Non-blocking Dynamic Unbounded Graphs with Worst-case Amortized Bounds

Bapi Chatterjee
Institute of Science and Technology Austria
bhaskerchatterjee@gmail.com

Komma Manogna
Indian Institute of Technology Hyderabad
cs18mtech11021@iith.ac.in

Sathya Peri
Indian Institute of Technology Hyderabad
sathya_p@cse.iith.ac.in

Muktikanta Sa
Télécom SudParis - Institut Polytechnique de Paris
muktikanta.sa@gmail.com

Abstract

Today's applications, in particular, the analytics tasks based on graph algorithms 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 current 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 counterpart. 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.2021.21

Related Version A full version of the paper is available at https://arxiv.org/abs/2003.01697

Acknowledgements 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).

1 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 [a]: **Pra**ctical **N**on-block**i**ng **Gra**ph Algorit**hm**s.

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 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 deletion of a node in the lock-free hash-table, whereas, addition/removal of an edge translates to the same operation in 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 to a graph data structure. 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] Panigraham is the Sanskrit translation of Marriage, which undoubtedly is a prominent event in our lives resulting in networks represented by graphs.

a big chunk 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 sets. (Section 3)
- After that, we introduce our novel framework as an interface operation with its correctness and progress guarantee (Section 4) 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 5). 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 orderly selection of graph operations, we achieve up to 80x lighter memory footprint compared to Stinger (Section 5). 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 6) 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 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: $v(\mathsf{v})$ 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 $(\mathsf{v}_1, \mathsf{v}_2|w) \in E$. We consider an ADT $\mathcal{A}$ as a set of operations: $\mathcal{A} = \{\textsc{PutV}(\mathsf{v}), \textsc{RemV}(\mathsf{v}), \textsc{GetV}(\mathsf{v}), \textsc{PutE}(\mathsf{v}_1, \mathsf{v}_2|w), \textsc{RemE}(\mathsf{v}_1, \mathsf{v}_2), \textsc{GetE}(\mathsf{v}_1, \mathsf{v}_2), \text{BFS}(\mathsf{v}), \text{SSSP}(\mathsf{v}), \text{BC}(\mathsf{v})\}$ on G.

1. A $\textsc{PutV}(\mathsf{v})$ updates V to $V \cup \mathsf{v}$ and returns `true` if $\mathsf{v} \notin V$, otherwise it returns `false` without any update.
2. A $\textsc{RemV}(\mathsf{v})$ updates V to $V - \mathsf{v}$ and returns `true` if $v(\mathsf{v}) \in V$, otherwise it returns `false` without any update.
3. A $\textsc{GetV}(\mathsf{v})$ returns `true` if $\mathsf{v} \in V$, and `false` if $\mathsf{v} \notin V$.
4. A $\textsc{PutE}(\mathsf{v}_1, \mathsf{v}_2|w)$
   (a) updates E to $E \cup (\mathsf{v}_1, \mathsf{v}_2|w)$ and returns $\langle$`true`$, \infty\rangle$ if $\mathsf{v}_1 \in V \land \mathsf{v}_2 \in V \land (\mathsf{v}_1, \mathsf{v}_2|\cdot) \notin E$,
   (b) updates E to $E - (\mathsf{v}_1, \mathsf{v}_2|z) \cup (\mathsf{v}_1, \mathsf{v}_2|w)$; returns $\langle$`true`$, z\rangle$ if $(\mathsf{v}_1, \mathsf{v}_2|z) \in E$,
   (c) returns $\langle$`false`$, w\rangle$ if $(\mathsf{v}_1, \mathsf{v}_2|w) \in E$ without updates,
   (d) returns $\langle$`false`$, \infty\rangle$ if $\mathsf{v}_1 \notin V \lor \mathsf{v}_2 \notin V$ without updates.
5. A $\textsc{RemE}(\mathsf{v}_1, \mathsf{v}_2)$ updates E to $E - (\mathsf{v}_1, \mathsf{v}_2|w)$ and returns $\langle$`true`$, w\rangle$ if $(\mathsf{v}_1, \mathsf{v}_2|w) \in E$, otherwise it returns $\langle$`false`$, \infty\rangle$ without any update.
6. A $\textsc{GetE}(\mathsf{v}_1, \mathsf{v}_2)$ returns $\langle$`true`$, w\rangle$ if $(\mathsf{v}_1, \mathsf{v}_2|w) \in E$, otherwise it returns $\langle$`false`$, \infty\rangle$.
7. A BFS(v), if $\mathsf{v} \in V$, returns a sequence of vertices reachable from v arranged in a BFS order as defined before. If $\mathsf{v} \notin V$ or $\nexists \mathsf{v}' \in V$ s.t. $(\mathsf{v}, \mathsf{v}'|\cdot) \in E$, it returns `NULL`.
8. An SSSP(v), if $\mathsf{v} \in V$, returns a set $S(\mathsf{v}) = \{d(\mathsf{v}_i)\}_{\forall \mathsf{v}_i \in V}$, where $d(\mathsf{v}_i)$ is the summation of the weights of the edges[b] on the shortest-path between v and v_i if $\mathsf{v}_i \hookleftarrow \mathsf{v}$, and $d(\mathsf{v}_i) = \infty$, if $\mathsf{v}_i \not\hookleftarrow \mathsf{v}$. Note that $d(\mathsf{v}) = 0$. There can be multiple paths between v and v_i with the same sum of edge-weights. If $\mathsf{v} \notin V$, it returns `NULL`.
9. A BC(v) returns the betweenness centrality of v as defined before, if $\mathsf{v} \in V$. It returns `NULL` if $\mathsf{v} \notin V$.

A precondition for $(\mathsf{v}_1, \mathsf{v}_2|w) \in E$ is $\mathsf{v}_1 \in V \land \mathsf{v}_2 \in V$.

3 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[c] $e := (u, v)$ 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 $u \in V$: $v \hookleftarrow u$ 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.

[b] We limit our discussion to positive edge-weights only.
[c] In this paper we confine the scope of discussion to directed graphs only.

```
struct bucket {
  VNode vn;
}
struct HNode {
  bucket [];
  int size;
  HNode pred;
}
class VNode {
  int v;
  VNode vnxt;
  ENode enxt;
  OpItem oi;
}
```

(a)

```
class OpItem {
  int ecnt, VisA [];
  ··· // Other fields
}
class ENode {
  int e;
  double w;
  ENode el, er;
  VNode ptv;
}
class SNode {
  VNode n;
  SNode nxt, p;
  int ecnt;
}
```

(b)

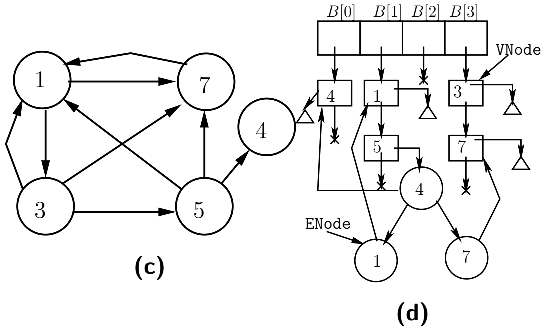


Figure 1 (a) and (b) Data structure components, (c) A sample directed graph, (d) Our implementation of (c).

3. **Betweeness Centrality (BC)**: Given a vertex $v \in V$, compute $BC(v) = \sum_{s,t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)}$, where $\sigma(s,t)$ is the number of shortest paths between vertices $s, t \in V$ and $\sigma(s,t|v)$ 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.

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 `ecnt`, and other algorithm specific indicators, which we describe in Section 4 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 thread. The counter `ecnt` 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 `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 `VNode` where the edge terminates; the key in the `ENode` is that of the `VNode`; thus each `ENode` delegates a directed edge. The `VNodes` are bagged in a linked-list being referred to by a pointer from the `buckets`, see Figure 1(a). A resizable hash-table is constructed of the arrays of these `buckets`, wherein arrays are linked in the form of a linked-list of `HNodes`.

At the bare-bones level, our resizable vertex-list derives from the lock-free hash-table of Liu et al. [37], whereas the edge-lists extend the lock-free BST of Howley et al. [28]. We introduce the `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 `OpItem` therein brings in nontrivial challenges.

The last but a significant component of our design is the class `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 `VNodes` visited during a scan along with two pointers `nxt` and `p` to keep track of the order of their visit. The field `ecnt` records the `ecnt` counter of the corresponding visited `VNode`, which enables checking if the visited `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 `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 `VNode`, shown as small rectangles therein, has a non-null pointer pointing to the root of a BST of `ENodes`. The `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 `VNode`-lists have two sentinel `VNodes`: `vh` and `vt` initialized with keys `-∞` and `∞`, respectively.

We adopt the well-known technique of *pointer marking* – using a single-word `CAS`– via bit-stealing [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 *logical removal* of a node and thereafter detaching it from the data structure. Specifically, an `HNode`, a `VNode`, and an `ENode` is logically removed by marking its `pred`, `vnxt`, and `el` pointer, respectively. We call a node *alive* which is not logically removed.

4 PANIGRAHAM Framework

In this section, we describe a non-blocking algorithm that implements the ADT $\mathcal{A}$. The operations M :={PutV, RemV, GetV, PutE, RemE, GetE } $\mathcal{A}$ 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 $Q := \{\text{BFS}, \text{SSSP}, \text{BC}\}$ $\mathcal{A}$. 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

```
Operation Op(v)
  tid ← GetThId();// get thread-id
  if (isMrkd(v)) then
    return NULL; //Vertex is not present
  return Scan(v, tid);//Invoke Scan
─────────────────────────────────────────
Method Scan(v, tid)
  list⟨SNode⟩ ot, nt ; //Trees to hold the nodes
  ot ← TreeCollect (v, tid); //1^st Collect
  while (true) do //Repeat the tree collection
    nt ← TreeCollect (v, tid); //2^nd Collect
    if (CmpTree (ot, nt)) then
      return nt;//return if two collects are equal
    ot ← nt;
─────────────────────────────────────────
Method CmpTree(ot, nt)
  if (ot = NULL ∨ nt = NULL) then
    return false;
  oit ← ot.head, nit ← nt.head;
  while (oit ≠ ot.tail ∧ nit ≠ nt.tail ) do
    if (oit.n ≠ nit.n ∨ oit.ecnt ≠ nit.ecnt ∨
oit.p ≠ nit.p) then
      return false; //Both the trees are not equal
    oit ← oit.nxt; nit ← nit.nxt;
  if (oit.n ≠ nit.n ∨ oit.ecnt ≠ nit.ecnt ∨ oit.p
≠ nit.p) then //Both the trees are not equal
    return false ;
  else return true ; //Both the trees are equal
─────────────────────────────────────────
Method ChkVisit(adjn, tid, count)
  if (adjn.oi.VisA [tid] = count) then
    return true;
  else return false ;
─────────────────────────────────────────
Method TreeCollect(v, tid)
  queue ⟨SNode ⟩ que; //Queue used for traversal
  list⟨SNode ⟩st; cnt ←cnt + 1; //List to keep
of the visited nodes
  v.oi.VisA [tid] ← cnt;
  sn←new CTNode(v,NULL,NULL,
  v.oi.ecnt);//Create a new SNode
  st.Add(sn);que.enque(sn);
  while (¬que.empty()) do //Iterate all vertices
    cvn ← que.deque(); // Get the front node
    if (isMrkd (cvn)) then
      continue;// If marked then continue
    itn ← cvn.n.enxt; //Get the root ENode
    stack ⟨ENode ⟩ S; // stack for inorder traversal
    /*Process all neighbors of cvn in the order of
    inorder traversal, as the edge-list is a BST*/
    while (itn ∨ ¬S.empty()) do
      while (itn ) do
        if (¬isMrkd(itn)) then
          S.push(itn); // push the ENode
        itn ← itn.el;
      itn ← S.pop();
      if (¬isMrkd(itn)) then //Validate it
        adjn ← itn.ptv;
        if (¬isMrkd (adjn)) then //Validate it
          if (¬ChkVisit (adjn, tid, cnt)) then
            adjn.oi.VisA [tid] ← cnt; //Mark it
            //Create a new SNode
            sn ← new CTNode(adjn,
             cvn,NULL,adjn.oi.ecnt);
            st.Add(sn); //Insert sn to st
            que.enque(sn); //Push sn into the que
      itn ← itn.er;
  return st; //The tree is returned to the Scan
```

Figure 2 Framework interface operation for graph queries.

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 $x_1, x_2, \ldots, x_n$. To represent pointer-marking, we define three procedures: (a) isMrkd(p) returns `true` if the last significant bit of the pointer p is set to 1, else, it returns `false`, (b) Mrk(p) sets the last significant bit of p to 1, and (c) UnMrk(p) sets the same to 0. An invocation of CVnode(`v`), CEnode(`e`) and CTNode(`v`), creates a new `VNode` with key `v`, a new `ENode` with key `e` and a new `SNode` with a `VNode` v(`v`) respectively. For a newly created `VNode`, `ENode` and `SNode` the pointer fields are initialized with `NULL` value.

The execution pipeline of Op is presented at lines 1 to 5 in Algorithm 2. 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 Op returning `NULL`, the LP is at the atomic step (a) where Op is invoked in case v was not in the data structre at that point, and (b) where v was logically removed using a `CAS` in case it was alive at the invocation of Op.

Now, if v was alive, it proceeds to perform the method Scan, Line 6 to 13. 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 TreeCollect, Line 29 to 59, whereas comparison of collected snapshots in performed by the method CmpTree, Line 14 to 24.

Method TreeCollect performs a BFS traversal in the data structure to collect pointers to the traversed `VNodes`, thereby forming a tree. A cell of `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 `VNodes` in facilitated by a queue: Line 30, whereas, exploring the outgoing edges at each `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 `SNode`s, where each `SNode` contains a pointer to a `VNode`, pointers to the next and previous `SNode`s and the value of the `ecnt` field of the `OpItem` of the `VNode`.

Method CmpTree essentially compares two snapshots in three aspects: whether the collected `SNode`s contain (a) pointers to the same `VNodes` (b) have the same `SNode` being pointed by previous and next, and (c) have the same `ecnt`. The three checks ensure that a consistent snapshot is the one which has its collection lifetime ***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 guranteed to be unchanged during the time of the last two 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 CmpTree returns.

Now, it is clear that any q *Q* does not engage in helping any other operation. Furthermore, an m *M* does not help a q *Q*. Thus, given an execution E as a collection of operation calls belonging to *A* , 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 *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.

5 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.

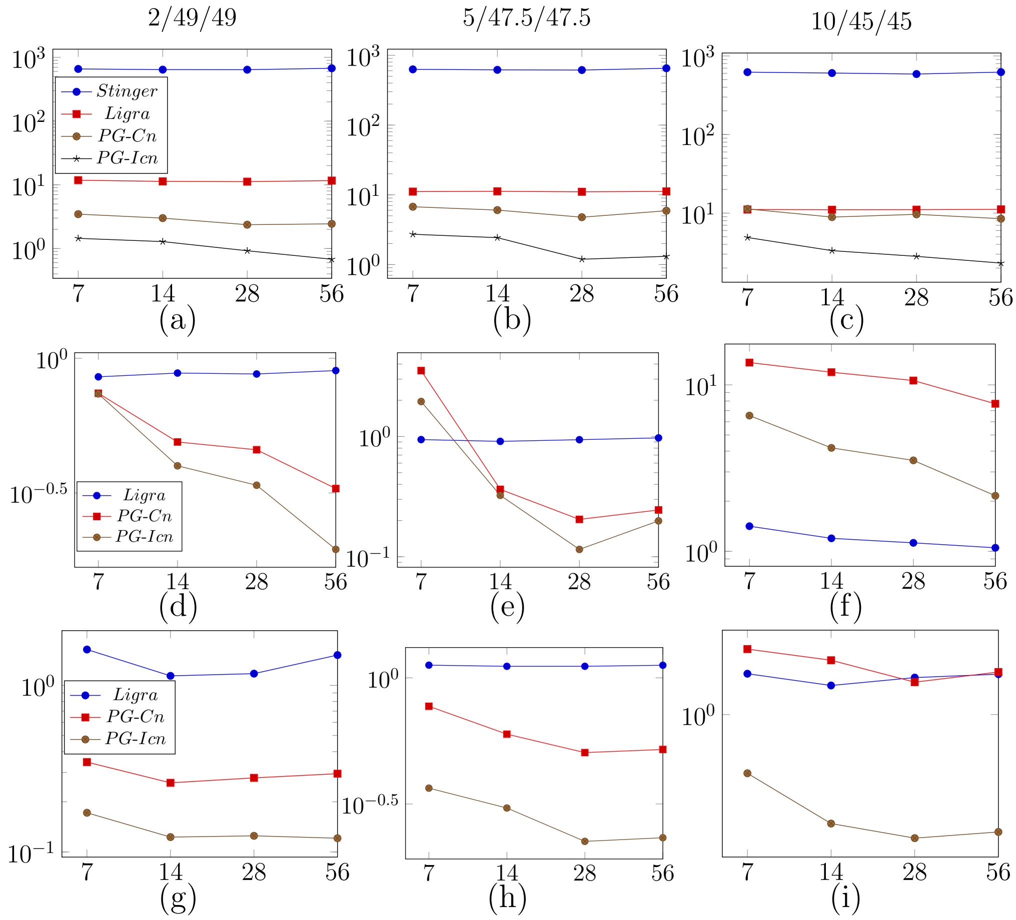


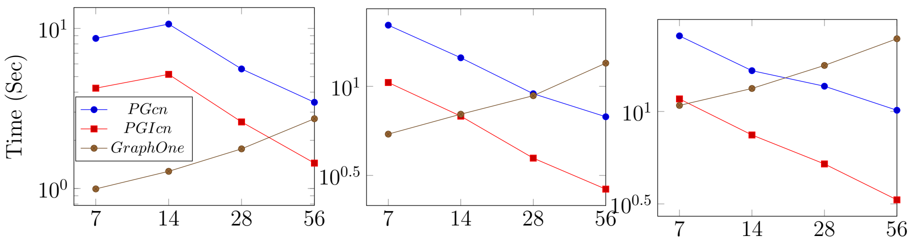
Figure 3 Latency of the executions containing Op: (1) BFS ((a), (b), and (c)) on a graph of size $|V| = 131K$ and $|E| = 2.44M$, (2) SSSP ((d), (e), and (f)) on a graph of size $|V| = 8K$ and $|E| = 80K$, and (3) BC ((g), (h), and (i)) on a graph of size $|V| = 16K$ and $|E| = 160K$. X-axis and Y-axis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

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[d] in C++ without garbage collection. We used Posix threads for multi-threaded implementation.

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 and then we took the mean. 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.



(a) 2/49/49 (b) 10/45/45 (c) 20/40/40

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 : $\{\text{BFS}: 2\%, \text{PutE}: 49\%, \text{RemE}: 49\%\}$. X-axis unit is the number of threads.

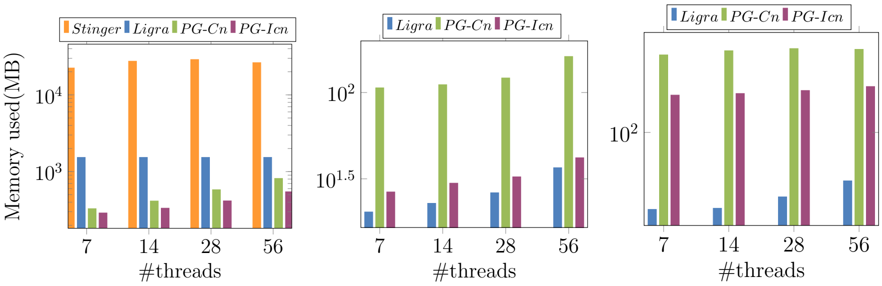
Workload Distribution: In each micro-benchmark, first we loaded a graph instance, thereafter performed warm-up operations, followed by an end-to-end run of 10^4 operations in total, assigned in a uniform random order to the threads. We used a range of distributions over an ordered (family of) set of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

[d] The source code is available on https://github.com/PDCRL/PANIGRAHAM

to a distribution $\{\textsc{Op} : 20\%, \{\textsc{PutV} : 30\%, \textsc{RemV} : 30\%\}, \{\textsc{PutE} : 10\%, \textsc{RemE} : 10\%\}\}$.

Experimental Observations and Discussion

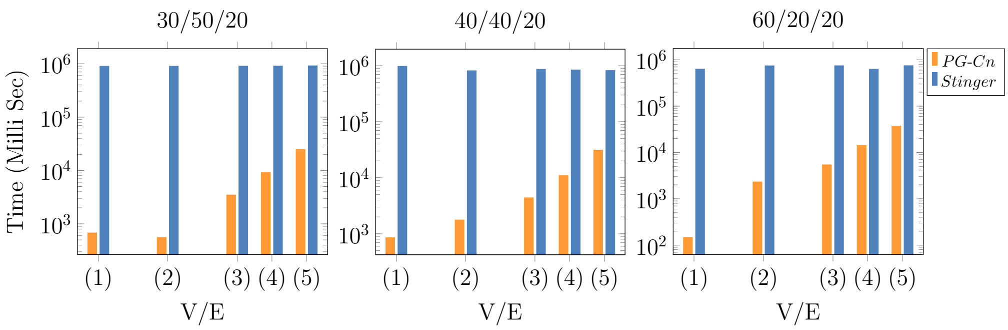
Figure 3 to 10 show the evaluation results. In the following we highlight the significant experimental observations.



(a) BFS (b) SSSP (c) BC

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:**



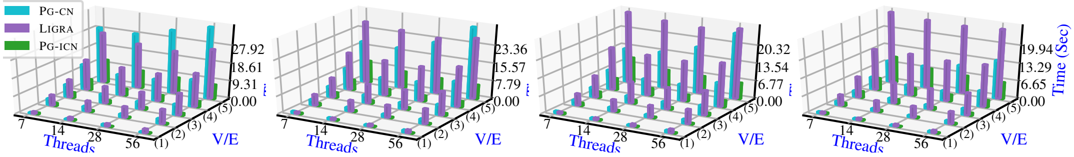
(a) (b) (c)

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: $\{(V/E), (1) : 1K/10K, (2) : 8K/80K, (3) : 16K/160K, (4) : 32K/320K, (5) : 65K/500K\}$.

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.

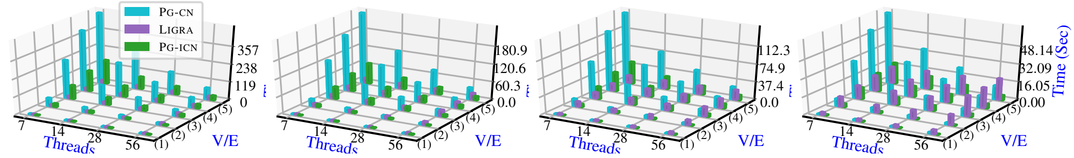
Memory footprint: We note in Figure 5 that Stinger has approximately 80x heavier memory footprint in comparison with PG-Cn or PG-Icn for executions with BFS queries. The reason can be traced in the design of Stinger, whereby it pessimistically allocates a large chunk of memory. For SSSP and BC queries, 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.



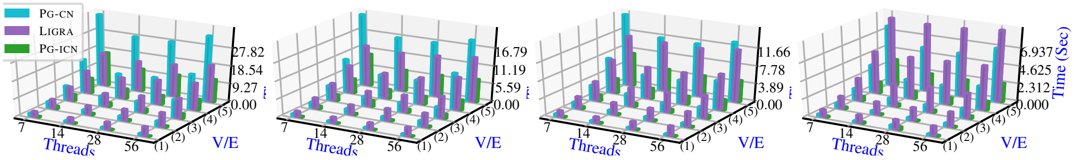
(a) 60/20/20 **(b)** 40/40/20 **(c)** 30/50/20 **(d)** 20/60/20

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



(a) 60/20/20 **(b)** 40/40/20 **(c)** 30/50/20 **(d)** 20/60/20

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 $\{(V/E), (1) : 1K/10K, (2) : 4K/30K, (3) : 8K/50K, (4) : 8K/70K, (5) : 8K/80K\}$.

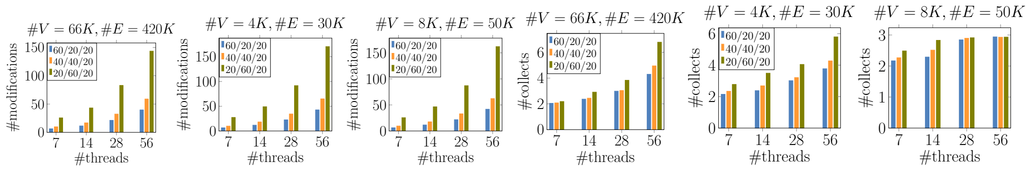


(a) 60/20/20 **(b)** 40/40/20 **(c)** 30/50/20 **(d)** 20/60/20

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 $\{(V/E), (1) : 1K/10K, (2) : 2K/20K, (3) : 4K/40K, (4) : 8K/80K, (5) : 16K/120K\}$.

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



(a) Op:BFS. (b) Op:SSSP. (c) Op:BC. (d) Op:BFS. (e) Op:SSSP. (f) Op:BC.

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.

6 Complexity Analysis

Given a graph $G = (V, E)$, denote $|V| = n$, $|E| = m$, $\delta = \max_{v \in V}(\delta_v)$, where δ_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.

The worst-case cost of PutV/RemV/GetV for PG is due to the hash-table, and that of PutE/RemE/GetE is due to the BST and hash-table. The worst-case cost of PutV/RemV for Ligra comes from copying and shifting the entire structure, whereas, that for PutE/RemE 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, δ 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[e], for an $o \in X$. We name the execution cases – PG-Cn, PG-Icn, Ligra, and Stinger – as in section 5. 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

[e] We are slightly adapting the original definitions of interval and point contention, where these notions are defined for processes invoking o, to our terminologies.

Algo.	PutV	RemV	PutE	RemE	GetV	GetE	BFS	SSSP	BC
PG	$O(n)$	$O(n)$	$O(n+\delta)$	$O(n+\delta)$	$O(n)$	$O(n+\delta)$	$O(n+m)$	$O(mn)$	$O(mn+n^2)$
Ligra	$O(n+m)$	$O(n+m)$	$O(m)$	$O(m)$	$O(1)$	$O(\log\delta)$	$O(n+m)$	$O(mn)$	$O(mn+n^2)$
Stinger	$O(n+m)$	$O(\delta)$	$O(\delta)$	$O(\delta)$	$O(1)$	$O(\delta)$	$O(n+m)$	–	–

Table 1 The static worst-case complexities.

peers though they may cause "cost escalation" which is captured by I_o. Denote $\widetilde{I_o} = (I_o - 1)$, 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:

I **Lemma 1.** *If an operation call $o \quad X$ is invoked at a state $S_{G,o} = (n, m, \delta)$ of G, the worst-case state of G that o can encounter is $\overline{S_{G,o}} = (O(n+\widetilde{I_o}), O(m+\widetilde{I_o}), O(\delta+\widetilde{I_o}))$.*

Denote $X_U = \{o \quad X \mid type(o) \quad U, U \quad A\}$, where A is the ADT as defined in Section 3. Let $I_{o,U}$ and $C_{o,U}$ denote the interval and point contentions, respectively, of o pertaining to the operation calls $o \quad \{X_U \quad \{o\}\}$. Without loss of generality, we consider executions X, s.t. $type(o) \quad M \quad \{q\} \quad o \quad X$, where $M = \{$PutV, RemV, PutE, RemE$\} \quad A$, $q \quad Q$ and $Q = \{$BFS, SSSP, BC$\} \quad A$. This execution represents our experiments in section 5. 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 $M_V = \{$PutV, RemV$\}$, $M_E = \{$PutE, RemE$\}$, and δ_o as the degree of vertex v_1, s.t. $o \quad \{$PutE$(v_1, v_2|w)$, RemE$(v_1, v_2)\}$. Using the fact that an operation $o \quad X$ s.t. $type(o) \quad M_V$ can be obstructed by only an operation $o \quad E$ s.t. $type(o) \quad M_V$ and similarly for the set M_E, following the standard accounting method [13] an amortization over the update operations X_M gives Lemma 2.

I **Lemma 2.** *The worst-case amortized cost per operation for an execution of X_M, denoted as A_{X_M} is*

$$O\left(\frac{C_{o,M_V}}{|X_M|}\sum_{o \quad X_{M_V}} W_{o,\overline{S_{G,o}}} + \frac{1}{|X_M|}\sum_{o \quad X_{M_E}} W_{o,\overline{S_{G,o}}} + \frac{C_{o,M_E}}{|X_M|}\sum_{o \quad X_{M_E}} O(\delta_e)\right).$$

Notice that here the worst-case costs for individual operation calls $W_{o,\overline{S_{G,o}}}$ 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 $o \quad X$. Furthermore, an $o \quad X_{M_E}$ 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

I **Theorem 3.** *Denote*

$$A_X(M) = \frac{C_{o,M_V}}{|E|}\sum_{o \quad X_{M_V}} W_{o,\overline{S_{G,o}}} + \frac{1}{|X|}\sum_{o \quad X_{M_E}} W_{o,\overline{S_{G,o}}} + \frac{C_{o,M_E}}{|E|}\sum_{o \quad X_{M_E}} O(\delta_e). \tag{1}$$

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

1. *For $q \quad Q$ performed by PG-Icn,*

$$A_X = A_X(M) + \frac{1}{|X|}\sum_{o \quad X_Q}\left(W_{o,\overline{S_{G,o}}} + I_{o,M}\right). \tag{2}$$

2. *For* $q \quad Q$ *performed by PG-Cn,*

$$A_X = A_X(M) + \frac{C_{o,M}}{|X|} \sum_{o \quad X_Q} \left(W_{o,\overline{S_{G,o}}} + I_o]_M \right). \tag{3}$$

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.

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

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

I 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 $o \quad X_Q$ 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.

I 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 $o \quad X_Q$ get speedup due to parallel executions, whereas $o \quad X_M$ are executed sequentially. If the parallel execution of an $o \quad X_Q$ 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 $O\left(\frac{1}{|X|}\sum_{o \quad X_M} W_{o,S_{G,o}} + \frac{1}{s|X|}\sum_{o \quad X_Q} W_{o,S_{G,o}}\right)$. 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.

7 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

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic Snapshots of Shared Memory. *Journal of the ACM*, 40(4):873–890, 1993.
2 Yehuda Afek, Gideon Stupp, and Dan Touitou. Long Lived Adaptive Splitter and Applications. *Distributed Comput.*, 15(2):67–86, 2002.
3 H. Attiya and A. Fouren. Alg. Adapting to Point Contention. *J. ACM*, 50(4):444–468, 2003.
4 Greg Barnes. A Method for Implementing Lock-free Shared-data Structures. In *SPAA*, pages 261–270, 1993.
5 Omar Batarfi, Radwa El Shawi, Ayman G Fayoumi, Reza Nouri, Ahmed Barnawi, Sherif Sakr, et al. Large Scale Graph Processing Systems: Survey and an Experimental Evaluation. *Cluster Computing*, 18(3):1189–1213, 2015.
6 Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *PPoPP*, pages 329–342, 2014.
7 A. Buluç, J. R. Gilbert, and V. B. Shah. Implementing Sparse Matrices for Graph Algorithms. In *Graph Algorithms in the Language of Linear Algebra*, volume 22, pages 287–313. 2011.
8 Hung Cao, Monica Wachowicz, and Sangwhan Cha. Developing an edge computing platform for real-time descriptive analytics. *2017 IEEE International Conference on Big Data (Big Data)*, pages 4546–4554, 2017.
9 Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *SDM*, pages 442–446, 2004.
10 Bapi Chatterjee, Nhan Nguyen, and Philippas Tsigas. Efficient Lock-free Binary Search Trees. In *PODC*, pages 322–331, 2014.
11 Bapi Chatterjee, Sathya Peri, and Muktikanta Sa. Dynamic Graph Operations: A Consistent Non-blocking Approach. *CoRR*, abs/2003.01697, 2020.
12 Bapi Chatterjee, Sathya Peri, Muktikanta Sa, and Nandini Singhal. A Simple and Practical Concurrent Non-blocking Unbounded Graph with Linearizable Reachability Queries. In *ICDCN*, pages 168–177, 2019.
13 Bapi Chatterjee, Ivan Walulya, and Philippas Tsigas. Help-optimal and Language-portable Lock-free Concurrent Data Structures. In *ICPP*, pages 360–369, 2016.
14 Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *EuroSys*, pages 85–98, 2012.
15 Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.
16 Nhan Nguyen Dang and Philippas Tsigas. Progress guarantees when composing lock-free objects. In *Euro-Par*, pages 148–159, 2011.
17 Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Low-Latency Graph Streaming Using Compressed Purely-Functional Trees. In *PLDI*, pages 918–934, 2019.
18 D. Ediger, R. McColl, J. Riedy, and D. A. Bader. STINGER: High Performance Data Structure for Streaming Graphs. In *HPEC*, 2012.
19 Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *PODC*, pages 131–140, 2010.
20 Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.
21 Timothy L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *DISC*, pages 300–314, 2001.

22 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A Lazy Concurrent List-Based Set Algorithm. *Parallel Processing Letters*, 17(4):411–424, 2007.

23 Danny Hendler, Nir Shavit, and Lena Yerushalmi. A Scalable Lock-free Stack Algorithm. In *SPAA*, pages 206–215, 2004.

24 Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *(ICDCS*, pages 522–529, 2003.

25 Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan K., 2008.

26 Maurice Herlihy and Nir Shavit. On the Nature of Progress. In *OPODIS*, pages 313–328, 2011.

27 Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

28 Shane V. Howley and Jeremy Jones. A Non-blocking Internal Binary Search Tree. In *SPAA*, pages 161–171, 2012.

29 Wole Jaiyeoba and K. Skadron. Graphtinker: A high performance data structure for dynamic graph processing. *IPDPS*, pages 1030–1041, 2019.

30 Nikolaos D. Kallimanis and Eleni Kanellou. Wait-Free Concurrent Graph Objects with Dynamic Traversals. In *OPODIS*, pages 1–27, 2015.

31 Miray Kas, Kathleen M. Carley, and L. Richard Carley. Incremental Closeness Centrality for Dynamically Changing Social Networks. In *ASONAM 2013*, pages 1250–1258, 2013.

32 Alex Kogan and Erez Petrank. Wait-Free Queues With Multiple Enqueuers and Dequeuers. In *PPOPP*, pages 223–234, 2011.

33 Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic Parallelism Requires Abstractions. In *PLDI*, pages 211–222, 2007.

34 P. Kumar and H. Huang. Graphone: A data store for real-time analytics on evolving graphs. In *FAST*, 2019.

35 Edya Ladan-Mozes and Nir Shavit. An Optimistic Approach to Lock-free FIFO Queues. *Distributed Computing*, 20(5):323–341, 2008.

36 Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. `http://snap.stanford.edu/data`, June 2014.

37 Yujie Liu, Kunlong Zhang, and Michael Spear. Dynamic-sized Nonblocking Hash Tables. In *PODC*, pages 242–251, 2014.

38 Maged M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *SPAA*, pages 73–82, 2002.

39 Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *PPoPP*, pages 317–328, 2014.

40 K A Obukhova, Iryna Zhuravska, and Volodymyr Burenko. Diagnostics of power consumption of a mobile device multi-core processor with detail of each core utilization. *TCSET*, pages 368–372, 2020.

41 P. Pan, C. Li, and M. Guo. Congraplus: Towards efficient processing of concurrent graph queries on numa machines. *IEEE TPDS*, 30(9):1990–2002, 2019.

42 Peitian Pan and Chao Li. Congra: Towards efficient processing of concurrent graph queries on shared-memory machines. In *ICCD*, pages 217–224, 2017.

43 Arunmoezhi Ramachandran and Neeraj Mittal. A Fast Lock-Free Internal Binary Search Tree. In *ICDCN*, pages 37:1–37:10, 2015.

44 Alberto G. Rossi, D. Blake, A. Timmermann, I. Tonks, and R. Wermers. Network centrality and delegated investment performance. *Journal of Financial Economics*, 128:183–206, 2018.

45 Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP*, pages 135–146, 2013.

46 Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+. In *DCC*, pages 403–412, 2015.

47 Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

48 Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-Free Linked-Lists. In *OPODIS*, pages 330–344, 2012.

49 John D. Valois. Lock-Free Linked Lists Using Compare-and-Swap. In *PODC*, pages 214–222, 1995.

50 Kunlong Zhang, Yujiao Zhao, Yajun Yang, Yujie Liu, and Michael F. Spear. Practical Non-blocking Unordered Lists. In *DISC*, pages 239–253, 2013.

51 J. Zhao, Y. Zhang, X. Liao, L. He, B. He, H. Jin, H. Liu, and Y. Chen. Graphm: An efficient storage system for high throughput of concurrent graph processing. In *SC*, 2019.

Appendix

A The Non-blocking Graph Algorithm

```
Operation PutV(v)
  return HashAdd(v);
--------------------------------------------------
Operation RemV(v)
  return HashRem(v);
--------------------------------------------------
Operation GetV(v)
  st, v ← HashCon(v);
  if (st = true) then
    return true, v ;
  else
    return false, NULL ;
--------------------------------------------------
Operation GetE(v1, v2)
  u, v, st ← ConVPlus(v1, v2);
  if (st = false) then
    return false, ∞ ;
  st, e ← BSTCon(v2, v.enxt);
  if (st = FOUND  ¬ HashCon(v1)  ¬ HashCon(v2)
  (e.ptv = u)) then
    z ← e.w;
    return true,z ;
  else return false, ∞ ;
--------------------------------------------------
Method ConVPlus(v1, v2)
  st1,u ← HashCon(v1); //Modified GetV, returns status along with ref
  st2,v ← HashCon(v2);
  if (st1 = true  st2 = true) then
    return u, v, true ;
  else return u, u, false ;
--------------------------------------------------
Operation PutE(v1, v2|w)
  u, v, st ← ConVPlus(v1, v2);
  if (st = false) then return false, ∞ ;
  while (true) do
    if ( isMrkd(u)  isMrkd(v)) then
      return false, ∞ ;
    st ← Find(v2, pe, peOp, ce , ceOp, u.enxt);
    if (GetFlag(pe) = MARKED) then
      continue;
    if (st = FOUND) then
      if (ce.w = w) then return false,w ;
      else
        z ← ce.w;
        CAS(ce.w, z, w);
        u.ecnt.FetchAndAdd (1);
        return true,z ;
    else
      ne ← CEnode (v2, w);
      ne.ptv ← v;
      Boolean ifLeft ← (st = NOTFOUND_L);
      ENode old ← ifLeft ? ce.left : ce.right;
      casOp ← new ChildCASOp(ifLeft, old, ne);
      if (CAS(ce.op,ceOp,Flag(casOp,CHILDCAS))) then
        u.ecnt.FetchAndAdd (1);
        HelpChildCAS(casOp, ce);
        return true, ∞ ;
--------------------------------------------------
Operation RemE(v1, v2)
  u, v, st ← ConVPlus(v1, v2);
  if (st = false) then
    return false, ∞ ;
  while (true) do
    if (isMrkd(u)  isMrkd(v)) then
      return false, ∞ ;
    st ← Find(v2, pe, peOp, ce , ceOp, u.enxt);
    if (st = FOUND) then
      return false, ∞ ;
    if (IsNull(curr.right)  IsNull( ce.left))) then
      if (CAS(ce.op, ceOp, Flag(ceOp, MARKED))) then
        u.ecnt.FetchAndAdd (1);
        HelpMarked(pe, peOp, ce);
        z ← ce.w;
        break;
    else
      st ← Find(v2, pe, peOp, ce , ceOp, u.enxt);
      if ((st = ABORT)  (ce.op = ceOp) then
        continue;
      relocOp ← new RelocateOp(ce, ceOp, v2, replace.e);
      if (CAS(replace.op, replaceOp, Flag (relocOp, RELOCATE)) then
        u.ecnt.FetchAndAdd (1);
        if (HelpRelocate(relocOp,pe,peOp,replace)) then
          z ← ce.w;
          break;
  return true,z ;
```

Figure 11 Pseudocodes of PutV, RemV, GetV, PutE, RemE, GetE and ConVPlus

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*. The 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`.

Based on the boolean parameter taken by Resize method, it decides the hash table either to grow or shrink. The initBkt method ensures all `VNodes` are physically present in the buckets. It relocates the `HNodes` to the hash table which are in the predecessor's list.

The i^{th} bucket of a given `HNode` h is initialized by initBkt method, by splitting or merging the buckets of h s predecessor `HNode` s, if s exists. The sizes of h and s are compared and then this method decides whether h is shrinking or expanding with reference to s. Then it freezes the respective bucket(s) of s before copying the `VNodes`. If h halves the size of s, then i^{th} and $(i + h.\texttt{size})^{th}$ buckets of s are merged together to form the i^{th} bucket of h. Otherwise, h doubles the size of s, then approximately half of the `VNodes` in the $(i \textbf{ mod } h.\texttt{size})^{th}$ bucket of s relocate to the i^{th} bucket of h. To avoid any races with the other helping threads while splitting or merging of buckets a `CAS` is used (Line 137).

The `ENode` structure is similar to that of a lock-free BST [28] with an additional edge weight `w` and a pointer field `ptv` which points to the corresponding `VNode`. This helps direct access to its `VNode` while doing a BFS traversal and also helps in deletion of the incoming edges. The operation `op` field stores if any changes are being made, which affects the `ENode`. To avoid the overhead of another field in the node structure, we use bit-manipulation: last significant bits of a pointer p, which are unused because of the memory-alignment of the shared-memory system, are used to store information about the state of the pointer shared by concurrent threads and executing an operation that would potentially update the pointee of the pointer. More specifically, in case of an x86-64 bit architecture, memory has a 64-bit boundary and the last three least significant bits are unused. So, we use the last two significant bits, which are enough for our purpose, of the pointer to store auxiliary data. We define four different methods to change an `ENode` pointer: IsNull(p) returns `true` if the last two significant bits of p make 00, which indicates no ongoing operation, otherwise, it returns `false`; isMrkd(p) returns `true` if the last two significant bits of p are set to 01, else it returns `false`, which indicates the node is no longer in the tree and it should be *physically* deleted; IsChildCAS (p) returns `true` if last two bits of p are set to 10, which indicates one of the child node is being modified, else it returns `false`; IsRelocate(p) returns `true` if the last two bits of p make 11, which indicates that the `ENode` is undergoing a node relocation operation.

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 `NULL`; if it is, initBkt method is invoked to initialize the bucket (Line 144). At the end, the return value is stored in the `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 HashAdd operation that checks for the size of the hash table with some *threshold* value, if it exceeds the threshold the size of the table is doubled. Similarly, a HashRem checks the threshold value, if it falls below threshold, it shrinks the hash table size to halves.

A GetV(`v`) operation, at Lines 5 to 10, invokes HashCon(`v`) to search a `VNode v` in the hash table. It starts by searching the given key `v` in the bucket b. If b is `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 HashCon is concurrently running with resizing of s. So, a double check at Line 123 is required to test whether s is `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 `NULL`, and then we perform the search in that bucket. If b is not `NULL`, then we simply return the presence of the corresponding `VNode` in the bucket b. Note that, at any point in time there are at most two `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 – PutE, RemE, and GetE – are presented in Figure 11. Before describing these operations, we detail the implementation of Find method, which is used by them. It is shown in Figure 13. The method Find, at Lines 199 to 227, tries to locate the position of the key by traversing down the edge-list of a `VNode`. It returns the position in *pe* and *ce*, and their corresponding `op` values in *peOp* and *ceOp* respectively. The result of the method Find can be one of the four values: (1) `FOUND`: if the key is present in the tree, (2) `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) `NOTFOUND_R`: similar to `NOTFOUND_L` but for the right child of *ce*, and (4) `ABORT`: if the search in a subtree is unable to return a usable result.

A PutE($\mathtt{v}_1, \mathtt{v}_2|w$) operation, at Lines 26 to 51, begins by validating the presence of $\mathtt{v}_1$ and $\mathtt{v}_2$ in the vertex-list. If the validations fails, it returns `false`, ∞ (Line 28). Once the validation succeeds, PutE operation invokes Find method in the edge-list of the vertex with key $\mathtt{v}_1$ to locate the position of the key $\mathtt{v}_2$. The position is returned in the variables *pe* and *ce*, and their corresponding `op` values are stored in the *peOp* and *ceOp* respectively. On that, PutE checks whether an `ENode` with the key $\mathtt{v}_2$ 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 PutE returns `false`, ∞ (Line 36). However, if it is present with a different edge weight, say z, PutE updates *ce*'s old weight z to the new weight w and returns `true`, z (Line 38). We update the edge-weight using a `CAS` to ensure the correct return in case there were multiple concurrent PutE operations trying to update the same edge. Notice that, here we are ***not*** freezing the `ENode` in anyway while updating its weight. The linearizability is still ensured, which we discuss in the next section.

If the key $\mathtt{v}_2$ is not present in the tree, a new `ENode` and a `ChildCASOp` object are created. Then using `CAS` the object is inserted logically into *ce* s `op` field (Line 48). If the `CAS` succeeds, it implies that *ce* s `op` field hadn't been modified since the first read. Which in turn indicates that all other fields of *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($\mathtt{v}_1, \mathtt{v}_2$) operation, at Lines 52 to 78, similarly begins by validating the presence of $\mathtt{v}_1$ and $\mathtt{v}_2$ 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 $\mathtt{v}_1$ to locate the position of the key $\mathtt{v}_2$. 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 with either a 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 v_2 with the node returned. This operation added to *replace* s `op` field safeguards it against a concurrent deletion while the ReмE 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 v_2s `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 ReмE returns `true`,*ce*.`w`

Similar to PutE and ReмE, a GetE(v_1, v_2) operation, at Lines 11 to 19, begins by validating the presence of v_1 and v_2 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 v_1 to locate the position of the key v_2. If it finds v_2, 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
    if (op.optype = Add) then
      resp ← op.key ∉ o.set;
      set ← o.set ∪ {op.key};
    else
      if (op.optype = Remove) then
        resp ← op.key ∈ o.set;
        set ← o.set \ {op.key};
    n ← new FSetNode(set, true);
    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); ]
  if (heuristic-policy) then
    Resize (true);
  return resp;
----------------------------------------------
Operation HashRem(key)
  resp ← Apply(Remove, key);
  if (heuristic-policy) then
    Resize (false);
  return resp;
```

Figure 12 Strucure of `FSet`, `FSetOp` and `HNode`. Pseudocodes of Invoke, Freeze, Add, and Remove methods based on dynamic sized non-blocking hash table[37].

```
118: Operation HashCon(key)
119:   t ← Head;
120:   b ← t.buckets[key mod t.size ];
121:   if (b = NULL) then
122:     s ← t.pred;
123:     if (s ≠ NULL ) then
124:       b ← s.buckets(key mod s.size);
125:     else
126:       b ← t.buckets(key mod t.size);
127:   return HasMember (b,key);
128: Method Resize(grow)
129:   t ← Head;
130:   if (t.size > 1 ∨ grow = true) then
131:     for (i ← 0 to t.size-1) do
132:       initBkt(t,i);
133:     t.pred ← NULL;
134:     size ← grow ? t.size ⋆ 2 : t.size/2;
135:     buckets ← new FSet[size ];
136:     t′ ← new HNode(buckets, size, t);
137:     CAS(Head, t, t′);
138: Method Apply(optype, key)
139:   op ← new FSetOp(optype, key, false, −);
140:   while (true) do
141:     t ← Head;
142:     b ← t.buckets[key mod t.size ];
143:     if (b = NULL) then
144:       b ← initBkt(t,key.mod t.size);
145:     if (Invoke(b, op)) then
146:       return GetResponse (op);
147: Method initBkt(t, key)
148:   b ← t.buckets[key];
149:   s ← t.pred;
150:   if (b = NULL ∧ s ≠ NULL) then
151:     if (t.size = s.size) then
152:       m ← s.buckets[i mod s.size];
153:       set ← Freeze(m) ∩ { x | x mod t.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(t.buckets[i], NULL, b′);
160:   return t.buckets[i];

struct Node{int key;Operation op; Node left,right;}
struct RelocateOp {int state,removeKey,replaceKey;
           Node dest; Operation destOp; }
struct ChildCASOp {boolean ifLeft;
          Node expected, update; }

161: Operation Add(key)
162:   Node pred, curr, newNode;
163:   Operation predOp, currOp, casOp;
164:   int result;
165:   while (true) do
166:     result ← Find(key, pred, predOp, curr, currOp, root);
167:     if (result = FOUND) then return false;
168:     newNode ← new Node(key);
169:     Boolean ifLeft ← (result = NOTFOUND_L);
170:     Node old ← ifLeft ? curr.left : curr.right;
171:     casOp ← new ChildCASOp(ifLeft, old, newNode);
172:     if (CAS(curr.op, currOp, Flag(casOp,CHILDCAS))) then
173:       HelpChildCAS(casOp, curr);
174:       return true;
175: Operation Remove(key)
176:   Nodepred, curr, replace;
177:   Operation predOp, currOp, replaceOp, relocOp;
178:   while (true) do
179:     if (Find(key, pred, predOp, curr, currOp, root) = FOUND) then
180:       return false;
181:     if (IsNull(curr.right   IsNull( curr.left))) then
182:       if (CAS(curr.op, currOp, Flag(currOp, MARKED))) then
183:         HelpMarked(pred, predOp, curr);
184:         return true;
185:     else
186:       if ((Find(key, pred, predOp, replace, replaceOp, curr) = ABORT)   (curr.op = currOp) then
187:         continue;
188:       relocOp ← new RelocateOp(curr, currOp, key, replace.key);
189:       if (CAS(replace.op, replaceOp, Flag(relocOp, RELOCATE)) then
190:         if (HelpRelocate(relocOp, pred, predOp, replace)) then
191:           return true;
192: Operation Contains(key)
193:   Node pred, curr;
194:   Operation predOp, currOp;
195:   if (Find(key, pred, predOp, curr, currOp, root) = FOUND) then
196:     return true;
197:   else
198:     return false;
199: Method Find(key, pred, predOp, curr, currOp, root)
200:   int result, currKey ; Node next, lastRight;
201:   Operation lastRightOp; result ← NOTFOUND_R;
202:   curr ← root; currOp ← curr.op;
203:   if (GetFlag(currOp) = NULL) then
204:     if (root = root) then
205:       HelpChildCAS(UnFlag(currOp), curr);
206:       goto Line 201;
207:     else return ABORT;
208:   next ← curr.right; lastRight ← curr;
209:   lastRightOp ← currOp;
210:   while (¬ IsNull(next)) do
211:     pred ← curr; predOp ← currOp;
212:     curr ← next; currOp ← curr.op;
213:     if (GetFlag(currOp) = NULL) then
214:       HelpList(pred, predOp, curr, currOp);
215:       goto Line 201;
216:     currKey ← curr.key;
217:     if (key < currKey) then
218:       result ← NOTFOUND_L; next ← curr.left;
219:     else
220:       if (key > currKey) then
221:         result ← NOTFOUND_R; next ← curr.right;
222:         lastRight ← curr; lastRightOp ← currOp;
223:       else
224:         result ← FOUND; break;
225:   if ((result = FOUND) ⋀ (lastRightOp = lastRight.op) then goto Line 201;
226:   if (curr.op = currOp) then goto Line 201;
227:   return result;
```

Figure 13 Strucure 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].