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

Outline of the Presentation

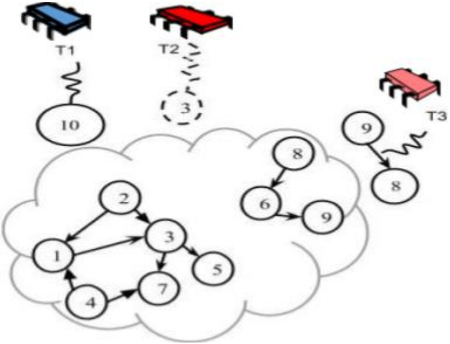
1. Introduction
2. The Data Structure
3. Design of Wait Freedom Algorithm
4. The ADT Operations
    - Part - I : Wait-Free Graph Algorithms
    - Part - II : Optimized Wait-Free Graph Algorithms
5. Correctness and Progress Guarantees
6. Simulation Results

Introduction

- 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.

Introduction

- 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.

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.

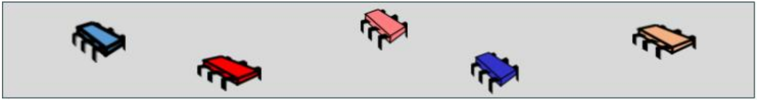
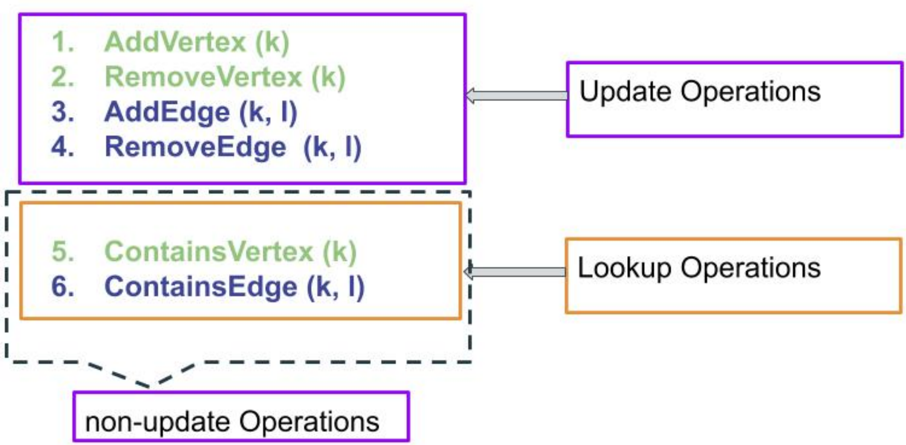


Figure: Concurrent Threads.

The ADT Operations [a]



1. AddVertex (k)
2. RemoveVertex (k)
3. AddEdge (k, l)
4. RemoveEdge (k, l)

Update Operations

5. ContainsVertex (k)
6. ContainsEdge (k, l)

Lookup Operations

non-update Operations

---

[a] Bapi Chatterjee, Sathya Peri, Muktikanta Sa, and Nandini Singhal. *A Simple and Practical Concurrent Non-blocking Unbounded Graph with Linearizable Reachability Queries*,ICDCN 2019.

1. AddVertex (k)
2. RemoveVertex (k)
3. Ad
4. Re

ations

Challenge

Making all ADT Ops to Wait-free

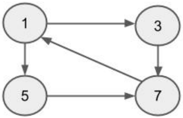
5. Co
6. Co

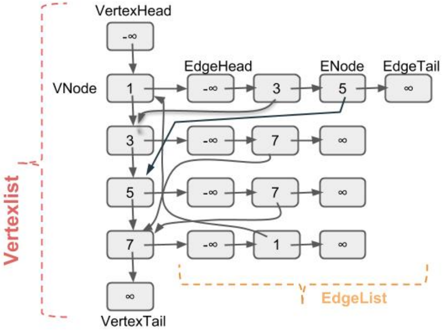
ions

non-update Operations

The Data Structure

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.





Correctness

---

[b] Maurice P. Herlihy and Jeannette M. Wing, *Linearizability: A Correctness Condition for Concurrent Objects*, TOPLAS-1990.

Correctness

- The inconsistency is due to violation of correctness.

---

[b] Maurice P. Herlihy and Jeannette M. Wing, *Linearizability: A Correctness Condition for Concurrent Objects*, TOPLAS-1990.

Correctness

- The inconsistency is due to violation of correctness.
- The correctness-criterion that we consider is *Linearizability*[b].

---

[b] Maurice P. Herlihy and Jeannette M. Wing, *Linearizability: A Correctness Condition for Concurrent Objects*, TOPLAS-1990.

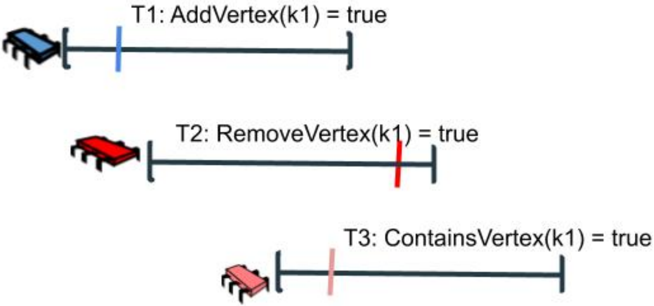
Correctness

- The inconsistency is due to violation of correctness.
- The correctness-criterion that we consider is *Linearizability*[b].
- A concurrent data-strcture *d* is linearizable if for any history (execution) *H* output by *d*:
  - Assign an atomic step as a linearization point (LP) inside the execution interval of each of the operations.
  - The history *H* is equivalent to a valid sequential execution obtained by ordering the operations by their LPs.

---

[b] Maurice P. Herlihy and Jeannette M. Wing, *Linearizability: A Correctness Condition for Concurrent Objects*, TOPLAS-1990.

Linearizability Example



T1: AddVertex(k1) = true

T2: RemoveVertex(k1) = true

T3: ContainsVertex(k1) = true

Progress Guarantee[c]

Wait-free

A method is wait-free if it guarantees that every call finishes its execution in a finite number of steps.

---

[c] Maurice P. Herlihy and Nir Shavit, *On the Nature of Progress*, OPODIS-2011.

Progress Guarantee[c]

Wait-free

A method is wait-free if it guarantees that every call finishes its execution in a finite number of steps.

Lock-free

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

---

[c] Maurice P. Herlihy and Nir Shavit, *On the Nature of Progress*, OPODIS-2011.

Help Mechanism and Difficulties

1. A common technique to achieve wait-freedom.

Help Mechanism and Difficulties

1. A common technique to achieve wait-freedom.
2. Multiple threads should be able to work concurrently on the same operation.

Help Mechanism and Difficulties

1. A common technique to achieve wait-freedom.
2. Multiple threads should be able to work concurrently on the same operation.
3. Many potential races.

Help Mechanism and Difficulties

1. A common technique to achieve wait-freedom.
2. Multiple threads should be able to work concurrently on the same operation.
3. Many potential races.
4. Difficult to design.

Help Mechanism and Difficulties

1. A common technique to achieve wait-freedom.
2. Multiple threads should be able to work concurrently on the same operation.
3. Many potential races.
4. Difficult to design.
5. Usually slower:

Help Mechanism and Difficulties

1. A common technique to achieve wait-freedom.
2. Multiple threads should be able to work concurrently on the same operation.
3. Many potential races.
4. Difficult to design.
5. Usually slower: **At times many threads are attempting to help the same operation.**

Help Mechanism

1. Each operation is assigned a dynamic age-based priority.

Help Mechanism

1. Each operation is assigned a dynamic age-based priority.
2. Each thread declares in a designated **state** array the operation it desires.

Help Mechanism

1. Each operation is assigned a dynamic age-based priority.
2. Each thread declares in a designated **state** array the operation it desires.
3. Many threads may attempt to execute it.

Help Mechanism for Each Thread

- Each thread accessing a concurrent graph data structure.

Help Mechanism for Each Thread

- Each thread accessing a concurrent graph data structure.
    - Chooses a monotonically increasing **phase** number.

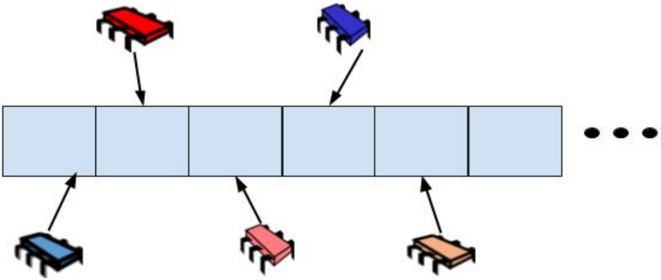
Help Mechanism for Each Thread

- Each thread accessing a concurrent graph data structure.
    - Chooses a monotonically increasing **phase** number.
    - Writes down its phase and operation information in a special **state** array.

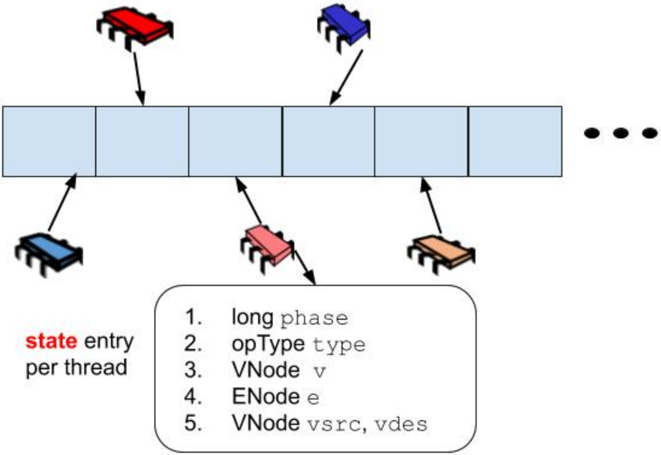
Help Mechanism for Each Thread

- Each thread accessing a concurrent graph data structure.
    - Chooses a monotonically increasing **phase** number.
    - Writes down its phase and operation information in a special **state** array.
    - Helps all threads with a non-larger phase to apply their operations.

State Array



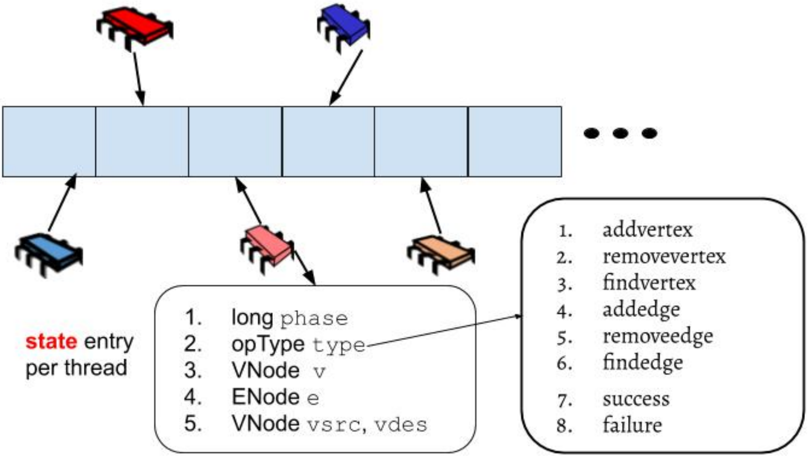
State Array



state entry per thread

1. long `phase`
2. opType `type`
3. VNode `v`
4. ENode `e`
5. VNode `vsrc`, `vdes`

State Array

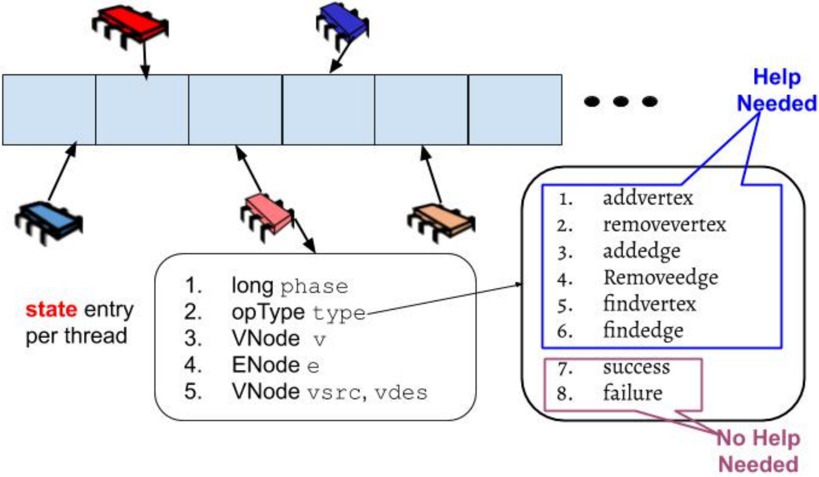


state entry per thread

1. long `phase`
2. opType `type`
3. VNode `v`
4. ENode `e`
5. VNode `vsrc`, `vdes`

1. addvertex
2. removevertex
3. findvertex
4. addedge
5. removeedge
6. findedge
7. success
8. failure

State Array



state entry per thread

1. long `phase`
2. opType `type`
3. VNode `v`
4. ENode `e`
5. VNode `vsrc`, `vdes`

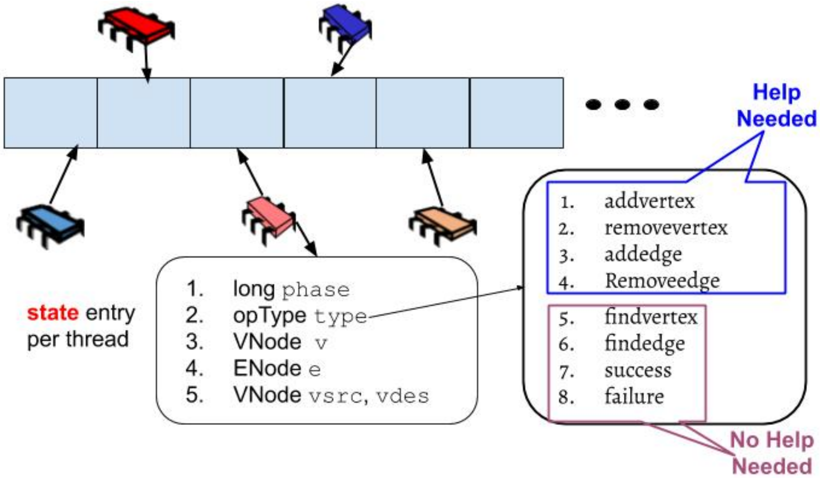
Help Needed

1. addvertex
2. removevertex
3. addedge
4. Removeedge
5. findvertex
6. findedge

No Help Needed

7. success
8. failure

State Array



state entry per thread

1. long `phase`
2. opType `type`
3. VNode `v`
4. ENode `e`
5. VNode `vsrc`, `vdes`

1. addvertex
2. removevertex
3. addedge
4. Removeedge
5. findvertex
6. findedge
7. success
8. failure

Outline

1. Introduction
2. The Data Structure
3. Design of Wait Freedom Algorithm
4. The ADT Operations
    - Part - I : Wait-Free Graph Algorithms
    - Part - II : Optimized Wait-Free Graph Algorithms
5. Correctness and Progress Guarantees
6. Simulation Results

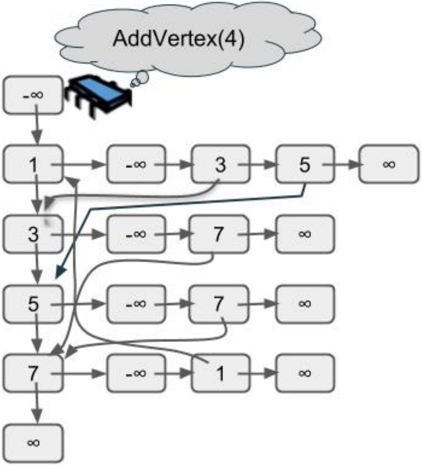
Part - I

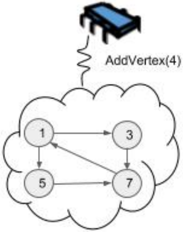
Wait-Free Graph Algorithms

The ADT Operations

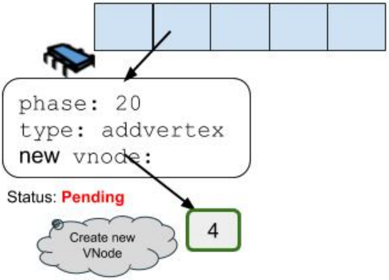
1. AddVertex
2. RemoveVertex
3. ContainsVertex
4. AddEdge
5. RemoveEdge
6. ContainsEdge

Working of AddVertex(u) Operation



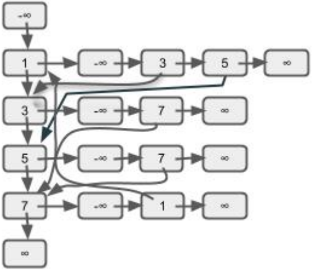


Working of AddVertex(u) Operation

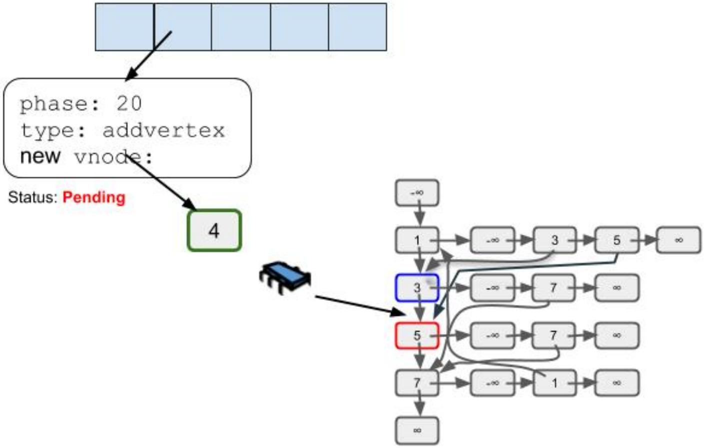


phase: 20
type: addvertex
new vnode:

Status: **Pending**

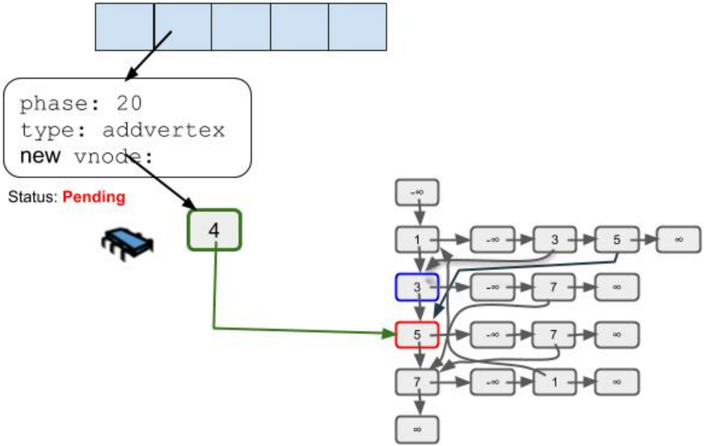


Working of Help AddVertex Method



1. **Locate**
2. Direct
3. Add
4. Report

Working of Help AddVertex Method

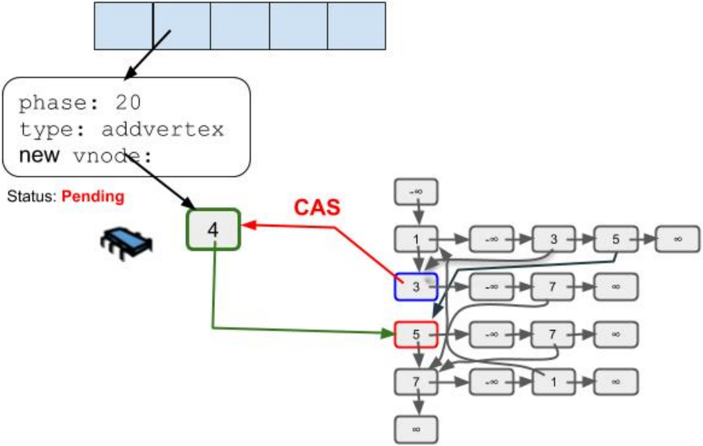


```
phase: 20
type: addvertex
new vnode:
```

Status: **Pending**

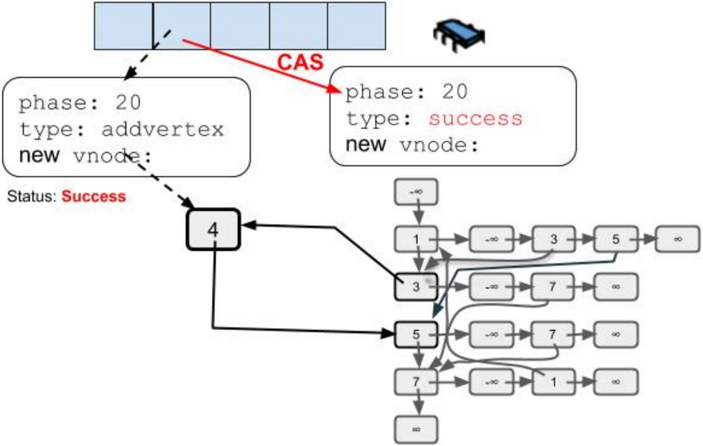
1. Locate
2. **Direct**
3. Add
4. Report

Working of Help AddVertex Method



1. Locate
2. Direct
3. **Add**
4. Report

Working of Help AddVertex Method



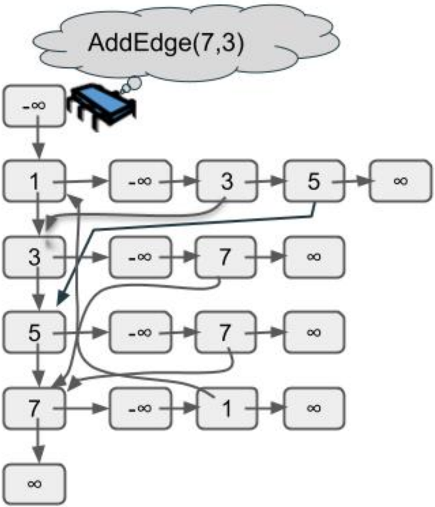
phase: 20
type: addvertex
new vnode:

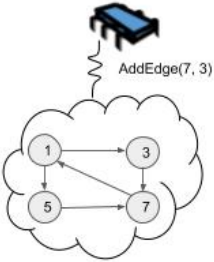
Status: **Success**

phase: 20
type: success
new vnode:

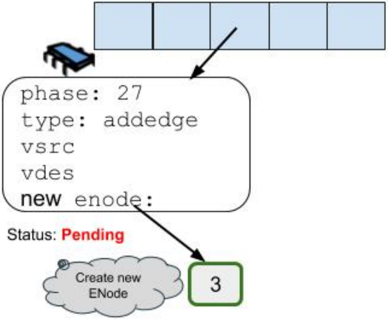
1. Locate
2. Direct
3. Add
4. **Report**

Working of AddEdge(u,v) Operation

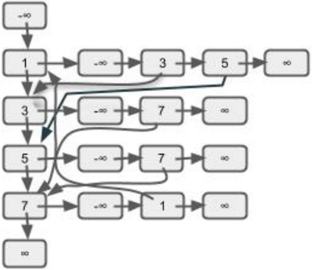




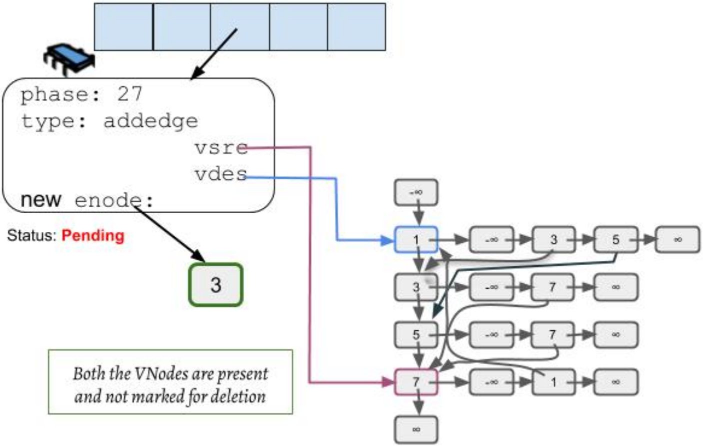
Working of AddEdge(u,v) Operation


```
phase: 27
type: addedge
vsrc
vdes
new enode:
```

Status: **Pending**



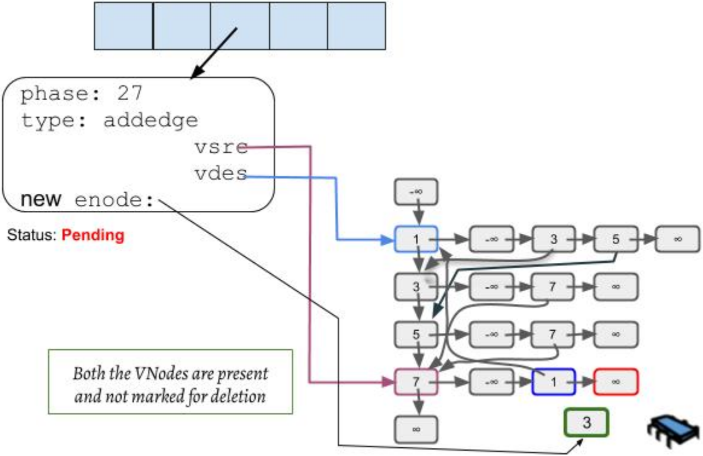
Working of Help AddEge Method



Both the VNodes are present and not marked for deletion

1. **VNode Validations**
2. Locate
3. Direct
4. Add
5. Report

Working of Help AddEge Method



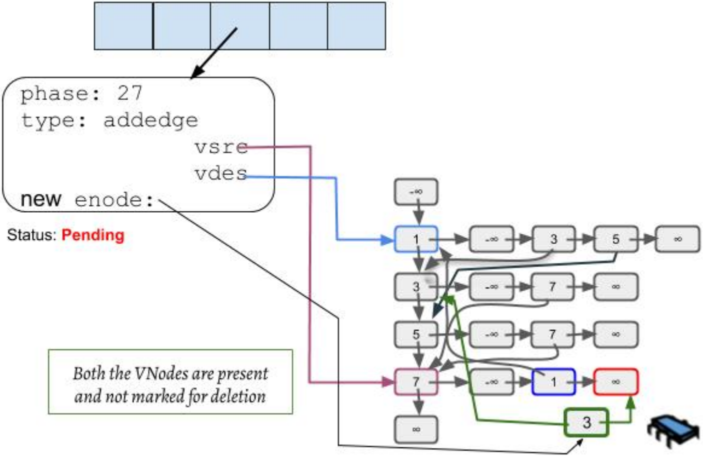
phase: 27
type: addedge
vsrc
vdes
new enode:

Status: **Pending**

Both the VNodes are present and not marked for deletion

1. VNode Validations
2. **Locate**
3. Direct
4. Add
5. Report

Working of Help AddEge Method

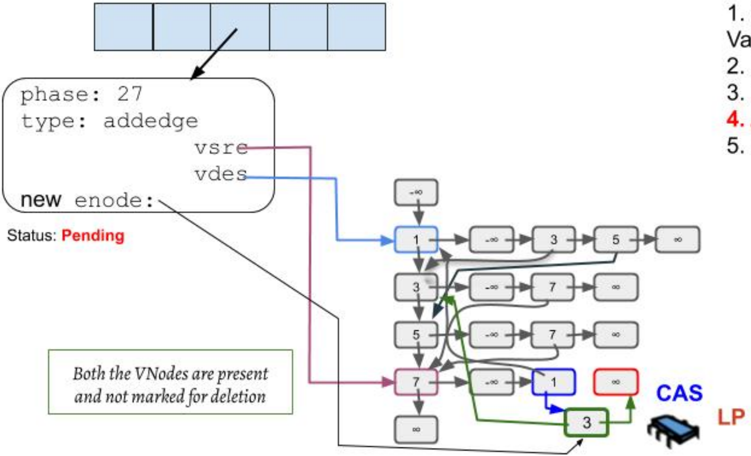


Status: **Pending**

Both the VNodes are present and not marked for deletion

1. VNode Validations
2. Locate
3. **Direct**
4. Add
5. Report

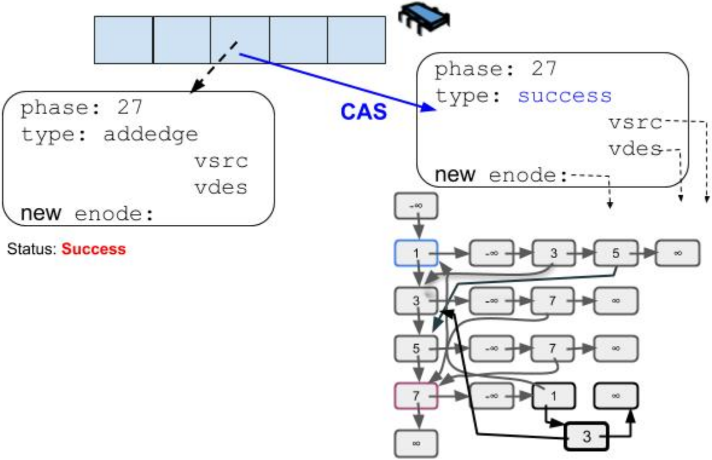
Working of Help AddEge Method



Both the VNodes are present and not marked for deletion

1. VNode Validations
2. Locate
3. Direct
4. **Add**
5. Report

Working of Help AddEge Method



```
phase: 27
type: addedge
            vsrc
            vdes
new enode:
```

Status: **Success**

```
phase: 27
type: success
            vsrc
            vdes
new enode:
```

1. VNode Validations
2. Locate
3. Direct
4. Add
5. **Report**

Outline

1. Introduction
2. The Data Structure
3. Design of Wait Freedom Algorithm
4. The ADT Operations
    - Part - I : Wait-Free Graph Algorithms
    - Part - II : Optimized Wait-Free Graph Algorithms
5. Correctness and Progress Guarantees
6. Simulation Results

Part - II

Optimized Wait-Free Graph Algorithms

Lock-free Vs Wait-free

Lock-free Vs Wait-free

1. Lock-free algorithms:
    - Among all threads trying to apply operations on the data structure, at least one will succeed.
    - Many *scalable* and *efficient* algorithms.
    - Global progress.

Lock-free Vs Wait-free

1. Lock-free algorithms:
    - Among all threads trying to apply operations on the data structure, at least one will succeed.
    - Many *scalable* and *efficient* algorithms.
    - Global progress.
2. Wait-free algorithms:
    - A thread completes its operation a bounded # steps: regardless of what other threads are doing.
    - Particularly important property in several domains e.g., real-time systems and operating systems.
    - Commonly regarded as too *inefficient* and *complicated* to design.
    - The overhead of wait-freedom is because of helping.

Reducing the Overhead of Helping

1. Ask for help only when you really need it.
    - i.e., after trying several times to apply the operation.

Reducing the Overhead of Helping

1. Ask for help only when you really need it.
    - i.e., after trying several times to apply the operation.
2. Help others only after giving them a chance to proceed on their own.
    - delayed helping.

An Optimized Fast Wait-free Graph Algorithm

1. Start operation by running its lock-free implementation.

An Optimized Fast Wait-free Graph Algorithm

1. Start operation by running its lock-free implementation.
    - **Fast path**

An Optimized Fast Wait-free Graph Algorithm

1. Start operation by running its lock-free implementation.
    - **Fast path**
2. Upon several failures, switch into a wait-free implementation

An Optimized Fast Wait-free Graph Algorithm

1. Start operation by running its lock-free implementation.
    - **Fast path**
2. Upon several failures, switch into a wait-free implementation → notify others that you need help

An Optimized Fast Wait-free Graph Algorithm

1. Start operation by running its lock-free implementation.
    - **Fast path**
2. Upon several failures, switch into a wait-free implementation → notify others that you need help → keep trying

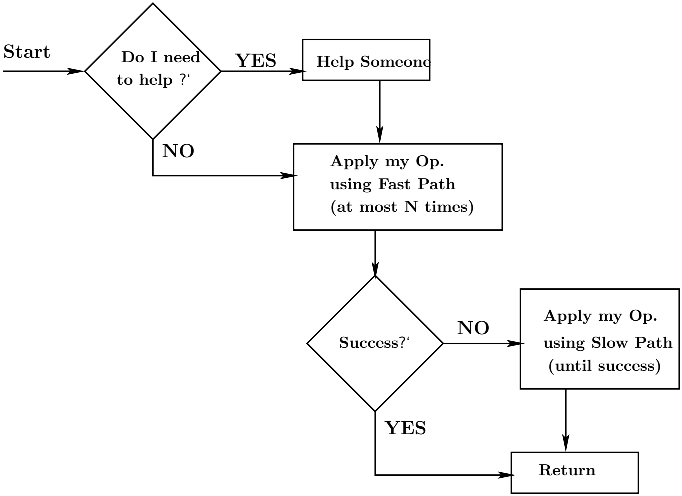
An Optimized Fast Wait-free Graph Algorithm

1. Start operation by running its lock-free implementation.
    - **Fast path**
2. Upon several failures, switch into a wait-free implementation → notify others that you need help → keep trying
    - **Slow path**

An Optimized Fast Wait-free Graph Algorithm

1. Start operation by running its lock-free implementation.
    - **Fast path**
2. Upon several failures, switch into a wait-free implementation → notify others that you need help → keep trying
    - **Slow path**
3. Once in a while, threads on the fast path check if their help is needed and provide help.

Optimized Fast Wait-free Algorithm Framework



Start

Do I need to help ?'

YES

Help Someone

NO

Apply my Op. using Fast Path (at most N times)

Success?'

NO

Apply my Op. using Slow Path (until success)

YES

Return

Correctness and Progress Guarantees

Theorem 1:

The ADT operations are linearizable.

Correctness and Progress Guarantees

Theorem 1:

The ADT operations are linearizable.

Theorem 2:

The ADT operations *AddVertex*, *RemoveVertex*, *ContainsVertex*, *AddEdge*, *RemoveEdge*, and *ContainsEdge* are Wait-free.

Correctness and Progress Guarantees

Theorem 1:

The ADT operations are linearizable.

Theorem 2:

The ADT operations *AddVertex*, *RemoveVertex*, *ContainsVertex*, *AddEdge*, *RemoveEdge*, and *ContainsEdge* are Wait-free.

Proofs of the Theorem 1 and 2 are shown in the paper.

Experiments

Experiments

- Intel(R) Xeon(R) E5-2690 v4 CPU containing 14 cores running at 2.60GHz on two sockets. Each core supports 2 logical threads.
  - Thus, a total of 56 logical cores.
- Implementation in C++ without any garbage collection. Multi-threaded implementation is based on Posix threads.

Experiments

- Intel(R) Xeon(R) E5-2690 v4 CPU containing 14 cores running at 2.60GHz on two sockets. Each core supports 2 logical threads.
    - Thus, a total of 56 logical cores.
- Implementation in C++ without any garbage collection. Multi-threaded implementation is based on Posix threads.
- Start experiments, with 1000 vertices and approximately 125000 edges added randomly.

Experiments

- Intel(R) Xeon(R) E5-2690 v4 CPU containing 14 cores running at 2.60GHz on two sockets. Each core supports 2 logical threads.
  - Thus, a total of 56 logical cores.
- Implementation in C++ without any garbage collection. 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.

Experiments

- Intel(R) Xeon(R) E5-2690 v4 CPU containing 14 cores running at 2.60GHz on two sockets. Each core supports 2 logical threads.
    - Thus, a total of 56 logical cores.
- Implementation in C++ without any garbage collection. 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 wit-free graph with its sequential, coarse-grained, hand-over-hand, lazy and lock-free graphs counterparts.

Workload Distributions

Graph Operations: AddVertex, RemoveVertex, ContainsVertex, AddEdge, RemoveEdge and ContainsEdge

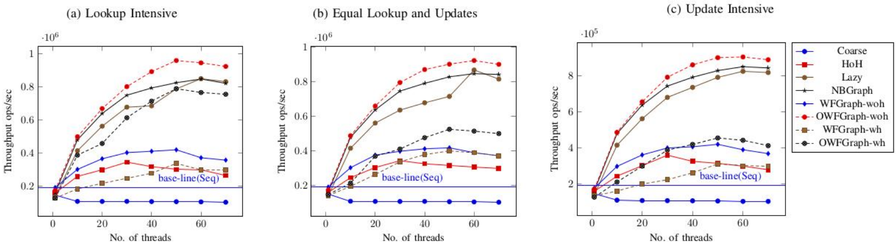
- **Lookup Intensive:** (2.5%, 2.5%, 45%, 2.5%, 2.5%, 45%)
- **Equal Lookup and Updates:** (12.5%, 12.5%, 25%,12.5%, 12.5%, 25%)
- **Update Intensive:** (22.5%, 22.5%, 5%, 22.5%, 22.5%, 5%)

Results

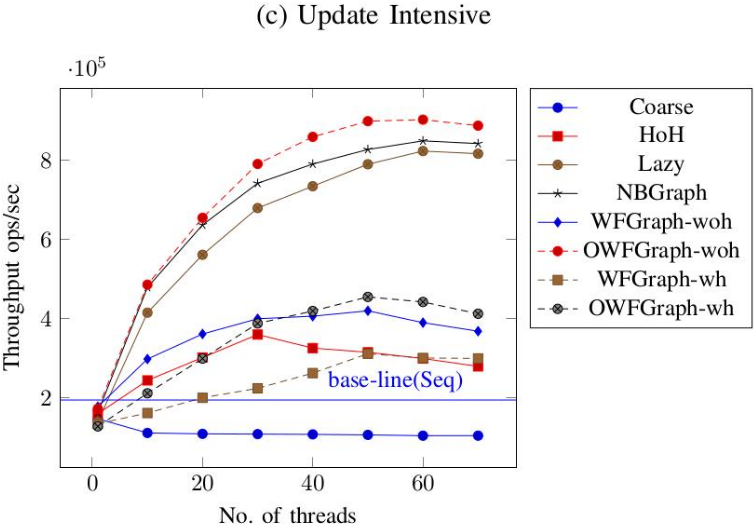
We have compared the following cases.

S. No	Label	Explanation
1	Seq	Sequential execution of all the operations
2	Coarse	Execution with a coarse grained lock [Ch. 9, AMP Book]
3	HoH	Execution with Hand-over-Hand lock [Ch. 9, AMP Book]
4	Lazy	Execution with Lazy-lock [Heller's Lazy List]
5	NBGraph	Based on non-blocking graph [Chatterjee's Non-blocking Graph]
6	WFGraph-woh	wait-free graph without helping of CONTAINSVERTEX & CONTAINSEDGE.
7	WFGraph-wh	wait-free graph with helping of CONTAINSVERTEX & CONTAINSEDGE.
8	OWFGraph-woh	Optimized wait-free graph without helping of CONTAINSVERTEX & CONTAINSEDGE.
9	OWFGraph-wh	Optimized wait-free graph with helping of CONTAINSVERTEX & CONTAINSEDGE.

Results


(a) Lookup Intensive
(b) Equal Lookup and Updates
(c) Update Intensive
$\cdot 10^6$
$\cdot 10^6$
$\cdot 10^5$
Throughput ops/sec
No. of threads
0
20
40
60
0.2
0.4
0.6
0.8
1
2
4
6
8
base-line(Seq)
Coarse
HoH
Lazy
NBGraph
WFGraph-woh
OWFGraph-woh
WFGraph-wh
OWFGraph-wh

Results


(c) Update Intensive
$\cdot 10^5$
Throughput ops/sec
8
6
4
2
0
20
40
60
No. of threads
base-line(Seq)
Coarse
HoH
Lazy
NBGraph
WFGraph-woh
OWFGraph-woh
WFGraph-wh
OWFGraph-wh

Conclusion

Conclusion

1. A practical wait-free directed graph data structure represented by its adjacency list which can grow without bound and sink at the runtime.
2. Provably all the methods are linearizable.

Conclusion

1. A practical wait-free directed graph data structure represented by its adjacency list which can grow without bound and sink at the runtime.
2. Provably all the methods are linearizable.
3. We implemented in a dynamic setting with threads helping each other using operator descriptors.
4. We also extended the wait-free graph to enhance the performance and achieve a fast wait-free graph: optimized wait-free graph.

Conclusion

1. A practical wait-free directed graph data structure represented by its adjacency list which can grow without bound and sink at the runtime.
2. Provably all the methods are linearizable.
3. We implemented in a dynamic setting with threads helping each other using operator descriptors.
4. We also extended the wait-free graph to enhance the performance and achieve a fast wait-free graph: optimized wait-free graph.
5. We extensively evaluate a sample C++ implementation of the algorithm through a number of micro-benchmarks.
6. Our experimental results show on an average of 9x improvement over the sequential implementation.

For More Information

1. The Technical Report is available at:
   https://arxiv.org/abs/1810.13325

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

Thank You!

For Further Reading..

- Chatterjee B. et al. *A Simple and Practical Concurrent Non-blocking Unbounded Graph with Linearizable Reachability Queries*. Proceedings of the 20th International Conference on Distributed Computing and Networking, ICDCN 2019
- Maurice P. et al. *Linearizability: A Correctness Condition for Concurrent Objects*. ACM Transactions on Programming Languages and Systems, Vol. 12, No. 3, July 1990, Pages 463-492.
- Y. Riany. et al. *Towards a practical snapshot algorithm*. Theoretical Computer Science, 269(1-2): 163-201, 2001.
- Timothy L. Harris. *A Pragmatic Implementation of Non-blocking Linked-Lists*. Distributed Computing, 15th International Conference, DISC 2001.
- Maurice Herlihy and Nir Shavit. *The Art of Multiprocesor Programming, Revised Print*. Imprinted Morgan Kaufmann,Elsevier, May 2012.
- A. Natarajan and N. Mittal, *Fast concurrent lock-free binary search trees* 19^{th} PPoPP, 2014, pp. 317–328.
- Arnab Sinha, Sharad Malik, *Runtime checking of serializability in software transactional memory*, Parallel & Distributed Processing (IPDPS), 2010
- Khanh Do Ba, *Wait-Free and Obstruction-Free Snapshot*, Dartmouth Computer Science Technical Report TR2006-578, June 2006.
- Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt and N. Shavit. *Atomic snapshots of shared memory*. Proc. ACM PODC , 1–14, 1990.
- Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, Nir Shavit. *A Lazy Concurrent List-Based Set Algorithm*. Parallel Processing Letters, volume 17, 4, 411–424, 2007,