# GraphIn: An Online High Performance Incremental Graph Processing Framework

Dipanjan Sengupta, Jeffrey Young, Matthew Wolf, Karsten Schwan
College of Computing, Georgia Tech,
{sengupta6, jyoung9}@gatech.edu; {mwolf, karsten.schwan}@cc.gatech.edu

Narayanan Sundaram, Xia Zhu, Theodore L. Willke
Intel Labs, Intel Corporation
{narayanan.sundaram, xia.zhu, theodore.l.willke} @intel.com

***Abstract*—With the massive explosion in social networks, the World Wide Web, and genomics, there has been a significant growth in the interest in graph analytics. An important aspect of real world graphs like those in social networks is that they are dynamic and are constantly evolving over time. Most prior work processes dynamic graphs by first storing the updates and then repeatedly running static graph analytics on them. Because of the extreme scale of real-world graphs and the high rate at which they evolve, it is undesirable and computationally infeasible to repeatedly run such static graph analytics on a sequence of versions, or snapshots, of the evolving graph. To address the problem of analyzing evolving graphs in near real-time, we propose a high performance 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 that is based on the gather-apply-scatter programming paradigm and that allows for implementing a large set of incremental graph processing algorithms seamlessly across multiple CPU cores. We further propose a novel property-based dual path execution optimization in the GraphIn framework to choose between an incremental vs static run over a particular update batch to achieve the best performance. Extensive experimental evaluations for a wide variety of graph inputs and algorithms demonstrate that GraphIn achieves up to 9.3 million updates/sec and over 400x speedup compared to static graph recomputation.**

***Keywords-* *Graph; Big data; Performance; Incremental Processing***

## I. 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 graph processing has grown substantially. Some examples of graph analytics include friend/product recommendations, anomaly and trend detection, online advertisement serving etc. This recent trend has given rise to many graph processing frameworks in both distributed, e.g. GraphLab[8], PowerGraph[9], Pregel[10], Giraph[11]; and single machine shared-memory environments, e.g. GraphMat[25], Graphchi[12], X-Stream[13], Galois[14], Ligra[15] etc. An important aspect of real-world graphs like facebook friend lists or twitter follower graphs is that they dynamically change with time. With billions of Facebook users sharing more than 100 billion photos and posts per month, billions of tweets from twitter, millions of blogs and their follower graphs, there is a huge need to quickly analyze this high velocity streaming graph data. Current graph analytics on such dynamic graphs follow a store-and-static-compute model that involves first storing batches of updates to a graph applied at different points in time and then repeatedly running static graph computations on multiple versions or snapshots of this *evolving* graph sequence. The key assumption made here is that the rate of change in graphs due to continuous updates is slower than the execution time of the static graph analytics. This assumption might not hold true for current real-world graphs. For instance, Facebook's representative social graph benchmark [31] is built on a self-reported update rate of 86,400 objects/second in 2013, while Twitter traffic [30] can peak at 143 thousand tweets (and associated updates) / per second and emails sent per second [32] can reach as high as 2.5 millions/sec. Hence, there are two fundamental challenges to applying static recomputation to these types of rapidly changing data sets. First, static graph analytics on a single version of the evolving graph, even when leveraging massive amount of parallelism offered by multiple cores in a high performance cluster, can be very slow due to the extreme scale of many real-world graphs (e.g., one Facebook graph purportedly has a trillion edges [28]) and/or because of the complexity of the graph queries that are traditionally both compute and memory intensive. Therefore, the cumulative cost of analyzing such large-scale versions with complex graph queries repeatedly can be substantially high. Second, there are real world graph analytics problems that inherently require soft or hard real time guarantees, e.g., real-time anomaly detection, disease spreading etc. So to conclude, the current high volume and velocity of graph data combined with the complexity of user queries has outstripped the traditional static graph analytics model on streaming graphs.

To address the above mentioned computational crisis in dynamic graph processing we propose an incremental graph processing framework called GraphIn. GraphIn 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 deletions) as a sequence of batches. The proposed I-GAS programming model is based on the popular gather-apply-scatter programming paradigm introduced in Pregel and PowerGraph for implementing a large set of incremental graph processing algorithms. Using I-GAS, an incremental graph problem can be reduced to a sub-problem, in GAS paradigm, to be run on a sub-graph of the current version of the evolving graph. Because the incremental logic, in many practical scenarios, affects only a portion of the graph, this reduction can result in tremendous performance benefit compared to static recomputation of the graph algorithm on the entire graph for many popular graph algorithms and real-world graphs.

Further, there are scenarios when updates to the graph affects a very large portion of the graph and incremental processing will not help much or may even be worse (due to overheads of incremental execution) compared to a static recomputation. As an example, in incremental BFS, updates that affect vertices close to the source node affect nearly the entire BFS tree. In this case, the incremental run can at best perform as good as the static re-run. To handle such scenarios, we introduce the notion of property-based dynamic switching between incremental and static graph processing called dual-path execution. GraphIn analyzes each update batch using certain built-in and user defined properties (e.g., vertex degree information) and dynamically decides whether to process the graph incrementally using I-GAS or to fall back to static recomputation.

Finally, GraphIn's design allows it to run on top of, and take advantage of, any GAS-based static graph analytics framework as its core. We build GraphIn on top of GraphMat [25], a high performance, open source graph processing framework. It is important to note that while we have used GraphMat in this work, the ideas for GraphIn translate well to other graph frameworks as well (such as Graphchi) and can under the hood seamlessly map to multiple cores in order to leverage available hardware parallelism.

Our contributions include:

- A high-performance incremental graph processing framework built on top of GraphMat, a static graph processing engine to process evolving graphs by avoiding the naive static graph recomputation approach.
- 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 to run on multi-core systems achieving a speedup of up to 3.4x with 12 cores.
- A property-based dual path execution optimization to dynamically decide between static and dynamic graph execution based on user defined and built-in graph properties resulting in speedups of up to 60x over a naïve streaming approach.
- An extensive evaluation of 3 popular graph algorithms on large scale real-world and synthetic graph datasets with various characteristics to measure the benefits of the framework and proposed optimization over static recomputation of snapshots of evolving graph over time. Compared to competitive frameworks such as STINGER, GraphIn achieves a speedup of up to 6.6x and over all achieves up to 9.3 million updates/sec and 400x speedup over static graph processing.

## II. BACKGRAOUND AND MOTIVATION

The Gather-Apply-Scatter (GAS) computational model is used by Pregel [10], PowerGraph [9], and GraphLab [8]. With GAS, a problem is described as a directed (sparse) graph, G = (V, E), where V denotes the vertex set and E denotes the directed edge set. A value is associated with each vertex $v_i \in V$, and each directed edge $e_{ij}$ is associated with a source vertex $v_i$ and a target vertex $v_j$: $e_{ij} = (v_i,v_j) \in E$. Given a directed edge $e_{ij} = (v_i,v_j)$, we refer to $e_{ij}$ as vertex $v_j$'s in-edge, and as vertex $v_i$'s out-edge. As shown in much prior work [9, 10], the GAS model is not only simple to use, but it is also sufficiently general to express a broad set of graph algorithms, ranging from PageRank to Connected Components, and from Heat Simulation to Sparse Linear Algebra.

Graph programs written as vertex programs typically follow a 3-phase Gather-Apply-Scatter (GAS) model. In the SCATTER phase, vertices can send a message to their neighbors along their edges (in-edges and/or out-edges). In the GATHER phase, incoming messages are processed and combined (reduced) into one. In the APPLY phase, vertices use the combined message to update their property/state.

At a higher level, the typical GAS computation has three stages [10]: (1) Initialization, (2) Iterations, and (3) Output. Initialization deals with initializing vertex/edge values and a starting computation frontier, which is defined as the set of active vertices for a given iteration. In the Iteration stage, a sequence of iterations are run, each gathering the values seen on the incoming edges, updating the values of elements, and then defining a new frontier for the next iteration. This process continues until the vertex state does not change anymore.

GraphMat [25] complements the use of the GAS model in GraphIn by taking vertex programs and compiling them into sparse matrix operations (typically generalized sparse matrix-sparse vector multiplication), and graph programs are specified by at least 4 user-defined functions (UDFs). The standard four functions are SEND_MESSAGE, PROCESS_MESSAGE, REDUCE and APPLY. Of these, Send_Message specifies the Scatter phase, whereas Apply refers to its eponymous phase. Process_Message and Reduce functions are required to characterize the Gather phase.

## III. DESIGN CHOICES

There are two major scenarios in terms of analysis of evolving graphs: 1) Offline evolving graph processing where multiple versions of the graph are stored and analyzed to observe the change in certain graph properties over time. 2) Online evolving graph processing that involve real-time continuous query processing over streaming updates on the evolving graph. GraphIn is a framework designed for online graph analytics. For the rest of the paper, evolving graph processing implies online graph analytics.

There are broadly three key characteristics of evolving graphs that dictate the design decisions for our framework:

1. Computation overlap in a sequence of evolving graph versions
2. Data or working set overlap in a sequence of evolving graph versions
3. Static vs dynamic execution runtime

### A. Computation overlap and Programming model

Between multiple versions or snapshots of an evolving graph the vertex states or values 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, while calculating out-degree of vertices, an addition or deletion of edge (vi,vj) only makes vertex vi inconsistent. For BFS however, addition of edge (vi,vj) makes vj and all vertices that are "downstream" from vj inconsistent. One can consider the entire vertex set V to be inconsistent by default. In many scenarios, changes affect only a very small subset of the graph (e.g. calculating out-degree) and computing the vertex states only for those inconsistent vertices and feeding the vertex states from the previous graph versions for the rest of the vertices will significantly reduce the computation time. This observation is the key motivation of the proposed I-GAS programming model for incremental graph processing that is discussed in detail in the system architecture section. To reduce overheads, I-GAS builds a set of inconsistent vertex sets and sub-graphs that are affected by an update batch and then reduce the incremental graph problem to a sub-problem in the GAS model.

### B. Working set overlap and data structure choice

When choosing the data structure to store the evolving graph with n vertices and m edges we have multiple options. *Adjacency matrices* allow for fast updates with both insertions and deletions taking O(1) time but require a lot of space of the order of O($n^2$). *Adjacency lists* are space efficient with O(m+n) space and allow fast updates but graph traversals are very inefficient due to non-contiguous memory nodes in the adjacency edge list. Compressed Sparse Row (CSR) formats provide both space efficiency combined with fast traversal (often can be easily parallelized) by storing offsets rather than all the valid fields in the adjacency matrix. But inserts and deletes are very expensive because each update requires shifting of the graph data throughout the compressed array to match the compressed format.

Another key observation to make here is that there is a huge overlap in the edge and vertex set between consecutive versions of an evolving graph. Formally, if the graph evolved from G to G' in certain time epoch t and let $\delta_1$ = G' - G (insertions), $\delta_2$ = G - G' (deletions) then G ∩ G' = G - $\delta_2$ = *G'* - $\delta_1$ is the overlap between the working set of the two consecutive versions.

In order to allow faster updates to the graph and run both the incremental and static graph algorithms more efficiently, GraphIn uses a *hybrid data structure* involving edge-lists to store incremental updates and compressed matrix format to store the previous static version of the graph. As mentioned above, the edge-list allows for faster updates, and it does this 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 section IV).

### C. Static vs Dynamic Runtime

Runtime of online graph analytics varies widely depending on the algorithm and the update list. There are scenarios when the incremental algorithm affects only a small or local portion of the graph (e.g., makes a small subset of the graph inconsistent) and changes to the graph require accesses proportional to the size of the update batch. As shown in [3, 4, 26], per-vertex properties that depend on a fixed radius affect only a local portion of the graph and hence the runtime is proportional to the update batch size (e.g., for triangle counting and in-degree calculation). On the other hand there are classes of incremental algorithms where the graph property depends on paths and can cause a large portion of the graph to become inconsistent, resulting in a complete recomputation of the graph. In this scenario, incremental processing will not achieve any performance benefit over static recomputation and might even result in a performance degradation due to the overheads associated with incremental execution. To handle both the scenarios efficiently, GraphIn uses heuristics to select either the incremental or static execution pathway. The decision is made dynamically and taken based on a set of built-in or user-defined graph property checks (e.g., vertex degree information) and the fraction of inconsistent vertices in the update batch that meet the criteria. More precisely, if the update is predicted to require access to and/or affect a small portion of the graph then the incremental execution path is taken, otherwise the update is merged with the static graph and the entire graph is recomputed. Going back to the BFS example, if 90% of the inconsistent vertices in an update batch are of high degree, a large portion of the graph is likely to be impacted, so the static execution path is taken. The metadata that is used to decide between static and incremental graph execution is discussed in detail in the architecture section.

## IV. GRAPHIN FRAMEWORK

The GraphIn framework can efficiently process evolving graphs that incrementally change over time due to edge or vertex insertions and/or deletions. The continuous stream of updates is divided into fixed size batches before being

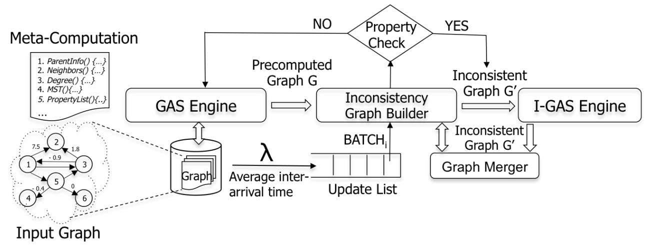


Figure 1: GraphIn Software Architecture

processed by GraphIn in the order of their arrival. GraphIn simplifies evolving graph analytics programming by supporting a multi-phase, dual path execution model described in detail in the following sections. Figure 1 shows the general software architecture of GraphIn which consists of five major components: GAS Engine, Inconsistency Graph Builder, I-GAS Engine and Graph Merger. All of the components support GAS-based APIs.

## *A. User interface*

As shown in Table I, 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 updates and run it on the target multi-core system. The user-defined functions include *meta_computation()*, *build_inconsistency_list()*, *CheckProperty()*, *frontier_activate()*, *update_inconsistency_list()* and *merge_state()*, corresponding to the different phases of GraphIn computation described below.

Five phases in the incremental graph computation:

1. Static graph preprocessing phase: Runs the GAS-based static version of the graph algorithm and any optional meta-computation information to be used later in incremental processing.
2. Inconsistency Graph Builder: Creates a list of vertices and optionally a user-defined inconsistent subgraph G' whose states became inconsistent after applying a particular update batch.
3. Property Check: Using the user-defined and in-built property list examines the current update batch to proactively decide whether to run incremental or static recomputation of graph algorithm.
4. Modified incremental GAS loop (I-GAS): Incremental version of GAS programming model that moves the computational frontier one step per iteration.
5. Merge Graph State: Merge the incremental and static graph states.

Figure 2 shows various phases in incremental BFS. We next describe each of these phases in detail.

## *B. Phase I: Static graph computation (GAS Engine)*

This phase is responsible for running 1) Static graph computation in GAS programming model and 2) Meta-computation to be used later in the incremental logic. The

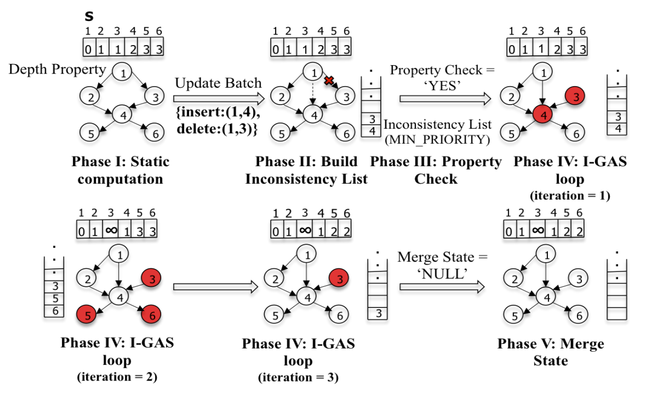

S
Depth Property
Update Batch
{insert:(1,4), delete:(1,3)}
Property Check = 'YES'
Inconsistency List (MIN_PRIORITY)
Phase I: Static computation
Phase II: Build Inconsistency List
Phase III: Property Check
Phase IV: I-GAS loop (iteration = 1)
Phase IV: I-GAS loop (iteration = 2)
Phase IV: I-GAS loop (iteration = 3)
Merge State = 'NULL'
Phase V: Merge State

Figure 2: Incremental BFS Phases

static graph computation follows the gather-apply-scatter (GAS) programming model and therefore any graph processing framework that supports GAS can be used as a Static Engine. We have used GraphMat as the core for our static graph computation. Meta computation (meta_computation()) involves the computation of static graph properties, like parents, neighbors, vertex degree, and minimum spanning tree (MST), which are needed by incremental algorithms for later phases of GraphIn. As an example, incremental BFS requires parent information during Phase IV.

## *C. Phase II: Inconsistency Graph Builder*

This is the phase where incremental graph processing begins and is responsible for marking the portions of the graph that become inconsistent after applying an update batch using the user-defined function *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.

## *D. Phase III: Property Check (Static vs Dynamic)*

As discussed before, 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 these classes, incremental processing will not achieve any performance benefit over static recomputation and may even result in performance degradation due to the overheads associated with incremental 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., degree, neighbor info, depth etc.) or define their own properties that they believe affect the runtime of the incremental algorithm in question. The framework will then

TABLE I. IMPLEMENTING APPLICATIONS IN GRAPHIN

| Application | GraphIn Phases and APIs | | | | | | |
|---|---|---|---|---|---|---|---|
| | | Phase I | Phase II | Phase III | Phase IV | | Phase V |
| | Type | *meta_computation()* | *build_inconsistency_list()* | *CheckProperty()* | *frontier_activate()* | *update_inconsistency_list()* | *merge_state()* |
| BFS [27] | All-merge | Parent id and vertex degree | **1.** Inconsistency list contains vertices with incorrect depth values with MIN_PRIORITY. **2.** G' = G | Check BFS depth property | Activate inconsistent vertices with minimum depth value-Ramalingam and Reps [27] | Remove frontier vertices and add inconsistent successors to inconsistency list | **1.** Apply all insertions and deletions to G. |
| Connected Components (CC) [4] | Delete-only-merge | Vertex degree | **1.** For each edge insertion add an edge in G' if the endpoints belong to different components [4, 26]. **2.** G' is also known as component graph. | Check disjoint component property | Activate all the vertices in G' | Clear inconsistency list | **1.** Apply only deletions to G. **2.** Relabel components in G using G' |
| Clustering Coefficient (CCof)[3] | No-merge | Vertex degree | **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 [3]. | Check vertex degree property | Activate all the vertices in G' | Clear inconsistency list | **1.** Applying insertions and deletions to G not required **2.** Update triangle counts and degree information in G using G' |

use the selected properties to decide whether to run incremental or static recomputation by calling the *property_check()* API for each update batch. This is called *property-based dual path execution*. *property_check()* 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, when running incremental BFS, the vertex depth could be one of the properties defined in the property list with property threshold depth of 2 and threshold fraction 0.3. In this case, GraphIn would switch to static BFS recomputation if >30% of inconsistent vertices have BFS depth < 2, in line with the idea that if an update affects a large number of vertices closer to the root of the BFS tree, it's better to run a full static recomputation. Note that the thresholds for these properties and the fraction of inconsistent vertices above (or below) which GraphIn would trigger a static recomputation are user-tunable parameters that are algorithm and dataset dependent and require training to derive their optimal values.

## E. *Phase IV: Incremental GAS Computation loop (I-GAS Engine)*

Incremental GAS or I-GAS phase ensures that only the inconsistent or affected 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 *frontier_activate*() and *update_inconsistency_list*() APIs whose prototypes are:
*frontier_activate*(G', inconsistency_list)
*update_inconsistency_list(G', inconsistency_list, frontier)*
As shown in Algorithm I, 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 *frontier_activate*() 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 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.

**Algorithm I: I-GAS computation loop per update batch**

```
while(!inconsistency_list.isempty())
  frontier = frontier_activate(G', inconsistency_list)
  IGAS(G')
  update_inconsistency_list(G', inconsistency_list, frontier)
```

## F. *Phase V: Merge Graph States (Graph Merger)*

This phase is responsible for both merging updated vertex property information (e.g., merging the new vertex depths calculated in incremental BFS with the original vertex property vector) 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

TABLE II. GRAPH DATASETS

| Graph Dataset | *Graph Properties* | | |
|---|---|---|---|
| | **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) | Real World | 2,937,612 | 41,919,708 |
| LiveJournal (LJ) | Real World | 4,847,571 | 68,475,391 |

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 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 over properties that are semi-localized within a graph, need to only accommodate per-batch deletes. Take for example a connected components algorithm, an algorithm that computes maximally-connected subgraphs. Insertions in connected components can be handled by creating a component graph [4, 26] 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 [3], triangle counting, calculating 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 G at the end of the I-GAS loop.

**2. Partial-Merge:** Either deletes or inserts are merged with G at the end of the I-GAS loop. 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 at the end of the I-GAS loop. The framework defers applying both inserts and deletes.

## V. EXPERIMENTAL EVALUATION

### *A. Experimental Setup*

**Evaluation Platform:** GraphIn is evaluated[1] on a dual-socket Intel node equipped with two Intel® Xeon® [2]E5-2608L six-core processors running at 2.0 GHz with 64 GB of DDR4 RAM and 16 MB of LLC. The host machine is running Ubuntu 14.04 with a 3.16 kernel. We used the Intel® C++ Composer XE 2015 compiler[3] to compile the native and benchmark codes. We used GraphMat [25] and STINGER [2, 3, 4] for performance comparisons. In order to utilize multiple threads on the CPU, GraphIn, GraphMat, and STINGER all use OpenMP. All the runs are compiled with the highest optimization level flag. Updates are provided in batches to GraphIn and STINGER where each batch size can range 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 Dataset.** As shown in Table II, we evaluate the performance and efficiency of GraphIn using a mix of real-world and synthetic datasets. Real-world datasets include Facebook interaction graph [16] and LiveJournal graph from the University of Florida Sparse Matrix collection [17]. The synthetic datasets are obtained from the Graph500 RMAT data generator [18] using scale 20, 21 and 22 graphs with average degree of 16 per vertex, labeled as G1M16M, G2M32M, and G4M64M, respectively.

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

### *B. Evaluation and Analysis*

*1) Benefits of Incremental graph computation:* In this set of experiments we showcase the benefits of incremental graph analytics over the offline static recomputation. From figures 3a, 4a and 5a we can observe that, GraphIn achieves maximum speedups of 407x, 40x and 82x over static computation across all the datasets for CCof, CC and BFS, respectively, with update batch size going as high as

[1]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

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

[3]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

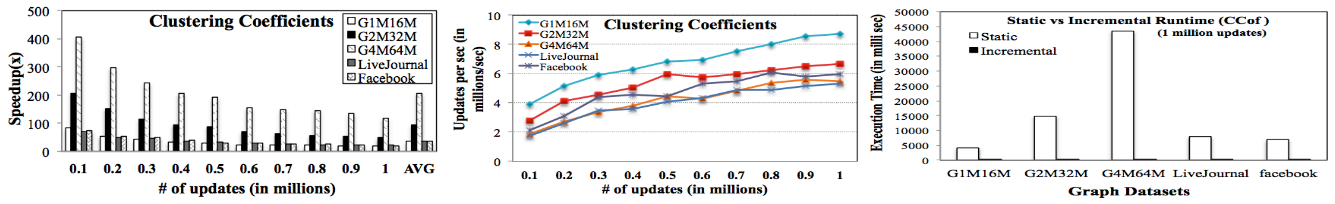


Figure 3: a) Incremental speedup over static execution vs. Batch size b) Update rate vs. Batch Size c) Incremental vs. Static Runtime for **CCof**

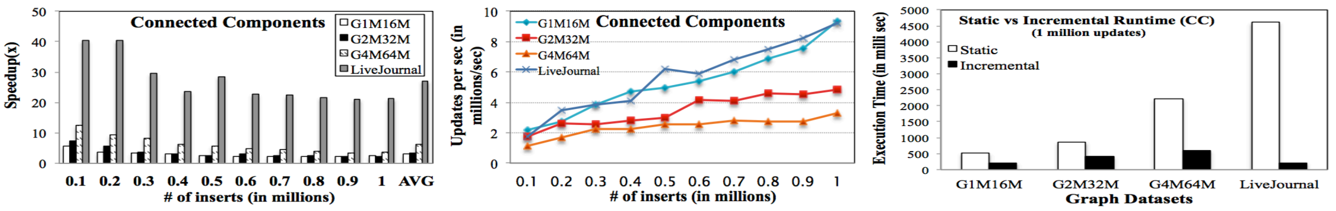


Figure 4: a) Incremental speedup over static execution vs. Batch size b) Update rate vs. Batch Size c) Incremental vs. Static Runtime for **CC**

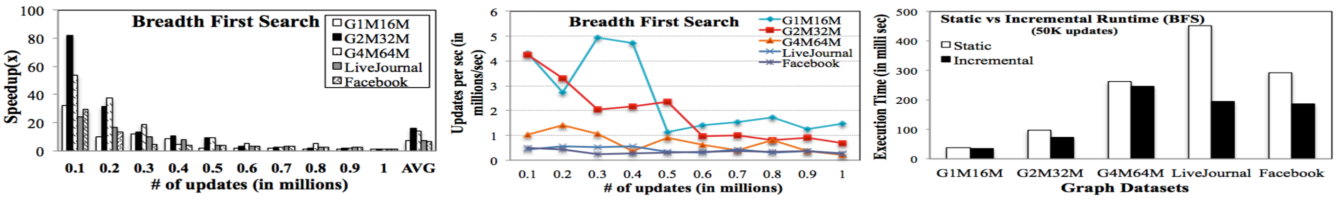


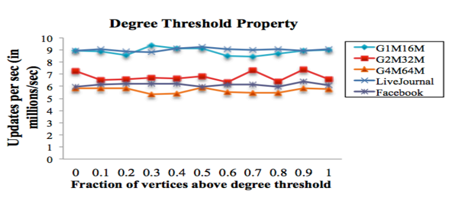
Figure 5: a) Incremental speedup over static execution vs. Batch size b) Update rate vs. Batch Size c) Incremental vs. Static Runtime for **BFS**

1 million updates. Figures 3b, 4b, and 5b show updates per second versus batch size; GraphIn achieves up to 9.3 million updates/sec. Figure 3c, 4c and 5c compares the incremental and static runtimes for CCof, CC and BFS for a fixed batch size. 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 as to how the performance of incremental execution varies with the algorithm type and update batch size.

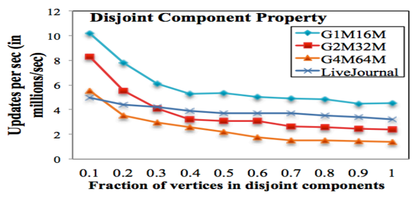
**Effect of graph algorithm.** The maximum speedups achieved over static recomputation occur with no-merge algorithms, such as clustering coefficients, followed by partial-merge and all-merge algorithms like CC and BFS respectively. Figure 5a shows that the maximum speedup of 82x for BFS occurs when the update batch size is much smaller (~50k updates) than the other two incremental algorithms (~1M updates). Again from the runtime comparison of incremental and static processing, shown in figure 3c, 4c, and 5c, we can observe that the relative performance of CCof is highest (119x) compared to the other two algorithms (21x and 2.3x for CC and BFS respectively). This variation in speedups across different algorithm types is because the fraction of the graph that becomes inconsistent after applying an update batch as the I-GAS loop unfolds increases in the order of no-merge, partial-merge, all-merge. In other words, no-merge or partial-merge algorithms affect the graph locally, so the incremental runtime is bounded by the size of the update batch. On the other hand, all-merge algorithms like BFS calculate a global property (i.e. depth), so the incremental computation affects a larger portion of the graph with a larger update batch size.

**Effect of update batch size.** For a particular incremental graph algorithm in GraphIn, the speedup achieved falls as the update batch size increases (Figures 3a-5a) because the problem size and hence the average incremental runtime increases with increasing the batch size. We can also observe that both runtime (because of decreasing speedup) and update rate increases with the batch size for CCof and CC (see figure 3b and 4b), which implies the decrease in speedup changes slower with respect to dramatic update rate increases for larger batches. Whereas in BFS both speedup and update rate falls with increase in batch size (Figure 5b). In case of BFS, the incremental computation affects a larger portion of the graph and causes the update rate to drop with respect to increases in batch size.

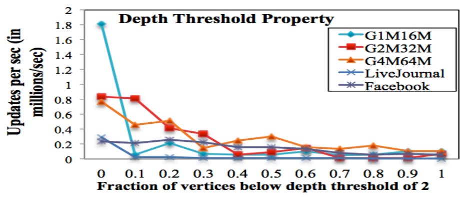
2) *Performance implications of graph properties:* In this section we evaluate how properties of the inconsistent vertices from a given update batch affect the average runtime and hence the update rate of incremental graph processing. This is the motivation for the dual-path execution of GraphIn (static recompute vs. incremental) where the framework decides on the correct path by examining the current update batch using pre-defined and/or user-defined graph properties. To demonstrate how these properties affect 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



a) Vertex Degree

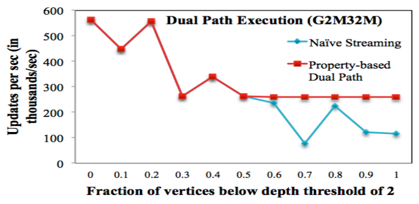


b) Disjoint Components

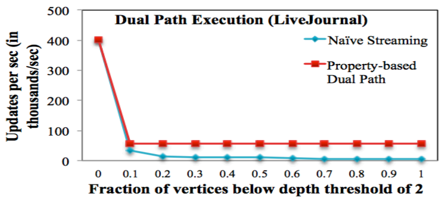


c) Depth value from source vertex

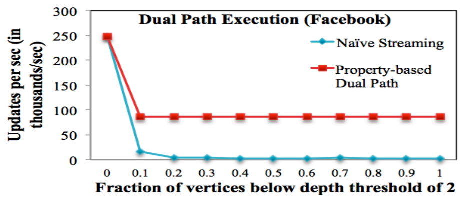
Figure 6: a) Effect of vertex degree, disjoint components and depth value from source vertex on the update rate in CCof, CC and BFS respectively



a) G2M32M



b) LiveJournal



c) Facebook

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

key role in the computation of the corresponding static graph algorithm we are comparing against. E.g. following is the expression to calculate clustering coefficient $C_v$ for vertex $v$

$$C_v = \frac{T_v}{d_v(d_v-1)} \quad (1)$$

$T_v$ = Total number of triangles in a graph with vertex $v$ as one of the endpoints.

$d_v$ = The degree of vertex $v$.

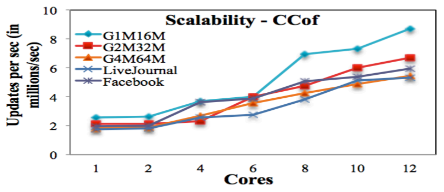
Hence vertex degree property is evaluated here for clustering coefficient due to its importance in its computation.

**Clustering Coefficient (Vertex Degree):** Figure 6a shows the change in the update rate for CCof versus the fraction of updates (insertions and deletions) affecting vertices with degree greater than a certain threshold (e.g. 750 for facebook graph). More precisely, we vary the fraction of inconsistent vertices with degree higher than the threshold degree in a given update batch and then evaluate its effect on the incremental runtime and the update rate. From figure 6a, we can see that the degree property does not have a dramatic effect on the CCof update rate. This is because CCof is a no-merge algorithm for both inserts and deletes to the graph, and the size of the sub-graph G' created in Phase II (see Table I), and thus the incremental runtime, is independent of the vertex degree. Therefore, the update rate remains relatively constant even when more edges are inserted and/or deleted near high degree vertices or supernodes.

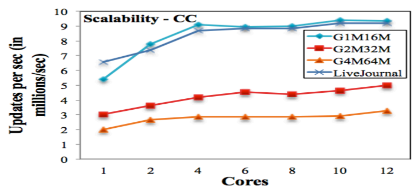
**Connected Components (Disjoint components):** From figure 6b we observe that the update rate for CC decreases as the fraction of edges inserted whose endpoints belong to different components in the original graph is increased, with a maximum slowdown of 3.97x across all datasets. This happens because 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 shown in Table I, GraphIn reduces incremental CC to static CC processing on the component graph and increasing the number of vertices with disjoint components proportionately increases the size of the component graph and subsequently, the incremental processing time.

**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. This slowdown results from more insertions and deletions on these lower-depth vertices closer to the root vertex, which results in the I-GAS loop making a much larger portion of the graph inconsistent with each increment and hence the runtime sharply increases. In addition to noting the general effect of the depth threshold, another observation is that the largest graphs with higher diameter values are affected the most. For example, the maximum slowdown (max to min ratio of the update rate) of 44.56x occurs in case of LiveJournal that has both maximum number of edges and maximum diameter among all input datasets. The reason behind this is for larger graphs with higher diameter imply the I-GAS loop iterates through more 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:** Next, we show how GraphIn uses property information and adapts to situations where the incremental processing performs worse than static recomputation. The threholds for these properties and the fraction of inconsistent vertices above (or below) which GraphIn would trigger a static recomputation are user-tunable parameters that are algorithm and dataset dependent and require training to derive their optimal values. From figure 7a, 7b and 7c we can observe that the performance of incremental BFS for naive streaming



a) Clustering Coefficients



b) Connected Components

Figure 8: Scalabilty of GraphIn using CCof and CC

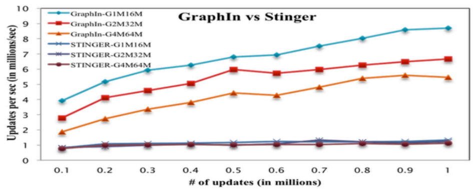


Figure 9: STINGER Comparison for CCof

without considering the property information of the current update batch 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. As discussed in previous section, this degradation in incremental performance is because a larger number of updates to these lower-depth vertices results in a large portion of the graph becoming inconsistent and hence significant increase in processing time. In phase III, GraphIn analyzes the current update batch for the depth threshold and if the batch has a fraction of vertices beyond certain thresholds, instead of proceeding to I-GAS incremental execution, processes the update batch with static recomputation. This ensures that the worst-case performance of an algorithm implemented in GraphIn has a lower bound of static recomputation runtime. We achieve a maximum speedup of 60x with dual path execution compared to naive streaming approach (Facebook).

*3) Comparison with STINGER:* Figure 9 shows the comparison between the update rate for GraphIn versus STINGER for clustering coefficients [3]. 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 6.6x speedup in throughput. GraphIn shows better scalability as batch size increases because of Graphin's hybrid data structure of edge-list for incremental updates and compressed matrix format for static versions of the graph. STINGER uses edge-list based data structures for both the static and incremental graph processing, which results in faster data structure update time but slower traversal time due to list traversal of the underlying array data structures. GraphIn's hybrid data structure thus enables faster updates (via the edge-list) as well as fast static computation on the clustering coefficient subproblem (via compressed matrix format). BFS and CC could not be compared since the publicly available version of STINGER [29] does not have incremental BFS and implements incremental CC through full static recomputation.

*4) Scalability*: Figures 8a and 8b demonstrate experiments that evaluate how the update rate varies with the number of threads of execution. In general, the update rate increases with number of threads, and the speedups achieved with 12 threads over single threaded execution are 3.4x and 1.72x for incremental CCof and CC respectively. These incremental graph algorithms implemented in GraphIn are reduced to a GAS-based graph algorithm on a subgraph (smaller problem size) of the input graph and hence can also benefit from the parallelism every GAS-based graph algorithm enjoys. CCof scales almost linearly with the threads or cores of execution because it is a no-merge algorithm while CC (Fig. 8b) flattens out at 6 cores. CC does not linearly scale because not all the insertions in CC result in an edge with endpoints in different components and hence the size of the component graph and the sub-problem size might be less than the update size.

## VI. 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 [19], GraphScope [21], and TEG [24] represent recent work in offline graph processing. Chronos is a high-performance system that 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.

**Real-time processing:** Continuous query processing over streaming updates [6] incurs memory constraints and restricts keeping multiple versions of the evolving graph. GIM-V [5] proposes an incremental graph processing model based upon generalized iterative matrix-vector multiplication. STINGER [2] defines an efficient data structure to represent streaming graphs that enables fast, real-time insertions and/or deletions to the graph and has been used to build applications like clustering coefficient [3] and connected components [4]. 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. Other real-time projects include real-time graph processing with PageRank using approximation techniques [1, 20] and a separate project that uses evolution aware clustering [7] to partition the evolving graph based on the recency of edges to speed up partitioning and merging of clusters.

**Distributed graph processing:** Distributed processing works by mapping large graphs across the combined memories of multiple machines. Pregel [10] provides a synchronous vertex-centric graph processing framework that is based on message passing. GraphLab [8] implements a framework for machine learning and data mining while

PowerGraph [9] exploits the power-law vertex degree distribution for efficient data placement and computation. ASPIRE [23] adopts an asynchronous mode of execution with a relaxed consistency to improve the remote access latency. These project are complimentary to GraphIn, in that they could be used to implement the static component of GraphIn computation while I-GAS can be leveraged to make these distributed frameworks more dynamic.

**Out-of-Memory graph processing:** GraphChi [12], X-Stream [13], and Turbograph[22] are some of the recently proposed out-of-memory frameworks that handle graphs that don't fit into a single machine's host memory. GraphChi is based on a vertex-centric implementation of graph algorithms where graphs are sharded onto SSD drives while X-Stream uses an edge-centric way to organize data for use with the GAS model. As GraphIn is framework-independent, these out-of-memory frameworks can be used as the static graph processing core for our framework.

## VII. 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, that is based on the gather-apply-scatter programming paradigm, 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 in the GraphIn framework to choose between an incremental vs static run over a particular update batch to achieve the best performance. GraphIn also enables seamless scaling of incremental graph processing to a very large dataset on massively parallel architecture. 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 400x speedup compared to static graph recomputation. Using dual-path execution GraphIn achieves up to 60x speedup compared to a naïve streaming approach. Compared to competing framework like STINGER, GraphIn achieves up to 6.6x speedup. Future work will look at extending the GraphIn framework to take advantage of on-node accelerators like Xeon Phi and multi-node clusters handling extreme-scale datasets.

## References

[1] B. Bahmani, R. Kumar, M. Mahdian, and E. Upfal. Pagerank on an evolving graph. KDD '12, pages 24–32, New York, NY, USA, 2012.

[2] D. Ediger, R. Mccoll, J. Riedy, and D. A. Bader. STINGER: High performance data structure for streaming graphs, 2012.

[3] D. Ediger, K. Jiang, J. Riedy, and D. A. Bader. Massive streaming data analytics: A case study with clustering coefficients, 2010.

[4] D. Ediger, J. Riedy, D. Bader, and H. Meyerhenke. Tracking structure of streaming social networks. IPDPSW, May 2011.

[5] T. Suzumura, S. Nishii, and M. Ganse. Towards large-scale graph stream processing platform. WWW Companion '14, Republic and Canton of Geneva, Switzerland, 2014.

[6] E. Zeitler and T. Risch. Massive scale-out of expensive continuous queries, 2011.

[7] M. Yuan, K.-L. Wu, G. Jacques-Silva, and Y. Lu. Efficient processing of streaming graphs for evolution-aware clustering. In Proceedings of the 22nd ACM CIKM '13, New York.

[8] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. Proc. VLDB Endow., Apr. 2012.

[9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: OSDI'12, Berkeley, CA. USENIX Association.

[10] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, and G. Inc. Pregel: A system for large-scale graph processing. SIGMOD, 2010.

[11] Giraph. Apache Giraph. http://giraph.apache.org/.

[12] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. OSDI'12, Berkeley, CA.

[13] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. SOSP '13, New York.

[14] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. SOSP '13, New York, NY, USA, 2013. ACM.

[15] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. PPoPP '13, pages 135–146, New York, NY, USA, 2013. ACM.

[16] C. Wilson, B. Boe, A. Sala, K. P. N. Puttaswamy, and B. Y. Zhao. User interactions in social networks and their implications. In EuroSys, 2009.

[17] T. Davis. The University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices.

[18] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500. Cray User's Group (CUG), 2010.

[19] 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 '14, New York, NY, USA, 2014.

[20] P. Desikan, N. Pathak, J. Srivastava, and V. Kumar. Incremental page rank computation on evolving graphs. WWW '05.

[21] 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.

[22] W-S. Han, S. Lee, K. Park, J-H. Lee, and M-S. Kim, and J. Kim and H. Yu. TurboGraph: A Fast Parallel Graph Engine Handling Billion-scale Graphs in a Single PC., KDD '13, New York, NY, USA, 2013.

[23] K. Vora, S. C. Koduru, and R. Gupta. Aspire: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based dsm. OOPSLA '14, New York, NY, USA.

[24] A. Fard, A. Abdolrashidi, L. Ramaswamy, and J. A. Miller. Towards efficient query processing on massive time-evolving graphs, 2012.

[25] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: high performance graph analytics made productive. *Proc. VLDB Endow.* (July 2015)

[26] R. McColl, O. Green, and D. Bader, "A new parallel algorithm for connected components in dynamic graphs," HiPC, December 2013

[27] G. Ramalingam and Thomas Reps. 1996. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms,* September 1996.

[28] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: graph processing at Facebook-scale. *Proc. VLDB Endow.* August 2015

[29] STINGER Github https://github.com/robmccoll/stinger.

[30] Twitter statistics, http://tinyurl.com/kcuhdcw

[31] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: a database benchmark based on the Facebook social graph. SIGMOD '13. ACM, New York.

[32] Email Statistics http://tinyurl.com/o7pch5f