Parallel Calculation to find Shortest Path by applying Dijkstra algorithm

Sadig Eyvazov
Institute of Computer Science
University of Tartu
Sadig.Eyvazov@tudeng.ut.ee

Abstract
This paper brings out the usefulness of Parallelizing the Shortest Path algorithm. In this paper, author makes a parallelization of Dijkstra algorithm which is implemented on Apache Spark. Nowadays it can be said that the search algorithms are involved in multiple actions in different areas such as online route planning systems, navigation systems, or timetable information systems and etc. Moreover, these algorithms can also be found in the network where you can discuss the real data transfer speed which is of great importance today. These algorithms should be efficient for running time aspects because searching for information is directly related to such type of algorithms. In this paper, author runs the Dijkstra algorithm in a parallel and sequential manner, then compares the result of them against each other. Overall, the results showed that running the algorithm in parallel is more time-efficient than sequential processing.

1 INTRODUCTION
Nowadays, research on graph theory is getting a wide range of attention with the development of computer and information technology. Therefore, many various numbers of graph structures and algorithms have been proposed [1]. One of the proposed graph algorithms is the Dijkstra algorithm [2] which is a graph search algorithm that finds the single source shortest path for a graph with non-negative edge path costs generating the shortest path tree. In fact, the algorithm calculates not only the shortest path from one vertex to another but also the shortest path tree which shows the shortest path from the source node to any other nodes of the tree.

Real-world networks are mostly very large in size and they consist of hundreds of thousands to billions of edges and vertices. Processing such big graphs take too much time and this is one of the most popular challenges of Computer Science. Thus, there is a need to optimize the graph algorithms for getting more time-efficient algorithm which will produce a better performance on a big graph data. Currently, the serial shortest path optimization reached the time limitation, therefore, one of another possible optimization ways is to run the algorithm in parallel because the sequential operations are not well enough for many big data as it takes more time than parallel computation. To run the algorithm in parallel, the operations should be run at the same time and separately not depending on each other.

In this paper, the Dijkstra algorithm is chosen to be computed in parallel for finding the shortest path tree in graph data and the algorithm is run in Apache Spark Cluster.

Apache Spark breaks the given task into the smaller parts and assigns them to its own executors. So, Spark executes the application in parallel. Spark’s main programming abstraction is resilient distributed datasets (RDD), and Spark RDD is a resilient, partitioned, distributed and immutable collection of data [3].

Spark breaks the RDD into smaller chunks of data which are called partitions. Thus, Spark executes the task in several different partitions of data in parallel and gets the result back from these executions.
In the following sections, we will talk more about the Spark and Dijkstra algorithm.

2 RELATED WORKS
The parallel processing is chosen as an optimization method for the Dijkstra Algorithm. Actually, it can be applied to different graph algorithms.

In the article [4], it has been talked about the parallelization problem of the Dijkstra algorithm. For the parallel implementation of the algorithm, the OpenMP (Open Multi-Processing) and OpenCL (Open Computing Language) are used. Performances of the algorithm are measured in different configurations. The results showed that parallel processing outperforms the sequential processing. However, in general, the results show that the average parallelization speed-up ratio is only 10% because the Dijkstra algorithm is sequential in itself and it is hard to parallelize the algorithm. This article [4] showed the disadvantage of the algorithm, and enhancing its performance would be perfect because the Dijkstra algorithm is widely used.

In [5], the parallelization method applied to the Dijkstra algorithm to see the advantages of parallel computation. Firstly, they transformed the DEM data and made experiments on these data. The traditional Dijkstra algorithm is transformed into parallel form according to the MapReduce programming model. Then, to get the running time and efficiency the different datasets are tested on the Hadoop cluster. As a result, a comparison between the traditional and parallel processing of the Dijkstra algorithm showed that parallel computation achieves more improvement.

They have used MapReduce programming model on different datasets and they made experiments on the Hadoop Cluster to get the run-time efficiency.

In [6] the researchers applied parallelization through the Dijkstra’s algorithm, Bellman-Ford algorithm, Floyd-Warshall algorithm, and Viterbi algorithm. The results of different test experiments are analyzed and talked about the comparison of those searching methods on graph systems. In addition, there was talked about the importance of each algorithm in our real world today mentioning that the Dijkstra algorithm is in the top 3rd place. Although there are explained several strategies for parallel solutions today, in conclusion, the proposed method is not usable because the proposed solutions are better.

The investigation in this paper is more related to the work which is done in [5]. The main goal is to observe how much improvement can be achieved by parallelizing the shortest path algorithm.

3 PARALLEL DIJKSTRA ALGORITHM
The experiments are implemented on a single local machine with an Intel Core i5 processor that has 1.60 GHz speed and 16G of random access memory (RAM) and on a Spark Cluster that consists of 4 virtual machines of 124G RAM and 32 cores each.

The shortest path algorithm that is used in this research is the Dijkstra algorithm. Dijkstra algorithm is a fundamental graph algorithm that allows computing shortest path from a source node to a target node in a directed, weighted graph, for example, the path with the smallest accumulated weight. And, the algorithm does not only compute the shortest path from source to the target vertex but also a shortest path tree which represents the shortest path from a source vertex to any reachable node.

The pseudocode for this algorithm is as follows:

1. for all v in G;
2. d(v) = infinity;
3. D = {s};
4. U = {G-s}; //all other nodes in G
5. For all u adjacent to s:
6. d(u) = w(u, s);
7. while U is not empty {
8. Let v be the node from U with minimum d(v);
9. \( U = U/v; \)
10. \( D = D \cup v; \)
11. For all \( u \) adjacent to \( v \) {
12. If \( d(u) > w(u,v) + d(v) \) then
13. \( d(u) = w(u,v) + d(v); \)
14. }

Figure 1. Pseudocode for Dijkstra Algorithm

The algorithm finds the shortest path between two vertices, source, and target, of the graph. Firstly, the algorithm labels all nodes with the infinity value. Then it chooses the “s” source vertex and puts this vertex into the list \( D \) which has been created as an empty list. And, puts the rest of the nodes into the list \( U \) which will be used during the next stages of the algorithm. All adjacent nodes of the source vertex \( s \) are labeled with the value which is the weight of an edge between the source and adjacent node. While the list \( U \) is not empty, it extracts the node \( v \) with the minimum value from the list \( U \) and puts into the list \( D \). In the next step, it starts to check the weights of the edges between \( v \) and all its adjacent vertices. If the weight of the edge is smaller than the current value(label) of the vertex, it sets this new value to the vertex. Therefore, all these steps are repeated unless all nodes are visited.

The time complexity of the algorithm depends on in which data structure you store the graph data. In our experiments, we use Binary heap for Priority Queue for sequential computing. The overall time complexity will be \( O(\text{E LogV}) \) where \( E \) and \( V \) is number of edges and vertices, respectively. Because the inner loop in the code will be executed \( O(\text{V+E}) \) times and each iteration will run an operation that takes \( O(\text{LogV}) \) time. So, the overall time complexity will be \( O((\text{E+V})*\text{LogV}) = O(\text{ELogV}). \) Worth to note that, if the Fibonacci Heap is used instead of the Binary Heap, then the time complexity of the algorithm will be reduced to \( O(\text{E+VLogV}). \)

In order to implement the parallelization on the Dijkstra algorithm, the Apache Spark is used. Spark extends the MapReduce model and efficiently supports more types of operations including interactive queries, stream processing and etc. Speed is a very important metric in processing the big data as it implies the difference between exploring data interactively and waiting hours. One of the primary advantages of the spark for speed is that it stores the data in memory and makes computations on them but also spark is more efficient than MapReduce running complex applications on the disk. Moreover, the data stored in RDD is fault-tolerant as RDDs can recover from a failure [3].

Spark has its own data structure which is called Resilient Distributed Dataset (RDD). When you create an RDD by loading some data, the spark automatically creates partitions of this data. And, the number of partitions is controllable by the user. If you have 5 partitions, it means that you can achieve 5 parallel processes at the most.

In order to distribute the workload over the executors, the workload needs to be partitioned. The graph data is stored in RDD which is the Spark’s main programming abstraction. The data is partitioned into subgraphs and sent to executors. The Driver process sends the data and application code into the executors, the tasks are run in parallel in 3 nodes (virtual machines), and the results gathered back in Driver node.

As can be seen in Figure 2, firstly, the spark application is started in the Driver node and the Cluster Managers take the data and code in order to distribute across the executors and the
results come back to the Driver node after the parallel execution.

4 RESULTS AND DISCUSSION

For running the algorithm in sequential, the pseudocode is used which is mentioned in Figure 1. For parallel processing, [7] is used. The data is split into several partitions and these partitions are distributed across the Spark Cluster during the parallel processing of the algorithm. After that, the Dijkstra algorithm is applied separately to these subgraphs. The expectation was to get 4 times more efficient results in parallel processing than sequential processing as there are 4 machines in a cluster. Although author could not get the expected result, the result of parallel processes was better than sequential processing. As mentioned in [4], one of the problems of Dijkstra is that it is hard to parallelize this algorithm and it appears that some inter-process interactions happen after splitting the data into subgraphs.

![Figure 3. The results of experiments from running the Dijkstra algorithm in parallel and sequential manner](image)

In Figure 3, it can be seen that the results of the parallel processing and sequential processing do not differ too much at the beginning of experiments when we process the algorithm on a small graph with 50-100 nodes.

As the number of nodes growths, the run time difference between the parallel and sequential processing starts to increase as it can be easily seen in Figure 3.

In this experiment, author has generated 10 different sizes of graphs having nodes ranging from 50 to 2000 by using a graph data generator tool. The algorithm is run 5 times for each different size datasets in a parallel and sequential manner, and these running results are averaged.

The experiments with the data in which the number of nodes is greater than 1000 showed that the parallel processing of the Dijkstra Shortest Path algorithm is really more efficient than sequential processing, approximately 2 times faster.

5 CONCLUSION AND FUTURE WORK

In this paper, author has applied parallelization on the Dijkstra algorithm by implementing the Apache Spark Cluster. The results of the experiments showed that it is possible to decrease the execution run time of the algorithm which is very valuable progress today. During the process of the experiments, it is observed that parallel computation is faster than sequential one and the quantity of efficiency depends on the number executors that you are running your tasks on. As future work, more data sets can be used in order to extend the comparative analysis by implementing the Dijkstra algorithm on Apache Spark to see the efficiency of the parallel and distributed processing.

REFERENCES

[6] Analysis of algorithms for shortest path problem in parallel, Bogdan Popa, Dan Popescu