Hindawi
Scientific Programming
Volume 2021, Article ID 6639008, 14 pages
https://doi.org/10.1155/2021/6639008

Research Article

DMGA: A Distributed Shortest Path Algorithm for Multistage Graph

Huanqing Cui,[1] **Ruixue Liu**,[1] **Shaohua Xu**,[1] **and Chuanai Zhou**[2]

[1]*College of Computer Science and Engineering, Shandong University of Science and Technology, Qingdao, Shandong, China*
[2]*College of Business, Qingdao Binhai University, Qingdao, Shandong, China*

Correspondence should be addressed to Huanqing Cui; cuihq@sdust.edu.cn

Received 15 October 2020; Accepted 22 May 2021; Published 1 June 2021

Academic Editor: Cristian Mateos

The multistage graph problem is a special kind of single-source single-sink shortest path problem. It is difficult even impossible to solve the large-scale multistage graphs using a single machine with sequential algorithms. There are many distributed graph computing systems that can solve this problem, but they are often designed for general large-scale graphs, which do not consider the special characteristics of multistage graphs. This paper proposes DMGA (Distributed Multistage Graph Algorithm) to solve the shortest path problem according to the structural characteristics of multistage graphs. The algorithm first allocates the graph to a set of computing nodes to store the vertices of the same stage to the same computing node. Next, DMGA calculates the shortest paths between any pair of starting and ending vertices within a partition by the classical dynamic programming algorithm. Finally, the global shortest path is calculated by subresults exchanging between computing nodes in an iterative method. Our experiments show that the proposed algorithm can effectively reduce the time to solve the shortest path of multistage graphs.

1. Introduction

With the continuous development of big data and information technology, graph has been widely applied in many applications, and various graph structures and algorithms have been proposed. Among them, the multistage graph is a special kind of weighted directed graphs, which are widely used in engineering technology, concurrency control, transportation, task schedule in high-performance computing, and other fields. Many coordination or dynamic scheduling problems can be transformed into multistage graph problems [1, 2].

Recently, the scale of graph data has grown tremendously, so it is difficult even impossible to store and process such large-scale graphs by a single computer or sequential processing method [3]. At this point, the distributed computing scheme became a must, and lots of dedicated graph-processing systems have been appearing [4–6], such as Pregel [7], PowerGraph [8], GraphX [9, 10], GraphLab [11], and PowerLyra [12]. These graph processing systems extend the computation by dividing the graph into multiple partitions and processing on multiple computing nodes in parallel. High-quality partition can reduce the communication cost and achieve the load balance [13–15], thus the processing time can be minimized subsequently. The current distributed graph processing systems and algorithms are usually designed for general graphs, and they do not consider the special structural properties of multistage graphs, so there are some disadvantages in applying them to multistage graphs, such as high communication cost and long solution time. The purpose of this paper is to present a distributed algorithm DMGA (Distributed Multistage Graph Algorithm) for the shortest path problem of multistage graphs to make full use of their characteristics. The main contributions are as follows:

(1) It presents a partitioning method for multistage graphs on distributed computing systems, which can make best use of characteristics of multistage graphs to achieve the best load balance and reduce the communication cost

(2) It designs a distributed algorithm of the shortest path problem of multistage graphs based on dynamic programming idea

(3) It performs extensive experiments to verify the performance of the proposed algorithm, compared to the classical parallel Dijkstra algorithm and the SSSP (single-source shortest path) algorithm on Pregel

Table 1 gives an overview about the notations used in this paper. The organizations of the rest of the paper are as follows. Section 2 introduces the related works, and Section 3 presents the statements of the shortest path problem of multistage graphs. Section 4 describes the proposed DMGA algorithm, and Section 5 introduces the experiments and analysis. Section 6 concludes the paper.

2. Related Works

2.1. The Shortest Path Algorithms. Finding the shortest path is a classical problem of graph theory, and the well-known sequential algorithms are Dijkstra, Floyd, and Bellman–Ford algorithms [16], which perform well in centralized computing. However, the large-scale graph needs distributed computing algorithms to obtain the shortest paths quickly.

The single-source shortest path (SSSP) is one of the most important shortest path problems. Peng et al. [1] defined a new graph model named by single-source-weighted multilevel graph and presented a parallel SSSP algorithm by constructing the vector-matrix multiple model, dividing into parallel tasks, and setting data communication's method. A·Davidson [17] developed three parallel SSSP algorithms for GPUs (Graphics Processing Unit): Workfront Sweep, Near-Farand, and Bucketing. These algorithms utilize different approaches to balance the trade-off between saving work and organizational overhead. S·Maleki [18] introduced a partially asynchronous parallel DSMR (Dijkstra Strip Mined Relaxation) algorithm for SSSP on shared and distributed memory systems. Busato and Bombieri [19] proposed a parallel Bellman–Ford algorithm based on frontier and active vertices that exploit the architectural characteristics of GPU architectures. Huang [20] gave a distributed Las Vegas algorithm on the classic scaling technique for the all-pairs' shortest paths on distributed networks. For the dynamic and stochastic graph models of the transportation network, Liu et al. [21] proposed an improved adaptive genetic algorithm by adjusting the encoding parameters to get the dynamic random shortest path. Ghaffari and Li [22] provided a distributed SSSP algorithm with less complexity, and it constitutes the first sublinear time algorithm for directed graphs. For the SSPP-MPN (Shortest Simple Path Problem with Must-Past Nodes), Su et al. [23] proposed a multistage metaheuristic algorithm based on k-opt move, candidate path search, conflicting nodes promotion, and connectivity relaxation.

The above algorithms do not consider the structural characteristics of multistage graphs, so they will produce a large amount of communication overhead, resulting in tedious execution time.

2.2. Graph Partitioning Algorithms. The basis of the distributed graph processing system is to partition the entire graph into a set of computing nodes. The graph partition algorithms are classified into vertex-cut and edge-cut. Edge-cut partitioning assigns each vertex to a unique partition, and the edge spanning partitions are called cut edge. As shown in Figure 1(a), edges $<b, d>$ are cut, and their two endpoints b and d are assigned to different partitions. Vertex-cut partitioning assigns edges uniquely to a certain partition, which results in vertex-cuts across multiple partitions [24]. As shown in Figure 1(b), vertex d is partitioned, both partitions $P1$ and $P2$ have copies of vertex d, and their references in each partition are also called mirrors [25].

The distributed graph processing systems often use the vertex-centric programming model [26, 27], where the computing node recursively operates its active vertices according to the user-defined graph function. Each vertex reads the statuses of its adjacent vertices or edges and updates its own status accordingly. In the iterative calculation of a graph, the partitions exchange intermediate results along edges. To some extent, the number of cut edges or mirror vertices can reflect the network communication overhead, which in turn affects the calculation efficiency. On the contrary, the load among computing nodes should be balanced to ensure that the computing nodes can achieve the results synchronously. Hence, both edge-cut and vertex-cut approaches aim to minimize cross-partition dependencies and achieve load balance [25].

The existing graph partitioning heuristic solutions are basically divided into offline and online partitioning strategies. The offline partitioning strategy refers to dividing the graph into several subgraphs before being loaded by the distributed system. F·Rahimian et al. [28] introduced the JA-BE-JA algorithm that uses local search and simulated annealing techniques for graph partitioning. The algorithm only needs to use part of the information to process the graph. Akhremtsev et al. [29] presented a multilevel shared-memory parallel graph partitioning algorithm that uses parallel label propagation for both coarsening and refinement, and it can balance the speed and quality of parallel graph partitioning.

The online partitioning strategy refers to partitioning the graph during the data loading process, where the input data is usually a vertex stream or an edge stream. Tsourakakis et al. [30] proposed the FENNEL algorithm based on locality-centric measures and balancing goals. Its core idea is to interpolate between maximizing the co-location of neighbouring vertices and minimizing that of non-neighbours. Petroni et al. [15] proposed the high-degree replicated first (HDRF) algorithm according to the characteristics of power-law graphs, which divide the vertices with high degrees in first. Zhang et al. [31] proposed the

Table 1: Notation overview.

Symbol	Definition
$G = (V, E, W)$	Graph with vertices set V, edges set E, and edge weights' set W
m	Number of stages in a multistage graph
V_i	Set of vertices of stage i of a multistage graph
n_i	Number of vertices of stage i of a multistage graph
$v_{i,j}$	The jth vertex in the ith stage of a multistage graph
$w_{i,j,i+1,k}$	Weight of edge $<v_{i,j}, v_{i+1,k}>$
E_i	Set of edges from V_i to V_{i+1}
$c_{k,i,l,j}$	Cost of the shortest path from vertex $v_{k,i}$ to $v_{l,j}$
CN_i	The ith partition or computing node
p	Number of partitions
Cap	Capacity of each computing node
ML	Maximum load of each partition
Sum	Number of edges of a partition
s_k	The first stage of CN_k
e_k	The last stage of CN_k
$f_{s_k,i,j,l}$	Index of the previous vertex of $v_{j,l}$ in the shortest path from $v_{s_k,i}$ to $v_{j,l}$
$sp_{s_k,j,e_k,i}$	List of vertices of the shortest path from $v_{s_k,j}$ to $v_{e_k,i}$
R	Set of partition IDs
SPL_k	Set of the shortest paths storing on CN_k

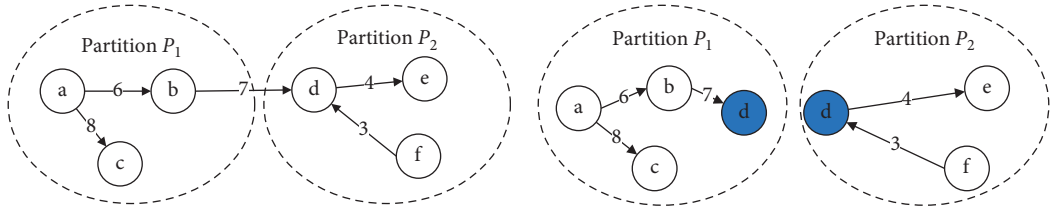

Figure 1: Two graph partitioning methods. (a) Edge-cut partition. (b) Vertex-cut partition.

AKIN algorithm based on the vertex similarity index, which exploits the similarity measure of the vertex degree to collect structure-related vertices in the same partition to further reduce the edge-cut rate. Wang et al. [32] analysed the locality of the graph and proposed the target-vertex sensitive Hash algorithm. The algorithm predivides the target vertices of the edge logically and then partitions the graph in parallel according to the target vertices. Ji et al. [33] proposed a two-stage local partitioning algorithm which introduces the concept of local partitions, emphasizing the impact of changes in the graph structure on the quality of partitions. Slota et al. [34] introduced XtraPuLP based on the scalable label propagation community detection technique. It can solve the multiple constraint and multiple objective graph partition problem on tera-scale graphs. Zhou et al. [35] proposed Geo-Cut which uses a cost-aware streaming heuristic and two partition refinement heuristics to reduce the cost and data transfer time of geo-distributed data centres.

The above graph partitioning algorithm are all designed for general graphs, which do not consider the special characteristics of multistage graphs, so it is necessary to design the graph partitioning algorithm for multistage graphs to accelerate the distributed processing.

3. Problem Statements

A multistage graph $G = (V, E, W)$ is a directed single-source and single-sink weighted connected graph, where V and E are, respectively, the set of vertices and edges and W is the weights of edges. The vertices are divided into disjoint stages, and each edge can only point from the vertex of the previous stage to the vertex of the succeeding stage. Formally, a multistage graph $G = (V, E, W)$ should satisfy

(1) $V = \cup_{i=1}^{m} V_i, \forall i \neq j$ and $V_i \cap V_j = \varnothing$, where m is the number of stages.

(2) $V_i = \{v_{i,j} | j = 1, 2, \ldots, n_i\}$, where n_i is the number of vertices of the ith stage.

(3) $E = \{v_{i,j}, v_{i+1,k} | i = 1, 2, \ldots, m-1; j = 1, 2, \ldots, n_i; k = 1, 2, \ldots, n_{i+1}\}$.

(4) $W = \{w_{i,j,i+1,k} | i = 1, 2, \ldots, m-1; j = 1, 2, \ldots, n_i; k = 1, 2, \ldots, n_{i+1}\}$, where $w_{i,j,i+1,k}$ is the weight of edge $v_{i,j}, v_{i+1,k}$. If $v_{i,j}, v_{i+1,k} \notin E$, $w_{i,j,i+1,k} = \infty$.

(5) $V_1 = \{v_{1,1}\}$, $V_m = \{v_{m,1}\}$, and $v_{1,1}$ and $v_{m,1}$ are, respectively, the source vertex and sink vertex.

Figure 2 is an example of the multistage graph, where the blue numbers above the graph are the numbers of edges. This

paper supposes that the multistage graphs are dense which means

$$|E_i| \approx n_i \times n_{i+1}, \tag{1}$$

where $E_i = \{v_{i,j}, v_{i+1,k} | j = 1, 2, \ldots, n_i; k = 1, 2, \ldots, n_{i+1}\}$ is the set of edges between Vi and Vi+1 and $|E_i|$ is the number of edges of Ei.

The shortest path problem of a multistage graph is to find the minimum cost path from the source vertex to the sink vertex. Let $c_{k,i,l,j}$ be the cost of the shortest path from vertex $v_{k,i}$ to $v_{l,j}$. Obviously, $c_{1,1,m,1}$ is the cost of the shortest path from the source vertex to the sink vertex, and

$$c_{1,1,k,i} = \begin{cases} 0, & \text{if } k = 1 \text{ and } i = 1, \\ \min\limits_{v_{k-1,j}, v_{k,i} \in E} \{c_{1,1,k-1,j} + w_{k-1,j,k,i}\}, & \text{if } k > 1 \text{ and } i = 1, 2, \ldots, n_k. \end{cases} \tag{2}$$

Given a large-scale multistage graph $G = (V, E, W)$, we need to partition it to a cluster of computing nodes. Each computing node stores a part of G, and each part is called a partition. Let p be the number of partitions, so G is divided into partitions $\{\mathrm{CN}_1, \mathrm{CN}_2, \ldots, \mathrm{CN}_p\}$ and CN_i is located on the ith computing node.

4. DMGA: The Proposed Algorithm

DMGA is run on the homogeneous cluster, which means all computing nodes have the same performance in terms of CPU, memory, and bandwidth. This algorithm partitions the entire graph to the given cluster first, and then, each computing node computes the shortest path of the partition on it. Finally, the computing nodes communicate with each other to obtain the shortest path of the whole graph.

Algorithm 1 gives the framework of DMGA. The details of each step of Algorithm 1 are given in the followingsections.

4.1. Multistage Graph Partition. In order to determine the graph partition strategy, we should analyse their impacts on the communication overhead after partition. According to the feature of multistage graphs, it is a better scheme to divide the vertices of the same stage into the same partition because it is easy to implement load balance and parallel shortest path solution. Suppose V_i and V_{i+1} are divided into two different partitions. If we use vertex-cut strategy, the number of mirror vertices is either n_i or n_{i+1}. If we use edge-cut strategy, the number of cut edges is $|E_i|$. According to (1), $n_i < |E_i|$ and $n_{i+1} < |E_i|$, which indicates that the vertex-cut strategy has less communication overhead than edge-cut strategy, so we adopt vertex-cut strategy to partition the graph. Figure 3 is an example. The edge-cut strategy produces 9 cut edges (Figure 3(a)), while the vertex-cut strategy only produces 3 mirror vertices (Figure 3(b)).

Since the multistage graphs studied in this paper are dense, we use the number of edges to represent the load of a partition. Let Cap be the capacity of each computing node, which is also the maximum number of edges that can be stored by a partition, then the number of partitions of a given G can be estimated as

$$p = \frac{|E|}{\mathrm{Cap}}. \tag{3}$$

The above equation gives the lower limit of the number of computing nodes. It may lead to the load of the last computing node far lower than those of the other computing nodes. For example, if $|E| = 10100$ and Cap = 1000, then $p = 11$. If the first 10 computing nodes are fully loaded, then the last computing node only has 100 edges, so the load is imbalanced. Hence, the maximum load of each partition is redefined as

$$\mathrm{ML} = \begin{cases} \mathrm{Cap}, & \text{if } p = \dfrac{|E|}{\mathrm{Cap}}, \\ \gamma \dfrac{|E|}{p}, & \text{otherwise}, \end{cases} \tag{4}$$

where γ is a predefined parameter to keep the load balance for different multistage graphs.

The idea of multistage graph partition is to assign the vertices of the same stage to the same partition. Figure 4 presents the flow diagram, and Algorithm 2 presents pseudocode. In this algorithm, Sum records the number of edges stored in the current partition. Lines 1 and 2 initialize the variables. Lines 3–16 divide G to a cluster of computing nodes. If Sum ≤ ML (line 5), lines 6–8 assign the edges of E_i to computing node CN_k, and lines 9 and 10, respectively, update e_k and i. If Sum > ML (line 11), which means the current partition CN_k will be overloaded if we assign the edges of E_i to it, lines 12–14 update variables to prepare for succeeding partition.

4.2. Local Shortest Path Calculation. After partitioning the graph, each computing node calculates the shortest path of the subgraph stored on it. The shortest path of each partition is referred to as the local shortest path, and the shortest path of the whole graph is referred to as the global shortest path.

Theorem 1. *If* $\{v_{1,1}, v_{2,i_2}, \ldots, v_{k,i_k}, v_{k+1,i_{k+1}}, \ldots, v_{l-1,i_{l-1}}, v_{l,i_l}, \ldots, v_{m,1}\}$ *is one of the shortest paths from* $v_{1,1}$ *to* $v_{m,1}$*, then* $\{v_{k,i_k}, v_{k+1,i_{k+1}}, \ldots, v_{l-1,i_{l-1}}, v_{l,i_l}\}$ *is one of the shortest paths from* v_{k,i_k} *to* v_{l,i_l}.

Proof. We prove it by using reduction to absurdity. Suppose $\{v_{k,i_k}, v_{k+1,i_{k+1}}, \ldots, v_{l-1,i_{l-1}}, v_{l,i_l}\}$ is not one of the shortest paths from v_{k,i_k} to v_{l,i_l}. There must exist a shortest path from v_{k,i_k} to v_{l,i_l}. Let $\{v_{k,i_k}, v_{k+1,i_{k+1}}', \ldots, v_{l-1,i_{l-1}}', v_{l,i_l}\}$ be one of the shortest paths from v_{k,i_k} to v_{l,i_l}. If we use $\{v_{k+1,i_{k+1}}', \ldots, v_{l-1,i_{l-1}}'\}$ to replace $\{v_{k+1,i_{k+1}}, \ldots, v_{l-1,i_{l-1}}\}$ in the path $\{v_{1,1}, v_{2,i_2}, \ldots, v_{k,i_k}, v_{k+1,i_{k+1}}, \ldots, v_{l-1,i_{l-1}}, v_{l,i_l}, \ldots, v_{m,1}\}$, then the cost of $\{v_{1,1}, v_{2,i_2}, \ldots, v_{k,i_k}, v^{k+1,i_{k+1}}, \ldots, v^{l-1,i_{l-1}}, v_{l,i_l}, \ldots, v_{m,1}\}$ is less than $\{v_{1,1}, v_{2,i_2}, \ldots, v_{k,i_k}, v_{k+1,i_{k+1}}, \ldots, v_{l-1,i_{l-1}}, v_{l,i_l}, \ldots, v_{m,1}\}$, so

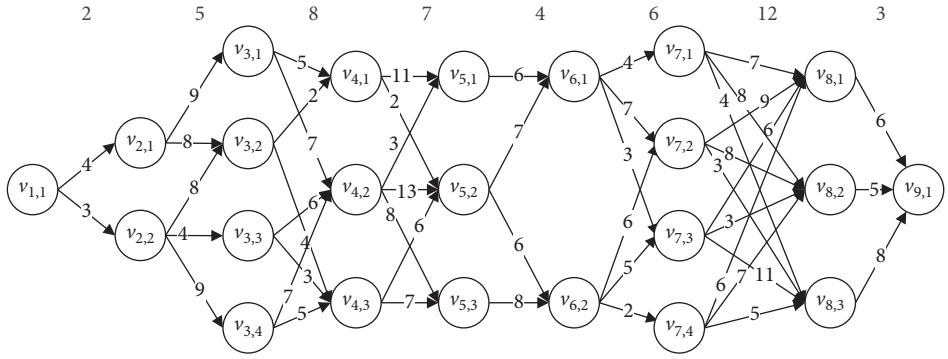

Figure 2: A multistage graph.

Require: A large-scale multistage graph $G = (V, E, W)$
Ensure: The shortest path from v1,1 to vm,1
(1) Partition G
(2) Get the local shortest path of each partition
(3) Get the global shortest path of the whole graph

Algorithm 1: DMGA.

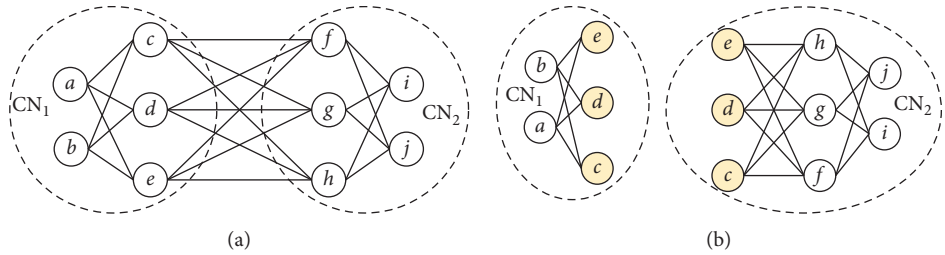

Figure 3: Two graph partitioning strategies. (a) Edge-cut strategy. (b) Vertex-cut strategy.

$\{v_{1,1}, v_{2,i_2}, \ldots, v_{k,i_k}, v_{k+1,i_{k+1}}, \ldots, v_{l-1,i_{l-1}}, v_{l,i_l}, \ldots, v_{m,1}\}$ is not one of the shortest paths from $v_{1,1}$ to $v_{m,1}$. This is a contradiction.

The above theorem shows that any part of the shortest path is also the shortest path, so the global shortest path is composed of the local shortest paths of all partitions, and

$$c_{1,1,m,1} = \min \sum_{k=1}^{m-1} c_{k,i,k+1,j}, \tag{5}$$

where $i = 1, 2, \ldots, n_k$ and $j = 1, 2, \ldots, n_{k+1}$. Subsequently, we have

$$c_{1,1,m,1} = \min \sum_{k=1}^{p} c_{s_k,i,e_k,j}, \tag{6}$$

where $i = 1, 2, \ldots, n_{s_k}$ and $j = 1, 2, \ldots, n_{e_k}$.

Based on the above equation, it is necessary to calculate the shortest paths between any pair of vertices of the first and last stages for each partition. Specifically, each computing node uses the idea of dynamic programming shown as (2) to solve the local shortest path.

Figure 5 presents the flow diagram, and Algorithm 3 presents pseudocode. Note that this algorithm is run by each computing node in parallel. On partition CN_k, it produces one of the shortest path from $v_{s_k,i}$ to $v_{e_k,j}$ for each vertex $v_{s_k,i} \in V_{s_k}$ and each vertex $v_{e_k,j} \in V_{e_k}$. In this algorithm, $f_{s_k,i,j,l}$ records the previous vertex in the shortest path from $v_{s_k,i}$ to $v_{j,l}$, i.e., $\{v_{s_k,i}, \ldots, v_{j-1,f_{s_k,i,j,l}}, v_{j,l}\}$ is one of the shortest path from $v_{s_k,i}$ to $v_{j,l}$. It calculates the shortest paths in a forward way. Lines 1 to 4 initialize c and f for the vertices in the first stage. Lines 5 to 17 calculate the shortest paths by a triple loop. The outermost loop is for all stages V_i $(i = s_{k+1}, s_{k+2}, \ldots, e_k)$ (line 5). The middle loop is for all vertices $v_{i,j}$ $(j = 1, 2, \ldots, n_i)$ of V_i (line 6), and the innermost loop is for all vertices $v_{s_k,l}$ $(l = 1, 2, \ldots, n_{s_k})$ of V_{s_k} (line 7). Given $v_{i,j}$ and $v_{s_k,l}$, lines 8 to 14 calculate the shortest path from $v_{s_k,l}$ to $v_{i,j}$, where lines 11 and 12, respectively, update $c_{s_k,l,i,j}$ and $f_{s_k,l,i,j}$. Lines 18 to 29 generate the shortest paths using f in a backtracking way. Given $v_{s_k,j}$ and $v_{e_k,i}$, it obtains the vertices' sequence $sp_{s_k,j,e_k,i}$ starting at vertex $v_{e_k,i}$. □

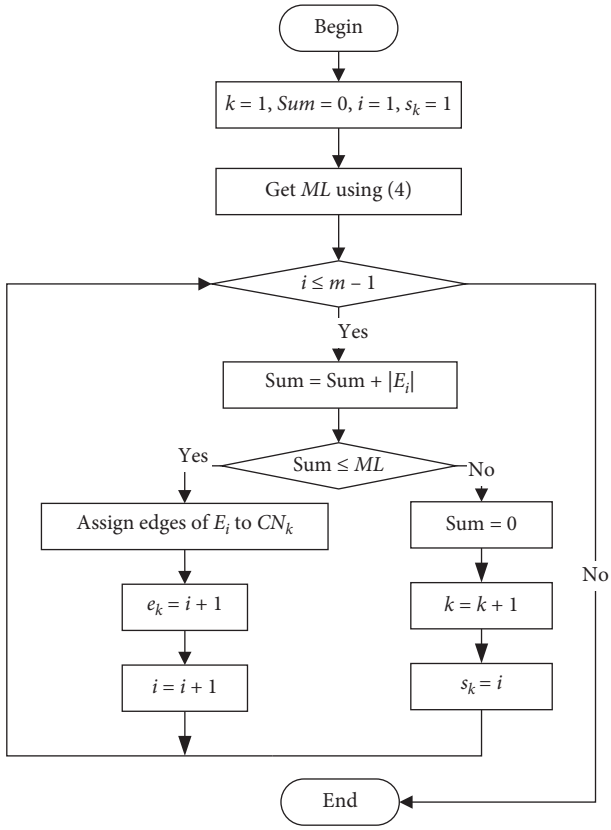

Figure 4: Flow diagram of graph partition.

Require: Multistage graph G
Ensure: Partition result of G
(1) Sum $\longleftarrow 0, k \longleftarrow 1, s_k \longleftarrow 1, i \longleftarrow 1$
(2) Calculate ML using (4)
(3) while $i \leq m-1$ do
(4) Sum $\longleftarrow$ Sum + $|E_i|$
(5) if Sum $\leq ML$ then
(6) for all $v_{i,j}, v_{i+1,l} \in E_i$ do
(7) Assign it to CN_k
(8) end for
(9) $e_k \longleftarrow i+1$
(10) $i \longleftarrow i+1$
(11) else
(12) Sum $\longleftarrow 0$
(13) $k \longleftarrow k+1$
(14) $s_k \longleftarrow i$
(15) end if
(16) end while

Algorithm 2: Multistage graph partitioning algorithm.

4.3. Global Shortest Path Calculation. After finding the shortest path of each partition, the global shortest path is calculated by message exchanging among computing nodes. Figure 6 depicts a sketch of the merging procedure of local shortest paths. This is an iterative procedure. In each iteration, a pair of computing nodes communicates where the "left" computing node sends its local shortest path to the "right" computing node, and the "right" computing node merges these two local shortest paths. Finally, CN_p gets the global shortest path.

Figure 7 presents the flow diagram, and Algorithm 4 presents the pseudocode. The set R records the indices of computing nodes participating in subresults' combination in each iteration. Initially, R contains all computing nodes (line 1). Lines 2 to 5 initialize two sets for each computing node. Lines 6 to 30 calculate the global shortest path by message exchanging among computing nodes with the basic idea as Figure 6. Firstly, the "left" computing nodes $CN_{R_{2i-1}}$ send $SPL_{R_{2i-1}}$ to the "right" neighbour computing node $CN_{R_{2i}}$ (lines 7 to 9), and this can be run in parallel for each pair of computing nodes. Secondly, the "right" computing nodes merge the two subresults to get a longer subresult (lines 10 to 28). Given a local shortest path of $SPL_{R_{2i-1}}$ (line 11) and $SPL_{R_{2i}}$ (line 12), if they can be merged, that is to say, the last vertex of the first path is the same as the starting vertex of the second path (line 13), it tries to merge them to be a longer path from $v_{a1,b1}$ to $v_{c2,d2}$. If the shortest path does not exist, the algorithm generates one and appends it to $SPL'_{R_{2i}}$ (lines 14 to 17). If the shortest path exists but it is longer than the current one, the algorithm updates the shortest path (lines 18 to 21). After the two innermost loops (lines 11 to 25), the algorithm replaces $SPL_{R_{2i}}$ with $SPL'_{R_{2i}}$ (line 26) and sets $SPL'_{R_{2i}}$ to be empty (line27) to prepare for the next iteration. Line 29 removes the "left" computing nodes from R. Finally CN_p returns $c_{1,1,m,1}$ and $sp_{1,1,m,1}$ as the global shortest path (line 31).

4.4. An Example. Let us take Figure 2 as an example to demonstrate the process of the above algorithm. Given Cap = 20, we have $p = 3$ according to (3). Set ML = 20 according to (4). Based on Algorithm 2, the graph will be partitioned to 3 partitions, as shown in Figure 8.

Next, each computing node calculates the local shortest path of its partition. Let us take CN_2 as an example to show the process.

(1) Initially, c and f of vertices of stage 4 are set to 0.
(2) The vertices of V_5 calculate c and f. For example, $c_{4,1,5,2} = c_{4,1,4,1} + w_{4,1,5,2} = 2$ and $f_{4,1,5,2} = 1$.
(3) The vertices of V_6 calculate c and f. For example, $c_{4,1,6,2} = \min\{c_{4,1,5,1} + w_{5,1,6,2}, \quad c_{4,1,5,2} + w_{5,2,6,2}, c_{4,1,5,3} + w_{5,3,6,2}\} = \min\{\infty, 8, \infty\} = 8$ and $f_{4,1,6,2} = 2$.
(4) The vertices of V_7 calculate c and f. For example, $c_{4,1,7,2} = \min\{c_{4,1,6,1} + w_{6,1,7,2}, c_{4,1,6,2} + w_{6,2,7,2}\} = \min\{16, 14\} = 14$ and $f_{4,1,7,2} = 2$.
(5) Finally, the vertices of V_7 backtrack to get the shortest paths. For example, to get the shortest path from $v_{4,1}$ to $v_{7,2}$, it backtracks to $v_{6,2}$ in first because $f_{4,1,7,2} = 2$. Secondly, it backtracks to $v_{5,2}$ and then $v_{4,1}$ finally. Therefore, the shortest path from $v_{4,1}$ to $v_{7,2}$ is $\{v_{4,1}, v_{5,2}, v_{6,2}, v_{7,2}\}$, and the cost is 14.

Similar to the above process, each CN calculates the local shortest paths, and these results are as follows:

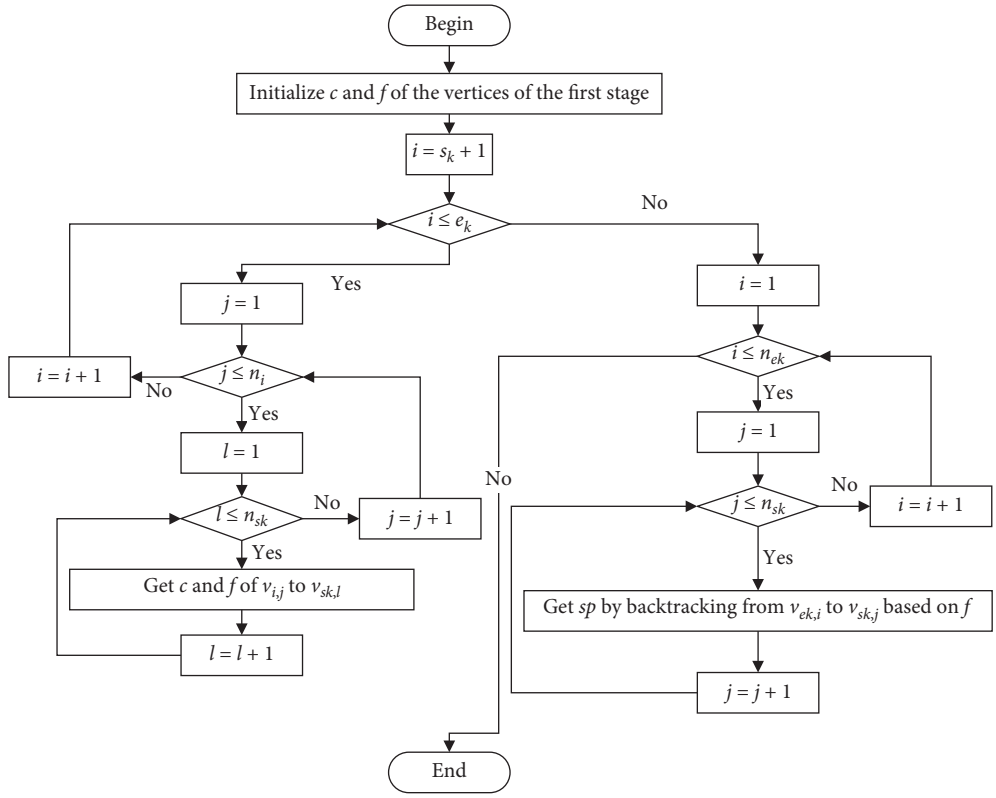

Figure 5: Flow diagram of local shortest path calculation.

Require: The graph partition CN_k
Ensure: $\{c_{s_k,i,e_k,j}, sp_{s_k,i,e_k,j} | i = 1, 2, \ldots, n_{s_k}; j = 1, 2, \ldots, n_{e_k}\}$ ▷list of shortest paths
(1) for $i \longleftarrow 1$ to n_{s_k} do
(2) $c_{s_k,i,s_k,i} \longleftarrow 0$
(3) $f_{s_k,i,s_k,i} \longleftarrow 0$
(4) end for
(5) for $i \longleftarrow s_k + 1$ to e_k do ▷for each stage
(6) for $j \longleftarrow 1$ to n_i do ▷for each vertex of stage V_i
(7) for $l \longleftarrow 1$ to n_{s_k} do ▷for each vertex of the first stage V_{s_k}
(8) $c_{s_k,l,i,j} \longleftarrow \infty$
(9) for all $v_{i-1,h}, v_{i,j} \in E$ do
(10) if $c_{s_k,l,i,j} > c_{s_k,l,i-1,h} + w_{i-1,h,i,j}$ then ▷a shorter path than before
(11) $c_{s_k,l,i,j} \longleftarrow c_{s_k,l,i-1,h} + w_{i-1,h,i,j}$
(12) $f_{s_k,l,i,j} \longleftarrow h$
(13) end if
(14) end for
(15) end for
(16) end for
(17) end for
(18) for $i \longleftarrow 1$ to n_{e_k} do ▷backtrack the shortest paths
(19) for $j \longleftarrow 1$ to n_{s_k} do
(20) $l \longleftarrow i$
(21) $h \longleftarrow e_k$
(22) $sp_{s_k,j,e_k,i} \longleftarrow \{v_{h,l}\}$
(23) while $h \neq s_k$ do
(24) $l \longleftarrow f_{s_k,j,h,l}$
(25) $h \longleftarrow h - 1$
(26) $sp_{s_k,i,e_k,j} \longleftarrow \{v_{h,l}\} \cup sp_{s_k,j,e_k,i}$
(27) end while
(28) end for
(29) end for

Algorithm 3: Local shortest path calculation.

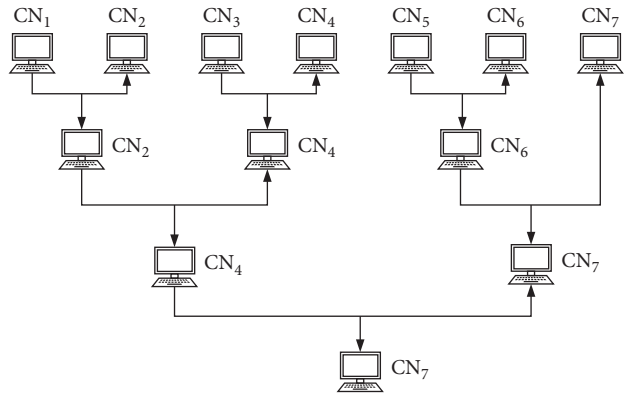

Figure 6: Local shortest paths merging.

(1) $SPL_1 = \{13, \{v_{1,1}, v_{2,2}, v_{3,2}, v_{4,1}\}, 13, \{v_{1,1}, v_{2,2}, v_{3,3}, v_{4,2}\}, 13, \{v_{1,1}, v_{2,2}, v_{3,3}, v_{4,3}\}\}$

(2) $SPL_2 = \{13, \{v_{4,1}, v_{5,2}, v_{6,1}, v_{7,1}\}, 14, \{v_{4,1}, v_{5,2}, v_{6,2}, v_{7,2}\}, 12, \{v_{4,1}, v_{5,2}, v_{6,1}, v_{7,3}\}, 10, \{v_{4,1}, v_{5,2}, v_{6,2}, v_{7,4}\}, 13, \{v_{4,2}, v_{5,1}, v_{6,1}, v_{7,1}\}, 16, \{v_{4,2}, v_{5,1}, v_{6,1}, v_{7,2}\}, 12, \{v_{4,2}, v_{5,1}, v_{6,1}, v_{7,3}\}, 18, \{v_{4,2}, v_{5,3}, v_{6,2}, v_{7,4}\}, 17, \{v_{4,3}, v_{5,2}, v_{6,1}, v_{7,1}\}, 18, \{v_{4,3}, v_{5,2}, v_{6,2}, v_{7,2}\}, 16, \{v_{4,3}, v_{5,2}, v_{6,1}, v_{7,3}\}, 14, \{v_{4,3}, v_{5,2}, v_{6,2}, v_{7,4}\}\}$

(3) $SPL_3 = \{12, \{v_{7,1}, v_{8,3}, v_{9,1}\}, 11, \{v_{7,2}, v_{8,3}, v_{9,1}\}, 8, \{v_{7,3}, v_{8,2}, v_{9,1}\}, 12, \{v_{7,4}, v_{8,2}, v_{9,1}\}\}$

The last step is to merge the subresults of all partitions. CN_1 sends SPL_1 to CN_2 to get the shortest paths from $v_{1,1}$ to vertices of V_7. Obviously, $c_{1,1,7,1} = \min\{c_{1,1,4,1} + c_{4,1,7,1}, c_{1,1,4,2} + c_{4,2,7,1}, c_{1,1,4,3} + w_{4,3,7,1}\} = \min\{26, 26, 27\} = 26$, and $sp_{1,1,7,1} = sp_{1,1,4,1} \cup sp_{4,1,7,1}$. Similarly, $c_{1,1,7,2} = 27$, $sp_{1,1,7,2} = sp_{1,1,4,1} \cup sp_{4,1,7,2}$, $c_{1,1,7,3} = 25$, $sp_{1,1,7,3} = sp_{1,1,4,1} \cup sp_{4,1,7,3}$, $c_{1,1,7,4} = 23$, and $sp_{1,1,7,4} = sp_{1,1,4,1} \cup sp_{4,1,7,4}$.

CN_2 stores the shortest paths after merging, and it sends the new SPL_2 to CN_3. CN_3 calculates $c_{1,1,9,1} = \min\{c_{1,1,7,1} + c_{7,1,9,1}, c_{1,1,7,2} + c_{7,2,9,1}, c_{1,1,7,3} + w_{7,3,9,1}, c_{1,1,7,4} + w_{7,4,9,1}\} = \min\{38, 38, 33, 35\} = 33$ and $sp_{1,1,9,1} = sp_{1,1,7,3} \cup sp_{7,3,9,1} = \{v_{1,1}, v_{2,2}, v_{3,2}, v_{4,1}, v_{5,2}, v_{6,1}, v_{7,3}, v_{8,2}, v_{9,1}\}$.

5. Experiments and Analysis

5.1. Experimental Setup. Because there are no public multistage graph datasets, we synthesize 5 datasets using Java on IntelliJ IDEA, where the number of vertices of each stage and the weights of edges are random values satisfying (1). Table 2 presents the basic data of these 5 datasets.

The shortest paths algorithms are run on Hadoop [36] in conjunction with the Spark [37] computing engine. Spark is a fast and general-purpose computing engine designed for large-scale data processing [38]. The cluster consists of 8 computers, and each computer has a 4-core Intel processor, 8 GB memory, and 1 TB storage. The operating system is CentOS 7, and the distributed environment is built using Hadoop2.7 and Spark2.1.

In order to compare the performance of DMGA with existing algorithms, all graphs are partitioned to 8 partitions, which is a little different from Algorithm 2 whose number of partitions depends on the scale of the graph. Hence, the ML of each partition is

$$\mathrm{ML} = \gamma \frac{|E|}{8}, \tag{7}$$

in the experiments, and line 2 uses (7) to replace (4) in Algorithm 2.

5.2. Experimental Results

5.2.1. Partitioning Quality. At present, there is no partition algorithm for multistage graphs, so we compare the partition quality of the DMGA algorithm with the Hash partitioning algorithm. Hash is the default partitioning algorithm in many distributed graph processing systems, and it is the basis of most of existing distributed algorithms for solving the shortest path.

For the vertex-cut partitioning method, the number of mirror vertices reflects the communication overhead. The fewer the mirror vertices, the less the communication overhead, and the corresponding calculation time will be reduced. Figure 9 shows the number of mirror vertices generated by two graph partitioning algorithms running on the above 5 datasets. It can be seen from the results that the number of mirror vertices of the two graph partitioning algorithms for small datasets is almost the same. With the increase of the scale of the dataset and the number of stages, the number of mirror vertices of the DMGA algorithm is significantly less than the Hash algorithm. Specifically, the number of mirror vertices produced by the Hash algorithm for Graph_3, Graph_4, and Graph_5 is, respectively, 37.25, 24.97, and 69.9 times of the DMGA algorithm.

This shows that the DMGA algorithm has a better partitioning result for multistage graphs than the Hash algorithm. It can be deduced that the number of mirror vertices will be reduced in further if (4) instead of (7) is used to guide partition in Algorithm 2.

For the homogeneous cluster which consists of computing nodes with the same configuration, the load of each computing node should be balanced as much as possible in order to reduce the calculation time. Table 3 shows the number of edges on each computing node of each dataset generated by two graph partitioning algorithms. The Hash algorithm partitions the graph according to the Hash function defined as

$$k = \left(\sum_{l=1}^{i-1} n_l + j\right) \bmod 8, \tag{8}$$

which means the vertex $v_{i,j}$ is assigned to partition CN_k. Thus, the load of each partition is almost the same. The DMGA algorithm assigns all vertices in the same stage to the same partition, and the numbers of vertices in each stage are different, so the edge distribution is not balanced.

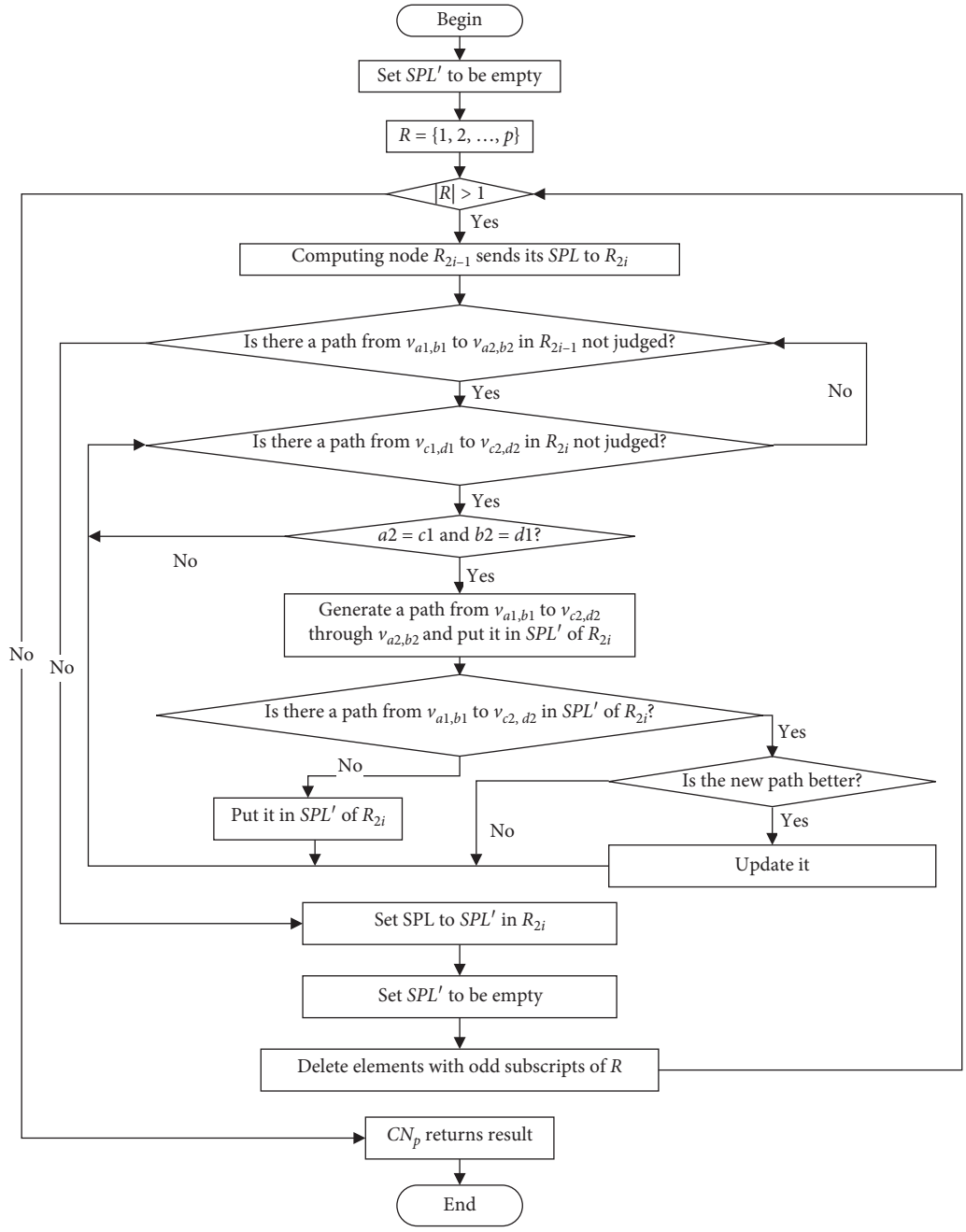

Figure 7: Flow diagram of global shortest path calculation.

The average deviation and standard deviation are further analysed to check the loads among partitions, and they are shown in Figure 10. In this figure, AVEDEV and STDEV, respectively, represent average deviation and standard deviation. We can see that the average and standard deviations of DMGA are 2.3 to 7.5 times of those of Hash. Graph_3 has the maximum difference among all graphs: the average deviation of DMGA is 6.65 times of that of Hash, and the standard deviation of DMGA is 7.51 times of that of Hash. Graph_1 has the minimal difference among all graphs: the average deviation of DMGA is 2.79 times of that of Hash, and the standard deviation of DMGA is 2.36 times of that of Hash. These results also show that Hash can produce more balanced partition.

5.2.2. Comparison of Running Time. We compare the execution time of four algorithms: single-machine, DMGA, SSSP [7] algorithm embedded in Pregel on Spark, and parallel Dijkstra [39–41]. The single-machine algorithm utilizes the sequential dynamic programming algorithm. The experiments are run 10 times, and the results are the average over these runs.

Table 4 presents the results, and Figure 11 shows the graphical comparison. It can be seen that DMGA has a

Require: The local shortest paths of each partition
Ensure: The global shortest path and its cost $c_{1,1,m,1}, sp_{1,1,m,1}$
(1) $R \longleftarrow \{1, 2, \ldots, p\}$
(2) for $k \longleftarrow 1$ to p do ▷all computing nodes do in parallel
(3) $\quad \mathrm{SPL}_k \longleftarrow \{c_{s_k,i,e_k,j}, sp_{s_k,i,e_k,j} | i = 1, 2, \ldots, n_{s_k}; j = 1, 2, \ldots, n_{e_k}\}$
(4) $\quad \mathrm{SPL}_k' \longleftarrow \varnothing$
(5) end for
(6) while $|R| > 1$ do
(7) $\quad$ for all $\{\mathrm{CN}_{R_{2i-1}} | i = 1, 2, \ldots, |R|/2\}$ do ▷all computing nodes do in parallel
(8) $\quad\quad \mathrm{CN}_{R_{2i-1}}$ sends $\mathrm{SPL}_{R_{2i-1}}$ to $\mathrm{CN}_{R_{2i}}$
(9) $\quad$ end for
(10) for all $\{\mathrm{CN}_{R_{2i}} | i = 1, 2, \ldots, |R|/2\}$ do ▷all computing nodes do in parallel
(11) $\quad$ for all $c_{a1,b1,a2,b2}, sp_{a1,b1,a2,b2} \in \mathrm{SPL}_{R_{2i-1}}$ do
(12) $\quad\quad$ for all $c_{c1,d1,c2,d2}, sp_{c1,d1,c2,d2} \in \mathrm{SPL}_{R_{2i}}$ do
(13) $\quad\quad$ if $a2 = c1 \wedge b2 = d1$ then
(14) $\quad\quad\quad$ if $sp_{a1,b1,c2,d2} = \varnothing$ then
(15) $\quad\quad\quad\quad c_{a1,b1,c2,d2} \longleftarrow c_{a1,b1,a2,b2} + c_{c1,d1,c2,d2}$
(16) $\quad\quad\quad\quad sp_{a1,b1,c2,d2} \longleftarrow sp_{a1,b1,a2,b2} \cup sp_{c1,d1,c2,d2}$
(17) $\quad\quad\quad\quad \mathrm{SPL}_{R_{2i}}' \longleftarrow \mathrm{SPL}_{R_{2i}}' \cup \{c_{a1,b1,c2,d2}, sp_{a1,b1,c2,d2}\}$
(18) $\quad\quad\quad$ else if $c_{a1,b1,c2,d2} > c_{a1,b1,a2,b2} + c_{c1,d1,c2,d2}$ then
(19) $\quad\quad\quad\quad c_{a1,b1,c2,d2} \longleftarrow c_{a1,b1,a2,b2} + c_{c1,d1,c2,d2}$
(20) $\quad\quad\quad\quad sp_{a1,b1,c2,d2} \longleftarrow sp_{a1,b1,a2,b2} \cup sp_{c1,d1,c2,d2}$
(21) $\quad\quad\quad\quad$ Update $c_{a1,b1,c2,d2}, sp_{a1,b1,c2,d2} \in \mathrm{SPL}_{R_{2i}}'$ with the new value
(22) $\quad\quad\quad$ end if
(23) $\quad\quad$ end if
(24) $\quad$ end for
(25) end for
(26) $\quad \mathrm{SPL}_{R_{2i}} \longleftarrow \mathrm{SPL}_{R_{2i}}'$
(27) $\quad \mathrm{SPL}_{R_{2i}}' \longleftarrow \varnothing$
(28) end for
(29) $\quad R \longleftarrow R - \{R_{2i-1} | i = 1, 2, \ldots, |R|/2\}$
(30) end while
(31) CN_p returns $c_{1,1,m,1}, sp_{1,1,m,1}$

Algorithm 4: Global shortest path calculation.

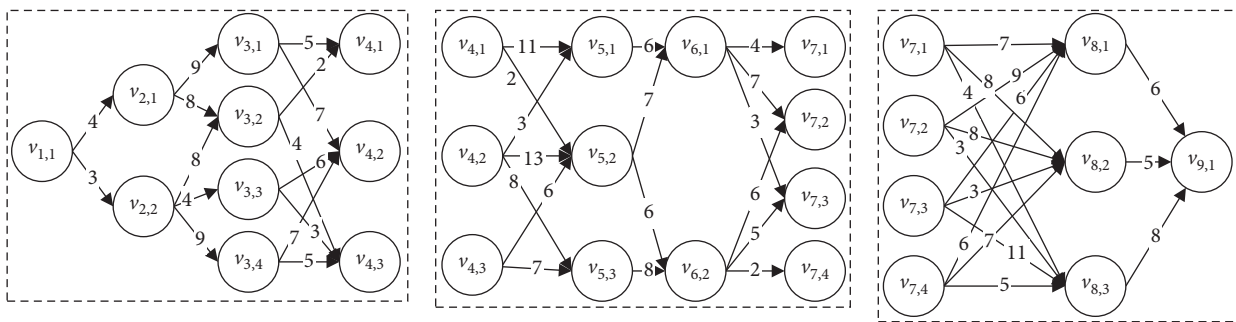

Figure 8: Partition of the example graph. (a) CN_1, 15 edges. (b) CN_2, 17 edges. (c) CN_3, 15 edges.

higher speedup ratio and time superiority over the sequential algorithm as the scale of dataset increases.

(i) Graph_1: the sequential algorithm is faster than DMGA because the communication among computing nodes of DMGA takes up a lot of time.

(ii) Graph_2: the running time of these two algorithms is almost the same.

(iii) Graph_3 and Graph 4: the running time of the sequential algorithm is, respectively, 1.72 and 3.13 times of that of DMGA.

Table 2: Multistage graph datasets.

Graph	Number of vertices	Number of edges	Number of stages
Graph_1	2000	5052	22
Graph_2	7002	21008	12
Graph_3	40002	199702	500
Graph_4	100000	499700	1000
Graph_5	1000020	5008710	10000

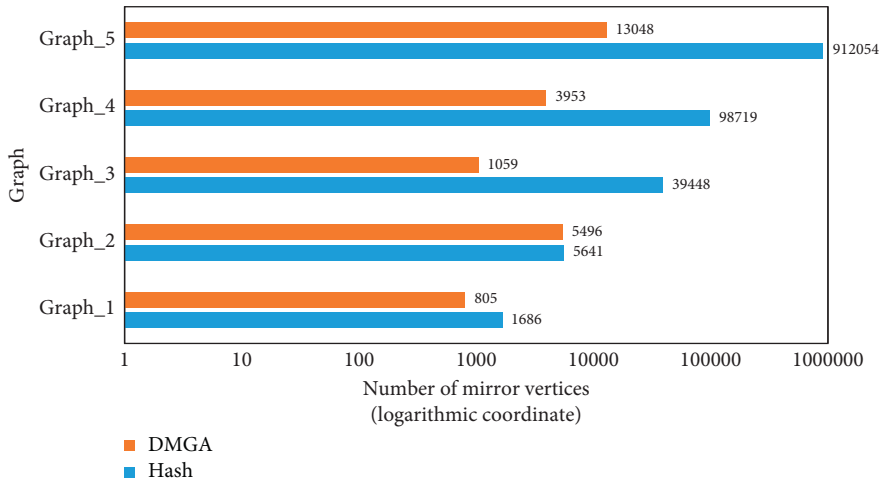

Figure 9: Number of mirror vertices.

Table 3: Edge distribution of graphs on computing nodes.

CN	Graph_1		Graph_2		Graph_3		Graph_4		Graph_5	
	Hash	DMGA	Hash	DMGA	Hash	DMGA	Hash	DMGA	Hash	DMGA
1	658	623	2553	2206	25138	25109	62395	62595	625049	620573
2	726	756	2835	3226	25139	26012	62677	60501	625274	605519
3	579	512	2661	2723	25145	24535	62279	64101	635371	646101
4	613	758	2544	3021	25078	26105	62039	61205	628324	612056
5	615	532	2579	2669	24901	25125	62901	63632	627591	636302
6	618	718	2572	2845	24590	24019	62410	61614	619432	621614
7	617	524	2636	2389	24836	22016	62836	63894	623853	639094
8	626	629	2628	1929	24875	26781	62163	62158	623816	627451

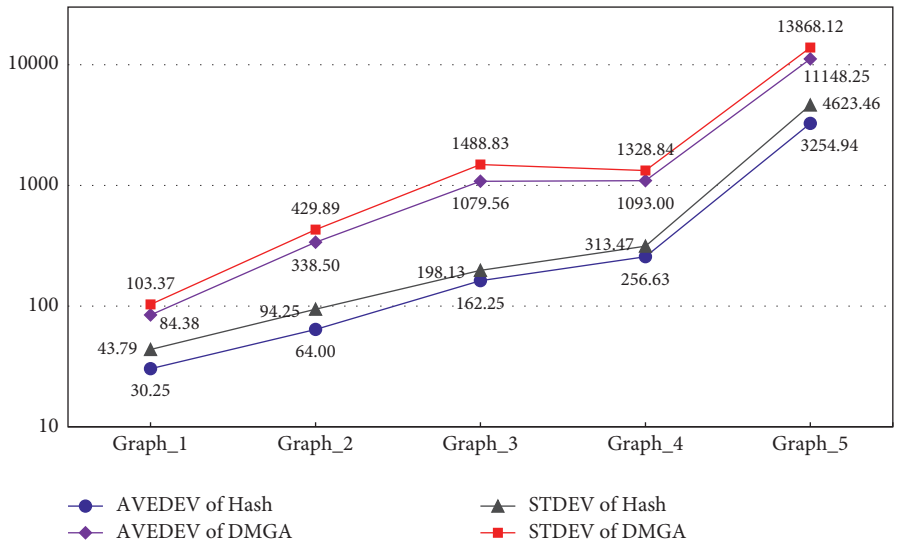

Figure 10: Average and standard deviations of edge distribution.

Table 4: Running time of four algorithms (unit: s).

Graph	Single machine	DMGA	SSSP	Parallel Dijkstra
Graph_1	4.06	5.56	8.56	101.57
Graph_2	17.91	15.52	11.14	452.36
Graph_3	59.13	34.46	1095.23	2642.58
Graph_4	381.56	122.06	1856.59	15893.45
Graph_5	–	7564.70	29324.73	85481.36

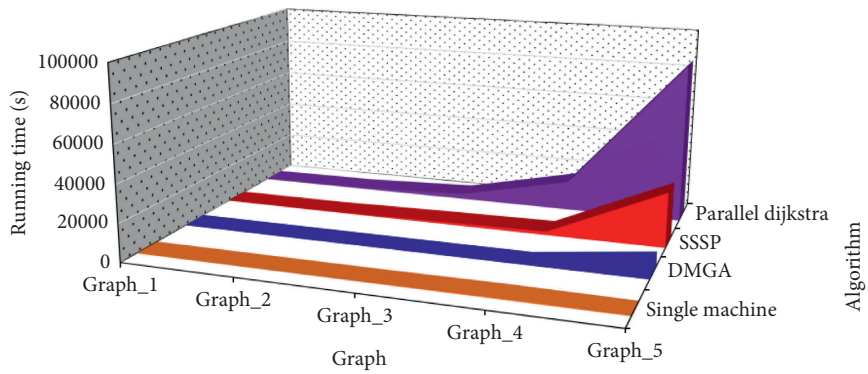

Figure 11: Comparison of running time of four algorithms.

(iv) Graph_5: the sequential algorithm meets "memory overflow" error, so this graph cannot be solved by a single machine. On the contrary, DMGA can obtain the shortest path.

In further, parallel Dijkstra has a relatively longer running time than the DMGA and SSSP, due to its high time and space complexity. SSSP uses Pregel's default graph partition method which does not take the structural feature of multistage graphs into account, so it has a large amount of communication. SSSP needs longer time for the larger scale graph, which implies that the communication overhead increases significantly with the increase of the scale of the graph specifically.

(i) Graph_2: SSSP has the least running time, and DMGA only needs 4 seconds longer than SSSP, but parallel Dijkstra needs more than 30 times of that of DMGA.

(ii) Other graphs: DMGA needs the least running time. The larger the scale of the graph, the more obvious the advantage of DMGA. For example, the running time of DMGA is, respectively, 25.8% and 8.8% of that of SSSP and parallel Dijkstra for Graph_5.

The above results show that DMGA makes full use of the special structural characteristic of multistage graphs, and it has extremely low communication overhead.

6. Conclusion and Future Work

Nowadays, graph models are widely applied in many fields, and the scale of the graph increases significantly. The existing distributed graph computing systems cannot make full use of the special characteristics of the multistage graphs. To this end, this paper proposes DMGA which is used to solve the shortest path problem of large-scale multistage graphs on a distributed computing system. DMGA consists of three phases: graph partition, local shortest path calculation, and global shortest path calculation. The experiment results demonstrate its high-performance. However, the load of DMGA is not balanced, and it only considers of the vertex-cut partitioning method. In future, we will focus on reducing the number of mirror vertices and solution time as much as possible under the premise of load balance and propose the special algorithm based on edge-cut partitioning idea.

Data Availability

The experimental data used to support the findings of this study are available from the corresponding author upon request and also provided in the supplementary information files.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported in part by the National Key R&D Plan of China under Grant 2018YFC1406203, Shandong Postgraduate Tutor Guidance Ability Improvement Project under Grant SDYY17040, and Science and Technology

Support Plan of Youth Innovation Team of Shandong Higher School under Grant 2019KJN024.

References

[1] Q. Peng, Y. Yu, and W. Wei, "The shortest path parallel algorithm on single source weighted multi-level graph," in *Proceedings of the 2nd International Workshop on Computer Science and Engineering (IWCSE)*, pp. 308–311, Vienna, Austria, October 2009.

[2] J. Tian, X. Wang, and H. Ding, "A genetic algorithm for solving booktitle graph problem," in *Proceedings of the 2016 Chinese Control and Decision Conference (CCDC)*, pp. 6395–6398, IEEE, Yinchuan, China, May 2016.

[3] H. Cui, J. Niu, C. Zhou, and M. Shu, "A multi-threading algorithm to detect and remove cycles in vertex- and arc-weighted digraph," *Algorithms*, vol. 10, no. 4, p. 115, 2017.

[4] A. Zafari, E. Larsson, and M. Tillenius, "DuctTeip: an efficient programming model for distributed task-based parallel computing," *Parallel Computing*, vol. 90, Article ID 102582, 2019.

[5] Y. Xu, H. Liu, and Z. Long, "A distributed computing framework for wind speed big data forecasting on Apache spark," *Sustainable Energy Technologies and Assessments*, vol. 37, Article ID 100582, 2020.

[6] D. Dai, Y. Chen, P. Carns et al., "GraphMeta: a graph-based engine for managing large-scale HPC rich metadata," in *Proceedings of the 2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 298–307, Taipei, Taiwan, September 2016.

[7] G. Malewicz, M. H. Austern, A. J. Bik et al., "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 135–146, ACM, Indianapolis, IN, USA, June 2010.

[8] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pp. 17–30, USENIX Association, Boston, MA ,USA, July 2012.

[9] J. E. Gonzalez, R. S. Xin, A. Dave et al., "GraphX: graph processing in a distributed dataflow framework," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pp. 599–613, USENIX Association, Boston, MA ,USA, July 2014.

[10] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: a resilient distributed graph system on spark," in *Proceedings of the 1st International Workshop on Graph Data Management Experiences and Systems (GRADES)*, pp. 1–6, ACM, Houston, TX, USA, May 2013.

[11] Y. Low, D. Bickson, J. Gonzalez et al., "Distributed GraphLab: a framework for machine learning and data mining in the cloud," *Proceeding of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[12] R. Chen, J. Shi, Y. Chen, and H. Chen, "PowerLyra: differentiated graph computation and partitioning on skewed graphs," in *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*, April 2018.

[13] J. Huang and D. J. Abadi, "Leopard: lightweight edge-oriented partitioning and replication for dynamic graphs," *Proceedings of the VLDB Endowment*, vol. 9, no. 7, pp. 540–551, 2016.

[14] K. Hu, G. Zeng, H. Jiang, and W. Wang, "Partitioning big graph with respect to arbitrary proportions in a streaming manner," *Future Generation Computer Systems*, vol. 80, pp. 1–11, 2018.

[15] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni, "HDRF: stream-based partitioning for power-law graphs," in *: Proceedings of the 24th ACM International on Conference on Information and Knowledge Management (CIKM '15)*, pp. 243–252, ACM, New York, NY, USA, May 2015.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, Cambridge, MA, USA, 3rd edition, 2009.

[17] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel GPU methods for single-source shortest paths," in *Proceedings of 2014 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 349–359, IEEE, Los Alamitos, CA, USA, June 2014.

[18] S. Maleki, D. Nguyen, A. Lenharth et al., "DSMR: a parallel algorithm for single-source shortest path problem," in *Proceedings of the 2016 International Conference on Supercomputing (ICS)*, ACM, New York, NY, USA, August 2016, 32.

[19] F. Busato and N. Bombieri, "An efficient implementation of the bellman-ford algorithm for kepler GPU architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2222–2233, 2016.

[20] C.-C. Huang, D. Nanongkai, and T. Saranurak, "Distributed exact weighted all-pairs shortest paths in $\tilde{O}(n^{5/4})$ rounds," in *Proceedings of the IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 168–179, Berkeley, CA, USA, May 2017.

[21] F. Liu, X. Tang, and Z. Yang, "An encoding algorithm based on the shortest path problem," in *Proceedings of the 14th International Conference on Computational Intelligence and Security (CIS)*, pp. 35–39, IEEE, Xi'an, China, December 2018.

[22] M. Ghaffari and J. Li, "Improved distributed algorithms for exact shortest paths," in *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pp. 431–444, ACM, Los Angeles, CA, USA, June 2018.

[23] Z. Su, J. Zhang, and Z. Lu, "A multi-stage metaheuristic algorithm for shortest simple path problem with must-pass nodes," *IEEE Access*, vol. 7, pp. 52142–52154, 2019.

[24] C. Mayer, R. Mayer, M. A. Tariq et al., "ADWISE: adaptive window-based streaming edge partitioning for high-speed graph processing," in *Proceedings of IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pp. 685–695, IEEE, Vienna, Austria, May 2018.

[25] Z. Abbas, V. Kalavri, P. Carbone, and V. Vlassov, "Streaming graph partitioning: an experimental study," *Proceedings of the VLDB Endowment*, vol. 11, pp. 1590–1603, 2018.

[26] N. M. Soudani, A. Fatemi, and M. Nematbakhsh, "An investigation of big graph partitioning methods for distribution of graphs in vertex-centric systems," *Distributed and Parallel Databases*, vol. 38, no. 1, pp. 1–29, 2020.

[27] L. A. R. Capelli, N. Brown, and J. M. Bull, "iPregel: strategies to deal with an extreme form of irregularity in vertex-centric graph processing," in *Proceedings of IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pp. 45–50, Denver, CO, USA, November 2019.

[28] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi, "A distributed algorithm for balanced graph partitioning," in *Proceedings of the IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems*, pp. 51–60, IEEE, Philadelphia, PA, USA, September 2013.

[29] Y. Akhremtsev, P. Sanders, and C. Schulz, "High-quality shared-memory graph partitioning," *IEEE Transactions on*

Parallel and Distributed Systems, vol. 31, no. 11, pp. 2710–2722, 2020.

[30] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "FENNEL: streaming graph partitioning for massive scale graphs," in *Proceedings of the 7th ACM International Conference on Web Search and Data Mining (WSDM '14)*, pp. 333–342, ACM, New York, NY, USA, June 2014.

[31] W. Zhang, Y. Chen, and D. Dai, "AKIN: a streaming graph partitioning algorithm for distributed graph storage systems," in *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 183–192, IEEE, Washington, DC, USA, May 2018.

[32] N. Wang, Z. Wang, Y. Gu, Y. Bao, and G. Yu, "TSH: easy-to-be distributed partitioning for large-scale graphs," *Future Generation Computer Systems*, vol. 101, pp. 804–818, 2019.

[33] S. Ji, C. Bu, L. Li, and X. Wu, "Local graph edge partitioning with a two-stage heuristic method," in *Proceedings of the 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 228–237, Dallas, TX, USA, July 2019.

[34] G. M. Slota, C. Root, K. Devine, K. Madduri, and S. Rajamanickam, "Scalable, multi-constraint, complex- objective graph partitioning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 12, pp. 2789–2801, 2020.

[35] A. C. Zhou, B. Shen, Y. Xiao, S. Ibrahim, and B. He, "Cost-aware partitioning for efficient large graph processing in geo-distributed datacenters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 7, pp. 1707–1723, 2020.

[36] D. Borthakur, J. Gray, and J. S. Sarma, "Apache hadoop goes real time at FaceBook," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 1071–1080, ACM, Vancouver, Canada, June 2011.

[37] J. G. Shanahan and L. Dai, "Large scale distributed data science using Apache Spark," in *Proceedings of the 21thACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 2323-2324, ACM, Sydney, Australia, August 2015.

[38] M. C. Lewis, "Big data analytics with spark," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE)*, p. 1242, March 2019.

[39] G. S. Brodal, J. L. Traff, and C. D. Zaroliagis, "A parallel priority data structure with applications," in *Proceedings of the 11th International Parallel Processing Symposium (IPPS)*, pp. 689–693, IEEE, Washington, DC, USA, April 1997.

[40] R. H. Mohring, H. Schilling, B. Schutz, D. Wagner, and T. Willhalm, "Partitioning graphs to speedup Dijkstra's algorithm," *ACM Journal of Experimental Algorithmics*, vol. 11, 2006.

[41] N. Jasika, N. Alispahic, A. Elma et al., "Dijkstra's shortest path algorithm serial and parallel execution performance analysis," in *Proceedings of the 35th International Convention MIPRO*, pp. 1811–1815, IEEE, Opatija, Croatia, May 2012.