HIGH PERFORMANCE COMPUTING
HYBRID PRACTICE

MARC PIÑOL

CONTENTS

1 INTRODUCTION

Our aim in this project was to speed-up the provided image convolution algorithm. To do this we used MPI to divide the data and send each division to a different node, and OpenMP to further parallelize the algorithm without changing its core specification and definition.

The MPI parallellization was the work of another team[1]. We used their method because ours was not working as well as theirs. On top of this, the chosen OpenMP approach was the -theoretically- most efficient one of the ones we tested before.

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

[1] The code can be found at `https://github.com/bergacat1/HCP/blob/master/convolutionMPI.c`

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

3 APPROACH

The approach to uniting MPI and OpenMP was based in a fusion of MPI to partition the data to work on and OpenMP to further parallelize the process.

As stated, the MPI part was originally programmed by a different team and we used their version due to it enjoying of a greater correctness. Our version had a problem where it shifted the original images by some pixels, thus making the convolution process incorrectly.

The OpenMP section of the program parallelized the outer loop of the convolution function (the loop which iterates over all the rows of the image). It was chosen this way because it was pointed to us on the first attempt at parallelizing this algorithm through OpenMP that this method was the most efficient of the three approaches we tried. The other two approaches involved parallelizing the convolution method calls and parallelizing the inner loop of the convolution function (which iterates over each pixel in a row) as well as the outer loop.

Overall, as an example of partitioning, an execution with N MPI partitions and 4 OpenMP cores would process its data in the following way:

1. The MPI master process would partition the data in N parts
2. The MPI master process would send each partition to its respective recipient
3. Each MPI slave would begin the convolution process, calling the OpenMP directives and parallelizing their outer loops
4. Each OpenMP parallel section would perform their respective calculations concurrently
5. Each MPI slave would send the results of their calculations to the master process
6. The MPI master process would receive each part and join them, sending the result to the output destination

This would, in theory, help to speed up the convolution process further.

4 RESULTS

To check how better is the approach of mixing MPI and OpenMP we tested our program with a different number of MPI nodes, each using 2 or 4 inner threads for the parallel convolution. Thus, we had 8 sets of results, enabling us to see the performance difference when using extra processing power. The total tests were the following:

- 2 nodes
  - 2 inner threads
  - 4 inner threads
- 4 nodes
  - 2 inner threads
  - 4 inner threads
- 6 nodes

  - 2 inner threads
  - 4 inner threads
- 8 nodes
  - 2 inner threads
  - 4 inner threads

As we can see in tables 1 to 8, the solution did not perform too well on smaller datasets or using few nodes. Its execution times were generally on par with the serial version, being slower on occasions. This meant that the solution had a poor speedup and efficiency on these small datasets. On the other hand, the solution performed better as the datasets and allocated nodes grew, reducing its execution time from the serial version to the parallel ones more as the problem to tackle grew. This is most likely due to the cost of synchronization and communication being too large for small problems: the communication cost offsets the parallelization gains.

Another thing to note is the generally low efficiency of the solution. Although adding nodes and cores to the execution environment helped gain a larger speedup the increase is not enough to maintain a good efficiency. When looking at the difference from 2 to 4 OpenMP cores this drawback is even more pronounced, nearly halving the efficiency of the solution for a marginal decrease in execution time.

Image	Kernel	Execution Time	Speedup	Efficiency
im01	3x3_Edge	0.215978	0.06	0.01
im01	5x5_Sharpen	0.262393	0.11	0.02
im01	25x25_random	0.974221	0.65	0.16
im01	49x49_random	2.779717	0.90	0.22
im01	99x99_random	9.409508	1.01	0.25
im02	3x3_Edge	0.266255	0.13	0.03
im02	5x5_Sharpen	0.381077	0.20	0.05
im02	25x25_random	1.978436	0.85	0.21
im02	49x49_random	6.897839	0.94	0.23
im02	99x99_random	25.280674	1.02	0.25
im03	3x3_Edge	0.485813	0.15	0.03
im03	5x5_Sharpen	0.440509	0.35	0.08
im03	25x25_random	3.774047	0.89	0.22
im03	49x49_random	13.382025	0.98	0.24
im03	99x99_random	51.883798	1.02	0.253
im04	3x3_Edge	6.811747	0.49	0.12
im04	5x5_Sharpen	11.897219	0.66	0.16
im04	25x25_random	177.946297	0.96	0.24
im04	49x49_random	684.835137	1.00	0.25
im04	99x99_random	2745.784052	1.02	0.25

Table 1: 2 Nodes, 2 Concurrent Loops Results

5 CONCLUSIONS

The parallelization process was successful: in general, we got a good speedup when using multiple nodes (see Figures 1 and 2).

On the other hand, as it has been said, this speedup may be not enough to justify the added cost of building and maintaining a cluster to run this program. As shown in tables 1 to 8 and Figure 3, the efficiency of this solution was low. It increased as the problem grew in size, though, so it may

Image	Kernel	Execution Time	Speedup	Efficiency
im01	3x3_Edge	0.331685	0.04	0.005
im01	5x5_Sharpen	0.294039	0.10	0.01
im01	25x25_random	0.898342	0.70	0.08
im01	49x49_random	2.787554	0.90	0.11
im01	99x99_random	9.353859	1.02	0.12
im02	3x3_Edge	0.235632	0.14	0.01
im02	5x5_Sharpen	0.372732	0.21	0.02
im02	25x25_random	1.94939	0.86	0.10
im02	49x49_random	6.584951	0.99	0.12
im02	99x99_random	25.225409	1.02	0.12
im03	3x3_Edge	0.38735	0.19	0.02
im03	5x5_Sharpen	0.382593	0.41	0.05
im03	25x25_random	3.834453	0.88	0.11
im03	49x49_random	13.488709	0.97	0.12
im03	99x99_random	51.825783	1.02	0.12
im04	3x3_Edge	6.692991	0.50	0.06
im04	5x5_Sharpen	11.569298	0.67	0.08
im04	25x25_random	177.672473	0.96	0.12
im04	49x49_random	683.845412	1.00	0.12
im04	99x99_random	2745.951213	1.02	0.12

Table 2: 2 Nodes, 4 Concurrent Loops Results

Image	Kernel	Execution Time	Speedup	Efficiency
im01	3x3_Edge	0.300273	0.04	0.005
im01	5x5_Sharpen	0.369031	0.08	0.01
im01	25x25_random	0.663908	0.95	0.11
im01	49x49_random	1.550763	1.62	0.20
im01	99x99_random	4.449808	2.15	0.26
im02	3x3_Edge	0.354271	0.09	0.01
im02	5x5_Sharpen	0.435234	0.18	0.02
im02	25x25_random	1.209939	1.39	0.17
im02	49x49_random	3.3867	1.93	0.24
im02	99x99_random	11.954807	2.16	0.27
im03	3x3_Edge	0.596967	0.12	0.01
im03	5x5_Sharpen	0.521758	0.30	0.03
im03	25x25_random	2.102116	1.61	0.20
im03	49x49_random	6.72129	1.96	0.24
im03	99x99_random	24.895074	2.13	0.26
im04	3x3_Edge	10.067833	0.33	0.041
im04	5x5_Sharpen	5.644899	1.39	0.17
im04	25x25_random	88.867451	1.92	0.24
im04	49x49_random	354.915332	1.93	0.24
im04	99x99_random	1365.750257	2.05	0.25

Table 3: 4 Nodes, 2 Concurrent Loops Results

be a good solution for much larger problems, where the speedup gained really offsets the parallelization costs. On the current state the OpenMP part does not provide a great increase between 2 and 4 dedicated threads. It could be tested on larger increases to check how worthwile is a larger amount of OpenMP threading.

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.

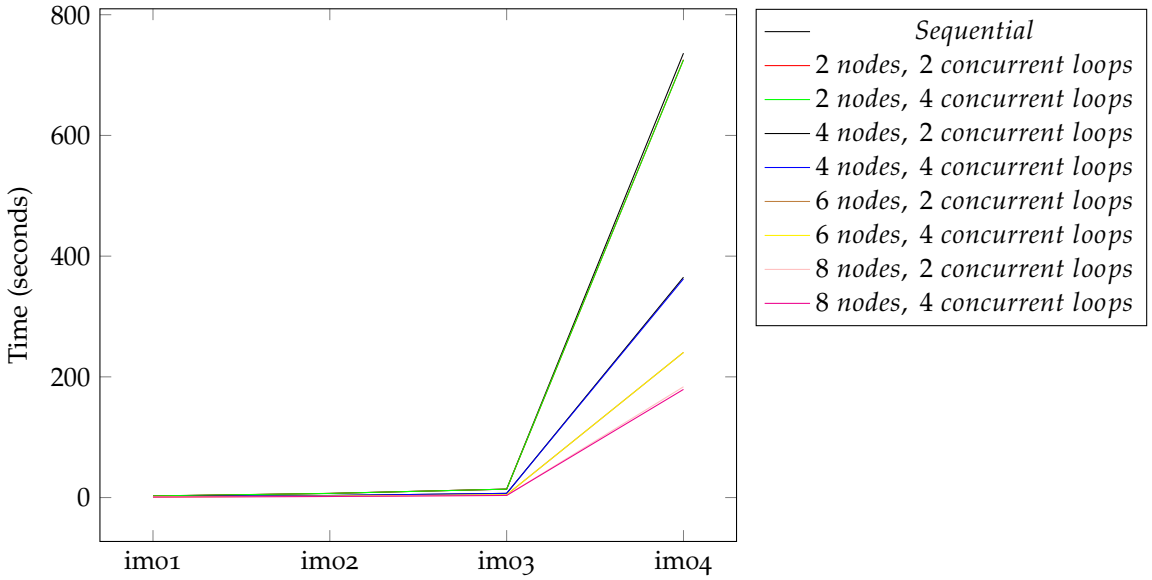

Figure 1: Average Execution Time

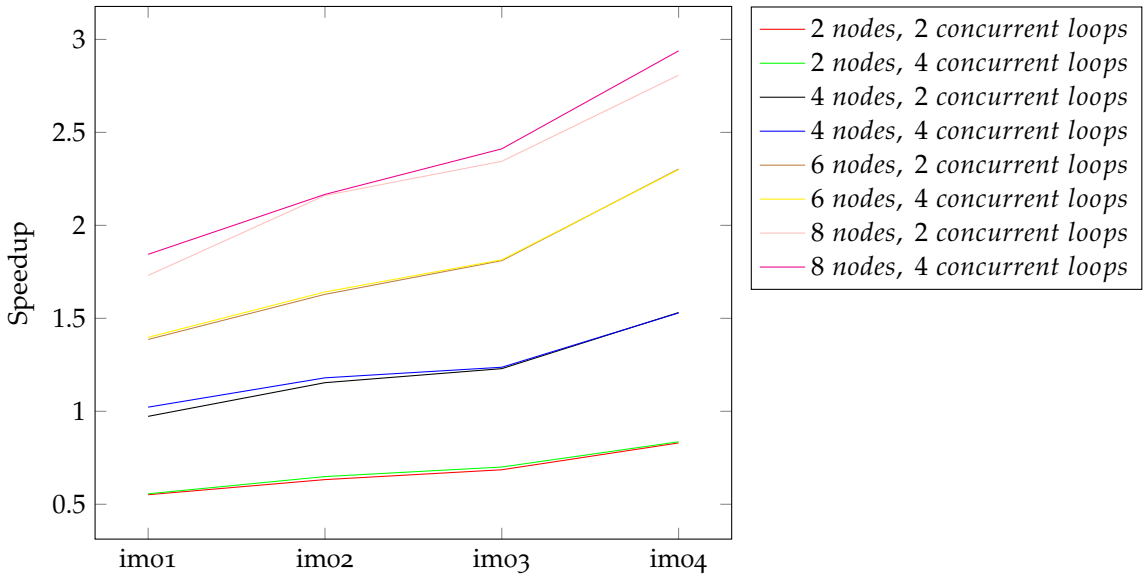

Figure 2: Average Speedup

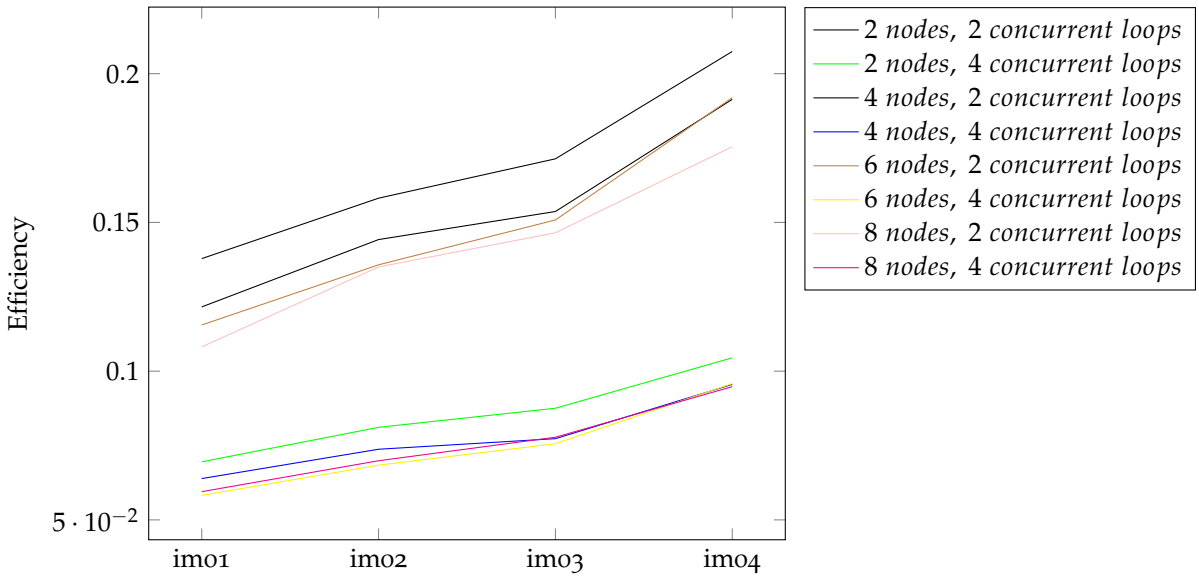

Figure 3: Average Efficiency

Image	Kernel	Execution Time	Speedup	Efficiency
im01	3x3_Edge	0.323265	0.04	0.002
im01	5x5_Sharpen	0.364402	0.08	0.005
im01	25x25_random	0.590528	1.07	0.06
im01	49x49_random	1.437216	1.75	0.10
im01	99x99_random	4.445669	2.15	0.13
im02	3x3_Edge	0.354148	0.09	0.006
im02	5x5_Sharpen	0.39442	0.20	0.012
im02	25x25_random	1.112092	1.51	0.09
im02	49x49_random	3.410877	1.91	0.11
im02	99x99_random	11.940191	2.16	0.13
im03	3x3_Edge	0.645836	0.11	0.007
im03	5x5_Sharpen	0.45487	0.34	0.02
im03	25x25_random	2.085934	1.62	0.10
im03	49x49_random	6.744715	1.95	0.12
im03	99x99_random	24.928732	2.13	0.13
im04	3x3_Edge	10.614925	0.31	0.01
im04	5x5_Sharpen	5.962193	1.31	0.08
im04	25x25_random	89.013436	1.92	0.12
im04	49x49_random	340.021656	2.02	0.12
im04	99x99_random	1366.259815	2.05	0.12

Table 4: 4 Nodes, 4 Concurrent Loops Results

Image	Kernel	Execution Time	Speedup	Efficiency
im01	3x3_Edge	0.283246	0.04	0.003
im01	5x5_Sharpen	0.278952	0.10	0.009
im01	25x25_random	0.551423	1.15	0.09
im01	49x49_random	1.123104	2.24	0.18
im01	99x99_random	2.837213	3.37	0.28
im02	3x3_Edge	0.492319	0.07	0.005
im02	5x5_Sharpen	0.404067	0.19	0.01
im02	25x25_random	0.925271	1.82	0.15
im02	49x49_random	2.445053	2.67	0.22
im02	99x99_random	7.65144	3.37	0.28
im03	3x3_Edge	0.417997	0.18	0.015
im03	5x5_Sharpen	0.46165	0.34	0.02
im03	25x25_random	1.48005	2.29	0.19
im03	49x49_random	4.546817	2.90	0.24
im03	99x99_random	15.986825	3.33	0.27
im04	3x3_Edge	3.960895	0.85	0.07
im04	5x5_Sharpen	4.749939	1.65	0.13
im04	25x25_random	59.458407	2.88	0.24
im04	49x49_random	227.672005	3.02	0.25
im04	99x99_random	906.618704	3.10	0.25

Table 5: 6 Nodes, 2 Concurrent Loops Results

6 FUTURE WORK

We have seen that the approaches chosen can be used to parallelize the convolution algorithm. Although it resulted in a performance gain, more tests should be performed to try to get greater efficiency. There are different approaches to obtain this extra performance, and they can be used stand-alone or combined

- Try other approaches to the OpenMP parallelization: section worksharing, stand-alone or coupled with parallel loops.
- Further parallelization of the kernel matrix loops. This could help on large problems, but it would likely be overkill on smaller (the overhead

Image	Kernel	Execution Time	Speedup	Efficiency
im01	3x3_Edge	0.284483	0.04	0.001
im01	5x5_Sharpen	0.277543	0.11	0.004
im01	25x25_random	0.477546	1.33	0.05
im01	49x49_random	1.129331	2.22	0.09
im01	99x99_random	2.931662	3.26	0.13
im02	3x3_Edge	0.366102	0.09	0.003
im02	5x5_Sharpen	0.392247	0.20	0.008
im02	25x25_random	0.891987	1.89	0.07
im02	49x49_random	2.427077	2.69	0.11
im02	99x99_random	7.774971	3.32	0.13
im03	3x3_Edge	0.652955	0.11	0.004
im03	5x5_Sharpen	0.478662	0.33	0.01
im03	25x25_random	1.480613	2.29	0.09
im03	49x49_random	4.395223	3.00	0.12
im03	99x99_random	15.97543	3.33	0.13
im04	3x3_Edge	4.13091	0.81	0.03
im04	5x5_Sharpen	4.743891	1.65	0.06
im04	25x25_random	59.327956	2.89	0.12
im04	49x49_random	226.735332	3.03	0.12
im04	99x99_random	907.150953	3.09	0.12

Table 6: 6 Nodes, 4 Concurrent Loops Results

Image	Kernel	Execution Time	Speedup	Efficiency
im01	3x3_Edge	0.347078	0.03	0.002
im01	5x5_Sharpen	0.362245	0.08	0.005
im01	25x25_random	0.503784	1.26	0.07
im01	49x49_random	0.87242	2.88	0.18
im01	99x99_random	2.188184	4.37	0.27
im02	3x3_Edge	0.413759	0.08	0.005
im02	5x5_Sharpen	0.309575	0.25	0.01
im02	25x25_random	0.865933	1.94	0.12
im02	49x49_random	1.724395	3.79	0.23
im02	99x99_random	5.476758	4.71	0.29
im03	3x3_Edge	0.982722	0.078	0.004
im03	5x5_Sharpen	0.452459	0.34	0.02
im03	25x25_random	1.19413	2.83	0.17
im03	49x49_random	3.423118	3.85	0.24
im03	99x99_random	11.576815	4.60	0.28
im04	3x3_Edge	25.162577	0.13	0.008
im04	5x5_Sharpen	4.298747	1.82	0.11
im04	25x25_random	44.49935	3.85	0.24
im04	49x49_random	169.888885	4.04	0.25
im04	99x99_random	673.931908	4.17	0.26

Table 7: 8 Nodes, 2 Concurrent Loops Results

is already large and margins are too low). It should be thoroughly tested.

- 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

Image	Kernel	Execution Time	Speedup	Efficiency
im01	3x3_Edge	0.377855	0.03	0.001
im01	5x5_Sharpen	0.355781	0.08	0.002
im01	25x25_random	0.425062	1.49	0.04
im01	49x49_random	0.813743	3.09	0.09
im01	99x99_random	2.126293	4.50	0.14
im02	3x3_Edge	0.435558	0.08	0.002
im02	5x5_Sharpen	0.420975	0.18	0.006
im02	25x25_random	0.788116	2.13	0.06
im02	49x49_random	1.806629	3.62	0.11
im02	99x99_random	5.384147	4.79	0.15
im03	3x3_Edge	1.060628	0.07	0.002
im03	5x5_Sharpen	0.418755	0.37	0.01
im03	25x25_random	1.112083	3.04	0.09
im03	49x49_random	3.332668	3.95	0.12
im03	99x99_random	11.578564	4.59	0.14
im04	3x3_Edge	3.769772	0.89	0.02
im04	5x5_Sharpen	4.603513	1.70	0.05
im04	25x25_random	44.60464	3.84	0.12
im04	49x49_random	168.85835	4.07	0.13
im04	99x99_random	674.240014	4.16	0.13

Table 8: 8 Nodes, 4 Concurrent Loops Results

- Optimisation for the used compiler. The currently used compiler, GCC, may translate some C, MPI 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.