A Simple and Practical Concurrent Non-blocking Unbounded Graph with Linearizable Reachability Queries

Bapi Chatterjee*, Sathya Peri†, Muktikanta Sa†, Nandini Singhal‡
*Institute of Science and Technology Austria, bapi.chatterjee@ist.ac.at
†Dept. of CS&E, IIT Hyderabad, India, {sathya_p, cs15resch11012}@iith.ac.in
‡Microsoft (R&D) Pvt. Ltd., Bangalore, India, nandini12396@gmail.com
Graphs are Everywhere...

Common real world objects can be modeled as graphs, which build the pairwise relations between objects.

Graph algorithms applied in many applications, including social networks, communication networks, VLSI design, graphics, etc.

Often these graphs are dynamic in nature and the updates are real-time.
The System Model

➔ Asynchronous shared-memory model with a finite set of $p$ processors accessed by a finite set of $n$ threads.

➔ The non-faulty threads communicate with each other by invoking methods on the shared objects.

➔ Execution on a shared-memory multi-processor system which supports atomic read, write, fetch-and-add (FAA) and compare-and-swap (CAS) instructions.
A directed graph $G = (V, E)$ represented by its adjacency list which enables it to grow (up to the availability of memory) and sink at the runtime.
The ADT Operations

1. AddVertex (k)
2. RemoveVertex (k)
3. AddEdge (k, l)
4. RemoveEdge (k, l)
5. ContainsVertex (k)
6. ContainsEdge (k, l)
7. GetPath(k, l)

Update Operations

Challenge
Consistency of a traversal with concurrent Updates in the Graph

non-update Operations

Graph Traversal
Consistency Example

T1: AddVertex(k1) = true

T2: RemoveVertex(k2) = true

T3: GetPath(k0, k3) = (... , k2, ...)
The ADT operations implemented by the data structure are represented by their *invocation* and *return* steps.

For an arbitrary concurrent execution of a set of ADT operations should satisfy the consistency framework *linearizability*.

Assign an atomic step as a *linearization point* (LP) inside the execution interval of each of the operations and show that the data structure invariants are maintained across the LPs.

An arbitrary concurrent execution is equivalent to a valid sequential execution obtained by ordering the operations by their LPs.
Linearizability Example

T1: $\text{AddVertex}(k1) = \text{true}$

T2: $\text{AddVertex}(k2) = \text{true}$

T3: $\text{ContainsVertex}(k1) = \text{true}$
A method is **wait-free** if it guarantees that every call finishes its execution in a finite number of steps.

A method is **lock-free** if it guarantees that infinitely often some method call finishes in a finite number of steps.

A method is **obstruction-free** if, from any point after which it executes in isolation, it finishes in a finite number of steps (method call executes in isolation if no other threads take steps).
Add Vertex

AddVertex(4)

Diagram showing the addition of vertex 4 to a graph with vertices labeled -∞, 1, 3, 5, and 7, and edges connecting them with weights.
Add Vertex

Found Location

-∞ 1 3 5 7 ∞

-∞ 3 -∞ 7 ∞

-∞ 5 -∞ 7 ∞

7 -∞ 1 ∞

∞
Add Vertex

Create new VNode
Add Vertex

$nv.vnxt \leftarrow cv$
Add Vertex

CAS(pv.vnxt, cv, nv)
Remove Vertex

RemoveVertex(5)

The diagram shows a graph with vertices labeled from 1 to 7 and edges connecting them with weights. The vertex 5 is being removed, as indicated by the cloud with the label 'RemoveVertex(5)' above it. The remaining graph structure and weights are depicted with arrows and edge weights.
Remove Vertex

Found Location

Diagram showing a graph with vertices labeled from -∞ to ∞, with edges connecting them.
Remove Vertex

Logically Mark the node

LP
Remove Vertex

Physical removal of VNode
Remove Vertex

All incoming and outgoing edges are logically removed.

CAS(pv.vnxt, cv, cv.vnxt)
AddEdge(7,3)
Find and verify the presence of vertices $v(7)$ and $v(3)$ in the vertex-list.
traverse down an edge-list of \( v(7) \) using \textit{locE} and physically removes two kind of logically removed ENodes: (a) logically removed VNode. (b) logically removed ENodes.
Add Edge

AddEdge(7,3)

Create new ENode ne
Add Edge

AddEdge(7,3)

\[ \text{AddEdge(7,3)} \]

\[ \text{ne.enxt} \leftarrow \text{ce} \]
Add Edge

AddEdge(7,3)

ne.ptv ← v(3)
Add Edge

AddEdge(7,3)

```
AddEdge(7,3)
```

```plaintext
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
CAS(pe.enxt, ce, ne)
```

```
AddEdge(7,3)
```

```
Add Edge

AddEdge(7,3)

Increment a counter at v(7)
AddEdge Conflicts with Vertex Modifications

T1: AddEdge\( (k, l) \) = EDGE ADDED

T2: RemoveVertex\( (k) \) = true

T3: AddVertex\( (l) \) = true

locV\( (k) \) = true

locV\( (l) \) = true

AddEdge\( (k, l) \)

Real time Not Possible
Synchronization of AddEdge with Vertex operations

T1: AddEdge(k, l) = EDGE ADDED

locV(k) = true

locV(l) = true verify v(k) and v(l) AddEdge(k, l)

T2: RemoveVertex(k) = true

T3: AddVertex(l) = true
Remove Edge

RemoveEdge(7, 1)
verify the presence of vertices \( v(7) \) and \( v(1) \) in the vertex-list using \texttt{ConVPlus}().

\texttt{RemoveEdge(7,1)}
traverse down an edge-list of \( v(7) \) using \( \text{locE} \) and physically removes two kind of logically removed ENodes:
(a). logically removed VNode.
(b) logically removed ENodes.

\[ \text{RemoveEdge}(7,1) \]
Remove Edge

RemoveEdge(7,1)

Logically Mark the node

LP
Remove Edge

RemoveEdge(7,1)

Physical removal of ENode
CAS(pe.enxt, ce, ce.enxt)
Increment a counter at \( v(7) \)

Remove Edge

RemoveEdge(7,1)
Reachability Query

GetPath(1,7)
Reachability Query

verify the presence of vertices $v(1)$ and $v(7)$ in the vertex-list using \texttt{ConVPlus}().

GetPath(1,7)
Reachability Query

Collect the Path(Single-Collect) using traversal algorithm: BFS

GetPath(1,7)
Problem with Single Collect

\[ t_0: \]

Initial Graph

T1: GetPath(3, 9)

\[ t_1: \]

T1: GetPath(3, 9)  
Starts First-Collect

T2: RemoveEdge(2, 1)

T3: AddEdge(7, 9)

\[ t_2: \]

Current Graph

T1: GetPath(3, 9)  
Returns path
Need Double Collect

$t_0$: Initial Graph

T1: GetPath(3, 9)

$t_1$: T1: GetPath(3, 9) Starts First-Collect

T2: RemoveEdge(2, 1)

T3: AddEdge(7, 9)

$t_2$: T1: GetPath(3, 9) First-Collect

$t_3$: T1: GetPath(3, 9) Second-Collect

OHH!
Still have a problem.

Return NULL
Double Collect Problem

**Initial Graph**

- T1: GetPath(3, 9)
- T2: RemoveEdge(7, 9)

**t₀:**

**t₁:**

- T3: AddEdge(1, 9)

**t₂:**

**T1: After First-Collect**
Graph has been restored

**$t_3$:**

**T4:** AddEdge(7,9)

**Graph has been restored**

**$t_4$:**

**T1:** GetPath(3, 9)

**Starts Second-Collect**

**T5:** RemoveEdge(1,9)

**T2:** RemoveEdge(7,9)
Problem ...

T3: AddEdge(1,9)

T5: RemoveEdge(1,9)

T4: AddEdge(7,9)

T1: After Second-Collections

Graph has been restored
To solve these issues...

1. We take double collect.
2. In each scan we collect BFS-tree which is a partial snapshot.

3. To capture the modifications.
   3.1. We have a counter associated with each vertex.
   3.2. Whenever any edge operations happen the counter incremented.

4. To verify the double collect we compare with BFS-tree alone with counter.
If the both the double collects are same

5.1. We have valid snapshot
5.2. We analyse the valid snapshot for the presence or absence of the path.
Reachability Query

GetPath(1, 7)

AddVertex(4)

RemoveVertex(5)

RemoveEdge(5, 7)

AddEdge(7, 3)
Reachability Query

**Multi-scan**
Stop when two consecutive scans match.

- AddVertex(4)
- RemoveVertex(5)
- RemoveEdge(5,7)
- AddEdge(7,3)
- GetPath(1,7)
Theorem 1: The ADT operations are linearizable.
Theorem 2: The ADT operations are non-blocking:
1. If the set of keys is finite, the operations ContainsVertex and ContainsEdge are wait-free.
2. The operation GetPath is obstruction-free.
3. The operations AddVertex, RemoveVertex, ContainsVertex, AddEdge, RemoveEdge, and ContainsEdge are lock-free.
Experimental Setup

- Intel(R) Xeon(R) E5-2690 v4 CPU containing 56 cores running at 2.60GHz. and each core supports 2 logical threads.
- Implementation has been done in C++ (without any garbage collection) and multi-threaded implementation is based on Posix threads.
- Start experiments, with 1000 vertices and approximately 125000 edges added randomly.
- 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 sequential and coarse-grained counterparts in two separate sets of experiments comprising:
  (a) The ADT operations excluding GetPath.
  (b) All the ADT operations.
Workload Distributions With GetPath

With GetPath: \{\text{AddVertex, RemoveVertex, ContainsVertex, AddEdge, RemoveEdge, ContainsEdge, GetPath}\}
A. Equal Lookup and Updates: (12.5\%, 12.5\%, 24\%, 12.5\%, 12.5\%, 24\%, 2\%)
B. Lookup Intensive: (2\%, 2\%, 45\%, 2\%, 2\%, 45\%, 2\%)
C. Update Intensive: (22.5\%, 22.5\%, 4\%, 22.5\%, 22.5\%, 4\%, 2\%)
Equal Lookup and Updates: (12.5%, 12.5%, 24%, 12.5%, 12.5%, 24%, 2%)
## Conclusion

- A novel concurrent directed Graph data structure represented by its adjacency list which can grow without bound and sink at the runtime.
- A simple and efficient non-blocking implementation of the ADT operations.

- The spotlight of our work is an obstruction-free reachability query.
- Provably all the methods are linearizable.

- We extensively evaluate a sample C/C++ implementation of the algorithm through a number of micro-benchmarks.

- Our experiments show that the proposed algorithm scales 5-7X with the number of threads in commonly available multi-core systems.
For More Information

And the complete source code is available at: https://github.com/PDCRL/ConcurrentGraphDS
Thank You!
References


For More Information

And the complete source code is available at: https://github.com/PDCRL/ConcurrentGraphDS
**Node Structure**

**Edge Node**

class ENode{
  int k;
  ENode enxt;
  VNode ptv;
};

**Vertex Node**

class VNode{
  int k;
  VNode vnxt;
  ENode enxt;
  int VisitedArray[];
  int ecnt;
};

**BFS Node**

class BFSNode{
  VNode n;
  BFSNode nxt;
  BFSNode p;
  Int locCnt;
};
Observations

A. The non-blocking algorithm is scalable with the number of threads in the system.
B. The performance of lock-based algorithm degrades with the increasing number of threads.
C. 5x-7x increase in the throughput in comparison to the sequential and lock-based counterparts.
**Workload Distributions Without GetPath**

**Without GetPath**: \{AddVertex, RemoveVertex, ContainsVertex, AddEdge, RemoveEdge, ContainsEdge\}

A. Equal Lookup and Updates: (12.5%, 12.5%, 25%, 12.5%, 12.5%, 25%)
B. Lookup Intensive: (2.5%, 2.5%, 45%, 2.5%, 2.5%, 45%)
C. Update Intensive: (22.5%, 22.5%, 5%, 22.5%, 22.5%, 5%)
After successfully checking of $v(1)$ and $v(7)$, it performs repeated BFS traversals by invoking the procedure \texttt{Scan}.

During any edge modification operations the atomic counter $\texttt{ecnt}$ of the source vertex is necessarily incremented.