P3

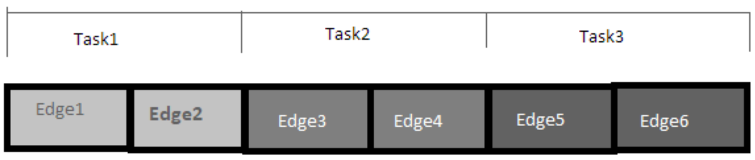
Problem: Sollin's Algorithm for minimum spanning tree(MST).

In each iteration, Sollin's algorithm:

1) For each component, find the cheapest edge that connects it to some other component.
2) Adds these edges to the MST.

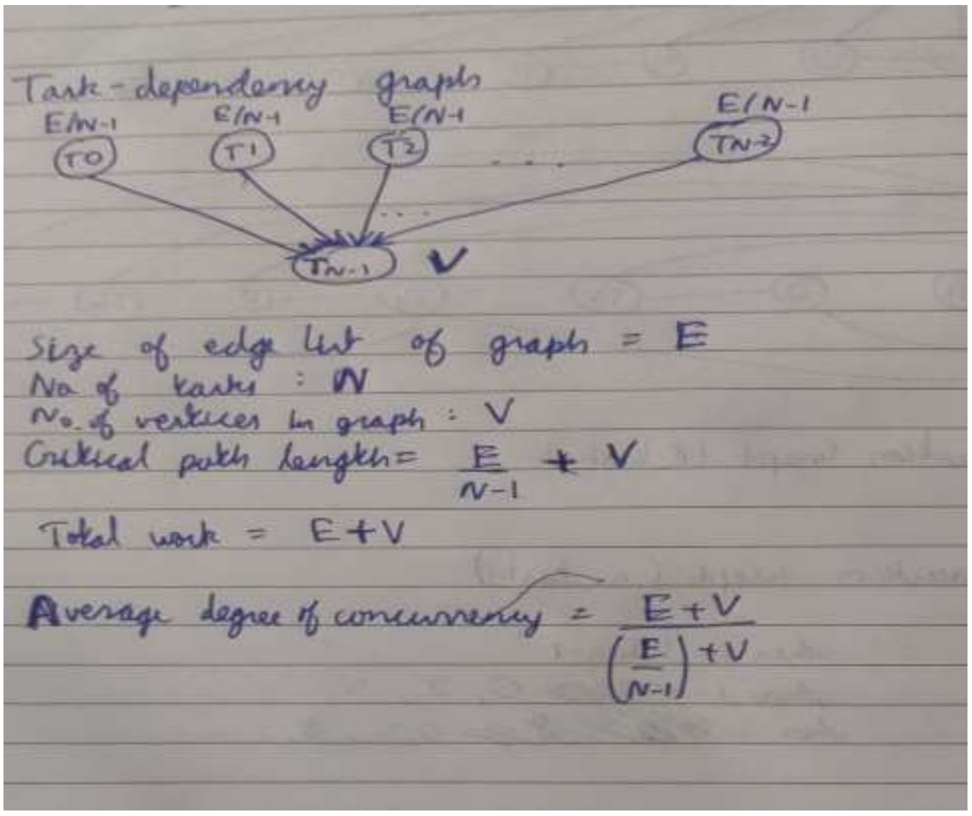
This is done till we have V-1 edges in the MST i.e., only one component is left.

For the 1st part, serial algorithm iterates over the entire edge list and maintains a list of cheapest edges (not in MST) for each component. We parallelize this process by dividing the edge list into regions and assign each of them to a task. This idea is based on **Input Data Decomposition**. We have used coarse-grained distribution.



To summarize, suppose we have N tasks, N-1 tasks can be run parallelly to complete the 1st part of each iteration and 1 task will be responsible for adding these edges to the MST.

Task-dependency graph, Critical path length & Average degree of concurrency:



Size of edge list of graph = E

No of tasks : N

No. of vertices in graph : V

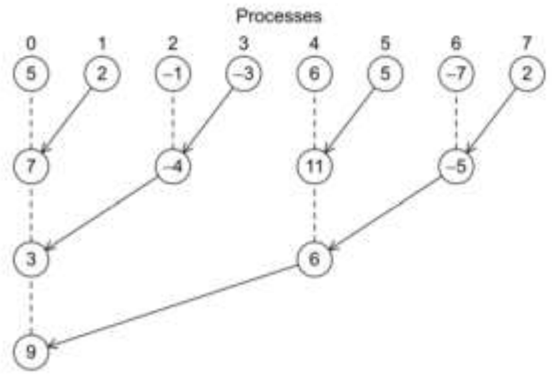
Critical path length = $\frac{E}{N-1} + V$

Total work = $E+V$

Average degree of concurrency = $\frac{E+V}{\left(\frac{E}{N-1}\right)+V}$

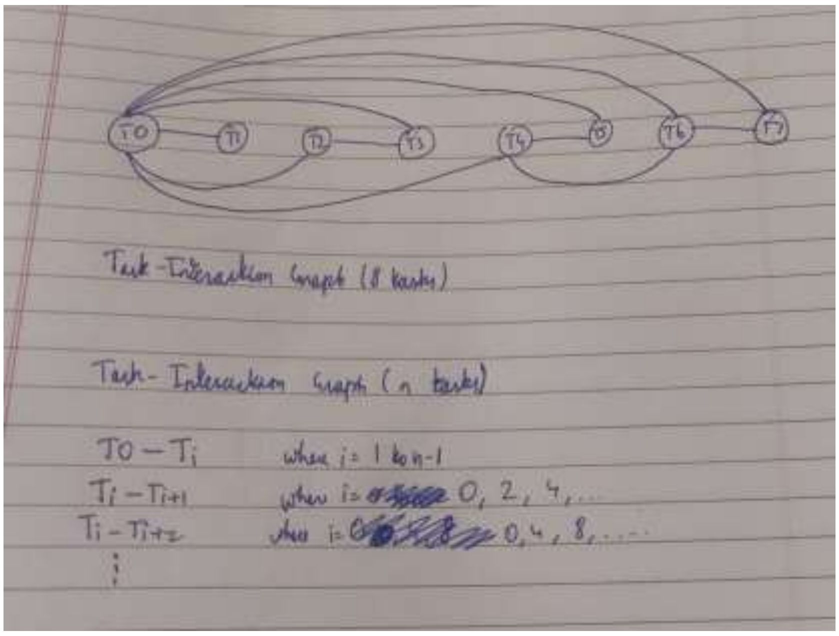
NOTE: Please see that the following task-dependency graph is for **only one iteration** of the algorithm. The same algorithm repeats for about log(V) times.

Each task will iterate over its region and maintain its own list of cheapest edges for each component. When all tasks have iterated over their region, the cheapest edge lists are compared using **tree-structured communication**.



The root process then adds the cheapest set of edges to the MST and broadcasts the updated MST.

Task-interaction graph:



Task-Interaction Graph (8 tasks)

Task- Interaction Graph (n tasks)

T0 — Ti when i = 1 to n-1
Ti — Ti+1 when i = 0, 2, 4, ...
Ti — Ti+2 when i = 0, 4, 8, ...
⋮

We have used **1-D Block Distribution** on the input data (edge list) because in this case, block distribution **does not lead to load imbalance**. Therefore, there was no need to perform cyclic distribution.

In the first level, each task is assigned to a process by owner computes rule since we have used data partitioning model. In the second level there is a single task which is assigned to the process ranked 0.

Explanation of Parallel Algorithm:

1) The input to the algorithm is a list of edges where each edge is represented in the from-to-weight format.
2) We use MPI_Scatterv to distribute this list into p processes where each process gets E/p*3 elements. We have multiplied by 3 because of the from-to-weight format.
3) Each process initializes an array parent to store the disjoint set number (or parent) for each vertex. Initially parent of each vertex will be -1. We have used Path Compression to optimize the disjoint set union.
4) Each process initializes an array to store the cheapest edge for each component and another array to use as receive buffer when it will receive the cheapest edge array from another process.
5) Inside the for loop: Each process iterates over its block. If an edge (not belonging to MST) joins 2 components and has minimum weight amongst other such edges, it is stored in the cheapest edge array for both the from and the to component.
6) Inside the for loop: Processes send their cheapest edge array in tree-structured communication so that the final cheapest edge array arrives to process 0.
7) Inside the for loop: Process 0 then uses this array to add edges to the MST.
8) This continues till no. of edges in MST reaches V-1.

Speedup, Efficiency, Cost and Scalability:

Ts = (E+V)*logV

Tp = (E/p+ V + p(ts + twV))*logV

Speedup = (E+V)/(E/p+V+p(ts+twV))

Efficiency = (E+V)/(E+pV+(p^2)*(ts+twV))

Cost = p*Tp = (E+pV+(p^2)*(ts+twV))

Scalability:

To/Ts = ((p-1)*V + (p^2)*(ts+twV))/(E+V)

Scalability of this algorithm depends on the density of the graph. If graph is dense i.e. E~V^2, then the efficiency will increase upon increasing V. If the graph is sparse i.e. E~V, then the algorithm is not scalable and will have decreasing efficiency with V.

Here,

E=no. of edges

V=no. of vertices

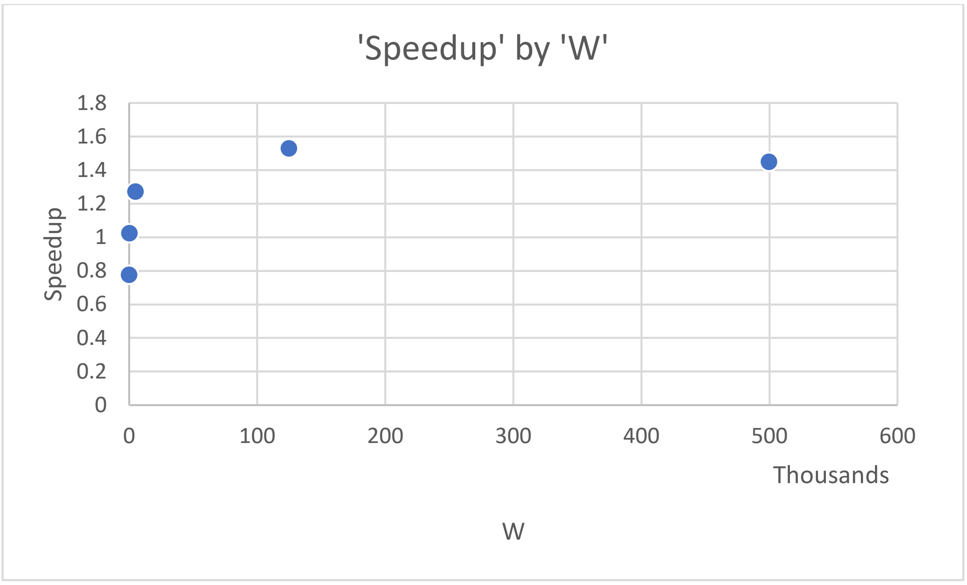
P=no. of processes

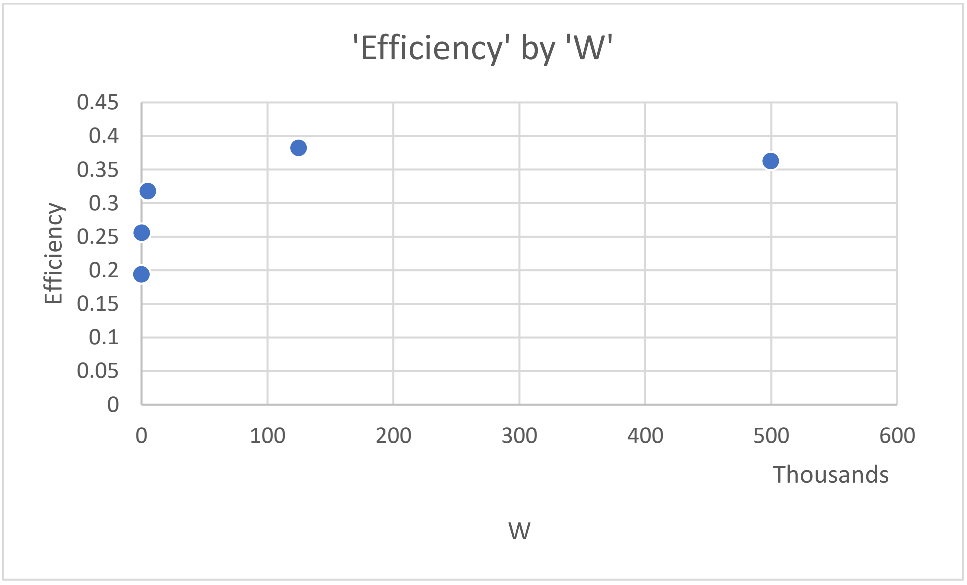
ts=startup time

tw=per-word transfer time

Size of edge list(E)	P=1(serial)	P=4	Speedup	Efficiency
10	0.0154ms	0.0198ms	0.776	0.194
190	0.0385ms	0.0376ms	1.024	0.256
4950	0.2056ms	0.1615ms	1.273	0.318
124750	3.456ms	2.8976ms	1.529	0.382
499500	12.782ms	8.8153ms	1.450	0.3625

NOTE: We have used complete (very dense) graphs. Due to this, the efficiency increases upon increasing E (and also V).


'Speedup' by 'W'
Speedup
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
100
200
300
400
500
600
Thousands
W


'Efficiency' by 'W'
Efficiency
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0
100
200
300
400
500
600
Thousands
W

END