CS3210 Assignment 1 Report

Ye Tong A0183760B

Yu Xiaoxue A0187744N

Section 1 Program Design

Program Design of Pthread Implementation

For Pthread implementation, we have tried two solutions. For the first solution, we parallelise loops involving *copy startWorld*, *copy invasionPlans* and *get the new states*. For this implementation, we take out these loops and write them into functions and for every generation, we will create new threads to execute them. However, the result for this approach is relatively bad. In general, all of our test cases run slower as compared to the sequential implementation. The graphs of this implementation can be seen in figure 9 and 10 in the appendix.

For the second solution, we decided to mainly focus on parallelising the *get new state* section inside the `goi()` function. For this implementation, we take out the ***generation*** loop and let all threads execute the new generation function containing this loop. The only section where we want all the threads to execute together is the *get new state* loop. To ensure that other sections are executed sequentially by only one thread, we pass in the `thread_id` and execute these segments only when `thread_id = 0`. We also implemented `pthread_barrier_wait(&barrier)` before and after the *get new state* loop, so that data is updated correctly. This approach is much faster and we are able to achieve execution time close to OpenMP implementation.

In order to speed up the sequential program efficiently, we need to distribute a suitable amount of workload to each thread during the implementation. The workload in this case is the outer loop, which is the total number of rows of the world. Each thread will be given a certain number of rows to process. One consideration is that each thread will take (no.of task/no.of thread) rows. For example, for 9 tasks with 5 threads, each thread will take 9/5 = 1 row first, and the last thread will take the remaining 4 tasks. This means that the first 4 threads will process one row each and the last thread will process 5 rows. However, this is not a very efficient distribution of workload as the last thread needs to process much more rows than the rest.

Therefore, we have come up with another consideration which is to distribute the workload equally as much as possible. We firstly allocate each thread with `ceil(9/5) = 2` rows. The total number of rows that the threads can process will be greater than the total number of rows by 1. Hence for the last thread, we only assign 1 row to it instead of 2. The workload is then equally distributed with the first 4 threads processing 2 rows and the last thread processing 1 row. This will help to speed up the program more efficiently.

Another consideration implemented was when the number of threads used is more than the number of rows in the input, the program will set the number of threads equal to the number of rows. This is to ensure that every thread declared will have at least one row to work on. This is also necessary for ensuring the correctness of the algorithm mentioned above.

Program Design of OpenMP Implementation

To efficiently speed up the sequential program, we have to consider loop segments of the code with few or no critical sections. From the `goi.c` code, we identified a few of the loop sections in the `goi()` function where we can implement OpenMP. The segments are loops to *copy startWorld*, *copy invasionPlans* and the *get new state*.

For *copying startWorld*, it will run N_ROWS * M_COLS number of iterations, which can be significant when N_ROWS and M_COLS are large. Whereas for *copying invasionPlans*, it will also run N_INVASIONS * N_ROWS * M_COLS number of iterations, which will also have a significant impact when N_INVASIONS, N_ROWS and M_COLS are large. Even if N_INVASIONS is 0, writing parallel code for this segment will not introduce additional overhead since it will not be run. For loops to *get new states*, it will run N_GENERATIONS * N_ROWS * M_COLS, which will be significant when N_GENERATIONS, N_ROWS, M_COLS are large. However, this parallel section will update a shared variable `deathToll`**.** Thus, `#pragma omp critical` is implemented to prevent multiple threads from updating `deathToll` at the same time.

Non-trivial Consideration

In the `getNextState()` function, there is a double for loop to count neighbors. We initially think that we can parallelise this segment to improve performance. However, this section only runs for a fixed number of 9 iterations. Implementing parallelism may create more overhead than speedup. Thus we decide to discard the idea.

Our test cases are designed with N_INVASIONS = 0. The main reason is that it is easier and more consistent to generate test cases and evaluate the execution time of the different implementations with no invasion (i.e. we will need test cases with N_INVASIONS ranging from 0 to 1000, which can be tedious to create). Therefore, although parallelism is implemented for copying the invasionPlans mentioned in OpenMP implementation, the overhead and speedup is not evaluated by our test cases.

Section 2 Reproduction of results

Inputs

Test cases `test0.in` to `test5.in` are cases with

1. N_GENERATIONS = 100000,
2. N_ROWS * M_COLS = 10*10, 20*20, 30*30, 40*40, 50*50 and 100*100 respectively,
3. N_INVASIONS = 0.

Test cases test6g0.in to test6g4.in are cases with

1. N_GENERATIONS = 100, 1000, 10 000, 100 000 and 1 000 000 respectively
2. N_ROWS * M_COLS = 50*50
3. N_INVASIONS = 0.

Each test case will run 3 times and the best result will be chosen.

We used threads number = 1,2,4,8,10,20 and 30 in our tests. Although up to 64 threads can be used, the execution overhead is too large and it will require too much time to run 64 threads. We decided to use 30 as the representative for trying to use more than available threads because the trend of the outcome for using 30 and 64 is similar and 30 is more manageable for conducting multiple repeated tests.

Outputs

Reference output of the test cases are generated with the sequential code. Both OpenMP and Pthread implementation's output are cross checked with the reference output produced by the sequential implementation to ensure correctness of the code.

Execution time measurement

Execution time measurement is conducted in the `goi()` function. The start time is obtained at the start of the function, before executing any other useful code. The end time is obtained at the end of the function, before executing the `return` statement. The execution time is calculated by obtaining the time difference between the start time and the end time. The purpose of capturing the executing time of the whole function is to take the overhead of all thread/OpenMP related operations into consideration of the speedup.

Section 3 Performance Measurement

Machine: soctf-pdc-004 Xeon Silver 4114

Sequential Implementation

Figure 1. Sequential Time of Various Input Size

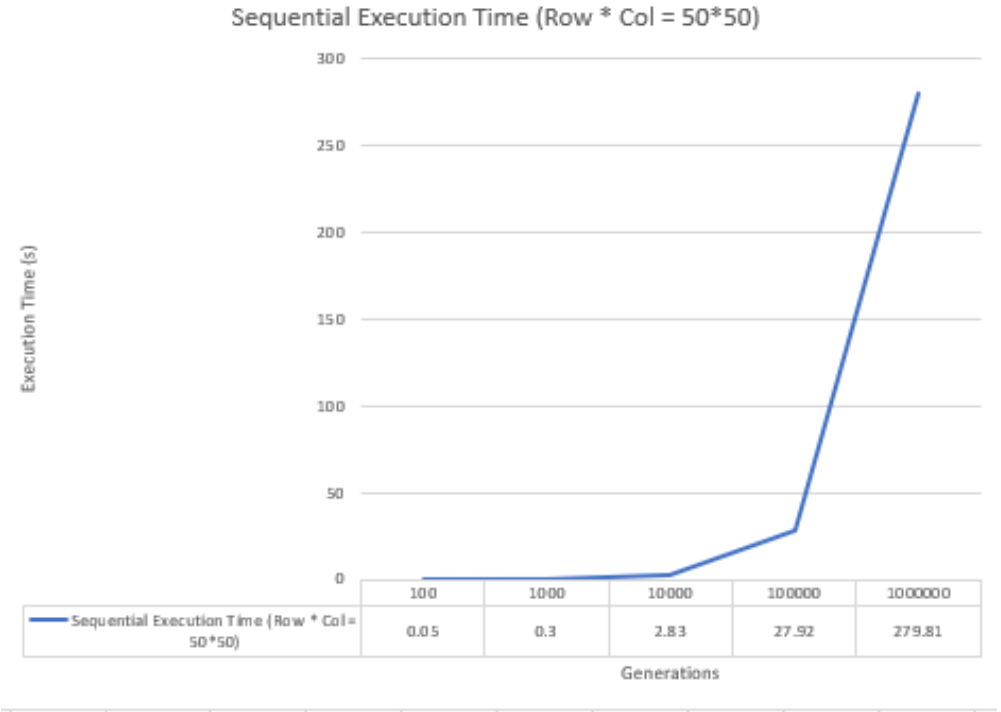
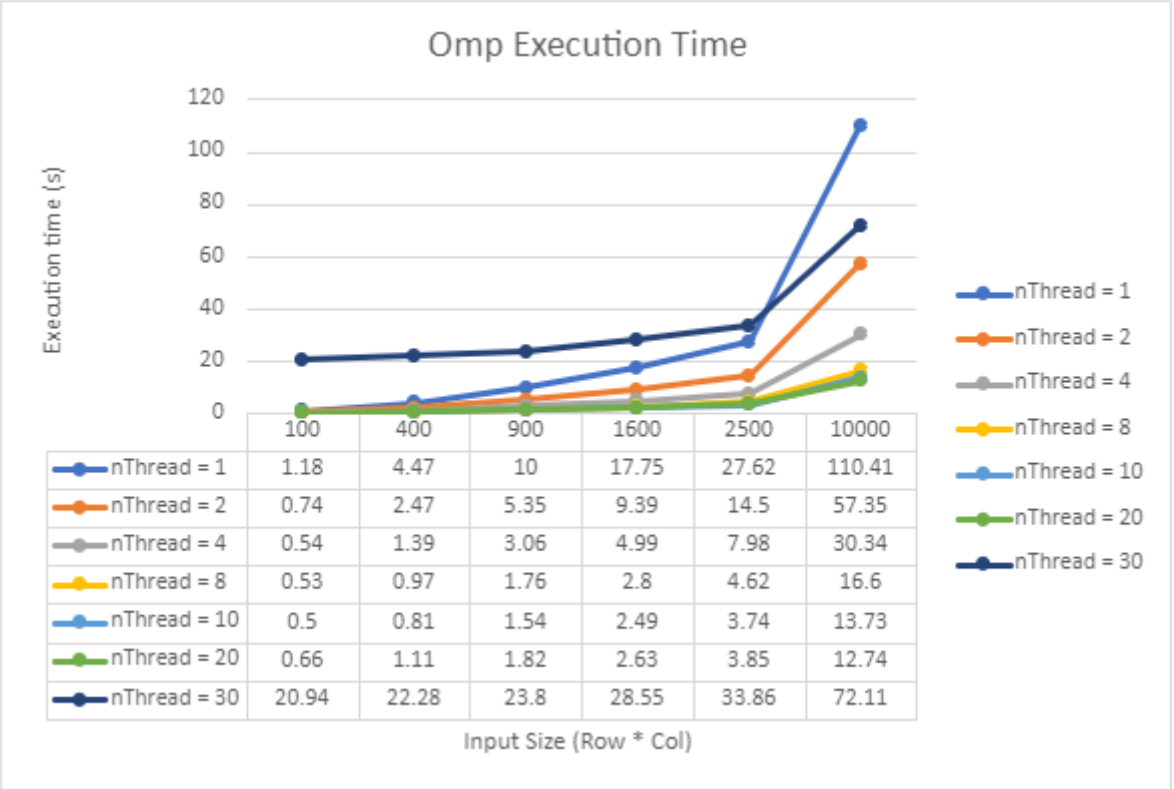


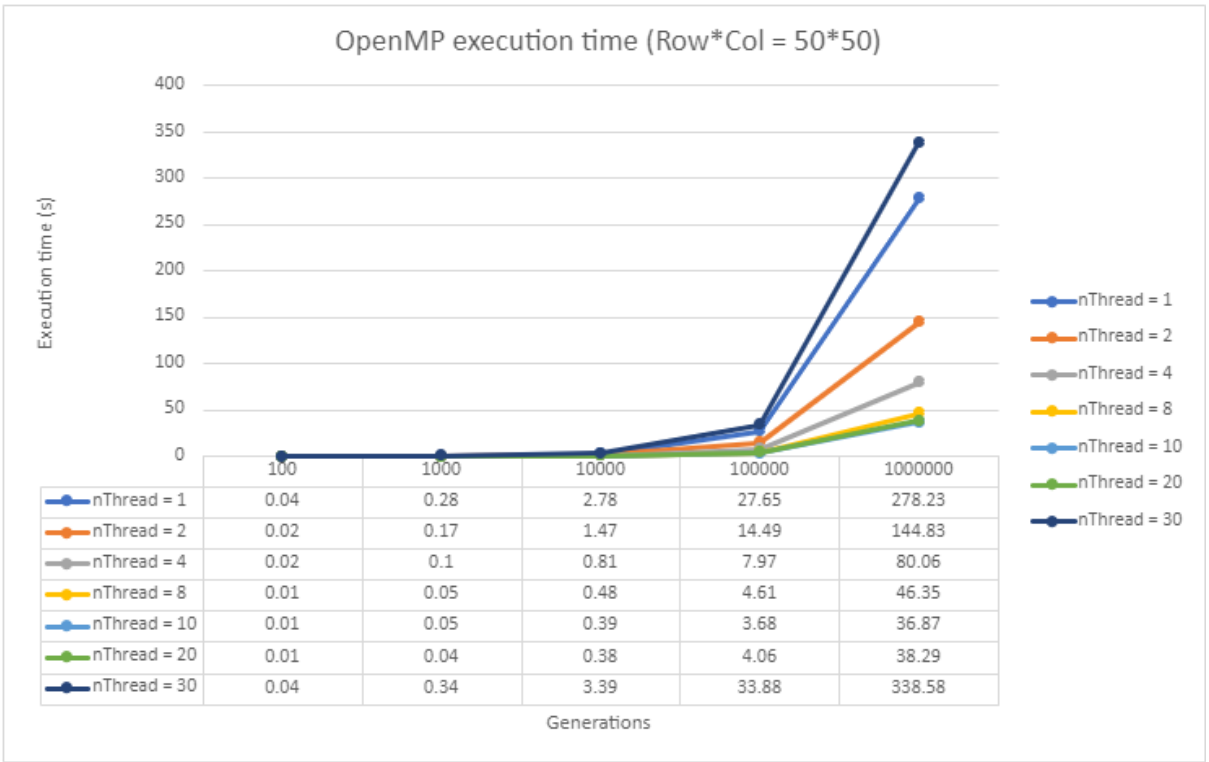
Figure 2. Sequential Time of Various Generations with Fixed Input Size

OpenMP Implementation



	100	400	900	1600	2500	10000
nThread = 1	1.18	4.47	10	17.75	27.62	110.41
nThread = 2	0.74	2.47	5.35	9.39	14.5	57.35
nThread = 4	0.54	1.39	3.06	4.99	7.98	30.34
nThread = 8	0.53	0.97	1.76	2.8	4.62	16.6
nThread = 10	0.5	0.81	1.54	2.49	3.74	13.73
nThread = 20	0.66	1.11	1.82	2.63	3.85	12.74
nThread = 30	20.94	22.28	23.8	28.55	33.86	72.11

Figure 3. OpenMP Execution Time of Various Input Sizes



	100	1000	10000	100000	1000000
nThread = 1	0.04	0.28	2.78	27.65	278.23
nThread = 2	0.02	0.17	1.47	14.49	144.83
nThread = 4	0.02	0.1	0.81	7.97	80.06
nThread = 8	0.01	0.05	0.48	4.61	46.35
nThread = 10	0.01	0.05	0.39	3.68	36.87
nThread = 20	0.01	0.04	0.38	4.06	38.29
nThread = 30	0.04	0.34	3.39	33.88	338.58

Figure 4. OpenMP Execution Time of Various Generations with Fixed Input Size

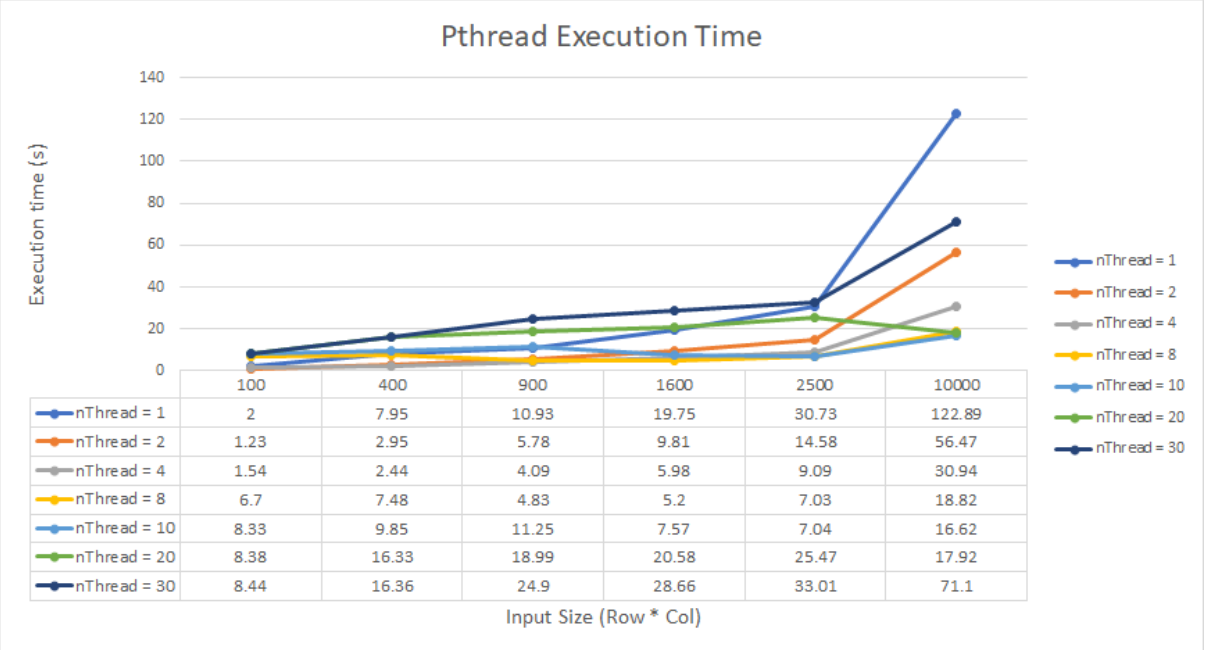
Figure 3 shows the execution time of the OpenMP implementation with various input sizes and figure 4 shows the execution time of fixed size with various generations. From figure 3, we can observe that when the thread number is 20, it has the fastest execution time, which is about 9 times faster than single thread performance. This is because the Xeon Silver 4114 processor has 10 cores with 20 threads. Hence, running with 20 threads helps to achieve the best performance.

From both figure 3 and 2, we observe that for thread numbers from 1 to 20, the execution time difference for small input sizes is small as sequential segments of the code are more significant with a smaller input size. However, for large input sizes, overheads and sequential segments become less significant. Thus we are able to achieve greater speedup. Therefore, the execution time of the program decreases significantly with more threads when the input sizes are big.

When the thread number is more than 20, the execution time for OpenMP implementation increases greatly. When declaring more than available threads, the program will still create the additional threads with additional overhead but the additional threads will not be doing any useful work as there is no actual thread available on the processor. Since the Xeon Silver 4114 processor has only 20 threads available, any number more than 20 will incur more overheads with no speedup.

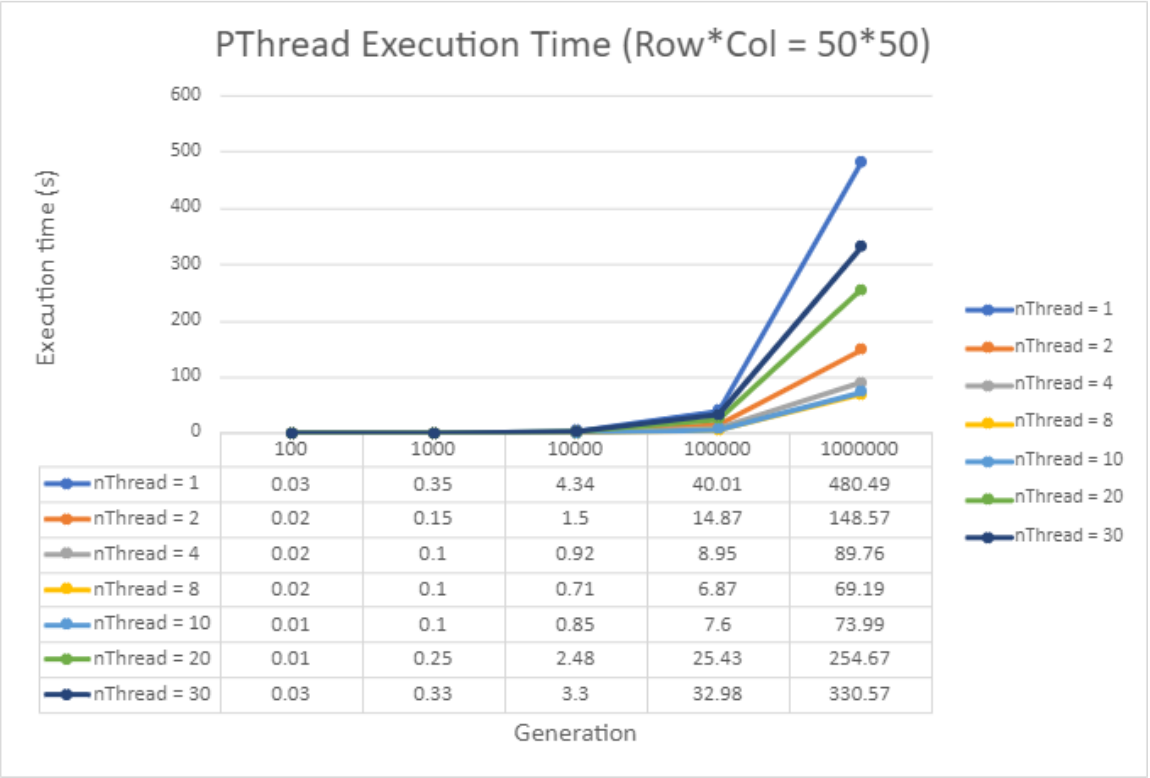
Comparing figure 1 and figure 3, for input size of 50 * 50 (generation = 10 000), OpenMP with 8 threads is able to execute 6 times faster. Comparing figure 2 and figure 4, for the number of generations = 1000000 (50*50), OpenMP with 8 threads is able to execute 6 times faster.

Pthread Implementation



	100	400	900	1600	2500	10000
nThread = 1	2	7.95	10.93	19.75	30.73	122.89
nThread = 2	1.23	2.95	5.78	9.81	14.58	56.47
nThread = 4	1.54	2.44	4.09	5.98	9.09	30.94
nThread = 8	6.7	7.48	4.83	5.2	7.03	18.82
nThread = 10	8.33	9.85	11.25	7.57	7.04	16.62
nThread = 20	8.38	16.33	18.99	20.58	25.47	17.92
nThread = 30	8.44	16.36	24.9	28.66	33.01	71.1

Figure 5. Pthread Execution Time of Various Input Sizes



	100	1000	10000	100000	1000000
nThread = 1	0.03	0.35	4.34	40.01	480.49
nThread = 2	0.02	0.15	1.5	14.87	148.57
nThread = 4	0.02	0.1	0.92	8.95	89.76
nThread = 8	0.02	0.1	0.71	6.87	69.19
nThread = 10	0.01	0.1	0.85	7.6	73.99
nThread = 20	0.01	0.25	2.48	25.43	254.67
nThread = 30	0.03	0.33	3.3	32.98	330.57

Figure 6. Pthread Execution Time of Various Generations with Fixed Input Size

Figure 5 shows the execution time of the Pthread implementation with various input sizes and figure 6 shows the execution time of Pthread implementation with various generations and fixed input size. From both figures, we can observe that when the thread is less than 20, increasing the number of threads also increases the performance (decrease the execution time) of the program.

From the graph, we observe that for thread numbers from 1 to 20, the execution time is longer for small input sizes as the overheads incurred is significant compared to the reduction in time, causing the execution time to be longer than single thread performance. However, for large input sizes (Rol * Col = 10000 and Generation = 1000000), the reduction in execution time is generally more significant as the overheads becomes less significant as compared to the reduction in time.

When the thread number is more than 20, the execution time for Pthread implementation increases greatly. When declaring more than available threads, the program will still create the additional threads with additional overhead (similar to OpenMP).

Comparing figure 1 and figure 5, for input size of 50 * 50 (generation = 10 000), Pthread with 8 threads is able to execute 6 times faster. Comparing figure 2 and figure 6, for number of generation is 100000 (Row * Col = 50*50), Pthread with 8 threads is able to execute 4 times faster.

OpenMP vs Pthread Implementation

From the graphs above, we can observe that OpenMP implementation is generally faster than Pthread implementation. Hence, OpenMP performs better than Pthread according to the execution time. This could be due to a few possible reasons.

OpenMP is more optimised and has less overhead dividing the workload among threads and thread synchronisation [1]. In OpenMP, the workload is divided evenly inside the API [1]. However, for

Pthread implementation, we have to manually divide the workload according to the number of threads and rows and global variables are used to share data among threads. This causes overheads which makes the implementation slower. Thread synchronisation for OpenMP is done internally inside the API whereas for Pthread, we need to use pthread_join() to wait for all threads to complete. This may also lead to overheads which slows down the execution.

Another reason is that OpenMP implementation has less cache miss than the Pthread Implementation. We conducted an experiment to measure the cache miss for both OpenMP and Pthread implementation for a thread number of 8. The result is shown in figure 7 and 8 below.

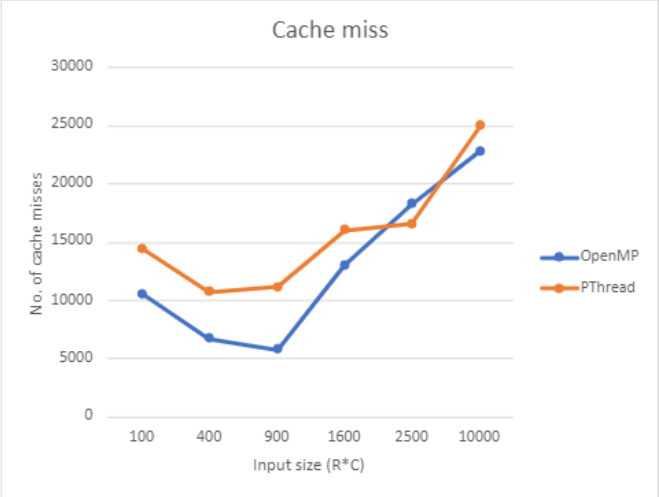


Figure 7. Cache Miss of OpenMP and Pthread Implementation with Various Input Sizes

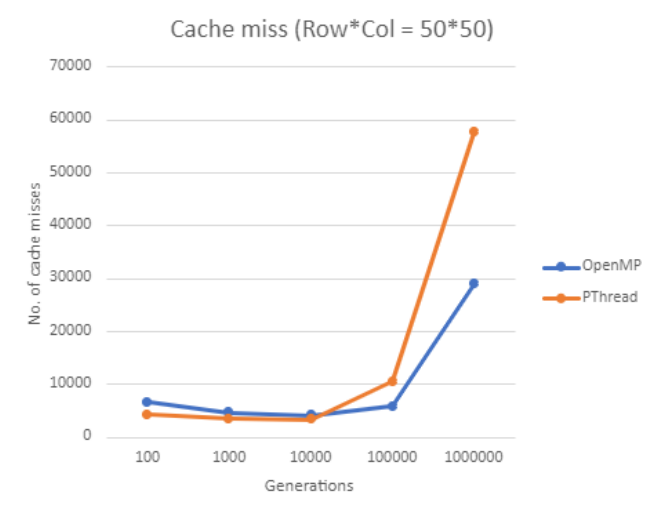


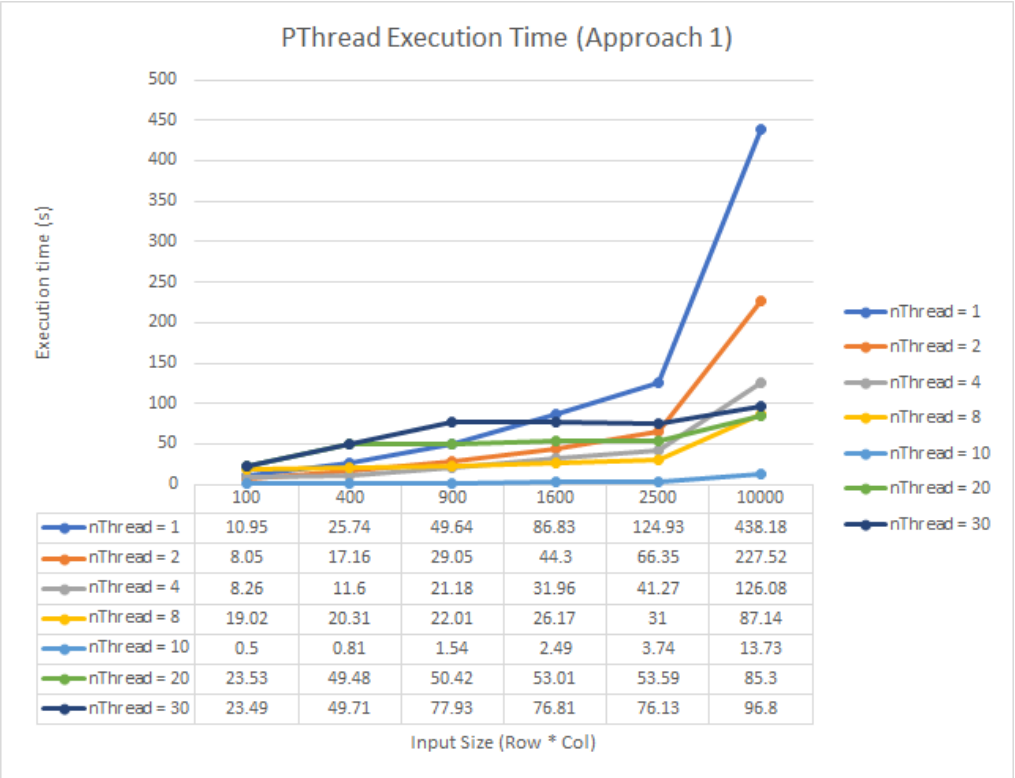
Figure 8. Cache Miss of OpenMP and Pthread Implementation with Various Generations

From figure 7 and 8, we can observe that Pthread implementation generally has a greater number of cache misses as compared to OpenMP. This will contribute greatly to the execution time since every cache miss will require fetching the data from the memory. Hence this also explains the difference in the execution time between OpenMP and Pthread implementation.

Section 4 Conclusion

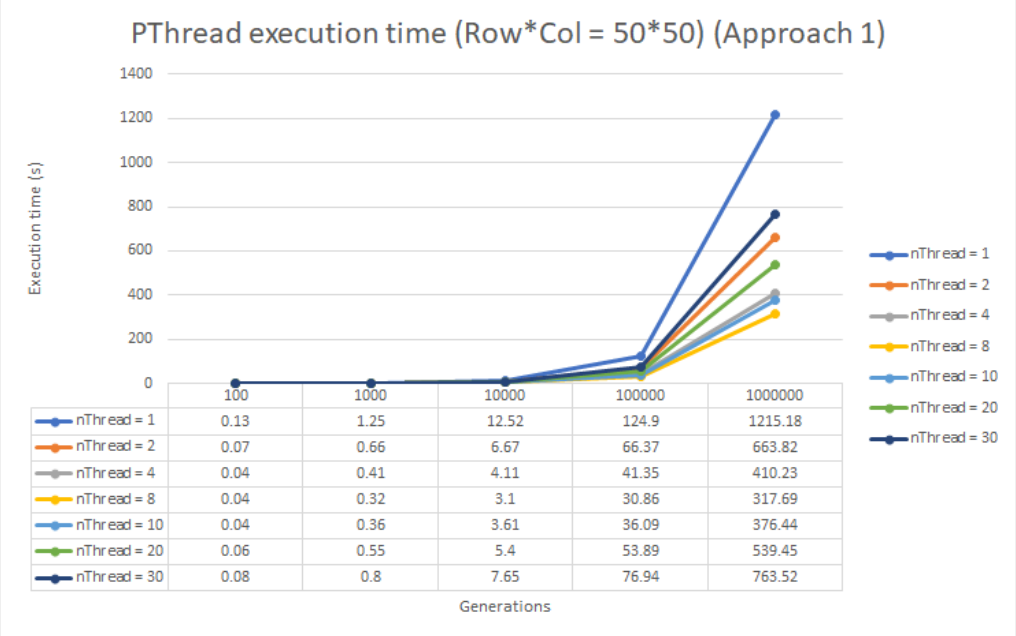
In conclusion, based on our experiment result, we can conclude that multi threads implementation is generally faster than sequential implementation with big input sizes as overheads incurred from multi threading is insignificant compared to the reduction in time. OpenMP also generally performs better than Pthread because OpenMP implementation is more optimised as compared to our Pthread implementation.

Appendix



	100	400	900	1600	2500	10000
nThread = 1	10.95	25.74	49.64	86.83	124.93	438.18
nThread = 2	8.05	17.16	29.05	44.3	66.35	227.52
nThread = 4	8.26	11.6	21.18	31.96	41.27	126.08
nThread = 8	19.02	20.31	22.01	26.17	31	87.14
nThread = 10	0.5	0.81	1.54	2.49	3.74	13.73
nThread = 20	23.53	49.48	50.42	53.01	53.59	85.3
nThread = 30	23.49	49.71	77.93	76.81	76.13	96.8

Figure 9. Pthread Execution Time of Various Input Sizes (Approach 1)



	100	1000	10000	100000	1000000
nThread = 1	0.13	1.25	12.52	124.9	1215.18
nThread = 2	0.07	0.66	6.67	66.37	663.82
nThread = 4	0.04	0.41	4.11	41.35	410.23
nThread = 8	0.04	0.32	3.1	30.86	317.69
nThread = 10	0.04	0.36	3.61	36.09	376.44
nThread = 20	0.06	0.55	5.4	53.89	539.45
nThread = 30	0.08	0.8	7.65	76.94	763.52

Figure 10. Pthread Execution Time of Various Generations (Approach 1)

Reference

1. Swahn, H., 2021. *Pthreads and OpenMP A performance and productivity study*. [online] Diva-portal.org. Available at: <http://www.diva-portal.org/smash/get/diva2:944063/FULLTEXT02> [Accessed 26 September 2021].