Sollin's Algorithm

CS F422: Parallel Computing

Guntaas Singh (2018A7PS0269P)
Harshit Garg (2018A7PS0218P)
K Vignesh (2018A7PS0183P)

March 23, 2021

The Sequential form for the Sollin's Algorithm

The sequential Sollin's Algorithm is given below which is used to find the MST of a given connected input graph:

```
1    procedure SOLLIN(G, V ,E)
2    begin
3        Initialize all the Vertices as their own component.
4        Initialize the MST to be empty
5        While there are more than one components, for each component:
6           Find the smallest weighted edge that connects this component to another component.
7           Add this edge to the MST.
            //JOIN COMPONENTS
8           Combine the two components connected by this edge by giving both the vertices
                the same component name.
            //Cleanup for next iteration.
9           Remove self-loops, all multiple edges b/w two components which is not the smallest one.
10       return MST
11   end SOLLIN
```

It can clearly be seen that at each iteration the problem size at least halves. At each iteration n components would get combined into at least n/2 components.

In the following Document there have been some important assumptions which need to be stated here

- the comparison operation and assignment operation in memory take unit amount of time and work
- When it is stated that a vertex is present in the edge list, it implies that at least one of the edges in the edge list has this vertex as an end-point.

Part A

Task dependency graph

We can granulate the problem into nearly five coarse tasks (the order they are mentioned does not carry any weight whatsoever right now)–

- For every vertex present in the edge lists, find the minimum edge with this vertex as an end-point.
- Find the root or component of each vertex. This task is necessary to avoid cycles.

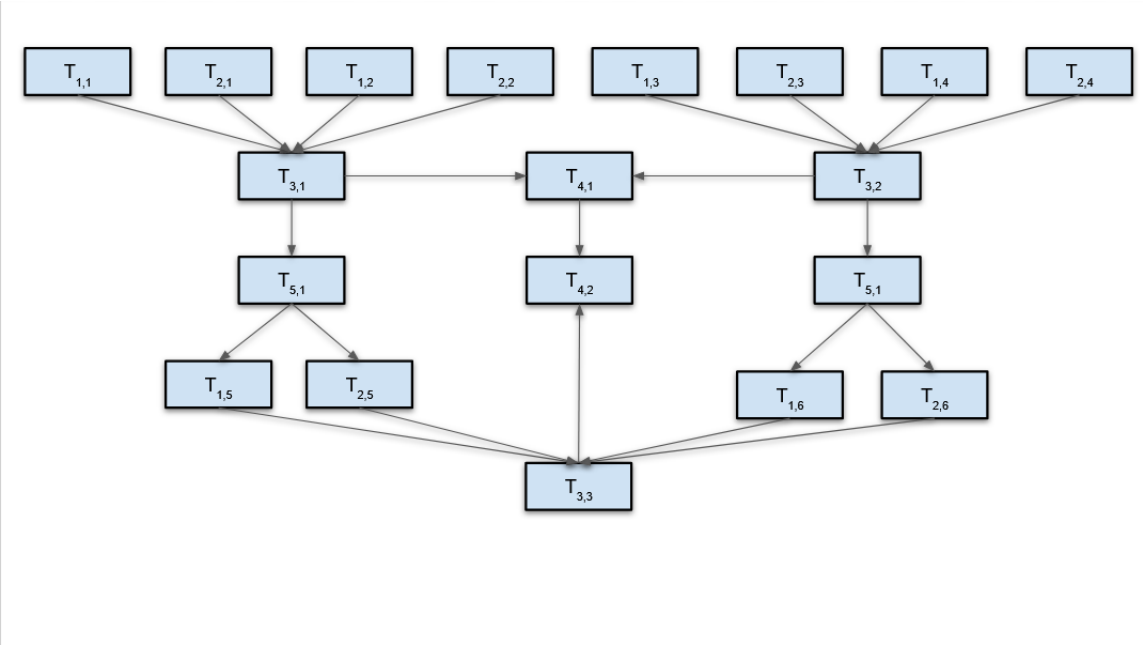


Figure 1: Task Dependency Graph

- Combine components connected by these shortest edges, if not already in the same root or component, then put them in the same root or component making them a single "supervertex" and rename them accordingly.
- Add the required edges into the MST and terminate if number of components is 1.
- Remove multiple edges by removing the non-smallest edges between two components and self-loops in these new components and update the edge list.Start a new iteration with the new edge lists.

Thus, the task dependency graph formed is -

Here the task $T_{i,j}$ defines the task i^{th} as stipulated above being done for the j^{th} time (with a different sets of input mostly).

Average Degree of Concurrency

For the given example in which there were 4 edge-lists parts which needed to be processed,

$$\text{Avg. Deg. of Concurrency} = \frac{TotalTasks}{d}$$

where d is the depth of the dependency graph from leaf to root or final answer. From the graph given above we can see that this evaluates to

$$= (\mathbf{8 + 2 + 3 + 4 + 1 + 1})/\mathbf{6} = \mathbf{19/6} = \mathbf{3.167}$$

And in general we can say that it evaluates for an n-edge system where $n = 2^m$

$$= \frac{6n\text{–}7 + log_2 n}{3log_2(n)} = \frac{6n\text{–}7 + m}{3m} \tag{1}$$

Critical Path length

When we see at the example graph we can clearly see that the longest path has a total length of 5 assuming that each edge weighs one unit. Here one unit can be defined as the work done in finding the minimum edge in an edge list part given to the task which is 1 in case of very fine granularity where each task 1 iteration at leaf level gets only one edge.
And in general we can say that it evaluates for an n-edge system where $n = 2^m$ Task 1(or 2) , 3, 5(or 4) each contribute $log_2(n)$ to the critical path length as they need to be done sequentially always but the last iteration of task 5 never takes place as a single component doesn't need to remove multiple edges and self loops. So the critical path length would be

$$= log_2 n + log_2 n + log_2 n - 1 = 3log_2 n - 1 \tag{2}$$

Task Interaction Graph

When we see at the granulation of tasks. we see that the tasks do not require any data exchange while their execution is going on and only require data when they start or they need to send the output data to another task. Thus the former is dependency of this task from others and the latter a dependency of the graph to others. This implies that the task interaction graph would be exactly the same as the task dependence graph. Thus, the graph interaction graph is .

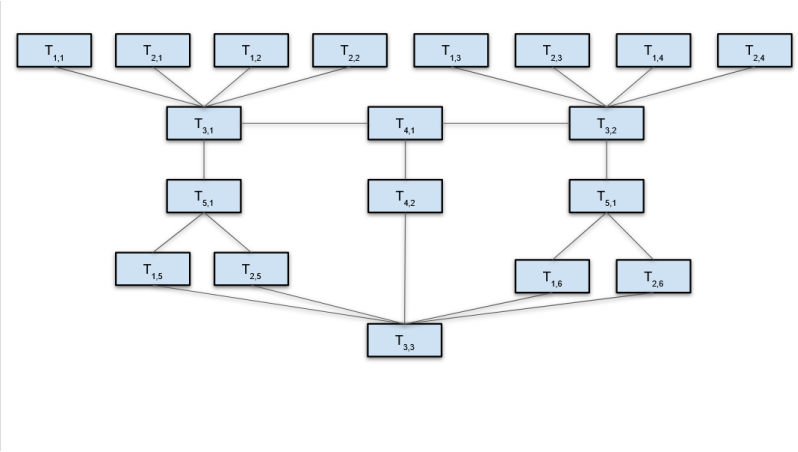


Figure 2: Task Interaction Graph

Part B

Decomposition

The Tasks can be decomposed in several ways into sub-tasks and then be given to processors, Here we would discuss one such decomposition for each task. Here p denotes the max number of sub tasks, E denotes the number of edges and V denotes the number of Vertices

- The first task can simply be input data decomposed into p sub-tasks with each sub-task having E/p edges in its own personal edge list. It was first tried that the decomposition be along the vertices instead of edges, but the communication overhead required to communicate the required data simply made the task decomposition very inefficient. The cost overheads are explained later.
- The second task can also be decomposed identical to task 1 where the owner of the vertex, in either the source or the destination of an edge in its part of the edge list, finds the root according to the edges in its own part of edge list. This implies that multiple sub-tasks may find different roots because of different parts of edge lists
- The third task and all its iteration done in parallel are not decomposed because the decomposition of this task caused us to have a very large amount of communication overhead in this and subsequent tasks because a lot of conflicting data was being passed to other processes.

  An example of this is that a vertex may have different roots on different and mostly unknown processes and thus data needs an AlltoAll Broadcast() after which there is a check to see in which component does this vertex really lie. After this we could send another AlltoAll Broadcast() to send the rectified data.This caused a very large communication overhead (a lot of messages but with low amount, or no amount in case of no conflicts, of data in each). This was also really hard to implement or maintain and thus we decided to do a gather on root and then execute the task 3 there and then do a OnetoAll Broadcast of this data for the fifth task.
- As we had decided that the components would be conflict-resolved and thus edges to be chosen in a single sub-task, it makes sense to also add edges into the MST in a sequential way after task 3 without decomposing this step.
- The task of removing multiple edges and self loops can be decomposed identical to task 1, p sub-tasks with nearly E/p edges each.

Mapping

The mapping of processes and sub-tasks need to be done by keeping two conditions in mind

1. Reduce the communication overhead as much as possible
2. Reduce idling as much as possible.

With keeping these two conditions, the decomposition and task interaction graph in mind I have decided to follow a master and slave model of Parallel algorithm and go with the following kind of mapping for each process

1. Any processor with rank $i > 0$ would be a slave process which would have the part of edge list containing edges indexed from $\frac{E(rank-1)}{p-1} to \frac{E(rank)}{p-1}$ These processes would be executing the task 1, 2 and 5 on their own respective part of edge list and sending the data of task 1 and 2 to the master process for task 3
2. the rank $i = 0$ process would be the master process which resolves the conflicts between components when it receives the data from the different slave processes, All the iterations of task 3 would be done by this master process only,and then sends the rectified data back to them and then later on adds the required used edges to the MST which also would be done by the same process only.

Reasoning

From the dependency graph it can be clearly seen that the task 3 needs to communicate the same data, namely the updated components to task 5 which this would then pass on to task 1 and 2 of the same iteration along with the updated edges by removing the self loops and multiple edges. This implies that there is a significant communication overhead between these tasks which can be totally avoided if all of them are on process which possess the same part of edge list to be used by task 1, 2 and 5. Thus the communication could consist of only of the updated components.That is, the i^{th} process would process task 1, 2, and 5 on the part of edge list containing edges indexed from $\frac{E(rank-1)}{p-1} to \frac{E(rank)}{p-1}$

From the previous decision we also see that the master process has been idle after it resolves the conflicts and gives the updated data to the slaves again in task 3. Thus to reduce this idling I had the master process add edges to MST while the slave process were removing multiple edges and self loops As the time complexity of both of these tasks ($O(V)$ vs $O(E/p)$) is comparable we see that the idling has been reduced significantly.

The Parallel form of Sollin's Algorithm

Considering the mapping and the degree of concurrency found in the previous sections, below is an algorithm for p processor for a connected graph with E edges and V vertices whose MST we need to find.

- For slave process
  1. Receive E/(p-1) edges from the master process
  2. Initialize all the vertices to be in a component 'i' for the i^{th} vertex if not already assigned.
  3. Find the Lightest edge and the component/root of each vertex from the part of edge list that this process has.
  4. Send the data found in step 3) to the processor of rank = 0
  5. Wait to receive the updated components from master and remove any multiple edge or self-loops caused because of the updated components by removing self-loops and all but the lightest of multiple edge between two components.
  6. repeat step 2-5 till all vertices are in the same component.
- For master process
  1. Send edge indexed from $\frac{E(i-1)}{p-1} to \frac{E(i)}{p-1}$ to the rank $i^{th}, i > 0$ processor
  2. Combine the data received from slave computers (of their step 4) to form a single component list by doing the following for each set of components received from a slave-
     (a) If two components , namely i and j, are joint by one of the smallest edges and i has only one vertex then change the component name of it to j. Store this edge in usededges.
     (b) if the components, namely I and j are joint by one of the edges received and this has a weight smaller than the weight by which component i was joint in previous iterations then change component name of all vertex in i to j. Modify usededges to add this edge and remove the previous iteration edge joining i and j.
  3. Broadcast the modified component list to all the slave processes.
  4. Add All the edges in usededges to MST.
  5. Wait for the slave processes to send the new lightest edges and component list and repeat step 2 to 5 till MST has V-1 edges.

This algorithm is initiated by the master process distributing the edges to the slaves and thus there is no circular dependency or deadlock where both master and slave process are waiting.

Possible Improvements and reasons for their absence

Instead of using a single processor for the decomposed sub-tasks of tasks 1, 2,and 5 we could use a an array of $p_{new} \approx E/p_{old}$ processor, considering that it was $E = O(V^2)$ and $p = O(V)$ processor or a total of $p_{total} = p_{old}^2 + 1$ to find the minimum and components in $O(log(E/p))$ instead of $O(E/p)$ time using a parallel search and similar logarithmic improvement for task 2 and 5 too. This would have decreased the time complexity considerably of the parallel algorithm but I could not use this because this would imply that We require a minimum of 5 cores to execute and check speedup of the program for full benefit of the algorithm(1 master, 2 slave for one part of edge list each finding minimum edges in part 1 of edge list , 2 more for the same purpose in the other half of edge list). I do not have a physical system which could check the speed up for such an algorithm and thus this idea was not implemented.

Parallel Time Complexity

The Algorithm can be broken down, for the purpose of finding time complexity, into three sequential parts of which each uses parallelism. Distributing the edges is a simple One to all broadcast thus giving

$$T_d = (t_s + t_w(E))log_2 p. \tag{3}$$

Part 1 is concurrently done by all slave processes where they find the lightest edge and root of all the vertices connected by the edges in their edge list and send this data to master. As this is a linear operation on a part of edge list, the time complexity of this step in worst case is

$$T_1 = E/p + t_s + 2t_w(E/p)). \tag{4}$$

Part 2 is the combining of all components by the master process which is done in linear time with respect to Vertices and the process count. We also send this data to the slaves again through a OnetoAll broadcast (Better time than Scatter). So, the time complexity of this part turns out to be .

$$T_2 = pV + (t_s + Vt_w)log_2 p \tag{5}$$

Last part is of removing the multiple edges and renaming all the vertices and components on the slave. This is a linear process with respect to edge count. The process on slave has a time complexity of O((2*E/p)). The process on master simply adds the edges to a MST in this time which is an O(V) operation, worst case we add all the edges in one go.

It was found that as during the edge finding and root finding part of slave process, we could complete the O(V) operation of adding and thus to prevent idling that was done concurrently with the multiple edge removal and new lightest edge finding. This resulted in the slave process becoming the dominant part in our calculation of time complexity.

$$T_3 = max(E/p, V) = E/p \tag{6}$$

The final time complexity can be given by the summation of all these 3 equations over a depth of $Log_2 e$ added with the distribution time.

$$T_p = (t_s + t_w(E))log_2 p + (E/p + t_s + 2t_w(E/p) + pV + (t_s + Vt_w)log_2 p + E/p)log_2(E)$$

$$T_p = (t_s + t_w(E))log_2 p + (t_S(1 + log_2 p) + \frac{(2E + pVlog_2 p)t_w}{p} + pV + \frac{2E}{p})log_2 E$$

The asymptotic growth function would be $T_p = O((t_s+t_w(E))log_2 p+(t_S(1+log_2 p)+\frac{(2E + pVlog_2 p)t_w}{p}+pV+\frac{2E}{p})log_2 E)$

If we try to do a growth asymptotic analysis while considering that $E = O(V^2)$ and $p = O(V)$

$$T_p = O(\frac{Elog_2(E)log_2(p)}{p})$$

Analysis of the Algorithm

The best sequential code for the MST problem is the kruskal algorithm for $T_s = O(Elog_2(V))$. This would be used a lot in further analysis.

Speed-up

Speed up is defined as the ratio of serial time to that of parallel time.

$$Speedup = \frac{T_s}{T_p} = \frac{Elog_2V}{(t_s + t_w(E))log_2p + (t_S(1 + log_2p) + \frac{(2E+pVlog_2p)t_w}{p} + pV + \frac{2E}{p})log_2E}$$

Doing an asymptotic analysis with earlier iterated equations.

$$Speedup \approx O(p/log_2P) \tag{7}$$

Efficiency

Efficiency is defined as the ratio of speedup to number of processor so

$$E = \frac{Speed - Up}{p} = \frac{\frac{p}{log_2p}}{p} = \frac{1}{log_2p} \tag{8}$$

Cost

Cost of an algorithm is defined as the Parallel time multiplied by the number of processors.

$$Cost = pT_p$$

$$Cost = p(\frac{Elog_2(E)log_2(p)}{p})$$

$$= O(Elog_2(E)log_2(p)) \tag{9}$$

This is not equal to the serial work thus this algorithm is not cost optimal.

Scalability

We can define a new variable $T_o = pT_p$ - T_s

$T_o = O(Elog_2(E)log_2(p))$ - $O(Elog_2(V))$

$T_s = O(Elog_2(V))$

The ratio of these two grows sub-linearly over limits of $p \longrightarrow \infty$ Thus we can say that this parallel system is scalable.

Graphs and Experimental data

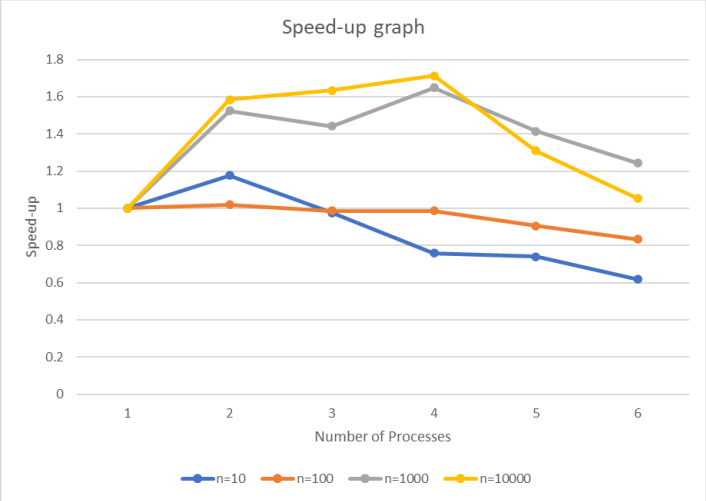
It should be known that the sequential time is not equal to that of time taken by the parallel code with one processor. The time taken by the best sequential code, Kruskal's, algorithm, was used. Some important points to be take into consideration while analysing the table or graph in next section.

- For the sequential and parallel time an average of three tries were taken to determine the value.
- For the sequential time, the code used was of a sequential code of Kruskal's algorithm because that was the best sequential code which gave the least amount of time.

- The code was executed on a 4-core system of intel i5 9300H(Acer Predator Helios 300 Laptop). Thus, we see that the there is a steady speed up till the number of process go till 4 after which the speed-up diminishes and even deteriorates because the communication overhead has increased with increased number of processes but there are no extra processors to compensate for it.

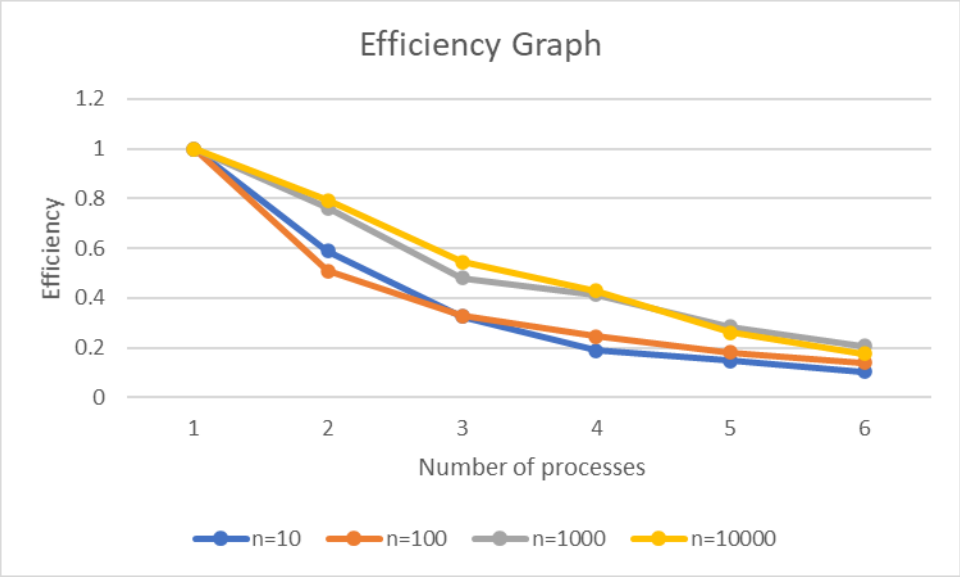
Num of Process	n=10	n=100	n=1000	n=10000
1	0.00012	0.00261	0.042637	0.068305
2	0.000102	0.00256	0.027991	0.04315
3	0.000123	0.002645	0.029591	0.041802
4	0.000158	0.002648	0.025864	0.03989
5	0.000162	0.002881	0.030131	0.052228
6	0.000194	0.003132	0.034297	0.064907

Table 1: Parallel time taken(in sec) for n-vertice graph



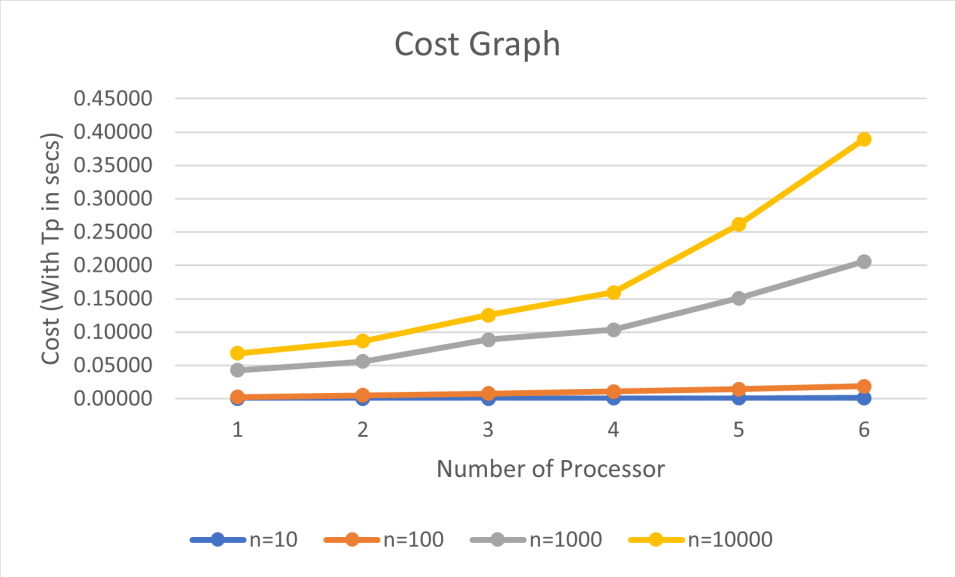
Num of Process	n=10	n=100	n=1000	n=10000
1	1	1	1	1
2	1.176	1.020	1.523	1.583
3	0.976	0.987	1.441	1.634
4	0.759	0.986	1.649	1.712
5	0.741	0.906	1.415	1.308
6	0.619	0.833	1.243	1.052

Table 2: Speed-up for n-vertices graph



Num of Process	n=10	n=100	n=1000	n=10000
1	100.00%	100.00%	100.00%	100.00%
2	58.82%	50.98%	76.16%	79.15%
3	32.52%	32.89%	48.03%	54.47%
4	18.99%	24.64%	41.21%	42.81%
5	14.81%	18.12%	28.30%	26.16%
6	10.31%	13.89%	20.72%	17.54%

Table 3: Efficiency for n-vertices graph



Num of Process	n=10	n=100	n=1000	n=10000
1	0.00012	0.00261	0.04264	0.06831
2	0.00020	0.00512	0.0559	0.08630
3	0.00037	0.00794	0.08877	0.12541
4	0.00063	0.01059	0.10346	0.15956
5	0.00081	0.01441	0.15066	0.26114
6	0.00116	0.01879	0.20578	0.38944

Table 4: Cost or Work for n-vertices graph