2024 IEEE 30th International Conference on Parallel and Distributed Systems (ICPADS) | 979-8-3315-1596-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICPADS63350.2024.00039

GraphFlow: A Fast and Accurate Distributed Streaming Graph Computation Model

Zheheng Liang[1,2], Yingying Zheng[3,4*], Sheng Bi[5], Chaosheng Yao[1,2], Jiayan Wang[5], Lijie Xu[3,4], Shuping Ji[6], Wei Wang[3,4,7,8], Shikai Duan[9]

[1]*Information Center Guangdong Power Grid Limited Liability Company, Guangzhou, China*
[2]*Joint Laboratory on Cyberspace Security, China Southern Power Grid, Guangzhou, China*
[3]*State Key Lab of Computer Science at ISCAS*, [4]*University of CAS, Beijing, China*
[5]*Guangzhou Power Supply Bureau, Guangzhou, China*
[6]*University of Toronto, Toronto, Canada*
[7]*Nanjing Institute of Software Technology*, [8]*University of CAS, Nanjing, China*
[9]*Ant Group-Payment Business Group, Hangzhou, China*
liangzheheng@qq.com, zhengyingying14@otcaix.iscas.ac.cn, 444681280@qq.com, 328965667@qq.com
peacockwang@163.com, {xulijie,wangwei}@otcaix.iscas.ac.cn, shuping@msrg.utoronto.ca, shikai.dsk@alibaba-inc.com

***Abstract*—Streaming graph computation has been widely applied in many fields, e.g., social network analysis and online product recommendation. However, existing streaming graph computation approaches still present limitations on accuracy and efficiency. To improve the accuracy, some distributed systems use the sequential graph update method based on an incremental computation model. However, these systems cannot handle the dynamic graph update concurrently. The speculation-based parallel updating model can parallelize the graph computation, however, it is restricted due to ignoring the original messages when updating a graph. Streaming graph computation usually requires high accuracy and low latency. As such, it is challenging to utilize incremental computation while simultaneously supplying concurrent processing guarantees.**

To overcome these challenges, in this paper, we first analyze a number of classical graph algorithms and summarize three principles that graph algorithms should satisfy in streaming scenarios. Based on these principles, we propose GraphFlow, a streaming graph computation model. GraphFlow achieves fast and accurate computation by utilizing incremental state update and propagation. To reduce the impact of concurrent update conflicts, GraphFlow provides a fine-grained lock based parallel update strategy. We implement GraphFlow framework and evaluate its performance and concurrent update conflict probability on real-world datasets. Meanwhile, we compare GraphFlow with two existing representative graph processing systems. Experimental results show GraphFlow achieves low latency and outperforms other graph processing systems given large datasets.

***Index Terms*—streaming graph computation, distributed computation**

I. Introduction

Nowadays, the amount of graph data has been increasing dramatically. For example, the number of vertices of the Web link graph reaches the terabyte level, and the number of edges reaches the petabyte level [1]. The quickly increasing graph data requires new efficient processing approaches. Existing graph computation systems, such as Pregel [2] and GraphX [3], are built on the Bulk Synchronized Parallel (BSP) model [4]. In this model, all updates on each iteration are computed in one SuperStep. Meanwhile, these systems only process static graph data in batches and need to re-compute the whole graph when some edges or vertices are updated. Thus, they cannot process real-time streaming graph data [5].

Streaming graph computation approaches can be divided into two categories, i.e., estimated computation [6]–[11] and accurate computation [5], [12], [13]. The estimated computation approaches, e.g., sampling and summarization, suffer from low accuracy in production environments. Chu et al. [13] indicate that for large-scale graphs that cannot be loaded into memory, the error rate of estimated methods is as high as 95%. KineoGraph [12] and IncGraph [5] use the sequential state update based incremental computation models to achieve high accuracy. However, the sequential incremental computation operations limit the concurrency of these systems. SpecGraph [13] improves these sequential incremental models and proposes a speculation-based parallel updating model. But this model assumes that the state of a vertex depends only on messages received from its neighbors, and the message that needs to be sent out is only related to the state of this vertex. This assumption largely simplifies the implementation of this concurrent system, however, the applicability to more graph algorithms of this system is deteriorated. GraphBolt [14] combines BSP semantics with an incremental computation model to process dynamic graph updates, but its synchronous processing semantic requires to wait for all processes to be completed in one SuperStep, which limits its performance. Therefore, we lack an effective approach to achieve high accuracy and low latency in streaming graph computation.

In this paper, we first analyze the features of a number of typical graph algorithms for streaming graph computation. Based on the feature analysis, we conclude three principles that streaming graph algorithms should satisfy: (1) incremental computation; (2) sequential consistency; and (3) commutative

*Yingying Zheng is the corresponding author. CAS is the abbreviation of Chinese Academy of Sciences. ISCAS is the abbreviation of Institute of Software Chinese Academy of Sciences.

979-8-3315-1596-6/24/$31.00 ©2024 IEEE
DOI 10.1109/ICPADS63350.2024.00039

law and associative law for vertex operators. Based on these three principles, we propose GraphFlow, a distributed streaming graph computation model. GraphFlow achieves efficient computation by utilizing incremental state update and propagation. Simultaneously, the state update strategies we designed for different state types prevent concurrent update conflicts on parallel graph processing and guarantee the accuracy of GraphFlow. GraphFlow can concurrently process incremental messages and compute a new state of a graph based on both the incoming messages and the original state of that graph.

To better characterize the streaming graph computation on GraphFlow model, we first propose three basic concepts: $event$, $state$, and $transform$. We abstract the requests to update a graph as an event stream, and the refined data of a graph at a specific time as a state, respectively. An event can transform a graph from a state to a new state. There are two types of states: independent state and associated state. We propose different state update strategies for independent state and associated state, respectively. We further propose a fine-grained lock based parallel update strategy to prevent concurrent updating conflicts on parallel graph processing and guarantee the accuracy of GraphFlow.

To evaluate the efficiency of GraphFlow, we implement four typical streaming graph computation algorithms, i.e., Degree Distribution, Triangle Count, Single Source Shortest Path, and PageRank, based on GraphFlow model. We evaluate the performance and concurrent update conflict probability of these four algorithms. Experiments on real-world datasets show the efficiency of GraphFlow: the average response times are within 2ms on a big dataset with 70M edges and the concurrent update conflict probability is only less than 3%. Compared with other graph systems, GraphFlow also presents better processing performance.

The main contributions of this paper are as follows:

- We analyze the computation features of typical graph algorithms. Based on the feature analysis, we propose three principles that streaming graph algorithms should satisfy.
- We propose a distributed streaming graph computation model GraphFlow. GraphFlow achieves fast processing speed through concurrent and incremental state update, and guarantees high accuracy simultaneously.
- We implement a streaming graph computation system based on GraphFlow model, and implement four graph algorithms on this system. Experiments on real-world datasets show that GraphFlow can achieve fast and accurate streaming graph computation.

II. Background

This section briefly introduces streaming graph computation and incremental graph computation.

A. Streaming Graph Computation

The streaming graph processing systems, such as GraphIn [15], KineoGraph [12], and IncGraph [5], process dynamic graph data that includes vertices, edges, the values of vertices, and the weights of edges. Dynamic graph data continuously flows into a graph processing system instead of being stored in files or databases. The graph structures may rapidly change when processing a flow of graph updates. Streaming graph data can be represented as a stream of events, i.e., $(A, (V_{source}, V_{sink}))$, in which A represents the action of addition or deletion, while V_{source} and V_{sink} stand for the source vertex and sink vertex of an edge, respectively. For example, $(+, (a, b))$ represents the event to insert an edge between the source vertex a and the sink vertex b.

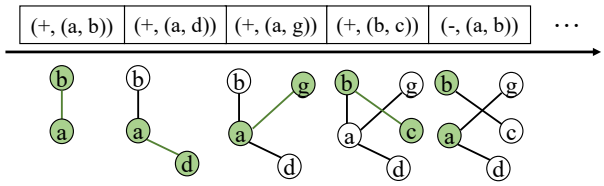


Fig. 1. An example of streaming graph computing

As shown in Fig. 1, events of streaming graph data flow into a graph processing system in a certain order. The signal '$+$' stands for inserting an edge, and the signal '$-$' stands for deleting an edge. The graph structures keep changing, as the continuous streaming graph data flows in. Compared to traditional static graph algorithms, the design of streaming graph algorithms is different and more difficult.

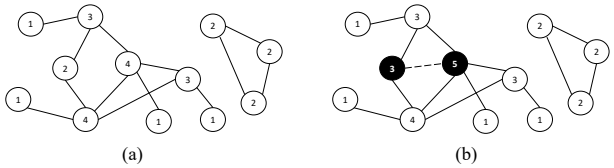
B. Incremental graph computation

Incremental computation means that for any incoming data, we only need to compute the changed data, without recalculating the whole data [14]. Therefore, incremental computation is very fast. Streaming graph computation conforms with the idea of incremental computation. As mentioned in Section II-A, streaming graph data acts as the incremental messages to update a graph. As such, the streaming graph computation inherits the advantages of incremental graph computation, e.g., fast processing capacity. However, the challenge on accuracy of incremental graph computation also exists for streaming graph computation.

KineoGraph [12] and IncGraph [5] use incremental graph computation models to achieve high accuracy, but the serial update method limits their performance. SpecGraph [13] utilizes a concurrent update method to support incremental graph computation. However, this solution assumes that the state of each vertex only depends on the current messages received from that vertex's neighbors and ignores the original state of that vertex. As a result, this assumption deteriorates the expressiveness of SpecGraph. GraphBolt [14] combines the BSP model with the incremental computation, however, it is limited by its synchronous processing semantics. As such, without better efficiency, the existing incremental graph computation solutions cannot support streaming graph computation well.

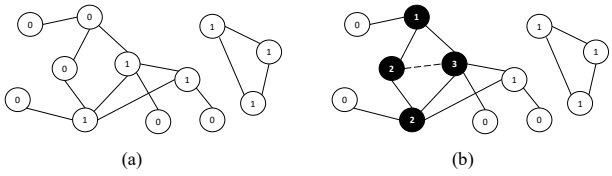
III. Feature Analysis of Graph Algorithms

To learn more about graph processing algorithms, in this section, we analyze the features of several classical graph



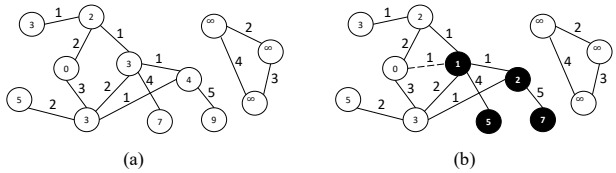
(a) (b)

Fig. 2. Degree distribution



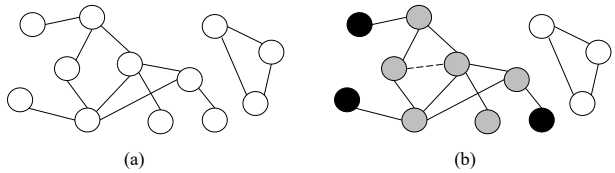
(a) (b)

Fig. 3. Triangle count



(a) (b)

Fig. 4. Single source shortest path



(a) (b)

Fig. 5. PageRank

processing algorithms, and summarize three principles that the graph processing algorithms should satisfy to achieve high performance.

A. Feature Analysis

To save space, we illustrate the features of graph processing algorithms only with four typical graph algorithms, i.e., Degree Distribution, Triangle Counting, Single Source Shortest Path, and PageRank.

(1) Degree Distribution (DD). DD algorithm is used to count the degree of each vertex in undirected graphs. Fig. 2 illustrates how to count degrees in stream processing scenarios. The circle represents a vertex in the graph, the line between two vertices represents an edge, and the value within a circle stands for the current degree of the corresponding vertex. We assume that the current graph is shown in Fig. 2 (a). In Fig. 2 (b), when the edge between the two black vertices is added, the degrees of source vertex and target vertex of this edge all increase by one. Thus, in DD algorithm, the action to add an edge e will only influence the source vertex and target vertex of edge e.

(2) Triangle Count (TC). TC algorithm is used to count the number of different triangles in undirected graphs. This algorithm has been widely used in community discovery and link prediction [16]. Fig. 3 shows how to count the number of triangles in stream processing scenarios. The value of a vertex represents the number of triangles owned by that vertex. Fig. 3 (a) shows the current state of a graph. When we add an edge e to this graph as shown in Fig. 3 (b), the number of triangles of the two vertices of e both increase by N. N is the number of common adjacent vertices of the two vertices of e. The number of triangles of all the N common adjacent vertices increase by one. As such, for TC algorithm, the action to add an edge influences not only two vertices of that edge, but also the common adjacent vertices of these two vertices.

(3) Single Source Shortest Path (SSSP). SSSP algorithm computes the shortest path from a source vertex to other vertices. It has been widely used in scenarios such as GPS navigation and social network [17]. As shown in Fig. 4, the value of an edge is the distance between these two vertices, the value of the source vertex is set to be 0, and the value of another vertex represents the shortest distance from the source vertex to this vertex. Fig. 4 (a) shows the current state of a graph for SSSP. If an edge e with weight 1 is added, we will get the new state of the graph, as shown in Fig. 4 (b). The newly added edge e is identified by the dotted line. This example tells us the effect of a newly added edge can be spread along certain paths (i.e., 0->1->2->7 and 0->1->5 in Fig. 4 (b)) for SSSP algorithm.

(4) PageRank (PR). PR algorithm [18] is used to calculate the importance of each Web page based on Web link inputs. If page A references to page B, then page A will contribute a certain score to page B. The score, a.k.a., the value of PR, presents the importance of a Web page. Web pages are cross-referenced. For PR algorithm, after several iterations, the score of each page will become stable. Suppose an initial graph running PR algorithm is shown in Fig. 5 (a). When a new edge identified by dotted line is added into this graph, the gray vertices are influenced firstly, and then the effects continue to be propagated to the black vertices as shown in Fig. 5 (b). After several iterations, the PR values of all vertices are stabilizing. Therefore, when a new edge is added into a graph running PR algorithm, all vertices in the connected subgraph where the inserted edge is located will be affected.

Feature Summary. The above section provides a detailed analysis of four classical and widely used graph algorithms. Based on the analysis, we summarize the graph computation features on four different dimensions: influence scope, update function, computation times, and update type. Influence scope refers to the vertices that can be influenced when an edge is changed. Update function refers to the main algebraic operation used to update the graph. Computation times are defined as the updating times of a vertex for each graph change. Update type indicates the update of a specific vertex

TABLE I
FEATURE ANALYSIS OF TYPICAL GRAPH ALGORITHMS

Algorithm	Influence Scope	Update Function	Computation Times	Update Type
DD	Source and target vertex	Add operation	Only once	Independent
TC	Source, target and the common adjacent vertices	Add operation	Only once	Associated
SSSP	Spread along certain paths	Minimize operation	Multiple times	Associated
PR	All vertices in the connected subgraph	Aggregate operation	Multiple times	Associated

is independent or associated with other vertices.

Table I concludes the features of DD, TC, SSSP, and PR algorithms in these four dimensions. Take the DD algorithm as an example. When a new edge is added into a graph, only the degrees of the source vertex and target vertex of that edge need to be increased by one, and this calculation only needs to be executed once. The value update of a vertex does not affect other vertices. Diving deep in the feature analysis, we conclude the following three principles to guide the design of streaming graph computation model for these algorithms.

- **Principle1:** The computing method needs to satisfy incremental computation.
- **Principle2:** Operational functions satisfy the commutative law and associative law. The computing sequence cannot affect computing results.
- **Principle3:** Computational sequence satisfies sequential consistency. The updating sequence of vertices cannot affect computing results.

We analyzed a number of graph algorithms, not only including DD, TC, SSSP, and PR, but also including other advanced graph algorithms. We find that some community detection algorithms, such as GN [19], Label Propagation Algorithm (LPA) [19], and FastUnfolding [20] also satisfy these three principles. So, we propose the GraphFlow model to support these algorithms. For any other algorithm, if Principle1 is not satisfied, that algorithm cannot be supported by GraphFlow or any other existing streaming graph processing model. If Principle2 or Principle3 is not satisfied, GraphFlow needs to expose inner interfaces for users to determine the update order to solve potentially introduced update conflicts. The solutions to support all types of graph algorithms using incremental parallel computing is out of the scope of this paper.

IV. GRAPHFLOW MODEL

In this section, we propose a flexible incremental streaming graph computation model GraphFlow, which supports the parallel processing for different graph algorithms. The overview of GraphFlow is shown in Fig. 6. As can be seen in this figure, GraphFlow maintains an event stream and a series of states. Conceptually, the major function of GraphFlow is to continuously transform the last state of a graph to a new state based on the event stream. In GraphFlow, we propose three basic concepts: `State`, `Event`, and `Transform`. `State` is used to represent the snapshot of a graph at a specific time. `Event` represents an update request to a graph. An `Event` can `Transform` a graph from one `State` to another `State`. The formal definitions of these three concepts are as follows.

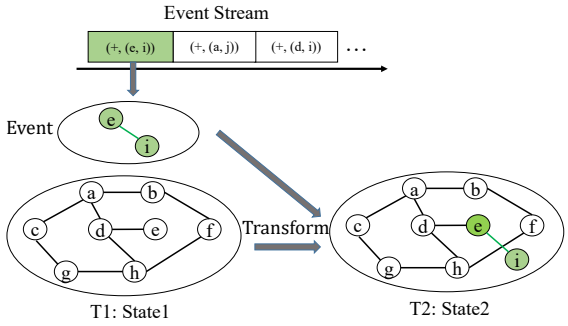


Fig. 6. Overview of the GraphFlow model

Definition 1 (State). A `State` is the screenshot of a graph at a specific time. It stores the refined data of a graph, other than the raw data of the whole graph. A `State` consists of several `Factors`, such as the degree of a vertex or any other user-defined messages. User-defined `State` is permitted in GraphFlow to achieve flexibility in supporting different graph computing algorithms.

Here is an example of a user-defined `State`: when we only want to count the number of edges in a graph, we can design `State` as a single counter, which represents the number of edges in the current graph. When an edge is added into or deleted from a graph, the counter will be simply increased or decreased by one. In this way, we only need to store the single counter state information so as to largely save the graph storage and computation cost.

Definition 2 (Event). An `Event` represents a request message to transform the $state_1$ of a graph G at time T_1 to a new $state_2$ of G at time T_2.

An `Event` message consists of an event type and an event value. Take the event "add an edge $e(v_1, v_2)$" as an example. Its event type is "add" and its event value is $e(v_1, v_2)$. The event value can be expressed in the form ($number_{vertex}$, $value_{vertex}$), or in the form ($source_{edge}$, $sink_{edge}$, $value_{edge}$). There are three event types, i.e., ADD, DELETE and UPDATE. As such, we have the following six types of event messages to update a graph: ADD vertex, DELETE vertex, UPDATE vertex, ADD edge, DELETE edge, and UPDATE edge. These six types of event messages will cover all kinds of graph changes.

Definition 3 (Transform). `Transform` is the action to update a graph, which can be triggered by an `Event`, and acts on a specific `State` of a graph.

TABLE II
STATE INTERFACES

Method Name	Description
Value GET_STATE(key)	Get value of the specified factor
SET_STATE(key, value)	Set value of the specified factor
Map GET_STATES()	Get all state from a graph
SPREAD(neighbor, state)	Propagate state to neighbors

TABLE III
EVENT INTERFACES

Method Name	Description
Edge GET_VALUE(event)	Get the value of event
Type GET_TYPE(event)	Get the type of event

In Fig. 6, suppose the state of the graph is $state_1$ at time T_1. When the graph receives an event to add edge e at time T_2, this event will trigger the `Transform` function to change the state of that graph to $state_2$. The transform function defines in detail how to change a graph from a state to another state according to an arriving event. Usually, a transform function has two types of operations: (1) update a specific edge or vertex; (2) propagate the updates to the neighboring vertices or even the whole graph.

Besides the proposal of the concepts of `State`, `Event`, and `Transform`, we also design their corresponding application programming interfaces (APIs). These interfaces are presented in Table II, Table III, and Table IV, respectively. Based on these interfaces, GraphFlow supports distributed streaming graph computation. Note that a key problem hidden behind is how to update and propagate state correctly, incrementally, and efficiently. In the next section, we will present our solution: use different state update strategies for different types of states.

V. STATE UPDATE STRATEGY

In this section, we first provide a state classification. Then we propose different state update strategies to supply accurate concurrent computation for different state types.

A. State Classification

Inspired by the influence scope and update type analysis of different streaming graph algorithms presented in Table 1, we classify state into two types: **independent state** and **associated state**. Specifically, (1) independent state refers to a state in which the change of a factor does not affect other factors. For example, the factor is defined as the degree of a vertex for the DD algorithm. When an edge is added into or deleted from a graph, except the source and target vertex of this edge, the degree of any other vertex does not need to be changed. (2) Associated state refers to a state constituted by interdependent factors, i.e., the changes of a factor will influence others. For instance, the factor is defined as the shortest distance between a vertex and the source vertex for the SSSP algorithm. When an edge is added into a graph, this event will not only influence the factors of the two vertices of this edge, but also the factors of their neighbors, and even all vertices of the connected subgraph.

TABLE IV
TRANSFORM INTERFACES

Method Name	Description
TRANSFORM(state, event)	Transform state by an event

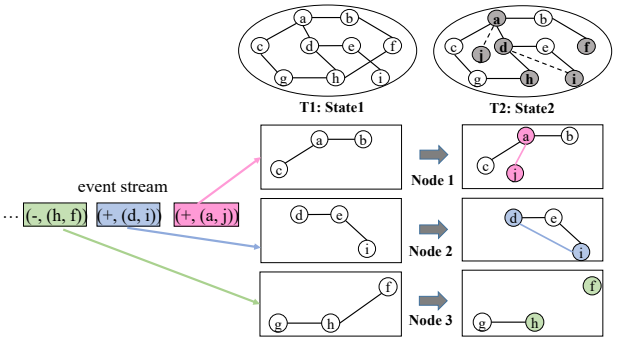


Fig. 7. Parallel processing of independent state updates

B. Update Strategy for Independent State

GraphFlow naturally supports the concurrent update of independent states in multiple computing nodes. One important reason is that the factors of independent state are not related to each other, which means the computing result of each node would not be affected by other nodes. As such, GraphFlow can compute new independent states in real time based on incremental computation. Meanwhile, GraphFlow stores independent states at distributed nodes to accelerate concurrent updating.

Fig. 7 demonstrates the parallel update processing of independent states. When several events arrive at the system, these events will be distributed to different nodes (such as the node 1, 2, 3 in Fig. 7) according to the hash-based graph partitioning algorithm. After that, different computing nodes can update their own states independently. When the $state_1^i$ of the computing node i at time T_1 is transformed to $state_2^i$ at time T_2, the transformation is not influenced by any computing node j where $i \neq j$.

C. Update Strategy for Associated State

Different from the independent state, a factor in the associated state could be relevant to many other factors, which means the changes of a factor may affect other factors. Therefore, the update of the associated state is more complex and challenging than the independent state. Parallel state updates on a graph may suffer from the problem of state update conflict when multiple events related to the same factor are processed in different nodes simultaneously. The easiest way to prevent concurrent update conflicts is serializing the execution of all the events. In this case, when an event e is being processed, other events need to wait until the finish of e. IncGraph [5]

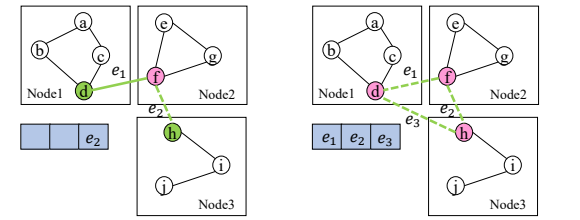
Algorithm 1: Fine-grained Lock Based Parallel Update

```
1  Function FGLUpdate(event, counter) do
2      edge ← GET_VALUE(event);
3      vertex ← GET_SOURCE(edge);
4      if isLocked(vertex) is true then
5          INCREMENT(counter) ;
6          WAIT(vertex) ;
7      end
8      LOCK(vertex);
9      value ← COMPUTE(GET_STATE(vertex));
10     SET_STATE(vertex, value) ;
11     UNLOCK(vertex) ;
12     foreach
         neighbor ∈ GET_NEIGHBORS(vertex) do
13         if isLocked(neighbor) is false then
14             SPREAD(neighbor, value) ;
15         end
16     end
17 end
```

adopts this naïve state update strategy, but the efficiency of IncGraph is highly limited due to the sequential state updates.

To overcome this limitation, we propose the *fine-grained lock based parallel update strategy* (FGL), which is inspired by the row lock mechanism in relational databases. FGL uses the fine-grained lock (i.e., vertex lock), in which only a single factor needs to be locked at a time. Therefore, the updates to the same vertex are atomically and consistently processed. Algorithm 1 presents the basic execution process of FGL. For different graph algorithms, the detailed implementation could be slightly different. The inputs of this algorithm are (1) a given event, and (2) a counter that is used to record the number of concurrent state update conflicts. By calling $GET\_VALUE(event)$, the edge of this event can be obtained (line 2). The source vertex of this edge can be got with $GET\_SOURCE(edge)$ (line 3). If the vertex is used by other events, the counter will be increased by one, and the execution of event needs to wait for the release of the lock on this vertex (lines 4-7). Otherwise, the current computing node gets the lock, calculate the state obtained from the function $GET\_STATE(vertex)$ to an updated value, updates the value of vertex with the function $SET\_STATE(vertex, value)$, and then releases the lock (lines 8-11). Finally, if some neighbors obtained from the function $GET\_NEIGHBORS(vertex)$ are affected by event, the updated value will be propagated to them (lines 12-16).

Fig. 8 shows an example of FGL. In this example, GraphFlow is running on a cluster with three parallel computing nodes: Node1, Node2, and Node3. Node1 is processing a subgraph with four vertices: a, b, c, and d. Node2 is processing a subgraph with three vertices: e, f, and g. Node3 is processing another subgraph with three vertices: h, i, and j. Suppose an event e_1 that updates the states of d and f is processed simultaneously with another event e_2 which will update the



(a) e_1 gets the lock of f; and e_2 waits for the releasing of this lock.

(b) e_1 gets the lock of d; e_2 gets the lock of f, and e_3 gets the lock of h, resulting in a cycle.

Fig. 8. The FGL update strategy example

states of f and h, as shown in Fig. 8 (a). In this case, an update conflict may happen in vertex f. This update conflict will not result into error since our FGL strategy can rightly handle it. If e_1 is processed before e_2, e_1 will get the lock in vertex a and f. e_2 needs to wait until the finishing of e_1. After e_1 releases the lock at f, e_2 will get the locks and further update the states of the vertex f and h.

Fig. 8 (b) shows a rare situation when the processing of the three events e_1, e_2, and e_3 results into a deadlock. FGL solves this problem by setting a lease time for each lock. In this way, these locks will be released when the expiration time arrives. Through this example, it can be seen the possible update conflict is successfully prevented by using FGL. This FGL based concurrency conflict avoidance and deadlock release mechanism introduces extra overhead, however, our experiments in Section VII-C show that for typical graph algorithms including the complex PR, the probability of concurrent update conflicts is less than 3%, and the probability of deadlock is lower.

FGL is an incremental update strategy. For each vertex, the update of each factor of a state is based on the last value of this factor. It means that the updates of different factors can also be executed concurrently by FGL. Only when multiple updates are related to the same factor, those updates need to be computed sequentially. Meanwhile, based on the Principle3, the order of multiple updates will not affect the final state of a node. This feature indicates that our FGL-based update strategy provides accuracy guarantee for associated state.

VI. Implementation

We implement a real system based on GraphFlow model, consisting of three layers: input layer, computation layer, and storage layer. These three layers cooperate together to achieve efficient streaming graph computation.

Input Layer. The input layer receives incoming update events. These events are hash-partitioned into each computing node and form subgraphs for each computing node. In order to achieve fast, flexible and robust graph update events processing, the input layer of GraphFlow adopts distributed cache mechanism.

Computation layer. The computation layer is performed as a computing engine that processes graph update events from the input layer for a certain graph algorithm. For example,

TABLE V
REAL-WORLD GRAPHS USED IN EVALUATIONS

Data Set	Edge	Vertices
LiveJournal (LJ)	70M	5M
Orkut (ORK)	117M	3M

when an edge update event is received, the computing engine first schedules jobs to update the source and target vertex of this edge, and then schedules jobs to update their neighboring vertices until all the relevant vertices are updated. Computation layer consists of a set of computing nodes, which can process different graph update events concurrently to achieve scalability. Different computing nodes run independently and in parallel to achieve high event processing speed. Each computing node keeps a subgraph of the whole graph, and incrementally transforms the current state of its subgraph to a new state based on our state update strategy. The communication overhead between different computing nodes is affected by the propagation range of the update messages. The subgraph states and update messages are synchronously or asynchronously stored into the storage layer for fault tolerance. Another advantage of our system is it provides simple interfaces at the computation layer to support most graph algorithms. Specifically, for any algorithms that satisfy the Principle1, Principle2, and Principle3, users can implement those algorithms directly on our GraphFlow framework according to the simple API.

Storage layer. Subgraphs and distributed caches are persisted into this layer. Subgraphs are stored using a key-value format, in which the key is a vertex, and the value is a set of target vertices in the outgoing edges of that vertex. Similar to the computation layer, this layer also contains a set of storage nodes. To achieve robustness, each storage node stores not only its own data, but also some other storage nodes' backup data. Meanwhile, the storage layer provides interfaces to persist the current state of a subgraph synchronously or asynchronously. Instead of providing the whole implementation of the storage layer from the scratch, we choose the mature open-source software Hazelcast [21], which is a distributed key-value storage and can provide distributed persistence for subgraph states and updated messages, as the distributed storage.

VII. EVALUATION

In this section, we choose four typical graph algorithms to evaluate the performance and concurrent update conflict probability of GraphFlow. Besides, we compare GraphFlow's performance with GraphX [3] and GraphBolt [14] using real-world datasets. GraphX is a batch processing system, while GraphBolt is an incremental graph processing system. Both of them are representative graph processing systems.

A. Experimental Setup

We choose Triangle Count (TC), Single Source Shortest Path (SSSP), PageRank (PR), and Degree Distribution (DD) to evaluate GraphFlow. Two real-world graphs, LiveJournal social network (LJ) [22], [23] and Orkut social network (ORK) [24] are used as our experimental datasets. Their scales are shown in Table V. The size of our experiment cluster is scaled from 4, 8, to 10 nodes. We call them 4-node, 8-node, and 10-node cluster. Each node has a 3.40GHz Intel® Core™ i7-2600 CPU with 8 cores, 16GB RAM and 2*1TB SATA Disks. The Operation System is Ubuntu-11.

We first compare the performance of GraphFlow with that of GraphX for TC, SSSP, PR, and DD algorithms. To verify their performances under different workloads, we generated 100K, 1M, and 70M datasets from the LJ dataset. Later, we compare the performance of GraphFlow with that of GraphBolt, which provides an open-source implementation for the PR algorithm. Both LJ and ORK datasets are used to compare their performances.

TABLE VI
AVERAGE RESPONSE TIMES (IN MILLISECONDS) OF GRAPHFLOW (GFL) AND GRAPHX (GX).

Response Time (ms)	4-node cluster			8-node cluster		
	100K	1M	70M	100K	1M	70M
TC-GX	2201	4384	100323	1830	3885	83557
TC-GFL	2.104	1.286	1.068	1.685	0.792	0.623
SSSP-GX	1709	12194	56268	1987	11681	47767
SSSP-GFL	1.812	0.629	0.42	1.47	0.752	0.3
PR-GX	4497	8617	97958	4854	7067	112836
PR-GFL	2.6	2.742	3.196	1.6	1.184	1.106
DD-GX	775	853	5022	628	852	4128
DD-GFL	0.385	0.379	0.382	0.219	0.2	0.206

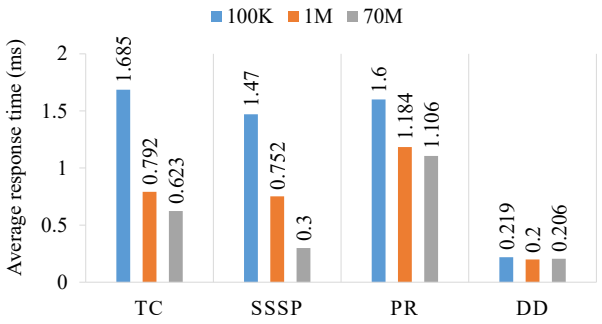


Fig. 9. GraphFlow's average response times (in milliseconds) for TC, SSSP, PR, and DD algorithms with 100K, 1M, and 70M datasets on the 8-node cluster

B. Performance

We evaluate the performance of GraphFlow for different graph algorithms by analyzing the average response times. Here, the average response times are calculated by dividing the total execution time by the number of processed events. Detailed experimental results are shown in Table VI.

1) Varing Workload Size: Fig. 9 shows GraphFlow's average response times for TC, SSSP, PR, and DD algorithms with 100K, 1M, and 70M datasets on the 8-node cluster. It can be seen that most requests can be processed within a low latency, a.k.a., 0.2ms to 1.7ms. It suggests that GraphFlow performs well across different cases. An interesting observation is that GraphFlow can achieve better performance for TC, SSSP, and

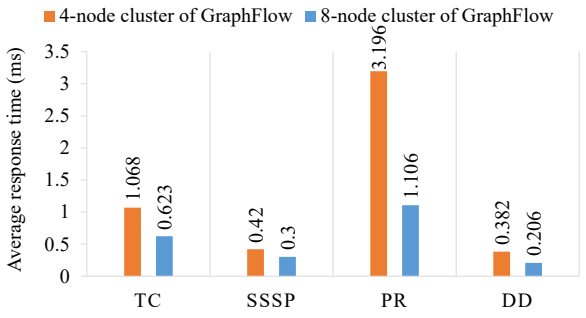


Fig. 10. GraphFlow's average response times (in milliseconds) for TC, SSSP, PR, and DD algorithms on the 4-node and 8-node cluster with 70M dataset

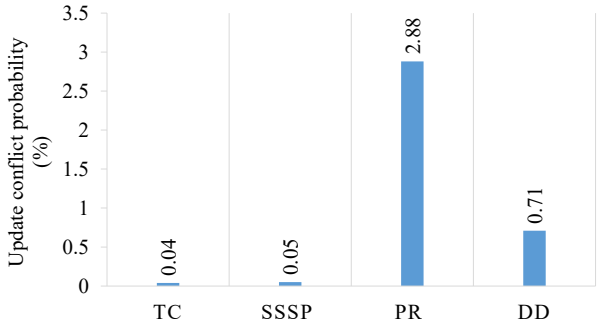


Fig. 11. GraphFlow's update conflict probability for TC, SSSP, PR, and DD algorithms with LJ dataset

DD algorithms in larger datasets. For example, the average response time for the SSSP algorithm running on the 70M dataset is 80% lower than that on the 1M dataset. However, for the PR algorithm, this trend happens on the 8-node cluster but does not happen on the 4-node cluster. This is because the PR algorithm needs more update computation than the other three algorithms and suffers from resource contention on the 4-node cluster.

It can also be seen that the dataset size has different effects for different algorithms. For example, DD's average response times are fixed at 0.2ms with 100K, 1M, and 70M datasets, while TC's average response times on the 70M dataset is 63% lower than that on the 100K dataset. The reason is that TC needs to calculate the neighboring intersection of the source and target vertex of an updated edge, while DD only needs to care about the source and target vertex of an updated edge. Because of the simple logic, DD presents stable performance when the dataset size changes.

2) Varying Cluster Size: Besides the workload size, we also run experiments to evaluate GraphFlow's scalability by varying the size of the cluster from 4 to 8 nodes. The experimental results on the 70M dataset are shown in Fig. 10. As we can see, GraphFlow performs better on the 8-node cluster than the 4-node cluster (as expected). GraphFlow's average response times on the 8-node cluster are 30%-65% lower than that on the 4-node cluster. Meanwhile, even on the 4-node cluster, the average response times across all cases are within 4ms. As such, we say GraphFlow achieves low latency and high scalability for different graph algorithms.

C. Concurrent Update Conflict Probability

To evaluate the probability of concurrent update conflict for TC, SSSP, PR, and DD algorithms, we sort the LJ dataset according to the source and target vertex, and then allocate them into 10 computing nodes using the round-robin mechanism to maximize the update conflict probability within each node. The concurrent update conflict probability is calculated by dividing the number of update conflicts by the number of requests.

As shown in Fig. 11, the concurrent update conflict probabilities for TC, SSSP, PR, and DD algorithms are all less than 3%. The influence scope of a graph algorithm can affect its update conflict. Fig. 11 shows that the concurrent update conflict probability of PR is largest. This is because the source and target vertex of a newly added edge need to propagate their updates to all vertices in the connected subgraph. During the parallel processing of updates, conflicts can be triggered. In contrast, the influence scope of SSSP is smaller, so the update conflict probability of SSSP is also lower than PR. The influence scope of DD is smallest, however, its update conflict probability is a little higher than SSSP and TC. This is because edges with common vertices are allocated to different computation nodes. More conflicts are triggered for DD when the state of the same vertex is updated by multiple computation nodes simultaneously.

D. Comparison with Existing Systems

In this section, we compare the performance of GraphFlow with the batch graph processing system GraphX and the incremental dynamic graph processing system GraphBolt.

1) Comparison with GraphX: GraphX is a distributed graph processing system following the Bulk Synchronous Parallel (BSP) [4] model, which processes graph data in micro-batch. Table VI shows the average response times of GraphX and GraphFlow across 100K, 1M, and 70M datasets in streaming scenarios.

These experimental results show that GraphFlow performs better than GraphX across all cases on both the 4-node cluster and the 8-node cluster. The experiments also show that GraphX's average response times increase linearly with the size of the dataset. For example, for the SSSP algorithm, GraphX's average response time on the 8-node cluster is about 12s and 48s when the size of the dataset is 1M and 70M, respectively. While GraphFlow's average response time is fixed at 1ms. GraphFlow outperforms GraphX because GraphX needs to re-compute all historical data to update the states of a graph for an incoming graph update event in streaming graph computation, while GraphFlow can incrementally compute the new states of a graph based on the last states. This incremental state update mechanism of GraphFlow reduces computing cost and improves performance.

Fig. 12 compares the response times of GraphX on the 4-node cluster and the 8-node cluster when the dataset size

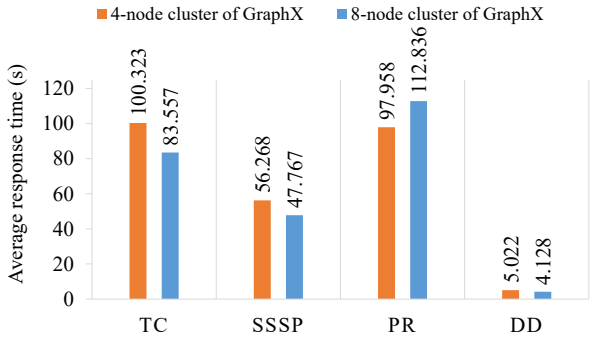


Fig. 12. GraphX's average response time (in seconds) for TC, SSSP, PR, and DD algorithms on the 4-node cluster and the 8-node cluster with 70M dataset

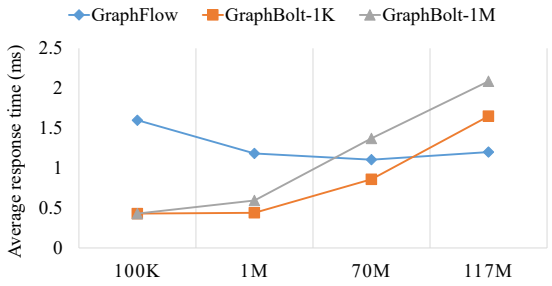


Fig. 13. Average response times (in milliseconds) of GraphFlow (GFL) and GraphBolt at different micro batches for the PR algorithm with different size of datasets

is 70M. As we can see, GraphX's response times on the 8-node cluster are around 16.5% lower than those on the 4-node cluster for TC, SSSP, and DD algorithms. Unsurprisingly, both GraphX and GraphFlow perform better on the 8-node cluster than on the 4-node cluster. However, varying the cluster size from 4 to 8 nodes, GraphFlow achieves more obvious performance improvement (with response times decreased by 30%-65%) than the performance improvement of GraphX (with response times decreased by 16.5%).

2) Comparison with GraphBolt: GraphBolt is an incremental graph processing system, and can process graph updates at different micro batches. Fig. 13 shows the average response times of GraphFlow and GraphBolt for the PR algorithm using 100K, 1M, 70M, and 117M datasets. It can be seen that GraphBolt presents better performance than GraphFlow on small datasets. The reason is that GraphFlow is a distributed solution, while GraphBolt is a single host solution. On small datasets, GraphFlow has extra communication overhead between different computing nodes. However, when the dataset size increases, GraphFlow outperforms GraphBolt. One reason is that GraphFlow can utilize the cluster's resources, while GraphBolt is limited by the resources of single node. The extra communication overhead for GraphFlow becomes less important. As shown in Fig. 13, when the dataset size is 117M, the average response time of GraphFlow is around 36% lower than that of GraphBolt. Meanwhile, both GraphFlow and GraphBolt present better performance than GraphX due to their supporting of incremental processing.

E. Discussion

Our experiments on real-world datasets show the GraphFlow achieves low latency and small update conflict probability for the implementations of TC, SSSP, PR, and DD algorithms. We also run experiments to compare GraphFlow with GraphX and GraphBolt. These experiments prove the efficiency of GraphFlow. To further investigate the performance of GraphFlow under different conditions, we have two future works. Firstly, due to the hardware limitation, the existing evaluations are limited by the size of clusters and datasets, so one future work is to run experiments on clusters with more computing nodes and using larger as well as more complex datasets. Secondly, we only implement and evaluate graph algorithms that satisfy the three principles we proposed on GraphFlow, so another future work is to evaluate the performance of GraphFlow for the graph algorithms that do not satisfy those three principles.

VIII. Related Work

Batch graph computation. Most of the existing batch graph processing systems [2], [3], [25]–[27] follow the Bulk Synchronous Parallel (BSP) [4] model, which adopts an iterative computation method and can only work on static graphs. These systems process graph update events by using synchronous computing and Vertex-Centric (VC) programming model [28]. GraphIn [29] supports online incremental graph processing. This system updates a graph in a micro-batch method. However, batch processing limits the performance of GraphIn for large-scale graphs. As such, the existing batch graph processing systems cannot efficiently process dynamical streaming graph updates and cannot meet the requirement of low latency.

Streaming graph computation. To support streaming graph processing, two popular approaches were proposed: the estimated computation approach [6]–[11] and accurate computation approach [5], [12], [13]. Sampling and summarization are the general estimated computation approaches. The sampling approach samples and processes the streaming data as it passes through, without any persistence. The works [6]–[8] belong to this sampling approach and are used to estimate the number of triangles in a graph. Spanner [9], [10], Sparsifier [11], [30], and Sketch [31]–[33] can be classified as the summarization approach. These estimated computation approaches can save memory and reduce the computation overhead. However, their processing results are usually not accurate. Moreover, it is also hard to control the error rate. For example, the work [34] indicates that the error rate of the estimated model can be as high as 95% for a large quantity of graph data that cannot be loaded into memory.

Incremental computation. There are many systems that achieve high accuracy, such as KineoGraph [12] and IncGraph [5]. The advantage of these systems is that they adopt the idea of incremental computation. As a result, their update overheads are reduced. However, they can only sequentially

process graph updates, which limits their performance and scalability. SpecGraph [13] follows a speculation-based concurrent update model. However, SpecGraph assumes that the state of a vertex only depends on messages received from its neighbors. This assumption makes the implementation of this concurrent system much easier but limits the expressiveness of this system. GraphBolt [14] provides incremental graph updates while supporting the BSP semantics on dynamic graphs analysis. However, synchronous processing would still cause performance shortboard effects. BLADYG [35] uses a block-centric approach to support dynamic graph computing. It relies on a single master node to propagate messages of all vertices, which causes the performance bottleneck for large-scale dynamic graph processing.

IX. Conclusion

In this paper, we propose a novel distributed streaming graph computation model GraphFlow, which overcomes the limitations of existing graph computation approaches. By providing the abstracts of event, state, and transform, GraphFlow supports most streaming graph algorithms. By utilizing incremental state update strategies, GraphFlow can rightly handle update conflicts and achieves high efficiency and accuracy. We implement the GraphFlow model, and complete four typical streaming graph algorithms, i.e., Degree Distribution, Triangle Count, Single Source Shortest Path, and PageRank on the system. Based on real-world datasets, we evaluate their performance and concurrent update conflict probability. Besides, we also compare GraphFlow with existing representative related works. Experiments prove the advantages of GraphFlow.

Acknowledgment

This work was supported by Guangdong Power grid limited liability company under Project 037800KC23090006.

References

[1] P. Yuan, X. Shu, C. Sha, and H. XU, "Research of large scale graph data management with in memory computing techniques," *Journal of East China Normal University*, 2014.

[2] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2010, pp. 135–146.

[3] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: a resilient distributed graph system on spark," in *Proceedings of First International Workshop on Graph Data Management Experiences and Systems (GRADES)*, 2013, p. 2.

[4] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[5] L. Shen, J. Xue, and Z. Qu, "Incgraph: Large-scale graph processing system for real-time and incremental computing," vol. 7, no. 12, pp. 1083–1092, 2013.

[6] Z. Bar-Yossef, R. Kumar, and D. Sivakumar, "Reductions in streaming algorithms, with an application to counting triangles in graphs," in *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2002, pp. 623–632.

[7] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos, "DOULION: counting triangles in massive graphs with a coin," in *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, 2009, pp. 837–846.

[8] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler, "Counting triangles in data streams," in *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 2006, pp. 253–262.

[9] S. Baswana, "Streaming algorithm for graph spanners - single pass and constant processing time per edge," *Inf. Process. Lett.*, vol. 106, no. 3, pp. 110–114, 2008.

[10] M. Elkin, "Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners," *ACM Trans. Algorithms*, vol. 7, no. 2, pp. 20:1–20:17, 2011.

[11] D. R. K. András A Benczúr and, "Approximating s-t minimum cuts in $\tilde{O}(n^2)$ time," in *Proceedings of the Twenty Eighth Annual Acm Symposium on Theory of Computing*, 2007.

[12] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: taking the pulse of a fast-changing and connected world," in *Proceedings of European Conference on Computer Systems (EuroSys)*, 2012, pp. 85–98.

[13] N. Jing, J. Xue, and Z. Qu, "Specgraph:a distributed graph processing system for dynamic result based on concurrent speculative execution," *Journal of Computer Research and Development*, 2014.

[14] M. Mariappan and K. Vora, "Graphbolt: Dependency-driven synchronous processing of streaming graphs," in *Proceedings of the Fourteenth EuroSys Conference (EuroSys)*, 2019, pp. 25:1–25:16.

[15] D. Sengupta, N. Sundaram, X. Zhu, T. L. Willke, J. S. Young, M. Wolf, and K. Schwan, "Graphin: An online high performance incremental graph processing framework," in *Proceedings of International Conference on Parallel and Distributed Computing*, 2016, pp. 319–333.

[16] M. A. Hasan and V. S. Dave, "Triangle counting in large networks: a review," *WIREs Data Mining Knowl. Discov.*, vol. 8, no. 2, 2018.

[17] Z. Zhang, "Study of shortest path problem on large-scale graph," Ph.D. dissertation, University of Science and Technology of China, 2014.

[18] L. Page, S. Brin, and T. Winograd, "The pagerank citation ranking: bringing order to the web," in *Stanford InfoLab*, 1999.

[19] M. Girvan and M. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 99, no. 12, 2002.

[20] V. D. Blondel, J. Guillaume, and R. Lambiotte, "Fast unfolding of communities in large networks: 15 years later," *CoRR*, vol. abs/2311.06047, 2023.

[21] "Hazelcast," https://hazelcast.org/, 2023.

[22] "Live journal," http://www.livejournal.com/, 2023.

[23] "Livejournal data set," http://snap.stanford.edu/data/soc-LiveJournal1.html, 2023.

[24] "Orkut data set," http://snap.stanford.edu/data/com-Orkut.html, 2023.

[25] "Graph processing with apache flink," https://flink.apache.org/2015/08/24/introducing-gelly-graph-processing-with-apache-flink/, 2015.

[26] C. Avery, "Giraph: Large-scale graph processing infrastructure on hadoop," *Proceedings of the Hadoop Summit*, vol. 11, no. 3, pp. 5–9, 2011.

[27] "Apache hama," https://attic.apache.org/projects/hama.html, 2023.

[28] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From "think like a vertex" to "think like a graph"," *Proc. VLDB Endow.*, vol. 7, no. 3, pp. 193–204, 2013.

[29] D. Sengupta, N. Sundaram, X. Zhu, T. L. Willke, J. S. Young, M. Wolf, and K. Schwan, "Graphin: An online high performance incremental graph processing framework," in *Proceedings of International Conference on Parallel and Distributed Computing*, 2016, pp. 319–333.

[30] D. A. Spielman and S. Teng, "Spectral sparsification of graphs," *SIAM J. Comput.*, vol. 40, no. 4, pp. 981–1025, 2011.

[31] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[32] P. Zhao, C. C. Aggarwal, and M. Wang, "gsketch: On query estimation in graph streams," *Proc. VLDB Endow.*, vol. 5, no. 3, pp. 193–204, 2011.

[33] N. Tang, Q. Chen, and P. Mitra, "Graph stream summarization: From big bang to big crunch," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2016, pp. 1481–1496.

[34] S. Chu and J. Cheng, "Triangle listing in massive networks and its applications," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2011, pp. 672–680.

[35] S. Aridhi, A. Montresor, and Y. Velegrakis, "BLADYG: A graph processing framework for large dynamic graphs," *Big Data Res.*, vol. 9, pp. 9–17, 2017.