Incremental Graph Processing for On-Line Analytics

Scott Sallinen[1], Roger Pearce[2], Matei Ripeanu[1]
[1]University of British Columbia, [2]Lawrence Livermore National Laboratory
scotts@ece.ubc.ca, rpearce@llnl.gov, matei@ece.ubc.ca

***Abstract*—Modern data generation is enormous; we now capture events at increasingly fine granularity, and require processing at rates approaching real-time. For graph analytics, this explosion in data volumes and processing demands has not been matched by improved algorithmic or infrastructure techniques. Instead of exploring solutions to keep up with the velocity of the generated data, most of today's systems focus on analyzing individually built historic snapshots. Modern graph analytics pipelines must evolve to become viable at massive scale, and move away from static, post-processing scenarios to support on-line analysis. This paper presents our progress towards a system that analyzes dynamic incremental graphs, responsive at single-change granularity. We present an algorithmic structure using principles of recursive updates and monotonic convergence, and a set of incremental graph algorithms that can be implemented based on this structure. We also present the required middleware to support graph analytics at fine, event-level granularity. We envision that graph topology changes are processed asynchronously, concurrently, and independently (without shared state), converging an algorithm's state (e.g. single-source shortest path distances, connectivity analysis labeling) to its deterministic answer. The expected long-term impact of this work is to enable a transition away from offline graph analytics, allowing knowledge to be extracted from networked systems in real-time.**

I. Introduction

We are witnessing a massive rise in globally connected data generated by an extremely diverse set of activities: from modelling networked information like the World Wide Web, to social networks and forums like Facebook and Reddit, to financial transactions such as those occurring on the Bitcoin network. When we analyze such connected data, we generally use a graph-based representation. Past research has led to remarkable advances in the processing speed and scalability of graph analytics; much of this work has used HPC systems [1][2][3] and is showcased by the Graph500 [4] and GraphChallenge [5] competitions.

However, as these past solutions were designed for static graphs, directly using them for dynamic graphs is not possible without using coarse and problematically inefficient static snapshots. This approach has multiple drawbacks: *(i)* it leads to high overheads due to storing multiple copies of the entire graph at different times [6][7] (or processing batch delta changes [8]), *(ii)* it loses information by removing the ability to query graph state in-between snapshots, and *(iii)* on occasion, it even abandons correctness (such as when events in a delta-change are modelled as concurrent, even when a total order exists).

Two factors drive our work: firstly, the increasing pressure to deliver the fast responses demanded by many applications (e.g. online recommendations, financial fraud detection, terrorism protection), and secondly, the need to model continuously evolving real-world systems represented as graphs. As examples, Twitter has seen a peak tweet rate of 143,199 tweets per second [9], the payment network Visa has been shown to handle up to 24,000 financial transactions per second [10], Facebook has activity in the range of 9,000 comments, 5,000 statuses, and 20,000 photos every second [11], and new scalable cryptocurrencies such as EOS have already seen transaction rates reach nearly 4,000 per second [12]. All these systems are naturally modelled as graphs.

To support analytics on such fast-evolving systems, graph processing systems should support both: *(i)* maintaining the topology of massive dynamic graphs, and *(ii)* near real-time analytics on these typologies. Dynamic graphs extend static graphs such that vertices and edges may be added or removed, and their attributes changed, at any point in time. Typical networked systems, such as finances, social networks, forums, and the World Wide Web all experience dynamism and evolve with time. Money moves, new friendships are made, posts and comments receive user interaction, and new web pages and hyperlinks are created. Despite the dynamism exhibited in all these systems, analytical tools that enable this dynamic nature to be captured are a relatively unexplored area. Existing systems face issues of scalability, adaptability, and latency – the existing systems that attempt to offer solutions are typically constrained to a single machine, and resort to batching or snapshotting [6][7][13].

The existence of a potentially more efficient avenue that avoids snapshotting, however, is suggested by past theoretical work on on-line algorithms for dynamic graphs [1] [14][15][8][16]. These algorithms dynamically maintain a (full or partial) solution to a user query and update it as the graph evolves. Thus, we hypothesize that new systems which aim to incorporate, support, and analyze graphs dynamically in a scalable fashion are feasible and will enable an entirely new class of opportunities for analyzing dynamic graphs: (near) real-time answers to analytics questions, and (near) real-time reactions to topology changes.

While being able to analyze and understand an evolving networked system in real-time is tantalizing, it poses a challenging problem: it is necessary to understand how to extrapolate common algorithms for static graphs into dynamic ones, design algorithms from scratch for such problems, or design entirely new algorithms for problems that emerge only in a dynamic graph context. Many graph problems such as Breadth First Search (BFS), are defined only in static terms, i.e. *"What is the level of all vertices?"* while the notion of "all vertices" has no clear definition in a dynamic environment. These types of problems in a dynamic environment are better thought of as time dependant. For example, in a human readable format, instead of asking *"What are all the vertices that I can reach?"* a dynamic algorithm would instead maintain *"What are all the vertices I can currently reach?"* This algorithm then gives a BFS result that changes over time, and hence, the answer itself is dynamic in nature, or can be thought of as "observable."

This format, wherein the observable problem solution evolves over time with the state of the data, gives rise to a query-based design for our system, where we can observe or *query* the

[1]Unfortunately, this work makes assumptions about the underlying abstract machine supporting these algorithms that preclude directly adopting these solutions – more concretely, assuming that topology events are each sequentially and atomically ingested precludes scaling to a distributed system, and may generate high overheads even in a shared-memory setup.

real-time data. Furthermore, algorithms designed for dynamic data can expand upon the traditional *"What"* questions, and can also answer *"When"* questions – a notion involving time that does not exist in a static environment. A simple example is *"When is vertex A connected to vertex B?"* offering a response in real-time based on when a condition has been met. This type of behaviour offers the ability for a time-sensitive reaction to occur. We target a design where the code implementing various algorithms is separated from the underlying infrastructure and multiple algorithms can be executed simultaneously (i.e. maintain their state) on the same underlying dynamic data structure, thus enabling support for multiple queries.

For this paper, we limit our inquiry to incremental "add-only" topological changes. Restricting the problem space allows us to focus on feasibility, algorithm design, and to explore the performance limits achievable, before adding further complexity. Additionally, many dynamic graphs in the real world are incremental-only, due to the nature of time-based systems only ever moving forward. For example, in discussion forums like Reddit, the bipartite graph between posts and users is only ever appended to as time moves forward; while a user/post visibility might change (e.g. due to moderation or privacy settings changes), the data itself is often never actually deleted. As another example, in a financial transaction network, a payment that happened in the past is never truly "reversed" (i.e. modelled as a delete) – instead a new, second payment is created for actions such as refunds or returning change.

This paper offers the following key contributions:

- *Identification of a class of algorithms that can be supported in an incremental graph context:* This paper showcases four algorithms, each designed to support live queries for incremental graphs: Breadth First Search, Single Source Shortest Path, Connected Components, and Multi S-T Connectivity (Section IV). These algorithms update the problem's solution incrementally and "on-the-fly," reacting to individual topological changes. These algorithms, which we dub REMO algorithms, belong to a class of algorithms that have key properties: *recursive updates* and *monotonic convergence* (Section II).
- *The "When" in graph processing:* We define and implement dynamic queries, which can trigger a user-defined callback on occurrence of a specified condition of an algorithm's vertex-*local* state (Section III-E). Different from *global* state (i.e. the state of the algorithm over the entire graph topology, which can also be discretized and collected as we show in Section III-D), triggers on local state offer the opportunity to add near real-time user-defined callbacks for feedback and analysis in a continuously evolving graph (Section II).
- *An event-based framework to support expressing and running algorithms:* Expanding upon a previous feasibility study for implementing a dynamic graph infrastructure [17], this paper further builds the infrastructure, including abstractions for several algorithm properties (Section III).
- *Evaluation:* This paper exposes the overheads (but also the opportunity) of executing static algorithms on a dynamically evolving data-structure, as well as the overhead of carrying a queryable, live algorithm state alongside real-time graph construction. We show that the framework can support the ingestion of hundreds of billions of edges with a maximum real-time rate of up to 1.3 billion edges per second (Section V). The showcased event rate suggests generous room for additional algorithm complexity. The paper also investigates the cost of global state collection, substantiating the intuition that it is more efficient to maintain and dynamically update algorithmic state as the graph evolves and collect it when needed, rather than use a static algorithm to construct on-demand algorithmic state from scratch (Section V-C).

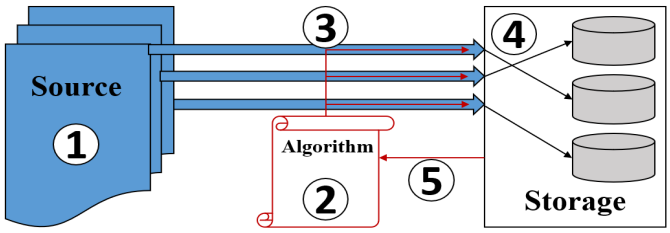

Fig. 1: *High level overview of the system. An incoming stream of events (1) that modify a graph (4) are ingested. During this process, an algorithm (2) hooks (3) into the stream, observing events (5), and updating its dynamic state accordingly.*

II. EVENT-BASED DESIGN FOR *ReMo* ALGORITHMS

To make reading easier, this section starts with a high-level overview of the how the supporting infrastructure is envisioned to work abstractly (with further detail in the subsequent section), and only then continues with discussing the challenges of supporting on-line algorithms on dynamic graphs and the subclass of algorithms that are the focus of this paper: algorithms that have two key properties, RE*cursion*, and MO*notonicity*, which we call REMO algorithms.

A. Overview of Supporting Infrastructure

We have chosen to support on-line dynamic graph processing through an event-centric design: when the graph structure or vertex/edge attributes change, the infrastructure *(i)* updates the topology information, and *(ii)* triggers an algorithmic event that allows a user-defined callback to perform the necessary algorithmic updates. Additionally, algorithmic events are not only generated upon topology changes, but can also be created by algorithmic-level behaviour as well.

We chose this event-based design for three key reasons: *(i)* while previous dynamic solutions could support serial graph changes [14][15], these solutions are sequential – each event is processed once the previous event has finished – while the natural world matches more closely to a concurrent and asynchronous system, *(ii)* an event-centric design enables deployment on a shared-nothing compute architecture, to allow scaling to support large graphs that need to be maintained in the aggregate memory of the system, and *(iii)* it offers a good match to a computing platform with multiple threads of computation.

This design is illustrated at a high level in Figure 1: a source (1) initiates asynchronous, concurrent updates to the underlying graph topology and attributes over one or multiple event streams. Events in the same stream are ordered while events across streams do not have a relative order. These events are observed by the algorithm engine (2). The algorithm engine, observing the topology/attribute change, executes user-defined logic (3) which will be passed along with the graph topology/attribute change to the storage layer which maintains state (4). Should

Algorithm 1: Recursive BFS Psuedocode

```
# Recursive BFS (recursive step)
def bfs_propagate(vertex, parent):
    if (vertex.level > parent.level + 1):
        vertex.level = parent.level + 1
        for nbr in vertex.nbrs:
            bfs_propagate(nbr, vertex)

# Setup for recursive BFS function (base case)
# Queue for propagation excluded for brevity.
def bfs(graph, start_vertex):
    for vertex in graph:
        vertex.level = sys.maxsize
    start_vertex.level = 1
    for nbr in start_vertex.nbrs:
        bfs_propagate(nbr, start_vertex)
```

the event require additional logic, it generates a new event that feeds back into the algorithm engine (5).

Finally, we note that a vertex-centric solution to support dynamic graphs would not realistically model graph evolution, as most graph evolution is edge-centric: an edge may appear between two already established vertices, and an edge may disappear without removing a vertex.

An Example. As a trivial example, consider a simple query that aims to track the degree of each vertex in a graph. In an event-centric design, we simply implement a callback on edge insertion and deletion: if an edge is added, increment a counter tracking the vertex degree; if removed, decrement it. With proper infrastructure support, a programmer will only have to write these two simple callbacks and be able to query a dynamic graph, resulting in a real-time analysis of a specific vertices degree or enabling a user-defined callback if the degree exceeds a certain threshold.

B. Recursive and Monotonic: ReMo Algorithms

At a high level, any algorithm designed for a dynamic graph will have two key issues: *(i)* it must execute on, and be tolerant of, graph topology and attribute changes, and *(ii)* it must return an answer in finite time (guarantee termination) despite future changes. These two properties make theoretical analysis of such algorithms complex, and require a rigorous understanding of the implications of a live state – as algorithm state can change at vertex/edge granularity during the processing of concurrent events.

We target a class of algorithms which we refer as REMO algorithms that have two key properties: RE*cursion*, and MO*notonicity*, which we explain in turn below. These two properties guide algorithm implementation in an event-centric system, and make it feasible to guarantee convergence to a correct result with algorithm termination.

Recursive Event Propagation. We chose a dynamic version of Breadth First Search (BFS) as our running example. BFS is a key problem for graphs, as it effectively exposes the overheads and behaviour associated with traversing a graph through neighbour iteration and is a building block of other common graph algorithms (e.g., Betweenness Centrality).

Consider an edge addition, where a new path in the graph is created: a link from vertex A to vertex B. Upon creating this link, one could imagine re-executing a static Breadth First Search traversal from the original source vertex upon the entire new graph to re-calculate the new level for all vertices. However, this would waste a large amount of work, as re-computing every value is the worst-case scenario.

Instead, consider the traditional top-down BFS algorithm in its recursive form: upon calculating the level for a vertex, propagate to all neighbours the calculated level. In a static graph, this event propagation implementation of BFS computes from the source vertex top-down, until termination, as shown in Algorithm 1. When considered this way in a dynamic context, this can extend to an edge addition event. Upon an edge addition, a user-defined callback simply propagates the vertices BFS level, as if it were the original source vertex, towards the newly created path (the termination proof is the same as for BFS).

The elegance comes from the abstraction of using the recursive step as the update function. The addition of an edge need only create a single event – after which, the event can, as defined by the callback, create and propagate new events only if necessary. For the BFS example, this can be described in a simple update event as follows in algorithm 2.

Algorithm 2: BFS update

```
# Hook into an edge being created from src to dst.
def add_event(dst, src):
    update_event(dst, src)

# Propagating update event.
def update_event(vertex, parent):
    if (vertex.level > parent.level + 1):
        vertex.level = parent.level + 1
        for nbr in vertex.nbrs:
            update_event(nbr, vertex)
```

This simple algorithm effectively becomes the BFS algorithm update within the algorithm engine: the update is initially activated by an edge event and can recursively propagate and trigger the same update event at other vertices in the graph. Note that the programmer does not have to consider how the event propagates: the complexities of the graph topology structure are hidden by the supporting framework.

This algorithm is efficient, resulting only in the execution of the computation required to "fix" the BFS tree, with no additional overhead. For example, imagine in the case where the event is passed between vertices that have the same level – in this case, the event will be triggered, but will not generate any additional events. Further, if a new vertex is added with an edge, it needs only a single step to calculate its own level. This is far more efficient than a full-scale re-computation of the algorithm over the whole graph upon each change (as we show in Section V).

Convex Monotonicity: Achieving Minimum State. The second property is *monotonicity*: a succession of events triggered by a graph topology/attribute update, both during propagation and once completed, impacts any state change of the algorithm only in a single direction. Further, there is an upper/lower bound for that state, and it is smooth – that is to say, the solution space is convex.

Monotonicity exists in BFS in the static top-down solution: the state (i.e. the BFS level of a vertex, the minimum number of hops required to reach that vertex) can only ever decrease. While an intermediate solution may present a path with N hops (thus the vertices' level being N), as the solution is refined it may only decrease – if it finds a path of length $N' < N$. At best, a vertex can reach a minimum of one hop, if it finds it can connect directly to the source. This property is what allows the BFS recursion to terminate: at some point in time, each vertex achieves its minimum state, and can make no further progress. The resulting state is the deterministic level according to the topology of the graph.

A dynamic algorithm supporting incremental edges for BFS can preserve the monotonicity property of the static algorithm.

When a new undirected edge is inserted between two vertices, it will fall into one of three cases: *(i)* it links two vertices of the same level, *(ii)* it links such that one's level = other's level + 1, or *(iii)* link such that one's level $>$ other's level + 1. (In the case of directed BFS, there are a few more trivial cases.) In case *(i)*, the current BFS solution continues to be valid and no changes need to be made. In case *(ii)*, the current solution also continues to be valid – a vertex has a new possible parent. In case *(iii)*, we have a situation where a vertex has a new, shorter path to the source. In this case, the recursion event starts at this vertex, as mentioned previously, and repairs the solution. However, all changes made continue to monotonically converge to the minimum state possible.

REMO ***Properties for Other Incremental Algorithms.*** Incremental algorithms for Single Source Shortest Path (SSSP), Connected Components (CC), and S-T Connectivity (S-T) can be designed in a similar way once the state that evolves monotonically is identified, as follows:

- *SSSP*. Monotonically evolving state: at each vertex the cost of the current minimal path to the source is stored. An algorithm to support edge addition for SSSP converges in the same way as for BFS, with the path being instead the sum of edge weights rather than the number of hops away from the source. Similar logic applies for edge updates limited only to reducing edge weight. In this case, the solution space is convex as any vertex may only become closer to the source, and there is a lower bound.
- *CC*. Monotonically evolving state: at each vertex the state is the vertex ID with the smallest label in the component it can reach. In the case of an edge addition, only one of two cases can occur: *(i)* the new edge connects vertices within the same component, or *(ii)* the new edge connects vertices between two components. In case *(i)* the change is trivial. In case *(ii)*, the component with the minimum state will propagate to the other component, and recursively apply the new possible minimum state into that component. This preserves the monotonicity of vertex state only ever decreasing convexly.
- *S-T*. Monotonically evolving state: each vertex stores the connectivity status to the source vertex. When adding new edges, a vertex may only become connected to target source vertex, directly or indirectly, and the state only evolves toward this connectivity. The state could be described by each vertex having an associated bit, where a 1 denotes no connectivity, and connecting flips it to a 0, creating a convex solution space. The same argument can be extended to multi S-T connectivity by using a bitmap.

C. Algorithm State: Global and Local State

An important facet of supporting algorithms with an event-based design is collecting results and presenting them to the application or user. We define two types of algorithm state collection: *global* and *local* state. These differ as follows.

Local state is defined as a state that can be observed on a per-vertex (and its associated edges) basis. At this level of granularity it is trivial to obtain a consistent view (by just making sure that no other events process the same vertex at the same time). Local state can be observed immediately, at a low cost, during algorithm execution. As an example, local state could be a vertex's connectivity: and a trigger/action functionality such as *"When the T vertex has a path to vertex S, perform this user-defined function"* can supported. This is further discussed in section III-E.

Global state is defined as the collective vertex and edge algorithm-related state after a defined set of events have been ingested and processed. As an example, it could refer to the entire BFS tree, and, as implied, contains global data: each vertex has a data point referring to its level in the BFS tree and its parent vertex. Global state is similar to the traditional data collection done in static graph analysis, and can be defined in terms similar to snapshotting: it is data associated with a discrete point in time, after some discrete set of events have been ingested. Designs for implementing this discretization are presented in section III-D.

D. Asynchronous Event Propagation

Asynchronous/concurrent event propagation does not impact the correctness of the above algorithms, as the algorithm state can only move in a single direction (towards minimum state), with potentially conflicting events being either independent or order-irrelevant: they change different parts of the graph independently, or the consequence of the two events can be combined or squashed. The effective solution space, when defined as convex, allows the resulting global algorithm state to arrive at the global minimum. This property is what allows concurrent events to be treated in parallel, and allows multiple update cascades to proceed concurrently.

We note that nondeterminism can be eliminated in a similar way as for static algorithms. For example, in BFS, if a vertex obtains two new possible parents, the one with minimum state will always be chosen regardless of the order of operations. However, if the parents are of equal state, and the algorithm designer wishes for a deterministic BFS tree, they need only define a second clause to discriminate between the two potential parents (similar to static algorithms, such as choosing the parent with the lowest vertex ID). With this clause, the global state at a specific time will become completely deterministic.

III. INFRASTRUCTURE SUPPORT

Enabling support for expressing and executing dynamic algorithms, as well as optimizing the distributed data structure that efficiently maintains the dynamic graph topology, are complex issues. Some of the challenges that emerge can be highlighted based on our understanding of challenges shown to emerge for static graph processing. First, graph algorithms have low compute-to-memory access ratio, which exposes fetching/updating the state of vertices (or edges) and fetching vertices topology information as the major overheads. Second, graph processing exhibits irregular and data-dependent memory access patterns, which lead to poor memory locality and reduces the effectiveness of caches and pre-fetching mechanisms. Finally, many graphs have a highly heterogeneous node degree distribution (i.e. they have power-law degree distribution, and are commonly named "scale-free") which makes partitioning the work among the processing nodes for access locality and load-balancing difficult. Enabling graph evolution further compounds these challenges, as locality and data placement optimizations need to be decided at run-time, and are not only dependent on algorithm behaviour but now also on topology evolution and time. As we model the behaviour of incoming edge-centric graph structure changes as events, we introduce new computation and storage requirements when operating in a distributed manner. The infrastructure must handle updating the

graph representation, generating and handling the corresponding application-level events, and triggering user-defined callbacks.

The rest of this section presents our event-centric programming model (Section III-A), the infrastructure to maintain the dynamically evolving graph topology (Section III-B), our solution for global and local state collection (Sections III-D and III-E), and our prototype implementation (Section III-F).

A. Programming Model

We provide a simple event-based programming model. Algorithms are designed as a set of used-defined callbacks triggered by events generated either by graph topology/attribute changes, or by other callbacks. We identify three key events [2] that are the minimum requirements to implement an incremental algorithm on a dynamic graph.

- *Edge Add* Event: The topology change of an edge addition triggers an *add* event at the source vertex of a directed edge (or at the first added vertex of an undirected edge, in which case the reverse-add always follows). Topology maintenance (Section III-B) is handled directly by the framework for *edge add/reverse-add events*.
- *Edge Reverse-Add* Event: In the case of an undirected (bi-directional) edge, the second vertex obtains a special notification event which enables it to add the edge as well.
- *Update* Event: This event does not represent a topology change, and is generated by user-defined behaviour. This mechanism gives a callback processing a vertex the ability to generate additional events, propagating information to all or to a selection of neighbouring vertices.

For each event, the programmer has the ability to define a callback to derive desired local algorithm state. Further, the programmer can decide exactly how the *update* event propagates: whether it is specifically across all edges to each neighbour, or only to some, and for which conditions. Along with all passed events, the programmer has access to key vertex properties to drive their algorithm: the *visiting vertex* being the identifier of the vertex creating the event, and it's current property at the time of event (i.e. it's own local state).

B. Node-local Topology Maintenance

Even with the large memory capacities of HPC systems, many graph applications require additional out-of-core storage. Such large memory requirements can result from a large graph, rich properties that decorate the topology, or large algorithm state. To accommodate such a data structure, we have incorporated a truly dynamic graph data structure called DegAwareRHH [18]. This graph structure adopts open addressing and compact hash tables with Robin Hood Hashing, and offers good data locality for vertices with a large degree. DegAwareRHH is degree aware, and uses a separate, compact data structure for low-degree vertices. This, in combination with the high degree access, allows distinct improvements in the number of accesses to out-of core storage (e.g. NVRAM) when needed.

The importance of this structure to store the dynamic graph data is twofold. First, the structure allows compressed, dynamic graph data to be stored in memory and spill to NVRAM only when needed, and second, it significantly improves the performance over a baseline implementation.

[2]For brevity we do not detail here additional events related to other topology changes: vertex related (which are set of edge changes), or attribute updates (which are similar to an addition).

C. Dynamic Partitioning

Partitioning the dynamic graph over a distributed set of compute nodes is challenging: since the graph is dynamic, there is no a priori information available to inform partitioning. As a first solution to explore, we have chosen a random partitioning technique for implementation simplicity and to obtain a baseline performance.

Our solution works as follows. To determine which process will be the owner of a particular vertex at run time, we use a simple form of consistent hashing [19] where we assume a cluster with a static process count P, and assign a vertex with ID V to a process via $hash(V) modulo(P)$. This way, as each process uses the same hash function, any process can determine in constant time which process owns a vertex. Consistent hashing produces a balanced, uniform partitioning in terms of the number of vertices, yet the resulting edge distribution may not be balanced – since we allocate vertices to processes, the power-law nature of many graphs [20] may create an uneven balance, with some processes responsible for vertices with many edges. However, our initial goal is simplicity; this naïve strategy allows a primary focus on the key issues related to supporting on-line algorithms and, importantly, provides a baseline lower bound for the performance that could be achieved using dynamic load balancing techniques in future work.

The key advantage of the above partitioning technique is that any process is able to insert a new directed edge at any time. (The directed edge will be co-located with the source vertex, since the source vertex must know it has an edge to the destination.) This makes it possible to support asynchronous directed edge creation and deletion.

A byproduct of this design is that we can enable the infrastructure to split the stream of incoming graph update events among all the participating nodes: each process can independently ingest pairs of [source, destination] graph structure changes (edge events) to increase ingestion throughput. However, when using multiple or split data streams, two important assumptions are made: *(i)* each individual stream presents its own events in-order, and *(ii)* events on different streams are treated as concurrent.

The mechanism we have described so far works directly for directed graphs, yet it requires additional attention for undirected graphs: since undirected edge creation leads to state updates at two vertices, these updates must be coordinated to maintain a consistent view of graph state. To this end, it is necessary to "serialize" undirected edge creation: process P sends the creation of edge $[a, b]$ to the owner A of vertex a, and then process A is responsible to send the reverse: the knowledge for the creation edge of $[b, a]$ to process B, owner of vertex b. Since the channel between processes A and B is first-in first-out (FIFO), and only the two processes are able to use the edge, we thus ensure that the edge is created before it is used at either end.

D. Collecting Global State

There are two main approaches to collect global state (previously defined as *the collective vertex and edge algorithm-related state after a defined set of events have been ingested and processed* in Section II-C). As the system handles events continuously, before actual collection occurs the desired time point must be discretized. This can be done by identifying an event for each

stream that is the last event to be processed in this collection. Then, the simple approach would be to wait for a period of quiescence. When no more events are generated as the outcome of previous events, the collection has concluded and the global state can be saved as a snapshot.

However, this simple approach would require pausing the incoming event stream. To avoid this, a continuous approach can use a variant of the Chandy-Lamport snapshot algorithm [21]. To this end, we also begin with a discretized time point, but do not halt or delay new events; instead, topology change events in the incoming event stream are tagged with a *previous* or *new* version identifier. When a vertex receives an event with a *new* version, the algorithm-related state of the vertex is split into S_{prev} and S_{new}. The events associated with the *new* version are only applied to S_{new}. If a vertex receives an event with a *previous* version then both the S_{prev} and the S_{new} state versions apply the state modifier. Subsequent algorithmic events generated as a result of the current event inherit the same version identifier. Then, akin to the simple approach, once all events associated with the *previous* version have concluded, the discretized state S_{prev} can be saved as a snapshot.

E. Observing Local State and Handling Queries

Local algorithm state, as opposed to global state, is not consistent from a global viewpoint, but is rather the viewpoint of each individual vertex. Local state informs global state in a causally consistent way, and informs any user-defined queries; when a query is defined, for example as *"When the current vertex is connected to source S"* local state can observe the instantiation of this property in constant time, and notify a user-defined event handler. These user-defined queries aim to offer real-time analysis to the programmer.

For REMO algorithms, when the local state observed is the algorithm state that guarantees convex monotonicity, one can guarantee two important properties: first, there are no false positives; for example, if S connects to T at some point in time, in the future it will not become unconnected (as a reminder, for now we operate in an edge add-only environment). Second, as a consequence of the same monotonic reasoning, the event triggering will also only occur once.

Defining a query based on the monotonic state of the algorithm is, admittedly, limiting. An example query that is not supported by the S-T connectivity algorithm would be *"Notify once all of N vertices have a path to S."* This query is not possible based on local state only, as the source does not have the knowledge of which vertices are currently connected to it, only the destination vertices know if they are connected to the source. As such, algorithm design and query design must go hand-in-hand to achieve the desired results: this query could instead be implemented on an algorithm that *accumulates* state on S, rather than *propagates* from S.

F. Prototype Implementation on Top of HavoqGT

To support our design, we have built on top of HavoqGT (Highly Asynchronous Visitor Queue Graph Toolkit) [22], an open-source graph analytics framework that targets parallel and distributed environments and large scale-free graphs. Past work [18][23][22] has demonstrated that this abstraction and its implementation provide good scalability for *static* graphs. A high-level architectural view of our prototype is presented in Figure 2.

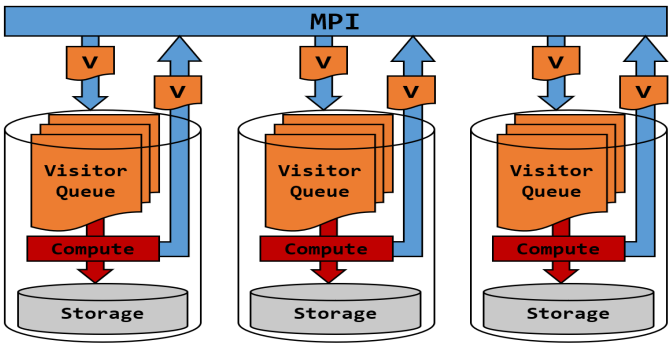

Fig. 2: *High-level structure of our prototype based on HavoqGT's visitor abstraction. Each process communicates with each other over MPI, using Visitor objects. These visitors are queued before entering the compute and storage layers of a process.*

A key enabler is HavoqGT's as a foundation is the visitor abstraction. This abstraction allows defining vertex-centric procedures that execute on vertex state, and offers a vertex the ability to create new "visitor" messages and pass them to its neighbours. HavoqGT is implemented on top of MPI and handles visitor message creation, as well as queue management for these messages. When an algorithm begins, an initial set of visitors are pushed on the queue and may generate other visitors in turn. Processing completes when all visitors have completed, which is determined by a distributed quiescence detection algorithm [24].

Our prototype extends HavoqGT to support dynamic graphs: it hides the complexities of ingesting, routing, and managing of the algorithmic events generated, and instead offers a simple programming abstraction for dynamic algorithm implementation as described in Section III-A. Algorithm 3 presents, at the high-level, how the topology evolution and algorithm-related events link into the HavoqGT infrastructure.

Prototype Limitations. Our current implementation has two major limitations: first, while the intention and design is to support multiple algorithms concurrently while analyzing an evolving graph topology, the current prototype only supports hooking in one algorithm. Second, our global state collection is a preliminary implementation of the algorithm described in Section III-D.

IV. DYNAMIC ALGORITHMS

We describe in detail four incremental REMO algorithms we have implemented based on our event-centric design (Section II) and programming model (Section III-A).

1. Breadth First Search: As previously described, BFS can be thought of as a recursive algorithm (Algorithm 1). In order to maintain a Breadth First search query upon an edge addition, we simply apply the *recursive step* starting from the new edge source (and using the edge source's present level). This recursive step becomes the update propagation event, and will flow to the component newly connected by the edge addition, terminating when the component is visited or the visited vertices all have closer or equal proximity to the source. This design, built upon the dynamic infrastructure, is shown in algorithm 4. Note the specification of the *init()* function, which needs to be called once to specify the source vertex, and can be initiated at any time.

2. Single Source Shortest Path: SSSP is similar to BFS, and unsurprisingly, uses almost identical code. The notable difference is the implication of edge weights; as SSSP evaluates

Algorithm 3: Dynamic Algorithm Infrastructure

```
class Vertex:
  int ID, value
  # Neighbour properties (e.g. 1 per edge).
  map[int nbr_ID, int nbr_value] nbrs
  static Graph* graph
  static Queue* q

  def VISIT(int
      vis_ID, int vis_val, enum VISIT_TYPE, int weight):
   switch (VISIT_TYPE):
    case 'INIT':
      init()
    case 'ADD':
      graph.insertEdge(vis_ID, this.ID, weight, this.nbrs)
      add(vis_ID, vis_val, weight)
      q.insert(vis_ID,
      this.ID, this.value, weight, 'REVERSE_ADD')
    case 'REVERSE_ADD':
      graph.insertEdge(vis_ID, this.ID, weight, this.nbrs)
      this.nbrs.set(vis_ID, vis_val)
      reverse_add(vis_ID, vis_val, weight)
    case 'UPDATE':
      this.nbrs.set(vis_ID, vis_val)
      update(vis_ID, vis_val, weight)

  def update_nbrs(int property):
    for (nbr : this.nbrs.iterator())
      q.insert(nbr, this.ID, property,
      graph.getEdgeWeight(this.ID, nbr), 'UPDATE')

  def update_single_nbr(int vertex_ID, int property):
    q.insert(vertex_ID, this.ID, property,
      graph.getEdgeWeight(this.ID, vertex_ID), 'UPDATE')

  virtual init();
  virtual add(int vis_ID, int vis_val, int weight);
  virtual
      reverse_add(int vis_ID, int vis_val, int weight);
  virtual update(int vis_ID, int vis_val, int weight);
```

Algorithm 4: Breadth First Search

```
class BFSVertex: public Vertex:
  def init(): # Begin traversal from this vertex.
    this.value = 1;
    update_nbrs(this.value);

  def add(int vis_ID, int vis_val, int weight):
    # If we are a new vertex, ensure level is inf.
    if (this.value == 0):
      this.value = MAX_INTEGER

  def reverse_add(int vis_ID, int vis_val, int weight):
    # If we are a new vertex, ensure level is inf.
    if (this.value == 0):
      this.value = MAX_INTEGER
    # The rest of the logic is the same as update step.
    update(vis_ID, vis_val)

  def update(int vis_ID, int vis_val, int weight):
    # Check if we have a lower level. (hop offset)
    if (this.value < vis_val - 1):
      # Notify back the visitor.
      update_single_nbr(vis_ID, this.value)

    # Check if they have a lower level. (hop offset)
    elif (this.value > vis_val + 1):
      this.value = vis_val + 1
      # Need to send our new level to all neighbours.
      update_nbrs(this.value)
```

the path distance from source to all vertices, the resulting "cost" of a vertex relative to the source is the minimum sum of edge weights required to reach the source. SSSP is implemented in the same way as BFS – with a recursive update step. When an edge is added to the graph, the source vertex already has a minimum cost to the destination vertex, and thus, the update effectively flows downward to update the newly connected vertices to, if possible, reduce their cost to the source. (Algorithm 5). The stark similarity in code to BFS is evident, however the actual execution path of an instantiated algorithm is more data dependant, as the

Algorithm 5: Single Source Shortest Path

```
class SSSPVertex: public Vertex:
  def init():  # Begin traversal from this vertex.
    this.value = 1
    update_nbrs(this.value)

  def add(int vis_ID, int vis_val, int weight):
    # If we are a new vertex, ensure cost is inf.
    if (this.value == 0):
      this.value = MAX_INTEGER

  def reverse_add(int vis_ID, int vis_val, int weight):
    # If we are a new vertex, ensure cost is inf.
    if (this.value == 0):
      this.value = MAX_INTEGER;
    # The rest of the logic is the same as update step.
    update(vis_ID, vis_val, weight)

  def update(int vis_ID, int vis_val, int weight):
    # Check if we have a lower cost.
    if (this.value < vis_val - weight):
      # Notify back the visitor.
      update_single_nbr(vis_ID, this.value);

    # Check if they have a lower cost.
    elif (this.value > vis_val + weight):
      this.value = vis_val + weight
      # Need to send our new cost to all neighbours.
      update_nbrs(this.value)
```

Algorithm 6: Connected Components

```
class CCVertex: public Vertex {
  def add(int vis_ID, int vis_val, int weight):
    # If we are a new vertex, label us.
    if (this.value == 0):
      this.value = hash(this.ID)

  def reverse_add(int vis_ID, int vis_val, int weight):
    # If we are unlabeled (new), label us.
    # (We know other vertex
      dominates us, since same hash for order of add.)
    if (this.value == 0):
      this.value = vis_val
    else:
      # Otherwise, logic is same as update step.
      update(vis_ID, vis_val, weight)

  def update(int vis_ID, int vis_val, int weight):
    # Check if our component is the dominator.
    if (this.value > vis_val):
      # Notify back the visitor.
      update_single_nbr(vis_ID, this.value)

    # Their component is the dominator.
    elif (this.value < vis_val):
      this.value = vis_val
      # Need to send our new label to all neighbours.
      update_nbrs(this.value)
```

edge weights play a key role in the topology. This can cause an entirely different data traversal pattern compared to BFS.

3. Connected Components: The CC algorithm does not require an initiating vertex. However, the algorithm still maintains the REMO properties like BFS and SSSP, by retaining a similar recursive update step. For CC, this involves a form of label propagation – each vertex primarily assumes it will "dominate" the component it is attached to, and contacts all neighbours, which will either accept and attempt to propagate the label further, or reject it and reply back with their own label (Algorithm 6). Notably, this algorithm requires more logic upon the addition of an edge (instead of through an *init()* function), where the algorithm applies a label to any new vertex added to the graph.

4. Multi S-T Connectivity: From a given source vertex S, a flow outwards is established, and any vertex T can identify if they are connected to the source. For our implementation of *multi* S-T connectivity, we can vary the set of sources S, and compute their connectivity for all participating vertices (i.e. T is the set of all V).

Algorithm 7: Multi S-T Connectivity

```
class STVertex: public Vertex:
  def init():  # Begin a source from this vertex.
    this.value = this.value ∪ this.ID
    update_nbrs(this.value)

  # Do nothing but wait.
  def add(int vis_ID, int vis_val, int weight):;

  def reverse_add(int vis_ID, int vis_val, int weight):
    # The logic is the same as update step.
    update(vis_ID, vis_val, weight)

  def update(int vis_ID, int vis_val, int weight):
    if (this.value == vis_val):
      pass  # do nothing
    # Check if our set is a pure SUPERset of theirs.
    elif ((this.value ∪ vis_val) == this.value):
      # Notify back the visitor.
      update_single_nbr(vis_ID, this.value)

    # Check if our set is a pure SUBset of theirs.
    elif ((this.value ∪ vis_val) == vis_val):
      # Apply their set, send to all neighbours.
      this.value = this.value ∪ vis_val
      update_nbrs(this.value)

    # There is a mix. Apply, broadcast to all.
    else:
      this.value = this.value ∪ vis_val
      update_nbrs(this.value)
```

The implementation of Multi S-T Connectivity is shown in algorithm 7. When a vertex becomes connected to another, each shares the subsets of S that they have connectivity to, and each compare. If a vertex is a superset (i.e. it already has connectivity to all sources that the other had) it does nothing. If a vertex is a subset (opposed to the other being the superset) it broadcasts its new connectivity. Finally, if there is a mix of connectivity, it also broadcasts, eventually causing an exchange of sets between the two.

Algorithm Design Insights. The above algorithms are supported well in an incrementally dynamic system, as each can be represented as a REMO strategy. Each employs some recursion: a base case (an edge change), followed by a propagation step (recursive update events). The second factor, monotonicity, is unique for each – each algorithm has a distinct, correct answer for a static graph (or a dynamic graph at some time T), and a different way to derive that answer. However, in all cases this answer is both convex and can be reached refined monotonically.

These REMO commonalities enable each algorithm to support concurrent, asynchronous updates, as each algorithm effectively "converges" to the correct state (i.e. the global minimum in the convex solution space); there is no need to keep track of ordering of events that impact different parts of the graph (only relative ordering is needed), and events that impact the same vertex are ordered in the infrastructure layer by the built-in visitor queue in FIFO ordering.

V. EVALUATION

To fully evaluate the system, it is important to: *(i)* define a credible baseline, as well as *(ii)* demonstrate the ability to scale to realistically-sized workloads. To this end, this section presents: *(i)* the comparison and analysis of the dynamic infrastructure compared to a static equivalent (executing based on the same framework), and *(ii)* scaling results when increasing the compute resources and size of the resulting data.

A. Experimental Platform, Methodology, Workload

Experimental platform. The experimental platform is the Catalyst cluster at Lawrence Livermore National Laboratory. Catalyst is an experimental data-intensive platform that has for each node dual 12-core Intel Xeon E5-2695v2 (2.4 GHz) processors, 128 GB of memory, and Intel 910 PCI-attached NAND Flash. In plots, one node means parallel MPI ranks over all 24 cores.

TABLE I: Graphs used in experiments. RMAT graphs (Graph500 parameters) have a 16x undirected (32x directed) edge factor. Graphs are made undirected with reverse edges where needed.

Name	#Vertices	#Edges	OnDiskSpace
Friendster [25]	65,608,366	3,612,134,270	61 GB
Twitter [20]	41,652,230	2,936,729,768	49 GB
SK2005 [26]	50,636,059	3,860,585,896	65 GB
Webgraph [27]	3,563,602,686	257,473,828,334	5.1 TB
RMAT($SCALE$)	$2^{(SCALE)}$	$2^{(SCALE)} * 32$	
e.g. RMAT31	2,147,483,648	68,719,476,736	

Experimental methodology. Plots of runtime present averages over 10 runs. For algorithms with an initiation vertex, a vertex is randomly pre-chosen so that is known to eventually lie within the largest connected component, and the same vertex is chosen if comparing across methods (e.g. static vs. dynamic). For dynamic executions, edges are pre-randomized and ingested by reading *[source, destination]* pairs from disk: these edges are read as fast as possible (parallelized into one stream per MPI rank), in order to understand the limitations of the dynamic construction system by feeding it as much as it can handle – each rank "pulling" a topology event as soon as local work is completed. This is thus a *saturation* test, representing the maximum possible event processing throughput. Any offered load lower than the reported maximum performance can be handled in real-time.

Workloads. For evaluation we use large real world and synthetic graphs, including the largest publicly-available real-world graphs. These are presented in detail in Table I. Note that dynamic ingestion of static graphs is done randomly, as these datasets provide only topology information.

B. Baseline Performance

Figure 3 showcases the result of several key runtime comparisons on a single node. There are several comparisons in this figure: first, the comparison of *static* construction (including compression from input presented as *[src, dst]* pairs to Compressed Sparse Row (CSR) format, and partitioning the components of the CSR structure: one per MPI process) to dynamic construction (bottom parts on the left and center bars); second, the comparison between running the static algorithm run-time on top of the optimized static graph construction and the graph constructed dynamically (top parts on the left and center bars); and, third, comparing both strategies mentioned so far (left and center, full bars) with building the graph and updating the BFS solution while doing it (right bar). We highlight the following comparisons.

Static vs. Dynamic – Graph Construction: One surprising result is that constructing the static graph for use with algorithms is only approximately 2x faster than the construction of a fully dynamic graph. As expected [28], the time to construct dwarfs algorithm time for BFS; however, for static graphs, a relatively large time to build the graph is often considered acceptable, as the construction need only occur once and the resulting graph can be used for multiple algorithms. This argument can also be applied to the construction time of the dynamic data-structure: as shown, one can use the constructed dynamic data-structure and execute *any known static algorithm* on top of it. However, the dynamic data-structure offers two clear advantages: *(i)* any change to

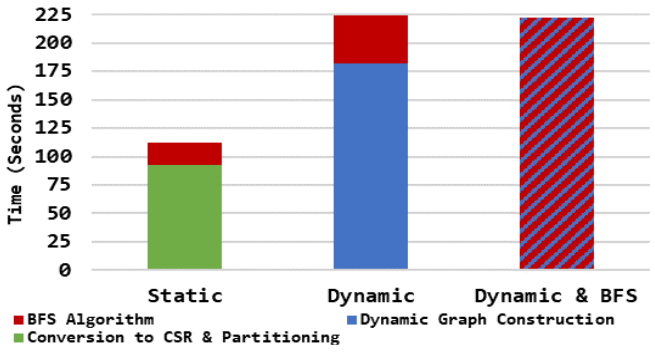

Fig. 3: *A comparison of static vs. dynamic strategies (1 node i.e. 24 cores / MPI ranks, Twitter dataset). Y-axis represents time to completion (seconds), and the X-axis the strategy used. The first bar presents the static processing time: the time to fully load the graph in memory (and perform the available optimizations, e.g. using the CSR format), stacked with one BFS execution. The second bar presents a scenario where the graph is loaded as a dynamic graph (ingesting edges) and one static BFS execution on the resulting final graph after all edges have been ingested. The third bar presents a dynamic case that overlaps graph construction with the BFS algorithm, keeping a live, real-time queryable result during graph evolution.*

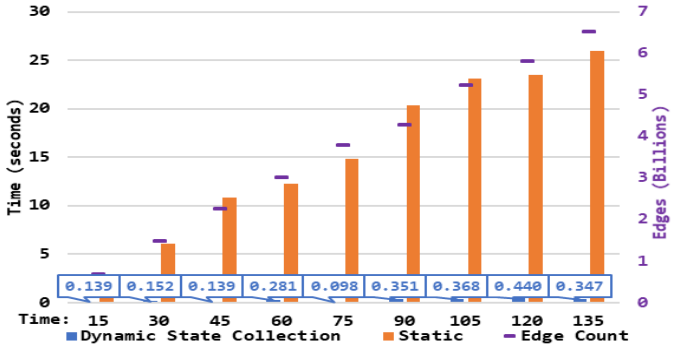

Fig. 4: *An evaluation of global algorithm state collection for BFS (16 nodes, RMAT edge generation). The X-axis represents different intervals (in seconds from beginning ingestion), and the Y-axis (left) the time for the solution to complete at that interval. The left bars (the precise numbers presented in callouts) present the dynamic state collection solution: the latency from asking for the global algorithm state, to the concluded state collected across all nodes. The right bars present how long it would take, given the same graph topology and execution environment, to run the algorithm statically (with no further edge ingestion). The size of the graph (in edges) at the interval is presented on the Y-axis (right).*

the graph can be applied incrementally as a set of events, at low cost, then any static algorithm can be applied to the new graph, and *(ii)* a live algorithm can offer an observable solution based on live events, where a query for local or global state can be initiated at any time before, during, or after construction.

Static vs. Dynamic – Algorithms: In Figure 3, we also present two different algorithms: static BFS executed on either the static or the dynamically built graph data-structure[3], (top bars, left and center) as well as a dynamic BFS query representing an observable query across the entire life span of graph construction (right bar). There is a clear overhead in executing a static algorithm on top of the dynamic structure, and the primary reason is that the static construction has an advantage of compression. As static graphs know a priori the degree of a vertex, and know it will not change, we can use the CSR format – decreasing memory requirements, and thus improving locality for neighbour iteration. A second advantage is state storage: the static BFS writes to a pre-defined buffer of size $V \in P$, the vertices within the given partition. This further increases locality for data writes for the static case. However, for the static BFS on the dynamic data-structure, each write is to a dynamic location within the DegAwareRHH structure (represented as a vertex property)[18], as the size of V is dynamic. Future work could target improving this performance for the static-on-dynamic use case, for example by intelligently gathering a local vertex count, allocating a static query buffer for the execution, and storing data there temporarily instead.

The right bar in Figure 3 presents the dynamic case that overlaps graph construction and maintains a live, real-time queryable result during graph evolution for the BFS algorithm. The key takeaway is that while there is no observable overhead compared to dynamically building the entire graph and then executing the static BFS (center bar), this strategy offers the key advantage that the algorithm state is observable during construction. This enables, for example, *(i)* simply triggering a callback immediately after a node is connected to the source (or has a path shorter than a specified length to the BFS source), or *(ii)* collecting BFS algorithm state, i.e. the BFS tree, at any point during construction, at a low cost, as we demonstrate below.

[3]Unlike the static graph, the dynamic graph data-structure is built one edge at the time, without any information about future edges in the stream (what they are or how many there will be). Thus the possible data layout and partitioning optimizations are severely limited.

C. Global State Collection

In section III-D, we explained how global algorithm state can be collected on-demand. To showcase the of this, in Figure 4, we perform an experiment ingesting random RMAT edges, and every 15 seconds, do two things: first, we measure the latency from requesting an on-the-fly collection at the interval's occurrence, until the dynamic global state has converged (i.e. the dynamically maintained algorithm data for BFS at that discrete time point is finalized). Second, we run a traditional static BFS algorithm on the given topology (as if it were a pre-loaded snapshot), in order to give a reference of how long it would take to compute the algorithm state from scratch. Note that in this comparison, the topology is in memory to begin with – a traditional snapshotting solution would first have to load topology, adding overhead.

As Figure 4 shows, the latency from the user-defined time-point until the global algorithm state is collected is in the order of hundreds of milliseconds, which is in stark contrast to the high overhead of computing a static algorithm on the same topology from scratch. We note that part of this advantage is obtained with high probability, but not guaranteed: an adversarial graph and query could be constructed to achieve a worst case scenario. For example, the event ingested right before initiating the collection of global state could trigger large cascading BFS events; in this case, the static algorithm presents an upper bound. A further limitation is prioritization between topology and algorithmic events: in the best case there is no overhead, but for shared resources there is a small tradeoff between latency and maximum event ingestion rate. For this

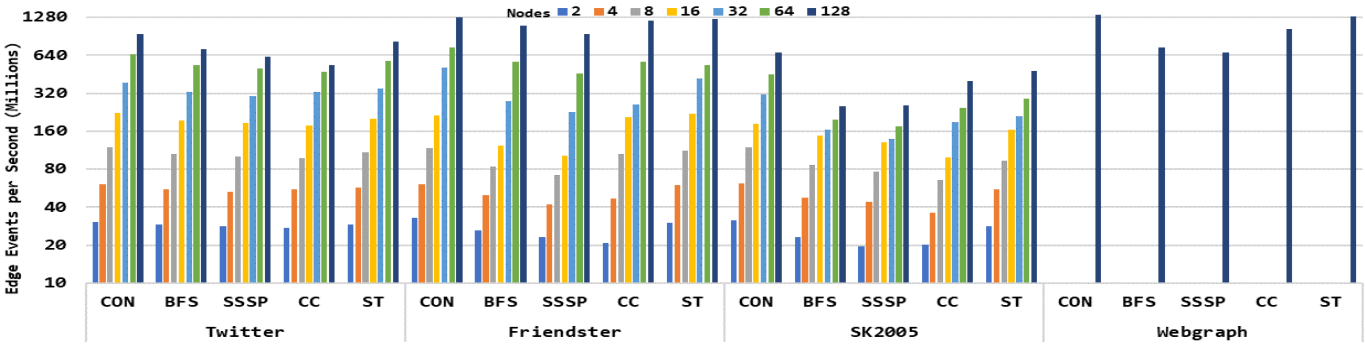

Fig. 5: *Performance results for queries on real-world graphs. The Y-axis denotes events per second (log scale), while the X-axis denotes the graph and algorithm applied (CON being construction only). Each bar represents the performance for scaling node count. Note: due to its size, the Webgraph only fits in memory on 128 nodes.*

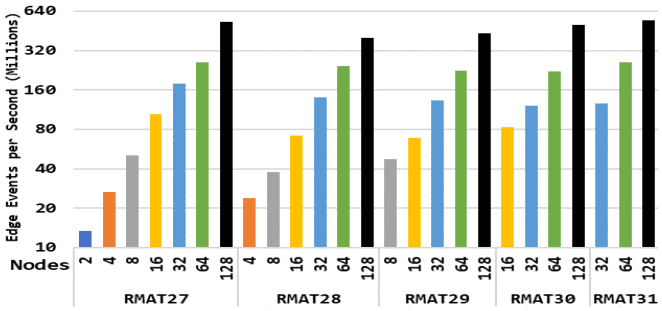

Fig. 6: *Synthetic scaling results, scaling up RMAT graph size and node count. The BFS algorithm state is maintained during the dynamic construction. The Y-axis (log scale) denotes events per second, while the X-axis (log scale) denotes the compute node count. The key take-away is that, given similar graph structure, the size of the graph does not impact event processing rate (good weak scaling).*

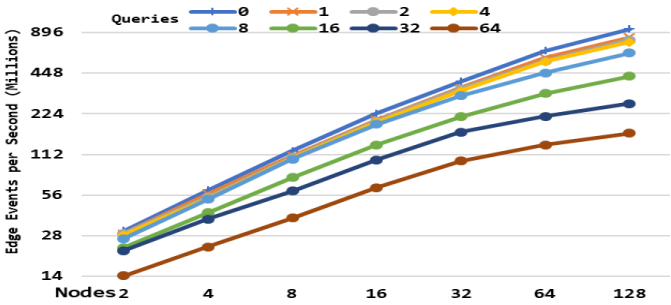

Fig. 7: *Scaling of multiple sources in S-T Connectivity, on the Twitter dataset. The Y-axis denotes events per second (log scale), while the X-axis denotes the compute node count. Each line represents a number of independent connectivity sources: from 0 (construction only) to 64.*

evaluation, we highlight absolute costs from ingestion saturation, but future work could seek to explore this tradeoff.

D. Dynamic Algorithm Query on Real Graphs

Figure 5 showcases each algorithm across the real-world datasets. The experiments highlight two key points: first, the cost of maintaining an algorithm with observable results during the construction had a low impact on performance compared to the construction-only execution (labeled CON). The reason for this is, by overlapping the graph construction with the updating algorithm, the cost of messaging a neighbour to update their state is often amortized by latching to the construction of the edge to that neighbour. Second, the resulting structure and topology of the dynamic graph created a slightly different performance pattern for each dataset, for both construction alone as well as with a dynamic algorithm.

E. Strong and Weak Scaling for Incremental BFS

Figure 6 shows the result of scaling on the synthetic RMAT datasets, while maintaining the BFS algorithm. The result shows good scalability: with an almost linear speedup with compute node count for the same graph, and, more importantly, the size of the graph did not significantly impact maximum event rate. This means scalability on two fronts: first, for a single event stream, doubling the node count will almost double the maximum ingestion rate. Second, as a graph grows in size over time, the maximum event rate is not significantly impacted due to the increased scale. In other words, the event ingestion rate is more closely tied with the structure of the graph topology (noting the differences in the real world graphs in figure 5), rather than the growth of the graph.

These properties are extremely desirable for this type of framework, as improving performance in a dynamic environment is paramount not only for the "wait for an answer" time, but also the source event rate which the system can handle. For these results, we show the system can scale to handle event rates of at least 400 million, and at best 1.3 billion edge events per second for a single, queryable algorithm on 128 nodes (3,072 cores). This high event processing rate suggests significant room to add complexity to algorithms, while remaining within processing requirements of real-time event sources.

F. Multi S-T Connectivity

The S-T Connectivity algorithm was designed to support multiple sources, and allows each vertex to independently know its connectivity status to any source S_i. Each source has an independent "flow" associated with it, and thus it makes a good baseline to gauge increasing the number of available queries: the connectivity queries enabled by this algorithm are dependant on the number

of sources chosen, as a query can execute user-defined behaviour when connectivity occurs from any vertex to any $S_i \in S$.

In figure 7 we show the impact of increasing the number of connectivity sources maintained dynamically, with a baseline of zero (construction only), on the Twitter dataset. The figure shows that, similar to the scaling results, doubling the node count gives a near doubling of maximum event rate, although the performance does taper off when the available work per node decreases too much. When doubling the number of sources however, the performance suffers non-linearly: the first few added sources do not greatly impact performance (for example, from one source to two induced less than a 10% cost), but the performance nearly halves after doubling the set of sources. The likely explanation for the difference in performance rate is due to seeing bottlenecks in messaging or buffer sizes. Regardless, the system is able to scale to over 160 million maximum events per second, for 64 concurrent sources.

VI. Discussion

While this paper demonstrates the feasibility and the benefits of *(i)* a graph processing system that maintains live algorithm state updated dynamically as a result of topology changes, and *(ii)* a class of algorithms that can be implemented in this style for incremental graphs, there are two important questions that should be discussed. The first is *"Why is this better than a batching solution?"* and the second is, *"How would one support delete events?"*

A. Comparison to Snapshot/Batching Solutions

Although difficult to quantitatively evaluate beyond the arguments brought forward in the previous section, it is important to stress the advantages of the approach we propose. In our system, if a user queries for global state at a discrete time point, the result returned is functionally equivalent to a snapshot (or processing of a batch) that ended at that specific time point. In figure 4, we chose specific intervals for global state collection – however, the design we provide supersedes a snapshotting solution, as it is continuous: *(i)* at *any* time desired a global state snapshot can be collected on-the-fly and in near real-time, and *(ii)* it further offers the ability to query and generate trigger events based on local vertex state over time. The vertex information offered via local state can be critical for real-time sensitive systems, and while the latency for snapshot systems offering a response is the entire time between snapshots, the continuous solution we present both captures causality and offers consistent, minimal latency.

In the experimental evaluation, we mostly focused on the event processing rate. It is important to make clear, however, that during each dynamic experiment the following system properties always held true:

- Any vertices' local state can be observed in constant time.
- Any user-defined callback could have been triggered whenever any vertices' local state matches a user-defined value.
- Any change(s) to the topology of the graph can be applied immediately, at any time.
- Any known static graph algorithm (not restricted to the set of algorithms presented here) could be applied on the dynamic graph whose evolution is paused or concluded.
- Global algorithm state can be collected at any discrete time before, during, or after modifications, without pausing or significantly affecting the on-going computations (as it can be done asynchronously in parallel).

B. Supporting Decremental Events

Although delete events are typically rarer in dynamic graphs (with many having none), an algorithm designer may wish to handle their occurrence asynchronously and concurrently with add events (note that it is trivial, yet costly, to handle delete events synchronously, by using a stop-the-world strategy). Here we outline a strategy that would provide this functionality.

Our solution for the ReMo algorithms is based on two of their properties: *(i)* each topology event (i.e. edge-add) can only monotonically move an identified algorithmic property towards a lower state, and *(ii)* upon recursion termination, the solution will converge to the minimum state possible, which is also the solution of the problem. These properties do not hold true anymore once we introduce delete events (e.g. in the case of BFS, while edge-add events will always reduce or maintain constant distance between the source and any graph node – thus maintaining monotonicity – edge deletes may increase this distance).

To address this issue we introduce state *generations*. We define the new monotonic state to be determined: *(i)* firstly by the *generation* of the algorithmic state, and only secondly by *(ii)* the actual algorithmic state. By having the generation as the primary factor, we can avoid situations that would otherwise break monotonicity: if an algorithmic action would break monitonicity (e.g., an edge delete leads to increasing distance to BFS source) we move the state into a new generation. Effectively, the action creates a new total state that is within a convex space lower than all possible other states within the current generation, and thus despite increasing its algorithm state, it is overall in a more minimal state.

While deletion events done in this generational fashion may have a high overhead, generally, the ratio of delete to add events is low, and many delete events can be treated as special cases (e.g. deletion of singletons, or edges connecting a leaf node). Further, a rewriting of data at this magnitude may also happen in an incremental only solution – but, similarly, only in the worst case. While we initially discuss here an operation that may often cause cascades, this provides a correct solution as a starting point and future optimizations will reduce complexity and reduce the occurrences of worst-case scenarios (e.g. such as maintaining spanning trees [15] and only reacting on tree cutting).

VII. Related work

Dynamic graphs have been well explored in a sequential context, with previous work being done to bound per-update timings for algorithms such as connectivity and spanning trees [14] [15]. This past work assumes global access to state (i.e. through shared memory or synchronization), and each edge change is processed atomically and synchronously. Rather than serializing events, our work targets real-time performance with concurrent asynchronous updates to topology – and thus requires an entirely different design space for algorithms that must be resilient to complex asynchronous interactions.

Comparison to other works, especially regarding performance, is difficult, with much of the previous work on "dynamic graphs" actually being solutions that use snapshotting rather than true on-line processing [6][7][13]. The closest related work is that of STINGER, a shared memory solution which can ingest structural changes at a rate of 10 million events per second with an updating kernel peak rate of around 1 million events per second [8][16]. Notably, in our implementation the algorithms run concurrently

with graph structure changes, so our events per second metrics presented are more closely comparable to the updating kernel rate.

There are multiple fundamental differences between STINGER and our work: *(i)* we do not batch or serialize events (each of our events are treated asynchronously, with the relative ordering defined: events in the same stream are ordered, events in different streams are not), *(ii)* we do not use shared memory (nor locking or atomics), *(iii)* our solution is real-time and on-line, thus none of our algorithms or designs require stopping the world (i.e. the event stream) for any reason, and *(iv)* the initialization or instantiation of our algorithms is done implicitly and at any time (as opposed to requiring update kernels), regardless of the current state of the world.

An algorithmic model for streaming graphs was also recently presented [29], which explores a similar design philosophy to our work, showing a set of algorithms that can be updated in a concurrent manner and maintain validity. Like our design, they target algorithms running concurrently with structural changes. However, the key difference in our philosophy is that of avoiding batching of update events and the avoidance of shared memory. With our REMO design, we show that if an algorithm can be made to always monotonically move towards a deterministic solution, the event concurrency can always be satisfied. Unfortunately, this work did not present an implementation to compare against.

VIII. CONCLUSION

As this work shows, offering better support for on-line algorithms on dynamically evolving graphs is a path that continues to be fruitful. First, this work demonstrates that many common graph algorithms belong to a class of algorithms that have two key properties, *recursive updates* and *monotonic convergence*, which enables their ability to support incremental processing and live queries. Second, this work presents an event-based framework to support this class of algorithms, demonstrating a good performance range compared to static solutions, and exceeds the existing event rates in existing systems by several orders of magnitude. Third, a dynamic graph data-structure is feasible to build and scale, while offering key advantages against static ones: namely, any change to the graph can be applied at low cost, then any static algorithm can be applied to the new graph, and any live algorithm can be observed at any time before, during, or after dynamic construction or modifications. Finally, the evaluation in this paper shows promising scalability results: the event-based framework scales nearly linearly, and shows support for real-time analysis on up to 1.3 billion edge events per second on 128 compute nodes – suggesting that a large amount of room for additional algorithmic complexity is available.

ACKNOWLEDGEMENT

This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DEAC52-07NA27344 and was supported by the LLNL-LDRD Program under Project No. 17-ERD-024. (LLNL-CONF-768197).

REFERENCES

[1] A. Yoo, A. Baker, R. Pearce, and V. Henson, "A scalable eigensolver for large scale-free graphs using 2D graph partitioning," in *Supercomputing*, pp. 1–11, 2011.

[2] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Supercomputing*, 2011.

[3] F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A. R. Choudhury, and Y. Sabharwal, "Breaking the speed and scalability barriers for graph exploration on distributed-memory machines," in *Supercomputing*, 2012.

[4] "Graph500, http://www.graph500.org/."

[5] "Graphchallenge, https://graphchallenge.mit.edu/challenges."

[6] A. P. Iyer, L. E. Li, T. Das, and I. Stoica, "Time-evolving graph processing at scale," in *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, p. 5, ACM, 2016.

[7] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: taking the pulse of a fast-changing and connected world," in *Proceedings of the 7th ACM european conference on Computer Systems*, pp. 85–98, ACM, 2012.

[8] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "Stinger: High performance data structure for streaming graphs," in *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pp. 1–5, IEEE, 2012.

[9] "Twitter engineering." https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how.html. Retrieved Oct 2018.

[10] "Visa." https://usa.visa.com/run-your-business/small-business-tools/retail.html. Retrieved Oct 2018.

[11] "Zephoria." https://zephoria.com/top-15-valuable-facebook-statistics/. Retrieved Oct 2018.

[12] "Eos block explorer." https://www.bloks.io/. Retrieved Oct 2018.

[13] C. Wickramaarachchi, A. Kumbhare, M. Frincu, C. Chelmis, and V. K. Prasanna, "Real-time analytics for fast evolving social graphs," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 829–834, May 2015.

[14] M. R. Henzinger, V. King, and V. King, "Randomized fully dynamic graph algorithms with polylogarithmic time per operation," *Journal of the ACM (JACM)*, vol. 46, no. 4, pp. 502–516, 1999.

[15] M. R. Henzinger and V. King, "Maintaining minimum spanning trees in dynamic graphs," in *International Colloquium on Automata, Languages, and Programming*, pp. 594–604, Springer, 1997.

[16] J. Riedy and D. A. Bader, "Multithreaded community monitoring for massive streaming graph data," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pp. 1646–1655, IEEE, 2013.

[17] S. Sallinen, K. Iwabuchi, S. Poudel, M. Gokhale, M. Ripeanu, and R. Pearce, "Graph colouring as a challenge problem for dynamic graph processing on distributed systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 30, IEEE Press, 2016.

[18] K. Iwabuchi, S. Sallinen, R. Pearce, B. V. Essen, M. Gokhale, and S. Matsuoka, "Towards a distributed large-scale dynamic graph data store," in *Graph Algorithms Building Blocks (GABB2016)*, 2016.

[19] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pp. 654–663, ACM, 1997.

[20] M. Cha, H. Haddadi, F. Benevenuto, and K. Gummadi, "Measuring user influence in twitter: The million follower fallacy," in *4th International AAAI Conference on Weblogs and Social Media (ICWSM)*, 2010.

[21] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 1, pp. 63–75, 1985.

[22] R. Pearce, M. Gokhale, and N. M. Amato, "Faster parallel traversal of scale free graphs at extreme scale with vertex delegates," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 549–559, IEEE Press, 2014.

[23] S. Poudel, R. Pearce, and M. Gokhale, "Towards scalable graph analytics on time dependent graphs," in *Third SDM Workshop on Mining Networks and Graphs (MNG)*, 2016.

[24] R. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, IEEE Computer Society, 2010.

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

[26] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: A scalable fully distributed web crawler," *Software: Practice and Experience*, vol. 34, no. 8, pp. 711–726, 2004.

[27] http://webdatacommons.org/hyperlinkgraph/.

[28] J. Malicevic, B. Lepers, and W. Zwaenepoel, "Everything you always wanted to know about multicore graph processing but were afraid to ask," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, (Santa Clara, CA), pp. 631–643, USENIX Association, 2017.

[29] C. Yin, J. Riedy, and D. A. Bader, "A new algorithmic model for graph analysis of streaming data," in *Proceedings of the 14th International Workshop on Mining and Learning with Graphs (MLG)*, 2018.