GraphIn: An Online High Performance Incremental Graph Processing Framework

Dipanjan Sengupta[1], Narayanan Sundaram[2], Xia Zhu[2], Theodore L. Willke[2], Jeffrey Young[1], Matthew Wolf[1], and Karsten Schwan[1]

[1] Georgia Institute of Technology, Atlanta GA, USA,
{dsengupta6, jyoung9}@gatech.edu, {mwolf, karsten.schwan}@cc.gatech.edu,
[2] Intel Labs, Hillsboro OR, USA,
{narayanan.sundaram, xia.zhu, theodore.l.willke}@intel.com

Abstract. The massive explosion in social networks has led to a significant growth in graph analytics and specifically in dynamic, time-varying graphs. Most prior work processes dynamic graphs by first storing the updates and then repeatedly running static graph analytics on saved snapshots. To handle the extreme scale and fast evolution of real-world graphs, we propose a dynamic graph analytics framework, GraphIn, that incrementally processes graphs on-the-fly using fixed-sized batches of updates. As part of GraphIn, we propose a novel programming model called I-GAS (based on gather-apply-scatter programming paradigm) that allows for implementing a large set of incremental graph processing algorithms seamlessly across multiple CPU cores. We further propose a property-based, dual-path execution model to choose between incremental or static computation. Our experiments show that for a variety of graph inputs and algorithms, GraphIn achieves up to 9.3 million updates/sec and over 400× speedup when compared to static graph recomputation.

Keywords: Graph, Big data, Performance, Incremental Processing

1 Introduction

With the increasing interest in many emerging domains such as social networks, the World Wide Web (e-commerce and advertising), and genomics, the importance of dynamic graph processing has grown substantially. This recent trend has given rise to many graph processing frameworks like GraphLab [13], PowerGraph [10], Graphchi [12], and X-Stream [19] that operate on time-varying real-world graphs. However, most current graph analytics on such dynamic graphs follow a store-and-static-compute model that involves storing batches of updates to a graph applied at different points in time and then repeatedly running static graph computations on the "snapshots" of the evolving graph. The key assumption made here is that the dynamic graph changes slower than the static processing rate. Extreme-scale applications like Facebook's representative social graph benchmark [4] reports an update rate of 86,400 objects/second in 2013, while Twitter traffic [3] can peak at 143 thousand tweets (and associated

updates) per second and emails sent [1] can reach as high as 2.5 millions/sec. This high volume of changes in dynamic graphs is further complicated by the need for soft or hard real-time guarantees for applications like real-time anomaly detection and disease spreading. Both the volume and complexity of queries have outstripped traditional static graph analytics.

To address the challenges of large scale dynamic graph processing, we propose an incremental graph processing framework called GraphIn. The GraphIn framework employs a novel programming model called Incremental-Gather-Apply-Scatter (or I-GAS) to incrementally process a continuous stream of updates (i.e., edge/vertex insertions and/or deletions) as a sequence of batches. The I-GAS programming model is based on the popular gather-apply-scatter (GAS) programming paradigm [10] and it allows an incremental graph problem to be reduced to a sub-problem that operates on a portion, or sub-graph, of the entire evolving graph. This sub-graph abstraction allows I-GAS to substantially outperform traditional static processing techniques.

Furthermore, GraphIn takes into account situations where incremental processing may perform worse than static recomputation, such as with incremental BFS, where updates may affect the entire BFS tree. To handle such scenarios, we introduce the notion of "dual-path execution" or property-based switching between incremental and static graph processing based on built-in and user defined properties (e.g., vertex degree information).

Finally, GraphIn's design allows it to run on top of, and take advantage of performance benefits of any GAS-based static graph analytics framework like X-Stream [19], GraphMat [27] (used in this work) or Graphchi [12].

This paper makes following contributions:

- A high-performance incremental graph processing framework built on top of GraphMat to process time-varying evolving graphs.
- A novel programming model called I-GAS for simplified implementation of incremental versions of many popular graph algorithms using the GraphIn framework that seamlessly generates parallel code for multi-core systems.
- An optimization heuristic to decide between static and dynamic graph execution based on built-in and user defined graph properties. This dual path execution results in speedups of up to 60× over a naïve streaming approach.
- An extensive evaluation of GraphIn for three popular graph algorithms that operate on large scale real-world and synthetic graph datasets. Compared to competitive frameworks such as STINGER [7], GraphIn achieves a speedup of up to 6.6×. Overall, GraphIn achieves up to 9.3 million updates/sec and 400× speedup over the naïve static graph recomputation approach.

The remainder of the paper is organized as follows: Section 2 discusses the background on graph analytics. Section 3 introduces our GraphIn framework. Section 4 presents the experimental setup and result analysis. Section 5 discusses the related work and Section 6 concludes with future work.

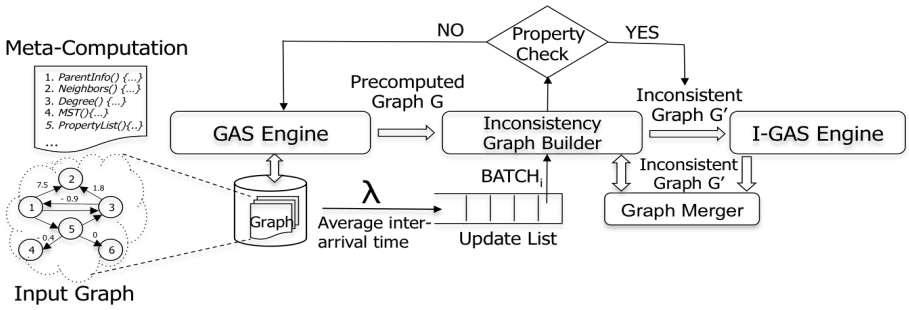

Fig. 1. GraphIn Software Architecture.

2 Background

Gather-Apply-Scatter (GAS) [10, 13, 14] is a computational model for graph processing sufficiently general to express a broad set of graph algorithms. With GAS, a problem is described as a directed graph, $G = (V, E)$, where V denotes the vertex set and E denotes the directed edge set and a property/state value is associated with each vertex $v_i \in V$. Graph programs written in GAS typically follow 3-phases: *Gather*, *Apply* and *Scatter*. In the Gather phase, incoming messages are processed and combined (reduced) into one message. In the Apply phase, vertices use the combined message to update their state. Finally, in the Scatter phase, vertices can send a message to their neighbors along their edges.

GraphMat [27] complements the use of the GAS model in GraphIn by taking vertex programs and compiling them into sparse matrix operations (e.g. sparse matrix-vector multiplication). The graph programs are specified by at least 4 user-defined functions (UDFs), `SEND_MESSAGE` for scatter phase, `PROCESS_MESSAGE` and `REDUCE` to specify Gather phase and `APPLY` for its eponymous phase.

We use 3 algorithms in this paper - Clustering co-efficient (CCof), Connected Components (CC) and Breadth-First Search (BFS). Clustering coefficient C_v for vertex v is defined as $C_v = \frac{T_v}{d_v(d_v-1)}$ where T_v is the total number of triangles in a graph with vertex v as one of the endpoints and d_v is its degree. CC refers to computing the weakly connected components in the graph. BFS assigns a level (equal to the number of edges traversed) to every vertex reachable from a root.

3 GraphIn Framework

The GraphIn framework can efficiently process evolving graphs by dividing the continuous stream of updates (edge or vertex insertions and/or deletions) into fixed size batches processed in the order of their arrival. It simplifies evolving graph analytics programming by supporting a multi-phase, dual path execution model. Figure 1 shows the general software architecture of GraphIn which consists of five major components: *GAS Engine*, *Inconsistency Graph Builder*, *Property Check*, *I-GAS Engine* and *Graph Merger*.

3.1 Graph Data-Structure

There are multiple options to store an evolving graph with n vertices and m edges. Adjacency matrices allow for fast updates with both insertions and deletions taking $O(1)$ time but require a lot of space ($O(n^2)$). Adjacency lists are

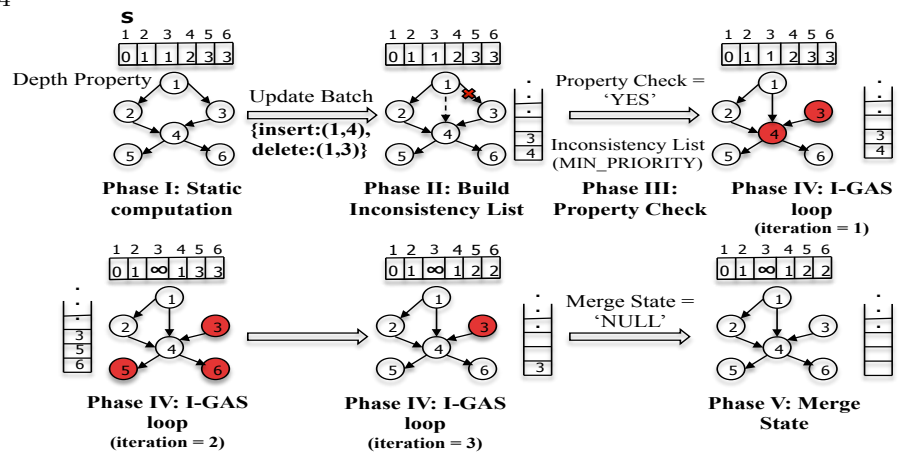

Fig. 2. Incremental BFS Phases.

space efficient with $O(m+n)$ space and allow fast updates but graph traversals are very inefficient due to non-contiguous memory nodes. Compressed Sparse Row (CSR) formats [5] provide both space efficiency combined with fast traversal (often easily parallelized) by storing offsets rather than all the valid fields in the adjacency matrix. However, updates are expensive because each update requires shifting of the graph data throughout the array to match the compressed format.

GraphIn adopts a hybrid data structure involving edge-lists to store incremental updates and compressed matrix format to store a static version of the graph. The edge-list allows for faster updates without adversely affecting the performance of incremental computation. The compressed format allows for faster parallel computation over the entire static version of the graph. The framework merges the update list and the static graph whenever required (see Phase V).

3.2 User Interface

As shown in Table 1, programmers can write a sequential algorithm by simply defining six functions for the different phases in GraphIn. GraphIn will then seamlessly generate parallelized code to incrementally process evolving graph on the target multi-core system. The user-defined functions include *meta_computation()*, *build_inconsistency_list()*, *check_property()*, *activate_frontier()*, *update_inconsistency_list()* and *merge_state()*, corresponding to the different phases of GraphIn computation described below in detail. Figure 2 shows various incremental BFS phases.

3.3 Phase I: Static Graph Computation (GAS Engine)

This phase is responsible for running 1) Static graph computation and 2) Meta-computation to be used later in the incremental logic. The static graph computation follows the GAS programming model, and therefore, any framework that supports GAS can be used as a Static Engine. We have used GraphMat framework for our static graph computation. Meta computation (*meta_computation()*) involves the computation of graph properties like vertex parents, vertex degree etc, which are needed by incremental algorithms for later phases of

	Graph Algorithms		
GraphIn APIs and Phases	Breadth First Search **(BFS)** [18]	Connected Components **(CC)** [8]	Clustering Coefficients **(CCof)** [6]
Type	All-merge	Delete-only-merge	No-merge
meta_computation() **(Phase I)**	Parent id and vertex degree	Vertex degree	Vertex degree
build_inconsistency_list() **(Phase II)**	**1.** Inconsistency list contains vertices with incorrect depth values with MIN_PRIORITY. **2.** G' = G	**1.** For each edge insertion add an edge in G' if the endpoints belong to different components [8, 15]. **2.** G' is also known as component graph.	**1.** Inconsistency list contains endpoints of every edge inserted and/or deleted and their respective neighbors. **2.** G' consists of inconsistent vertices and edge incident on them in G [6]
check_property() **(Phase III)**	Check BFS depth property	Check disjoint component property	Check vertex degree property
activate_frontier() **(Phase IV)**	Activate inconsistent vertices with minimum depth value-Ramalingam and Reps [18]	Activate all the vertices in G'	Activate all the vertices in G'
update_inconsistency_list() **(Phase IV)**	Remove frontier vertices and add inconsistent successors to inconsistency list	Clear inconsistency list	Clear inconsistency list
merge_state() **(Phase V)**	**1.** Apply all insertions and deletions to G.	**1.** Apply only deletions to G. **2.** Relabel components in G using G'.	**1.** Applying insertions and deletions to G not required. **2.** Update triangle counts and degree information in G using G'.

Table 1. Implementing Graph Applications in GraphIn

GraphIn. As an example, incremental BFS requires parent information during Phase IV.

3.4 Phase II: Inconsistency Graph Builder

Between multiple versions or snapshots of an evolving graph the vertex states for many vertices remain the same over time and therefore their recomputation is essentially redundant. We define an *inconsistent vertex* to be a vertex for which one or more properties are affected when the update batch is applied. For example in BFS, addition of edge (v_i, v_j) can potentially make v_j and all vertices that are downstream from v_j inconsistent. This phase is responsible for marking the portions of the graph that become inconsistent using the user-defined function(UDF) *build_inconsistency_list()*. This UDF takes two parameters: the *update batch* and a *user-defined priority*. The update batch consists of edge or vertex insertions and/or deletions from which a list of inconsistent vertices is built after applying the updates. Vertices in this inconsistency list are assigned the user-defined priorities and by default all the inconsistent vertices have equal priority. This phase also optionally builds a user-defined sub-graph G' to be used in the later phases. By default G' is same as the original graph.

3.5 Phase III: Property Check (Static vs. Dynamic)

Runtime of online graph analytics depends both on the algorithm and the particular choice of updates. There are classes of incremental algorithms that cause large portions of the graph to become inconsistent and hence can result in recomputation over the entire graph. For them, incremental processing will not achieve any performance benefit over static recomputation and may even result in performance degradation due to the overheads associated with incre-

Algorithm 1 : I-GAS computation loop per update batch

```
1: while(!inconsistency_list.isempty()):
2:    frontier = activate_frontier(G', inconsistency_list)
3:    IGAS(G')
4:    update_inconsistency_list(G', inconsistency_list, frontier)
```

mental execution. To deal with such situations, we allow the user to define a heuristic for determining when one form of computation is used over the other. The user may select from a set of predefined graph properties (e.g. vertex degree) or define their own properties that affect the runtime of the incremental algorithm. The framework will then use the selected properties to decide whether to run incremental or static recomputation by calling *check_property()* for each update batch. This is called "property-based dual path execution". *check_property()* takes four parameters: *inconsistency_list*, *property_list*, *threshold_vector*, and *threshold_fraction*. *Property list* defines the set of graph properties under consideration. *Threshold vector* defines a set of thresholds for properties in the property list, above (or below) which the performance of incremental processing will drastically degrade. Finally, *threshold fraction* defines the fraction of inconsistent vertices that are above (or below) the corresponding property threshold. For example, in BFS, the vertex depth could be one of the properties defined in the property list with depth threshold 2 and threshold fraction 0.3. Then, GraphIn would switch to static BFS recomputation if $> 30\%$ of inconsistent vertices have BFS depth < 2, in line with the idea that if updates affect a large number of vertices closer to the root of the BFS tree, its better to run a full static recomputation. Note that these thresholds for static recomputation are algorithm and dataset dependent user-tunable parameters that require training to derive their optimal values.

3.6 Phase IV: Incremental GAS Computation (I-GAS Engine)

The incremental GAS (or I-GAS) phase ensures that only the inconsistent part of the graph is recomputed incrementally, not the entire graph. The I-GAS Engine identifies the overlap between two consecutive versions of the evolving graph and incrementally processes the graph by opening only the new computational frontier. Here the user implements the I-GAS program as well as the *activate_frontier()* and *update_inconsistency_list()* APIs whose prototypes are: *activate_frontier*(G, inconsistency_list), *update_inconsistency_list*(G, inconsistency_list, frontier).

As shown in Algorithm 1, the I-GAS loop is comprised of three basic steps that are iterated over until the inconsistency list becomes empty. It starts with a set of inconsistent vertices and calls *activate_frontier()* to activate or open the next computational frontier using the vertex priority defined in Phase II and then runs an I-GAS program. An I-GAS program consists of incremental versions of the gather, apply and scatter functions. By default, the I-GAS program is the same as the GAS program for static execution, but the user can choose to override it if necessary. Finally, the new computational frontier information is used to update the vertex inconsistency list.

Graphs	Type	# Vertices	# Edges
RMAT Scale 20 (G1M16M)	Synthetic	1,048,576	15,700,394
RMAT Scale 21 (G2M32M)	Synthetic	2,097,152	31,771,509
RMAT Scale 22 (G4M64M)	Synthetic	4,194,304	64,155,735
Facebook (FB) [29]	Real World	2,937,612	41,919,708
LiveJournal (LJ) [2]	Real World	4,847,571	68,475,391

Table 2. Graph Datasets.

3.7 Phase V: Merge Graph States (Graph Merger)

This phase is responsible for both merging updated vertex property information (e.g., new vertex depths calculated in incremental BFS) and inserting/deleting edges into the most recent version of the static graph G, thereby generating the next version of the graph. GraphIn can perform the merger in several ways to accommodate different types of graph algorithms and different levels of tolerance for inconsistency in the system.

Some incremental algorithms must accommodate both inserts and deletes from the latest update batch in each iteration of the algorithm. These updates must be applied to G before the next update batch is considered. In general, this is the case for graph algorithms that calculate global properties, like BFS, which needs to consider any added or removed edges before recalculating vertex depths. Other algorithms, often ones that compute properties that are semi-localized within a graph, need to only accommodate deletes e.g. in connected components algorithm, insertions can be handled by creating a component graph [8, 15] G' in the Phase II (see Table I), where each edge insertion (u, v) in G results in an edge in G' if u and v belong to separate components or results in a self-edge, which is ignored in the component graph. Therefore, a batch of insertions has no dependency on insertions from prior batches; they can be processed at any point (e.g., deferred) without affecting the accuracy of results. But deletes must be applied before the next update batch is considered. Finally, there are algorithms where each incremental iteration has no dependency on inserts or deletes from the previous batch. This is often the case for algorithms that only utilize local properties. Examples include the computation of clustering coefficients [6], triangle counting, vertex degree, etc. In such cases, both inserts and deletes may be deferred. Based on these observations, we support three merge patterns:

1. **All-Merge:** Both inserts and deletes are merged with static graph G.

2. **Partial-Merge:** Either deletes or inserts are merged with G. The framework defers applying the rest of the updates to the original graphs.

3. **No-Merge:** Neither inserts nor deletes from the update batch are merged with G. The framework defers applying both inserts and deletes.

4 Experimental Evaluation

4.1 Experimental Setup

Evaluation Platform. GraphIn is evaluated[3] on a dual-socket Intel node equipped with two Intel® Xeon® [4]E5-2608L six-core processors running at 2.0 GHz with 64 GB of DDR4 RAM. We used the Intel® C++ Composer XE 2015 Compiler[5] to compile the native and benchmark codes. We used GraphMat [27] and STINGER [6–8] for performance comparisons. In order to utilize multiple threads on the CPU, GraphIn, GraphMat and STINGER all use OpenMP. Updates are provided in batches of size ranging from 10,000 up to one million with 1% of all updates being deletions (except for CC where we use only insertions). The endpoints of the edges used for batch updates are generated randomly.

Graph Datasets. As shown in Table 2, we evaluate the performance of GraphIn using a mix of real-world and synthetic datasets. The synthetic datasets are obtained from the Graph500 RMAT data generator [16] for scales 20, 21 and 22 with an average degree of 16 per vertex.

Evaluated Algorithms. Three widely used graph algorithms are evaluated, including Clustering Coefficient (CCof), Connected Components (CC) and Breadth First Search (BFS). As shown in Table 1, these algorithms are classified as no-merge (CCof), partial-merge (CC) and all-merge (BFS) algorithm defined in the previous section. Algorithms requiring undirected graphs as inputs, e.g., connected components, are stored as pairs of directed edges.

4.2 Evaluation and Analysis

1) Benefits of Incremental graph computation: From figures 3a, 4a, and 5a we can observe that, GraphIn achieves maximum speedups of 407×, 40× and 82× over static computation across all the datasets for CCof, CC and BFS, respectively, with update batch size going as high as 1 million updates. Figures 3b, 4b, and 5b show updates per second versus batch size; GraphIn achieves up to 9.3 million updates/sec. These speedups result from the use of incremental computation in the IGAS execution model to compute the vertex properties/states for only the inconsistent vertices, as opposed to executing the graph algorithm for the entire input graph in the static case. Furthermore, we can draw key inferences

[3] Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance

[4] Intel and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

[5] Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel micro-architecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

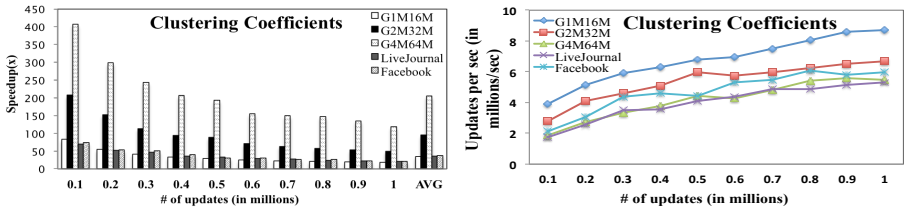

Fig. 3. a) Incremental speedup over static execution vs. update batch size b) Update rate vs. batch size for **Clustering Coefficient**.

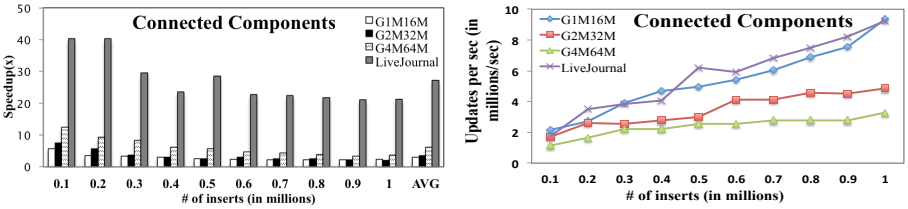

Fig. 4. a) Incremental speedup over static execution vs. update batch size b) Update rate vs. batch size for **Connected Components**.

as to how the performance of incremental execution varies with the algorithm type and update batch size, which we discuss next in detail.

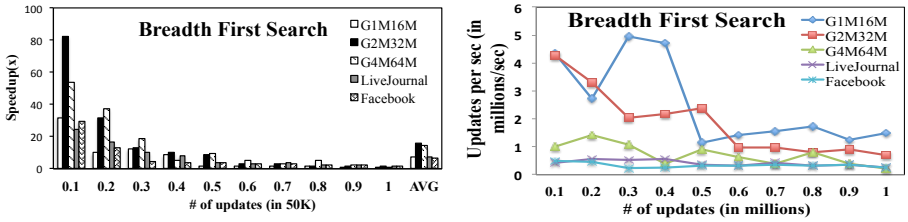

Fig. 5. a) Incremental speedup over static execution vs. update batch size b) Update rate vs. batch size for **BFS**.

Effect of graph algorithm. From Figure 3a-5a we can observe that the maximum speedups occur with no-merge algorithms, such as clustering coefficients, followed by partial-merge and all-merge algorithms like CC and BFS respectively. Note that the maximum speedup for BFS occurs when the update batch size is much smaller (∼50k updates) compared to other two incremental algorithms (∼1M updates). This variation in speedups occurs because the fraction of the graph that becomes inconsistent after applying an update batch increases in the order of no-merge, partial-merge, and all-merge. No-merge or partial-merge algorithms affect the graph locally whereas all-merge algorithms like BFS calculate a global property (i.e. depth), so the incremental computation affects a larger portion of the graph and incurs higher incremental processing time.

Effect of Update Batch Size. For a particular incremental algorithm in GraphIn, the speedup achieved falls as the update batch size increases (figures 3a-5a) because the problem size and the average incremental runtime increases with the batch size. Also, both runtime (because of decreasing speedup) and update rate increase with the batch size for CCof and CC (see figure 3b

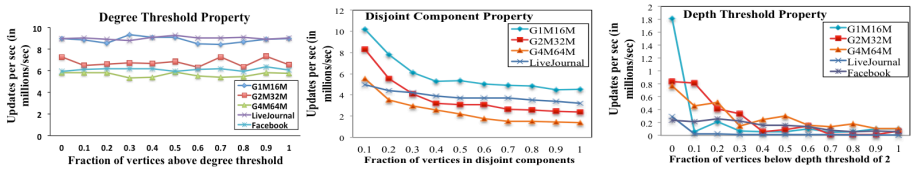

Fig. 6. Effect of a) vertex degree, b) disjoint components and c) depth value from source vertex on the update rate in CCof, CC and BFS respectively

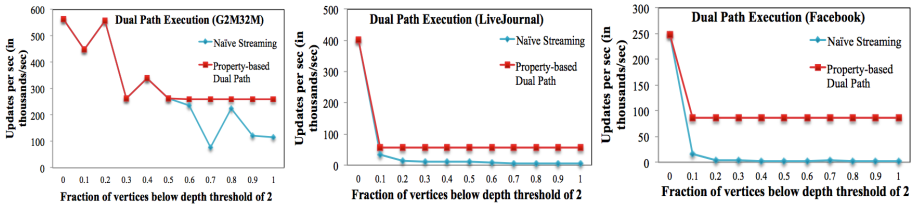

Fig. 7. Dual path execution vs. naïve streaming in incremental BFS using vertex depth property for G2M32M, LiveJournal and Facebook graph

and 4b), which implies the decrease in speedup changes slower with respect to dynamic update rate increases for larger batches. Whereas in BFS both speedup and update rate falls with increase in batch size (figure 5b) as the incremental computation affects a larger portion of the graph with the increase in batch size causing the update rate to drop.

2) Performance implications of graph properties: To demonstrate how properties of the inconsistent vertices in an update batch affect the average runtime in GraphIn, we have chosen graph properties: vertex degree (CCof), vertices with disjoint components (CC) and vertex depth from the source (BFS). The rationale behind choosing these properties is that they play a key role in the computation of the corresponding static graph algorithm we are comparing against.

Clustering Coefficient (Vertex Degree): Figure 6a shows the change in the update rate versus the fraction of vertices with degree greater than a certain threshold (e.g. 750 for the Facebook graph) that are affected by the updates. We can observe that the degree property does not have a dramatic effect on the CCof update rate which remains relatively constant. This is because CCof is a no-merge algorithm for both inserts and deletes to the graph and thus the incremental runtime is independent of the vertex degree.

Connected Components (Disjoint Components): As shown in figure 6b, the update rate for CC decreases as the fraction of edges inserted whose endpoints belong to different components in the original graph is increased, with a maximum slowdown of 3.97× across all datasets. This happens because such edges have endpoints in multiple components, meaning that there is a corresponding edge in the component graph, as opposed to self edges where endpoints of an edge fall in the same component. As described in Table 1, GraphIn reduces incremental CC to static CC processing on the component graph and increasing the number of vertices with disjoint components increases the size of the component graph and subsequently, the incremental processing time.

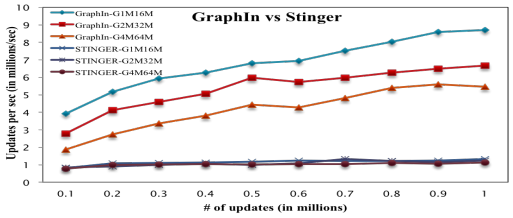

Fig. 8. STINGER Comparison for CCof

BFS (Depth): From figure 6c we can observe that increasing the fraction of vertices with depth threshold below 2 causes a sharp decline in the update rate. E.g. in G1M16M even with a small increment of 10% in inconsistent vertices below the depth of 2 results in 29x decline in update rate. When more insertions and deletions occur on these lower-depth vertices closer to the root vertex, it results in the I-GAS loop making a much larger portion of the graph inconsistent with each increment and hence the runtime increases sharply. Another observation to make here is that the largest graphs with higher diameter values are affected the most, e.g. LiveJournal, which has the maximum number of edges and the maximum diameter among all input datasets has the largest max to min ratio of update rate . For larger graphs with higher diameter the I-GAS loop iterates through larger number of BFS levels with more work per iteration (large number of edges) until the inconsistency vertex set becomes empty.

Benefits of property-based dual-path execution: Figure 7 shows how GraphIn adapts to situations where the incremental processing performs worse than static recomputation. The performance of incremental BFS for naïve (no property checks) streaming falls relative to static processing beyond a threshold fraction of vertices with depth threshold below 2. For G2M32M, LiveJournal and Facebook this threshold fraction is 0.6, 0.1 and 0.1, respectively. This degradation in incremental performance is because of a larger number of updates to these lower-depth vertices resulting in a larger portion of the graph becoming inconsistent incurring longer processing times. In such scenarios, GraphIn processes the update batch with static recomputation, which ensures that the worst-case performance of GraphIn is no worse than static recomputation. With dual path execution we achieve a maximum speedup of 60× (Facebook input).

3) Comparison with STINGER: As shown in figure 8, clustering coefficients [6] using STINGER shows a max update rate of 1.32 million versus 8.7 million updates per sec with GraphIn for the G1M16M case, which results in 6.6× speedup in throughput. This is because unlike STINGER, which uses a single edge-list based data structures for both static and incremental graph processing, GraphIn's hybrid data structure of edge-list for incremental updates and compressed matrix format for static versions of the graph enables faster updates as well as fast static computation on the clustering coefficient subproblem.

5 Related Work

Dynamic graph processing can be broken down into offline and online processing; GraphIn is a framework designed to addresses the latter problem.

Offline Processing: Chronos [11], GraphScope [26], and TEG [9] represent recent work in offline graph processing. Chronos supports incremental processing

on temporal graphs using a graph representation that places graph vertex data from different versions together leading to good cache locality. GraphScope proposes encoding for evolving graphs community discovery and anomaly detection.

Online Processing: Continuous query processing over streaming updates [30] incurs memory constraints and restricts keeping multiple versions of the evolving graph. GIM-V [28] proposes an incremental graph processing model based upon generalized iterative matrix-vector multiplication. STINGER [7] defines an efficient data structure to represent streaming graphs that enables fast, real-time insertions and/or deletions to the graph. Unlike STINGER which uses a single data structure for both static and dynamic graph analysis, GraphIn uses a hybrid data structure that allows for incremental computation on edge lists and a compressed format for static graph computation.

Static Graph Processing: Pregel [14], PowerGraph [10] and GraphLab [13] are some of the distributed frameworks that work by mapping large graphs across the combined memories of multiple machines. GraphChi [12] and X-Stream [19] are recently proposed out-of-memory frameworks that handle graphs that don't fit into a single machine's host memory. GraphReduce [22, 23] framework can efficiently process graphs that cannot fit into the limited GPU memory [20, 21] by mapping sub-graphs to the different memory abstractions of slow and fast memory [24]. These projects are complimentary to GraphIn, as they could be used to implement the static component of GraphIn while I-GAS can be leveraged to make these frameworks more dynamic.

6 Conclusion and Future Work

In this paper we present GraphIn, a high performance incremental graph processing framework for time-evolving graphs. Using its novel programming model called I-GAS, GraphIn incrementally computes the required graph properties only for the affected subgraph and thereby eliminates redundant computation. We further propose a user-tunable, property-based dual path execution optimization to choose between an incremental and a static run to achieve the best performance. Extensive experimental evaluations for a wide variety of graph inputs and algorithms demonstrate that GraphIn achieves a throughput of up to 9.3 million updates/sec and over 400× speedup compared to static graph recomputation. Using property-based dual-path execution GraphIn achieves up to 60× speedup compared to a naïve streaming approach. Future work will look at extending GraphIn to take advantage of on-node accelerators like Nvidia GPU [25] and multi-node clusters handling extreme-scale datasets [17].

References

1. Email Statistics: http://tinyurl.com/o7pch5f.
2. The University of Florida Sparse Matrix Collection: http://tinyurl.com/me4w55.
3. Twitter Statistics: http://tinyurl.com/kcuhdcw.
4. T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: A database benchmark based on the facebook social graph. SIGMOD '13, NY, USA.
5. N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

6. D. Ediger, K. Jiang, J. Riedy, and D. Bader. Massive streaming data analytics: A case study with clustering coefficients. In *IPDPSW 2010*, pages 1–8, April 2010.
7. D. Ediger, R. McColl, J. Riedy, and D. Bader. Stinger: High performance data structure for streaming graphs. In *HPEC*, Sept 2012.
8. D. Ediger, J. Riedy, D. Bader, and H. Meyerhenke. Tracking structure of streaming social networks. In *IPDPSW 2011*, May 2011.
9. A. Fard, A. Abdolrashidi, L. Ramaswamy, and J. Miller. Towards efficient query processing on massive time-evolving graphs. In *CollaborateCom*, Oct 2012.
10. J. E. Gonzalez, Y. Low, H. Gu, et al. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI '12*, Hollywood, CA, 2012. USENIX.
11. 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. EuroSys, 2014.
12. A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. OSDI'12, Berkeley, CA, USA, 2012. USENIX Association.
13. Y. Low, D. Bickson, J. Gonzalez, et al. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, Apr. 2012.
14. G. Malewicz, M. H. Austern, A. J. Bik, et al. Pregel: A system for large-scale graph processing. SIGMOD '10, New York, NY, USA, 2010. ACM.
15. R. McColl, O. Green, and D. Bader. A new parallel algorithm for connected components in dynamic graphs. In *HiPC*, Dec 2013.
16. R. C. Murphy, K. Wheeler, B. Barrett, and J. A. Ang. Introducing the graph 500. In *Cray Users Group (CUG)*, 2010.
17. S. J. Plimpton and K. D. Devine. Mapreduce in mpi for large-scale graph algorithms. *Parallel Comput.*, 37(9):610–632, Sept. 2011.
18. G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, Sept. 1996.
19. A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. SOSP '13, New York, NY, USA, 2013. ACM.
20. D. Sengupta, R. Belapure, and K. Schwan. Multi-tenancy on gpgpu-based servers. VTDC '13, pages 3–10, New York, NY, USA, 2013. ACM.
21. D. Sengupta, A. Goswami, K. Schwan, and K. Pallavi. Scheduling multi-tenant cloud workloads on accelerator-based systems. SC '14, NJ, USA, 2014. IEEE.
22. D. Sengupta, S. L. Song, K. Agarwal, and K. Schwan. Graphreduce: Processing large-scale graphs on accelerator-based systems. SC '15, NY, USA, 2015. ACM.
23. D. Sengupta, S. L. Song, K. Agarwal, and K. Schwan. Graphreduce: Processing large-scale graphs on accelerator-based systems. SC '15, NY, USA, 2015. ACM.
24. D. Sengupta, Q. Wang, H. Volos, et al. A framework for emulating non-volatile memory systemswith different performance characteristics. ICPE '15, NY, USA.
25. G. M. Slota, S. Rajamanickam, and K. Madduri. High-performance graph analytics on manycore processors. In *IPDPS '15 IEEE International*, pages 17–27, May 2015.
26. J. Sun, C. Faloutsos, S. Papadimitriou, and P. S. Yu. Graphscope: Parameter-free mining of large time-evolving graphs. KDD '07, New York, NY, USA, 2007. ACM.
27. N. Sundaram, N. Satish, M. M. A. Patwary, et al. Graphmat: High performance graph analytics made productive. *Proc. VLDB Endow.*, 8(11), July 2015.
28. T. Suzumura, S. Nishii, and M. Ganse. Towards large-scale graph stream processing platform. WWW '14 Companion, Republic and Canton of Geneva, Switzerland.
29. C. Wilson, B. Boe, A. Sala, K. P. Puttaswamy, and B. Y. Zhao. User interactions in social networks and their implications. EuroSys '09, New York, NY, USA. ACM.
30. E. Zeitler and T. Risch. Massive scale-out of expensive continuous queries. *PVLDB*, 4(11):1181–1188, 2011.