HIGH PERFORMANCE COMPUTING
OPENMP 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 OpenMP to parallelize the algorithm without changing its core specification and definition.

This parallellization was done in an incremental way, parallellizing a part of the algorithm and testing for its performance increase and correctness before moving to the next section.

In the end, given the time frame and scope of the project, we ended up with 3 iterations of the parallellizing-testing approach:

1. The first one was a parallelization of the outer loop of the `convolve2D` function. This loop iterated through each row of the input matrix. This was the most obvious approach and provided some speed up.

2. The next approach was parallelizing the 3 calls to the previous function, given that each one worked with independent data (each RGB channel), and was an easy and fast way to improve the performance of the program.

3. In the last iteration we parallelized the inner loop, which iterated through each column of the input matrix. This approach, having done the first one, did not require a lot of work but improved the performance of the algorithm.

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: parallelizing the main `convolve2D` loop*

The first chosen approach was parallellizing the outer loop of the convolution function. This required identifying how to divide the input data and how to write to the appropriate output locations.

The loop was initially parallelized using 8 threads and the `parallel for` OpenMP directive. This divided the input matrix in 8 sets of rows (when available) and worked on all of them at the same time. The algorithm was essentially the same, but the access pointers had to be updated to adapt to the modifications.

A new pointer (`base`) was created, to always point to the initial location of the input pointers, and two additional calculations were added before the second loop:

```
outPtr = i * dataSizeX + out;
inPtr = inPtr2 = i * dataSizeX + base;
```

Where `out` and `base` where the initial positions of the output and input pointers, shared between all threads, `i` was the iteration counter and `dataSizeX` was the size of the rows of the image. Thus, these calculations set the pointers to the position where the data of the current row will be found.

These calculations found out the initial position of the current row with an independence of previous calculations that was not initially there: the previous algorithm was thought using a sequential increment of the input and output positions that relied on the previous iteration to know where to read and write. With these changes the algorithm was successfully parallelized.

3.2 *Second approach: parallelizing the RGB `convolve2D` calls*

The second approach was straightforward: using the `parallel sections` OpenMP directive the three method calls were successfully parallelized. There were not any shared data that had to be secured against concurrent access, so the addition of the directive was all that was required to improve the performance of the algorithm.

3.3 *Third approach: parallelizing the inner `convolve2D` loop*

The third approach consisted in parallelizing the next loop of the `convolve2D` function. The loop traversed the rows of the input matrix and performed the required computations, so it was similar to approach 1. Two more pointers, `baseOut` and `baseIn`, were created. These pointers would hold the position of the input and output pointers at the iteration of each row:

```
baseOut = i * dataSizeX + out;
baseIn = i * dataSizeX + base;
```

The calculations were the same as in approach 1. Then, to manage the data positions in each sub-thread, an extension of the arithmetic found in approach 1 was chosen:

```
outPtr = baseOut + j;
inPtr = inPtr2 = baseIn + j;
```

And the pointer increment lines were changed, from:

```
inPtr = ++inPtr2;
outPtr++;
```

to

```
inPtr = inPtr2;
// outPtr++;  (removed)
```

These calculations offloaded the pointer increment to the start of each loop, incrementing each pointer by the required amount to make it point to the correct position, and allowed us to parallelize the loop with no further problems. In this approach the previous number of threads was lowered to keep the global number of threads manageable: each loop was set to 4 threads. To enable nested loop concurrency we also had to set the corresponding OpenMP flag with `omp_set_nested(1);`

4 RESULTS

4.1 *Approach 1*

As we can see in Table 1, the first approach was not very good. There was some degree of speedup, specially when working on larger problems, but in general the performance was near the sequential version of the program, probably due to parallel overhead and dealing with problems too small to gain performance with this kind of parallelization.

4.2 *Approach 2*

The second approach gave better results, with a speedup generally greater than 1, although they were not a great improvement over the serial version.

Still, as we can see in Table 2, the overall performance was good and the good results of the parallelization were beginning to show.

4.3 *Approach 3*

The third approach gave us the best results. Times dropped considerably, as seen in Table 3, and speedup and efficiency improved over the other versions.

This version was still not really efficient. It gained performance, but the added cores required a larger increase to end up with a better efficiency.

Image	Kernel	Execution Time	Speedup	Efficiency
im01	3x3_Edge	0.006533	2.065	0.25
im01	5x5_Sharpen	0.03646	0.83	0.10
im01	25x25_random	0.1721038	3.69	0.46
im01	49x49_random	2.641003	0.95	0.11
im01	99x99_random	9.442556	1.01	0.12
im02	3x3_Edge	0.037999	0.92	0.11
im02	5x5_Sharpen	0.93287	0.08	0.01
im02	25x25_random	2.053	0.82	0.10
im02	49x49_random	6.761707	0.96	0.12
im02	99x99_random	25.783499	1.002	0.12
im03	3x3_Edge	0.082865	0.92	0.11
im03	5x5_Sharpen	0.183167	0.86	0.10
im03	25x25_random	3.976196	0.85	0.10
im03	49x49_random	13.443378	0.98	0.12
im03	99x99_random	52.757174	1.009	0.12
im04	3x3_Edge	1.30791	2.58	0.32
im04	5x5_Sharpen	9.365253	0.83	0.10
im04	25x25_random	123.700748	1.38	0.17
im04	49x49_random	722.613292	0.95	0.11
im04	99x99_random	2780.58432	1.01	0.12

Table 1: Approach 1 Results

Image	Kernel	Execution Time	Speedup	Efficiency
im01	3x3_Edge	0.008592	1.57	0.19
im01	5x5_Sharpen	0.127653	0.23	0.02
im01	25x25_random	0.253023	2.51	0.31
im01	49x49_random	1.415637	1.77	0.22
im01	99x99_random	8.312749	1.15	0.14
im02	3x3_Edge	0.016083	2.18	0.27
im02	5x5_Sharpen	0.249756	0.31	0.03
im02	25x25_random	1.032954	1.63	0.20
im02	49x49_random	7.136904	0.91	0.11
im02	99x99_random	8.821495	2.92	0.36
im03	3x3_Edge	0.030389	2.52	0.31
im03	5x5_Sharpen	0.290592	0.54	0.06
im03	25x25_random	2.007264	1.68	0.21
im03	49x49_random	4.829885	2.73	0.34
im03	99x99_random	47.16127	1.12	0.14
im04	3x3_Edge	1.233093	2.74	0.34
im04	5x5_Sharpen	6.28277	1.25	0.15
im04	25x25_random	177.757679	0.96	0.12
im04	49x49_random	337.306133	2.03	0.25
im04	99x99_random	2013.79167	1.3	1

Table 2: Approach 2 Results

5 CONCLUSIONS

The parallelization process was successful: in general, we got a speedup of 2, but it was not very efficient (see Figure 3).

Of the three methods tested, the third (parallelizing the source matrix and the computations for each RGB channel) was the best performing, outclassing the other versions more as the size of the problem grew (Figures 1 and 2). 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.

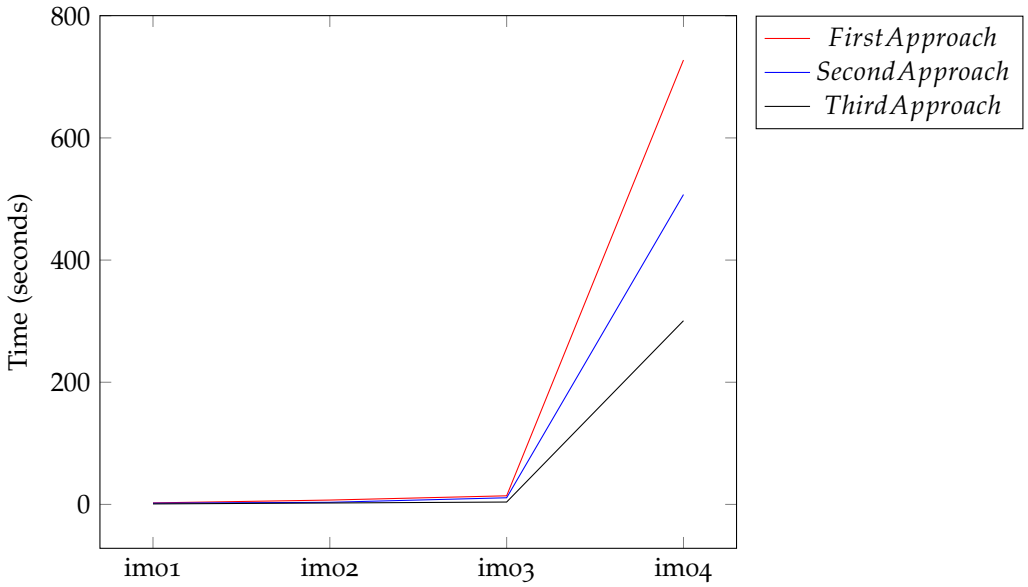

Figure 1: Average Execution Time

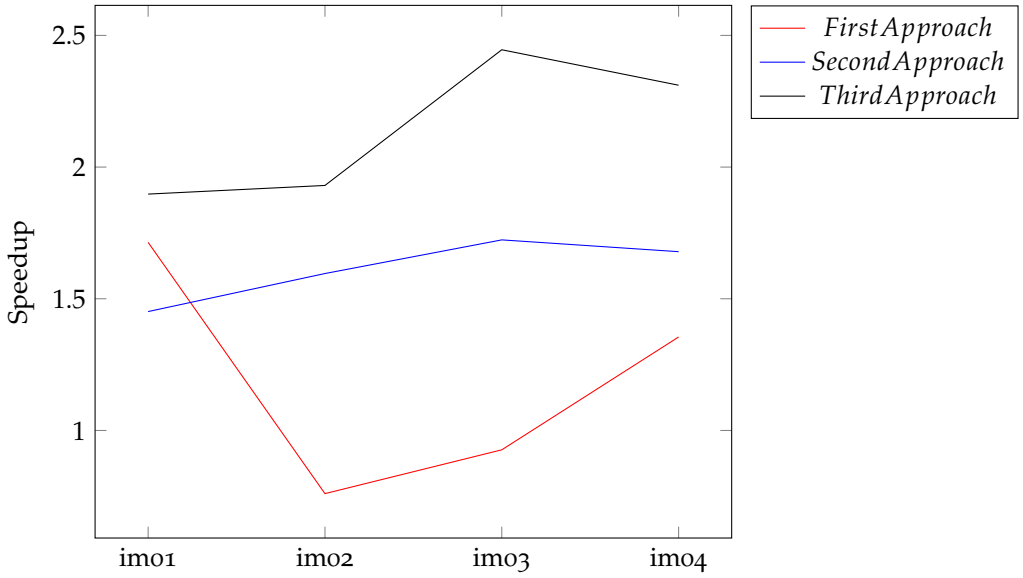

Figure 2: Average Speedup

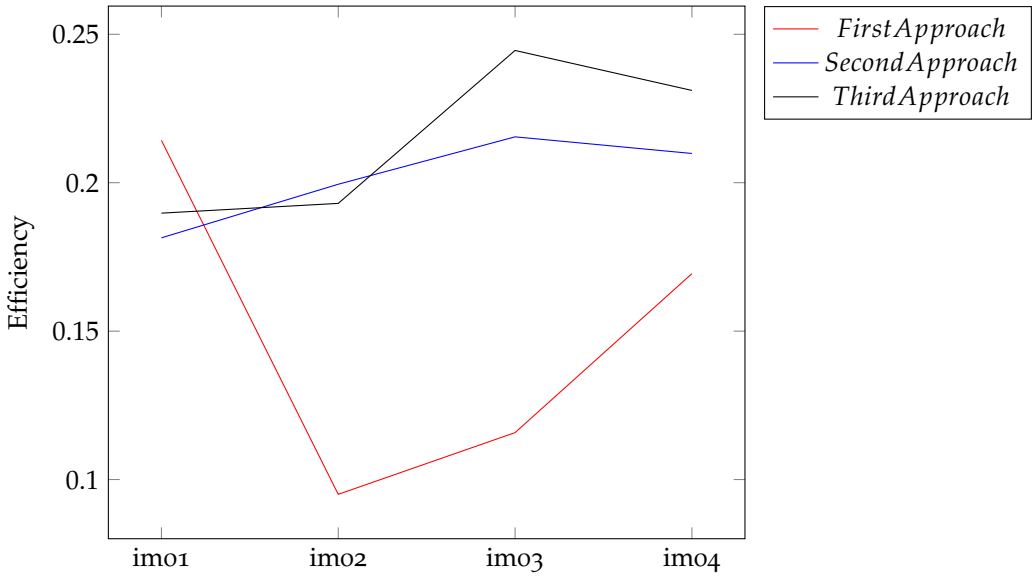

Figure 3: Average Efficiency

Image	Kernel	Execution Time	Speedup	Efficiency
im01	3x3_Edge	0.02055	0.002	0
im01	5x5_Sharpen	0.024666	1.23	0.12
im01	25x25_random	0.607795	1.04	0.10
im01	49x49_random	0.712449	3.53	0.35
im01	99x99_random	2.615008	3.66	0.36
im02	3x3_Edge	0.027994	1.25	0.12
im02	5x5_Sharpen	0.068694	1.15	0.11
im02	25x25_random	0.99596	1.69	0.16
im02	49x49_random	3.598474	1.81	0.18
im02	99x99_random	6.920743	3.73	0.37
im03	3x3_Edge	0.088827	0.86	0.08
im03	5x5_Sharpen	0.189426	0.83	0.08
im03	25x25_random	1.049154	3.23	0.32
im03	49x49_random	3.679489	3.58	0.35
im03	99x99_random	14.34835	3.71	0.37
im04	3x3_Edge	2.170963	1.55	0.15
im04	5x5_Sharpen	3.910536	2.01	0.20
im04	25x25_random	53.268599	3.21	0.32
im04	49x49_random	701.627165	0.98	0.09
im04	99x99_random	742.590118	3.78	0.37

Table 3: Approach 3 Results

6 FUTURE WORK

We have seen that the approach chosen is a good one 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 e greater efficiency. There are different approaches to obtain this extra performance, and they can be used stand-alone or combined

- Parallelization of the kernel matrix loops. This would help when facing large parallel matrices, a bottleneck of the current approach, and would increase the scalability of the algorithm.
- 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.