Non-Blocking Dynamic Unbounded Graphs with Worst-Case Amortized Bounds

Bapi Chatterjee  
Indraprastha Institute of Information Technology Delhi, India

Sathya Peri  
Indian Institute of Technology Hyderabad, India

Muktikanta Sa  
Télécom SudParis – Institut Polytechnique de Paris, France

Komma Manogna  
Indian Institute of Technology Hyderabad, India

Abstract

Today’s graph-based analytics tasks in domains such as blockchains, social networks, biological networks, and several others demand real-time data updates at high speed. The real-time updates are efficiently ingested if the data structure naturally supports dynamic addition and removal of both edges and vertices. These dynamic updates are best facilitated by concurrency in the underlying data structure. Unfortunately, the existing dynamic graph frameworks broadly refurbish the static processing models using approaches such as versioning and incremental computation. Consequently, they carry their original design traits such as high memory footprint and batch processing that do not honor the real-time changes. At the same time, multi-core processors—a natural setting for concurrent data structures—are now commonplace, and the analytics tasks are moving closer to data sources over lightweight devices. Thus, exploring a fresh design approach for real-time graph analytics is significant.

This paper reports a novel concurrent graph data structure that provides breadth-first search, single-source shortest-path, and betweenness centrality with concurrent dynamic updates of both edges and vertices. We evaluate the proposed data structure theoretically—by an amortized analysis—and experimentally via a C++ implementation. The experimental results show that (a) our algorithm outperforms the current state-of-the-art by a throughput speed-up of up to three orders of magnitude in several cases, and (b) it offers up to $80x$ lighter memory-footprint compared to existing methods. The experiments include several counterparts: Stinger, Ligra and GraphOne. We prove that the presented concurrent algorithms are non-blocking and linearizable.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms

Keywords and phrases concurrent data structure, linearizability, non-blocking, directed graph, breadth-first search, single-source shortest-path, betweenness centrality

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2021.20


Supplementary Material Software (Source Code): https://github.com/sngraha/PANIGRAHAM

Funding This work was partially funded by National Supercomputing Mission, Govt. of India under the project “Concurrent and Distributed Programming primitives and algorithms for Temporal Graphs” (DST/NSM/R&D_Exascale/2021/16).

Introduction

A graph represents the pairwise relationships between objects or entities that underlie the complex frameworks such as blockchains, social networks, semantic-web, biological networks and many others. The contemporary applications of graph algorithms in real-time analytics,
such as product recommendation or influential user tracking [31] over social network graphs, demand dynamic addition and removal of vertices and/or edges over time. Existing approaches for graph analytics can be broadly classified in batch analytics, e.g. GraphTinker [29], where a graph operation is performed over a static temporal snapshot of the data structure, and stream analytics e.g. Kineograph [14], where a temporal window of incoming data is studied. In general, these approaches inherently assume that the dynamic updates are monotonic; the structure of the graph largely remaining unaffected. A deviation from the assumed data ingestion pattern severely affects their design optimizations. Notwithstanding, in such techniques concurrency is predominantly limited in a “true” real-time sense. Furthermore, in anticipation of growing number of edges, they allocate a large chunk of memory. Recent trends show that there is an emerging niche for the analytics tasks closer to the data sources, such as mobile and edge devices [8]. On such platforms, though multi-core is getting progressively ubiquitous [40], unlike data-center-based settings, memory is limited and therefore the graph applications with unlimited dynamic updates must aim to have a lightweight memory footprint. In substance, the pursuit of an efficient lightweight real-time concurrent graph analytics framework with a fresh design approach is imperative.

Concurrent Data Structures

With the rise of multi-core computers, concurrent data structures have become popular, for they are able to harness the power of multiple cores effectively. Several concurrent data structures have been developed in recent years such as: stacks [23], queues [4, 24, 32, 35], linked-lists [13, 21, 22, 48, 49, 50], hash tables [37, 38], binary search trees [6, 10, 13, 19, 39, 43], etc. On concurrent graphs, Kallimanis et al. [30] presented dynamic traversals and Chatterjee et al. [12] presented reachability queries. However, graph analytics queries, for example, single-source-shortest-path (SSSP) queries, which appertain to link-prediction in social networks or betweenness centrality, which finds applications in stock markets [44], are much more complex than reachability. The aforementioned queries inherently scan through (almost) the entire graph. In a dynamic setting, a concurrent update of a vertex or an edge can potentially render the output of such queries inconsistent.

To elucidate, consider computing the shortest path between two vertices. It requires exploring all possible paths between them, followed by returning the set of edges that make the shortest path. It is easy to see that an addition of an edge to another path can make it shorter than the one returned, and similarly, a removal of an edge (from it) could make it no longer the shortest. Imagine the addition and removal to be concurrent with the query, which can certainly benefit the application. Clearly, the return of the query can be inconsistent with the latest state of the graph.

In a concurrent dynamic graph, we require the updates and queries be consistent. To motivate, consider the computation of the risk-adjusted performance of a stock-portfolio via betweenness centrality [44]. In a dynamic setting, where the results of such analytics tasks influence the high-stake financial decisions, it is significant that a user is supplied with a consistent query result.

A commonly accepted correctness-criterion for concurrent data structures is linearizability [27], which intuitively infers that the output of a concurrent execution of a set of operations should appear as executed in a certain sequential order. Separating a graph query from concurrent updates by way of locking the shared vertices and edges can achieve linearizability. However, locking the portion of the graph that requires access by a query, which often could very well be its entirety, would obstruct a large number of concurrent fast updates. Even an effortful interleaving of the query- and update-locks at a finer granularity
does not protect against pitfalls such as deadlock, convoying, etc [25]. A more attractive option is to implement non-blocking (lock-free) progress, which ensures that some non-faulty (non-crashing) threads complete their operations in a finite number of steps [26]. Surprisingly, non-blocking linearizable design of queries that synchronize with concurrent updates in a dynamic graph are difficult.

**Proposed work.** In this paper, we describe the design and implementation of a graph data structure, which provides (a) three useful operations – breadth-first search (BFS), single-source shortest-path (SSSP), and betweenness centrality (BC), (b) dynamic updates of edges and vertices concurrent with the operations, (c) non-blocking progress with linearizability, and (d) a light memory footprint. We call it PANIGRAHAM: Practical Non-blocking Graph Algorithms.

**Algorithm Overview**

In a nutshell, we implement a concurrent non-blocking dynamic directed graph data structure as an adjacency-list formed by a composition of lock-free sets: a lock-free hash-table and multiple lock-free binary search trees (BSTs). The set of outgoing edges $E_v$ from a vertex $v \in V$ is implemented by a BST, whereas, $v$ itself is a node of the hash-table (as shown in Figure 1). Addition/removal of a vertex amounts to the same operation of a node in the lock-free hash-table, whereas, addition/removal of an edge translates likewise to a lock-free BST. Although lock-free progress is composable [16], thereby ensuring lock-free updates in the graph, however, optimizing these operations is nontrivial as shown by us in this paper. The operations – BFS, SSSP, BC – are implemented by specialized partial snapshots of the composite data structure. In a dynamic concurrent non-blocking setting, we apply multi-scan/validate [1] to ensure the linearizability of a partial snapshot. We prove that these operations are non-blocking. The empirical results show the effectiveness of our algorithms.

**Related work**

Libraries of parallel implementation of graph operations are abundant in literature. A relevant survey can be found in [5]. To mention a few well-known ones: PowerGraph [20], Galois [33], Ligra [45], Ligra+ [46], MGraph [51], Congra [42], Congra+ [41]. However, they primarily focus on static queries and natively do not allow updates to the data structure, let alone concurrency.

Broadly, these libraries use the compressed sparse row (CSR) format, a read-only representation, to implement a graph. In principle, the basic designs of an adjacency list and the CSR are almost identical [47], however, in practice the CSR exhibits better cache efficiency due to locality [7]. In a dynamic setting, a serious drawback of the CSR format is the need for reprocessing the entire structure for vertex updates and the array that stores edge information for edge updates.

To our knowledge, Stinger [18] was the first large-scale practical implementation that supported dynamic updates in a graph. They implement a graph as an edge-list: edges incident on a vertex are stored in a linked-list of edge-blocks. The vertices constitute a logical vertex array, thereby the edge-blocks are referenced. The edge-blocks contain the metadata such as timestamps and mark of valid edges. In practice, they allocate a big chunk

---

*Panigrahm is the Sanskrit translation of Marriage, which undoubtedly is a prominent event in our lives resulting in networks represented by graphs.*

---

*OPODIS 2021*
Non-Blocking Dynamic Unbounded Graphs

of memory (by default, half of the available system memory) to minimize cost of allocation for addition of edges after initialization. The removal of both edges and vertices is provided via metadata-based marks. However, vertex addition requires copying the entire structure. Furthermore, by design update operations are not allowed to be concurrent with queries. In contrast, PANIGRAHAM is concurrent, non-blocking, uses a hash-table for vertex-list and BSTs to contain edge-nodes. Moreover, the memory consumption is determined by the actual data contained in the data structure. Stinger was extended and optimized in some recent works such as GraphOne [34], GraphTinker [29]. GraphOne hybridizes an edge-list and an adjacency-list to support batch processing. Their methodology maintains versions of these lists to provide intermittent batch updates to an analytics engine. Clearly, for real-time lightweight settings, this method suffers from similar drawbacks as Stinger. On the other hand, GraphTinker builds upon Stinger and replaces linear probing with better hashed searches on the edge-lists. Their approach also shows some load balancing as the data structure grows. Nevertheless, none of these methods are efficient for dynamic vertex additions, and updates and queries are inherently sequentialized. By contrast, PANIGRAHAM provides fully concurrent queries and updates as a fundamental design component and ensures correctness (linearizability). Aspen [17] is another recent framework that extends Stinger to support graph updates with graph queries. However, the interface provided by them is very different - acquire, set and release. It is not immediately clear how to use their framework for concurrent graph updates and compare it with our framework.

Contributions and paper summary

- First, we describe the non-blocking directed graph data structure as a composition of lock-free hash table and binary search trees. (Section 2)

- After that, we introduce our novel framework as an interface operation with its correctness and progress guarantee (Section 3) followed by the descriptions of concurrent implementation of BFS, SSSP, and BC.

- We present an experimental evaluation of our algorithm comparing it against the existing parallel graph libraries Ligra [45] and Stinger [18] with respect to the throughput and memory footprint (Section 4). Our experiments demonstrate the power of non-blocking concurrency for dynamic updates in an application. Utilizing the parallel compute resources – 56 threads – in a standard multi-core machine, our implementation performs in some cases (a) up to 5x better than GraphOne (b) up to 10x better than Ligra and (c) 40x better than Stinger for BFS, SSSP, and BC algorithms. Significantly, for an identically initialized data structure and an identical random order selection of graph operations, we achieve up to 80x lighter memory footprint compared to Stinger (Section 4). In comparative terms, the most recent counterpart of our work is GraphTinker [29], who report up to 4x speedup in comparison with Stinger. Thus, the presented algorithm outperforms its latest competitor.

- Finally, we present an amortized analysis (Section 5) to theoretically contrast the worst case cost of our method against that of Ligra and Stinger. To the best of our knowledge, this is the first work on amortized upper bound for concurrent dynamic graph operations.
2 Non-blocking Graph Data Structure

Preliminaries

Our discussion uses a standard shared-memory model that supports atomic read, write, fetch-and-add (FAA), and compare-and-swap (CAS) instructions.

Background on the Graph Operations. A graph is represented as a pair \( G = (V, E) \), where \( V \) is the set of vertices and \( E \) is the set of edges. An edge \( e \in E \), \( e := (u, v) \) represents a pair of vertices \( u, v \in V \). In a directed graph\(^b\) \( e := (u, v) \) is an ordered pair, thus has an associated direction: emanating (outgoing) from \( u \) and terminating (incoming) at \( v \). We denote the set of outgoing edges from \( v \) by \( E_v \). Thus, \( \bigcup_{v \in V} E_v = E \). Each edge \( e \in E \) has a weight \( w_e \). A node \( v \in V \) is said reachable from \( u \in V \): \( v \leftrightarrow u \) if there are consecutive edges \( \{e_1, e_2, \ldots, e_n\} \subseteq E \) such that \( e_1 \) emanates from \( u \) and \( e_n \) terminates at \( v \).

1. **Breadth First Search (BFS):** Given a query vertex \( v \in V \), output each vertex \( u \in V - v \) reachable from \( v \). The collection of vertices happens in a BFS order: those at a distance \( d_1 \) from \( v \) are collected before those at a distance \( d_2 > d_1 \).

2. **Single Source Shortest Path (SSSP):** Given a vertex \( v \in V \), find a shortest path with respect to total edge-weight from \( v \) to every other vertex \( u \in V - v \). Note that, given a pair of nodes \( u, v \in V \), the shortest path between \( u \) and \( v \) may not be unique.

3. **Betweenness Centrality (BC):** Given a vertex \( v \in V \), compute \( BC(v) = \sum_{s \in V} \sum_{t \in V} \sigma(s, t \mid v) \) where \( \sigma(s, t) \) is the number of shortest paths between vertices \( s, t \in V \) and \( \sigma(s, t \mid v) \) is that passing through \( v \). \( BC(v) \) indicates the prominence of \( v \) in \( V \) and finds several applications where influence of an entity in a network is to be measured.

The Abstract Data Type (ADT)

Consider a weighted directed graph \( G = (V, E) \) as defined before. A vertex \( v \in V \) has an immutable unique key drawn from a totally ordered universe. For brevity, we denote a vertex with key \( v \): \( v(v) \) by \( v \) itself. Extending on the notations used in Section 1, we denote a directed edge with weight \( w \) from the vertex \( v_1 \) to \( v_2 \) as \( (v_1, v_2|w) \in E \). We consider an ADT \( \mathcal{A} \) as a set of operations: \( \mathcal{A} = \{ \text{PutV}(v), \text{RemV}(v), \text{GetV}(v), \text{PutE}(v_1, v_2|w), \text{RemE}(v_1, v_2), \text{GetE}(v_1, v_2), \text{BFS}(v), \text{SSSP}(v), \text{BC}(v) \} \) on \( G \).

1. A \( \text{PutV}(v) \) updates \( V \) to \( V \cup v \) and returns true if \( v \notin V \), otherwise it returns false without any update.

2. A \( \text{RemV}(v) \) updates \( V \) to \( V - v \) and returns true if \( v(v) \in V \), otherwise it returns false without any update.

3. A \( \text{GetV}(v) \) returns true if \( v \in V \), and false if \( v \notin V \).

4. A \( \text{PutE}(v_1, v_2|w) \)
   1. updates \( E \) to \( E \cup (v_1, v_2|w) \) and returns \( \langle \text{true}, \infty \rangle \) if \( v_1 \in V \land v_2 \in V \land (v_1, v_2|\cdot) \notin E \),
   2. updates \( E \) to \( E - (v_1, v_2|z) \cup (v_1, v_2|w) \) and returns \( \langle \text{true}, z \rangle \) if \( (v_1, v_2|z) \in E \),
   3. returns \( \langle \text{false}, w \rangle \) if \( (v_1, v_2|w) \notin E \) without updates,
   4. returns \( \langle \text{false}, \infty \rangle \) if \( v_1 \notin V \lor v_2 \notin V \) without updates.

5. A \( \text{RemE}(v_1, v_2) \) updates \( E \) to \( E - (v_1, v_2|w) \) and returns \( \langle \text{true}, w \rangle \) if \( (v_1, v_2|w) \in E \), otherwise it returns \( \langle \text{false}, \infty \rangle \) without any update.

6. A \( \text{GetE}(v_1, v_2) \) returns \( \langle \text{true}, w \rangle \) if \( (v_1, v_2|w) \in E \), otherwise it returns \( \langle \text{false}, \infty \rangle \).

\(^b\) In this paper we confine the scope of discussion to directed graphs only.
struct bucket {
    VNode vn;
}

struct HNode {
    bucket[];  // Other fields
    int size;
    HNode pred;
}

class VNode {
    int v;
    VNode vnxt;
    ENode enxt;
    OpItem oi;
}

class OpItem {
    int cnt, VisA[];
    ... // Other fields
}

class ENode {
    int e;
    double w;
    ENode el, er;
    VNode ptv;
}

class SNode {
    VNode n;
    SNode nxt, p;
    int cnt;
}

Figure 1 (a) and (b) Data structure components, (c) A sample directed graph, (d) Our implementation of (c).

7. A BFS(v), if v ∈ V, returns a sequence of vertices reachable from v arranged in a BFS order as defined before. If v /∈ V, it returns NULL.

8. An SSSP(v), if v ∈ V, returns a set S(v) = {d(v_i)}_{i ∈ V}, where d(v_i) is the summation of the weights of the edges on the shortest-path between v and v_i if v_i ← v, and d(v_i) = ∞, if v_i /← v. Note that d(v) = 0. There can be multiple paths between v and v_i with the same sum of edge-weights. If v /∈ V, it returns NULL.

9. A BC(v) returns the betweenness centrality of v as defined before, if v ∈ V. It returns NULL if v /∈ V.

A precondition for (v_1, v_2 | w) ∈ E is v_1 ∈ V ∧ v_2 ∈ V.

Data Structure Components

To facilitate both an efficient traversal and lock-freedom, we build the data structure based on a composition of a lock-free hash-table implementing the vertex-list, and lock-free BSTs implementing the edge-lists. On a skeleton of this composition, we include the design components for efficient traversals and (partial) snapshots. This is a more efficient design as compared to Chatterjee et al.’s approach [12] where the component dictionaries are implemented using lock-free linked-lists only.

More specifically, the nodes of the vertex-list are instances of the class VNode, see Figure 1(a). A VNode contains the key of the corresponding vertex along with a pointer to a BST implementing its edge-list. The most important member of a VNode is a pointer to an instance of the class OpItem, which facilitates anchoring of the traversals as described above.

The OpItem class, see Figure 1(b), encapsulates an array VisA of the size equal to the number of threads in the system, a counter cnt, and other algorithm specific indicators, which we describe in Section 3 while specifying the queries. An element of VisA simply keeps a count of the number of times the node is visited by a query performed by the corresponding

---

We limit our discussion to positive edge-weights only.
thread. The counter \texttt{cnt} is incremented every time an outgoing edge is added or removed at the vertex. This serves an important purpose of notifying a thread if the same edge is removed and added since the last visit.

The class \texttt{ENode}, see Figure 1(b), structures the nodes of an edge-list. It encapsulates a key, the edge-weight, the left- and right-child pointers and a pointer to the associated \texttt{VNode} where the edge terminates; the key in the \texttt{ENode} is that of the \texttt{VNode}; thus each \texttt{ENode} delegates a directed edge. The \texttt{VNodes} are bagged in a linked-list being referred to by a pointer from the \texttt{buckets}, see Figure 1(a). A resizable hash-table is constructed of the arrays of these \texttt{buckets}, wherein arrays are linked in the form of a linked-list of \texttt{HNodes}.

At the bare-bones level, our resizable vertex-list derives from the lock-free hash-table of Liu et al. \cite{37}, whereas the edge-lists extend the lock-free BST of Howley et al. \cite{28}. We introduce the \texttt{OpItem} fields in hash-table nodes. To facilitate non-recursive traversal in the lock-free BST, we use stacks. As we explain later, aligning the operations of the hash-table to the state of \texttt{OpItem} therein brings in nontrivial challenges.

The last but a significant component of our design is the class \texttt{SNode}, see Figure 1(b). It encapsulates the information to validate a scan of the graph to output a consistent specialized partial snapshot. More specifically, it packs the pointers to \texttt{VNodes} visited during a scan along with two pointers \texttt{nxt} and \texttt{p} to keep track of the order of their visit. The field \texttt{cnt} records the \texttt{cnt} counter of the corresponding visited \texttt{VNode}, which enables checking if the visited \texttt{VNode} has had any addition or removal of an edge since the last visit.

### Non-blocking Data Structure Construction

Having these components in place, we construct a non-blocking graph data structure in a modular fashion. Refer to Figure 1(d) depicting a partial implementation of a sample directed graph shown in Figure 1(c). The \texttt{ENodes}, shown as circles in Figure 1(d), with their children and parent pointers make lock-free internal BSTs corresponding to the edge-lists. For simplicity we have only shown the outgoing edges of vertex 5 in Figure 1(d) while the edges of other vertices are represented by small triangles. Thus, whenever a vertex has outgoing edges, the corresponding \texttt{VNode}, shown as small rectangles therein, has a non-null pointer pointing to the root of a BST of \texttt{ENodes}. The \texttt{VNodes} themselves make sorted lock-free linked-lists connected to the buckets of a hash-table. The buckets are cells of a bucket-array that implement the lock-free hash-table. When required, we add/remove bucket-arrays for an unbounded resizable dynamic design. The lock-free \texttt{VNode}-lists have two sentinel \texttt{VNodes}: \texttt{vh} and \texttt{vt} initialized with keys \(-\infty\) and \(\infty\), respectively.

We adopt the well-known technique of \texttt{pointer marking} – using a single-word \texttt{CAS} – via bit-stealing \cite{28,37} to perform lazy non-blocking removal of nodes. Concretely, on a common x86-64 architecture, memory has a 64-bit boundary and the last three least significant bits are unused; this allows us to use the last significant bit of a pointer to indicate first a \texttt{logical removal} of a node and thereafter detaching it from the data structure. Specifically, an \texttt{HNode}, a \texttt{VNode}, and an \texttt{ENode} is logically removed by marking its \texttt{pred}, \texttt{vnxt}, and \texttt{el} pointer, respectively. We call a node \texttt{alive} which is not logically removed.

### 3 PANIGRAHAM Framework

In this section, we describe a non-blocking algorithm that implements the ADT \(\mathcal{A}\). The operations \(\mathcal{M} := \{\text{PutV, RemV, GetV, PutE, RemE, GetE}\} \subseteq \mathcal{A}\) use the interface of the hash-table and BST with interesting non-trivial adaptation to our purpose. In the permitted space we describe the execution, correctness and progress property of the operations.
20:8  Non-Blocking Dynamic Unbounded Graphs

1: Operation Op(v)
2:   tid ← GetThreadID(); //get thread-id
3:   if (isMRRKd(v)) then
4:      return NULL; //Vertex is not present
5:   return SCAN(v, tid); //Invoke Scan
6: Method SCAN(v, tid)
7:   list(SNode) ot, nt; //Trees to hold the nodes
8:   ot ← TreeCollect(v, tid); //1st Collect
9:   while (true) do //Repeat the tree collection
10:      nt ← TreeCollect(v, tid); //2nd Collect
11:      if (CmpTree(ot, nt)) then
12:         return nt; //return if two collects are equal
13:      else if (ot = NULL ∧ nt = NULL) then
14:         return false; //Both the trees are not equal
15:      else
16:         ot ← ot.head; nt ← nt.head;
17:         while (ot = ot.tail ∧ nt = nt.tail) do
18:            if (ot.n ≠ nt.n ∨ ot.ecnt ≠ nt.ecnt ∨ ot.p ≠ nt.p) then
19:               return false; //Both the trees are not equal
20:            return true; //Both the trees are equal
21:      if (adjn.o.i.Visa[tid] = count) then
22:         return true;
23:      else
24:         return false;
Method
25:   CHKVISIT(adjn, tid, count)
26:   if (adjn.o.i.Visa[tid] = count) then
27:      return true;
28:   else
29:      return false;
Method
30:   TREECOLLECT(v, tid)
31:   queue(SNode) que; //Queue used for traversal
32:   list(SNode) st; cnt ← cnt + 1; //List to keep
33:   of the visited nodes
34:   v.o.i.Visa[tid] ← cnt;
35:   sn ← new CTNODE(v, NULL, NULL, v.o.i.ecnt); //Create a new SNode
36:   st.Add(en) (en enquene sn);
37:   while (~queue.empty()) do //Iterate all vertices
38:      v ← queue.deque(); //Get the front node
39:      if (isMRRKd(v)) then
40:         continue; //If marked then continue
41:      itn ← v.n.oxt; //Get the root ENode
42:      stack (ENode) S; //stack for inorder traversal
43:      /*Process all neighbors of v in the order of
44:       inorder traversal, as the edge-list is a BST*/
45:      while (itn ∨ ~S.empty()) do
46:         if (!isMRRKd(itn)) then
47:            S.push(itn); //push the ENode
48:         itn ← itn.el;
49:         if (!isMRRKd(itn)) then //Validate it
50:            adjn ← itn.ptr;
51:         if (!isMRRKd(adjn)) then //Validate it
52:            adjn.o.i.Visa[tid] ← cnt; //Mark it
53:            //Create a new SNode
54:            sn ← new CTNODE(adjn, v.n.oxt, v.o.i.ecnt);
55:            st.Add(sn); ///Insert sn to st
56:         que.enqueue(sn); //Push sn into the que
57:         ifn ← itn.ear;
58:         return st; //The tree is returned to the SCAN

\[ \mathcal{D} := \{\text{BFS, SSSP, BC} \} \subset \mathcal{A} \]. To de-clutter the presentation, we encapsulate the three queries in a unified framework. The framework comes with an interface operation Op. Op is specialized to the requirements of the three queries. The functionality of Op is presented in pseudo-code in Figures 2. Due to space constraints pseudo-code of the operations BFS, SSSP, and BC and detail descriptions are presented in the technical report [11].

Before describing the algorithm, it is important to specify its correctness and progress guarantee. In essence, we need to establish that during any execution the invariants corresponding to a consistent state of the data structure are satisfied, which are: (a) each edge-list maintains a BST order based on the ENode’s key e, and alive ENodes are reachable from enxt of the corresponding VNode, (b) a VNode that holds a pointer to a BST containing any alive ENode is itself alive, (c) each alive VNode is reachable from vh and vertex-lists connected to buckets are sorted based on the VNode’s keys v, and (d) an HNode which contains a bucket holding a pointer to an alive VNode is itself alive and an alive HNode is always connected to the linked-list of HNodes.

To prove linearizability [27], we describe the execution generated by the data structure as a collection of method invocation and response events. We assign an atomic step between the invocation and response as the linearization point (LP) of a method call (operation). Ordering the operations by their LPs provide a sequential history of the execution. We prove the correctness of the data structure by assigning a sequential history to an arbitrary
execution which is valid, i.e., it maintains the invariants. Furthermore, we argue that the data structure is non-blocking by showing that the queries would return in a finite number of steps if no update operation happens and hence is obstruction-free [26]. Moreover in an arbitrary execution at least one update operation returns in a finite number of steps and as a result is lock-free [26]. The details are provided in technical report [11].

Pseudo-code convention. We use \( p.x \) to denote the member field \( x \) of a class object pointer \( p \). To indicate multiple return objects from an operation we use \( \langle x_1, x_2, \ldots, x_n \rangle \).

To represent pointer-marking, we define three procedures: (a) \text{ISRMRKD}(p) returns \text{true} if the last significant bit of the pointer \( p \) is set to 1, else, it returns \text{false}, (b) \text{MRK}(p) sets the last significant bit of \( p \) to 1, and (c) \text{UNMRK}(p) sets the same to 0. An invocation of \text{CVNODE}(v) returns \text{false}, \text{CENODE}(e) and \text{CTNODE}(v), creates a new \text{VNode} with key \( v \), a new \text{ENode} with key \( e \) and a new \text{SNode} with a \text{VNode} \( v(v) \) respectively. For a newly created \text{VNode}, \text{ENode} and \text{SNode} the pointer fields are initialized with NULL value.

The execution pipeline of \text{OP} is presented at lines 1 to 5 in Algorithm 2. \text{OP} intakes a query vertex \( v \). It starts with checking if \( v \) is alive at Line 3. In the case \( v \) was not alive, it returns NULL. For this execution case, which results in \text{OP} returning NULL, the LP is at the atomic step (a) where \text{OP} is invoked in case \( v \) was not in the data structure at that point, and (b) where \( v \) was logically removed using a CAS in case it was alive at the invocation of \text{OP}.

Now, if \( v \) was alive, it proceeds to perform the method \text{SCAN}, Line 6 to 13. \text{SCAN} repeatedly performs (specialized partial) snapshot collection of the data structure along with comparing every consecutive pair of scans, stopping when a consecutive pair of collected snapshots are found identical. Snapshot collection is structured in the method \text{TREECOLLECT}, Line 29 to 59, whereas comparison of collected snapshots in performed by the method \text{CMPTREE}, Line 14 to 24.

Method \text{TREECOLLECT} performs a BFS traversal in the data structure to collect pointers to the traversed \text{VNodes}, thereby forming a tree. A cell of \text{VisA} corresponding to thread-id is marked on visiting it; notice that it is adaptation of the well-known use of node-dirty-bit for BFS [15]. The traversal over \text{VNodes} in facilitated by a queue: Line 30, whereas, exploring the outgoing edges at each \text{VNode}, equivalently, traversing over the BST corresponding to its edge-list uses a stack: Line 40. The snapshot collection for the queries BFS and BC are identical. For SSSP, where edge-weights are to be considered, the snapshot collection is optimized in each consecutive scan based on the last collection. At the core, the collected snapshot is a list of \text{SNodes}, where each \text{SNode} contains a pointer to a \text{VNode}, pointers to the next and previous \text{SNodes} and the value of the \text{ecnt} field of the \text{OpItem} of the \text{VNode}.

Method \text{CMPTREE} essentially compares two snapshots in three aspects: whether the collected \text{SNodes} contain (a) pointers to the same \text{VNodes} (b) have the same \text{SNode} being pointed by previous and next, and (c) have the same \text{ecnt}. The three checks ensure that a consistent snapshot is the one which has its collection lifetime \text{not concurrent} to (a) a vertex either added to or removed from it, (b) a path change by way of addition or removal of an edge, and, (c) an edge removed and then added back to the same position, respectively. Thus, at the completion of these checks, if two consecutive snapshots match, it is guaranteed to be unchanged during the time of the last two \text{TREECOLLECT} operations. Clearly, we can put a linearization point just after the atomic step where the last check is done: Line 19 or 22, where \text{CMPTREE} returns.

Now, it is clear that any \( q \in \mathcal{D} \) does not engage in helping any other operation. Furthermore, an \( m \in \mathcal{M} \) does not help a \( q \in \mathcal{D} \). Thus, given an execution \( E \) as a collection of operation calls belonging to \( \mathcal{M} \), by the fact that the data-structures hash-table and BST are
lock-free, and whenever no PutV, PutE, RemV, and RemE happen, a \( q \in Q \) returns, we infer that the presented algorithm is non-blocking. In Appendix A, we present the details of each of the operations.

Fundamentally, the functionalities of BFS, SSSP and BC queries are tailored by specialized construction of corresponding OpItem objects according to the requirements of their respective partial snapshots. As mentioned above, these queries are obstruction-free. In the technical report [11], we present the details of each of the queries.

4 Experiments

In this section, we describe the experimental evaluation of our non-blocking graph algorithms against three well-known existing batch analytics methods: (a) Stinger [18], (b) Ligra [45], and (c) GraphOne [34].

Dataset. We use (a) a standard synthetic dataset – R-MAT graphs [9] – with power-law distribution of degrees, and (b) real-life SNAP{EmailEuAll, Slashdot0811, socEpinions1, and WikiVot} [36] graph dataset.

Algorithms. While Stinger provides dynamic edge addition and removal and vertex removal operations, Ligra is built for static queries. However, these libraries do not allow concurrent updates with queries: we execute dynamic vertex and edge updates by intermittent sequential addition and removal. As explained earlier, we needed the repeated snapshot collection and validation methodology to guarantee linearizability of graph queries. However, if the consistency requirement is not as strong as linearizability, we can still have non-blocking progress if we collect the snapshots only once, i.e., we stop the scan algorithm after a single round of snapshot collection. At the cost of theoretical consistency, we gain a lot in terms of throughput, which is the primary demand of the analytics applications, who often go for approximate queries. Thus, the experiments include the following methods: (1) PG-Cn: Linearizable PANIGRAHAM, (2) PG-Icn: Inconsistent PANIGRAHAM, (3) Ligra, (4) Stinger, and (5) GraphOne. Note that, while all the libraries provide BFS queries, only Ligra supports SSSP and BC.

The choice of the competitors. One clear advantage of PANIGRAHAM over each of the lately developed dynamic graph frameworks, such as GraphOne [34] and GraphTinker [29], is that in a dynamic setting these frameworks do not provide any direct or intuitive method for vertex removal. The dynamic property of the graph in these frameworks is primarily with regards to the edges. While keeping the requirement for having support of dynamic vertex and edges, we zeroed on Ligra [18] and Stinger [45] for comparisons. We also compare the results of BFS on GraphOne [34] with concurrent PutE, and RemE operations by keeping a fixed number of vertices.

Experimental Setup. We conducted our experiments on a system with Intel(R) Xeon(R) E5-2690 v4 CPU packing 56 cores with a clock speed of 2.60GHz. There are 2 logical threads for each core and each having a private 64KB L1 and 256KB L2 cache. The 35840KB L3 cache is shared across the cores. The system has 32GB of RAM and 1TB of hard disk. It runs on a 64-bit Linux operating system. All implementations are in C++ without garbage collection. We used Posix threads for multi-threaded implementation.

\( ^d \) The source code is available on https://github.com/PDCRL/PANIGRAHAM.
The experiments start with a graph instance populating the data structure. At the execution initialization, we spawned a fixed set of threads (7, 14, 28 and 56). During the runtime, each thread randomly performed a set of operations chosen from a certain random workload distribution. The random workload pre-constructed and the same across all experiments. Each experiment was executed for 5 iterations. After ignoring the initial 5% operations for warm-up and we took the average of the remaining operations. We considered two evaluation metrics: (i) the latency: total time taken to complete the set of operations, after a fixed warm-up – 5% of the total number of operations, and (ii) the memory footprint.

Workload Distribution. In each micro-benchmark, first we loaded a graph instance, thereafter performed warm-up operations, followed by an end-to-end run of $10^4$ operations in total, assigned in a uniform random order to the threads. We used a range of distributions over an ordered (family of) set of operations: \{\text{Op}, \text{Vertex-Updates}:=\{\text{PutV}, \text{RemV}\}, \text{Edge-Updates}:=\{\text{PutE}, \text{RemE}\}\}. A sample label, say, 20/60/20 on a performance plot refers to a distribution \{\text{Op}: 20\%, \{\text{PutV}: 30\%, \text{RemV}: 30\\}, \{\text{PutE}: 10\%, \text{RemE}: 10\\}\}. 

---

**Figure 3** Latency of the executions containing Op: (1) BFS ((a), (b), and (c)) on a graph of size $|V|= 131K$ and $|E|= 2.44M$, (2) SSSP ((d), (e), and (f)) on a graph of size $|V|= 8K$ and $|E|= 80K$, and (3) BC ((g), (h), and (i)) on a graph of size $|V|= 16K$ and $|E|= 160K$. X-axis and Y-axis units are the number of threads and time in second, respectively.
Non-Blocking Dynamic Unbounded Graphs

Figure 4  Latency of the executions containing OP: BFS ((a), (b), and (c)) on a graph of size $|V|=65K$ and $|E|=500K$. Total $10^4$ operations were performed with given distributions. The distributions for each cases is: BFS/PUTE/REME, e.g., 2/49/49: {BFS: 2%, PUTE: 49%, REME: 49%}. X-axis unit is the number of threads.

Experimental Observations and Discussion

Figure 3 to 10 show the evaluation results. In the following we highlight the significant experimental observations.

Figure 5  The memory footprint during the run-time corresponding to the workload distribution 10/45/45 as plotted in Figures 3(c), 3(f) and 3(i).

Scalability.  See Figure 3; the concurrent methods scale well with the number of threads irrespective of the workload and graph size, whereas Stinger shows negligible scalability. With higher proportion of queries in the workload, Ligra starts scaling. This shows that concurrency in dynamic analytics is a natural way to scale-up.

GraphOne vs PANIGRAHAM.  GraphOne [34] does not allow vertex updates. Unlike Stinger and Ligra, wherein copying the allocated graph structure to a new memory-location was a workaround, the GraphOne interface does not let the structure of the allocated graph be retrieved, thus disallowing any obvious possibility of PUTV and REMV operations. Thus, the only experimental comparison of PG-Cn and PG-Icn with GraphOne was in regards to concurrent executions of BFS, PUTE and REME operations by fixing the number of vertices. Figure 4 depicts different workloads and with a graph having a fixed number of vertices. We observe that for the chosen graphs sizes and the common workload, that well represent a dynamic size, GraphOne exhibits severely limited scalability with the number of threads, in comparison to PG-Cn and PG-Icn.
Figure 6 Consistent concurrent analytics of PG-Cn against Stinger: the execution latency of BFS is plotted for 56 threads for different graph-sizes as labeled on x-axis: \{(V/E), (1) : 1K/10K, (2) : 8K/80K, (3) : 16K/160K, (4) : 32K/320K, (5) : 65K/500K\}.

Memory footprint. Figure 5 shows that Stinger has approximately 80x heavier memory footprint in comparison with PG-Cn or PG-Icn for executions with BFS queries. The reason can be traced in the design of Stinger, whereby it pessimistically allocates a large chunk of memory. For SSSP and BC queries, wherein the OpItem object gets bigger to facilitate partial snapshot collection, Ligra gets advantage of compact CSR representation. However, in no case the allocated memory by PG-Cn or PG-Icn spills as drastically as Stinger.

Concurrency vs. Batch analytics. Figure 6 shows that PG-Cn offers two to four orders of magnitude speed-up in comparison with state-of-the-art Stinger for a given standard system setting. It clearly implies that a concurrent analytics framework can vastly improve on the existing methods of batch analytics.

Figure 7 Latency of the executions containing Op: BFS. In the 3D-plots, x-axis indicates the total time in seconds for an end-to-end run of \(10^6\) operations uniformly distributed according to the respective distributions. The dataset sizes as labeled on the y-axis are \{(V/E), (1) : 1K/10K, (2) : 8K/80K, (3) : 16K/160K, (4) : 32K/320K, (5) : 65K/500K\}.

Overall advantage of Concurrency. Having seen the comparative performance of Stinger-based batch analytics and the proposed consistent concurrent analytics framework, now we compare both the consistent and high performing inconsistent variants of PANIGRAHAM with a lightweight static framework Ligra adopted to the batch analytics setting (as noted earlier, Stinger and GraphOne do not support SSSP and BC queries). See Figures 7, 8, and 9. For smaller datasets, as well as for higher update workloads, both PG-Cn and PG-Icn outperform Ligra. As the query workload grows i.e., the overall workload gets closer to static, CSR based method exploits inline parallelization with lower cache misses, thus Ligra gets advantage. Still, PG-Icn decisively performs better than Ligra over the entire range of graph sizes and workload distribution. Notice that, in some cases, as we move from 28 to 56 threads, hyperthreading activates leading to cache thrashing, which limits CSR’s optimization; in the same way, in some cases, for higher thread contention PG-Cn’s performance also suffers.
20:14 Non-Blocking Dynamic Unbounded Graphs

- **Figure 8** End-to-end latency of the executions containing Op: SSSP. The plotting description is similar to that of Fig. 7. The graph sizes as labeled on the y-axis are \{(V/E), (1) : 1K/10K, (2) : 4K/30K, (3) : 8K/50K, (4) : 8K/70K, (5) : 8K/80K\}.

- **Figure 9** End-to-end latency of the executions containing Op: BC. The plotting description is similar to that of Fig. 7. The graph sizes as labeled on the y-axis are \{(V/E), (1) : 1K/10K, (2) : 2K/20K, (3) : 4K/40K, (4) : 8K/80K, (5) : 16K/120K\}.

- **Figure 10** Average number of concurrent modifications and scans during a query. Legends 60/20/20, etc. are identical to those in Fig. 7, 8 and 9.

5 Complexity Analysis

Given a graph \(G = (V, E)\), denote \(|V| = n, |E| = m, \delta = \max_{v \in V}(\delta_v)\), where \(\delta_v\) is the degree of vertex \(v\). As PANIGRAHAM (PG) consists of a hash-table and BSTs, Ligra uses CSR format, and Stinger uses edge-lists to represent \(G\), the worst-case cost of operations by each of them in a static setting are given in Table 1.

<table>
<thead>
<tr>
<th>Table 1 The static worst-case complexities.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algo.</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>PG</td>
</tr>
<tr>
<td>Ligra</td>
</tr>
<tr>
<td>Stinger</td>
</tr>
</tbody>
</table>

The worst-case cost of PUTO/REMOV/GETV for PG is due to the hash-table, and that of PUTO/REMEM/GETE is due to the BST and hash-table. The worst-case cost of PUTO/REMOV for Ligra comes from copying and shifting the entire structure, whereas, that for PUTO/REMEM
comes from shifting the edge array. GetV and GetE in Ligra relate to lookup in an index and a sorted array, respectively. Stinger behaves similar to Ligra in terms vertex operations, however, the edge-lists facilitate the worst-case linear cost in maximum degree δ only for the operations RemV/PUTE/REME/GETE here. The queries in each of the data-structure designs behave identically.

We define the state of a graph G as a tuple \( S_G = (n, m, \delta) \), where \( n, m, \delta \) are as aforementioned. Essentially, \( S_G \) captures the size and shape of \( G \). Now consider an execution – set of operation calls \( -X \) such that invocations and responses of operations \( \{ o \in X \} \) form a valid history \( H \) [25]. Thus, for an \( o \in X \), \( type(o) \in \mathcal{A} \), where \( type(o) \) denotes the type of \( o \) and \( \mathcal{A} \) is the ADT as described earlier. Denote the worst-case cost of \( o \), given \( o \) is invoked at an atomic time point when state of \( G \) was \( S_G \) by \( W_{o,S_G} \). The states of \( G \), being tuples, are ordered by dictionary order. In a dynamic setting, \( W_{o,S_G} \) is upper-bounded by the cost of \( o \) as performed in a static setting over the worst-case state, during the lifetime of \( o \), of \( G \) as defined in Lemma 1.

Let \( I_o \) and \( C_o \) be the interval contention [2] and point contention [3], respectively\(^*\), for an \( o \in X \). We name the execution cases – PG-Cn, PG-Icn, Ligra, and Stinger – as in section 4. Notice that, for an execution of Ligra and Stinger, \( I_o = C_o = 1 \) as there is no concurrency. For PG-Icn, \( C_o = 1 \) as even though the operations are performed concurrently, they are essentially not obliged to maintain any consistency, thus, do not cause “restart” to their peers though they may cause “cost escalation” which is captured by \( I_o \). Denote \( \tilde{I}_o = (I_o - 1) \), the total number of concurrent operation calls other than \( o \) itself (those responsible for a possible cost escalation) that were invoked between the invocation and response of \( o \). Lemma 1 is immediate:

**Lemma 1.** If an operation call \( o \in X \) is invoked at a state \( S_{G,o} = (n, m, \delta) \) of \( G \), the worst-case state of \( G \) that \( o \) can encounter is \( \bar{S}_{G,o} = (O(n + \tilde{I}_o), O(m + \tilde{I}_o), O(\delta + I_o)) \).

Denote \( X_\mathcal{M} = \{ o \in X \ | \ type(o) \in \mathcal{M}, \mathcal{M} \subseteq \mathcal{A} \} \), where \( \mathcal{A} \) is the ADT as defined in Section 2. Let \( I_o, \mathcal{U} \) and \( C_o, \mathcal{U} \) denote the interval and point contentions, respectively, of \( o \) pertaining to the operation calls \( o \in \{ X_\mathcal{U} \cup \{ o \} \} \). Without loss of generality, we consider executions \( X \), s.t. \( type(o) \in \mathcal{M} \cup \{ q \} \ \forall o \in X \), where \( \mathcal{M} = \{ \text{PUTV, REMV, PUTE, REME} \} \subseteq \mathcal{A}, q \in \mathcal{D} \) and \( \mathcal{D} = \{ \text{BFS, SSSP, BC} \} \subseteq \mathcal{A} \). This execution represents our experiments in section 4. The worst-case cost \( W_{o,S_G,o} \) of operation calls belonging to different \( type(o) \), in a static setting, are as listed in Table 1. Denote \( \mathcal{M}_V = \{ \text{PUTV, REMV} \}, \mathcal{M}_E = \{ \text{PUTE, REME} \} \), and \( \delta_o \) as the degree of vertex \( v_1 \), s.t. \( o \in \{ \text{PUTE}(v_1, v_2), \text{REME}(v_1, v_2) \} \). Using the fact that an operation \( o \in X \) s.t. \( type(o) \in \mathcal{M}_V \) can be obstructed by only an operation \( o' \in E \) s.t. \( type(o') \in \mathcal{M}_V \) and similarly for the set \( \mathcal{M}_E \), following the standard accounting method [13] an amortization over the update operations \( X_\mathcal{M} \) gives Lemma 2.

**Lemma 2.** The worst-case amortized cost per operation for an execution of \( X_\mathcal{M} \), denoted as \( A_{X_\mathcal{M}} \) is

\[
O \left( \frac{C_o,\mathcal{M}_V}{|X_\mathcal{M}|} \sum_{o \in X_\mathcal{M}} W_{o,S_G,o} + \frac{1}{|X_\mathcal{M}|} \sum_{o \in X_\mathcal{M}} W_{o,S_G,o} + \frac{C_o,\mathcal{M}_E}{|X_\mathcal{M}|} \sum_{o \in X_\mathcal{M}} O(\delta_e) \right)
\]

\(^*\) We are slightly adapting the original definitions of interval and point contention, where these notions are defined for processes invoking \( o \), to our terminologies.
Non-Blocking Dynamic Unbounded Graphs

Notice that the worst-case costs for individual operation calls $W_{o,S_G,o}$ consider a dynamic setting. Lemma 2 essentially infers that a careful accounting for amortization should consider only those concurrent operations that cause a CAS failure and thereby restart of an $o \in X$. Furthermore, an $o \in X_{\neq}$ restarts only from the vertex $v_1$ as mentioned above. Using a similar technique, and the fact that the queries by PG-Icn do not restart, we have Theorem 3.

**Theorem 3.** Denote

$$A_X(\mathcal{M}) = \frac{C_{o,\mathcal{M}}}{|E|} \sum_{o \in X_{\neq}} W_{o,S_G,o} + \frac{1}{|X|} \sum_{o \in X_{\neq} \mathcal{M}} W_{o,S_G,o} + \frac{C_{o,\mathcal{M}}}{|E|} \sum_{o \in X_{\neq}} O(\delta_c). \quad (1)$$

The worst-case amortized cost per operation $A_X$ for an execution of $o$ s.t. $type(o) \in \mathcal{M} \cup \{q\}$ for all $o \in X$, and $q \in \mathcal{Q} = \{BFS, SSSP, BC\}$ is

1. For $q \in \mathcal{Q}$ performed by PG-Icn,

$$A_X = A_X(\mathcal{M}) + \frac{1}{|X|} \sum_{o \in X_{\neq}} \left(W_{o,S_G,o} + \bar{I}_{o,\mathcal{M}}\right). \quad (2)$$

2. For $q \in \mathcal{Q}$ performed by PG-Cn,

$$A_X = A_X(\mathcal{M}) + \frac{C_{o,\mathcal{M}}}{|X|} \sum_{o \in X_{\neq}} \left(W_{o,S_G,o} + \bar{I}_{o,\mathcal{M}}\right). \quad (3)$$

Now, different from the concurrent analytics by PANIGRAHAM, in a batch analytics setting, the updates and queries selected at a random order are essentially performed sequentially, thus we have Theorem 4.

**Theorem 4.** For $q \in \mathcal{Q}$ performed by Ligra or Stinger is $O\left(\frac{1}{|X|} \sum_{o \in X} W_{o,S_G,o}\right)$.

Plugging in the worst-case costs from Table 1 gives the amortized costs in terms of the parameters $n, m, \delta$ of $G$.

**Remark 5 (Observed contention).** Notice that the worst-case amortized cost per operation for PG-Cn can be tightened by more careful accounting as one restart of a query execution $o \in X_{\neq}$ corresponds to all the modifications in $G$ that might have happened during its scan phase. We experimentally obtained the average number of concurrent modifications and scans as in TREECOLLECT for the queries as shown in Figure 10. Clearly, the average number of scans before a linearized response is much less than the average number of concurrent modifications during the lifetime of a query.

**Remark 6 (Parallel speedup).** Assuming that there are $p$ non-faulty threads in the shared-memory system, and each atomic step can be executed in a unit time-step, the worst-case amortized number of time-steps per operation for an execution of both PG-Cn and PG-Icn is roughly $A_X/p$, where $A_X$ is as given in Theorem 3. For Ligra and Stinger, the operation calls $o \in X_{\neq}$ get speedup due to parallel executions, whereas $o \in X_{\neq}$ are executed sequentially. If the parallel execution of an $o \in X_{\neq}$ has a speedup $s \leq p$, then the worst-case amortized number of time-steps per operation for an execution of Ligra or Stinger will be $O\left(\frac{1}{|X|} \sum_{o \in X_{\neq}} W_{o,S_G,o} + \frac{1}{s|X|} \sum_{o \in X_{\neq}} W_{o,S_G,o}\right)$. Clearly, the theoretical insights from the amortized analysis is corroborated by our experiments where we observed that even for a moderately sized graph, PG-Icn performs better than Ligra, whereas, for smaller graphs despite of costly consistent queries, PG-Cn outperforms the batch analytic methods.
6 Conclusion

In this paper, we presented a novel framework of concurrent dynamic graph analytics PANIGRAHAM. We implemented commonly used graph algorithms: breadth-first search, single-source-shortest-path, and betweenness centrality over this framework. The presented framework is versatile enough such that it can be extended to other graph algorithms that process the global information in a graph and are usually found in graph-based analytics. We proved that the presented algorithms are non-blocking and linearizable. From the perspective of higher performance at the cost of consistency, we presented an inconsistent variant as well. We extensively evaluated a C++ implementation of the algorithms that shows scalability of the method with parallel resources. Another important contribution of this paper is an amortized analysis of the graph operations in a concurrent consistent non-blocking setting.

To the best of our knowledge, this is the first work to provide amortized upper bound for concurrent dynamic graph operations. Unlike the well-known parallel batch analytics libraries, our framework honors the real-time order of updates and most significantly provides fully dynamic vertex additions, which has largely been unavailable previously. Its memory footprint is up to 80x lighter compared to Stinger and it provides up-to-three orders of magnitude better performance than Stinger.

The present work motivates two very important future works: (a) implementing lock-free variant of CSR representation of graphs to take advantage of cache efficiency and concurrency, and, (b) an amortized average-case analysis of these algorithm, which gives a more realistic picture of the implementations with respect to their theoretical behavior.

References


In this section, we present a detailed implementation of our non-blocking directed graph algorithm. The non-blocking graph composes on the basic structures of the dynamic non-blocking hash table [37] and non-blocking internal binary search tree [28]. For a self-contained reading, we present the algorithms of non-blocking hash table and BST. Because it derives and builds on the earlier works [37] and [28], many keywords in our presentation are identical to theirs. One key difference between our non-blocking BST design from [28] is that we maintain a mutable edge-weight in each BST node, thereby not only the implementation requires extra steps but also we need to discuss extra cases in order to argue the correctness of our design. Furthermore, we also perform non-recursive traversals in the BST for snapshot collections, which were already discussed as part of the graph queries. The pseudo-codes pertaining to the non-blocking hash-table are presented in Figure 12, whereas those for the non-blocking BST are presented in Figures 13.

### A.1 Structures

The declarations of the object structures that we use to build the data structure are listed in Figure 12 and 13. The structures FSet, FSetOp, and HNode are used to build the vertex-list, whereas Node, RelocateOp, and ChildCASOp are the component-objects of the edge-list.
structure FSet, a freezable set of VNodes that serves as a building block of the non-blocking hash table. An FSet object builds a VNode set with PUTV, REMV, and GETV operations, and in addition, provides a FREEZE method that makes the object immutable. The changes of an FSet object can be either addition or removal of a VNode. For simplicity, we encode PUTV and REMV operation as FSetOp objects. The FSetOp has a state optype (PUTV or REMV), the key value, done a boolean field that shows the operation was applied or not, and resp a boolean field that holds the return value.

The vertex-list is a dynamically resizable non-blocking hash table constructed with the instances of VNodes, and it is a linked-list of HNodes (Hash Table Node). The HNode composed of an array of buckets of FSet objects, the size field stores the array length and the predecessor HNode is pointed to by the pred pointer. The head of the HNode is pointed to by a shared Head pointer.

For clarity, we assume that a RESIZE method grows (doubles) or shrinks (halves) the size of the HNode which amount to modifying the length of the bucket array. The hash function uses modular arithmetic for indexing in the hash table, e.g. index = key mod size.

Figure 11 Pseudocodes of PUTV, REMV, GETV, PUTE, REME, GETE and CONVPLUS.
Based on the boolean parameter taken by the 

The non-blocking vertex operations  

To avoid the overhead of another field in the node structure, we use bit-manipulation: last weight

A.2 The Vertex Operations

A ChildCASOp object holds sufficient information for another thread to finish an operation that made changes to one of the child – right or left – pointers of a node. A node’s op field holds a flag indicating an active ChildCASOp operation. Similarly, a RelocateOp object holds sufficient information for another thread to finish an operation that removes the key of a node with both the children and replaces it with the next largest key. To replace the next largest key, we need the pointer to the node whose key is to be removed, the data stored in the node’s op field, the key to replacement and the key being removed. As we did in case of a ChildCASOp, the op field of a node holds a flag with a RELocate state indicating an active RelocateOp operation.

A.2 The Vertex Operations

The working of the non-blocking vertex operations PutV, RemV, and GetV are presented in Figure 11. A PutV(v) operation, at Lines 1 to 2, invokes HashAdd(v) to perform an insertion of a VNode v in the hash table. A RemV(v) operation at lines 3 to 4 invokes HashRem(v) to perform a deletion of VNode v from the hash table. The method Apply, which tries to modify the corresponding buckets, is called by both HashAdd and HashRem, see Line 109 and 114. It first creates a new FSetOp object consisting of the modification request, and then constantly tries to apply the request to the respective bucket b, see Lines
138 to 146. Before applying the changes to the bucket it checks whether \( b \) is \texttt{NULL}; if it is, \texttt{initBkt} method is invoked to initialize the bucket (Line 144). At the end, the return value is stored in the \texttt{resp} field.

The algorithm and the resizing hash table are orthogonal to each other, so we used heuristic policies to resize the hash table. As a classical heuristic we use a \texttt{HASHADD} operation that checks for the size of the hash table with some \texttt{threshold} value, if it exceeds the threshold the size of the table is doubled. Similarly, a \texttt{HASHREM} checks the threshold value, if it falls below threshold, it shrinks the hash table size to halves.

A \texttt{GetV(v)} operation, at Lines 5 to 10, invokes \texttt{HashCon(v)} to search a \texttt{VNode v} in the hash table. It starts by searching the given key \( v \) in the bucket \( b \). If \( b \) is \texttt{NULL}, it reads \( t \)'s predecessor (Line 122) \( s \) and then starts searching on it. At this point it could return an incorrect result as \texttt{HashCon} is concurrently running with resizing of \( s \). So, a double check at Line 123 is required to test whether \( s \) is \texttt{NULL} between Lines 120 and 122. Then, we re-read that bucket of \( t \) (Line 124 or 126), which must be initialized before \( s \) becomes \texttt{NULL}, and then we perform the search in that bucket. If \( b \) is not \texttt{NULL}, then we simply return the presence of the corresponding \texttt{VNode} in the bucket \( b \). Note that, at any point in time there are at most two \texttt{HNodes}: only one when no resizing happens and another to support resizing – halving or doubling – of the hash table.

### A.3 The Edge Operations

The non-blocking graph edge operations – \texttt{PutE}, \texttt{RemE}, and \texttt{GetE} – are presented in Figure 11. Before describing these operations, we detail the implementation of \texttt{Find} method, which is used by them. It is shown in Figure 13. The method \texttt{Find}, at Lines 199 to 227, tries to locate the position of the key by traversing down the edge-list of a \texttt{VNode}. It returns the position in \( pe \) and \( ce \), and their corresponding \texttt{op} values in \( peOp \) and \( ceOp \) respectively. The result of the method \texttt{Find} can be one of the four values: (1) \texttt{FOUND}: if the key is present in the tree, (2) \texttt{NOTFOUND_L}: if the key is not in the tree but might have been placed at the left child of \( ce \) if it was added by some other threads, (3) \texttt{NOTFOUND_R}: similar to \texttt{NOTFOUND_L} but for the right child of \( ce \), and (4) \texttt{ABORT}: if the search in a subtree is unable to return a usable result.

A \texttt{PutE(v1, v2|w)} operation, at Lines 26 to 51, begins by validating the presence of \( v1 \) and \( v2 \) in the vertex-list. If the validations fails, it returns \langle \texttt{false},\infty \rangle (Line 28). Once the validation succeeds, \texttt{PutE} operation invokes \texttt{Find} method in the edge-list of the vertex with key \( v1 \) to locate the position of the key \( v2 \). The position is returned in the variables \( pe \) and \( ce \), and their corresponding \texttt{op} values are stored in the \( peOp \) and \( ceOp \) respectively. On that, \texttt{PutE} checks whether an \texttt{ENode} with the key \( v2 \) is present. If it is present containing the same edge weight value \( w \), it implies that an edge with the exact same weight is already present, therefore \texttt{PutE} returns \langle \texttt{false},\infty \rangle (Line 36). However, if it is present with a different edge weight, say \( z \), \texttt{PutE} updates \texttt{ce}'s old weight \( z \) to the new weight \( w \) and returns \langle \texttt{true},z \rangle (Line 38). We update the edge-weight using a \texttt{CAS} to ensure the correct return in case there were multiple concurrent \texttt{PutE} operations trying to update the same edge. Notice that, here we are not freezing the \texttt{ENode} in anyway while updating its weight. The linearizability is still ensured, which we discuss in the next section.

If the key \( v2 \) is not present in the tree, a new \texttt{ENode} and a \texttt{ChildCASOp} object are created. Then using \texttt{CAS} the object is inserted logically into \texttt{ce}'s \texttt{op} field (Line 48). If the \texttt{CAS} succeeds, it implies that \texttt{ce}'s \texttt{op} field hadn’t been modified since the first read. Which in turn indicates that all other fields of \texttt{ce} were also not changed by any other concurrent
thread. Hence, the CAS on one of the ce’s child pointer should not fail. Thereafter, using a call to HELPCHILDCAS method the new ENode ne is physically added to the tree. This can be done by any thread that sees the ongoing operation in ce’s op field.

A REME(v1, v2) operation, at Lines 52 to 78, similarly begins by validating the presence of v1 and v2 in the vertex-list. If the validation fails, it returns (false, ∞). Once the validation succeeds, it invokes FIND method in the edge-list of the vertex having key v1 to locate the position of the key v2. If the key is not present it returns (false, ∞). If the key is present, one of the two paths is followed. The first path at Lines 63 to 67 is followed if the node has less than two children. In case the node has both its children present a second path at Lines 69 to 77 is followed. The first path is relatively simpler to handle, as single CAS instruction is used to mark the node from the state NONE to MARKED at this point the node is considered as logically deleted from the tree. After a successful CAS, a HELPMARKED method is invoked to perform the physical deletion. It uses a ChildCASOp to replace pe’s child pointer to ce’s only child pointer, or a NULL pointer if ce is a leaf node.

The second path is more difficult to handle, as the node has both the children. Firstly, FIND method only locates the children but an extra FIND (Line 69) method is invoked to locate the node with the next largest key. If the FIND method returns ABORT, which indicates that ce’s op field was modified after the first search, so the entire REME operation is restarted. After a successful search, a RelocateOp object replace is created (Line 72) to replace ce’s key v2 with the node returned. This operation added to replace’s op field safeguards it against a concurrent deletion while the REME operation is running by virtue of the use of a CAS (Line 73). Then HELPRELOCATE method is invoked to insert RelocateOp into the node with v2’s op field. This is done using a CAS, after a successful CAS the node is considered as logically removed from the tree. Until the result of the operation is known the initial state is set to ONGOING. If any other thread either sees that the operation is completed by way of performing all the required CAS executions or takes steps to perform those CAS operations itself, it will set the operation state from ONGOING to SUCCESSFUL, using a CAS. If it has seen other value, it sets the operation state from ONGOING to FAILED. After the successful state change, a CAS is used to update the key to new value and a second CAS is used to delete the ongoing RelocateOp from the same node. Then next part of the HELPRELOCATE method performs cleanup on replace by either marking it if the relocation was successful or clearing its op field if it has failed. If the operation is successful and ce is marked, HELPMARKED method is invoked to excise ce from the tree. At the end REME returns (true, ce.w)

Similar to PutE and REME, a GETE(v1, v2) operation, at Lines 11 to 19, begins by validating the presence of v1 and v2 in the vertex-list. If the validation fail, it returns (false, ∞). Once the validation succeeds, it invokes FIND method in the edge-list of the vertex with key v1 to locate the position of the key v2. If it finds v2, it checks if both the vertices are not marked and also the ceOp not marked; on ensuring that it returns (true, ce.w), otherwise, it returns (false, ∞).
struct FSetNode { int set; boolean ok; }
struct FSet { FSetNode node; }
struct FSetOp { int optype, key; boolean resp; }
struct HNode { FSet buckets; int size; HNode pred; }

Method GetResponse(op)
return op.resp;

Method HasMember(b, k)
o ← b.node; // local copy of b
return k ∈ o.set;

Method INVOKE(b, op)
o ← b.node; // local copy of b
while (o.ok) do
resp ← op.key ∉ o.set;
set ← o.set ∪ {op.key};
else
if (op.optype = ADD) then
resp ← op.key ∈ o.set;
set ← o.set \ {op.key};
if (CAS(b.node, o.n)) then
op.resp ← resp;
return true;
o ← b.node;
return false;

Method Freeze(b)
o ← b.node; // local copy of b
while (o.ok) do
n ← new FSetNode(o.set, false);
if (CAS(b.node, o.n)) then
break;
o ← b.node;
return o.set;

Operation HASHADD(key)
resp ← APPLY(ADD, key); }

Operation HASHREM(key)
resp ← APPLY(REMOVE, key);
if (heuristic-policy) then
RESIZE(true);
return resp;

Operation HASHCON(key)
t ← Head;
while (o.ok) do
s ← t.pred;
if (s ≠ NULL) then
b ← s.buckets(key mod s.size);
else
b ← t.buckets(key mod t.size);
return HasMember(b, key);

Method RESIZE(grow)
t ← Head;
if (t.size > 1 ∨ grow = true) then
for (i ← 0 to t.size-1) do
INITBKT(t.i);
t.pred ← NULL;
s.size ← grow ? t.size + 2 : t.size/2;
buckets ← new FSet[size];
t′ ← new HNode(buckets, size, t);
CAS(Head, t, t′);

Method APPLY( optype, key)
op ← new FSetOp( optype, key, false, t);}
while (true) do
118: if (heuristic-policy) then
119: RESIZE(true);
120: return resp;
121: Operation HASHREM(key)
122: resp ← APPLY(REMOVE, key);
123: if (heuristic-policy) then
124: RESIZE(false);
125: return resp;
126: Operation HASHCON(key)
127: t ← Head;
128: b ← t.buckets[key mod t.size ];
129: if (b = NULL) then
130: s ← t.pred;
131: if (s ≠ NULL) then
132: b ← s.buckets(key mod s.size);
133: else
134: b ← t.buckets(key mod t.size);
135: return HasMember(b, key);
136: Method RESIZE(grow)
137: t ← Head;
138: if (t.size > 1 ∨ grow = true) then
139: for (i ← 0 to t.size-1) do
140: INITBKT(t.i);
141: t.pred ← NULL;
142: t.size ← grow ? t.size + 2 : t.size/2;
143: buckets ← new FSet[size ];
144: t′ ← new HNode(buckets, size, t);
145: CAS(Head, t, t′);
146: Operation APPLY( optype, key)
op ← new FSetOp( optype, key, false, t);}
147: while (true) do
148: 149: t ← Head;
150: b ← t.buckets[key mod t.size ];
151: if (b = NULL) then
152: b ← INITBKT(t.key mod t.size);
153: if (INVOKE(b, op)) then
154: return GetResponse(op);
155: Figure 12 Structure of FSet, FSetOp and HNode. Pseudocode of INVOKE, Freeze, Add, and Remove methods based on dynamic sized non-blocking hash table[37].
\begin{verbatim}
147: Method INITBkt(i, key)
148:  b ← f.buckets[key];
149:  s ← f.pred;
150: if (b = NULL ∧ s ≠ NULL) then
151:    if (f.size = s.size) then
152:      m ← s.buckets[i mod s.size];
153:      set ← FREEZE(m) ∩ { x | x mod f.size = i };
154:    else
155:      m ← s.buckets[i];
156:      m′ ← s.buckets[i + s.size];
157:      set ← FREEZE(m) ∪ FREEZE(m′);
158:      b′ ← new FSet(set, true);
159:      CAS(f.buckets[i], NULL, b′);
160: return f.buckets[i];
161: struct Node(int key;Operation op; Node left,right;
162: struct RelocateOp (int state,removeKey,replaceKey;
163: Node dest, Operation destOp;
164: struct ChildCASOp (boolean istLeft;
165: Node expected, update; )
166: Operation Add(key)
167:  Node pred, curr, newNode;
168:  Operation predOp, currOp, casOp;
169:  int result;
170:  while (true) do
171:    result ← Find(key, predPredOp, currOp, currOp, root);
172:    if (result = FOUND) then return false;
173:    newNode ← new Node(key);
174:    Boolean istLeft ← (result = NOTFOUND_L);
175:    Node old ← istLeft ? curr.left : curr.right;
176:    casOp ← new ChildCASOp(istLeft, old, newNode);
177:    if (CAS(currOp, currOp, FLARM(currOp, ChildCAS)))
178:        then
179:            HELPCASASOp(casOp, curr);
180:            return true;
181:    Operation Remove(key)
182:  Node pred, curr, replace;
183:  Operation predOp, currOp, replaceOp, relocOp;
184:  while (true) do
185:    if (Find(key, pred, predOp, curr, currOp, root) ≠ NOTFOUND) then
186:        return false;
187:    if (isNOTNULL(curr.right) ∧ isNOTNULL (curr.left))
188:        then
189:            HELPMARKED(pred, predOp, curr);
190: else
191:    if (CAS(currOp, currOp, FLAG(currOp, MARKED)))
192:        then
193:            HELPMARKED(pred, predOp, curr);
194: else
196:        then
197:            HELPRELOCATE(relocOp, pred, predOp, replaceOp) then
198:                return true;
199:    else
200:        HELPMARKED(pred, predOp, curr);
201:    return false;
202: Method FIND(key, pred, predOp, curr, currOp, root)
203:  int result, currKey ; Node next, lastRight;
204:  Operation lastRightOp; result ← NOTFOUND_R;
205:  curr ← root; currOp ← currOp;
206:  if (!GETFLAG(currOp) ≠ NULL) then
207:    if (root = root) then
208:        HELPCHILDASOp(UNFLAG(currOp), curr);
209:    goto Line 201;
210:  else return ABORT;
211:    next ← curr.left; lastRight ← curr;
212:    lastRightOp ← currOp;
213:    while (~ (isNULL(next)) do
214:        pred ← curr; predOp ← currOp;
215:        curr ← next; currOp ← currOp;
216:        if (GETFLAG(currOp) ≠ NULL) then
217:            HELPLAST(pred, predOp, currOp, currOp);
218:        goto Line 201;
219:        currKey ← curr.key;
220:        if (key < currKey) then
221:            result ← NOTFOUND_R; next ← curr.left;
222: else
223:    if (key > currKey) then
224:        result ← NOTFOUND_R; next ← curr.right;
225:        lastRight ← curr; lastRightOp ← currOp;
226: else
227:    result ← FOUND; break;
228:    if (result ≠ FOUND) ∧ (lastRightOp ≠ lastRightOp) then goto Line 201;
229:    if (currOp ≠ currOp) then goto Line 201;
230:    return result;
\end{verbatim}

Figure 13 Structure of Node, RelocateOp and ChildCASOp. Pseudocodes of Add, Remove, Contains and Find methods based on non-blocking binary search tree[28]. Pseudocodes of Contains, Resize, Apply and initBkt methods based on dynamic sized non-blocking hash table[37].