An Efficient Practical Concurrent Wait-Free Unbounded Graph

Sathya Peri[1], Chandra Kiran Reddy[2], Muktikanta Sa[3]
Department of Computer Science & Engineering
Indian Institute of Technology Hyderabad, India
{[1]sathya_p, [2]cs15btech11012, [3]cs15resch11012}@iith.ac.in

***Abstract*—In this paper, we propose an efficient concurrent wait-free algorithm to construct an unbounded directed graph for shared memory architecture. To the best of our knowledge that this is the first wait-free algorithm for an unbounded directed graph where insertion and deletion of vertices and/or edges can happen concurrently. To achieve wait-freedom in dynamic setting, threads help each other to perform the desired takss using operator descriptors by other threads. We also prove that all graph operations are wait-free and linearizable. We implemented our algorithms in C++ and tested its performance through several micro-benchmarks. Our experimental results show an average of 9x improvement over the global lock-based implementation.**

***Index Terms*—concurrent data-structure, lazy-list, directed graph, locks, lock-free, wait-free, fast-path-slow-path**

I. Introduction

Graphs are very useful structures that have wide variety of applications. They are usually represented as pairwise relationships among objects, with the objects as vertices and relationships as edges. They are applicable in various research areas such as social networking (facebook, twitter etc.), semantic, data mining, image processing, VLSI design, road network, graphics, blockchains and many more.

In many of these application, the graphs are very *large* and *dynamic* in nature, that is, they undergo changes over time like addition and removal of vertices and/or edges [2]. Hence, to precisely model these applications, we need an efficient data-structure which supports dynamic changes and can expand at run-time depending on the availability of memory in the machine.

Nowadays, with multicore systems becoming ubiquitous *concurrent data-structures (CDS)* [7] have become popular. CDS such as concurrent stacks, queues, hash-tables etc. allow multiple threads to operate on them concurrently while maintaining correctness, *linearizability* [8]. These structures can efficiently harness the power multi-core systems. Thus, a multi-threaded concurrent unbounded graph data-structure can effectively model dynamic graphs as described above.

The correctness-criterion for CDS is linearizability [8] which ensures that the affect of every method seems to take place somewhere at some atomic step between the invocation and response of the method. The atomic step is referred to as *linearization point (LP)*. Coming to progress conditions, a method of a CDS is *wait-free* [6], [7] if it ensures that the method finishes its execution in a finite number of steps. A *lock-free* CDS ensures, at least one of its methods is guaranteed to complete in a finite number of steps. A lock-free algorithm never enter deadlocks but can possibly starve. On the other hand, wait-free algorithms are starvation-free. In many of the wait-free and lock-free algorithms proposed in the literature, threads help each other to achieve the desired tasks.

Concurrent graph data-structures have not been explored in detail in the literature with some work appearing recently [1], [9]. Applications relying on graphs mostly use a sequential implementation while some parallel implementations synchronize using global locks which causes severe performance bottlenecks.

In this paper, we describe an efficient practical concurrent wait-free unbounded directed graph (for shared memory system) which supports concurrent insertion/deletion of vertices and edges while ensuring linearizability [8]. The algorithm for wait-free concurrent graph data-structure is based on the non-blocking graph by Chatterjee et al. [1] and wait-free algorithm proposed by Timnat et al. [14]. Our implementation is not a straightforward extension to lock-free/wait-free list implementation but has several non-trivial key supplements. This can be seen from the LPs of edge methods which in many cases lie outside their method and depend on other graph operations running concurrently. We believe the design of the graph data-structure is such that it can help to identify other useful properties on a graph such as reachability, cycle detection, shortest path, betweenness centrality, diameter, etc.

To enhance performance, we also developed an optimized wait-free graph based on the principle of fast-path-slow-path [11]. The basic idea is that the lock-free algorithms are fast as compare to the wait-free algorithms in practice. So, instead of always executing in wait-free manner (slow-path), threads normally execute methods in lock-free manner (fast-path). If a thread executing a method in lock-free manner, fails to complete in certain threshold number of iterations, switches to wait-free execution and eventually terminates.

A. Contributions

In this paper, we present an efficient practical concurrent wait-free unbounded directed graph data-structure. The main contributions of our work are summarized below:

1) We describe an Abstract Data Type (ADT) that maintains a wait-free directed graph $G = (V, E)$. It comprises of the following methods on the sets V and E: (1) Add Vertex:

WFADDV (2) Remove Vertex: WFREMV, (3) Contains Vertex: WFCONV (4) Add Edge: WFADDE (5) Remove Edge: WFREME and (6) Contains Edge: WFCONE. The wait-free graph is represented as an adjacency list similar in [1] (Section III).

2) We implemented the directed graph in a dynamic setting with threads helping each other using operator descriptors to achieve wait-freedom (Section IV).
3) We also extended the wait-free graph to enhance the performance and achieve a fast wait-free graph based on the principle of fast-path-slow-path proposed by Kogan et al. [11] (Section V).
4) Formally, we prove for the correctness by showing the operations of the concurrent graph data-structure are linearizable [8]. We also prove the wait-free progress guarantee of the operations WFADDV, WFREMV, WFCONV, WFADDE, WFREME, and WFCONE (Section VI).
5) We evaluated the wait-free algorithms in C++ implementation and tested through several micro-benchmarks. Our experimental results show on an average of 9x improvement over the sequential and global lock implementation (Section VII).

B. Related Work

Kallimanis and Kanellou [9] presented a concurrent graph that supports wait-free edge updates and traversals. They represented the graph using adjacency matrix, with an upper bound on number of vertices. As a result, their graph data-structure does not allow any insertion or deletion of vertices after initialization of the graph. Although this might be useful for some applications such as road networks, this may not be adequate for many real-world applications which need dynamic modifications of vertices as well as unbounded graph size.

A recent work by Chatterjee et al. [1] proposed a non-blocking concurrent graph data-structure which allowed multiple threads to perform dynamic insertion and deletion of vertices and/or edges in lock-free manner. Our paper extends their data-structure while ensuring that all the graph operations are wait-free.

II. SYSTEM MODEL

The Memory Model. We consider an asynchronous shared-memory model with a finite set of p processors accessed by a finite set of n threads. The threads communicate with each other by invoking atomic operations on the shared objects such as atomic `read`, `write`, `fetch-and-add` (FAA) and `compare-and-swap` (CAS) instructions.

An $\texttt{FAA}(x, a)$ instruction atomically increments the value at the memory location x by the value a. Similarly, a $\texttt{CAS}(x, a, a')$ is an atomic instruction that checks if the current value at a memory location x is equivalent to the given value a, and only if true, changes the value of x to the new value a' and returns `true`; otherwise the memory location remains unchanged and the instruction returns `false`. Such a system can be perfectly realized by a Non-Uniform Memory Access (NUMA) computer with one or more multi-processor CPUs.

```
class VNode {
   int vkey; // immutable key field
   VNode vnext; // atomic refe., pointer to the next VNode
   ENode enext; // atomic ref., pointer to the edge-list
}
class ENode {
   int ekey; // immutable key field
   VNode pointv; // pointer from the ENode to its VNode.
   ENode enext; // atomic ref., pointer to the next ENode
}
class ODA {
   unsigned long phase; // phase number of each operation.
   opType  type; // type of the operation.
   VNode   vnode; // pointer to the VNode.
   ENode   enode; // pointer to the ENode.
   VNode   vsrc, vdest; // pointer to the source and destination VNode
}

VNode VertexHead, VertexTail; // Sentinel nodes for the vertex-list
unsigned long maxph; // atomic variable which keeps track of op. number.
ODA state []; // global state array for posting operations, array size is
                same as number of threads
```

Fig. 1: Structure of `ENode`, `VNode` and `ODA`.

Correctness. We consider *linearizability* introduced by Herlihy & Wing [8] as the correctness criterion for the graph operations. We assume that the execution generated by a data-structure is a collection of method invocation and response events. Each invocation of a method call has a subsequent response. An execution 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 execution in which the methods are ordered by their LPs [8].

Progress. The progress properties specify when a thread invoking operations on the shared memory objects completes in the presence of other concurrent threads. In this context, we provide the graph implementation with methods that satisfy wait-freedom, based on the definitions in Herlihy and Shavit [6]. A method of a CDS is wait-free if it completes in finite number of steps. A data-structure implementation is wait-free if all its methods are wait-free. This ensures per-thread progress and is the most reliable non-blocking progress guarantee in a concurrent system. A data-structure is lock-free if its methods get invoked by multiple concurrent threads, then one of them will complete in finite number of steps.

III. THE UNDERLYING GRAPH DATA-STRUCTURE

In this section, we give a detailed construction of the graph data-structure, which is a combination of non-blocking graph based on [1] and wait-free construction based on [14], [15]. We represent the concurrent directed graph as an adjacency list representations. Hence, it is constructed as a collection set of vertices stored as linked-list manner wherein each vertex also holds a list of neighboring vertices which it has outgoing edges.

The VNode, ENode and ODA structures depicted in Figure 1. The VNode consists of two atomic pointers vnext and enext and an immutable key vkey. The vnext points to the next VNode in the vertex-list, whereas, enext points to the head of the edge-list which is the list of outgoing neighboring vertices. Similarly, an ENode also has an atomic pointer enext points to the next ENode in the edge-list and a pointer pointv points to the corresponding VNode. Which helps direct access to its VNode while doing any traversal like BFS, DFS and also helps to delete the incoming edges, detail regarding this described in Section IV. We assume that all the vertices in the vertex-list have unique identification key which captured by vkey field. Similarly, all the edge nodes for a vertex in the edge-list have unique identification key which captured by ekey field.

Besides VNodes and ENodes, we also have an state array with an operation-descriptor(ODA) for each thread. Each thread's state entry recounts its current state. An ODA composed of six fields: a phase number phase, an operation type type indicates the current operation executed by this thread, possible operation types given at the end of this section, a pointer to the vertex node vnode, which used for any vertex operations, a pointer to the edge node enode, which is used for any edge operations, and a pair of VNodes pointers vsrc and vdest, used for any edge operations to store the source and destination VNodes of an edge.

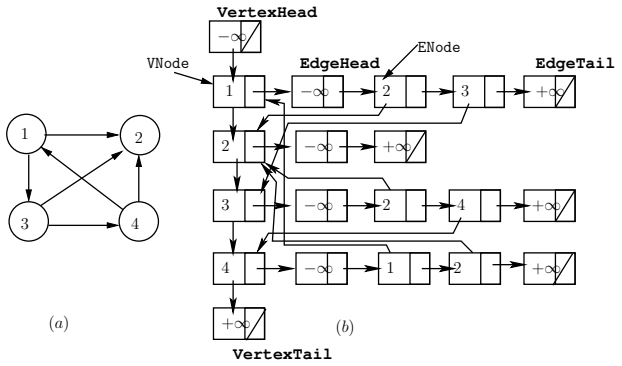

Fig. 2: (a) A directed Graph (b) The wait-free graph representation for (a).

Our wait-free concurrent directed graph data-structure supports six major operations: WFAddV, WFRemV, WFConV, WFAddE, WFRemE, and WFConE. We use the helping mechanism like [10], [14], [15] to achieve the wait-freedom for all our graph operations. Before begin to execute an operation, a thread starts invoking a special state array state similar as Timnat et al. [14]. This state shared among the threads. All threads can see the details of the operation they are running during their execution. Whenever an operation starts execution it publishes its operation in the state, then all other threads try to help to finish its execution. When the operation finished its execution, the outcome result is also announced to the state, using a CAS, which substitutes the old existing type to the new one.

We initialize the vertex-list with dummy head(VertexHead) and tail(VertexTail) (called sentinels) with values -∞ and ∞ respectively. Similarly, each edge-lists is also initialized with a dummy head(EdgeHead) and tail(EdgeTail)(see Figure 2).

Our wait-free graph data-structure maintains some *invariants*: (a) the vertex-list is sorted based on the VNode's key value vkey and each unmarked VNode is reachable from the VertexHead, and (b) also each edge-lists are sorted based on the ENode's key value ekey and unmarked ENodes are reachable from the EdgeHead of the corresponding VNode.

A. The Abstract Data Type(ADT)

A wait-free graph is defined as a directed graph $G = (V, E)$, where V is the set of vertices and E is the set of directed edges. Each edge in E is an ordered pair of vertices belonging to V. A vertex $v \in V$ has an immutable unique key vkey denoted by $v(\texttt{vkey})$. A directed edge from the vertex $v(\texttt{ekey}_1)$ to $v(\texttt{ekey}_2)$ is denoted as $e(v(\texttt{ekey}_1), v(\texttt{ekey}_2)) \in E$. For simplicity, we denote $e(v(\texttt{ekey}_1), v(\texttt{ekey}_2))$ as $e(\texttt{ekey}_2)$, which means the $v(\texttt{ekey}_1)$ has a neighbouring vertex $v(\texttt{ekey}_2)$. We also defined an ADT for operations on G which are exported by the wait-free graph data-structure, are given bellow:

1) The WFAddV(vkey) operation adds a vertex vkey to the graph, only if $v(\texttt{vkey}) \notin V$ and then returns true, otherwise it returns false.
2) The WFRemV(vkey) operation deletes a vertex $v(\texttt{vkey})$ from V, only if $v(\texttt{vkey}) \in V$ and then returns true, otherwise it returns false. Once a vertex $v(\texttt{vkey})$ is deleted successfully all its outgoing and incoming edges also removed.
3) The WFConV(vkey) operation returns true, if $v(\texttt{vkey}) \in V$; otherwise returns false.
4) The WFAddE($\texttt{ekey}_1, \texttt{ekey}_2$) adds an edge $e(v(\texttt{ekey}_1), v(\texttt{ekey}_2))$ to E, only if $e(v(\texttt{ekey}_1), v(\texttt{ekey}_2)) \notin E$ and $v(\texttt{ekey}_1) \in V$ and $v(\texttt{ekey}_2) \in V$ then it returns EDGE ADDED. If $v(\texttt{ekey}_1) \notin V$ or $v(\texttt{ekey}_2) \notin V$, it returns VERTEX NOT PRESENT. If $e(v(\texttt{ekey}_1), v(\texttt{ekey}_2)) \in E$, it returns EDGE ALREADY PRESENT.
5) The WFRemE($\texttt{ekey}_1, \texttt{ekey}_2$) deletes the edge $e(v(\texttt{ekey}_1), v(\texttt{ekey}_2))$ from E, only if $e(v(\texttt{ekey}_1), v(\texttt{ekey}_2)) \in E$ and $v(\texttt{ekey}_1) \in V$ and $v(\texttt{ekey}_2) \in V$ then it returns EDGE REMOVED. If $v(\texttt{ekey}_1) \notin V$ or $v(\texttt{ekey}_2) \notin V$, it returns VERTEX NOT PRESENT. If $e(v(\texttt{ekey}_1), v(\texttt{ekey}_2)) \notin E$, it returns EDGE NOT PRESENT.
6) The WFConE($\texttt{ekey}_1, \texttt{ekey}_2$) if $e(v(\texttt{ekey}_1), v(\texttt{ekey}_2)) \in E$ and $v(\texttt{ekey}_1) \in V$ and $v(\texttt{ekey}_2) \in V$ then it returns EDGE PRESENT, otherwise it returns VERTEX OR EDGE NOT PRESENT.
7) The HelpGphDS(phase) operation ensures that each thread completes its own operation and helps in completing all the pending operations with lower phase numbers.

When the operations WFADDV, WFREMV, WFCONV, WFADDE, WFREME, and WFCONE start execution they will get a new phase number and post their operation in the state array and then invoke the HELPGPHDS(phase) (Help Graph data-structure)operation. All the helping methods check if their phase number is the same as the thread's phase number in state, otherwise, they return false.

The possible operation type for the ODA state values are:

1) addvertex: To insert a VNode into the vertex-list and it requested for help.
2) remvertex: To delete a VNode from the vertex-list and it requested for help.
3) findvertex: To find a VNode from the vertex-list and it requested for help.
4) addedge: To insert an ENode into the edge-list of a vertex and it requested for help.
5) remedge: To delete an ENode from the edge-list of a vertex and it requested for help.
6) findedge: To find an ENode from the edge-list of a vertex and it requested for help.
7) success: If any of the graph operation finished its execution successfully.
8) failure: If any of the graph operation unable finished its execution.

The first six states used to ask for help from other threads whereas the last two states thread does not ask for any help from other threads.

IV. THE WAIT-FREE GRAPH ALGORITHM

In this section, we present the technical details of all wait-free graph operations. The design of wait-free graph data-structure based on the adjacency list representation. Hence, it is implemented as a collection (list) of vertices wherein each vertex holds a list of vertices to which it has outgoing edges. The implementation is a linked list of VNode and ENode as shown in Figure 2. The implementation of each of these lists based on the non-blocking graph [1] and wait-free construction based on [14], [15] and non-blocking list concurrent-set [3], [4], [12], [16]. The wait-free graph algorithm depicted in Figure 3, 4, 5, and 6.

Pseudo-code convention: We use $p.x$ to access the member field x of a class object pointer p. To return multiple variables from an operation we use $\langle x_1, x_2, \ldots, x_n \rangle$. To avoid the overhead of another field in the node structure, we use bit-manipulation: last one significant bit of a pointer p. In case of an x86-64 bit architecture, memory has a 64-bit boundary and the last three least significant bits are unused. So, we use the last one significant bit of the pointer. We define three methods ISMRKD(p) return true if last significant bit of pointer p is set to 1, else, it returns false, MRKDRF(p), UNMRKDRF(p) sets last significant bit of the pointer p to 1 and 0 respectively. An invocation of CVNODE(vkey) creates a new VNode with key vkey. Similarly, an invocation of CENODE(ekey) creates a new ENode with key ekey. For a newly created VNode and ENode the pointer fields are initialised with NULL value.

A. The Helping Procedure

When an operation is invoked by a thread to start its execution, at first, it chooses a unique phase number which is the higher than all previously chosen phase numbers by other threads. The main objective of assigning a unique phase number is to help the operations with lower phase number. This means whenever a thread started its execution with a new phase number, it tries to help other unfinished operations whose phase number is lower. Which allows all operations to get help to finish their execution to ensures the starvation-free. The phase selection procedure, in Line 1 to 4, executed by reading the current phase number and then atomically increments the maxph, using an FAA. Once the phase numbers is chosen, the thread publishes the operation in the state array by updating its entry. Then it invokes the HELPGPHDS (in Line 5 to 24) procedure where it traverses through the state array and tries to help the operations whose phase number is lower than or equal to its, which ensures the unfinished operations gets help from other threads to finish the execution. This ensures wait-freedom.

B. The Vertex Methods

The wait-free vertex operations WFADDV, WFREMV, and WFCONV depicted in Figure 3 and their corresponding helping procedures HELPADDV, HELPREMV, and HELPCONV depicted in Figure 4, and 5. If the vertex set keys are finite(up to available memory in the system), we also have an optimized case where the WFCONV neither helps nor accepts any help from other threads. This because to achieve higher throughput, we assume WFCONV is called more frequently than the WFADDV and WFREMV operations, so it does not allow any help. Without being affected by each other, all the vertex operations are wait-free.

A WFADDV(vkey) operation invoked by passing the vkey to be inserted, in Line 25 to 37. A WFADDV operation starts by choosing a phase number, creating a new VNode by invoking CVNODE(vkey), posting its operation on the state array. Then the thread calls HELPGPHDS(phase) to invoke the helping mechanism. After that, it traverses the state array and helps all pending operations and tries to complete its operation. In the next step the same thread(or a helping thread) enters HELPADDV(phase) and verifies the phase number and type of operation opType, if they match with addvertex then it invokes HELPLOCV(phase) to traverse the vertex-list until it finds a vertex with its key greater than or equal to v(vkey). In the process of traversal, it physically deletes all logically deleted VNodes, using a CAS. Once it reaches the appropriate location checks whether the vkey is already present. If the ekey is not present earlier it attempts a CAS to add the new VNode (Line 202). On an unsuccessful CAS, it retries. After the operation completes HELPADDV(phase) success or failure reported to the state array.

In the process of traversal, it is possible that the threads which are helping might have been inserted the v(vkey) but not publish success. Also, it is possible that the v(vkey) we

```
1:  procedure MaxPhase()
2:    maxph.FetchAndAdd(1);
3:    return maxph;
4:  end procedure
5:  Operation HelpGphDS (phase)
6:    for (tid ← 0 to state.end()) do
7:      ODA desc ← state[tid];
8:      if (desc.phase ≤ phase) then
9:        if (desc.type = addvertex) then
10:         HelpAddV(tid, desc.phase);
11:       else if (desc.type = remvertex) then
12:         HelpRemV(tid, desc.phase);
13:       else if (desc.type = addedge) then
14:         HelpAddE(tid, desc.phase);
15:       else if (desc.type = remedge) then
16:         HelpRemE(tid, desc.phase);
17:       else if (desc.type = findvertex) then
18:         WFConV(tid, desc.phase);
19:       else if (desc.type = findedge) then
20:         WFConE(tid, desc.phase);
21:       end if
22:     end if
23:   end for
24: end Operation
25: Operation WFAddV(key)
26:   tid ← ThreadID.get();
27:   phase ← MaxPhase();
28:   nv ← new VNode(key);
29:   ODA op ← new ODA(phase, addvertex, nv);
30:   state[tid] ← op;
31:   HelpGphDS(phase);
32:   if (state[tid].type = success) then
33:     return true;
34:   else
35:     return false;
36:   end if
37: end Operation
38: Operation WFRemV(key)
39:   tid ← ThreadID.get();
40:   phase ← MaxPhase();
41:   nv ← new VNode(key);
42:   ODA op ← new ODA(phase, remvertex, nv);
43:   state[tid] ← op;
44:   HelpGphDS(phase);
45:   if (state[tid].type = success) then
46:     return true;
47:   else
48:     return false;
49:   end if
50: end Operation
51: Operation WFAddE(key1, key2)
52:   tid ← ThreadID.get();
53:   phase ← MaxPhase();
54:   ⟨v1, v2, flag⟩ ← LocateUV(key1, key2);
55:   if (flag = false ⋁ isMrkd(v1) ⋁ isMrkd(v2)) then
56:     return VERTEX NOT PRESENT
57:   end if
58:   ne ← new ENode(key2);
59:   ODA op ← new ODA (phase, addedge, ne, v1, v2);
60:   state[tid] ← op;
61:   HelpGphDS(phase);
62:   if (state[tid].type = success) then
63:     return EDGE ADDED;
64:   else
65:     return EDGE ALREADY PRESENT;
66:   end if
67: end Operation
68: Operation WFRemE(key1, key2)
69:   tid ← ThreadID.get();
70:   phase ← MaxPhase();
71:   ⟨v1, v2, flag⟩ ← LocateUV(key1, key2);
72:   if (flag = false ⋁ isMrkd(v1) ⋁ isMrkd(v2)) then
73:     return VERTEX NOT PRESENT;
74:   end if
75:   ne ← new ENode(key2);
76:   ODA op ← new ODA (phase, remedge, ne, v1, v2);
77:   state[tid] ← op;
78:   HelpGphDS(phase);
79:   if (state[tid].type = success) then
80:     return EDGE REMOVED;
81:   else
82:     return EDGE NOT PRESENT;
83:   end if
84: end Operation
85: Operation WFConV(key)
86:   tid ← ThreadID.get();
87:   phase ← MaxPhase();
88:   nv ← new VNode(key);
89:   ODA op ← new ODA(phase, findvertex, nv);
90:   state[tid] ← op;
91:   HelpGphDS(phase);
92:   if (state[tid].type = success) then
93:     return true;
94:   else
95:     return false;
96:   end if
97: end Operation
```

Fig. 3: Pseudo-codes of WFAddV, WFRemV, WFAddE, WFRemE and WFConV.

are trying to insert was already inserted and then removed by some other thread and then a different VNode with vkey was inserted to the vertex-list. We properly handled these cases.

To identify these cases, we check the VNodes that was discovered during the process of traversal. If that is the same as v(vkey) that we are trying to insert, then we reported success to the state (see Line 185). Also, we check if that VNode is marked by invoking isMrkd procedure for deletion, means the v(vkey) already inserted and then marked for deletion, then we also reported success to the state array (see Line 197), else we try to report failure.

Like WFAddV, a WFRemV(vkey) operation invoked by passing the vkey to be deleted, in Line 38 to 50. It starts by choosing a phase number, announcing its operation on the state array to delete the v(vkey). Then the thread calls HelpGphDS(phase) to invoke the helping mechanism. Like a WFAddV operation, it traverses the state array and helps all pending operations and tries to complete its operation. In the next step the same thread(or a helping thread) enters HelpRemV(phase) and verifies the phase number and type of operation opType, if they match with remvertex then it invokes HelpLocV(phase) to traverse the vertex-list until it finds a vertex with its key greater than or equal to v(vkey). In the process of traversal, it physically deletes all logically deleted VNodes, using a CAS. Once it reaches the appropriate location checks whether the vkey is already present. If the ekey is present it attempts to remove the VNode in two steps (like [4]), (a) atomically marks the vnext of current VNode, using a CAS (Line 226), and (b) atomically updates the vnext of the predecessor VNode to point to the vnext of current VNode, using a CAS (Line 229). On any unsuccessful CAS it will cause the operation to restart from the HelpRemV procedure. After the operation completes HelpRemV will update success to the state array.

A WFConV(vkey) operation is much simpler than WFAddV and WFRemV. We have two cases based on with and without help. For the helping case a WFConV(vkey) operation, in Line 85 to 97, first starts publishing the operation in the state array like other operations. Then any helping thread will search it in the vertex-list, If the searching key is present and not been marked it reported success, else it reported failure, using a CAS to the state array. The HelpConV procedure, in Line 136 to 152, guarantees that the list traversal does not affected from infinite insertion of VNodes, this is because other threads will first help this operation before inserting a new VNode. Unlike a HelpAddV and HelpRemV, a HelpConV procedure does not need a loop to update the state state if any failure of the CAS,

For the without helping case a ConV(vkey) operation, in Line 332 to 338, first traverses the vertex-list in a wait-free manner skipping all logically marked VNodes until it finds a vertex with its key greater than or equal to vkey. Once it reaches the appropriate VNode, checks its key value equals to vkey, and it is unmarked, then it returns true otherwise returns false. We do not allow ConV for any helping in the process of traversal. This is because if the vertex set keys

```
98: Operation WFConE(key1, key2)
99:   tid ← ThreadID.get();
100:  phase ← MaxPhase();
101:  ⟨v1, v2, flag⟩ ← LocateUV(key1, key2);
102:  if (flag = false ⋁ isMrkd(v1) ⋁ isMrkd(v2)) then
103:    return VERTEX NOT PRESENT;
104:  end if
105:  ne ← new ENode(key2);
106:  ODA op ← new ODA (phase, findedge, ne, v1, v2);
107:  state[tid] ← op;
108:  HelpGphDS(phase);
109:  if (state[tid].type = success) then
110:    return EDGE PRESENT;
111:  else
112:    return VERTEX OR EDGE NOT PRESENT;
113:  end if
114: end Operation

115: procedure WFLocV(key)
116:  while (true) do
117:    v1 ← VertexHead;
118:    v2 ← v1.vnext;
119:    while (true) do
120:      v3 ← v2.vnext;
121:      while (isMrkd(v3)) do
122:        flag ← CAS(v1.vnext, v2, v3);
123:        if (¬flag) then
124:          goto Line 116;
125:        end if
126:        v2 ← v3; v3 ← v2.vnext;
127:      end while
128:      if (v2.vkey ≥ key) then
129:        return ⟨v1, v2⟩;
130:      end if
131:      v1 ← v2;
132:      v2 ← v3;
133:    end while
134:  end while
135: end procedure

136: procedure HelpConV(tid, phase)
137:  ODA op ← state[tid];
138:  if (¬(op.type = findvertex ∧ op.phase = phase)) then
139:    return;
140:  end if
141:  v1 ← op.vnode;
142:  ⟨pred, curr⟩ ← WFLocV(v1.vkey);
143:  if (curr.vkey = v1.vkey) ∧ (¬isMrkd (curr.vnext)) then
144:    ODA succ ← new ODA (phase, success);
145:    (CAS(state[tid], op, succ));
146:    return;
147:  else
148:    ODA fail ← new ODA (phase, failure);
149:    (CAS(state[tid], op, fail));
150:    return;
151:  end if
152: end procedure

153: procedure ConE(key1, key2)
154:  VNode v1, v2;
155:  ENode e;
156:  ⟨v1, v2, flag⟩ ← LocateUV(key1, key2);
157:  if (flag = false) then
158:    return VERTEX NOT PRESENT;
159:  end if
160:  if (isMrkd(v1) ∨ isMrkd(v2)) then
161:    return VERTEX NOT PRESENT;
162:  end if
163:  e ← EdgeHead;
164:  while ((e.ekey < key2)) do
165:    e ← UnMrkdRf(e.enext) ;
166:  end while
167:  if ((e.ekey = key2) ∧ (¬isMrkd(e.enext) ∧ (¬isMrkd(v1.vnext) ∧ (¬isMrkd(v2.vnext))) then
168:    return EDGE PRESENT;
169:  else
170:    return VERTEX OR EDGE NOT PRESENT;
171:  end if
172: end procedure

173: procedure HelpAddV(tid, phase)
174:  while (true) do
175:    ODA op ← state[tid];
176:    if (¬(opType = addvertex∧op. phase = phase)) then
177:      return;
178:    end if
179:    VNode v1 ← op.vnode;
180:    VNode v2 ← v1.vnext;
181:    ⟨pred, curr⟩ ← WFLocV(v1.vkey);
182:    if (curr.vkey = v1.vkey) then
183:      if (curr = v1 ∨ isMrkd( curr.vnext )) then
184:        ODA succ ← new ODA (phase, success);
185:        if CAS(state[tid], op, succ) then
186:          return;
187:        end if
188:      else
189:        ODA fail ← new ODA (phase, failure);
190:        if (CAS(state[tid], op, fail)) then
191:          return;
192:        end if
193:      end if
194:    else
195:      if ((isMrkd(v1.vnext))) then
196:        ODA succ ← new ODA (phase, success);
197:        if (CAS(state[tid], op, succ)) then
198:          return;
199:        end if
200:      end if
201:      CAS(v1.vnext, v2, curr);
202:      if (CAS(pred.vnext, curr, v1)) then
203:        ODA succ ← new ODA (phase, success);
204:        if (CAS(state[tid], op, succ)) then
205:          return;
206:        end if
207:      end if
208:    end if
209:  end while
210: end procedure
```

Fig. 4: Pseudo-codes of WFConE, WFLocV, HelpConV, ConE and HelpAddV.

are finite(upto available memory in the system) to achieve higher throughput and ConV is called more frequently than the WFAddV and WFRemV operations, so it does not allow any help.

C. The Edge Methods

The wait-free graph edge operations WFAddE, WFRemE, and WFConE are depicted in Figure 3 and 4 and their corresponding helping procedures HelpAddE, HelpRemE, and HelpConE are defined in Figure 5.

A WFAddE($\texttt{ekey}_1, \texttt{ekey}_2$) operation, in Line 51 to 67, begins by validating the presence of the $v(\texttt{ekey}_1)$ and $v(\texttt{ekey}_2)$ in the vertex-list by invoking LocateUV and are unmarked. If the validations fail, it returns VERTEX NOT PRESENT. Once the validation succeeds, WFAddE operation starts by choosing a phase number, creating a new ENode by invoking CEnode(ekey), posting its operation on the state array alone with the vsrc and vdest vertices. Then the thread calls HelpGphDS(phase) to invoke the helping mechanism. After that it traverses the state array and helps all pending operations and tries to complete its own operation. In the next step the same thread(or a helping thread) enters HelpAddE(phase)(in Line 256 to 298) and verifies the phase number and type of operation opType, if they match with the addedge then it invokes LocE($v(\texttt{ekey}_1), \texttt{ekey}_2$)(Line 269) to traverse the edge-list until it finds an ENode with its key greater than or equal to $\texttt{ekey}_2$. In the process of traversal, it physically deletes two kinds of logically deleted ENodes, (a) the ENodes whose VNode is logically deleted, using a CAS, and (b) the logically deleted ENodes, using a CAS. Once it reaches the appropriate location checks if the $\texttt{ekey}_2$ is already present or not. If present, then it attempts a CAS to add the new ENode (Line 290). On an unsuccessful CAS, it retries. After the operation completes HelpAddE(phase) success or failure reported to the state array.

In the process of traversal in the HelpAddE procedure, it is possible that the threads which are helping might have been inserted the $e(\texttt{ekey}_2)$ but not publish success. Also it is possible that the $e(\texttt{vkey}_2)$ we are trying to insert was already inserted and then removed by some other thread and then a different ENode with $\texttt{ekey}_2$ was inserted to the edge-list of $v(\texttt{ekey}_1)$. We properly handled these cases. Like WFAddV and WFRemV, we check the ENodes that was discovered during the process of traversal is the same key $\texttt{ekey}_2$ that we are trying to insert, then we reported success to the state array(see Line 273). Also we check if that ENode is marked(by invoking isMrkd), means the $e(\texttt{ekey}_2)$ already inserted and then marked for deletion, then we also reported success to state array(see Line 285), else we try to report failure.

```
211: procedure HELPREMV(tid, phase)
212:   while (true) do
213:     ODA op ← state[tid];
214:     if (¬(op.type = remvertex ∧ op. phase = phase)) then
215:       return;
216:     end if
217:     VNode v1 ← op.vnode;
218:     ⟨pred, curr⟩ ← WFLOCV(v1.vkey);
219:     cnext ← curr.vnext
220:     if (curr.vkey ≠ v1.vkey) then
221:       ODA fail ← new ODA (phase, failure);
222:       if (CAS(state[tid], op, fail)) then
223:         return;
224:       end if
225:     else
226:       if (¬CAS(curr.vnext, cnext, MRKDRF(cnext))) then
227:         goto Line 212;
228:       end if
229:       if (¬(CAS(pred.vnext, curr, cnext)) ) then
230:         goto Line 212;
231:       end if
232:       ODA succ ← new ODA (phase, success);
233:       if (CAS(state[tid], op, succ)) then
234:         return;
235:       end if
236:     end if
237:   end while
238: end procedure

239: procedure HELPCONE(tid, phase)
240:   ODA op ← state[tid];
241:   if (¬(op.type = findedge ∧ op.phase = phase)) then
242:     return;
243:   end if
244:   ENode e2 ← op.enode;
245:   ⟨pred, curr⟩ ← LOCE(op.vsrc, e2.ekey );
246:   if ((curr.ekey = e2.ekey) ∧ ISMRKD(curr.enext)) then
247:     ODA succ ← new ODA (phase, success);
248:     (CAS(state[tid], op, succ));
249:     return true;
250:   else
251:     ODA fail ← new ODA (phase, failure);
252:     (CAS(state[tid], op, fail));
253:     return false;
254:   end if
255: end procedure

256: procedure HELPADDE(tid, phase)
257:   while (true) do
258:     ODA op ← state[tid];
259:     if (¬(op.type = addedge ∧ op.phase = phase)) then
260:       return;
261:     end if
262:     if (ISMRKD(op.vsrc) ∨ ISMRKD(op.vdest )) then
263:       ODA fail ← new ODA (phase, failure);
264:       CAS(state[tid], op, fail);
265:       return;
266:     end if
267:     VNode v1 ← op.vsrc;
268:     ENode e2 ← op.enode, e3 ← e2.enext;
269:     ⟨pred, curr⟩ ← LOCE(v1, e2.ekey);
270:     if (curr.ekey = e2.ekey) then
271:       if (curr = e2) ∨ (ISMRKD(curr.enext )) then
272:         ODA succ ← new ODA (phase, success);
273:         if (CAS(state[tid], op, succ)) then
274:           return;
275:         end if
276:       else
277:         ODA fail ← new ODA (phase, failure);
278:         if (CAS(state[tid], op, fail)) then
279:           return;
280:         end if
281:       end if
282:     else
283:       if (ISMRKD(e2.enext)) then
284:         ODA succ ← new ODA (phase, success);
285:         if (CAS(state[tid], op, succ)) then
286:           return;
287:         end if
288:       end if
289:       CAS(e2.enext, e3, curr);
290:       if (CAS(pred.enext, curr, e2)) then
291:         ODA succ ← new ODA (phase, success);
292:         if (CAS(state[tid], newOP, succ)) then
293:           return;
294:         end if
295:       end if
296:     end if
297:   end while
298: end procedure

299: procedure HELPREME(tid, phase)
300:   while (true) do
301:     ODA op ← state[tid];
302:     if (¬(op.type = remedge ∧ op.phase = phase)) then
303:       return;
304:     end if
305:     if (ISMRKD(op.vsrc) ∨ ISMRKD(op. vdest)) then
306:       ODA fail ← new ODA (phase, failure);
307:       CAS(state[tid], op, fail);
308:       return;
309:     end if
310:     enode e2 ← op.enode;
311:     ⟨pred, curr⟩ ← LOCE(op.vsrc, e2. ekey);
312:     cnext ← curr.enext;
313:     if (curr.ekey ≠ e2.ekey) then
314:       ODA fail ← new ODA (phase, failure);
315:       if (CAS(state[tid], op, fail)) then
316:         return;
317:       end if
318:     else
319:       if (¬CAS(curr.enext, cnext, MRKDRF(cnext))) then
320:         goto Line 300;
321:       end if
322:       if (¬CAS(pred.enext, curr, cnext))) then
323:         goto Line 300;
324:       end if
325:       ODA succ ← new ODA (phase, success);
326:       if (CAS(state[tid], op, succ)) then
327:         return;
328:       end if
329:     end if
330:   end while
331: end procedure

332: procedure CONV(key)
333:   vnode v ← read(VertexHead);
334:   while (v.vkey < key) do
335:     v ← UNMRKDRF(v.vnext) ;
336:   end while
337:   return (v.vkey = key) ∧ (ISMRKD(v) = false );
338: end procedure
```

Fig. 5: Pseudo-codes of HELPREMV, HELPCONE, HELPADDE, HELPREME and CONV.

In each start of the **while**(Line 257) it is necessary to check for the presence of vertices op.vsrc and op.vdest in Line 262 by calling ISMRKD procedure. The reason explained in [1]. This is one of the several differences between an implementation trivially extending lock-free and wait-free list and ours. In fact, it can be seen that if this check is not performed, then it can result in the algorithm to not be linearizable.

A WFREME($\texttt{ekey}_1, \texttt{ekey}_2$) operation proceeds almost identical to the WFADDE, in Line 68 to 84, begins by validating the presence of the $v(\texttt{ekey}_1)$ and $v(\texttt{ekey}_2)$ and are unmarked. If the validations fail, it returns VERTEX NOT PRESENT. Once the validation succeeds, WFREME operation starts by choosing a phase number, creating a new ENode, posting its operation on the state array alone with the vsrc and vdest vertices. Then it gone through helping mechanism HELPGPHDS(phase). After that it traverses the state array and helps all pending operations and tries to complete its own operation. In the next step the same thread(or a helping thread) enters HELPREME(phase)(in Line 299 to 331) and verifies the phase number and type of operation opType, if they match with the remedge then it traverse the edge-list until it finds an ENode with its key greater than or equal to $\texttt{ekey}_2$. Like HELPADDE, in the process of traversal, it physically deletes two kinds of logically deleted ENodes, (a) the ENodes whose VNode is logically deleted, and (b) the logically deleted ENodes. Once it reaches the appropriate location checks if the $\texttt{ekey}_2$ is already present or not, if it is, attempts to remove the $e(\texttt{ekey}_2)$ in two steps like WFREMV operation, (a) atomically marks the enext of the current ENode, using a CAS (Line 319), and (b) atomically updates the enext of the predecessor ENode to point to the enext of current ENode, using a CAS (Line 322). On any unsuccessful CAS, it reattempted. Like HELPADDE, after the operation completes HELPREME(phase) success or failure reported to the state array.

Like WFCONV, a WFCONE operations also has two cases based on with and without help. For the helping case a WFCONE ($\texttt{ekey}_1, \texttt{ekey}_2$) operation, in Line 98 to 114, does similar work like WFADDE and WFREME operation. It

publishes the operation in the state and then invokes the HELPGPHDS(phase). If the current ENode is equal to the $\texttt{ekey}_2$ and $e(\texttt{ekey}_2)$ unmarked and $v(\texttt{ekey}_2)$ and $v(\texttt{ekey}_2)$ also unmarked, it updates the state to success, using a CAS. Otherwise failure is updated to the state.

For the without helping case a CONE($\texttt{ekey}_1, \texttt{ekey}_2$) operation, in Line 153 to 172, validates the presence of the corresponding VNodes. Then it traverses the edge-list of $v(\texttt{ekey}_1)$ in a wait-free manner skipping all logically marked ENodes until it finds an edge with its key greater than or equal to $\texttt{ekey}_2$. Once it reaches the appropriate ENode, checks its key value equals to $\texttt{ekey}_2$, and $e(\texttt{ekey}_2)$ is unmarked, and $v(\texttt{ekey}_1)$ and $v(\texttt{ekey}_2)$ are unmarked, then it returns EDGE PRESENT otherwise it returns VERTEX OR EDGE NOT PRESENT.

```
339: procedure LocE(srcv, key)
340:   while (true) do
341:     e1 ← srcv.enext;
342:     e2 ← e1.enext;
343:     while (true) do
344:       e3 ← e2.enext;
345:       v ← e2.pointv;
346:       while (isMrkd(v) ∧
     ¬isMrkd(e3)) do
347:         if (¬CAS (e2.enext, e3,
     MrkdRf (e3))) then
348:           goto Line 340;
349:         end if
350:         if (¬CAS (e1.enext, e2,
     e3)) then
351:           goto Line 340;
352:         end if
353:         e2 ← UnMrkdRf(e3);
354:         e3 ← e2.enext;
355:         v ← e2.pointv;
356:       end while
357:       while (isMrkd(e3)) do
358:         if (¬CAS (e1.enext, e2,
     e3)) then
359:           goto Line 340;
360:         end if
361:         e2 ← UnMrkdRf(e3);
362:         e3 ← e2.enext;
363:         v ← e2.pointv;
364:       end while
365:       if (isMrkd(v)) then
366:         goto Line 343;
367:       end if
368:       if (e2.ekey ≥ key) then
369:         return⟨e1, e2⟩;
370:       end if
371:       e1 ← e2; e2 ← e3;
372:     end while
373:   end while
374: end procedure

375: procedure LocateUV(k1, k2)
376:   if (k1 < k2) then
377:     v1 ← VertexHead;
378:     while (v1.vkey < k1) do
379:       v1 ← v1.vnext;
380:     end while
381:     if (v1.vkey ≠ k1 ∨
     isMrkd(v1)) then
382:       return ⟨v1, v2, false⟩;
383:     end if
384:     v2 ← v1.vnext;
385:     while (v2.vkey < k2) do
386:       v2 ← v2.vnext;
387:     end while
388:     if (v2.vkey ≠ k2 ∨
     isMrkd(v2)) then
389:       return ⟨v1, v2, false⟩;
390:     end if
391:   else
392:     v2 ← VertexHead;
393:     while (v2.vkey < k2) do
394:       v2 ← v2.vnext;
395:     end while
396:     if (v2.vkey ≠ k2 ∨
     (isMrkd(v2))) then
397:       return ⟨v1, v2, false⟩;
398:     end if
399:     v1 ← nv.vnext;
400:     while (v1.vkey < k1) do
401:       v1 ← v1.vnext;
402:     end while
403:     if (v1.vkey ≠ k1 ∨
     (isMrkd(v1))) then
404:       return ⟨v1, v2, false⟩;
405:     else
406:       return ⟨v1, v2, true⟩;
407:     end if
408:   end if
409: end procedure
```

Fig. 6: Pseudo-codes of LOCE, LOCATEUV, OWFADDV and OWFREMV.

V. AN OPTIMIZED FAST WAIT-FREE GRAPH ALGORITHM

In this section, we present the optimized version of our wait-free concurrent graph data-structure, which is designed based on the fast-path-slow-path algorithm by Kogan et al. [11]. The fast-path-slow-path algorithm is a combination of two parts, the first part is a lock-free algorithm which is usually fast and the second one is a wait-free algorithm which is slow. Pragmatically the lock-free algorithms are fast as compare to the wait-free algorithms as they don't require helping always. So, to enhance the performance and achieve a fast wait-free graph we adopted lock-free graph by Chatterjee et al. [1] and wait-free graph which described in previous Section IV. The basic working principle of the optimized wait-free graph is as follows: (1). Before an operation begins the fast-path lock-free algorithm, it inspects whether help needed for any other operations in the slow-path wait-free algorithm, (2). Then the operation starts running with its fast-path lock-free algorithms while it keeps tracking the number times it fails, which is nothing but the number of failed CAS. Generally, if very less number of CAS failed happen then helping is not necessary and hence the execution finishes just after completing the fast-path lock-free algorithm. (3). If the operation is unable to finish its execution after trying a certain number of CAS, then it allows entering the slow-path algorithm to finish the execution.

The slow-path wait-free algorithms described in Section IV. Each operation chooses its phase numbers then it publish the operation in the state array by updating its corresponding entry. Then it traverse through the state array and try to help the operations whose phase number is lower than or equals to its own phase number, which ensures the unfinished operations gets help from other threads to finish the execution. This ensures wait-freedom. The maximum number of tries in the fast-path is upper bounded by a macro MAX_FAIL similar to [11], we choose MAX_FAIL to 20. On average, we achieve high throughput with this upper bound value. Due to lack of space, the detailed implementation described in the technical report[a].

VI. CORRECTNESS AND PROGRESS GUARANTEE

In this section, we prove the correctness of our concurrent wait-free graph data-structure based on LP [8] events inside the execution interval of each of the operations.

Theorem 1: The concurrent wait-free graph operations are linearizable.

Proof. Based on the return values of the operations we discuss the LPs.

1) WFADDV(vkey): We have two cases:
   a) true: The LP be the successful CAS execution at the Line 202.
   b) false: The LP be the atomic read of the vnext pointing to the vertex $v(\texttt{vkey})$.
2) WFREMV(vkey): We have two cases:
   a) true: The LP be the successful CAS execution at the Line 226 (logical removal).
   b) false: If there is a concurrent WFREMV operation *op*, that removed $v(\texttt{vkey})$ then the LP be just after the LP of *op*. If $v(\texttt{vkey})$ did not exist in the vertex-list then the LP be at the invocation of WFREMV.
3) WFCONV(vkey): We have two cases:
   a) true: The LP be the atomic read of the vnext pointing to the vertex $v(\texttt{vkey})$.
   b) false: The LP be the same as returning false WFREMV, the case 2b.

[a]The technical report is available: https://arxiv.org/abs/1810.13325

4) WFADDE(k_1, k_2): We have three cases:
   a) `EDGE ADDED`:
      i) With no concurrent WFREMV(k_1) or WFREMV(k_2): The LP be the successful `CAS` execution at the Line 290.
      ii) With concurrent WFREMV(k_1) or WFREMV(k_2): The LP be just before the first remove's LP.
   b) `EDGE ALREADY PRESENT`:
      i) With no concurrent WFREMV(k_1) or WFREMV(k_2): The LP be the atomic read of the `enext` pointing to the `ENode` $e(k_2)$ in the edge-list fo the vertex $v(k_1)$.
      ii) With concurrent WFREMV(k_1) or WFREMV(k_2) or WFREME(k_1, k_2) : The LP be just before the first remove's LP.
   c) `VERTEX NOT PRESENT`:
      i) At the time of invocation of WFADDE(k_1, k_2) if both vertices $v(k_1)$ and $v(k_2)$ were in the vertex-list and a concurrent WFREMV removes $v(k_1)$ or $v(k_2)$ or both then the LP be the just after the LP of the earlier WFREMV.
      ii) At the time of invocation of WFADDE(k_1, k_2) if both vertices $v(k_1)$ and $v(k_2)$ were not present in the vertex-list, then the LP be the invocation point itself.
5) WFREME(k_1, k_2): We have three cases:
   a) `EDGE REMOVED`:
      i) With no concurrent WFREMV(k_1) or WFREMV(k_2): The LP be the successful `CAS` execution at the Line 319(logical removal).
      ii) With concurrent WFREMV(k_1) or WFREMV(k_2): The LP be just before the first remove's LP.
   b) `EDGE NOT PRESENT`: If there is a concurrent WFREME operation removed $e(k_1, k_2)$ then the LP be the just after its LP, otherwise at the invocation of WFREME(k_1, k_2) itself.
   c) `VERTEX NOT PRESENT`: The LP be the same as the case WFADDE returning "`VERTEX NOT PRESENT`"4c.
6) WFCONE(k_1, k_2): Similar to WFREME, we have three cases:
   a) `EDGE PRESENT`:
      i) With no concurrent WFREMV(k_1) or WFREMV(k_2): The LP be the atomic read of the `enext` pointing to the `ENode` $e(k_2)$ in the edge-list fo the vertex $v(k_1)$.
      ii) With concurrent WFREMV(k_1) or WFREMV(k_2) or WFREME(k_1, k_2) : The LP be just before the first remove's LP.
   b) `VERTEX NOT PRESENT`: The LP be the same as that of the WFADDE's returning "`VERTEX NOT PRESENT`" case 4c.
   c) `VERTEX OR EDGE NOT PRESENT`: The LP be the same as that of the WFREME's returning "`EDGE NOT PRESENT`" and WFADDE's returning "`VERTEX NOT PRESENT`" cases.

From the above discussion one can notice that each operation's LPs lies in the interval between the invocation and the return steps. For any invocation of a WFADDV(`vkey`) operation the traversal terminates at the `VNode` whose key is just less than or equal to `vkey`. Similar reasoning also true for invocation of an WFADDE(k_1, k_2) operation. Both the operations do the traversal in the sorted vertex-list and edge-list to make sure that a new `VNode` or `ENode` does not break the invariant of the wait-free graph data-structure. The WFREMV and WFREME do not break the sorted order of the lists. Similarly, the non-update operations do not modify the data-structure. Thus we concluded that all wait-free graph operations maintain the invariant across the LPs. This completes the proof of the linearizability. □

Theorem 2: The presented concurrent graph operations WFADDV, WFREMV, WFCONV, WFADDE, WFREME, and WFCONE are wait-free.

Proof. To show the concurrent graph algorithms to be wait-freedom, we have to make sure that the helping procedure terminates with a limited number try in concurrent with update operations. As we discussed in Section IV that each operation chooses its `phase` numbers larger than all the previous operations and then publishes the operation in the `state` array by updating its entry. After that, it traverses through the `state` array and tries to help the operations whose `phase` number is lower than or equal to its `phase` number, which ensures the unfinished operations gets help from other thread to finish the execution. So that all the threads help the pending operations and finished the execution with a limited number of steps. This ensures wait-freedom. Therefore, the graph operations WFADDV, WFREMV, WFCONV, WFADDE, WFREME, and WFCONE are wait-free. □

VII. EXPERIMENTS AND ANALYSIS

Experimental Setup: We conducted our experiments on a processor with Intel(R) Xeon(R) E5-2690 v4 CPU containing 56 cores running at 2.60GHz. Each core supports 2 logical threads. Every core's L1-64K, L2-256K cache memory is private to that core; L3-35840K cache is shared across the cores, 32GB of RAM and 1TB of hard disk, running 64-bit Linux operating system. All the implementation[b] were written in C++ (without any garbage collection) and multi-threaded implementation is based on Posix threads.

Running Strategy: In the experiments, we start with an initial graph of 1000 vertices and nearly $\binom{1000}{2}/4 \approx 125000$ edges added randomly, rather than starting with an empty graph. When the program begins, it creates a fixed set of threads (1, 10, 20, 30, 40, 50, 60 and 70) and each thread

[b]The source code is available on https://github.com/PDCRL/ConcurrentGraphDS.

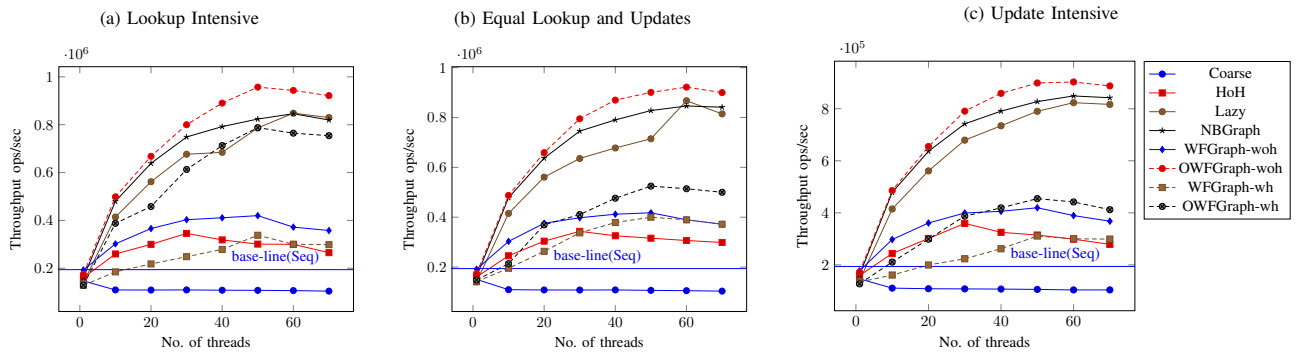

Fig. 7: Experimental Results for Wait-free and Optimised Wait-free Graph.

randomly performs a set of operations chosen by a particular workload distribution, as given below. The metric for evaluation is the number of operations completed in a unit time, i.e. throughput. We measured throughput by running the experiment for 20 seconds. Each data point is obtained by averaging over 5 iterations.

Workload Distribution: To compare the performance with several micro benchmarks, we used the following distributions over the ordered set of operations {WFAddV, WFRemV, WFConV, WFAddE, WFRemE, WFConE }:

1) *Lookup Intensive*: (2.5%, 2.5%, 45%, 2.5%, 2.5%, 45%), see the Figure 7a.
2) *Equal Lookup and Updates*: (12.5%, 12.5%, 25%, 12.5%, 12.5%, 25%), see the Figure 7b.
3) *Update Intensive*: (22.5%, 22.5%, 5%, 22.5%, 22.5%, 5%), Figure 7c.

We compare the following concurrent graph algorithms:

1) **Seq**: Sequential execution of all the operations.
2) **Coarse**: Execution with a coarse grained lock [7, Ch. 9].
3) **HoH**: Execution with Hand-over-Hand lock [7, Ch. 9].
4) **Lazy**: Execution with Lazy-lock [5].
5) **NBGraph**: Based on non-blocking graph [1].
6) **WFGraph-woh**: The wait-free graph algorithm with ConV and ConE (without helping of contains vertex and edge operations) Section IV.
7) **WFGraph-wh**: The wait-free graph algorithm with WFConV and WFConE (with helping of contains vertex and edge operation) Section IV.
8) **OWFGraph-woh**: The optimized version of wait-free graph algorithm with ConV and ConE (without helping of contains vertex and edge operations) Section V.
9) **OWFGraph-wh**: The optimized version of wait-free graph algorithm with WFConV and WFConE (with helping of contains vertex and edge operation) Section V.

In the plots shown in Figure 7, we observe that the `WFGraph-woh` and `WFGraph-wh` algorithm does not scale well like `NBGraph` with the number of threads in the system, on the other hand, the `OWFGraph-woh` variant scales well compare to others.

VIII. Conclusion and Future Directions

In this paper, we presented an efficient, practical wait-free algorithm to implement a concurrent graph data-structure, which allows threads to insert and delete the vertices/edges concurrently. We also developed an optimized version of wait-free graph using the concept of fast-path-slow-path algorithm developed by Kogan et al. [11]. We extensively evaluated the C++ implementation of our algorithm and the optimized variant through several micro-benchmarks. We compared wait-free graph and optimized wait-free graph algorithms with sequential, coarse-lock, hand-over-hand lock, lazy lock, and non-blocking concurrent graphs. The optimized wait-free graph without helping of contains vertex and edge operations achieves nearly up to 9x speedup on throughput with respect to locking counterparts and nearly 1.5x speedup with respect to non-blocking counterpart. Currently, our implementation does not have any garbage collection mechanism. In future, we plan to enhance our implementation with a garbage collection procedure similar to [13].

References

[1] Bapi Chatterjee, Sathya Peri, Muktikanta Sa, and Nandini Singhal. A Simple and Practical Concurrent Non-blocking Unbounded Graph with Linearizable Reachability Queries. In *ICDCN 2019, Bangalore, India, January 04-07, 2019*, pages 168–177, 2019.

[2] Camil Demetrescu, Irene Finocchi, and Giuseppe F. Italiano. Dynamic Graphs. In *Handbook of Data Structures and Applications*. 2004.

[3] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004*, pages 50–59, 2004.

[4] Timothy L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings*, pages 300–314, 2001.

[5] 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.

[6] 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.

[7] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., 1st edition, 2012.

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

[9] Nikolaos D. Kallimanis and Eleni Kanellou. Wait-Free Concurrent Graph Objects with Dynamic Traversals. In *OPODIS 2015, December 14-17, 2015, Rennes, France*, pages 27:1–27:17, 2015.

[10] Alex Kogan and Erez Petrank. Wait-Free Queues With Multiple Enqueuers and Dequeuers. In *PPOPP*, pages 223–234. ACM, 2011.
[11] Alex Kogan and Erez Petrank. A Methodology for Creating Fast Wait-free Data Structures. In *PPoPP '12*, pages 141–150, New York, NY, USA, 2012. ACM.
[12] Maged M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *SPAA*, pages 73–82, 2002.
[13] Maged M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004.
[14] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-Free Linked-Lists. In *OPODIS*, volume 7702 of *Lecture Notes in Computer Science*, pages 330–344. Springer, 2012.
[15] Shahar Timnat and Erez Petrank. A Practical Wait-Free Simulation for Lock-Free Data Structures. In *PPOPP*, pages 357–368. ACM, 2014.
[16] John D. Valois. Lock-Free Linked Lists Using Compare-and-Swap. In *PODC 1995*, pages 214–222, 1995.