Parallelization of Borůvka Algorithm by Spark

MSBD 5003 Big Data Computing
Minimum Spanning Tree – Deep Project Report

Wang, Xiangyu	Chen, Yixue	Liang, Minghui	Yang, Yi
20711306	20732518	20731679	20710742
xwanggb@connect.ust.hk	ychenhz@connect.ust.hk	mliangae@connect.ust.hk	yyangeh@connect.ust.hk

1. INTRODUCE

A minimum spanning tree (MST) is a subset of the edges in a connected, edge-weighted, and undirected graph that connects all the vertices, without any cycles and with the minimum total edge weight. In other words, it is a spanning tree whose sum of edge weights is as small as possible.

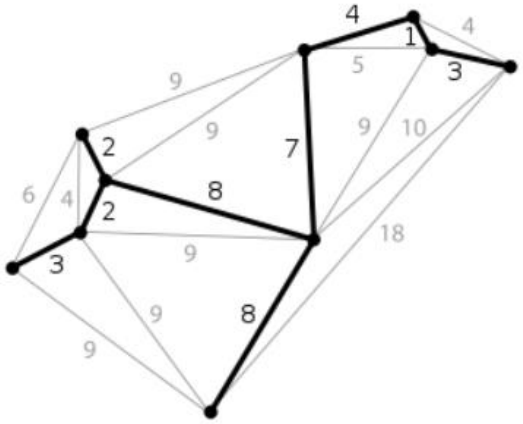

Figure 1: The Minimum Spanning Tree

The minimum spanning tree has many important applications. For example, to lay optical cables between N cities, the main goal is to enable any two of these N cities to communicate, but the cost of laying optical cables is very high, and the cost of laying optical cables varies between cities. Thus, another goal is to minimize the total cost of laying fiber optic cables. This requires finding the minimum spanning tree with weights.

Parallelization plays an important role in the improvement of program performance because the speed of CPU flattens as the transistor count increases. To accelerate the process of finding a MST, this work attempts to implement a parallel version of Borůvka algorithm[1] by PySpark. Then, the performance of different number of spark executors on different size of Erdos-Renyi(ER) random graphs is evaluated by the run-time. Finally, the conclusion is given.

2. RELATED WORK

Many algorithms can be used to compute MST. Prim[2] and Kruskal[3] algorithm are regarded as the most two classic method. However, they are not suitable for parallel programming. On the contrary, an old algorithm published by Borůvka in 1926 (Borůvka algorithm) is easier to parallelize and shown as follows:

```
Borůvka(G(E, V)):
    T = [] # An MST of G, initialized as []
    F = Forest([tree(v) for v in V])
    set the weight of the cheapest edge for each tree in F as ∞
while Ture:
    for (u, v) in E:
        if u and v belong to two different trees in F:
            if w_uv < the weight of the cheapest edge for the tree of u:
                set (u, v) as the cheapest edge for the tree of u
            if w_uv < the weight of the cheapest edge for the tree of v:
                set (u, v) as the cheapest edge for the tree of v
    # For each Tree in F, find the cheapest edge from a node in it to a.
      node outside of it.
    for (u, v) in list(the cheapest edge of each tree in F):
        if u and v belong to two different trees in F:
            Connect these two trees by (u, v)
            Add (u, v) to T.
            If F only have 1 tree left:
                return T
```

Table 1: Borůvka algorithm

Thus, this work selects the Borůvka algorithm for parallel implement, which is expected to significantly reduce the running time when the graph size is huge.

3. METHODOLOGY

3.1. Parallelization by Spark

When finding the cheapest edge for each tree in forest, dividing edges into p partitions and assign them to p executors at first. Secondly, each executor finds the

local cheapest edge for each tree in forest from $\frac{|E|}{p}$ edges. Thirdly, dividing the trees in forest into p partitions and assign them to *p* executors. At last, each executor selects the cheapest edge for each tree in the corresponding partition. The overview is as below shown:

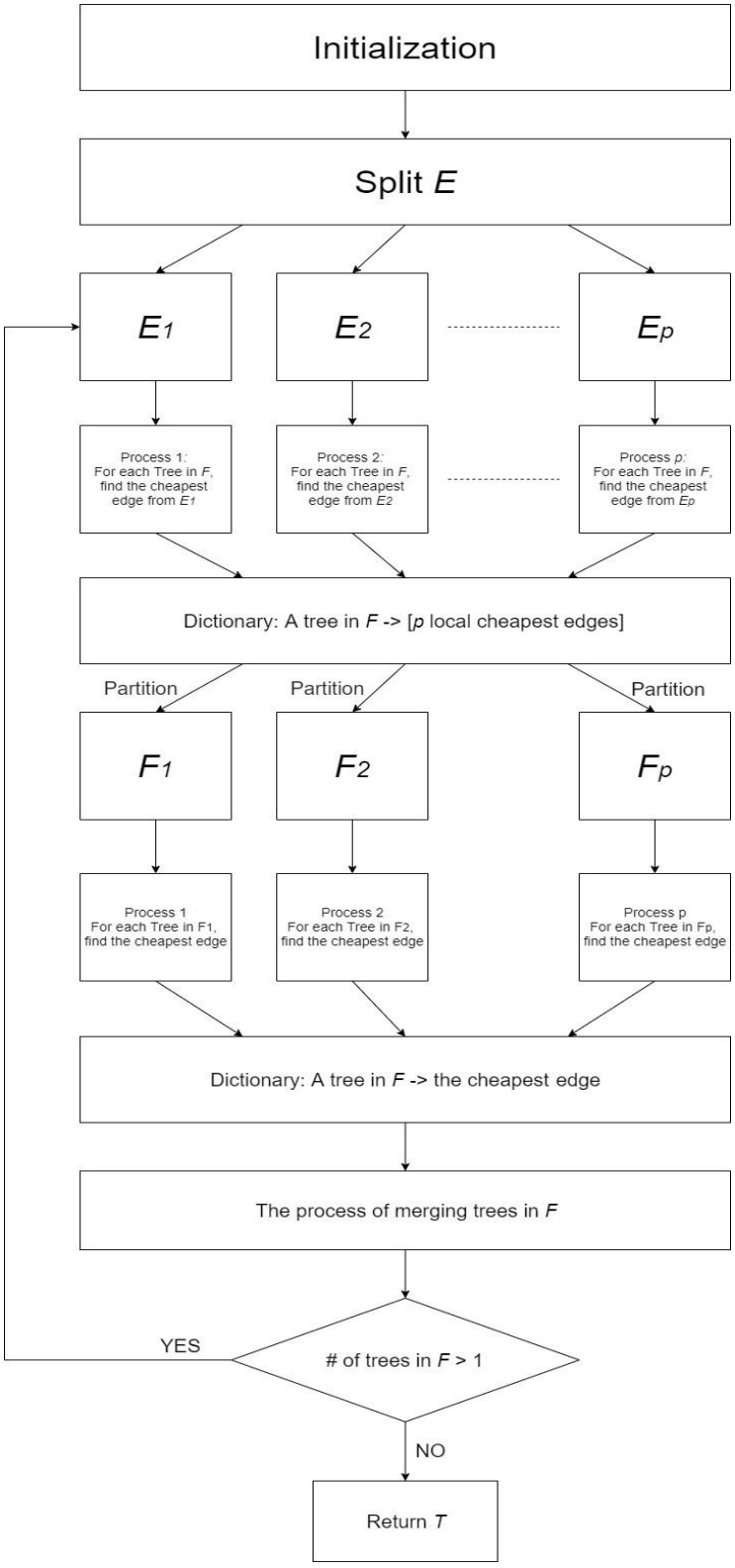

Figure 2: The Parallelization of Borůvka Algorithm

3.2. Optimization

Each tree in the Forest has a root, which identifies the tree. Two nodes belong to the same tree if and only if they share the same root. To reduce the cost of finding the root of a tree, balance and path halving are used.

3.2.1. Balance

When merging two trees, adding the lower one to the higher one to make each tree balanced. As a result, the run-time of check whether two nodes are on the same tree is reduced to $O(\log|V|)$.

3.2.2. Path Halving

Caching the root of vertices after it is found, which makes every node in the path directly point to the root node.

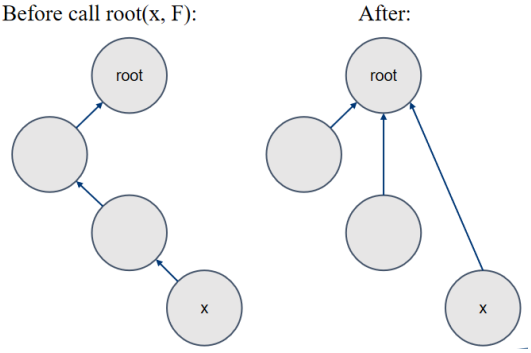

Figure 3: Path Halving

Thus, for single-process Python implement, the cost of checking whether two nodes belong to the same tree is $O(1)$ [4]. However, the forest is a closure in Spark, in other words, each executor caches the root of vertices on its own copy of forest. Thus, the cost is worse than $O(1)$ even if there is only one executor. As the number of executors increases, the cost of finding the root of vertex will be closer to $O(\log|V|)$.

3.3. Run-time Analysis

Let N_t denote the # of trees in F after t while-loops, then:

$$N_{t+1} < \frac{N_t}{2}$$

Thus, the number of loops is at most $\log|V|$. Then, let a denote the run-time of checking whether two vertices belong to the same tree, and p denote the parallelism. For the t-th while-loop, the run-time is:

$$a \times O\left(\frac{|E|}{p}\right) + O(N_{t-1}) = O\left(\frac{a|E|}{p}\right) + O\left(\frac{|V|}{2^{t-1}}\right)$$

For single-process Python implement, p = 1 and a = $O(1)$, so the total run-time is:

$$O(|E|\log|V|) + O(|V|) = O(|E|\log|V|)$$

For spark implement, a = $O(\log|V|)$ and the communication cost is $O(\log|V|\log p)$, so the total run-time is:

$$O\left(\frac{|E|}{p}(\log|V|)^2\right) + O(|V|) + O(\log|V|\log p)$$

As 3.2.2 says, when p increases, the cost of checking whether two vertices belong to the same tree and communication increases. In addition, the updating of T cannot be parallelized. Thus, the speedup of parallelization is predicted to be less than the parallelism p. In other words, the performance converges when the number of executors is big enough.

4. EXPERIMENT

4.1. DATASET

ER random graphs are used. For an ER random graph of N vertices, if the probability that each pair of vertices are connected p is greater than $\frac{\ln N}{N}$, then this graph is connected with the high probability [5]. In this work, 10 ER random graphs as follows are used for experiment.

Graph	N	$p = \frac{2\ln N + 1}{N}$
1	100 000	2.403×10^{-4}
2	200 000	1.271×10^{-4}
3	300 000	8.741×10^{-5}

4	400 000	6.700×10^{-5}
5	500 000	5.449×10^{-5}
6	600 000	4.602×10^{-5}
7	700 000	3.988×10^{-5}
8	800 000	3.523×10^{-5}
9	900 000	3.158×10^{-5}
10	1 000 000	2.863×10^{-5}

Weight of edge = random integer from 1 to 1000

Table 2: ER random graphs for the experiment

4.2. RESULT

Macbook Pro 2019 with 6-core Intel i7(2.6GHz) is used for experiment. The run-time of Borůvka algorithm of single-process python and diferent number of Spark executors on 10 different ER random graphs is shown as below (unit: second):

Size	Python	1 executor	2 executors	3 executors	4 executors	5 executors	6 executors
100 000	6.909645	10.87301	6.869639	5.921837	5.413907	5.323189	5.177478
200 000	17.06008	23.48749	16.65845	14.26385	13.06128	12.42765	12.13227
300 000	26.78964	35.73987	25.31435	21.53227	19.4827	18.3182	17.86619
400 000	39.05842	53.3129	35.80504	29.83924	27.48708	25.49258	24.54813
500 000	65.98864	72.87215	51.80383	42.37993	37.91482	35.19552	34.30122
600 000	72.77661	92.77113	60.71162	50.07953	44.45692	43.34088	38.69196
700 000	91.84878	113.2181	73.62824	59.89156	55.52747	48.16842	45.86183
800 000	101.6704	131.7085	86.99804	72.52439	61.40062	59.56124	53.42683
900 000	117.0658	153.6786	103.3067	80.91528	72.44891	66.24233	61.11932
1 000 000	135.8309	188.5855	131.3851	111.0032	100.4292	96.80093	93.6585

Table 3: The run-time of python and spark

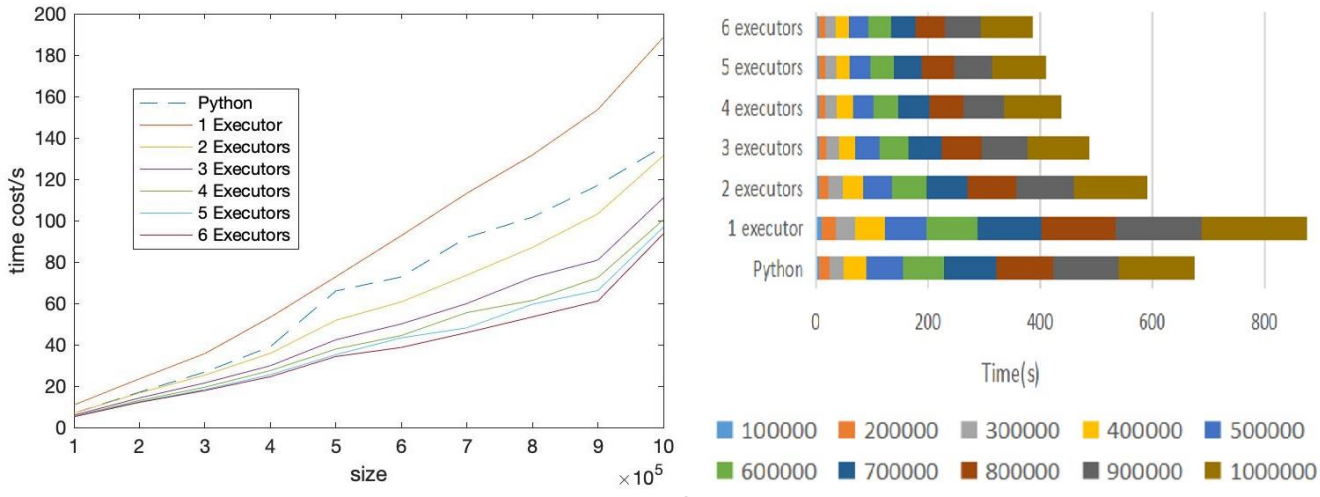

Figure 3: The run-time of python and spark

4.3. DISCUSSION

The run-time of one spark executor is longer than that of single-process Python. In addition, the run-time of spark implement decreases as the number of executors increases but converges when there are more than 4 executors. These results match the expectation as 3.2.2 and 3.3 say.

5. FUTURE WORK

Perhaps sharing the forest to each executor improves the

performance because it halves the path as single-process Python do, which **may** reduce the cost of finding the root of a vertex. However, Spark doesn't provide an interface for the complex shared variables, so this work requires to build a local or remote server to maintain and update the forest, which means the process of identifying a tree cannot be parallelized. In order words, each executer have to wait for the completion of the previous one. As a result, the communication cost will increase, and an unexpected result may occur. Although the implement is complex, we will attempt it in the future.

6. CONCLUSION

This works implements the Borůvka algorithm by Single-Process Python and Spark. The performance of Python and Spark are evaluated by the run-time of getting a minimum spanning tree (MST). In the experiment, different sizes of ER random graphs and different number of executors are attempted. The result shows that the run-time decreases as the number of Spark executors increases but converges when there are more than 4 executors. In the future, we will attempt the shared memory for the potential improvements.

7. ACKNOWLEDGEMENTS

This work is guide by Professor Yi. He recommended the Borůvka Algorithm to us.

This work is division is as follow: Xiangyu Wang is responsible for selecting the topic, coding, and experiment. Yang Yi is responsible for presentation and part of report writing. Yixue Chen and Minghui Liang are responsible for the report writing.

8. REFERENCE

[1] Borůvka, Otakar (1926). " About a certain minimal problem". Práce Mor. Přírodověd. Spol. V Brně III (in Czech and German). 3: 37–58.

[2] Prim, R. C. (Nov 1957), "Shortest connection networks And some generalizations", *Bell System Technical Journal*, 36 (6): 1389–1401

[3] Kruskal, J. B. (1956). "On the shortest spanning subtree of a graph and the traveling salesman problem". *Proceedings of the American Mathematical Society*. 7 (1): 48–50

[4] Tarjan, Robert E.; van Leeuwen, Jan (1984). "Worst-case analysis of set union algorithms". Journal of the ACM. 31 (2): 245–281

[5] Erdős, P.; Rényi, A. (1960). "On the evolution of random graphs". Publications of the Mathematical Institute of the Hungarian Academy of Sciences. 5: 17–61

9. CODE

https://github.com/xyw6/MST.git