Q3

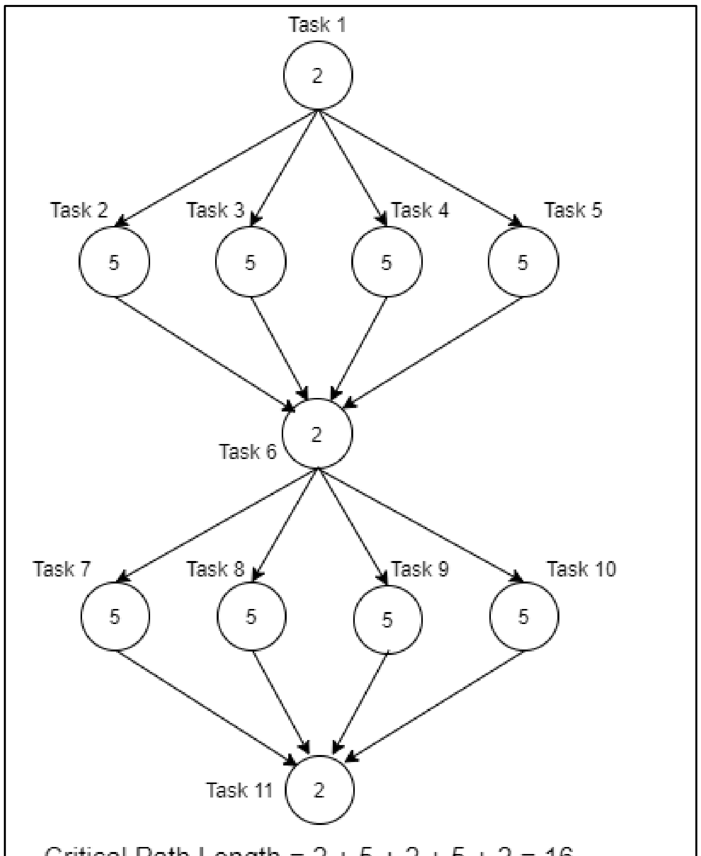
(a) In the sequential version of Sollin's algorithm, for atmost *logV* iterations, we perform a set of operations over the set of edges in the graph. The number of iterations is not known beforehand, and depends completely on the input. This calls for the use of some kind of *dynamic task generation.*

- We notice here that the state of our problem changes after each iteration. (The no. of trees in our disjoint forest keeps reducing with the addition of edges). This means we can use *exploratory decomposition* as we iterate over the states. We terminate when the MST is created (when we only have 1 tree in our forest).
    - Speculative decomposition is not relevant here as the dependencies are well-defined in our case; and at no point do we need to roll-back any already completed tasks.
- In our exploratory decomposition across the iterations, we can incorporate some *input data decomposition* since we know beforehand the set of edges (given as input) that need to be processed in each iteration.
    - This means we will end up with some kind of *hybrid decomposition* (mix of *exploratory* and *data* decompositions).
- At the end of each iteration, we need to update the disjoint forest's state and broadcast it to the other tasks for subsequent computation.

Thus, each iteration will correspond to 2 levels of nodes in our task dependency graph (first level with multiple tasks, each of which work on its allotted set of edges, and another level with 1 task for updating the forest's state).

On testing for np=4, it was found that the average time taken for a task to handle its allotted set of edges was 0.000005sec., while the average time taken for a task to update the forest's state was 0.000002sec

Using the above result we can use weights of 5 and 2 for the first and second levels respectively. Also, it may be noted that the maximum height of our task-dependency graph is *logV*, where V -> no. of vertices.



Critical Path Length = 2 + 5 + 2 + 5 + 2 = 16

Average Degree of Concurrency = 46 / 16 = 2.875

Figure 1: Task Dependency Graph (for 2 iterations)

Task interactions happen between each level of nodes, alternating between MPI_Bcast() operation and MPI_Reduce() operation.

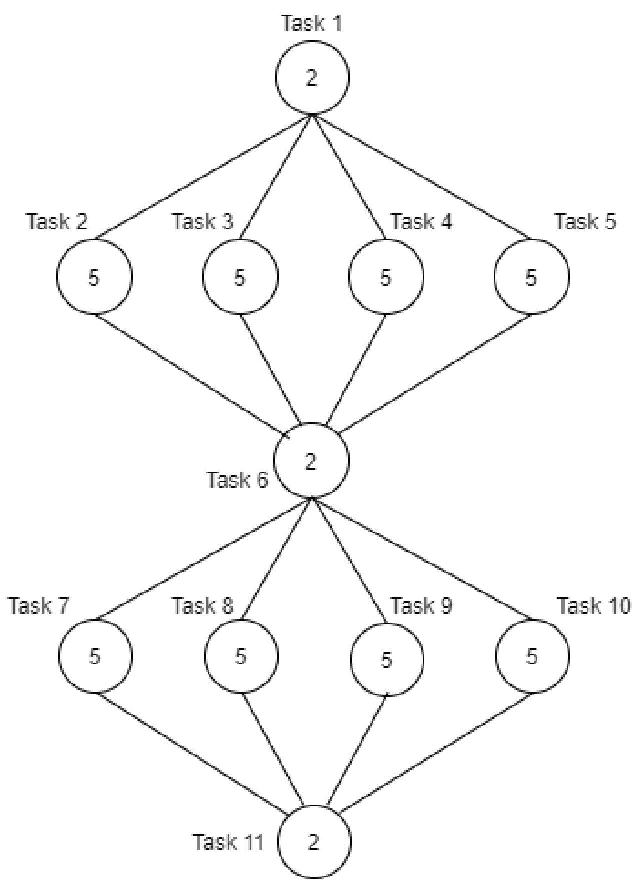


Figure 2: Task Interaction Graph

Even here it may be noted that the manner in which MPI optimizes MPI_Bcast() and MPI_Reduce() is not known to the programmer.

(b) Although the tasks are dynamically generated at runtime, their sizes are known and the interactions follow a regular pattern.

- The tasks are already partitioned based on input-data, so the mapping can also be based on this same data partitioning.
- In our hybrid decomposition scheme, within each iteration we perform static mapping based on input data partitioning; i.e. we can use a simple block distribution for distributing the array of edges across the processes uniformly, since we know that each task has to iterate over all edges and thus has the same cost.
- The single task of combining all results and updating the forest's state is something that cannot be parallelized and needs to be done in a sequential manner (since it involves the Union operation in our disjoin set which needs the edges to be processed sequentially).
- After the end of an iteration, the subsequent tasks can be mapped to the same processes as they were before.

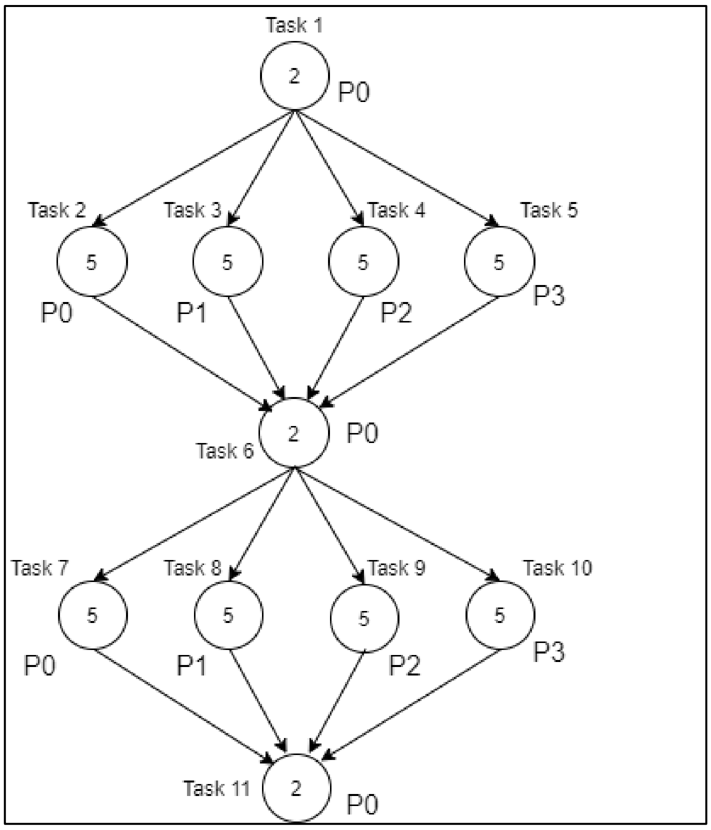


Figure 3: Task-process mapping

Idling that is present during updation of forest state is unavoidable. But for the different tasks in same iteration, load can be evenly balanced across each process using 1 to 1 mapping of the tasks.

(c) The only part that can be parallelized in Sollin's algorithm is the process of iterating over all edges and computing the minimal edge for each vertex that can be added to the MST without creating a cycle. This operation as a whole has to be performed for atmost *logV* iterations in a sequential manner.

Initially our root process initializes the disjoint forest data structure and broadcasts it to all processes.

Subsequently, every process in our algorithm does the following in each iteration:

- Iterate over all edges allotted to it one by one.
- For each edge, check if it is the minimal edge for its source and destination that can be added to the MST without creating a cycle.
- After this apply the reduce operation to compute the final minimal set of edges to be added (handles the case of selecting the least weighted edge for each vertex in case multiple processes select different edges for that vertex).
- The above result is gathered at rank-0 process, which uses it to update the forest's state and broadcasts it back to all the processes so that the next iteration can happen.

(d), (e) The speedup, efficiency and cost were computed for n=4 processes using varying sizes of input (completely connected graphs with no. of vertices varying from 5 to 700). The graphs are shown below.

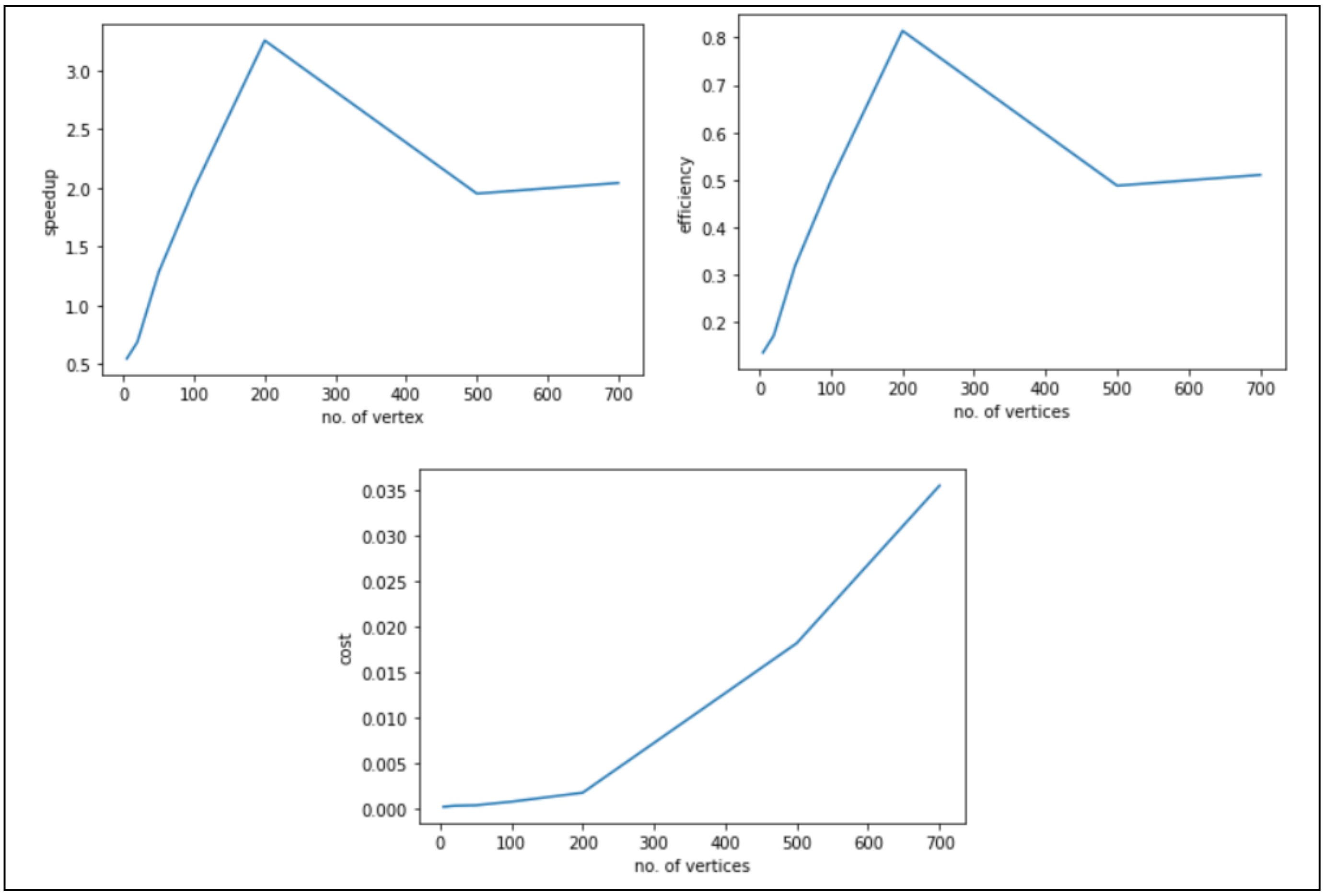


Figure 4: Speedup, efficiency & cost vs input size

- Keeping no. of processes constant, the cost increases with increase in input size as expected.
- The speedup and efficiency peak at around 200 vertices. For bigger inputs, there isn't any increase in speedup and efficiency probably because of the bottleneck of reading large inputs from a file.