Parallel Computing Assignment 1
Heat Diffusion OpenMP and MPI

Hamzah Al Zubi s1047768
Nadezhda Dobreva s1033115
Seraph Jin s1032019
Tim Gerritsen s1041423

May 27, 2021

1 Introduction

After a lot of struggles, all of our main measurements were performed on the cluster that was provided. The exact specifications are given in the course material but we will repeat them here for the sake of completeness:

- 2x Intel Xeon Silver 4214, 12 cores each (1-3.2GHz[1]).
- 128GB RAM.
- 1TB NVME SSD.
- 8x NVIDIA RTX 2080 GPU, 12GB VRAM each (irrelevant for this assignment).

All of our code was also compiled on the cluster, so the following (extremely outdated) software was used:

- gcc 7.5.0.
- OpenMP 4.5.
- OpenMPI 2.1.1.

This report will have the following structure:

- In section 2, we will introduce some preliminary information about the methods we used throughout the assignment.
- In section 3, we will first evaluate the performance of the original code that was provided with the assignment, describe how we optimized it, and finally compare the optimized version to the original.
- In section 4, we will describe our OpenMP implementation and also compare it using both the optimized and unoptimized versions.
- In section 5, we will do the same for our MPI implementation.
- In section 6, we will discuss our testing in much more detail and dive into the data and represent it, comparing all different versions, and discuss how everything performs.
- Finally, in section 7 we will discuss how we worked as a team.

Our code, scripts, and relevant material for the report can be found in our git repository on Github and also in our submission.

[1] Unfortunately, we have no control over this frequency. So the fact that it can vary on its own is the first of many reasons why we think this cluster is not appropriate for this assignment.

2 Preliminaries

2.1 Wall Clock Time Measurements

All of our time measurements were made using the code that was provided for this purpose on Brightspace. We implemented those functions in a separate file that we include in each of our programs. We always start the measurement before the main loop of the algorithm, and end it immediately after. So we disregard the time of initialization in our experiments.

2.2 GFLOPS Calculations

We used the following formula for calculating the number of floating point operations per second for a run of the program with `iter` iterations and a matrix of size `n` that took `t` seconds:

$$FLOPS = \frac{9 \cdot iter \cdot (n-2)^2}{t}$$

This is because in every cell of the matrix we perform 9 floating point operations: five multiplications and four additions. And we have $(n-2)^2$ cells that we update, because we update all cells in a square matrix of size n except for the border cells. And finally all of this is repeated `iter` times.

To be as clear as possible, we always mean "billion floating point operations per second" when we say "GFLOPS".

2.3 Measurement Aggregation

For all of our experiments, we took 5 measurements. We took the median of those measurements in all cases. We did this because the average would not be representative of the values due to outliers caused by the extreme inconsistencies in runtime on the cluster which severely limited our options of aggregation methods. We found this method to be sufficient while still allowing for the process to be easily automated, which was needed due to the amount of experiments we ran.

3 Sequential Version

3.1 Evaluation of Original Code

To get started, we need to establish a baseline. So we ran the given code with more or less no alterations (other than cleaning it up and inserting our measurement code). We compiled it using gcc with no flags. Table 1 shows the wall clock time and GFLOPS of this version as a function of matrix size. Note that for all of those measurements, the number of iterations was fixed at 5.

Matrix Dimensions	256 * 256	512 * 512	1024 * 1024	2048 * 2048	4096 * 4096	8192 * 8192	16384 * 16384	32768 * 32768
Size in MB	0.5	2	8	32	128	512	2048	8192
Wall Clock Time (ms)	6	16	55	156	615	2547	10881	45804
GFLOPS	0.48	0.73	0.85	1.21	1.23	1.19	1.11	1.05

Table 1: The wall clock time and GFLOPS as a function of matrix size for the original sequential version.

If we look at the GFLOPS in the context of the frequency of the processor we used, we can see that we are probably doing less than one floating point operation per clock cycle, which does not sound great. So there is definitely room for improvement!

3.2 Optimizing the Code

We tried to optimize the sequential version by vectorizing it. We took two steps to do this:

- Allocating the matrix as a two dimensional array: meaning that the matrix now becomes an array of arrays, where each inner array corresponds to a row.
- Adding optimization flags to our compilation: namely `-O3` and `-march=native`.

We found that the second step alone already vectorizes the code. But the first step was necessary to make that vectorization play nicely with the OpenMP and MPI versions. So we implemented both steps. Table 2 shows the wall clock time and GFLOPS of this version as a function of matrix size. Note that for all of those measurements, the number of iterations was fixed at 5.

Matrix Dimensions	256 * 256	512 * 512	1024 * 1024	2048 * 2048	4096 * 4096	8192 * 8192	16384 * 16384	32768 * 32768
Size in MB	0.5	2	8	32	128	512	2048	8192
Wall Clock Time (ms)	0	1	6	33	134	540	2273	10580
GFLOPS	2.90	11.70	7.83	5.71	5.63	5.59	5.31	4.57

Table 2: The wall clock time and GFLOPS as a function of matrix size for the vectorized sequential version.

This is already looking much better! The data for small matrices is a bit messy because the runtimes are too small, but in the rest we can see that there is a speedup of 4. And the GFLOPS are looking much better in the context of our clock speed. We are comfortably doing around 2 floating point operations per clock cycle.

4 OpenMP Implementations

4.1 Code

We have added a third argument to the OMP program that allows the user to specify the number of threads to use. We use `omp_set_num_threads` in the code with that argument to initialize the omp threads.

We have focused on parallelizing the relax method. We added the following OMP pragma directly before the outer for loop: `#pragma omp parallel for schedule(guided,1) collapse(2)`. This divides the work into the different threads using the `guided` scheduling strategy. It is difficult to know for sure whether the differences in the measurements we saw between different strategies were due to one of them having an edge, or just inconsistencies from the cluster. But `guided` seemed to have the lowest runtimes according to our experiments. The `collapse(2)` clause is used in order to parallelize both levels (the outer and the inner loop).

4.2 Evaluation

To continue establishing our baseline, we need to compare the OMP versions running with one thread to the sequential versions (both original and vectorized). We used the exact same strategy we discussed in section 3.2 for vectorizing the OMP version. Figure 1 shows a comparison between the sequential versions and the OMP versions running with one thread. This measurement is for a matrix of size 8192, and the number of iterations was fixed at 5.

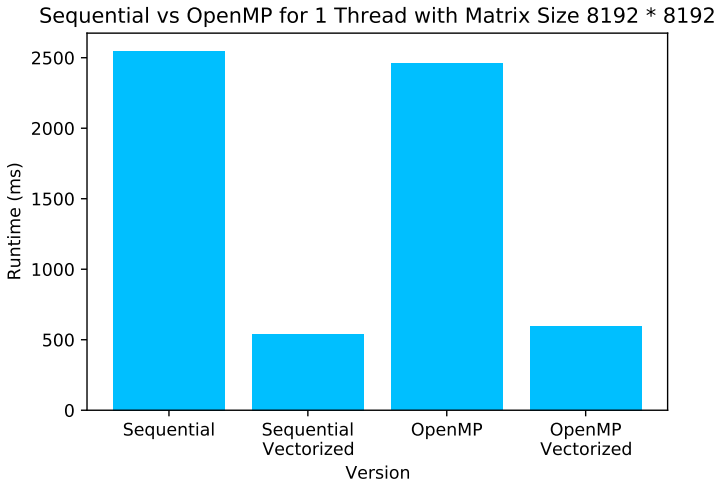

Figure 1: Comparison of wall clock time between the sequential versions and OpenMP versions with 1 thread. The matrix size was fixed at 8192 and the number of iterations was fixed at 5.

We can see that there is practically no overhead, and the two versions match. Our optimization strategy seems to have worked properly! We will discuss more in depth measurements and comparisons with other versions and variables in section 6.

5 MPI Implementations

5.1 Code

We went through many iterations while working on the MPI implementation but we ended up with the following strategy:

- The matrix is sliced as equally as possible between all ranks, meaning that each rank will calculate a certain number of rows.
- Each rank has an `in` and an `out` matrix. Those matrices only contain the slice of the original matrix that the respective rank is concerned with. This is done to conserve memory. Note that those matrices are not square, so we had to modify quite a large portion of the code to support this.
- Rank 0 has a `master` matrix with size n, this matrix will contain the final result.
- Each rank only communicates to its "neighboring" ranks, i.e. the ranks that calculate neighboring slices. Only the borders between slices are communicated after each iteration. This is done using `MPI_Send` and `MPI_Recv`.
- After all iterations finish, `MPI_Gatherv` is used to collect all the output slices from all ranks to the `master` matrix in rank 0.

5.2 Evaluation

We repeated the same experiment of establishing the baseline for MPI. So we compared running it with one rank to the sequential versions. Again, we still used the same strategy discussed in section 3.2 for vectorizing the MPI version. Figure 2 shows a comparison between the sequential versions and the MPI versions running with one rank. This measurement is for a matrix of size 8192, and the number of iterations was fixed at 5.

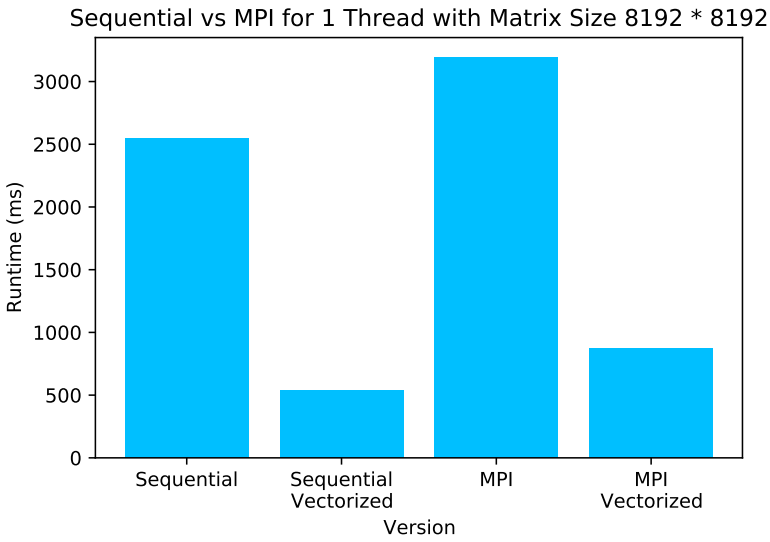

Figure 2: Comparison of wall clock time between the sequential versions and MPI versions with 1 rank. The matrix size was fixed at 8192 and the number of iterations was fixed at 5.

Unfortunately there seems to be a bit of overhead. Obviously, this is quite understandable in an MPI implementation due to the cost of communication, especially on the cluster. We have installed OpenMPI locally and were unable to reproduce this difference in performance between the sequential code and the MPI code with one rank. So it is a real possibility that the cluster is contributing to those differences. Again, we will discuss more detailed testing in section 6.

6 Performance

6.1 Method

We wrote a bash script that compiles all of our programs and then performs all of our tests on the cluster. We used `sbatch` to get this running. The script starts by varying the matrix size, the possible values it can take are 256, 512, 1024, 2048, 4096, 8192, 16384, and 32768. For each of those values, we measured the performance (5 times) of all of our programs for all thread/rank counts in the set: 1, 2, 4, 8, 16, and 32. The script then measures the performance when the size of the matrix is fixed, but the number of iterations changes. But we have found that this increases almost linearly with the number of iterations. So it is not the most interesting data.

6.2 Data

Figure 3 shows how the different OMP and MPI versions scale as we increase the thread/rank count for a fixed matrix size of 8192 and a fixed number of iterations of 5. Since we saw that the versions with one thread/rank already more or less match the sequential versions, we can see that using more processors reduces the runtime beyond the best case for the sequential version, so we can say for sure that our parallelization is practically useful!

To get a better understanding of how these implementations scale, we can compare their speedups over the sequential versions as well as their efficiencies. Figures 4 and 5 show a comparison of this. Clearly, it is not exactly perfect... or even close. Especially for the vectorized versions.

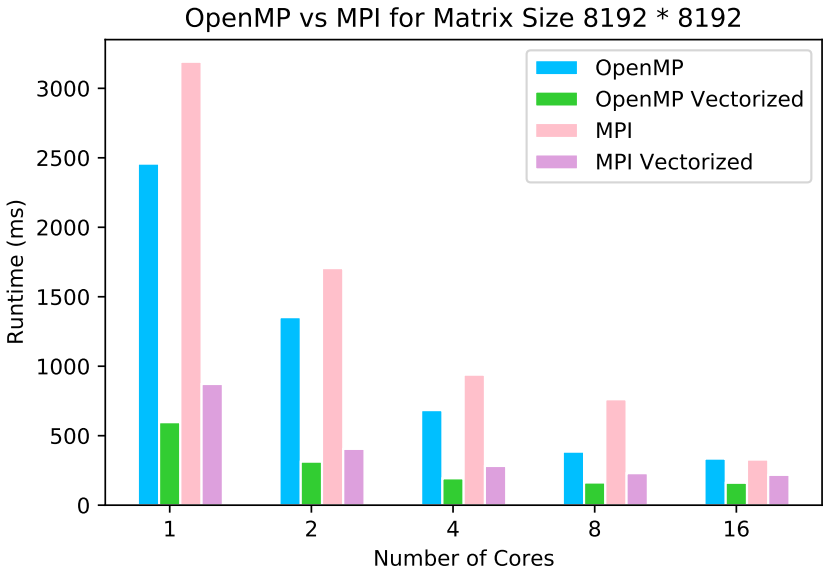

Figure 3: Comparison of wall clock time between the OMP and MPI versions as a function of thread/rank count. The matrix size was fixed at 8192 and the number of iterations was fixed at 5.

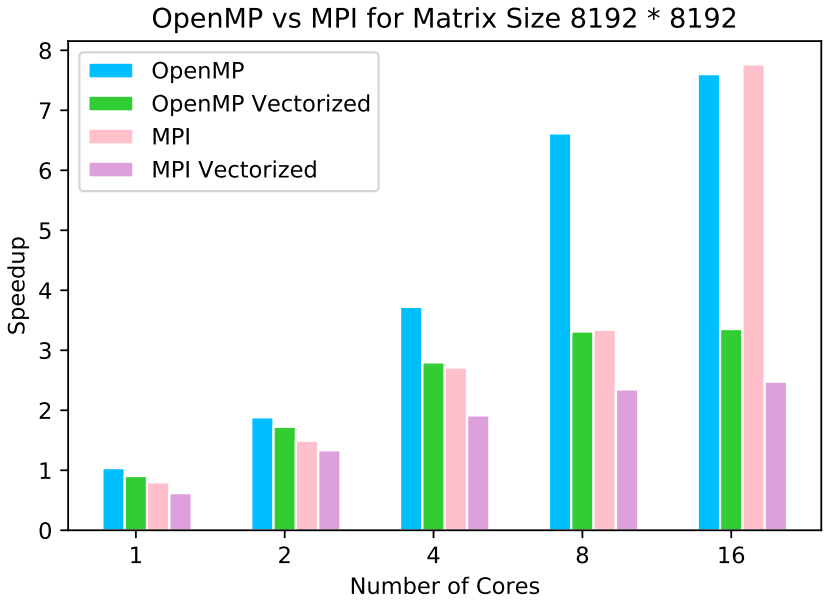

Figure 4: Comparison of performance speedup between the OMP and MPI versions over the respective sequential versions as a function of thread/rank count. The matrix size was fixed at 8192 and the number of iterations was fixed at 5.

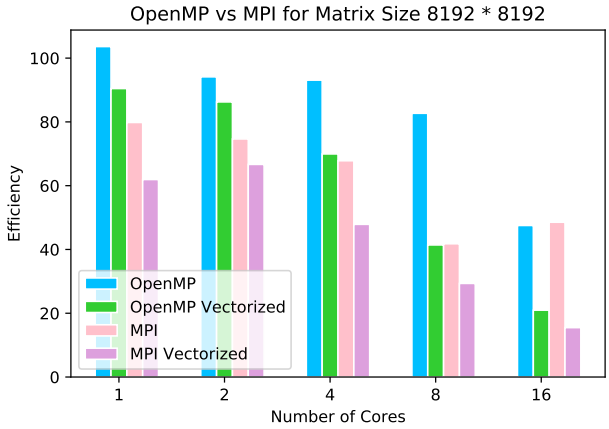

Figure 5: Comparison of efficiency between the OMP and MPI versions as a function of thread/rank count. The matrix size was fixed at 8192 and the number of iterations was fixed at 5.

It is also useful to look into how performance changes when both the matrix size and the core count changes. Especially for determining weak scaling. Tables 3 through 6 show the runtimes for the unoptimized OMP version, vectorized OMP version, unoptimized MPI version, and vectorized MPI version respectively. Tables 7 through 10 show the speedups.

Matrix Dimensions	1024 * 1024	2048 * 2048	4096 * 4096	8192 * 8192	16384 * 16384	32768 * 32768
1 core	53ms	150ms	593ms	2458ms	10534ms	44809ms
2 cores	31ms	100ms	323ms	1353ms	5432ms	24186ms
4 cores	19ms	59ms	187ms	684ms	2771ms	11727ms
8 cores	14ms	42ms	107ms	385ms	1508ms	6295ms
16 cores	21ms	36ms	99ms	335ms	1323ms	5703ms
32 cores	80ms	102ms	122ms	318ms	977ms	4418ms

Table 3: Wall clock times of unoptimized OMP version as a function of matrix size and core count.

Matrix Dimensions	1024 * 1024	2048 * 2048	4096 * 4096	8192 * 8192	16384 * 16384	32768 * 32768
1 core	11ms	46ms	143ms	597ms	2663ms	12546ms
2 cores	7ms	28ms	91ms	313ms	1458ms	7094ms
4 cores	4ms	17ms	52ms	193ms	779ms	3847ms
8 cores	3ms	14ms	42ms	163ms	599ms	3945ms
16 cores	10ms	20ms	60ms	161ms	747ms	3519ms
32 cores	97ms	94ms	123ms	251ms	895ms	4320ms

Table 4: Wall clock times of vectorized OMP version as a function of matrix size and core count.

Matrix Dimensions	1024 * 1024	2048 * 2048	4096 * 4096	8192 * 8192	16384 * 16384	32768 * 32768
1 core	56ms	203ms	712ms	3191ms	13534ms	54600ms
2 cores	29ms	109ms	358ms	1706ms	7531ms	30800ms
4 cores	16ms	65ms	208ms	939ms	3944ms	18128ms
8 cores	13ms	55ms	139ms	762ms	2414ms	9945ms
16 cores	8ms	30ms	89ms	328ms	2114ms	6627ms
32 cores	9ms	27ms	86ms	320ms	1692ms	6811ms

Table 5: Wall clock times of unoptimized MPI version as a function of matrix size and core count.

Matrix Dimensions	1024 * 1024	2048 * 2048	4096 * 4096	8192 * 8192	16384 * 16384	32768 * 32768
1 core	8ms	40ms	161ms	872ms	4856ms	20475ms
2 cores	5ms	25ms	101ms	405ms	3706ms	16385ms
4 cores	4ms	18ms	71ms	282ms	2009ms	10090ms
8 cores	3ms	16ms	58ms	230ms	1390ms	6237ms
16 cores	3ms	13ms	53ms	218ms	1245ms	4986ms
32 cores	4ms	13ms	55ms	210ms	1005ms	4957ms

Table 6: Wall clock times of vectorized MPI version as a function of matrix size and core count.

Matrix Dimensions	1024 * 1024	2048 * 2048	4096 * 4096	8192 * 8192	16384 * 16384	32768 * 32768
1 core	1.04	1.04	1.04	1.04	1.03	1.02
2 cores	1.77	1.56	1.90	1.88	2.00	1.89
4 cores	2.89	2.64	3.29	3.72	3.93	3.91
8 cores	3.93	3.71	5.75	6.62	7.22	7.28
16 cores	2.62	4.33	6.21	7.60	8.22	8.03
32 cores	0.69	1.53	5.04	8.01	11.14	10.37

Table 7: Speedup of unoptimized OMP version relative to the original unoptimized sequential code as a function of matrix size and core count.

Matrix Dimensions	1024 * 1024	2048 * 2048	4096 * 4096	8192 * 8192	16384 * 16384	32768 * 32768
1 core	0.55	0.72	0.94	0.90	0.85	0.84
2 cores	0.86	1.18	1.47	1.73	1.56	1.49
4 cores	1.50	1.94	2.58	2.80	2.92	2.75
8 cores	2.00	2.36	3.19	3.31	3.79	2.68
16 cores	0.60	1.65	2.23	3.35	3.04	3.01
32 cores	0.06	0.35	1.09	2.15	2.54	2.45

Table 8: Speedup of vectorized OMP version relative to the original vectorized sequential code as a function of matrix size and core count.

Matrix Dimensions	1024 * 1024	2048 * 2048	4096 * 4096	8192 * 8192	16384 * 16384	32768 * 32768
1 core	0.98	0.77	0.86	0.80	0.80	0.84
2 cores	1.90	1.43	1.72	1.49	1.44	1.49
4 cores	3.44	2.40	2.96	2.71	2.76	2.53
8 cores	4.23	2.84	4.42	3.34	4.51	4.61
16 cores	6.88	5.20	6.91	7.77	5.15	6.91
32 cores	6.11	5.78	7.15	7.96	6.43	6.73

Table 9: Speedup of unoptimized MPI version relative to the original unoptimized sequential code as a function of matrix size and core count.

Matrix Dimensions	1024 * 1024	2048 * 2048	4096 * 4096	8192 * 8192	16384 * 16384	32768 * 32768
1 core	0.75	0.82	0.83	0.62	0.47	0.52
2 cores	1.20	1.32	1.33	1.33	0.61	0.65
4 cores	1.50	1.83	1.89	1.91	1.13	1.05
8 cores	2.00	2.06	2.31	2.35	1.64	1.70
16 cores	2.00	2.54	2.53	2.48	1.83	2.12
32 cores	1.50	2.54	2.44	2.57	2.26	2.13

Table 10: Speedup of vectorized MPI version relative to the original vectorized sequential code as a function of matrix size and core count.

6.3 Scalability

The data we used for Figure 3 gives us information for determining strong scalability, since we are keeping the size of the matrix constant but varying the thread count. We can see that doubling the number of threads reduces (in some cases even halves!) the runtime. This is more or less the case for the unoptimized OMP and MPI versions, but is not that pronounced in the vectorized versions.

To achieve strong scaling, doubling the number of threads will result in halving the runtime, quadrupling the thread count will result in reducing the runtime to a quarter of the runtime when using a single thread, and so on. For unoptimized OMP that is indeed the case when the number of cores is less than 16 which can also be seen in table 7. For unoptimized MPI this also holds when the number of cores less than 8, which is seen in table 9. For the vectorized versions this is not happening: for both of them, using 2 threads means half the runtime of using a single thread, but for any larger number of threads they plateau, as can be seen in tables 8 and 10. In conclusion, we obviously do not achieve perfect strong scaling (especially in the case of the vectorized versions, which have no strong scaling whatsoever). But the unoptimized versions show potential and accomplish some strong scaling for a small number of threads. This is further illustrated by figure 5 which shows that the efficiency is quite high for a small number of threads and then degrades drastically after the work is distributed between more than 8 threads.

The tables provide information about weak scaling as well. To judge that we need to check whether increasing the size of the matrix and the thread count proportionally will result in more or less equal runtimes (so the work is equally distributed between the threads). For example, we can analyze and compare the runtimes of executing the code on 1 core for n=1024, on 4 cores for n=2048 and on 16 cores for n=4096.

Looking at those values in tables 3 and 4, we can see that there is some weak scaling up until 16 cores when it comes to the OMP versions. Looking at the aforementioned cells in tables 5 and 6, we can see that the unoptimized MPI version also has weak scaling up until 16 cores, while the vectorized MPI does not achieve any weak scaling. In conclusion, none of the four versions achieve it perfectly, but for smaller number of threads they do have some weak scaling.

6.4 Discussion

Admittedly, we may have too many comments about working on the cluster in this report. We are aware that there was a lot of effort by the teaching team to make this as smooth as possible, which is greatly appreciated.

With that said, we truly can not make a single well-founded conclusion about the data we got due to how inconsistent the measurements are. Not only do multiple runs with the exact same code and arguments have error margins that are many multiples of the measurements themselves, most of the time it is flat out impossible to even measure anything due to the cluster being overloaded or unavailable.

Obviously, an alternative is testing things locally. But the problem with that is that all members of our group can only realistically provide 4 cores at the very best. We still did this anyway, and as mentioned above, the data made much more sense than what we got from the cluster. But again, we cannot draw any conclusions from that, because the core count is too low. So we decided to go with the cluster data.

We do not know whether our MPI baseline is worse than the sequential versions (as can be seen in Figure 2) due to a flawed implementation, or flaws in the testing environment. But as we mentioned above, it seemed way better when we ran it locally using the latest version of OpenMPI.

The same applies to scalability. We do not know whether the plateauing in speedup is due to flaws in our implementation for the same reasons. Additionally, even though everything scaled well when ran locally, we still cannot make any conclusions from that because the core count we

have available is too low, and we already see scalability for low core counts in the cluster anyway.

Despite the quality of the cluster, we found the assignment to be very interesting and helpful for learning OMP and MPI. So it really is a shame we did not get to properly test our implementations. Obviously we believe that more controlled and consistent testing should be the priority when thinking about future directions of this work.

6.5 Programming and Debugging Efforts

It goes without saying that, in general, debugging this sort of program is a nightmare :). But to go into more detail: optimizing the sequential code using proper array allocation took some digging, we also misled ourselves by trying to optimize the algorithm itself first as opposed to the allocation. As for OMP, the most difficult part was the fact that we were not really taught all of the tools that are needed to actually parallelize this job (e.g. the `collapse` clause). So figuring that out took a bit of effort. Finally, working on MPI was absolutely exhausting. A lot of effort was needed to work out the intricacies of slicing, communication, and debugging.

7 Work Distribution

7.1 Hamzah

I organized the work that we have to do and managed the project. I mainly focused on implementing the MPI versions (task 3). I helped the rest of the team a little bit with the other tasks as well. Finally I contributed by writing a major part of the report. If I were to distribute 16 points among us, I would give each member 4 because everyone did their part and we worked smoothly as a team!

7.2 Nadia

I worked on task 2 - creating the OpenMP versions, and running experiments. I also worked on creating tables and graphs for the visualization of the performance of our code. Furthermore, I did some of the analysis of the results. Finally, I participated in writing the report. I believe that all of us should get 4 points, because we contributed equally to the project.

7.3 Seraph

I was helping write the shell script to get the output that was used for performance evaluation. I also assisted to create some graphs, and help running experiments on the cluster. Discuss ideas with members. I would like to give all members a 4 because everyone was trying their best. I think we all did a great job, and we deserve the points.

7.4 Tim

I worked on task 1, vectorizing the sequential version. I created the script to parse all output from our testing in order to use it for visualisation through fancy graphs and tables. After the script I also took part in the actual visualisation it self. I believe we all deserve equal points as we all contributed equally.