Date of publication SEP-30, 2025, date of current version JUN-05, 2025.
www.computingonline.net / computing@computingonline.net

Print ISSN 1727-6209
Online ISSN 2312-5381
DOI 10.47839/ijc.19.3.XX

Improved Computing Performance for Floyd-Warshall Algorithm in the MapReduce architectures

NGUYEN DINH LAU[1], LE THANH TUAN[2]

[1]University of Education and Science, The University of Da Nang, Danang City, Vietnam
[2] Master student of Information Systems, University of Education and Science, The University of Da Nang, Danang City, Vietnam

Corresponding author: Nguyen Dinh Lau (e-mail: ndlau@ued.udn).

ABSTRACT The main result of this paper is building a new parallel algorithm based on Floyd-Warshall algorithm to find the Shortest Path for all-pair. The problem of finding All Pair Shortest Path (APSP). APSP is performed on many structures such as: MPI, OpenMP, Cuda and others. When the input data is large, we need to find ways to improve computing power to reduce calculation time for the APSP algorithm. We present a contribution for the APSP in the MapReduce architectures. The idea of this algorithm is to multi Workers to work in parallel by Floyd-Warshall algorithm. There is one Master assigning Map and Reduce. The mapper simultaneously executes their work and sends their data to the reducer until the job is finished. The MapReduce algorithm has two functions: Map and Reduce. They receive key/value pairs based on an adjacency list. The algorithm performs in MapReduce and our results prove that the proposed approach improved Computing Performance for Algorithm Finding the Shortest Path for all-pair. Some fundamental results are systematized and proved.

KEYWORDS shortest path; graph; algorithm; hadoop; mapreduce

I. INTRODUCTION

In the field of graph theory, the problem of finding short paths receives a lot of attention and has many practical applications. There are many different algorithms to find the shortest path such as: Dijkstra algorithm; Bellman-Ford algorithm, the Bellman-Ford algorithm seems superior because it can handle graphs with negatively weighted edges; Floyd's algorithm is an algorithm that finds the shortest distance of all vertices; Floyd-Warshall algorithm is an algorithm to find the shortest path of all vertices.

With the Floyd-Warshall algorithm, the sequential algorithm has complexity equal to $O(n^3)$; n is the number of vertices in the graph (|V|=n), however, in the case of the Hadoop framework, to reduce the computational time of the algorithm when running in parallel on multiple machines, the algorithm has to be modified.

There are many published articles underpinned by Floyd-Warshall algorithm on different parallel environments such as: MPI, OpenMP, Cuda, [15, 16, 18]. With the advent of MapReduce architecture in cloud computing environments, researchers are often interested in this structure because MapReduce architecture is suitable for problems with massive data for the following reasons:

- MapReduce is one of the innovative technologies of the big data revolution, a programming model and tool introduced by Google in 2004.
- Mapreduce can be understood as an execution method to help applications quickly process large amounts of data in a distributed environment.
- These computers execute parallel but independently of each other to reduce processing time.

In papers [3], [4], authors construct parallel all-pairs shortest path algorithm with a MapReduce architecture. Parallel shortest path of an A* algorithm with a MapReduce architecture are implemented in [5], [6], [7], [8]. In paper [9], [10], [11], [12], authors perform parallel data-processing paradigm with Hadoop. In papers [15], [16], [18] authors construct parallel all-pairs shortest path algorithm with MPI, OpenMP, Cuda architecture.

This paper aims at presenting a parallel formulation of an All Pair Shortest Path Problem (APSPP) algorithm: the modified adjacency list based algorithm on MapReduce architecture.

II. HADOOP AND MAPREDUCE

MapReduce on cluster: A cluster has hundreds to thousands of common servers connected to each other via LAN. The hardware stored on these servers does not require high performance, just regular hard disks connected via the IDE standard. The unit of work in the MapReduce program is called a job package (job). Each job has many tasks that are transferred from a common distribution system to servers in the cluster.

The Map task is performed and distributed across storage nodes. The distributed process is performed automatically through the input data being split. The Reduce task is also distributed through intermediate key/value pairs being grouped into pairs with similar keys (Figure 1) [9], [19], [20], [21], [22].

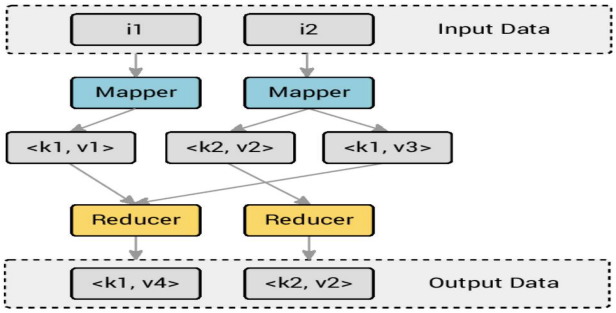

Figure 1. A general model of performed of Mapper and Reducer

MapReduce cluster has a Master node and Worker nodes. The Master node is responsible for managing and regulating Workers (See Figure 2) [20]. Compute nodes are workers that read data from an input file, usually from the block itself stored on a local drive. The Map task generates a set of intermediate key/value pairs stored in internal memory. At configured intervals, intermediate key/value pairs are written to the local hard drive, divided into R groups (R is the number of nodes running the Reduce task). The location of these pairs is notified to the Master server so that the Master is responsible for returning this address to the servers doing the Reduce task.

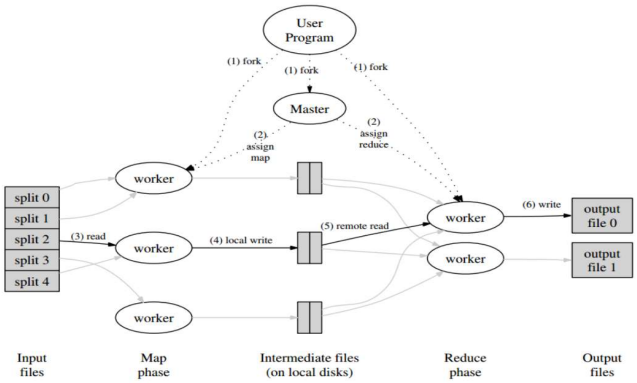

Figure 2. The performance model of Master and Worker

When Workers performing the Reduce task are notified of the location of intermediate key/value pairs, these Workers read the data. When the read is finished, the Worker sorts the intermediate key/value pairs by key. The Worker performs Reduce sequential calculations on the sorted data. The output of the Reduce task is written to the output file. When all Map and Reduce tasks are finished, the Master reports the results back to the user program [23], [24], [25]

III. ALL-PAIR SHORTEST PATH ALGORITHM (APSP)

A. ADJACENCY – LIST

Let graph G=(V, E), $\forall i \in V$, find edges (i,j) $\in$E with weight w(i,j). It maintains an array of list called the Adjacency – List Example 1: An undirected graph with 4 vertices and its adjacency list are shown.

Table 1. Adjacency–List with nodeID (4 vertices)

NodeID (i)	j, W(i,j)	k, W(i,k)
1	2,7	3,5
2	1,7	4,6
3	1,5	4,11
4	2,6	3,11

B. ALL-PAIR SHORTEST PATH PROBLEM ALGORITHM

We use Floyd-Warshall algorithm to find the shortest path between every pair of vertices of a weighted graph G. The matrix D, all-pair shortest distance matrix, can be computed from W, the weighted adjacency matrix. The matrix P, the paths matrix is set with p[i][j]=j for each (i,j). If there is no edge from i to j then p[i][j]=null.

Input: G=(V, E, w), V={1, 2,…,n}, w(i, j) for edge (i, j)
Output: Matrix D=[d(i,j)], In which d(i,j) is the shortest path from i to j $\forall$ (i,j). Matrix P=[p(i,j)], used to determine the shortest path.

The sequential all-pair shortest-path algorithm can be described as the following

Algorithm 1 APSP(W[n][n],P[n][n])

```
1. {
2.     D[n][n]=W[n][n];
3.     for(int k=0;k<=n;k++)
4.       {
5.          for(int i=0;i<=n;i++)
6.                for(int j=0;j<=n;j++)
7.                if(D[i][j]>D[i][k]+D[k][j])
8.                  {
9.                          D[i][j]=D[i][k]+D[k][j];
10.                         P[i][j]=P[i][k];
11.                 }
12.       }
13. }
```

APSP algorithm is correct and the complexity of APSP algorithm is $O(n^3)$

Example 2: An undirected graph with 4 vertices in example 1 has weighted matrix D and path matrix P show

Table 2. Weighted matrix D and path matrix P

Weighted matrix D

	1	2	3	4	
1	inf	7	5	inf	
2	7	inf	inf	6	
3	5	inf	inf	11	
4	inf	6	11	inf	

Path Matrix P

	1	2	3	4
1	null	2	3	null
2	1	null	null	4
3	1	null	null	4
4	null	2	3	null

In which D_k and P_k are matrices obtained from D and P after the k^{th} iteration for k=1,2,3,4. Matrix D_4 determines the length of the shortest path between every pair of vertices while Matrix P_4 identifies the shortest path between every pair of vertices. Matrix D_4 and path matrix P_4 are shown in Table 3.

Table 3. Weighted matrix D4 and path matrix P4

Weighted Matrix D4

	1	2	3	4
1	10	7	5	13
2	7	12	12	6
3	5	12	10	11
4	13	6	11	12

Path Matrix P4

	1	2	3	4
1	3	2	3	2
2	1	4	1	4
3	1	1	1	4
4	2	2	3	2

IV. ALL-PAIR SHORTEST PATH ALGORITHM ON MAPREDUCE

A. IDEA OF MAPREDUCE OF APSP ALGORITHMS

Map stage:

The mapper class takes the entire file input and parses it line by line. Let the consider nodeId be "k". For a vertex "i" adjacent to "k" it emits a new node. For generating the new node, the algorithm iterates through the nodes adjacent to "k" and for each j adjacent to k it sums the distances D(i,k) and D(k,j) and sets D(i,j)= D(i,k) + D(k,j)

"Mark" is a variable to mark the shortest path from "Key" to "j".

Reduce stage:

The output of the mapper will be the input to the reducer class. The reducer class takes the minimum of all the path weights and adds it to the adjacency list of the keyId node.

B. MAPREDUCE APSP ALGORITHMS

Algorithm 2: *APSP Mapper*

Input: Value={∀j| j adjacent NodeID , w(NodeID, j), [Mark]}.

Data representation:

(Key, Value): {NodeID, {∀j| j adjacent NodeID , w(NodeID, j), Mark}.

NodeID is all nodes of the graph (Key, Value): Seperated

key-value pairs denote adjacent nodeId and edge weight connecting them.

"|" separated key-value pair

Output: (Key, Value)

//Emit (Key, Value)

1. *{*
2. For all k in Key (Key, Value) =(key, Value)
3. For all k in Key
4. For all j ∈J| J is set node adjacent k
5. Emit (key, Value)
6. Key: =j,
7. $Value_1$: ={j_1, w(k,j_1), [mark]}
8. $Value_h$={j_h, w(k, j_1)+w(k, j_h), [mark]} with h>1
9. }.

C. DESCRIPTION OF MAPPER ALGORITHM

The input of the Mapper algorithm is the pairs (Key, Value) with the structure: {NodeID, {∀j| j adjacent NodeID , w(NodeID, j), Mark } with Key=NodeID ∈[1…n]|n=|V|. and Value={∀j| j adjacent NodeID, w(NodeID, j), Mark }, The Value includes all nodes j adjacent to NodeID along with the weight w(NodeID, j) of the edge (NodeID, j) and adds the variable Mark to mark the number of vertices that NodeID has passed through. Mark is used to find the path after the Reduce algorithm ends.

- "Mark" is a variable to mark the processed vertices along the path from "Key" to "j"| ∀j| j adjacent Key. Mark=1,2,3…,n. With n being the number of vertices in the graph. If (Key, Value) pairs are not marked (Mark=null), the path from "Key" to "j" is via "j".

- Also for each vertex it maintains a boolean variable "check" to indicate whether that vertex has been used to update distances to its adjacent vertices. If Mark is updated for a particular vertex, the shortest distance from the source vertex to the processed vertex also changes.

Algorithm 3: *APSP* Reducer

---

1. Input: The output of the mapper will be the input to the reducer class
2. Output: (Key, Value)
3. //Emit (Key, Value)
4. For j=1 to |V|
5. $\forall s_i \in S$ with set S= {Key}
6. If (s_i==j) then
7. {
8. Key=j;
9. Value={k\∀k adjacent j, $Min\{w(j,k)\}$, [mark]}
10. Emit (Key, Value)
11. }

/* The reducer class takes the minimum of all the path weights and adds it to the adjacency list of the keyId node. */

12. Sorts the data by keys.

D. DESCRIPTION OF REDUCER ALGORITHM

Keys and Values pairs from the output of the Mapper algorithm become the Input data of the Reducer algorithm. These Keys and Values pairs have many identical Keys. Therefore, the S set contains all the values in the Keys. Note that duplicate Keys are still saved in the S set. The Reducer algorithm choose only a Key from all identical Keys and the Values is created by taking the Min of all the weights of the edges (j,k). and emit the new Key and Value pair.

Then the Reducer algorithm sorts in ascending order based on Keys

E. EXAMPLE

An undirected graph with 4 vertices in example 1, then (Key, Value) is the adjacency list of the graph

Table 4. (Key, Value) is adjacency list of graph

Key	$Value_1$|	$Value_2$|
1	2,7|	3,5|
2	1,7|	4,6|
3	1.5|	4,11|
4	2,6|	3,11|

Output emitted by mapper:

Table 5. Output: (Key, Value)

Key	$Value_1$ |	$Value_2$ |		
1	2,7|	3,5|		
2	1,7|	4,6|		
3	1.5|	4,11|		
4	2,6|	3,11|		
2	1,7|	3,12,mark=1|		
3	1,5|	2,12,mark=1|		
1	2,7|	4,13,mark=2|		
4	2,6|	1,13,mark=2|		
1	3,5|	4,16,mark=3|		
4	3,11|	1,16,mark=3|		
2	4,6|	3,17,mark=3|		
3	4,11|	2,17,mark=3|		

"Mark": It is used to mark the shortest path from "Key" to "j"

Table 6. Input to the reducer class

Key	$Value_1$ |	$Value_2$ |		
1	2,7|	3,5|		
2	1,7|	4,6|		
3	1.5|	4,11|		
4	2,6|	3,11|		
2	1,7|	3,12,mark=1|		
3	1,5|	2,12,mark=1|		
1	2,7|	4,13,mark=2|		
4	2,6|	1,13,mark=2|		
1	3,5|	4,16,mark=3|		
4	3,11|	1,16,mark=3|		
2	4,6|	3,17,mark=3|		
3	4,11|	2,17,mark=3|		

Table 7. The output emitted by the reducer

Key	$Value_1$ |	$Value_2$ |	$Value_3$ |
1	2,7|	3,5|	4,13,mark=2|
2	1,7|	3,12,mark=1|	4,6|
3	1.5|	2,12,mark=1|	4,11|
4	1,13,mark=2|	2,6|	3,11|

The output of the reducer iteration serves as the input for the mapper's next iteration.

Table 8. Output emitted by algorithm Mapper

Key	$Value_1$ |	$Value_2$ |	$Value_3$ |
1	2,7|	3,5|	4,13,mark=2|
2	1,7|	3,12,mark=1|	4,6|
3	1,5|	2,12,mark=1|	4,11|
4	1,13,mark=2|	2,6|	3,11|
2	1,7|	3,12,mark=1|	4,18,mark=1,2|
3	1,5|	2,12,mark=1|	4,18,mark=1,2|
4	1,13,mark=2|	2,20,mark=2,1|	3,18,mark=2, 1|
1	2,7|	3, 18,mark=2,1|	4,13,mark=2|
3	2,12,mark=1|	1,19,mark=1,2|	4, 18,mark=1,2|
4	2,6|	1,13,mark=2|	3, 18,mark=1|
1	3,5|	2,17,mark=3,1|	4, 16,mark=3|
2	3, 12,mark=1|	1,17,mark=1,3|	4, 23,mark=1,3|
4	3,11|	1,16,mark=3,4|	2, 23,mark=3,1|
1	4,13,mark=2|	2, 19,mark=2,4|	3, 24,mark=2,4|
2	4,6|	1,19,mark=4,2|	3, 17,mark=4|
3	4,11|	1,24,mark=2,4|	2,17,mark=4|

Table 9. The final output after all the iterations have been completed

Key	$Value_1$ |	$Value_2$ |	$Value_3$ |
1	2,7|	3,5|	4,13,mark=2|
2	1,7|	3,12,mark=1|	4,6|
3	1,5|	2,12,mark=1|	4,11|
4	1,13,mark=2|	2,6|	3,11|

The results of shortest paths between all pairs of vertices and paths show:

(1,2): shortest distances 7, path 1→2 (mark=null)
(1,3): shortest distances 5, path 1→5
(1,4): shortest distances 13, path 1→ 2 (mark =2)→4
(2,1): shortest distances 7, path 2→1
(2,3): shortest distances 12, path 2→1→3
(2,4): shortest distances 6, path 2→4
(3,1): shortest distances 5, path 3→1
(3,2): shortest distances 12, path 3→1→2
(3,4): shortest distances 11, path 3→4
(4,1): shortest distances 13, path 4→2→1
(4,2): shortest distances 6, path 4→2
(4,3): shortest distances 11, path 4→3

Property 2. Proposed MapReduce of APSP algorithms is correct.

Proof: In Proposed MapReduce of APSP algorithms, we used 4 stage

1. Input stage: Adjacency list of the initial graph
2. Map stage: computation of intermediate (Key, Value) pairs
3. Reduce stage: Takes the minimum of all the path weights and adds it to the adjacency list
4. Output stage: storage of full shortest distance and path.

In the input stage, all the vertices are connected to each vertex on a linked list that is associated with that vertex. Map stage, the sequential all pair shortest-path algorithm by Floyd-Warshow, could be current in this stage at the work the algorithm iterates through the nodes adjacent to "k" and for each j adjacent to k it sums the distances D(i,k) and D(k,j) and sets D(i,j)= D(i,k) + D(k,j). In the Reduce stage, the output of the mapper will be the input to the reducer class, the reducer class takes the minimum (Emit (Key, Value) of all the path weights and adds it to the adjacency list of the keyId node. "Mark" is a variable to mark the shortest path from "Key" to "j". Mark=1,2,3…,n. With n being the number of vertices in the graph. If (Key, Value) pairs are not marked (Mark=0), the path from "Key" to "j" is via "j". Also for each vertex, it maintains a boolean variable "check" to indicate whether that vertex has been used to update distances to its adjacent vertices. If the path weights are not updated, the current distance may not be the shortest path. In the last stage (output stage), the final output after all the iterations have been completed, the master node uploads the full shortest distance and path into the HDFS.

F. COMPLEXITY ANALYSIS AND ASSESSMENT

Complexity of Floyd-Warshall algorithm is $O(n^3)$. The number of assignments D[i][j]=D[i][k]+D[k][j] in algorithm 1 is n^3 because it is in 3 loops. Therefore, the complexity is $O(n^3)$.

The computation time of MapReduce is the total computation time of the Mapper and Reducer.

$$TMR_{FW}=TM_{FW}+TR_{FW}$$

Depending on the divided numbers of blocks of the input graph data, the computation time of Mapper is TM_{FW} and the computation time of Reducer is TR_{FW}. The computation time of Mapper and Reducer varies.

Let N^B be the numbers of blocks, G^S be the input data and Num be the size of each block.

$$N^B = \left\lceil \frac{G^S}{Num} \right\rceil$$

Let C_M be the complexity of the Mapper algorithm, let m be the number of vertices adjacent to the processed vertex j. We

have C_M=O(n.m^2). Because in the Mapper algorithm there are 3 nested loops to Emit (Key, Value) which are lines 3, 4 and 8. In which line 3, 4 and 8 has the number of operations n, m and h (h=m) respectively.

Let C_R be the complexity of the Reducer algorithm, we have C_M=O(n^2.m). Because in the Reducer algorithm there are 3 nested loops to Emit (Key, Value) as lines 4, 5 and 9. In which loops 4, 5 have the same number of calculations n and line 9 has the number of k (k=m).

In the case of a typical graph, the value of m is much smaller than n but we have m=n when the graph is complete.

We get, $\boldsymbol{TM_{FW} = \frac{C_M}{N^B}, and\ TR_{FW} = \frac{C_R}{N^B}}$

Theoretically, the complexity on MapReduce is:

$$\boldsymbol{TMR_{FW} = Max\left\{\frac{C_M}{N^B}, \frac{C_R}{N^B}\right\} = Max\left\{\frac{O(n.m^2)}{N^B}, \frac{O(n^2.m)}{N^B}\right\}}$$

The actual running time in Mapreduce architecture is:

$$\boldsymbol{\frac{O(n.m^2)}{N^B} + \frac{O(n^2.m)}{N^B}}$$

F. DATASET

Random graphs (Figure 3) are created as our database to test the algorithms.

- Input: NumNode, Expansion coefficient.

Example: Input: NumNode=6, Expansion coefficient=3

- Output: Node 1 adjacent Node 2 Node 3 and Node 4; Node 2 adjacent Node 3, Node 4 and Node 5; Node 3 adjacent Node 4, Node 4 and Node 6; Node 4 adjacent Node 5 and Node 6, Node 5 adjacent Node 6. With w(Node i, Node j) is random (see Algorithm 4)

Algorithm 4: A random graph creates an algorithm

Input: NumNode, Expansion coefficient
Output: Graph (Namefile.txt)

```
BEGIN
        ofstream f ("Namfile.txt");
        for(int i=1;i<= NumNode;i++)
          Begin
            f<<i<<"       ";
            for(int j=i+1;j<=i+Expansion coefficient;j++)
                if(j<= NumNode)
                  Begin
                    srand(Munber);
                    int w = rand()
                    Number=Number+1;
                    f<<j<<","<<w<<"|";
                  End;
             f<<endl;
          End;
        f.close();
END.
```

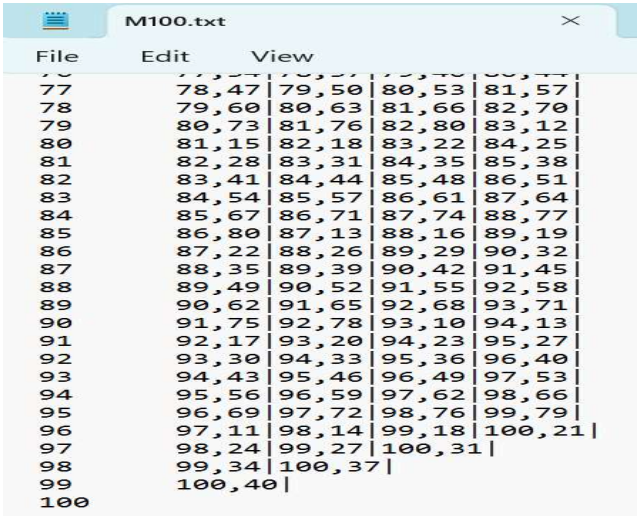

Figure 3. Graph with NumNode =100, EC=4

Figure 3 is the randomly generated input file with 100 vertices and expansion coefficient = 4. With the number of edges 100*4-(1+2+3+4)=390 edges.

We experimentally random graphs nodes as follows: The graph corresponds to 12000 nodes, 71979 edges (Expansion coefficient=6); 17000 nodes, 101979 edges (Expansion coefficient=6) and 22000 nodes, 131979 edges (Expansion coefficient=6). The simulation result demonstrates that the runtime of parallel algorithms in the MapReduce architectures is better than a sequential algorithm.

G. EXPERIMENTAL RESULTS

The performance tests were conducted on nodes Hadoop cluster. The tests have been achieved on Hadoop 3.3.0. All computations have been executed fifteen times and the presented values are the average values of the executions. The graph data covers all types of road networks, and contains weighted edges to estimate the travel distances. The experimental results show that the approach achieves a significant gain in computation time.

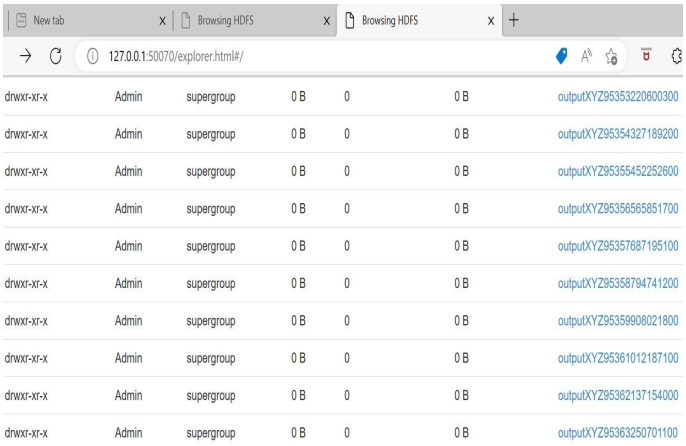

Figure 4. Output file in HDFS

Figure 4 reveals the results of running and saving on the HDFS system.

The simulation result demonstrates that the runtime of parallel algorithms on large graphs is better than small graphs.

Table 10 Illustrates the execution time of graphs on Mapreduce structure.

Table 10. The execution time

Graph	12000 nodes	17000 nodes	22000 nodes
Time	246 mins	319 mins	407 mins

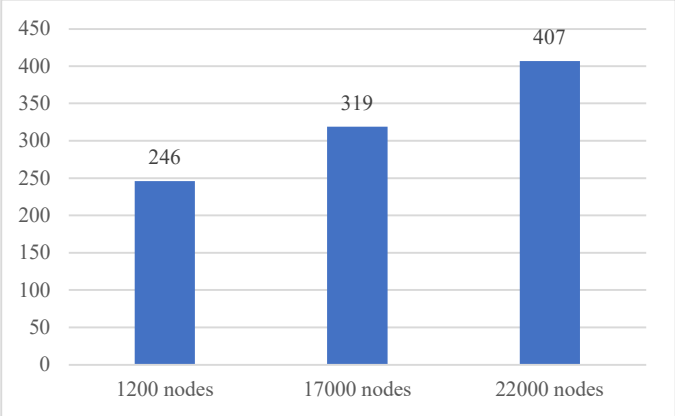

Figure 5. Chart performs the run time of graphs

Figure 5 outlines the result of comparing the execution time of the graphs

V. COMPARISON WITH PREVIOUS STUDIES

In the paper [15], a parallel algorithm was built to find the shortest path of every pair of vertices on the graph on Pipelining. The authors also pointed out that the complexity by the number of steps to send and receive the result is $O(n^2 \log k)$ and the computation time is $O(n^3/k)$. In our algorithm, it is $\frac{O(n.m^2)}{N^B} + \frac{O(n^2.m)}{N^B}$ with m being much smaller n, so the computation time that we have built is significantly reduced.

In the paper [16], the authors built the algorithm on GPU with CUDA structure. This requires the computer to use a NVIDIA Graphic Cards, so it is not convenient and costly as compared to MapReduce. Moreover, in the paper [16], the authors did not compare the various parallel processing blocks with different computation time. In [16], the authors have not proved the complexity of the algorithm.

The paper [18] outlines the designing algorithms on MPI and OpenMP structures but the authors have not run on random graphs to compare input graphs with arbitrary numbers of vertices and edges. While our parallel algorithm experiments on large data sets with randomly generated edges and vertices in the algorithm 4 above.

VI. CONCLUSIONS

This paper presents an innovative approach that exploits a MapReduce framework for APSP algorithm. It is proved that the parallel computing concept is suited to face with APSP algorithm (Property 2). MapReduce APSP algorithms are presented in detail with particular experimental examples. In addition, the basic results are throughly systematized and proved.

This paper presents new parallel algorithms (algorithm 2, 3 and 4) based on the actual requirements, proving soundness. In addition, thesis also does parallelization for existing algorithms, then indicates the advantages of the new ones over previous algorithms.

As part of future work, We will focus on the following tasks:

- Quantify the time complexity taken by MapReduce APSP algorithm for a given graph size.
- Apply of MapReduce APSP algorithm approach on a real road network.
- In the future, we will experiment on a dataset with input of traffic routes in Vietnam and taken from the link: https://download.geofabrik.de/asia/vietnam.html

References

[1] S. H. Roosta, *Paralell Processing and Parallel Algorithm Theory and Computation*, Springer, 2000, 347 p.

[2] N. D. Lau, T. Q. Chien, L. M. Thanh, "Improved computing performance for algorithm finding the shortest path in extended graph," *Proceedings of the International Conference on Foundations of Computer Science (FCS'14)*, USA, 2014, pp. 14-20.

[3] V. Dragomir, "All-pair shortest path modified matrix multiplication based algorithm for a one-chip MapReduce architecture," *U.P.B. Sci. Bull., Series C*, 78, 4, pp. 95-108, 2016.

[4] V. Dragomir, G. M. Ștefan, "All-pair shortest path on a hybrid Map-Reduce based architecture," *Proceedings of the Romanian Academy, Series A, the publishing House of the Romanian Academy*, vol.20, no. 4, pp. 411–417, 2019.

[5] S. Aridhi, V. Benjamin, P. Lacomme, L. Ren, "Shortest path resolutionusing hadoop," *MOSIM'14*, Nancy – France, 2014.

[6] W.Y.H. Adoni, T. Nahhal, B. Aghezzaf, A. Elbyed, "The MapReduce-based approach to improve the shortest path computation in large-scale road networks: the case of A* algorithm," *Journal of Big Data*, vol. 5, p. 16, 2018. https://doi.org/10.1186/s40537-018-0125-8.

[7] W.Y.H. Adoni, T. Nahhal, B. Aghezzaf, A. Elbyed, "MRA*: Parallel and distributed path in large-scale graph using MapReduce-A* based approach," In: Sabir, E., García Armada, A., Ghogho, M., Debbah, M. (eds) Ubiquitous Networking. UNet 2017. *Lecture Notes in Computer Science*, vol 10542, 2027. Springer, Cham. https://doi.org/10.1007/978-3-319-68179-5_34.

[8] S. Aridhi, P. Lacomme, L. Ren, B. Vincent, "A MapReduce-based approach for shortest path problem in large-scale networks," *Journal of Engineering Applications of Artificial Intelligence*, vol. 41, pp. 151–165, 2015. https://doi.org/10.1016/j.engappai.2015.02.008.

[9] Apache Hadoop, *Welcome to Apache Hadoop*. [Online]. Available at: http://hadoop.apache.org/

[10] S. Ghemawat, H. Gobioff, S.T. Leung, "The Google file system," *ACM SIGOPS Operating Systems Review*, vol. 37. New York: ACM; 2003. p. 29–43, 2003. https://doi.org/10.1145/1165389.945450.

[11] J. Dean, S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun ACM*, vol. 51, issue 1, pp. 107–13, 2018. https://doi.org/10.1145/1327452.1327492.

[12] V.K. Vavilapalli, Sıo Seth, Bıo Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, E. Baldeschwieler, A.C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, "Apache Hadoop YARN: yet another resource negotiator," *Proceedings of the 4th ACM Annual Symposium on Cloud Computing*, Santa Clara: 2013, pp. 1– 16. https://doi.org/10.1145/2523616.2523633.

[13] M. Hena, N. Jeyanthi, A Three-Tier, "Authentication scheme for kerberized Hadoop environment," *Cybernetics and Information Technologies*, vol. 21, no. 4, pp. 119-136, 2021. https://doi.org/10.2478/cait-2021-0046.

[14] D. Petrosyan, H. Astsatryan, "Serverless high-performance computing over cloud," *Cybernetics and Information Technologies*, vol. 22, no. 3, pp. 82-92, 2022. https://doi.org/10.2478/cait-2022-0029.

[15] H. Wang, L. Tian, and C.-H. Jiang, "Practical parallel algorithm for all-pair shortest path based on pipelining," *Journal of Electronic Science and Technology of China*, vol 6, no 3, 2008.

[16] D. Kulkarni, N. Sharma, P. Shinde, V. Varma, "Parallelization of shortest path finder on GPU: Floyd-Warshall," *International Journal of Computer Applications*, vol. 118, no. 20, 2015. https://doi.org/10.5120/20858-3547.

[17] S.-C. Han, F. Franchetti, and M. Puschel, "Program generation for the all-pairs shortest path problem," *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2006, pp. 222–232. https://doi.org/10.1145/1152154.1152189.

[18] Z. Yan, Q. Song, "An implementation of parallel Floyd-Warshall algorithm based on hybrid MPI and OpenMP," *Proceedings of the International Conference on Electronics, Communications and Control*, 2012, pp. 2461-2466.

[19] K. Kalia, N. Gupta, "Analysis of Hadoop MapReduce scheduling in heterogeneous environment," *Ain Shams Engineering Journal*, No 12, pp. 1101-1110, 2021. https://doi.org/10.1016/j.asej.2020.06.009.

[20] J. Dean, S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM January*, vol. 51, no. 11, pp. 107-113, 2008. https://doi.org/10.1145/1327452.1327492.

[21] X. Xiaobing, B. Chao, and C. Feng, "An insight into traffic safety management system platform based on cloud computing," *Procedia - Soc. Behav. Sci.*, vol. 96, pp. 2643–2646, 2013. https://doi.org/10.1016/j.sbspro.2013.08.295.

[22] S.N. Khezr, N.J. Navimipour, "MapReduce and its applications, challenges, and architecture: A comprehensive review and directions for future research," *Journal of Grid Computing*, vol. 15, no. 3, pp. 295–321, 2017. https://doi.org/10.1007/s10723-017-9408-0.

[23] N. S. Naik, A. Negi, and V. N. Sastry, ''A review of adaptive approaches to MapReduce scheduling in heterogeneous environments," *Proceedings of the 2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, Delhi, India, 2014, pp. 677-683, https://doi.org/10.1109/ICACCI.2014.6968497.

[24] M. W. ur Rahman, N. S. Islam, X. Lu, D. Shankar, and D. K. Panda, ''MRAdvisor: A comprehensive tuning, profiling, and prediction tool for MapReduce execution frameworks on HPC clusters," *Journal Parallel Distributed Computing*, vol. 120, pp. 237–250, 2018. https://doi.org/10.1016/j.jpdc.2017.11.004.

[25] H. Singh, S. Bawa, "A MapReduce-based scalable discovery and indexing of structured big data," *Future Generation Computer Systems*, vol. 73, pp. 32–43, 2017. https://doi.org/10.1016/j.future.2017.03.028.

NGUYEN DINH LAU *Born in 1978 in Dien Ban, Quangnam, Vietnam. He graduated from Maths_IT faculty of Hue university of science in 2000. He got master of science (IT) at Danang university of technology and hold Ph.D Degree in 2015 at Danang university of technology. His main major: Applicable mathematics in transport, parallel and distributed process, discrete mathemetics, graph theory, grid Computing and distributed programming.*

LE THANH TUAN *Born in 1978 in Hai Chau, Danang, Vietnam. He graduated of IT at Danang University of Technology. His main major: Parallel and distributed process, discrete mathematics, graph theory, grid Computing and distributed programming.*