HIGH PERFORMANCE COMPUTING
MPI PRACTICE

MARC PIÑOL

CONTENTS

1 INTRODUCTION

Our aim in this project was to speed-up the image convolution algorithm that was given to us. To do this we used MPI to parallelize the algorithm without changing its core specification and definition.

This parallellization was done in two different ways: the first one divided the problem space statically, in as many chunks as worker threads were available. The second approach divided the problem space in several chunks and gave one to each worker thread. When a thread finished, more work was assigned to it until the problem was solved.

2 DESCRIPTION OF THE TARGET MACHINE

The target machine is a 12-node heterogeneous cluster. Its nodes can be classified in two groups:

1. cabinet-0, comprised of 8 nodes with the following characteristics
   - Intel Core i5
     - 4 cores @ 3.10 GHz
   - 4 Gb of RAM
   - Total: 32 computation units with 1 Gb of main memory each
2. cabinet-1, comprised of 4 nodes with the following characteristics
   - AMD Athlon
     - 1 core @ 2 GHz
   - 1 Gb of RAM
   - Total: 4 computation units with 1 Gb of main memory each

There are 3 queues used to run the programs in the cluster

1. high.q - All cabinet-0 nodes

2. low.q - All cabinet-1 nodes
3. all.q - All cabinet-0 and cabinet-1 nodes

3 APPROACHES

3.1 *First approach: static task mapping*

The first chosen approach mapped the tasks statically: each one worked in a part of the problem until it ended. When all tasks solved their part, the problem was solved.

This approach implied dividing the problem space in as many chunks as threads were available. It was decided this would be done by dividing the number of rows of the source matrix between the amount of workers available:

```
row_number = source_rows \ thread_number
```

Once this was done, `row_number` would be used to divide the source matrix in an equitative way. Being `rank` the rank of the worker thread (beginning at 0):

```
initial_pixel = rank * row_number
final_pixel = ((rank + 1) * row_number) - 1
```

This divided the problem space in equal divisions. The next problem to address was that the convolution process used more data than the one that would be output: the convolution relied on the surrounding pixels to calculate the value of each individual pixel, so we had to add extra data to allow this. The amount of data required varied between each kernel.

The calculations needed to find out this extra data were straightforward:

```
initial_needed_pixel = initial_pixel - (kernel_rows / 2)
final_needed_pixel = final_pixel + (kernel_rows / 2)
```

This gave us the needed buffer, but had 2 caveats: the variables could get out of the scope of the problem. This was solved with an additional check:

```
if(initial_needed_pixel < 0) initial_needed_pixel = 0
if(rank == thread_number) final_needed_pixel = final_pixel
```

The checks put the out-of-bounds indexes back at the start or end of the problem space.

Once the previous calculations were done, the master thread sent the required data to each worker. To do this three messages were used: one to send them the problem space information and the previous calculations, one to send them the kernel, and one to send them their chunk of problem space.

The workers received this data, performed the convolution process, and returned the result to the master thread, who collected this result and outputted it.

3.2 *Second approach: dynamic task mapping*

The second approach mapped the tasks dynamically: the problem space was divided in `N` partitions, and each worker worked on one. As the number of workers did not have to match the number of partitions, once a worker finished with its partition a new one was sent to it. This was repeated until there were no more partitions.

The data was divided in the same way as in the previous example, but instead of relying on the rank of each worker, the job number -a sequential counter of the amount of partitions computed until then- was used. Thus, the calculations were:

```
initial_pixel = last_job * row_number
final_pixel = ((last_job + 1) * row_number) - 1
```

The rest of calculations were the same.

To send the data different messages were used: the kernel and general problem information only had to be sent one time because it did not change, so it was broadcast at the start of the process.

The data regarding each problem partition had to be sent each time the worker thread needed a new task to do, so it was sent in the same way as in the first approach.

To signal the end of the execution, once each worker finished with their calculations and sent the result to the master, they waited for a flag sent from the master. The master thread, before sending anything to the workers, sent the "continue" flag. When all processing was done, it sent the "stop" flag, making them end the receiving-processing loop.

The rest of the work was as described on approach 1.

4 RESULTS

4.1 *Approach 1*

As we can see in Tables 1, 2 and 3, the first approach was good. There was a good speedup, specially when working on larger problems, and in general the efficiency was good as well. The performance improved with more threads, up until 12, giving this approach a good scalability. The overall average efficiency was between 50% and 70%, confirming the scalability of this approach.

Image	Kernel	Execution Time	Speedup	Efficiency
im01	3x3_Edge	0.012968	1.04	0.26
im01	5x5_Sharpen	0.012197	2.50	0.62
im01	25x25_random	0.232739	2.73	0.68
im01	49x49_random	0.854111	2.94	0.73
im01	99x99_random	3.051154	3.14	0.78
im02	3x3_Edge	0.025814	1.36	0.34
im02	5x5_Sharpen	0.03059	2.58	0.64
im02	25x25_random	0.555916	3.032	0.75
im02	49x49_random	2.201425	2.97	0.74
im02	99x99_random	8.195096	3.15	0.78
im03	3x3_Edge	0.092329	0.83	0.20
im03	5x5_Sharpen	0.069117	2.28	0.57
im03	25x25_random	1.137446	2.98	0.74
im03	49x49_random	4.479072	2.94	0.73
im03	99x99_random	17.076639	3.11	0.77
im04	3x3_Edge	5.19732	0.65	0.16
im04	5x5_Sharpen	7.093388	1.10	0.27
im04	25x25_random	57.472689	2.98	0.74
im04	49x49_random	234.157448	2.93	0.73
im04	99x99_random	873.587621	3.21	0.80

Table 1: Approach 1, 4 cores Results

Image	Kernel	Execution Time	Speedup	Efficiency
im01	3x3_Edge	0.221204	0.06	0.007
im01	5x5_Sharpen	0.266803	0.11	0.01
im01	25x25_random	0.11263	5.65	0.70
im01	49x49_random	0.68542	3.67	0.45
im01	99x99_random	1.00156	9.56	1.19
im02	3x3_Edge	0.04134	0.84	0.10
im02	5x5_Sharpen	0.062541	1.26	0.15
im02	25x25_random	0.275622	6.11	0.76
im02	49x49_random	0.952075	6.87	0.85
im02	99x99_random	3.109084	8.31	1.03
im03	3x3_Edge	0.115362	0.66	0.08
im03	5x5_Sharpen	0.085746	1.84	0.23
im03	25x25_random	0.562908	6.02	0.75
im03	49x49_random	1.912208	6.89	0.86
im03	99x99_random	6.738495	7.90	0.98
im04	3x3_Edge	5.178435	0.65	0.08
im04	5x5_Sharpen	3.551915	2.21	0.27
im04	25x25_random	26.688912	6.42	0.80
im04	49x49_random	102.750342	6.69	0.83
im04	99x99_random	685.254862	4.10	0.51

Table 2: Approach 1, 8 cores Results

Image	Kernel	Execution Time	Speedup	Efficiency
im01	3x3_Edge	0.22176	0.02	0.001
im01	5x5_Sharpen	0.221707	0.13	0.011
im01	25x25_random	0.082309	7.73	0.64
im01	49x49_random	0.226532	11.11	0.92
im01	99x99_random	0.670621	14.28	1.19
im02	3x3_Edge	0.241268	0.14	0.012
im02	5x5_Sharpen	0.244429	0.32	0.026
im02	25x25_random	0.415895	4.05	0.33
im02	49x49_random	0.574135	11.40	0.95
im02	99x99_random	1.775303	14.55	1.21
im03	3x3_Edge	0.125285	0.61	0.05
im03	5x5_Sharpen	0.100182	1.57	0.13
im03	25x25_random	0.382112	8.87	0.73
im03	49x49_random	1.21435	10.86	0.90
im03	99x99_random	3.96039	13.44	1.12
im04	3x3_Edge	4.406454	0.76	0.06
im04	5x5_Sharpen	4.389595	1.79	0.14
im04	25x25_random	18.078297	9.48	0.79
im04	49x49_random	65.668183	10.47	0.87
im04	99x99_random	425.92365	6.59	0.54

Table 3: Approach 1, 12 cores Results

4.2 *Approach 2*

The second approach gave generally worse results than the first one, shining mostly when the problems began to grew to a large input matrix and kernel. When the problems were small, as we can see in Tables 4, 5 and 6, the performance was worse than that of the sequential version.

This approach found a good speedup as more cores were added, but the average efficiency decreased inversely to the speedup, making this approach not very scalable. The added overhead of the extra message passing and calculations made it shine at large problems, where each worker could work for more time and offset said overhead, but on small problems, as it has been said, it's not a good approach.

Image	Kernel	Execution Time	Speedup	Efficiency
im01	3x3_Edge	0.117009	0.11	0.02
im01	5x5_Sharpen	0.18066	0.16	0.04
im01	25x25_random	0.407152	1.56	0.39
im01	49x49_random	1.01586	2.47	0.61
im01	99x99_random	2.854895	3.35	0.83
im02	3x3_Edge	0.153198	0.22	0.05
im02	5x5_Sharpen	0.163225	0.48	0.12
im02	25x25_random	1.01563	1.65	0.41
im02	49x49_random	3.87523	1.68	0.42
im02	99x99_random	6.98148	3.70	0.92
im03	3x3_Edge	0.255013	0.30	0.075
im03	5x5_Sharpen	0.284521	0.55	0.13
im03	25x25_random	1.87532	1.80	0.45
im03	49x49_random	6.58752	2.002	0.50
im03	99x99_random	20.23584	2.63	0.65
im04	3x3_Edge	2.75666	1.22	0.30
im04	5x5_Sharpen	3.818362	2.05	0.51
im04	25x25_random	69.918277	2.45	0.61
im04	49x49_random	217.26584	3.16	0.79
im04	99x99_random	658.39556	4.26	1.06

Table 4: Approach 2, 4 cores Results

Image	Kernel	Execution Time	Speedup	Efficiency
im01	3x3_Edge	0.36906	0.03	0.004
im01	5x5_Sharpen	0.356907	0.08	0.01
im01	25x25_random	0.724838	0.87	0.10
im01	49x49_random	2.50698	1.004	0.12
im01	99x99_random	2.598742	3.68	0.46
im02	3x3_Edge	0.310422	0.11	0.01
im02	5x5_Sharpen	0.426463	0.18	0.02
im02	25x25_random	1.25846	1.33	0.16
im02	49x49_random	3.123548	2.09	0.26
im02	99x99_random	4.598752	5.61	0.70
im03	3x3_Edge	0.424332	0.18	0.022
im03	5x5_Sharpen	0.439674	0.35	0.044
im03	25x25_random	3.58953	0.94	0.11
im03	49x49_random	4.65482	2.83	0.35
im03	99x99_random	8.25684	6.45	0.80
im04	3x3_Edge	3.446107	0.98	0.12
im04	5x5_Sharpen	3.411108	2.30	0.28
im04	25x25_random	35.865808	4.78	0.59
im04	49x49_random	131.376722	5.23	0.65
im04	99x99_random	539.846683	5.20	0.65

Table 5: Approach 2, 8 cores Results

5 CONCLUSIONS

The parallelization process was successful: in general, we got a speedup of 6 with 12 cores, with a good efficiency (see Figure 3).

Image	Kernel	Execution Time	Speedup	Efficiency
im01	3x3_Edge	0.639015	0.06	0.005
im01	5x5_Sharpen	0.708734	0.043	0.003
im01	25x25_random	0.644235	0.98	0.08
im01	49x49_random	0.774566	3.25	0.27
im01	99x99_random	0.983256	9.745	0.81
im02	3x3_Edge	0.925649	0.037	0.003
im02	5x5_Sharpen	0.935554	0.084	0.007
im02	25x25_random	1.158965	1.45	0.12
im02	49x49_random	1.189532	5.50	0.45
im02	99x99_random	2.486582	10.39	0.86
im03	3x3_Edge	0.695581	0.11	0.009
im03	5x5_Sharpen	1.064752	0.14	0.01
im03	25x25_random	3.014789	1.12	0.09
im03	49x49_random	1.963258	6.71	0.55
im03	99x99_random	5.865125	9.08	0.75
im04	3x3_Edge	4.340012	0.77	0.06
im04	5x5_Sharpen	4.483373	1.75	0.14
im04	25x25_random	20.033829	8.56	0.71
im04	49x49_random	68.376147	10.05	0.83
im04	99x99_random	273.40289	10.28	0.85

Table 6: Approach 2, 12 cores Results

Of the two methods tested, the first was the best performing in general, outclassing the other but getting closer as the size of the problem grew (Figures 1 and 2).

Once the size was getting very big (at im04), the first approach showed a decrease in efficiency and speedup but the second one grew on both metrics. This is a sign that the first approach is best used on relatively small problems (althoug it does not suffer a lot on larger ones) and the second should only be used when the scope of the program is very big, making this approach less flexible.

This difference is likely due to the overhead added by the extra synchronisation needed by the second approach being larger than the performance gained by the parallelization until the problem was big enough to offset this overhead.

We could not test the different methods in really big problems due to unexpected circumstances and restrictions on our testing environment, but the trend is likely to continue, with bigger returns the bigger the problem is. The demonstration of this theory remains to be performed in future work.

6 FUTURE WORK

We have seen that the approaches chosen work well to parallelize the convolution algorithm. Although it performs well, more tests could be performed to try to get an extra performance increase, especially with larger problems, and get greater efficiency. There are different approaches to obtain this extra performance, and they can be used stand-alone or combined

- Parallelize further the algorithm at the level of each worker: OpenMP can be used for this approach, requiring less message transport overhead due to the shared memory approach that uses OpenMP.
- Further parallelization of the kernel matrix loops. This could help on large problems, but it would likely be overkill on smaller (the overhead is already large and margins are too low). It should be thoroughly tested.

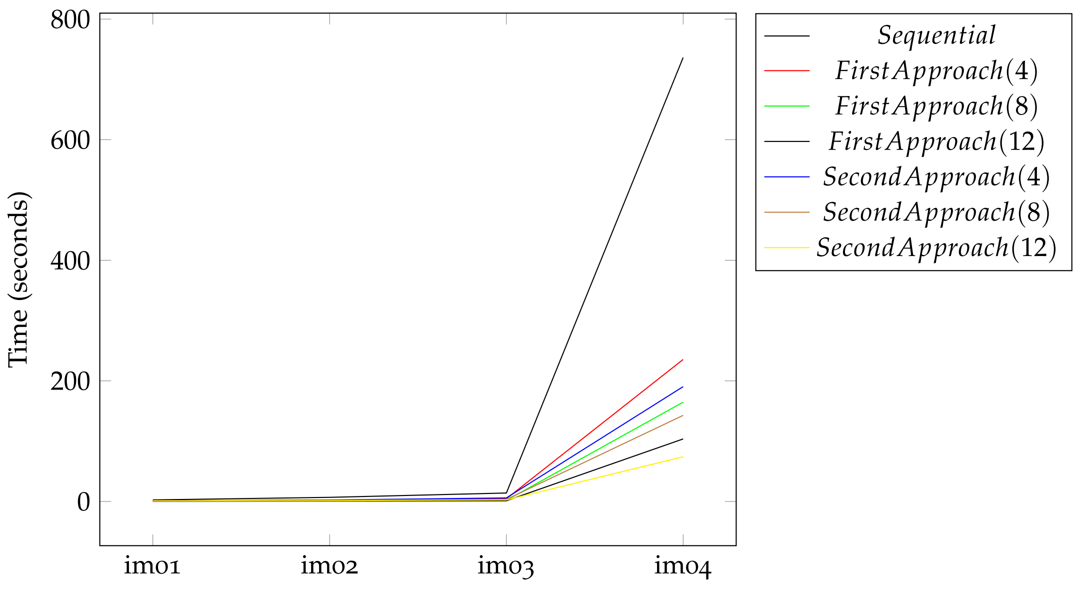

Figure 1: Average Execution Time

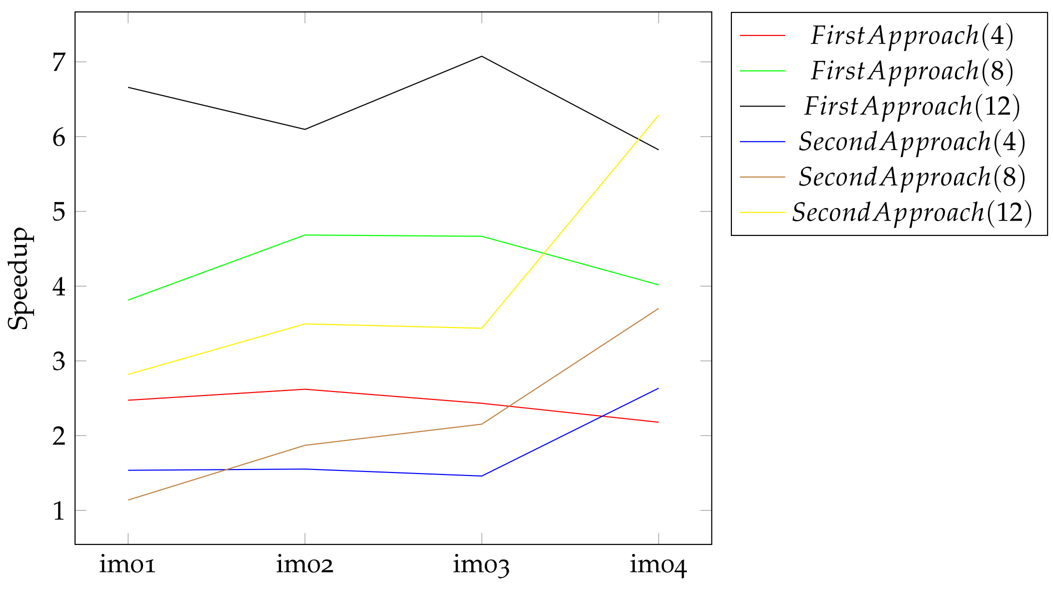

Figure 2: Average Speedup

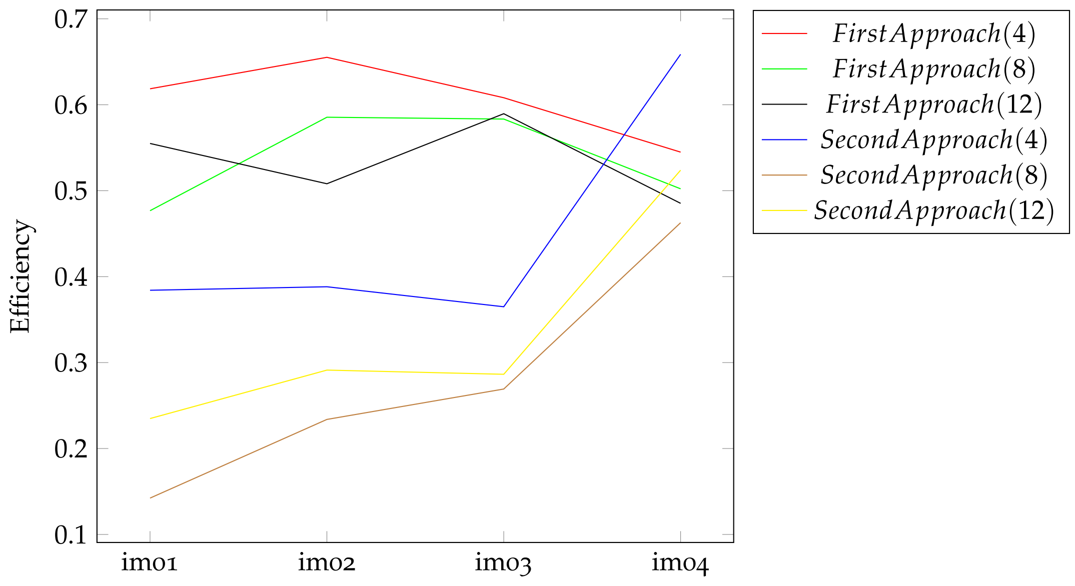

Figure 3: Average Efficiency

- Tailoring to the target machine. A greater customization to take into account all the details of the target machine could be explored. This includes
  - Checking if parallel I/O would be suitable
  - Tweaking the algorithm to tailor it to the different ways the target machine's cache works
  - Optimising the algorithm to take into account the architecture of the target machine
- Optimisation for the used compiler. The currently used compiler, GCC, may translate some C and OpenMP expressions to less than optimal bytecode. If these pitfalls could be detected and addressed, an increase in performance would be assured, at the expense of portability.