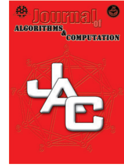
Journal of Algorithms and Computation

journal homepage: http://jac.ut.ac.ir



Negative Cost Girth Problem using Map-Reduce Framework

Mahboubeh Shamsi*[1], Abdolreza Rasouli Kenari†[1] and Roghayeh Aghamohammadi‡[1]

[1]Faculty of Electrical and Computer Engineering, Qom University of Technology

ABSTRACT

On a graph with a negative cost cycle, the shortest path is undefined, but the number of edges of the shortest negative cost cycle could be computed. It is called Negative Cost Girth (NCG). The NCG problem is applied in many optimization issues such as scheduling and model verification. The existing polynomial algorithms suffer from high computation and memory consumption. In this paper, a powerful Map-Reduce framework implemented to find the NCG of a graph. The proposed algorithm runs in $O(\log k)$ parallel time over $O(n^3)$ on each Hadoop nodes, where n, k are the size of the graph and the value of NCG, respectively. The Hadoop implementation of the algorithm shows that the total execution time is reduced by 50% compared with polynomial algorithms, especially in large networks concerning increasing the numbers of Hadoop nodes. The result proves the efficiency of the approach for solving the NCG problem to process big data in a parallel and distributed way.

Keyword: Map-Reduce, Parallelism, Negative Cost Girth, Sequential, Message Passing Interface

ARTICLE INFO

Article history:
Research paper
Received 3, August 2021
Received in revised form 23, November 2021
Accepted 30 November 2021
Available online 30, December 2021

*shamsi@qut.ac.ir
†Corresponding author: A. Rasouli Kenari. Email:rasouli@qut.ac.ir
‡aghamohammadi.r@qut.ac.ir

AMS subject Classification: 05C78.

1 Introduction

Graph algorithms are a large group of widely used algorithms in computer networks, such as social nets. Due to their high degree polynomial time, when the graphs become large, the processing time will increase. Therefore, parallelism is an efficient approach to increase the speed up of the graph algorithms [12].
One of the most challenging graph algorithms is the Negative Cost Girth (NCG) problem. This problem is well-studied in operational research and theoretical computer science. It is also applied in different applications such as constraint-solving, real-time scheduling, and program verification [8, 11, 14, 6, 16, 18, 4]. Based on the definition in [23, 22], The Negative Cost Girth (NCG) problem is defined as the number of edges of the negative cost cycle or the negative cost cycle with the fewest number of edges in a weighted directed graph.
The rest of the paper will continue with related work in section 2, the preliminaries in section 3 including the problem formulation and the Map-Reduce programming paradigm, our proposed algorithm and two examples are described in section 4, the analytical results in section 5 continues with the experimental results in section 6, and it will end with a final conclusion.

2 Related Work

Several studies investigate finding NCG using different approaches. The proposed approaches are classified into two major groups, including sequential and parallel algorithms. The first polynomial time algorithm proposed in [19] was based on a dynamic programming method. They trace path cost increment from each node to itself to find the NCG. The time complexity of this algorithm is estimated $O(n^3. \log k)$ where n is the number of vertices and k is the value of NCG.
Moreover, edge progression algorithm and edge relaxation algorithm have been proposed in [20]. They use matrix multiplication to find path length from each node to itself and check that the negative cost cycle exists or not. The difference between the two algorithms is their different methods to extend paths. In the edge progression algorithm, all paths of incoming vertices y to x should be identified to find the shortest path between u and x including paths from u to y and y to x. In the edge relaxation algorithm, it will relax each edge to find the shortest non-positive cost path between each pair of vertices. While the time complexity of edge progression algorithm is $O(n^2.k + m.n.k)$, edge relaxation algorithm is $O(m.n.k)$ where m is the number of edges and n is the number of vertices in the graph and k equals to the value of the NCG [20]. The comparison results indicate that both edge progression and edge relaxation algorithms perform better compared to [19] in sparse graph. Furthermore, the proposed algorithm in [19] is superior to both algorithms

in [20] for a dense graph. Eventually, in all cases, the edge relaxation algorithm works better than edge progression. It is noted that all three algorithms terminate when the first negative cost cycle is found.
Moreover, a divide and conquer algorithm is proposed for the NCG problem in planar networks which runs in $O(n^{1.5k})$ when k is the NCG length[22]. The algorithm is based on the well-known LiptonTarjan planar separator theorem [22].
In addition to sequential algorithms, parallel algorithms have been developed to solve the NCG problem. In [24], the focus is on finding all elementary negative cycles. It provides a heuristic algorithm based on divide and conquers framework producing sub-problems to solve the main problem. It shows the reasonable CPU time and more efficiency than some proposed recursive algorithms with increasing the number of vertices up to some hundred vertices.
In [16], there are two fast randomized algorithms for finding the shortest negative cost cycle in a directed and weighted graph. It proposed two randomized algorithms using some bits to indicate the indices of vertices and will permute them randomly to find out all shortest paths using k edges from vertices in at least probability $O(1-\frac{1}{e})$. Therefore, the first proposed algorithm runs in $O(m.n.\log n)$ time in at least probability of $(1-\frac{1}{e})$, in a network using n vertices and m edges. The second proposed algorithm reduced the randomized bits. It runs in $O(m.n.\log n)$ time using $O(n)$ random bits. Both of these algorithms are faster than the proposed algorithm in [20].
In [9], a crossover-based algorithms is applied. They mentioned that using repairing mechanisms or appropriate selection strategies can speed up the running process. The result showed that, repairing infeasible offspring improved time complexity to $O(n^{3.2}.(\log n)^{0.2})$. Also, selecting parents that guarantee feasible offspring will improve the time complexity to $O(n^3 \log n)$. The most important characteristic of the evolutionary algorithms is their parallelism capabilities.
In [23], a parallel implementation has been proposed using Message Passing Interface (MPI) which runs in $O(\log k. \log n)$ time on $O(n^3)$ processors. It uses a matrix multiplication approach for the NCG problem. It modified the sequential algorithm which is explained in [19], to run in parallel mode.
Furthermore, three type algorithms for the NCG problem including the All-pair approach proposed in [23], the Single-Source approach proposed in [22], and the Randomized approach proposed in[16] are compared in [2]. It is addressed that the Randomized approach could be superior to others for all graph classes.
All the related works studied are summarized in Table 1.

2.1 Motivation and Contribution

Due to the enormous growth of computer networks and related graphs, the use of parallel algorithms has a higher priority than sequential algorithms. Moreover, Map-Reduce is considered an efficient model of parallelism which is easy to use even for programmers without experience with parallel and distributed systems. Since the Map-Reduce model hides the details of parallelism, as well as enhances fault-tolerance, locality optimization,

Table 1: Summary of related works

Ref, Year	Applied Method	Complexity
[24] , 2002	Heuristic algorithm - divide and conquers	-
[19] , 2009	Dynamic Programming	$O(n^3. \log^k)$
[20] , 2013	Edge progression	$O(n^2.k + m.n.k)$
[20] , 2013	Edge relaxation	$O(m.n.k)$
[9] , 2013	Crossover-based algorithms	$O(n^{3.2}.(\log n)^{0.2})$
[23] , 2015	Message Passing Interface	$O(\log k. \log n)$ on $O(n^3)$ processors
[22] , 2015	divide and conquers	$O(n^{1.5k})$
[16] , 2018	randomized fast algorithms	$O(m.n. \log n)$ with probability $(1 - \frac{1}{e})$
[16] , 2018	reduced randomized bits	$O(m.n. \log n)$ on $O(n)$ random bits

and load balancing, it is applied as an effective approach in a large variety of problems [10, 15, 5, 13]. Therefore, Map-Reduce is used as a powerful model to find the NCG, effectively.
The main contribution of this paper is an efficient algorithm based on a Map-Reduce programming approach using completing and modifying the matrix multiplication to find all shortest paths between vertices. We focus on proposing a Map-Reduce based parallelism for discovering the NCG problem. It is implemented in the Hadoop environment and its performance is analysed and compared.

3 Preliminary

In this section, the problem formulation, Map-Reduce, and Hadoop are described. Also, all preliminaries which are used in this paper are explained.

3.1 Problem Formulation

In this section, the Negative Cost Girth problem is explained as proposed in [23]. Additionally, all assumptions used in this algorithm are presented. It is noted that the procedure and assumptions are employed as the basis of the proposed algorithm using the Map-Reduce method.
Suppose that a network is defined by $G =< V, E, C >$ where V and E denote vertex-set $\{v_1, v_2, ..., v_n\}$ and edge set $\{e_{ij} | 1 \leq i, j \leq n\}$, respectively. Besides, C defines a cost function $(C : E \to Z)$ which is an integer weight to each edge between i and j vertices. Moreover, the adjacency matrix of a graph G is denoted by $A = (a_{ij})$ where $a_{ij} = C(e_{ij})$ for existing edge between vertices i and j, otherwise $a_{ij} = \infty$.
In addition to the principal definition of the network graph, matrix $D^{(k)} = d^k_{ij}$ is the cost of the shortest path between each pair of vertices passing at most k edges where

$1 \leq k \leq n$. Note that, $d^1_{ij} = 0$ if $i = j$, $d^1_{ij} = a_{ij}$ for an edge $e_{ij} \in E$ between i and j and otherwise $d^1_{ij} = \infty$.
The proposed algorithm is performed recursively by computing the value of d^k_{ij} from the value of d^{k-1}_{ij}. To do so, each path breaks into two sub-paths including

1. The shortest path from v_i to v_r using at most $(k-1)$ edges, where v_r is the neighbour of v_j
2. and the edge from v_r to v_j.

It is formulated as equation 1 [23]:

$$d^k_{ij} = \min_{(1 \leq r \leq n, r \neq j)} (d^{(k-1)}_{ir} + a_{rj}) \quad (1)$$

where a_{rj} denotes the cost of the edge from v_r to v_j. It should be noticed that the cost of a self-loop equals 0. In fact, the number of edges will be increased in each iteration and the cost likewise will be raised. It checks the shortest paths of any vertices to itself, if there is a negative cost then the NCG is found [23]. After detecting the first negative cost cycle, the algorithm should be terminated. If the process continues, the shortest path crosses from some vertices or edges twice or more, thus the path will not a simple path. Note that the simple path is defined as the path which does not comprise any repeated vertices. In fact, to compute the shortest simple path correctly, any NCG algorithm should be terminated after detecting the first negative cost cycle.
Based on the explained algorithm in [23], to reduce the running time of the proposed algorithm, square power of matrix $D^{(2)}, D^{(4)}, ...$ are computed instead of using all power of matrix $D^{(k)}$. In the case of detecting NCG in $D^{(l)}$ matrix, NCG equals l at most and $l/2 + 1$ at least due to executing the binary search. A binary search will explore the interval $[l/2 + 1, l]$ for finding the smallest value of r that $D^{(r)}$ has a negative cost cycle.

3.2 Map-Reduce

Map-Reduce introduced by Google is a paradigm for programming and an architecture model for parallel implementation in distributed computation systems [10]. In this model, data is divided into smaller sections and it is allocated to the different nodes in distributed systems to perform an independent sub-process on them. In fact, Map-Reduce divides the problem into sub-problems to solve them in the parallel model leading to enhanced processing speed up. Therefore, Map-Reduce is useful for big data problems that could be implemented in a parallel way [3].
Map-Reduce assigns computational tasks to several machines communicating over the network through exchanging messages among them. The input dataset should be distributed among these machines. [21]. The algorithm will perform in several rounds (sometimes called a job). Each job is consisting of two main phases, Map, and Reduce [21]. After completing the Map phase, the Reduce phase will be started. To do so, the output of the Map phase is considered as the input of the Reduce phase. After computing the Reduce phase, the final outputs will be produced.

3.2.1 Map phase

In the Map phase, the computation task takes place among different machines through exchanging data among them. It is noted that each machine with local storage produces a list of key-value pairs (k, v) extracted from the input of its local storage. The values of k is numeric and the v includes other information, respectively [21].
It is supposed that a list of key-value pairs is produced by all machines in the Map phase. Key-value pairs with the same keys like $(k, v_1), (k, v_2), ..., (k, v_n)$ will be sent to the same machine in the reduce phase [21].

3.2.2 Reduce phase

Note that network communication should not be considered in this phase resulting in performing calculations on the local storage of each machine. Each machine should combine the key-value pairs of the previous phase into its local storage leading to processing on the local data. After completing the Reduce phase of each machine, the current iteration will be finished [21].
After performing each round, the local storage of each machine will be delivered to a Distributed File System (DFS) as a disk on the cloud which never fails. It is due to improving the robustness of the system in the case of failing machines during running algorithm by replacing the failed machine with the other machine. To do so, by requesting a new machine, the previous load on the old storage is transferred to the new storage. Therefore, the current round will be implemented again in the case of failing the old machine [21].

3.3 Hadoop

Hadoop is an open-source software framework that implements distributed processing big data on some clusters of servers. This framework is based on Java language using for distributed processing on thousands of machines that are robust against failures [7].
The main framework of Hadoop includes below modules[17]:

- Hadoop commons part: They consist of necessary libraries and utilities of other Hadoop modules.
- Hadoop Distributed File System (HDFS): It is a distributed file system which stores data on the cluster machines makings wide bandwidth
- Hadoop YARN: It is a platform for resource management to handle the computation resources in the clusters.
- Hadoop Map-Reduce: It is a programming model for data processing on a large scale.

Eventually, Hadoop makes a distributed file system storing data on several servers and distributing tasks among them. It is noted that Hadoop is a more well-known open-source implementation for Map-Reduce.

4 The proposed algorithm based on Map-Reduce

This section explains the parallel implementation of the proposed algorithm using Map-Reduce for solving the NCG problem. Our implementation is like the Message Passing Interface (MPI) implementation, proposed in [23]. But our algorithm uses a parallel implementation based on Map-Reduce. To do so, we should define the input file. After defining the structure of the input file, we should perform two main phases including Map and Reduce to reduce running time.
The input file is made with the adjacency matrix. Each record inside the file includes three items, the row id, the column id, and the matrix value corresponds to the row and column. The record is represented as (i, j, a_{ij}) in the input file.
In the Map phase, key-value pairs will be produced from the input data. This process is implemented in Algorithm 1. As shown in Algorithm 1, for each record (i, j, a_{ij}) in an input file, we produce two key-value pairs marking with "1" and "2". The first key-value pairs, built in line 5 of Algorithm 1. The key is an ordered pair (i, z) where i is the row number of matrix, and $z = 1, 2, 3, ..., n$. Here, n is the number of Matrix columns. Value is built as $(1, j, a_{ij})$ which is marked by "1". j and a_{ij} comes from input file as mentioned above. Furthermore, the second key-value pairs , shown in line 6 of Algorithm 1, is extracted by the column number of matrix (j, z) as the key and value of $(2, i, a_{ij})$ marking by "2".

Algorithm 1 Map Phase Algorithm.

1: Inputs: Adjacency Matrix as the input file;
2: Outputs:set of (key,value)
3: **for** each element of input matrix (i, j, a_{ij}) **do**
4:     **for** z = 1,2,3,.. to the size of input square matrix(n) **do**
5:         produce (key, value) pairs as $((i, z),\ (1, j, a_{ij}))$
6:         produce (key, value) pairs as $((z, j),\ (2, i, a_{ij}))$
7:     **end for**
8: **end for**
9: **return** set of (key,value) pairs that each key has a list with values $(1, j, a_{ij})$ and $(2, i, a_{ij})$

After producing key-value pairs, we classify them into distinct groups corresponding to the same keys. In fact, key-value pairs with the same key are collected in the same group. In the next step, the extracted first key-value pairs in which their column numbers are equal to the row numbers of the second key-value pairs are transmitted to the reduce phase which is presented in Algorithm 2. To do so, we classify each produced group of the Map phase into two main lists named list1 and list2 according to the marking values. Therefore, list1 includes key-value pairs marking with "1" and list2 consists of items starting with "2". Then, these two lists beginning with 1 and 2 should be sorted, depicted in line 5, 6 of Algorithm 2.
After creating two sorted lists, the value of cells form list1 and list2 with the same j are added, shown in line 7. j value indicates the column numbers in value cells of list1 and row numbers in value cells of list2. It is meaning that cell values in which the row numbers in list1 are equivalent to the column number of value cells in list2 should be added. It is similar to what happened in the matrix multiplication. Then, as shown in

line 8 of Algorithm 2, it finds minimum value among all the results of the last step (sum of values). Therefore, it scans all paths using at most k edges and finds the minimum cost paths. k is a number that is a power of 2 and related to the round number of running Map-Reduce phase. In the last step, it is necessary to check the diameter of the output matrix for finding a negative cost girth. So, if the row number of matrix (i) is equal to the column number of matrix (j) and the minimum value is less than zero, a negative cost cycle is successfully detected which is illustrated in line 9 of Algorithm 2.

Algorithm 2 Reduce Function Algorithm.

```
1: Inputs: list1 includes key-value pairs marking with "1" and list2 consists of items starting with "2";
2: Outputs:(key, minimum value);
3: Initialization:
4: for each key do
5:     sort values begin with 1 by j in list1
6:     sort values begin with 2 by i in list2
7:     sum of a_ij and a_jk values (column number in list1 = row number in list2)
8:     find minimum of sum results
9:     if i = j and the minimum value < 0 then found = true
10: end for
11: return (key, minimum value)
```

For running the program, first, a job is created. After reading the input file, the Map and Reduce functions are executed. If no negative cost girth is found, a new job is created by reading the output of the previous Reduce function, and then, Map and Reduce functions will be executed recursively. In other words, the output from a Reduce function will be sent as input to the next Map function. It means that after completion of any Map and Reduce functions, a new matrix that contains the length of the shortest paths using at most k edges will be created. This iteration will be terminated when the first negative cost cycle is detected.

When a negative cost cycle using l length (l is a power of 2) is detected, we need to know the exact value of the NCG. We remember that the number of edges used in the NCG is at most l and at least $l/2+1$. Therefore we search this interval by the binary search to find the precise value of the NCG.

Algorithm 3 explains how a binary NCG search is used to find the correct value of k. In the initialization phase, variables are set in lines 4 and 5. In the while loop, a new job is created.

Note that after first iteration, there are two inputs for Binary NCG Search algorithm and we should combine them to create one input file for Map phase in line 8. Two inputs are combined and sent to the Map function. In each round, $D^{(low)}, D^{(\frac{k}{r})}$ are sent to Map and Reduce functions as inputs, thus the shortest path between each pair of vertices is found using at most $(low+\frac{k}{r})$ edges through $D^{(low+\frac{k}{r})}$ matrix. Note that low, k, r variables are changed and prepared in each round. $D^{(low)}, D^{(\frac{k}{r})}$ are matrices including the shortest path between each pair of vertices using at most low and $(\frac{k}{r})$ edges, respectively. These matrices are produced in the previous Map-Reduce iteration.

In other words, the search continues in the lower or upper half of the interval, recursively, eliminating the other half from consideration if there is no NCG in that half.

Algorithm 3 Binary NCG search algorithm.

```
1: Inputs: D^(low),D^(k/r);
2: Outputs: k;
3: Initialization:
4: found = false; high = k; low = k/2; r = 4;
5: mid = (low + high) / 2;
6: while true do
7:     Create job with inputs: D^(low),D^(k/r) ;
8:     Do Map-Reduce;
9:     if "Negative Cost Cycle is found" then
10:        found = true;
11:    end if
12:    if found = true then
13:        if mid is even then
14:            high = mid;
15:            r = 2 * r;
16:            mid = (low + high) / 2;
17:            found = false;
18:        else
19:            return(mid);
20:        end if
21:    else
22:        if mid is even then
23:            low = mid;
24:            r = 2 * r;
25:            mid = (low + high) / 2;
26:        else
27:            return(mid + 1);
28:        end if
29:    end if
30: end while
```

After completion of Map and Reduce phases in each round, it checks if the NCG is found and mid is an even number then k is equal to mid, or if mid is an even number and the NCG is not found, then k equals to mid+1. The conditions are shown in Algorithm 3 in lines 12 to 29.
Fig 1 shows the Map and Reduce phases in the proposed Algorithms. First, there is a loop for finding the interval of the NCG. Then, a binary NCG search begins that uses the previous outputs to search the range and to find the smallest value of the NCG.

4.1 Two examples

In this section, two examples are provided to comprehend the process of the proposed algorithm. For the first example, it is assumed that the adjacency matrix is given as follows:

$$A = \begin{bmatrix} 0 & 1 \\ -2 & 0 \end{bmatrix}$$

The Map phase produces key-value pairs that values are grouped based on the keys. As depicted in Table 2, different groups are indicated by specified colors. Therefore, the list of a different group is identified as follows:

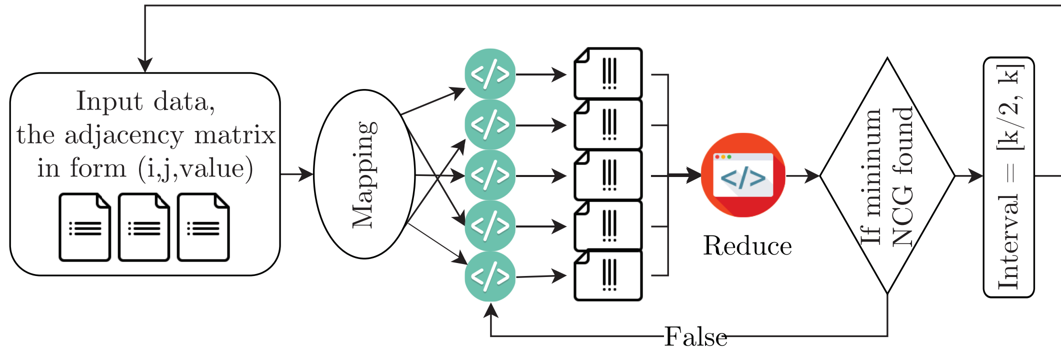


Figure 1: Map and Reduce phases in the proposed algorithm based on Map-Reduce

$$\begin{cases} (1,1): & (1,1,0),(2,1,0),(1,2,1),(2,2,-2) \\ (1,2): & (1,1,0),(2,1,1),(1,2,1),(2,2,0) \\ (2,1): & (2,1,0),(1,1,-2),(2,2,-2),(1,2,0) \\ (2,2): & (2,1,1),(1,1,-2),(1,2,0),(2,2,0) \end{cases} \tag{2}$$

Table 2: KEY-VALUE pairs produced by Map Function

KEY	VALUES
(1,1)	(1,1,0)
(1,1)	(2,1,0)
(1,2)	(1,1,0)
(2,1)	(2,1,0)
(1,1)	(1,2,1)
(1,2)	(2,1,1)
(1,2)	(1,2,1)
(2,2)	(2,1,1)
(2,1)	(1,1,-2)
(1,1)	(2,2,-2)
(2,2)	(1,1,-2)
(2,1)	(2,2,-2)
(2,1)	(1,2,0)
(1,2)	(2,2,0)
(2,2)	(1,2,0)
(2,2)	(2,2,0)

The produced key-values of the Map phase is classified into two main lists defined by list1 and list2 according to the marking values as depicted in Table 3 step 1. Therefore, list1 and list2 include key-value pairs marking with “1” and “2”, respectively.

Table 3: Example 1 execution step by step

STEP 1

$(1,1)$	$j=1$	$j=2$
$list1$	$(1,0)$	$(2,1)$
$list2$	$(1,0)$	$(2,-2)$
	$0+0=0$	$1+(-2)=-1$
	$\Rightarrow Minimum=-1$	

STEP 2

$(1,2)$	$j=1$	$j=2$
$list1$	$(1,0)$	$(1,1)$
$list2$	$(2,1)$	$(2,0)$
	$0+1=1$	$1+0=1$
	$\Rightarrow Minimum=1$	

STEP 3

$(2,1)$	$j=1$	$j=2$
$list1$	$(1,-2)$	$(2,0)$
$list2$	$(1,0)$	$(2,-2)$
	$-2+0=-2$	$0+(-2)=-2$
	$\Rightarrow Minimum=-2$	

STEP 4

$(2,2)$	$j=1$	$j=2$
$list1$	$(1,-2)$	$(2,0)$
$list2$	$(1,1)$	$(2,0)$
	$-2+1=-1$	$0+0=0$
	$\Rightarrow Minimum=-1$	

It means that each key indicates (row, column) numbers, and items in list1 are in (column, value) form and items in list2 are in (row, value) form produced in the previous step. So, it is extensible the form (row, column, value) in each step using each presented key and value separately.
For instance, Fig 2 shows that there is a cycle with considering cost between each pair, which is illustrated in equation Table 3 step 1.
We conduct the same process for other groups to find the minimum value in each group, which is shown in Table 3 steps 2-4.
The output of the Reduce phase for key-value pairs of the example are as below:

$1,1,-1$
$1,2,1$
$2,1,-2$
$2,2,-1$

that contains a negative number in the diagonal. We can represent it as a matrix. They are the shortest path between each pair of vertices using at most 2 edges. So, the NCG

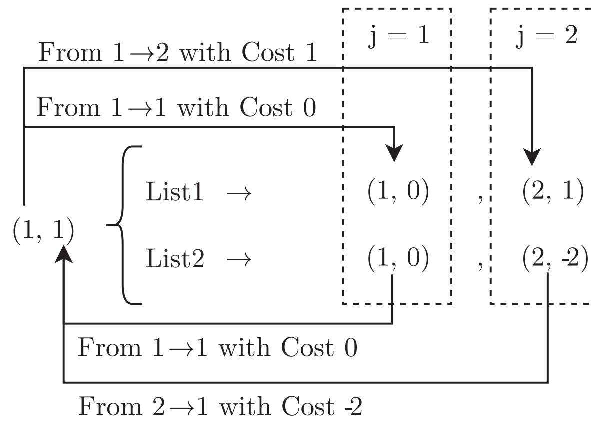


Figure 2: Details of equation 3

is equal to 2 which is shown in the matrix form.

$$A = \begin{bmatrix} -1 & 1 \\ -2 & -1 \end{bmatrix}$$

In the second example, Fig 3 shows a graph sample with a negative cost cycle. In this graph, the NCG of the network is 3; the fewest edges that make a negative cost girth is equal to 3. The cycle corresponding to the NCG of each network is indicated with dashed lines.

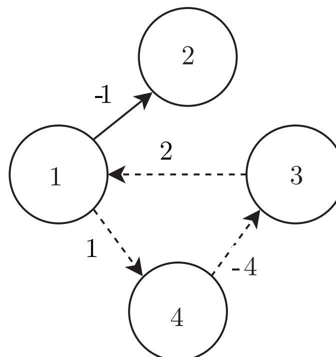


Figure 3: A sample graph with negative cost girth

First, an adjacency matrix is created for this graph, and then according to this matrix, an input file is provided for the algorithm. The initial adjacency matrix is formed as follows:

$$A = \begin{bmatrix} 0 & -1 & \infty & 1 \\ \infty & 0 & \infty & \infty \\ 2 & \infty & 0 & \infty \\ \infty & \infty & -4 & 0 \end{bmatrix}$$

Table 4: The input and output of each iteration

The initial input	The output of the first round The input of the second round	The output of the second round	The output of the binary search
(1,1,0)	(1,1,0)	(1,1,-1)	(1,1,-1)
(1,2,-1)	(1,2,-1)	(1,2,-2)	(1,2,-1)
(1,3,∞)	(1,3,-3)	(1,3,-3)	(1,3,-3)
(1,4,1)	(1,4,1)	(1,4,0)	(1,4,1)
(2,1,∞)	(2,1,∞)	(2,1,∞)	(2,1,∞)
(2,2,0)	(2,2,0)	(2,2,0)	(2,2,0)
(2,3,∞)	(2,3,∞)	(2,3,∞)	(2,3,∞)
(2,4,∞)	(2,4,∞)	(2,4,∞)	(2,4,∞)
(3,1,2)	(3,1,2)	(3,1,1)	(3,1,1)
(3,2,∞)	(3,2,1)	(3,2,1)	(3,2,1)
(3,3,0)	(3,3,0)	(3,3,-1)	(3,3,-1)
(3,4,∞)	(3,4,3)	(3,4,3)	(3,4,3)
(4,1,∞)	(4,1,-2)	(4,1,-2)	(4,1,-2)
(4,2,∞)	(4,2,∞)	(4,2,-3)	(4,2,-3)
(4,3,-4)	(4,3,-4)	(4,3,-5)	(4,3,-4)
(4,4,0)	(4,4,0)	(4,4,-1)	(4,4,-1)

The initial input which is produced according to the adjacency matrix is presented in the first column of Table 4. It contains row number, column number and value of each cell; in the form $(i, j, value)$.

After creating a job and performing the first Map and Reduce phases, the output would be produced which contains all the shortest paths using at most 2 edges, shown in the following matrix. Although the output of the first round is depicted in $(i, j, value)$ form, the matrix is presented to understand more clearly.

$$A = \begin{bmatrix} 0 & -1 & -3 & 1 \\ \infty & 0 & \infty & \infty \\ 2 & 1 & 0 & 3 \\ -2 & \infty & -4 & 0 \end{bmatrix}$$

In this step, the matrix will be checked to identify if a negative number exists in the diagonal, it means there is a negative cost cycle in the graph. In the output matrix of our example, there is no negative number on the matrix diagonal. Therefore, The input file of the second round which was the output of the first round in form $(i, j, value)$ is shown in the second column of Table 4.

The new job will be created and Map and Reduce phases are performed again for the second iteration. The input of the second Map-Reduce phases is considered as the output

produced by the first Map-Reduce round. By performing the second round, all the shortest paths using at most 4 edges are found. The output of the second iteration is presented in the form of matrix as follows:

$$A = \begin{bmatrix} -1 & -2 & -3 & 0 \\ \infty & 0 & \infty & \infty \\ 1 & 1 & -1 & 3 \\ -2 & -3 & -5 & -1 \end{bmatrix}$$

Moreover, it can be seen that there exist negative values on the diagonal of the output matrix. So, a negative cycle is found in this step and iteration will be terminated. The output of the second round is shown in the $(i, j, value)$ form which is depicted in the third column of Table 4.

The NCG of the network is underlined in the output of the second round. This output shows the shortest path between each pair of vertices using at most 4 edges. That is a power of 2 meaning that the NCG is at most 4 and at least 4/2+1. In fact, we can use binary search in the interval $[4/2+1, 4]$ to find the smallest value of the NCG. Therefore, based on the binary NCG search, the shortest paths using at most 3 edges should be found.

The inputs of the first round of the binary NCG search algorithm are $D^{(2)}$, $D^{(1)}$, that they are matrices produced by previous Map-Reduce tasks. During this round, all the shortest paths using at most 3 edges are found. Now the NCG is found and the binary NCG search is terminated. The output of the proposed binary search for $D^{(3)}$ is as follows:

$$A = \begin{bmatrix} -1 & -1 & -3 & 1 \\ \infty & 0 & \infty & \infty \\ 2 & 1 & -1 & 3 \\ -2 & -3 & -4 & -1 \end{bmatrix}$$

Therefore, the output of the proposed binary NCG search (it will be in $(i, j, value)$ form) for $D^{(3)}$ is presented in the last column of Table 4.

5 Analytical results

In this section, the analytical performance of the proposed algorithm based on Map-Reduce is conducted to compare with the nave sequential algorithm (A Hadoop with only one node) and MPI.

To analyse the performance, the total runtime is estimated by two phases as described in equation 3 and equation 4:

$$Total\ Map\,time = \frac{(n^2 * z)}{nodes} \tag{3}$$

$$Total\ Reduce\,time = \frac{(n^2 * z)}{nodes} \tag{4}$$

Where n is the number of graph nodes and z is the number of columns or rows of the adjacency matrix, as well as *nodes* is the number of Hadoop nodes. It is concluded that the total job execution time is computed as equation 5:

$$Total_Time = r * (Map_Time + Reduce_Time) \tag{5}$$

where r is the number of iterations for finding the final NCG value.
It is assumed that there are n lines in the input file in the job function. Each line of the input file should be read and be sent to the Map function. The Map function includes a loop with size z, which is equal to the number of columns or rows of input square matrix; it is clear that the value of z is equal to n value. Thus the total Map time is shown as equation 6:

$$Map_Time = \frac{(n^2 * n)}{nodes} \rightarrow \frac{n^3}{nodes} \tag{6}$$

where *nodes* is the number of Hadoop nodes which is variable.
After completing the Map phase, the Reduce phase will capture any mapped item, and there are n^2 keys which are received from the Map phase; so the Reduce function will repeat capturing operation as n times to find similar j in two lists. By the end of the receiving operation, it will calculate and summarize the captured items. The other elements of the Reduce function are the same as the Map function.
The total job execution time is equal to the time consumed for the Map and Reduce phases. Execution of the job will repeat r rounds. In fact, the binary NCG search function will produce the jobs r times. To be more specific, r determines the times of creating and running a new job until finding the NCG.
For the first time, r is the power of 2 and continues to 4,... and then it repeats job-creating until finding the smallest value for the NCG by the binary search; each one of these operations has $O(\log n)$ time order at most. So, the value of r equals to $2\log n$ at most. Therefore, we considered the maximum r value as $2\log n$.(See equation 7)

$$(2\log n)(\tfrac{n^3}{nodes} + \tfrac{n^3}{nodes}) \rightarrow (2\log n)(\tfrac{2n^3}{nodes}) \rightarrow (\tfrac{4n^3\log n}{nodes}) \tag{7}$$

For more precise analysis, to find the NCG wherein it crosses k edges, at the first, $2\log k$ rounds will be iterated to find the NCG which progresses by the power of 2 matrices and after it. After that, the binary NCG search will be executed $\log k$ rounds to find the smallest value of k in the interval $[k/2+1, k]$. It will be deduced that finding the NCG is faster in the smaller value of k because it is clear that lower rounds will be iterated to find the NCG and algorithm will be terminated earlier.
So, we have in equation 8:

$$(2\log k + 1)(\frac{2n^3}{nodes}) \rightarrow O(\log k)(\frac{2n^3}{nodes}) \tag{8}$$

In order to compare the effect of k on time complexity, we should define the following scenario. Fig 4 depicts the number of performed operations versus number of vertices

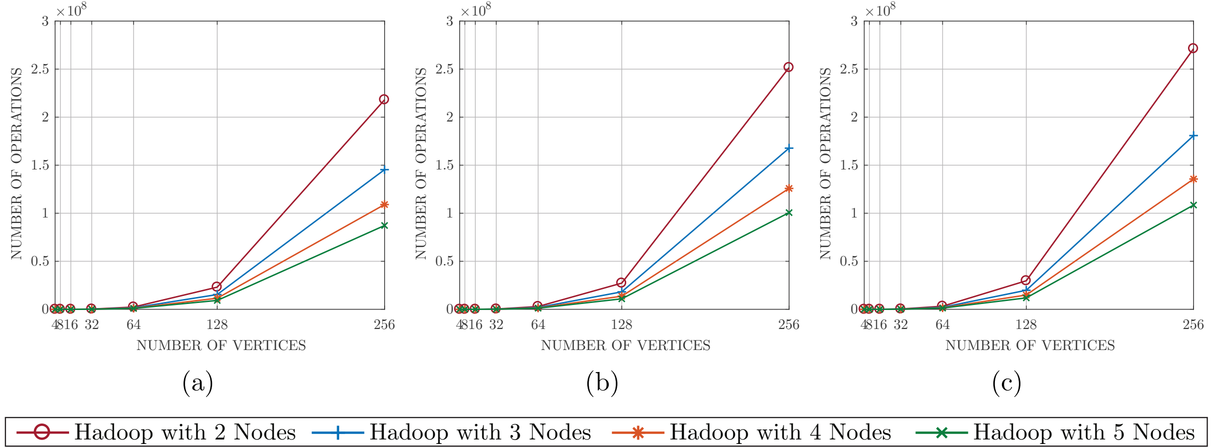


Figure 4: Number of operations when a) $k = 0.25 \times n$, b) $k = 0.5 \times n$, c) $k = 0.75 \times n$

$n = 32, 64, 128, 256$ considering different number of edges used in the first negative cost cycle $k = 0.25 \times n, 0.5 \times n, 0.75 \times n$.

As can be seen, by increasing the denominator, which is defined as the number of Hadoop nodes, the numbers of performed operations are decreased. In fact, there is a fast descending with 5 denominators compared with the other denominators when the number of vertices reaches 256. Also, differences among various values of denominators in the low number of vertices are less than the higher number of vertices. Thus, it is beneficial to large problems.

Actually, it is obvious that decreasing the number of denominators causes an increase in the numbers of performed operations. For instance, the number of operations in the case of $k = 0.25 \times n$ are very more than $k = 0.75 \times n$. It is due to the fact that total execution time is increased when k is decreased which is expressed as equation 9:

$$\begin{array}{ll} (2\log(\frac{1}{4}n)+1)(\frac{2n^3}{nodes}) & k = 0.25 \times n \\ (2\log(\frac{1}{2}n)+1)(\frac{2n^3}{nodes}) & k = 0.5 \times n \\ (2\log(\frac{3}{4}n)+1)(\frac{2n^3}{nodes}) & k = 0.75 \times n \\ (2\log n+1)(\frac{2n^3}{nodes}) & k = 1 \times n \end{array} \tag{9}$$

6 Experimental results

In this section, the performance of the proposed algorithm based on Map-Reduce is evaluated through implementing different graphs with respect to various parameters, including several vertices and varying number of Hadoop nodes. The graphs are connected and generated randomly; they have at least one negative cost cycle.

In addition to preparing the experimental setup, the proposed algorithm based on Map-Reduce is implemented using Java code to compare the performance of the proposed algorithm with the implemented nave sequential algorithm (A Hadoop with only one

node). Moreover, our testing platform is Hadoop-2.8.0 that is installed on Ubuntu 14.04 operating system. In total, there is one NameNode (1GB RAM, 3Core Processors) and one, two, three, and four DataNodes (1GB RAM, 3 Core Processors) respectively.
All the implementations are built on VMware machines on a computer. In order to get final results precisely, we try to limit some restrictions that are made by the running virtual machines on a computer. The following restrictions are eliminated.

- When we run on over one virtual machine on a computer, the power of the computer will be decreased because the power of the processor and memory and other items will be divided among them.
- We get the imposed cost of running several virtual machines on a computer to overcome the restrictions and to achieve more exact results.
- Each virtual machine that runs on the computer will impose some overloads leading to an increase in the final execution time. In fact, it will be increased for any virtual machines which we will run on the same computer. So, we ignore this value from any got execution time.

We let the number of vertices n starting with 32 and keep doubling up to 128. Also, the number of Hadoop nodes is increased up to 5 nodes (1 master and 1 to 4 slaves). The number of edges used in the first negative cost cycle is shown as k, where $k = 0.25 \times n, k = 0.5 \times n, k = 0.75 \times n, k = 1 \times n$.
The important selected metric of the performance evaluation is time complexity. Each reported result is tested on five independent samples and the average execution time is reported approximately as shown in Table 5.
It is observed that in the large graphs, the execution time is grown in sequential implementation, while parallel implementation is lower than the time consumed in the sequential approach. Besides, low improvement is occurred by increasing the number of Hadoop nodes in the low number of vertices due to forming a small graph with a low number of vertices. Furthermore, the emission and collection times or communication costs could increase the running time. In other words, the effect of communication cost is ignored in this case. Therefore, the parallelism is useful in the large type of graphs which communication cost is negligible.
Fig 5 is provided to show the comparison of proposed algorithm based on Map-Reduce and sequential algorithm versus different number of vertices $32, 64, 128$ with respect to different k values including $k = 0.25 \times n, k = 0.5 \times n, k = 0.75 \times n, k = 1 \times n$.
In fact, it is obvious that although the execution time is increased more rapidly in the sequential approach, there is smooth time enhancement in parallel implementations. Besides, in a low number of graph vertices, there is a low time reduction, even by five Hadoop nodes. It concludes that we should select the number of Hadoop nodes in relation to the number of graph vertices for better results. Moreover, in large and complex problems, the Hadoop paradigm is helpful. However, the real restriction makes execution time longer than theory. Subtracted value varying from 0-0.5 to at most 14-20 seconds dependent on k value is ignored to overcome the imposed restrictions.

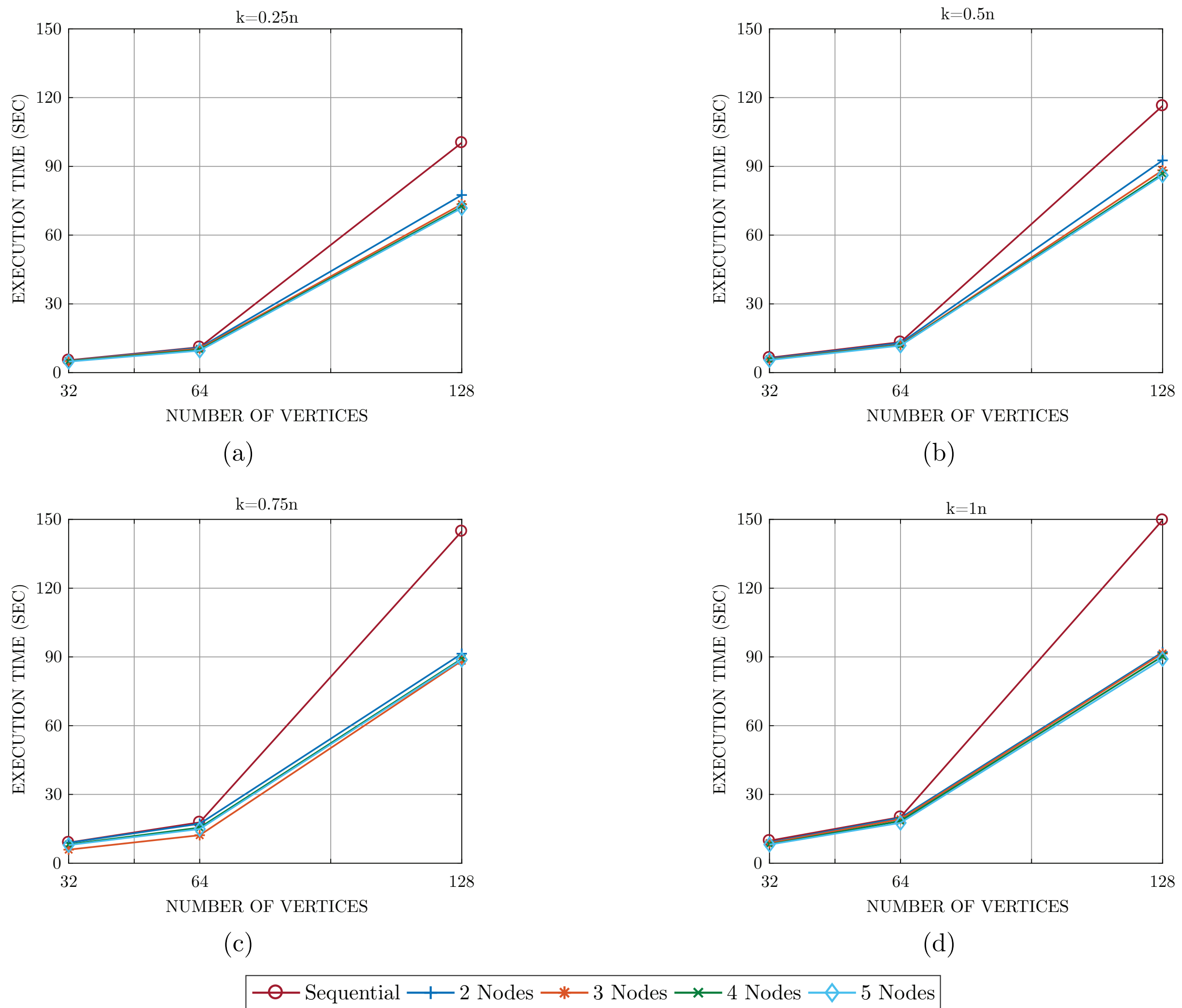


Figure 5: Execution time when a) $k = 0.25 \times n$ b) $k = 0.5 \times n$ c) $k = 0.75 \times n$ d) $k = 1 \times n$

Table 5: Execution Time (seconds)

	Hadoop Nodes	32	64	128
k= 0.25n	sequential	5.4	11.1	100.3
	2 nodes	5.2	10.9	77.5
	3 nodes	5.1	10.5	73.5
	4 nodes	5	10	72.4
	5 nodes	4.8	9.6	71.8
k= 0.5n	sequential	6.5	13.3	116.4
	2 nodes	6.2	12.9	92.6
	3 nodes	5.9	12.2	88.3
	4 nodes	5.7	12	86.8
	5 nodes	5.5	11.8	86.1
k= 0.75n	sequential	9	17.8	144.8
	2 nodes	8.9	17.2	91.4
	3 nodes	8.5	16.2	90.8
	4 nodes	8.2	15.5	89.2
	5 nodes	7.9	14.9	88.8
k= 1n	sequential	9.8	20.1	149.7
	2 nodes	9.1	19.8	92.1
	3 nodes	8.8	19.2	91.5
	4 nodes	8.3	18.4	90.3
	5 nodes	8.1	17.6	89.1

6.1 Map-Reduce vs MPI

A parallel algorithm will have a faster execution time than a sequential algorithm. So, we have to compare our results with a parallel algorithm such as MPI. In paper [23] a parallel implementation has been proposed using MPI which runs in $O(\log k . \log n)$ time on $O(n^3)$ processors as our based paper. The comparison result of our Map-Reduce algorithm vs the reported results in paper [23] has shown in Table 6. Note that, we choose only the case of a graph with 128 vertices with 1, 2, and 4 Hadoop nodes and we compare our result with the same configured graph with 1, 2, and 4 parallel processors reported in their paper. Although they applied 128 processors in their experimentation, our hardware limitation for creating more Hadoop nodes did not allow us to compare the result with more than 4 processors.

Note that, the execution time in these two cases are not comparable, because paper [23] experiments were implemented on the Frost IBM Blue Gene/L supercomputer consists of 8,192 processors, and our platform was a simple Hadoop with one NameNode (1GB RAM, 3Core Processors). But, the result again proves that our theoretical improvement. The execution time of the MPI algorithm depends on $O(\log n)$, whereas our Map-Reduce algorithm depends on $O(\log k)$. Since the Map-Reduce algorithm is theoretically faster than the existing parallel implementation, we think that the execution time in the same

Table 6: Execution time in Map-Reduce vs MPI (seconds)

		Map-Reduce	MPI
k= 0.25n	1 Node vs 1 Processor	100.3	0.62
	2 Node vs 2 Processors	77.5	0.45
	4 Node vs 4 Processors	72.4	0.22
k= 0. 5n	1 Node vs 1 Processor	116.4	0.76
	2 Node vs 2 Processors	92.6	0.59
	4 Node vs 4 Processors	86.8	0.29
k= 0.75n	1 Node vs 1 Processor	144.8	0.92
	2 Node vs 2 Processors	91.4	0.72
	4 Node vs 4 Processors	89.2	0.35
k= 1n	1 Node vs 1 Processor	149.7	0.92
	2 Node vs 2 Processors	92.1	0.72
	4 Node vs 4 Processors	90.3	0.35

platforms will be significantly less than the parallel algorithms.

6.2 Speedup analysis

Speedup measurement is one of the most important metrics to analyse the performance of parallel algorithms like Map-Reduce. It computes how faster a parallel algorithm runs than the sequential one. There are two main methods to measure the speedup, using execution time and Amdahls Law [1]. Speedup measures by execution time using equation 10.

$$Speedup = \frac{T(1)}{T(n)} \tag{10}$$

where n is the number of Hadoop nodes, $T(1)$ is the execution time with one node, and $T(n)$ is the execution time using n Hadoop nodes parallelly.
Amdahls Law defines speedup as equation 11.

$$Speedup = \frac{W_s + W_p}{W_s + \frac{W_p}{n}} \tag{11}$$

where n is the number of Hadoop nodes, W_s is the portion of the process that cannot be parallelized, and W_p is the parallelizable work of the whole work.
Because of our hardware limit, we implement a Hadoop with 5 nodes and at most a graph with 128 vertices. Thus our speedup regarding the execution time is equal to $\frac{T(1)}{T(5)}$ where the graph vertices are 128. The speedup for different values of NCG(k) are shown in Table 7.
As we know there is an upper bound for speedup and the speedup will converge to an upper bound when n increases. Our result does not converge, and it means that the 5

Table 7: Speedup of proposed Map-Reduce algorithm for different values of NCG(k)

NCG value	k = 0.25n	k = 0.5n	k = 0.75n	k = 1n
Speedup	1.396	1.351	1.630	1.680

Hadoop nodes are not sufficient to find the final speedup of the algorithm. Therefore, we try to find the final speedup with Amdahl's Law. The whole process is divided into the following sub process:

1. Create the input file
2. First round to find the upper bound of k among 2, 4, 8, ... including
   - Create a new job
   - Map phase
   - Reduce phase
   - Pass the output to the next Map
3. Binary NCG search algorithm in range $[\frac{k}{2} + 1, k]$
   - Combine two matrices as input file
   - Create a new job
   - Map phase
   - Reduce phase
   - Pass the output to the next Map

Among all jobs described above, only the Map and Reduce processes are parallel. It means that the portion of parallel works is 40%. Based on Amdahl's Law the speedup is equal to $\frac{1}{0.6+\frac{0.4}{n}}$ and it's upper bound is around 1.666. But Consider that most of the time the algorithm runs on the Map and Reduce phase. So the $\frac{W_p}{W_s}$ ratio is more than 40% and the experimental evidence shows that it is more than 50%. If we assume that the ratio is around 50%, then the speedup upper bound is 2. These facts are shown in Fig 6. Briefly, the speedup is between 1.66 and 2, which is confirmed by experimental results.

7 Conclusion

The NCG problem is defined to find the fewest number of edges making a negative cost cycle. As discussed in this paper, existing algorithms check all shortest paths between pairs until finding the first negative cost cycle. In the big and dense graphs, it is more difficult. So, parallelism is an efficient method to find NCG in the graph.
Parallelism is a potential solution for processing complicated and massive data. Map-Reduce is a new paradigm to parallelize big data problems. To do so, Map-Reduce divides

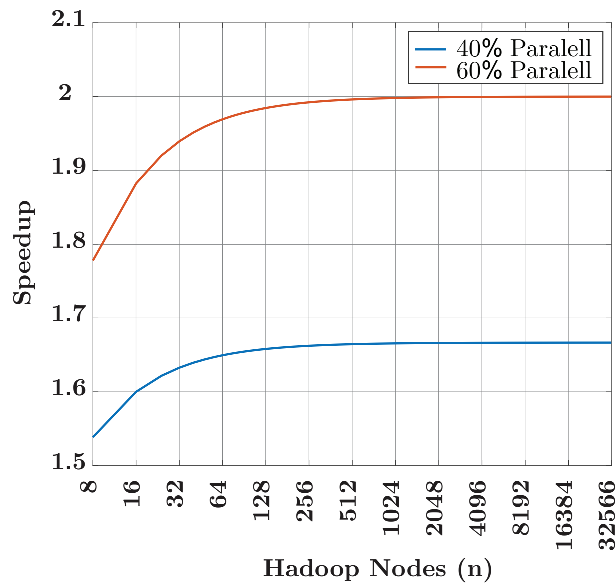


Figure 6: Speedup according to Amdahl's Law

the task into subtasks, which should be allocated to each one of multiple processors in order to perform one of the partitioned tasks using separated data. Therefore, Map-Reduce causes the total execution time to reduce significantly.
In this paper, we focus on the NCG problem in networks using a Map-Reduce based algorithm. In fact, we implemented a Map-Reduce based algorithm for the NCG problem on the Hadoop framework. The evaluation results show that our algorithm gets superior results than the other parallel approach. In our algorithm, if we select the number of Hadoop nodes with respect to the number of graph vertices, we get significant results reaching half of the running time of sequential implementation.
As future work, we can increase the number of participating machines and consider the limitations in order to achieve better speed. Also, we can test the algorithm on real clusters to check our performance evaluations. Besides, through selecting larger graphs, we can increase the number of Map and Reduce tasks in the job to investigate its effects on the performance.

References

[1] Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483485, New York, NY, USA. Association for Computing Machinery.

[2] Anderson, M., Williamson, M., and Subramani, K. (2019). Empirical analysis of algorithms for the shortest negative cost cycle problem. *Discrete Applied Mathematics*, 253:167–184.

[3] Barreto, M., Nesmachnow, S., and Tchernykh, A. (2018). Hybrid algorithms for 3-sat optimisation using mapreduce on clouds. *International Journal of Innovative Computing and Applications*, 9(1):44–64.

[4] Chandrachoodan, N., Bhattacharyya, S. S., and Liu, K. (1999). Negative cycle detection in dynamic graphs. Technical report, University of Maryland Institute for Advanced Computer Studies.

[5] Chen, W., Song, Y., Ba, H., Lin, C., and Chang, E. Y. (2011). Parallel spectral clustering in distributed systems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(3):568–586.

[6] Cotton, S. and Maler, O. (2006). Fast and flexible difference constraint propagation for dpll (t). In *International Conference on Theory and Applications of Satisfiability Testing*, pages 170–183. Springer.

[7] Dean, J. and Ghemawat, S. (2010). Mapreduce: A flexible data processing tool. *Commun. ACM*, 53:72–77.

[8] Dechter, R., Meiri, I., and Pearl, J. (1991). Temporal constraint networks. *Artificial intelligence*, 49(1-3):61–95.

[9] Doerr, B., Johannsen, D., Kotzing, T., and NeumannMadeleine, T. (2013). More effective crossover operators for the all-pairs shortest path problem. *Theoretical Computer Science*, 471(3).

[10] Ghemawat, S. and Dean, J. (2004). Mapreduce: Simplified data processing on large clusters. *OSDI, 2004.*

[11] Han, C.-C. and Lin, K.-J. (1992). Scheduling real-time computations with separation constraints. *Information Processing Letters*, 42(2):61–66.

[12] Kajdanowicz, T., Kazienko, P., and Indyk, W. (2014). Parallel processing of large graphs. *Future Generation Computer Systems*, 32:324–337.

[13] Memishi, B., Ibrahim, S., Perez, M., and Antoniu, G. (2016). *Fault Tolerance in MapReduce: A Survey*, chapter 11, pages 205–240. Springer.

[14] Morris, P., Muscettola, N., and Vidal, T. (2001). Dynamic control of plans with temporal uncertainty. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI01, page 494499, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[15] Ni, E. C., Ciocan, D. F., Henderson, S. G., and Hunter, S. R. (2015). Comparing message passing interface and mapreduce for large-scale parallel ranking and selection. In *2015 Winter Simulation Conference (WSC)*, pages 3858–3867.

[16] Orlin, J. B., Subramani, K., and Wojciechowki, P. (2018). Randomized algorithms for finding the shortest negative cost cycle in networks. *Discrete Applied Mathematics*, 236:387–394.

[17] Pelucchi, M., Psaila, G., and Toccu, M. (2018). Hadoop vs. spark: Impact on performance of the hammer query engine for open data corpora. *Algorithms*, 11(12).

[18] Subramani, K. (2007). A zero-space algorithm for negative cost cycle detection in networks. *Journal of Discrete Algorithms*, 5(3):408–421.

[19] Subramani, K. (2009). Optimal length resolution refutations of difference constraint systems. *Journal of Automated Reasoning*, 43(2):121–137.

[20] Subramani, K., Williamson, M., and Gu, X. (2013). Improved algorithms for optimal length resolution refutation in difference constraint systems. *Formal Aspects of Computing*, 25(2):319–341.

[21] Tao, Y., Lin, W., and Xiao, X. (2013). Minimal mapreduce algorithms. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 529–540.

[22] Williamson, M. and Subramani, K. (2015a). On the negative cost girth problem in planar networks. *Journal of Discrete Algorithms*, 35:40–50.

[23] Williamson, M. and Subramani, K. (2015b). A parallel implementation for the negative cost girth problem. *International Journal of Parallel Programming*, 43(2):240–259.

[24] Yamada, T. and Kinoshita, H. (2002). Finding all the negative cycles in a directed graph. *Discrete Applied Mathematics*, 118(3):279–291.