A Concurrent Unbounded Wait-Free Graph

Sathya Peri, Chandra Kiran Reddy, Muktikanta Sa

Department of Computer Science & Engineering
Indian Institute of Technology Hyderabad, India
{sathya_p, cs15btech11012, cs15resch11012}@iith.ac.in

Abstract
In this paper, we propose a concurrent non-blocking unbounded directed graph (for shared memory architecture) that is concurrently being updated by threads adding/deleting vertices and edges. All the operations of the algorithm are wait-free in nature. We extend the wait-free list implementation of concurrent set for achieving this. We have compared the performance of the proposed concurrent data-structure with coarse-grained, hand-over-hand, fine-grained locking and lock-free implementation and achieves significant improvement. We also extended our implementation using fast-path-slow-path based on the algorithm proposed by Timnat et al.

Keywords: concurrent data structure; lazy-list; directed graph; locks, lock-free, wait-free; fast-path-slow-path;

1 Introduction
A graph represents pairwise relationships between objects along with their properties. Due to their usefulness, graphs are being used in various fields like genomics various kinds of networks such as social, semantic etc. Generally, these graphs are very large and dynamic in nature. Dynamic graphs are the one’s which are subjected to a sequence of changes like insertion, deletion of vertices and/or edges [2]. Online social networks (facebook, linkedin, google+, twitter, quora, etc.), are dynamic in nature with the users and the relationships among them changing over time.

In this paper, we develop a concurrent non-blocking unbounded directed graph (for shared memory architecture) that is concurrently being updated by threads adding/deleting vertices and edges while ensuring linearizability [7]. We show experimental analysis of the concurrent graph structure under varying workload distributions which demonstrate the concurrency obtained against different locking and lock-free strategies. The algorithm we designed for the concurrent graph data structure [1] is based on the wait-free implementation proposed by Timnat et al. [13]. Our implementation is not a straight forward extension to wait-free list implementation but has several non-trivial additions. This can be seen from the linearization points of edge methods which lie outside their method and depend on other concurrently executing graph methods. Moreover we believe the design of the graph data-structure is such that it can help identify other useful properties on graph such as reachability, cycle detection, shortest path, minimum spanning tree, etc.

Our main contribution of this paper is a practical, linearizable, wait-free unbounded directed graph. Also we have designed fast-path-slow-path algorithm based on [13].

2 The Graph Data-structure
In this section, we describe the graph data-structure based on [1]. It is based on the adjacency list representation. Hence, it is implemented as a collection (list) of vertices wherein each vertex in turn holds a list of vertices to which it has outgoing edges.

The problem addressed in this paper is as follows: A concurrent directed graph $G = (V, E)$, which is dynamically being modified by a fixed set of concurrent threads. In this setting, threads may perform insertion / deletion of vertices or edges to the graph.
The data structure, we consider a shared-memory system consisting of a finite set of processors accessed by a finite set of threads that run in a completely asynchronous manner. The threads communicate with each other by invoking operations on shared objects and getting corresponding responses. The pointers and other fields of the various nodes are implemented by the shared objects. The system supports atomic read, write, fetch-and-add (FAA) and compare-and-swap (CAS) instructions.

The structures of the VNode and ENode are given in Figure 1 similar as [1]. A VNode consists of two pointers vnext and enext in addition to an immutable key val. vnext points to the next VNode in the vertex-list, whereas, enext points to the head of the edge-list. In the edge-list, an ENode a pointer enext points to the next ENode in the edge-list in addition to an immutable key field val. We assume that all the vertices have unique identification key (captured by val field).

Our wait-free concurrent graph data-structure supports six major operations like [1]: WFAddVertex, WFRemoveVertex, WFCcontainsVertex, WFAddEdge, WFRmoveEdge and WFCcontainsEdge. The helping mechanism is required like [4,9,13,15] to achieve the wait-freedom. So our graph data-structure uses helping mechanism for all six methods. Before starting to execute a method, a thread starts invoking a special state array we called it as an Operation Descriptor Array (ODA) same as Timnat et al. [13]. This ODA is shared among all the threads and can view the details of the method it is executing. Once a method is published, all other threads can try to help it for execution. When the method completed it’s execution, the result is reported to the ODA, by doing a CAS, which substitutes the old existing operation descriptor with the new one.

class VNode {
    int val; // immutable key field
    VNode vnext; // atomic ref., pointer to the next VNode
    ENode enext; // atomic ref., pointer to the edge-list
}
class ENode {
    int val; // immutable key field
    ENode enext; // atomic ref., pointer to the next ENode
}

Figure 1: Structure of ENode and VNode.

Figure 2: (a) A directed Graph (b) The concurrent graph data structure representation for (a).

2.1 Graph Methods and Sequential Specification

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

1. The WFAddVertex(u) method adds a vertex u to the graph, returns success if the key is not present earlier otherwise it returns failure.

2. The WFRmoveVertex(u) method deletes vertex u from the graph, if it is present in the graph and returns success. If the vertex is not in the graph, it returns failure.
3. The WFContainsVertex\((u)\) returns success, if the graph contains the vertex \(u\); otherwise returns failure.

4. The WFAddEdge\((u, v)\) method adds a directed edge \((u, v)\) to the graph if the edge \((u, v)\) is not present earlier in the graph and returns success. If either of the vertices \(u\) or \(v\) is not present or the edge \((u, v)\) is already in the graph, it returns failure.

5. The WFRemoveEdge\((u, v)\) method deletes the directed edge \((u, v)\) from the graph if it is present earlier and returns success. If either of the vertices \(u\) or \(v\) is not present or the edge \((u, v)\) is not present in the graph, it returns failure.

6. The WFContainsEdge\((u, v)\) returns success, if the graph contains the edge \((u, v)\); If either of the vertices \(u\) or \(v\) or the edge \((u, v)\) is not present in the graph, it returns failure.

7. The HelpGraphDS\((phase)\) method ensures that each thread completes its own operation and helps in completing all the pending operations with lower phase numbers.

Each of the method will get a new phase number and post their operation in ODA and invoke HelpGraphDS\((phase)\) method. All the helping methods check if their phase is same as the threads phase in ODA, else they return failure.

3 The Graph Algorithm

In this section, we describe the details implementation of the concurrent graph structure and the working of the operations. We represent the graph using adjacency list representation, which is a list of linked lists as illustrated in the Figure 2. The underlying adjacency list representation is an adaptation of the lock-free [4] and wait-free [13] based concurrent linked-list. All the fields in the structure are declared atomic. This ensures that operations on these variables happen atomically. In the context of a particular application, the node structure can be easily modified to carry useful data (like weights etc). We defined all the pseudo code used concurrent graph data structure from the Algorithm 1-12 in the paper by Chatterjee et al. [1].

3.1 High-level Overview of the data-structure and Algorithm

In this section we provide an overview of the design wait-free graph data structure. It is based on the adjacency list representation. Hence, it is implemented as a collection (list) of vertices wherein each vertex in turn holds a list of vertices to which it has outgoing edges. The implementation is a linked list of VNode and ENode as shown in the Table 1. The implementation of each of these lists are based on the non-blocking list implementation of the a concurrent-set [3, 5, 8, 10, 12, 14, 16, 17].

The ENode class has three fields. The val field is the key value of the edge\((u, v)\) (edge from \(u\) to \(v\)), stores the key value of \(v\). The edge nodes are sorted in order of the val field. This helps efficiently detect when a ENode is absent in the edge list. The enext pointer in each node can be marked using a special marked bit, to signify that the entry in the node is logically deleted.

The marked field is of type boolean which indicates whether that ENode is in the edge list or not. The enext field is a reference to the next ENode in the edge list.

Similarly, the VNode class has five fields. The val field is the key value of the vertex \(u\). The vertex nodes are sorted in the order of the val field which helps detect presence/absence of a VNode in the vertex list (like the the sorted ENode list). The marked field is a boolean marked field indicating whether that VNode is in the vertex list or not. The vnext field is a reference to the next VNode in the vertex list. The EdgeHead field is a sentinel ENode at the start of the each edge list for each VNode has the smallest possible key value (−\(\infty\)). The EdgeTail field is sentinel ENode at the end of the each edge list has the largest possible key value (+\(\infty\)).

We assume the enext and marked are treated as a single atomic unit: any attempt to update the enext field when the marked field is true will fail. Similarly, the vnext and marked fields of a VNode are treated as a single atomic unit.

Our wait-free concurrent graph data-structure supports six major operations: AddVertex, RemoveVertex, ContainsVertex, AddEdge, RemoveEdge and ContainsEdge. All of these operations are helped
by their unique supporting methods which in turn make them to run in wait-free manner. The non-blocking list algorithm uses the same abstraction map as the lazy list algorithm: a key is in the set if, and only if it is in an unmarked reachable node from the VertexHead.

Our algorithms are in pseudo-code on a mix of c++ and JAVA and designed for execution on a shared-memory multi-processor with fixed number of threads, the system supports atomic read, write and compare-and-swap (CAS) operations.

As stated earlier our graph data-structure uses helping mechanism for all six methods to achieve the wait-freedom. So, before starting to execute a method, a thread starts invoking a special state array we called it as an Operation Descriptor Array (ODA) same as Timnat et. al. [14]. This ODA is shared among all the threads and can view the details of the method it is executing. Once a method is published, all other threads can try to help it for execution. When the method completed it’s execution, the result is reported to the ODA, by doing a CAS, which substitutes the old existing operation descriptor with the new one.

We also maintain an ODA for each thread. The ODA entry for each thread describes its current state. And it’s class is defined in the Table 1. The ODA class has seven fields, a phase field phase (phase number of the operation), the OpType field signifying which operation is currently being executed by this thread, a pointer to a vnode denoted en, pointer to two enode denoted en1 and en2, which serve the insert and delete operations, EWindow for searching result of enode denoted ESearchResult a pair of pointers (prev, curr) for recording the result of a search operation for any enode and VWindow for searching result of vnode denoted VSearchResult a pair of pointers (prev, curr) for recording the result of a search operation for any vnode. We also maintain an array with type ODA named state.

3.2 The Vertex Methods

The WFAddVertex\((u)\) method starts by posting its operation on ODA array. The thread calls HelpGraphDS\((phase)\) to invoke the helping mechanism. The thread traverses ODA array and helps all pending operations try to complete its own operations. In the next step the same thread (or a helping thread) enters HelpAddVertex\((phase)\) and verifies the phase number and type of operation opType, if they match the thread will invoke HelpLocateVertex\((phase)\) to traverse the vertex list until it finds a vertex with its key greater than or equal to \(u\), say \(ucurr\) and it’s predecessor, say \(upred\). If the \(ucurr\) holds a \(val\) equal to the \(u\) the vertex to be added, then it returns the failure. Otherwise the vertex should be inserted between \(ucurr\) and \(upred\). This is done by first updating the new VNode’s \(vnext\) pointer to point to the \(ucurr\), and then updating the \(upred\)’s \(vnext\) to point to it. The later one is done using a CAS to prevent the race condition and any failure of the CAS will cause the operation to restart from the HelpLocateVertex method. Finally, when the new VNode has been added, the WFAddVertex returns success. After the operation completes HelpAddVertex\((phase)\) will update the ODA to success or failure.

The WFRmoveVertex\((u)\) starts by posting its operation on ODA array like WFAddVertex. The thread calls HelpGraphDS\((phase)\) to invoke the helping mechanism. The thread traverses ODA array and helps all pending operations try to complete its own operations. It executed in two stages. First the VNode to be removed is chosen. To do this, the HelpLocateVertex method is called. It finds a vertex with its key greater than or equal to \(u\), say \(ucurr\) and it’s predecessor, say \(upred\). If the \(ucurr\) holds a \(val\) not equal to the \(u\) the vertex to be removed, then it returns the failure. Otherwise the VNode to be removed is announced in the ODA array and then update it so that concurrent helping thread will not remove two different VNodes as the VNode which unable to help this operation is determined to be the single VNode that is announced in the ODA array.

In the second stage the, the WFRmoveVertex is executed similar to Harris’s lock-free list [4], the \(ucurr\)’s \(vnext\) is marked using CAS called logical removed and any failure of the CAS will cause the operation to restart from the HelpLocateVertex method, and then physically removed from the vertex-list, by making \(upred\)’s \(vnext\) point to \(ucurr\)’s \(vnext\), this is done by CAS as well. After successful logical removal success is reported to the ODA array.

The WFCcontainsVertex method is much simpler than WFAddVertex and WFRmoveVertex. First it start publishing the operation. Any helping thread will then search for it in the vertex-list, If the searching key is present and not marked it report success else it reported failure to the ODA.
3.3 The Edge Methods

The WFAddEdge\((u, v)\) method is similar to vertex operations by posting the operation in ODA and invoking HelpGraphDS\((phase)\). It verifies the phase number and type of operation opType if they match the thread will invoked HelpAddEdge\((u, v)\) which starts by checking for the presence of vertices \(u\) and \(v\) in the vertex-list. After this, once again \(u\) and \(v\) are validated to be unmarked. The reason for this is explained by an example in Figure 3. 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.

Once the vertices \(u\) and \(v\) have been validated to be reachable and unmarked in the vertex list, the thread traverses the edge-list of vertex \(u\) until an edge node with key greater than \(v\) has been encountered, say ecurr and it’s predecessor say epred. If the ecurr holds a val equal to the \(v\) the ENode to be added, then it returns the failure. Otherwise the ENode should be inserted between ecurr and epred. This is done by first updating the new ENode’s enext pointer to point to the ecurr, and then updating the epred’s enext to point to it. The later one is done using a CAS to prevent the race condition and any failure of the CAS will cause the operation to restart from the HelpAddEdge method. Finally, when the new ENode has been added, the WFAddEdge return success. After the operation completes HelpAddEdge\((phase)\) will update the ODA to success or failure.

![Figure 3](image)

Figure 3: This figure depicts why we need an additional check to locate vertices in WFAddEdge\((u, v)\). A thread \(T_1\) trying to perform WFAddEdge\((u, v, success)\), first invokes HelpLocateVertex. Just after \(T_1\) has verified vertex \(u\), thread \(T_2\) deletes vertex \(u\). Also vertex \(v\) gets added by thread \(T_3\) just before \(T_1\) verifies it. So, now thread \(T_1\) has successfully tested for the presence of vertices \(u\) and \(v\) in the vertex list, and then it proceeds to add edge \((u, v)\), returning success. However, as is evident, no possible sequentially generated history of the given concurrent execution is correct. Hence an additional check must be performed before proceeding to actually add the edge.

The WFRmoveEdge\((u, v)\) method proceeds similar to the WFAddEdge\((u, v)\) by posting the operation in ODA and invoking HelpGraphDS\((phase)\). If the ecurr holds a val is not equal to the \(v\) the ENode to be removed, then it returns the failure. Otherwise the ENode should be removed between ecurr and epred like WFRmoveVertex in two stages: logical and physical remove. After the operation completes HelpRemoveEdge\((phase)\) will update the ODA to success or failure.

Like other edge operation the WFCcontainEdge\((u, v)\) also post the operation in ODA and then invokes the HelpGraphDS\((phase)\). If the ecurr holds a val is equal to the \(v\) the ENode and not marked, it returns the success. Otherwise returns failure. After the operation completes HelpContainsEdge\((phase)\) will update the ODA to success or failure.

3.4 The Fast-Path-Slow-Path Variation

Generally the lock-free algorithms are fast as compare to the wait-free one. The idea behind the fast-path, slow-path [9] approach is 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. We use the Harris’s lock-free linked-list [4] approach for the fast-path. The fast-path algorithm begins by a check whether helping is required for any
operation in the slow-path. The operation start running with its lock-free algorithm while counting the
number of contentions that end with a failed CAS. Generally, very less number of failures occur and helping not
required, and so the execution terminates after running the faster lock-free algorithm. If this fast-path fails
to make progress the execution moves to the slow-path, which runs the slower wait-free algorithm described
in the above Section 3. So requesting help using ODA and making sure the operation eventually terminates.
The number of CAS failures allowed in the fast-path is limited by a parameter called MAX_FAIL. The help
is provided by threads running fast-path-slow-path ensures wait-freedom. The full implementation of the
fast-path-slow-path variation of the concurrent graph data structure is described in the technical report.

4 Working of Concurrent Graph Methods

In this section, we describe the implementation of the concurrent graph structure and the working of the
various methods. We represent the graph using adjacency list representation, which is a list of linked
lists. Almost all the fields in the structure are atomic. For application use the structure can be altered to carry
custom data.

Notations used in PseudoCode:

\(\downarrow, \uparrow\) denote input and output arguments to each method respectively. The shared memory is accessed only by
invoking explicit \textit{read()} and \textit{write()} methods. The flag is a local variable which returns the status of each
operation. We use \(e_1, e_2\) to represent the ENode reference nodes and \(v_1, v_2, v\) to represent VNode references.

Algorithm 1 maxPhase Method: Takes \(currMaxPhase\) and returns next value by doing \(currMaxPhase + 1\).

1. \textbf{procedure} \textit{MaxPhase}(\textit{CurrMaxPhase} \downarrow, \textit{MaxPhase} \uparrow)
2. \hspace{1em} \textit{MaxPhase} \leftarrow \textit{CurrMaxPhase};
3. \hspace{1em} \textit{CAS}(\textit{CurrMaxPhase}, \textit{MaxPhase}, \textit{MaxPhase} + 1);
4. \textbf{end procedure}

Algorithm 2 Help Method: Takes \textit{Phase} number larger than all previously chosen phase numbers. It make
sure that old operations receive help and complete before new operations are executed.

5. \textbf{procedure} \textit{HelpGraphDS} (\textit{Phase} \downarrow)
6. \hspace{1em} \textit{tid} \leftarrow 0;
7. \hspace{1em} \textbf{while} (\textit{tid} < \textit{state}.length()) \textbf{do}
8. \hspace{2em} \textit{ODA desc} \leftarrow \textit{state}[\textit{tid}];
9. \hspace{2em} \textbf{if} (\textit{desc.phase} \leq \textit{Phase}) \textbf{then}
10. \hspace{3em} \textbf{if} (\textit{desc.type} = \textit{OpType.WFAddVertex}) \textbf{then}
11. \hspace{4em} \textit{HelpAddVertex}(\textit{desc.phase});
12. \hspace{3em} \textbf{else if} (\textit{desc.type} = \textit{OpType.WFRemoveVertex}) \textbf{then}
13. \hspace{4em} \textit{HelpRemoveVertex}(\textit{desc.phase});
14. \hspace{3em} \textbf{else if} (\textit{desc.type} = \textit{OpType.WFContainsVertex}) \textbf{then}
15. \hspace{4em} \textit{HelpContainsVertex}(\textit{desc.phase});
16. \hspace{3em} \textbf{else if} (\textit{desc.type} = \textit{OpType.WFAddEdge}) \textbf{then}
17. \hspace{4em} \textit{HelpAddEdge}(\textit{desc.phase});
18. \hspace{3em} \textbf{else if} (\textit{desc.type} = \textit{OpType.WFRemoveEdge}) \textbf{then}
19. \hspace{4em} \textit{HelpRemoveEdge}(\textit{desc.phase});
20. \hspace{3em} \textbf{else if} (\textit{desc.type} = \textit{OpType.WFContainsEdge}) \textbf{then}
21. \hspace{4em} \textit{HelpContainsEdge}(\textit{desc.phase});
22. \hspace{2em} \textbf{end if}
23. \hspace{1em} \textbf{end if}
24. \hspace{1em} \textbf{end while}
25. \textbf{end procedure}
class enode{
    int val;
    enode enext; // atomic
    boolean marked; // atomic
    enode(int key){
        val = key;
        marked = false;
        enext = null;
    }
};
enum OpType{AddVertex, RemoveVertex,
            ContainVertex, AddEdge,
            RemoveEdge, ContainEdge,
            success, failure};
class ODA{
    int phase;
    OpType type;
    vnode vn;
enode en1,en2;
EWindow ESearchResult; // for edge
VWindow VSearchResult;// for vertex
    /* Constructor for vnode */
    ODA(int ph, OpType tp, vnode v,
         VWindow result){
        phase = ph;
        type = tp;
        vn = v;
en1 = NULL;
en2 = NULL;
VSearchResult = result;
    }
    /* Constructor for enode */
    ODA(int ph, OpType tp, enode e1,
         enode e2, EWindow result){
        phase = ph;
        type = tp;
en1 = e1;
en2 = e2;
v = NULL;
ESearchResult = result;
    }
};
class vnode{
    int val;
    vnode vnext; // atomic
    enode EdgeHead;
enode EdgeTail;
    boolean marked; // atomic
    vnode(int key){
        val = key;
        marked = false;
vnext = null;
    EdgeHead = new enode(-infinity);
    EdgeTail = new enode(+infinity);
    EdgeHead.enext = EdgeTail;
    }
};
class VWindow{
    vnode pred;
    vnode curr;
    VWindow(vnode p, vnode c){
        pred = p;
curr = c;
    }
};
vnode Vhead, Vtail; // sentinel nodes
ODA state; // atomic
int long currMaxPhase; //
init(){
currMaxPhase = 0;
Vhead = new vnode(MIN);
vnext = new vnode(MAX);
Vhead.vnext = Vtail;
state = new ODA[numThread];
for( int i = 0 to state.length())
    state[i].ODA(0, success, NULL, NULL);
}
Algorithm 3 isSearchStillPending Method: Takes Phase number larger than all previously chosen phase numbers. It make sure that old operations receive help and complete before new operations are executed.

26. procedure isSearchStillPending(tid ↓, Phase ↓, flag ↑)
27.     ODA curr ← state[tid];
28. if ((curr.type = OpType.WFAddVertex) ∨ (curr.type = OpType.WFRemoveVertex) ∨
        (curr.type = OpType.WFContainsVertex) ∨ (curr.type = OpType.WFAddEdge) ∨ (curr.type =
         OpType.WFRemoveEdge) ∨ (curr.type = OpType.WFContainsEdge) ∧ (curr.Phase =
         Phase.WFContainsEdge)) then
29.     flag ← true;
30. else
31.     flag ← false;
32. end if
33. end procedure

Algorithm 4 WFValidVertex Method: Takes two vertices, \(v_1, v_2\), each of type VNode as input and validates for presence in vertex list and returns true or false.

34. procedure WFValidVertex (v_1 ↓, v_2 ↓, flag ↑)
35. if (read(v_1.marked) = false) ∧ (read(v_2.marked) = false) ∧ (read(v_1.vnext) = v_2) then
36.     flag ← true; // validation successful
37. else
38.     flag ← false; // validation fails
39. end if
40. return; //return flag
41. end procedure
**Algorithm 5** WFLocateVertex Method: Takes $key$ as input and returns the corresponding pair of neighboring $VNode \langle v_1, v_2 \rangle$. Initially $v_1$ and $v_2$ are set to $null$.

```plaintext
procedure WFLOCATEVERTEX (key ↓, phase ↓, v1 ↑, v2 ↑)
    bool snip;
    while (true) do
        $v_1$ ← read(VertexHead);
        $v_2$ ← read($v_1$.vnext);
        /*Find left and right vnode*/
        while (($read(v_2.marked) = false) ∧ ($read(v_2.val) < key)) do
            $v_1$ ← $v_2$;
            $v_2$ ← read($v_2$.vnext);
        end while
        /*Check vnodes are adjacent*/
        if (!WFValidVertex($v_1 ↓, v_2 ↓, flag ↑)) then
            go to Line 44;
        else
            return;
        end if
        /* Remove one or more marked vnodes, by changing the $v_1$.vnext to $v_2$.vnext if $v_2$.vnext is unmarked*/
        snip ← CAS($v_1$.vnext, $v_2$.vnext, $v_2$.vnext, false, false);
        if (!isSearchStillPending(phase)) then
            return NULL; // to ensure wait-freedom
        end if
        if (!snip) then
            go to Line 44;
        else
            return;
        end if
    end while
end procedure
```

**Algorithm 6** WFAddVertex Method: Successfully adds $VNode(key)$ to the vertex list, if it is not present earlier.

```plaintext
procedure WFADDVERTEX (key ↓, flag ↑)
    tid ← ThreadID.get();
    phase ← maxPhase(); // setting the phase number for the
    newv ← new vnode(key);
    ODA op ← new ODA(phase, OpType.WFADDVERTEX, newv, NULL);
    state[tid] ← op; // publish the operation
    HelpGraphDS(phase);
    if (state[tid].type = OpType.success) then
        flag ← true;
    else
        flag ← false;
    end if
end procedure
```
Algorithm 7 HELPAddVertex Method:

procedure HELPAddVertex (Phase ↓)

\[ tid \leftarrow \text{ThreadId.get();} \]
\[ ODA \ op; \]

while (true) do

\[ op \leftarrow \text{state[tid];} \]
\[ \text{if } (! (\text{op.type} = \text{OpType.WFAddVertex} \wedge \text{op.phase} = \text{Phase})) \text{ then} \]
\[ \text{return;} \]
\[ \text{end if} \]
\[ \text{vnode } v_1 \leftarrow \text{op.vn;} // \text{the vnode } vn \text{ to be inserted} \]
\[ \text{vnode } v_2 \leftarrow v_1.vnext; \]
\[ \text{WFLocateVertex}(v_1.val ↓, phase ↓, pred ↑, curr ↑); \]
\[ \text{if } (\text{curr.val} = v_1.val) \text{ then} // \text{failure may have happened} \]
\[ ODA \ success \leftarrow \text{new ODA(Phase, OpType.success, } v_1, \text{NULL);} \]
\[ \text{if } (\text{CAS(state[tid], op, success)}) \text{ then} \]
\[ \text{return;} \]
\[ \text{end if} \]
\[ \text{else} // \text{vnode } nv \text{ not added yet : failure} \]
\[ ODA \ failure \leftarrow \text{new ODA(Phase, OpType.failure, } v_1, \text{NULL);} \]
\[ \text{if } (\text{CAS(state[tid], op, failure)}) \text{ then} \]
\[ \text{return;} \]
\[ \text{end if} \]
\[ \text{end if} \]
\[ \text{else} \]
\[ \text{if } ((\text{curr.vnext.marked} = \text{true})) \text{ then} // \text{already added and then deleted: success} \]
\[ ODA \ success \leftarrow \text{new ODA(Phase, OpType.success, } v_1, \text{NULL);} \]
\[ \text{if } (\text{CAS(state[tid], op, success)}) \text{ then} \]
\[ \text{return;} \]
\[ \text{end if} \]
\[ \text{end if} \]
\[ \text{end if} \]
\[ \text{end while} \]
\[ \text{end procedure} \]
Algorithm 8 WFRmoveVertex Method: \( \text{WNode}(key) \) gets removed from the vertex list if it is already present. Initially \( \text{flag} \) is set to \( \text{true} \).

```plaintext
procedure WFRmoveVertex (\( key \downarrow, \text{flag} \uparrow \))
    \( tid \leftarrow \text{ThreadID.get()} \);
    \( \text{phase} \leftarrow \text{maxPhase()} \); // setting the phase number for the
    \( ODA \text{ op} \leftarrow \text{new ODA(phase, OpType.WFRmoveVertex, new vnode(key), NULL)} \);
    \( \text{state}[tid] \leftarrow \text{op}; \) // publish the operation
    \( \text{help(phase)} \);
    if (\( \text{state}[tid].\text{type} = \text{OpType.success} \)) then
        \( \text{flag} \leftarrow \text{true} \);
    else
        \( \text{flag} \leftarrow \text{false} \);
    end if
end procedure
```

Algorithm 9 HelpRemoveVertex Method:

```plaintext
procedure HelpRemoveVertex (\( \text{Phase} \downarrow \))
    \( tid \leftarrow \text{ThreadID.get()} \);
    \( ODA \text{ op} \);
    \( \text{while (true)} \) do
        \( \text{op} \leftarrow \text{state}[tid] \);
        if (! ((\( \text{op.type} = \text{OpType.WFRmoveVertex} \) \&\& \( \text{op.phase} = \text{Phase} \))) ) then
            \( \text{return} \);
        end if
        \( \text{vnode} \ v_1 \leftarrow \text{op.nv} \);
        \( \text{WFLocateVertex} (v_1.\text{val} \downarrow, \text{phase} \downarrow, \text{pred} \uparrow, \text{curr} \uparrow) \);
        if (\( \text{curr.val} \neq v_1.\text{val} \)) then
            \( \text{ODA failure} \leftarrow \text{new ODA(Phase, OpType.failure, v_1, NULL)} \);
            if (\( \text{CAS(state[tid], op, failure)} \)) then
                \( \text{return} \);
            end if
        end if
        else
/* Changing the \text{curr.marked} to \text{true} if both \text{curr} and \text{pred} are unmarked*/
        if (!((\( \text{CAS(pred.vnext, curr, curr, false, true)} \))) ) then // logically removed
            go to Line 145;
        end if
        \( \text{WFLocateVertex} (v_1.\text{val} \downarrow, \text{phase} \downarrow, \text{pred} \uparrow, \text{curr} \uparrow) \); // physically removed
        \( \text{ODA success} \leftarrow \text{new ODA(Phase, OpType.success, v_1, NULL)} \);
        if (\( \text{CAS(state[tid], op, success)} \)) then
            \( \text{return} \);
        end if
    end while
end procedure
```
Algorithm 10  \textbf{WFContainsVertex} Method: Returns \textit{true} if \texttt{VNode(key)} is present in vertex list and returns \textit{false} otherwise.

\begin{verbatim}
\textbf{procedure} WFContainsVertex(key ↓, flag ↑)
\textbf{tid} ← ThreadID.get();
\textbf{phase} ← maxPhase(); // setting the phase number for the
\textbf{newv} ← new vnode(key);
\textbf{ODA op} ← new ODA(phase, OpType.WFContainsVertex, newv, NULL);
\textbf{state}[\textbf{tid}] ← op; // publish the operation
\textbf{help}(phase);
\textbf{if} (state[\textbf{tid}].type = OpType.success) \textbf{then}
\textbf{flag} ← true;
\textbf{else}
\textbf{flag} ← false;
\textbf{end if}
\textbf{return};
\end{verbatim}

Algorithm 11  \textbf{HelpContainsVertex} Method:

\begin{verbatim}
\textbf{procedure} HelpContainsVertex (Phase ↓)
\textbf{tid} ← ThreadID.get();
\textbf{ODA op};
\textbf{op} ← state[\textbf{tid}];
\textbf{if} (! (op.type = OpType.WFContainsVertex ∧ op.phase = Phase)) \textbf{then}
\textbf{return};
\textbf{end if}
\textbf{vnode v1} ← op.nv;
WFLocateVertex(v1.val ↓, phase ↓, pred ↑, curr ↑);
\textbf{if} ((curr.val = v1.val) ∧ (curr.vnext.marked = false)) \textbf{then}
\textbf{(CAS(state[\textbf{tid}], op, success)); return;}
\textbf{else}
\textbf{ODA failure} ← new ODA(Phase, OpType.failure, v1, NULL);
\textbf{(CAS(state[\textbf{tid}], op, failure)); return;}
\textbf{end if}
\textbf{return};
\textbf{end procedure}
\end{verbatim}

Algorithm 12  \textbf{WFValidEdge} Method: Takes two \texttt{enode e1, e2} and validates for presence in edge list.

\begin{verbatim}
\textbf{procedure} WFValidEdge(e1 ↓, e2 ↓, flag ↑)
\textbf{if} (read(e1.marked) = false) ∧ (read(e2.marked) = false) ∧ (read(e1.enext) = e2)) \textbf{then}
\textbf{flag} ← true; // validation successful
\textbf{else}
\textbf{flag} ← false; //validation fails
\textbf{end if}
\textbf{return}; //return flag
\textbf{end procedure}
\end{verbatim}
Algorithm 13 HelpSearchEdge Method: This method helps to optimise searching of edge vertices. It compare the key1 and key2, starts searching based on smaller key value. Takes two keys, key1 and key2, as input and returns the VNode \((v_1, v_2)\) corresponding to them. Initially \(v_1, v_2\) are set to null and flag is set to true.

```plaintext
procedure WFHelpSearchEdge (key1 ↓, key2 ↓, v1 ↑, v2 ↑, flag ↑)
if (key1 < key2) then
    v1 ← read(VertexHead); // starting from VertexHead
    while (read(v1.val) < key1) do
        v1 ← read(v1.vnext);
    end while
    if (read(v1.val ≠ key1) ∨ (read(v1.marked))) then // VNode(key1) not present or marked
        flag ← false;
        return;
    end if
    v2 ← read(v1.vnext);
    while (read(v2.val) < key2) do
        v2 ← read(v2.vnext);
    end while
    if (read(v2.val ≠ key2) ∨ (read(v2.marked))) then // VNode(key2) not present or marked
        flag ← false;
        return;
    end if
else
    v2 ← read(VertexHead); // starting from VertexHead
    while (read(v2.val) < key2) do
        v2 ← read(v2.vnext);
    end while
    if (read(v2.val ≠ key2) ∨ (read(v2.marked))) then // VNode(key2) not present or marked
        flag ← false;
        return;
    end if
    v1 ← read(n_v.vnext);
    while (read(v1.val) < key1) do
        v1 ← read(v1.vnext);
    end while
    if (read(v1.val ≠ key1) ∨ (read(v1.marked))) then // VNode(key1) not present or marked
        flag ← false;
        return;
    end if
end if
end procedure
```
Algorithm 14 WFLocateEdge Method: Takes two keys, $key_1$ and $key_2$, as input and returns the pair of adjacent ENode $\langle e_1, e_2 \rangle$. If VNode $v_1$ or $v_2$ or ENode $e_2$ is not present, it returns false. Initially enodes $e_1, e_2$ are set to null and flag is set to true.

247: procedure WFLOCATEEDGE ($key_1 \downarrow, key_2 \downarrow, phase \downarrow, e_1 \uparrow, e_2 \uparrow, flag \uparrow$)
248:   bool snip;
249:   HelpSearchEdge($key_1 \downarrow, key_2 \downarrow, v_1 \uparrow, v_2 \uparrow, flag \uparrow$);
250:   if ($flag = false$) then
251:     return;  // $v_1$ or $v_2$ not found, returns flag
252:   end if
253:   if ($read(v_1.marked) \lor read(v_2.marked)$) then
254:     flag ← false;
255:     return;
256:   end if
257:   while (true) do
258:     $e_1 \leftarrow read(v_1.enext)$;
259:     $e_2 \leftarrow read(e_1.enext)$;
260:     /*Find left and right enode */
261:     while (($read(e_2.marked) = false) \land (read(e_2.val) < key_2)$) do
262:       $e_1 \leftarrow e_2$;
263:       $e_2 \leftarrow read(e_2.enext)$;
264:     end while
265:     /*Check enodes are adjacent */
266:     if (!WFValidEdge($e_1 \downarrow, e_2 \downarrow, flag \uparrow$)) then
267:       go to Line 257;
268:     else
269:     return;  // returns true if validation succeeds.
270:   end if
271:   /*Remove one or more marked enodes, by changing the $e_1.enext$ to $e_2.enext$ if $e_2.enext$ is unmarked*/
272:   snip ← CAS($e_1.enext, e_2, e_2.enext, false, false$);
273:   if (!$isSearchStillPending(phase)) then
274:     return NULL;  // to ensure wait-freedom
275:   end if
276:   if (!$snip) then
277:     go to Line 257;
278:   else
279:     return;
280:   end if
281: end while
282: end procedure
Algorithm 15 WFAddEdge Method: \(E\text{Node}(key_2)\) gets added to the edge list of \(V\text{Node}(key_1)\), if it is not present. Initially, \(flag\) is set to true.

```plaintext
283: procedure WFAddEdge (key_1 ↓, key_2 ↓, flag ↑)
284:   tid ← ThreadID.get();
285:   phase ← maxPhase(); // setting the phase number for the
286:   newe_1 ← new enode(key_1);
287:   newe_2 ← new enode(key_2);
288:   ODA op ← new ODA(phase, OpType.WFAddEdge, newe_1, newe_2, NULL);
289:   state[tid] ← op; // publish the operation
290:   WFLocateEdge(key_1 ↓, key_2 ↓, e_1 ↑, e_2 ↑, flag ↑);
291: if (flag = false) then // \(V\text{Node}(key_1)\) or \(V\text{Node}(key_2)\) not found
292:   return;
293: end if
294: help(phase);
295: if (state[tid].type = OpType.success) then
296:   flag ← true;
297: else
298:   flag ← false;
299: end if
300: end procedure
```
Algorithm 16 HelpAddEdge Method:

16

procedure HelpAddEdge (Phase ↓)

301: tid ← ThreadID.get();
302: ODA op;
303: while (true) do
304: op ← state[tid];
305: if (! (op.type = OpType.WFAddEdge ∧ op.phase = Phase)) then
306: return;
307: end if
308: enode e_1 ← op.ne_1;
309: enode e_2 ← op.ne_2;
310: enode e_3 ← e_2.enext;
311: WFLocateEdge(e_1.val ↓ , e_2.val ↓ , phase ↓ , pred ↑ , curr ↑ );
312: if (curr.val = e_2.val) then // failure may have happens
313: if ((curr = e_2) ∨ (curr.enext.marked = true)) then // success
314: ODA success ← new ODA(Phase, OpType.success, e_1, e_2, NULL);
315: if (CAS(state[tid], op, success)) then
316: return;
317: end if
318: else // node ne_2 not added yet : failure
319: ODA failure ← new ODA(Phase, OpType.failure, e_1, e_2, NULL);
320: if (CAS(state[tid], op, failure)) then
321: return;
322: end if
323: end if
324: else
325: if ((curr.enext.marked = true)) then // already added and then deleted: success
326: ODA success ← new ODA(Phase, OpType.success, e_1, e_2, NULL);
327: if (CAS(state[tid], op, success)) then
328: return;
329: end if
330: end if
331: /* the version of the next pointer to avoid the ABA problem, used in later CAS Line 340*/
332: int ver ← pred.enext.version;
333: ODA newOP ← new ODA(Phase, OpType.WFAddVertex, e_1, e_2, NULL);
334: if (! (CAS(state[tid], op, newOP)) then
335: go to Line 304; // operation might have already reported as failure
336: end if
337: CAS(e_2.enext, e_3, curr, false, false);
338: /* is successful: WFAddEdge is linearized here*/
339: if (CAS(pred.enext, ver, e_2.enext, e_2, false, false)) then
340: ODA success ← new ODA(Phase, OpType.success, e_1, e_2, NULL);
341: if (CAS(state[tid], newOP, success)) then
342: return;
343: end if
344: end if
345: end while
346: end procedure
Algorithm 17 WFRemoveEdge Method: \texttt{E\texttt{Node}$(key_2)$} gets removed from the edge list of \texttt{V\texttt{Node}$(key_1)$}, if it is present. Returns successful if the edge is not present earlier.

```plaintext
procedure WFRemoveEdge (key$_1$ ↓, key$_2$ ↓, flag ↑)
  WFLocateEdge(key$_1$ ↓, key$_2$ ↓, e$_1$ ↑, e$_2$ ↑, flag ↑);
  /* \texttt{V\texttt{Node}$(key_1)$} or \texttt{V\texttt{Node}$(key_2)$} not found*/
  if (flag = false) then
    return;
  end if
  tid ← ThreadID.get();
  phase ← maxPhase(); // setting the phase number for the
  ODA op ← new ODA(phase, OpType.WFRemoveEdge, new \texttt{enode}$(key_1)$, new \texttt{enode}$(key_2)$, NULL);
  state[tid] ← op; // publish the operation
  help(phase);
  if (state[tid].type = OpType.success) then
    flag ← true;
  else
    flag ← false;
  end if
end procedure
```

Algorithm 18 HelpRemoveEdge Method:

```plaintext
procedure HelpRemoveEdge (Phase ↓)
  tid ← ThreadID.get();
  ODA op;
  while (true) do
    op ← state[tid];
    if (! (op.type = OpType.WFRemoveEdge ∧ .phase = Phase)) then
      return;
    end if
    enode e$_1$ ← op.ne$_1$;
    enode e$_2$ ← op.ne$_2$;
    WFLocateEdge(e$_1$.val ↓, e$_2$.val ↓, phase ↓, pred ↑, curr ↑);
    if (curr.val ≠ e$_2$.val) then
      ODA failure ← new ODA(Phase, OpType.failure, e$_1$, e$_2$, NULL);
      if (CAS(state[tid], op, failure)) then
        return;
      end if
    end if
    else
      /* Changing the curr.marked to true if both curr and pred are unmarked*/
      if (! (CAS(pred.enext, curr, curr, false, true))) then // logically removed
        go to Line 369;
      end if
    end if
    WFLocateEdge(e$_1$.val ↓, e$_2$.val ↓, phase ↓, pred ↑, curr ↑); // physically removed
    ODA success ← new ODA(Phase, OpType.success, e$_1$, e$_2$, NULL);
    if (CAS(state[tid], op, success)) then
      return;
    end if
  end while
end procedure
```
Algorithm 19 WFContainsEdge Method: Returns true if E\(\text{Node}(key_2)\) is part of the edge list of V\(\text{Node}(key_1)\) and returns false otherwise.

\[
\begin{align*}
\text{procedure} & \quad \text{WFContainsEdge}(key_1 \downarrow, key_2 \downarrow, \text{flag} \uparrow) \\
& \quad \text{WFHelpSearchEdge}(key_1 \downarrow, key_2 \downarrow, v_1 \uparrow, v_2 \uparrow, \text{flag} \uparrow); \\
& \quad /* \text{VNode}(key_1) \text{ or VNode}(key_2) \text{ not found */} \\
& \quad \text{if} (\text{flag} = \text{false}) \text{ then} \\
& \quad \quad \text{return;} \\
& \quad \text{end if} \\
& \quad \text{tid} \leftarrow \text{ThreadID.get}(); \\
& \quad \text{phase} \leftarrow \text{maxPhase}(); // \text{setting the phase number for the} \\
& \quad e_1 \leftarrow \text{new enode}(key_1); \\
& \quad e_2 \leftarrow \text{new enode}(key_2); \\
& \quad \text{ODA op} \leftarrow \text{new ODA(Phase, OpType.WFContainsEdge, } e_1, e_2\text{NULL)}; \\
& \quad \text{state}[\text{tid}] \leftarrow \text{op}; // \text{publish the operation} \\
& \quad \text{help(Phase);} \\
& \quad \text{if} (\text{state}[\text{tid}].\text{type} = \text{OpType.success}) \text{ then} \\
& \quad \quad \text{flag} \leftarrow \text{true;} \\
& \quad \text{else} \\
& \quad \quad \text{flag} \leftarrow \text{false;} \\
& \quad \text{end if} \\
& \text{end procedure}
\end{align*}
\]

Algorithm 20 HelpContainsEdge Method:

\[
\begin{align*}
\text{procedure} & \quad \text{HelpContainsEdge } (\text{Phase} \downarrow) \\
& \quad \text{tid} \leftarrow \text{ThreadID.get}(); \\
& \quad \text{ODA op;} \\
& \quad \text{op} \leftarrow \text{state}[\text{tid}]; \\
& \quad \text{if} (! (\text{op.type} = \text{OpType.WFContainsVertex} \land \text{op.phase} = \text{Phase})) \text{ then} \\
& \quad \quad \text{return;} \\
& \quad \text{end if} \\
& \quad \text{enode } e_1 \leftarrow \text{op.ne1}; \\
& \quad \text{enode } e_2 \leftarrow \text{op.ne2}; \\
& \quad \text{WFLocateEdge}(e_1.\text{val} \downarrow, e_2 \downarrow, \text{phase} \downarrow, \text{pred} \uparrow, \text{curr} \uparrow); \\
& \quad \text{if} ((\text{curr.val} = e_2.\text{val}) \land (\text{curr.enext.marked} = \text{false})) \text{ then} \\
& \quad \quad \text{ODA success} \leftarrow \text{new ODA(Phase, OpType.success, } e_1, e_2, \text{NULL)}; \\
& \quad \quad (\text{CAS(state}[\text{tid}], \text{op, success}) ; \\
& \quad \quad \text{return;} \\
& \quad \text{else} \\
& \quad \quad \text{ODA failure} \leftarrow \text{new ODA(Phase, OpType.failure, } e_1, e_2, \text{NULL)}; \\
& \quad \quad (\text{CAS(state}[\text{tid}], \text{op, failure}) ; \\
& \quad \quad \text{return;} \\
& \quad \text{end if} \\
& \text{end procedure}
\end{align*}
\]

5 Experiments & Analysis

We performed our tests on a workstation 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. The tests were performed in a controlled environment, where we were the sole users of the system. The implementation\textsuperscript{a} has been done in C/C++

\textsuperscript{a}The complete source code of our implementation is available on Github [11].
In the experiments, we start with an initial graph of 1000 vertices. When the program begins, it creates a fixed number of threads (1, 10, 20, 30, 40, 50, 60 and 70) and each thread randomly performs a set of operations chosen by a particular workload distribution. The evaluation metric used is the number of operations completed in a unit time. We measure throughput obtained on running the experiment for 20 seconds. Each data point is obtained by averaging over 5 iterations. We compare the non-blocking graph with its coarse [7], hand-over-hand(HoH) [7], lazy [6] locking and lock-free [4] counterparts.

In the the distribution over the ordered set of operations \{WFAddVertex, WFRemoveVertex, WFContainsVertex, WFAddEdge, WFRemoveEdge, WFContainsEdge\} are (1) Lookup Intensive: (2.5%, 2.5%, 45%, 2.5%, 2.5%, 45%), see the Figure 4a. (2) Equal Lookup and Updates: (12.5%, 12.5%, 25%, 12.5%, 12.5%, 25%), see the Figure 4b. (3) Update Intensive: (22.5%, 22.5%, 5%, 22.5%, 22.5%, 5%), Figure 4c.

In the plots shown in the Figure 4, we observe that the wait-free algorithm is not scalable like lock-free with the number of threads in the system, on the other hand the fast-path-slow-path variant is scalable.

References

[1] Bapi Chatterjee, Sathya Peri, Muktikanta Sa, and Nandini Singhal. A Simple and Practical Concurrent Non-blocking Unbounded Graph with Reachability Queries. CoRR, abs/1809.00896, 2018.

[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 Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, 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 Proceedings of the 15th International Conference on Distributed Computing, DISC ’01, pages 300–314, London, UK, UK, 2001. Springer-Verlag.

[5] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings, pages 300–314, 2001.

[6] 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.

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

[8] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueueers and dequeueers. In PPOPP, pages 223–234. ACM, 2011.
Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. In Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012, pages 141–150, 2012.

Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In SPAA, pages 73–82, 2002.

Sathya Peri, MuktiKanta Sa, and Nandini Singhal. ConcurrentGraphDs. https://github.com/PDCRL/ConcurrentGraphDs, 2017.

Philippe Stellwag, Alexander Ditter, and Wolfgang Schröder-Preikschat. A wait-free queue for multiple enqueuers and multiple dequeuers using local preferences and pragmatic extensions. In SIES, pages 237–248. IEEE, 2009.

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.

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.

Shahar Timnat and Erez Petrank. A practical wait-free simulation for lock-free data structures. In ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ’14, Orlando, FL, USA, February 15-19, 2014, pages 357–368, 2014.

John D. Valois. Lock-free linked lists using compare-and-swap. In Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995, pages 214–222, 1995.

Kunlong Zhang, Yujiao Zhao, Yajun Yang, Yujie Liu, and Michael F. Spear. Practical non-blocking unordered lists. In DISC, volume 8205 of Lecture Notes in Computer Science, pages 239–253. Springer, 2013.