GraPU: Accelerate Streaming Graph Analysis through Preprocessing Buffered Updates

Feng Sheng, Qiang Cao, Haoran Cai, Jie Yao and Changsheng Xie
Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System,
School of Computer Science and Technology, Huazhong University of Science and Technology
CorrespondingAuthor:caoqiang@hust.edu.cn

ABSTRACT

Streaming graph analysis extracts timely insights from evolving graphs, and has gained increasing popularity. For current streaming graph analytics systems, incoming updates are simply cached in a buffer, until being applied onto existing graph structure to construct a new snapshot. Iterative graph algorithms then work on the new snapshot to produce up-to-date analysis result. Nevertheless, we find that for widely used monotonic graph algorithms, the buffered updates can be effectively preprocessed to achieve fast and accurate analysis on new snapshots.

To this end, we propose GraPU, a streaming graph analytics system for monotonic graph algorithms. Before applying updates, GraPU preprocesses buffered updates in two sequential phases: 1) *Components-based Classification* first identifies the effective graph data that are actually affected by current updates, by classifying the vertices involved in buffered updates according to the predetermined connected components in underlying graph; 2) *In-buffer Precomputation* then generates safe and profitable intermediate values that can be later merged onto underlying graph to facilitate the convergence on new snapshots, by precomputing the values of vertices involved in buffered updates. After all updates are applied, GraPU calculates new vertex values in subgraph-centric manner. GraPU further presents *Load-factors Guided Balancing* to achieve subgraph-level load balance, by efficiently reassigning some vertices and edges among subgraphs beforehand. Evaluation shows that GraPU outperforms state-of-the-art KineoGraph by up to 19.67x, when running four monotonic graph algorithms on real-word graphs.

CCS CONCEPTS

• **Theory of computation → Graph algorithms analysis**; • **Computer systems organization → Real-time system architecture**; *Distributed architectures*;

KEYWORDS

Distributed graph analysis, streaming graphs, preprocessing

SoCC '18, October 11–13, 2018, Carlsbad, CA, USA

ACM ISBN 978-1-4503-6011-1/18/10…$15.00
https://doi.org/10.1145/3267809.3267811

ACM Reference Format:
Feng Sheng, Qiang Cao, Haoran Cai, Jie Yao, and Changsheng Xie. 2018. GraPU: Accelerate Streaming Graph Analysis through Preprocessing Buffered Updates. In *SoCC '18: ACM Symposium on Cloud Computing, October 11–13, 2018, Carlsbad, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3267809.3267811

1 INTRODUCTION

Many online services in clouds have been built on evolving graphs, such as social network analysis [15], natural language processing [41], and web information retrieval [20]. These graphs exhibit drastic increases in both the scale and changing frequency, for example, about 0.8 million new users were created and 235 million new friendships were established for the social networks of Facebook per day in 2017 [10]. While performing analysis over these fast evolving graphs can provide valuable insights for users, it also poses great challenges to underlying systems, since analysis results should be timely and correct as *graph-structured* updates (e.g., a set of newly added edges) are continuously streaming in.

Some systems [9, 21, 28, 31, 36] have recently emerged to analyze evolving graphs at scale. They generally run iterative graph algorithms on a snapshot, which is a full version of the graph in consistency and integrity. Incoming updates are applied onto the graph structure of last snapshot to construct a new snapshot. For example, KineoGraph [4] developed an epoch-based commit protocol to periodically apply updates onto graph structure of the last snapshot, and an incremental graph-computation engine to improve computational efficiency on the new snapshot. When incoming updates involve edge deletions, KickStarter [35] can produce safe and profitable intermediate values if running monotonic graph algorithms on the new snapshot, by trimming the values of vertices affected by deleted edges.

However, in these systems, incoming updates cannot be applied immediately, and they are merely cached in a fixed-sized buffer until being applied in a batch to construct next snapshot. This method separates graph updates from graph computations, but decreases the responsiveness of querying on latest updates. Even so, we find that for extensively used *monotonic* graph algorithms, the values of vertices involved in buffered updates can be precomputed to produce a set of safe intermediate values, which will be later merged onto underlying graph to enable faster convergence on new snapshots. Moreover, we observe that each batch of updates typically exhibit spatial locality [9], and they often affect only a subset of underlying graph when being applied. Both the observations inspire us to execute some preprocessing operations for buffered updates, to shorten the merging time of applying updates onto underlying

graph and the convergence time of running graph algorithms on new snapshots.

This paper strives to sufficiently exploit the implicit information within buffered updates and the processing capability of underlying system, to accelerate graph analysis on new snapshots. For this end, we propose GraPU, a streaming graph analytics system for monotonic graph algorithms. Before applying each batch of updates, GraPU preprocesses buffered updates in two sequential phases. After the updates are applied, GraPU calculates new vertex values in subgraph-centric manner, with a subgraph-level load balancing scheme performed beforehand.

Two-phase Preprocessing. GraPU preprocesses buffered updates with Components-based Classification and In-buffer Precomputation. GraPU first identifies underlying graph as a series of connected components in which one or more vertices can reach each other. In Components-based Classification, each batch of updates are then classified according to predetermined components in the graph. In this way, the vertices within buffered updates and underlying graph can be clearly correlated or demarcated based on the components. This classification avoids loading an amount of irrelevant graph data when applying updates, and reserve more memory space for analysis on new snapshots. After the classification, In-buffer Precomputation will precompute the values of vertices involved in buffered updates, to obtain a set of intermediate values. Once buffered updates are applied, those intermediate values can be merged onto corresponding components in underlying graph to achieve faster convergence on new snapshots.

Subgraph-level Balancing. To improve the efficiency of calculating vertex values in new snapshots, each component in underlying graph is further partitioned into a set of subgraphs where a subgraph is the computational unit of graph algorithms. However, compared to the subgraph-centric manner in static graph analysis [29, 32], the subgraph-level load imbalance of streaming graph analysis is more severe, as different numbers of vertices and edges are added (or removed) in each subgraph when applying updates. To achieve subgraph-level load balance, GraPU presents the Load-factors Guided Balancing, which adjusts some vertices and edges among subgraphs beforehand according to the *load factor*, a key metric that characterizes the load intensity of each subgraph by counting the number of included edges. This balancing scheme also tries to maintain data locality by reassigning a vertex to the subgraph containing most of its neighbors. As a result, GraPU effectively accelerate the convergence of graph algorithms.

To be specific, we make the following contributions:

(1) We propose two novel phases for sequentially preprocessing buffered updates, namely Components-based Classification and In-buffer Precomputation respectively.

(2) We present a subgraph-level load balancing scheme designated for streaming graph analysis, namely Load-factors Guided Balancing.

(3) We implement a prototype GraPU based on a distributed subgragh-centric graph analytics system GoFFish [29]. With four popular monotonic graph algorithms and two typical datasets, our experiments demonstrate that GraPU outperforms state-of-the-art system KineoGraph by up to 19.67x.

The rest of this paper is organized as follows. Section 2 overviews the background and motivation. Section 3 introduces the two-phase preprocessing and subgraph-level balancing proposed in GraPU. Section 4 reports the experimental setup and evaluation results. We discuss related works in Section 5 and conclude this paper in Section 6.

2 BACKGROUND AND MOTIVATION

2.1 Architecture Overview

Figure 1 demonstrates the architecture of existing streaming graph analytics systems [28], which is mainly composed of three different types of nodes, namely *buffer nodes*, *worker nodes* and *master nodes* respectively.

Incoming graph-structured updates are first cached on the buffer nodes. Under certain conditions (explained in Subsection 2.2), buffer nodes assign all buffered updates to corresponding worker nodes, according to a predefined partitioning scheme that determines the mapping of vertices to worker nodes [5]. The partitioning scheme is stored on a distributed file system for message passing and vertices reassignment, and is accessible to all nodes. In general, the buffer nodes have similar hardware configurations with worker nodes.

When the updates are received, the worker nodes load whole graph structure of last snapshot from external storage, and apply received updates onto the graph structure to construct a new snapshot. Then, iterative graph algorithms work on the new snapshot in the Bulk Synchronous Parallel (BSP) model [33], to calculate up-to-date vertex values. Finally, the analysis result of new snapshot is stored into external storage for fault tolerant [23].

The master nodes are responsible for answering user queries. When a user query arrives, master nodes push this query to corresponding worker nodes according to the shared partitioning scheme, the worker nodes then create extra threads to retrieve analysis result requested by the query. The attained result is finally sent to users by master nodes. Moreover, master nodes can collect execution statics from each worker node for load balance [12].

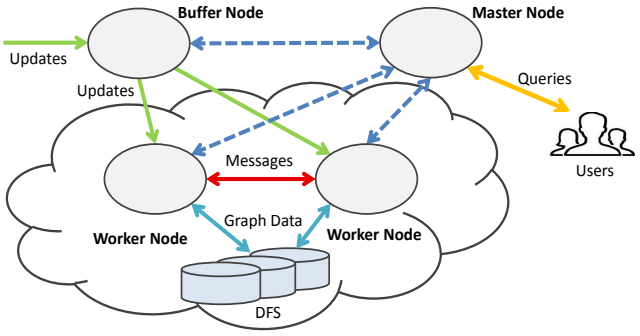


Figure 1: Architecture of existing streaming graph analytics systems.

2.2 Workflow of Answering Queries

Figure 2 depicts the workflow of applying incoming updates and generating analysis results for user queries, which typically comprises three steps:

I. Incoming updates are first cached in a fixed-sized buffer.
II. Under specific conditions, current batch of buffered updates are applied onto the graph structure of last snapshot (e.g., the *Snapshot-N* in Figure 2), to construct a new snapshot (e.g., the *Snapshot-(N+1)*).
III. Graph algorithms work on the new snapshot, to generate up-to-date analysis result for user queries.

Note that, there are two main conditions that trigger the application of buffered updates onto underlying graph. In most cases, updates are periodically applied when the buffer is filled or a time interval is reached, which we term as the *general condition*. But in some special cases, users may issue ad-hoc queries [28] to retrieve the exact result at that moment, leading to immediate application of buffered updates even though the buffer is not filled or the time interval is not met, which we term as the *ad-hoc condition*.

Furthermore, as the vertex values right before the updates are often closer to the final result of new snapshot than initial values (i.e., more profitable), most systems maintain the intermediate values of iterative computations on the recent snapshot. Therefore, when buffered updates are applied, the accurate result of new snapshot can be easily obtained by performing iterative computations starting from the intermediate values, as a form of *incremental computation* that corrects the errors in intermediate values [34]. This method of incremental computation effectively avoid performing computations on new snapshots from the scratch.

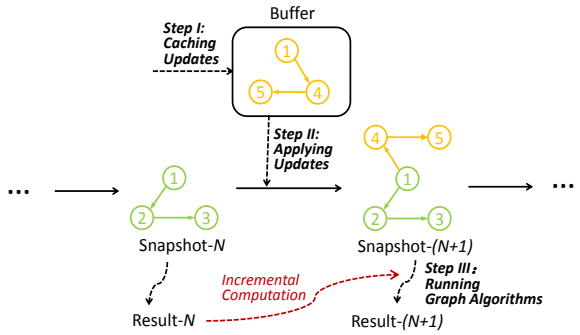


Figure 2: The workflow of applying buffered updates and generating up-to-date analysis results for user queries.

2.3 Opportunities and Challenges

For existing systems, all incoming updates are merely cached in a fixed-sized buffer, until being applied onto underlying graph under general or ad-hoc condition. However, we observe that the vertices and edges involved in buffered updates typically form a *local* graph topology, which is a subset of the graph structure of new snapshot. Exploiting the local graph topology within updates can compute values of some vertices beforehand, potentially to generate profitable intermediate values that are closer to the final result of new snapshot. Moreover, the underlying graph often comprises many connected components where one or more vertices can reach each other. Based on the property of connected components, each update will only affect the components that its involved vertices belong to. Classifying vertices in both updates and underlying graph according to corresponding components, can identify the effective graph data that are actually affected by current updates. These observations inspire us to develop some preprocessing approaches for buffered updates, to facilitate the application of updates (i.e., the *Step II* in Figure 2) and to accelerate the convergence on new snapshots (i.e., the *Step III* in Figure 2).

Nevertheless, preprocessing updates poses new challenges to the infrastructure of underlying system, in terms of the correctness and timeliness of analysis results. For the *correctness*, as some graph algorithms (e.g., PageRank) require all in-edges of a vertex to compute its new value, exploiting the local graph topology within updates may provide unsafe intermediate values, which can potentially lead to incorrect result for these algorithms. For the *timeliness*, preprocessing should satisfy three features to achieve efficiency: 1) Transparent, preprocessing does not affect the tasks running on worker and master nodes. 2) Low-overhead, preprocessing incurs no large overhead that delays the application of updates, or causes no additional hardware investment for buffer nodes. 3) Profitable, as a baseline, the convergence performance with preprocessing is no lower than that without preprocessing.

2.4 Monotonic Graph Algorithms

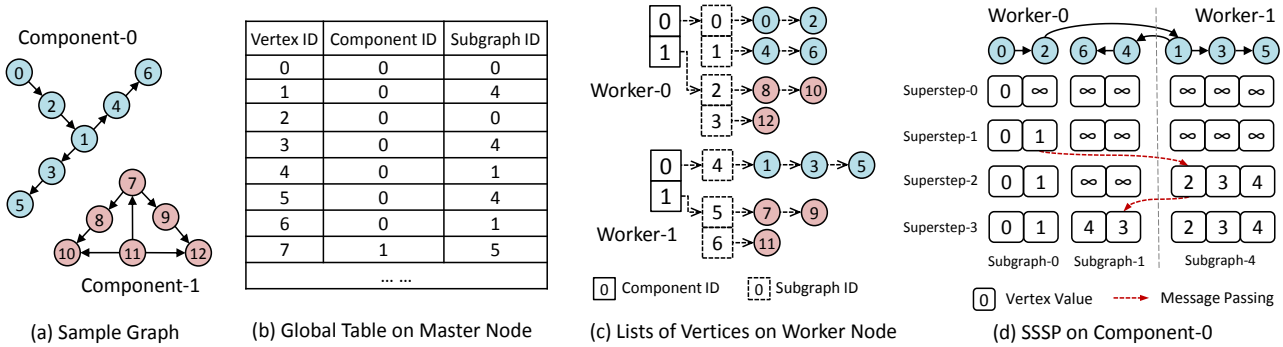
As to the challenge on correctness, we find that exploiting the local graph topology within updates can provide safe intermediate values for widely used monotonic graph algorithms. There are a wide range of monotonic graph algorithms, in which every vertex value increases/decreases consistently across supersteps, as the examples listed in Table 1. Furthermore, as observed by previous work [35], for monotonic graph algorithms, the value of a vertex is typically selected from one of its in-edges. In essence, the update function of each vertex is a selection function that compares all its in-edges and choose the maximal/minimal one as the new value. Therefore, the new value of a vertex depends on one of its in-edges that exists within either underlying graph or buffered updates. If the critical in-edge lies in underlying graph, preprocessing will produce an ineffective intermediate value, and thus makes no difference to final value of the vertex. But if the in-edge exists within buffered updates, preprocessing can produce a safe intermediate value that contributes to final value of the vertex, which will be introduced in Subsection 3.2.

3 DESIGN OF GRAPU

To overcome the challenge on timeliness, this section proposes GraPU, a streaming graph analytics system built on the architecture in Figure 1. To fully exploit each batch of buffered updates, GraPU adopts two novel preprocessing phases, namely *Components-based Classification* and *In-buffer Precomputation* respectively. After the updates are applied, GraPU employs the subgraph-centric manner to improve computational efficiency on new snapshots, and further presents *Load-factors Guided Balancing* to achieve subgraph-level load balance.

3.1 Components-based Classification

The underlying graph generally consists of many *connected components* [24]. A vertex can reach all other vertices in the same component via some specific paths (ignoring the directions of edges



(a) Sample Graph (b) Global Table on Master Node (c) Lists of Vertices on Worker Node (d) SSSP on Component-0

Figure 3: The data structure used to store the component information of vertices involved in the sample graph (a). Master nodes store the full component information of all vertices using the table (b), each worker node stores the component information of its assigned vertices using a series of lists (c). Subfigure (d) illustrates the supersteps when running SSSP on the Component-0 in subgraph-centric manner, assuming the starting vertex is the vertex ⓪ and all edge weights equal to 1.

along the paths), but is unreachable from any vertices in a different component. As an example, there are two connected components in Subfigure 3(a). Due to the characteristic of connected components, a graph update only has impact on the underlying components that its involved vertices belong to. For instance, a newly added edge will affect the components containing its source and destination vertices. To identify the underlying components affected by current updates, the vertices within updates and underlying graph are both classified according to corresponding components.

3.1.1 Classifying buffered updates

When the original graph is loaded and distributed over worker nodes, the *Weakly Connected Component (WCC)* [24] algorithm is performed on the graph (add reverse edges if it is a directed graph), to *initialize* the component IDs for all vertices in underlying graph. Afterwards, master nodes collect the generated component IDs from every worker node, and maintain the full component information of all vertices to provide fast lookup for buffer nodes. Besides, each worker node also stores the component information of its assigned vertices, to identify the underlying components affected by current updates.

For each batch of buffered updates, the involved vertices are classified and applied in following steps:

I. Buffer nodes send the vertex IDs involved in buffered updates to master nodes.
II. Master nodes return the component IDs related with these vertices to buffer nodes.
III. Buffer nodes assign each update along with relevant component IDs to corresponding worker nodes, according to the shared partitioning scheme.
IV. Worker nodes load the graph structure of relevant components according to received component IDs, and then apply updates onto the loaded graph structure.

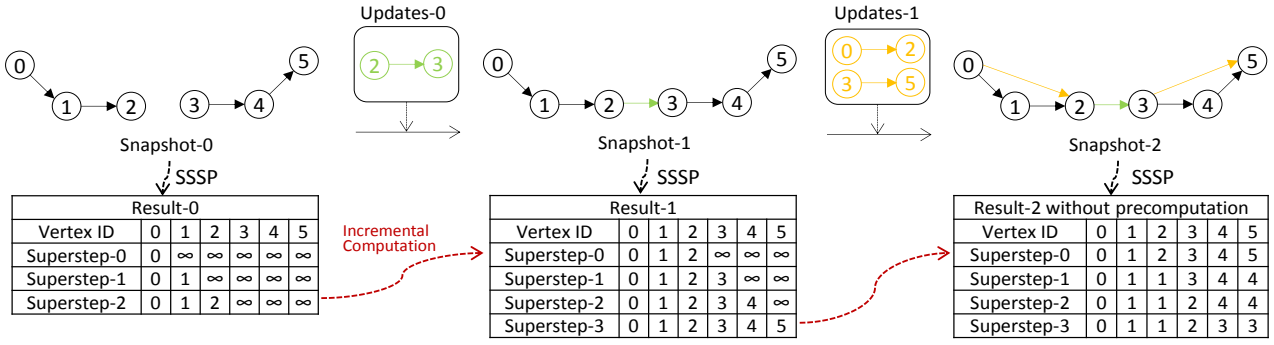
3.1.2 Maintaining component information

To quickly locate the component ID of each vertex requested by buffer nodes, master nodes employ a hash table keyed by vertex IDs to store the component information of all vertices, as illustrated in Subfigure 3(b). Differently, to sequentially load all vertices belonging to a certain component when applying updates, every worker node adopts a series of lists indexed by component IDs to store the component information of its assigned vertices, as depicted in Subfigure 3(c).

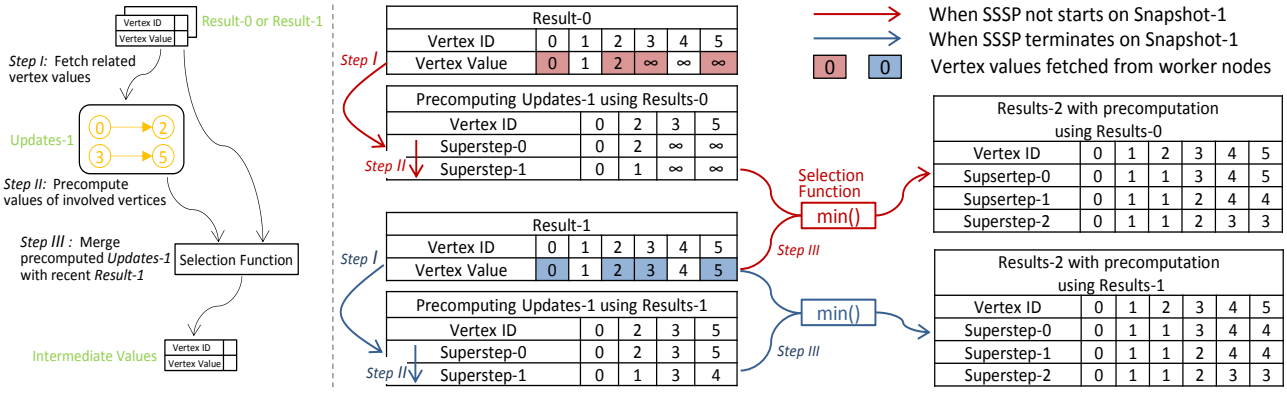
Case of Edge Additions. When a batch of updates are applied, worker nodes can determine whether a newly added edge has linked two separate components, according to the component IDs of source and destination vertices of the edge. If the component IDs are different, then two separate components are linked by the edge, and thus corresponding lists of vertices belonging to these two components are merged. This method of merging lists can determine new component IDs for affected vertices on-the-fly, avoiding the expensive cost of recalculating component IDs using the WCC algorithm. Finally, the new component IDs of affected vertices are informed to master nodes.

Case of Edge Deletions. Note that if buffered updates involve edge deletions, the source and destination vertices of a deleted edge should belong to the same component, because there are no edges spanning two different components. Therefore, similar to the case of edge additions when applying updates, the graph structure of corresponding components that deleted edges belong to is loaded into memory to construct a new snapshot. Nevertheless, the deleted edges may arbitrarily divide a loaded component into several new components. To detect the potential division of each loaded component, the WCC algorithm has to be performed on the new snapshot, which causes non-negligible overhead of recalculating new component IDs. Finally, the new component IDs of affected vertices are informed to master nodes.

To summarize, the benefit of Components-based Classification is two-fold. First, the updates cached during a time period often exhibit spatial locality [9] (e.g., a new social network user often builds many friendships when created), thus they are usually applied onto a limited number of underlying components. Owing to the classification, the graph structure of irrelevant components is not loaded from external storage when applying updates, except that their vertex values computed from the preceding snapshot are



(a) The workflow of calculating the *Result-2* without precomputing the *Updates-1*.



(b) The workflow of calculating the *Result-2*, with precomputing the *Updates-1* using the *Result-0* or *Result-1* respectively.

Figure 4: A comparison between workflows of calculating the final *Result-2* for SSSP algorithm, with and without precomputing the *Updates-1* respectively. The starting vertex for SSSP is the vertex ⓪ and all edge weights equal to 1.

loaded for query services (these vertex values will not be changed by current updates). As comparison, existing systems have to load whole graph structure from external storage whenever applying updates. Unloading irrelevant components avoids a number of unnecessary IOs, thus greatly shortens the time of applying updates. Moreover, this further reserves more memory space for the analysis on new snapshots, leading to faster convergence of graph algorithms. Second, the ad-hoc queries issued at a certain moment often concern a few components [37] (e.g., a hot topic interested by the public at present). When ad-hoc queries arrive, only the updates related with the components that the ad-hoc queries concern will be applied. In contrast, existing systems have to apply all buffered updates when ad-hoc queries arrive. This prioritization facilitates the application of updates that are relevant with ad-hoc queries, significantly decreasing the latency in answering ad-hoc queries.

3.2 In-buffer Precomputation

Case of Edge Additions. After the classification, the vertices within buffered updates can be further precomputed, to produce a set of intermediate values that are closer to the final result of new snapshot. To this end, buffer nodes should fetch current values of involved vertices and edges from corresponding worker nodes. However, while the values of edges are usually invariant during running time of graph algorithms, the values of vertices often change frequently across supersteps. When buffer nodes fetch related vertex values from worker nodes, graph algorithms are potentially still running on the last snapshot. Fortunately, owing to the monotonicity of vertex values for monotonic graph algorithms, the vertex values fetched at different supersteps would not affect the safety of generated intermediate values, if the updates only involve edge additions. Adding new edges preserves existing graph structure, and thus will not break the monotonicity of vertex values, as demonstrated in [35].

Figure 4 depicts how to precompute buffered updates with an example of SSSP algorithm, which calculates the lengths of paths from a given starting vertex to all other vertices in a weighted graph. In this example, there are two batches of buffered updates denoted as *Updates-0* and *Updates-1* respectively, which separately generate two snapshots denoted as *Snapshot-1* and *Snapshot-2* respectively (the original graph is denoted as *Snapshot-0*). For simplicity, we assume that the starting vertex for SSSP algorithm is the vertex ⓪ and all edge weights in each snapshot equal to 1. Subfigure 4(a) illustrates the workflow of calculating the *Result-2* without precomputation. As it shows, because of the incremental computation, the intermediate values (i.e., the vertex values in the *Superstep-0*) for

analysis on the new snapshot totally derive from the converged result of the proceeding snapshot.

Subfigure 4(b) illustrates the workflow of calculating the *Result-2* with precomputing the *Updates-1*. First, to precompute the *Updates-1*, current values of its involved vertices {⓪, ②, ③, ⑤} need be fetched from corresponding worker nodes (i.e., the *Step I* in Subfigure 4(b)). Here, for simplicity, we show two opposite situations where SSSP on the last *Snapshot-1* has not yet started and has already finished respectively, and other situations are in between. Therefore, when fetching related vertex values from worker nodes, if SSSP has not started running on the *Snapshot-1*, corresponding vertex values $\{0, 2, \infty, \infty\}$ in the previous converged *Result-0* will be used. Otherwise, if SSSP has already completed on the *Snapshot-1*, corresponding vertex values $\{0, 2, 3, 5\}$ in the recent converged *Result-1* will be used. Based on the local graph topology within the *Updates-1* (i.e., the edges ⓪ → ② and ③ → ⑤) and the fetched vertex values, the values of all involved vertices {⓪, ②, ③, ⑤} can be precomputed on buffer nodes (i.e., the *Step II* in Subfigure 4(b)).

As stated earlier, the value of each vertex for monotonic graph algorithms is determined by one of its in-edges. For a given vertex, if the in-edge exists within underlying graph, the value in recent converged result would be closer to the final value than the value obtained from precomputation. Thus, to provide more profitable intermediate values for running SSSP on the new *Snapshot-2*, the result of precomputing the *Updates-1* on buffer nodes and the recent converged *Result-1* on worker nodes are *merged* when applying updates, by using the predefined selection function *min*() to select the minimal value from these two results for each vertex (i.e., the *Step III* in Subfigure 4(b)). Although precomputing the *Updates-1* using the *Result-0* or *Result-1* respectively both produces profitable intermediate values that lead to less number of supersteps on the new snapshot, precomputing based on the recent *Result-1* enables the generated intermediate values closer to the final *Result-2*. In general, precomputing updates based on the most recent result can lead to faster convergence of graph algorithms on new snapshots.

Case of Edge Deletions. The situation becomes much more complicated when updates involve edge deletions, as the graph-structured changes may break the monotonicity and invalidate the generated intermediate values. To overcome the challenge caused by edge deletions, we borrow the trimming techniques proposed in KickStarter [35], which produces safe intermediate values by constructing the *value dependence trees* for vertices in current snapshot. Upon an edge deletion, the vertices whose values are affected by the deleted edge can be identified by traversing the subtree rooting the destination vertex of the edge. KickStarter further computes intermediate values for these affected vertices by re-executing the update function starting from the destination vertex.

To conclude, precomputing buffered updates with current vertex values obtained from worker nodes, can produce profitable intermediate values that accelerate the convergence of graph algorithms on new snapshots, especially when buffered updates involve a large proportion of newly added vertices.

3.3 Subgraph-level Load Balancing

To further improve the efficiency of calculating new vertex values after all buffered update are applied, we employ the *subgraph-centric* computation model [32]. Intuitively, a subgraph is defined as a *local* component within a partition. That is, within one graph partition, a vertex can reach all other vertices in the same subgraph, but is unreachable from any vertices in a different subgraph. However, two subgraphs on different partitions may have boundary edges that connect their vertices, as long as these two subgraphs belong to the same connected component. In terms of the hierarchy, some subgraphs form a connected component, and then all connected components form the whole graph structure.

3.3.1 Subgraph-centric computation

In subgraph-centric manner, a subgraph is the computational unit within a superstep. Specifically, within each superstep, the vertices in one subgraph first communicate with each other and calculate values independently, then the subgraph sends messages to remote subgraphs at the synchronization barrier in BSP model. Iterative computations terminate when there are no active vertices and new incoming messages in all subgraphs. Subfigure 3(d) illustrates the supersteps when running SSSP algorithm on the *Component-0* in Subfigure 3(a) using subgraph-centric manner, assuming that the starting vertex is the vertex ⓪ and all edges weights equal to 1. As it shows, the vertex values within a subgraph can be determined within one superstep, and messages are exchanged among subgraphs. The subgraph information is initialized by performing the WCC algorithm on each graph partition, and is maintained on-the-fly using the same method as connected components. To sequentially load the vertices in a given subgraph, the *subgraphID*s are introduced into the lists of vertices maintained by worker nodes, as shown in Subfigure 3(c).

Compared to traditional vertex-centric manner, the ability to access vertices in a whole subgraph enables subgraph-centric manner to avoid many unnecessary message exchanges between vertices, and require less number of supersteps to reach convergence.

3.3.2 Load-factors guided balancing

Relative to the subgraph-centric manner in static graph analysis, load imbalance of streaming graph analysis can be more severe, as various numbers of vertices and edges are added into each subgraph when applying updates. The time taken by a superstep in subgraph-centric manner usually depends on the slowest subgraph, therefore the computational efficiency of subgraph-centric manner would decrease as updates are periodically applied. To eliminate subgraph-level load imbalance, we present Load-factors Guided Balancing, which is briefly described in Algorithm 1.

The first step of balancing is to detect load imbalance (line 1-5). To this end, a metric that can quantitatively characterize the imbalance among subgraphs is in need. Our previous work [27] has demonstrated that, the difference of processing times can represent the load imbalance. Moreover, as advocated in many papers [18, 42], the processing time for graph analysis is mainly decided by the number of edges rather than vertices. Based on these concepts, we propose a novel metric *load factor* that characterizes the load intensity of each subgraph by estimating two parts of the processing time, namely the time on processing local edges and the time on sending remote messages respectively. Specifically, the load factor

Algorithm	Update Function	Selection Function	Type
MC	$v.value \leftarrow \max(v.value, \max_{e \in InEdges(v)}(e.source.value))$	*max()*	Label Propagation
BFS	$v.depth \leftarrow \min_{e \in InEdges(v)}(e.source.depth + 1)$	*min()*	Graph Traversal
SSSP	$v.path \leftarrow \min_{e \in InEdges(v)}(e.source.path + e.weight)$	*min()*	Graph Traversal
SSWP	$v.path \leftarrow \max_{e \in InEdges(v)}(\min(e.source.path, e.weight))$	*max()*	Graph Traversal

Table 1: Monotonic graph algorithms

of a given subgraph i on the worker node j is calculated as follow:

$$loadFactor_i = \frac{innerE_i + boundaryE_i}{processRate_j} + \frac{boundaryE_i}{transferRate_j} \quad (1)$$

where the $innerE_i$ is the number of inner edges whose source and destination vertices are both in the subgraph i, the $boundaryE_i$ is the number of boundary edges connecting one vertex on a remote subgraph. Besides, the $processRate_j$ and $transferRate_j$ are respectively the throughput of processing local edges and sending remote messages in the node j, since both variables may fluctuate at runtime, we use the values measured in the last superstep to calculate current load factor. Since the edges involved in buffered updates will affect the workload distribution over subgraphs in new snapshot, after all updates are applied and the new snapshot is constructed, the number of inner and boundary edges in each subgraph are counted to calculate current load factor.

Once the load factors are calculated to characterize the load intensity of every subgraph, the difference of all load factors should be quantified to determine whether the workload distribution over subgraphs is severely imbalanced. For this purpose, we adopt the *Variation Coefficient (VC)* of all load factors to quantify such difference, which is calculated as follow:

$$VC = \left[\frac{1}{N} \sum_{i=1}^{N} \left(\frac{loadFactor_i - \overline{loadFactor}}{\overline{loadFactor}} \right)^2 \right]^{\frac{1}{2}} \quad (2)$$

where $\overline{loadFactor}$ is the average load factor, and N is the total number of subgraphs. The Variation Coefficient has shown to be an effective metric to detect load imbalance in our previous work [27]. To limit the overhead of handling load imbalance, we further introduce a threshold to ignore some slight imbalance where the cost is expected to be greater than the benefit of balancing. Specifically, a load imbalance is determined only if the VC of all load factors is greater than the predefined threshold(line 6).

Once a load imbalance is detected, the VC of current load factors should be reduced below the threshold by *reassigning* some vertices and edges among subgraphs, to provide a balanced workload distribution for subgraphs in the new snapshot. To this end, the subgraph with the largest load factor (i.e., the heaviest-loaded) first migrates some of its vertices and edges to those subgraphs whose load factors are lower than the average $\overline{loadFactor}$ (i.e., light-loaded). Moreover, to decrease the number of messages transferred over networks when running graph algorithms on new snapshot, the vertices connecting much more boundary edges than inner edges are prioritized to be migrated (line 9). Besides, to achieve the data locality of accessing vertices in new snapshot, each reassigned vertex is migrated to the light-loaded subgraph that has the most neighbors of this vertex (line10). A certain number of vertices and edges in the heaviest-loaded subgraph are migrated, until its load factor decreases to a given range (e.g., 5% above the $\overline{loadFactor}$ in our evaluation). Then, the next heavy-loaded subgraph begins its vertices reassignment.

Such iterative reassignment of vertices migrated from heavy-loaded subgraphs to light-loaded subgraphs terminates, when the VC of current load factors is reduced below the threshold. Finally, the new locations of reassigned vertices are stored on the shared partitioning scheme for future message passing, and the new subgraph IDs of reassigned vertices are informed to master nodes. Note that after buffered updates are applied, the workflows of Load-factors Guided Balancing for edge additions and deletions are the same, except that edge additions increase the load factors of subgraphs while edge deletions decrease the load factors.

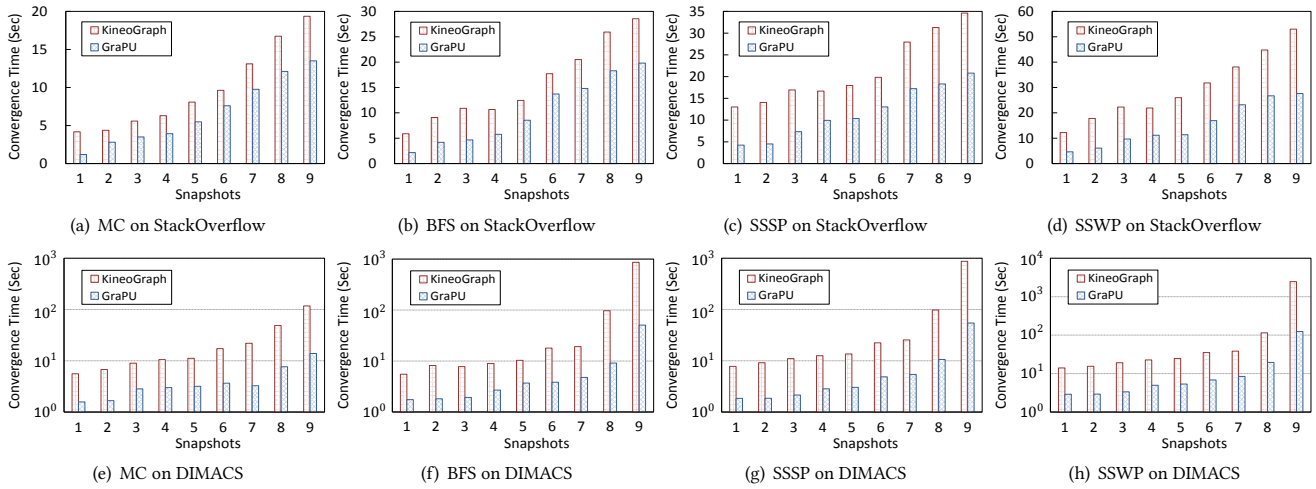
Algorithm 1: Load-factors Guided Balancing

```
   Input: Set of subgraphs S, Threshold VC_thr
   Output: Set of balanced subgraphs S'
1  foreach subgraph i ∈ S do
2      Count innerE_i and boundaryE_i of the subgraph i;
3      Calculate loadFactor_i of the subgraph i;
4  end
5  Calculate VC of all loadFactors;
6  while VC >VC_thr do
7      Get the subgraph s ∈ S with the largest loadFactor;
8      while loadFactor_s >105% * avg(loadFactor) do
9          Get the vertex v ∈ s with largest boundaryE − innerE;
10         Reassign v to the subgraph d ∈ S , where d has the largest
             numOfNeighbors(v) and
             loadFactor_d < avg(loadFactor);
11         Record v.location on the shared partitioning scheme;
12         Inform v.subgraphID to master nodes;
13         Recalculate loadFactor_s;
14     end
15     Recalculate VC of current loadFactors;
16 end
17 S' ⇐ S
```

4 EVALUATION

4.1 Experimental Setup

We implemented GraPU based on a distributed subgraph-centric graph analytics system GoFFish [29]. While GoFFish has several



(a) MC on StackOverflow (b) BFS on StackOverflow (c) SSSP on StackOverflow (d) SSWP on StackOverflow

(e) MC on DIMACS (f) BFS on DIMACS (g) SSSP on DIMACS (h) SSWP on DIMACS

Figure 5: The convergence times of two monotonic graph algorithms on different snapshots using KineoGraph and GraPU respectively, in the context of general condition.

implementation versions and only works on static graphs, GraPU is built on the *GoFFish-Giraph* with the support for analysis on evolving graphs. We compare GraPU with state-of-the-art KineoGraph [4], which periodically applies buffered updates without preprocessing. Since KineoGraph is not openly available, we prototype its architecture on *Apache Giraph* [7]. The experiments are conducted on an in-house cluster composed of 1 master node, 1 buffer node and 10 worker nodes. Each node has one Intel Xeon E5-2650 processor (8 cores), 8GB DRAM and one 500GB 7200rpm HDD, all nodes are connected via 1Gb Ethernet.

Algorithms: We select four representative monotonic graph algorithms, as listed in Table 1. *Max Computation (MC)* finds the maximal vertex value in a graph. *Breadth First Search (BFS)* traverses whole graph to compute the depths of all vertices from a designated starting vertex. *Single Source Shortest Paths (SSSP)* calculates the lengths of paths from a given starting vertex to all other vertices in a weighted graph. *Single Source Widest Paths (SSWP)* searches a path between a specific starting vertex and every other vertex in a weighted graph, to maximize the weight of the minimum-weight edges in the path.

These graph algorithms share some common characteristics: 1) Vertex values are monotonic across supersteps. 2) In every superstep, the value of an active vertex is determined by one of its in-edges. 3) Only a subset of vertices are active in each superstep. 4) graph algorithms terminate when there are no active vertices and new incoming messages.

Datasets: Two real-word graphs listed in Table 2 are used in our experiments. *StackOverflow* [17] is a temporal graph that tracked interactions of users on the community website Stack Overflow over 2774 days, in which an edge $u \rightarrow v$ with a timestamp t represents that the user u commented on the user v at the time t. Differently, *DIMACS* [6] is a non-temporal graph that records the USA road networks, in which each vertex represents a city and weight of each edge is the distance between a pair of cities. We randomly assign each edge with a timestamp, to simulate the build time of a road.

The main differences between the graph structures of these two graphs are: 1) The graph structure of StackOverflow is dense (i.e., higher per-vertex degree) while its diameter is short, DIMACS has a sparse graph structure but a long diameter. 2) The degrees of vertices in StackOverflow exhibit the power-law distribution [8], while the vertices in DIMACS connect a similar number of edges. 3) Tens of thousands of weakly connected components exist within StackOverflow, but there is only one in DIMACS.

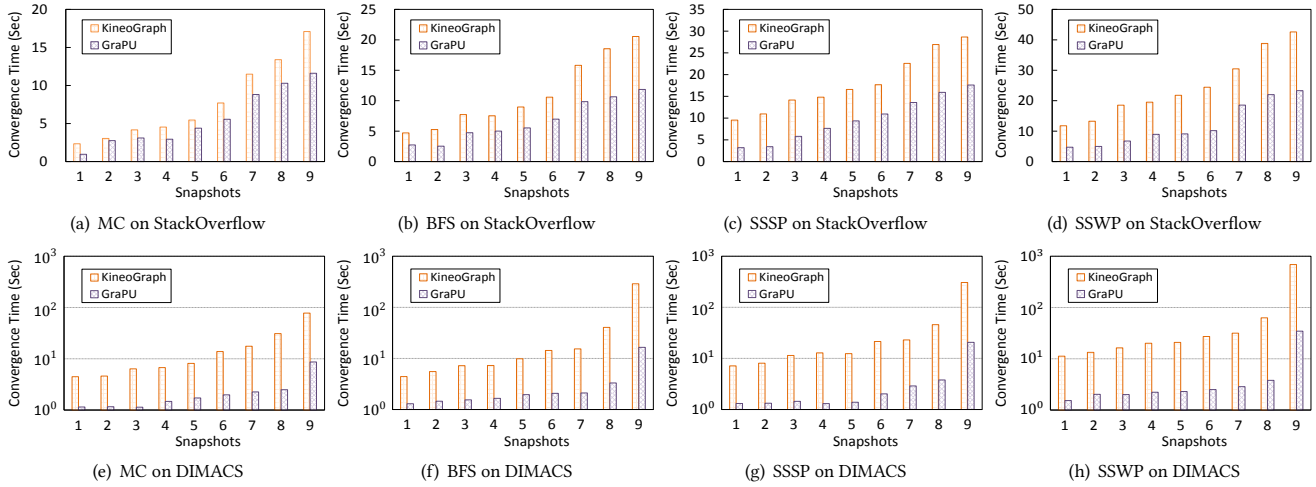
Dataset	$\|V\|$	$\|E\|$	Avg.Degree	Diameter	WCC
StackOverflow	2.6M	63M	24.4	26	23,580
DIMACS	14M	34M	2.4	3,838	1

Table 2: Datasets

Methodology: We respectively divide the edges in each graph into 10 groups according to the ranges of their timestamps. The first group of edges form the original snapshot 0, remaining groups separately form 9 batches of buffered updates and each of them is applied one at a time to construct a new snapshot. We compares the convergence times of graph algorithms on various snapshots using KineoGraph and GraPU respectively, under two different conditions. In the general condition, the 9 batches of updates are applied sequentially and integrally. To evaluate the systemic performance under the ad-hoc condition, we issue a query to retrieve the values of 10 random vertices in each snapshot. Note that for the case of edge deletions, the overhead of recalculating new component IDs with WCC algorithm and the reference of computing intermediate values with trimming technique [35] cause GraPU to be comparative with state-of-the-art systems (e.g., KickStarter). Therefore, we evaluate the performance of GraPU when buffered updates only involve edge additions.

4.2 Performance under General Condition

Figure 5 compares the runtime performance of the two systems on different snapshots. As it shows, GraPU consistently outperforms

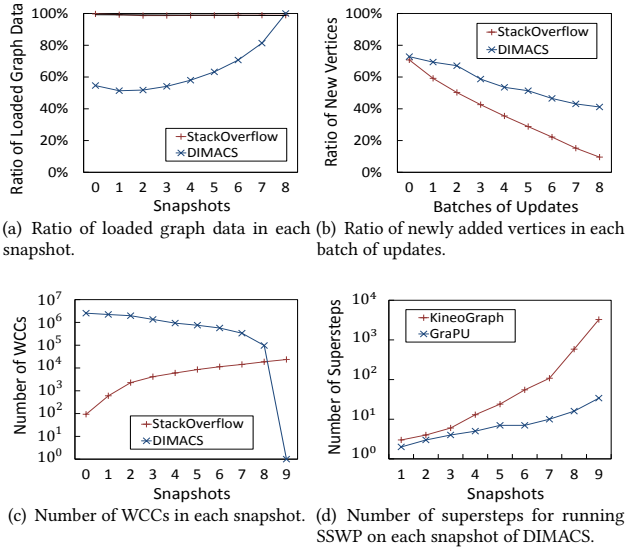


(a) MC on StackOverflow (b) BFS on StackOverflow (c) SSSP on StackOverflow (d) SSWP on StackOverflow

(e) MC on DIMACS (f) BFS on DIMACS (g) SSSP on DIMACS (h) SSWP on DIMACS

Figure 6: The convergence times of two monotonic graph algorithms on different snapshots using KineoGraph and GraPU respectively, in the context of ad-hoc condition.

KineoGraph on various combinations of algorithms and datasets. When running MC, BFS, SSSP, and SSWP on StackOverflow, GraPU exceeds KineoGraph by average 1.68x, 1.78x, 2.05x, and 2.13x respectively. The limited improvement of GraPU is caused by three factors: 1) Due to the dense graph structure of StackOverflow (a vertex connects average 24 edges), each batch of updates have impact on almost all vertices in underlying graph. As shown in Subfigure 7(a), nearly all graph data in last snapshot are loaded from disks when applying each batch of updates to construct a new snapshot, which invalidates the effectiveness of Components-based Classification. 2) Since the value of a vertex for monotonic graph algorithms is determined by one of its in-edges, the values of many newly added vertices can be calculated in advance by In-buffer Precomputation, especially for the first several batches of updates where most vertices are newly added, as illustrated in Subgraph 7(b). However, as the ratio of new vertices decreases and each vertex connects more in-edges, the benefit of precomputation is reduced. 3) The relative advantage of running graph traversal algorithms in subgraph-centric manner is further offset by the short diameter of StackOverflow. Nevertheless, GraPU can still benefit from Load-factors Guided Balancing, especially for the latter several snapshots where the vertices are unevenly distributed over subgraphs. Note that, the convergence times of all graph algorithms increase with the size of snapshots rather than updates, which echoes the finding in previous research [4].



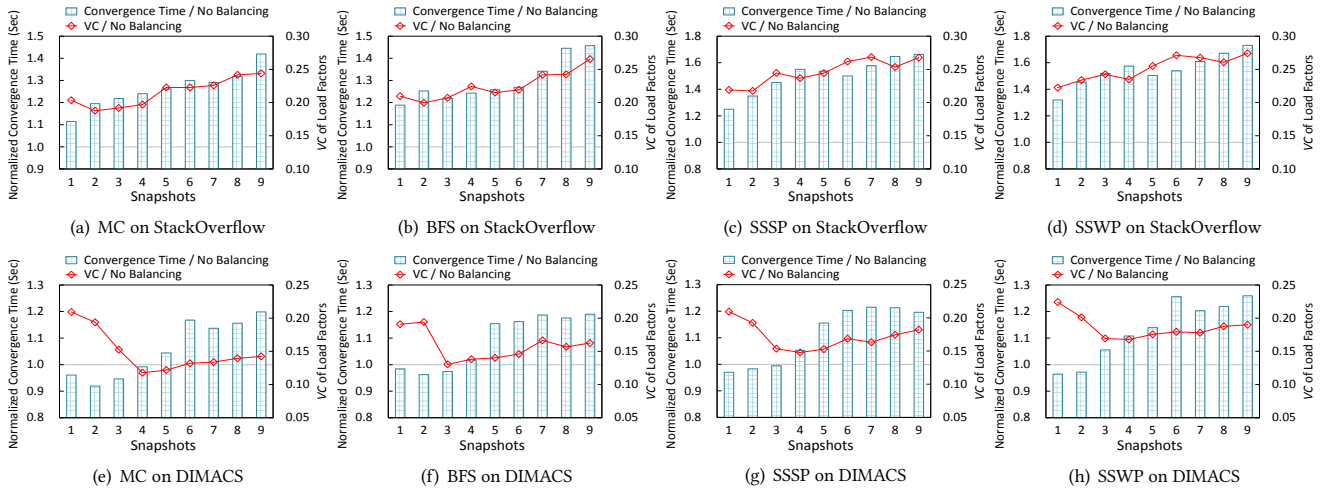
(a) Ratio of loaded graph data in each snapshot. (b) Ratio of newly added vertices in each batch of updates.

(c) Number of WCCs in each snapshot. (d) Number of supersteps for running SSWP on each snapshot of DIMACS.

Figure 7: Variations of some metrics when running graph algorithms on different snapshots under general condition.

In terms of running MC, BFS, SSSP, and SSWP on DIMACS, GraPU attains significant speedups of average 4.15x, 5.89x, 6.41x, and 6.71x respectively over KineoGraph. The great improvement of GraPU also derives from three factors: 1) The sparse graph structure of DIMACS leads to relatively low ratio of loaded graph data, especially for the first several snapshots, as demonstrated in Subfigure 7(a). 2) Because of the high ratio of new vertices in each batch of updates, the values of many newly added vertices are determined in advance by precomputation. 3) The long diameter of DIMACS further expands the advantage of subgraph-centric manner, resulting in greatly less number of supersteps and synchronization overhead. As an example shown in Subfigure 7(d), when running SSWP on the snapshot 9 of DIMACS, the number of superstep for GraPU is two orders of magnitude less than that for KineoGraph, leading to the maximal speedup of 19.67x achieved by GraPU.

Note that, the convergence times of these graph algorithms increase dramatically over the latter several snapshots of DIMACS (e.g., when running SSWP on DIMACS, KineoGraph takes 114s on the snapshot 8 and 2459s on the snapshot 9), which is attributed to two factors: 1) Since the vertices in DIMACS are sparsely



(a) MC on StackOverflow (b) BFS on StackOverflow (c) SSSP on StackOverflow (d) SSWP on StackOverflow

(e) MC on DIMACS (f) BFS on DIMACS (g) SSSP on DIMACS (h) SSWP on DIMACS

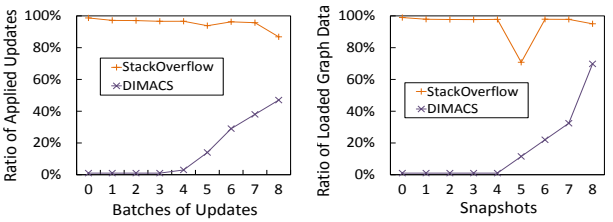
Figure 8: The convergence times and the VCs of load factors when running graph algorithms on different snapshots, using GraPU without Load-factors Guided Balancing. The convergence times are normalized to that with balancing.

connected, millions of connected components co-exist within underlying graph, as shown in Subfigure 7(c)), but many of them contains only a few vertices. Thus, when running graph traversal algorithms on these snapshots, almost all connected components are inactive except for the one containing the starting vertex. However, as more updates are applied, multiple connected components are merged into a larger one, which results in drastic increases in both the number of active vertices and the convergence times. Second, the diameter of DIMACS increases substantially over the latter several snapshots, which leads to much more supersteps and synchronization overhead.

In the context of general condition, the classification and precomputation of GraPU work well on sparse graphs where the per-vertex degree is relatively low. The long diameter of a graph further expands the advantage of subgraph-centric manner.

4.3 Performance under Ad-hoc Condition

When an ad-hoc query arrives, KineoGraph applies all buffered updates while GraPU only applies the updates that are related with the ad-hoc query. As presented in Figure 6, the performance advantage of GraPU over KineoGraph varies significantly across different datasets.



(a) Ratio of applied updates in each batch of updates (b) Ratio of loaded graph data in each snapshot

Figure 9: Variations of some metrics when running graph algorithms on different snapshots under ad-hoc condition.

When running graph algorithms on each snapshot of the dense graph StackOverflow, the convergence times under ad-hoc condition are similar with that under general condition. Although there are tens of thousands of connected components in StackOverflow, a few of them contain most vertices in the graph while others include only several vertices, as a form of the power-law distribution. As a consequence, to calculate the values of 10 random vertices requested by each ad-hoc query, a majority of updates are applied (as depicted in Subfigure 9(a)) because their involved vertices are in the same connected components with the 10 requested vertices, and almost all graph data in last snapshot are loaded (as shown in Subfigure 9(b)) to construct the new snapshot. Therefore, the benefit of Components-based Classification is offset by the high connectivity of vertices in StackOverflow.

As for running graph algorithms on each snapshot of DIMACS, the performance advantage of GraPU over KineoGraph becomes even larger under ad-hoc condition, compared to that under general condition. To calculate the values of 10 random vertices, only a fraction of graph data in last snapshot are loaded due to the loosely connected vertices in DIMACS, as shown in Subfigure 9(b). Furthermore, because the vertices in DIMACS are uniformly distributed over connected components, a low proportion of updates that are related with each ad-hoc query are applied, as illustrated in Subfigure 9(a). However, as more vertices are added and many connected components are merged, both the ratio of loaded graph data and the ratio of applied updates increase steadily, which sightly decreases the performance advantage of GraPU on latter several snapshots of DIMACS.

To conclude, when running graph algorithms under ad-hoc condition, the performance advantage of GraPU over KineoGraph further increases on sparse graphs, owing to the classification that identifies the updates and graph data related with ad-hoc queries.

4.4 Effect of Balancing

To evaluate the effect of Load-factors Guided Balancing on performance of GraPU, we compare the convergence times of graph algorithms on different snapshots with and without balancing respectively (the threshold for balancing is set to 0.10). As shown in Figure 8, the subgraph-level balancing effectively improves performance of GraPU when running graph algorithms on StackOverflow, but brings limited benefit for DIMACS.

Because of the power-law degree distribution exhibited by vertices in StackOverflow, a few subgraphs contain much more vertices than other subgraphs, which leads to severe subgraph-level load imbalance, especially for latter several snapshots. As a result, the performance of GraPU on StackOverflow is effectively improved by Load-factors Guided Balancing, for example, SSWP converges faster by average 1.55x with balancing.

Differently, since the vertices in DIMACS are more uniformly distributed over subgraphs, GraPU benefits less from subgraph-level balancing. Particularly, when running on the first several snapshots of DIMACS, the convergence times of graph algorithms with balancing is even longer than that without balancing, because of the overhead of reassigning vertices. Note that, due to a small number of vertices contained in each subgraph, the VCs of load factors in the first several snapshots are unstable and relatively higher. However, as more vertices and edges involved in buffered updates are added onto underlying subgraphs, the subgraph-level load imbalance emerges, and the benefit of balancing gradually dominates on the latter several snapshots of DIMACS.

5 RELATED WORKS

Static Graph Analysis. A large body of research focus on performing fast analysis on large-scale static graphs, in which the graph structure does not change over time. These works either scale up graph analysis in a single machine for efficiency and cost-effectiveness [1, 18, 19, 40], or scale out on a cluster of machines for performance and scalability [29, 38, 39, 42]. Some of them further leverage the massive amount of parallelism provided by GPU and specific accelerators [25, 26]. Even though they are incompetent to perform iterative analysis over evolving graphs, their contributions on some aspects (e.g., reduce random disk accesses [14], limit synchronization overhead [16], and improve network efficiency [3]) can be incorporated to accelerate streaming graph analysis.

Temporary Graph Analysis. Temporary graph analysis works on *a series of* snapshots to capture the changes of graphs during a time period (e.g., the traffic changes of a road network during a day), which is different from streaming graph analysis that performs graph algorithms on a recent snapshot to retrieve the timely attributes of graphs (e.g., the traffic of a road network at present). Chronos [9] designed an in-memory data layout to preserve time-locality within the snapshots of a temporary graph, and carefully scheduled iterative computations on these snapshots to leverage the time-locality. Version Traveler [11] stores the differences among multiple snapshots of a temporary graph as deltas, and construct the next snapshot by integrating current snapshot with a corresponding delta, in order to reduce the snapshot-switching cost. Differently, [30] proposed a temporally iterative BSP programming abstraction to process time-series graphs, in which the graph structure is nearly invariable but the attributed values of vertices can vary significantly across different snapshots.

Streaming Graph Analysis. Recently, many systems have been developed to facilitate analysis on large-scale evolving graphs. KineoGraph [4] proposed an epoch commit protocol to periodically generate snapshots, and support incremental computations on vertices in both push and pull model. Naiad [21] presented a new computation model *timely dataflow*, which enables asynchronous incremental computations. UNICORN [31] designed the GIM-V model to incrementally process evolving graphs with Matrix-Vector operations. Tornado [28] organizes the iterative computations on evolving graphs into a main loop and several branch loops, in which the main loop ingests updates and maintains intermediate values while branch loops are forked from the main loop to obtain accurate results. KickStarter [35] produces safe and profitable intermediate values for monotonic graph algorithms, by trimming the values of vertices that are affected by deleted edges. Graspan [36] is a disk-based parallel graph system that analyzes program graphs with constant edge additions. Many other studies concentrate on online graph partitioning as updates continuously stream in [2, 5], or optimizations for specific algorithms on evolving graphs [13, 22]. However, incoming updates in these systems are periodically applied without preprocessing, which inspires our work to sufficiently exploit the buffered update to accelerate analysis on new snapshots.

6 DISCUSSIONS AND CONCLUSIONS

In this paper, we strive to fully leverage the buffered updates to accelerate streaming graph analysis. To this end, we design and implement a streaming graph analytics system GraPU that works for a general class of monotonic graph algorithms. GraPU efficiently preprocesses buffered updates with Components-based Classification and In-buffer Precomputation, and it performs analysis on new snapshots in subgraph-centric manner enhanced with Load-factors Guided Balancing. Our experimental results show that, by effectively exploiting buffered update, GraPU outperforms state-of-the-art KineoGraph by up to 19.67x, when running four monotonic graph algorithms on real-word graphs.

We do recognize some limitations of GraPU. First, there are many graph algorithms that performs non-monotonic computations over vertices, thus we plan to expand GraPU with the support for non-monotonic graph algorithms (e.g., PageRank and Graph Coloring). Second, although some techniques have been introduced in the work to handle edge deletions involved in updates, we realize their poor efficiency and high overhead, and therefore consider more optimized preprocessing methods for edge deletions as future work.

7 ACKNOWLEDGMENTS

We thank the anonymous reviewers, and the fellow researchers Keval Vora, Mingxing Zhang and Xu Zhou for their valuable comments and suggestions. This work is supported in part by Nature Science Foundation of China under Grant No. 61872156 and No. 61821003, the Fundamental Research Funds for the Central Universities No. 2018KFYXKJC037, and Alibaba Group through Alibaba Innovative Research (AIR) Program.

REFERENCES

[1] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. 2017. Squeezing out All the Value of Loaded Data: An Out-of-core Graph Processing System with Reduced Disk I/O. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC)*. 125–137.

[2] Nikos Armenatzoglou, Huy Pham, Vasilis Ntranos, Dimitris Papadias, and Cyrus Shahabi. 2015. Real-Time Multi-Criteria Social Graph Partitioning: A Game Theoretic Approach. In *Proceedings of the 2015 ACM International Conference on Management of Data (SIGMOD)*. 1617–1628.

[3] Maciej Besta, Michal Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. 93–104.

[4] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the Seventh European Conference on Computer Systems (EuroSys)*. 85–98.

[5] Dong Dai, Wei Zhang, and Yong Chen. 2017. IOGP: An Incremental Online Graph Partitioning Algorithm for Distributed Graph Databases. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. 219–230.

[6] DIMACS. 2006. 9th DIMACS Implementation Challenge - Shortest Paths. http://www.dis.uniroma1.it/challenge9/. (2006).

[7] Apache Software Foundation. 2016. The Apache Giraph Project. http://giraph.apache.org/. (2016).

[8] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, (OSDI)*. 17–30.

[9] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*. 1:1–1:14.

[10] Zephoria Inc. 2018. The Top 20 Valuable Facebook Statistics. https://zephoria.com/top-15-valuable-facebook-statistics/. (2018).

[11] Xiaoen Ju, Dan Williams, Hani Jamjoom, and Kang G. Shin. 2016. Version Traveler: Fast and Memory-Efficient Version Switching in Graph Processing Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC)*. 523–536.

[12] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. 2013. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the Eighth European Conference on Computer Systems (EuroSys)*. 169–182.

[13] Nicolas Kourtellis, Gianmarco De Francisci Morales, and Francesco Bonchi. 2016. Scalable online betweenness centrality in evolving graphs. In *Proceedings of the 32nd IEEE International Conference on Data Engineering (ICDE)*. 1580–1581.

[14] Pradeep Kumar and H. Howie Huang. 2016. G-store: high-performance graph store for trillion-edge processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 830–841.

[15] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue B. Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th International Conference on World Wide Web (WWW)*. 591–600.

[16] Michael LeBeane, Shuang Song, Reena Panda, Jee Ho Ryoo, and Lizy K. John. 2015. Data partitioning strategies for graph workloads on heterogeneous clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 56:1–56:12.

[17] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (2014).

[18] Hang Liu and H. Howie Huang. 2017. Graphene: Fine-Grained IO Management for Graph Computing. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*. 285–300.

[19] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woon-Hak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*. 527–543.

[20] Einat Minkov, William W. Cohen, and Andrew Y. Ng. 2006. Contextual search and name disambiguation in email using graphs. In *Proceedings of the 29th Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 27–34.

[21] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the 24th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 439–455.

[22] Muhammad Usman Nisar, Sahar Voghoei, and Lakshmish Ramaswamy. 2017. Caching for Pattern Matching Queries in Time Evolving Graphs: Challenges and Approaches. In *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2352–2357.

[23] Mayank Pundir, Luke M. Leslie, Indranil Gupta, and Roy H. Campbell. 2015. Zorro: zero-cost reactive failure recovery in distributed graph processing. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC)*. 195–208.

[24] Vibhor Rastogi, Ashwin Machanavajjhala, Laukik Chitnis, and Anish Das Sarma. 2013. Finding connected components in map-reduce in logarithmic rounds. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE)*. 50–61.

[25] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 622–636.

[26] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. 2015. GraphReduce: processing large-scale graphs on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 28:1–28:12.

[27] Feng Sheng, Qiang Cao, Haoran Cai, Jie Yao, and Changsheng Xie. 2017. Laro: Lazy repartitioning for graph workloads on heterogeneous clusters. In *Proceedings of the 36th IEEE International Performance Computing and Communications Conference (IPCCC)*. 1–8.

[28] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. 2016. Tornado: A System For Real-Time Iterative Analysis Over Evolving Data. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*. 417–430.

[29] Yogesh Simmhan, Alok Gautam Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi S. Raghavendra, and Viktor K. Prasanna. 2014. GoFFish: A Sub-graph Centric Framework for Large-Scale Graph Analytics. In *Proceedings of the 20th European Conference on Parallel Processing (Euro-Par)*. 451–462.

[30] Yogesh L. Simmhan, Neel Choudhury, Charith Wickramaarachchi, Alok Gautam Kumbhare, Marc Frîncu, Cauligi S. Raghavendra, and Viktor K. Prasanna. 2015. Distributed Programming over Time-Series Graphs. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 809–818.

[31] Toyotaro Suzumura, Shunsuke Nishii, and Masaru Ganse. 2014. Towards large-scale graph stream processing platform. In *Proceedings of the 23rd International World Wide Web Conference (WWW)*. 1321–1326.

[32] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From "Think Like a Vertex" to "Think Like a Graph". *PVLDB* 7, 3 (2013), 193–204.

[33] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (1990), 103–111.

[34] Keval Vora, Rajiv Gupta, and Guoqing (Harry) Xu. 2016. Synergistic Analysis of Evolving Graphs. *TACO* 13, 4 (2016), 32:1–32:27.

[35] Keval Vora, Rajiv Gupta, and Guoqing (Harry) Xu. 2017. 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 (ASPLOS)*. 237–251.

[36] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing (Harry) Xu, and Ardalan Amiri Sani. 2017. Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 389–404.

[37] Kai Wang, Guoqing (Harry) Xu, Zhendong Su, and Yu David Liu. 2015. GraphQ: Graph Query Processing with Abstraction Refinement - Scalable and Programmable Analytics over Very Large Graphs on a Single PC. In *Proceedings of the 2015 USENIX Annual Technical Conference, (USENIX ATC)*. 387–401.

[38] Lei Wang, Liangji Zhuang, Junhang Chen, Huimin Cui, Fang Lv, Ying Liu, and Xiaobing Feng. 2018. Lazygraph: lazy data coherency for replicas in distributed graph-parallel computation. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 276–289.

[39] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. 2016. Exploring the Hidden Dimension in Graph Processing. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 285–300.

[40] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. 2018. Wonderland: A Novel Abstraction-Based Out-Of-Core Graph Processing System. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 608–621.

[41] Linhong Zhu, Aram Galstyan, James Cheng, and Kristina Lerman. 2014. Tripartite graph clustering for dynamic sentiment analysis on social media. In *Proceedings of the 2014 International Conference on Management of Data (SIGMOD)*. 1531–1542.

[42] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 301–316.