GraphFly: Efficient Asynchronous Streaming Graphs Processing via Dependency-Flow

Dan Chen*, Chuangyi Gui*, Yi Zhang, Hai Jin, Long Zheng§, Yu Huang, Xiaofei Liao
National Engineering Research Center for Big Data Technology and System/Services Computing Technology and System Lab/Clusters and Grid Computing Lab, Huazhong University of Science and Technology, Wuhan, China
{cdhust, chygui, zyyy, hjin, longzh, yuh, xfliao}@hust.edu.cn

***Abstract*—Existing streaming graph processing systems typically adopt two phases of refinement and recomputation to ensure the correctness of the incremental computation. However, severe redundant memory accesses exist due to the unnecessary synchronization among independent edge updates. In this paper, we present GraphFly, a high-performance asynchronous streaming graph processing system based on dependency-flows. GraphFly features three key designs: 1) *Dependency trees* (D-trees), which helps quickly identify independent graph updates with low cost; 2) Dependency-flow based processing model, which exploits the space-time dependent co-scheduling for cache efficiency; 3) Specialized graph data layout, which further reduces memory accesses. We evaluate GraphFly, and the results show that GraphFly significantly outperforms state-of-the-art systems KickStarter and GraphBolt by $5.81\times$ and $1.78\times$ on average, respectively. Also, GraphFly scales well with different sizes of update batch and compute resources.**

***Index Terms*—streaming graphs, incremental processing, redundant memory accesses**

I. Introduction

Graphs are widely used in real life to model complex relationships between entities [1]–[3]. In real scenarios, graph data is continuously changed, called streaming graphs. Streaming graph processing appears to be particularly important for analyzing dynamically changing graphs in real time, for example, financial fraud detection [4] and real-time content recommendations in social networks [5]. Existing streaming graph processing systems such as Tornado [6], Kineograph [7], KickStarter [8], and GraphBolt [9] are proposed to handle fast changing graphs as batches of graph updates arrive. They adopt incremental computation techniques for real-time analysis, reusing prior results to accelerate convergence.

Although incremental computation significantly reduces the response time, directly reusing prior results usually leads to incorrect results. The incorrectness remains when new graph updates arrive. Generating the correct result becomes the focus of state-of-the-art streaming graph processing systems. The prevailing solution is to transform the previous results into a refined intermediate approximation that meets the correct convergence requirements before the computation starts. Incremental recomputation is then applied to the intermediate approximation to produce correct final results. This *refinement-recomputation* progress is common and the core in state-of-the-arts, such as GraphIn [10], KickStarter [8], GraphBolt [9], and DZiG [11]. Existing works have proposed many novel techniques to optimize the *intrinsic* problems of the refinement or recomputation, such as utilizing the dependency of intermediate values to reduce the number of impacted vertices that need to be processed [8], [9], and a hybrid computation model to leverage computation sparsity in different iterations [11].

However, the *refinement-recomputation* execution mode remains oblivious to the redundant memory accesses across two phases. Specifically, a batch of graph updates usually contains millions of edges. After graph updating, the refinement follows the dependency to traverse the graph for identifying and resetting the values of impacted vertices to recoverable approximations. All the edges and values of these impacted vertices are read and written to the memory in the refinement. When performing incremental computations for impacted vertices, these data will be fetched again from memory, resulting in redundant memory accesses.

There exists an opportunity to reuse the data accessed by refinement in recomputation before writing it into memory. However, the impact of different edge updates may influence each other and thus existing systems take a conservative strategy by inserting a synchronization barrier on all graph updates between two phases. Therefore, recomputation will only start after all the edge updates in a batch have been refined. This means, even two edge updates are totally independent, i.e., there is no overlap between the sets of vertices impacted by these two edge updates, the refinement and recomputation are still separated and thus both phases require access to the same data from memory and cannot enjoy the benefits of reusing data in the cache. Besides, due to the irregularity of graph structure, memory accesses in both phases are random and unpredictable, which limits the performance.

We observe that the impacted vertices spread their influence through the outgoing edges, which forms a dependency-flow. The incremental changes of impacted vertices in both refinement and recomputation are propagated in certain directions of flow, which usually affects only a small fraction of vertices. Processing the update streams guided by dependency-flows can embrace high parallelism and memory efficiency. Edge updates in independent dependency-flows can be concurrently processed without synchronization. Edge updates falling into the same dependency-flow can reuse data in the cache, which reduces the redundant random memory accesses. However, it is

*Dan Chen and Chuangyi Gui contributed equally to this work.
§Long Zheng is the corresponding author of this paper.

only after the refinement phase we can extract the dependency-flows precisely, which does not help us to reuse the data in refinement and recomputation. How to identify the dependency-flows before refinement is the primary difficulty. Maintaining both correctness and efficiency is another challenge.

In this paper, we present GraphFly, a novel runtime technique that breaks the unnecessary synchronization barrier for independent edge updates between the refinement and recomputation phases to eliminate redundant memory accesses. The core of GraphFly is to dynamically capture and adjust the dependency-flow as the graph is updated during runtime. Specifically, GraphFly is characterized by the following key designs. First, by exploring the topological features of graphs, we propose a decomposition method to reorganize graphs into *D-trees* based on the concept of elimination trees [12]–[14], which helps to quickly identify the dependency-flows of the graph before refinement. The dependency-flow can be easily expressed by D-trees and also D-trees can be easily maintained incrementally as graph changes. Second, we develop a space-time dependent co-scheduling model to divide the oversized dependency-flow. For graphs stored as matrices, we derive dependency-flows by the dependency of the lower triangular matrix while managing the execution order of dependency-flows using the upper triangular matrix. Third, we design a specialized data layout for storing the dependency-flows by putting the vertices and edges inside a dependency-flow together as blocks. Such data layout provides high memory access efficiency and improves the effectiveness of cache for processing the dependency-flows. We implement a prototype of GraphFly and achieve significant performance improvement over state-of-the-art streaming graph processing systems.

The key contributions of this paper are as follows:

- It introduces the dependency-flow to break the synchronization between refinement and recomputation for reducing redundant memory accesses in streaming graph processing.
- It proposes D-trees based graph decomposition and execution strategies, overcoming the challenges of applying the concept of dependency-flows.
- It develops GraphFly, a high-performance stream graph processing system that eliminates redundant memory accesses for various algorithms. Experimental results show that GraphFly outperforms KickStarter and GraphBolt by 5.81$\times$ and 1.78 $\times$, respectively.

II. Background and Motivation

This section reviews streaming graph processing models and discusses the key observations motivating our approach.

A. Streaming Graph Processing

A streaming graph G, as shown in Fig. 1, is continuously updated by changing the structural connections or modifying values on vertices and edges. Streaming graph processing aims to maintain the latest query results while graph updates keep arriving. Since vertex updates can be transformed into a series of edge updates, in this paper, we focus only on edge additions and deletions. For example, a vertex deletion can be understood as deleting all the edges containing this vertex. A vertex addition can be modeled by adding the first edge of this vertex. In existing streaming graph processing systems [8]–[11], these graph updates are batched for evaluation (denoted as Δ_1 and Δ_2 in Fig. 1). Before receiving updates, initial graph G_0 is fully evaluated as in the static graph processing to generate converged results R_0. Each batch of updates is then applied and evaluated to get new converged results (e.g., R_1 and R_2 in Fig. 1). Instead of starting from scratch, which is wasteful due to large amounts of unnecessary computations, existing systems reuse previous results to incrementally compute the changes based on the algorithmic features of different algorithms. Previous results can be understood as a kind of intermediate states during the progress to convergence. This process repeats when new updates arrive.

Despite the efficiency of incremental computation, maintaining correctness of the incremental results is challenging. Not all intermediate states lead to correct results, refinement must be applied to previous results to obtain a recoverable approximation (e.g., $Refine(R_0)$ and $Refine(R_1)$ in Fig. 1). This can be illustrated through the SSSP example in Fig. 2. Before the deletion of edge Ⓐ→Ⓓ, the lengths of shortest paths from vertex Ⓐ to other vertices are computed by updating the vertices values when these vertices receive a shorter path than their current states. Convergence results recorded as initial R_0 in Fig. 2(b). The expected values after the edge deletion are presented in Fig. 2(c). After the edge deletion, directly reusing the values in R_0 to compute the new lengths will result in no changes, because the old values already converged in the updated graph. One solution is to refine the affected vertices to their initial values. However, simply resetting vertices on updated edges is not enough in this example, only resetting Ⓓ still leads to wrong results because the impact is passed to other vertices. In order to solve the problem, all impacted

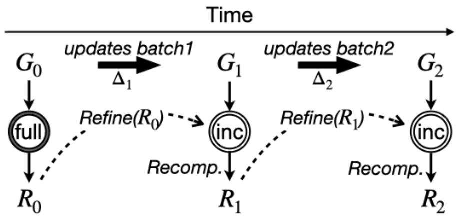

Fig. 1. The conception of incremental streaming graph processing. Batched graph updates keep arriving. Initial graph is computed as static processing, and later graph updates are processed incrementally by reusing previous results.

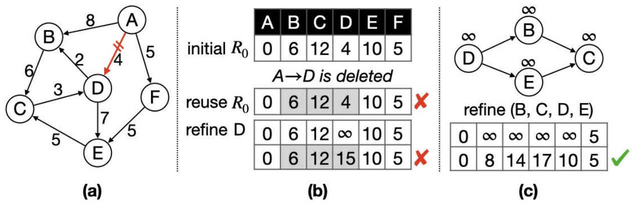

Fig. 2. Example of incremental processing: a case study of SSSP. (a) Initial graph with a deleted edge. (b) Directly reusing the results or simply resetting the vertex on updated edge both result in wrong results. (c) Detecting and refining possible impacted vertices lead to correctness.

vertices are correctly detected first in existing systems, i.e., {Ⓑ, Ⓒ, Ⓓ, Ⓔ} in Fig. 2(c), and then they are refined based on algorithmic properties to guarantee correctness.

This work incorporates existing refinement solution of identifying necessarily impacted vertices for correctness, while enabling high efficiency on concurrent processing of edge updates. The proposed techniques may also be integrated into existing systems to improve the performance.

B. *Problem: Redundant Memory Accesses*

While incremental refinement and recomputation guarantee the correctness, redundant memory accesses between these two phases limit the performance in existing systems. The redundancies come from two folds: 1) vertex values after refinement have to be written into memory and then fetched again for recomputation on the same impacted vertices, 2) the edges of these impacted vertices are first traversed to identify impacts in refinement and then accessed again to propagate computed values in recomputation.

This problem can be detailed in Fig. 3(a) and Fig. 3(b). In this example, multiple edge deletions are applied to the initial graph (i.e., deleting edge Ⓐ→Ⓑ and Ⓗ→Ⓘ). For deleting Ⓐ→Ⓑ, consider the global vertex values are stored continuously in the memory. During the refinement, the edges of vertex B are accessed to identify impacted vertices by Ⓑ, i.e., Ⓑ→Ⓒ and Ⓑ→Ⓓ. At the same time, the value of Ⓑ is loaded from memory and adjusted. The refine function may determine that Ⓓ is affected by Ⓑ, and then continue on Ⓓ following the outgoing edges, Ⓓ→Ⓕ and Ⓕ→Ⓖ, until vertex Ⓖ is refined. Latest refined values of {Ⓑ, Ⓓ, Ⓕ, Ⓖ} are currently in the local space and will be written back to the global vertex values before recomputation. In the recomputation phase, in order to compute these impacted vertices, the edges traversed in the refinement phase and the latest values of {Ⓑ, Ⓓ, Ⓕ, Ⓖ} are requested repeatedly. In fact, these data related to one graph update can be directly reused by recomputation before written into memory. Existing systems take a conservative strategy by synchronizing between two phases, losing the opportunity to remove redundancies. As shown in Fig. 3(b), the refinements of Ⓐ→Ⓑ and Ⓗ→Ⓘ are synchronized. Besides, the vertex values are scattered at different locations, repeated memory accesses add more pressure on memory bandwidth. We profiled the impact of redundant memory accesses in GraphBolt as shown in Fig. 4(a). As we can see, redundant memory accesses consume $> 68\%$ of the running time on average in two phases.

Existing works optimize the refinement phase of an edge update by either utilizing heuristics to reduce the number of necessary vertices that need to be refined and thus the recomputations, or providing better refined value on impacted vertices to accelerate the convergence. However, the inherent redundant memory access across refinement and recomputation still exists. In this work, we aim to solve the redundant memory access problem by exploring the opportunities to break the barrier of refinement and recomputation among concurrent edge updates. Existing optimizations inside refinement phase are orthogonal to our focus.

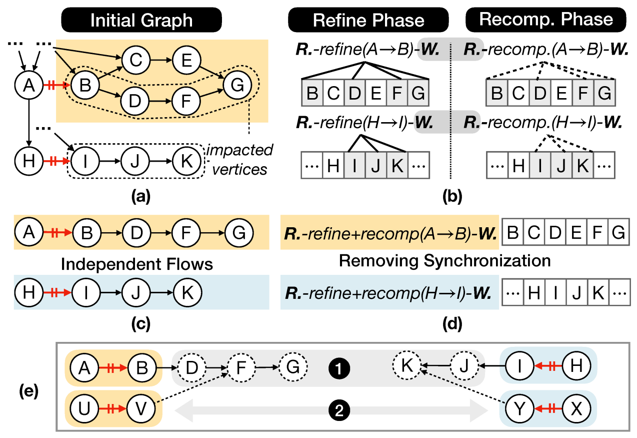

Fig. 3. A motivation example of redundant memory accesses. (a) Two edges are deleted from the initial graph. (b) Synchronization between two phases causes the vertex values of impacted vertices repeatedly read from and written into memory. (c) Impacted vertices of two deleted edges fall into independency-flows, and two edges can thus be processed concurrently. (d) By removing the synchronization of two deletions, the values of impacted vertices are reused in two phases before being written into memory. (e) Challenges of processing batched updates: ❶ Identifying independent updates A→B and H→I without traversal on {D, F, G} and {J, K} during runtime is difficult; ❷ Impacted vertices induced from two updates may overlap, e.g., A→B and U→V both influence vertex F and G.

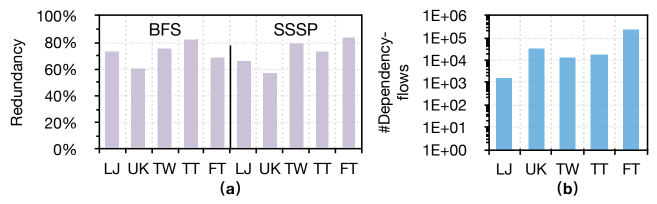

Fig. 4. (a) The execution time ratio of redundant memory accesses by deleting 100K edges for each batch, and (b) The number of dependency-flows for different graphs

C. *Insights: Dependency-Flow*

The barrier between the refinement and recomputation is due to the possible dependencies among edge updates. If we can identify the edges that are independent of each other, then the barrier among those edges can be removed. We achieve our goal based on the following observation.

Observation. The incremental updates of a vertex impact other vertices via certain directions in a dependency-flow, which usually consists of a small fraction of vertices. Given a graph update, the impacted vertex influences other vertices following outgoing edges, all possible impacted vertices and their connected edges form a dependency-flow. As shown in Fig. 3(a), the shaded part of the graph represents a dependency-flow starting from impacted vertex Ⓑ. Vertices on this dependency-flow are possibly affected by Ⓑ. The dependency-flow induced by deleted edge Ⓗ→Ⓘ includes {Ⓘ, Ⓙ, Ⓚ}. If the value on Ⓑ increases then the change will also cause the values on Ⓓ, Ⓕ, and Ⓖ to increase. Once a dependency-flow is identified, we can safely operate on inside vertices using the local information of the flow. Fig. 4(b) counts the number of dependency-flows for different graphs. We observe many dependency-flows ranging from 1,496 to

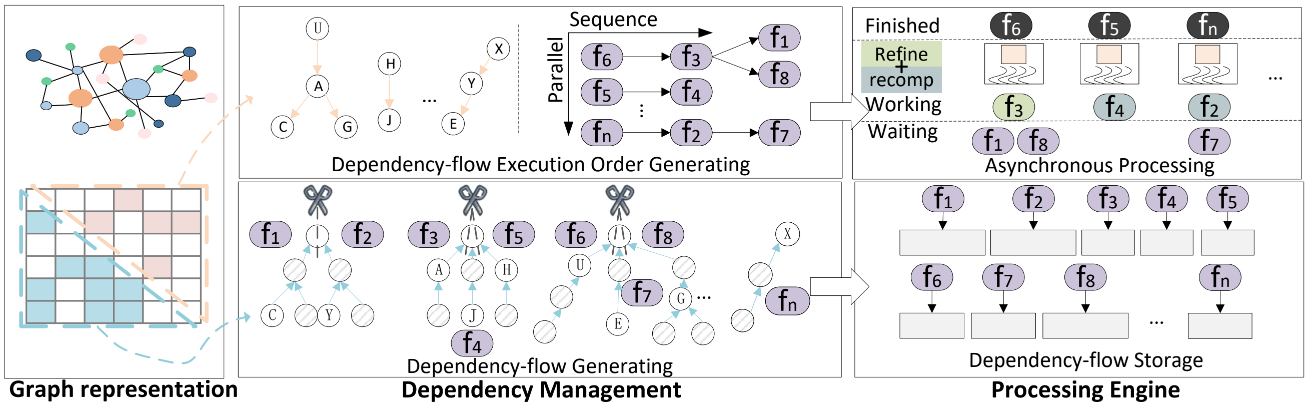

Fig. 5. The overview of GraphFly

211,348 according to different graphs, showing significant optimization opportunities.

Based on the observation, we can partition the graph into dependency-flows and put the vertex values and structural information of a flow together. Each dependency-flow can be loaded into the cache to avoid frequent memory accesses. If edge updates are distributed into different flows, then they can be totally processed in parallel and thus no synchronization is needed. If two edge updates fall into the same dependency-flow, then the information required can be found directly in the cache even though they rely on each other to recompute. As shown in Fig. 3 (c), the influenced paths consisting of impacted vertices over Ⓐ→Ⓑ and Ⓗ→Ⓘ are independent of each other. There is no need to synchronize the refine phase for two edges, and the recomputation can start immediately after the refinement before writing the results into memory (as shown in Fig. 3(d)). All the necessary information for computation can be fetched during the traversal for refinement, thus the amount of memory accesses is largely reduced.

Challenges. Although the dependency-flow provides opportunities to optimize redundant memory accesses, there are still several challenges to be solved. First, as the graph changes, the dependency-flow composed of vertices also changes. Generally, we need to regenerate the dependency-flows; however, this will limit the performance gained by reducing redundant memory accesses with breaking the two-stage synchronization of refinement and recomputation. How to incrementally maintain the dependency-flow as the graph structure changes is important. Second, due to the strongly connected vertices in the graph, there exists oversized dependency-flows, limiting the performance gain brought by asynchronous processing. A finer-grained execution must be considered by parallelizing the sub-flows of the oversized dependency-flow. However, correctness must be maintained, e.g., Ⓧ→Ⓨ has impacts on Ⓚ in the sub-flows of Ⓗ→Ⓘ as shown in Fig. 3(e). In such cases, the connections of different sub-flows must be considered to maintain correctness. In this work, GraphFly elegantly handles above challenges via an extension of the elimination trees, called D-trees, and efficient asynchronous processing. The two challenges are solved in Section IV and Section V respectively.

III. Overview

Based on our insights, we propose the dependency-flow based processing method for streaming graphs, called GraphFly. Given an arbitrary graph and a batch of streaming updates, GraphFly can fast exploit the relations of these updates via dependency-flows and asynchronously execute the refinement and recomputation to reduce redundant memory accesses. We realize above goals with two main modules, as shown in Fig. 5.

Dependency Management. This part first uses a concept of *D-trees* to generate the dependency-flows, and then organizes these flows in efficient space-time co-scheduling orders. Consider an initial graph viewed as a matrix.

• Generating dependency-flows. The theory of *elimination trees*, which is the basis of D-trees, helps fast represent the dependencies of vertices inside the lower triangular matrix into forests. Dependency-flows are extracted by dividing these D-trees at root vertices (i.e., the initial graph is divided as f_1 to f_n). Two vertices on opposite flows are considered independent of each other.

• Scheduling dependency-flows. Flows extracted from the lower triangular matrix actually present the dependency of vertices in structural space. In order to maintain correctness, the flows must be executed in certain orders during runtime, which is constrained by upper triangular matrix. In the example of Fig. 5, there is Ⓧ→Ⓨ→Ⓔ, such that flow f_n is executed before f_2 and f_7. These space-time dependent co-scheduling orders are then fed to processing engine to execute.

Processing Engine. Underlying the efficiency of GraphFly is the locality of the specialized storage management and highly parallel asynchronous processing model via the dependency-flow.

• Dependency-flow storage management. When accessing the vertex data in a dependency-flow, these vertex data may be distributed in different memory regions. In order to improve the effectiveness of memory access, we design a storage format for the dependency-flow. Let the vertices data in the dependency-flow be stored closely in memory, allowing to merge memory accesses and improving memory access effectiveness.

• Asynchronous processing model. A batch of streaming updates may be scattered into different dependency-flows.

GraphFly loads dependency-flows according to the scheduling orders and executes them asynchronously. At the same time, different flows may work in different phases, e.g., f_3 is in refinement phase while f_2 and f_4 are in recomputation phase. In this way, synchronization among all update edges is relaxed.

IV. Constructing Dependency-Flow via D-trees

This section first introduces the definition of D-trees derived from the theory of elimination trees, and then describes the procedure for efficiently generating and maintaining D-trees.

A. Definition of D-trees

The D-tree proposed in this work is based on the theory of elimination tree, which is widely used for exploring parallelisms in the area of high performance data analytics [12]–[14], e.g., LU factorization of unsymmetric matrices. D-trees inherits the advantages of the elimination trees while overcoming its limitations when applied to arbitrary graphs.

The Foundation of D-trees. Let L be the lower triangular matrix $n \times n$. The elimination tree $T(L)$ of L is defined as a tree with n nodes $\{1, ..., n\}$, where (i, k) is an edge iff.

$$k = min\{r > i | l_{ri} \neq 0\} \tag{1}$$

l_{ri} denotes the non-zero entry in the i-th row and r-th column of L, and L must satisfy the following condition:

CONDITION 1: If $i < j < k$ and $l_{ji}, l_{ki} \neq 0$, then $l_{kj} \neq 0$.

Fig. 6(a) presents a standard lower triangular matrix L and it satisfies *CONDITION 1*. Fig. 6 (b) shows its directed graph. According to the procedure in Eq. (1), we can generate the elimination tree $T(L)$ for the matrix L as shown in Fig. 6(c).

In fact, the elimination tree should be the forest, because there may be more than one root. But for simplicity in expression, it is uniformly called tree in this paper. Elimination tree has the following property [15]:

PROPERTY 1: Let x and y be children of z in T. SubT(x) and SubT(y) are the set of vertices in the subtree with x and y as root, respectively. Then, there does not exist any edge in $G(L)$ between SubT(x) and SubT(y).

Generally, multiple elimination trees generated for a lower triangular matrix are independent of each other. An elimination tree can be regarded as dependency-flows natively, e.g., elimination trees with root vertices ⑦ and ⑧ can be treated as two dependency-flows in Fig. 6(c). Moreover, according to *PROPERTY 1*, sub-trees cut from the root vertex of an elimination tree can also represent different dependency-flows ending with the same vertex. Consider the example in Fig. 6(c). ⑦ is the root of the directed tree and also the parent of ② and ④. Hence, there is no edge between SubT(②) and SubT(④) (*PROPERTY 1*). Cutting on ⑦, we can obtain two dependency-flows ⓪ $\rightarrow$ ① $\rightarrow$ ② $\rightarrow$ ⑦ and ③ $\rightarrow$ ④ $\rightarrow$ ⑦ which together affect only ⑦.

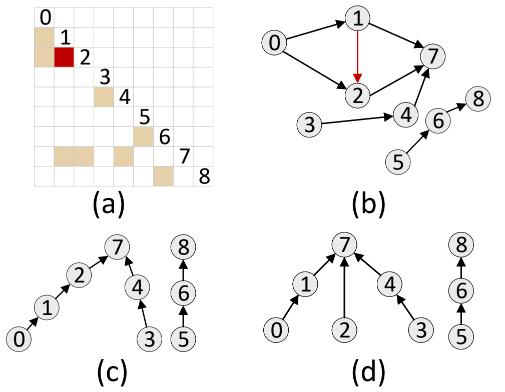

Fig. 6. An example of the elimination tree: (a) A lower triangular matrix L. (b) The directed Graph $G(L)$ of L. (c) The elimination tree $T(L)$ of L. (d) The elimination tree generated by directed graph after deleting 1→2

If a graph satisfies *CONDITION 1*, we can directly use elimination tree as in Eq. (1) to extract dependency-flows. However, due to the irregularity of real-world graphs, this condition is usually not satisfied, e.g., Fig. 6(d) shows the elimination tree induced by Eq. (1) after l_{12} is deleted. ⓪ can reach ② while the tree consider them as independent. Besides, Eq. (1) also fails when the graph is asymmetric.

The Characteristics of D-trees. The essence of *CONDITION 1* is that, for each vertex, only one of its edges is chosen, and this edge can reach the destination vertex of the other edges. For any lower triangular matrix, edges not satisfying *CONDITION 1* should be recorded and added to the elimination tree to capture all dependencies. However, this results in the elimination tree becoming a complex graph, and *PROPERTY 1* is destroyed.

Facing arbitrary graphs, in order to preserve the *PROPERTY 1*, D-trees extend the elimination trees by introducing the concept of *hyper vertex*. The intuition here is to group a vertex having multiple parents with its parent vertices as a hyper vertex, ensuring that each has at most one parent. For example, D-trees can group ⓪, ①, and ② into a hyper vertex in Fig. 6(d). If there are no vertices to be merged as a hyper vertex, D-trees degenerate into elimination trees. In other words, the elimination trees is a special case of D-trees.

A hyper vertex in D-trees is a whole part and not separable, i.e., all vertices within a hyper vertex should be put into the same dependency-flow. With the concept of hyper vertex, D-trees can capture the vertex dependency for any lower triangular matrix and still guarantee the *PROPERTY 1* of elimination trees, i.e., there is no dependency between the child vertices of any vertex.

B. Generating D-trees

Checking for each vertex whether the outgoing edges match *CONDITION 1* is time-consuming, when creating D-trees. For efficiently generating D-trees, we rely on the insight that the outgoing neighbors of the vertices are always partitioned

Algorithm 1: D-tree Generation and Maintenance

```
Input: Input graph – $G = (V, E)$
Output: D-trees – $T$
1  Function DtreeGeneration($G$):
2      for $vertex\ v$ do
3          for $u\ in\ v.outEdges$ do
4              $AddToDtrees(T, v, u)$
5          if $v.out\_degree > 1$ then
6              mergeHyperVertexInDTree ($T, v$)
   /* Accumulative algorithms                     */
7  Function edgeAddition($e$):
8      AddToDtrees($T, e$)
9      CheckMergeHyperVertex($T, e$)
   /* Accumulative algorithms                     */
10 Function edgeDeletion($e$):
11     DeleteEdgeInDtrees($T, e$)
12     CheckSeparateHyperVertex($T, e$)
   /* Selective algorithms                        */
13 Function edgeAddition($e$):
14     if $e\ is\ keyEdge$ then
15         AddToDtrees($T, e$)
   /* Selective algorithms                        */
16 Function edgeDeletion($e$):
17     if $e\ is\ keyEdge$ then
18         DeleteEdgeInDtrees($T, e$)
```

into the same dependency-flow, regardless of whether *CONDITION 1* is satisfied. If the outgoing edge meets condition one, the vertex and its neighbors will be in the same branch of the D-tree and partitioned into the same dependency-flow. Otherwise, they are packaged into hyper vertex and these vertices are also partitioned into the same dependency-flow due to the inseparability of the hyper vertex. Therefore, we can directly package the vertex and its neighbors to hyper vertex regardless of whether they meet the *CONDITION 1.*

Algorithm 1 implements the procedure of generating D-trees for the lower triangular matrix. Rows are considered as source vertices, columns are considered as destination vertices. Given a vertex, it traverses all outgoing edge paths of the vertex and adds all outgoing edge paths to D-trees (Lines 3-4). Then it checks whether this vertex needs to be merged as hyper vertex (Lines 5-6). If the vertex contains multiple parents (i.e., contains multiple outgoing edges), D-trees merges this vertex and its parent vertices into a hyper vertex. The complexity of the algorithm is $O(N + E)$. More importantly, our approach can be maintained incrementally as the graph mutates, and the time complexity for a deleted or incremental edge is only $O(1)$. The details of incremental maintenance will be described in Section IV-C.

For accumulative algorithms, the vertex state is derived from the neighbors of all its incoming edges. D-trees can be generated directly from graph structure. While for the selective algorithms, the vertex only selects one edge to compute its state. Such selected edges are called key edges, while other edges that are not used to compute vertex values are useless dependencies. If we generate D-trees directly using graph structure, the useless dependencies can cause the non-dependent vertices to lie in a dependency-flow. Hence, we track key edges to generate D-trees for selective algorithms. We record these key edges during the runtime like Kick-Starter [8] and then use them for the next batch updates.

Note that D-trees are generated for a triangular matrix. However, graphs are usually asymmetric and sparse such that they can not be used directly to generate D-trees. In this work, we separate a sparse asymmetric matrix into a lower triangular matrix and an upper triangular matrix, such that Algorithm 1 can be used to generate D-trees for upper and lower triangular matrices, respectively.

Although the directed acyclic graph can be turned into a lower triangle by the topological sort [16], streaming graphs are often continuously changed. The sorting requires frequent reordering, incurring non-negligible overheads. Also, real-world graphs are often non-acyclic. Therefore, the sorting can be ineffective in many cases, particularly for real scenarios.

C. Dynamically Maintaining D-trees

If a graph change occurs in the lower triangular part of the matrix, we only need to maintain the lower triangular D-tree, and vice versa. For two types of algorithms, the maintaining procedure is demonstrated in Algorithm 1.

Accumulative Algorithms. After receiving a batch of edge updates, for edge additions, we first add the edge to D-trees and then check whether the vertex needs to be merged into a hyper vertex (Lines 7-9). We remove the edge from D-trees for edge deletion and then check whether the vertex needs to be separated from the hyper vertex (Lines 10-12).

If an edge addition leads a connected vertex to have more than one parents on the D-tree, all parents and this connected vertex will be merged into a hyper-vertex. For example, consider an edge $a \to b$ in a D-tree. For the case of adding an edge $a \to c$, the connected vertex a of this added edge will have two parents b and c. Thus, the connected vertex a and its parent vertices b and c will be merged into one hyper-vertex.

If one associated vertex of an edge addition has already been in a hyper-vertex, another one will be incrementally merged into this hyper-vertex. Suppose multiple edge additions compose a complete subgraph. All the associated vertices will be merged into a resulting hyper-vertex. Note that vertices for each edge addition are incrementally added to a hyper-vertex. Therefore, each edge addition affects vertices in a constant magnitude, yielding the $O(1)$ maintenance complexity. The edge deletion is similar.

Selective Algorithms. For the selective algorithm, we generate D-trees only using key edges. Therefore, D-trees need not be modified when the edge additions and deletions are not recorded as key edges. For edge additions and deletions that are key edges, we only need to add the edge (Lines 13-15) or delete the edge (Lines 16-18) in the D-tree. Since each vertex is affected by only one edge, there are no hyper vertices in D-trees. In this case, D-trees are actually elimination trees. The algorithm complexity of adding and deleting an edge is $O(1)$, which results in negligible overhead.

V. Dependency-flow Processing

In this section, we first introduce the space-time dependent co-scheduling execution model to avoid large dependency-flows, and then present a specialized storage format to enable efficient memory access.

A. *Space-Time Dependent Co-Scheduling Model*

A naive way to analyze the dependencies of an asymmetric graph is to add D-trees of the upper triangular matrix to D-trees of the lower triangular matrix, yet causing the D-trees to become a complex graph. This may result in a large dependency-flow that degrades cache efficiency. Consider a worst-case scenario where all vertices are dependent on each other. The entire graph is finally considered as a single dependency-flow. The incremental computation still suffers from frequent memory accesses in two phases.

Space-Time Dependency. We expect the dependency-flow to remain fine-grained and yet correctly break the synchronization between the refinement and recomputation phases. We propose a space-time dependent co-scheduling approach. Specifically, we partition the graph structure into dependency-flows directly by D-trees of the lower triangular matrix, and then control the execution order of these dependency-flows through D-trees generated by the upper triangular matrix. For each edge addition or edge deletion, we can quickly determine the range of vertices it can affect, including vertices that are in other dependency-flows. Specifically, based on the place of edge deletion or addition, we can easily determine the impacted vertices in the dependency-flow. Because these impacted vertices may have outgoing edges to other dependency-flow. We can identify other impacted dependency-flows by D-trees of the upper triangle and generate the impact relationship between them. When determining the impacted dependency-flows, we merge the impacts if there are also edge deletions and additions in the same flow. That is, we process the deletion and addition of edges in the dependency-flow and the impact from other dependency-flows all at once.

Example. As shown in Fig. 7(b), for deleting ④→⑤, ②→⑦, and ⑪→⓪, we can confirm that ④→⑤ may affect ⑥ and ⑨ in the dependency-flow. We can further know other dependency-flows affected through D-trees of the upper triangle. There are no other flows affected by ⑥, while ⑨ affects vertex ⑦ (in dependency-flow f_2). ⑦ becomes the impacted vertex, and we can find such dependency-flows until all affected vertices are identified. Finally we can generate the relationships among affected dependency-flow, i.e., $f_1 \Leftrightarrow f_2 \rightarrow f_3$. Because f_2 also contains edge deletion ②→⑦, we can merge their impacts as shown in Fig. 7(d).

Eventually we can get multiple set of relationships among dependency-flows, i.e., $f_1 \Leftrightarrow f_2 \rightarrow f_3$ and $f_4 \rightarrow f_5$. For dependency-flows that form a cycle, if we break the synchronization between refinement and recomputation for such dependency-flow, it may lead to wrong results. For example,

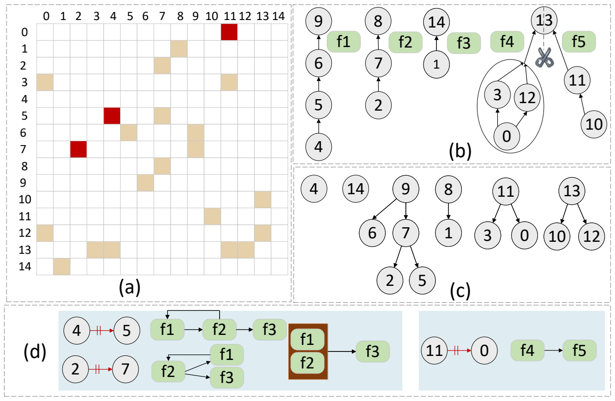

Fig. 7. An example of space-time dependent co-scheduling model: (a) An asymmetric matrix; (b) D-trees for lower triangular matrix; (c) D-trees for upper triangular matrix; (d) Execution order of dependency-flows

there are two dependency-flows that affect each other; if one dependency-flow executes recomputation without waiting for refinement on other flows, some of the vertex values it uses are likely to be reset by the other dependency-flows, which leads to wrong results. Hence we merge such dependency-flows and consider them as a whole dependency-flow, e.g., we consider $f_1 \Leftrightarrow f_2$ as a dependency-flow when deleting ④→⑤. The dependency-flows are executed in order according to their dependencies. This ensures that the refinement of other dependency-flows that affect the dependency-flow has been executed, and thus correctness are guaranteed when asynchronously processing the refinement and recomputation. Different sets of related dependency-flows can be executed in parallel, because they will not affect each other.

Discussion. In the extreme case, the real-world graph may have edges distributed mainly in the upper triangular matrix, with only a few non-zero elements in the lower triangle, or vice versa. Our space-time dependent co-scheduling approach can be well adapted to this situation. We can switch the roles of the upper and lower triangles. D-trees of the upper triangle also can be used to partition the dependency-flow in graph, and the lower triangle controls the order of execution. In addition, GraphFly can also support accelerating undirected graphs, which can often be represented as a directed graph where each edge is decomposed with two directed edges.

For the selective algorithms, when edge deletion occurs, the target vertex of the deleted edge will select an edge from its other edges to compute its vertex state. The source vertices of other edges may be located in other dependency-flows, and we cannot confirm whether the values of source vertices of other dependency-flows are correct. We use pull-style processing for edges within a dependency-flow and push the results for edges between dependency-flow, i.e., we select an edge from within the same dependency-flow as its recovery value.

B. *Storage Management for Dependency-flows*

Although the dependency-flows can break the synchronization between refinement and recomputation and reduce the redundant accesses to memory, vertex values may be stored in different memory locations. To further improve the efficiency

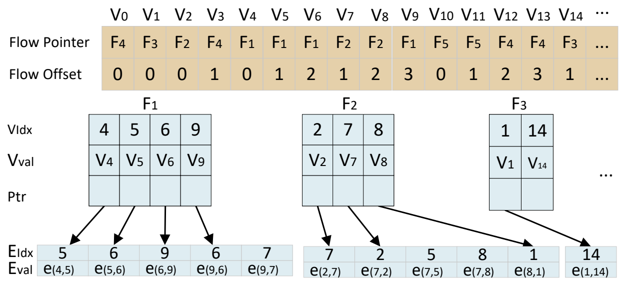
V0 V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12 V13 V14 ...
Flow Pointer
F4 F3 F2 F4 F1 F1 F1 F2 F2 F1 F5 F5 F4 F4 F3 ...
Flow Offset
0 0 0 1 0 1 2 1 2 3 0 1 2 3 1 ...
F1
F2
F3
Vidx
4 5 6 9
2 7 8
1 14
Vval
V4 V5 V6 V9
V2 V7 V8
V1 V14
...
Ptr
Eidx
Eval
5 6 9 6 7
e(4,5) e(5,6) e(6,9) e(9,6) e(9,7)
7 2 5 8 1
e(2,7) e(7,2) e(7,5) e(7,8) e(8,1)
14
e(1,14)

Fig. 8. Specialized graph layout

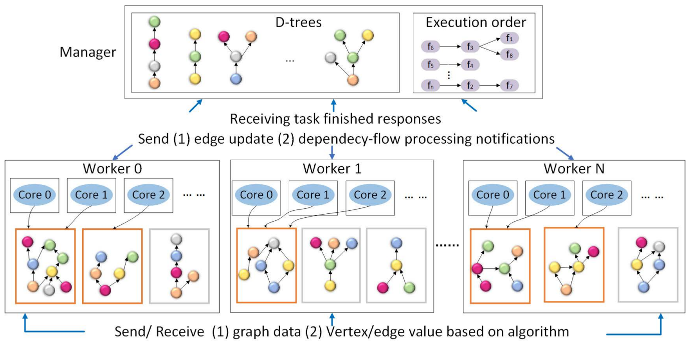
Manager
D-trees
...
Execution order
Receiving task finished responses
Send (1) edge update (2) dependecy-flow processing notifications
Worker 0
Worker 1
Worker N
Core 0
Core 1
Core 2
......
Send/ Receive (1) graph data (2) Vertex/edge value based on algorithm

Fig. 9. The GraphFly workflow

of memory access latency, we design a specialized layout for storing dependency-flows as shown in Fig. 8. We store the vertex data of dependency-flow data compactly.

V_{idx} is used to store the indices of vertices contained in the dependency-flow, and the values of the vertices are stored in V_{val}. Storing dependent vertices together can improve cache efficiency and reduce memory access latency. The edges and the edge values are stored in E_{idx} and E_{val}, respectively. Each vertex has a pointer Ptr to store the index of its first edge. Besides, we also maintain the *Flow Pointer* and *Flow Offset* for each vertex to access its data in dependency-flow quickly. *Flow Pointer* is a pointer to the starting address of the stored dependency-flow, and *Flow Offset* is the offset value of the vertex stored in the dependency-flow. Fig. 8 illustrates how to store the dependency-flows of the directed graph in Fig. 7(a). In this way, when performing the refinement and recomputation of the dependency-flow, both the vertex data and the edge data are accessed in a coalesced way.

VI. Implementation

Fig. 9 shows the high-level workflow overview of GraphFly, which includes two major components for parallel computing: a *Manager* task and several *Worker* tasks. The Manger task is responsible for maintaining D-trees and scheduling the execution order of dependency-flows when a batch of streaming edges arrives. The *Worker* tasks simultaneously execute the refinement and computation phases of edge updates by following the scheduling order generated by the Manager task. This Manager-Worker abstraction can work effectively in parallel for both a single machine and a distributed setting.

```
processEdgeStream(Updates, G, flowHandler, AlgType):
  parallel for e ∈ Updates do
    /* Below are two concurrent steps in flowHandler */
      DTree, Orders ← updateDtree(e), generateOrders(e)
      DepencencyFlow ← updateDFlow(e)
flowHandler.execute(refineEdge, computeEdge, Updates):
  parallel for FlowOrder ∈ Orders do
    for each level ∈ FlowOrder do      /* Asynchronous
      parallel for f ∈ level do          execution */
        run(refineEdge, computeEdge, f)
refineEdge(G, Edge<u, v, w>);      /* Refine definition */
computeEdge(G, Edge<u, v, w>);  /* Compute definition */
```

Fig. 10. Pseudo-code for the parallel execution of GraphFly

Single Machine Workflow. Initially, we generate the D-trees of a graph offline. The initial graph is divided into many dependency-flows. Fig. 10 shows the execution procedure of GraphFly. When a batch of edge updates arrives, the Manager maintains the D-tree and generates the execution order of dependency-flows based on edge updates while the workers update the graph data in parallel. After the graph is updated, the Manager will assign independent dependency-flows to different Worker threads based on the execution order. If a dependency-flow has many update edges, multiple threads can be used to utilize its inherent vertex-level parallelism. The Manager thread will be informed when a worker has completed its local dependency-flow tasks, and assigned new executable dependency-flows to the worker until all dependency-flows are completed. The above workflow will be repeated when a new edge updates batch arrives.

Distributed Implementations. The distributed workflow on GraphFly is similar to its workflow on the single machine environment. GraphFly selects one node to run the Manager task while others are Workers. We next focus on describing the differences as follows.

- *Data Management*. A graph is partitioned into many partitions (i.e., dependency-flows). Those dependency-flows arising from the same D-tree (which can be divided further as discussed in V-A) will be preferably distributed in the same Worker node. In the distributed version, the Manager node also has a flow-worker table to locate which Worker node the dependency-flow resides in. When a new edge update arrives, Manager will first look up the flow-worker table to find the corresponding Worker that resides the dependency-flow of this edge update and then send the new edge to the identified Worker.
- *Communication*. The distributed GraphFly follows a message-passing communication model. Each Manager node and Worker node run one MPI process to communicate with other nodes. The Manager node sends the edge update and its dependency-flow processing notification to a corresponding Worker and receives the task completion responses from Workers. Workers communicate with each other in two cases: (1) Suppose a large dependency-flow f has many small slices distributed in different Workers.

When the upstream data of f in a Worker is modified by an edge update, the resulting vertex and/or edge value in this Worker will be sent to the downstream Worker(s) to update the downstream data of f; (2) When a dependency-flow is moved to another node for load balancing, the corresponding graph data will be moved as well.

- *Workload Balancing*. As the graph is constantly changing, the total number of vertices among different Worker nodes may become significantly different because of the power-law distribution of the graph, leading to load imbalance issue. To ensure load balance, GraphFly tries to dynamically ensure a similar vertex count in total on each worker node, which is monitored by the Manager. This happens in graph update, hence the overhead of dependency-flow movement can be overlapped. Note that the load imbalance issue between Worker nodes does not occur frequently as there are periods of relative calm when the graph is changing [17]. At runtime, there may be the case that some workers are idle while others are overloaded with tasks. GraphFly also enables work-stealing to fetch executable dependency-flows from a busy node to an idle Worker like the previous work [18].

Programming. To support GraphFly, we modify the computing engine and data scheduler in existing streaming graph frameworks. Specifically, as shown in Fig. 10, this is achieved by adding a specialized `flowHandler` that automatically maintains and schedules dependency-flows. When a batch of streaming edges arrives, `processEdgeStream` invokes `flowHandler` to maintain the graph. Then, `flowHandler` schedules dependency-flows to be processed by `refineEdge` and `computeEdge`. These modifications are made in the streaming graph system kernels that are transparent to users. Therefore, users can follow the existing programming model (e.g., edge-centric model [8], [9]) to write a graph algorithm by easily using the same graph APIs (i.e., `refineEdge` and `computeEdge`) provided by the underlying graph frameworks without any application-level code modification.

VII. Evaluation

A. Experimental Setup

Workloads. We use five real-world graph datasets listed in Table I. For graph algorithms, we consider selective and accumulative algorithms. We use four selective graph algorithms: *Single Source Shortest Paths* (SSSP) [19], [20], *Single Source Widest Paths* (SSWP) [21], *Breadth-First Search* (BFS) [22], [23], and *Connected Components* (CC) [24], [25]. These applications are representatives of selective algorithms and they perform selection operations based on the values of neighboring vertices. *PageRank* [26], [27] and *Label Propagation* (LP) [28] are selected to represent the accumulative algorithm.

Like previous work [8], we use 50% of the graph as the initial graph, and the rest of the edges are added with graph mutations. The graph updates also include the deletion of edges. In addition, the edges are deleted from the graph with 0.1 probability. Edge deletions and edge additions are combined for batch updates simultaneously. In our experiments, we vary the proportional number of deleted edges and the size of the batch to evaluate the effectiveness of GraphFly.

TABLE I
Real World Graph Datasets

Graphs	#Edges	#Vertices
Friendster (FT) [29]	2.5B	68.3M
Twitter MPI (TT) [30]	2.0B	52.6M
Twitter (TW) [31]	1.5B	41.7M
UKDomain (UK) [32]	1.0B	39.5M
LiveJournal (LJ) [33]	69M	4.8M

Environment. Our single-machine trials run on a server with two 14-core Intel Xeon E5-2680v4 processors, 256GB memory, and 512GB SSD. The running system is operated on the 64-bit Ubuntu 18.04 with kernel 5.4. The applications are compiled with gcc 5.5.0 at optimization level 'O3'. Our distributed experiments run on a 16-node cluster connected via a 10Gbps network. Each node has the same configuration as the single machine server. We use OpenMP 3.0.0 for multi-threading and MPI for distributed communications.

Methodology. We compare two state-of-the-art systems as the baselines: 1) KickStarter [8]: refine vertex error values triggered by edge deletion for monotonic algorithms; 2) GraphBolt [9]: provide *Bulk Synchronous Parallel* (BSP) support via dependency-driven incremental processing. We compare with KickStarter and GraphBolt for monotonic and accumulative algorithms, respectively. Note that GraphBolt is open-sourced as a single-node version and KickStarter is embedded into GraphBolt, so we only compare the performance of both systems on a single machine.

B. Performance and Characteristics

We first present the overall performance of GraphFly over KickStarter and GraphBolt. Then we show the benefits of proposed techniques for memory access efficiency.

Overall Performance. Fig. 11 shows the execution time for KickStarter, GraghBolt, and GraphFly with 100K edge mutations. The execution time in Kickstart and GraghBolt includes refinement and recomputation only. In addition to refinement and recomputation, the execution time in GraphFly additionally includes the incremental maintenance overhead of D-trees. As we can see, GraphFly outperforms both KickStarter and GraphBolt. This is mainly because our dependency-flow breaks the synchronization between refinement and recomputation. Recomputation can reuse the data accessed from memory by refinement, avoiding redundant memory accesses in both phases. In addition, GraphFly stores vertices compactly in a dependency-flow, which further improves the efficiency.

We observe different speedups in various algorithms. For selective-type algorithms, GraphFly outperforms KickStarter by 5.81$\times$ on average, while only outperforms GraphBolt by 1.78$\times$ on average for accumulative-type algorithms. The reason behind is that, for the accumulative-type algorithms, the vertex state is derived from the neighbors of all its incoming edges. Therefore D-trees are generated based on all

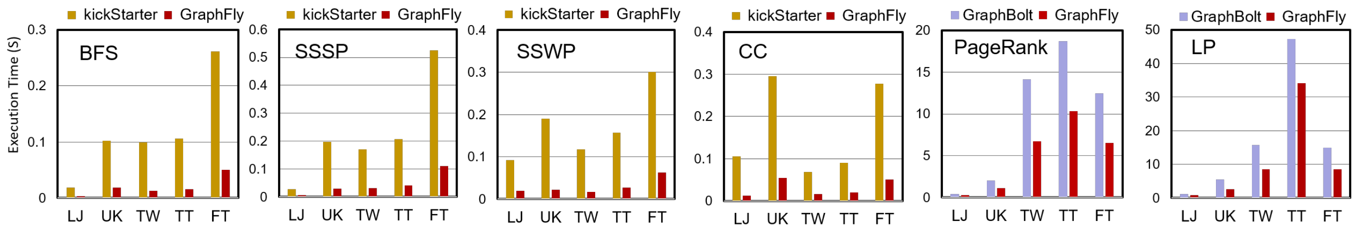

Fig. 11. Executions time for KickStarter, GraphBolt, and GraphFly with 100K edge mutations

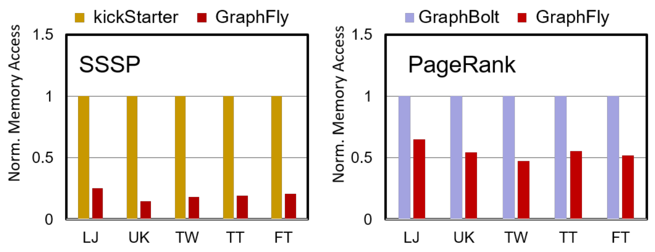

Fig. 12. Normalized memory accesses comparison

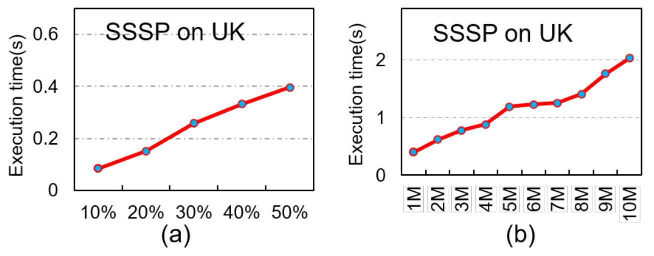

Fig. 14. (a) Execution time with different percentages of deletions for SSSP on UK; (b) Execution time with varying batch sizes for SSSP on UK

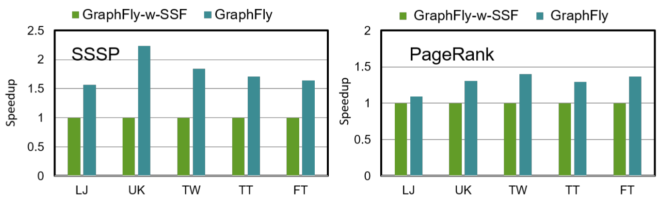

Fig. 13. Execution time of GraphFly with/without specialized storage format

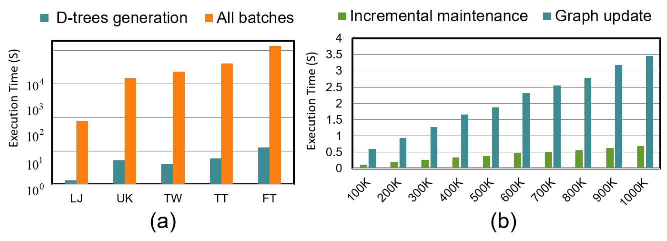

Fig. 15. (a) Generation time of D-trees and the total incremental computation time for all batches; (b) Execution time for D-trees incremental maintenance and graph update

connected edges. This results that the generated dependency-flows become too large, even though the space-time dependent co-scheduling approach limits the data reuse in the cache for both refinement and recomputation. For selective types of algorithms, each vertex is affected by only one key edge, and our D-trees are generated based on these key edges only. Therefore, the generated dependency-flow is fine-grained, which facilitates the data reuse in the cache by both refinement and recomputation.

Memory Access Efficiency. Fig. 12 further shows the memory access efficiency of GraphFly against KickStarter using SSSP algorithm and GraphBolt using PageRank algorithm. GraphFly reduces memory accesses by 80.19% compared to KickStarter and by 38.02% compared to GraphBolt. This demonstrates the effectiveness of dependency-flows in GraphFly. Compared to the whole graph, the dependency-flow has better cache locality. Once the dependency-flow executes refinement to access data from memory, the following recomputation does not need to access data from memory again. It can improve cache hits rate and reduce the number of expensive memory accesses. In addition, the compact storage of the dependency-flow further reduces the memory accesses.

C. *Effectiveness of Specialized Storage Format*

The performance of GraphFly with and without specialized storage format (GraphFly-w-SSF) is shown in Fig. 13. We can see that GraphFly is on average 1.81× and 1.29× faster than GraphFly-w-SSF on SSSP and PageRank. Even if we know the dependencies among vertices, these vertices are located in different memory locations. Accessing these vertices requires multiple memory accesses, since they are not compactly stored together. With the specialized storage format, we can reduce the number of accesses, further improving the effectiveness of memory accesses, i.e., accessing a memory block, the rate of useful vertices is higher. Moreover, our specialized storage format also improves the effectiveness of the cache by maximizing the usefulness of data storage in the cache, which means more cache hits.

D. *Sensitivity of Edge Deletions and Batch Size*

This part studies the sensitivity of the effectiveness of GraphFly to the number of deletions of edges in the graph update. Fig. 14(a) shows the execution time under different percentages of deleted edges in the graph update batch from 10% to 50%, while keeping the same batch size as 100k edge streams. It is important to note that our technique can maintain a relatively stable execution time in many cases. This is because dependency-flow accurately expresses the vertex dependencies and knows exactly which vertices are affected by

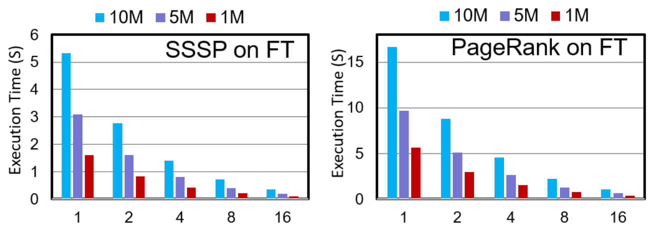

Fig. 16. Execution time by varying the number of nodes

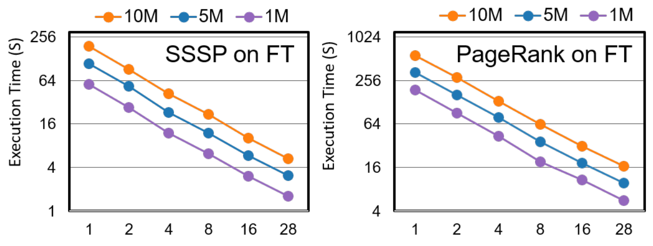

Fig. 17. Execution time by varying the number of available cores

the deleted edges. Even if the number of deleted edges in the dependency-flow increases, the amount of memory accesses is reduced, and the cache hits rate is improved.

Fig. 14(b) shows the variation with the edge updates from 1M to 10M while the percentage of edge deletion is set to 30%. As the number of edges keeps increasing, there are no significant impacts on the executions time. This is because the memory access overhead in the dependency-flow is stable even though the number of edge updates keeps changing, benefiting from the fact that GraphFly eliminates redundant memory accesses for refinement and recomputation.

E. Overhead of Generating and Maintaining D-trees

Fig. 15(a) shows the D-tree generation overhead and the total incremental computation time for all batches. We see that the D-tree generation overhead takes only 0.47% of the total incremental computation time, which are reasonable in practice. Actually, this small generation overhead incurs only once and can still be amortized by multiple runs. For example, in real scenarios the graph is continuously changing. Thus, the number of batches will keep increasing and the generated D-trees can be repeatedly used. Fig. 15(b) shows the incremental maintenance overhead and graph update overhead with different batch sizes. The graph structure needs to be changed when the edge updates arrive. Incremental maintenance of D-trees and graph updates can be performed in parallel. The incremental maintenance overhead of D-trees is less than graph updating overhead, and can be overlapped with graph update.

F. Scalability

Fig. 16 shows the execution time of SSSP and PageRank on FT as the number of nodes increases. To fully evaluate the scalability of GraphFly, we also vary batch sizes from 1M to 10M. As we can see, the execution time decreases dramatically as the number of compute nodes increases. We assign dependency-flows to different nodes. These closely related vertices are grouped into the same node, which incurs less communication overhead between nodes. Fig. 17 further characterizes the performance of GraphFly with different number of cores available on a single machine. We see more available cores achieve faster runtime with the dependency-flow ideology that enhances data reusability in the cache. Also, our dependency-flows expose finer-grained parallelism that can be exploited more fully by threads. Therefore, GraphFly can be easily scaled with respect to the number of available cores.

VIII. Related Work

A large amount of research works have emerged to support dynamic graph processing. For fast-changing streaming graph processing or historical analysis on evolving graphs, existing works propose many novel ideas in terms of graph computational models and storage methods.

Streaming Graph Processing Systems: Existing systems mainly adopt incremental computation to process streaming graphs for its efficiency. Kineograph [7] and Tornado [6] maintain a “main loop” to constantly compute approximate results, a branch is forked when user query arrives to continue processing based on the approximations until it converges. Naiad [34] adopts differential data flow to incrementally compute on recorded value changes for general-purpose processing, and thus supports streaming graphs. Tripoline [35] also incorporates the incremental computation while supports results reusing for different vertex-specific queries via the concept of “graph triangle inequality”. These systems are limited to the situation where only addition edges are applied, and encounter correctness problems while facing edge deletions.

GraphIn [10] proposed a tagging method to identify all possible affected vertices and reset them to initial values to guarantee correctness. Instead of tagging all possible vertices, better approximation methods for identifying impacted vertices and refining values can reduce necessary computations to convergence. KickStarter [8] proposed a dependence tracking approach and trimming process to generate useful approximations in asynchronous execution. GraphBolt [9] further devised dependency-driven refinement to provide *Bulk Synchronous Parallel* (BSP) support. Recent work DZiG [11] explored sparsity in iterative algorithms to avoid computations on already converged vertices during incremental refinement. While most of the work focus on correctness problem, GraphFly targets redundant memory accesses problem across refinement and recomputation phases. Existing optimizations on two phases can also be absorbed into GraphFly.

Dynamic Graph Data Structures and Storage Systems: Storing dynamic graphs requires flexible and efficient data structures to handle fast arriving updates. Usually, graphs are partitioned into edge blocks in these systems and represented as adjacency lists to guarantee high-throughput mutation and access. Representing systems include LLAMA [36], STINGER [37], and GraphOne [38]. Aspen [39] supports concurrently updating and processing dynamic graphs via *C-Trees*. LiveGraph [40] is a transactional graph storage system which supports both efficient graph updates and analytics. ElGA [41] supports highly elastic and scalable dynamic graph analysis using a shared-nothing architecture and consistent hashing, and efficiently handles high degree vertices by applying sketches. The specialized graph layout proposed in GraphFly also enables fast mutation and access. The dependency-flow oriented storage provides better locality during processing.

Systems for Evolving Graph Analytics: Instead of always querying the latest results, evolving graph analytics aims to mine historical information, e.g., the difference of results between different graph versions overtime. Evolving graphs are stored as different versions. The historical analysis is more like static graph processing on different graph versions. State-of-the-art systems like GraphTau [42], ImmortalGraph [43], and Chronos [44] usually reuse the results of previous graph version and adopt incremental computation to accelerate the convergence. Because the value propagation also obeys the dependency-flow in the incremental computations, organizing the graph versions into dependency-flows may also improve the performance in evolving graph analytics because the differences of snapshots also propagate following the dependencies.

IX. Conclusion

In this work, we present GraphFly, an approach that eliminates the redundant memory access across the refinement and recomputation phases in streaming graph processing. GraphFly proposes the D-trees to capture the dependency-flows, which makes breaking the barrier between refinement and recomputation possible. Based on D-trees, GraphFly implements a highly paralleled asynchronous processing model to embrace efficiency and correctness. The proposed techniques are orthogonal to previous research and can also benefit existing streaming graph systems. Evaluation on real-world graphs shows that GraphFly can significantly outperform state-of-the-art systems KickStarter by $5.81\times$ for monotonic algorithms and Graphbolt by $1.78\times$ for accumulative algorithms on average. GraphFly also scales well in distributed settings.

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful comments. This work is supported by the NSFC (No. 61832006, 62072195, 61825202, and 61929103) and Huawei Technologies Co., Ltd.

References

[1] T. Steil, T. Reza, K. Iwabuchi, B. W. Priest, G. Sanders, and R. Pearce, “Tripoll: computing surveys of triangles in massive-scale temporal graphs with metadata,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 67:1–67:12, 2021.

[2] Y. Huang, L. Zheng, P. Yao, Q. Wang, X. Liao, H. Jin, and J. Xue, “Accelerating graph convolutional networks using crossbar-based processing-in-memory architectures,” in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*, pp. 1029–1042, 2022.

[3] L. Zheng, X. Li, Y. Zheng, Y. Huang, X. Liao, H. Jin, J. Xue, Z. Shao, and Q. Hua, “Scaph: Scalable gpu-accelerated graph processing with value-driven differential scheduling,” in *Proceedings of the 2020 USENIX Annual Technical Conference*, pp. 573–588, 2020.

[4] H. Wang, Z. Li, P. Zhang, J. Huang, P. Hui, J. Liao, J. Zhang, and J. Bu, “Live-streaming fraud detection: A heterogeneous graph neural network approach,” in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 3670–3678, 2021.

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

[6] X. Shi, B. Cui, Y. Shao, and Y. Tong, “Tornado: A system for real-time iterative analysis over evolving data,” in *Proceedings of the International Conference on Management of Data*, pp. 417–430, 2016.

[7] 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 the 7th ACM European Conference on Computer Systems*, pp. 85–98, 2012.

[8] K. Vora, R. Gupta, and G. Xu, “KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 237–251, 2017.

[9] M. Mariappan and K. Vora, “GraphBolt: Dependency-driven synchronous processing of streaming graphs,” in *Proceedings of the 14th European Conference on Computer Systems*, pp. 25:1–25:16, 2019.

[10] D. Sengupta, N. Sundaram, X. Zhu, T. L. Willke, J. Young, M. Wolf, and K. Schwan, “GraphIn: An online high performance incremental graph processing framework,” in *Proceedings of the 22nd European Conference on Parallel Processing*, pp. 319–333, 2016.

[11] M. Mariappan, J. Che, and K. Vora, “DZiG: Sparsity-aware incremental processing of streaming graphs,” in *Proceedings of the 16th European Conference on Computer Systems*, pp. 83–98, 2021.

[12] J. R. Gilbert and J. W. Liu, “Elimination structures for unsymmetric sparse LU factors,” *SIAM Journal on Matrix Analysis and Applications*, vol. 14, no. 2, pp. 334–352, 1993.

[13] P. Sao, X. S. Li, and R. Vuduc, “A communication-avoiding 3D LU factorization algorithm for sparse matrices,” in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pp. 908–919, 2018.

[14] D. W. Margo and M. I. Seltzer, “A scalable distributed graph partitioner,” *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1478–1489, 2015.

[15] P. Heggernes, “Minimal triangulations of graphs: A survey,” *Discrete Mathematics*, vol. 306, no. 3, pp. 297–317, 2006.

[16] Y. Zheng, Y. Bian, S. Li, and S. E. Li, “Cooperative control of heterogeneous connected vehicles with directed acyclic interactions,” *IEEE Intelligent Transportation Systems Magazine*, pp. 127–141, 2021.

[17] K. Gabert, K. Sancak, M. Y. Özkaya, A. Pinar, and Ü. V. Çatalyürek, “EIGA: elastic and scalable dynamic graph analysis,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 50:1–50:15, 2021.

[18] P. Yao, L. Zheng, Z. Zeng, Y. Huang, C. Gui, X. Liao, H. Jin, and J. Xue, “A locality-aware energy-efficient accelerator for graph mining applications,” in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 895–907, 2020.

[19] V. T. Chakaravarthy, F. Checconi, F. Petrini, and Y. Sabharwal, “Scalable single source shortest path algorithms for massively parallel systems,” in *Proceedings of the 28th International Parallel and Distributed Processing Symposium*, pp. 889–901, 2014.

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

[21] A. Dumitrescu, “Dynamic widest path selection for connection admission control in core-stateless networks,” in *Proceedings of the International Conference on Quality of Service in Heterogeneous Wired/Wireless Networks*, pp. 102–111, 2004.

[22] S. Hong, S. Salihoglu, J. Widom, and K. Olukotun, “Simplifying scalable graph processing with a domain-specific language,” in *Proceedings of Annual International Symposium on Code Generation and Optimization*, pp. 208–218, 2014.

[23] S. Beamer, K. Asanovic, and D. A. Patterson, “Direction-optimizing breadth-first search,” in *Proceedings of the International Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–12, 2012.

[24] L. di Stefano and A. Bulgarelli, “A simple and efficient connected components labeling algorithm,” in *Proceedings of the International Conference on Image Analysis and Processing*, pp. 322–327, 1999.

[25] P. Yao, L. Zheng, Y. Huang, Q. Wang, C. Gui, Z. Zeng, X. Liao, H. Jin, and J. Xue, "Scalagraph: A scalable accelerator for massively parallel graph processing," in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*, pp. 199–212, 2022.

[26] B. Sergey, P. Lawrence, R. Motwami, and W. Terry, "The PageRank citation ranking: Bringing order to the web," in *Proceedings of the 14th International Conference on World Wide Web*, pp. 567–574, 2005.

[27] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[28] C. Ye, Y. Li, B. He, Z. Li, and J. Sun, "GPU-accelerated graph label propagation for real-time fraud detection," in *Proceedings of the International Conference on Management of Data*, pp. 2348–2356, 2021.

[29] "Friendster network dataset." http://konect.uni-koblenz.de/networks/friendster. KONECT, 2015.

[30] M. Cha, H. Haddadi, F. Benevenuto, and P. K. Gummadi, "Measuring user influence in twitter: The million follower fallacy," in *Proceedings of the 4th International Conference on Weblogs and Social Media*, pp. 10–17, 2010.

[31] H. Kwak, C. Lee, H. Park, and S. B. Moon, "What is Twitter, a social network or a news media?," in *Proceedings of the 19th International Conference on World Wide Web*, pp. 591–600, 2010.

[32] P. Boldi and S. Vigna, "The webgraph framework I: compression techniques," in *Proceedings of the 13th International Conference on World Wide Web*, pp. 595–602, 2004.

[33] L. Backstrom, D. P. Huttenlocher, J. M. Kleinberg, and X. Lan, "Group formation in large social networks: membership, growth, and evolution," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 44–54, 2006.

[34] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pp. 439–455, 2013.

[35] X. Jiang, C. Xu, X. Yin, Z. Zhao, and R. Gupta, "Tripoline: Generalized incremental graph processing via graph triangle inequality," in *Proceedings of the 16th European Conference on Computer Systems*, pp. 17–32, 2021.

[36] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "LLAMA: Efficient graph analytics using large multiversioned arrays," in *Proceedings of the International Conference on Data Engineering*, pp. 363–374, 2015.

[37] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "STINGER: High performance data structure for streaming graphs," in *Proceedings of the IEEE Conference on High Performance Extreme Computing*, pp. 1–5, 2012.

[38] P. Kumar and H. H. Huang, "GraphOne: A data store for real-time analytics on evolving graphs," *ACM Transactions on Storage*, vol. 15, no. 4, pp. 29:1–29:40, 2020.

[39] L. Dhulipala, G. E. Blelloch, and J. Shun, "Low-latency graph streaming using compressed purely-functional trees," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 918–934, 2019.

[40] X. Zhu, G. Feng, M. Serafini, X. Ma, J. Yu, L. Xie, A. Aboulnaga, and W. Chen, "LiveGraph: A transactional graph storage system with purely sequential adjacency list scans," *Proceedings of the VLDB Endowment*, vol. 13, no. 7, pp. 1020–1034, 2020.

[41] K. Gabert, K. Sancak, M. Y. Özkaya, A. Pinar, and U. V. Çatalyürek, "EIGA: Elastic and scalable dynamic graph analysis," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 50:1–50:15, 2021.

[42] A. P. Iyer, L. E. Li, T. Das, and I. Stoica, "Time-evolving graph processing at scale," in *Proceedings of the 14th International Workshop on Graph Data Management Experiences and Systems*, pp. 5:1–5:6, 2016.

[43] Y. Miao, W. Han, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, E. Chen, and W. Chen, "ImmortalGraph: A system for storage and analysis of temporal graphs," *ACM Transactions on Storage*, vol. 11, no. 3, pp. 14:1–14:34, 2015.

[44] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen, "Chronos: A graph engine for temporal graph analysis," in *Proceedings of the 9th European Conference on Computer Systems*, pp. 1–14, 2014.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

This paper includes a series of experiments that run Single Source Shortest Paths (SSSP), Single Source Widest Paths (SSWP), Breadth-First Search (BFS), Connected Components (CC), PageRank, and Label Propagation (LP) on public real-world graphs.

For performance comparison, we run GraphFly on a single node, which has 2× 14-core Intel Xeon E5-2680v4 processors with 256GB RAM and 512GB SSD, and runs 64-bit Ubuntu 18.04. Both GraphFly and the baselines (GraphBolt, https://github.com/pdclab/graphbolt) are implemented under the same setting and compiled with g++ >= 5.3.0 with -O3 option.

The graph datasets evaluated can be downloaded in Table 1. These graphs are stored as lists of edge tuples ($< source, destination >$). In order to simulate streaming graphs, we randomly select edges from the original static graphs as edge deletions and additions. These edge updates are then stored in a file as edge streams.

In order to execute the experiments, GraphFly uses a script to convert the original graphs into dependency flows and to sample the edge streams. Dependency flows are stored in the same format as original graphs. For both GraphFly and baselines, the original graphs and edge streams are used. The dependency flows are only used for GraphFly. For label propagation, a seeds file is provided.

Experimental Setup:

- Relevant hardware details: Intel(R) Xeon(R) CPU E5-2680 2.10GHz with 256 GB RAM
- Operating systems and versions: Ubuntu 18.04
- Compilers and versions: GCC 5.5.0, Python3
- Applications and versions: GraphFly, GraphBolt
- Input datasets and versions: SNAP datasets, WebGraph datasets, Konect datasets

Table 1: Real World Graph Datasets

Graphs	URLs
Friendster (FT)	http://konect.cc/networks/friendster/
Twitter MPI (TT)	http://konect.cc/networks/twitter_mpi/
Twitter (TW)	https://snap.stanford.edu/data/twitter-2010.html
UKDomain (UK)	https://law.di.unimi.it/webdata/uk-2005/
LiveJournal (LJ)	https://snap.stanford.edu/data/soc-LiveJournal1.html

AUTHOR-CREATED OR MODIFIED ARTIFACTS:

Artifact 1

Persistent ID: `https://doi.org/10.5281/zenodo.6616276`
Artifact name: GraphFly

Reproduction of the artifact without container: Commands to compile the applications for GraphFly:

```
$ cd apps
$ make -j
```

Example commands to run the applications for GraphFly on LiveJournal graph:

```
$ ./BFS -source 1 -numberOfUpdateBatches 1 -nEdges 100000
-streamPath ../inputs/live-journal/[edge streams file]
-outputFile /tmp/output/bfs_log ../inputs/livejournal/
[dependency flows file]

$ ./SSSP -source 1 -numberOfUpdateBatches 1 -nEdges 1000-
00 -streamPath ../inputs/livejournal/[edge streams file]
-outputFile /tmp/output/sssp_log ../inputs/livejournal/
[dependency flows file]

$ ./SSWP -source 1 -numberOfUpdateBatches 1 -nEdges 1000-
00 -streamPath ../inputs/livejournal/[edge streams file]
-outputFile /tmp/output/sswp_log ../inputs/livejournal/
[dependency flows file]

$ ./CC -source 1 -numberOfUpdateBatches 1 -nEdges 100000
-streamPath ../inputs/livejournal/[edge streams file]
-outputFile /tmp/output/cc_log ../inputs/livejournal/
[dependency flows file]

$ ./PageRank -numberOfUpdateBatches 1 -nEdges 100000
-streamPath ../inputs/livejournal/[edge streams file]
-outputFile /tmp/output/pr_log ../inputs/livejournal/
[dependency flows file]

$ ./LabelPropagation -numberOfUpdateBatches 1 -nEdges
100000 -streamPath ../inputs/livejournal/[edge streams
file] -seedsFile ../inputs/livejournal/lj-seeds-file
-outputFile /tmp/output/lp_log ../inputs/livejournal/
[dependency flows file]
```