National University
of computer and emerging sciences

Project Phase 2

CL-220 Operating Systems

BS(CS) – A

Batch 2018

Submitted By:

Hassan Shahzad 18i-0441

Submitted to:

Sir Mehran Khan

Date of Submission:

20-12-20

PHASE 1 Analysis

For implementation of **Phase 1**, I used 3 different data structures:

1. Queues
2. Vectors
3. Priority-Queues

I have used **A* algorithm**. In my implementation, I have used 3 different classes and 1 main class ***(making 4 in total)***.

1. board.cpp
2. multithreading.cpp
3. queues.cpp
4. main.cpp

We start off by creating a randomized board which is **4x4**. For this permutation was used which initially initializes a board with 0-15 values sequentially and then each entry is swapped with another randomized entry to make it a random board. After the implementation of board, the A* algorithm is implemented in which we are using a hash function. The purpose of this hash function is to randomly create a board combination after each step and checks its cost, the one with the lowest cost is selected and rest are stored in hash function. Then the hash function ensures that no combination is revisited. Similarly, we implemented a priority-queue and then we created another class of queue which has a queue of priority queues. Then we implemented multithreading in which user passed the number of threads as an argument and then the generic function takes the argument as a parameter and initialized that many threads which later on perform the desired task.

To run the file, follow the following steps:

- **g++ main.cpp -pthread -o main**
- **./main n** *{here **"n"** is the number of parameters i.e., 1,2,3,4,……..,n}*

It will then display a menu as follows:

Do you want to run a test file?

1: Yes

2: No

Choose your option =

(If you choose 1, it will ask you to enter the filename (including extension) whereas if you choose 2, it will randomly generate a board.)

As my workspace is core i5 10th generation, it has ***4 cores*** and ***8 threads***. So, I conducted test on each file 8 times using threads from 1-8.

➢ Benchmark:

Test # 1:

In this test, I created a file named **"easy.txt"**. The basic purpose of this test was to create a file with easy combination of the puzzle and see how effective multi-threading works. The combination that I used in "easy.txt" was:

9	8	11	
6	1	10	13
3	14	2	4
15	12	7	5

The time taken to execute the program is as follows:

No of threads	Time Taken (ms)
1	4431 (4.43s)
2	4422 (4.42s)
3	3833 (3.83s)
4	3763 (3.76s)
5	3704 (3.7s)
6	3284 (3.28s)
7	3236 (3.23s)
8	3158 (3.16s)

Test # 2:

In this test, I created a file named **"med.txt"**. The basic purpose of this test was to create a file with a bit tougher combination of the puzzle than the "easy.txt" and see how effective multi-threading works. The combination that I used in "med.txt" was:

8	14	2	7
3	15		6
1	11	13	12
4	9	5	10

The time taken to execute the program is as follows:

No of threads	Time Taken (ms)
1	15578 (15.58s)
2	15308 (15.31s)
3	14083 (14.08s)
4	12946 (12.95s)
5	12271 (12.27s)
6	12020 (12.02s)
7	11730 (11.73s)
8	10597 (10.59s)

Test # 3:

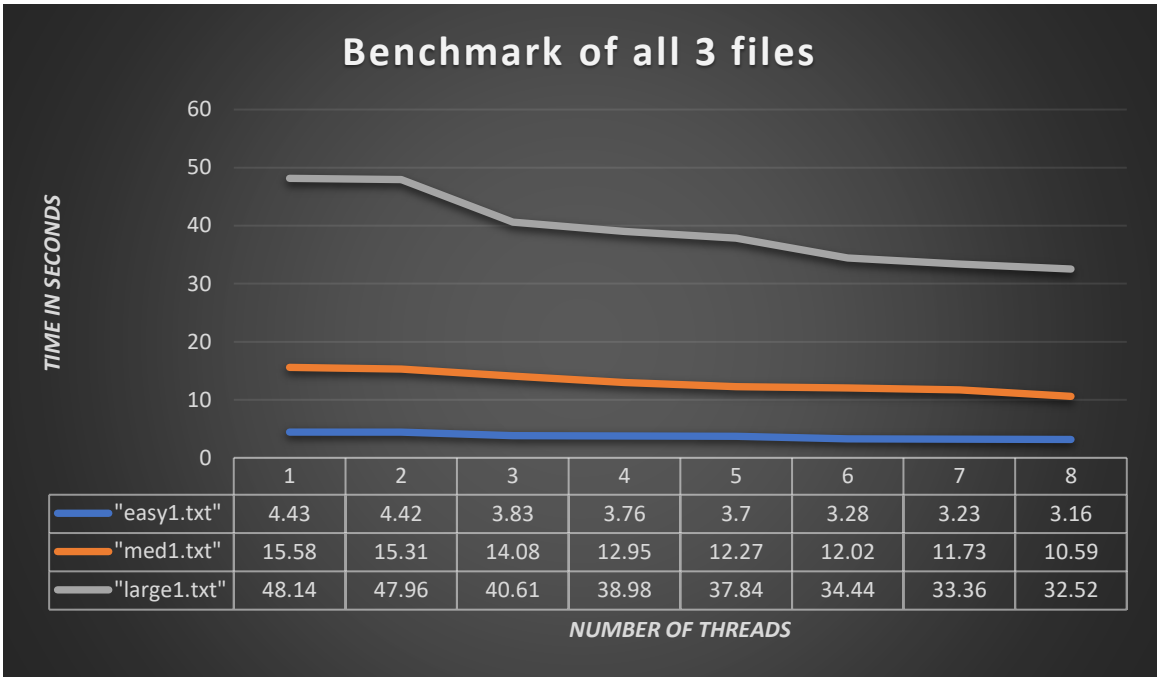
In this test, I created a file named **"hard.txt"**. The basic purpose of this test was to create a file with a really hard combination of the puzzle and see how effective multi-threading works. The combination that I used in "hard.txt" was:

10	3	6	7
12	2	9	11
14	1	13	8
4		15	5

The time taken to execute the program is as follows:

No of threads	Time Taken (ms)
1	48138 (48.14s)
2	47969 (47.96s)
3	40608 (40.61s)
4	38980 (38.98s)
5	37841 (37.84s)
6	34438 (34.44s)
7	33357 (33.36s)
8	32524 (32.52s)

The graph of the above-mentioned tests is:



	1	2	3	4	5	6	7	8
"easy1.txt"	4.43	4.42	3.83	3.76	3.7	3.28	3.23	3.16
"med1.txt"	15.58	15.31	14.08	12.95	12.27	12.02	11.73	10.59
"large1.txt"	48.14	47.96	40.61	38.98	37.84	34.44	33.36	32.52

➢ Conclusion:

As can be seen from the above-mentioned tests and graph, the y-axis has time that has been converted into seconds from milliseconds for ease. As can be seen in the above-mentioned graph, if we keep on increasing the number of threads, a great difference is seen in the efficiency of the program. With more threads, program becomes more efficient and the puzzle is solved in shorter period of time. This also proves that multi-threading has been implemented perfectly. As thread count increases, the execution time decreases. We can see a **29% improvement** in execution speed as we move from 1 thread to 8 threads in file ***"easy.txt"***. Similarly, we can see a **32% improvement** in execution speed as we move from 1 thread to 8 threads in file ***"med.txt"*** and We can see a **33% improvement** in execution speed as we move from 1 thread to 8 threads in file ***"hard.txt"***.

Hence, we can say that:

$$\text{Number of threads} \quad \alpha \quad \frac{1}{\text{time to execute}}$$

PHASE 3 Analysis

In Phase 3, I decided to modify my implementation. Initially, each thread was being given a priority queue which helps it in implementation of A* algorithm. I kept thinking, researching and came up with different ideas to make our program even more efficient. I thought of using ***Dijkstra Algorithm*** to implement a better solution but then suddenly something clicked my mind and an idea just emerged. I thought to myself that if a mall owner has 10 employees. The work load will be distributed and the time taken to execute their jobs will be reduced but what if each of the 10 employees is given further 3 assistants too? Then the time taken would be even minimized.

Let's take an example of a bank. A bank has top-level managers, but to distribute the workload even more efficiently there are further area managers, branch managers and then each branch manager further has employees to assign task too. This way the work load is also divided along with time efficiency.

After analyzing the above scenarios, I decided that instead of giving each thread a fixed number of queues, why shouldn't I give them more than 1 queues each? After detailed experimentation and analysis, I came to the conclusion that the more the number of priority-queues each thread has, the lesser the time it will take to execute the program. Hence, for this implementation, I am taking the number of queues per thread from the user and then I'm assigning that number of queues to each thread and the result turned out to be amazingly better both in terms of performance and time efficiency.

To run the file, follow the following steps:

- g++ main.cpp -pthread -o main
- ./main n *{here **"n"** is the number of parameters i.e., 1,2,3,4,……..,n}*

It will then display a menu as follows:

Do you want to run a test file?

1: Yes

2: No

Choose your option =

(If you choose 1, it will ask you to enter the filename (including extension) whereas if you choose 2, it will randomly generate a board.)

Then it will display:

Enter the number of queues per thread =

As my workspace is core i5 10th generation, it has ***4 cores*** and ***8 threads***. So, I conducted test on each file 8 times using threads from 1-8.

➢ Benchmark:

Test # 1:

In this test, I created a file named **"easy.txt"**. The basic purpose of this test was to create a file with easy combination of the puzzle and see how effective multi-threading works. The combination that I used in "easy.txt" was:

9	8	11	
6	1	10	13
3	14	2	4
15	12	7	5

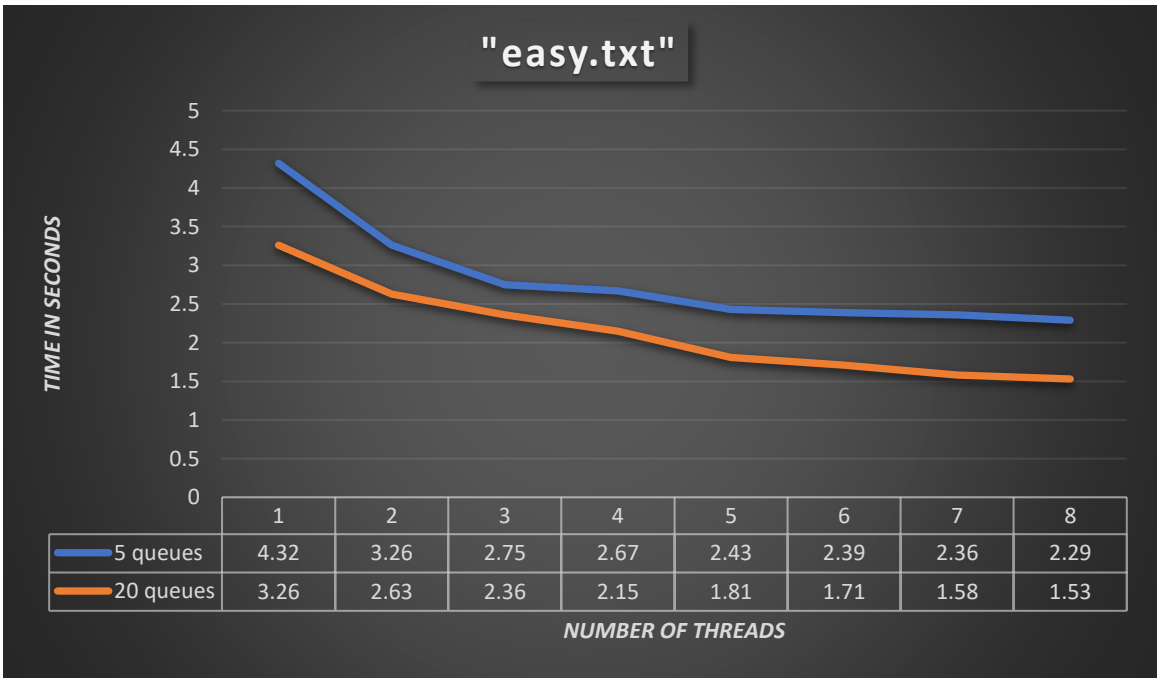
The time taken to execute the program with 5 queues per thread is as follows:

No of threads	Queues per Thread	Time Taken (ms)
1	5	4321 (4.32s)
2	5	3264 (3.26s)
3	5	2759 (2.75s)
4	5	2676 (2.67s)
5	5	2435 (2.43s)
6	5	2397 (2.39s)
7	5	2368 (2.36s)
8	5	2298 (2.29s)

The time taken to execute the program with 20 queues per thread is as follows:

No of threads	Queues per Thread	Time Taken (ms)
1	20	3264 (3.26s)
2	20	2631 (2.63s)
3	20	2362 (2.36s)
4	20	2150 (2.15s)
5	20	1817 (1.81s)
6	20	1712 (1.71s)
7	20	1585 (1.58s)
8	20	1532 (1.53s)

The graph of this test is as follows:



We can see how increasing the number of queues per thread decreases the time of execution even further.

Test # 2:

In this test, I created a file named **"med.txt"**. The basic purpose of this test was to create a file with a bit tougher combination of the puzzle than the "easy.txt" and see how effective multi-threading works. The combination that I used in "med.txt" was:

8	14	2	7
3	15		6
1	11	13	12
4	9	5	10

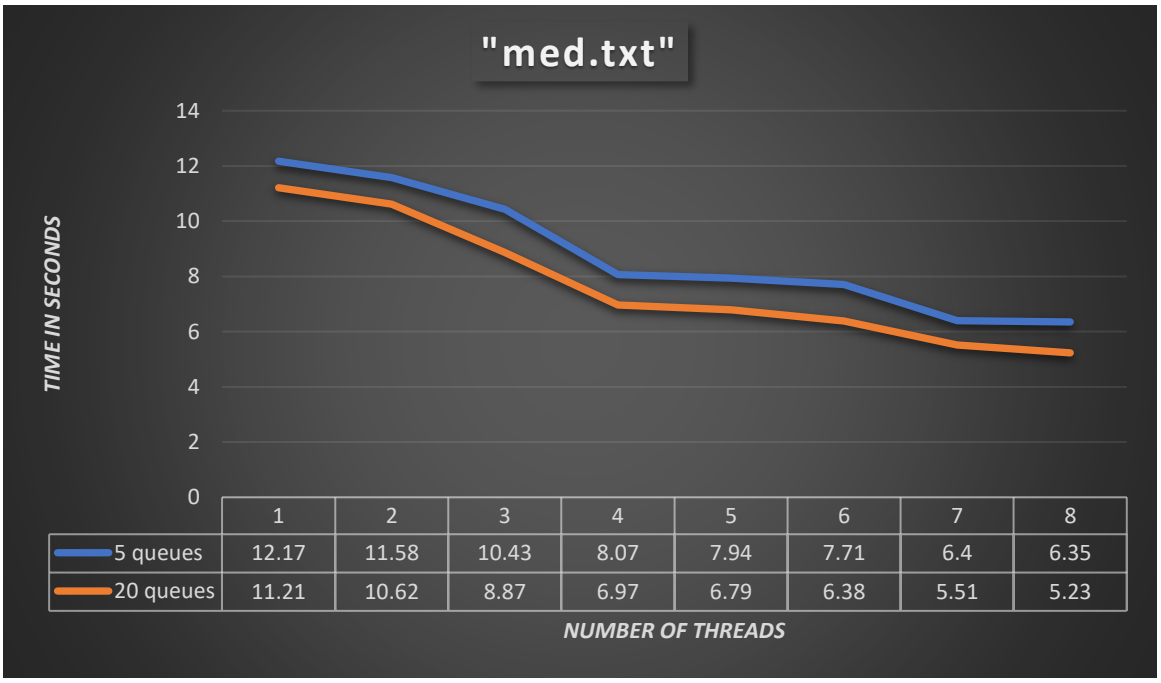
The time taken to execute the program with 5 queues per thread is as follows:

No of threads	Queues per Thread	Time Taken (ms)
1	5	12170 (12.17s)
2	5	11582 (11.58s)
3	5	10439 (10.43s)
4	5	8070 (8.07s)
5	5	7949 (7.94s)
6	5	7714 (7.71s)
7	5	6405 (6.40s)
8	5	6352 (6.35s)

The time taken to execute the program with 20 queues per thread is as follows:

No of threads	Queues per Thread	Time Taken (ms)
1	20	11210 (11.21s)
2	20	10622 (10.62s)
3	20	8870 (8.87s)
4	20	6979 (6.97s)
5	20	6799 (6.79s)
6	20	6389 (6.38s)
7	20	5519 (5.51s)
8	20	5236 (5.23s)

The graph of this test is as follows:



We can see how increasing the number of queues per thread decreases the time of execution even further.

Test # 3:

In this test, I created a file named **"hard.txt"**. The basic purpose of this test was to create a file with a really hard combination of the puzzle and see how effective multi-threading works. The combination that I used in "hard.txt" was:

10	3	6	7
12	2	9	11
14	1	13	8
4		15	5

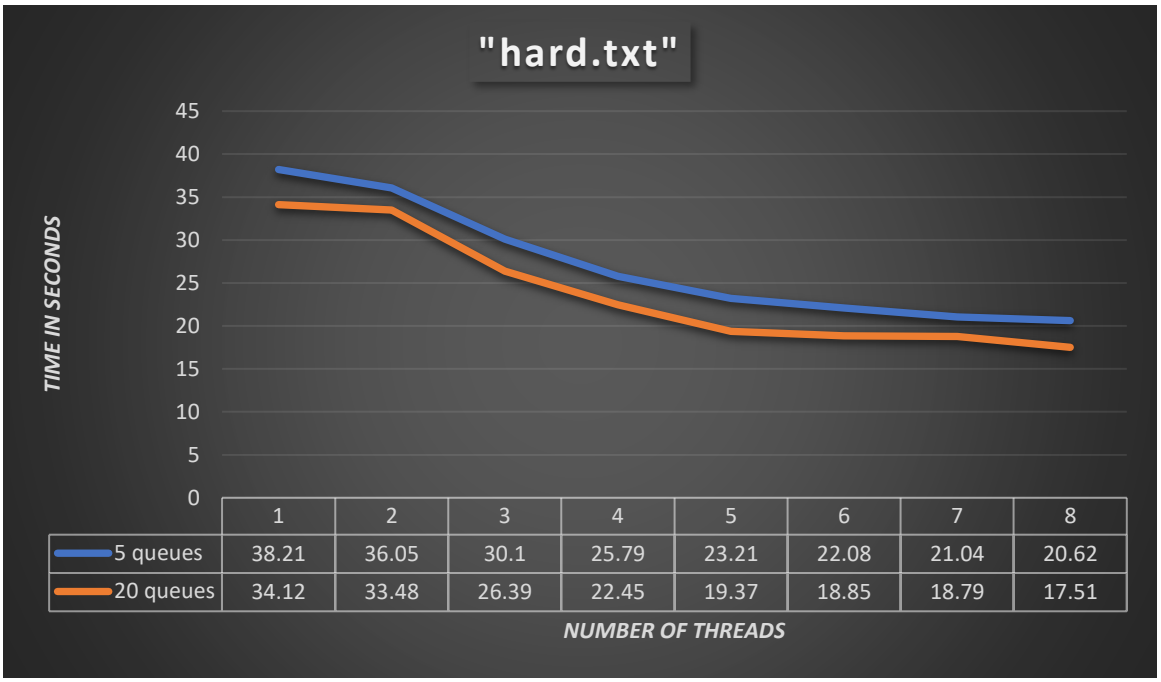
The time taken to execute the program with 5 queues per thread is as follows:

No of threads	Queues per Thread	Time Taken (ms)
1	5	38213 (38.21s)
2	5	36052 (36.05s)
3	5	30105 (30.10s)
4	5	25792 (25.79s)
5	5	23216 (23.21s)
6	5	22083 (22.08s)
7	5	21042 (21.04s)
8	5	20623 (20.62s)

The time taken to execute the program with 20 queues per thread is as follows:

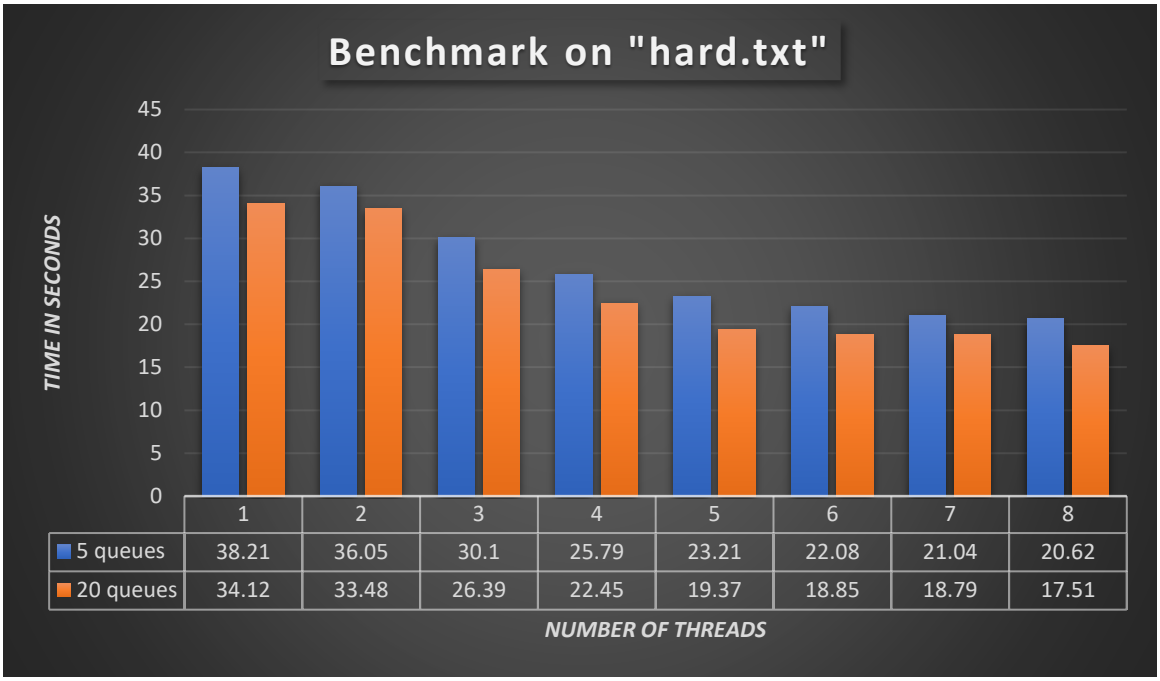
No of threads	Queues per Thread	Time Taken (ms)
1	20	34127 (34.12s)
2	20	33489 (33.48s)
3	20	26399 (26.39s)
4	20	22455 (22.45s)
5	20	19337 (19.37s)
6	20	18856 (18.85s)
7	20	18793 (18.79s)
8	20	17510 (17.51s)

The graph of this test is as follows:



We can see how increasing the number of queues per thread decreases the time of execution even further.

Now we conclude our testing by a final analysis of the file “hard.txt”:



	1	2	3	4	5	6	7	8
5 queues	38.21	36.05	30.1	25.79	23.21	22.08	21.04	20.62
20 queues	34.12	33.48	26.39	22.45	19.37	18.85	18.79	17.51

➢ Conclusion:

As can be seen from all of the tests performed above and all of the statistics and graphs mentioned above, we can see that as we increase the number of queues per thread, we see a significant decrease in the execution time of the program. Similarly, after side-by-side analysis of ***“hard.txt”,*** we analyzed that increasing the number of queues per thread from *5 to 20* gives us a significant 15% decrease in time of execution with efficient performance.

Similarly, increasing the number of queues per thread from *1 to 20* gives us a significant 46.2% decrease in the time of execution with efficient performance.

Hence, we can say that:

$$\text{Number of queues per thread} \quad \alpha \quad \frac{1}{\text{time to execute}}$$

THE END ☺