Minimum Spanning Trees

Submitted By

Muhammed Ihsan Corak
Mazen Annatshi
Muhammed Furkan AVCI

CSC770: Parallel Computing
City University of New York – The College of Staten Island

Minimum Spanning Trees

A minimum spanning tree (MST) is a fundamental concept in graph theory. Given a connected, undirected graph with weighted edges, a spanning tree is a subgraph that is a tree and connects all the vertices together without creating any cycles. A minimum spanning tree of a graph is a spanning tree with the smallest possible sum of edge weights. The maximum of edges in a MST will be equal to number of vertex V - 1

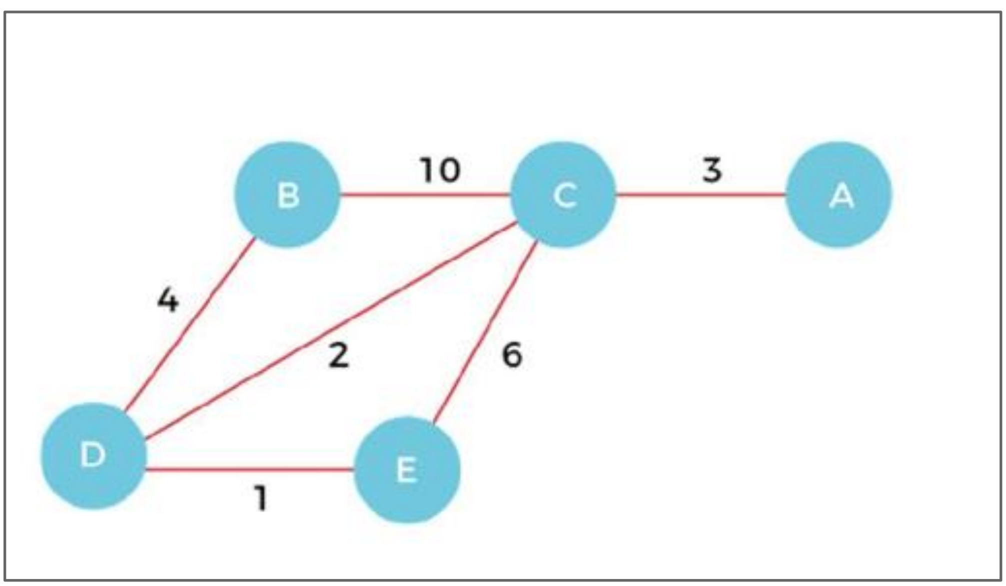
Maximum Edges in MST = V - 1

Prim’s algorithm

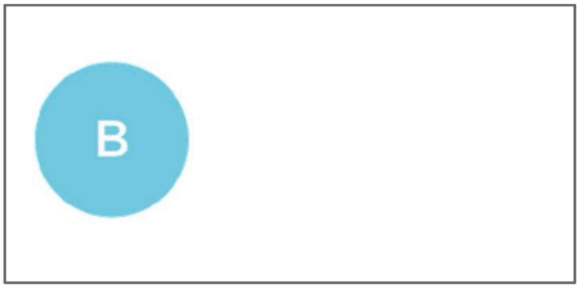
It falls under the umbrella of algorithms called greedy algorithms that find the local optimum solution in the hope of finding global maxima.

Working:

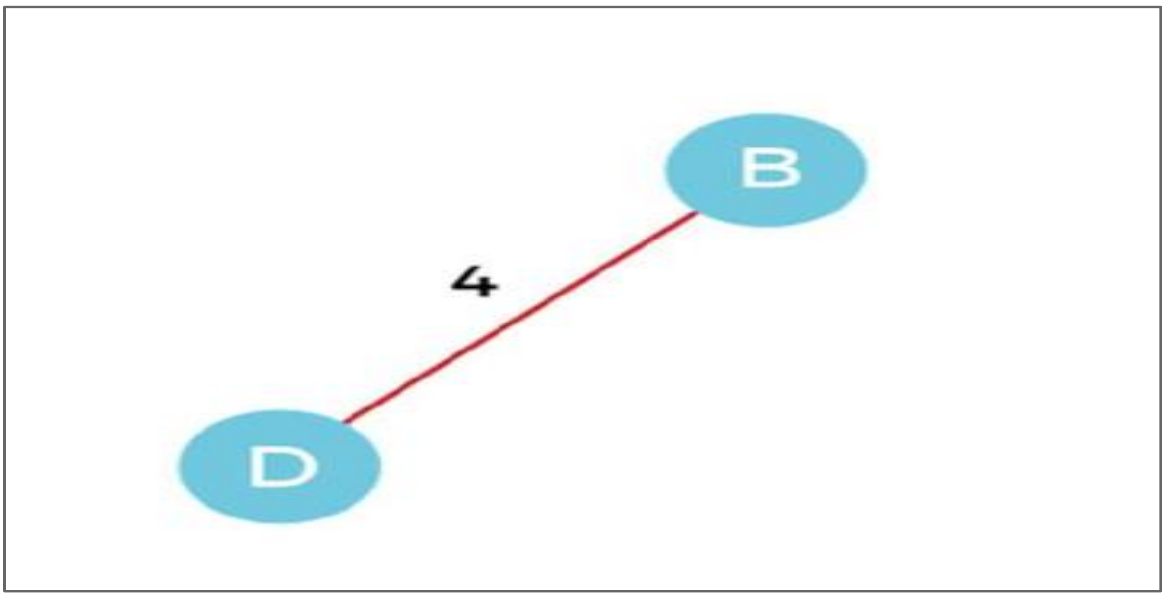
Let us discuss the working of the algorithm through an example. Let's consider the below graph.



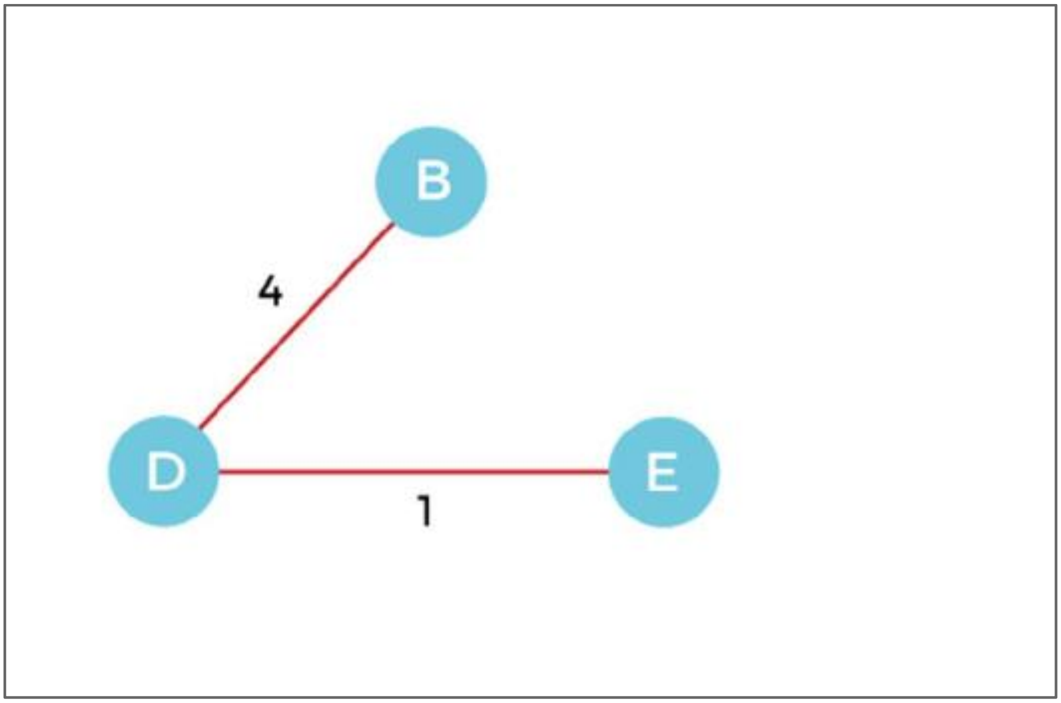
Step 1: Start by selecting a vertex from the given graph. For instance, let's pick vertex B.



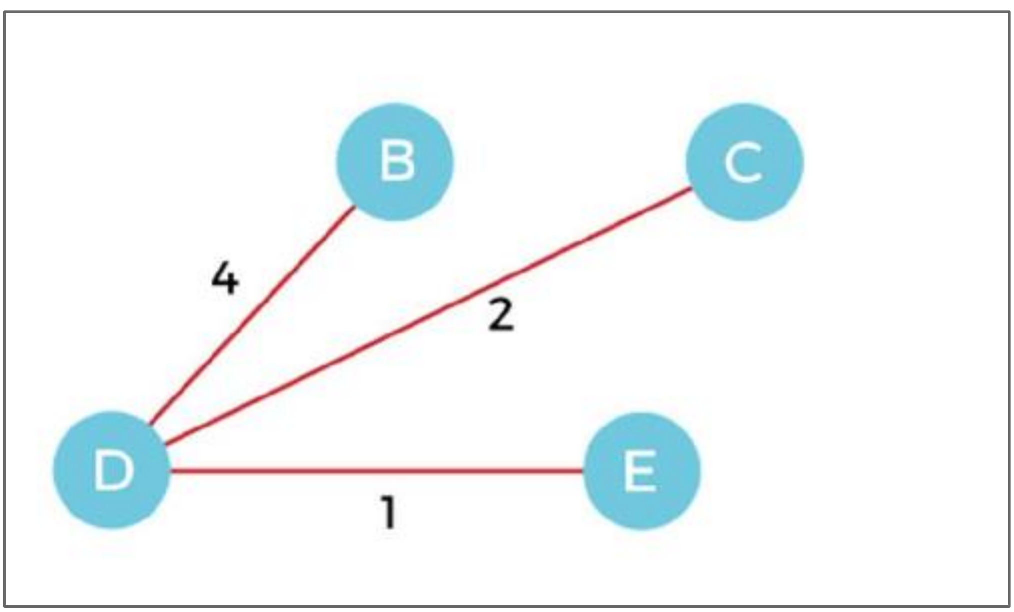
Step 2: Next, select and include the shortest edge connected to vertex B. Vertex B has two adjacent vertices, C and D, with edges of weights 10 and 4 respectively. Choose the edge BD, as it has the smallest weight, and add it to the minimum spanning tree (MST).



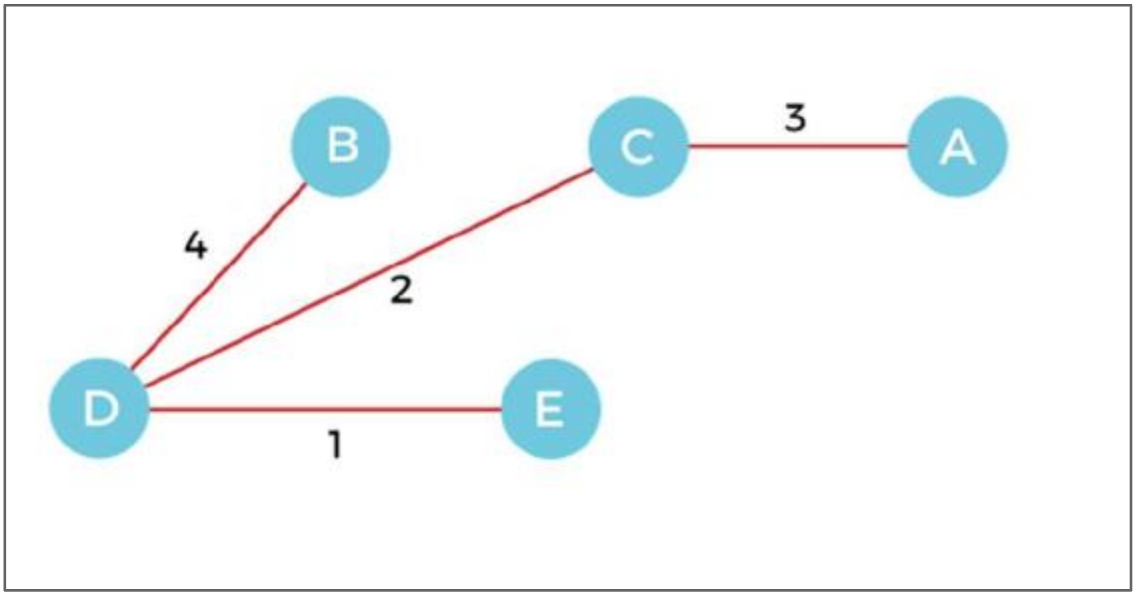
Step 3: Continuing, identify the edge with the smallest weight among the remaining edges. Here, we edge DE and DC and BC with a weight of 1,2,10 respectively. We picked the edge DE since it's the minimum and does not form a circle.



Step 4: Next**,** we edge EC, DC, and BC with a weight of 6, 2,10 respectively. We choose the edge DC as it's the minimum among the given options and does not form a circle.



Step 5: Next**,** we edge EC,CA and BC with a weight 6, 3,10 respectively. We picked the edge CA as it's the minimum among these and does not form a circle.



The cost of the minimum spanning tree is the Sum of Weight of the Edges which is given the sum of edge DB, DE, DC, CA (4, 1, 2, 3 = 10).

Algorithm:

Step 1: Initialize the minimum spanning tree with a vertex chosen at random.

Step 2: Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree if it is not forming the circle.

Step 3: Keep repeating step 2 until we get a minimum spanning tree.

Data Structure Used

- Array Graph Representation of a Weight Graph.
- Array MST for keeping track of vertices.
- Array of boolean for keeping record of visited vertices.

Parallel Implementation Details:

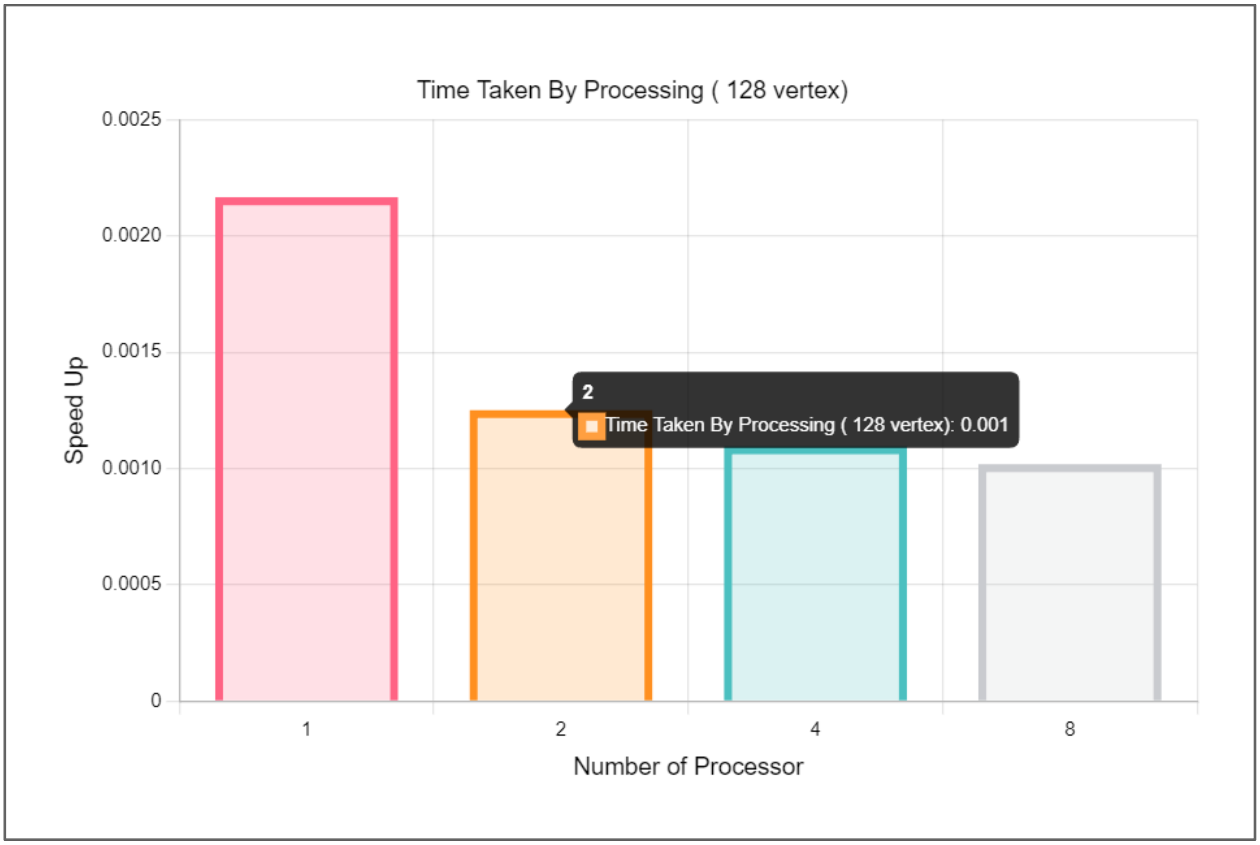
1. Partition the input set V into p subsets, such that each subset contains n/p consecutive vertices and their edges, and assign each process a different subset. Each process also contains part of array d for vertices in its partition.
2. Let Vi be the subset assigned to process pi, and di part of array d which pi maintains. Every process pi finds a minimum-weight edge ei (candidate) connecting MST with a vertex in Vi.
3. Every process pi sends its ei edge to the root process using all-to-one reduction.
4. From the received edges, the root process selects one with a minimum weight (called global minimum-weight edge minimum), adds it to MST and broadcasts it to all other processes.
5. Processes mark vertices connected by minimum as belonging to MST and update their part of array d.
6. Repeat steps 2–5 until every vertex is in M.

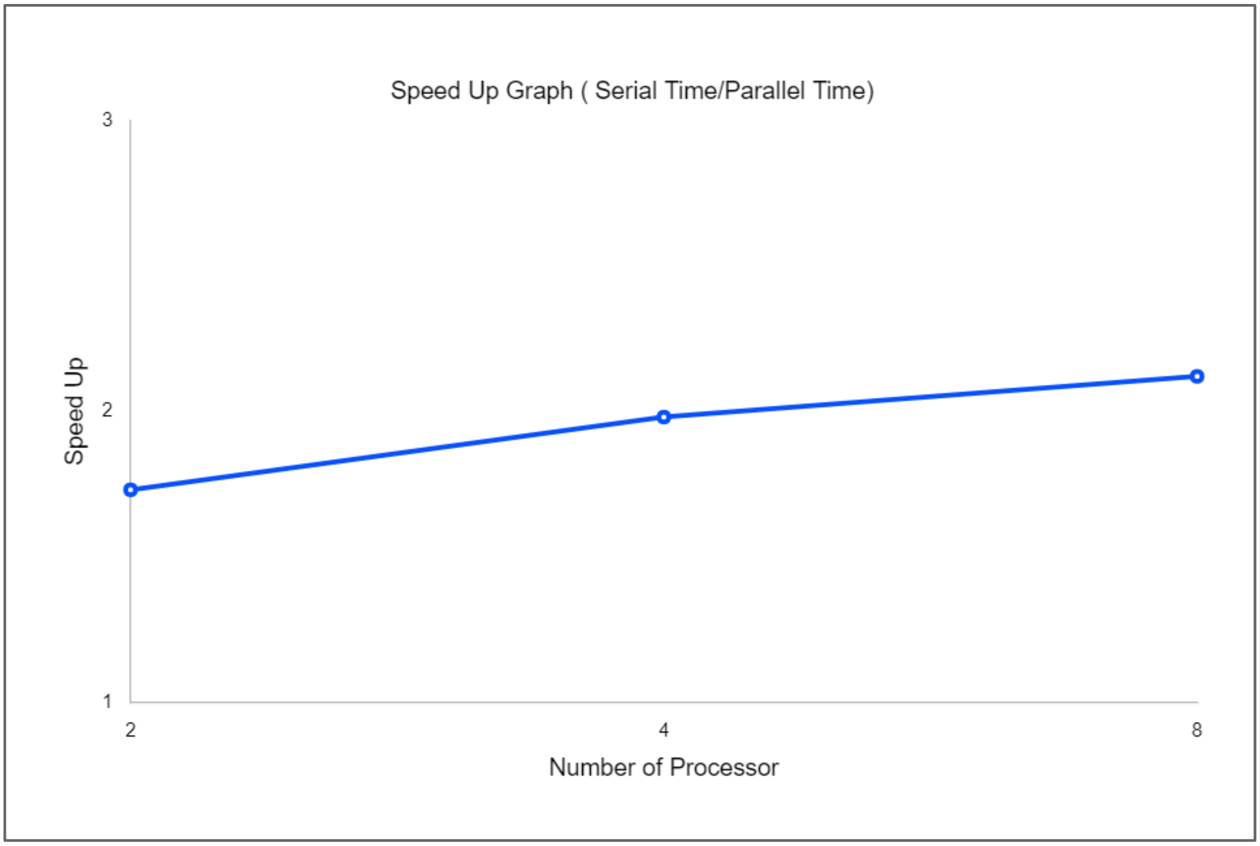
Performance Calculation & Graph:

In this case, **Number of vertices of the graph is constant at 128.**

Number Of Processor	Time Time	Speed Up(Serial time/ Parallel time)
1	**0.002167**	**-(Serial Time)**
2	0.001253	**1.73**
4	0.001097	**1.98**
8	0.001020	**2.12**

Graph:


Time Taken By Processing (128 vertex)
0.0025
0.0020
0.0015
0.0010
0.0005
0
Speed Up
2
Time Taken By Processing (128 vertex): 0.001
1
2
4
8
Number of Processor

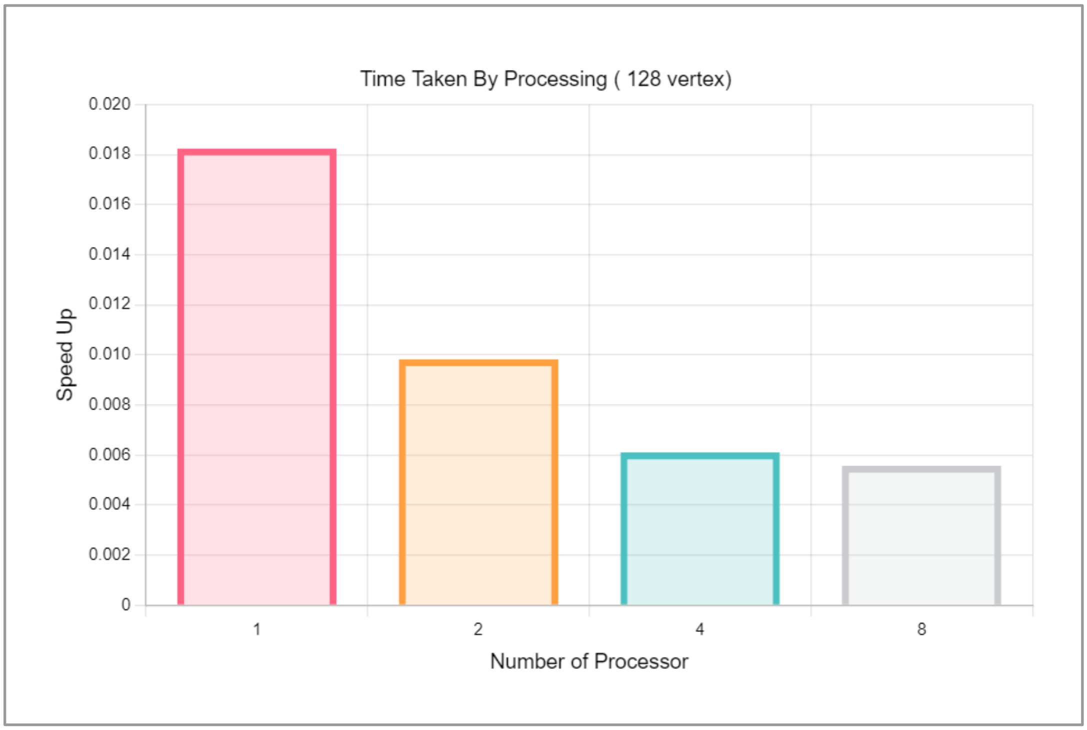


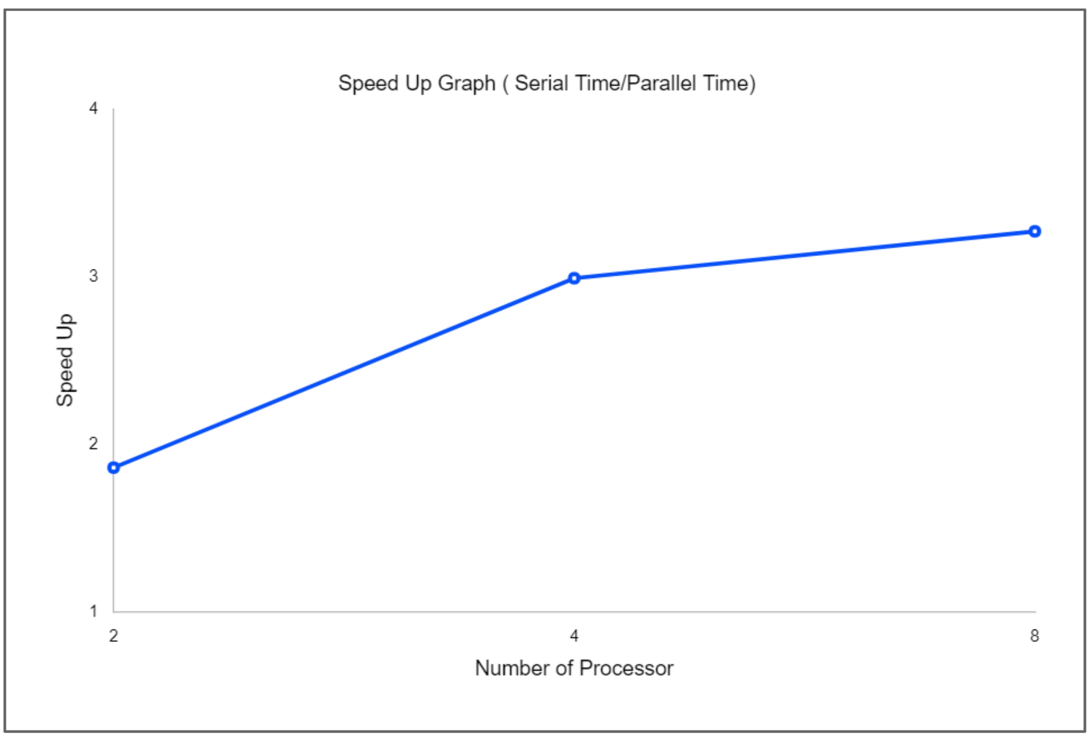
Performance Calculation & Graph:

Increasing the Number of Vertices = 256.

Number Of Processor	Time Time	Speed Up(Parallel time/ serial time)
1	**0.018251**	**-(Serial Time)**
2	0.009830	**1.86**
3	0.006106	**2.99**
4	0.005575	**3.27**

Graph:


Time Taken By Processing (128 vertex)
0.020
0.018
0.016
0.014
0.012
0.010
0.008
0.006
0.004
0.002
0
Speed Up
1
2
4
8
Number of Processor

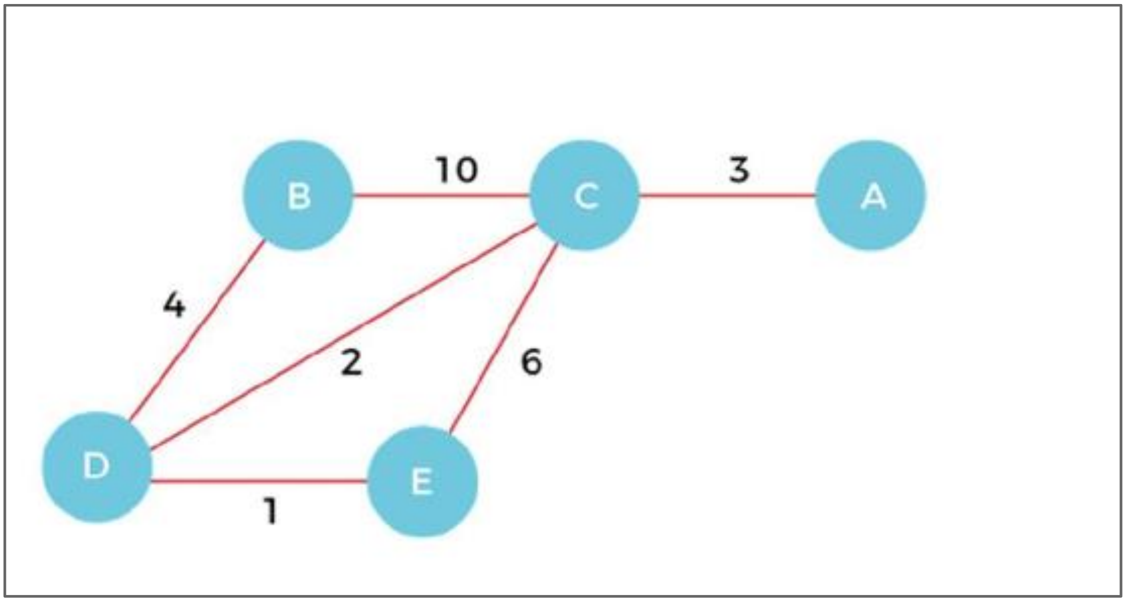

Speed Up Graph (Serial Time/Parallel Time)
4
3
2
1
Speed Up
2
4
8
Number of Processor

Kruskal's Algorithm

It also falls under the umbrella of algorithms called greedy algorithms that find the local optimum solution in the hope of finding global maxima.

Working:

Let us discuss the working of the algorithm through an example. Let's consider the below graph.



Step 1: We first Sort all the edges of the graph into a list in terms of increasing weight value. Here are the Edges after the sorting.

DE - 1, DC - 2, CA - 3, DB - 4 , EC - 6, BC - 10

Step 2: We start picking the edges from the sorting list and add them in case they are already not in MST. We first pick the edge DE and check that it's not in MST and add it.

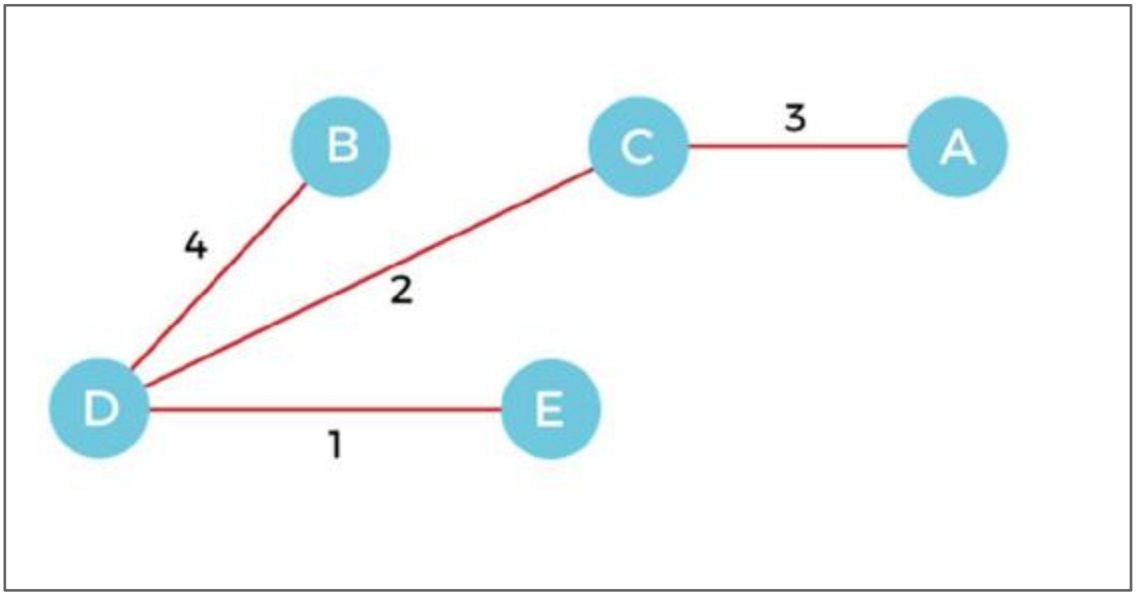
Step 3: Next We pict DC and add it to the MST.

Step 4: Next We pict CA and add it to the MST.

Step 5: Next we BD and add it to the MST.

Step 6: Since, We reached the maximum edges in MST(5-1 = 4 edges), We end the loop here.

Here is the final MST.



Algorithm:

Step 1: Arrange all edges in ascending order of their weights.
Step 2: Select the smallest edge.
Step 3: Verify if adding this edge would create a cycle with the existing spanning tree. If not, incorporate it into the tree. Otherwise, disregard it.
Step 4: Repeat step 2 until the end of vertexes or maximum vertices are found for MST.

Data Structure Used

- Aray Graph Representation of a Weight Graph.
- Array of Edges.
- Merge Sort Algorithm
- Union Algorithm for finding cycle in a non-connected graph.

Parallel Implementation:

Step 1: Since the heart of the algorithm is sorting. So, We sort all the vertices in parallel in terms of increasing weight. First, we divide the edges among processors. Here let's say we have 8 vertices and 2 processors. We divide 8/2 = 4 edges among each processor.

- Each process will then sort its edges separately using merge sort.
- After the sort, Each process sends the sorted edge to root, Root will combine the edges to global sorted array.

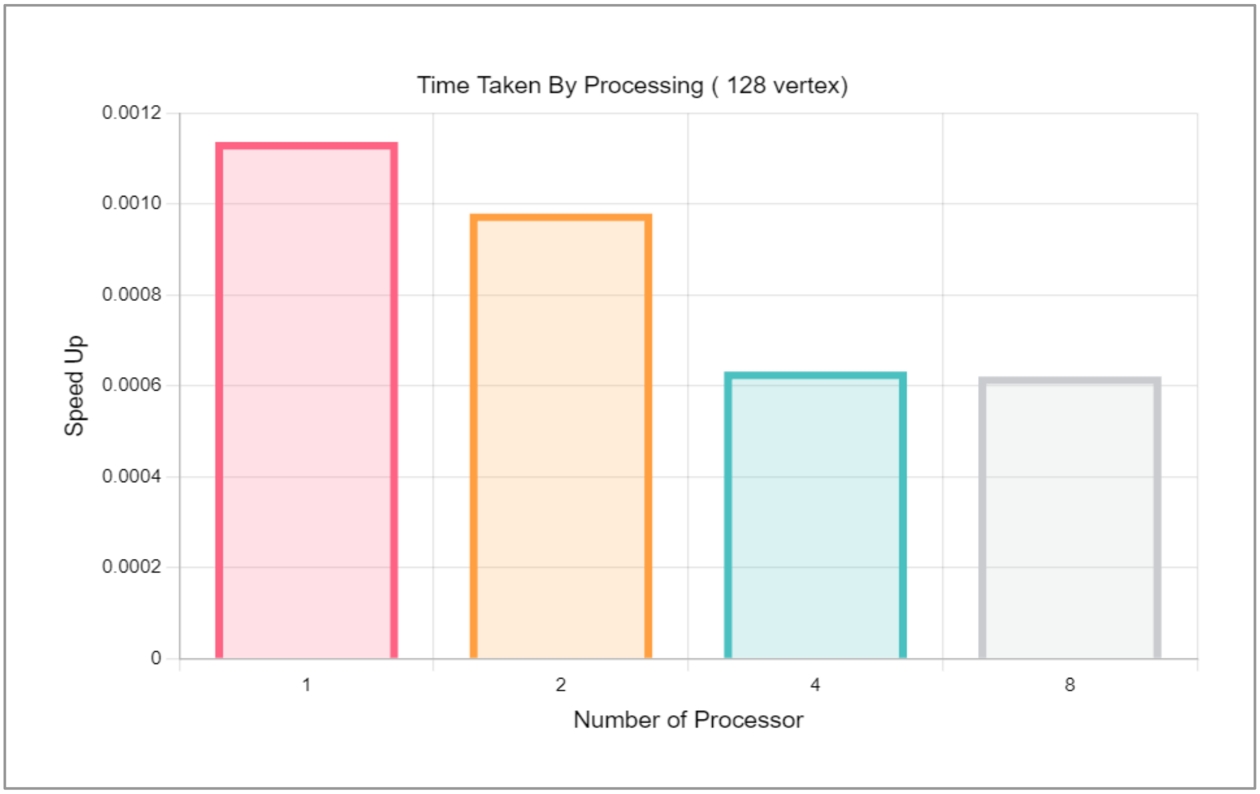
Step 2: Root will simply iterate over the sorted array in O(N) alteration to find the MST. In each iteration following steps are performed.

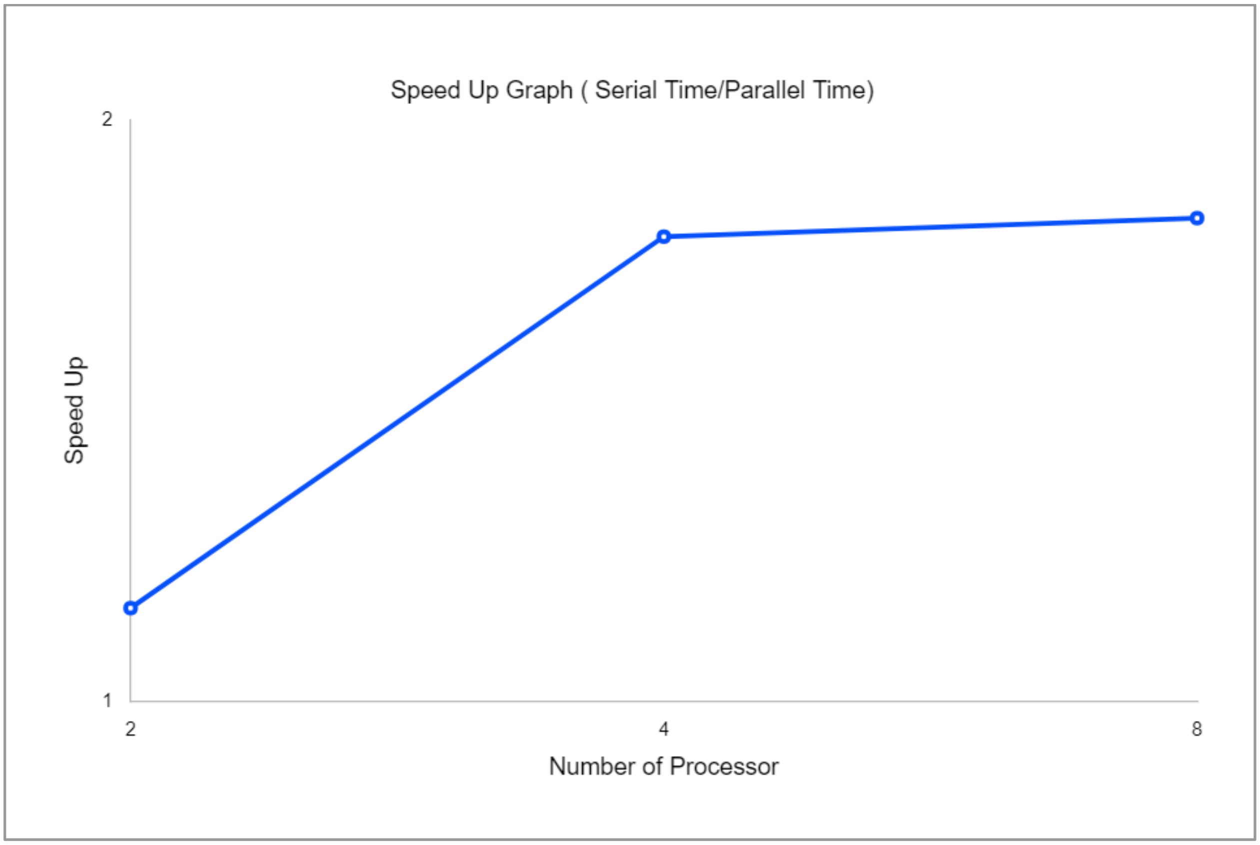
- It picks an edge, will first check if it forms the circle or not using the union algorithm, if the edge does not form a circle and is not in MST, then it's added.
- The step is repeated until we reach the maximum edges in MST or the all the edges are checked.

Performance Calculation & Graph:

Number Of Processor	**Time Time**	**Speed Up(Serial time/ Parallel time)**
1	**0.001137**	**-(Serial Time)**
2	0.000980	**1.16**
4	0.000632	**1.80**
8	0.000621	**1.83**

Graph:


Time Taken By Processing (128 vertex)
Speed Up
0.0012
0.0010
0.0008
0.0006
0.0004
0.0002
0
1
2
4
8
Number of Processor


Speed Up Graph (Serial Time/Parallel Time)
Speed Up
2
1
2
4
8
Number of Processor

Parallel Time Complexities:

- n represents the total amount of data (vertices or edges).
- m represents the total number of edges.
- p represents the number of processors.

Algorithm	Operation	Time Complexity
Prim's Algorithm	Scattering Data	$O(n/p)$
	Minimum Edge Calculation	$O(n/p)$
	Allreduce & Broadcast	$O(\log p)$
	Overall Complexity	$O(n/p) + O(\log p)$
Kruskal's Algorithm	Scattering Data	$O(m/p)$
	Sorting Edges	$O(n \log(n/p))$
	Merging Sorted Lists	$O(\log p)$
	Overall Complexity	$O(m/p) + O(n \log(n/p)) + O(\log p)$

Algorithm 1:

```
// command to compile: mpicc -o algo1 algo2.c
// command to run: mpirun -np <numberProcs> algo1
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>
#include <time.h>

// size of matrix or number of vertices of the graph
#define mSize 128

int *MatrixChunk; // chunk of matrix belong to each processor
typedef struct
{
    int v1;
    int v2;
} Edge;

void primAlgorithm(int rank, int sendcounts, int size){
    // start to calculate running time
    double start= MPI_Wtime();
    // minimum spanning tree including the selected edges
    int* MST = (int *)malloc(sizeof(int) * mSize);
```

```
    // initialize the MST array
    for (int i = 0; i < mSize; ++i)
    {
        MST[i] = -1;
    }

    // the first vertex is always the root of the MST
    MST[0] = 0;
    int minWeightOfGraph = 0;

    // the minimum edge of the graph
    int min, v1 = 0, v2 = 0;
    struct
    {
        int min;
        int rank;
    } minRow, row;
    Edge edge;

    // the main loop of the algorithm
    for (int k = 0; k < mSize - 1; ++k)
    {
        min = INT_MAX;

        // find the minimum edge of the graph, and the vertices of that edge
        for (int i = 0; i < sendcounts; ++i)
        {
            // if the vertex is already in the MST(already not visited)
            if (MST[i + rank*sendcounts] != -1)
            {
                // find the minimum edge of the vertex
                for (int j = 0; j < mSize; ++j)
                {
                    // if the vertex is not in the MST
                    if (MST[j] == -1)
                    {
                        // if the MatrixChunk[mSize*i+j] is less than min
value
                        if (MatrixChunk[mSize * i + j] < min &&
MatrixChunk[mSize * i + j] != 0)
                        {
                            min = MatrixChunk[mSize * i + j];
                            // change the current edge
                            v2 = j;
                            v1 = i;
                        }
                    }
                }
            }
        }
        // find the minimum edge of the graph
        row.min = min;
        row.rank = rank;
        // find the minimum edge of the graph
        MPI_Allreduce(&row, &minRow, 1, MPI_2INT, MPI_MINLOC,
MPI_COMM_WORLD);
        // if the current processor has the minimum edge
```

```
        edge.v1 = v1 + rank*sendcounts;
        edge.v2 = v2;
        // broadcast the minimum edge to all processors
        MPI_Bcast(&edge, 1, MPI_2INT, minRow.rank, MPI_COMM_WORLD);
        // if the edge is not in the MST
        MST[edge.v2] = edge.v1;
        // add the weight of the edge to the minWeightOfGraph
        minWeightOfGraph += minRow.min;
    }

    double finish, calc_time;
    finish = MPI_Wtime();
    calc_time = finish - start;

    // rank 0 will writte its own process time, and values
    if (rank == 0)
    {
        FILE* f_result = fopen("output.txt", "w");
        fprintf(f_result, "\nNumber of processors: %d\nNumber of vertices:
%d\nTime of execution: %f\n", size, mSize, calc_time);
        fprintf(f_result, "The minimun Weight is %d\n", minWeightOfGraph);
        for (int i = 1; i < mSize; ++i)
        {
            fprintf(f_result, "Edge %d %d\n", i, MST[i]);
        }
        fclose(f_result);
    }
    free(MST);

}

int main(int argc, char *argv[])
{
    // rank of current processor
    int size,rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    // to use same random value each time we are not seading the random value
with time
    // srand(time(NULL) + rank);
    int sendcounts = mSize / size;
    // if number of vertices is not a multiply of number of processors then
we need to handle the remainder
    // which is left as for future implementation to reduce the complexity of
the problem
    int *matrix;
    if (rank == 0)
    {
        matrix = (int *)malloc(mSize * mSize * sizeof(int));
        for (int i = 0; i < mSize * mSize; i++)
        {
            matrix[i] = 0;
        }

        // we are generating the value randomly here
        for (int i = 0; i < mSize; ++i)
```

```
        {
            matrix[mSize * i ] = 0;
            for (int j = i + 1; j < mSize; ++j)
            {
                matrix[mSize * i + j] = matrix[mSize * i + j] = rand() % 10;
            }
        }
    }

    // after this each processor needs its own chunk of data
    MatrixChunk = (int *)malloc(sendcounts * mSize * sizeof(int));
    // here the chunk each processor needs will be scatter to it
    MPI_Scatter(matrix, sendcounts*mSize,MPI_INT, MatrixChunk,
sendcounts*mSize, MPI_INT, 0, MPI_COMM_WORLD);
    primAlgorithm(rank, sendcounts, size);
    free(MatrixChunk);
    MPI_Finalize();
    return 0;
}
```

Result of Algorithm 1:

```
Number of processors: 8
Number of vertices: 128
Time of execution: 0.001206
The minimun Weight is 154
Edge 1 0
Edge 2 1
Edge 3 0
Edge 4 0
Edge 5 0
Edge 6 5
Edge 7 4
Edge 8 0
Edge 9 6
Edge 10 0
Edge 11 0
Edge 12 11
Edge 13 11
Edge 14 12
Edge 15 10
Edge 16 1
Edge 17 8
Edge 18 11
Edge 19 0
Edge 20 8
Edge 21 0
Edge 22 21
Edge 23 20
Edge 24 23
Edge 25 10
Edge 26 10
Edge 27 0
Edge 28 27
Edge 29 26
Edge 30 8
Edge 31 8
Edge 32 29
Edge 33 8
Edge 34 21
Edge 35 34
Edge 36 20
Edge 37 21
Edge 38 21
Edge 39 0
Edge 40 25
Edge 41 15
Edge 42 32
Edge 43 0
Edge 44 24
Edge 45 8
Edge 46 39
Edge 47 22
Edge 48 28
Edge 49 35
Edge 50 39
Edge 51 26
Edge 52 45
Edge 53 0
Edge 54 49
Edge 55 15
Edge 56 35
Edge 57 37
Edge 58 15
Edge 59 42
Edge 60 0
Edge 61 35
Edge 62 35
Edge 63 25
Edge 64 15
Edge 65 45
Edge 66 32
Edge 67 10
Edge 68 34
Edge 69 57
Edge 70 32
Edge 71 37
Edge 72 58
Edge 73 10
Edge 74 29
Edge 75 34
Edge 76 42
Edge 77 56
Edge 78 10
Edge 79 15
Edge 80 29
Edge 81 22
Edge 82 37
Edge 83 56
Edge 84 21
Edge 85 25
Edge 86 54
Edge 87 10
Edge 88 0
Edge 89 25
Edge 90 47
Edge 91 39
Edge 92 42
```

```
Edge 93 29
Edge 94 25
Edge 95 42
Edge 96 22
Edge 97 26
Edge 98 38
Edge 99 26
Edge 100 10
Edge 101 22
Edge 102 49
Edge 103 35
Edge 104 37
Edge 105 43
Edge 106 0
Edge 107 29
Edge 108 22
Edge 109 58
Edge 110 15
Edge 111 43
Edge 112 10
Edge 113 0
Edge 114 15
Edge 115 46
Edge 116 0
Edge 117 25
Edge 118 21
Edge 119 41
Edge 120 21
Edge 121 61
Edge 122 10
Edge 123 0
Edge 124 32
Edge 125 29
Edge 126 0
Edge 127 0
```

Algorithm 2:

```
// command to compile: mpicc -o algo2 algo2.c
// command to run: mpirun -np <numberProcs> algo2 <input file>
// Include of libraries
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <mpi.h>

const int ELEMENT_UNSET = -1;

// MPI variables, rank and size
int rank, size;

// Data structures for the graph and the minimum spanning tree
// Set to store find the Disjoint and union
typedef struct Set
{
   int elements;
   int *canonicalElements;
   int *rank;
} Set;

typedef struct Graph
{
   int edges;
   int vertices;
   int *edgeList;
} Graph;

// allocate memory on heap for the grapg
void newGraph(Graph *graph, const int Vertices, const int Edges)
{
   graph->edges = Edges;
   graph->vertices = Vertices;
   graph->edgeList = (int *)calloc(Edges* 3, sizeof(int));
}

// read a graph from a file and create the graph
void readDataOfGraphFromFile(Graph *graph, const char inputFileName[])
{
   // open the file for reading
   FILE *inputFile = fopen(inputFileName, "r");
   if (inputFile == NULL)
   {
      fprintf(stderr, "Couldn't open input file, exiting!\n");
      exit(EXIT_FAILURE);
   }
   int fscanfResult;
   // first line contains number of vertices and edges
   int vertices = 0, edges = 0;
   fscanfResult = fscanf(inputFile, "%d %d", &vertices, &edges);
```

```
    newGraph(graph, vertices, edges);

    int from, to, weight;
    for (int i = 0; i < edges; i++)
    {
        fscanfResult = fscanf(inputFile, "%d %d %d", &from, &to, &weight);
        graph->edgeList[i * 3] = from;
        graph->edgeList[i * 3 + 1] = to;
        graph->edgeList[i * 3 + 2] = weight;
    }
    fclose(inputFile);
}

// print all edges of the graph correct fromat
void printGraph(const Graph *graph)
{
    printf("##################################################\n");
    for (int i = 0; i < graph->edges; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            printf("%d\t", graph->edgeList[i * 3 + j]);
        }
        printf("\n");
    }
    printf("##################################################\n");
}

// Create a set
void createSet(Set *set, const int elements)
{
    set->elements = elements;
    set->canonicalElements = (int *)malloc(elements * sizeof(int));
    memset(set->canonicalElements, ELEMENT_UNSET, elements * sizeof(int));
    set->rank = (int *)calloc(elements, sizeof(int));
}

// return the element of a vertex with path compression
//  what is canonical element?
//  A canonical element is the representative of a set. It is the element
that is used to identify the set.
//  For example, in the set {1, 2, 3}, 1 is the canonical element.
//  Path compression is a technique used to optimize the find operation in a
disjoint-set data structure.
//  It involves updating the parent of each node in the path to the root of
the set during the find operation.
//  This way, subsequent find operations on the same set will be faster.
int FindSet(const Set *set, const int vertex)
{
    if (set->canonicalElements[vertex] == ELEMENT_UNSET)
    {
        return vertex;
    }
    else
    {
        set->canonicalElements[vertex] = FindSet(set, set-
>canonicalElements[vertex]);
```

```
        return set->canonicalElements[vertex];
    }
}

// merge the set of parent1 and parent2 with union by rank
void unionSet(Set *set, const int parent1, const int parent2)
{
    int root1 = FindSet(set, parent1);
    int root2 = FindSet(set, parent2);

    if (root1 == root2)
    {
        return;
    }
    else if (set->rank[root1] < set->rank[root2])
    {
        set->canonicalElements[root1] = root2;
    }
    else if (set->rank[root1] > set->rank[root2])
    {
        set->canonicalElements[root2] = root1;
    }
    else
    {
        set->canonicalElements[root1] = root2;
        set->rank[root2] = set->rank[root1] + 1;
    }
}

// copy the value of the Edge from to to from
void copyEdge(int *To, int *From)
{
    memcpy(To, From, 3 * sizeof(int));
}

// merge sorted lists, start and end are inclusive
void merge(int *edgeList, const int start, const int end, const int pivot)
{
    int length = end - start + 1;
    int *working = (int *)malloc(length * 3 * sizeof(int));
    // copy first part
    memcpy(working, &edgeList[start * 3], (pivot - start + 1) * 3 *
sizeof(int));
    // copy second part reverse to simpify merge
    int workingEnd = end + pivot - start + 1;
    for (int i = pivot + 1; i <= end; i++)
    {
        copyEdge(&working[(workingEnd - i) * 3], &edgeList[i * 3]);
    }
    int left = 0;
    int right = end - start;
    for (int k = start; k <= end; k++)
    {
        if (working[right * 3 + 2] < working[left * 3 + 2])
        {
            copyEdge(&edgeList[k * 3], &working[right * 3]);
            right--;
```

```
        }
        else
        {
            copyEdge(&edgeList[k * 3], &working[left * 3]);
            left++;
        }
    }
    // clean up the memory
    free(working);
}

// sort the edge list using merge sort, start and end are inclusive
void mergeSort(int *edgeList, const int start, const int end)
{
    if (start != end)
    {
        // recursively divide the list in two parts and sort them
        int pivot = (start + end) / 2;
        mergeSort(edgeList, start, pivot);
        mergeSort(edgeList, pivot + 1, end);

        merge(edgeList, start, end, pivot);
    }
}

// sort the edges of the graph in parallel with mergesort in parallel
void sort(Graph *graph)
{
    int elements;
    bool parallel = size != 1;
    // send number of elements
    if (rank == 0)
    {
        elements = graph->edges;
    }
    MPI_Bcast(&elements, 1, MPI_INT, 0, MPI_COMM_WORLD);
    // scatter the edges to sort
    int elementsPart = (elements + size - 1) / size;
    int *edgeListDivided = (int *)malloc(elementsPart * 3 * sizeof(int));
    if (parallel)
    {
        if (elements % size != 0)
        {
            if (rank == 0){
                fprintf(stderr, "Please make sure number of edges are divible by
number of Processor!\n");
            }
            MPI_Finalize();
            exit(0);
        }
        MPI_Scatter(graph->edgeList, elementsPart * 3, MPI_INT,
edgeListDivided, elementsPart * 3, MPI_INT, 0, MPI_COMM_WORLD);
    }
    else
    {
        edgeListDivided = graph->edgeList;
    }
```

```
    // sort the part
    mergeSort(edgeListDivided, 0, elementsPart - 1);

    if (parallel)
    {
        // merge all parts
        int from, to, elementsRecieved;
        for (int step = 1; step < size; step *= 2)
        {
            if (rank % (2 * step) == 0)
            {
                from = rank + step;
                if (from < size)
                {
                    MPI_Recv(&elementsRecieved, 1, MPI_INT, from, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                    edgeListDivided = realloc(edgeListDivided, (elementsPart +
elementsRecieved) * 3 * sizeof(int));
                    MPI_Recv(&edgeListDivided[elementsPart * 3], elementsRecieved
* 3, MPI_INT, from, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                    merge(edgeListDivided, 0, elementsPart + elementsRecieved - 1,
elementsPart - 1);
                    elementsPart += elementsRecieved;
                }
            }
            else if (rank % step == 0)
            {
                to = rank - step;
                MPI_Send(&elementsPart, 1, MPI_INT, to, 0, MPI_COMM_WORLD);
                MPI_Send(edgeListDivided, elementsPart * 3, MPI_INT, to, 0,
MPI_COMM_WORLD);
            }
        }

        // edgeListDivided is the new edgeList of the graph, cleanup other
memory
        if (rank == 0)
        {
            free(graph->edgeList);
            graph->edgeList = edgeListDivided;
        }
        else
        {
            free(edgeListDivided);
        }
    }
    else
    {
        graph->edgeList = edgeListDivided;
    }
}

// find a MST of the graph using Kruskal's algorithm
void kruskalAlgorithm(Graph *graph, Graph *mst)
{
    // create needed data structures
```

```
    Set *set = &(Set){.elements = 0, .canonicalElements = NULL, .rank = NULL};
    createSet(set, graph->vertices);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // sort the edges of the graph, we are sorting the edge of the graph in
parallel
    sort(graph);

    if (rank == 0)
    {
        // add edges to the MST
        int currentEdge = 0;
        for (int edgesMST = 0; edgesMST < graph->vertices - 1 || currentEdge <
graph->edges;)
        {
            // check for loops if edge would be inserted
            int ElementFrom = FindSet(set, graph->edgeList[currentEdge * 3]);
            int ElementTo = FindSet(set, graph->edgeList[currentEdge * 3 + 1]);
            if (ElementFrom != ElementTo)
            {
                // add edge to MST
                copyEdge(&mst->edgeList[edgesMST * 3], &graph-
>edgeList[currentEdge * 3]);
                unionSet(set, ElementFrom, ElementTo);
                edgesMST++;
            }
            currentEdge++;
        }
    }

    // clean the allocated memory
    free(set->canonicalElements);
    free(set->rank);
}

// main program
int main(int argc, char *argv[])
{
    // MPI variables and initialization
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (argc < 2)
    {
        if (rank == 0)
        {
            printf("Usage:  mpirun -np <numberProcs> %s <input file>\n",
argv[0]);
        }
        MPI_Finalize();
        return 0;
    }

    // graph Variables
    Graph *graph = &(Graph){.edges = 0, .vertices = 0, .edgeList = NULL};
```

```
   Graph *MST = &(Graph){.edges = 0, .vertices = 0, .edgeList = NULL};
   if (rank == 0)
   {
      // read the graph from the file
      readDataOfGraphFromFile(graph, argv[1]);
      newGraph(MST, graph->vertices, graph->vertices - 1);
   }

   double start = MPI_Wtime();
   // use Kruskal's algorithm
   kruskalAlgorithm(graph, MST);
   if (rank == 0)
   {
      printf("Time elapsed: %f s\n", MPI_Wtime() - start);
      // print the edges of the MST
      printf("Minimum Spanning Tree (Kruskal):\n");
      unsigned long weightMST = 0;
      for (int i = 0; i < MST->edges; i++)
      {
         weightMST += MST->edgeList[i * 3 + 2];
      }
      printf("MST weight: %lu\n", weightMST);
      printGraph(MST);
      // cleanup memory
      free(graph->edgeList);
      free(MST->edgeList);
   }
   MPI_Finalize();
   return 0;
```

Result of Algorithm 2:

```
[[annatshi@csremote1 final]$ mpirun -np 4 ./algorithm2 testgraph.txt
Time elapsed: 0.000074 s
Minimum Spanning Tree (Kruskal):
MST weight: 18
#################################################################
0       1       1
1       4       2
0       5       3
0       3       5
1       2       7
#################################################################
```

```
[annatshi@csremote1 final]$ mpirun -np 8 ./algorithm2 largegraph.txt
Time elapsed: 0.000887 s
Minimum Spanning Tree (Kruskal):
MST weight: 127
##################################################
0       10      1
0       21      1
0       39      1
0       43      1
0       53      1
0       60      1
0       88      1
0       106     1
0       113     1
0       116     1
0       123     1
0       126     1
0       127     1
1       16      1
1       32      1
1       33      1
1       39      1
1       50      1
1       56      1
1       62      1
1       72      1
1       114     1
2       27      1
2       29      1
2       30      1
2       37      1
2       71      1
2       74      1
2       78      1
2       81      1
2       95      1
2       115     1
3       10      1
3       14      1
3       23      1
3       69      1
3       87      1
3       91      1
3       109     1
3       122     1
4       8       1
4       9       1
4       39      1
4       41      1
4       58      1
4       89      1
4       100     1
4       118     1
4       121     1
4       124     1
5       6       1
5       13      1
5       17      1
5       24      1
5       40      1
5       50      1
5       90      1
5       96      1
5       107     1
5       108     1
6       64      1
6       68      1
6       86      1
6       104     1
7       8       1
7       25      1
7       42      1
7       47      1
7       71      1
7       112     1
8       20      1
8       45      1
8       49      1
8       55      1
8       67      1
8       82      1
8       102     1
9       15      1
9       31      1
9       85      1
9       98      1
9       110     1
10      26      1
10      73      1
11      13      1
11      18      1
11      35      1
11      97      1
11      103     1
11      120     1
```

```
12      14      1
12      19      1
12      28      1
12      77      1
13      52      1
13      63      1
13      75      1
13      99      1
14      57      1
14      70      1
14      117     1
15      79      1
16      36      1
16      76      1
16      93      1
16      94      1
17      101     1
18      59      1
18      65      1
19      22      1
19      38      1
19      44      1
19      66      1
20      54      1
21      34      1
21      84      1
24      46      1
26      51      1
28      48      1
29      80      1
29      125     1
35      61      1
36      83      1
41      119     1
42      92      1
43      105     1
43      111     1
##################################################
```

Future Work:

Currently, The algorithm is not working When the division of vertex is not uniform. In other words, When vertexes % Processes != 0, In that case the scatter function will not work, We need to improve the algorithm to support that.

Further Notes:

The communication overhead becomes quite significant with a large number of processors with small verteces, As in that case serial would do a better job on small graphs as compared to using n number of processors.

Conclusion:

To conclude, Minimum Spanning Tree (MST) algorithms like Prim's and Kruskal's play a pivotal role in network design, including applications such as network traffic minimization, the design of efficient routing protocols, and infrastructure development. Both algorithms are examples of greedy techniques used to find globally optimal solutions by making a sequence of locally optimal choices.

Prim's algorithm is efficient in scenarios where the graph has a dense adjacency matrix, as it incrementally builds the MST by adding the cheapest edge that expands the tree. On the other hand, Kruskal's algorithm is generally more efficient for sparse graphs, as it builds the MST by sorting all the edges first and then selecting the smallest edges that do not form a cycle.

The parallel implementations of these algorithms demonstrate significant improvements in computational efficiency, which is particularly notable as the size of the graph increases. This makes parallel computing a valuable approach for dealing with large-scale problems in real-time applications. However, the efficiency gains from parallelization must be carefully weighed against the overhead of process communication, especially when dealing with smaller graphs or a large number of processors.

Reference:

- Osipov, V., Sanders, P., & Singler, J. (2009, January). The filter-kruskal minimum spanning tree algorithm. In *2009 Proceedings of the Eleventh Workshop on Algorithm Engineering and Experiments (ALENEX)* (pp. 52-61). Society for Industrial and Applied Mathematics.
- Kershenbaum, A., & Van Slyke, R. (1972, August). Computing minimum spanning trees efficiently. In *Proceedings of the ACM annual conference-Volume 1* (pp. 518-527).
- Mamun, A. A., & Rajasekaran, S. (2016, June). An efficient minimum spanning tree algorithm. In *2016 IEEE Symposium on Computers and Communication (ISCC)* (pp. 1047-1052). IEEE.
- https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/
- https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/
- https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/
- https://linegraphmaker.co/
- https://linegraphmaker.co/bar-graph