Brief Announcement: Non-Blocking Dynamic Unbounded Graphs with Worst-Case Amortized Bounds

Bapi Chatterjee
Institute of Science and Technology, Klosterneuburg, Austria

Sathya Peri
Indian Institute of Technology, Hyderabad, India

Muktikanta Sa
Télécom SudParis – Institut Polytechnique de Paris, France

Abstract

This paper reports a new concurrent graph data structure that supports updates of both edges and vertices and queries: Breadth-first search, Single-source shortest-path, and Betweenness centrality. The operations are provably linearizable and non-blocking.

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.DISC.2021.52

Related Version *Full Version*: https://arxiv.org/abs/2003.01697

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).

1 Introduction

Dynamic graph data structures with concurrent query operations and updates can readily boost important real-world applications such as social networks [6], semantic-web [5], biological networks [10], blockchains [3], and many others. The existing libraries of graph queries, which support dynamic updates, for example, Stinger [12], GraphOne [16], GraphTinker [15], Kineograph [9], GraphTau [14], Kickstarter [18], Aspen [11], etc. face limitations such as blocking concurrency, no native support for vertex updates, and high memory-footprint.

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 [13], and (d) a light memory footprint. We call it PANIGRAHAM [a]: **Pra**ctical **N**on-blocki**ng** **Gra**ph Algorit**hm**s. 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. Addition/removal of a vertex translates to the same operation in the lock-free hash-table,

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

```
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); //1st Collect
  while (true) do //Repeat the tree collection
    nt ← TreeCollect (v, tid); //2nd 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 1 Framework interface operation for graph queries.

whereas, addition/removal of an edge translates to the same operation in a lock-free BST. The operations – BFS, SSSP, BC – are implemented by specialized partial snapshots. In a dynamic concurrent 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.

2 PANIGRAHAM

Algorithm Overview. We implement an ADT $\mathscr{A} = \mathscr{S} \cup \mathscr{Q}$, wherein the set operations $\mathscr{S} := \{\textsc{PutV}, \textsc{RemV}, \textsc{GetV}, \textsc{PutE}, \textsc{RemE}, \textsc{GetE}\}$ use lock-free hash-table and BST and the queries $\mathscr{Q} := \{\text{BFS}, \text{SSSP}, \text{BC}\}$ use partial snapshot. To de-clutter the presentation, we encapsulate the three queries in a unified framework with an interface operation Op– presented in pseudo-code in Figure 1. Op is specialized to the requirements of the three queries. We have explained the pseudo-code using line-comments in Figure 1. For detail of the ADT operations please see the full version [8], wherein we also present their proofs of linearizability and non-blocking progress.

Experimental Results and Discussion. We experimentally evaluate our non-blocking graph against two well-known existing batch analytics methods: (a) **Stinger** [12], and (b) **Ligra** [17]. To analyze the trade-off between consistency and performance, in addition to the presented linearizable algorithm PANIGRAHAM (**PG-Cn**), we include its inconsistent variant (**PG-Icn**). The results are based on a standard dataset **R-MAT** graphs [7]. Each micro-benchmark displays the latency of an end-to-end run of 10^4 operations on a loaded graph, assigned in a uniform random order to the threads. We used a range of workload distributions. A sample label, say, 2/49/49 on the top of a column of performance plots refers to a distribution $\{\textsc{Op} : 2\%, \{\textsc{PutV} : 24.5\%, \textsc{RemV} : 24.5\%\}, \{\textsc{PutE} : 24.5\%, \textsc{RemE} : 24.5\%\}\}$. We used a multi-core system with 28 cores (56 logical threads). The results shown in Figure 2 demonstrate the scalability of the proposed methods. We observe that the presented algorithm outperforms **Stinger** and **Ligra** in several cases by orders of magnitude. In the full version [8] we present additional results on real-life graphs as well as experimental comparison of the memory-footprints of the methods that further highlights the efficacy of our method.

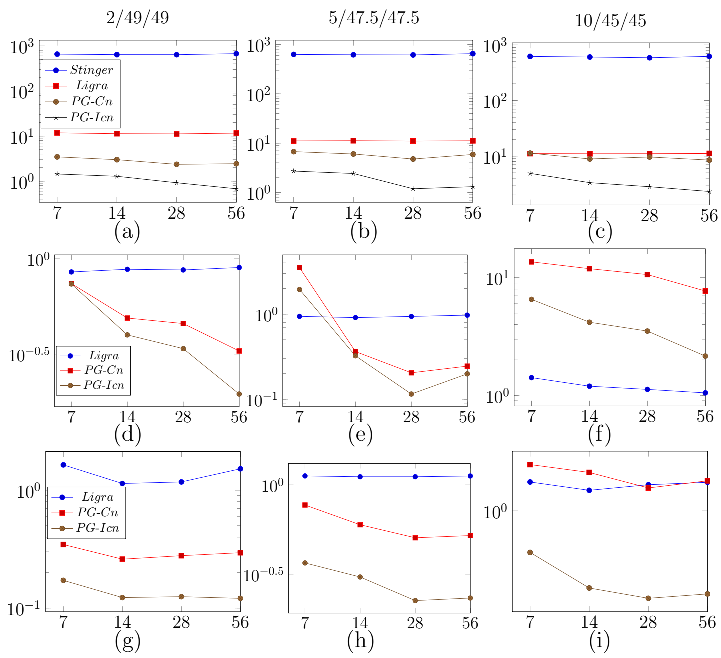

Figure 2 Latency of the executions containing $\textsc{Op}$: (1) BFS ((a), (b), and (c)) on a graph of size $|V|= 131K$ and $|E|= 2.44M$, (2) SSSP ((d), (e), and (f)) on a graph of size $|V|= 8K$ and $|E|= 80K$, and (3) BC ((g), (h), and (i)) on a graph of size $|V|= 16K$ and $|E|= 160K$. X-axis and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

Complexity Analysis. Given a graph $G = (V, E)$, denote $|V|= n$, $|E|= m$, $\max_{\mathtt{v}\in V}(\delta_{\mathtt{v}}) = \delta$, where $\delta_{\mathtt{v}}$ is the degree of vertex $\mathtt{v}$. Define the (static) *state* of a graph G as a tuple $S_G = (n, m, \delta)$. Let X be a concurrent execution given as a set of operations. Thus, for an $o \in X$, $type(o) \in \mathscr{A}$, where $type(o)$ denotes the type of o and $\mathscr{A}$ is the ADT. Let I_o and C_o be the *interval contention* [2] and *point contention* [4], respectively, for an $o \in X$. 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. 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}. W_{o,S_G} for each operation type is given in Table 1 of [8]. The states of G, being tuples, are ordered by dictionary order. In a dynamic setting, W_{o,S_G} is upper-bounded by the worst-case cost of o as performed in a static setting over the *worst-case state*, during the lifetime of o, of G. It can be shown that the worst-case state of G that o can encounter is $\overline{S_{G,o}} = (O(n + \widetilde{I_o}), O(m + \widetilde{I_o}), O(\delta + \widetilde{I_o}))$.

▶ **Theorem 1.** *Denote* $\mathscr{Q}$=$\{BFS, SSSP, BC\}$, $\mathscr{M}_V$=$\{\textsc{PutV}, \textsc{RemV}\}$, $\mathscr{M}_E$=$\{\textsc{PutE}, \textsc{RemE}\}$, *and* $\mathscr{M} = \mathscr{M}_V \cup \mathscr{M}_E$. *Let* δ_e *be the degree of vertex whose edge modification happens. Denote* $X_{\mathscr{U}} = \{o \in X \mid type(o) \in \mathscr{U}, \mathscr{U} \subseteq \mathscr{A}\}$, *where* $\mathscr{A}$ *is the ADT as defined before. Let* $I_{o,\mathscr{U}}$ *and* $C_{o,\mathscr{U}}$ *denote the interval and point contentions, respectively, of* o *pertaining to the operation calls* $o \in \{X_{\mathscr{U}} \cup \{o\}\}$. *Accordingly,* $\widetilde{I_{o,\mathscr{U}}} = I_{o,\mathscr{U}} - 1$. *Considering the queries* $q \in \mathscr{Q}$ *performed by PG-Cn, the worst-case amortized cost per operation* A_X *for an execution* X *s.t.* $type(o) \in \mathscr{M} \cup \{q\}\ \forall o \in X$, *and* $q \in \mathscr{Q}$ *is* $A_X = A_X(\mathscr{M}) + \frac{C_{o,\mathscr{M}}}{|X|} \sum_{o \in X_{\mathscr{Q}}} \left(W_{o,\overline{S_{G,o}}} + \widetilde{I_{o,\mathscr{M}}} \right)$, *where* $A_X(\mathscr{M}) = \frac{C_{o,\mathscr{M}_V}}{|X|} \sum_{o \in X_{\mathscr{M}_V}} W_{o,\overline{S_{G,o}}} + \frac{1}{|X|} \sum_{o \in X_{\mathscr{M}_X}} W_{o,\overline{S_{G,o}}} + \frac{C_{o,\mathscr{M}_E}}{|X|} \sum_{o \in X_{\mathscr{M}_E}} O(\delta_e)$.

The proof of Theorem 1 is available in the full version [8].

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. Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. An Efficient Framework for Optimistic Concurrent Execution of Smart Contracts. In *(PDP) 2019*, pages 83–92, 2019.
4. Hagit Attiya and Arie Fouren. Algorithms Adapting to Point Contention. *J. ACM*, 50(4):444–468, 2003.
5. Sumit Bhatia, Bapi Chatterjee, Deepak Nathani, and Manohar Kaul. A Persistent Homology Perspective to the Link Prediction Problem. In *Complex Networks*, page (to appear), 2019.
6. Salvatore Catanese, Pasquale De Meo, Emilio Ferrara, Giacomo Fiumara, and Alessandro Provetti. Crawling Facebook for Social Network Analysis Purposes. In *(WIMS)*, page 52, 2011.
7. Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *SDM*, pages 442–446, 2004.
8. Bapi Chatterjee, Sathya Peri, and Muktikanta Sa. Dynamic Graph Operations: A Consistent Non-blocking Approach. *CoRR*, abs/2003.01697, 2020.
9. 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.
10. Antonio del Sol, Hirotomo Fujihashi, and Paul O'Meara. Topology of Small-world Networks of Protein–Protein Complex Structures. *Bioinformatics*, 21(8):1311–1315, 2005.
11. Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *40th PLDI*, pages 918–934, 2019.
12. D. Ediger, R. McColl, J. Riedy, and D. A. Bader. STINGER: High Performance Data Structure for Streaming Graphs. In *HPEC*, 2012.
13. Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
14. Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-evolving Graph Processing at Scale. In *GRADES*, page 5, 2016.
15. Wole Jaiyeoba and K. Skadron. Graphtinker: A high performance data structure for dynamic graph processing. *IPDPS*, pages 1030–1041, 2019.
16. P. Kumar and H. Huang. Graphone: A data store for real-time analytics on evolving graphs. In *FAST*, 2019.
17. Julian Shun and Guy E. Blelloch. Ligra: a Lightweight Graph Processing Framework for Shared Memory. In *PPoPP*, pages 135–146, 2013.
18. Keval Vora, R. Gupta, and Guoqing Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. *ASPLOS*, 2017.