ACGraph: Accelerating Streaming Graph Processing via Dependence Hierarchy

Zihan Jiang[1], Fubing Mao[1*] , Yapu Guo[1], Xu Liu[1], Haikun Liu[1], Xiaofei Liao[1], Hai Jin[1], Wei Zhang[2]
[1]National Engineering Research Center for Big Data Technology and System,
Services Computing Technology and System Lab, Cluster and Grid Computing Lab,
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China
[2]Department of Electronic and Computer Engineering, HKUST, Hong Kong
{jiangzihan, fbmao, guoyapu, liuxu2021, hkliu, xfliao, hjin}@hust.edu.cn, wei.zhang@ust.hk

2023 60th ACM/IEEE Design Automation Conference (DAC) | 979-8-3503-2348-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/DAC56929.2023.10247904

***Abstract*—Streaming graph processing needs to timely evaluate continuous queries. Prior systems suffer from massive redundant computations due to the irregular order of processing vertices influenced by updates. To address this issue, we propose ACGraph, a novel streaming graph processing approach for monotonic graph algorithms. It maintains dependence trees during runtime, and makes affected vertices processed in a top-to-bottom order in the hierarchy of the dependence trees, thus normalizing the state propagation order and coalescing of multiple propagation to the same vertices. Experimental results show that ACGraph reduces the number of updates by 50% on average, and achieves the speedup of 1.75∼7.43× over state-of-the-art systems.**

***Index Terms*—graph processing, streaming graph, incremental computation, state propagation**

I. Introduction

Graph plays a significant role in many domains due to its ability to capture relationships between objects, such as social networks [1], recommendation systems [2], and complex relational networks. However, most real-world graphs are evolving over time, and need to be updated timely to keep the result up to date. Graph like this is called streaming graph, where a stream of updates consisting of edge (or vertex) additions and edge (or vertex) deletions are applied to the graph in batches to generate new results [3]. Streaming graph processing mainly contains two phases: update and computation [4]. In the update phase edge updates are first ingested into the graph structure, then in the computation phase the latest results of the graph are computed. When graph evolves, the computation can be restarted after updating the graph structure, which treats the graph as a static one. However, a batch of updates only influence a small part of vertices and meanwhile graph changes are arriving over time. Thus, processing streaming graph in above-mentioned manner will lead to expensive computation latency.

To efficiently process streaming graphs, a large number of software systems [1], [5]–[11] are proposed to quickly process batches of update. These systems adopt an efficient technique of incremental computation, which incrementally refines the result on a snapshot of the previous graph. In other words, after applying updates to graph structure, vertices affected by edge additions and edge deletions will first be identified, and only their new states will be propagated to their neighbors until the whole graph converges to an up-to-date state. However, existing software systems still suffer from low convergence speed due to redundant computation. Specifically, new states of the affected vertices are usually propagated in an irregular order to enable high parallelism. However, state changes propagated from affected vertices usually pass through the same subset of vertices, and this characteristic causes multiple updates of these vertices. In addition, multiple updates of a vertex's state will cause multiple propagation of its state to its neighbors in a chain reaction, which exacerbates performance degradation.

Currently, many innovative solutions have been proposed to accelerate static graph processing [12]–[15]. For example, Digraph [13] and Depgraph [12] divide a directed graph into several paths and process each path in an asynchronous manner. Grasp [14] designs a domain-specialized cache policy to keep hot vertices from cache thrashing. However, these approaches rely on consistent graph structure, making them not suitable for streaming graphs.

As for streaming graphs, many recently proposed systems and accelerators focus on storage [10], [16], batch preprocessing [9], [17], and dynamic repartitioning [18]. In order to speed up computation stage, KickStarter [6] uses a trimming technique based on dependence tree to reduce recomputations in edge deletion phase, which is further adopted by most of recent systems. However, their methods contribute nothing to edge addition phase, which still suffers from extensive redundant computation. TDGraph [19] utilizes a topology-driven approach to synchronize incremental computations of all affected vertices, and consequently reduce redundant computations. However, this approach needs a pre-traversal from all affected vertices, which introduces expensive overhead if implemented in software. Drawbacks of these systems indicate that a lightweight and efficient approach is needed to speed up streaming graph computation.

In our work, we analyse the propagation characteristics of monotonic graph algorithms when facing streaming updates

* Fubing Mao is the corresponding author. This work is supported by National Key Research and Development Program of China under grant No. 2022YFB2404202, National Natural Science Foundation of China under grant No. 61832006, Huawei Technologies Co., Ltd (No.YBN2021035018A6) and Hong Kong Research Grants Council GRF under Grant 16213521.

and observe that not only can the dependence tree be used to trim deletion computations, but it can also be adopted to greatly optimize both addition and deletion computation. When streaming updates arrive, the level of each vertex in dependence tree will only change within limited cases, allowing us to process active vertices (e.g., affected by edge updates) in a top-to-bottom order based on their levels in dependence trees. Following this way, a large part of redundant computation caused by irregular process order of vertices will be removed.

Based on our observation, we propose ACGraph, an efficient runtime approach to greatly accelerate streaming graph processing for monotonic algorithms. ACGraph maintains dependence trees of vertices and utilizes them to significantly accelerate convergence of the whole graph when processing batches of updates. In addition, ACGraph introduces barely no extra overhead to achieve the optimization and can also guarantee high parallelism. We evaluate ACGraph using four monotonic algorithms and five real-world graphs. The results indicate that our approach reduces the number of vertices updates by 50% on average, and achieves the speedup of about 1.75$\sim$7.43$\times$ over state-of-the-art streaming graph systems. Especially, we improve the runtime by up to 22.8$\times$ for edge additions.

In summary, our main contributions are as follows:

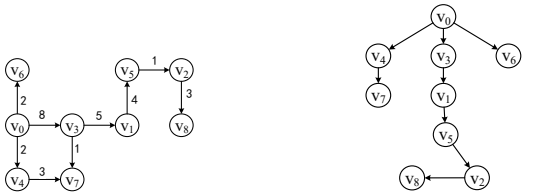
- We comprehensively analyze the characteristics of vertex dependencies evolving for monotonic algorithms in streaming graph processing.
- We propose an effective runtime approach to reduce the redundant computation while guaranteeing high parallelism with barely no extra overhead introduced.
- We develop ACGparh to support low-latency incremental processing for monotonic algorithms. Experimental results reveal that our proposed ACGparh outperforms state-of-the-art methods.

The remainder of this paper is organized as follows. Section II describes the background and our motivation. Section III introduces the details of our observations and approaches, along with the implementation of ACGraph. We conduct experiments and evaluate the performance of ACGraph in Section IV. Finally, we conclude our work in Section V.

II. Background

A. Streaming Graph Analytics

Incremental Computation Workflow. In streaming graph processing, graph updates usually only affect a small fraction of vertices. Thus, incremental computation is widely adopted to achieve a fast response to a query [19]. Streaming updates are usually cached in a buffer and applied in batches. When the latest result is needed, a batch of buffered updates will be applied to the graph and affected vertices will be reset to a safe approximate value while the other vertices remain unchanged. The new result will be generated by restarting graph algorithm in an incremental manner. Graph updates usually consist of edge additions and deletions, which are processed in different ways. Edge additions suit for the incremental computation well and are easy to process since they bring no effect that needs to be reverted while edge deletions do not. If an edge is deleted, we first need to cancel all the influences made by this edge before conducting incremental computation. For accumulative algorithms, its easier to do this. We can just send the inverse value of the previous converged state of the source vertex along the deleted edge to eliminate its influence. For monotonic (i.e. selective) algorithms, edge deletions can result in incorrect results and are not supported until KickStarter is proposed.



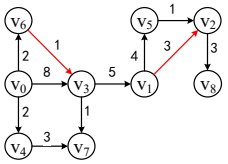
(a) An example graph (b) The corresponding dependence trees

Fig. 1. An example graph and its corresponding dependence trees after running SSSP

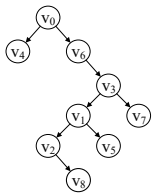
Dependence Tree. Dependence tree is first proposed by KickStarter to support active value dependence tracking and calculate new safe approximate values for vertices affected by edge deletions. It relies on an important characteristic of monotonic algorithms that each vertex's state is dependent on a single in-neighbor. Take the graph in Fig. 1(a) as an example, we conduct the *Single Source Shortest Path* (SSSP) algorithm on this graph with v_0 being the source vertex. After static evaluation, each vertex will reach a convergence state corresponding to its shortest path originating from the source vertex. Every shortest path indicates the dependency between vertices, which can be represented as dependence trees shown in Fig. 1(b). The initial levels of all the vertices are set to be maximum and the levels of roots are 0. Each node on the trees with level l is dependent on its direct parent with level $l-1$ and transitive parents whose levels are less than $l-1$. If a vertex is not reachable from the source vertex, its level will remain at the initial level. Dependence trees are currently only used in processing edge deletions. If a vertex v is affected by edge deletion, we find an edge subset by selecting every incoming edge (u, v) where $level(u) <= level(v)$. Then we re-execute the update function on these edges to obtain a safe approximate value for vertex v and restart the algorithm. More details can be referred in KickStarter [6].

B. Problems and Motivation

Despite a large number of methods proposed to optimize streaming graph analytics, existing systems [1], [3], [5], [7], [8], [11] still suffer from low convergence speed. Specifically, they usually process vertices in a random order to ensure high parallelism and load balance among multiple threads/workers, which will also bring extensive redundant computations. About 90% of vertex state updates are useless in state-of-the-art software systems [19], i.e., KickStarter [6], GraphBolt [7], DZiG [8]. When updates arrive, directly affected vertices are set to be active and need to propagate their new states



(a) An example for edge additions



(b) The new dependence trees

vertex	v0	v1	v2	v3	v4	v5	v6	v7	v8
old levels	0	2	4	1	1	3	1	2	5
new levels	0	3	4	2	1	3	1	3	5

(c) The changes of vertices' levels in the dependence trees

Fig. 2. An example to show edge additions and how the dependence trees and vertices' levels evolve

to neighbors. However, due to the dependencies between vertices, those who are active and located in the latter part of a dependence chain will receive state propagation from the former vertices and usually have to conduct update and propagation multiple times. Considering graph updates in Fig. 2(a), two new edges $v_6 \rightarrow v_3$ and $v_1 \rightarrow v_2$ are added, their destination vertices v_3 and v_2 are updated and set to be active. If we process them in the order of vertex id, v_2 will first propagate its state to v_8, followed by v_3 to v_1. Then v_1 will update itself and propagate new state to v_2 and v_8 again.

To address the above issue, we observe that this type of computations can be eliminated by changing the processing order of vertices to a top-to-bottom order according to their levels on the dependence trees. In the example shown in Fig. 2(a), we can cancel the multiple updates of v_2 if we process v_3 before v_2, because v_3 has a smaller level than v_2 on the dependence trees. However, there are several challenges to achieve this approach. First, the levels of affected vertices may be constantly changing when the process of computation goes on. Therefore, utilizing the dependence trees of last snapshot may not work well. Second, how to guarantee parallelism when the processing order is serialized. Third, how to achieve this goal with minimum overhead. We will discuss the solutions to these challenges in section III.

III. Overview of Our Approach

With the observations mentioned above, we further analyze the dependence trees when graph changes and then present our proposed approach for high-performance streaming graph processing. The main idea of our approach is to reduce redundant computations in monotonic algorithms by regularizing the processing order of all affected vertices so that multiple state propagation along dependence chains can be coalesced.

A. Analysis of Evolving Dependency

Generally a batch of updates usually include edge deletions and additions. When edge deletions arrive, all vertices that are directly affected are first reset to initial value and their new approximate values will be calculated using dependence trees. Then these vertices reset the states of their neighbors, which will be treated as active ones in the next round of computation. Active vertices will be recalculated in the same way until no new active vertices appear. As for edge additions, the matters are easier. Directly affected vertices will be set to be active and propagate their new states to their successor neighbors as incremental computation does. Next we will discuss how the levels of all these affected vertices evolve in these two scenarios. We only focus on the case where vertex level will change. It means that adding an edge will surely make the destination vertex dependent on the source vertex, and also deleting an edge will remove the existing dependency between the two vertices.

Edge Additions. When processing edge additions, we first update directly affected vertices, including their values and levels on the dependence trees. Then these active vertices in the first round of computation will update their neighbors' values and levels. Fig. 2(a) provides an example of edge additions. Two new edges $v_6 \rightarrow v_3$ and $v_1 \rightarrow v_2$ are added to the graph in Fig. 1(a), making v_3 and v_2 active and the level of v_3 change from 1 to 2. v_3 will further update the levels of v_1 and v_7 to 3. The final dependence trees and changes of each vertex's level are shown in Fig. 2(b) and Fig. 2(c). We make several observations from this process. First, if an active vertex u updates its neighbor v, then $level(v)$ will definitely be set to $level(u) + 1$. Second, if the source and destination vertices belong to the same sub-tree in the last dependence trees, their dependency will remain the same in the new trees. Third, an active vertex may turn new vertices on the other sub-trees into its children whose levels are smaller than its level, but those who are originally dependent on it will remain the same.

Edge Deletions. As for deletions, each vertex affected directly will first be reset to initial value and then acquire an approximate value and a new level using dependence trees. After a vertex is set to be active, its neighbors that are dependent on it will also be reset and processed in the same way. Assuming that the two edges $v_6 \rightarrow v_3$ and $v_1 \rightarrow v_2$ have already been added in Fig. 2(a), we delete them and recover the dependence trees to the previous ones. After processing, v_2 and v_3 will become children of v_5 and v_0 respectively. The final tree is the same with Fig. 1(b) and the old levels and new levels in Fig. 2(c) exchange. We also gain several observations. First, the problem of redundant computation of edge deletions is not as severe as that of edge additions. Since in most cases, deletion of an edge will block the propagation chain, the vertices before the deleted edge on the chain will not cause multiple times of state update. Second, level changes are more unpredictable in deletions. Active vertices are prone to find new parents on other sub-trees, but their new parents are not known until their computations complete. This is different from additions in which destination vertices are bound to be dependent on source vertices and their original children remain dependent on them.

B. Accelerate Incremental Computation via Dependence Tree

The basic idea of our approach is to process affected vertices in the order of top-to-bottom based on their levels on the dependence trees. A standard way is to resort all vertices

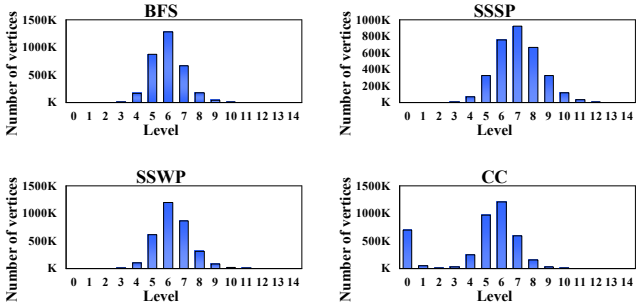


Fig. 3. Distribution of vertex levels

Algorithm 1 ACGraph execution flow

Input: Graph $G = (V, E)$, Active vertex set V_a, Vertex level L

```
1: level ← 0
2: while V_a ≠ ∅ do
3:     parallel_for each vertex v_i ∈ V do
4:         if v_i ∈ V_a and L(v_i) == level do
5:             process(v_i)
6:         end if
7:     end parallel_for
8:     level = level + 1
9: end while
```

based on their levels on the previous trees and process one by one. However, resorting introduces high overhead and the information of levels are outdated with no parallelism guaranteed. Hence we have two main challenges. First, how to ensure high parallelism with minimum overhead if we regularize the processing order of vertices. Second, what we have are only the dependence trees generated in the last graph snapshot, which will change a lot after applying new updates. It means that we cannot adopt the previous trees directly. Thus, we need to consider the change rules of vertices' levels to develop fitting solutions.

For the first challenge, without loss of generality, we analyse the distribution of vertex levels of dependence trees for four monotonic algorithms running on LiveJournal graph, and the results are shown in Fig. 3. We can see that most levels have more than 10K vertices, which means generally there are enough vertices to guarantee parallelism in the same level. Based on this, we adopt a new execution flow combining both serial and parallel modes to process each level in the algorithms, which is given in Algorithm 1. Specifically, we process each level serially and all vertices in the same level parallelly. The level processed increases by one each round (Line 8), and within each round, we parallelly process each vertex with the same level (Line 3). If a vertex is active and its level is the same as current level (Line 4), then update and propagation operations are applied to this vertex (Line 5). Following this strategy, the parallelism can be ensured and no extra overhead will be brought as we do not need any preprocessing and only utilize an integer to represent the current level.

For the second challenge, we propose two different execution approaches for edge additions and deletions respectively.

Edge Additions. The execution flow for edge additions is

Algorithm 2 Execution flow of edge additions

Input: Graph $G = (V, E)$, Active vertex set V_a, Vertex level L, Parents T

```
1: level ← 0
2: while V_a ! = ∅ do
3:     parallel_for each vertex v_i ∈ V do
4:         u_i ← T(v_i)
5:         if L(v_i) ≠ L(u_i) + 1 then
6:             L(v_i) ← L(u_i) + 1
7:         if v_i ∈ V_a and L(v_i) == level then
8:             processAddition(v_i)
9:         end if
10:    end parallel_for
11:    level = level + 1
12: end while
```

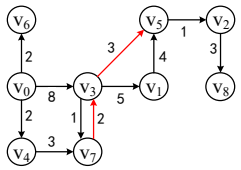


Fig. 4. Showing the scenario where the level of destination vertex gets bigger after edge addition: $v_3 \rightarrow v_5$ and $v_7 \rightarrow v_3$ are two newly added edges. v_5 has a smaller level than v_3, but process v_5 first will cause the sub-tree rooted at v_5 be processed twice.

shown in Algorithm 2. Assuming an edge $u \rightarrow v$ is added, we need to consider two main scenarios. First, adding this edge causes the level of v to be smaller than its previous level. In this case, all vertices of the sub-tree rooted at v and new vertices joining from other sub-trees will be updated eventually and obtain new levels as analysed above, but all these vertices' new levels will be no smaller than u, so changes of the sub-tree and the impact of $u \rightarrow v$ will be coalesced. Thus, when processing each level, we can simply compare each vertex's new level with current level (Line 7) and process those vertices whose levels are equal (Line 8). Due to the observation that every vertex v updated by its in-neighbor u will get a level of $level(u) + 1$, every newly activated vertex will definitely be processed in the next round, which guarantees that each level only needs to be traversed once and no new active vertices will be missed.

Second, the level of v becomes bigger than before after edge addition. This only happens when the two vertices of an added edge (i.e. u and v) belong to different sub-trees and may cause extra computations. Considering the edge additions in Fig. 4, the level of v_5 will be updated from 3 to 2, while level of v_3 being updated from 1 to 3. v_5 has a smaller level so it will be processed before v_3, but it will be processed twice after v_3 propagates its new state to v_5. This kind of redundant computation cannot be eliminated like the previous case. To address this issue, we adopt a level adjustment strategy performed before vertex processing. In each round, for each vertex v we first obtain its parent u on the dependence trees (Line 4). If $level(v) \neq level(u) + 1$, it means that u is an affected vertex and v will also be activated eventually (Line 5). We then adjust $level(v)$ to $level(u) + 1$ either it is active or not to indicate that v is also affected (Line 6). After adjustment, sub-tree rooted at u will be processed only once because every vertex's level will be adjusted to be bigger than $level(u)$ and will not be processed before u. This step does not produce incorrect result because it is equivalent to

TABLE I
GRAPH USED IN EVALUATION

Datasets	Vertices	Edges	Avg Degree
com-DBLP [20]	317,080	1,049,866	7
web-Google [20]	875,713	5,105,039	12
wiki-topcats [20]	1,791,489	28,511,807	32
com-LiveJournal [20]	3,997,962	34,681,189	17
com-Orkut [20]	3,072,441	117,185,083	76

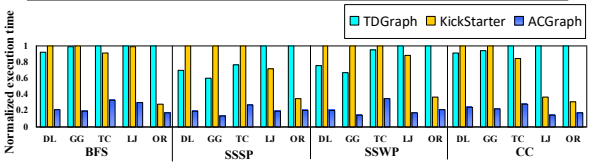


Fig. 5. Total execution time of various solutions

updating the level of v to its final result earlier than its value, which will stay unchanged until v is activated.

Edge Deletions. The process of edge deletions is similar to Algorithm 1. As for edge deletions, we have made the observation that level changes are unpredictable. If an edge $u \rightarrow v$ is deleted, every vertex on the sub-tree rooted at v has to find a new parent or be reset to initial state, and their new parents can be any other vertices. Different from additions, there will be no vertices moved from other sub-trees to this sub-tree, which is an important feature that can be utilized. Hence, it is inappropriate to process vertices based on their new levels on the fly like what we do in edge additions. Instead, we adopt the old dependence trees of the previous graph snapshot to obtain their levels, which has two main benefits. First, all vertices' levels are certain and fixed in the previous trees, so we do not need to track their new levels complicatedly. Second, a deleted edge will only affect the sub-tree rooted at the destination vertex, which means that it will not set a vertex with smaller level to be active. Thus, we also only need to process each level once and no active vertex will be missed. The only difference with the total execution flow in Algorithm 1 is that the input vertex level is changed to the level on the dependence trees of the previous snapshot.

IV. EVALUATION

A. Experimental Setup

We implement ACGraph based on GraphBolt [7], which is the state-of-the-art software system supporting streaming graph processing. GraphBolt supports both monotonic and accumulative graph algorithms, but it is not better than KickStarter in monotonic algorithms. We compare ACGraph with KickStarter [6] and TDGraph [19], which is the cutting-edge hardware accelerator for streaming graph processing. TDGraph also aims at reducing redundant computation overhead of streaming graph processing, but utilizes a different method. We implement a software version of TDGraph for comparison. Our programs were compiled using GCC 7.2 with -O3 flag. All experiments were performed on a 32-core Linux server, which is equipped with Intel Xeon CPU E5-2680 v4 CPU and 256GB of main memory.

Benchmarks. We evaluate ACGraph using four representative monotonic graph algorithms: *Breadth First Search* (BFS) [6], *Single Source Shortest Paths* (SSSP) [6], *Single Source Widest Paths* (SSWP), and *Connected Components* (CC) [6].

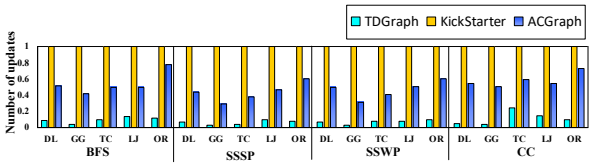


Fig. 6. The normalized number of vertex state updates

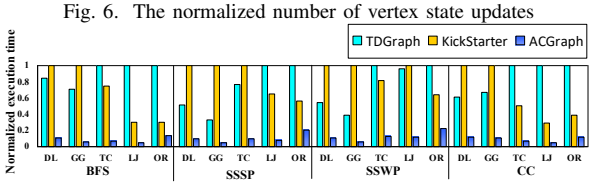


Fig. 7. Execution time for edge additions

Datasets. Table I lists five real-world graph datasets [20] used in our experiments. Like previous systems, we first load 50% of the edges to generate an initial fixed point, and the rest 50% edges are treated as edge additions that are streamed in. Meanwhile, the deleted edges are randomly selected from the loaded edges. The edge additions and deletions are mixed to generate streaming batches together.

B. Performance Comparison

Overall Performance. We compare the execution times of different methods. Fig. 5 shows the normalized execution times for ACGraph, KickStarter, and TDGraph on each dataset respectively. The result shows that ACGraph outperforms both KickStarter and TDGraph by $1.75 \sim 7.43\times$ and $2.85 \sim 7.03\times$, respectively. TDGraph is not much better than KickStarter and even has worse performance in some datasets. The reason is that TDGraph suffers from high runtime cost when implemented in software, especially when graph is large. By contrast, ACGraph reduces redundant computations using dependence trees with barely no overhead introduced, and can achieve higher performance in software.

Number of Updates. We evaluate the total number of vertex state updates, and the results are shown in Fig. 6. The number of updates performed by ACGraph is 50% of that by KickStarter on average. TDGraph has the minimum number of updates because it adopts the topology information to synchronize computations so that every vertex must wait until all the propagation from its neighbors arrives, which guarantees that there are barely no extra computations. However, it requires an expensive preprocessing to achieve this. Fig. 5 shows that the execution time of TDGraph is still long despite its minimum number of updates. By contrast, ACGraph cannot completely eliminate all redundant computations. However, the advantages of ACGraph are that it only utilizes the dependence trees existing in original computation without any expensive preprocessing, and its execution flow can ensure high parallelism. By this means, ACGraph achieves much higher acceleration compared with TDGraph.

Performance for Edge Additions and Deletions. We analyse the performance for additions and deletions separately

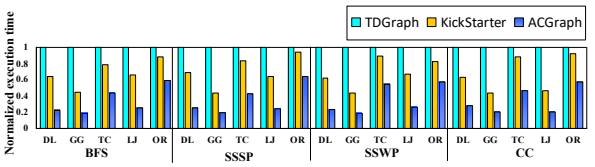
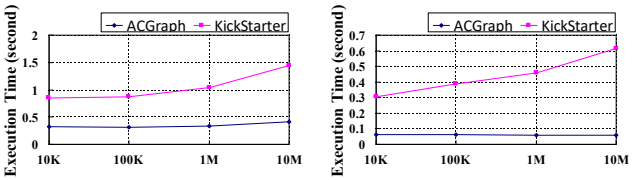


Fig. 8. Execution time for edge deletions



(a) Edge additions and deletions (b) Edge additions only

Fig. 9. Execution time varies with batch size on LiveJournal graph

since they have different overhead for computations. Fig. 7 shows the execution time for edge additions. When updates only consist of edge additions, ACGraph achieves $3.7\sim20.1\times$ speedup over KickStarter, and $5.0\sim22.8\times$ speedup over TDGraph. The result indicates that it has much more advantage when processing edge additions, since edge additions provide more chances for multiple updates to be coalesced. Fig. 8 reveals that ACGraph achieves $1.6\sim2.8\times$ speedup over KickStarter, and $1.7\sim5.3\times$ speedup over TDGraph when processing edge deletions. As we have analysed in Section III-A, the issue of redundant computation in edge deletions is not as severe as in edge additions, so the speedup of ACGraph is not that much as in edge additions. Hence, our approach achieves better performance when facing massive edge additions.

C. Sensitivity Analysis to Batch Size

We conduct performance analysis for batch sizes in two scenarios where edge deletions are included and excluded. Fig.9 reveals the execution times of ACGraph and KickStarter for SSSP running on LiveJournal graph when varying the batch size from 10K to 10M. Since Kickstarter has better performance than TDGraph on LiveJounal as shown in Fig.5, we only compare with KickStarter. The result demonstrates that ACGraph has better stability when batch size changes from small to large, while KickStarter suffers from more severe performance degradation when batch size becomes large. The reason is that ACGraph always process updates in a top-to-bottom order according to vertex levels, so more updates will be coalesced when batch size is large. Fig. 9(b) indicates that ACGraph is more stable and achieves much higher speedup when there are only edge additions. This is consistent with our previous conclusion that ACGraph has better performance when processing edge additions.

V. Conclusion

In this paper, we propose ACGraph, an efficient streaming graph system for monotonic algorithms. ACGraph adopts a runtime approach that maintains dependence trees during runtime, and leverages the dependencies between vertices to normalize the state propagation order, thus multiple states propagated to the same vertex can be coalesced, which greatly reduces redundant computations and achieves high performance in streaming graph processing with barely no overhead introduced. ACGraph can be augmented to existing systems to enable faster processing of graph updates. Experimental results show that our approach reduces the number of vertices updates by 50% on average, and achieves speedup up to $7.43\times$ over state-of-the-art streaming graph systems.

References

[1] 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 EuroSys'12*, pp. 85–98, 2012.

[2] A. Sharma, J. Jiang, P. Bommannavar, B. Larson, and J. Lin, "Graphjet: Real-time content recommendations at twitter," in *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1281–1292, 2016.

[3] S. Rahman, M. Afarin, N. Abu-Ghazaleh, and R. Gupta, "Jetstream: Graph analytics on streaming data with event-driven hardware accelerator," in *Proceedings of MICRO'21*, pp. 1091–1105, 2021.

[4] A. Basak, J. Lin, R. Lorica, X. Xie, Z. Chishti, A. Alameldeen, and Y. Xie, "Saga-bench: Software and hardware characterization of streaming graph analytics workloads," in *Proceedings of ISPASS'20*, pp. 12–23, 2020.

[5] X. Shi, B. Cui, Y. Shao, and Y. Tong, "Tornado: A system for real-time iterative analysis over evolving data," in *Proceedings of SIGMOD'16*, pp. 417–430, 2016.

[6] K. Vora, R. Gupta, and G. Xu, "Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations," in *Proceedings of ASPLOS'21*, pp. 237–251, 2017.

[7] M. Mariappan and K. Vora, "Graphbolt: Dependency-driven synchronous processing of streaming graphs," in *Proceedings of EuroSys'19*, pp. 1–16, 2019.

[8] M. Mariappan, J. Che, and K. Vora, "Dzig: sparsity-aware incremental processing of streaming graphs," in *Proceedings of EuroSys'21*, pp. 83–98, 2021.

[9] F. Sheng, Q. Cao, and J. Yao, "Exploiting buffered updates for fast streaming graph analysis," *IEEE Transactions on Computers*, vol. 70, no. 2, pp. 255–269, 2020.

[10] L. Dhulipala, G. E. Blelloch, and J. Shun, "Low-latency graph streaming using compressed purely-functional trees," in *Proceedings of PLDI'19*, pp. 918–934, 2019.

[11] S. Gong, C. Tian, Q. Yin, W. Yu, Y. Zhang, L. Geng, S. Yu, G. Yu, and J. Zhou, "Automating incremental graph processing with flexible memoization," in *Proceedings of the VLDB Endowment*, vol. 14, no. 9, pp. 1613–1625, 2021.

[12] Y. Zhang, X. Liao, H. Jin, L. He, B. He, H. Liu, and L. Gu, "Depgraph: A dependency-driven accelerator for efficient iterative graph processing," in *Proceedings of HPCA'21*, pp. 371–384, 2021.

[13] Y. Zhang, X. Liao, H. Jin, B. He, H. Liu, and L. Gu, "Digraph: An efficient path-based iterative directed graph processing system on multiple GPUs," in *Proceedings of ASPLOS'19*, pp. 601–614, 2019.

[14] P. Faldu, J. Diamond, and B. Grot, "Domain-specialized cache management for graph analytics," in *Proceedings of HPCA'20*, pp. 234–248, 2020.

[15] H. Jin, P. Yao, and X. Liao, "Towards dataflow based graph processing," *Science China Information Sciences*, vol. 60, pp. 1–3, 2017.

[16] P. Pandey, B. Wheatman, H. Xu, and A. Buluc, "Terrace: A hierarchical graph container for skewed dynamic graphs," in *Proceedings of SIGMOD'21*, pp. 1372–1385, 2021.

[17] A. Basak, Z. Qu, J. Lin, A. R. Alameldeen, Z. Chishti, Y. Ding, and Y. Xie, "Improving streaming graph processing performance using input knowledge," in *Proceedings of MICRO'21*, pp. 1036–1050, 2021.

[18] A. McCrabb, E. Winsor, and V. Bertacco, "Dredge: Dynamic repartitioning during dynamic graph execution," in *Proceedings of DAC'19*, pp. 1–6, 2019.

[19] J. Zhao, Y. Yang, Y. Zhang, X. Liao, L. Gu, L. He, B. He, H. Jin, H. Liu, X. Jiang, and H. Yu, "Tdgraph: a topology-driven accelerator for high-performance streaming graph processing," in *Proceedings of ISCA'22*, pp. 116–129, 2022.

[20] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.