CS5300: Parallel and Concurrent Programming

Final Project

- RISHABH JAIN : CS23MTECH12007
CHANDRA PRAKASH : CO21BTECH11003
ANKIT MALIK : MS23MTECH11003

Parallel Implementation of MST(PRIMS):

Problem Statement:
The challenge is to discover the Minimum Spanning Tree, which is a spanning tree with the least total weight, for a provided undirected weighted connected graph using multiprocessing.

Implementation:
I've developed a parallelized iteration of Prim's algorithm. The primary reason for choosing this algorithm lies in its utilization of the Cut Property of MST. (A set of edges linking two vertex sets in a graph is termed a Cut of the graph)
Sequential Prim's Algorithm:

1. Form a set to keep track of vertices already incorporated into the Minimum Spanning Tree (MST).
2. Assign key values to all vertices in the given graph, initializing them all as INFINITE. Set the key value of the first vertex to 0, ensuring its selection as

the initial pick.

3. While the set 'from' does not encompass all vertices:

   a. Select a vertex 'u' that is not yet included in 'from' and possesses the minimum key value (based on the cut property).

   b. Add 'u' to the MST set.

   c. Revise the key values of all adjacent vertices to 'u'. To update these key values, iterate through all adjacent vertices. For each adjacent vertex 'v', if the weight of the edge 'u-v' is less than the previous key value of 'v', update the key value to the weight of 'u-v'

Parallel Prim's Algorithm:

Parallelization of Prim's Algorithm involves distributing the task of identifying the vertex with the minimum key value among multiple threads.

Each thread is responsible for determining the minimum weight for a specific segment of the graph. The final step encompasses finding the overall minimum among the results obtained by each thread.

It is essential to note that the process of determining the minimum from all segments requires accessing the global variable 'min' and should, therefore, be enclosed within a critical section.

The parallelization extends to the phase where each thread concurrently updates the distances of neighboring vertices for a specific portion of vertex iteration related to vertex 'u.'

Program Design(Primes1.cpp/ Primes2.cpp□vertices constant):

Primary Procedure:

1. Examine the adjacency matrix, either provided by the user or chosen randomly based on the second argument.
2. Configure the number of threads for OpenMP using the user-provided input.
3. Record and display the runtime of both the sequential and parallel programs, presenting the outcomes in sequential_MST and parallel_MST correspondingly.
4. Display the weight of the minimum spanning tree generated by the Sequential and Parallel algorithms, ensuring correctness by checking the output from the printMST() function.

PrimMSTPar Method:

1. Utilize two sets, "from" and "visited," to monitor vertices already incorporated and those pending inclusion in the Minimum Spanning Tree (MST) respectively.

2. Employ the "key" set to manage the minimum edge weight to each vertex from the segmented graph.

3. Designate the 0th vertex as the root, adding it to the MST while updating its key value to 0.

4. Implement a for loop with V-1 iterations (as the 0th vertex is already included), adding one vertex to the MST in each iteration.

5. Identify the vertex with the minimum key value through the minKeyPar() method and remove it from the set tracking vertices not yet included in the MST.

6. Parallelize the update of key values for the neighbors of the aforementioned vertex with the minimum key value using an OpenMP for loop, distributing the execution across multiple threads.

7. Ultimately, return the vertices forming the resultant MST.

MinKeyPar Method:

1. Employ the parallel implementation of OpenMP to identify the minimum key value for each part, utilizing the specified number of threads. This step utilizes two local variables that store the minimum key value and its corresponding index. Since this implementation does not involve shared memory (global variables), there is no critical section for the threads.

2. Determine the overall minimum among the local minimums obtained in the previous step. Introduce a shared variable, "min," which each thread updates after comparing it with its local minimum key value. Due to the involvement of shared variables, this process must be enclosed in a critical section of the OpenMP framework.

3. Conclude the process by returning the index of the minimum key.

MSTPrint Method:

1. This function is designed to display the Minimum Spanning Tree (MST) generated through both Sequential and Parallel implementations, saving the results using File I/O into distinct files.

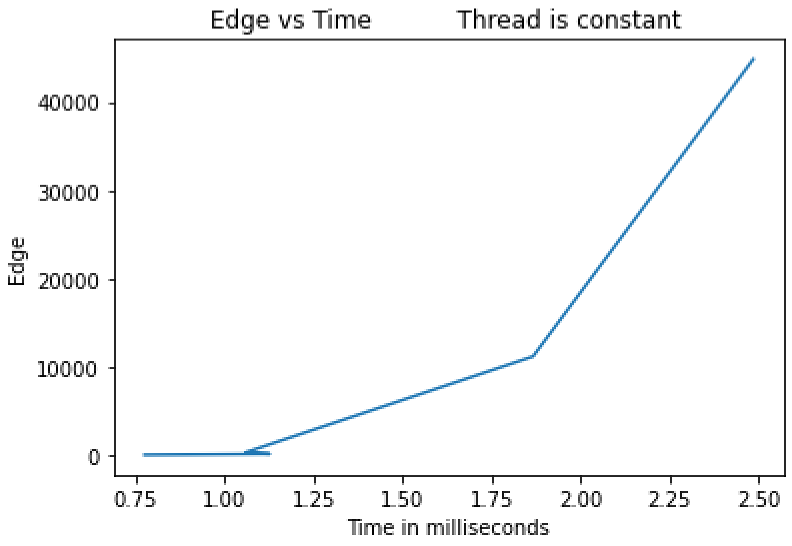
2. Additionally, this method computes and returns the total weight of the generated MST.

Performance of Parallel MST Algorithm:

1. Keeping Number of Threads Constant = 8

Data Values:

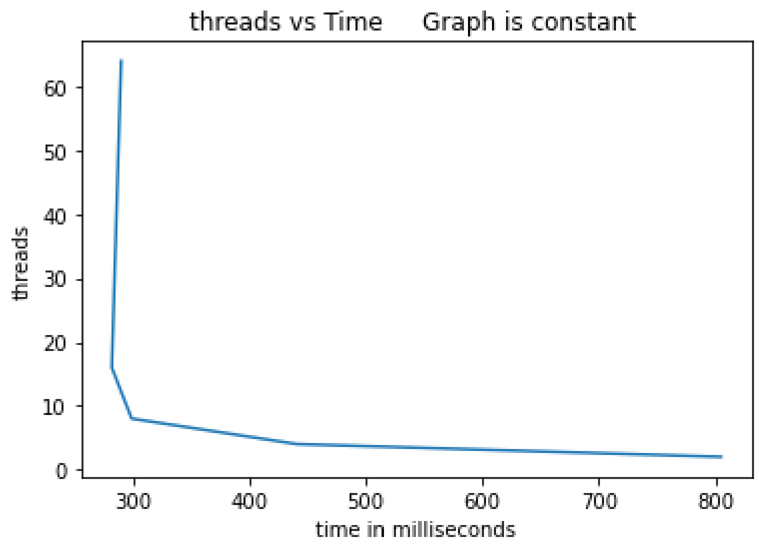
Number of Vertices(V)/ **Number of Edges(E)= num_edges = V * (V - 1) / 2**	Time for Parallel Execution milliseconds
5,10	0.776299999
16,120	1.1272
25,300	1.0602
150,11175	1.8663
200,44850	2.4844


Edge vs Time
Thread is constant
Edge
0
10000
20000
30000
40000
0.75
1.00
1.25
1.50
1.75
2.00
2.25
2.50
Time in milliseconds

Keeping Number of Vertices Constant(V = 100)

Data Values:

Number of Threads(N)	Time for Parallel Execution seconds
2	805
4	441
10	299
15	282
20	280
50	282
100	291
500	300



BORUVKA ALGORITHM

In the Boruvka's algorithm, the major task that is parallelized among threads is the process of finding the closest edges for each vertex subset.

Parallelized Work:

Closest Edge Calculation for Each Vertex:
-Threads are assigned the venture of computing the nearest part for unique subsets of vertices concurrently.
-The work is divided a number of the threads based on the quantity of threads detailed(numThreads).
-Each thread focuses on a subset of the vertices and checks edges related to those vertices to locate the closest edges.
-As data structure (closestEdge) can be access and update by multiple threads so to avoid data races mutex is used.

How Parallelization is Achieved:

Thread Creation:

The code initializes a vector of threads (thread) to handle parallel computation. It creates threads equal to the specified “numThreads”.

Work Division:

Each thread is responsible for examining a subset of vertices and their corresponding edges. These subsets are assigned to threads in a manner that ensures even distribution and parallel execution.

Thread Execution:

Each thread independently processes its assigned subset of edges to determine the closest edge for each vertex subset. They update the closestEdge vector (protected by the mutex) with the closest edges found.

Thread Joining:

Once all threads have finished their computations, they are joined before further processing the results.

User Input:

The code expects an input file "graph_input.txt" with the following format-
First line: Number of vertices and number of edges (separated by spaces).
Subsequent lines: Edge details (source, destination, weight), one edge per line.

Output Details:

MST Edges and Weights:

Displays the edges of the Minimum Spanning Tree along with their respective weights.
Shows the total weight of the constructed Minimum Spanning Tree.

Data observed data for different scenario while implementing parallelization using

Boruvks's Algo for finding MST

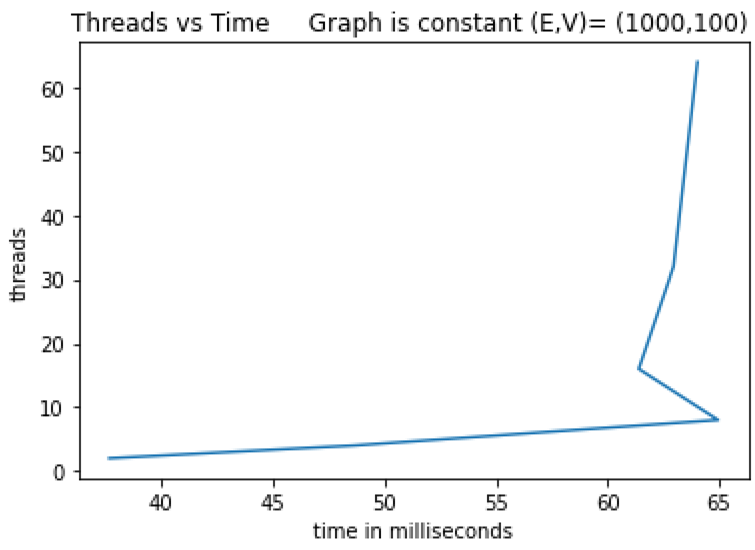
No of threads are constant = 8

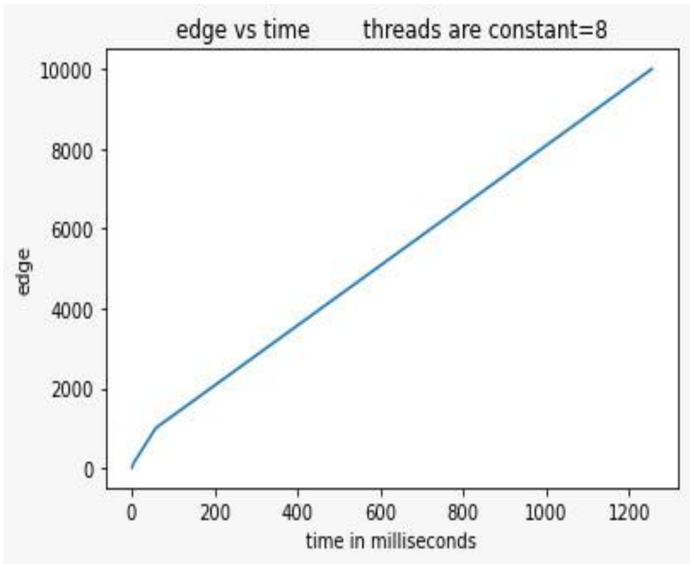
No of Edges and vertices	Time in miliiseconds
10,5	1.209787
100,20	3.615066
1000,100	58.303191
10000,200	1254.886935
100000,1000	104037.993766

No of Edge and Vertices are constant (E,V) = (1000,100)

No of threads	Time in milliseconds
2	37.662397
4	48.629732
8	64.954347
16	61.399566
32	62.959609
64	64.029930

Below is the Plot for time vs number of threads





PARALLEL KRUSKAL ALGORITHM

Problem statement :

Finding the minimum spanning tree for the given undirected weighted connected graph using multiprocessing

Implementation :

I have implemented a parallel version of Kruskal algorithm . The main feature for selecting this algorithm is here I am optimizing the time complexity of Kruskal algorithm from O(mlogn) to O(ma(n)) where a(n) is the ackerman function by using the union and find data structure and sorting the edges using parallelized merge sort

Parallel Kruskal algorithm :

- **Every process p_i basically sorts edges contained in its partition V_i.**
- **Every process p_i finds a local minimum spanning tree F_i using edges in its partition V_i applying the Kruskal algorithm.**
- **Processes merge their local MST. Merging is performed in the following manner. Let b and c denote two processes which are to emerge their local trees and let Fb and Fc denote their respective set of local MST edges. Process b sends set Fb to c, which forms a new local MST (or MSF) from Fb . After merging, process b is no longer involved in computation and can terminate.**
- **Merging continues until only one process remains**

Program Design(code.c)

Main Method:

- **MPI is initialized using 'MPI_Init' and the process rank and size are obtained using 'MPI_Comm_rank' and 'MPI_Comm_size'.**
- **The memory is been allocated for two instances of the 'WeightedGraph' struct: 'graph' and 'mst'.**
- **The root process reds a weighted graph from a file that is specified in the command line argument named as 'graph.txt'. The edges of the original graph are printed using the 'printWeightedGraph' function.**
- **An empty graph is created with the same number of vertices as the original but with one less edge.**

- All process participate in the execution of the ‘mstkruskal’ function which compute the minimum spanning tree.
- MPI is finalized using ‘MPI_Finalize’.

The NewWeightedGraph method :

- This function is to initialize a ‘WeightedGraph’ structure with the specified number of vertices and edges and allocate memory for the edge list.

The ‘readGraphFile’ function

- The function opens the specified input file in read mode using ‘fopen’
- The first line of the function is expected to contain the number of vertices and edges. The ‘newWeightedGraph’ function is called to initialize the number of vertices and edges.
- A loop is used to read information for each edge from the file.

The ‘printWeightedGraph’ function:

- This function works by iterating over theb edges and their components in a ‘WeightedGraph’ structure and printing the information in a tabular format.

The ‘newSet’ function :

- The firstb line sets the ‘elements’ field of the ‘Set’ structure using ‘set->elements=elements’ as it represent the totoal number of elements in the set.
- The second line allocate the memory for the canonical elements
- In the next line indicates a ‘memset’ function which is used to initialize the entire ‘canonicalElemets’ array with the value ‘UNSET_ELEMENT’.
- Memory is allocated for the rank array .

The ‘findset’ function:

- **The purpose of this function is to efficiently find and return the representative of the set to which a given ‘vertex’ belongs in a disjoint set data structure.**
- **Path compression is to optimize the subsequent find operation by updating the canonical elements during the recursive process.**

The ‘unionSet’ function:

- **The function first find the root of the sets containing ‘parent1’ and ‘parent2’ using the ‘findset’ function**
- **The function checks if ‘root1’ and ‘root2’ are already the same.**

The ‘copyEdge’ function:

- **This function utilized ‘memcpy’ to copy the contents of an edge from one memory location to another.**

The ‘scatterEdgeList’ function :

- **It uses MPI to distribute a global edge list among process , adjusting the size of the portion for thje last process and checking for the alst process and checking for the process combination .**
- **It is used in parallel computing scenario where data needs to be distributed among multipkle process.**

The ‘sort’ function:

- **The first line starts by setting up MPI , obtaining the process rank and the total number of process**
- **The ‘parallel’ variable is set to true if the number of process is greater than 1 , indiacating that the MPI is being used in parallel.**
- **The total number of elements in the graph is broadcasted to all processes.**
- **The edges are then scattered among the processes using the ‘scatteeEdgeList’ function.**

- **Each process performs a local sort on its subset of edges using the ‘Mergesort’ function.**
- **Processes participate in a parallel merge operation to combine the locally sorted subsets into a globally sorted edge list.**

The ‘mstKruskal’ function :

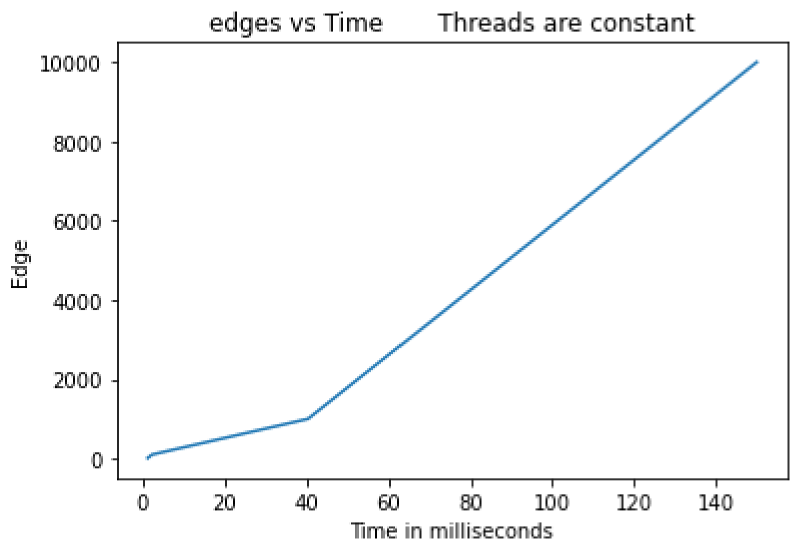
- **A ‘set’ data structure is created and initialized using dynamic memory allocation.**
- **The ‘newset’ function initializes the set with require number of elements**
- **The parallelization is achieved by distributing the edge amomg MPI processes during sorting and merging the results.**
- **The root process is responsible for constructing the MST.**
- **The code ensures that each edge is added only once only if does not create a cycle**

Performance of Parallel Kruskal algorithm

No of threads are constant = 10

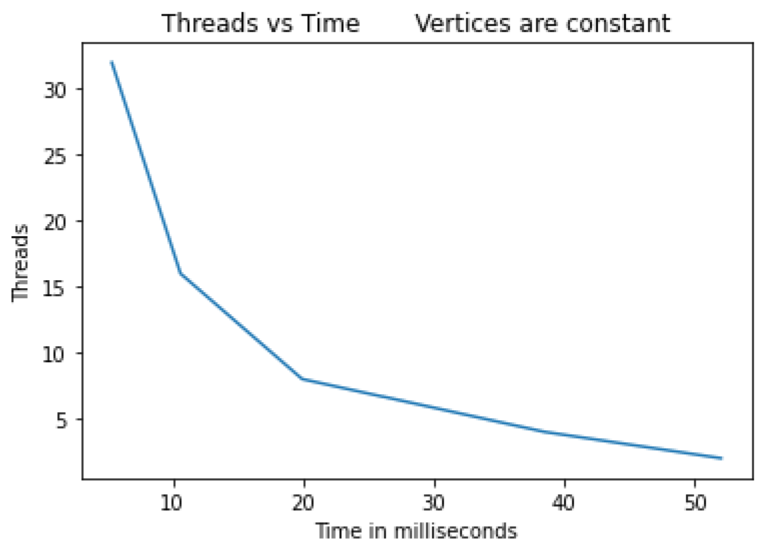
Number of Edge	Time for Parallel Execution milliseconds
5,10	1.10
16,120	2.20
25,300	40.34

150,11175	150
200,44850	2.4844



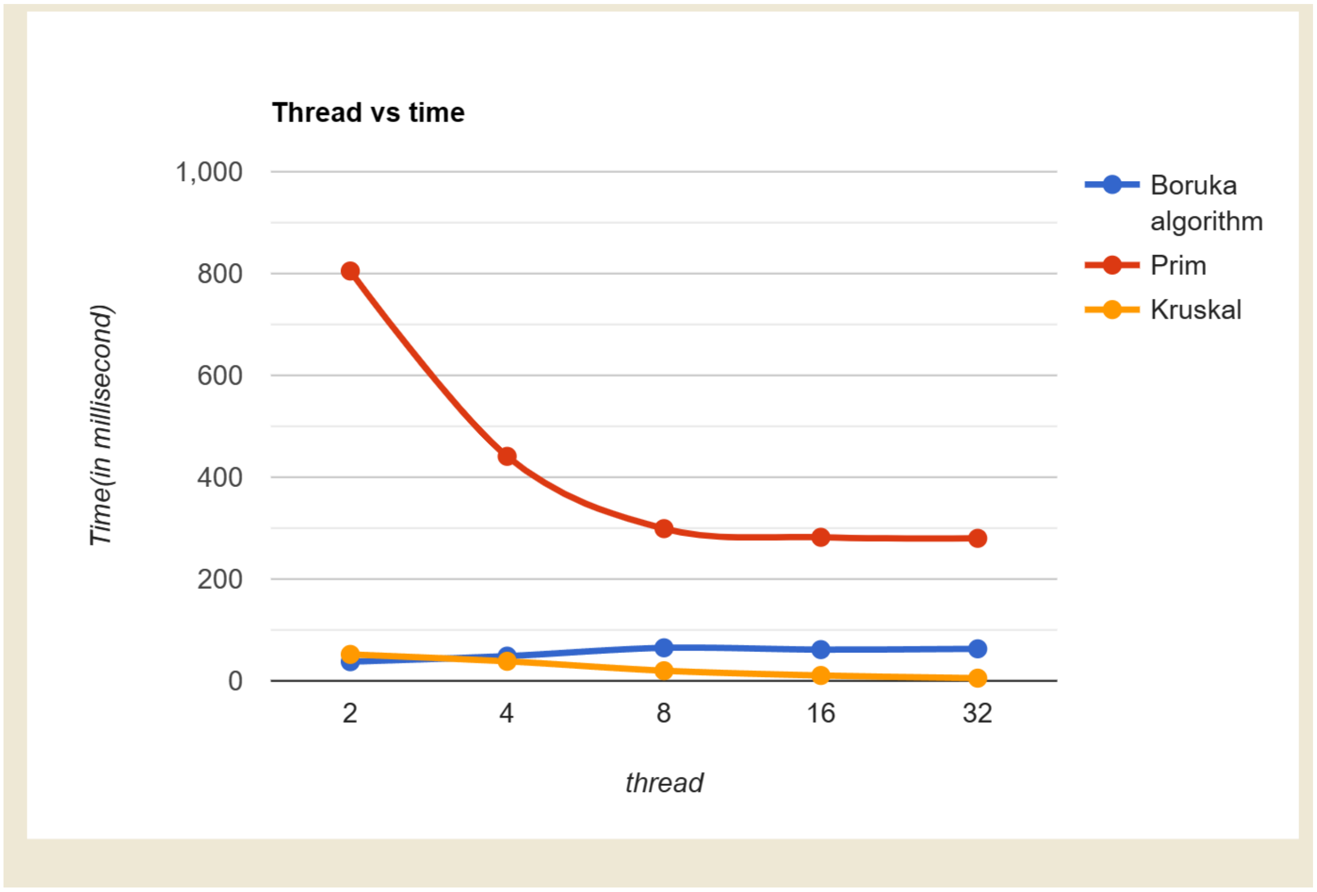
No of Vertices are constant =100

Number of Threads(N)	Time for Parallel Execution milliseconds
2	52
4	38.46
8	19.94
16	10.608
32	5.342



Comparison of different parallel algorithm

When number of vertices are constant


Thread vs time
1,000
800
600
400
200
0
Time(in millisecond)
2
4
8
16
32
thread
Boruka algorithm
Prim
Kruskal