**	**600x400**	**3000x2000**	**6000x4000**	**30000x20000**
THREADS	**Serial**	4,58	113,51	453,55	11337,24	44,88	1122,9	4487,22	112290	448,55	11207,4	44855	1120743
	1	0,740535	17,4896	77,4285	1748,99	6,821734	169,721	683,175	16971,3	80,6849	1697,32	8139,41	171223
	2	1,074771	19,9385	78,7453	1993,87	8,340337	203,565	836,032	20356,9	67,1468	2035,68	6773,76	205357
	4	1,653315	26,8415	94,5993	2685,78	9,64869	233,053	964,81	23305,7	87,6271	2330,81	8839,73	235120
	8	1,355942	29,7363	103,703	2976,12	9,780186	212,561	978,698	21257,1	80,6618	2125,98	8137,09	214466
	16	2,184625	37,051	102,689	3705,66	11,375947	208,346	1136,201	20834,8	79,8506	2089,99	8055,25	210836

Table 7: Execution table for each features with Dynamic mapping of tasks and 8 processes.

		ITERATIONS											
		10^4				10^5				10^6			
16 PROCESSES		**600x400**	**3000x2000**	**6000x4000**	**30000x20000**	**600x400**	**3000x2000**	**6000x4000**	**30000x20000**	**600x400**	**3000x2000**	**6000x4000**	**30000x20000**
THREADS	**Serial**	4,58	113,51	453,55	11337,24	44,88	1122,9	4487,22	112290	448,55	11207,4	44855	1120743
	1	0,442223	15,1301	52,4509	1514,98	3,210093	81,271845	321,121	8127,08	31,6878	812,59	3196,64	81973,2
	2	0,544728	13,2613	58,1015	1326,77	3,830822	102,786163	383,923	10276,1	35,2845	1027,04	3559,46	103606
	4	0,811092	14,0271	58,2458	1402,56	4,038936	109,137722	403,921	10913	40,6709	1091,25	4102,84	110804
	8	0,846968	16,7185	61,7413	1671,66	4,521432	81,112435	452,542	8112,02	43,2987	811,222	4367,93	81835,2
	16	0,86334	21,6718	67,408	2169,02	5,465427	91,95516	547,621	9199,56	43,9785	920,012	4435,65	92809,8

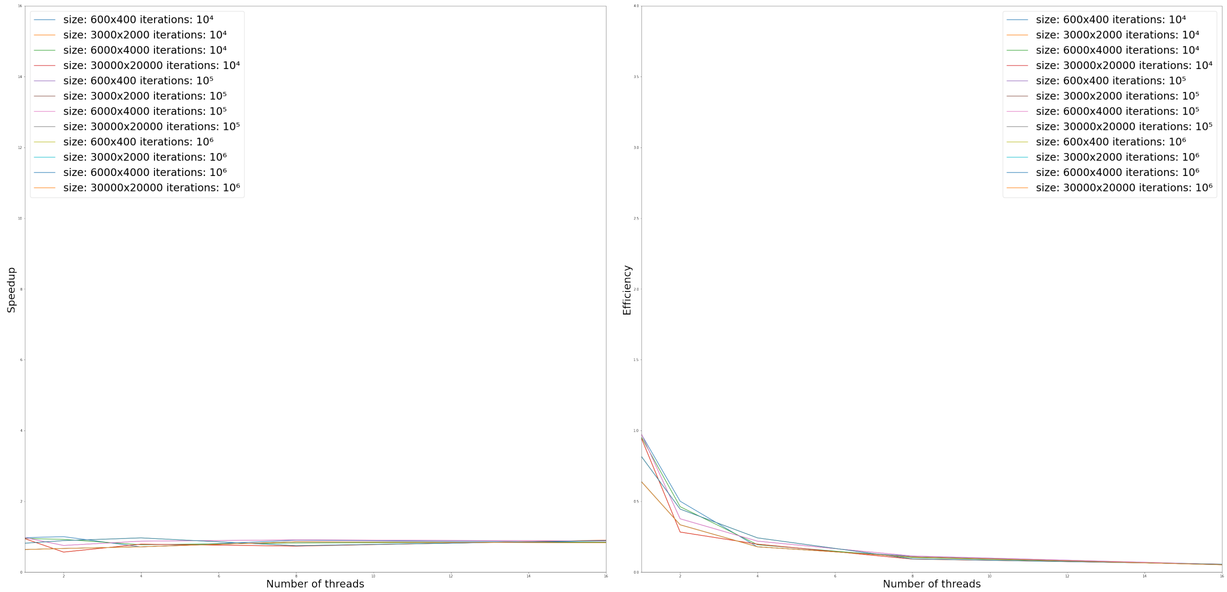
Table 8: Execution table for each features with Dynamic mapping of tasks and 16 processes.

In the same way as with the dynamic, once we have made the calculations we can observe the following. The first thing that visually impacts us is that as we increase the number of threads the performance gets drastically worse, this may be due to how the dynamic decomposition is done row by row, the volume of task to parallelize with openmp is minimal, and it takes more time to create and synchronize all the threads than to do the computation itself.

Similar to static mapping, the program reaches its highest performance when the number of processes increases, but unlike the previous one it reaches it with 16 processes and 1 thread, while in the case of static mapping it reaches it with 16 processes and 16 threads.

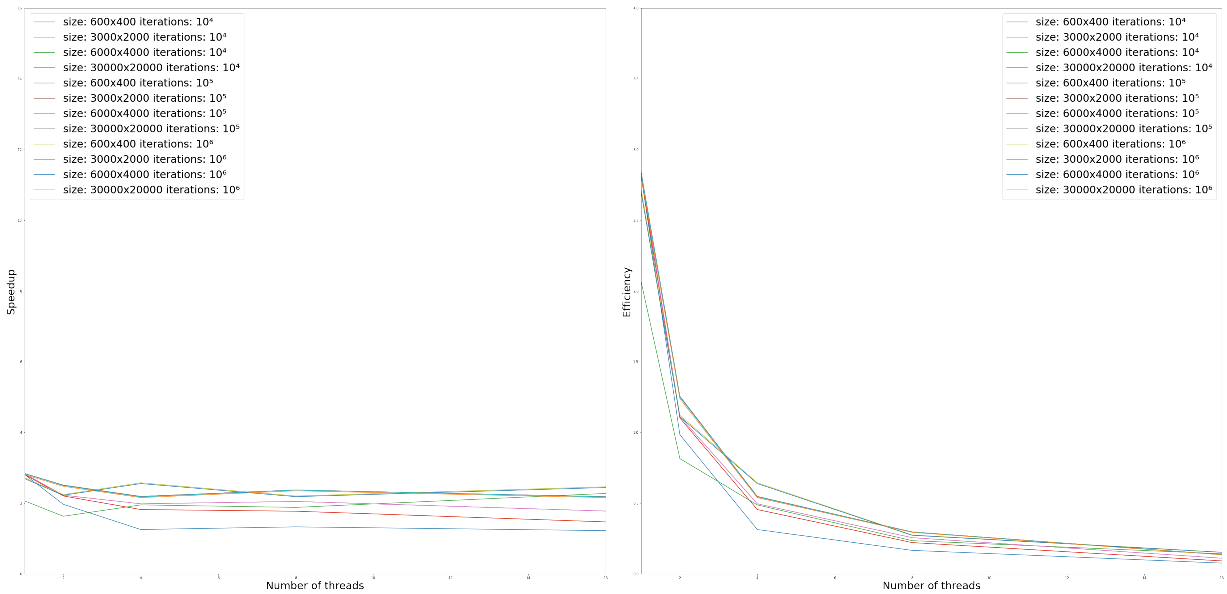
3.1 Speedup obtained for different number of cores and problem size

In the following figures we can see the **dynamic mapping speedup** and **efficiency** obtained using **2, 4, 8, 16 processes** and applying the experimentation configurations that we mention in **section 1**:



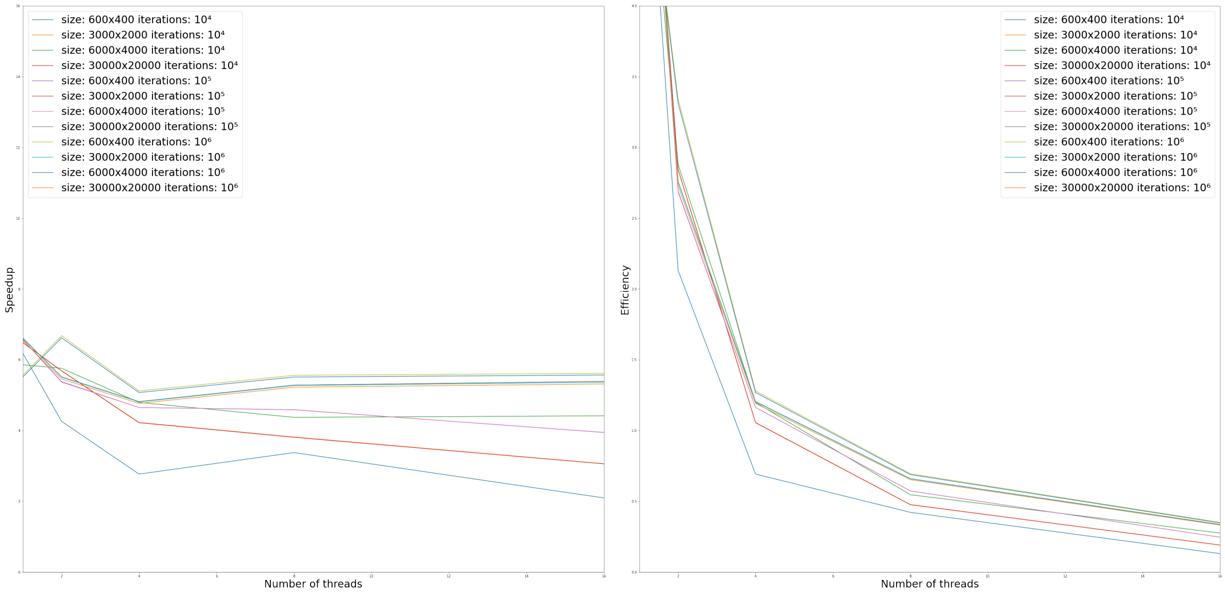
(a) Dynamic speedup (b) Dynamic efficiency

Figure 7: plots of speedup and efficiency with 2 processes



(a) Dynamic speedup (b) Dynamic efficiency

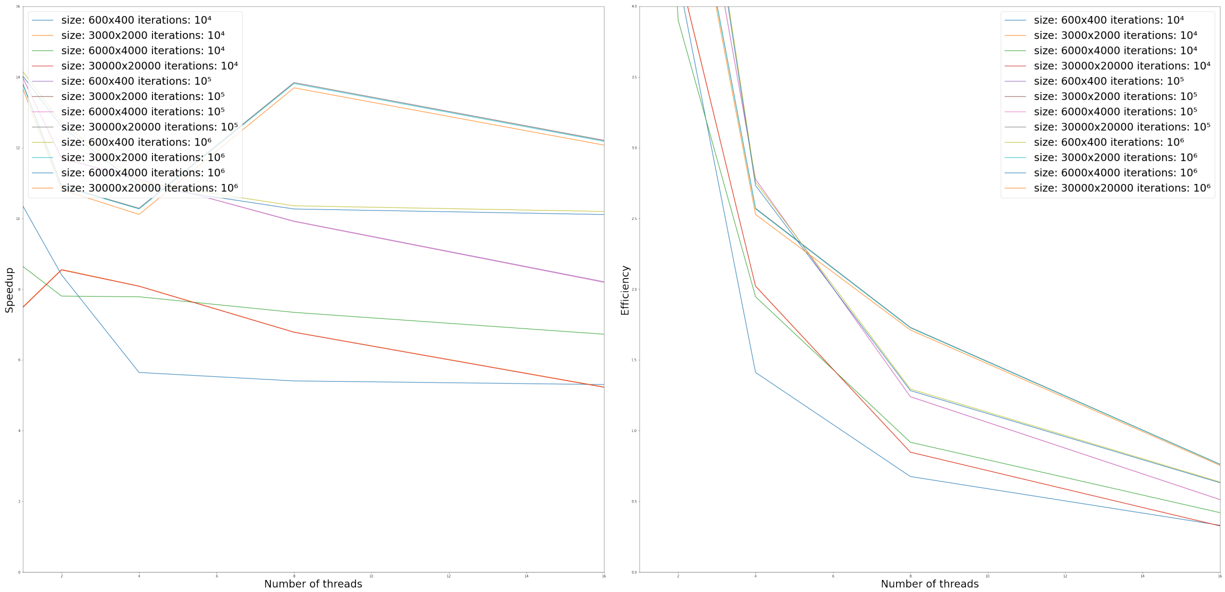
Figure 8: plots of speedup and efficiency with 4 processes



(a) Dynamic speedup

(b) Dynamic efficiency

Figure 9: plots of speedup and efficiency with 8 processes



(a) Dynamic speedup

(b) Dynamic efficiency

Figure 10: plots of speedup and efficiency with 16 processes

If we compare the previous figures with those shown in **subsection 2.1**, we can see that mostly for this problem static mapping gives a better speedup and efficiency than dynamic mapping regardless of the number of the processes applied.

3.2 Load balancing

In this case, it was a matter of partially breaking down the entire task into chunks. In our case we have broken it down into rows, where **each chunks ends up being a row**. In the case of the static strategy, all the processes were involved in the computation of the complex numbers, but in this case it is not like that since we have a master-slave architecture, where **the master assigns task and the slaves make the necessary computations**.

The logic that follows this strategy is the following. The **master** (process 0) waits to receive requests from the **slaves** (processes > 0). It can receive two types of requests:

- Request to be assigned a task (chunk)
- Request to deliver results for the task (chunk)

In the case that we want to assign a task to a slave, we will first check that there is a task available, that is, that there is still work to be done. If so, what we will do is to send you the chunk that you have to compute and it will make the necessary calculations, and then increase the next chunk to send. If we don't have work to do, we will send a **-1**, which means that you don't have to do any more calculations and therefore you can finish your execution.

If the case is that the results are being sent to us, what we will do is receive the pixels and save them in a three-dimensional matrix, in the corresponding place, so that later we can write them in the output file.

Every time we receive a computed task, we will increase the chunk counter, when we have received all the chunks and have finished all the slaves, then we can proceed to generate the output mandelbrot file and finish the master process.

We've talked a lot about the master, but haven't mentioned much about what the slave really does. What the slave does is ask for work indefinitely, until the master tells him that he has no more work to do, he keeps asking for work, making calculations and sending him the results.

4 Strengths and Weaknesses of each implementation

Mapping techniques used in parallel algorithms can be broadly classified into two categories: **static** and **dynamic**. The parallel programming paradigm and the characteristics of tasks and the interactions among them determine whether a static or a dynamic mapping is more suitable.

4.1 Static

Static mapping is often, though not exclusively, used in conjunction with a decomposition based on data partitioning. Static mapping is also used for mapping certain problems that are expressed naturally by a static task-dependency graph.

Strengths

- It takes advantage of all the processes to carry out the calculations.
- Low communication between processes, important when the number of rows is higher.
- Simple to implement and easy to understand.

Weaknesses

- Fixed data decomposition. Inefficient when the computation of the tasks is very variable.

4.2 Dynamic

Dynamic mapping is necessary in situations where a static mapping may result in a highly imbalanced distribution of work among processes or where the task-dependency graph itself if dynamic, thus precluding a static mapping. Since the primary reason for using a dynamic mapping is balancing the workload among processes, dynamic mapping is often referred to as dynamic load-balancing. Dynamic mapping techniques are usually classified as either centralized or distributed.

Strengths

- Flexible when computing tasks with different load balance. Promotes the equitable distribution of tasks and therefore equalizes the execution times of all slaves.

Weaknesses

- It does not use all the processes, since the master only manages the results of the slaves and therefore does not make any calculations.
- Over-communication between processes, increases the waiting time for the slaves when data decompisiton is very granular (master can only manage 1 request at the same time).
- More complex to implement, there are a greater number of race conditions to control.

5 Conclusions

To make a little synthesis and conclude, we have been able to observe the results for both types of mapping, static and dynamic, and we have been able to draw positive things from both as well as negative things.

For the static mapping method we have been able to see that it is undoubtedly the most suitable for this type of problem, given that the computations are very similar and do not require an excess of communication between the processes that make the calculations, although it is also interesting to say that this method is not very tolerant in terms of the real workloads that are assigned to each process. Another positive point is that all the processes are used, including the master, while the dynamic mapping one is not.

For the dynamic mapping method we can say that there is an excess of communication since we are constantly waiting for requests from the slaves to assign work or to receive the results. These processes are blocking, so until one is finished we cannot continue attending the others and consequently we limit our concurrence. A positive aspect of this type of decomposition is that we can assign a volume of specific tasks for each process if it has a different computing capacity than the rest or if we are simply interested in doing so for performance reasons.

We have also observed that the number of processes has a significant influence on the performance of both mappings, unlike the number of threads that seems to have more impact on the dynamic *(for the worse)* than on the static.

Regarding the results of the past practice, we have noticed a huge improvement from the hybrid (openmp + mpi) to the openmp, decreasing exponentially the execution times and taking better advantage of the cores.

So, to conclude, **we believe that the best solution for this problem is to use a hybrid solution that involves both MPI and OpenMP and perform a static mapping of the tasks as well as parallelizing the loop for openmp with a dynamic schedule.**

6 Bibliography

References

[1] *Static & Dynamic mapping of tasks:*

Consultation time: 5:05 p.m. on May 26, 2020 Available at:

http://www.scielo.org.co/scielo.php?script=sci_arttext&pid=S0012-73532014000300004

Consultation time: 3:16 p.m. on May 27, 2020 Available at:

https://ieeexplore.ieee.org/document/5335672/

Consultation time: 11:02 p.m. on May 28, 2020 Available at:

https://www.ece.uic.edu/~dutt/courses/ece566/lect-notes/lect7-task-part-map.pdf

Consultation time: 9:26 p.m. on May 29, 2020 Available at:

http://parallelcomp.uw.hu/ch03lev1sec4.html

Consultation time: 10:19 p.m. on May 30, 2020 Available at:

https://www.researchgate.net/publication/286159742_Static_and_dynamic_task_mapping_onto_network_on_chip_multiprocessors