Building Efficient Concurrent Graph Object through Composition of List-based Set*

Sathya Peri, Muktikanta Sa, Nandini Singhal

Department of Computer Science & Engineering,
Indian Institute of Technology Hyderabad, India
{sathya_p, cs15resch11012, cs15mtech01004}@iith.ac.in

Abstract

In this paper, we propose a generic concurrent directed graph (for shared memory architecture) that is concurrently being updated by threads adding/deleting vertices and edges. The graph is constructed by the composition of the well known concurrent list-based set data-structure from the literature. Our construction is generic, in the sense that it can be used to obtain various progress guarantees, depending on the granularity of the underlying concurrent set implementation - either blocking or non-blocking. We prove that the proposed construction is linearizable by identifying its linearization points. Finally, we compare the performance of all the variants of the concurrent graph data-structure along with its sequential implementation. We observe that our concurrent graph data-structure mimics the performance of the concurrent list based set.

Keywords: concurrent data structure; lazy set; directed graph; non-blocking; locks; lock-freedom;

1 Introduction

A graph represents pairwise relationships between objects along with their properties. Due to their usefulness, graphs are being used in various kinds of networks such as social, semantic, genomics, etc. Generally, these graphs are very *large* and *dynamic* in nature. Dynamic graphs are the one's which are subject to a sequence of changes like insertion, deletion of vertices and/or edges [1]. Online social networks (facebook, linkedin, google+, twitter, quora, etc.), are dynamic in nature with the users and the relationships among them changing over time. In this paper, we develop a generic concurrent directed graph data-structure, which allows threads to concurrently add/delete or perform contains on vertices/edges while ensuring linearizability [6]. The graph is constructed by the composition of the well known concurrent list-based set implementation from the literature. To the the best of our knowledge, this is the first work to propose an adjacency list representation of a graphs by composing list-based sets. The other known work on concurrent graphs by Kallimanis & Kanellou [7] works on adjacency matrix representation and assume an upper-bound on the number of vertices while we make no such assumption. Our construction is generic, in the sense that it can be used to obtain different progress guarantees, depending on the granularity of the underlying concurrent set implementation - either blocking or non-blocking. The blocking list implementation is taken from Heller et. al [3] which is also popularly known as the lazy implementation whereas the non-blocking variant is by Harris [2]. Our design is not a straight forward extension of the concurrent list-based set implementation but has several non-trivial additions. This can be seen from the Linearization Points of edge update methods (described later) which lie outside their method code & depend on other concurrently executing graph methods. We believe the design of the concurrent graph data-structure is such that it can help identify other useful properties on graph such as cycle detection, shortest path, reachability, minimum spanning tree, strongly connected components, etc.

Roadmap - In Section 2, we describe the system model & preliminaries detailing definitions of the terminology used in the paper. In Section 3, we describe the construction of our generic concurrent graph data structure. Later on, in Section 4, we describe the working of each of the methods and their linearization points. In Section 5, we describe the evaluation of the throughput of the several variants - sequential, using coarse-grained locking, hand-over-hand locking, lazy list-based set, lock-free implementation. Finally we conclude in Section 6.

2 System Model & Preliminaries

In this paper, we assume that our system consists of finite set of p processors, accessed by a finite set of n threads that run in a completely asynchronous manner and communicate using shared objects. The threads communicate with each other by invoking methods on the shared objects and getting corresponding responses. Consequently, we make

*This work is currently in progress and the technical report of this paper can be found on Arxiv [9].

no assumption about the relative speeds of the threads. We also assume that none of these processors and threads fail. Our algorithm is designed for execution on a shared-memory multi-processor system which supports atomic *read*, *write* and *compare-and-swap(CAS)* operations.
Safety: To prove a concurrent data-structure to be correct, *linearizability* proposed by Herlihy & Wing [6] is the standard correctness criterion in the concurrent world. They consider a history generated by a data-structure which is collection of method invocation and response events. Each invocation of a method call has a subsequent response. A history is linearizable if it is possible to assign an atomic event as a *linearization point* (LP) inside the execution interval of each method such that the result of each of these methods is the same as it would be in a sequential history in which the methods are ordered by their LPs [4].
Progress: The *progress* properties specifies when a thread invoking methods on shared objects completes in presence of other concurrent threads. Some progress conditions used in this paper are mentioned here which are based on the definitions in Herlihy & Shavit [5]. The progress condition of a method in concurrent object is defined as: (1) Blocking: In this, an unexpected delay by any thread (say, one holding a lock) can prevent other threads from making progress. (2) Deadlock-Free: This is a **blocking** condition which ensures that **some** thread (among other threads in the system) waiting to get a response to a method invocation will eventually receive it. (3) Lock-Free: This is a **non-blocking** condition which ensures that **some** thread waiting to get a response to a method (among multiple other threads), eventually receives it.

3 Construction of Concurrent Generic Graph Data-Structure

In this section, we describe the construction of our generic concurrent graph data-structure. We represent our data-structure using the adjacency list representation of the graph. It is implemented as a list of lists, i.e., composition of lists. Each vertex in the list holds a list of vertices to which it has outgoing edges. This is depicted pictorially in Figure 1.

The problem addressed in this paper is described as follows: A concurrent directed graph $G = (V, E)$, where $G(V)$ is a set of vertices and $G(E)$ is a collection of directed edges. Each edge connects an ordered pair of vertices belonging to $G(V)$. And this G is dynamically modified by a fixed set of concurrent running threads. In this setting, threads may perform insertion / deletion of vertices or edges to the graph. We assume that all the vertices have unique identification key (captured by *val* field) as shown below. We also assume that a vertex once deleted is not inserted again with the same key.

In Table 3a, we describe the structure of the Node class. In the concurrent graph data-structure, a *GNode* could be used to represent either the vertex node or the edge node. Thus we add an additional next field to indicate the list we wish to traverse. In other words, if the *GNode* is a vertex node, we will use its *vnext* to point to the next vertex node and *enext* to direct to its edge head sentinel. Similarly, if the GNode is a edge node, we will only use its enext to go to the next edge node in the edge list. The remaining fields of the GNode class are the same as the fields of concurrent linked list based on the lazy or the lock-free implementation. The construction shown here is keeping in mind the lazy & lock free list-based set implementation which makes use of the additional marked field. However, it can be modified appropriately to obtain other variants.

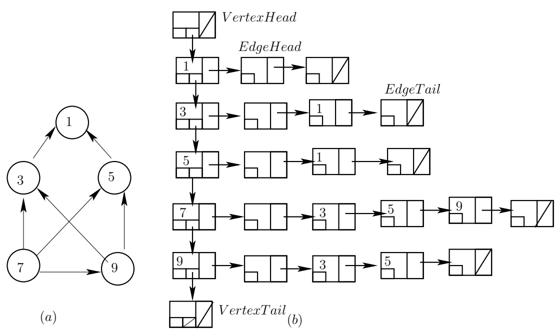

Figure 1: (a) A directed Graph (b) The concurrent graph data structure representation for (a).

Table 3a: Structure of GNode.

```
class GNode{
    int val;
    bool marked;
    GNode vnext;
    GNode enext;
    GNode(int key){
        val = key;
        marked = false;
        vnext = enext = null;
    }
};
```

The Table 3b depicts the structure of the list-based set class. It creates two sentinel nodes for each list, called head and tail. A list can be uniquely identified using its head node. An object of the List class exports the add(x), remove(x) & contains(x) methods. Each of these methods can be implemented using various well known synchronization techniques of the concurrent set implementation - fine-grained, optimistic, lazy or lock-free. The only difference is that we also pass the head of the list on which each of these operation should be performed. With our modified Node class, each GNode could possibly direct to either a vertex node or a edge node; we must identify correctly on which list we wish to perform the operation. To do this, we pass the head of the list we wish to operate on. If the argument passed is the head of the vertex list, we use the vnext to direct to the next GNode, otherwise enext.

Finally for our purposes, as illustrated in table 3c, the concurrent graph data-structure exports the specified methods. These are explained in the next subsection.

Table 3b: Structure of List.

```
class List{
    GNode Head, Tail;
    List(){
        //Initialize sentinels
    }
    bool Add(GNode Head, int key);
    bool Remove(GNode Head, int key);
    bool Contains(GNode Head, int key);
    /* performing operation on the list represented
    by Head */
};
```

Table 3c: Structure of Graph.

```
class Graph extends List{
  bool AddVertex(int key);
  bool RemoveVertex(int key);
  bool ContainsVertex(int key);
  bool AddEdge(int key1, int key2);
  bool RemoveEdge(int key1, int key2);
  bool ContainsEdge(int key1, int key2);
  };

Graph G;
```

3.1 Methods Exported & Sequential Specification

In this sub-section, we describe the methods exported by the concurrent directed graph data-structure along with their sequential specification. The specification as the name suggests shows the behavior of the graph when all the methods are invoked sequentially.

1. The $AddVertex(u)$ method adds a vertex u to the graph, returning $true$. This follows directly from our assumption that all the vertices are assigned distinct keys. Once added, the method will never invoke addition on this key again.
2. The $RemoveVertex(u)$ method deletes vertex u from the graph, if it is present in the graph and returns true. By deleting this vertex u, this method ensures that all the incoming and outgoing edges of u are deleted as well. If the vertex is not present in the graph, it returns $false$.
3. The $ContainsVertex(u)$ returns $true$, if the graph contains the vertex u; otherwise returns $false$.
4. The $AddEdge(u, v)$ method adds a directed edge (u, v) to the graph if the edge (u, v) is not already present in the graph and returns $true$. If the edge is already in the graph it simply returns $true$. But if either the vertices u or v are not present, it returns $false$.
5. The $RemoveEdge(u, v)$ method deletes the directed edge (u, v) from the graph structure if it is present and returns $true$. If the edge (u, v) is not present in the graph but the vertices u & v are in the graph it still returns $true$. But, if either of the vertices u or v are not present in the graph, it returns $false$.
6. The $ContainsEdge(u, v)$ returns $true$, if the graph contains the edge (u, v); otherwise returns $false$.

4 Working of Concurrent Graph Methods

In this section, we describe the working of the various methods on the generic concurrent graph data structure. As explained earlier, we represent the graph using adjacency list representation, which is a list of list-based sets as illustrated in the Figure 1. A set is implemented as a linked list of nodes. In each list, nodes are in sorted in key order, providing an efficient way to detect when an item is absent. The next field is a reference to the next node in the list. Each list has two kinds of nodes. In addition to regular nodes that hold items in the set, we use two sentinel nodes, called head and tail and their keys are the minimum and maximum integer values. All the fields in the structure are declared atomic. This ensures that operations on these variables happen atomically. In the context of a particular application, the node structure can be easily modified to carry useful data (like weights etc). Algorithm 1-7 describe all the pseudo code used in our generic concurrent graph data structure.

Notations used in Pseudo-Code:

We use ↓, ↑ to denote the input and output arguments to each method respectively. The shared memory is accessed only by invoking explicit *read()*, *write()* and *CAS* methods. The $flag$ is a local variable which returns the status of each operation.

4.1 *Update Vertex Methods* - AddVertex & RemoveVertex

The $AddVertex(u)$ method is given in the Algorithm 1. It simply invokes the Add method of the list-based set by passing the head of the vertex list and the key to be added.

Algorithm 1 AddVertex Method: Successfully adds $GNode(key)$ to the vertex list, if it is not present earlier, else it ignores.

```
1: procedure AddVertex (key ↓, flag ↑)
2:   G.Add(G.Head ↓, key ↓, status ↑);
3:   flag ← status
4: end procedure
```

Algorithm 2 RemoveVertex Method: removes the $GNode(key)$ from the vertex list if it is already present. Otherwise it returns $false$

```
5: procedure RemoveVertex (key ↓, flag ↑)
6:   G.Remove(G.Head ↓, key ↓, status ↑);
7:   flag ← status;
8:   RemoveIncomingEdges(key↓);        /*optional method*/
9: end procedure
```

Algorithm 3 RemoveIncomingEdge Method: This method helps remove all the incoming edges of a deleted vertex $GNode(key)$ from the graph.

```
10: procedure RemoveIncomingEdges(key ↓)
11:   node ← G.Head
12:   while (node != G.Tail)) do
13:     G.Remove(node.enext ↓, key ↓, flag ↑);
14:     node ← node.vnext;
15:   end while
16: end procedure
```

When a thread wants to delete a vertex from the concurrent graph, it invokes the Remove method of the list-based set by passing the head of the vertex list and the key of the vertex to be deleted. This is described in Algorithm 2. Once a vertex has been deleted from the vertex list, its outgoing edges are logically removed automatically. This is because any operation in the edge list of the deleted vertex, will first verify for the presence of the vertex in the graph. After the deletion of the vertex in the vertex list, we must also delete the incoming edges to the deleted vertex. This is described in Algorithm 3. The RemoveIncomingEdges method performs a traversal of the entire vertex list, to check if any of the existing reachable vertices contains an edge node corresponding to the deleted vertex in their edge list. If such an edge node is present, it is simply deleted from its edge list.

It is to be noted that performing the deletion of incoming edges of deleted vertices is an optional step as this does not affect the correctness of the algorithm. In other words, even if edge nodes corresponding to the deleted vertices are still reachable, no other method's correctness is affected by their presence. In later section, we present results of algorithms without removing their incoming edges.

4.2 ***Update Edge Methods*** - AddEdge & RemoveEdge

Algorithm 4 AddEdge Method: $GNode(key_2)$ gets added to the edge list of $GNode(key_1).enext$, if it is not present and returns true.

```
17: procedure AddEdge (key1 ↓,key2 ↓, flag ↑)
18:   G.Contains(G.Head ↓, key1 ↓, u ↑, status1 ↑);
19:   G.Contains(G.Head ↓, key2 ↓, v ↑ status2 ↑);
20:   if ( status1 = false ∨ status2 = false) then
21:     flag ← false;
22:     return;
23:   end if
24:   G.Contains(G.Head ↓, key1 ↓, u ↑, status1 ↑);
25:   if (status1 = false) then
26:     flag ← false;
27:     return;
28:   end if
29:   G.Add(u.enext ↓, key2 ↓, status ↑);
30:   flag ← status;
31: end procedure
```

Algorithm 5 RemoveEdge Method: $GNode(key_2)$ gets removed from the edge list of $GNode(key_1).enext$, if it is present. Returns unsuccessful if the edge is not present earlier.

```
32: procedure RemoveEdge (key1 ↓, key2 ↓, flag ↑)
33:   G.Contains(G.Head ↓, key1 ↓, u ↑, status1 ↑);
34:   G.Contains(G.Head ↓, key2 ↓, v ↑, status2 ↑);
35:   if ( status1 = false ∨ status2 = false) then
36:     flag ← false;
37:     return;
38:   end if
39:   G.Contains(G.Head ↓, key1 ↓, u ↑, status1 ↑);
40:   if (status1 = false) then
41:     flag ← false;
42:     return;
43:   end if
44:   G.Remove(u.enext ↓, key2 ↓, status ↑);
45:   flag ← status;
46: end procedure
```

The $AddEdge(u, v)$ method starts by checking for the presence of vertices u and v in the vertex list of the graph by invoking the $Contains$ method in the Lines 18 & 19 respectively. After this, once again u is validated in the Line 24, the reason for this is explained by an example in Figure 2.

After successful checking for the presence of both the vertices $AddEdge$ invoked the $Add(v)$ in the u's adjacency list($enext$) of respective data-structure in the Line 29. It adds the edge (u, v), if not present earlier and then it returns $true$. If the edge is already in the graph it simply returns $true$. But if either the vertices u or v is not present, it returns $false$. The generic $AddEdge$ method is given in the Algorithm 4.

The $RemoveEdge(u, v)$ method proceeds similar to the $AddEdge(u, v)$, it first checks for the presence of vertices u and v in the vertex list of the graph by invoking the $Contains$ method in the Lines 33 & 33 respectively. After this, once again u is checked for presence in the Line 39 for the same reason as described earlier. After successful checking for the presence of both the vertices, $RemoveEdge$ invokes the $Remove(v)$ in the u's adjacency list($enext$). It removes the edge (u, v), if it is present in the graph and then it returns $true$. But if either the vertices u or v are not present, it returns $false$. The generic $RemoveEdge$ method is given in the Algorithm 5.

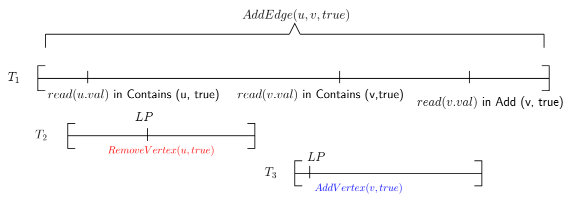

Figure 2: This figure depicts why we need an additional check to locate vertex in AddEdge. A thread T_1 tries to perform AddEdge (u, v, true), first invokes $contains(u)$. Just after T_1 has verified vertex u to be present in vertex list, thread T_2 deletes vertex u. Also vertex v gets added by thread T_3 just before T_1 verifies it. Now thread T_1 has successfully tested for the presence of vertices u and v in the vertex list, and then it proceeds to add edge (u, v), returning true. However, as is evident, in no possible sequential history equivalent to the given concurrent execution will both the vertices u and v exist together. Hence an additional check must be performed before proceeding to actually add the edge. With this additional check, in this scenario, AddEdge(u,v) will return false on checking that vertex u has been deleted.

4.3 *Read-Only Methods* - ContainsVertex & ContainsEdge

Method $ContainsVertex(u)$ simply invokes the Contains method of the list-based set by passing the head of the vertex list and the vertex key to be searched. The $ContainsEdge(u, v)$ method first invokes the Contains method of the list-based set for each of the vertex keys. If they are found in the vertex list, it then calls for Contains in the edge list of the first vertex. These methods return *true* if the vertex/edge node it was searching for is present and unmarked, otherwise returns $false$. The generic $ContainsVertex$ & $ContainsEdge$ methods are given in the Algorithm 6 & 7 respectively. These read-only methods are wait-free based on our assumption that the *Contains* method of the underlying list is wait-free.

Algorithm 6 ContainsVertex Method: Returns $true$ if $GNode(key)$ is present in vertex list and returns $false$ otherwise.

```
procedure CONTAINSVERTEX (key ↓, flag ↑)
 G.Contains(G.Head ↓, key ↓, u ↑, status ↑);
 flag ← status;
end procedure
```

Algorithm 7 ContainsEdge Method: Returns $true$ if $GNode(key_2)$ is part of the edge list of $GNode(key_1)$ and returns $false$ otherwise.

```
procedure CONTAINSEDGE (key1 ↓, key2 ↓, flag ↑)
 G.Contains(G.Head ↓, key1 ↓, u ↑, status1 ↑);
 G.Contains(G.Head ↓, key2 ↓, v ↑, status2 ↑);
 if ( status1 = false ∨ status2 = false) then
  flag ← false;
  return;
 end if
 G.Contains(u.enext ↓, key2 ↓, x ↑, status ↑);
 flag ← status;
end procedure
```

4.4 Correctness: Linearization Points

In this subsection, we describe the *Linearization Points*(LP) [6] of all methods of our concurrent graph data structure. Due to lack of space, the proof of the correctness of concurrent graph data structure is given in the full paper [8]. We try to formalise the proof of our concurrent graph data structure based on LP events of the methods and is also based on [11].

The LP of $AddVertex(key, true)$, $RemoveVertex(key, true)$, $RemoveVertex(key, false)$, $ContainsVertex(key, true)$ & $ContainsVertex(key, false)$ is the linearization point of $Add(key, true)$, $Remove(key, true)$, $Remove(key, false)$, $Contains(key, true)$ & $Contains(key, false)$ in the corresponding concurrent list-based set implementation.

We linearize a successful $AddEdge(key_1, key_2, true)$ method call within its execution interval: (1) if there is no successful concurrent delete vertex on key_1 and key_2, the LP is defined as the LP of the $Add(key, true)$; (2) if there is a successful concurrent delete vertex on key_1 or key_2 or both, the LP is the point immediately before the first LP of successful concurrent delete on vertex key_1 or key_2.

In case of an unsuccessful $AddEdge(key_1, key_2, false)$ method call, the LP is defined to be within its execution interval: (1) if there is no successful concurrent add vertex on key_1 and key_2, LP is defined as the LP of the $Contains(key, false)$ when there is no concurrent Add method; (2) if there is a successful concurrent add vertex on key_1 or key_2 or both, it is linearized at the point immediately before the LP of the first successful concurrent add vertex on key_1 or key_2.

We linearize a successful *RemoveEdge* method call within its execution interval: (1) if there is no successful concurrent delete vertex, the LP is defined as the LP of the $Remove(key, true)$; (2) if there is a successful concurrent delete vertex on key_1 or key_2 or both, it is linearized just before the LP of the first successful concurrent delete vertex on key_1 or key_2. In case of an unsuccessful $RemoveEdge(key_1, key_2, false)$ method call, the LP is defined to be the same as the LP of $AddEdge(key_1, key_2, false)$ within its execution interval.

We linearize a successful $ContainsEdge(key_1, key_2)$ method call within its execution interval: (1) if there is no successful concurrent delete vertex on key_1 and key_2, the LP is defined as the LP of $Contains(key, true)$; (2) if there is a successful concurrent delete vertex on key_1 or key_2, it is linearized immediately before the LP of the first successful concurrent delete on the corresponding vertex. In case of an unsuccessful $ContainsEdge(key_1, key_2, false)$ method call, the LP is defined to be within its execution interval: (1) if there is no successful concurrent add edge (key_1, key_2), the LP is the LP of the $Contains(key, false)$; (2) if there is a successful concurrent add edge on (key_1, key_2), it is linearized immediately before the LP of that successful concurrent add edge.

Lemma 1 *The history H generated by the interleaving of any of the methods of the concurrent graph data structure, is linearizable.*
Proof in [8].

Lemma 2 *The methods of the concurrent graph data structure have the same progress guarantees as provided by the concurrent list-based set implementation used to implement it.*
Proof in [8].

5 Simulation Results & Analysis

We performed our tests on 2 sockets, 10 cores per socket, Intel Xeon (R) CPU E5-2630 v4 running at 2.20 GHz frequency. Each core supports 2 hardware threads. Every core's L1 has 64k, L2 has 256k cache memory are private

(a) CV:15%, CE:15%, AddE:25%, DelE:10%, AddV:25% & DelV:10%.

(b) CV:40%, CE:40%, AddE:7%, DelE:3%, AddV:7% & DelV:3%

(c) AddE:50%, DelE: 50% and rest are 0%

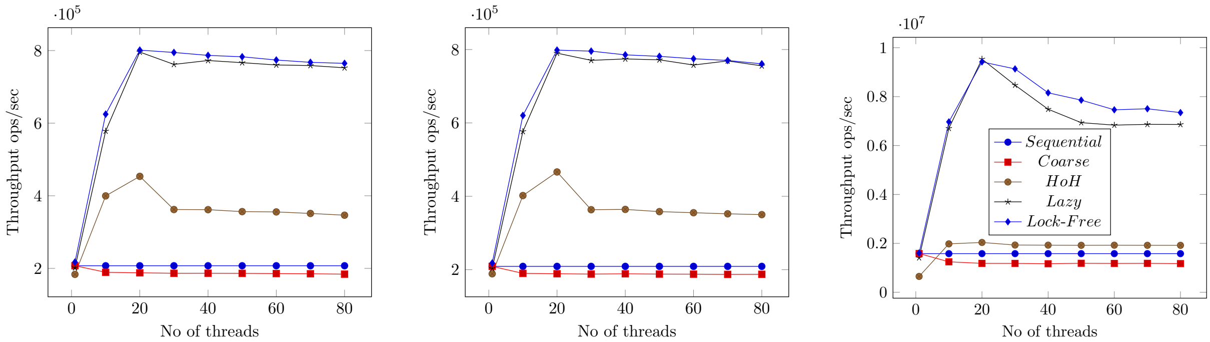

Figure 3: Generic Graph Data-Structure Results

to that core; L3 cache (25MB) is shared across all cores of a processors. The tests were performed in a controlled environment, where we were the sole users of the system. The implementation[a] has been done in C++ (without any garbage collection) and threading is achieved by using Posix threads and all the programs were optimized at O3 level.

In the experiments conducted, we start with an initial complete graph. When the program starts, it creates fixed number of threads and each thread randomly performs a set of operations chosen by a particular workload distribution. Here, the evaluation metric used is the number of operations completed in unit time. We measure throughput obtained on running the experiment for 20 seconds and present the results for the following workload distributions: (1) *Update-dominated*: 25% $AddVertex$, 25% $AddEdge$, 10% $RemoveVertex$, 10% $RemoveEdge$, 15% $ContainsVertex$ and 15% $ContainsEdge$; (2) *Contains-dominated*: 40% $ContainsVertex$, 40% $ContainsEdge$, 7% $AddVertex$, 7% $AddEdge$, 3% $RemoveVertex$ and 3% $RemoveEdge$; (3) *Edge-updates*: 50% $AddEdge$, 50% $RemoveEdge$ and rest are 0%. Figure 3 depicts the results for the data-structure methods. Each data point is obtained after averaging for 5 iterations. We assume that duplicate vertices are not inserted.

It is to be noted that all the variants of the concurrent data-structure are implemented without Deletion of Incoming Edges (DIE) for deleted vertices since it achieves higher throughput than the one with DIE. This can be attributed to the observation that it is cost inefficient to traverse all the vertices to search for the incoming edges of the deleted vertices. We tested different variants of the data-structure for different number of threads - LockFree implementation [2], Lazy list-based set implementation [3], hand-over-hand coupling list, CoarseLock [4, Chap 9]: which supports concurrent operations by acquiring a global lock and the sequential implementation. The figures depict that the performance of the concurrent data structure is similar to that of the lazy list-based set performance. We noted on an average 5x increased throughput.

6 Conclusion & Future Direction

In this paper, we have shown how to construct a fully dynamic concurrent graph data structure, which allows threads to concurrently add/delete vertices/edges. The graph is constructed by the composition of the well known concurrent list-based set data structure from the literature. Our construction is generic, in the sense that it can be used to obtain various progress guarantees, depending on the granularity of the underlying concurrent set implementation - either blocking or non-blocking. We believe that there are many applications that can benefit from this concurrent graph structure. An important application that inspired us is SGT in Databases and Transactional Memory. For proving linearizability, we have identified the linearization points of all the methods. We have compared the performance of the different variants of concurrent data-structure and we achieve on an average 5x increased throughput.

References

[1] Camil Demetrescu, Irene Finocchi, and Giuseppe F. Italiano. Dynamic graphs. In *Handbook of Data Structures and Applications*. 2004. URL: http://dx.doi.org/10.1201/9781420035179.ch36, `doi:10.1201/9781420035179.ch36`.

[2] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings*, pages 300–314, 2001. URL: https://doi.org/10.1007/3-540-45414-4_21, `doi:10.1007/3-540-45414-4_21`.

[3] 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. URL: http://dx.doi.org/10.1142/S0129626407003125, `doi:10.1142/S0129626407003125`.

[4] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[a] The complete source code is available on Github [10].

[5] Maurice Herlihy and Nir Shavit. On the nature of progress. In *Principles of Distributed Systems - 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings*, pages 313–328, 2011. URL: http://dx.doi.org/10.1007/978-3-642-25873-2_22, doi:10.1007/978-3-642-25873-2_22.

[6] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:http://doi.acm.org/10.1145/78969.78972.

[7] Nikolaos D. Kallimanis and Eleni Kanellou. Wait-free concurrent graph objects with dynamic traversals. In *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*, pages 27:1–27:17, 2015. URL: http://dx.doi.org/10.4230/LIPIcs.OPODIS.2015.27, doi:10.4230/LIPIcs.OPODIS.2015.27.

[8] Sathya Peri, Muktikanta Sa, and Nandini Singhal. Maintaining acyclicity of concurrent graphs. *CoRR*, abs/1611.03947, 2016. URL: http://arxiv.org/abs/1611.03947.

[9] Sathya Peri, Muktikanta Sa, and Nandini Singhal. Building efficient concurrent graphs through composition of set objects. *CoRR*, abs/1710.08296, 2017. URL: https://arxiv.org/abs/1710.08296.

[10] Sathya Peri, Muktikanta Sa, and Nandini Singhal. Concurrentgraphds. https://github.com/PDCRL/ConcurrentGraphDS, 2017.

[11] Sathya Peri, Muktikanta Sa, and Nandini Singhal. Proving correctness of concurrent objects by validating linearization points. *CoRR*, abs/1705.02884, 2017. URL: http://arxiv.org/abs/1705.02884.