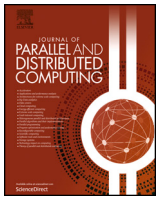
Journal of Parallel and Distributed Computing 186 (2024) 104830

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc



An efficient SSSP algorithm on time-evolving graphs with prediction of computation results

Yongli Cheng [a,e,f], Chuanjie Huang [a], Hong Jiang [b], Xianghao Xu [c,*], Fang Wang [d]

[a] College of Computer and Data Science, FuZhou University, Fuzhou, China
[b] Department of Computer Science Engineering, University of Texas at Arlington, USA
[c] School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing, China
[d] Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, China
[e] Fujian Key Laboratory of Network Computing and Intelligent Information Processing (Fuzhou University), Fuzhou, China
[f] Engineering Research Center of Big Data Intelligence, Ministry of Education, Fuzhou, China

ARTICLE INFO

Keywords:
Time-evolving graph
SSSP, Grouper
Predicted computation results

ABSTRACT

Many applications need to execute Single-Source Shortest Paths (SSSP) algorithm on each snapshot of a time-evolving graph, leading to long waiting times experienced by the users of such applications. However, these applications are often time-sensitive, the delayed computation results can lead to the loss of best decision-making opportunities. To address this problem, in this paper we propose an efficient SSSP algorithm for time-evolving graphs, called V-Grouper. The main idea of V-Grouper is to avoid the redundant computations of the same vertex in different snapshots. Our experimental results over real-world time-evolving graphs show that, due to the high similarity of consecutive snapshots, the computation results of one vertex in neighboring snapshots are equal with a high probability. At the beginning of computation, V-Grouper first divides all the versions of a given vertex in different snapshots into vertex groups, where the computation result of each version is predicted based on the aforementioned insight of neighboring snapshots having equal results. The versions of the vertex in each group have the same predicted computation result. During the computation process for each vertex group, only one version needs to participate in computation, avoiding a large number of redundant computations. Experimental results show that V-Grouper is up to 64.31× faster than the state-of-the-art SSSP algorithm.

1. Introduction

A time-evolving graph (TEG) consists of a series of snapshots, which has been increasingly used to predict future trends in the real world, by mining the inherent law among the historic snapshots of real-world graphs [16,26,42]. Take Fig. 1 as an example, a TEG algorithm typically consists of two steps: (1) executing an identical graph algorithm, such as the single-source shortest paths (SSSP) algorithm, on each snapshot respectively; (2) studying the development law of real world and predicting the future trend, based on the computation results of the snapshots. The goal of this paper is to efficiently compute the SSSPs on the snapshots of the time-evolving graph. The second step above is not our focus.

SSSP is a fundamental algorithm that has been used in many applications [21], such as transportation networks, communication, social networks, etc. For example, it is the building block for computing other important network analysis properties such as betweenness centrality [21]. In order to compute SSSP for each snapshot of a time-evolving graph, traditional algorithms (such as Dijkstra's algorithm) can only handle graphs with a small size, due to their high time complexity [21]. However, with real-world graphs that grow rapidly with time, traditional graph algorithms are not suitable for processing graphs with a large size. In order to address this problem, existing algorithms for large graphs are usually designed based on the bulk synchronous parallel (BSP) computation model [22]. This computation model is highly efficient and very user-friendly for coding and debugging large graph algorithms.

However, BSP-based SSSP algorithm is designed for static graph (single snapshot), which is inefficient for time-evolving graph with a large number of snapshots. Compared with a static graph computation task, a time-evolving graph computation task needs to execute an identical graph algorithm on multiple snapshots, leading to the execution time increasing with the number of snapshots. However, many applications

* Corresponding author.
E-mail address: xianghao@njust.edu.cn (X. Xu).

https://doi.org/10.1016/j.jpdc.2023.104830
Received 14 April 2023; Received in revised form 8 December 2023; Accepted 15 December 2023
Available online 19 December 2023

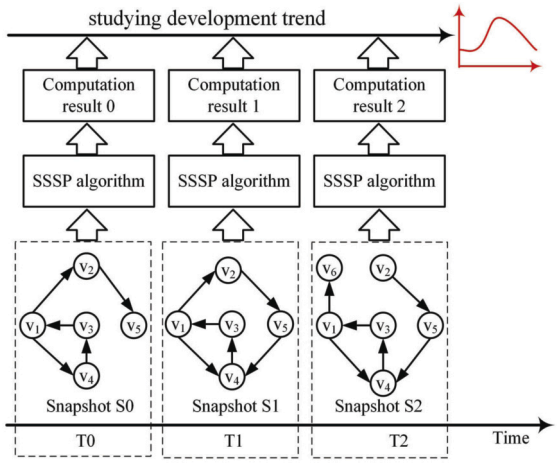


Fig. 1. Time-evolving graph algorithm (take SSSP as an example).

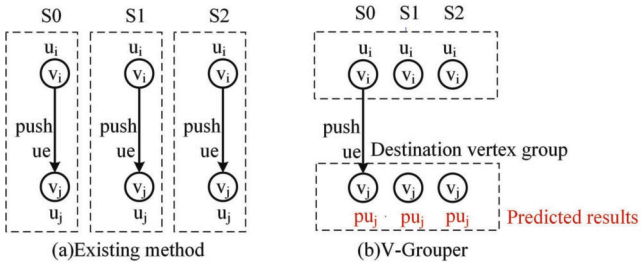


Fig. 2. Key difference between V-Grouper and existing BSP-based SSSP algorithm for time-evolving graph.

are time-sensitive, the delayed computation results can lead to the loss of best decision-making opportunities [32].

In this paper, we propose an efficient SSSP algorithm for time-evolving graphs, called V-Grouper. The main idea of V-Grouper is to avoid the redundant computations of the same vertex in different snapshots. Take the edge $\langle v_i, v_j \rangle$ in Fig. 2(a) as an example, existing BSP-based SSSP algorithm needs to execute three *push* operations in total, corresponding to the snapshots of S0, S1 and S2. In contrast, V-Grouper coalesces such operations by dividing the destination vertex v_j in different snapshots into vertex groups according to the predicted values. For the example of Fig. 2(b), where there is only one vertex group, V-Grouper ensures that the vertices in the vertex group will obtain the same computation result (pu_j) after computation finishes, via its prediction method as detailed in Section 3. For each vertex group, V-Grouper only needs to execute one *push* operation for the first snapshot to compute the current shortest path of destination vertex v_j; other vertices in the vertex group share the same computation directly.

The high performance of V-Grouper stems from the avoided *push* operations in each superstep of the execution process. Because the large number of *push* operations during the execution process of each snapshot dominate the execution time in both the distributed and single-node in-memory computation environments. In addition to the computation task, a *push* operation requires one round of communication from the source vertex to the destination vertex. In a distributed computation environment, this means inter-machine communication, typically by sending a message. The inter-machine communication is inefficient [6]. In a single-node in-memory computation environment, the communication between any pair of vertices requires one random memory access (read) from the source vertex and one random memory access (write) to the destination vertex. Random memory accesses are much less efficient than sequential memory accesses [16]. More importantly, modern computers with multi-core CPUs are largely based on the popular NUMA architecture [40,43] that provides DRAM on each of the multiple NUMA nodes and enables enlarged DRAM capacity by connecting these nodes with high-bandwidth links. A *push* operation requires a remote memory access from local NUMA node to one of the remote NUMA nodes, with a high probability. The remote memory access is inefficient, because each remote access has a penalty of high latency that is approximately 10× higher than the case of local memory access [25].

This paper makes the following contributions:

- **An efficient SSSP algorithm for time-evolving graphs.** We propose an efficient SSSP algorithm for time-evolving graph, called V-Grouper. Existing BSP-based SSSP algorithm computes the SSSPs of each snapshot of the time-evolving graph independently, resulting in a long execution time. Different from existing BSP-based SSSP algorithm, V-Grouper can accurately predict the computation results of the same vertex in different snapshots, and then avoid the redundant computations of the vertex that has the same predicted computation in different snapshots, leading to a significant performance improvement.
- **The implementation of V-Grouper.** We implemented V-Grouper on the single-node in-memory environment that relies on the NUMA-architecture. This graph computation environment is currently popular and used by many recent studies [37,43]. Experimental results show that V-Grouper is up to 64.31× faster than the state-of-the-art SSSP algorithm. Although we believe that it can also be built in a distributed environment, it is beyond the scope of this paper and we leave it as a topic of future study.

The rest of the paper is structured as follows. Background and motivation are presented in Section 2. Section 3 introduces our V-Grouper, a high-performance SSSP algorithm for time-evolving graph. We discuss several implementation details of V-Grouper in Section 4. Experimental evaluations of V-Grouper are presented in Section 5. We discuss related work in Section 6 and conclude the paper in Section 7.

2. Background and motivation

In this section, we first present a brief introduction to the Bulk Synchronous Parallel (BSP) computation model in Section 2.1, and then the BSP-based SSSP algorithm for time-evolving graphs in Section 2.2. We finally discuss the efficiency issue of this algorithm in Section 2.3. The insights gained through the efficiency issue of existing BSP-based SSSP algorithm for time-evolving graph help motivate us to propose the V-Grouper algorithm that eliminates the redundant computations incurred by existing BSP-based SSSP algorithm.

2.1. Bulk Synchronous Parallel (BSP) computation model

The Bulk Synchronous Parallel (BSP) model [22] is an excellent solution to process large-scale graphs [23,44], which works as follows. As mentioned before, to process a static graph G = (V, E), the graph computation job consists of a number of supertsteps; in each superstep, all vertices are executed in parallel by using the vertex-centric computation mode, that is, each vertex updates the statuses of its neighbors by sending messages [28] or accessing its neighbors directly by using *pull* or *push* modes [41]; the updated statuses will be used by the neighbors of the vertex in next superstep. The graph computation job proceeds and ends at the superstep where the convergence condition is met.

BSP computation model has been employed by many applications to implement different graph algorithms, such as Pagerank [24], due to the two following reasons. First, it is of high scalability, because the vertices can be assigned to more cores to obtain a higher performance. Second, the graph computation job can reach the convergence condition by using a limited number of superstep if the graph algorithm is well designed [28]. Although BSP computation model is designed for the static graph (single snaphot) at first, it can be extended easily for the algorithm of time-evolving graphs.

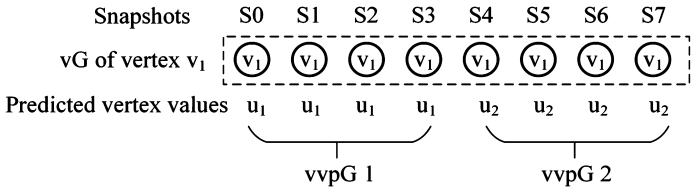


Fig. 3. Vertex group (vG) and vertex value predicted group (vvpGs).

2.2. BSP-based SSSP for time-evolving graph

Algorithm 1: BSP-based SSSP of time-evolving graph.

```
   Input: N; TEG = {S_0, S_1, ..., S_{N-1}};
   Output: sssps of each Snapshot(S_i); 0 ≤ i < N;
1  for each Snapshot S_i ∈ TEG do
       /* Initialize other vertices.                          */
2      for each vertex v ∈ S_i do
3          v.u = +∞;
4      end
       /* Initialize source vertex vs.                        */
5      vs.u = 0;
       /* Execute supersteps.                                 */
6      convergenceFlag = 0;
7      while convergenceFlag = 0 do
8          convergenceFlag = 1;
           /* vertices are executed in parallel.              */
9          for each vertex v ∈ V_i and v is active do
               /* Execute push operations. ue is the weight of
                  directed edge<v,v'>                          */
10             for each outgoing neighbor vertex v' of v do
11                 if v'.u > v.u + ue then
12                     v'.u = v.u + ue;
13                     convergenceFlag = 0;
14                 end
15             end
16         end
17     end
18 end
```

Algorithm 1 shows the BSP-based SSSP for time-evolving graph, which works as follows. The input of the BSP-based SSSP algorithm for time-evolving graph is a snapshot set $TEG = \{S_0, S_1, \cdots, S_{N-1}\}$. Each snapshot can be defined as $S_i = (V_i, E_i)$, where $0 \leq i < N$, V_i is the vertex set of S_i, E_i is the directed edge set of S_i. The BSP-based SSSP algorithm for time-evolving graph needs to execute N computation tasks to compute the single-source shortest paths for each snapshot S_i (Lines 1∼18). Each computation task consists of two stages. In the initialization stage (Lines 2∼5), the value of source vertex vs is set to 0, the values of other vertices are set to $+\infty$; this means that the distance from the source vertex vs to any one of other vertices is currently unknown. In the computation stage, the computation task proceeds with a number of supersteps (Lines 6∼17); in each superstep, all active vertices are executed in parallel (Lines 9∼16); when executing each vertex v (Lines 10∼15), there is one *push* operation for each outgoing neighbor v'; the *push* operation updates the value of v' if a shorter path has been received; the updated values of v' will be used in next superstep by its outgoing neighbors. An *active vertex* is defined as the vertex that has received a shorter path from its incoming neighbors in the previous superstep; the source vertex vs is special, which is only active in the first superstep. The graph computation job ends at the superstep where each vertex does not receive a shorter distance.

2.3. Motivation

Like other time-evolving graph algorithms, the key problem of the BSP-based SSSP for time-evolving graph is the long execution time. As shown in Algorithm 1, the time-evolving graph computation job needs to execute each snapshot of the time-evolving graph respectively. During the execution of each snapshot, there are a number of superstep. In each superstep, each vertex needs to execute one *push* operation for each of its out-going neighbors. Hence, there are a large number of *push* operations during the whole execution process of the algorithm. As mentioned before, the large number of *push* operations during the execution process of each snapshot dominate the execution time in both the distributed and single-node in-memory computation environments.

Motivated by the issue of the algorithm execution efficiency above, we propose V-Grouper, an efficient SSSP algorithm for time-evolving graphs. V-Grouper first divides all the versions of a given vertex in different snapshots into vertex groups, where the computation result of each version is predicted based on the aforementioned insight of neighboring snapshots having equal results. The versions of the vertex in each group have the same predicted computation result. During the computation process for each vertex group, only one version needs to participate in computation, avoiding a large number of redundant computations.

3. V-Grouper

In this Section, we first define three terminologies in subsection 3.1. This helps us present the V-Grouper algorithm in subsection 3.2. We detail the key component of V-Grouper in subsection 3.3, i.e., predicting and grouping. Performance analysis of V-Grouper is presented in subsection 3.4.

3.1. Definitions

Three terminologies are defined as follows.

- **Time-evolving graph (TEG).** A TEG is set of snapshots, which can be represented as $TEG = \{S_i\}$, where $0 \leq i < N$, N is the number of snapshots. Each snapshot $S_i = (V_i, E_i)$ is composed of a set of vertices V_i and a set of directed edges $E_i \in V_i \times V_i$.
- **Vertex group (vG).** A TEG with N snapshots have a number of vertex groups (vGs). A vG can be defined as follows. For each vertex v, the vertex group vG has N elements; the ith element represents that snapshot S_i contains vertex v, where $0 \leq i < N$. Take Fig. 3 as an example, there are 8 snapshots; the vertex group of v_1 can be represented as $\{v_1, v_1, v_1, v_1, v_1, v_1, v_1, v_1\}$; if snapshot S_1 does not contain vertex v_1, the vertex group of v_1 can be represented as $\{v_1, null, v_1, v_1, v_1, v_1, v_1, v_1\}$.
- **Vertex value predicted group (vvpG).** Each vertex group vG can be further divided into vertex groups with smaller sizes before executing computation, by predicting the computation results of one vertex in different snapshots. We define each vertex group with a small size as a vvpG. All the vertices in a vvpG will obtain an equal computation result after the computation finished. Take Fig. 3 as an example, vG of v_1 has two vvpGs, vvpG 1 and vvpG 2. All the vertices in vvpG 1 have the same computation result u_1; similarly, all the vertices in vvpG 2 have the same computation result u_2.

3.2. Algorithm

As shown in Algorithm 2, V-Grouper works as follows. The input of V-Grouper is a snapshot set $TEG = \{S_0, S_1, \cdots, S_{N-1}\}$. In the initialization stage (Lines 1∼6), each snapshot S_i is initialized as follows. The

value of source vertex vs is set to 0, the values of other vertices are set to $+\infty$; this means that the distance from the source vertex vs to each of other vertices is currently unknown. In the computation stage, the computation task proceeds with a number of supersteps (Lines 7∼21); in each superstep, all active vertex groups are executed in parallel (Lines 9∼20). An *active vertex group* is defined as the vertex group that has received at least one shorter path from its incoming neighbors in the previous superstep; the source vertex group is special, which is only active in the first superstep. When executing each vertex group vG, it needs to update each of its outgoing neighbor vG'. For each outgoing neighbor vG', V-Grouper first divides vG' into vvpGs by predicting each computation result of vertex v' in vG' (Line 12, see Section 3.3); the computation results of vertex v' in same vvpG will be equal after computation finished. V-Grouper then only need to execute one *push* operation for each vvpG (Lines 13∼18); the *push* operation updates the value if a shorter path has been received (Lines 14∼17); the updated values of v' will be used in next superstep by its outgoing neighbors. The graph computation job ends at the superstep where each vertex does not receive a shorter distance.

Algorithm 2: V-Grouper.

```
   Input: N; TEG = {S_0, S_1, ..., S_{N-1}};
   Output: sssps of each Snapshot(S_i); 0 ≤ i < N;
   /* Initialize vertices of all snapshots.                    */
1  for each Snapshot S_i ∈ TEG do
       /* Initialize all vertices.                             */
2      for each vertex v ∈ S_i do
3          v.u = +∞;
4      end
       /* Initialize source vertex vs.                         */
5      vs.u = 0;
6  end
   /* Execute supersteps for vertex group vGs.                 */
7  convergenceFlag = 0;
8  while convergenceFlag = 0 do
9      convergenceFlag = 1;
       /* vGs are executed in parallel.                        */
10     for each active vertex group vG do
11         for each outgoing neighbor vG' of vG do
12             divide vG' of vertex v' into vvpGs;
13             for each vvpG do
                   /* Execute push operation. ue is the weight of
                      directed edge<v,v'>                       */
14                 if v'.u > v.u + ue then
15                     v'.u = v.u + ue;
16                     convergenceFlag = 0;
17                 end
18             end
19         end
20     end
21 end
```

3.3. Predicting and grouping

We discuss the key problem, that is, a vertex group vG is divided into vvpGs before executing computation, where the vertices in one vvpG will obtain an equal value after the computation finished. Take Fig. 4(a) as an example, the TEG has 9 snapshots, there are two vertex groups, i.e., vG of vertex v_1 and vG of vertex v_2. Vertex v_2 is one of the outgoing neighbors of vertex v_1. Assume that we need to compute the shorter path of vertex v_2 in any one of the snapshots, a *push* operation should be executed, as shown in Equation (1). Where v_1.u is the current value of vertex v_1, eu is the weight of edge $\langle v_1, v_2\rangle$, and v_2.u is the current value of vertex v_2, v_2.u' is the new value of vertex v_2.

$$push: v_2.u' = min(v_1.u + eu, v_2.u) \tag{1}$$

According to Equation (1), in order to predict that the vertices in one vvpG will obtain an equal computation result, three conditions should be satisfied. First, the current values of vertex v_1 in the vvpG should be equal. Second, current values of vertex v_2 should be also equal. Similarly, the weights of edge $\langle v_1, v_2\rangle$ should be equal. In fact, the weight of edge rarely changes in many real-world graph, such as the road graph [20].

Hence, the predicting and grouping can be implemented easily by sequentially accessing the current values of vertex v_1, the current values of vertex v_2, and the weights of edge$\langle v_1, v_2\rangle$. Initially, the first vvpG (vvpG 1) is generated. During the process of each sequential access, if a new value is checked, a new vvpG will be generated. For example, at the point of S3, vertex v_1 obtains the new value of u_2, vvpG 2 is generated; at the point S6, the edge$\langle v_1, v_2\rangle$ obtains the new value of eu_2, the vvpG 3 is generated.

3.4. Performance analysis

Avoided *push* operations. By predicting the computation results, each destination vertex group (vG) is divided into vvpGs. Our experimental results show that each vvpG has a large size. V-Grouper only needs to execute one *push* operation for each vvpG, avoiding a large number of *push* operations. In addition to the computation task, a *push* operation requires one round of costly communication from the source vertex to the destination vertex both in the distributed and single-node in-memory computation environments. Take the latter as an example, modern computers with multi-core CPUs are largely based on the popular NUMA architecture [37,43] that provides DRAM on each of the multiple NUMA nodes and enables enlarged DRAM capacity by connecting these nodes with high-bandwidth links. A *push* operation requires a remote memory access from local NUMA node to one of the remote NUMA nodes, with a high probability. The remote memory access is inefficient, because each remote access has a penalty of high latency that is approximately 10× higher than the case of local memory access [4].

Low overhead of predicting. Compared with the gain from the avoided *push* operations, the extra overhead of predicting and grouping is negligible. As detailed in Section 3.3, in order to divide each destination vertex group (vG) into vvpGs, V-Grouper only needs to access the current values of source vertex group, the current values of destination vertex group, and the weights of edge group. This process is highly efficient. First, there are no communications among the three groups. Second, the values in each group are grouped together, which can be accessed sequentially to obtain a high efficiency. More importantly, V-Grouper does not need to access all the values in the group when accessing each group. Take vertex group of v_1 in Fig. 4(b) as an example, V-Grouper only needs to access the compact data structure which has only a small number of data units. Each data unit, such as "u_1 3", means that a number of consecutive values are all equal.

Effect of snapshot similarity. The performance of V-Grouper is dominated mainly by the similarity among the snapshots of the time-evolving graph. Hence, V-Grouper can obtain a high performance when processing the time-evolving graphs with different size, if the time-evolving graphs have a high snapshot similarity. Because a high similarity among the consecutive snapshots means that the vertices in each vertex group will obtain the same computation result with a high probability. Real-world time-evolving graphs have a high snapshot similarity, e.g., only about 0.09% of edges between any two consecutive snapshots of the Weibo dataset are different. A high snapshot similarity will lead to a large average size of vvpG, avoiding more *push* operations. For example, when executing V-Grouper on the Flickr dataset with 100 snapshots, the average size of vvpG is 61.

4. Implementation

In this section, we discuss several key implementation issues that are important for V-Grouper.

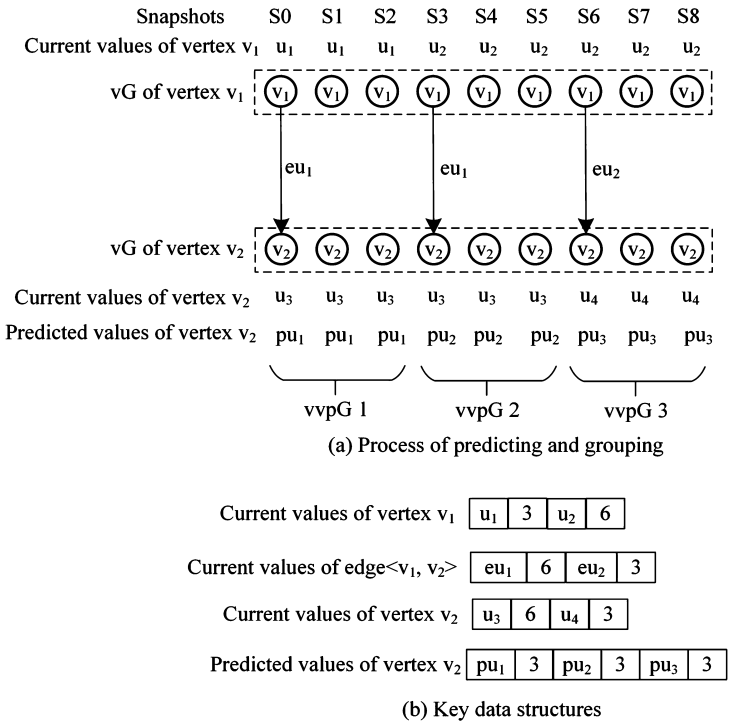


Fig. 4. Predicting and grouping.

4.1. System architecture

There are three kinds of mainstream graph-processing architectures as follows.

Distributed system architecture. Distributed graph-processing systems, such as Pregel [28], can obtain a higher performance by adding compute nodes. However, to scale out to obtain a high performance, these systems often require excessive investment in compute nodes, leading to high hardware cost. Furthermore, the scalability of these systems is limited due to the inefficient communication process that needs to send high-frequency and fine-grained messages [5,7].

Disk-based system architecture. Disk-based graph-processing systems, such as GraphChi [24], can handle large-scale graphs on just a personal computer by using the large storage space of magnetic disks. However, the performance of these systems is poor due to the costly disk I/O [41], in spite of many efforts have been done. V-Grouper, as a time-evolving graph algorithm, needs to process a large number snapshots, requiring a more efficient system architecture to obtain an acceptable performance.

In-memory system architecture. In-memory graph-processing systems, such as Ligra [36], can process large-scale graphs by using a server with large memory. The advantages of these systems are three-fold. First, the memory space of a current server can be easily extended to several TB with the reducing price, which is large enough for most real-world graphs [15]. Second, these systems avoid the excessive investment that distributed graph-processing systems suffer from. More importantly, these systems can obtain a high performance by using the large number of cores, without the costly communication of distributed graph-processing systems.

Based on the insights of three kinds of graph-processing architectures, like existing in-memory graph-processing systems, we choose the in-memory system architecture to implement our V-Grouper.

4.2. Memory utilization

Memory remains an important resource when implementing V-Grouper, in spite of the memory space of a current server is large enough to accommodate most real-world graphs. Because, as a time-evolving graph algorithm, V-Grouper needs to process a large number of snapshots in memory. To address this problem, we carefully design the memory data structures of the graph structure and the intermediate results.

Graph structure. For a time-evolving graph with N snapshots, each object (vertex or edge) does not need to be stored N times. Instead, each object is stored only once with a bitmap of N bits. Each bit of the bitmap for an object corresponds to a snapshot and indicates the presence/absence (1/0) of this object in the snapshot, making it very efficient to determine if an object belongs to a certain snapshot or not. By using this design, only one bit is needed to represent a repeated object in a snapshot. However, the bitmaps still require large memory space if the algorithm needs to load a large number of snapshots into memory. To address this problem, we borrow an efficient compression method for bitmaps, called PLWAH [10]. It can obtain a high compression ratio of 5× ∼100× according to the size of the bitmaps and the feature of the graph, while only leading to a slight performance degradation when accessing the objects.

Intermediate results. As shown in Fig. 4, the main intermediate results in each superstep include three parts, i.e., the values of each vertex in different snapshots, the values of each edge in different snapshots, the predicted values of each vertex in different snapshots. To improve the memory utilization, we carefully design the data structures for the three parts of intermediate results. Take the values of vertex v_1 in Fig. 4(b) as an example, we do not need to store N values for a vertex group. Instead, they are stored by using two data elements. For example, the data element of "u_1, 3" represents that the value u_1 is repeated 3 times. This design is simple but effective because, for the real-world time-evolving graph, the size of each vvpG is usually large, requiring only a small number of data elements to store the values of each vertex group.

Due to the design of high memory utilization, V-Grouper can load snapshots into memory as many as possible according to the requirement of applications. In the case of that the snapshots exceed the memory space of the server, V-Grouper divides them into snapshot groups with a smaller size, each can be loaded into memory.

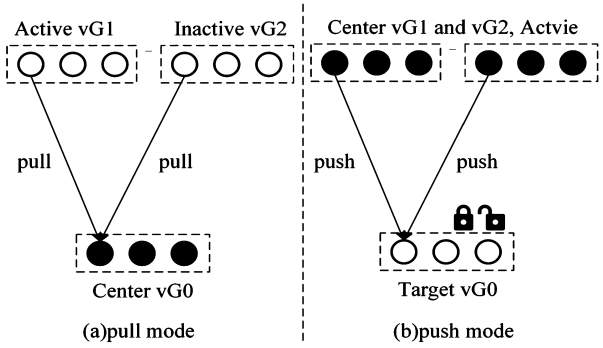


Fig. 5. Interactions between vertex groups.

4.3. Interactions between vertex groups

As mentioned in Section 3.2, in each superstep, all vertex groups are executed in parallel, each needs to exchange messages with its neighbors. There are two interaction modes, i.e., *pull* and *push* [41]. As shown in Fig. 5(a), if the *pull* mode is used, each vertex group (center vG0) needs to finish its computation by accessing all its incoming neighbors (vG1 and vG2). This mode is efficient for the graph algorithms, such as Pagerank [30], where most vertex groups need to interact with its neighbors. However, the *pull* mode is inefficient for V-Grouper because, a large number of vertex groups are inactive in many supersteps. The inactive vertices do not receive the shorter distance in the previous superstep. The values of the inactive vertex group do not contribute to the computation result of the center vertex vG0.

We then turn to the *push* mode. As shown in Fig. 5(b), by using this mode, only active vertex groups need to interact with its outgoing neighbors, avoiding the useless computation of inactive vertex groups. However, conflict occurs when several vertex groups update one of their common neighbors simultaneously. Hence, a lock is required for each vertex group. The mechanism of lock can bring an extra performance overhead. In order to reduce the extra performance overhead, we employ an efficient lock mechanism, called compare-and-swap (CAS) [36], to provide the atomic operation. Experimental results show that, compared with the large number of useless computations, this extra performance overhead is negligible.

5. Experimental evaluation

Extensive experiments have been conducted to evaluate the efficiency of V-Grouper. In this section, we first present the experimental setup in subsection 5.1. We then study the effect of the avoided *push* operations in subsection 5.2, and further study the efficiency of the interaction modes in subsection 5.3. We compare V-Grouper with the existing BSP-based SSSP algorithm for time-evolving graphs in subsection 5.4. We compare V-Grouper with incremental computation method in subsection 5.5, and evaluate the memory utilization of V-Grouper in subsection 5.6. Effect of snapshot similarity is evaluated in subsection 5.7.

5.1. Experimental setup

We conduct experiments on a server equipped with two NUMA nodes, which is a popular architecture of modern computers. Each node contains an Intel(R) Xeon(R) E5-2650 CPU (2.50GH) processor with 24 cores and 128 GB DRAM, i.e., 48 cores and 256 GB DRAM in total. Like several previous studies [16,19,33], we use three real-world time-evolving graph datasets, which are summarized in Table 1. The time-evolving graph dataset Flickr [11] consists of 100 snapshots; the average snapshot has 22974 vertices and 329316 edges; this dataset with 100 snapshots has 32931637 edges in total; the interval between any two consecutive snapshots is one day. The time-evolving graph dataset Youtube [11] consists of 100 snapshots; the average snapshot has 32235 vertices and 92289 edges; this dataset with 100 snapshots has 9228909 edges in total; the interval between any two consecutive snapshots is one day. The time-evolving graph dataset Weibo [9] consists of 27 snapshots; the average snapshot has 66201 vertices and 15669304 edges; this dataset with 27 snapshots has 423071214 edges in total; the interval between any two consecutive snapshots is one day. We implemented the existing BSP-based SSSP algorithm for time-evolving graphs, in order to compare it with V-Grouper.

Table 1
Datasets of time-evolving graphs.

DataSet	Flickr [11]	Youtube [11]	Weibo [9]
# of snapshots	100	100	27
Interval	1 day	1 day	1 day
Edges/snapshot	329316	92289	15669304
Edges(total)	32931637	9228909	423071214

5.2. Effect of avoided push operations

As mentioned in Section 3, in each superstep of the execution process of V-Grouper, all vertex groups are executed in parallel. For a time-evolving graph with N snapshots and M vertices, V-Grouper creates M vGs, each has N elements; for each element i in a vertex group vG, it is set to the vertex ID if this vertex belongs to snapshot i, otherwise it is set to -1 (null). vvpGs are generated during the execution process of the algorithm. We use C++ OpenMP architecture to execute the vGs in parallel, which can assign vGs to multiple threads. When executing each vertex group vG, it needs to interact with each of its neighbors vG'; during the process of interaction, V-Grouper divides the vG' into smaller groups, called vertex value predicted groups (vvpGs); the vertices in each vvpG will obtain an identical value after the computation finished. Hence, for each vvpG, V-Grouper only needs to execute one *push* operation to finish the interaction, avoiding a large number of *push* operations. In this section, we study the relationship between the performance of V-Grouper and the number of the avoided *push* operations.

Experiments are conducted by running V-Grouper on the time-evolving graph dataset Youtube with the number of snapshots varying from 1 to 100. V-Grouper reaches convergence at 13^{th}, 14^{th}, 15^{th}, 17^{th}, 18^{th} and 18^{th} superstep respectively when the number of snapshots is 1, 20, 40, 60, 80 and 100. In each experiment, we record the number of avoided *push* operations, and the execution time of an average snapshot. We take the case of one snapshot as the baseline, and normalize the experimental results to an average snapshot. As shown in Fig. 6(a), at the point of one snapshot, since the number of avoided *push* operations is 0, no improvement can be obtained in this case. V-Grouper obtains a performance improvement of 14.28× over baseline, at the point of 20 snapshots. The reason is that V-Grouper avoids 19197569 *push* operations in this experiment. V-Grouper obtains a continuous performance improvement with the number of snapshots increasing, and reaches its peak performance at the point of 100 snapshots. The speedup is 27.2× in this case. The reason is that the size of the average vvpG increases when executing the time-evolving graph with a larger number of snapshots, indicating that more *push* operations can be avoided. Hence, V-Grouper can obtain a larger speedup when executing the time-evolving graph with a larger number of snapshots. From Fig. 6(a), we also observe that the speedup increases slowly from the point of 60 snapshots to 100 snapshots. The reason is that the maximum value of the size of the average vvpG has been achieved.

Experiments are also conducted by running V-Grouper on the time-evolving graph dataset Flickr with the number of snapshots varying from 1 to 100. Compared with Youtube, V-Grouper obtains a higher performance improvement in each experiment. As shown in Fig. 6(b), V-Grouper is 17.46× faster than the baseline when executing 20 snap-

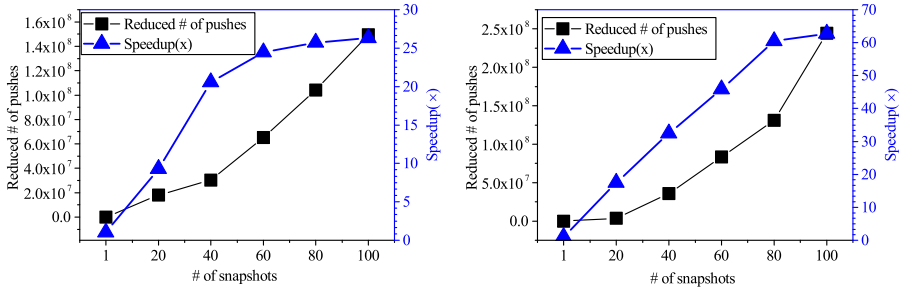


Fig. 6. Effect of avoided push operations.

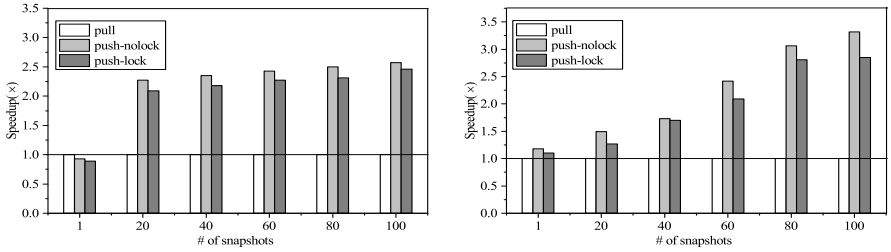


Fig. 7. Efficiency of interactions of vertex groups.

shots. At the point of 100 snapshots, it obtains a performance improvement of 62.71×. The speedup of Youtube is 27.2× at the point of 100 snapshots. The reason is that, compared with Youtube, the difference between any two consecutive snapshots of Flickr is smaller, leading to a larger size of the average vvpG. A vvpG with a larger size will avoid a larger number of *push* operations during the execution process of the algorithm. For example, at the point of 100 snapshots, the size of the average vvpG of Flickr is 73, while the size of the average vvpG of Youtube is 59 only. During the execution process, V-Grouper only needs to execute one *push* operation for each vvpG, regardless of the size of the vvpG. Hence, The larger size of vvpG can avoid more *push* operations, resulting a higher performance. For example, V-Grouper avoids 1.42×10^8 and 2.38×10^8 *push* operations respectively on Youtube and Flickr, at the point of 100 snapshots, resulting in the different performances of the algorithm.

5.3. Efficiency of interactions between vertex groups

We study V-Grouper in terms of efficiency of the interactions between vertex groups. As mentioned in section 3, in each superstep, all vertex groups (vGs) are executed in parallel. When executing each vG, it needs to interact with its neighbors; for each neighbor vG', it is divided into vvpGs. There are two interaction modes when vG exchanges message with each vvpG, i.e., *pull* and *push*. We implement three interaction modes to study the performance of V-Grouper, that is, *pull*, *push-lock*, and *push-nolock*. The *push-nolock* is implemented to study the lock overhead of the *push* mode by comparing it with *push* mode. It can only be used in experimental environment. Because it will lead to incorrect computation result.

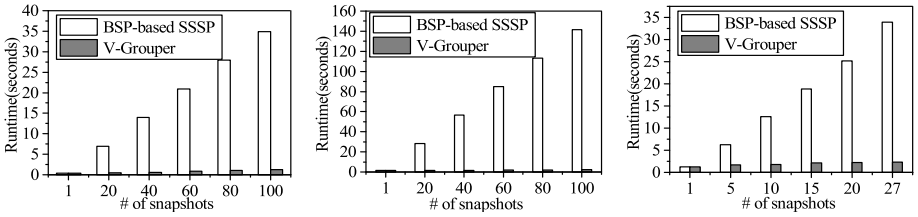
Experiments are first conducted by running V-Grouper on Youtube with the number of snapshots varying from 1 to 100. We take the execution time of *pull* as the baseline. As shown in Fig. 7(a), at the point of 1 snapshot, *push-lock* mode is slightly slower than *pull* mode. However, at the point of 20 snapshots, *push-lock* is 2.09× faster than the *pull* mode. The reason is that a larger number of *push* operations is avoided in the latter case. We also observe that the difference between the *push-lock* and *push-nolock* is small. The *push-nolock* is only 1.03× faster than *push-lock*, indicating that the overhead of lock of *push* mode is small. We also obtain similar results from the experiments at the points of 40 snapshots, 60 snapshots and 100 snapshots.

Experiments are then conducted by running V-Grouper on Flickr with the number of snapshots varying from 1 to 100. Compared with Youtube, as shown in Fig. 7(b), there are two differences in the experimental results. First, at the point of 1 snapshot, the *push-lock* and *push-nolock* modes are faster than the *pull* mode. The reason is that the degree of the average vertex of Flickr is larger than that of Youtube. An average vertex of Youtube has 2.86 neighbors, while the case of Flickr is 14. A larger degree makes the *pull* mode to execute more useless computations. Second, the *push-lock* and *push-nolock* obtain a performance improvement in case of Flickr than the case of Youtube. The reason is that, the size of an average vvpG of Flickr is larger than that of Youtube.

5.4. Compared with BSP-based SSSP

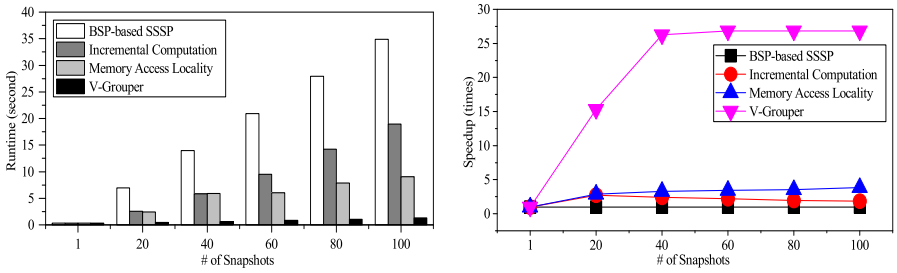
We implemented BSP-based SSSP algorithm for time-evolving graphs to compare V-Grouper with it in terms of runtime. V-Grouper is implemented with *push-lock* mode. In each experiment, we take the BSP-based SSSP algorithm as baseline. We first run BSP-based SSSP and V-Grouper on the time-evolving graph dataset Youtube by using 1, 20, 40, 60, 80 and 100 snapshots respectively. As shown in Fig. 8(a), at the point of 1 snapshot, the runtime of V-Grouper is similar to that of baseline. However, at the point of 20 snapshots, the runtime of v-Grouper is 0.48 seconds that is 14.5× faster than the case of baseline (6.98 seconds). V-Grouper continue to obtain a higher performance improvement with the number of snapshots increasing. At the point of 100 snapshots, V-Grouper is 26.84× faster than BSP-based SSSP. The experimental results indicate that V-Grouper can obtain a higher performance when processing more snapshots. The reason is that it can avoid more *push* operations in this case.

We then run BSP-based SSSP and V-Grouper on the time-evolving graph dataset Flickr by using 1, 20, 40, 60, 80 and 100 snapshots respectively. We also observe the similar results from the experiments. As shown in Fig. 8(b), the difference between two datasets is that, in each experiment, the speedup of Flickr is higher than that of Youtube. At



(a) Youtube (b) Flickr (c) Weibo

Fig. 8. Compared with BSP-based SSSP algorithm.



(a) Runtime (b) Speedup

Fig. 9. Compared with other methods.

the point of 100 snapshots, V-Grouper is 64.31× faster than BSP-based SSSP. As mentioned before, the reason is that the size of an average vvpG of Flickr is larger than that of Youtube. The larger size of vvpG can avoid more *push* operations, leading to a higher performance improvement.

We finally run BSP-based SSSP and V-Grouper on the time-evolving graph dataset Weibo by using 1, 5, 10, 15, 20 and 27 snapshots respectively. As mentioned in Section 5.1, time-evolving graph dataset Weibo has 27 snapshots only. As shown in Fig. 8(c), V-Grouper is 1.08×, 3.61×, 6.87×, 8.68×, 11.32× and 11.45× faster than baseline respectively at the points of 1, 5, 10, 15, 20 and 27 snapshots. The experimental results indicate that V-Grouper can also obtain a significant performance improvement when processing snapshots with a smaller size.

5.5. Compared with other methods

We first compare V-Grouper with incremental computation method that is another method to optimizing graph computation efficiency on a series of snapshots [13]. Incremental computation method first selects the first snapshot (S_0) as the baseline snapshot and computes the SSSPs for the baseline snapshot. Then, other snapshots are executed based on the computation result of the baseline snapshot. We then compare V-Grouper with memory locality optimization method [16]. This method organizes the information for the same vertex across different snapshots together. During the execution process of algorithm, the information for the same vertex across different snapshots can be accessed continuously in memory, reducing the number of cache misses. Experiments are conducted on the time-evolving graph dataset Youtube by using 1, 20, 40, 60, 80 and 100 snapshots respectively. In each experiment, we take the BSP-based SSSP algorithm as baseline.

As shown in Fig. 9, at the point of 1 snapshot, the runtime of incremental computation method is similar to that of baseline. The incremental computation method obtains 2.7×, 2.4×, 2.2×, 1.96× and 1.84× speedup respectively at the points of 20, 40, 60, 80 and 100 snapshots. The reason is that the incremental computation method can obtain a higher efficiency when executing the snapshots that are closer to the baseline snapshot. When executing a snapshot that is far from the baseline snapshot, the difference between the baseline snapshot and the executed snapshot is also large. Hence, algorithm needs to execute a large number of supersteps to reach convergence, based on the computation result of the baseline result. More importantly, when executing a snapshot based on the computation result of the baseline snapshot, a number of vertices that have reached convergence will turn to active status again. The SSSPs of them need to be recomputed. This not only adds the computation workload of each superstep, but also requires a larger number of superstep to reach convergence.

As shown in Fig. 9, at the point of 1 snapshot, the runtime of the memory locality optimization method is similar to that of baseline. The incremental computation method obtains 2.86×, 3.26×, 3.45×, 3.56× and 3.58× speedup respectively at the points of 20, 40, 60, 80 and 100 snapshots. The experimental results show that the memory locality optimization method can obtain a higher efficiency when executing the task with more snapshots. The reason is that the longer information for the same vertex across different snapshots can be accessed continuously, leading the smaller number of cache misses.

The runtime of V-Grouper is also similar to that of baseline at the point of 1 snapshot. Different from incremental computation method, V-Grouper obtains 15.29×, 26.95×, 26.67×, 26.84× and 26.80× speedup respectively at the points of 20, 40, 60, 80 and 100 snapshots. The reason is as follows. When executing a larger number of snapshots, more *push* operations can be avoided for each vertex group, resulting a higher performance. In these experiments, V-Grouper outperforms the incremental computation method by 5.64×, 11.23×, 12.2×, 13.69× and 14.58× respectively at the points of 20, 40, 60, 80 and 100 snapshots. It also outperforms the memory locality optimization method by 5.34×, 8.05×, 7.77×, 7.53× and 7.50× respectively at the points of 20, 40, 60, 80 and 100 snapshots.

5.6. Memory utilization

We conduct experiments to evaluate the memory utilization of V-Grouper. As mentioned in Section 4.2, we have carefully designed the memory data structures of the graph structure and the intermediate results to improve the memory utilization. Hence, we first implemented the baseline version of V-Grouper without the compression methods of graph structure and intermediate results, and then the version of V-

Table 2
Memory utilization.

# of Snapshots	Baseline	Structure_C	V-Grouper	C_ratio
1	252.22 MB	346.09 MB	330.24 MB	0.76×
9	2329.21 MB	355.28 MB	339.98 MB	6.85×
18	4646.75 MB	346.80 MB	348.76 MB	13.32×
27	6894.57 MB	375.04 MB	357.87 MB	19.26×

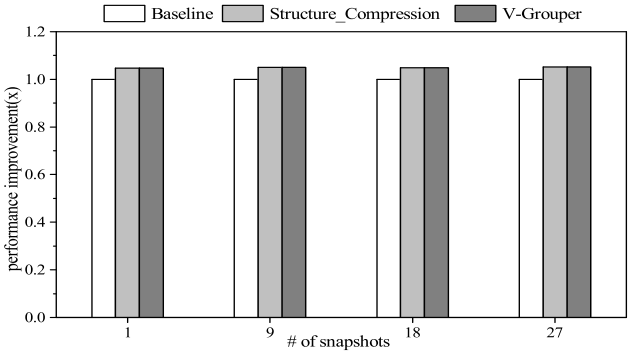


Fig. 10. Performance effect of compression.

Grouper only with graph structure compression. We run the baseline version, the time-evolving graph structure compression and V-Grouper on the time-evolving graph dataset Weibo by using 1, 9, 18, and 27 snapshots respectively. The experimental results are as follows.

As shown in Table 2, at the point of 1 snapshot, the memory consumption of the baseline version is 252.22 MB. The memory consumption of V-Grouper version is 330.24 MB. The compression ratio is 0.76. However, the compression ratio of V-Grouper is 6.85×, 13.32× and 19.26× respectively at the points of 9 snapshots, 18 snapshots and 27 snapshots. The reasons are as follows. At the point of 1 snapshot, the gain of the compression cannot cover the extra overhead of the bitmaps and extra overhead of the counters that are used to encode the time-evolving graph structure and the intermediate results. However, the compression ratio is increasing with the number of the snapshots. The reason is that, for a time-evolving graph with N snapshots, each object (vertex or edge) is stored only once with a bitmap of N bits. By using this design, only one bit is needed to represent a repeated object in a snapshot. Hence, V-Grouper can obtain a higher compression ratio when processing the time-evolving graph with a large number snapshots. Experiments are also repeated on the datasets of Youtube and Flickr. Similar results are observed.

From the experimental results, we observe that the version with graph structure compression is only 1.047×, 1.051×, 1.048× and 1.052× slower than the baseline respectively, at the points of 1 snapshot, 9 snapshots, 18 snapshots, and 27 snapshots, as shown in Fig. 10. For example, at the point of 1 snapshot, the runtimes of baseline and the version with graph structure compression are 1.257 seconds and 1.316 seconds. The difference is small. The reasons are as follows. First, during the execution process of algorithm, both the baseline and time-evolving graph structure compression need to determine if an object belongs to a certain snapshot or not. Second, PLWAH [10] is an efficient compression method for bitmaps, which incurs only minor performance overhead. We also observe that the difference between the version only with graph structure compression and V-Grouper is negligible, as shown in Fig. 10. The reasons are as follows. During the execution process of algorithm, when the algorithm needs to write one vertex value predicted group with m repeated data, the baseline needs m write operations. V-Grouper only needs to write the data only one time with m. Experimental results show that, both methods are very efficient, resulting in the negligible performance difference. Experiments are also repeated on the datasets of Youtube and Flickr. Similar results are observed.

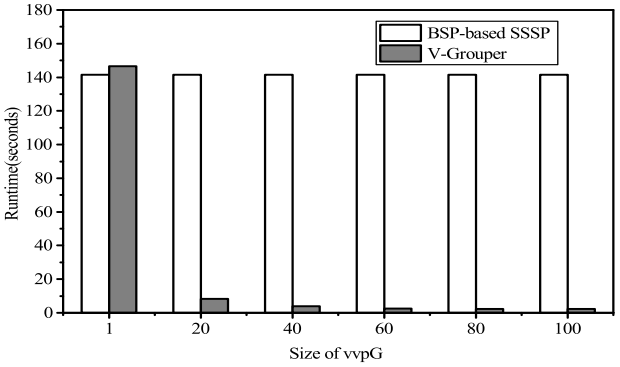


Fig. 11. Effect of snapshot similarity.

5.7. Effect of snapshot similarity

In this section, we study performance effect of the snapshot similarity of time-evolving graph. Real-world time-evolving graphs have a high snapshot similarity. Hence, it is difficult to find a series of datasets of real-world time-evolving graphs with different similarities. However, experiments can be designed according to the size of the vvpG. The reason is that, during the execution process of V-Grouper, a high snapshot similarity will lead to a large average size of vvpG, avoiding more push operations. Hence, we can conduct different experiments on a real-world time-evolving graph. In each experiment, the size of vvpG is set as an upper value artificially. Specifically, experiments are conducted on the time-evolving graph dataset Flickr with 100 snapshots, by using the upper values of vvpG of 1, 20, 40, 60, 80 and 100 respectively. We take the BSP-based SSSP algorithm as baseline.

Experimental results show that the runtime of baseline is 141.5 seconds. The runtime of V-Grouper is 147.5 seconds when the upper value of vvpG is set as 1. In this case, V-Grouper is 1.042× slower than the baseline. The reasons are as follows. First, V-Grouper cannot obtain performance gain when the upper value of vvpG is set as 1. Because each vertex in the vertex group needs to execute *push* operations to its neighbors. The performance degradation stems mainly from the extra overhead of predicting computation results. However, V-Grouper is 17.15×, 35.19×, 54.06×, 64.31× and 64.31× faster than baseline when the upper size is set as 20, 40, 60, 80 and 100 respectively. The reason is that the larger size of vvpG can avoid more *push* operations, resulting in a higher performance improvement. We also observe that the speedup is reduced from the point 60 to the point 80. V-Grouper then reaches its peak performance at the points of 80 and 100. The peak performance depends mainly on the snapshot similarity of time-evolving graph. Experiments are also repeated on the datasets of Youtube and Weibo. Similar results are observed (Fig. 11).

6. Related work

In this section, we briefly discuss the related works that are not mentioned in Section 2.

Incremental Computation Method. Incremental computation method is another method to optimizing graph computation on a series of snapshots [13,16]. In order to improve the execution efficiency of TEG algorithms, incremental computation method first selects and executes the baseline snapshot. Typically, the first snapshot is selected as baseline. Then, each of other snapshots is executed based on the computation result of baseline snapshot. The main idea of this method is that the computation result of the baseline snapshot can be reused by other snapshots. However, the performance of this method is limited because, the granularity of the whole snapshot is too large, leading to a large number of vertices needing to be recomputed when executing each of other snapshots. As mentioned before, this not only adds the computation workload of each superstep, but also requires a larger

number of superstep to reach convergence. Tornado [35] and Tripoline [18]) only support incremental computations for edge additions while more recent systems (such as Kickstarter [38] and GraphBolt [29]) also support edge deletions. CommonGraph [2] is the system to convert deletions into additions for evolving graph analysis and thus reduce the cost of graph mutation as well as incremental computation (via work sharing) significantly. Different from these methods, V-Grouper is a fine-grained algorithm. During the execution process of each superstep, in order to obtain shorter paths of the vertices of each vertex group, V-Grouper predicts the vertices that will obtain the same computation, avoiding the redundant computations of the vertices in the vertex group. Hence, our method can obtain a higher and more stable performance, as shown in Section 5.5.

Improving memory access locality. Han et al. [16] present Chronos to reduce the execution time of TEG algorithms by improving the efficiency of memory accesses. They design the in-memory data structures carefully by putting the information for the same vertex across different snapshots together. The main idea of this method is that the information for the same vertex across different snapshots can be accessed continuously in memory, reducing the number of cache misses. However, the efficiency of this method is limited by an upper limit because, the access mode of small random memory accesses is only 10× slower than that of the sequential memory accesses in ideal condition [25]. Furthermore, the efficiency of this method can be reduced in the NUMA architecture. Modern computers with multi-core CPUs are largely based on the popular NUMA architecture [37,43] that provides DRAM on each of the multiple NUMA nodes and enables enlarged DRAM capacity by connecting these nodes with high-bandwidth links. When a vertex needs to access its neighbors that reside in other NUMA nodes, there is a penalty of high latency that is approximately 10× higher than the case of local memory access [4]. Compared with this method, our V-Grouper overcomes two limitations above by avoiding a large number *push* operations, obtaining a significant performance improvement.

GPU-Based Implementation. Yangzihao Wang, et al. [39] proposed a high-performance graph library, called Gunrock that provides a data-centric abstraction of a set of vertices or edges. They develop a three-step architecture to compute SSSP algorithm on GPU. Federico Busato, et al. [3] proposed an efficient implementation of the Bellman-Ford algorithm using two queues on the Kepler GPU. Although this algorithm exploits dynamic parallelism, it also uses atomic operation that makes part of the code serial due to a feature of modern Kepler GPUs. Akif Rehman Busato, et al. [31] designed a GPU-based accelerator to speed up the execution of SSSP algorithm. Anurag Ingole and Rupesh Nasre [17] implemented a dynamic incremental and decremental SSSP algorithm by developing the feature of GPU. Like most graph algorithms and graph processing systems, V-Grouper has implemented based on CPUs which are more popular. V-Grouper improves the performance by optimizing the algorithm itself.

Agent: Polyer [8] divides a vertex with large number of edges into agent vertices according to the number of NUMA nodes. In each superstep of the execution process, the agent vertices are executed like other vertices that are not divided. At the end of each superstep, one agent vertex aggregates the computation results of other agents, and sends the aggregated computation result to other agents. Powergraph [14] also adopts this method. This method can reduce the number of communications among the NUMA nodes or the compute nodes of the distributed graph systems. However, due to the overheads of aggregating computation results and synchronizing the aggregated computation result, only the vertices with a large number edges can obtain a performance improvement. Different from this method, V-Grouper is designed for time-evolving graphs, which can obtain a stable performance improvement by avoiding the redundant *pull* [41] operations of the same vertex in different snapshots.

Message buffering: Distributed graph processing systems usually suffer from high-frequency and fine-grained communications among the machines, leading to limited performance. In order to address this problem, message buffering is usually employed by distributed graph processing systems, such as Pregel [28] and GPS [34], to reduce the communication overhead, which buffers the messages and then sends the message batches to the remote machines together. Essentially, the interactions among vertices are achieved by sending messages over edges. Different from distributed graph processing systems, V-Grouper is an efficient SSSP algorithm for time-evolving graph, which reduces the number of communications among the NUMA nodes or the computation nodes of distributed graph processing systems by carefully designing the algorithms.

Graph partitioning: Another method to reduce the communication overhead among the NUMA nodes or the computation nodes of distributed graph processing systems is to employ graph partitioning algorithms. For large-scale graph data with complex structure, distributed graph processing systems need to choose a proper graph partitioning algorithm that can significantly reduce the number inter-machine edges, and effectively balance the workload among the machines [1,12]. However, there are two limitations as follows. First, graph partitioning is a NP-hard problem. It is difficult to obtain a good solution that not only significantly reduces the edges across different computation nodes, but also well balances the computation overloads among the computation nodes, resulting in a limited performance gain. V-Grouper overcomes the performance constraint of the methods of graph partitioning by avoiding the redundant computations of the vertices in the vertex group to improve the algorithm efficiency.

Adding computation resources. In recent years, many distributed graph processing systems, such as Pregel [28], have been proposed to speedup the graph computation task by increasing the number of compute nodes. Most of these systems are designed for the static graph (single snapshot) computation task [27,28,36]. However, there are two limitations when graph algorithm engineers implement their graph algorithms based on these systems. First, distributed graph computation is expensive. In order to speed up the time-evolving graph computation task that needs to compute a large number of snapshots, a large number of compute nodes are required. The investments of compute nodes that are either beyond the financial capability of or unprofitable for most small and medium-sized organizations, making the deployment of their large-scale graph-computing jobs difficult if not impossible [7]. Second, distributed graph process systems are of low scalability due to the inefficient communication [6,41]. Compared with the method above, V-Grouper has a higher cost-effectiveness. Since it improves the performance by optimizing the algorithm itself.

Avoiding redundant computations among snapshots. Chenghui Ren, et al. [32] proposed FVF algorithm to compute shortest path for any pair of vertices in each snapshot of the time-evolving graph. For a group of snapshots, FVF first computes the lower limit of the shortest path value and upper limit of the shortest path value among the snapshots. If the lower and upper limit values are equal, the snapshots between two bounded snapshots do not need to be computed. However, this method is not suitable for time-evolving graph SSSP algorithm due to the reasons as follows. First, FVF is employed to compute the shortest path of any pair of vertices, while SSSP algorithm is to compute the shortest paths from the source vertex to other vertices. They are two different algorithms. Second, the prediction granularity is too large. If the number of the snapshots between the two bounded snapshots is too small, the performance gain can likely be negative due to the computation costs of the bounded snapshots. Different from the method above, as mentioned before, V-Grouper is a fine-grained algorithm. During the execution process of each superstep, in order to obtain shorter paths of the vertices of each vertex group, V-Grouper predicts the vertices that will obtain the same computation, avoiding the redundant computations of the vertices in the vertex group. This can obtain a higher performance improvement.

7. Conclusion

In this paper, we have proposed V-Grouper, a highly efficient SSSP algorithm for time-evolving graphs, which can obtain a significant performance improvement by avoiding the redundant computations of vertices across the snapshots that existing BSP-based SSSP algorithm suffers from. Experimental results show that V-Grouper is up to 64.31× faster than BSP-based SSSP algorithm.

V-Grouper has been designed to efficiently compute the SSSPs for each snapshot of time-evolving graph. SSSP algorithm is the building block for computing other important network analysis. The main idea of V-Grouper is to avoid the redundant computations of the same vertex in different snapshots, by predicting computation results. As for future work, we plan to design other important algorithms for time-evolving graphs by using the main idea of v-Grouper, such as PageRank (PR), Community Detection (CD) and Connected Components (CC). We will first research the essential conditions that can be used to predict the computation results according to the key feature of the algorithm, and then design and implement the algorithm based on the essential conditions. SSSP is a fundamental algorithm that can be used in many applications, such as transportation networks and communication. As for future work, we also consider the concrete applications of V-Grouper.

CRediT authorship contribution statement

Yongli Cheng: Methodology, Project administration, Supervision, Writing – original draft, Writing – review & editing. **Chuanjie Huang:** Data curation, Software. **Hong Jiang:** Formal analysis, Writing – review & editing. **Xianghao Xu:** Funding acquisition, Project administration, Supervision, Writing – review & editing. **Fang Wang:** Formal analysis, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgment

This work is supported by Natural Science Foundation of Fujian Province under Grant No. 2020J01493. This work is also supported by the Open Project Program of Wuhan National Laboratory for Optoelectronics NO. 2022WNLOKF017.

References

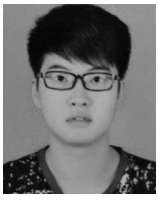
[1] Z. Abbas, V. Kalavri, P. Carbone, V. Vlassov, Streaming graph partitioning: an experimental study, Proc. VLDB Endow. 11 (11) (2018) 1590–1603.
[2] M. Afarin, C. Gao, S. Rahman, N. Abu-Ghazaleh, R. Gupta, Commongraph: graph analytics on evolving data, in: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 2, 2023, pp. 133–145.
[3] F. Busato, N. Bombieri, An efficient implementation of the Bellman-Ford algorithm for Kepler gpu architectures, IEEE Trans. Parallel Distrib. Syst. 27 (8) (2015) 2222–2233.
[4] Y. Cheng, W. Chen, Z. Wang, X. Yu, Performance-monitoring-based traffic-aware virtual machine deployment on numa systems, IEEE Syst. J. 11 (2) (2015) 973–982.
[5] Y. Cheng, H. Jiang, F. Wang, Y. Hua, D. Feng, Blitzg: exploiting high-bandwidth networks for fast graph processing, in: IEEE INFOCOM 2017-IEEE Conference on Computer Communications, IEEE, 2017, pp. 1–9.
[6] Y. Cheng, H. Jiang, F. Wang, Y. Hua, D. Feng, W. Guo, Y. Wu, Using high-bandwidth networks efficiently for fast graph computation, IEEE Trans. Parallel Distrib. Syst. 30 (5) (2018) 1170–1183.
[7] Y. Cheng, F. Wang, H. Jiang, Y. Hua, D. Feng, Y. Wu, T. Zhu, W. Guo, A highly cost-effective task scheduling strategy for very large graph computation, Future Gener. Comput. Syst. 89 (2018) 698–712.
[8] V. Dadu, S. Liu, T. Nowatzki, Polygraph: exposing the value of flexibility for graph processing accelerators, in: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), IEEE, 2021, pp. 595–608.
[9] Datasets for social network analysis, https://www.aminer.cn/data-sna#Weibo-NetTweet. (Accessed 21 November 2023).
[10] F. Deliège, T.B. Pedersen, Position list word aligned hybrid: optimizing space and performance for compressed bitmaps, in: Proceedings of the 13th International Conference on Extending Database Technology, 2010, pp. 228–239.
[11] Dynamic networks, https://networkrepository.com/dynamic.php. (Accessed 21 November 2023).
[12] W. Fan, M. Liu, C. Tian, R. Xu, J. Zhou, Incrementalization of graph partitioning algorithms, Proc. VLDB Endow. 13 (8) (2020) 1261–1274.
[13] S. Gandhi, Y. Simmhan, An interval-centric model for distributed computing over temporal graphs, in: 2020 IEEE 36th International Conference on Data Engineering (ICDE), IEEE, 2020, pp. 1129–1140.
[14] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, Powergraph: distributed graph-parallel computation on natural graphs, in: 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), 2012, pp. 17–30.
[15] L. Han, Z. Shen, D. Liu, Z. Shao, H.H. Huang, T. Li, A novel reram-based processing-in-memory architecture for graph traversal, ACM Trans. Storage 14 (1) (2018) 1–26.
[16] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, E. Chen, Chronos: a graph engine for temporal graph analysis, in: Proceedings of the Ninth European Conference on Computer Systems, 2014, pp. 1–14.
[17] A. Ingole, R. Nasre, Dynamic shortest paths using javascript on gpus, in: IEEE 22nd International Conference on High-Performance Computing (HiPC), 2015, pp. 1–5.
[18] X. Jiang, C. Xu, X. Yin, Z. Zhao, R. Gupta, Tripoline: generalized incremental graph processing via graph triangle inequality, in: Proceedings of the Sixteenth European Conference on Computer Systems, 2021, pp. 17–32.
[19] H. Jung, J.-G. Lee, N. Lee, S.-H. Kim, Comparison of fitness and popularity: fitness-popularity dynamic network model, J. Stat. Mech. Theory Exp. 2018 (12) (2018) 123403.
[20] R. Karimi, D.M. Koppelman, C.J. Michael, Gpu road network graph contraction and sssp query, in: Proceedings of the ACM International Conference on Supercomputing, 2019, pp. 250–260.
[21] A. Khanda, S. Srinivasan, S. Bhowmick, B. Norris, S.K. Das, A parallel algorithm template for updating single-source shortest paths in large-scale dynamic networks, IEEE Trans. Parallel Distrib. Syst. 33 (4) (2021) 929–940.
[22] D. Krizanc, A. Saarimaki, Bulk synchronous parallel: practical experience with a model for parallel computing, Parallel Comput. 25 (2) (1999) 159–181.
[23] P. Kumar, H.H. Huang, Graphone: a data store for real-time analytics on evolving graphs, ACM Trans. Storage 15 (4) (2020) 1–40.
[24] A. Kyrola, G. Blelloch, C. Guestrin, {GraphChi}:{Large-Scale} graph computation on just a {PC}, in: 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), 2012, pp. 31–46.
[25] V. Leis, P. Boncz, A. Kemper, T. Neumann, Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age, in: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, 2014, pp. 743–754.
[26] K. Lerman, R. Ghosh, J.H. Kang, Centrality metric for dynamic networks, in: Proceedings of the Eighth Workshop on Mining and Learning with Graphs, 2010, pp. 70–77.
[27] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, J. Hellerstein, Distributed graphlab: a framework for machine learning and data mining in the cloud, Proc. VLDB Endow. 5 (8) (2012) 716–727.
[28] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, 2010, pp. 135–146.
[29] M. Mariappan, K. Vora, Graphbolt: dependency-driven synchronous processing of streaming graphs, in: Proceedings of the Fourteenth EuroSys Conference 2019, 2019, pp. 1–16.
[30] L. Page, S. Brin, R. Motwani, T. Winograd, The pagerank citation ranking: Bringing order to the web, Tech. rep., Stanford InfoLab, 1999.
[31] A. Rehman, M. Ahmad, O. Khan, Exploring accelerator and parallel graph algorithmic choices for temporal graphs, in: Proceedings of the Eleventh International Workshop on Programming Models and Applications for Multicores and Manycores, 2020, pp. 1–10.
[32] C. Ren, E. Lo, B. Kao, X. Zhu, R. Cheng, On querying historical evolving graph sequences, Proc. VLDB Endow. 4 (11) (2011) 726–737.
[33] R. Rossi, N. Ahmed, The network data repository with interactive graph analytics and visualization, in: Twenty-Ninth AAAI Conference on Artificial Intelligence, 2015.
[34] S. Salihoglu, J. Widom, Gps: a graph processing system, in: Proceedings of the 25th International Conference on Scientific and Statistical Database Management, 2013, pp. 1–12.

[35] X. Shi, B. Cui, Y. Shao, Y. Tong, Tornado: a system for real-time iterative analysis over evolving data, in: Proceedings of the 2016 International Conference on Management of Data, 2016, pp. 417–430.
[36] J. Shun, G.E. Blelloch, Ligra: a lightweight graph processing framework for shared memory, in: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2013, pp. 135–146.
[37] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt, E. Tune, Optimizing Google's warehouse scale computers: the numa experience, in: 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2013, pp. 188–197.
[38] K. Vora, R. Gupta, G. Xu, Kickstarter: fast and accurate computations on streaming graphs via trimmed approximations, in: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, 2017, pp. 237–251.
[39] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, J.D. Owens, Gunrock: a high-performance graph processing library on the gpu, in: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2016, pp. 1–12.
[40] Y. Wang, D. Jiang, J. Xiong, Numa-aware thread migration for high performance nvmm file systems, in: 36th International Conference on Massive Storage Systems and Technology (MSST 2020), 2020.
[41] X. Xu, F. Wang, H. Jiang, Y. Cheng, D. Feng, Y. Zhang, A hybrid update strategy for i/o-efficient out-of-core graph processing, IEEE Trans. Parallel Distrib. Syst. 31 (8) (2020) 1767–1782.
[42] L. Yang, L. Qi, Y.-P. Zhao, B. Gao, T.-Y. Liu, Link analysis using time series of web graphs, in: Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, 2007, pp. 1011–1014.
[43] K. Zhang, R. Chen, H. Chen, Numa-aware graph-structured analytics, in: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2015, pp. 183–193.
[44] Y. Zhang, X. Liao, H. Jin, L. He, B. He, H. Liu, L. Gu, Depgraph: a dependency-driven accelerator for efficient iterative graph processing, in: 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), IEEE, 2021, pp. 371–384.

Yongli Cheng received the BE degree from the Chang'an University, Xi'an, China, in 1998, the MS degree from the FuZhou University, FuZhou, China, in 2010, and the PhD degree from the Huazhong University of Science and Technology, Wuhan, China, 2017. He is currently a teacher of the College of Computer and Data Science, FuZhou University currently. His current research interests include computer architecture and graph computing. He has several publications in major international conferences and journals, including HPDC, IWQoS, INFOCOM, ICPP, the Future Generation Computing Systems, IEEE/ACM Transactions on Networking, and Frontiers of Computer Science.

Chuanjie Huang received the BE degree from the Chengyi College of Jimei University, XiaMen, China, in 2020. He is currently working toward the MS degree majoring in computer technology at FuZhou University, FuZhou, China. His current research interest is mainly to accelerate the execution efficiency of time-evolving graph algorithms.

Hong Jiang received the BE degree from the Huazhong University of Science and Technology, Wuhan, China, in 1982, the MASc degree from the University of Toronto, Toronto, Canada, in 1987, and the PhD degree from the Texas A&M University, College Station, Texas, in 1991. He is Wendell H. Nedderman endowed professor and chair of the Department of Computer Science and Engineering, University of Texas at Arlington. His research interests include computer architecture, computer storage systems, and parallel/distributed computing. He serves as an associate editor of the IEEE Transactions on Parallel and Distributed Systems. He has more than 200 publications in major journals and international Conferences in these areas, including the IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Computers, ACM Transactions on Storage, ACM Transactions on Architecture and Code Optimization, Journal of Parallel and Distributed Computing, ISCA, MICRO, FAST, USENIX ATC, USENIX LISA, SIGMETRICS, MIDDLEWARE, ICDCS, IPDPS, OOPLAS, ECOOP, SC, ICS, HPDC, and ICPP.



Xianghao Xu received the PhD degree from Huazhong University of Science and Technology, Wuhan, China, 2021. He is currently an assistant professor in School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing, China. His current research interests include graph processing, computer architecture and big data analytics. He has several publications in major international conferences and journals, including IEEE-TPDS, IEEE-TBD, ICPP, IWQoS and FCS.

Fang Wang received the BE and master's degrees in computer science from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 1994 and 1997, respectively, and the PhD degree in computer architecture from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2001, where she is currently a professor of computer science and engineering. Her research interests include distribute file systems, parallel I/O storage systems, and graph processing systems. She has more than 50 publications in major journals and conferences, including the Future Generation Computing Systems, the ACM Transactions on Architecture and Code Optimization, HiPC, ICDCS, HPDC, and ICPP.