

# A Shape Analysis for Optimizing Parallel Graph Programs

Dimitrios Proutzos<sup>1</sup>

Roman Manevich<sup>2</sup>

Keshav Pingali<sup>1,2</sup>

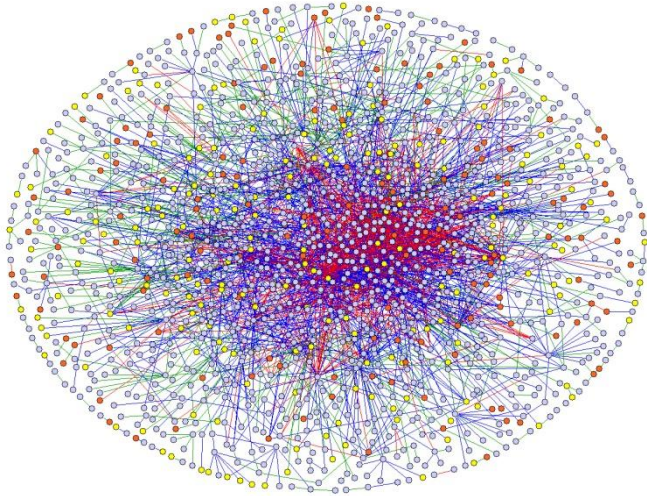
Kathryn S. McKinley<sup>1</sup>

1: Department of Computer Science, The University of Texas at Austin

2: Institute for Computational Engineering and Sciences, The University of  
Texas at Austin

# Motivation

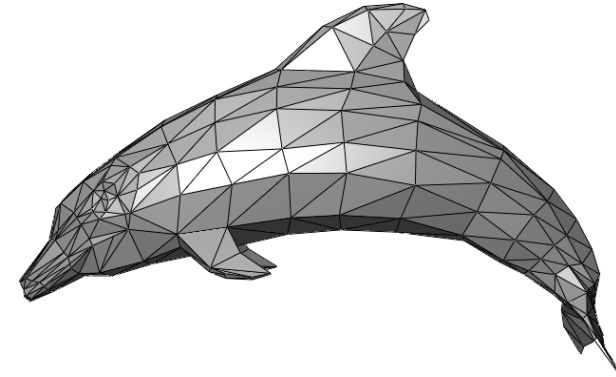
- Graph algorithms are ubiquitous



Computational  
biology



Social Networks




Computer  
Graphics

- **Goal:** Compiler analysis for optimization of parallel graph algorithms



# Organization

- Parallelization of graph algorithms in Galois system
  - Speculative execution
  - Example: Boruvka MST algorithm
- Optimization opportunities
  - Reduce speculation overheads
  - Analysis problem: LockSet shape analysis
- Lockset shape analysis
  - Abstract Data Type (ADT) modeling
  - Hierarchy summarization abstraction
  - Predicate discovery
- Evaluation
  - Fast and infers all available optimizations
  - Optimizations give speedup up to 12x




# Organization

- Parallelization of graph algorithms in Galois system
  - Speculative execution
  - Example: Boruvka MST algorithm
- Optimization opportunities
  - Reduce speculation overheads
  - Analysis problem: LockSet shape analysis 
- Lockset shape analysis
  - Abstract Data Type (ADT) modeling
  - Hierarchy summarization abstraction
  - Predicate discovery
- Evaluation
  - Fast and infers all available optimizations
  - Optimizations give speedup up to 12x





# Organization

- Parallelization of graph algorithms in Galois system
  - Speculative execution
  - Example: Boruvka MST algorithm
- Optimization opportunities
  - Reduce speculation overheads
  - Analysis problem: LockSet shape analysis 
- Lockset shape analysis
  - Abstract Data Type (ADT) modeling
  - Hierarchy summarization abstraction 
  - Predicate discovery
- Evaluation
  - Fast and infers all available optimizations
  - Optimizations give speedup up to 12x






# Organization

- Parallelization of graph algorithms in Galois system
  - Speculative execution
  - Example: Boruvka MST algorithm
- Optimization opportunities
  - Reduce speculation overheads
  - Analysis problem: LockSet shape analysis 
- Lockset shape analysis
  - Abstract Data Type (ADT) modeling 
  - Hierarchy summarization abstraction 
  - Predicate discovery
- Evaluation
  - Fast and infers all available optimizations
  - Optimizations give speedup up to 12x

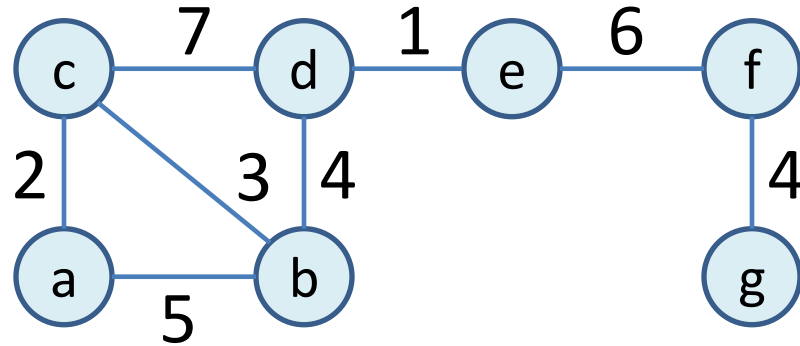
# Organization

- Parallelization of graph algorithms in Galois system
  - Speculative execution
  - Example: Boruvka MST algorithm
- Optimization opportunities
  - Reduce speculation overheads
  - Analysis problem: LockSet shape analysis 
- Lockset shape analysis
  - Abstract Data Type (ADT) modeling 
  - Hierarchy summarization abstraction 
  - Predicate discovery
- Evaluation 
  - Fast and infers all available optimizations
  - Optimizations give speedup up to 12x

# Organization

- Parallelization of graph algorithms in Galois system
  - Speculative execution
  - Example: Boruvka MST algorithm
- Optimization opportunities
  - Reduce speculation overheads
  - Analysis problem: LockSet shape analysis 
- Lockset shape analysis
  - Abstract Data Type (ADT) modeling 
  - Hierarchy summarization abstraction 
  - Predicate discovery
- Evaluation
  - Fast and infers all available optimizations 
  - Optimizations give speedup up to 12x 

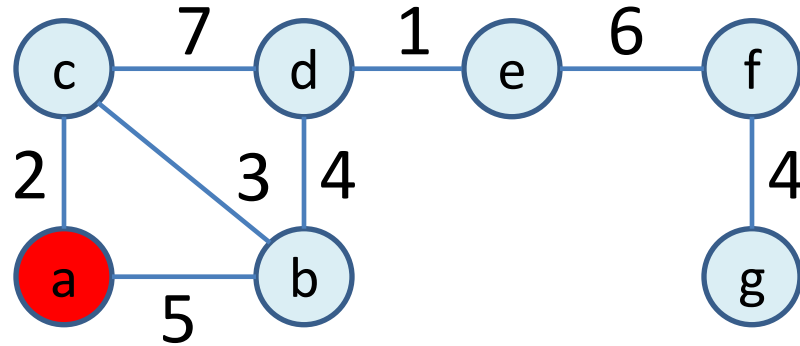
# Boruvka's Minimum Spanning Tree Algorithm



## Build MST bottom-up

```
repeat {  
  pick arbitrary node 'a'  
  merge with lightest neighbor 'lt'  
  add edge 'a-lt' to MST  
} until graph is a single node
```

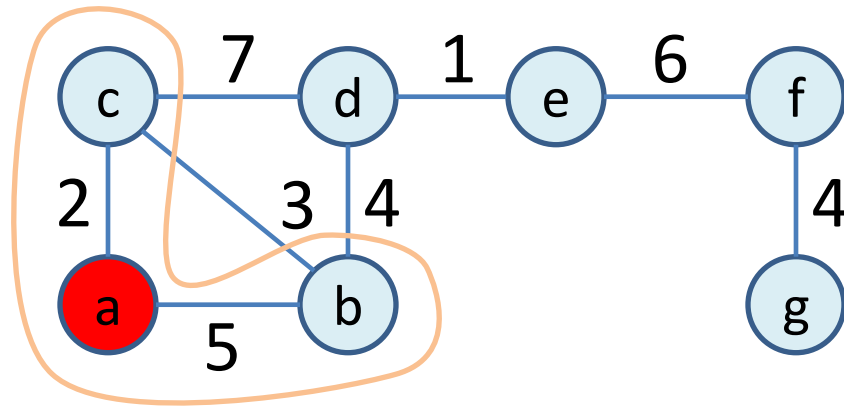
# Boruvka's Minimum Spanning Tree Algorithm



## Build MST bottom-up

```
repeat {  
    pick arbitrary node 'a'  
    merge with lightest neighbor 'lt'  
    add edge 'a-lt' to MST  
} until graph is a single node
```

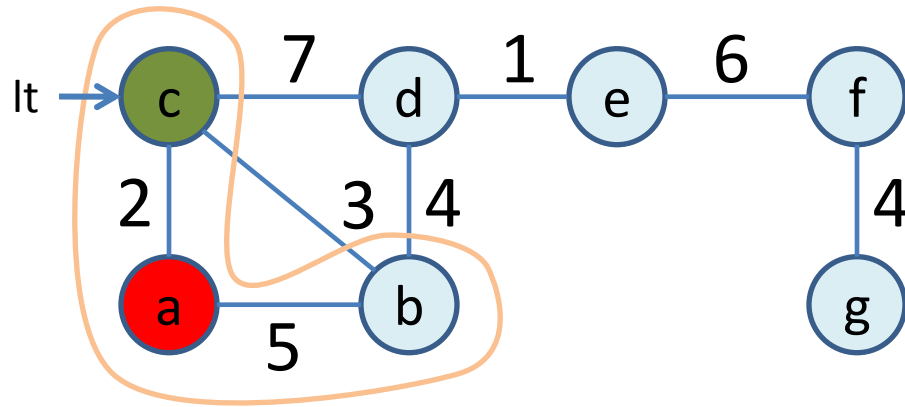
# Boruvka's Minimum Spanning Tree Algorithm



## Build MST bottom-up

```
repeat {  
  pick arbitrary node 'a'  
  merge with lightest neighbor 'lt'  
  add edge 'a-lt' to MST  
} until graph is a single node
```

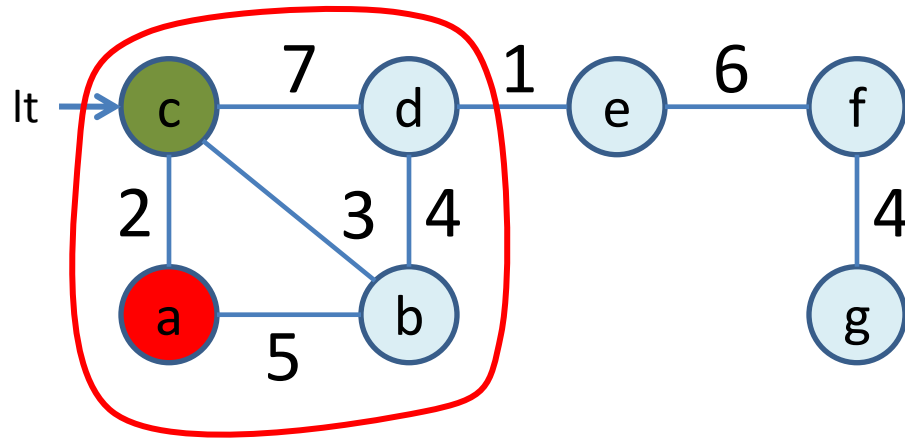
# Boruvka's Minimum Spanning Tree Algorithm



## Build MST bottom-up

```
repeat {  
    pick arbitrary node 'a'  
    merge with lightest neighbor 'lt'  
    add edge 'a-lt' to MST  
} until graph is a single node
```

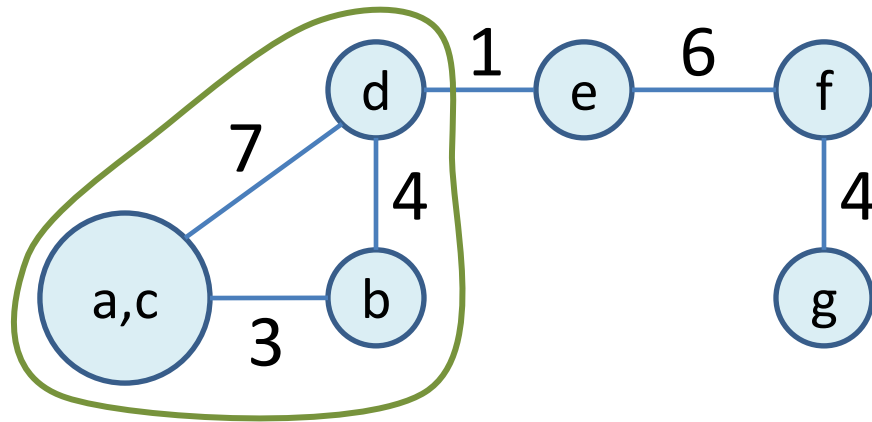
# Boruvka's Minimum Spanning Tree Algorithm



## Build MST bottom-up

```
repeat {  
  pick arbitrary node 'a'  
  merge with lightest neighbor 'lt'  
  add edge 'a-lt' to MST  
} until graph is a single node
```

# Boruvka's Minimum Spanning Tree Algorithm



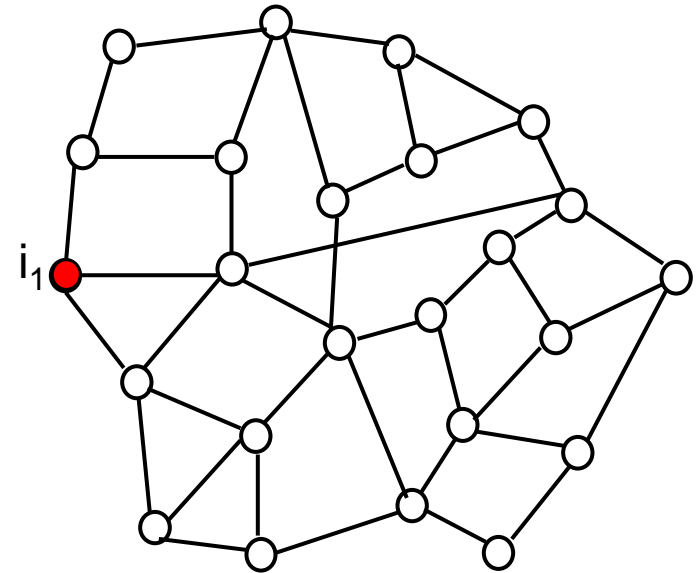
## Build MST bottom-up

```
repeat {  
  pick arbitrary node 'a'  
  merge with lightest neighbor 'lt'  
  add edge 'a-lt' to MST  
} until graph is a single node
```

# Parallelism in Boruvka

- Algorithm = repeated application of operator to graph

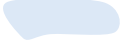
- Active node: ●
  - Node where computation is needed
- Activity:
  - Application of operator to active node
- Neighborhood:  
  - Sub-graph read/written to perform activity
- Unordered algorithms:
  - Active nodes can be processed in any order

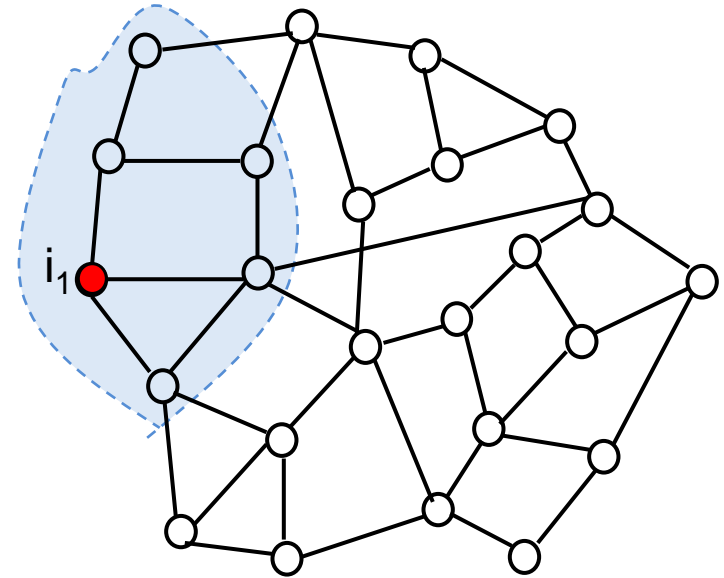


- Amorphous data-parallelism
  - Parallel execution of activities, subject to neighborhood constraints
- Neighborhoods are functions of runtime values
  - Parallelism cannot be uncovered at compile time in general

# Parallelism in Boruvka

- Algorithm = repeated application of operator to graph

- Active node: ●
  - Node where computation is needed
- Activity:
  - Application of operator to active node
- Neighborhood: 
  - Sub-graph read/written to perform activity
- Unordered algorithms:
  - Active nodes can be processed in any order

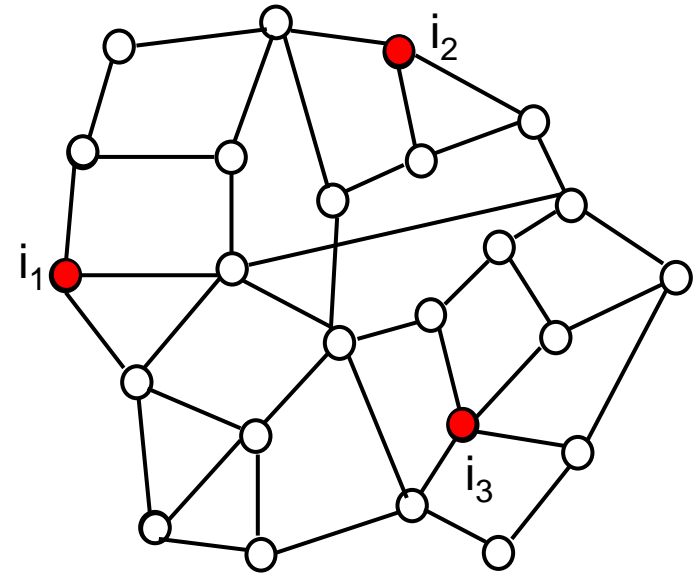


- Amorphous data-parallelism
  - Parallel execution of activities, subject to neighborhood constraints
- Neighborhoods are functions of runtime values
  - Parallelism cannot be uncovered at compile time in general

# Parallelism in Boruvka

- Algorithm = repeated application of operator to graph

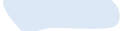
- Active node: ●
  - Node where computation is needed
- Activity:
  - Application of operator to active node
- Neighborhood:  
  - Sub-graph read/written to perform activity
- Unordered algorithms:
  - Active nodes can be processed in any order

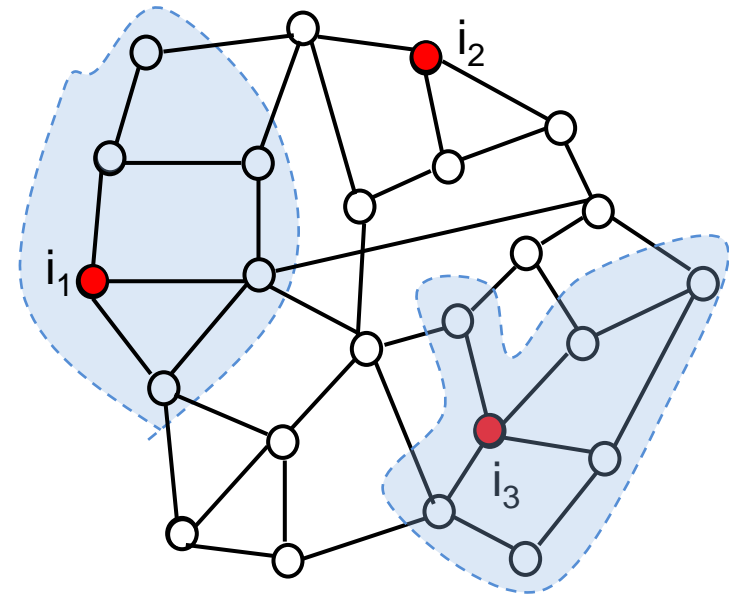


- Amorphous data-parallelism
  - Parallel execution of activities, subject to neighborhood constraints
- Neighborhoods are functions of runtime values
  - Parallelism cannot be uncovered at compile time in general

# Parallelism in Boruvka

- Algorithm = repeated application of operator to graph

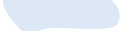
- Active node: ●
  - Node where computation is needed
- Activity:
  - Application of operator to active node
- Neighborhood: 
  - Sub-graph read/written to perform activity
- Unordered algorithms:
  - Active nodes can be processed in any order

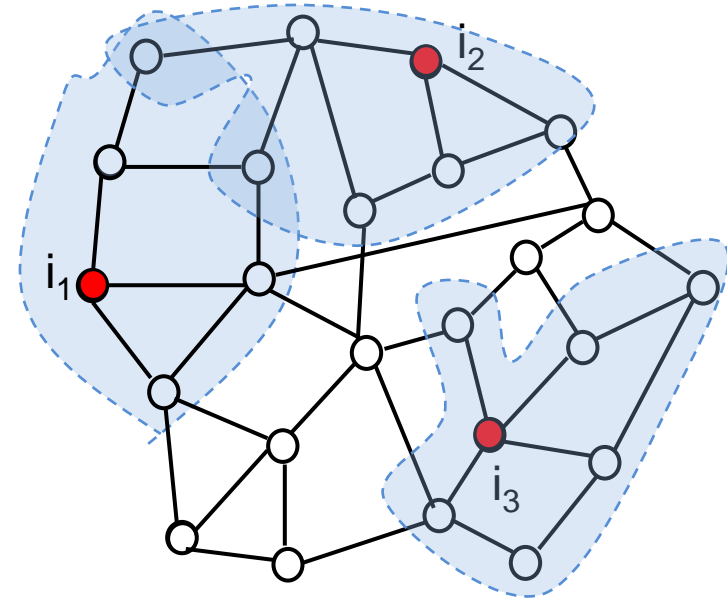


- Amorphous data-parallelism
  - Parallel execution of activities, subject to neighborhood constraints
- Neighborhoods are functions of runtime values
  - Parallelism cannot be uncovered at compile time in general

# Parallelism in Boruvka

- Algorithm = repeated application of operator to graph

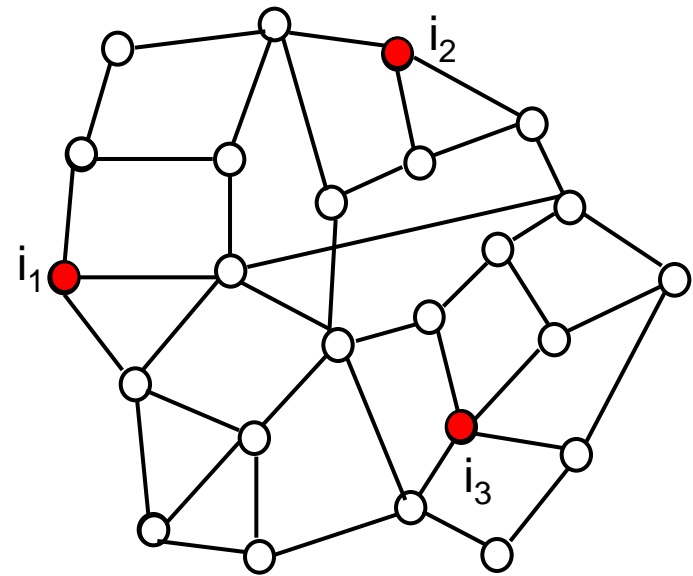
- Active node: ●
  - Node where computation is needed
- Activity:
  - Application of operator to active node
- Neighborhood: 
  - Sub-graph read/written to perform activity
- Unordered algorithms:
  - Active nodes can be processed in any order



- Amorphous data-parallelism
  - Parallel execution of activities, subject to neighborhood constraints
- Neighborhoods are functions of runtime values
  - Parallelism cannot be uncovered at compile time in general

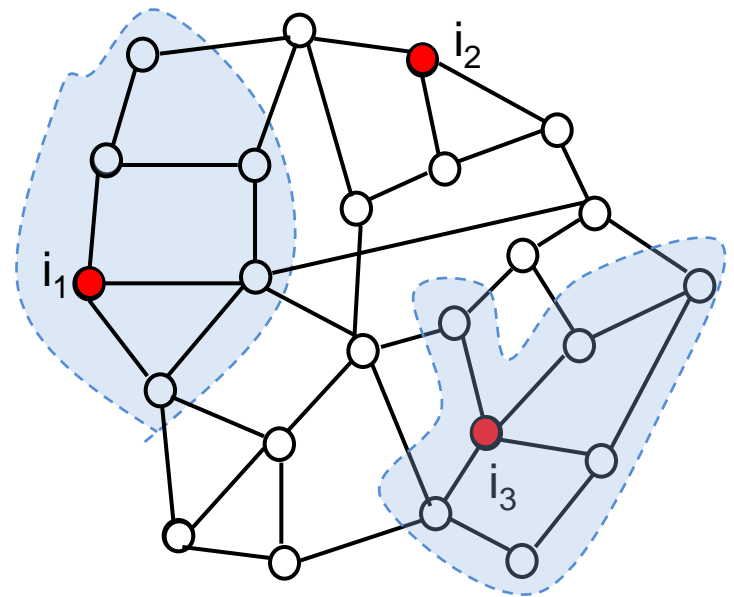
# Optimistic Parallelization in Galois

- Programming model
  - Client code has sequential semantics
  - Library of concurrent data structures
- Parallel execution model
  - Thread-level speculation (TLS)
  - Activities executed speculatively
- Conflict detection
  - Each node/edge has associated **exclusive** lock
  - Graph operations acquire locks on read/written nodes/edges
  - Lock owned by another thread → conflict → iteration rolled back
  - All locks released at the end
- Two main overheads
  - Locking
  - Undo actions



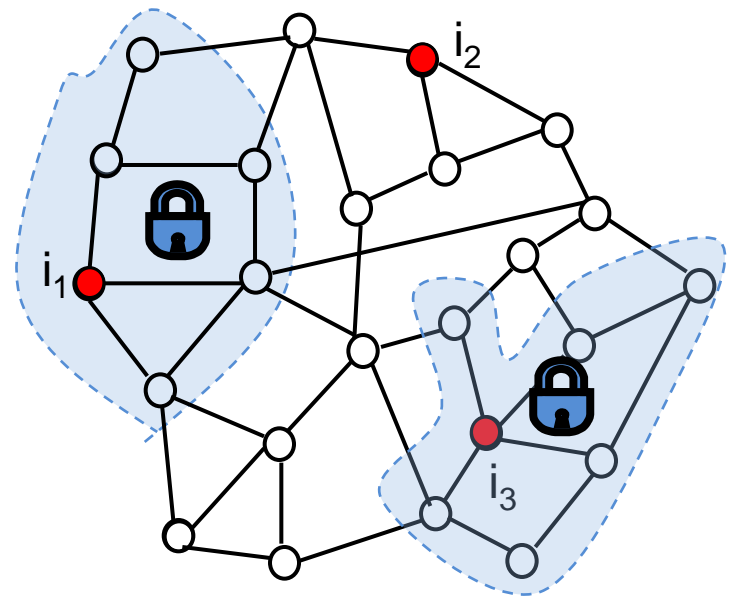
# Optimistic Parallelization in Galois

- Programming model
  - Client code has sequential semantics
  - Library of concurrent data structures
- Parallel execution model
  - Thread-level speculation (TLS)
  - Activities executed speculatively
- Conflict detection
  - Each node/edge has associated **exclusive** lock
  - Graph operations acquire locks on read/written nodes/edges
  - Lock owned by another thread → conflict → iteration rolled back
  - All locks released at the end
- Two main overheads
  - Locking
  - Undo actions



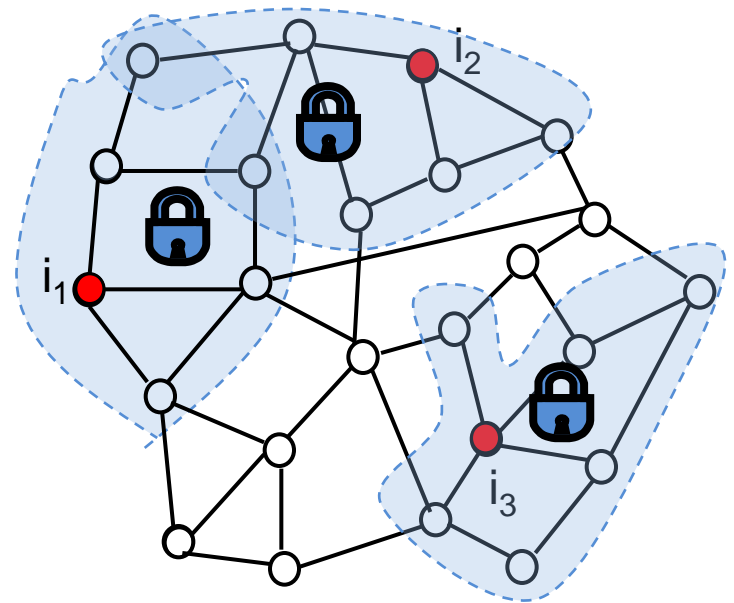
# Optimistic Parallelization in Galois

- Programming model
  - Client code has sequential semantics
  - Library of concurrent data structures
- Parallel execution model
  - Thread-level speculation (TLS)
  - Activities executed speculatively
- Conflict detection
  - Each node/edge has associated **exclusive** lock
  - Graph operations acquire locks on read/written nodes/edges
  - Lock owned by another thread → conflict → iteration rolled back
  - All locks released at the end
- Two main overheads
  - Locking
  - Undo actions



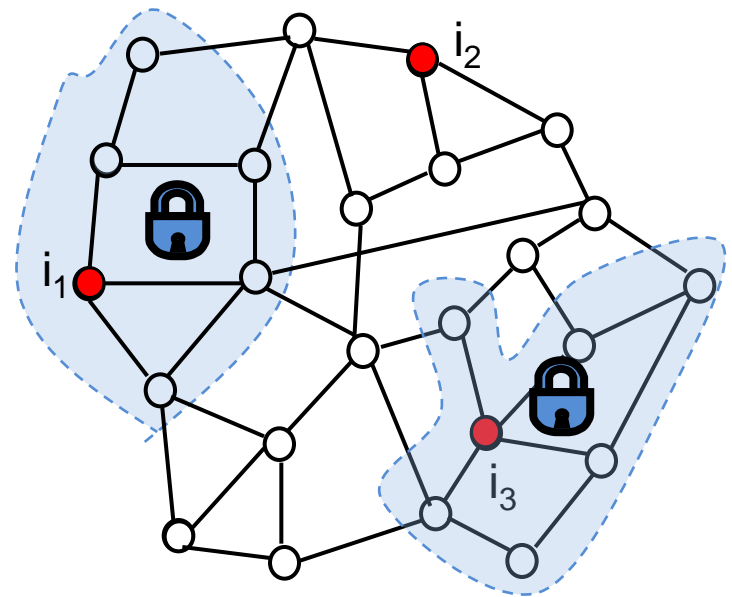
# Optimistic Parallelization in Galois

- Programming model
  - Client code has sequential semantics
  - Library of concurrent data structures
- Parallel execution model
  - Thread-level speculation (TLS)
  - Activities executed speculatively
- Conflict detection
  - Each node/edge has associated **exclusive** lock
  - Graph operations acquire locks on read/written nodes/edges
  - Lock owned by another thread → conflict → iteration rolled back
  - All locks released at the end
- Two main overheads
  - Locking
  - Undo actions



# Optimistic Parallelization in Galois

- Programming model
  - Client code has sequential semantics
  - Library of concurrent data structures
- Parallel execution model
  - Thread-level speculation (TLS)
  - Activities executed speculatively
- Conflict detection
  - Each node/edge has associated **exclusive** lock
  - Graph operations acquire locks on read/written nodes/edges
  - Lock owned by another thread → conflict → iteration rolled back
  - All locks released at the end
- Two main overheads
  - Locking
  - Undo actions



# Overheads (I): Locking

- Optimizations
  - Redundant locking elimination
  - Lock removal for iteration private data
  - Lock removal for lock domination

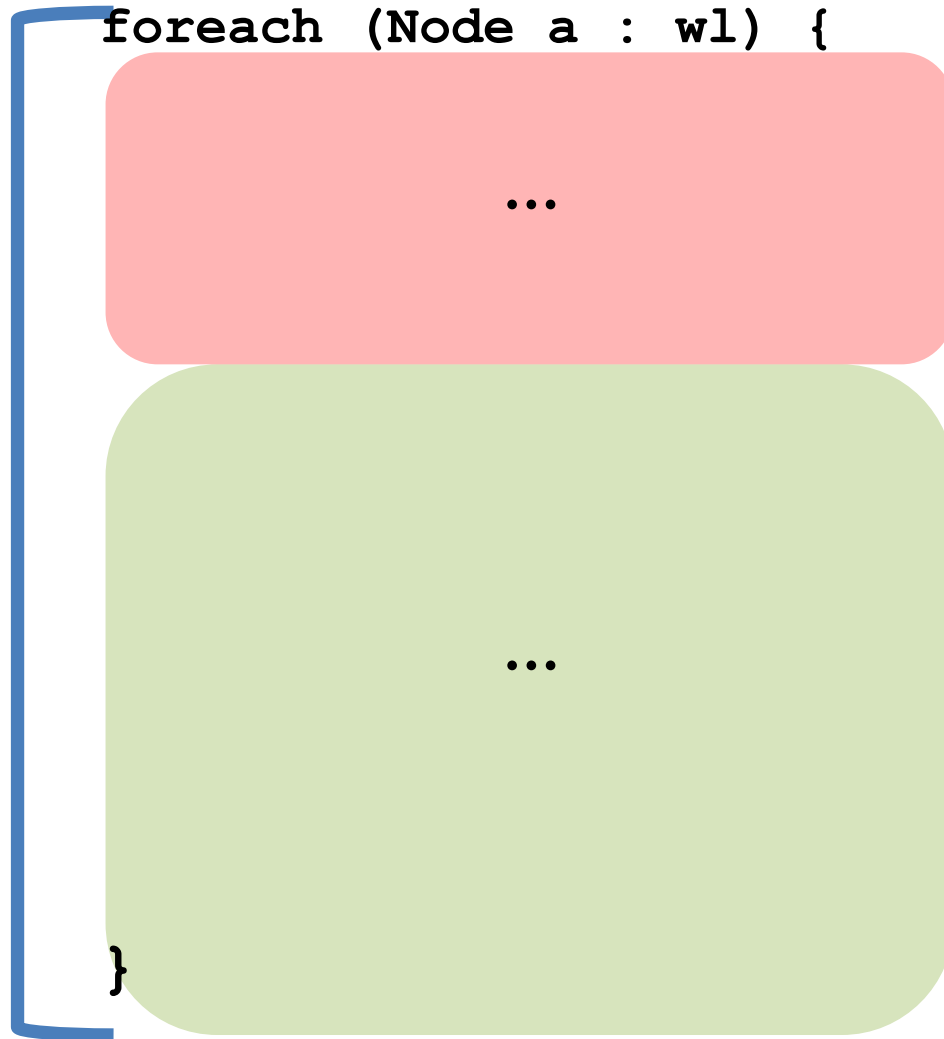
# Overheads (I): Locking

- Optimizations
  - Redundant locking elimination
  - Lock removal for iteration private data
  - Lock removal for lock domination

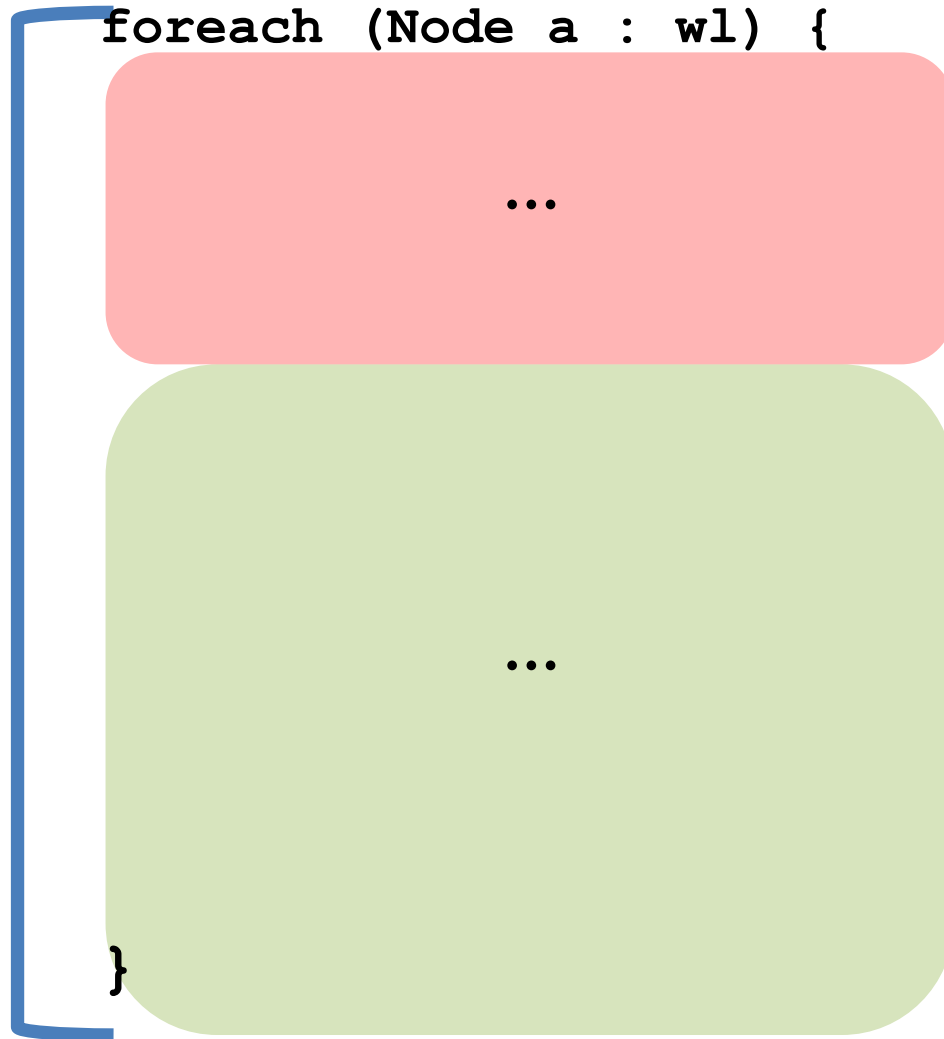
# Overheads (I): Locking

- Optimizations
  - Redundant locking elimination
  - Lock removal for iteration private data
  - Lock removal for lock domination
- $ACQ(P)$ : set of *definitely acquired* locks per program point  $P$
- Given method call  $M$  at  $P$ :  
 $Locks(M) \subseteq ACQ(P) \Rightarrow$  Redundant Locking

# Overheads (II): Undo actions

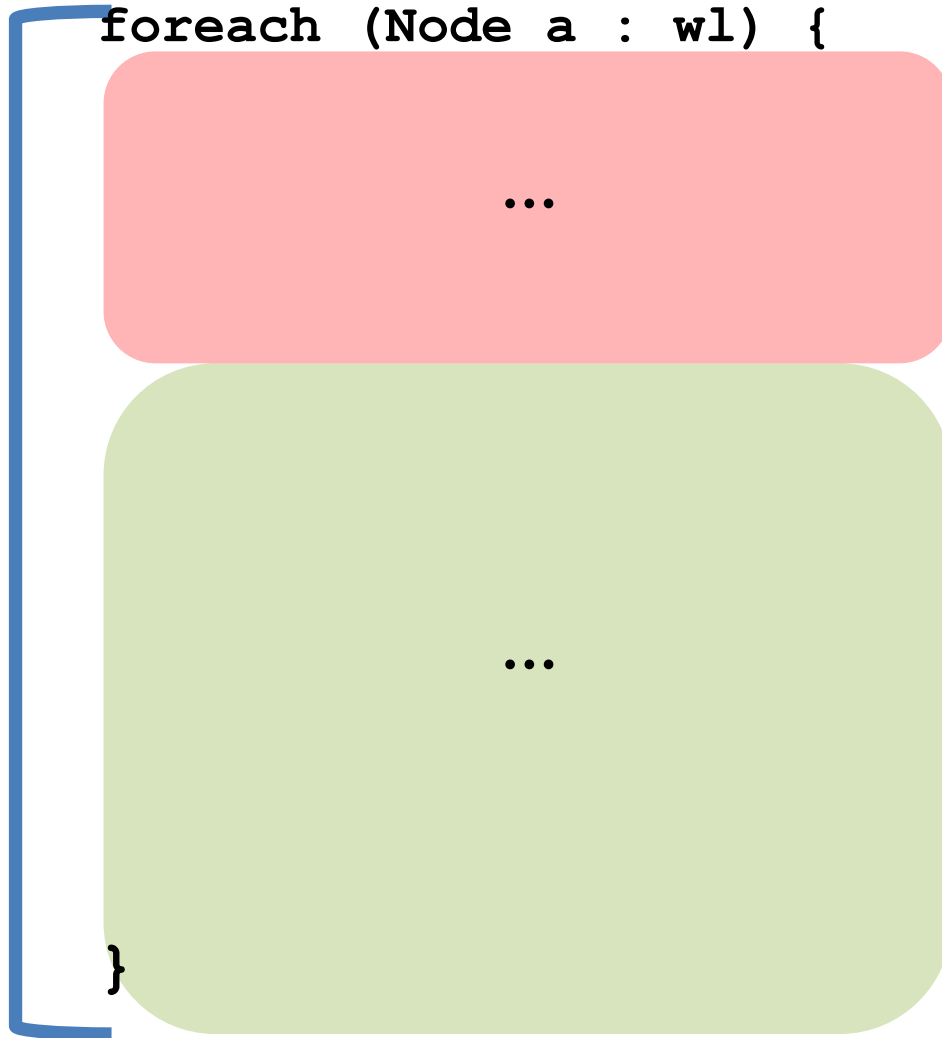
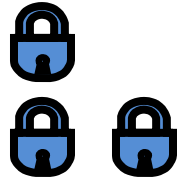


# Overheads (II): Undo actions



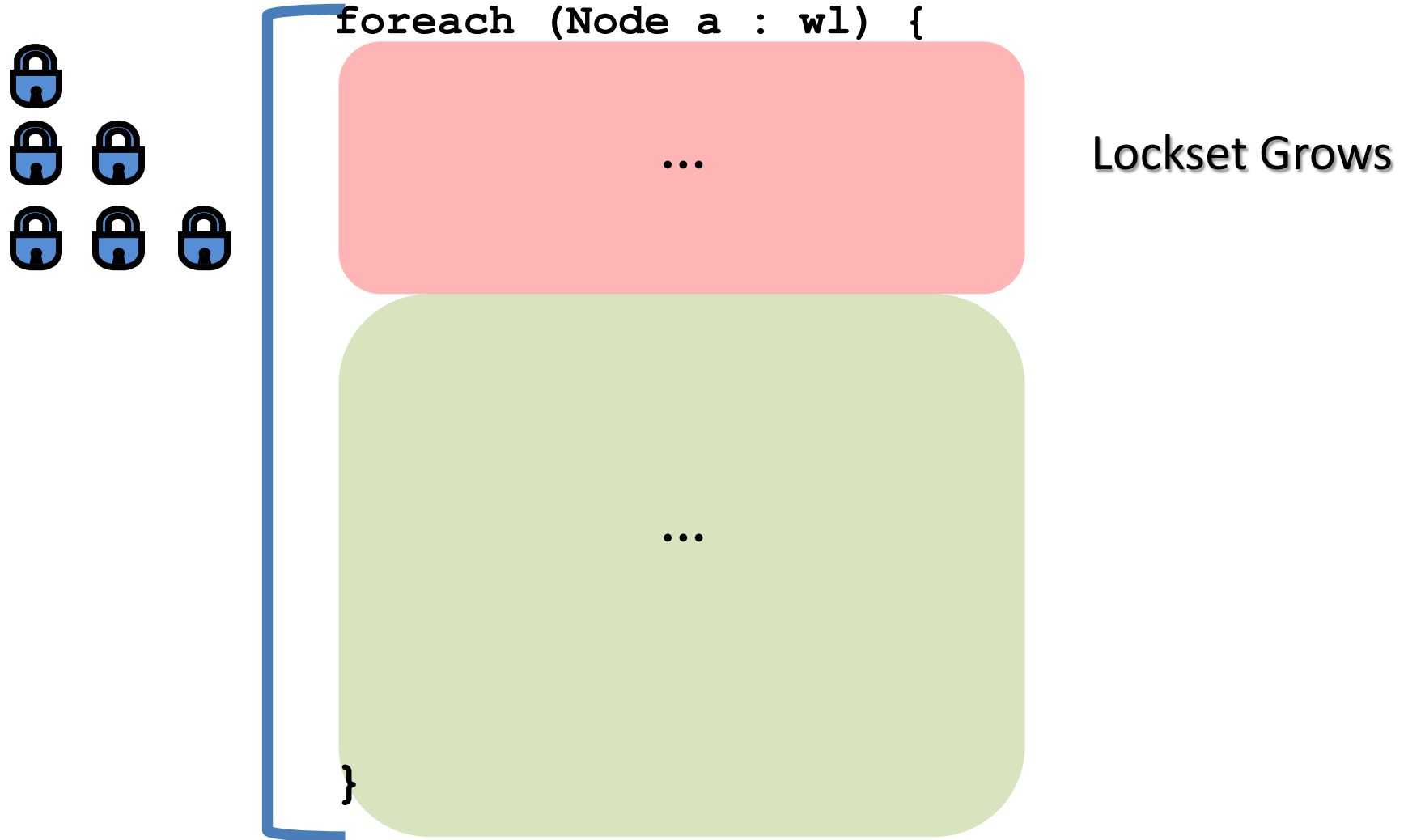
Lockset Grows

# Overheads (II): Undo actions

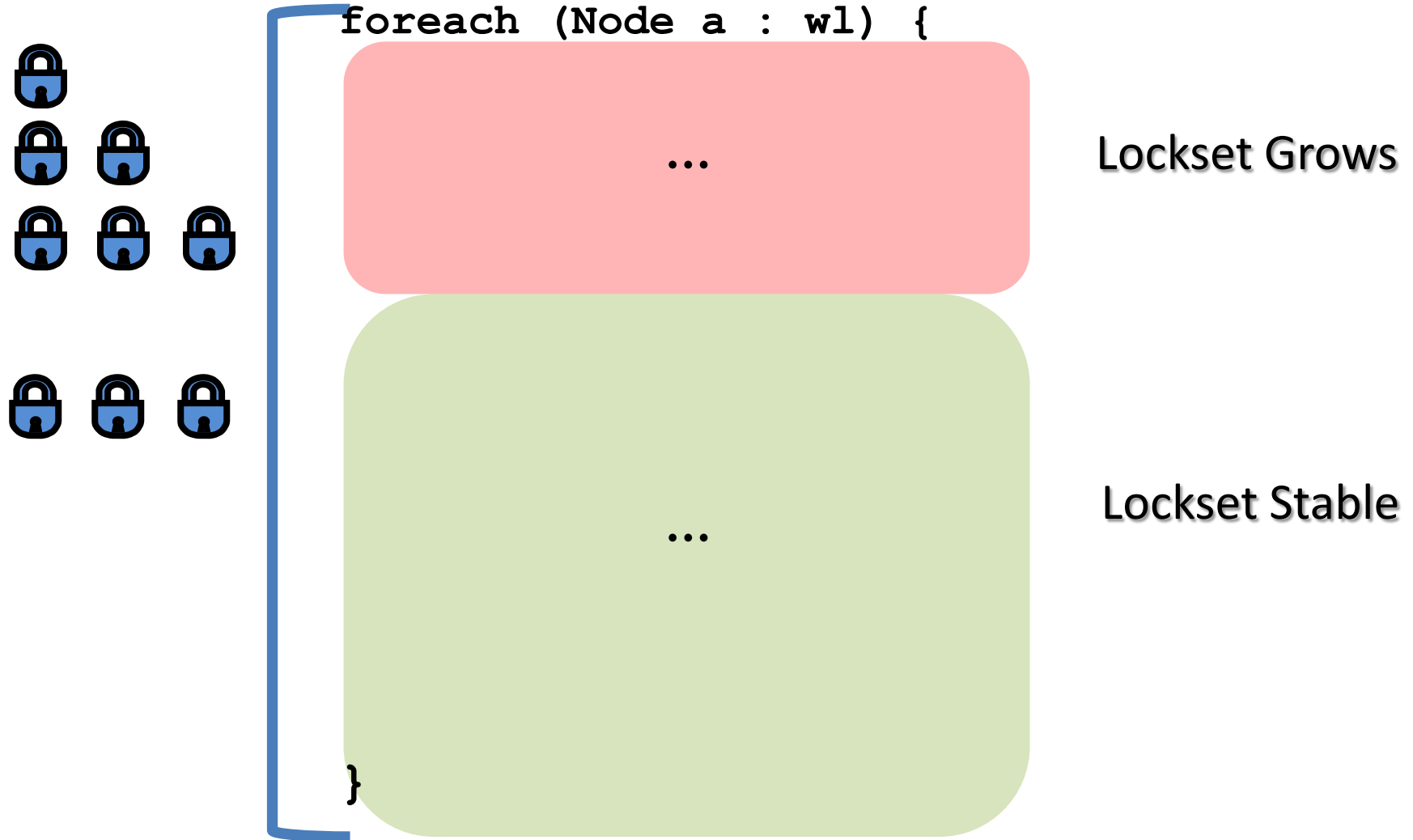


Lockset Grows

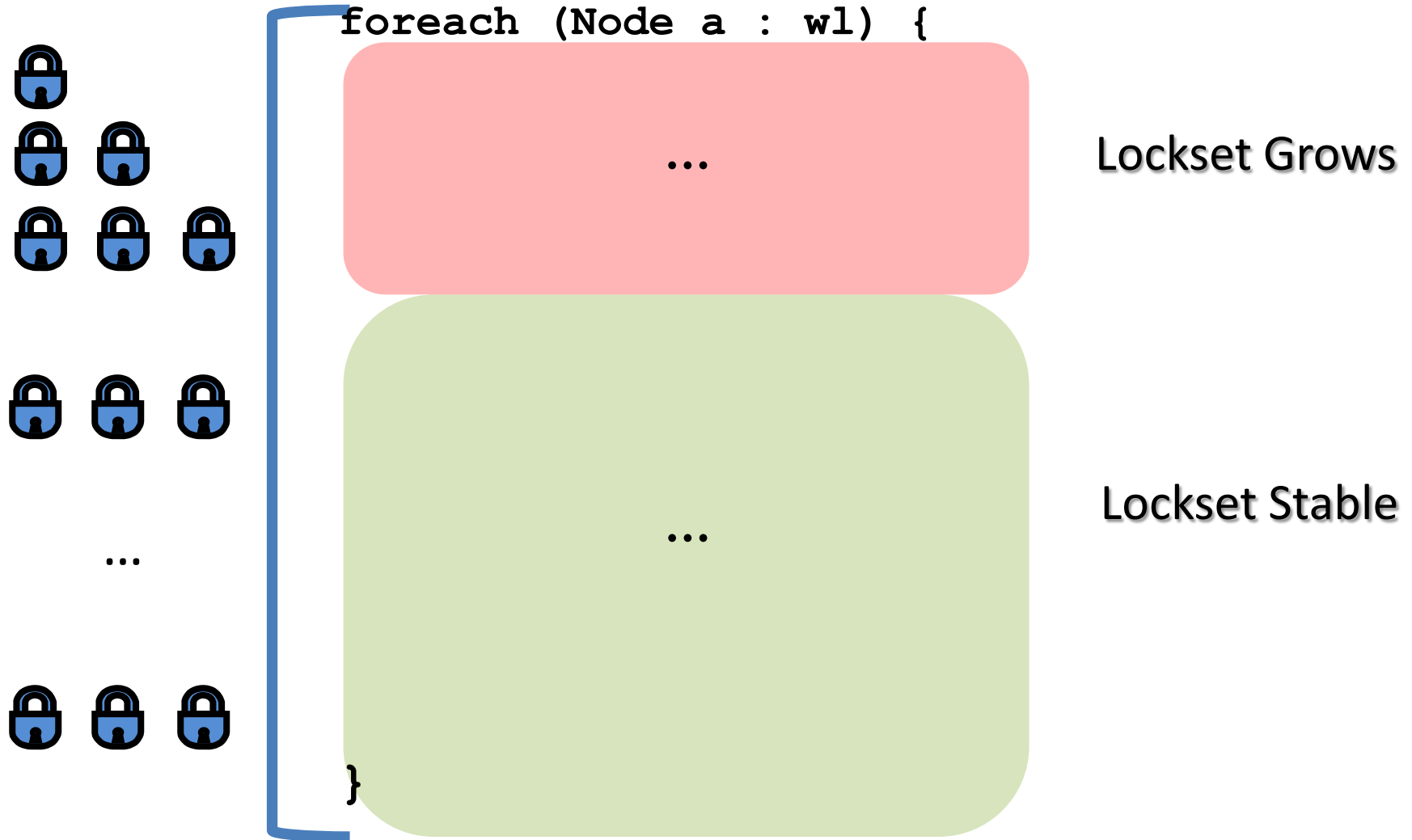
# Overheads (II): Undo actions



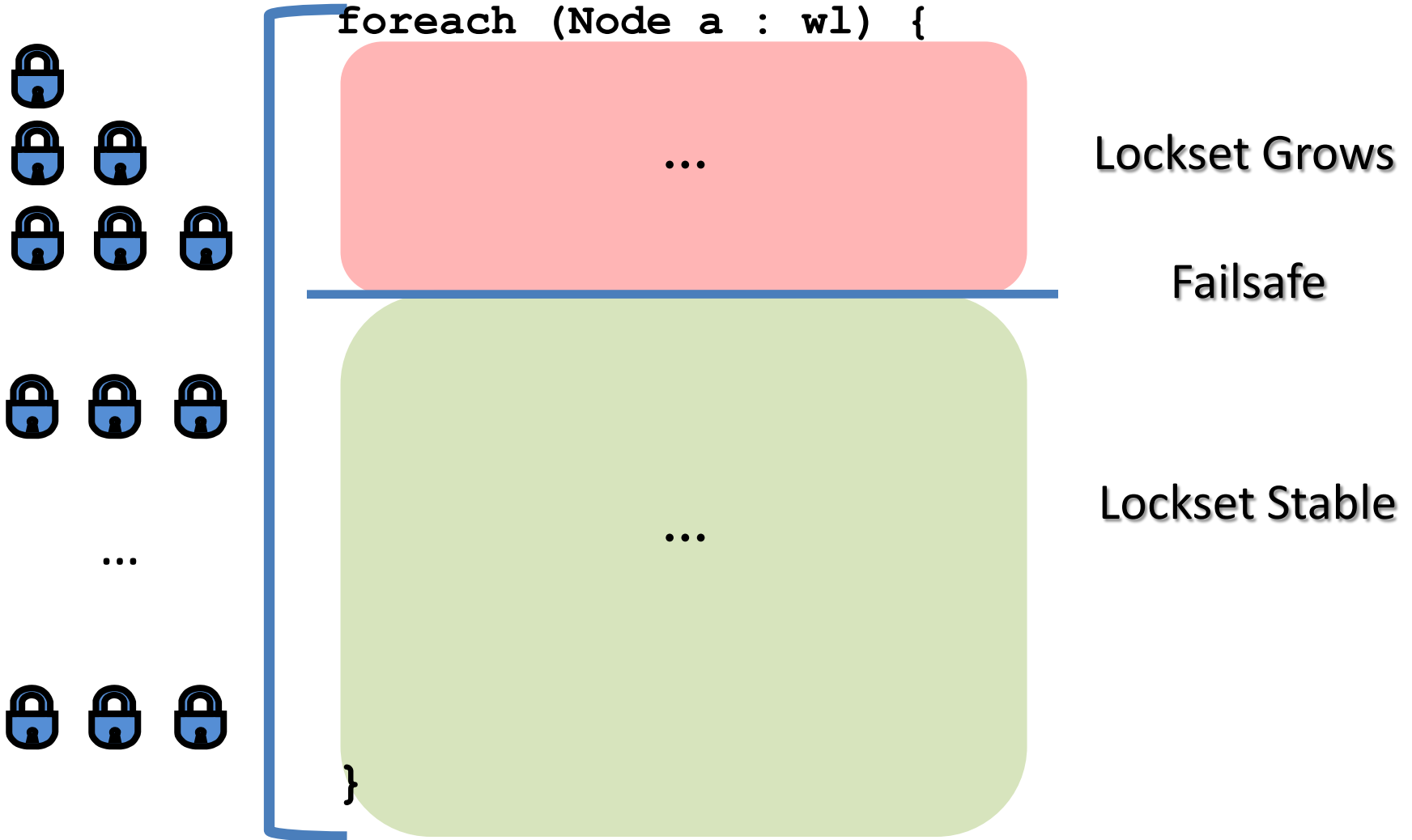
# Overheads (II): Undo actions



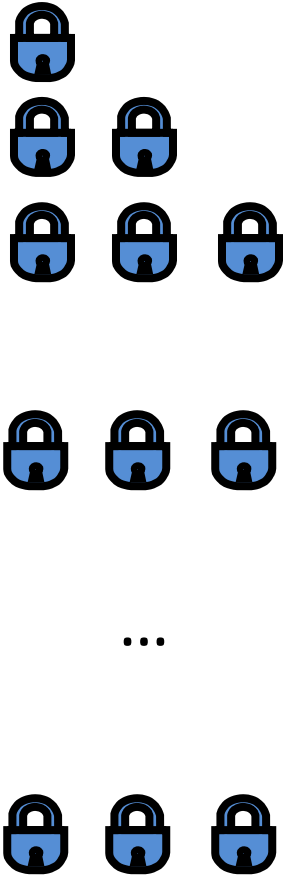
# Overheads (II): Undo actions



# Overheads (II): Undo actions



# Overheads (II): Undo actions



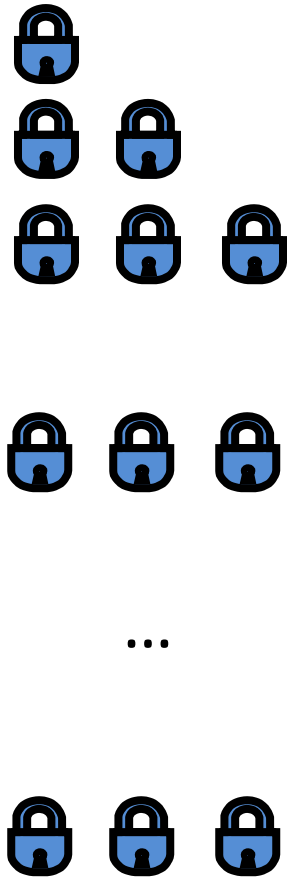
```
foreach (Node a : wl) {  
    Set<Node> aNghbrs = g.neighbors(a);  
    Node lt = null;  
    for (Node n : aNghbrs) {  
        minW,lt = minWeightEdge((a,lt), (a,n));  
    }  
  
    g.removeEdge(a, lt);  
    Set<Node> ltNghbrs = g.neighbors(lt);  
    for (Node n : ltNghbrs) {  
        Edge e = g.getEdge(lt, n);  
        Weight w = g.getEdgeData(e);  
        Edge an = g.getEdge(a, n);  
        if (an != null) {  
            Weight wan = g.getEdgeData(an);  
            if (wan.compareTo(w) < 0)  
                w = wan;  
            g.setEdgeData(an, w);  
        } else {  
            g.addEdge(a, n, w);  
        }  
    }  
    g.removeNode(lt);  
    mst.add(minW);  
    wl.add(a);  
}
```

Lockset Grows

Failsafe

Lockset Stable

# Overheads (II): Undo actions



```

foreach (Node a : wl) {
    Set<Node> aNghbrs = g.neighbors(a);
    Node lt = null;
    for (Node n : aNghbrs) {
        minW,lt = minWeightEdge((a,lt), (a,n));
    }

    g.removeEdge(a, lt);
    Set<Node> ltNghbrs = g.neighbors(lt);
    for (Node n : ltNghbrs) {
        Edge e = g.getEdge(lt, n);
        Weight w = g.getEdgeData(e);
        Edge an = g.getEdge(a, n);
        if (an != null) {
            Weight wan = g.getEdgeData(an);
            if (wan.compareTo(w) < 0)
                w = wan;
            g.setEdgeData(an, w);
        } else {
            g.addEdge(a, n, w);
        }
    }
    g.removeNode(lt);
    mst.add(minW);
    wl.add(a);
}
    
```

Lockset Grows

Failsafe

Lockset Stable

Program point  $P$  is *failsafe* if:

$\forall Q : \text{Reaches}(P, Q) \Rightarrow \text{Locks}(Q) \subseteq \text{ACQ}(P)$

# Lockset Analysis

- Redundant Locking

- $Locks(M) \subseteq ACQ(P)$

- Undo elimination


- $\forall Q : Reaches(P, Q) \Rightarrow Locks(Q) \subseteq ACQ(P)$

- Need to compute  $ACQ(P)$

```
GSet<Node> wl = new GSet<Node>();
wl.addAll(g.getNodes());
GBag<Weight> mst = new GBag<Weight>();

foreach (Node a : wl) {
    Set<Node> aNghbrs = g.neighbors(a);
    Node lt = null;
    for (Node n : aNghbrs) {
        minW, lt = minWeightEdge((a, lt), (a, n));
    }

    g.removeEdge(a, lt);
    Set<Node> ltNghbrs = g.neighbors(lt);
    for (Node n : ltNghbrs) {
        Edge e = g.getEdge(lt, n);
        Weight w = g.getEdgeData(e);
        Edge an = g.getEdge(a, n);
        if (an != null) {
            Weight wan = g.getEdgeData(an);
            if (wan.compareTo(w) < 0)
                w = wan;
            g.setEdgeData(an, w);
        } else {
            g.addEdge(a, n, w);
        }
    }
    g.removeNode(lt);
    mst.add(minW);
    wl.add(a);
}
```

 : Runtime overhead

# Lockset Analysis

- Redundant Locking

- $Locks(M) \subseteq ACQ(P)$

- Undo elimination


- $\forall Q : Reaches(P, Q) \Rightarrow Locks(Q) \subseteq ACQ(P)$

- Need to compute  $ACQ(P)$

```
GSet<Node> wl = new GSet<Node>();
wl.addAll(g.getNodes());
GBag<Weight> mst = new GBag<Weight>();

foreach (Node a : wl) {
    Set<Node> aNghbrs = g.neighbors(a);
    Node lt = null;
    for (Node n : aNghbrs) {
        minW,lt = minWeightEdge((a,lt), (a,n));
    }

    g.removeEdge(a, lt);
    Set<Node> ltNghbrs = g.neighbors(lt);
    for (Node n : ltNghbrs) {
        Edge e = g.getEdge(lt, n);
        Weight w = g.getEdgeData(e);
        Edge an = g.getEdge(a, n);
        if (an != null) {
            Weight wan = g.getEdgeData(an);
            if (wan.compareTo(w) < 0)
                w = wan;
            g.setEdgeData(an, w);
        } else {
            g.addEdge(a, n, w);
        }
    }
    g.removeNode(lt);
    mst.add(minW);
    wl.add(a);
}
```

: Runtime overhead

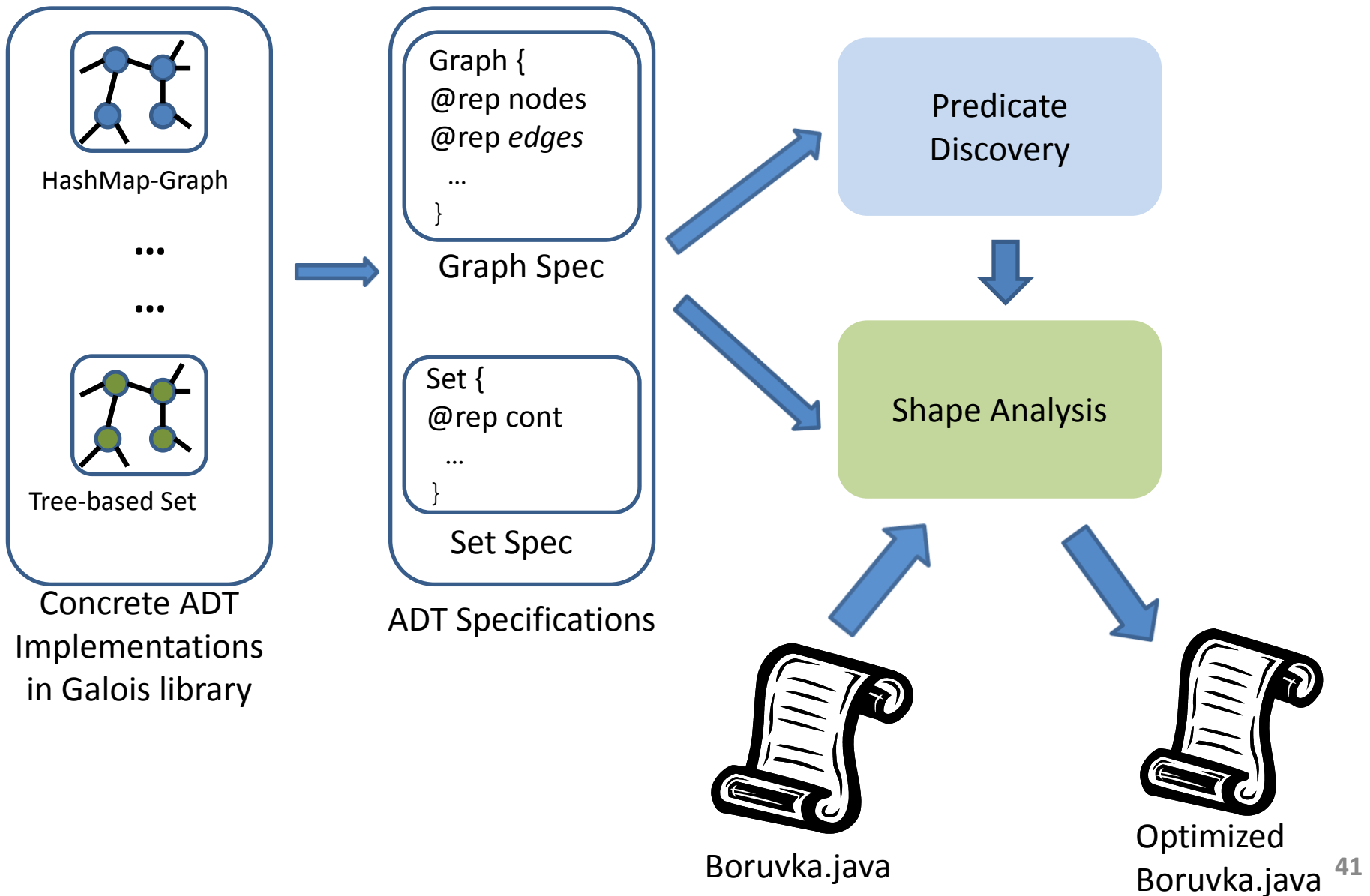
# Analysis Challenges

- The usual suspects:
  - Unbounded Memory → Undecidability
  - Aliasing, Destructive updates
- Specific challenges:
  - Complex ADTs: unstructured graphs
  - Heap objects are locked
  - Adapt abstraction to ADTs
- We use Abstract Interpretation [CC'77]
  - Balance precision and realistic performance

# Organization

- Parallelization of graph algorithms in Galois system
  - Speculative execution
  - Example: Boruvka MST algorithm
- Optimization opportunities
  - Reduce speculation overheads
  - Analysis problem: LockSet shape analysis
- Lockset shape analysis
  - Abstract Data Type (ADT) modeling
  - Hierarchy summarization abstraction
  - Predicate discovery
- Evaluation
  - Fast and infers all available optimizations
  - Optimizations give speedup up to 12x

# Shape Analysis Overview



# ADT Specification

```
...  
Set<Node> S1 = g.neighbors(n);  
...
```

Boruvka.java

```
Graph<ND,ED> {
```

```
@rep set<Node> nodes
```

```
@rep set<Edge> edges
```

```
Set<Node> neighbors(Node n);
```

```
}
```

Graph Spec

# ADT Specification

```
...  
Set<Node> s1 = g.neighbors(n);  
...
```

Boruvka.java

```
Graph<ND,ED> {
```

```
@rep set<Node> nodes  
@rep set<Edge> edges
```

```
Set<Node> neighbors(Node n);
```

```
}
```

Graph Spec

Abstract ADT state by  
virtual set fields

# ADT Specification

```
...  
Set<Node> s1 = g.neighbors(n);  
...
```

Boruvka.java

```
Graph<ND,ED> {  
  
  @rep set<Node> nodes  
  @rep set<Edge> edges  
  
  @locks(n + n.rev(src) + n.rev(src).dst +  
         n.rev(dst) + n.rev(dst).src)  
  
  Set<Node> neighbors(Node n);  
  
}
```

Abstract ADT state by virtual set fields

Graph Spec

# ADT Specification

Abstract ADT state by  
virtual set fields

```
...  
Set<Node> s1 = g.neighbors(n);  
...
```

Boruvka.java

```
Graph<ND,ED> {  
  
  @rep set<Node> nodes  
  @rep set<Edge> edges  
  
  @locks(n + n.rev(src) + n.rev(src).dst +  
         n.rev(dst) + n.rev(dst).src)  
  @op(  
    nghbrs = n.rev(src).dst + n.rev(dst).src ,  
    ret = new Set<Node<ND>>(cont=nghbrs)  
  )  
  Set<Node> neighbors(Node n);  
  
}
```

Graph Spec

# ADT Specification

Abstract ADT state by  
virtual set fields

```
...  
Set<Node> s1 = g.neighbors(n);  
...
```

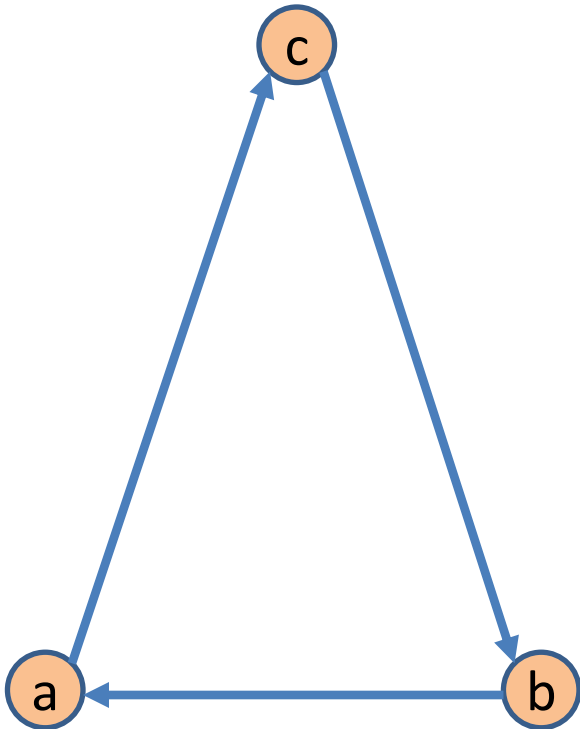
Boruvka.java

```
Graph<ND,ED> {  
  
  @rep set<Node> nodes  
  @rep set<Edge> edges  
  
  @locks(n + n.rev(src) + n.rev(src).dst +  
         n.rev(dst) + n.rev(dst).src)  
  @op(  
    nghbrs = n.rev(src).dst + n.rev(dst).src ,  
    ret = new Set<Node<ND>>(cont=nghbrs)  
  )  
  Set<Node> neighbors(Node n);  
  
}
```

Graph Spec

Assumption: Implementation satisfies Spec

# Modeling ADTs



```
Graph<ND,ED> {
```

```
@rep set<Node> nodes
```

```
@rep set<Edge> edges
```

```
@locks(n + n.rev(src) + n.rev(src).dst +  
        n.rev(dst) + n.rev(dst).src)
```

```
@op(
```

```
  nghbrs = n.rev(src).dst + n.rev(dst).src ,  
  ret = new Set<Node<ND>>(cont=nghbrs)
```

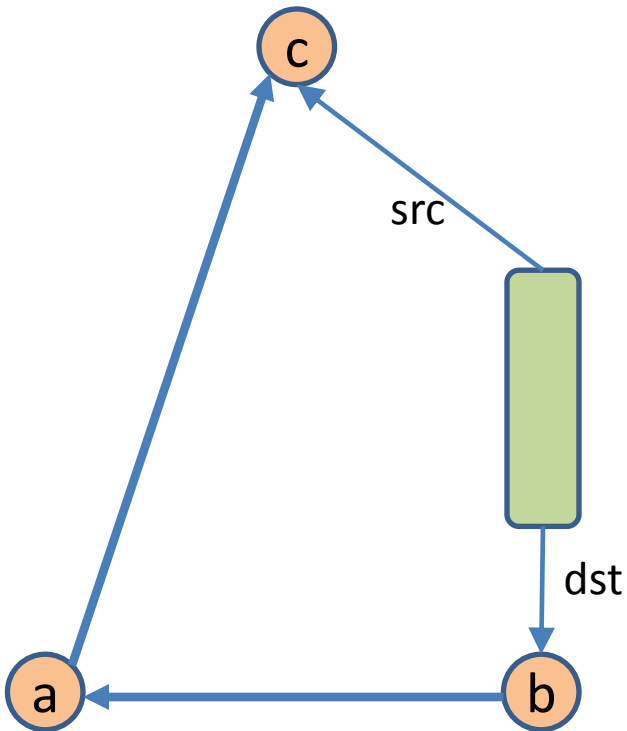
```
)
```

```
Set<Node> neighbors(Node n);
```

```
}
```

Graph Spec

# Modeling ADTs



```
Graph<ND,ED> {
```

```
@rep set<Node> nodes
```

```
@rep set<Edge> edges
```

```
@locks(n + n.rev(src) + n.rev(src).dst +  
        n.rev(dst) + n.rev(dst).src)
```

```
@op(
```

```
  nghbrs = n.rev(src).dst + n.rev(dst).src ,  
  ret = new Set<Node<ND>>(cont=nghbrs)
```

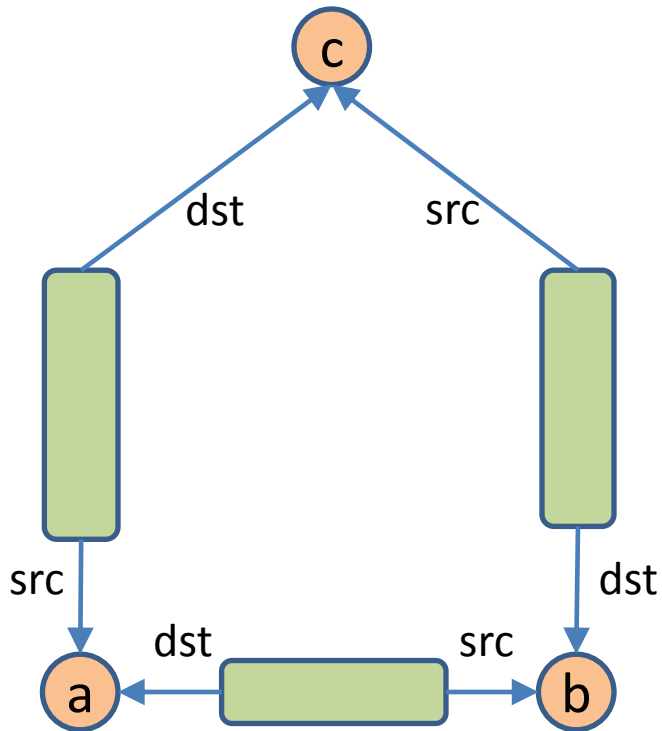
```
)
```

```
Set<Node> neighbors(Node n);
```

```
}
```

Graph Spec

# Modeling ADTs



```
Graph<ND,ED> {
```

```
@rep set<Node> nodes
```

```
@rep set<Edge> edges
```

```
@locks(n + n.rev(src) + n.rev(src).dst +  
n.rev(dst) + n.rev(dst).src)
```

```
@op(
```

```
  nghbrs = n.rev(src).dst + n.rev(dst).src ,  
  ret = new Set<Node<ND>>(cont=nghbrs)
```

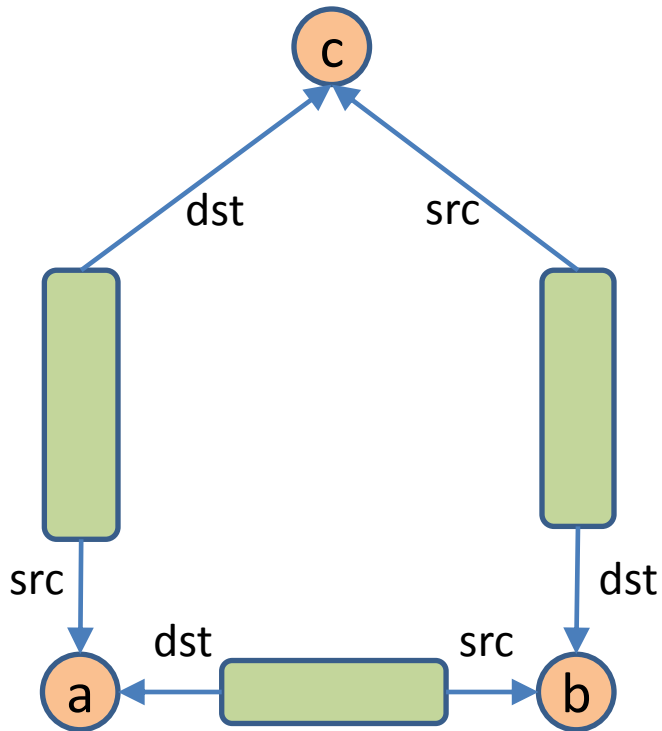
```
)
```

```
Set<Node> neighbors(Node n);
```

```
}
```

Graph Spec

# Modeling ADTs



```
Graph<ND,ED> {
```

```
@rep set<Node> nodes
```

```
@rep set<Edge> edges
```

```
@locks(n + n.rev(src) + n.rev(src).dst +  
       n.rev(dst) + n.rev(dst).src)
```

```
@op(
```

```
  nghbrs = n.rev(src).dst + n.rev(dst).src ,  
  ret = new Set<Node<ND>>(cont=nghbrs)
```

```
)
```

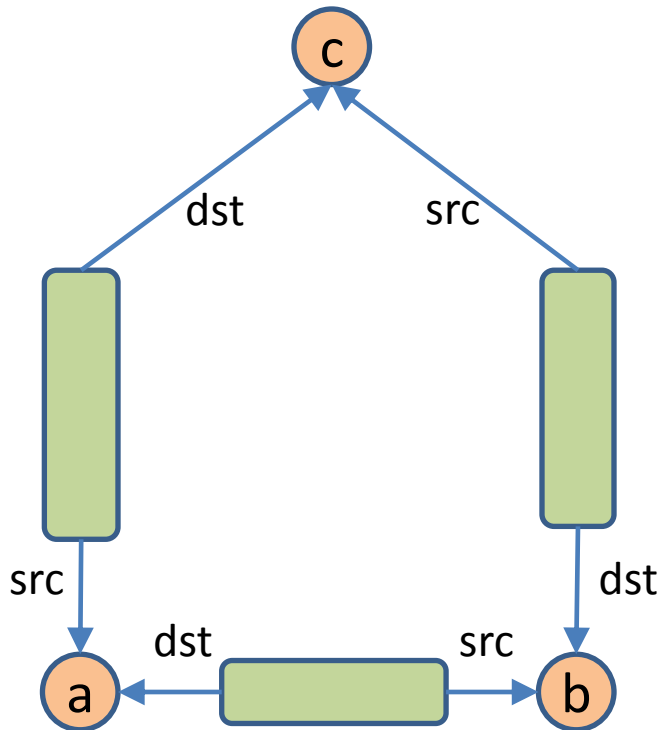
```
Set<Node> neighbors(Node n);
```

```
}
```

Graph Spec

# Modeling ADTs

Abstract State



```
Graph<ND,ED> {
```

```
@rep set<Node> nodes
```

```
@rep set<Edge> edges
```

```
@locks(n + n.rev(src) + n.rev(src).dst +  
       n.rev(dst) + n.rev(dst).src)
```

```
@op(
```

```
  nghbrs = n.rev(src).dst + n.rev(dst).src ,  
  ret = new Set<Node<ND>>(cont=nghbrs)
```

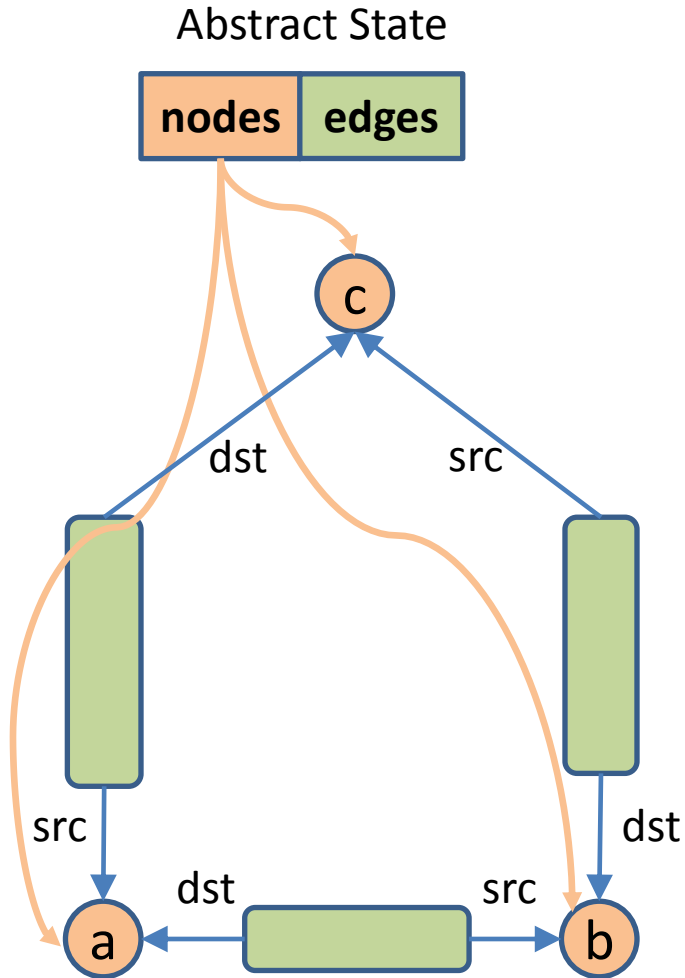
```
)
```

```
Set<Node> neighbors(Node n);
```

```
}
```

Graph Spec

# Modeling ADTs



```
Graph<ND,ED> {
```

```
@rep set<Node> nodes
```

```
@rep set<Edge> edges
```

```
@locks(n + n.rev(src) + n.rev(src).dst +  
n.rev(dst) + n.rev(dst).src)
```

```
@op(
```

```
  nghbrs = n.rev(src).dst + n.rev(dst).src ,  
  ret = new Set<Node<ND>>(cont=nghbrs)
```

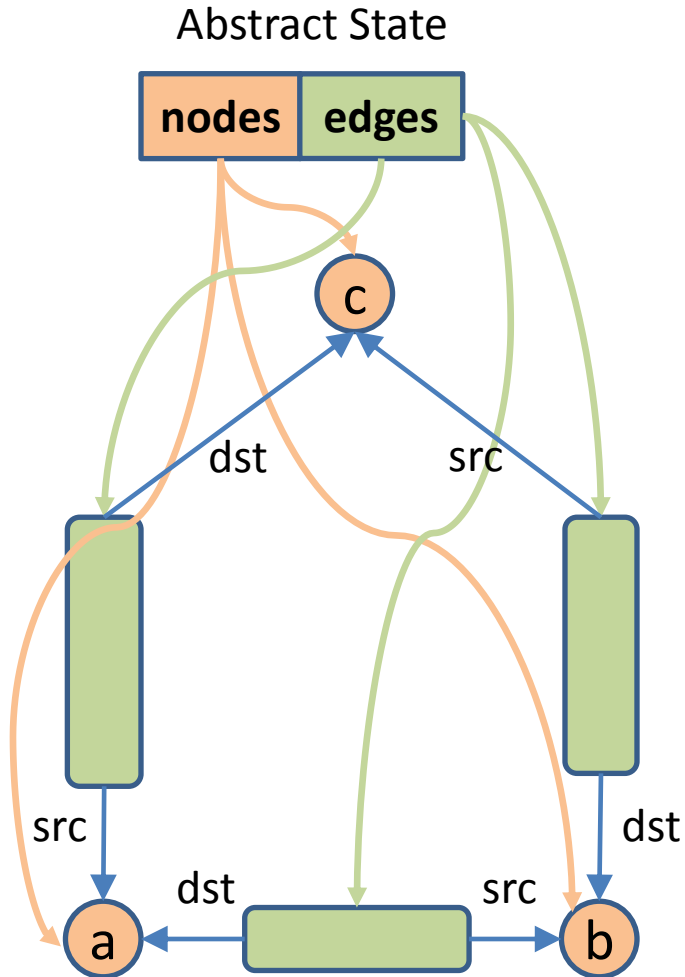
```
)
```

```
Set<Node> neighbors(Node n);
```

```
}
```

Graph Spec

# Modeling ADTs



```
Graph<ND,ED> {
```

```
@rep set<Node> nodes
```

```
@rep set<Edge> edges
```

```
@locks(n + n.rev(src) + n.rev(src).dst +  
n.rev(dst) + n.rev(dst).src)
```

```
@op(
```

```
  nghbrs = n.rev(src).dst + n.rev(dst).src ,  
  ret = new Set<Node<ND>>(cont=nghbrs)
```

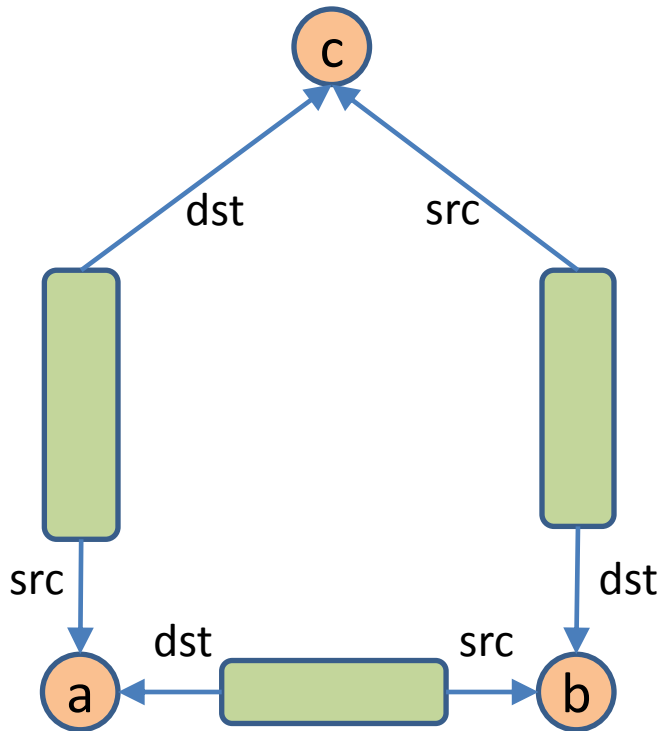
```
)
```

```
Set<Node> neighbors(Node n);
```

```
}
```

Graph Spec

# Modeling ADTs



```
Graph<ND,ED> {
```

```
@rep set<Node> nodes
```

```
@rep set<Edge> edges
```

```
@locks(n + n.rev(src) + n.rev(src).dst +  
       n.rev(dst) + n.rev(dst).src)
```

```
@op(
```

```
  nghbrs = n.rev(src).dst + n.rev(dst).src ,  
  ret = new Set<Node<ND>>(cont=nghbrs)
```

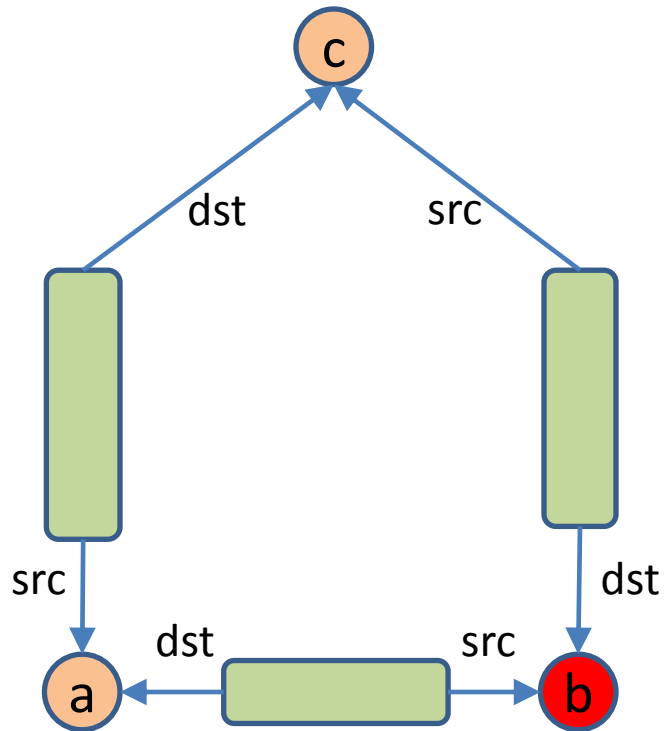
```
)
```

```
Set<Node> neighbors(Node n);
```

```
}
```

Graph Spec

# Modeling ADTs



```
Graph<ND,ED> {
```

```
@rep set<Node> nodes
```

```
@rep set<Edge> edges
```

```
@locks(n + n.rev(src) + n.rev(src).dst +  
n.rev(dst) + n.rev(dst).src)
```

```
@op(
```

```
  nghbrs = n.rev(src).dst + n.rev(dst).src ,  
  ret = new Set<Node<ND>>(cont=nghbrs)
```

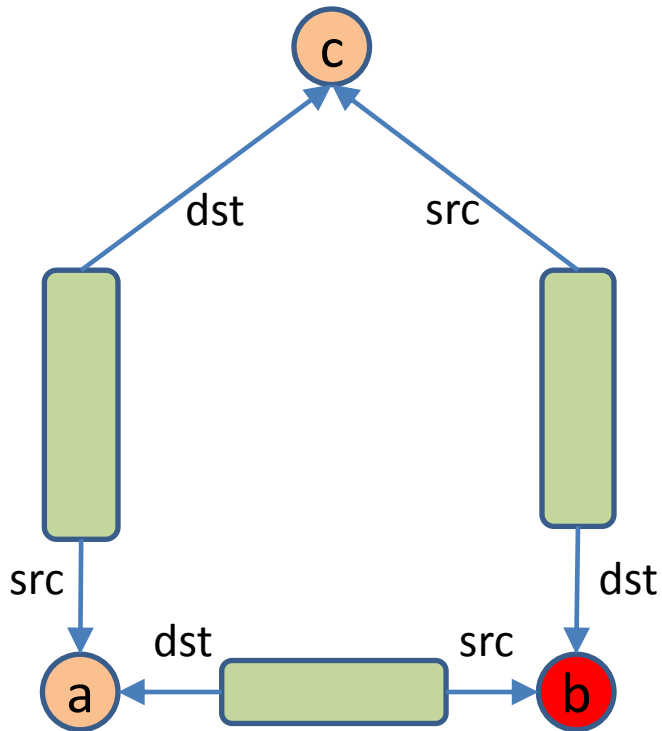
```
)
```

```
Set<Node> neighbors(Node n);
```

```
}
```

Graph Spec

# Modeling ADTs



```
Graph<ND,ED> {
```

```
@rep set<Node> nodes
```

```
@rep set<Edge> edges
```

```
@locks(n + n.rev(src) + n.rev(src).dst +  
n.rev(dst) + n.rev(dst).src)
```

```
@op(
```

```
  nghbrs = n.rev(src).dst + n.rev(dst).src ,  
  ret = new Set<Node<ND>>(cont=nghbrs)
```

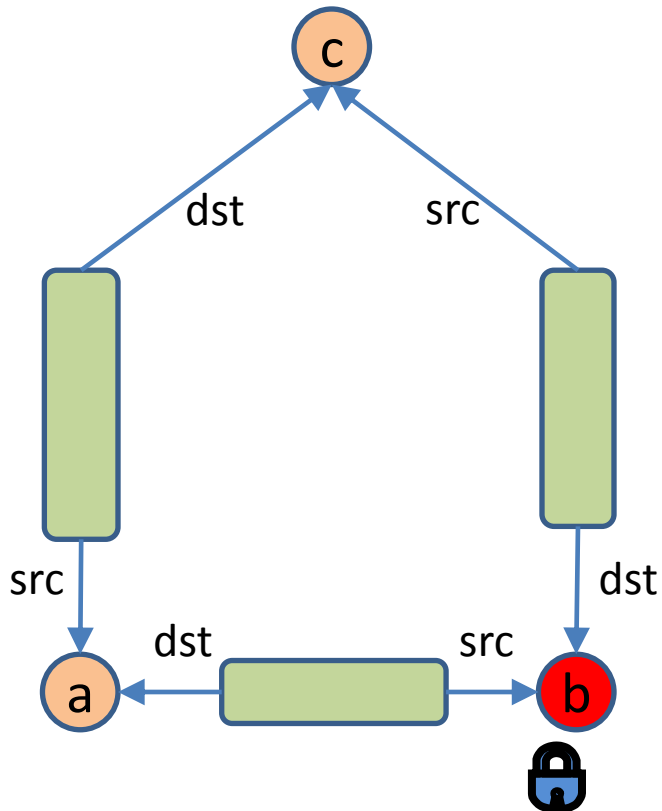
```
)
```

```
Set<Node> neighbors(Node n);
```

```
}
```

Graph Spec

# Modeling ADTs



```
Graph<ND,ED> {
```

```
@rep set<Node> nodes
```

```
@rep set<Edge> edges
```

```
@locks(n + n.rev(src) + n.rev(src).dst +  
n.rev(dst) + n.rev(dst).src)
```

```
@op(
```

```
  nghbrs = n.rev(src).dst + n.rev(dst).src ,  
  ret = new Set<Node<ND>>(cont=nghbrs)
```

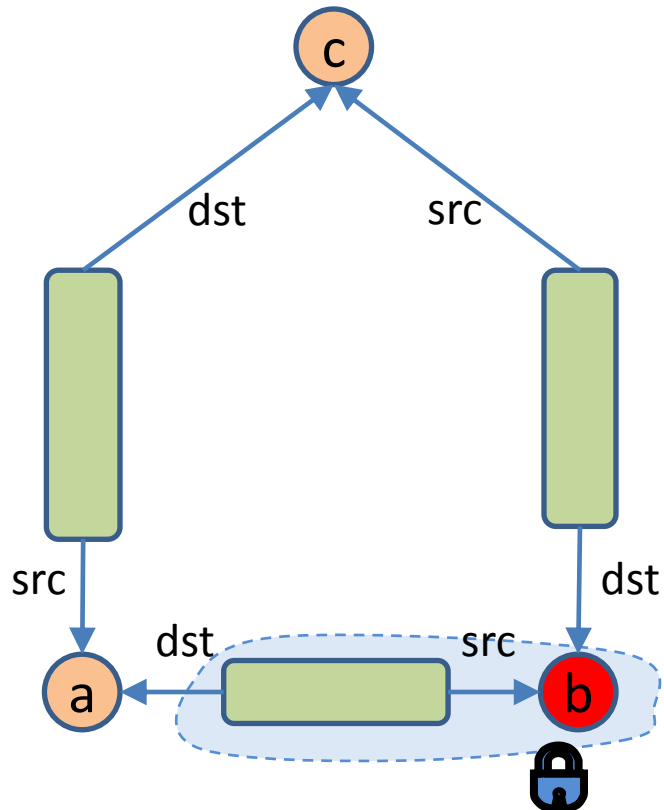
```
)
```

```
Set<Node> neighbors(Node n);
```

```
}
```

Graph Spec

# Modeling ADTs



```
Graph<ND,ED> {
```

```
@rep set<Node> nodes
```

```
@rep set<Edge> edges
```

```
@locks(n + n.rev(src) + n.rev(src).dst +  
n.rev(dst) + n.rev(dst).src)
```

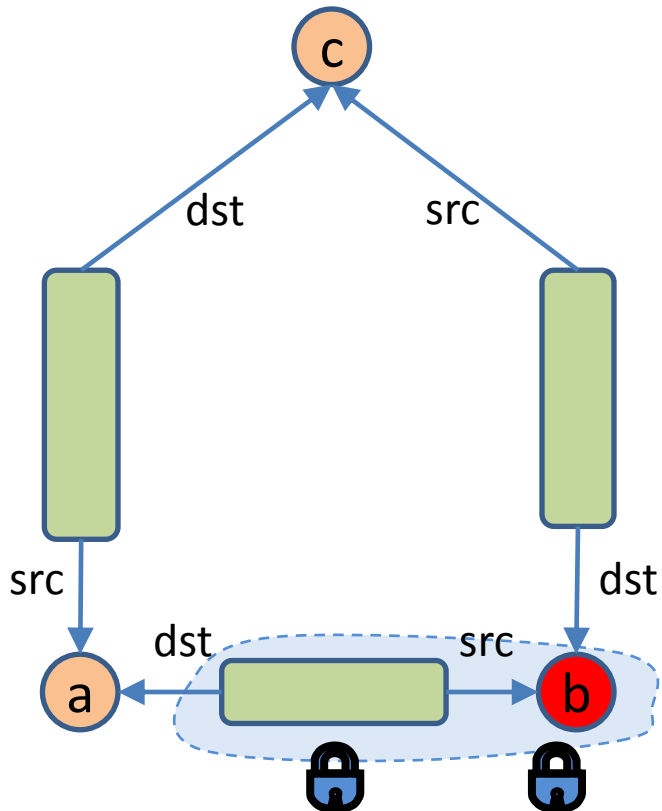
```
@op(  
  nghbrs = n.rev(src).dst + n.rev(dst).src ,  
  ret = new Set<Node<ND>>(cont=nghbrs)  
)
```

```
Set<Node> neighbors(Node n);
```

```
}
```

Graph Spec

# Modeling ADTs



```
Graph<ND,ED> {
```

```
@rep set<Node> nodes
```

```
@rep set<Edge> edges
```

```
@locks(n + n.rev(src) + n.rev(src).dst +  
n.rev(dst) + n.rev(dst).src)
```

```
@op(
```

```
  nghbrs = n.rev(src).dst + n.rev(dst).src ,  
  ret = new Set<Node<ND>>(cont=nghbrs)
```

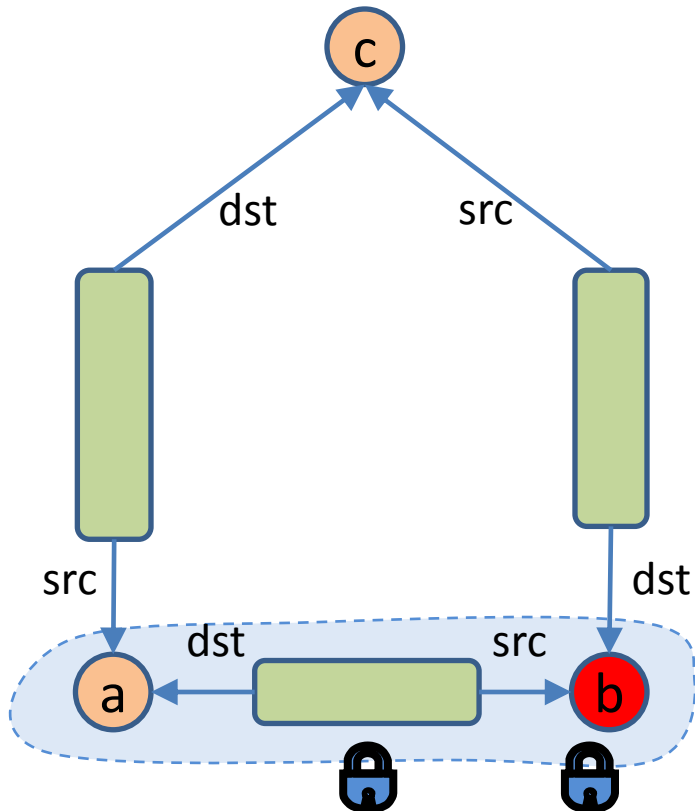
```
)
```

```
Set<Node> neighbors(Node n);
```

```
}
```

Graph Spec

# Modeling ADTs



```
Graph<ND,ED> {
```

```
@rep set<Node> nodes
```

```
@rep set<Edge> edges
```

```
@locks(n + n.rev(src) + n.rev(src).dst +  
n.rev(dst) + n.rev(dst).src)
```

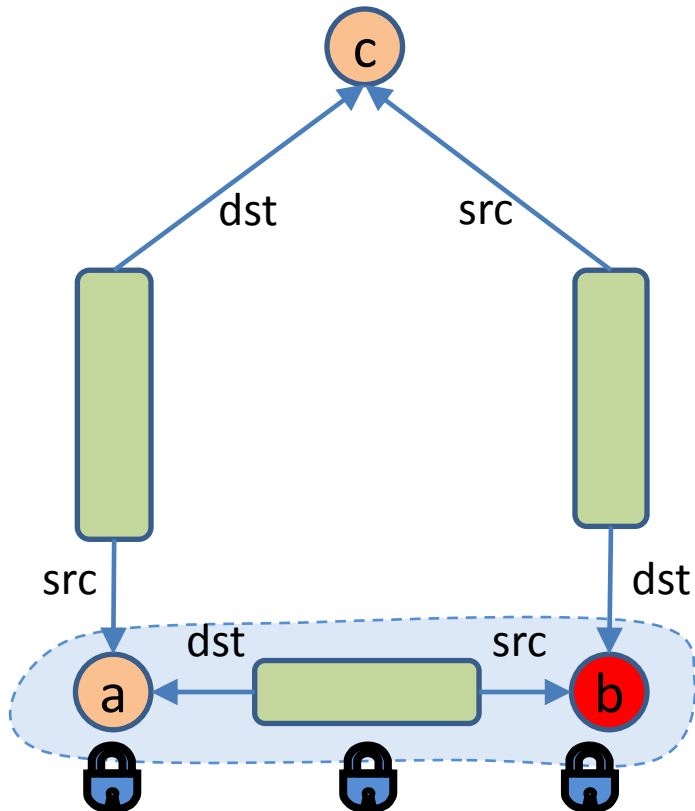
```
@op(  
  nghbrs = n.rev(src).dst + n.rev(dst).src ,  
  ret = new Set<Node<ND>>(cont=nghbrs)  
)
```

```
Set<Node> neighbors(Node n);
```

```
}
```

Graph Spec

# Modeling ADTs



```
Graph<ND,ED> {
```

```
@rep set<Node> nodes
```

```
@rep set<Edge> edges
```

```
@locks(n + n.rev(src) + n.rev(src).dst +  
n.rev(dst) + n.rev(dst).src)
```

```
@op(
```

```
  nghbrs = n.rev(src).dst + n.rev(dst).src ,  
  ret = new Set<Node<ND>>(cont=nghbrs)
```

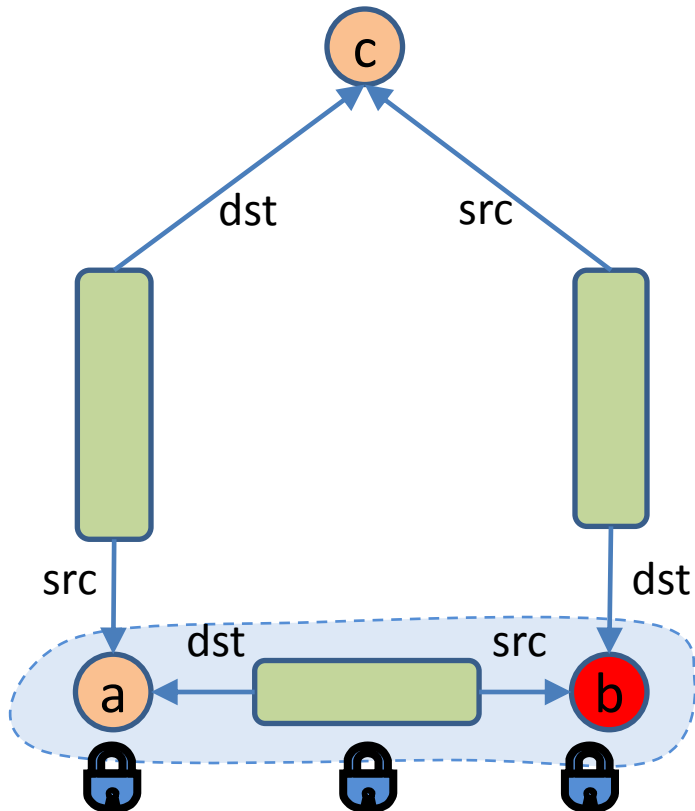
```
)
```

```
Set<Node> neighbors(Node n);
```

```
}
```

Graph Spec

# Modeling ADTs



```
Graph<ND,ED> {
```

```
@rep set<Node> nodes
```

```
@rep set<Edge> edges
```

```
@locks(n + n.rev(src) + n.rev(src).dst +  
n.rev(dst) + n.rev(dst).src)
```

```
@op(
```

```
  nghbrs = n.rev(src).dst + n.rev(dst).src ,  
  ret = new Set<Node<ND>>(cont=nghbrs)
```

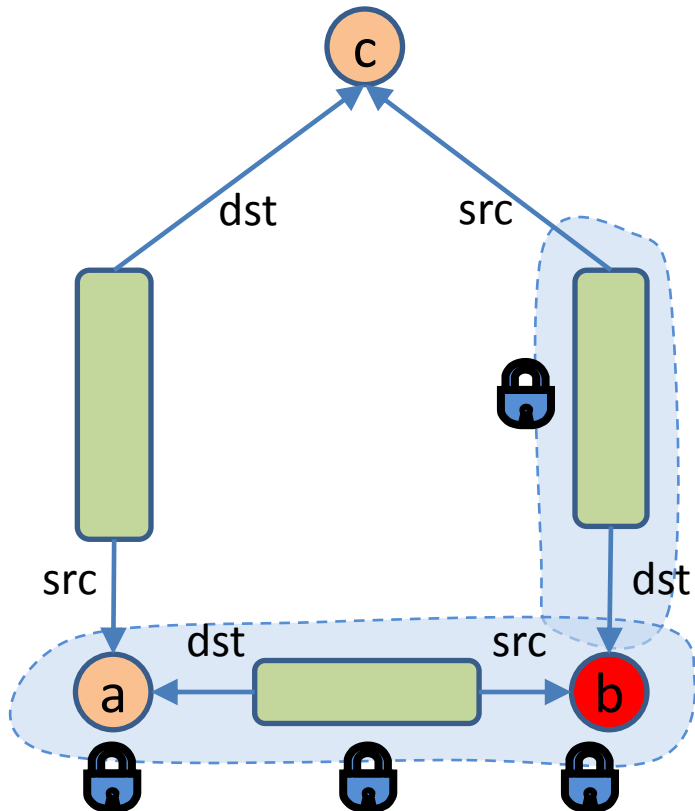
```
)
```

```
Set<Node> neighbors(Node n);
```

```
}
```

Graph Spec

# Modeling ADTs



```
Graph<ND,ED> {
```

```
@rep set<Node> nodes
```

```
@rep set<Edge> edges
```

```
@locks(n + n.rev(src) + n.rev(src).dst +  
n.rev(dst) + n.rev(dst).src)
```

```
@op(
```

```
  nghbrs = n.rev(src).dst + n.rev(dst).src ,  
  ret = new Set<Node<ND>>(cont=nghbrs)
```

```
)
```

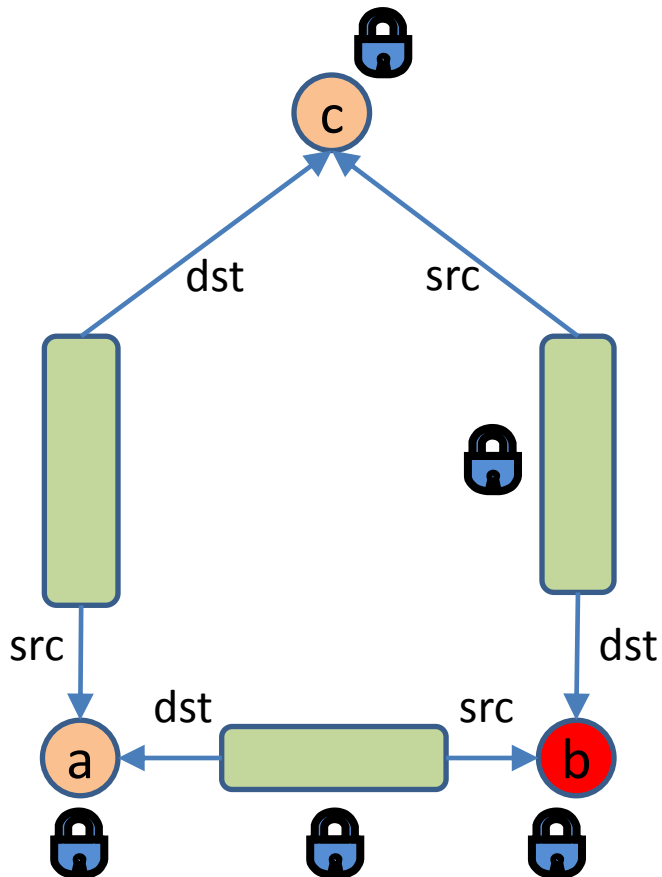
```
Set<Node> neighbors(Node n);
```

```
}
```

Graph Spec



# Modeling ADTs



```
Graph<ND,ED> {
```

```
@rep set<Node> nodes
```

```
@rep set<Edge> edges
```

```
@locks(n + n.rev(src) + n.rev(src).dst +  
n.rev(dst) + n.rev(dst).src)
```

```
@op(  
  
```

```
  nghbrs = n.rev(src).dst + n.rev(dst).src,  
  ret = new Set<Node<ND>>(cont=nghbrs)
```

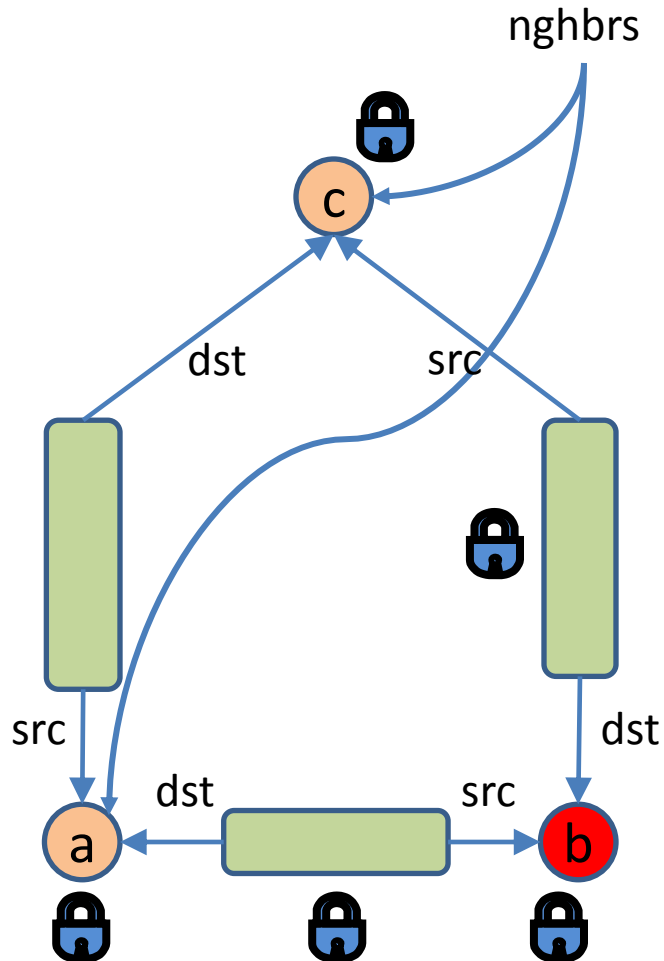
```
)
```

```
Set<Node> neighbors(Node n);
```

```
}
```

Graph Spec

# Modeling ADTs



```
Graph<ND,ED> {
```

```
@rep set<Node> nodes
```

```
@rep set<Edge> edges
```

```
@locks(n + n.rev(src) + n.rev(src).dst +  
n.rev(dst) + n.rev(dst).src)
```

```
@op(
```

```
nghbrs = n.rev(src).dst + n.rev(dst).src,  
ret = new Set<Node<ND>>(cont=nghbrs)
```

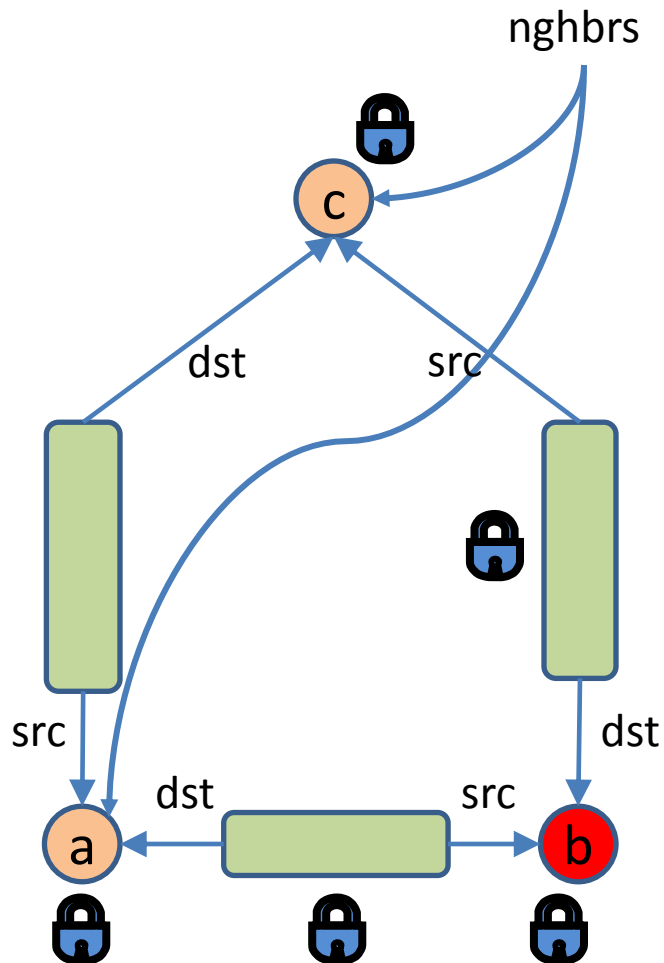
```
)
```

```
Set<Node> neighbors(Node n);
```

```
}
```

Graph Spec

# Modeling ADTs



```
Graph<ND,ED> {
```

```
@rep set<Node> nodes
```

```
@rep set<Edge> edges
```

```
@locks(n + n.rev(src) + n.rev(src).dst +  
n.rev(dst) + n.rev(dst).src)
```

```
@op(  
nghbrs = n.rev(src).dst + n.rev(dst).src ,
```

```
ret = new Set<Node<ND>>(cont=nghbrs);
```

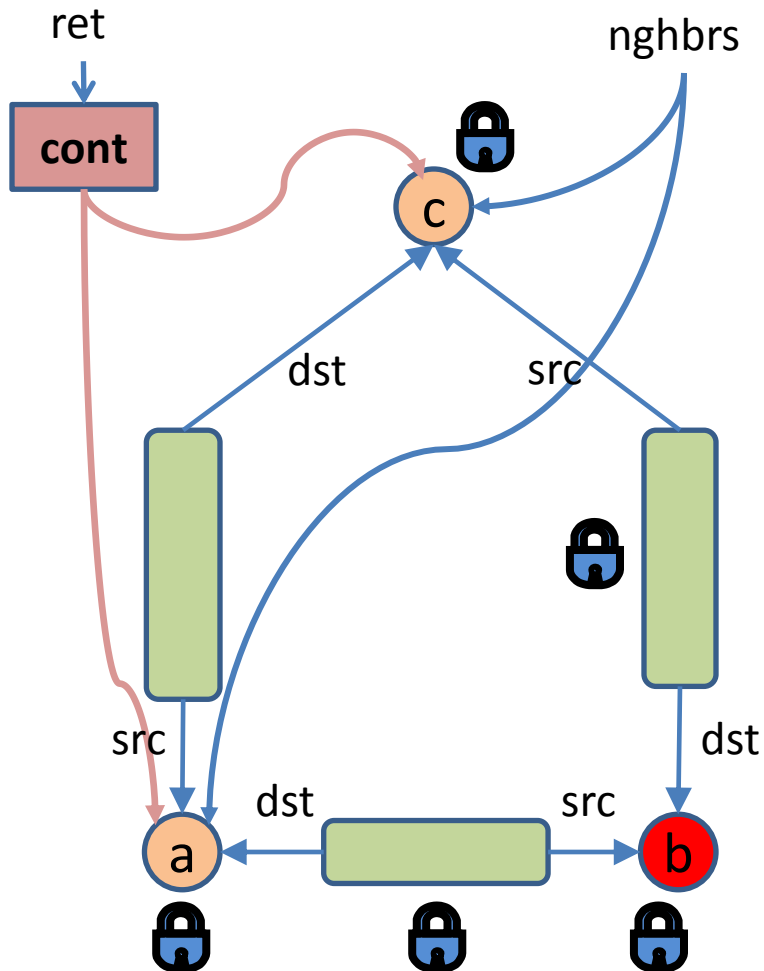
```
)
```

```
Set<Node> neighbors(Node n);
```

```
}
```

Graph Spec

# Modeling ADTs



```
Graph<ND,ED> {
```

```
@rep set<Node> nodes
```

```
@rep set<Edge> edges
```

```
@locks(n + n.rev(src) + n.rev(src).dst +  
n.rev(dst) + n.rev(dst).src)
```

```
@op(
```

```
nghbrs = n.rev(src).dst + n.rev(dst).src ,  
ret = new Set<Node<ND>>(cont=nghbrs)
```

```
)
```

```
Set<Node> neighbors(Node n);
```

```
}
```

Graph Spec

# Organization

- Parallelization of graph algorithms in Galois system
  - Speculative execution
  - Example: Boruvka MST algorithm
- Optimization opportunities
  - Reduce speculation overheads
  - Analysis problem: LockSet shape analysis
- Lockset shape analysis
  - Abstract Data Type (ADT) modeling
  - Hierarchy summarization abstraction
  - Predicate discovery
- Evaluation
  - Fast and infers all available optimizations
  - Optimizations give speedup up to 12x

# Abstraction Scheme

- Parameterized by set of *LockPaths*:

$$L(\text{Path}) \triangleq \forall o . o \in \text{Path} \Rightarrow \text{Locked}(o)$$

– Tracks subset of must-be-locked objects

- Abstract domain elements have the form:

$$\textit{Aliasing-configs} \rightarrow 2^{\textit{LockPaths}} \times \dots$$

# Abstraction Scheme

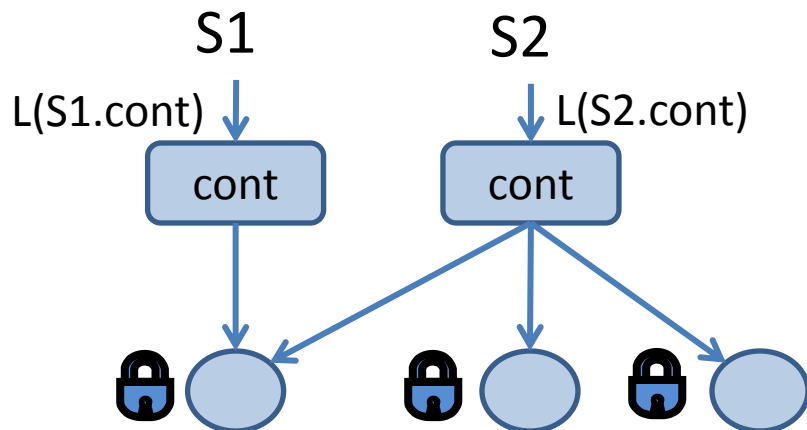
- Parameterized by set of *LockPaths*:

$$L(\text{Path}) \triangleq \forall o . o \in \text{Path} \Rightarrow \text{Locked}(o)$$

– Tracks subset of must-be-locked objects

- Abstract domain elements have the form:

$$\text{Aliasing-configs} \rightarrow 2^{\text{LockPaths}} \times \dots$$



# Abstraction Scheme

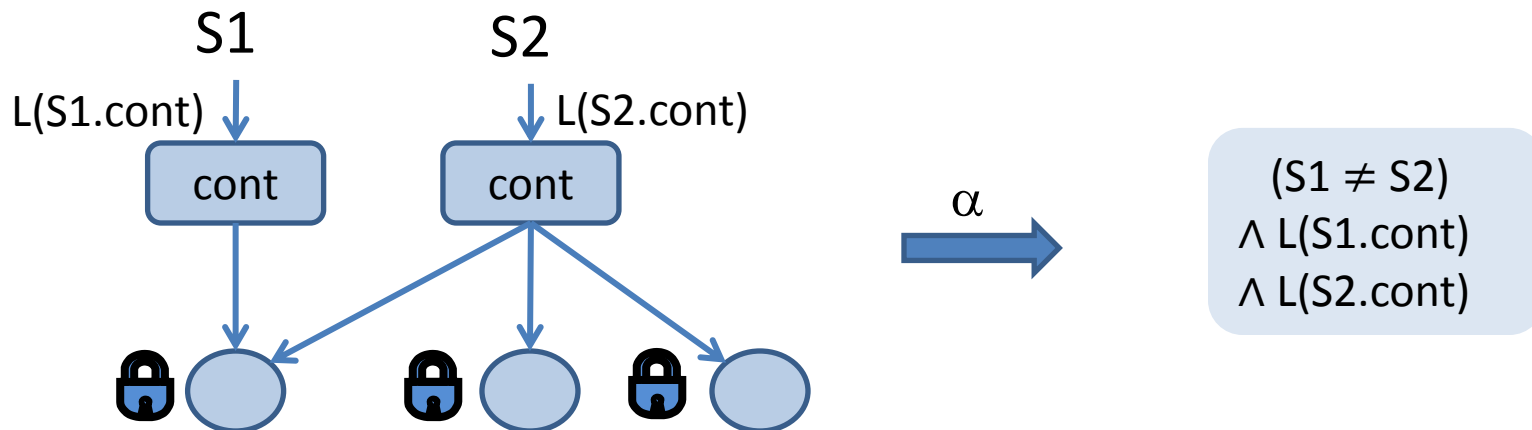
- Parameterized by set of *LockPaths*:

$$L(\text{Path}) \triangleq \forall o . o \in \text{Path} \Rightarrow \text{Locked}(o)$$

– Tracks subset of must-be-locked objects

- Abstract domain elements have the form:

$$\text{Aliasing-configs} \rightarrow 2^{\text{LockPaths}} \times \dots$$



# Joining Abstract States

$( (x \neq y) \wedge L(y.nd) ) \sqcup ( (x \neq y) \wedge L(y.nd) \wedge L(x.rev(src)) ) \sqcup ( (x = y) \wedge L(x.nd) )$

# Joining Abstract States

$( (x \neq y) \wedge L(y.nd) ) \sqcup ( (x \neq y) \wedge L(y.nd) \wedge L(x.rev(src)) ) \sqcup ( (x = y) \wedge L(x.nd) )$

# Joining Abstract States

$( (x \neq y) \wedge L(y.nd) ) \sqcup ( (x \neq y) \wedge L(y.nd) \wedge L(x.rev(src)) ) \sqcup ( (x = y) \wedge L(x.nd) )$

# Joining Abstract States

$( (x \neq y) \wedge L(y.nd) ) \sqcup ( (x \neq y) \wedge L(y.nd) \wedge L(x.rev(src)) ) \sqcup ( (x = y) \wedge L(x.nd) )$

# Joining Abstract States

$( (x \neq y) \wedge L(y.nd) ) \sqcup ( (x \neq y) \wedge L(y.nd) \wedge L(x.rev(src)) ) \sqcup ( (x = y) \wedge L(x.nd) )$

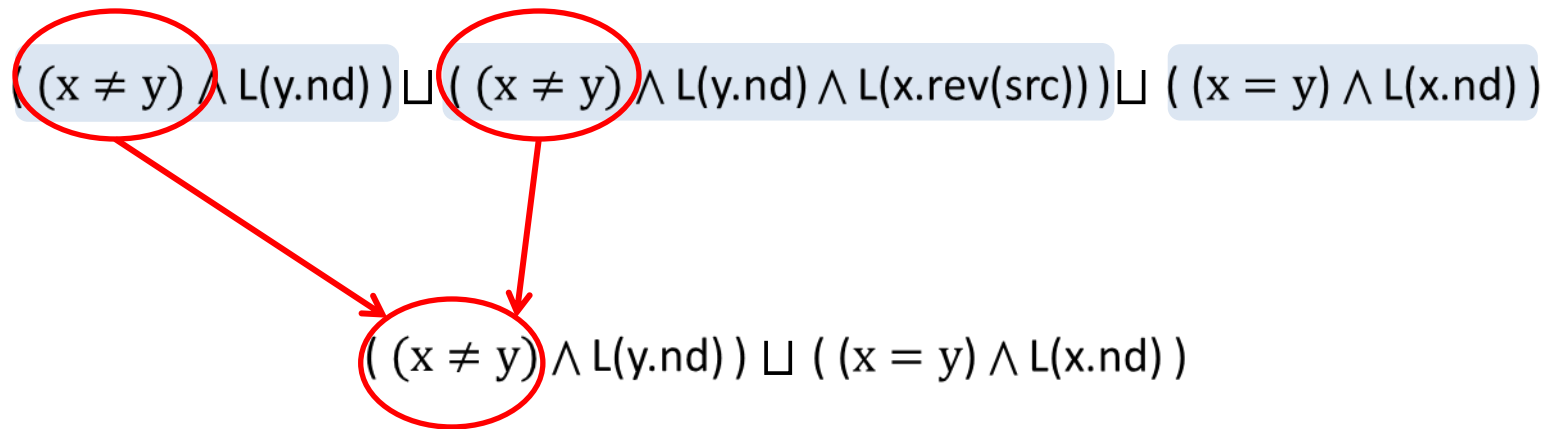
$( (x \neq y) \wedge L(y.nd) ) \sqcup ( (x = y) \wedge L(x.nd) )$

# Joining Abstract States

$$((x \neq y) \wedge L(y.nd)) \sqcup ((x \neq y) \wedge L(y.nd) \wedge L(x.rev(src))) \sqcup ((x = y) \wedge L(x.nd))$$

$$((x \neq y) \wedge L(y.nd)) \sqcup ((x = y) \wedge L(x.nd))$$

# Joining Abstract States



# Joining Abstract States

$$((x \neq y) \wedge L(y.nd)) \sqcup ((x \neq y) \wedge L(y.nd) \wedge L(x.rev(src))) \sqcup ((x = y) \wedge L(x.nd))$$

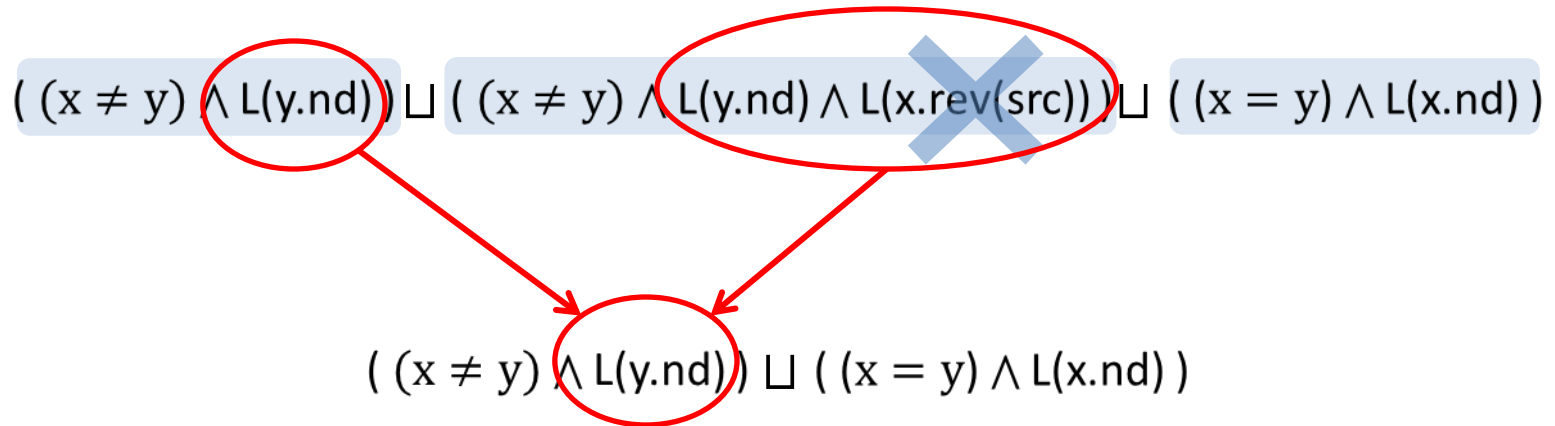
$$((x \neq y) \wedge L(y.nd)) \sqcup ((x = y) \wedge L(x.nd))$$

# Joining Abstract States

$$((x \neq y) \wedge L(y.nd)) \sqcup ((x \neq y) \wedge L(y.nd) \wedge L(x.rev(src))) \sqcup ((x = y) \wedge L(x.nd))$$

$$((x \neq y) \wedge L(y.nd)) \sqcup ((x = y) \wedge L(x.nd))$$

# Joining Abstract States



# Joining Abstract States

$( (x \neq y) \wedge L(y.nd) ) \sqcup ( (x \neq y) \wedge L(y.nd) \wedge L(x.rev(src)) ) \sqcup ( (x = y) \wedge L(x.nd) )$

$( (x \neq y) \wedge L(y.nd) ) \sqcup ( (x = y) \wedge L(x.nd) )$

# Joining Abstract States

$( (x \neq y) \wedge L(y.nd) ) \sqcup ( (x \neq y) \wedge L(y.nd) \wedge L(x.rev(src)) ) \sqcup ( (x = y) \wedge L(x.nd) )$

$( (x \neq y) \wedge L(y.nd) ) \sqcup ( (x = y) \wedge L(x.nd) )$

$( (x \neq y) \wedge L(y.nd) ) \vee ( (x = y) \wedge L(x.nd) )$

# Joining Abstract States

$$((x \neq y) \wedge L(y.nd)) \sqcup ((x \neq y) \wedge L(y.nd) \wedge L(x.rev(src))) \sqcup ((x = y) \wedge L(x.nd))$$

$$((x \neq y) \wedge L(y.nd)) \sqcup ((x = y) \wedge L(x.nd))$$

$$((x \neq y) \wedge L(y.nd)) \vee ((x = y) \wedge L(x.nd))$$

# Joining Abstract States

$( (x \neq y) \wedge L(y.nd) ) \sqcup ( (x \neq y) \wedge L(y.nd) \wedge L(x.rev(src))) \sqcup ( (x = y) \wedge L(x.nd) )$

$( (x \neq y) \wedge L(y.nd) ) \sqcup ( (x = y) \wedge L(x.nd) )$

$( (x \neq y) \wedge L(y.nd) ) \vee ( (x = y) \wedge L(x.nd) )$

Aliasing is crucial for precision  
May-be-locked does not enable our optimizations  
#Aliasing-configs : small constant ( $\approx 6$ )

# Example Invariant in Boruvka

```
GSet<Node> wl = new GSet<Node>();
wl.addAll(g.getNodes());
GBag<Weight> mst = new GBag<Weight>();

foreach (Node a : wl) {
    Set<Node> aNghbrs = g.neighbors(a);
    Node lt = null;
    for (Node n : aNghbrs) {
        minW,lt = minWeightEdge((a,lt), (a,n));
    }

    g.removeEdge(a, lt);
    Set<Node> ltNghbrs = g.neighbors(lt);
    for (Node n : ltNghbrs) {
        Edge e = g.getEdge(lt, n);
        Weight w = g.getEdgeData(e);
        Edge an = g.getEdge(a, n);
        if (an != null) {
            Weight wan = g.getEdgeData(an);
            if (wan.compareTo(w) < 0)
                w = wan;
            g.setEdgeData(an, w);
        } else {
            g.addEdge(a, n, w);
        }
    }
    g.removeNode(lt);
    mst.add(minW);
    wl.add(a);
}
```

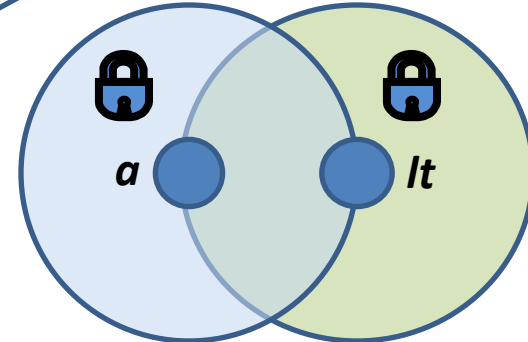
# Example Invariant in Boruvka

```
GSet<Node> w1 = new GSet<Node>();
w1.addAll(g.getNodes());
GBag<Weight> mst = new GBag<Weight>();

foreach (Node a : w1) {
    Set<Node> aNghbrs = g.neighbors(a);
    Node lt = null;
    for (Node n : aNghbrs) {
        minW,lt = minWeightEdge((a,lt), (a,n));
    }

    g.removeEdge(a, lt);
    Set<Node> ltNghbrs = g.neighbors(lt);
    for (Node n : ltNghbrs) {
        Edge e = g.getEdge(lt, n);
        Weight w = g.getEdgeData(e);
        Edge an = g.getEdge(a, n);
        if (an != null) {
            Weight wan = g.getEdgeData(an);
            if (wan.compareTo(w) < 0)
                w = wan;
            g.setEdgeData(an, w);
        } else {
            g.addEdge(a, n, w);
        }
    }
    g.removeNode(lt);
    mst.add(minW);
    w1.add(a);
}
```

The immediate neighbors  
of  $a$  and  $lt$  are locked



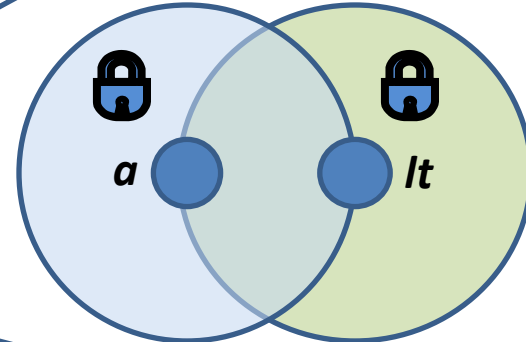
# Example Invariant in Boruvka

```
GSet<Node> wl = new GSet<Node>();
wl.addAll(g.getNodes());
GBag<Weight> mst = new GBag<Weight>();

foreach (Node a : wl) {
    Set<Node> aNghbrs = g.neighbors(a);
    Node lt = null;
    for (Node n : aNghbrs) {
        minW,lt = minWeightEdge((a,lt), (a,n));
    }

    g.removeEdge(a, lt);
    Set<Node> ltNghbrs = g.neighbors(lt);
    for (Node n : ltNghbrs) {
        Edge e = g.getEdge(lt, n);
        Weight w = g.getEdgeData(e);
        Edge an = g.getEdge(a, n);
        if (an != null) {
            Weight wan = g.getEdgeData(an);
            if (wan.compareTo(w) < 0)
                w = wan;
            g.setEdgeData(an, w);
        } else {
            g.addEdge(a, n, w);
        }
    }
    g.removeNode(lt);
    mst.add(minW);
    wl.add(a);
}
```

The immediate neighbors  
of  $a$  and  $lt$  are locked



$(a \neq lt)$

$\wedge L(a) \wedge L(a.rev(src)) \wedge L(a.rev(dst))$   
 $\wedge L(a.rev(src).dst) \wedge L(a.rev(dst).src)$   
 $\wedge L(lt) \wedge L(lt.rev(dst)) \wedge L(lt.rev(src))$   
 $\wedge L(lt.rev(dst).src) \wedge L(lt.rev(src).dst)$

.....

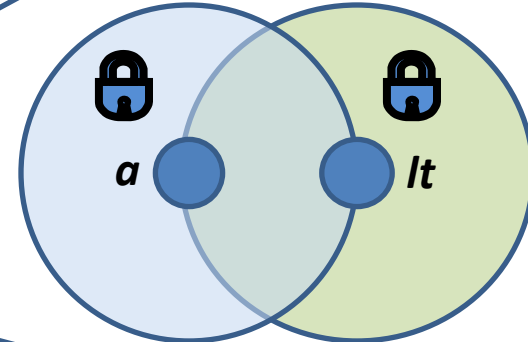
# Example Invariant in Boruvka

```
GSet<Node> wl = new GSet<Node>();
wl.addAll(g.getNodes());
GBag<Weight> mst = new GBag<Weight>();

foreach (Node a : wl) {
    Set<Node> aNghbrs = g.neighbors(a);
    Node lt = null;
    for (Node n : aNghbrs) {
        minW,lt = minWeightEdge((a,lt), (a,n));
    }

    g.removeEdge(a, lt);
    Set<Node> ltNghbrs = g.neighbors(lt);
    for (Node n : ltNghbrs) {
        Edge e = g.getEdge(lt, n);
        Weight w = g.getEdgeData(e);
        Edge an = g.getEdge(a, n);
        if (an != null) {
            Weight wan = g.getEdgeData(an);
            if (wan.compareTo(w) < 0)
                w = wan;
            g.setEdgeData(an, w);
        } else {
            g.addEdge(a, n, w);
        }
    }
    g.removeNode(lt);
    mst.add(minW);
    wl.add(a);
}
```

The immediate neighbors  
of  $a$  and  $lt$  are locked



$(a \neq lt)$

$\wedge L(a) \wedge L(a.rev(src)) \wedge L(a.rev(dst))$   
 $\wedge L(a.rev(src).dst) \wedge L(a.rev(dst).src)$   
 $\wedge L(lt) \wedge L(lt.rev(dst)) \wedge L(lt.rev(src))$   
 $\wedge L(lt.rev(dst).src) \wedge L(lt.rev(src).dst)$

.....

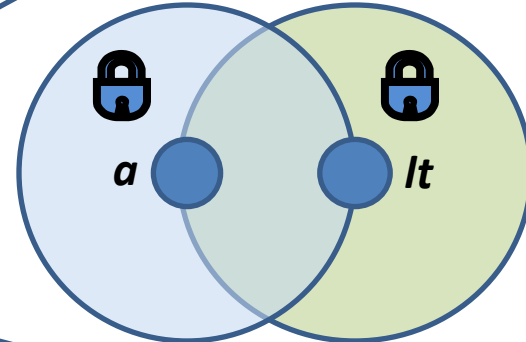
# Example Invariant in Boruvka

```
GSet<Node> wl = new GSet<Node>();
wl.addAll(g.getNodes());
GBag<Weight> mst = new GBag<Weight>();

foreach (Node a : wl) {
    Set<Node> aNghbrs = g.neighbors(a);
    Node lt = null;
    for (Node n : aNghbrs) {
        minW,lt = minWeightEdge((a,lt), (a,n));
    }

    g.removeEdge(a, lt);
    Set<Node> ltNghbrs = g.neighbors(lt);
    for (Node n : ltNghbrs) {
        Edge e = g.getEdge(lt, n);
        Weight w = g.getEdgeData(e);
        Edge an = g.getEdge(a, n);
        if (an != null) {
            Weight wan = g.getEdgeData(an);
            if (wan.compareTo(w) < 0)
                w = wan;
            g.setEdgeData(an, w);
        } else {
            g.addEdge(a, n, w);
        }
    }
    g.removeNode(lt);
    mst.add(minW);
    wl.add(a);
}
```

The immediate neighbors  
of  $a$  and  $lt$  are locked



$(a \neq lt)$

$\wedge L(a) \wedge L(a.rev(src)) \wedge L(a.rev(dst))$

$\wedge L(a.rev(src).dst) \wedge L(a.rev(dst).src)$

$\wedge L(lt) \wedge L(lt.rev(dst)) \wedge L(lt.rev(src))$

$\wedge L(lt.rev(dst).src) \wedge L(lt.rev(src).dst)$

.....

# Heuristics for Finding Paths

- **Hierarchy Summarization (HS)**

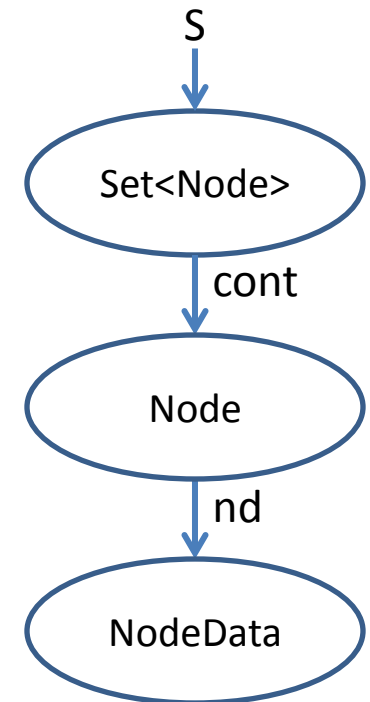
- $x.(fld)^*$

- Type hierarchy graph acyclic  $\rightarrow$   
bounded number of paths

- Preflow-Push:

- $L(S.cont) \wedge L(S.cont.nd)$

- Nodes in set S and their data are locked



# Footprint Graph Heuristic

- **Footprint Graphs (FG)** [*Calcagno et al. SAS'07*]
  - All acyclic paths from arguments of ADT method to locked objects
  - $x.( fld \mid rev(fld) )^*$
  - Delaunay Refinement:  
 $L(S.cont) \wedge L(S.cont.rev(src)) \wedge L(S.cont.rev(dst))$   
 $\wedge L(S.cont.rev(src).dst) \wedge L(S.cont.rev(dst).src)$
  - Nodes in set S and all of their immediate neighbors are locked
- **Composition of HS, FG**
  - Preflow-Push:  $L(a.rev(src).ed)$

# Footprint Graph Heuristic

- **Footprint Graphs (FG)** [Calcagno et al. SAS'07]
  - All acyclic paths from arguments of ADT method to locked objects
  - $x.( fld \mid rev(fld) )^*$
  - Delaunay Refinement:  
 $L(S.cont) \wedge L(S.cont.rev(src)) \wedge L(S.cont.rev(dst))$   
 $\wedge L(S.cont.rev(src).dst) \wedge L(S.cont.rev(dst).src)$
  - Nodes in set S and all of their immediate neighbors are locked
- **Composition of HS, FG**
  - Preflow-Push:  $L(a.\underbrace{rev(src).ed}_{FG})$

# Footprint Graph Heuristic

- **Footprint Graphs (FG)** [Calcagno et al. SAS'07]
  - All acyclic paths from arguments of ADT method to locked objects
  - $x.(fld \mid rev(fld))^*$
  - Delaunay Refinement:  
 $L(S.cont) \wedge L(S.cont.rev(src)) \wedge L(S.cont.rev(dst))$   
 $\wedge L(S.cont.rev(src).dst) \wedge L(S.cont.rev(dst).src)$
  - Nodes in set S and all of their immediate neighbors are locked

- **Composition of HS, FG**

- Preflow-Push:  $L(a.rev(src).ed)$



# Organization

- Parallelization of graph algorithms in Galois system
  - Speculative execution
  - Example: Boruvka MST algorithm
- Optimization opportunities
  - Reduce speculation overheads
  - Analysis problem: LockSet shape analysis
- Shape analysis
  - Abstract Data Type modeling
  - Hierarchy summarization abstraction
  - Predicate discovery
- Evaluation
  - Fast and infers all available optimizations
  - Optimizations give speedup up to 12x

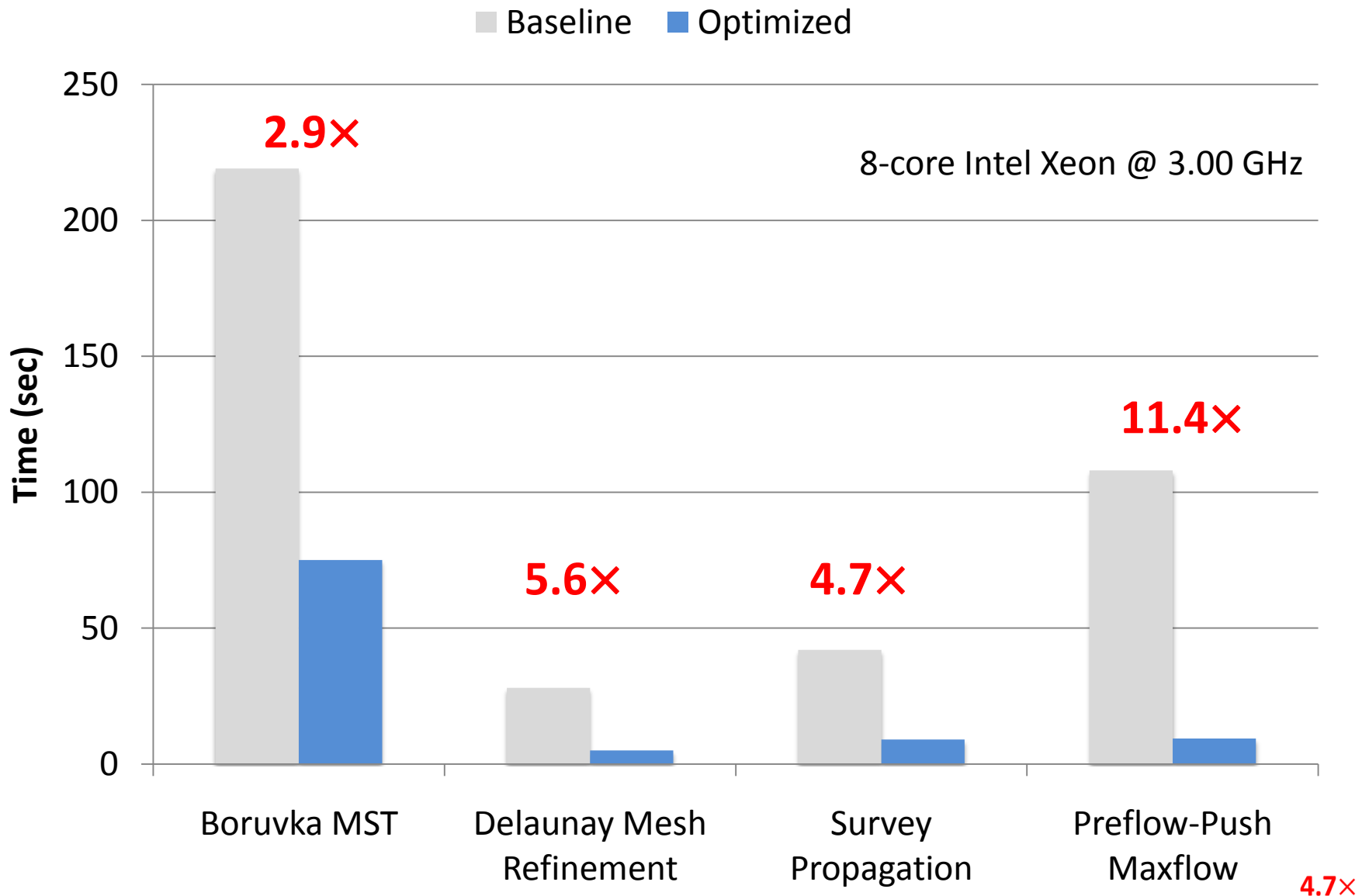
# Experimental Evaluation

- Implement on top of TVLA
  - Encode abstraction by 3-Valued Shape Analysis  
[*SRW TOPLAS'02*]
- Evaluation on 4 Lonestar Java benchmarks

Benchmark	Analysis Time (sec)
Boruvka MST	6
Preflow-Push Maxflow	7
Survey Propagation	12
Delaunay Mesh Refinement	16

- Inferred all available optimizations
- # abstract states practically linear in program size

# Impact of Optimizations for 8 Threads



# Related Work

- Safe programmable speculative parallelism [*Prabhu et al. PLDI'10*]
  - Focused on value speculation on ordered algorithms
  - Different rollback freedom condition
- Transactional Memory compiler optimizations [*Harris et al. PLDI'06, Dragojevic et al. SPAA'09*]
  - Similar optimizations
  - Don't target rollback freedom
  - Imprecise for unbounded data-structures
- Optimizations for parallel graph programs [*Mendez-Lojo et al. PPOPP'10*]
  - Manual optimizations
  - Failsafe subsumes cautious
- Verifying conformance of ADT implementation to specification
  - The Jahob project (*Kuncak, Rinard, Wies et al.*)

# Conclusion

- New application for static analysis
  - Optimization of optimistically parallelized graph programs
- Novel shape analysis
  - Utilize observations on the structure of concrete states and programming style
- Enables optimizations crucial for performance

Thank You!