CS5300: Parallel and Concurrent Programming
Final Project

- **CS18BTECH11001**
- **CS18BTECH11016**

PLAGIARISM STATEMENT

I certify that this assignment/report is my own work, based on my personal study and/or research and that I have acknowledged all material and sources used in its preparation, whether they be books, articles, reports, lecture notes, and any other kind of document, electronic or personal communication. I also certify that this assignment/report has not previously been submitted for assessment in any other course, except where specific permission has been granted from all course instructors involved, or at any other time in this course, and that I have not copied in part or whole or otherwise plagiarised the work of other students and/or persons. I pledge to uphold the principles of honesty and responsibility at CSE@IITH. In addition, I understand my responsibility to report honor violations by other students if I become aware of it.

Name: A. Venkata Sai Mahesh, J. Deepak Reddy

Date: 23/12/2020

Signature: A V S Mahesh, J Deepak Reddy

TT(Time Tampering)-Lock:

TT-Lock (Time Tampering lock) is a lock mechanism with FCFS property based on the CPU time in which they are requesting to access the critical section of a specific program. Each thread writes its time of request for the critical section that is a timestamp in a priority queue with an earlier timestamp prior to later timestamp. We are being precise in FCFS by considering time stamps because we believe that every program will differ in nanoseconds of execution time and the corresponding relative order of execution of threads affects the expected output in an unexpected way. In this lock execution mechanism, the threads will be given access to the critical section based on the order of time stamps corresponding to relative threads in the priority queue. We are using priority queue because there may be a situation where the threads may request the access to critical section earlier but failed to write into the queue subsequently, So we intended to use priority queue which ensures that the threads with earlier timestamps are given more priority when compared with the threads with the later time stamps, in turn, ensuring the FCFS property.

Program Design:

Main Method:

1. In **main()** function, function **initialisation()** which is used to initialize the required data variables is called and also the threads were initialized to perform **testalgo()** function.
2. Computation regarding total average CS enter times and exit times were also performed in this function.

Initialization Method:

1. Data variables n(total no. of threads) and k(no. of times a thread can request lock) are read from the file inp-params.txt.
2. The files **output.txt**(to output the activity of the threads with a time-stamp), **Avg-enter_times.txt**(to output average time taken to enter CS by each thread), **Avg-exit_times.txt**(to output average time taken to exit CS by each thread), **clock_order.txt**(to output the CS request order of the threads based on the time-stamp) are opened in "write" mode.

Testalgo Method:

1. In this function, each thread will call the functions **lok()** and **unlok()** to acquire and release lock correspondingly and the in-between times of these requests of locks and unlock were noted for further study.
2. Each thread will print its activity(requesting to enter CS, entering to CS, requesting to exit CS, exiting CS) in the file **output.txt,** so that the user can confirm whether the lock is satisfying mutual exclusion or not.
3. Between lok() and unlok() calls, a thread will be forced to go to sleep for 1sec, so that the user can easily interpret the activity of the threads from the file **output.txt**.

Lok Method:

In this method, each thread requesting for the lock must wait in the priority queue(with the priority of lowest time-stamp). To complete this operation **map** data structure is used. Here time-stamps of the threads are mapped

to the thread-id. The map data structure ensures that these time-stamps when popped would be in ascending order. Internally **map** uses a red-black tree, and these time-stamps are treated as nodes to this tree. This **map** data-structure abstracts the complexity and gives away the feeling of the priority queue and satisfies our requirement of implementation and execution of priority queue (where we cannot map time stamps with corresponding thread-ids). To ensure that these time-stamps are added atomically into the **map**, a tree-based synchronization primitive is used. Now to check whether current thread is the head of the priority queue or not, we compared thread id and the id which has been mapped by the lowest time-stamp of the queue containing the time of CS requests of the threads. If the ids are equal then the thread is allowed to acquire the lock else the threads must wait until the time of request for the lock is least in the current waiting queue.

Unlok Method:

In **unlok()** method, each thread writes its time of request for the lock in the file **clock_order.txt** so that user can interpret and can conclude that the order of the statements printed must be in the order of the time of request for the lock which is the main motive of this lock(**FCFS** property). To release the lock, a thread must remove its node from the queue so that the thread with next least time-stamp can access the critical section.

Observations:

output.txt

```
1   1st CS Entry Request at 16:59:37 by thread 1
2   1st CS Entry at 16:59:37 by thread 1
3   1st CS Entry Request at 16:59:37 by thread 2
4   1st CS Entry Request at 16:59:37 by thread 10
5   1st CS Entry Request at 16:59:37 by thread 9
6   1st CS Entry Request at 16:59:37 by thread 3
7   1st CS Entry Request at 16:59:37 by thread 8
8   1st CS Entry Request at 16:59:37 by thread 7
9   1st CS Entry Request at 16:59:37 by thread 5
10  1st CS Entry Request at 16:59:37 by thread 6
11  1st CS Entry Request at 16:59:37 by thread 4
12  1st CS Exit Request at 16:59:38 by thread 1
13  1st CS Entry at 16:59:38 by thread 2
14  1st CS Exit at 16:59:38 by thread 1
15  1st CS Exit Request at 16:59:39 by thread 2
16  2nd CS Entry Request at 16:59:39 by thread 1
17  1st CS Exit at 16:59:39 by thread 2
18  1st CS Entry at 16:59:39 by thread 10
19  2nd CS Entry Request at 16:59:40 by thread 2
20  1st CS Exit Request at 16:59:40 by thread 10
21  1st CS Exit at 16:59:40 by thread 10
22  1st CS Entry at 16:59:40 by thread 9
23  2nd CS Entry Request at 16:59:41 by thread 10
```

Fig1: A snapshot of **output.txt**

As one can see that there are no CS enter messages by other threads between the CS enter and CS exit request message by a thread which indicates that this lock algorithm ensures the **mutual exclusion** property and also one can observe that every thread is not given access to the lock right away after requesting which suggests that this lock is **not wait-free**. The **while** loop in the **lok()** function suggests that the threads may **spin** while waiting for the lock to be released. If the thread which entered the CS

is not holding the CPU or died for whatever the reason, then the other threads cannot acquire the lock, this suggests that this algorithm is **blocking**.

Avg-enter_times.txt

```
1   The average CS entry-time of thread-1 = 16396634Us
2   The average CS entry-time of thread-2 = 16929641Us
3   The average CS entry-time of thread-3 = 18333812Us
4   The average CS entry-time of thread-4 = 20769287Us
5   The average CS entry-time of thread-5 = 19814842Us
6   The average CS entry-time of thread-6 = 20304068Us
7   The average CS entry-time of thread-7 = 19352813Us
8   The average CS entry-time of thread-8 = 18796612Us
9   The average CS entry-time of thread-9 = 17897230Us
10  The average CS entry-time of thread-10 = 17443933Us
11
12  The total average CS entry-time = 18603887Us
```

Fig2: A snapshot of **Avg-enter_times.txt**

Avg-exit_times.txt

```
1   The average CS exit-time of thread-1 = 746Us
2   The average CS exit-time of thread-2 = 733Us
3   The average CS exit-time of thread-3 = 174Us
4   The average CS exit-time of thread-4 = 197Us
5   The average CS exit-time of thread-5 = 627Us
6   The average CS exit-time of thread-6 = 363Us
7   The average CS exit-time of thread-7 = 102Us
8   The average CS exit-time of thread-8 = 231Us
9   The average CS exit-time of thread-9 = 522Us
10  The average CS exit-time of thread-10 = 2174Us
11
12  The total average CS exit-time = 586Us
```

Fig3: A snapshot of **Avg-exit_times.txt**

clock_order.txt

```
1   The thread-1 requested to entry for CS at 3199
2   The thread-2 requested to entry for CS at 3308
3   The thread-10 requested to entry for CS at 3714
4   The thread-9 requested to entry for CS at 4071
5   The thread-3 requested to entry for CS at 5283
6   The thread-8 requested to entry for CS at 8071
7   The thread-7 requested to entry for CS at 9815
8   The thread-5 requested to entry for CS at 10350
9   The thread-6 requested to entry for CS at 19824
10  The thread-4 requested to entry for CS at 20379
11  The thread-1 requested to entry for CS at 1024718
12  The thread-2 requested to entry for CS at 1526185
13  The thread-10 requested to entry for CS at 2023218
14  The thread-9 requested to entry for CS at 2537038
15  The thread-3 requested to entry for CS at 3049502
16  The thread-8 requested to entry for CS at 3568025
17  The thread-7 requested to entry for CS at 4067070
18  The thread-5 requested to entry for CS at 4597622
19  The thread-6 requested to entry for CS at 5102665
20  The thread-4 requested to entry for CS at 5662989
21  The thread-1 requested to entry for CS at 6143304
22  The thread-2 requested to entry for CS at 6641680
23  The thread-10 requested to entry for CS at 7154468
24  The thread-9 requested to entry for CS at 7649024
25  The thread-3 requested to entry for CS at 8167133
```

Fig4: A snapshot of **clock_order.txt**

As one can see in **Fig4** that the threads were given access to the critical section in the order of their time of request for the access of the critical section. These messages are printed into the **clock_order.txt** in the **unlok()** function which means the thread still has access to the lock and printed its time of request for access to the critical section which proves property of **FCFS** precisely based on the time-stamps that is a time of

request for the critical section which is the main of the implementation of this lock algorithm.

Conclusion:

It can be concluded that the CS enter-times and CS exit-times increase as the value of **n increases**(**k** is constant). This lock isn’t meant to be an efficient locking algorithm but is designed to ensure that the thread with the least time of request for the critical section is to be given access to the critical section first.

Parallel Implementation of MST:

Problem Statement:

Finding Minimum Spanning Tree(spanning tree of minimum total weight) for the given undirected weighted connected graph using Multiprocessing.

Implementation:

I have implemented a parallel version of Prim's algorithm. The main feature for selecting this algorithm is that it uses the Cut Property of MST.

Cut Property:

Group of edges that connects two sets of vertices in a graph is called a Cut of graph.

Sequential Prim's Algorithm:

1. Create a set *from* that keeps track of vertices already included in MST.
2. Assign a *key* value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
3. While *from* doesn't include all vertices
    a. Pick a vertex u which is not there in from and has minimum key value(cut property).
    b. Include u to mstSet.
    c. Update key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v

Parallel Prim's Algorithm:

1. Prim’s Algorithm can be parallelised in step 3a(to find vertex with minimum key value) by using each thread to find the minimum weight for a part of the graph and finally finding the minimum of all these parts of the graph taken by each thread. The finding minimum of all parts of graph calculated by each thread involves an access for global variable *min* which therefore should be enclosed in a critical section.
2. We even parallelize the step 3c where each thread will update the distance of part of neighbours of vertex u(each thread will take a part of vertex iteration)

Program Design(Primes.cpp):

Main Method:

1. Consider the adjacency matrix either inputted from the user or selected randomly as per the second argument in the *input_params.txt*
2. Set the number of threads to be used by the OpenMp with the input entered by the user.
3. Note the time taken by running the sequential program and parallel program and print the output of both these algorithms in *sequential_MST.txt* and *parallel_MST.txt* respectively.
4. Print the weight of the minimum spanning tree constructed by the Sequential and Parallel algorithm that is returned from the *printMST()* to check the correctness.

PrimMSTPar Method:

1. Consider 2 sets - *from* and *visited* to keep track of vertices already included in the MST and not included in the MST respectively.
2. Consider set - *key* to maintain minimum weight of edge to each vertex from the cut graph.
3. Consider 0^{th} vertex as root and add it into the MST along with updating its key value to 0.
4. Consider a for loop of V-1 iterations(where one vertex will be added into the MST and as 0^{th} vertex is already added)
5. Each time find the vertex with minimum key value using the *minKeyPar()* method and remove it from the set that maintains vertices not included in MST.
6. Now parallelise the updation of key values of the neighbours of the above calculated minimum key value vertex using OpenMP for loop which divides the execution to be executed by multiple threads.
7. Finally return the thus formed vertices in MST.

MinKeyPar Method:

1. Find the minimum key value of each part using the parallel implementation of OpenMP with the inputted number of threads. This part uses two local variables that store the minimum key value and the index containing it respectively. As the threads don't involve any shared memory(global variables) in this implementation it is not involved in the critical section.
2. Now to find the minimum of all the local minimums calculated above. Consider a shared variable *min* to be updated by each thread after

checking it with their local minimum key values. As this involves shared variables this should be enclosed in a critical section part of OpenMP.

3. Finally return the minimum key index.

PrintMST Method:

1. This method will print the MST formed in both Sequential and Parallel implementation using the File IO into different files.
2. This method will calculate the weight of the thus formed MST and return it.

Observations:

Difference in outputs of Sequential and Parallel Executions:

We find that the output files **sequential_MST.txt** and *parallel_MST.txt* may not be equal. These files are equal if there exists only one MST for the given graph. But if there are multiple MST's for the given graph, the finding of minimum key value index may consider other different edges with the same minimum weight(when there are multiple edges of minimum weight) while using threads(as the order is not maintained - not sequential). So while uncommenting and running the **diff** command in the **run.sh,** it wouldn't show any difference when we can ensure that there is only one MST.

Correctness:

We couldn't guarantee the correctness of the result obtained by Parallel implementation by comparing the edges of MST with the result of Sequential implementation due to the above reason. But we could able to

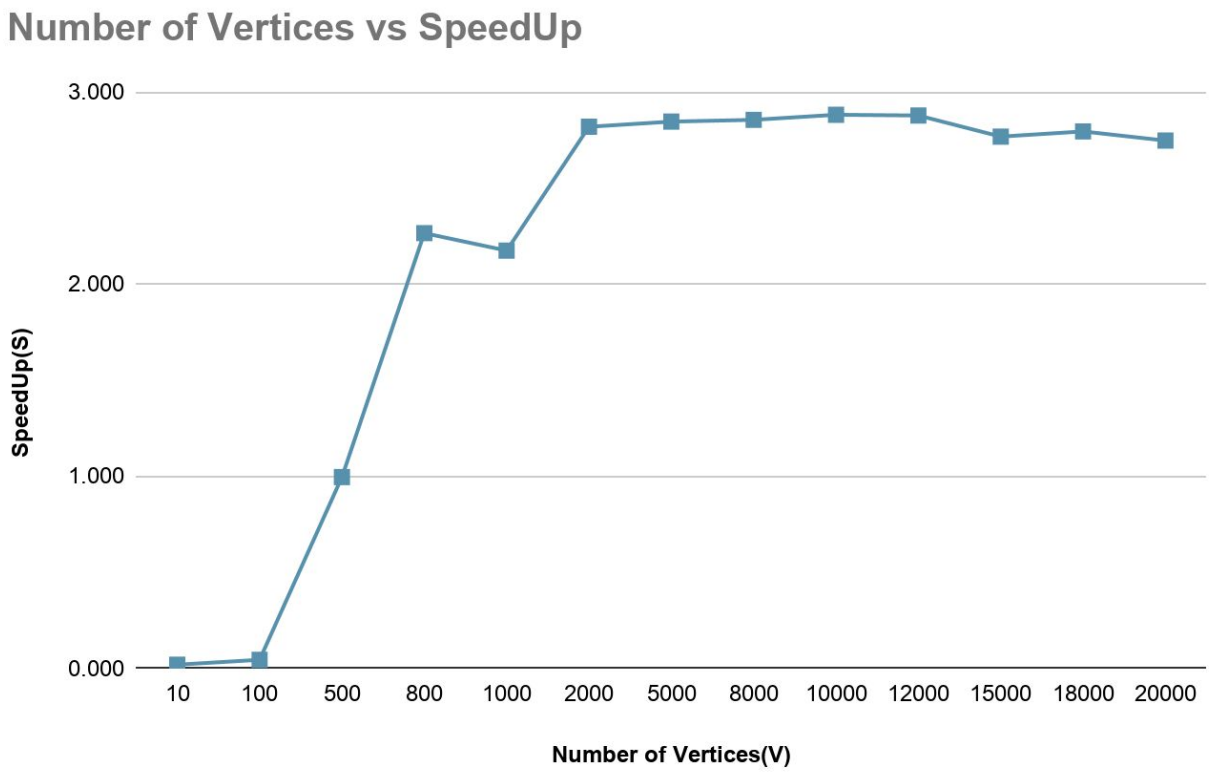
check that the weight of the whole MST must be the same for all MST's which are printed onto the terminal. We can observe that both the values for sequential and parallel execution are equal and thus proved that our algorithm is correct.

Performance of Parallel MST Algorithm:

1. Keeping Number of Threads Constant(N = 10)

Data Values:

Number of Vertices(V)	Time for Sequential Execution (tseq)	Time for Parallel Execution (tpar)	SpeedUp due to parallelisation(S)
10	7 us	369 us	0.019 x
100	216 us	4860 us	0.044 x
500	4245 us	4264 us	0.995 x
800	13221 us	5837 us	2.265 x
1000	17348 us	7979 us	2.174 x
2000	66150 us	23468 us	2.818 x
5000	341430 us	119987 us	2.845 x
8000	833496 us	291977 us	2.854 x
10000	1283824 us	445587us	2.881 x
12000	1889787 us	656792 us	2.877 x
15000	2823357 us	1092454 us	2.767 x
18000	4279823 us	1531600 us	2.794 x
20000	5302587 us	1930461 us	2.746 x



Analysis:

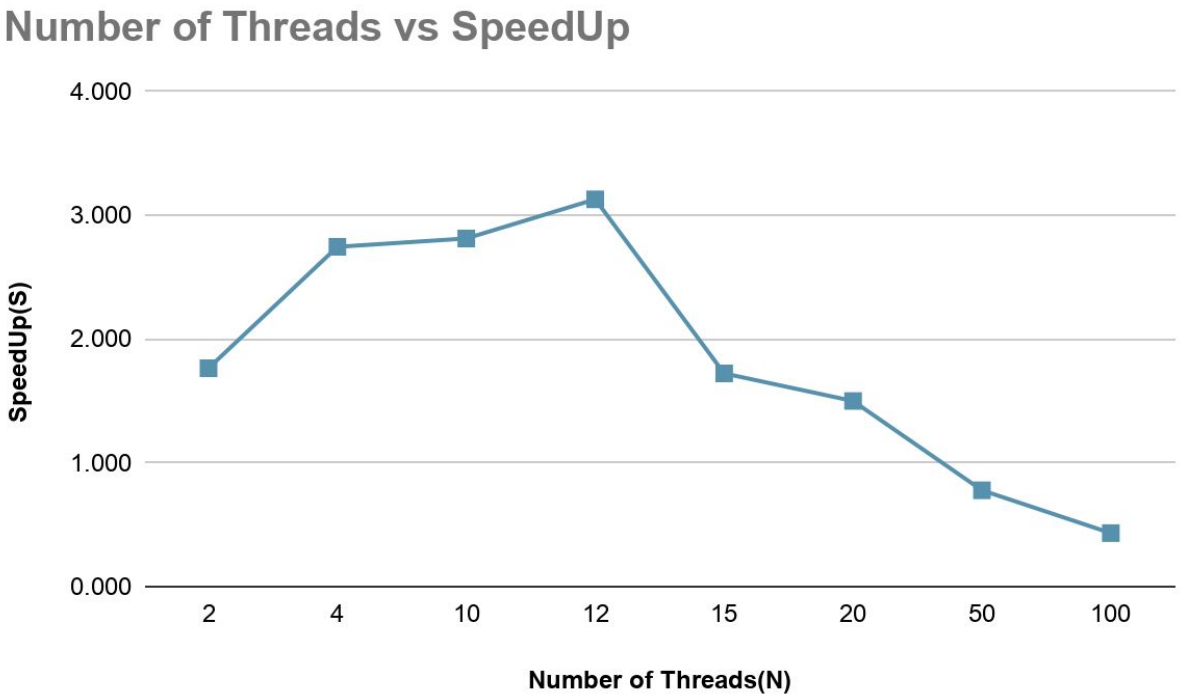
We can observe that the speedup is less than zero for the lesser number of Vertices(V). This means that the sequential method is taking less than the parallelised method. This is because the context switch between the threads is dominating than the execution time for the given input.

We can also observe that the speedUp remains nearly constant(~2.8x) for the higher values of input. This is due to the balancing of work load time with the context switch time and from the graph we can also say that the maximum speed up achieved by our Parallel MST algorithm is 2.8x.

2. Keeping Number of Vertices Constant(V = 20000)

Data Values:

Number of Threads(N)	Time for Sequential Execution (tseq)	Time for Parallel Execution (tpar)	SpeedUp due to parallelisation(S)
2	5302587 us	3004317 us	1.765 x
4	5302587 us	1931330 us	2.746 x
10	5302587 us	1884959 us	2.813 x
12	5302587 us	1694336 us	3.130 x
15	5302587 us	3077781 us	1.723 x
20	5302587 us	3535145 us	1.500 x
50	5302587 us	6810107 us	0.779 x
100	5302587 us	12260544 us	0.432 x



Analysis:

We can see that initially as the number of threads increases, the speedUp is also increased and reaches maximum at N=12 with a speed up of ~3.1x. After that the SpeedUp is continuously decreasing. This is due to the frequent context switching between threads that is dominating the execution time of the given input.

Conclusion:

By Using this Parallel implementation of MST, we could achieve a
Maximum speedUp = 3.8 x
Optimal number of threads = 12

References:

1. https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/
2. https://github.com/parthvshah/parallel-prims

THE END