Maintenance of Strongly Connected Component in Shared-memory Graph

Muktikanta Sa

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

Abstract. In this paper, we present an on-line full dynamic algorithm for maintaining strongly connected component of a directed graph in a shared memory architecture. The edges and vertices are added or deleted concurrently by fixed number of threads. To the the best of our knowledge, this is the first work to propose using linearizable concurrent directed graph and is build using both ordered and unordered list-based set. We provide an empirical comparison against sequential and course-grained. The results show our algorithm performance is similar to the lazy linked list and the throughput is increased between 3 to 7x depending on different workload distributions and applications. We believe that there are huge applications in the on-line graph. Finally, we show how the algorithm can be extended to community detection in on-line graph.

Keywords: concurrent data structure; directed graph; strong connected components, locks; connectivity on directed graphs; dynamic graph algorithms;

1 Introduction

Generally the real-world practical graph always dynamically change over time. Dynamic graphs are the one's which are subjected to a sequence of changes like insertion, deletion of vertices and/or edges [5]. Dynamic graph algorithms are used extensively and it has been studied for several decades. Many important results have been achieved for fundamental dynamic graph problems and some of these problems are very challenging i.e,finding cycles, graph coloring, minimum spanning tree, shortest path between a pair of vertices, connectivity, 2-edge & 2-vertex connectivity, transitive closure, strongly connected components, flow network, etc (see, e.g., the survey in [5]). The biological graph networks are very complicated due to their complex layered architecture and size. Graphs Networks, typically involving finding the perfect match or similarities on gene expression for evolution like disease progression [16] are even more challenging to the research community.

We have been specifically motivated by largely used problem of fully dynamic evolution *Strongly Connected Components*(SCC). Detection of SCC in dynamically changing graph affects a large community both in the theoretical computer science and the network community. SCC detection on static networks fails to capture the natural phenomena and important dynamics. Discovering SCCs on dynamic graph helps uncover the laws in processes of graph evolution, which have

been proven necessary to capture essential structural information in on-line social networking platforms (facebook, linkedin, google+, twitter, quora, etc.). SCC often merges or splits because of the changing friendship over time. A common application of SCC on these social graph is to check weather two members belong to the same SCC (or community). So, we define the *SCC(id1,id2)*: which checks if there is a directed path from $id1$ to $id1$ and the other way round. In general, a social network graph handles the concurrency control over a set of users or threads running concurrently. A thread as a block of code is invoked by the help of methods to access multiple shared memory objects atomically.

In this paper, we present a new shared-memory algorithm called as *SMSCC* for maintaining SCC in fully dynamic directed graphs.The following are the key contributions of this work:

1. Firstly, we designed a new incremental algorithm(SMDSCC) for maintaining SCC dynamically. i.e, after inserting an edge or a vertex how quickly we update the SCC.
2. Secondly, we designed a decremental algorithm(SMISCC) for maintaining SCC dynamically. i.e, after deleting an edge or a vertex how quickly we update the SCC.
3. Thirdly, a algorithm for maintaining fully dynamic SCC(SMSCC).
4. An empirical comparison against sequential and course-grained.
5. Our algorithm is work-efficient for most on-line graphs.
6. An application suite : community detection on-line graph.

We have not found any comparable concurrent data-structure for solving this strongly connected components problem in shared-memory architecture. Hence we crosscheck against sequential and course-grained implementations.

1.1 Background and Related Work

Let a concurrent directed graph $G = (V, E)$, $G(V)$ is a set of vertices and set of $G(E)$ directed edges. We use *Adj(u)* to denote the set of neighbors of a vertex u. The $G(E)$ is collection of both outgoing and incoming neighbors, i.e., *Adj(u)* = { for outgoing edges $v : \langle u, v\rangle$ & for incoming edges $w : \langle w, u\rangle \in G(E)$}. Each edge connects an ordered pair of vertices currently belongs to $G(V)$. And this G is dynamically being modified by a fixed set of concurrent running threads. Our dynamic graph setting, threads can perform insertion/deletion of edges and insertion of vertices. We assume that all the vertices have unique identification key, which is captured by *val* field in Gnode structure as shown in the Section 4.

Definition 1.(Reachability),*Given a graph G=(V,E) and two vertices* $u, v \in G(V)$*, a vertex v is reachable from another vertex u if there is a path from u to v.*

Definition 2.(SCC)*, A graph is strongly connected if there is a* directed path *from any vertex to every other vertex. Formally,*
Let $G = (V, E)$ *be a directed graph. We say two nodes* $u, v \in G(V)$ *are called strongly connected iff v is reachable from u and also u is reachable from v. A strongly connected component(or SCC) of G is a set* $C \subseteq G(V)$ *such that:*

1. *C is not empty.*
2. *For any $u, v \in C$: u and v are strongly connected.*
3. *For any $u \in C$ and $v \in V - C$: u and v are not strongly connected.*

Apart from the definition, SCC also satisfies the equivalence relation on the set of vertices: *Reflexive:* every vertex v is strongly connected to itself. *Symmetric:* if u is strongly connected to v, then v is strongly connected to u. *Transitive:* if u is strongly connected to v and v is strongly connected to w, then u is also strongly connected to w.

1.2 Related Work

There have been many parallel computing algorithms proposed for computing SCC both in directed and undirected graphs. Hopcroft and Tarjan [11] presented the first algorithm to compute the connected components of a graph using the depth first searches(DFS) approaches. Hirschburg et al. [10] presented a novel parallel algorithm for finding the connected components in an undirected graph. Shiloach and Vishkin [14] proposed an parallel computing *O(logV)* algorithm. In 1981, Shiloach and Even [13] presented a first decremental algorithm that finds all connected components in dynamic graphs, only edges are deleted.

Henzinger and King [7] also proposed a new algorithm that maintains spanning tree for each connected components, which helps them to update the data-structure quickly only when deletion of edge occurs.

The main drawback of these algorithms is, they are expensive and need more space for each change to the data-structure. Also they don't utilize the advantages of multi-core or multi-processor architecture.

In 2014, Slota, et. al., proposed a parallel multistep based algorithm using both BFS and coloring technique to detect the SCC in large graphs. Later they used the trimming methodology to reduce the search space of the graph to achieve better performance. Recently Bender et. al, [2] proposed incremental algorithm to maintain the SCC of a dynamic graph. Also Bloemen. et.al., [3] proposed a novel parallel on-the-fly algorithm for SCC decomposition in multi-core system, they used advantages of Tarjan's algorithm.

Bader. et. al., [1] developed a data-structure known as STRINGER for dynamic graph problems. They used combination of both adjacency matrices and Compressed Sparse Row (CSR) representation of graph. And they claimed that the STRINGER helps faster insertions and better spatio-temporal locality as compare to the adjacency lists representations. Later they also developed a CUDA version of the STRINGER called that cuSTRINGER [6], which supports dynamic graph algorithms for GPUs.

None of above proposed algorithms clarify how the internal share-memory access is achieved by the multi-threads/processors and how the memory is synchronized, whether the data-structure is linealizable or not, etc. In this paper we able to address these problems.

The rest of the paper is organized as follows. In the Section 2, we define the system model, preliminaries and design principles. In Section 3 we define the high level overview of the algorithm AddEdge & RemoveEdge. We define the

data-structure of SCC-graph in Section 4 and in the Section 5 we define technical details of all our algorithms and some of the pseudo-codes. In the Section 6 we give high level correctness proof and in the Section 7 we analyze the experimental results. Finally we concluded in the Section 8 along with future direction and discussion.

2 System Model & Preliminaries

In this paper, we have considered that our system consists of fixed set of p processors, accessed by a finite set of n threads T_1, T_2 ...., T_n that run in a completely asynchronous manner and communicate through shared objects on which they perform atomic *read*, *write*, *fetch-and-add*(FAA) operations. A FAA operation takes two arguments (loc, incVal), where `loc` is the address location from where it fetches the value, then adds `incVal` to it and then writes back to the result `loc`.

We assume that each thread has a unique identifier, it is assigned at the time of thread creation. Each thread invokes a method which may be composed of shared-memory objects and local cipherings. We make no assumptions about the relative speeds of the threads and assume none of these processors and threads fail.

As we said earlier our proposed algorithms are implementations of shared objects and a share object is an abstraction set of methods defined as `SCC` class in the Section 4. It has set of methods and each method has its sequential specification. To prove a concurrent data structure to be correct, *linearizability* proposed by Herlihy & Wing [9] is the standard correctness criterion. Anytime a thread invokes a method for an object, it follows until it receives a response. In may be the case a method's invocation is pending if has not received a response. For any sequential history in which the methods are ordered by their LPs.
Progress: An execution is *deadlock-free* if it guarantees minimal progress in every *crash-free* [8] execution, and maximal progress if it is starvation-free. An execution is *crash-free* if it guarantees minimal progress in every uniformly isolating history, and maximal progress in some such history [8].
Design Principles: We developed a set of correct behaviour for our algorithm and implementation.

1. *thread-safety*: The SCC-graph data-structure can be shared by fixed number of multiple threads at all times, which ensures all fulfill their requirement specifications and behave properly without unintended interaction.
2. *lock-freedom*: apply non-blocking techniques to provide an implementation of thread-safe C++ dynamic array based on the current C++ memory model.
3. *portability*: Generally our algorithms do not rely on specific hardware architectures, rather it is based on asynchronous memory model.
4. *simplicity*: The algorithm keeps the implementation simple to allow the correctness verification, like linearizability or model-based testing.

Notations: We denoted $\downarrow$, $\uparrow$ as input and output arguments to each method respectively and our pseudo-code is mixed of C++ and JAVA language format.

3 An Overview of the Algorithm

Before getting into the technical details of the algorithm, we first provide an overview of the design. The SCC class supports some basic operations: AddVertex, AddEdge, RemoveEdge, checkSCC, blongsTo, etc. and all of these methods are dead-lock free. The high-level overview of the AddEdge and RemoveEdge methods are given bellow and the technical details are in the Section 5.

AddEdge (u, v):

1. Checks the presence of vertex u and v in the SCC-Graph. If both are present, adds v in the u's edge list and adds -u in the v's edge list. Else returns false.
2. After adding the edge successful, checks the SCC id, *ccid* of each vertices.
3. if $u.ccid = v.ccid$, returns true, as no changes to the current SCC. Else goto step 4.
4. Checks the reachable path from vertex v to u, if it is, goto step 5, else return true, as no changes to the current SCC.
5. Runs the limited version of Tarjan's algorithm, process the affected SCCs with its vertices and edges, merge then to create a new SCC.
   - Firstly, create new scc with any single old vertex, later adds rest of vertices to that newly created SCC, and then disconnects from old SCC.

RemoveEdge (u, v):

1. Checks the presence of vertex u and v in the SCC-Graph. If both are present, removes v from the u's edge list and removes -u from the v's edge list. Else returns false.
2. After deleting the edge, checks the *ccid* of both the vertices.
3. if $u.ccid \neq v.ccid$ returns true, as no changes to the current SCC. Else goto step 4.
4. Run the forward and backward DFS algorithm(limited version of Kosaraju's algorithm), process all the affected vertices in that SCC and creates new SCCs.
   - For each new iteration.
     - Firstly, creates new scc with any one old vertex belongs to it, later adds rest of vertices to that newly created SCC and then disconnects from the old SCC.

4 Construction of SCC-Graph structure

In this section we present the node structures of vertex, edge and scc to construct the SCC graph. The node structures are all based on the same basic idea of lazy set implementation using linked list. This data-structure is designed similarly based on the adjacency list representation of any graph. It is implemented as a collection (list) of SCCs, wherein each SCC holds the list of vertex set belongs to it, and each vertex holds the edge list (both incoming and outgoing edges). We represent all incoming edges with negative sign followed by `val` and outgoing edges with the `val`, as shown in the Fig 1b.

The `Gnode` structure is a normal node and has five fields. The `val` field is the actual value of the node. If it is a vertex node, it stores the vertex id, if

it is an outgoing edge, it stores the `val` of the destination vertex, if it is an incoming edge, it stores the negative of source vertex's `val`. The main idea of storing both incoming and outgoing edges for each vertex helps to explore the graph backward and forward manner respectively. And also it helps to trim the SCC-Graph after deleting a vertex, i.e, once a thread successfully deleted a vertex all its incoming and and outgoing edges needs to be removed quickly instate of iterating over whole SCC-Graph. The vertex and edge nodes are sorted in the `val` (lower to higher) order, it provides an efficient way to search when an item is absent. The boolean `marked` field is used to set the node and helps traversal to the target node without lock, we maintain an invariant that every unmarked node is reachable from the sentinel node `Head`. If a node is marked, then that is not logically present in the list. Each node has a `lock` field, that helps to achieve the *fine-grained* concurrency. Each node can be locked by invoking `lock()` and `unlock()` methods.It just a fine-grained locking technique, helps multiple threads can traverse the list concurrently. The `vnext` & `enext` fields are the atomic references to the next vertex node in the vertex list and the next edge node in the edge list of a vertex respectively.

```
unsigned long ccid;
unsigned long ccCount
typedef struct Gnode{
  long val;
  bool marked;
  Lock lock;
  struct Gnode *vnext;
  struct Gnode *enext;
}slist_t;
typedef struct CCnode{
  long ccno;
  bool marked;
  Lock lock;
  struct Gnode* vnext;
  struct CCnode *next;
}cclist_t
 class SCC{
  CCnode CCHead, CCTail;
  bool AddVertex(u);
  bool RemoveVertex(u)
  bool AddEdge(u, v);
  bool RemoveEdge(u, v);
  bool checkSCC(u,v);
  int  blongsTo(v);
};
```

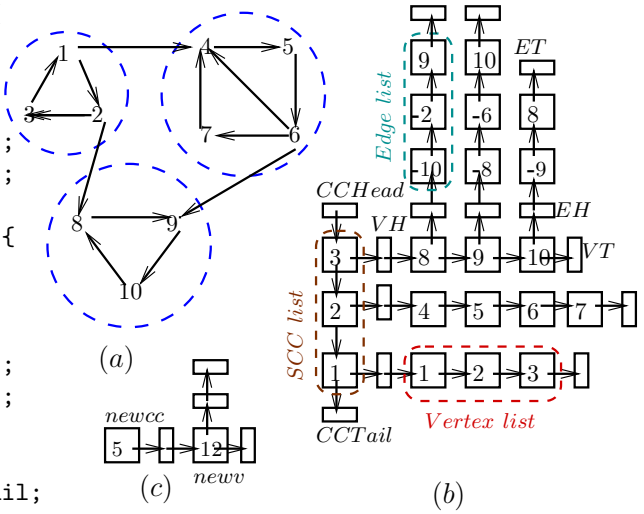

Fig. 1: (a). An example of a directed graph having three SCCs. (b). The SCC-Graph representation of (a), Each SCCs have their own `ccno` and vertex list, each vertex have their own adjacency vertex(both incoming(-ve) and outgoing(+ve)) represent in edge list. e.g. vertex 10 present in SCC 3 & has an incoming edge(-9) and an outgoing edge(8). (c). Structure of new SCC: whenever a new vertex is added, a new SCC is created with new vertex and then inserted at the beginning of the SCC-Graph.

The `CCnode` structure is used for holding all vertices belonging to a SCC. Like `Gnode`, it has five fields. The `ccno` field is the actual scc key value and unique for each SCC. Once a key assigned to a SCC, same key will never generate again.

We assume our system provides infinite number of unique key and had no upper bound. The boolean `marked` and `lock` have same meaning as `Gnode`. The `vnext` and `next` fileds are the atomic references to vertex head(VH) and next `CCnode`. We have two atomic variables `ccid` and `ccCount` used to hold the unique id for each CCnode and total number of SCCs respectively.

Finally the `SCC` class is the actual abstract class, which coordinates all operation activities. This class uses two type of nodes, `Gnode` and `CCnode`. The vertex and edge nodes are represent by Gnode and the SCC nodes are represented by CCnode and also has two sentinel nodes `CCHead` and `CCTail`. The *SCC* class supports four basic graph operations *AddVertex, AddEdge* and *RemoveVertex, RemoveEdge*, and also supports some application specific methods, *checkSCC, blongsToCommunity*, etc. The detail working and pseudo code is given in the next section.

5 Algorithms

In this section we present SMSCC, the actual algorithm for maintaining strongly connected components of fully dynamic directed graph in a shared memory system. The edges and vertices are added/removed concurrently by fixed set of threads. In Section 3 we discussed the high level overview of two methods. The technical details of all the methods are discussed here

- *AddEdge (u, v)*: Creates a new edge if not present earlier and then checks whether the SCC is affected, if it is, tries to restore the SCC, if unable to do that returns false.
- *RemoveEdge (u, v)*: Tries to delete the edge if it is already present and then checks whether the SCC is affected, if it is, try to restore the SCC, if unable to do that returns false.
- *AddVertex (u)*: Adds a new SCC having the vertex u to the SCC-graph, if the vertex is not present earlier. We assume there is no duplicate vertex in the SCC-graph, means once a vertex is added, same vertex will never be invoked by AddVertex method on this key again.
- *RemoveVertex (u)*: Removes a vertex u in a SCC from the SCC-graph, if the vertex is present earlier. Then checks whether the SCC is affected, if it is, try to restore the SCC, if unable to do that returns false. After that it deletes all outgoing and incoming edges.
- *checkSCC(u, v)*: Checks whether u and v are in the same strongly connected component at a given instance. This method used for different applications, to check whether two he persons are present in the same community or not.
- *blongsToCommunity(u)*: returns the *ccno* of an SCC, in which u is belongs to. This method is used for applications, to find the community of a person.

5.1 Edge or Vertex Insertion

After inserting an edge to the graph, how quickly we update the SCC instate of starting everything from the scratch. The details of the algorithm is given bellow. If we allowed only insertion of edges or vertices, called it as an incremental algorithm. For this we used the modified version of Tarjan's [11] algorithm to restore the affected SCC after inserting an edge iff it violates the SCC-Graph. Whereas AddVertex will not affect the SCC-Graph.

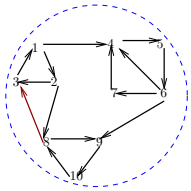

Fig. 2: An example after addition of the edge $(8, 3)$ in the Fig 1(a)

To add an edge, we invoke the $AddEdge(u, v)$ method presented in the Algorithm 15. First it checks the presence of vertices u and v by invoking the *locateSCC* method (Algorithm 3) from Line 215 to 223. If any one of these vertices is not present or the edge is present, we simply return false. After successful check of u and v, in the the Line 224 we try to add the edge node v(outgoing edge) in the u's edge list and the edge node -u(incoming edge) in the v's edge list. After successful addition of both the edges, we check if any changes to the SCC-Graph. For that, first we check the `ccno` of both the vertices(in the Line 226). If they are equal, no changes to the SCC-Graph as it is added within a SCC, and we return true. On the other hand we check if there is a reachable path from vertex v to u, if it is, we merge all the SCCs which are in the reachable path by invoking the method *findSCCafterAddE()*(in the Line 232). If the reachable path is not exist, the SCC remain unchanged, we simply return true.
While merging all the affected SCCs by invoking *findSCCafterAddE()* method, we only consider the the vertices and edges which are affected due to addition of edge (u, v). We used the modified version of the Tarjan's algorithm [11] because all the vertices which are in the reachable path are pushed to the stack *Stk* in one iteration and then popped all to build a new SCC in one iteration as well. When the findSCCafterAddEmethod is called, it creates a local stack `Stk` and other variables for processing the the Tarjan's algorithm and merging the affected SCCs to single SCC. In the process of merging, for the first popped vertex , say x, we create a new SCC, say `newcc`(with its edges), and add it at the beginning of the SCC-Graph and then disconnect it from the old SCC, this is done by the method *createNewSCCwithOldV ()*, which is invoked in the Line 206, it is similar to the *addVertexSCC ()* method(Algorithm 9). From the second popped vertex onwards, we just add the vertex(with its edges) to the `newcc` to the sorted position and detached the link from the old SCC, this is done by the method *addOldVInSCC ()*, which is invoked in the Line 210. Any time we are inserting or detaching a vertex from the list we validate as some other threads concurrent may add or delete to the predecessor or successor of that vertex. We always maintain a invariant that any unmarkable node is always reachable from the respective `Head` of the list. In the Fig 2, we have shown an example after addition of the edge $(8, 3)$ in the Fig 1(a), how all three SCCs are merged to form a new SCC.
A new vertex `newv` is added by invoking *AddVertex* method. Each time this method is called with new vertex id, which is generated from the last vertex

id plus one. This increment is done by atomic operation fetch-and-add (FAA). We assume all vertices have unique id and the system has unbounded number of such keys, once it is added to SCC-Graph, will never assign this id to any other vertex. Each time a new SCC also created with new `ccno`, say `newcc`. After that `newv` is added to `newcc` and then `newcc` is inserted at the beginning of the SCC-Graph and it never affects the properties of SCC-Graph. The structure of `newcc` is shown in the Fig 1(c).

5.2 Edge or Vertex Deletion

Like AddEdge, after deleting an edge or a vertex from the graph, how quickly we update the SCC. If we allowed only deletion of edges or vertices to the SCC-graph, called it as decremental algorithm. We used the limited version of Kosaraju's [4, Chap 22, Sec 22.5] algorithm to resort the affected SCC after deleting an edge or a vertex only iff it violates the SCC-Graph. To remove an edge, we invoke the *RemoveEdge*(u, v) method (Algorithm 16). First we check the presence of vertices u and v in the SCC-Graph by invoking the *locateSCC* method (Algorithm 3) from Line 242 to 250. If any one of these vertices or the edge is not present, we simply return false. After successful presence of u and v(in the the Line 251) we try to remove the edge node v(outgoing edge) in the u's vertex list and the edge node -u(incoming edge) in the v's vertex list, if present earlier.

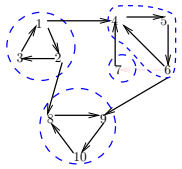

Fig. 3: An example after deletion of the edge $(8, 7)$ in the Fig 1(a), the SCC breaks to two new SCCs.

Algorithm 1 It takes the input `pred` & `curr` of type ¡T¿, where ¡T¿ is either a slist_t(vertex or edge node) or cclist_t node. It returns true with the invariant an unmarkable node is reachable from Head or else it returns false

```
1: procedure bool VALIDATE (< T > pred ↓, < T > curr ↓ )
2:   return (pred.marked = false ∧ curr.marked = false ∧ pred.next = curr);
3: end procedure
```

Algorithm 2 It takes input as vertex `key` & it's SCC, returns the exact location of key in vertex list of scc.

```
4: procedure LOCATEV (key ↓, currc ↓, predv ↑, currv ↑)
5:   predv ← currc.vnext;
6:   currv ← predv.vnext;
7:   while (currv.val < key ) do
8:     predv ← currv;
9:     currv ← currv.vnext;
10:  end while
11: end procedure
```

Algorithm 3 It takes input as `key`, returns false if key is not present, else it returns true and the references to predc & currc of SCC and the references to predv & currv vertex having val == key.

```
12: procedure bool locateSCC (key ↓, predc ↑, currc ↑, predv ↑, currv ↑ )
13:   predc ← CCHead;
14:   currc ← predc.next;
15:   while (currc ≠ CCTail) do
16:     predv ← currc.vnext;
17:     currv ← predv.vnext;
18:     while (currv.val ≤ key ) do
19:       if (currv.val = key ∧ currv.marked = false) then
20:         return true;
21:       end if
22:       predv ← currv;
23:       currv ← currv.vnext;
24:     end while
25:     predc ← currc;
26:     currc ← currc.next;
27:   end while
28:   return false;
29: end procedure
```

Algorithm 4 It inserts a newcc with old vertex to SCC-Graph at the CCHead position.

```
30: procedure bool createNewSCCwithOldV (predv ↓, currv ↓)
31:   predc ← CCHead;
32:   currc ← predc.next;
33:   newcc ← createNewSCCwithNewV (currv↓);
34:   predc.lock(); currc.lock();
35:   predv.lock(); currv.lock();
36:   if (ValidateC (predc↓, currc↓) ∧ ValidateV (predv↓, currv↓)) then
37:     predv.vnext ← currv.vnext
38:     newcc.next ← currc;
39:     predc.next ← newcc;
40:     currv.unlock(); predv.unlock();
41:     currc.unlock(); predc.unlock();
42:     ccCount ← ccCount + 1; // atomic increment (FAA(1))
43:     return true;
44:   else
45:     currv.unlock(); predv.unlock();
46:     currc.unlock(); predc.unlock();
47:     return false;
48:   end if
49: end procedure
```

Algorithm 5 It takes input as `key` & EHead of a vertex, returns the exact location of key in the edge list.

```
50: procedure locateE (key ↓, EHead ↓, prede ↑, curre ↑)
51:   prede ← EHead;
52:   curre ← prede.enext;
53:   while (curre.val < key ) do
54:     prede ← curre;
55:     curre ← curre.enext;
56:   end while
57: end procedure
```

Algorithm 6 It inserts the old vertex to SCC.

```
58: procedure addOldVInSCC (newcc ↓, predv ↓, currv ↓);
59:   flag1 ← locateV (currv.val ↓, newcc ↑, predv1 ↑, currv1 ↑); //Algorithm 2
60:   predv1.lock(); currv1.lock();
61:   predv.lock(); currv.lock();
62:   if (flag1 = true ∧ ValidateV (predv1↓, currv1↓) ∧ ValidateV (predv↓, currv↓)) then
63:     predv.vnext ← currv.vnext; //detach the link
64:     currv.vnext ← currv1; // logical insertion
65:     predv1.vnext ← currv; // physical insertion
66:     currv.unlock(); predv.unlock();
67:     currv1.unlock(); predv1.unlock();
68:   else
69:     currv.unlock(); predv.unlock();
70:     currv1.unlock();predv1.unlock();
71:   end if
72: end procedure
```

Algorithm 7 It takes input as *key* & EHead of a vertex. If key is not present, adds the edge node in the EHead list, else returns false.

```
73: procedure bool addENode (key ↓, EHead ↓)
74:   locateE (key ↓, EHead ↓, prede ↑, curre ↑); //Algorithm 5
75:   newe ← createE (key)
76:   prede.lock(); curre.lock();
77:   if (curre.val ≠ key ∧ ValidateE (prede↓, curre↓)) then
78:     newe.enext ← curre; // logical insertion
79:     prede.enext ← newe; // physical insertion
80:     curre.unlock(); prede.unlock();
81:     return true;
82:   else
83:     curre.unlock(); prede.unlock();
84:     return false;
85:   end if
86: end procedure
```

Algorithm 8 It takes input as *key* & EHead of a vertex. If key is present, removes the edge node from the EHead list, else returns false.

```
87: procedure bool remENode (key ↓, EHead ↓)
88:   locateE (key ↓, EHead ↓, prede ↑, curre ↑);
89:   prede.lock(); curre.lock();
90:   if (curre.val = key ∧ ValidateE (prede↓, curre↓)) then
91:     curre.marked ← true; // logical deletion
92:     prede.enext ← curre.enext; // physical deletion
93:     curre.unlock(); prede.unlock();
94:     return true;
95:   else
96:     curre.unlock(); prede.unlock();
97:     return false;
98:   end if
99: end procedure
```

After successful deletion, we check if any changes to SCC-Graph. For that we check the `ccno` of both the vertices(Line 253). If the `ccnos` of both vertices are unequal, the SCC will not be affected, as edge (u, v) added between two SCCs. If the `ccnos` are equal, then there may be breaking of that single SCC to multiple SCCs. For that we use the modified version of Kosaraju's algorithm for find the all SCCs in that old SCC using forward and backward depth first

search(DFS) algorithm, for that we invoke the *findSCCafterRemoveE()* method in the Line 256.

In the Algorithm 13 we define to find all new SCCs after removal of the edge (u, v). We only process the vertices and the edges on that SCC. For each iteration we use DFS algorithm twice, one for forward DFS invoked by *DFSFW* (Algorithm 10 in the Line 162), and other invoked by *DFSBW* (Algorithm 11) in the Line 163. The DFSFW and DFSBW method locally processed the out-going and incoming edges respectively. In each iteration a new SCC is created(*createNewSCCwithOldV ()*) with old vertex(with its edges), say *newcc* and inserted at the beginning of the SCC-Graph, and then detached from the old SCC. For the next subsequent vertices belongs to newcc's id are added by the *addOldVInSCC ()* method to `newcc`. Like AddEdge, anytime we insert or detach a vertex from the vertex list we check the validation, because some threads concurrently may add or delete to the predecessor or successor of that vertex node. In the Fig 3, we have shown an example after deletion of the edge $(8, 7)$ in the Fig 1(a), how a single SCC is breakdown to two new SCCs.
For removing a vertex we invoke the *RemoveVertex* method and then use the limited version of Kosaraju's algorithm to resort the affected SCC, iff it violates the SCC-Graph. The mythology be similar as RemoveEdge, the pseudo code is shown in the Algorithm 18.

5.3 Check Community

The *checkSCC(id1, id2)*, *blongsToCommunity(id)*, etc. are used for different applications. For any online social networking it takes ids of two person and checks by invoking checkSCCmethod whether these two persons are belong to same community, if they are, either system admin or one of the person can send friendship suggestion or request to other person. Similarly, blongsToCommunitymethod reads a person id and tells which community he/she is belongs to, based on that it can do some activities in the community. Like this there are a huge number of SCC application in the dynamic graph and the requirement of efficient graph algorithms as well. The checkSCCand blongsToCommunitymethods are shown in the Algorithm 23 and 24 respectively.

5.4 Memory management

Our proposed algorithm depends on a explicit garbage collector(GC) for better memory management. We defined a separate GC method which is invoked by an independent thread in regular intervals of time. Our GC method is similar to Michael's Hazard Pointers technique [12] although it was designed for lock-free objects and we reclaim all three types of node. This GC thread does not affect the execution time.

Algorithm 9 If the key is not present earlier return false, else creates a new SCC, adds a new vertex to it, returns true. Inserts the new SCC at the beginning of the SCC-graph, just after the CCHead.

```
100: procedure bool addVertexSCC (key ↓ )
101:  flag ← locateSCC (key ↓, predc ↑, currc ↑, predv ↑, currv ↑); //Algorithm 3
102:  if (flag = true) then
103:   return false;
104:  else
105:   newcc ← createNewSCCwithNewV (key);
106:   predc ← CCHead;
107:   currc ← predc.next;
108:   predc.lock(); currc.lock();
109:   if (ValidateC (predc↓, currc↓)) then
110:    newcc.next ← currc; // logical insertion
111:    predc.next ← newrc; // physical insertion
112:    currc.unlock(); predc.unlock();
113:    ccCount ← ccCount + 1;
114:    return true;
115:   else
116:    currc.unlock(); predc.unlock();
117:    return false;
118:   end if
119:  end if
120: end procedure
```

Algorithm 10 DFS of forward traversal. Only process the outgoing edges

```
121: procedure DFSFW (slHead ↓, sl_edge ↓, num_cc ↓, cc ↓, SUCC ↓↑)
122:  SUCC[sl_edge-¿val] ← num_cc;
123:  for it ← sl_edge.enext.enext to it.next ≠ NULL do
124:   if (it.val > 0) then // checks for outgoing edges
125:    flag ← locateSCC (it.val ↓, predc ↑, currc ↑, predv ↑, currv ↑); //Algorithm 3
126:    if (SUCC[it.val] = cc ∧ flag = true) then
127:     DFSFW (slHead ↓, sl_edge ↓, num_cc ↓, cc ↓, SUCC ↓↑); //Algorithm 10
128:    end if
129:   end if
130:  end for
131: end procedure
```

Algorithm 11 DFS of the backward traversal. Only process the incoming edges

```
132: procedure DFSBW (slHead ↓, sl_edge ↓, num_cc ↓, cc ↓, PREC ↓↑)
133:  PREC[sl_edge-¿val] ← num_cc;
134:  for it ← sl_edge.enext.enext to it.next ≠ NULL do
135:   if (it.val < 0) then// checks for incoming edges
136:    flag ← locateSCC ((-1) ∗ it.val ↓, predc ↑, currc ↑, predv ↑, currv ↑); //Algorithm 3
137:    if (PREC[(-1)∗ it.val] = cc ∧ flag = true) then
138:     DFSBW (slHead ↓, sl_edge ↓, num_cc ↓, cc ↓, PREC ↓↑); //Algorithm 11
139:    end if
140:   end if
141:  end for
142: end procedure
```

Algorithm 12 It adds all affected SCCs to a SCC after AddEdge.

```
143: procedure bool findSCCafterAddE(scc1 ↓, scc2 ↓, n ↓)
144:   Stk ← new slist_t[n]; visited ← new bool[n];
145:   Root ← new long[n]; Comp ← new long[n];
146:   for i ← 0 to n do
147:     visited[i] ← false; Root[i] ← +∞; Comp[i] ← -1;
148:   end for
149:   currv ← scc2.vnext;
150:   mergeSCC (currv ↓, Stk↓, Root↓, Comp↓, visited↓); // Algorithm 14
151: end procedure
```

Algorithm 13 Finds all SCCs after RemoveEdge. Iterate only affected vertices and edges. The logic is based on modified version of Kosaraju's algorithm

```
152: procedure bool findSCCafterRemoveE(slHead ↓, cc ↓, n ↓)
153:   num_cc ← ccid;
154:   SUCC ← new long[n];
155:   PREC ← new long[n];
156:   for i ← 0 to n do
157:     SUCC[i] ← cc;
158:     PREC[i] ← cc;
159:   end for
160:   for it ← slHead.vnext to it.vnext ≠ NULL do
161:     if (SUCC[it.val] = cc) then
162:       DFSFW (slHead ↓, it ↓, num_cc ↓, cc ↓, SUCC ↓↑); // Algorithm 10
163:       DFSBW (slHead ↓, it ↓, num_cc ↓, cc ↓, PREC ↓↑); // Algorithm 11
164:       bool st ← true;
165:       cclist_t newcc;
166:       for (j ← 1 to n) do
167:         if (locateSCC (key ↓, predc ↑, currc ↑,predv ↑, currv ↑)) then //Algorithm 3
168:           if (SUCC[j] ≠ PREC[j]) then
169:             SUCC[j] ← cc;
170:             PREC[j] ← cc;
171:           else
172:             if (SUCC[j] = num_cc ∧ PREC[j] = num_cc)) then
173:               if (st = true) then // for first vertex
174:                 newcc ← createNewSCCwithOldV (predv ↓, currv ↓);// Algorithm 4
175:                 st ← false;
176:               else// for rest of vertices with same num_cc
177:                 addOldVInSCC (newcc ↓, predv ↓, currv ↓); // Algorithm 6
178:               end if
179:             end if
180:           end if
181:         end if
182:       end for
183:       num_cc ← num_cc +1;
184:       ccid ← num_cc;
185:     end if
186:   end for
187: end procedure
```

Algorithm 14 Finds all SCCs after AddEdge. Iterate only the vertices and edges which are affected and the logic is based on limited version of Tarjan's algorithm.

```
188: procedure MERGESCC (currv ↓, Stk ↓, Root ↓, Comp ↓, visited ↓)
189:   visited[currv.val] ← true;
190:   Root[currv.val] ← currv.val;
191:   Comp[currv.val] ← -1;
192:   Stk.push(currv)
193:   for it ← currv.enext.enext to it.enext ≠ NULL do
194:    if (it.val > 0) then
195:     if (visited[it.val] = false) then
196:      if (locateSCC (it.val ↓, predc ↑, currc ↑, predv ↑, currv ↑)) then //Algorithm 3
197:       mergeSCC (currv ↓, Stk ↓, Root ↓, Comp ↓, visited ↓); //Algorithm 14
198:      end if
199:     end if
200:     if (Comp[it.val] = -1) then
201:      Root[currv.val] ← Root[currv.val] < Root[it.val] ? Root[currv.val] : Root[it.val]
202:     end if
203:    end if
204:   end for
205:   if (Root[currv.val] = currv.val)) then
206:    newcc ← createNewSCCwithOldV (predv ↓, currv ↓); //Algorithm 4
207:    ccCount ← ccCount + 1; // atomic increment(FAA(1))
208:    repeat
209:     w ← Stk.pop();
210:     addOldVInSCC (newcc ↓, w ↓); //Algorithm 6
211:    until (w.val ≠ currv.val)
212:   end if
213: end procedure
```

Algorithm 15 Adds both incoming and outgoing edges to the edge list of vertex key_1 ,if it is not present earlier and then update the affected SCCs.

```
214: procedure bool ADDEDGE (key1 ↓,key2 ↓)
215:   flag1 ← locateSCC (key1 ↓, predc1 ↑, currc1 ↑, predv1 ↑, currv1 ↑); //Algorithm 3
216:   flag2 ← locateSCC (key2 ↓, predc2 ↑, currc2 ↑, predv2 ↑, currv2 ↑); //Algorithm 3
217:   if ( flag1 = false ∨ flag2 = false) then
218:    return false;
219:   end if
220:   flag1 ← locateSCC (key1 ↓, predc1 ↑, currc1 ↑, predv1 ↑, currv1 ↑); //Algorithm 3
221:   if (flag1 = false) then
222:    return false;
223:   end if
224:   flag ← addENode (curr1.enext, key2) ∧ addENode (curr2.enext, (−1) ∗ key1); //Algorithm 7
225:   if (flag = true) then
226:    if (currc1.ccno = currc2.ccno) then
227:     return true;
228:    else
229:     if (!isReachable(currc2,currc1)) then // Checks reachable path from currc2 to currc1
230:      return true;
231:     else
232:      if (findSCCafterAddE(currc2 ↓, currc1 ↓, n ↓)) then //Algorithm 12
233:       return true;
234:      else
235:       return false;
236:      end if
237:     end if
238:    end if
239:   end if
240: end procedure
```

Algorithm 16 Removes both incoming and outgoing edges from the edge list of vertex key_1,if it is present and then update the affected SCCs.

```
241: procedure bool RemoveEdge (key1 ↓,key2 ↓)
242:  flag1 ← locateSCC (key1 ↓, predc1 ↑, currc1 ↑, predv1 ↑, currv1 ↑); //Algorithm 3
243:  flag2 ← locateSCC (key2 ↓, predc2 ↑, currc2 ↑, predv2 ↑, currv2 ↑); //Algorithm 3
244:  if ( flag1 = false ∨ flag2 = false) then
245:   return false;
246:  end if
247:  flag1 ← locateSCC (key1 ↓, predc1 ↑, currc1 ↑, predv1 ↑, currv1 ↑); //Algorithm 3
248:  if (flag1 = false) then
249:   return false;
250:  end if
251:  flag ← remENode (curr1.enext, key2) ∧ remENode (curr2.enext, (−1) ∗ key1); //Algorithm 8
252:  if (flag = true) then
253:   if (currc1.ccno ≠ currc2.ccno) then
254:    return true;
255:   else
256:    if (findSCCafterRemoveE(currc1 ↓, n ↓)) then //Algorithm 13
257:     return true;
258:    else
259:     return false;
260:    end if
261:   end if
262:  end if
263: end procedure
```

Algorithm 17 Removes the vertex along its incoming and outgoing edges from SCC-Graph,if it is present else returns false

```
264: procedure bool remVNode (key ↓, cc ↑)
265:  flag ← locateSCC (key ↓, predc ↑, currc ↑, predv ↑, currv ↑); //Algorithm 3
266:  if (flag = false) then
267:   return false;
268:  else
269:   predv.lock(); currv.lock();
270:   if ValidateE (predv↓, currv↓)) then
271:    currv.marked ← true; // logical deletion
272:    predv.enext ← currv.enext; // physical deletion
273:    cc ← currc;
274:    currv.unlock(); predv.unlock();
275:    return true;
276:   else
277:    currv.unlock(); predv.unlock();
278:    return false;
279:   end if
280:  end if
281: end procedure
```

Algorithm 18 update the SCC-Graph after successful remVNode

```
282: procedure bool RemoveVertex (key ↓)
283:  flag ← remVNode (key ↓, cc ↑); //Algorithm 17
284:  if (flag = false) then
285:   return false;
286:  else
287:   return findSCCafterRemoveE(cc ↓, cc.ccno) ↓)) //Algorithm 13;
288:  end if
289: end procedure
```

Algorithm 19 It inserts a newcc with new vertex to SCC-Graph at the CCHead position

```
290: procedure bool addSCC (key ↓)
291:   predc ← CCHead;
292:   currc ← predc.next;
293:   newcc ← createSCC (key↓);  //Algorithm 21;
294:   predc.lock(); currc.lock();
295:   predv.lock(); currv.lock();
296:   if (ValidateC (predc↓, currc↓) ∧ ValidateV (predv↓, currv↓)) then
297:    predv.vnext ← currv.vnext
298:    newcc.next ← currc; // logical insertion
299:    predc.next ← newcc; // physical insertion
300:    currv.unlock(); predv.unlock();
301:    currc.unlock(); predc.unlock();
302:    ccCount ← ccCount + 1; // atomic increment (FAA(1))
303:    return true;
304:   else
305:    currv.unlock(); predv.unlock();
306:    currc.unlock(); predc.unlock();
307:    return false;
308:   end if
309: end procedure
```

Algorithm 20 Adds a new vertex to the SCC-Graph

```
310: procedure bool AddVertex (key ↓)
311:   return addSCC (key ↓); // Algorithm 19
312: end procedure
```

Algorithm 21 Initialize a new SCC

```
313: procedure cclist_t createSCC (key ↓)
314:   VHead.val ← INT_MIN;
315:   VHead.vnext ← NULL;
316:   VHead.enext ← NULL;
317:   VHead.marked ← false;
318:   VTail.val ← INT_MAX;
319:   VTail.vnext ← NULL;
320:   VTail.enext ← NULL;
321:   VTail.marked ← false;
322:   EHead.val ← INT_MIN;
323:   EHead.vnext ← NULL;
324:   EHead.enext ← NULL;
325:   EHead.marked ← false;
326:   ETail.val ← INT_MAX;
327:   ETail.vnext ← NULL;
328:   ETail.enext ← NULL;
329:   ETail.marked ← false;
330:   EHead.enext ← ETail;
331:   newv.val ← key;
332:   newv.vnext ← VTail;
333:   newv.enext ← EHead;
334:   newv.marked ← false;
335:   newcc.vnext ← newv;
336:   newcc.next ← NULL;
337:   newcc.ccno ← ccid;
338:   ccid ← ccid + 1;
339:   return newcc;
340: end procedure
```

Algorithm 22 Removes all empty SCCs from SCC-Graph, i.e., SCC having empty vertex

```
341: procedure bool removeSCC ()
342:   predc ← CCHead;
343:   currc ← predc.next;
344:   while (currc ≠ CCTail) do
345:     if (currc.vnext.vnext.vnext = NULL) then
346:       predc.lock();
347:       currc.lock();
348:       if (ValidateC (predc↓, currc↓) ) then
349:         currc.marked ← true; // logical deletion
350:         predc.next ← currc; // physical deletion
351:         currc.unlock();
352:         predc.unlock();
353:         ccCount ← ccCount - 1 ; // atomic increment (FAA(-1))
354:         continue;// goto Line 344;
355:       else
356:         currc.unlock();
357:         predc.unlock();
358:         predc ← currc;
359:         currc ← currc.next;
360:       end if
361:     else
362:       predc ← currc;
363:       currc ← currc.next;
364:     end if
365:   end while
366: end procedure
```

Algorithm 23 Checks whether two ids are in the same strongly connected component at a given instance.

```
367: procedure bool checkSCC(key1 ↓, key2 ↓)
368:   flag1 ← locateSCC (key1 ↓, predc1 ↑, currc1 ↑, predv1 ↑, currv1 ↑); //Algorithm 3
369:   flag2 ← locateSCC (key2 ↓, predc2 ↑, currc2 ↑, predv2 ↑, currv2 ↑); //Algorithm 3
370:   if ( flag1 = false ∨ flag2 = false) then
371:     return false;
372:   end if
373:   flag1 ← locateSCC (key1 ↓, predc1 ↑, currc1 ↑, predv1 ↑, currv1 ↑); //Algorithm 3
374:   if (flag1 = false) then
375:     return false;
376:   else
377:     locateE (key2 ↓, currv1.enext ↓, prede ↑, curre ↑); //Algorithm 5
378:     if (curre.val = key2 ∧ curre.marked = false) then
379:       return true;
380:     else
381:       return false;
382:     end if
383:   end if
384: end procedure
```

Algorithm 24 Checks id belongs to which SCC, it returns the *ccno* of an SCC,

```
385: procedure bool blongsToCommunity(key ↓)
386:   flag ← locateSCC (key ↓, predc ↑, currc ↑, predv ↑, currv ↑); //Algorithm 3
387:   if (flag = true ∧ currv.marked = false) then
388:     return true;
389:   else
390:     return false;
391:   end if
392: end procedure
```

6 The Correctness Proof

We now describe how our proposed algorithm SMSCC is correct. A full and detail proof is available in the full paper and it is based on Timnat. et.al.'s, full paper [15]. We think the detail proof is very much important for concurrent data-structure and algorithms as without that, it is very hard to understand the races. Any directed graph is represented as SCC-Graph and it is collection of three types of lists. First, the *SCC-list*, each SCC is a node in the SCC-list. Secondly, *Vertex-list*, each SCC has a vertex set, stoored in the vertex list and finally, *Edge-list*, each vertex has its adjacency edge list. The SCC-Graph is interfaced with node id or key value `val`, boolean `marked` filed and `next` field. At any instance of time a node is considered to be part of SCC-Graph, if it is unmarked.

Proof Methodology We define the abstract SCC-Graph which always holds two invariant. Once the invariant holds for a node, it remain true. The first invariant is that, the node(SCC or vertex or edge) can only physically change by pointer(`next` or `vnext` or `enext`) and the key value of the node never change after initialization. Second, once a node is marked, it remain to be marked and it's next pointer never change until GC. For proving the correctness we use the four stages of any node similar like Timnat. et.al.'s, [15]. *Logical remove*: changing the `marked` filed false to true. *Physical remove*: delinking the node from the list. *Logical insertion*: Connecting new node's pointer to the node list. *Physical Insertion*: making new logical node to a physical node, i.e. actual insertion. We prove our algorithm using mathematical induction.

Lemma 1. *The history H generated by the interleaving of any of the methods of the SCC-Graph, is linearizable.*
Proof in the Appendix.

Lemma 2. *The methods $AddVertex$, $RemoveVertex$, $AddEdge$ and $RemoveEdge$ are deadlock-free.*

Proof Sketch: We prove all the $AddVertex$, $RemoveVertex$, $AddEdge$ and $RemoveEdge$ methods are deadlock-free by direct argument based of the acquiring lock on both the current and predecessor nodes.

1. $AddVertex$: the $AddVertex(key)$ method is deadlock-free because a thread always acquires lock on the *vnode* with smaller keys first. Which means, if a thread say T_1 acquired a lock on a $vnode(key)$, it never tries to acquire a lock on a *vnode* with key smaller than or equal to $vnode(key)$. This is true because the $AddVertex$ method acquires lock on the predecessor *vnode* from the $LocateVertex$ method.
2. $RemoveVertex$: the $RemoveVertex(key)$ method is also deadlock-free, similar argument as $AddVertex$.
3. $AddEdge$: the $AddEdge(key_1, hey_2)$ method is deadlock-free because a thread always acquires lock on the *enode* with smaller keys first. Which means, if a thread say T_1 acquired a lock on a $enode(key_2)$, it never tries to acquire a

lock on a *enode* of the vertex $vnode(key_1)$ with key smaller than or equal to $enode(key_2)$. This is true because the *AddEdge* method acquires lock on the predecessor edge nodes of the vertex $vnode(key_1)$ from the *locateE* method.

4. *RemoveEdge*: the $RemoveEdge(key_1, key_2)$ method is also deadlock-free, similar argument as *AddEdge*.

□

Lemma 3. *The methods checkSCC and blongsToCommunity are wait-free.*

Proof. The $blongsToCommunity(key)$ method scans the vertex list of the graph starting from the $VertexHead$, ignoring whether *vnode* are marked or not. It returns a boolean $flag$ either $true$ or $false$ depending on $vnode(key)$ greater than or equal to the sought-after key. If the desired *vnode* is unmarked, it simply returns $true$ and this is correct because the vertex list is sorted. On the other hand, it returns $false$ if $vnode(key)$ is not present or has been marked. This $blongsToCommunity$ method is wait-free, because there are only a finite number of vertex keys that are smaller than the one being searched for. By the observation of the code, a new *vnode* with lower or equal keys is never added ahead of it, hence they are reachable from $VertexHead$ even if vertex nodes are logically removed from the vertex list. Therefore, each time the $blongsToCommunity$ moves to a new vertex node, whose key value is larger key than the previous one. This can happen only finitely many times, which says the traversal of $blongsToCommunity$ method is wait-free.

Similarly, the $checkSCC(key_1, key_2)$ method first scans the vertex list of the graph starting from the $VertexHead$, ignoring whether vertex nodes are marked or not. It returns a boolean $flag$ either $true$ or $false$ depending on $enode(key_2)$ greater than or equal to the sought-after key in the edge list of the vertex $vnode(key_1)$. If the desired *enode* is unmarked, it simply returns $true$ and this is correct because the vertex list is sorted as well as the edge list of the vertex $vnode(key_1)$ is also sorted. On the other hand it returns $false$ if either $vnode(key_1)$ or $vnode(key_2)$ is not present or has been marked in the vertex list or $enode(key_2)$ is not present or has been marked in the edge list of the vertex $vnode(key_1)$. This $checkSCC$ method is wait-free, because there are only a finite number of vertex keys that are smaller than the one being searched for as well as a finite number of edge keys that are smaller than the one being searched for in edge list of any vertex. By observation of the code, a new *enode* with lower or equal keys is never added ahead of $enode(key_2)$ in the edge list of the vertex $vnode(key_1)$, hence they are reachable from $VertexHead$ even if vertex nodes or edge nodes of $vnode(key_1)$ are logically removed from the vertex list. Therefore, each time the $ContainsEdge$ moves to a new edge node, whose key value is larger key than the previous one. This can happen only finitely many times, which says the traversal of $checkSCC$ method is wait-free.

6.1 Linearization Points

In this section we identify the linearization point(LP) of our proposed methods. Before identifying the LP, we first consider locateSCC, as it is used by most of

the methods. It returns true if key value is present along with pair of SCC pointers(`predc` & `currc`) and pair of vertex pointers (`predv` & `currv`). For successful *locateSCC* return the LP be Line 19 where key is found and for unsuccessful the LP be Line 26 last read of `currc.next`.
The LP of a successful *AddEdge* with no successful concurrent *RemoveVertex*, is either in the Line 39 when a new SCC is created with the first old vertex, it is the physical insertion `predc.next` ← `newcc` at the `CCHead` position. Or the LP be in the Line 65 `predv1.vnext` ← `currv` physical insertion of old vertex to `newcc`. If there is a successful concurrent removal of either one of the vertex or both, we linearized just before the LP of the first successful concurrent RemoveVertex. For unsuccessful *AddEdge*, the LP is inside *locateSCC* method where either of the vertex is not found in the Line 26 the last read of `currc.next` or inside the *locateE* method in the Line 55, the last read of *curre.enext* inside the while loop, if the edge node is present.

Similarly, the LP of a successful *RemoveEdge* with no successful concurrent *RemoveVertex* is same as successful *AddEdge* with no successful concurrent *RemoveVertex*. If there is a successful concurrent removal of either one of the vertex or both, we linearized just before the LP of the first successful concurrent RemoveVertex. For unsuccessful *AddEdge*, the LP is inside *locateSCC* method where either of the vertex key is not found in the Line 26 the last read of `currc.next`, or inside the *locateE* method in the Line 55, the last read of *curre.enext* inside the while loop, where the edge node is not present.

7 Performance Analysis

In this section, we evaluate the performance of our SMSCC algorithm. The source code available at https://github.com/Mukti0123/SMSCC. It contains both fully and partial dynamic SCC with & without deletion of incoming edges(DIE) and some applications, such as community detection. We compare throughput with sequential and coarse-grain.

The methods are evaluated on a dual-socket, 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 to that core; L3 cache (25MB) is shared across all cores of a processors. All the codes are compiled using the GCC C/C++ compiler (version 5.4.0) with -O3 optimization and Posix threads execution model.
Workload & methodology: we ran each experiment for 20 seconds, and measured the overall number of operations executed by all the threads(starting from 1, 10, 20 to 60). The graphs shown in the Fig 4 & 5 are the total number of operations executed by all threads. In all the tests, we ran each evaluation 8 times and took the average.
The algorithms we compare are, (1). Sequential(only one thread and no lock) with partial (without removing vertices, *Seq-woDV*) and fully(*Seq*) dynamic, (2). Coarse-grained(only one spin lock) with partial (without removing vertices, *Coarse-woDV*) and fully(*Coarse*) dynamic, (3). SMSCC with partial (without

removing vertices, *SMSCC-woDV*) and fully(*SMSCC*) dynamic(with and without DIE). Each thread performed, in the Fig 4a, 50% add(V+E) and 50% rem(V+E), in the Fig 4b, 90% add(V+E) and 10% rem(V+E) and 10% add(V+E) and in the Fig 4c, 90% rem(V+E). The Fig 5 shows the throughputs of *Seq*, *Coarse*, *SMSCC*(without DIEs) and *SMSCC-DIE*(with DIEs). Similarly, the Fig 5a and 5b depicts the incremental(SMISCC) & decremental(SMDSCC) throughputs respectively and Fig 5c shows the community detection(80% check and 20% add & rem).

After executing all above micro benchmarks, SMSCC (with & without DIEs) perform efficiently over Sep and Coarse. The Fig 4 and 5 shows the performance is similar to the lazy linked list and the throughput is increased between 3 to 7X depending on different workload distributions and applications.

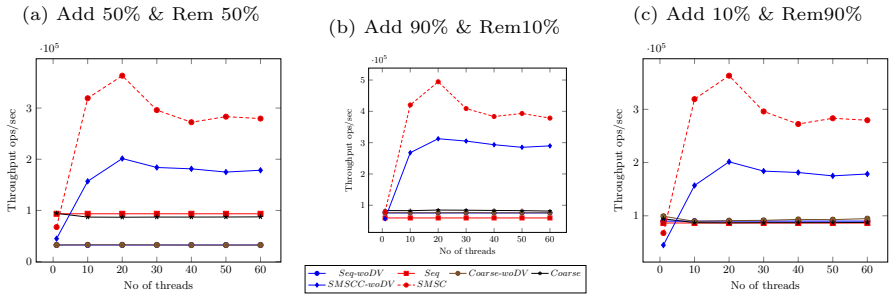

Fig. 4: SMSCC Execution with different workload distributions

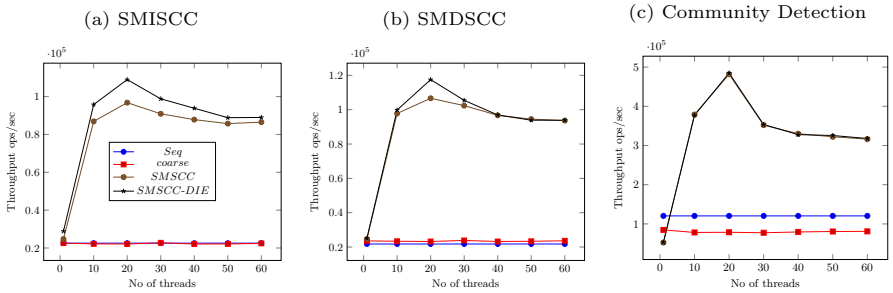

Fig. 5: (a)Incremental SCC(100% Add(V+E)), (b)Decremental SCC(100% Rem(V+E) and (c). Community detection(checking 80% + update 20%

8 Conclusion & Future Direction

In this paper, we proposed a full dynamic algorithm(SMSCC) for maintaining strongly connected component of a directed graph in a shared memory architecture. The edges/vertices are added or deleted concurrently by fixed number of threads. To the the best of our knowledge, this is the first work to propose using linearizable concurrent data-structure. We have constructed SCC-Graph using

three type of nodes, `SCC node`, `vertex & edge node`, which were build using list-based set, first one is unordered and later two are ordered list. We provide an empirical comparison against sequential, course-grained, with different workload distributions. Also we compare the result with delete & without delete incoming edges. We concluded that the performance show in the Fig. 4 is similar to the lazy linked list and the throughput is increased between 3 to 7X. Also In Fig 5 depictes one application of SCC-Graph to identify community in a random graph. We believe that there are huge applications in the on-line graph.

Currently the proposed update algorithms are blocking and deadlock-free. In the future, we plan to explore non-blocking(lock-free & wait-free) variant of all the methods of SCC-Graph. We believe that one can develop a better optimization techniques to handle the SCC restoring after the edges/vertices are added or deleted. Also we plan for other real world social graph applications.

References

1. David A. Bader, Jonathan W. Berry, Daniel Chavarría-Miranda, Kamesh Madduri, and Steven C. Poulos. Stinger: Spatio-temporal interaction networks and graphs (sting) extensible representation. 2009.
2. Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. A New Approach to Incremental Cycle Detection and Related Problems. *ACM Trans. Algorithms*, 12(2):14, 2016. URL: http://doi.acm.org/10.1145/2756553, doi:10.1145/2756553.
3. Vincent Bloemen, Alfons Laarman, and Jaco van de Pol. Multi-core on-the-fly scc decomposition. *SIGPLAN Not.*, 51(8):8:1–8:12, February 2016. URL: http://doi.acm.org/10.1145/3016078.2851161, doi:10.1145/3016078.2851161.
4. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009. URL: http://mitpress.mit.edu/books/introduction-algorithms.
5. Camil Demetrescu, David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Algorithms and theory of computation handbook. chapter Dynamic Graph Algorithms, pages 9–9. Chapman & Hall/CRC, 2010. URL: http://dl.acm.org/citation.cfm?id=1882757.1882766.
6. Oded Green and David A. Bader. custinger: Supporting dynamic graph algorithms for gpus. *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2016.
7. Monika R. Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, July 1999. URL: http://doi.acm.org/10.1145/320211.320215, doi:10.1145/320211.320215.
8. 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.
9. 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.
10. D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Commun. ACM*, 22(8):461–464, August 1979. URL: http://doi.acm.org/10.1145/359138.359141, doi:10.1145/359138.359141.

11. John Hopcroft and Robert Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, June 1973. URL: http://doi.acm.org/10.1145/362248.362272, doi:10.1145/362248.362272.
12. Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004. URL: http://dx.doi.org/10.1109/TPDS.2004.8, doi:10.1109/TPDS.2004.8.
13. Yossi Shiloach and Shimon Even. An on-line edge-deletion problem. *J. ACM*, 28(1):1–4, January 1981. URL: http://doi.acm.org/10.1145/322234.322235, doi:10.1145/322234.322235.
14. Yossi Shiloach and Uzi Vishkin. An o(logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57 – 67, 1982. URL: http://www.sciencedirect.com/science/article/pii/0196677482900086, doi:https://doi.org/10.1016/0196-6774(82)90008-6.
15. Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In *Principles of Distributed Systems, 16th International Conference, OPODIS 2012, Rome, Italy, December 18-20, 2012. Proceedings*, pages 330–344, 2012. URL: https://doi.org/10.1007/978-3-642-35476-2_23, doi:10.1007/978-3-642-35476-2_23.
16. A Gitter Z Bar Joseph and I Simon. Studying and modelling dynamic biological processes using time-series gene expression data. In *Nature Publishing Group, a division of Macmillan Publishers Limited. All Rights Reserved.*, pages 552–564, 2012. URL: https://www.ncbi.nlm.nih.gov/pubmed/22805708.