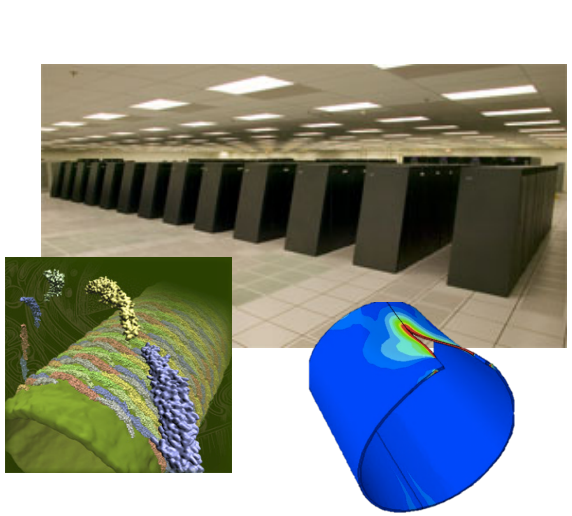
Galois: A System for Parallel Execution of Irregular Algorithms

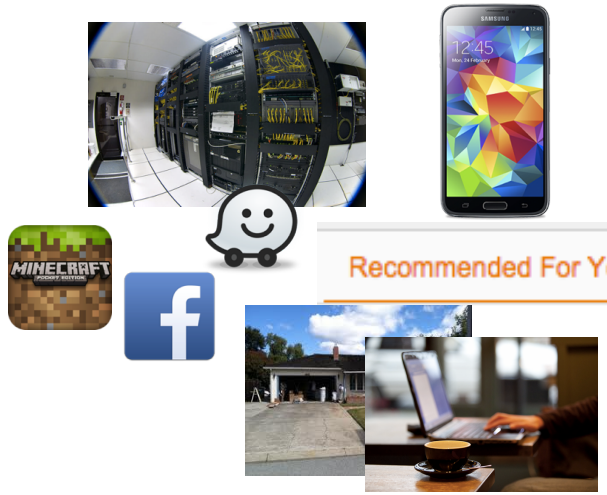
Donald Nguyen

The University of Texas at Austin

Parallel Programming is Changing



Old World

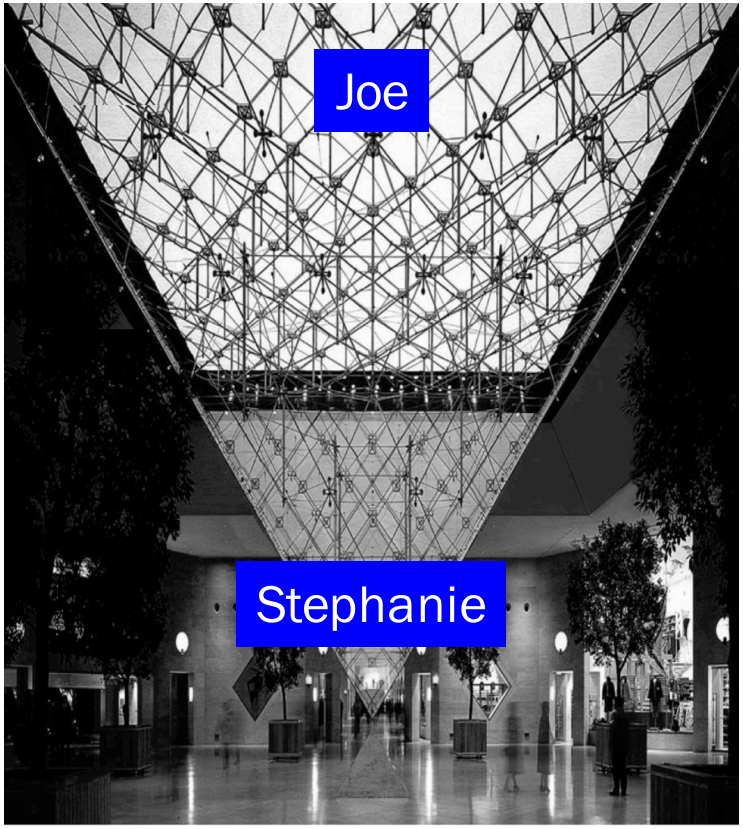


New World

- Data
  - Structured (vector, matrix) versus unstructured (graphs)
- Platforms
  - Dedicated clusters versus cloud, mobile
- People
  - Small number of highly trained scientists versus large number of self-trained parallel programmers

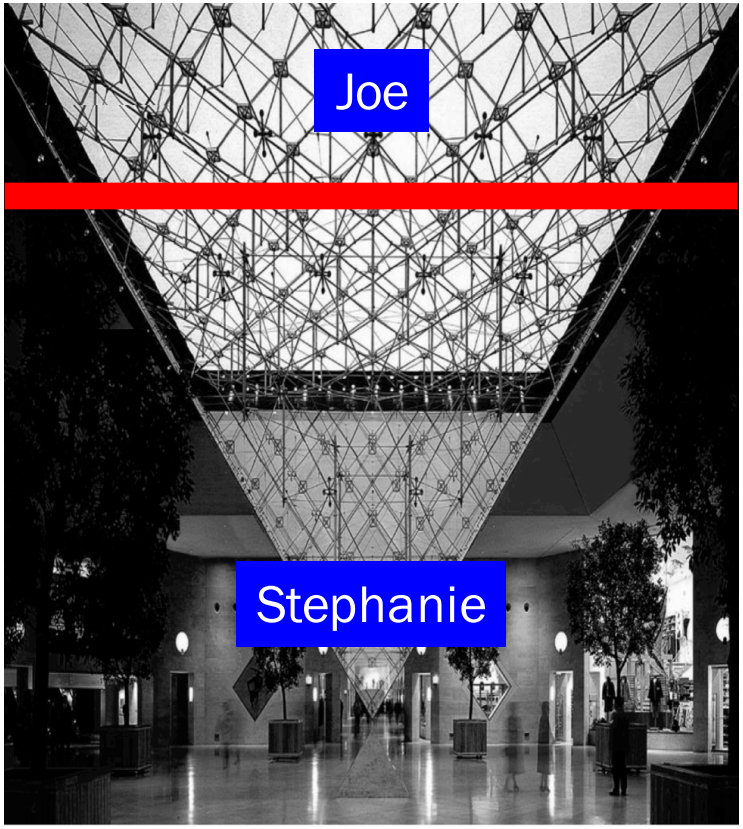
The Search for Good Programming Models

- Tension between productivity and performance
  - Support large number of application programmers with small number of expert parallel programmers
  - Performance comparable to hand-written codes
- Galois project



The Search for Good Programming Models

- Tension between productivity and performance
  - Support large number of application programmers with small number of expert parallel programmers
  - Performance comparable to hand-written codes
- Galois project

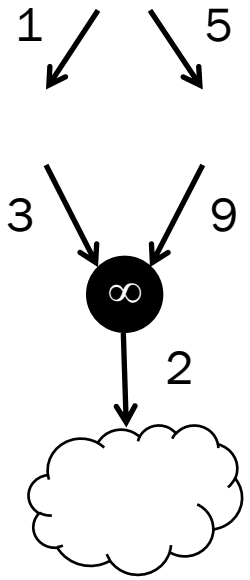


Outline

- Parallel Program = Operator + Schedule + Parallel Data Structure
- Galois system implementation
  - Operator, data structures, scheduling
- Galois studies
  - Intel study
  - Deterministic scheduling
  - Adding a scheduler: sparse tiling
  - Comparison with transactional memory
  - Implementing other programming models

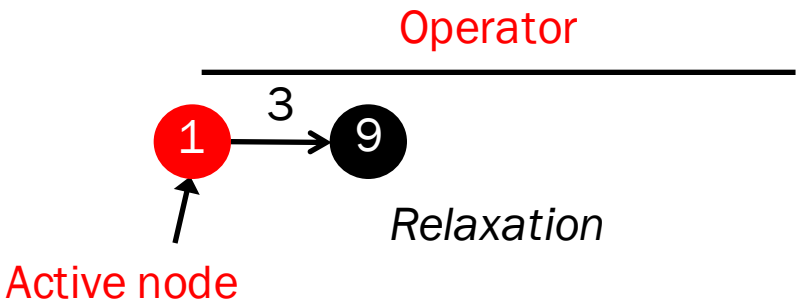
SSSP

- Find the shortest distance from source node to all other nodes in a graph
  - Label nodes with tentative distance
  - Assume non-negative edge weights
  - BFS is a special case
- Algorithms
  - Chaotic relaxation O(2^V)
  - Bellman-Ford O(VE)
  - Dijkstra's algorithm O(E log V)
    - Uses priority queue
  - Δ-stepping
    - Uses sequence of bags to prioritize work
    - Δ=1: Dijkstra
    - Δ=∞: Chaotic relaxation
- Different algorithms are different schedules for applying relaxations
  - Improved work efficiency
  - Input sensitivity
  - Locality



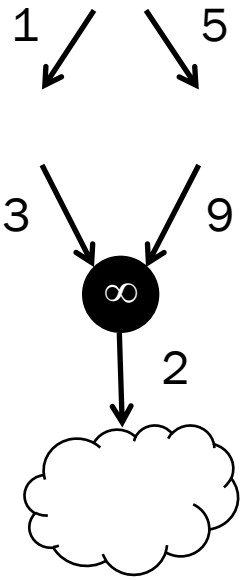
SSSP

- Find the shortest distance from source node to all other nodes in a graph
  - Label nodes with tentative distance
  - Assume non-negative edge weights
  - BFS is a special case



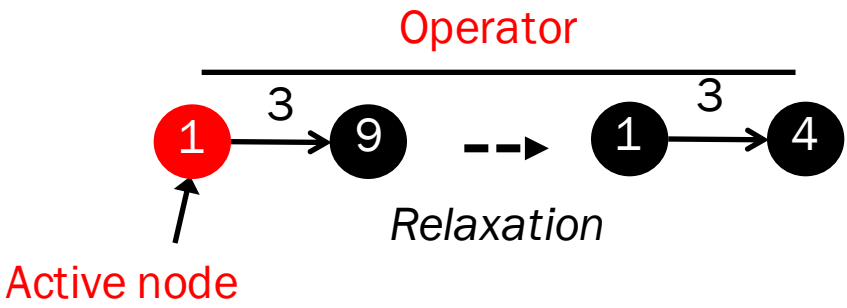
- Algorithms
  - Chaotic relaxation O(2^V)
  - Bellman-Ford O(VE)
  - Dijkstra's algorithm O(E log V)
    - Uses priority queue
  - Δ-stepping
    - Uses sequence of bags to prioritize work
    - Δ=1: Dijkstra
    - Δ=∞: Chaotic relaxation

- Different algorithms are different schedules for applying relaxations
  - Improved work efficiency
  - Input sensitivity
  - Locality



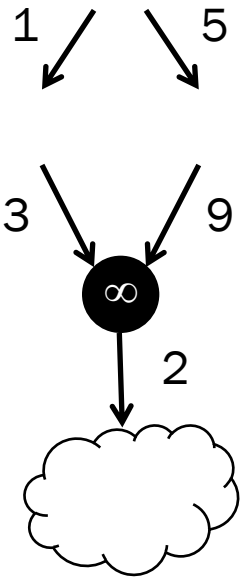
SSSP

- Find the shortest distance from source node to all other nodes in a graph
  - Label nodes with tentative distance
  - Assume non-negative edge weights
  - BFS is a special case



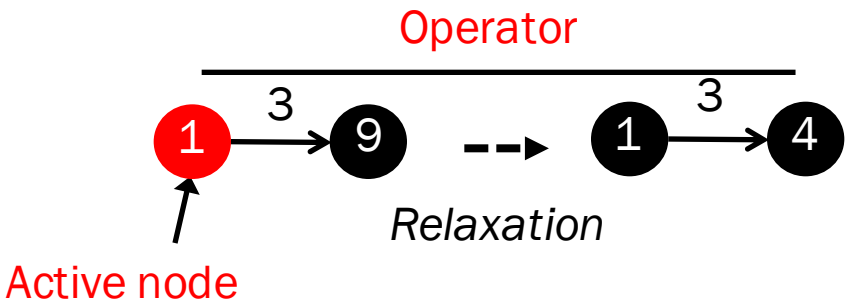
- Algorithms
  - Chaotic relaxation $O(2^V)$
  - Bellman-Ford $O(VE)$
  - Dijkstra's algorithm $O(E \log V)$
    - Uses priority queue
  - Δ-stepping
    - Uses sequence of bags to prioritize work
    - Δ=1: Dijkstra
    - Δ=∞: Chaotic relaxation

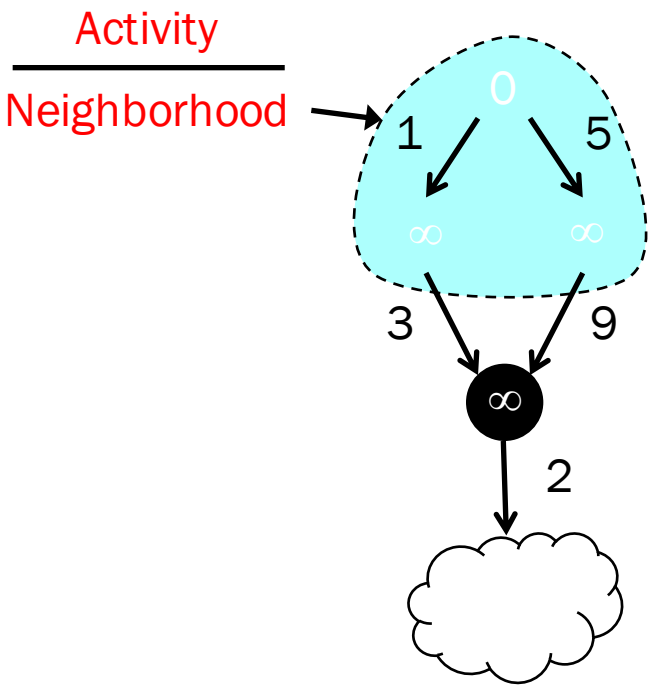
- Different algorithms are different schedules for applying relaxations
  - Improved work efficiency
  - Input sensitivity
  - Locality



SSSP

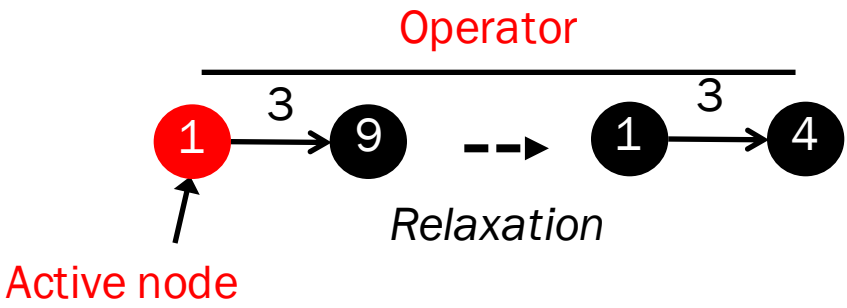
- Find the shortest distance from source node to all other nodes in a graph
  - Label nodes with tentative distance
  - Assume non-negative edge weights
  - BFS is a special case
- Algorithms
  - Chaotic relaxation O(2^V)
  - Bellman-Ford O(VE)
  - Dijkstra's algorithm O(E log V)
    - Uses priority queue
  - Δ-stepping
    - Uses sequence of bags to prioritize work
    - Δ=1: Dijkstra
    - Δ=∞: Chaotic relaxation
- Different algorithms are different schedules for applying relaxations
  - Improved work efficiency
  - Input sensitivity
  - Locality



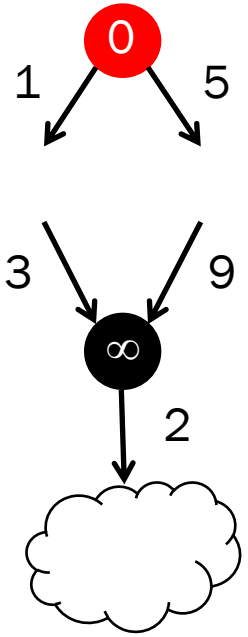


SSSP

- Find the shortest distance from source node to all other nodes in a graph
  - Label nodes with tentative distance
  - Assume non-negative edge weights
  - BFS is a special case



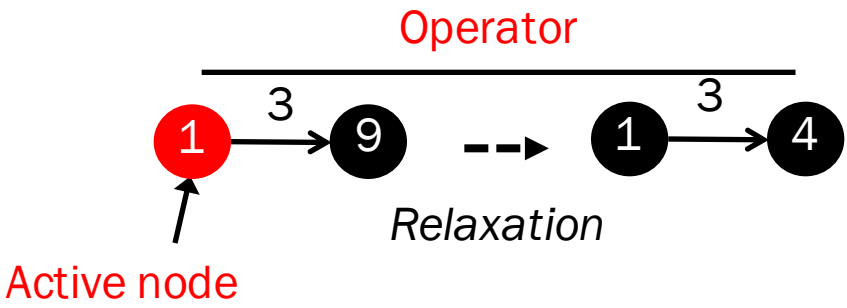
- Algorithms
  - Chaotic relaxation O(2^V)
  - Bellman-Ford O(VE)
  - Dijkstra's algorithm O(E log V)
    - Uses priority queue
  - Δ-stepping
    - Uses sequence of bags to prioritize work
    - Δ=1: Dijkstra
    - Δ=∞: Chaotic relaxation



- Different algorithms are different schedules for applying relaxations
  - Improved work efficiency
  - Input sensitivity
  - Locality

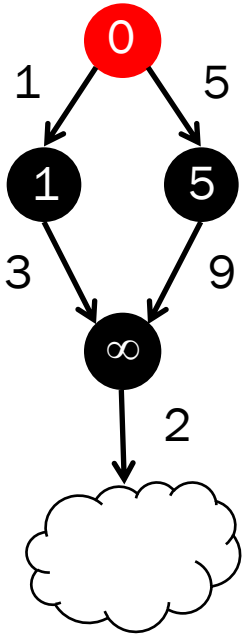
SSSP

- Find the shortest distance from source node to all other nodes in a graph
  - Label nodes with tentative distance
  - Assume non-negative edge weights
  - BFS is a special case



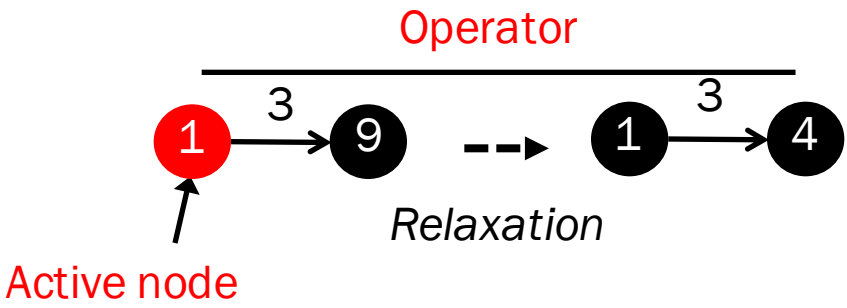
- Algorithms
  - Chaotic relaxation $O(2^V)$
  - Bellman-Ford $O(VE)$
  - Dijkstra's algorithm $O(E \log V)$
    - Uses priority queue
  - Δ-stepping
    - Uses sequence of bags to prioritize work
    - Δ=1: Dijkstra
    - Δ=∞: Chaotic relaxation

- Different algorithms are different schedules for applying relaxations
  - Improved work efficiency
  - Input sensitivity
  - Locality

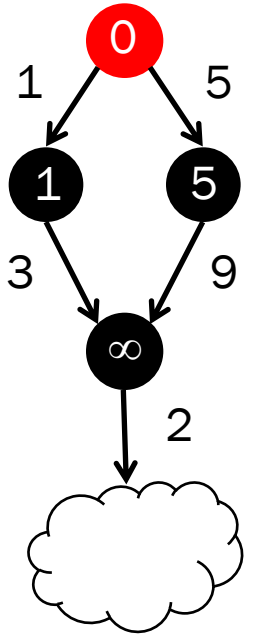


SSSP

- Find the shortest distance from source node to all other nodes in a graph
  - Label nodes with tentative distance
  - Assume non-negative edge weights
  - BFS is a special case



- Algorithms
  - Chaotic relaxation O(2^V)
  - Bellman-Ford O(VE)
  - Dijkstra's algorithm O(E log V)
    - Uses priority queue
  - Δ-stepping
    - Uses sequence of bags to prioritize work
    - Δ=1: Dijkstra
    - Δ=∞: Chaotic relaxation

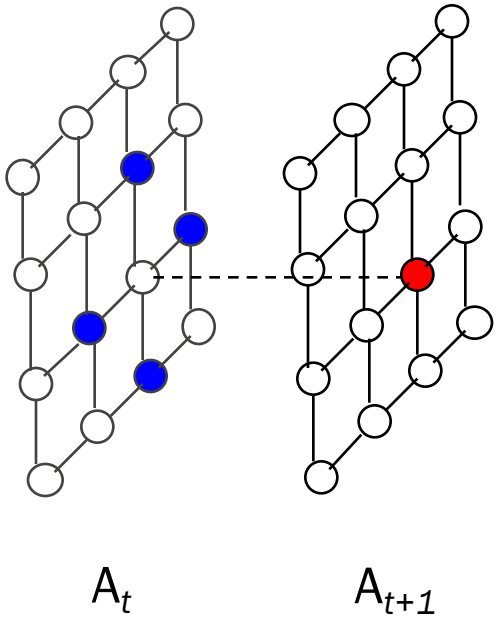


- Different algorithms are different schedules for applying relaxations
  - Improved work efficiency
  - Input sensitivity
  - Locality

Algorithm = Operator + Schedule

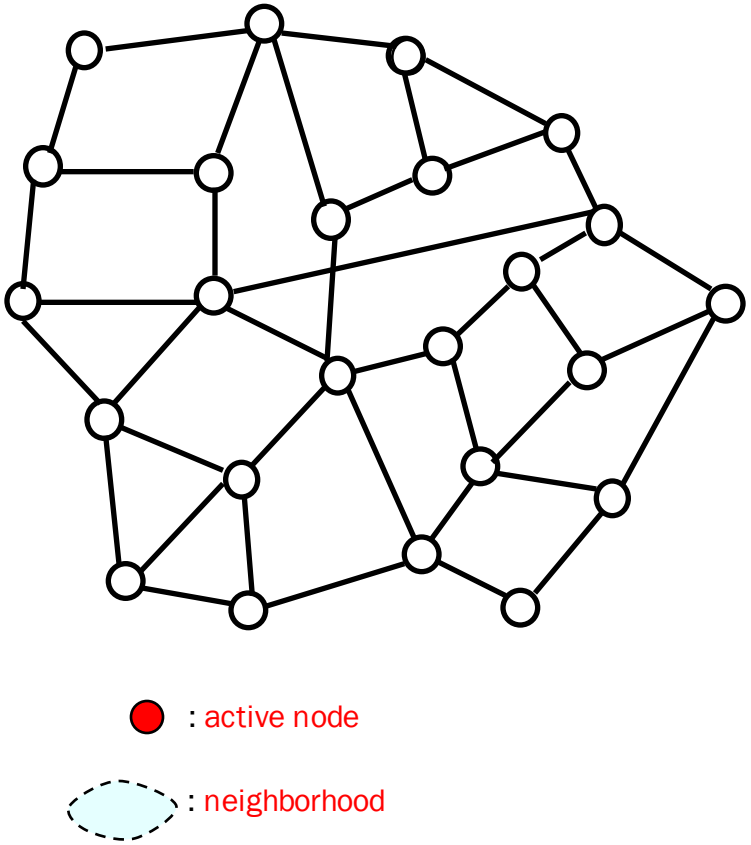
Stencil Computation

- Finite difference computation
  - Active nodes: nodes in A_{t+1}
  - Operator: 5 point stencil
  - Different schedules have different locality
- Regular application
  - Grid structure and active nodes known statically
  - Can be parallelized by a compiler



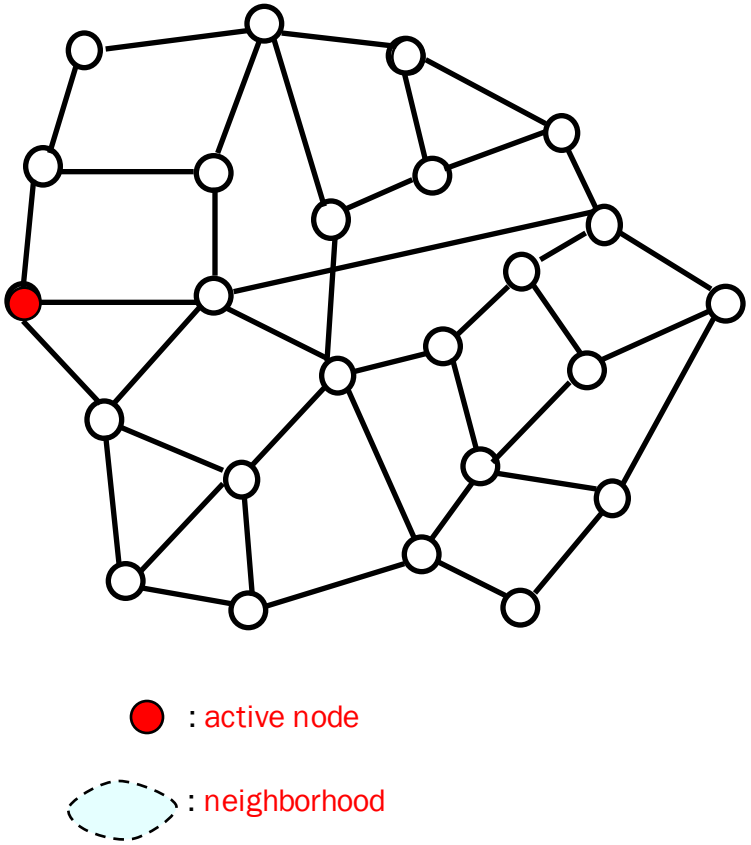
Abstraction of Algorithms

- Operator formulation
  - Active elements: nodes or edges where there is work to be done
  - Operator: computation at active element
    - Activity: application of operator to active element
    - Neighborhood: graph elements read or written by activity
  - Ordering: order in which active elements must appear to have been processed
    - Unordered algorithms: any order is fine (e.g., chaotic relaxation) but may have soft priorities
    - Ordered algorithms: algorithm-specific order (e.g., discrete event simulation)
  - Parallelism: process activities in parallel subject to neighborhood and ordering constraints
    - Data parallelism is a special case (all nodes active, no conflicts)



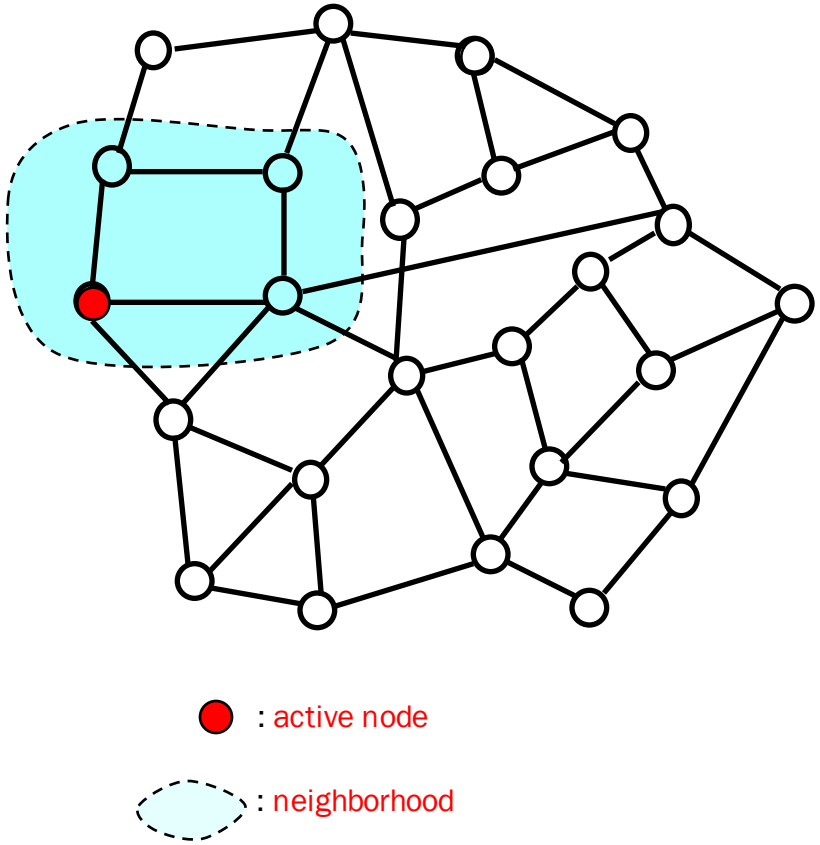
Abstraction of Algorithms

- Operator formulation
  - Active elements: nodes or edges where there is work to be done
  - Operator: computation at active element
    - Activity: application of operator to active element
    - Neighborhood: graph elements read or written by activity
  - Ordering: order in which active elements must appear to have been processed
    - Unordered algorithms: any order is fine (e.g., chaotic relaxation) but may have soft priorities
    - Ordered algorithms: algorithm-specific order (e.g., discrete event simulation)
  - Parallelism: process activities in parallel subject to neighborhood and ordering constraints
    - Data parallelism is a special case (all nodes active, no conflicts)



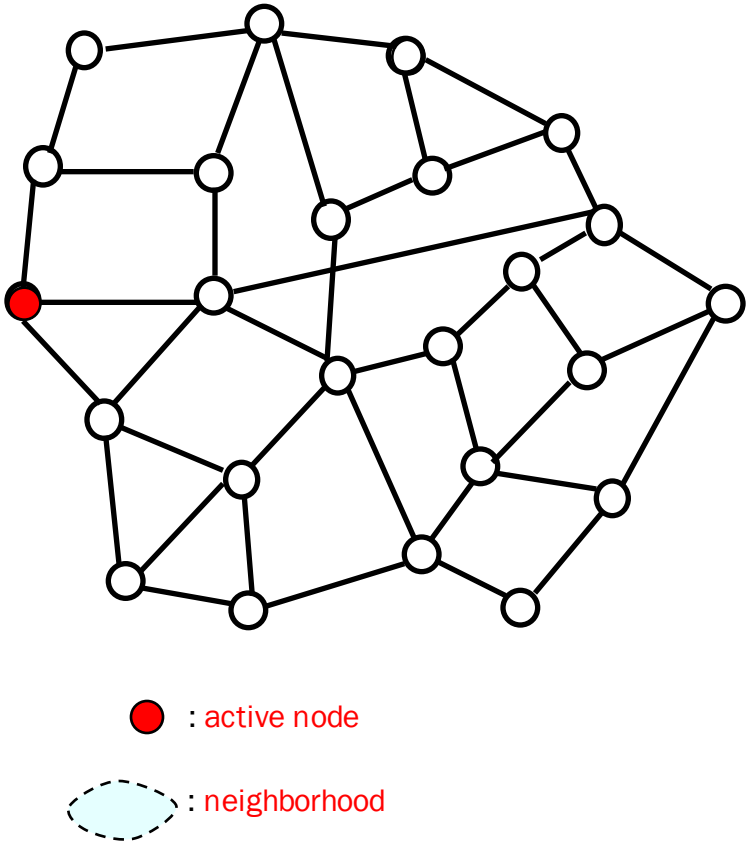
Abstraction of Algorithms

- Operator formulation
  - Active elements: nodes or edges where there is work to be done
  - Operator: computation at active element
    - Activity: application of operator to active element
    - Neighborhood: graph elements read or written by activity
  - Ordering: order in which active elements must appear to have been processed
    - Unordered algorithms: any order is fine (e.g., chaotic relaxation) but may have soft priorities
    - Ordered algorithms: algorithm-specific order (e.g., discrete event simulation)
  - Parallelism: process activities in parallel subject to neighborhood and ordering constraints
    - Data parallelism is a special case (all nodes active, no conflicts)



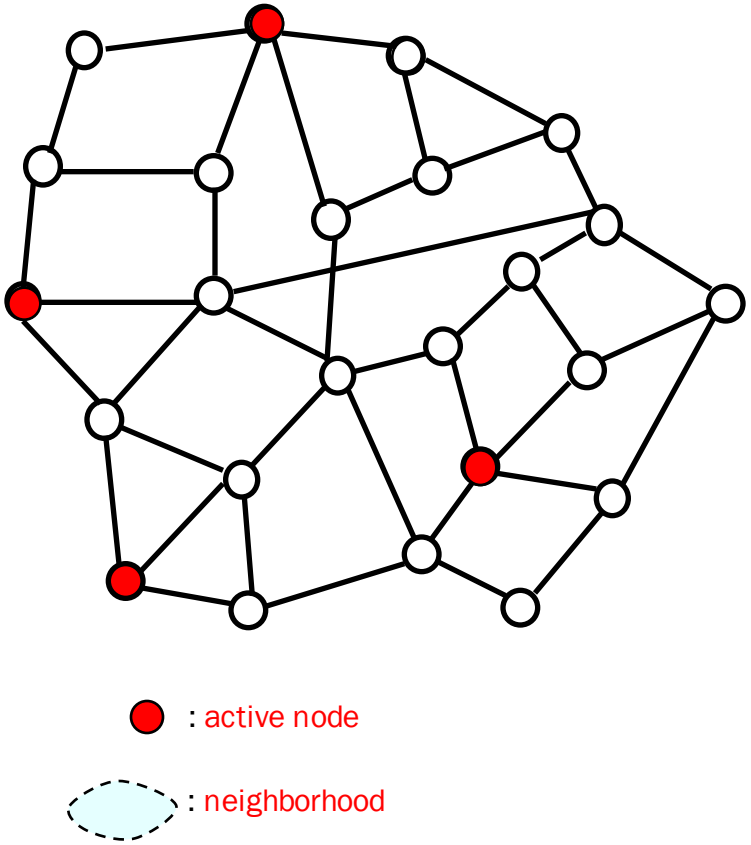
Abstraction of Algorithms

- Operator formulation
  - Active elements: nodes or edges where there is work to be done
  - Operator: computation at active element
    - Activity: application of operator to active element
    - Neighborhood: graph elements read or written by activity
  - Ordering: order in which active elements must appear to have been processed
    - Unordered algorithms: any order is fine (e.g., chaotic relaxation) but may have soft priorities
    - Ordered algorithms: algorithm-specific order (e.g., discrete event simulation)
  - Parallelism: process activities in parallel subject to neighborhood and ordering constraints
    - Data parallelism is a special case (all nodes active, no conflicts)



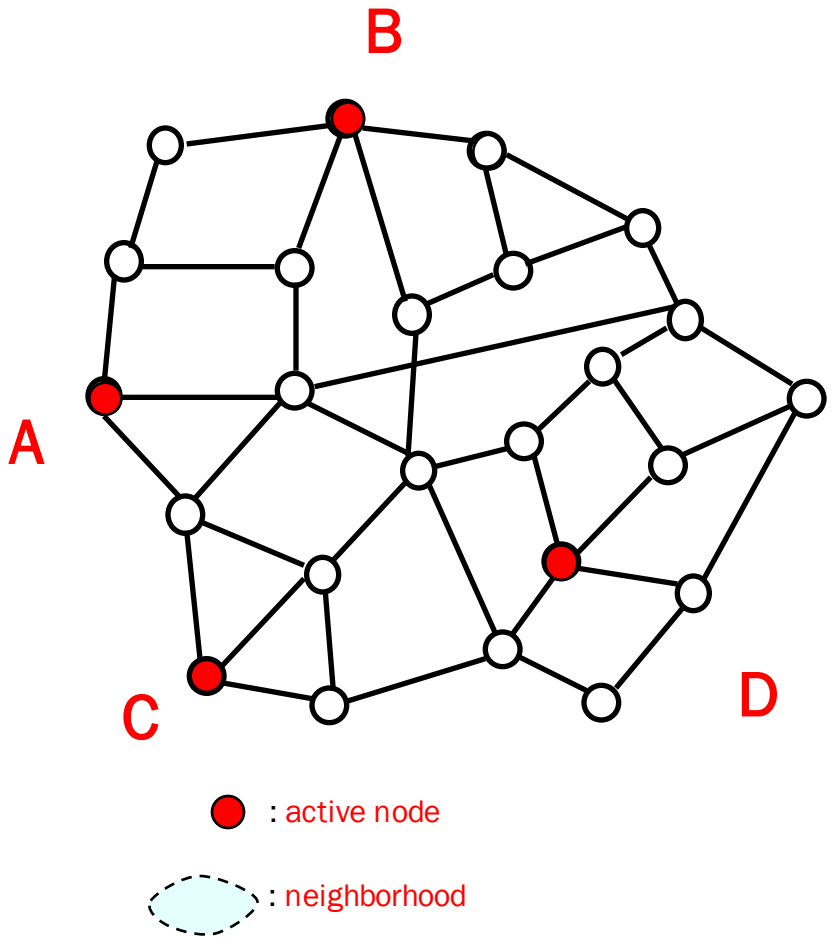
Abstraction of Algorithms

- Operator formulation
  - Active elements: nodes or edges where there is work to be done
  - Operator: computation at active element
    - Activity: application of operator to active element
    - Neighborhood: graph elements read or written by activity
  - Ordering: order in which active elements must appear to have been processed
    - Unordered algorithms: any order is fine (e.g., chaotic relaxation) but may have soft priorities
    - Ordered algorithms: algorithm-specific order (e.g., discrete event simulation)
  - Parallelism: process activities in parallel subject to neighborhood and ordering constraints
    - Data parallelism is a special case (all nodes active, no conflicts)



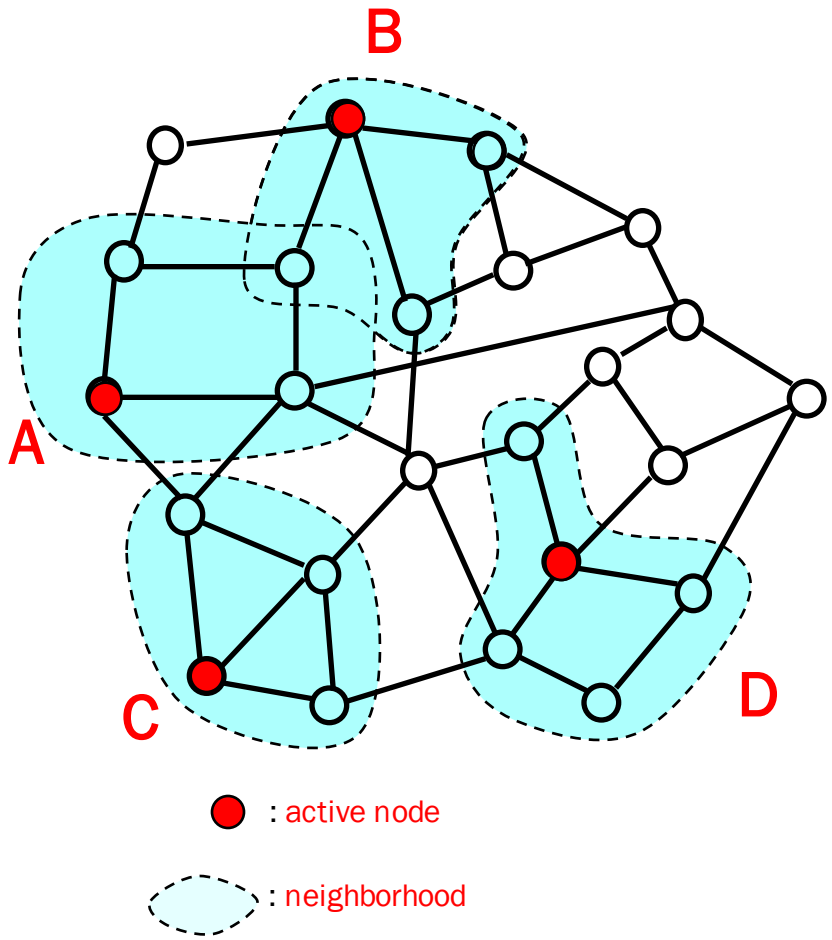
Abstraction of Algorithms

- Operator formulation
  - Active elements: nodes or edges where there is work to be done
  - Operator: computation at active element
    - Activity: application of operator to active element
    - Neighborhood: graph elements read or written by activity
  - Ordering: order in which active elements must appear to have been processed
    - Unordered algorithms: any order is fine (e.g., chaotic relaxation) but may have soft priorities
    - Ordered algorithms: algorithm-specific order (e.g., discrete event simulation)
  - Parallelism: process activities in parallel subject to neighborhood and ordering constraints
    - Data parallelism is a special case (all nodes active, no conflicts)

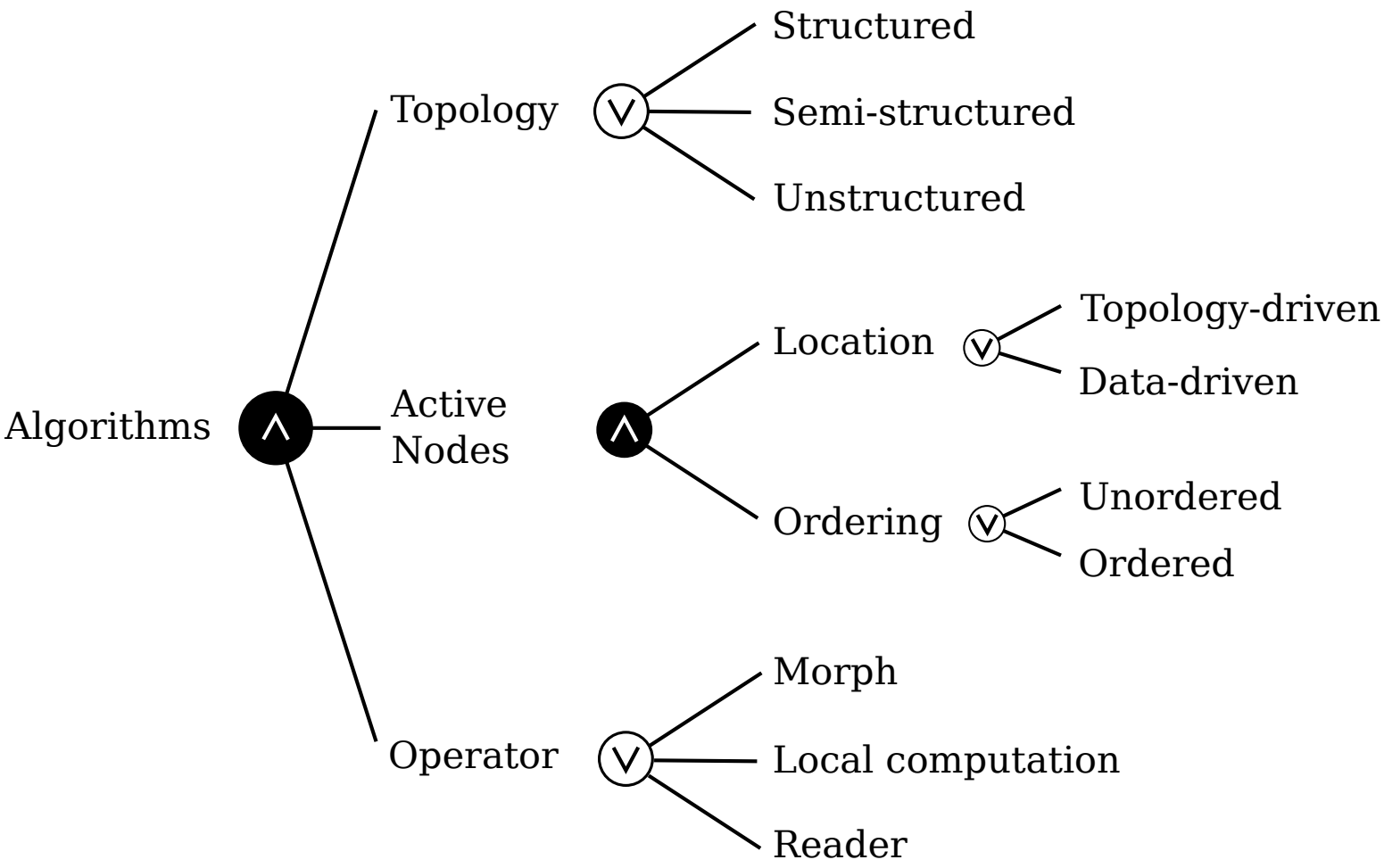


Abstraction of Algorithms

- Operator formulation
  - Active elements: nodes or edges where there is work to be done
  - Operator: computation at active element
    - Activity: application of operator to active element
    - Neighborhood: graph elements read or written by activity
  - Ordering: order in which active elements must appear to have been processed
    - Unordered algorithms: any order is fine (e.g., chaotic relaxation) but may have soft priorities
    - Ordered algorithms: algorithm-specific order (e.g., discrete event simulation)
  - Parallelism: process activities in parallel subject to neighborhood and ordering constraints
    - Data parallelism is a special case (all nodes active, no conflicts)

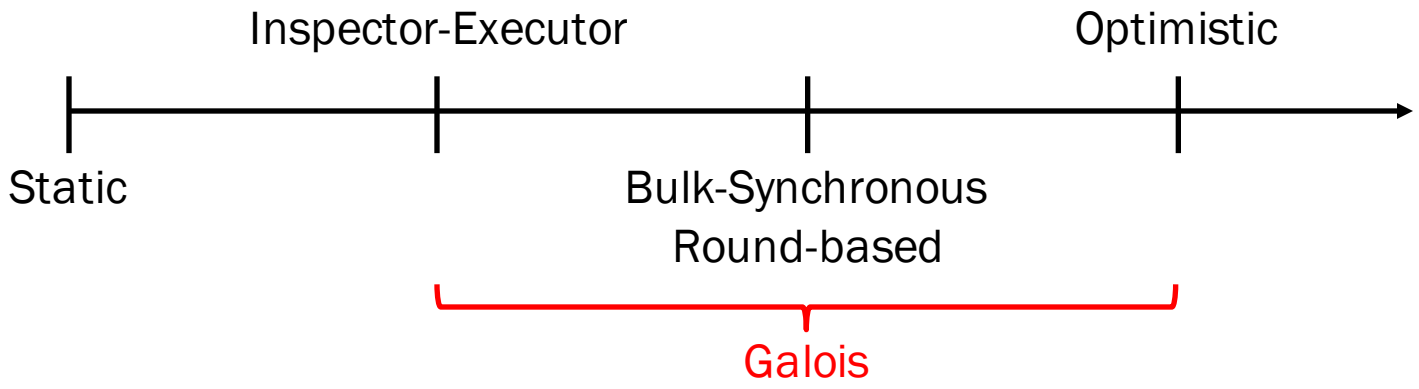


TAO of Parallelism



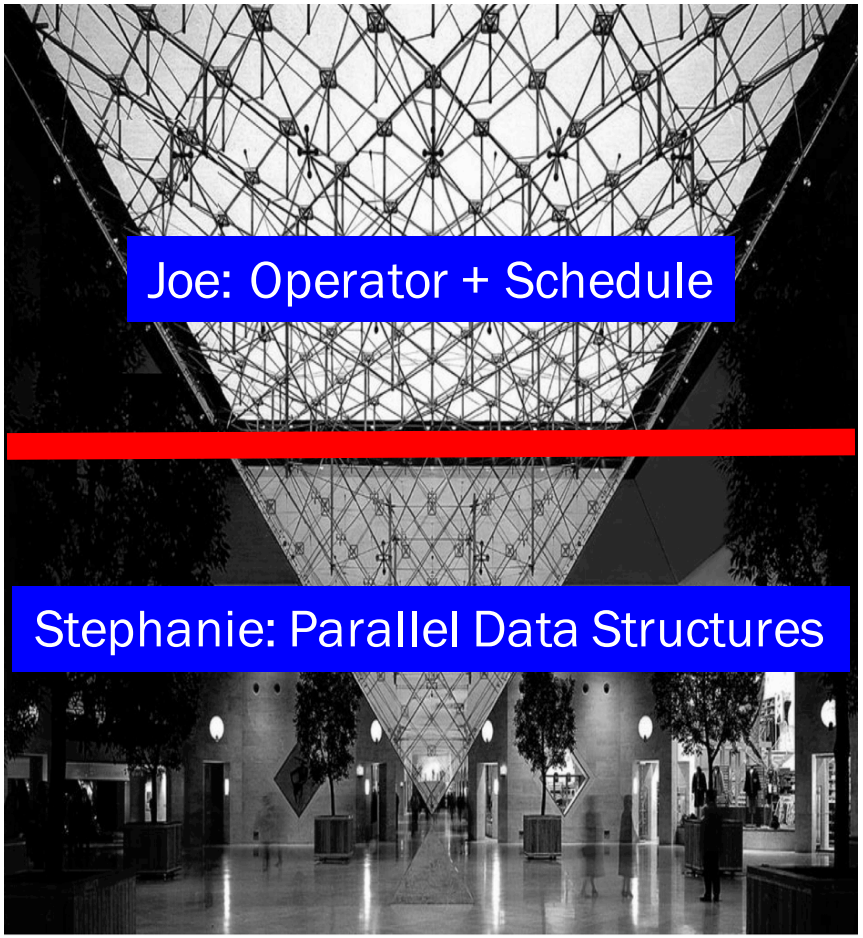
Pingali, Nguyen, et al. The tao of parallelism in algorithms. PLDI 2011.

TAO of Parallelism



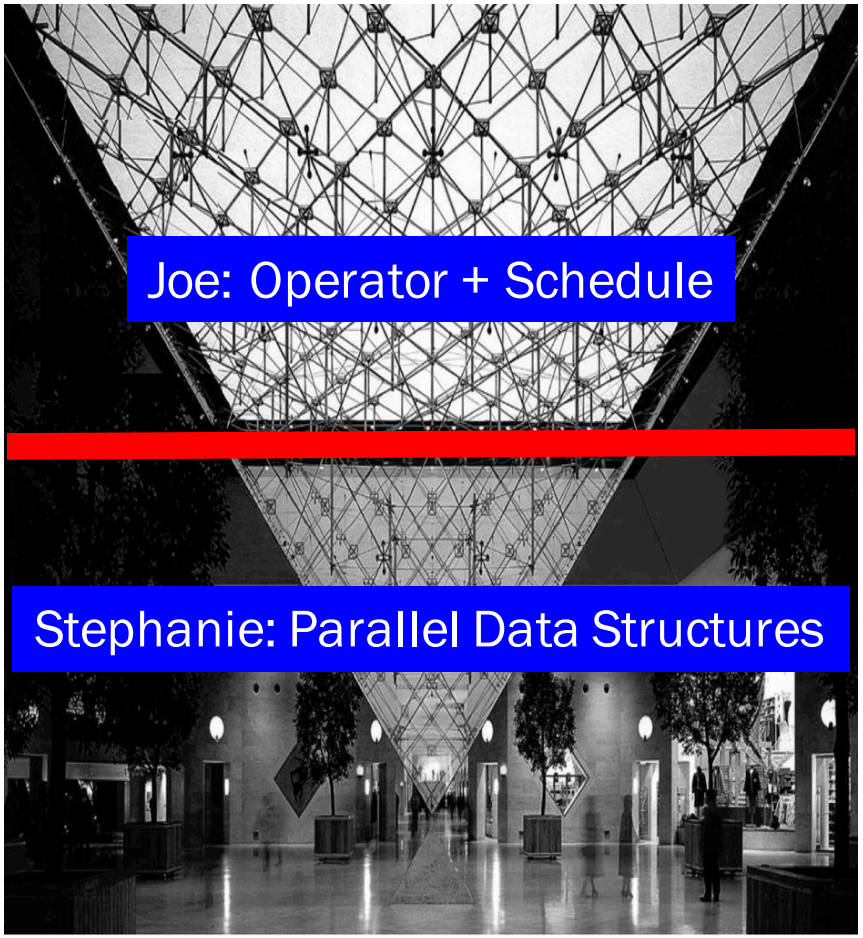
Pingali, Nguyen, et al. The tao of parallelism in algorithms. PLDI 2011.

Galois Project



Program = Algorithm + Data Structure

Galois Project



Program = Algorithm + Data Structure

Parallel Program = Operator + Schedule + Parallel Data Structure

My Contribution

Design and implementation of a programming model for high-performance parallelism by factoring programs into

operator + schedule + data structure

Themes

- Joe
  - Program = Operator + Schedule + Data Structure
- Stephanie
  - Scalability principle 1: disjoint access
    - Accesses to distinct logical entities should be disjoint physically as well
    - "Do no harm"
  - Scalability principle 2: virtualize tasks
    - Execution of tasks should not be eagerly tied to particular physical resources
    - "Be flexible"
- Challenges
  - Easy to introduce inadvertent sharing if not looking at entire system stack
  - Small task sizes (~ 100s of instructions)

Publications

- M. Amber Hassaan, Donald Nguyen, Keshav Pingali. Kinetic Dependence Graphs (to appear). ASPLOS 2015.
- Konstantions Karantasis, Andrew Lenharth, Donald Nguyen, et al. Parallelization of reordering algorithms for bandwidth and wavefront reduction. SC 2014.
- Damian Goik, Konrad Jopek, Maciej Paszyński, Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Graph grammar based multi-thread multi-frontal direct solver with galois scheduler. ICCS 2014.
- Donald Nguyen, Andrew Lenharth, Keshav Pingali. Deterministic Galois: On-demand, parameterless and portable. ASPLOS 2014.
- Donald Nguyen, Andrew Lenharth, Keshav Pingali. A lightweight infrastructure for graph analytics. SOSP 2013.
- Keshav Pingali, Donald Nguyen, Milind Kulkarni, et al. The tao of parallelism in algorithms. PLDI 2011.
- Donald Nguyen, Keshav Pingali. Synthesizing concurrent schedulers for irregular algorithms. ASPLOS 2011.
- Milind Kulkarni, Donald Nguyen, Dimitrios Prountzos, et al. Exploiting the commutativity lattice. PLDI 2011.
- Xin Sui, Donald Nguyen, Martin Burtscher, Keshav Pingali. Parallel Graph Partitioning on Multicore Architectures. LCPC 2010.
- Mario Méndez-Lojo, Donald Nguyen, Dimitrios Prountzos, et al. Structure-driven optimizations for amorphous data-parallel programs. PPOPP 2010.
- Shih-wei Liao, Tzu-Han Hung, Donald Nguyen, et al. Machine learning-based prefetch optimization for data center applications. SC 2009.

Publications

- M. Amber Hassaan, Donald Nguyen, Keshav Pingali. Kinetic Dependence Graphs (to appear). ASPLOS 2015.
- Konstantions Karantasis, Andrew Lenharth, Donald Nguyen, et al. Parallelization of reordering algorithms for bandwidth and wavefront reduction. SC 2014.
- Damian Goik, Konrad Jopek, Maciej Paszyński, Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Graph grammar based multi-thread multi-frontal direct solver with galois scheduler. ICCS 2014.
- Donald Nguyen, Andrew Lenharth, Keshav Pingali. Deterministic Galois: On-demand, parameterless and portable. ASPLOS 2014.
- Donald Nguyen, Andrew Lenharth, Keshav Pingali. A lightweight infrastructure for graph analytics. SOSP 2013.
- Keshav Pingali, Donald Nguyen, Milind Kulkarni, et al. The tao of parallelism in algorithms. PLDI 2011.
- Donald Nguyen, Keshav Pingali. Synthesizing concurrent schedulers for irregular algorithms. ASPLOS 2011.
- Milind Kulkarni, Donald Nguyen, Dimitrios Prountzos, et al. Exploiting the commutativity lattice. PLDI 2011.
- Xin Sui, Donald Nguyen, Martin Burtscher, Keshav Pingali. Parallel Graph Partitioning on Multicore Architectures. LCPC 2010.
- Mario Méndez-Lojo, Donald Nguyen, Dimitrios Prountzos, et al. Structure-driven optimizations for amorphous data-parallel programs. PPOPP 2010.
- Shih-wei Liao, Tzu-Han Hung, Donald Nguyen, et al. Machine learning-based prefetch optimization for data center applications. SC 2009.

Outline

- Parallel Program = Operator + Schedule + Parallel Data Structure
- Galois system implementation
  - Operator, data structures, scheduling
- Galois studies
  - Intel study
  - Deterministic scheduling
  - Adding a scheduler: sparse tiling
  - Comparison with transactional memory
  - Implementing other programming models

Galois Programming Model

- Galois system (Stephanie)
  - Runtime and library of concurrent data structures
- Galois system (Joe)
  - Set iterators express parallelism
  - Operator: body of iterator
  - ADT (e.g., graph, set)
- Unordered execution
  - Could have conflicts
  - Some serialization of iterations
  - New iterations can be added

```
Graph g
for_each (Edge e : wl)
  Node n = g.getEdgeDst(e)
  …
```

- Ordered execution
  - See Hassaan, Nguyen and Pingali. Kinetic Dependence Graphs (to appear). ASPLOS 2015.

Hello Graph

```
#include “Galois/Galois.h”
#include “Galois/Graph/Graph.h”

typedef Galois::Graph::LC_CSR_Graph<int, int> Graph;
typedef Graph::GraphNode GNode;

Graph graph;

struct P {
  void operator()(GNode& n, Galois::UserContext<GNode>& c) {
    int& d = graph.getData(n);
    if (d.value++ < 5)
      c.push(n);
  }
};

int main(int argc, char** argv) {
  Galois::Graph::readGraph(graph, argv[1]);
  …
  Galois::for_each(initial, P(), Galois::wl<WL>());
  return 0;
}
```

Hello Graph

```
#include “Galois/Galois.h”
#include “Galois/Graph/Graph.h”

typedef Galois::Graph::LC_CSR_Graph<int, int> Graph;
typedef Graph::GraphNode GNode;

Graph graph;

struct P {
  void operator()(GNode& n, Galois::UserContext<GNode>& c) {
    int& d = graph.getData(n);
    if (d.value++ < 5)
      c.push(n);
  }
};

int main(int argc, char** argv) {
  Galois::Graph::readGraph(graph, argv[1]);
  …
  Galois::for_each(initial, P(), Galois::wl<WL>());
  return 0;
}
```

Data structure declarations

Operator

Galois Iterator

Hello Graph

```
#include “Galois/Galois.h”
#include “Galois/Graph/Graph.h”

typedef Galois::Graph::LC_CSR_Graph<int, int> Graph;
typedef Graph::GraphNode GNode;

Graph graph;

struct P {
  void operator()(GNode& n, Galois::UserContext<GNode>& c) {
    int& d = graph.getData(n);
    if (d.value++ < 5)
      c.push(n);
  }
};

int main(int argc, char** argv) {
  Galois::Graph::readGraph(graph, argv[1]);
  …
  Galois::for_each(initial, P(), Galois::wl<WL>());
  return 0;
}
```

Data structure declarations

Operator

Galois Iterator

Sequential program semantics

Hello Graph

```
#include “Galois/Galois.h”
#include “Galois/Graph/Graph.h”

typedef Galois::Graph::LC_CSR_Graph<int, int> Graph;
typedef Graph::GraphNode GNode;

Graph graph;

struct P {
  void operator()(GNode& n, Galois::UserContext<GNode>& c) {
    int& d = graph.getData(n);
    if (d.value++ < 5)
  }
};

int main(int argc, char** argv) {
  Galois::Graph::readGraph(graph, argv[1]);
  …
  Galois::for_each(initial, P(), Galois::wl<WL>());
  return 0;
}
```

Data structure declarations

Operator

Neighborhood defined by graph API calls

Galois Iterator

Sequential program semantics

Hello Graph

```
#include “Galois/Galois.h”
#include “Galois/Graph/Graph.h”

typedef Galois::Graph::LC_CSR_Graph<int, int> Graph;
typedef Graph::GraphNode GNode;

Graph graph;

struct P {
  void operator()(GNode& n, Gc
    int& d = graph.getData(n);
    if (d.value++ < 5)
  }
};

int main(int argc, char** argv) {
  Galois::Graph::readGraph(graph, argv[1]);
  …
  Galois::for_each(initial, P(), Galois::wl<WL>());
  return 0;
}
```

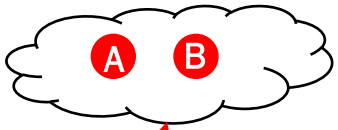
Neighborhood defined by graph API calls

```
struct TaskIndexer {
  int operator()(GNode& n) {
    return graph.getData(n) >> shift;
  }
};

using namespace Galois::WorkList;

typedef dChunkedFIFO<> WL;
typedef OrderedByIntegerMetric<TaskIndexer> WL;
```

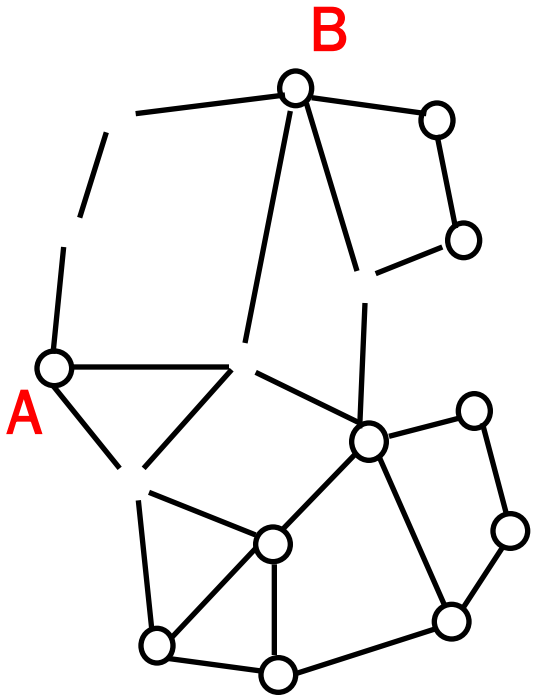
Scheduling DSL

Sequential program semantics

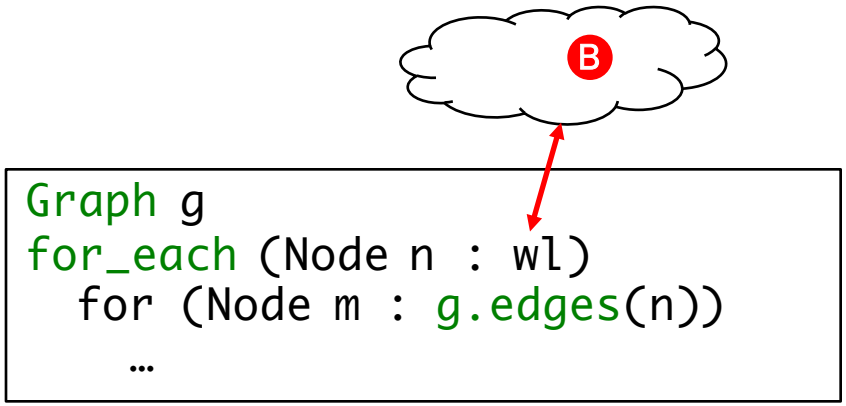
Galois Operator

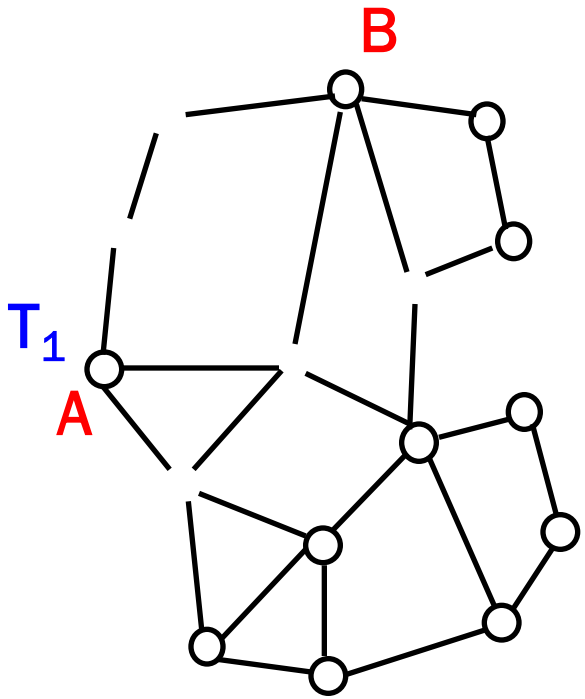



```
Graph g
for_each (Node n : wl)
  for (Node m : g.edges(n))
    …
```

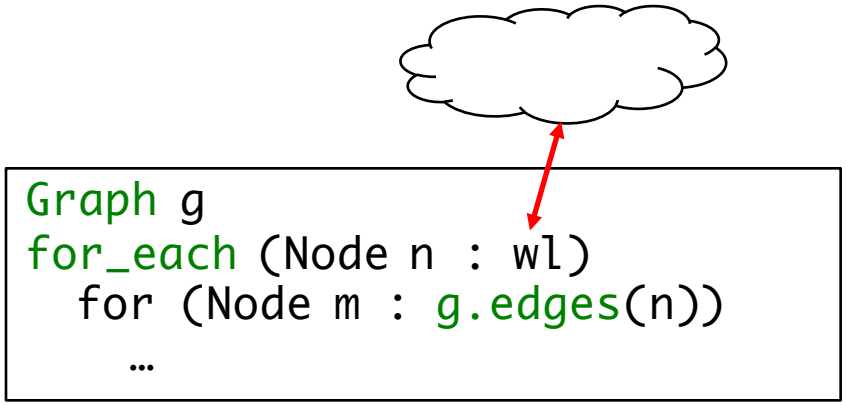


Galois Operator

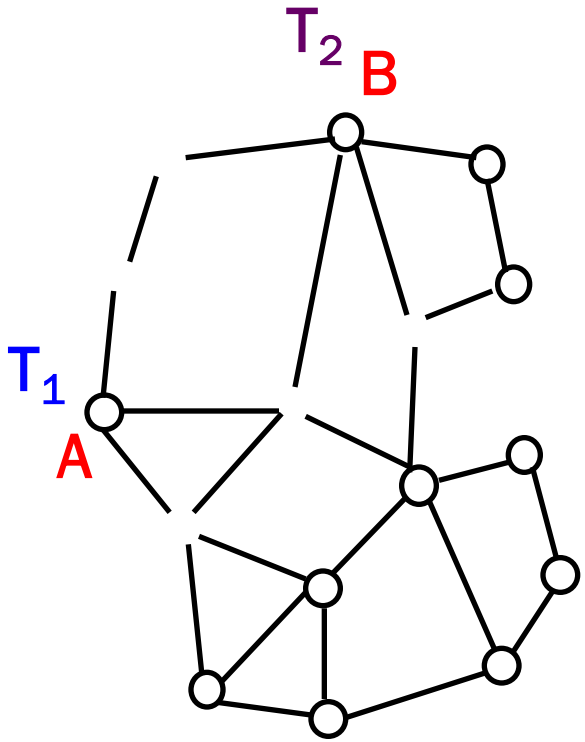




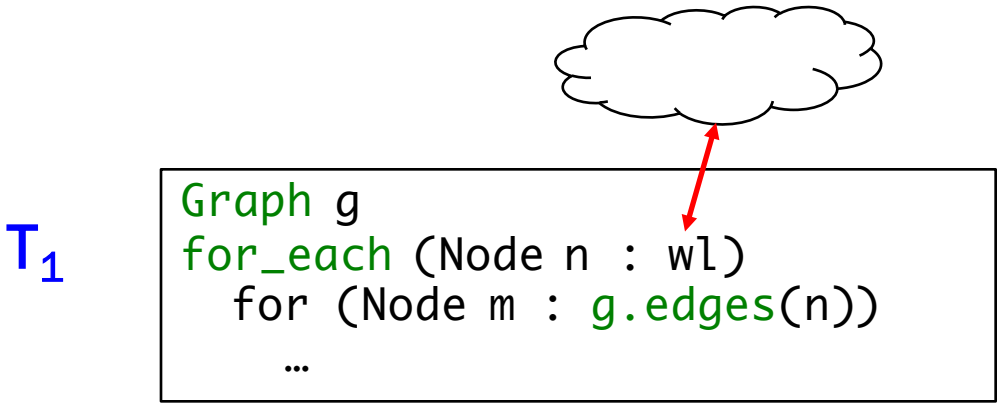
Galois Operator

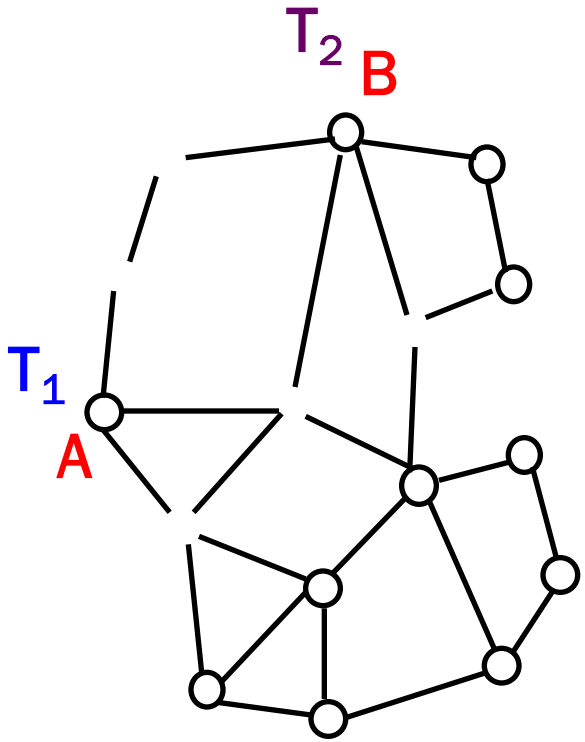


```
Graph g
for_each (Node n : wl)
  for (Node m : g.edges(n))
    …
```

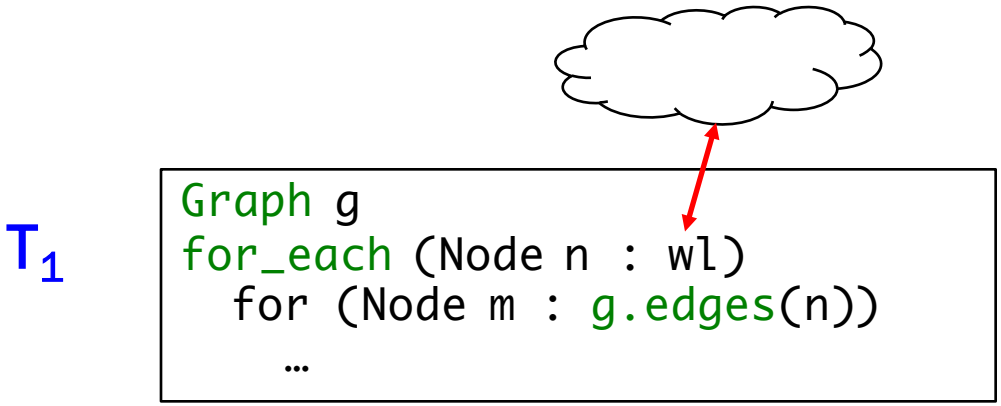


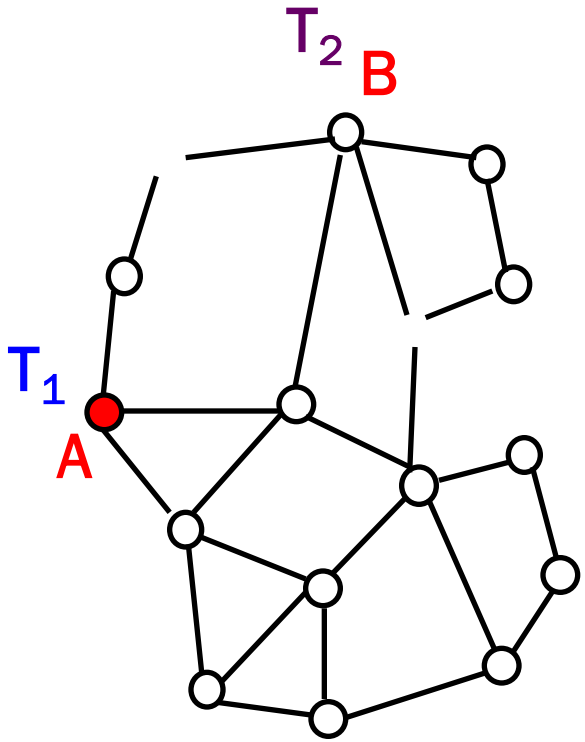
Galois Operator



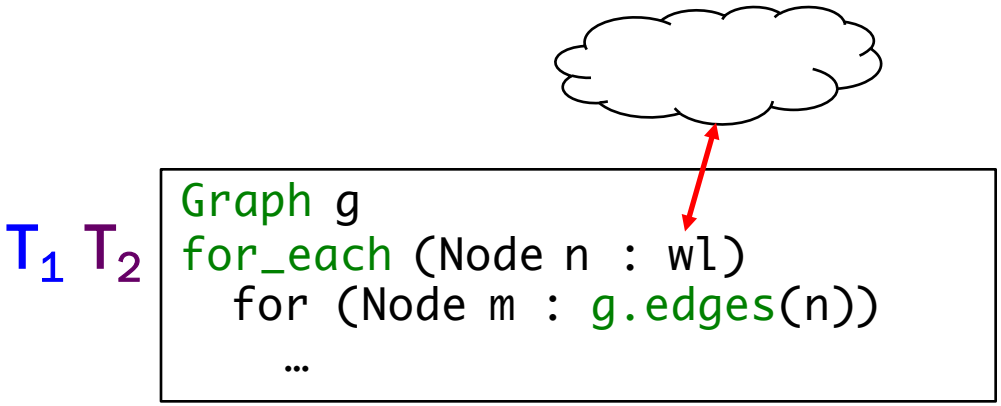


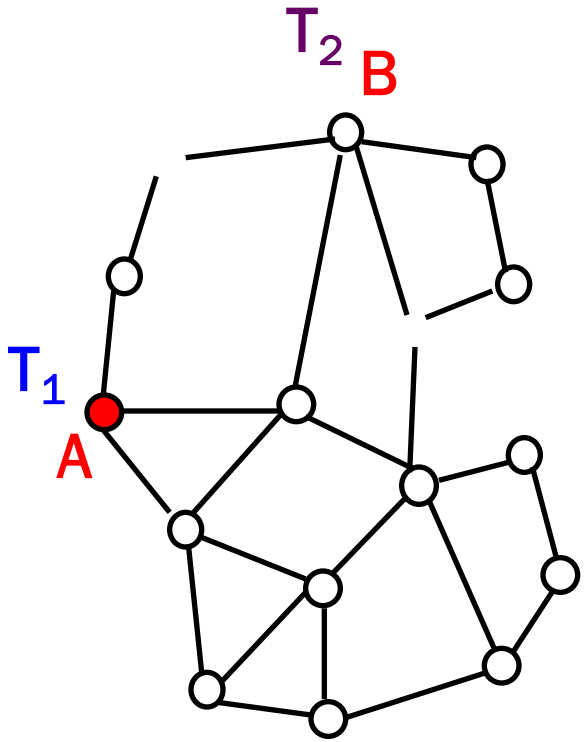
Galois Operator



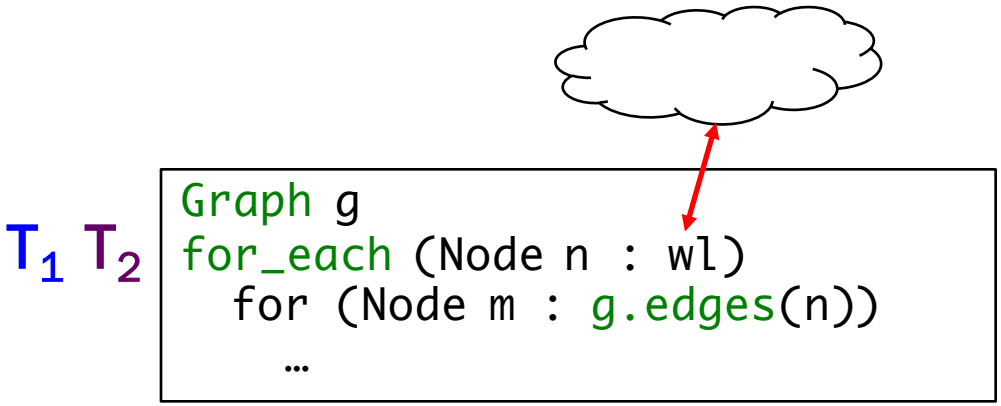


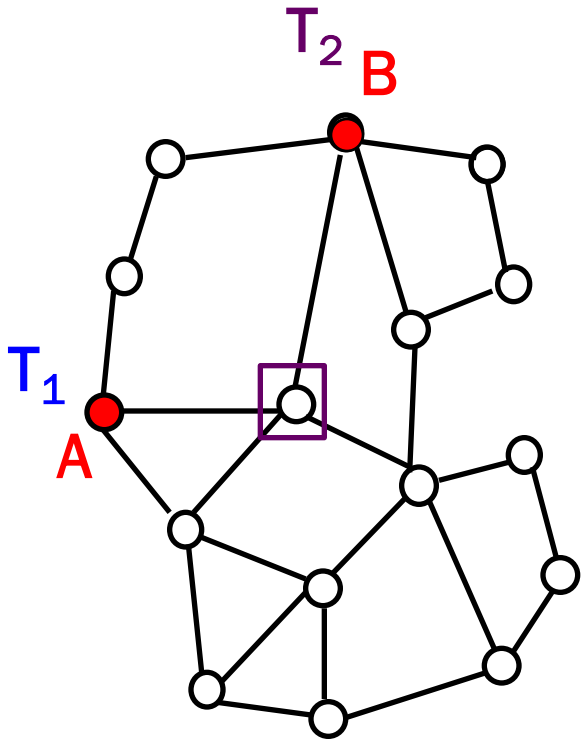
Galois Operator



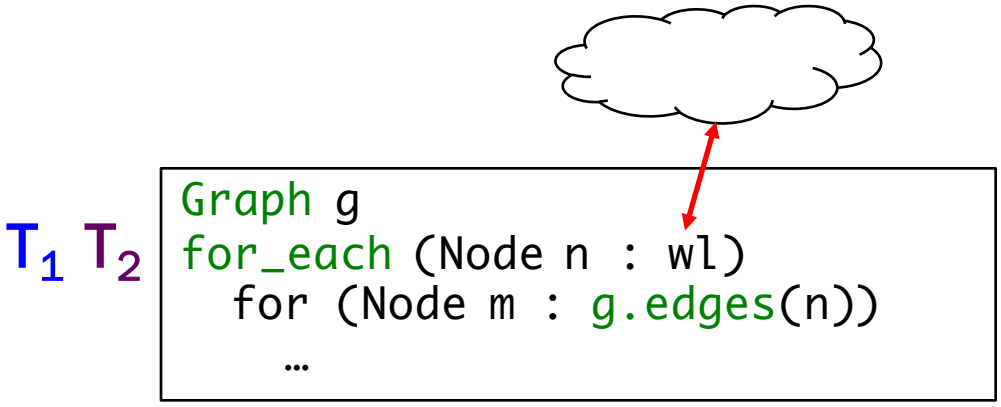


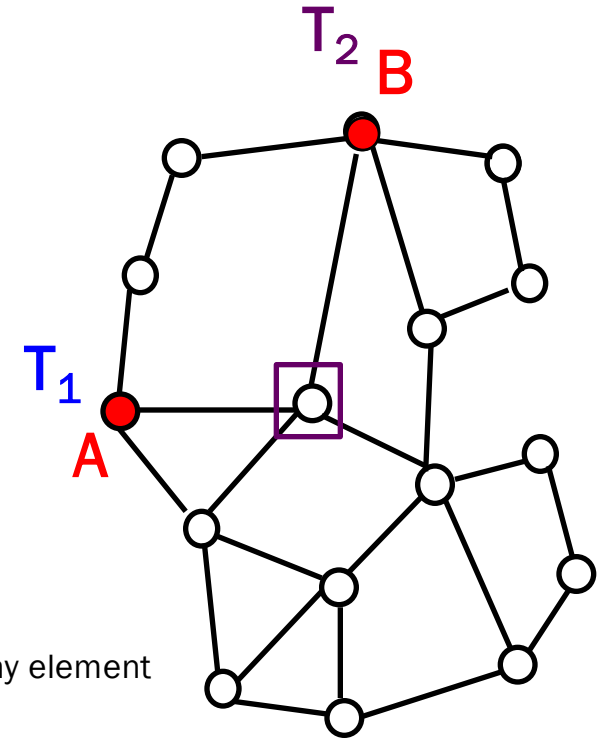
Galois Operator





Galois Operator





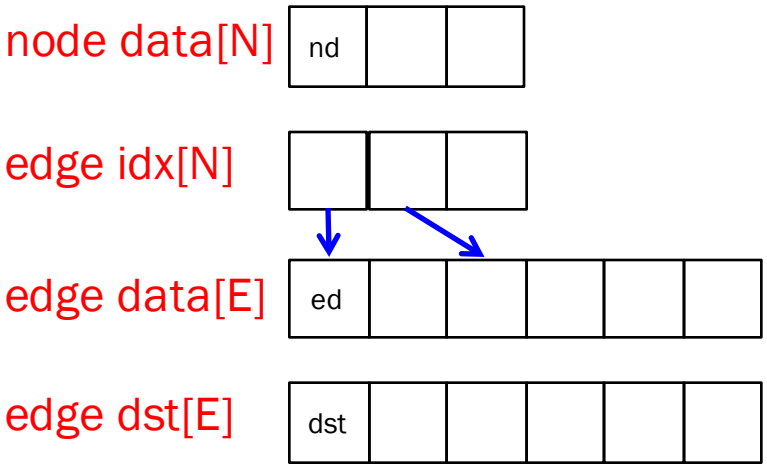
- All data accesses go through API
  - Galois data implementations maintain marks
  - When marks overlap, rollback activity and try later
- Cautious
  - Commonly operators read their entire neighborhood before modifying any element
  - Failsafe point: point in operator before first write
- Non-deterministic
- Difference from Thread Level Speculation [Rauchwerger and Padua, PLDI 1995]
  - Iterations are unordered
  - Operators are cautious
  - Conflicts are detected at ADT level rather than memory level

Galois Data Structures

- Graphs
  - May require multiple loads to access data for a node
  - Exploit local computation structure
  - Simple allocation schemes may be imbalanced in physical memory

Galois Data Structures

- Graphs
  - May require multiple loads to access data for a node
  - Exploit local computation structure
  - Simple allocation schemes may be imbalanced in physical memory



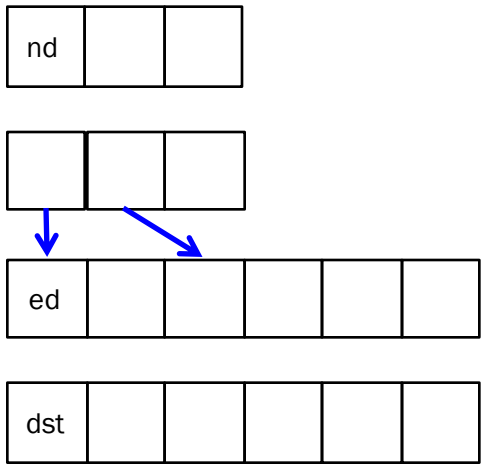
CSR

nd: node data
ed: edge data
dst: edge destination
len: # of edges

Galois Data Structures

- Graphs
  - May require multiple loads to access data for a node
  - Exploit local computation structure
  - Simple allocation schemes may be imbalanced in physical memory

Inlining



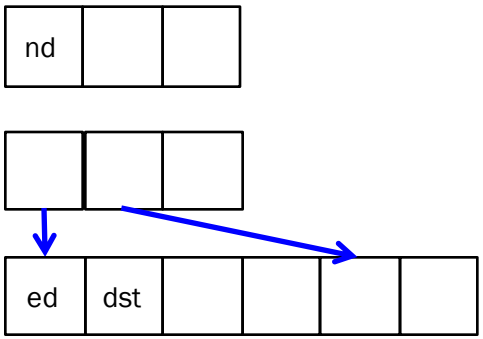
nd: node data
ed: edge data
dst: edge destination
len: # of edges

Inlining: Andrew Lenharth

Galois Data Structures

- Graphs
  - May require multiple loads to access data for a node
  - Exploit local computation structure
  - Simple allocation schemes may be imbalanced in physical memory

Inlining

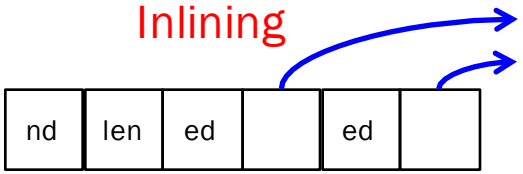


nd: node data
ed: edge data
dst: edge destination
len: # of edges

Inlining: Andrew Lenharth

Galois Data Structures

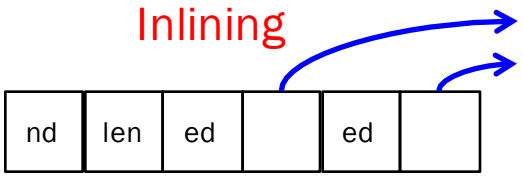
- Graphs
  - May require multiple loads to access data for a node
  - Exploit local computation structure
  - Simple allocation schemes may be imbalanced in physical memory



nd: node data
ed: edge data
dst: edge destination
len: # of edges

Inlining: Andrew Lenharth

Galois Data Structures

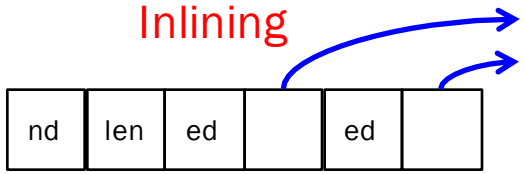


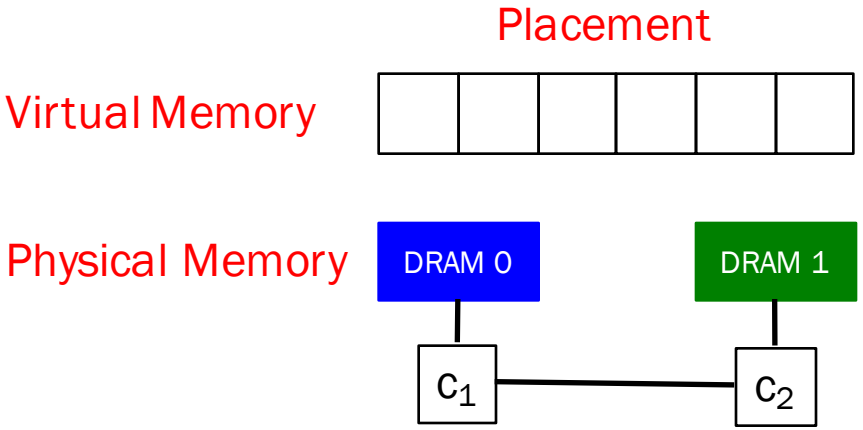
- Graphs
  - May require multiple loads to access data for a node
  - Exploit local computation structure
  - Simple allocation schemes may be imbalanced in physical memory

Inlining: Andrew Lenharth

Galois Data Structures

- Graphs
  - May require multiple loads to access data for a node
  - Exploit local computation structure
  - Simple allocation schemes may be imbalanced in physical memory





Inlining: Andrew Lenharth

Scheduling in Galois

- Users write high-level scheduling policies
  - DSL for specifying policies
- Separation of concerns
  - Users understand their operator but have vague understanding of scheduling
  - Schedulers are difficult to implement efficiently
- Galois system synthesizes scheduler by composing elements from a scheduler library

Nguyen and Pingali. Synthesizing concurrent schedulers for irregular algorithms. ASPLOS 2011.

Scheduling Components and Policies

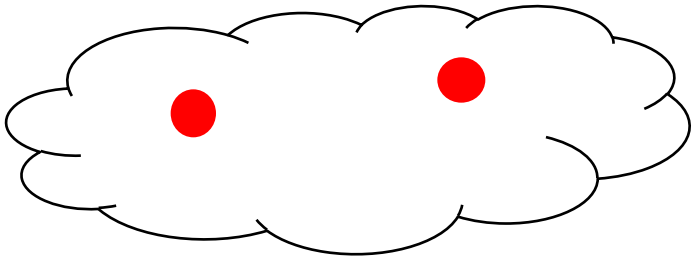
- Components: FIFO, LIFO, BulkSynchronous, ChunkedFIFO, OrderedByMetric, …

Algorithm	Scheduling Policy	Used By
DMR	OrderedByMetric(λ*t*.minangle(*t*)) FIFO	Shewchuk96
	Global: ChunkedFIFO(*k*) Local: LIFO	Kulkarni08
DT	OrderedByMetric(λ*p*.round(*p*)) FIFO	Amenta03
	Random	Clarkson89
PFP	FIFO	Goldberg88
	OrderedByMetric(λ*n*.height(*n*))	Cherkassy95
PTA	FIFO	Pearce03
	BulkSynchronous	Nielson99
SSSP	FIFO	Bellman-Ford
	OrderedByMetric(λ*n*.*w*(*n*)/Δ + (light(*n*) ? 0 : ½)) FIFO	Δ-stepping

Scheduling Bag

Abstraction

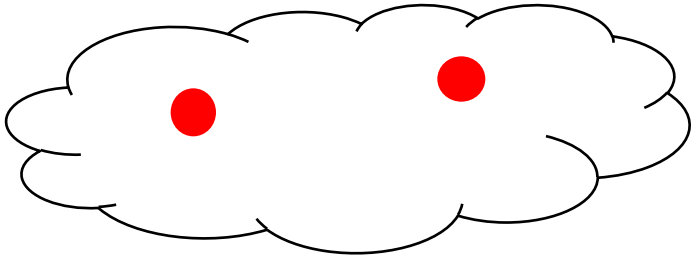
Bag



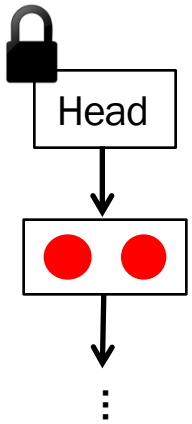
Scheduling Bag

Abstraction

Bag



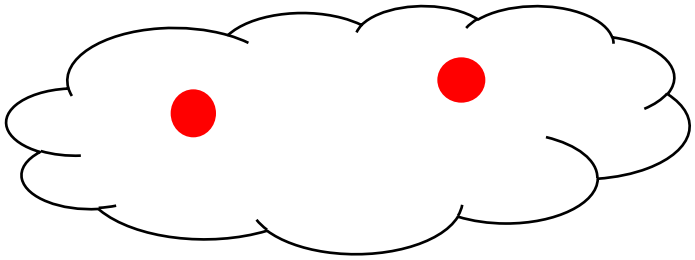
Implementation



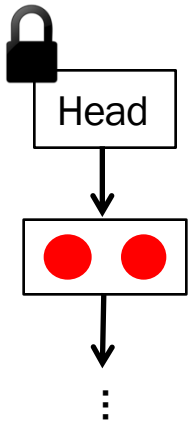
Scheduling Bag

Abstraction

Bag



Implementation

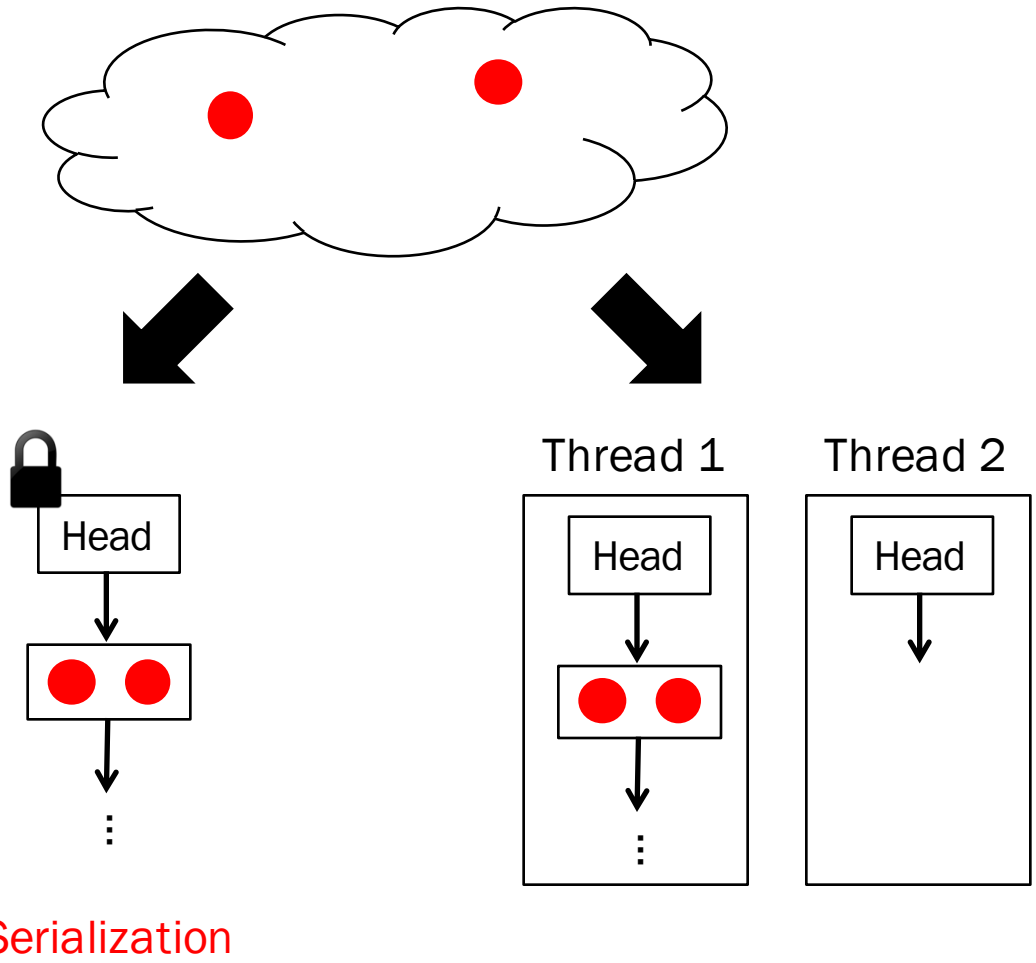


Serialization

Scheduling Bag

Abstraction

Implementation

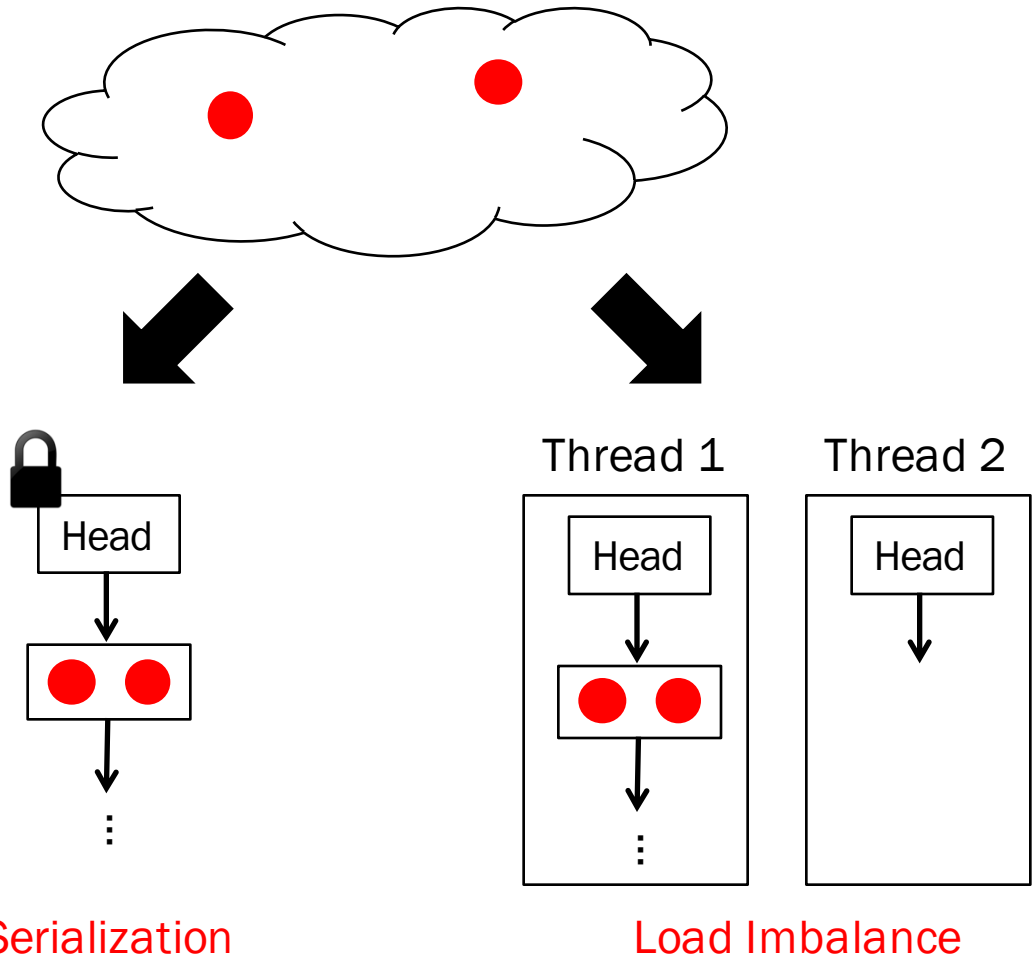


Serialization

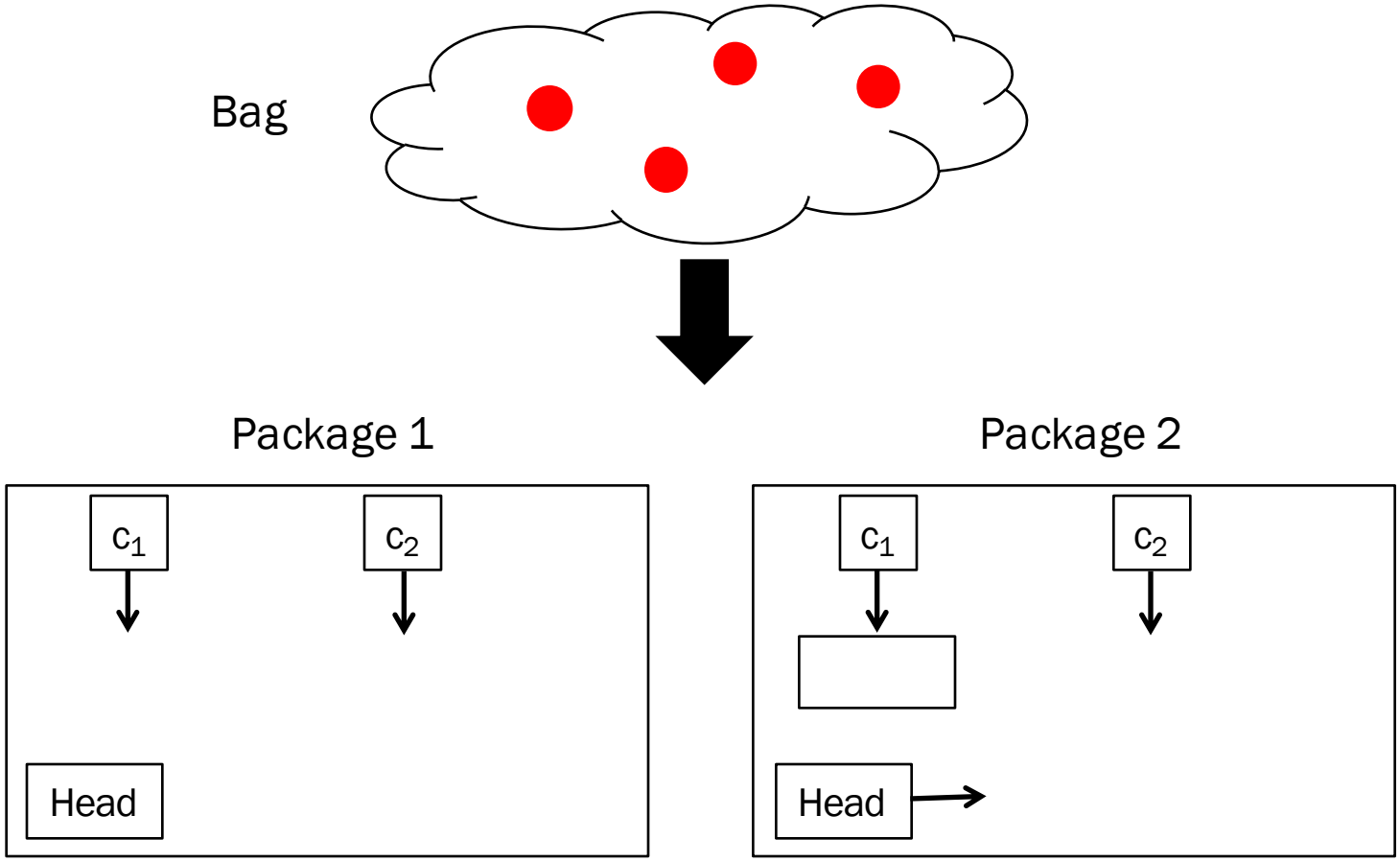
Scheduling Bag

Abstraction

Implementation

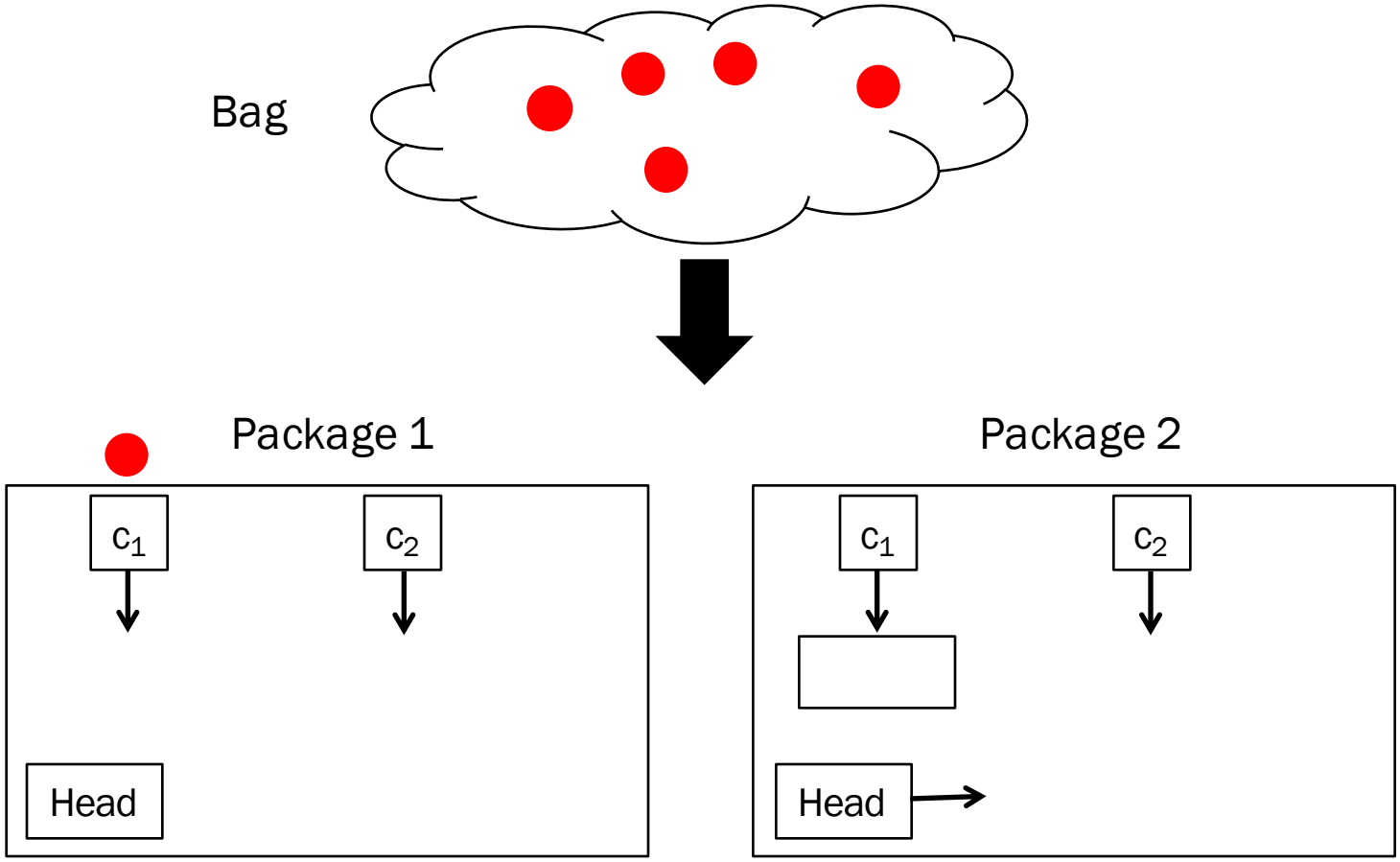


Scheduling Bag



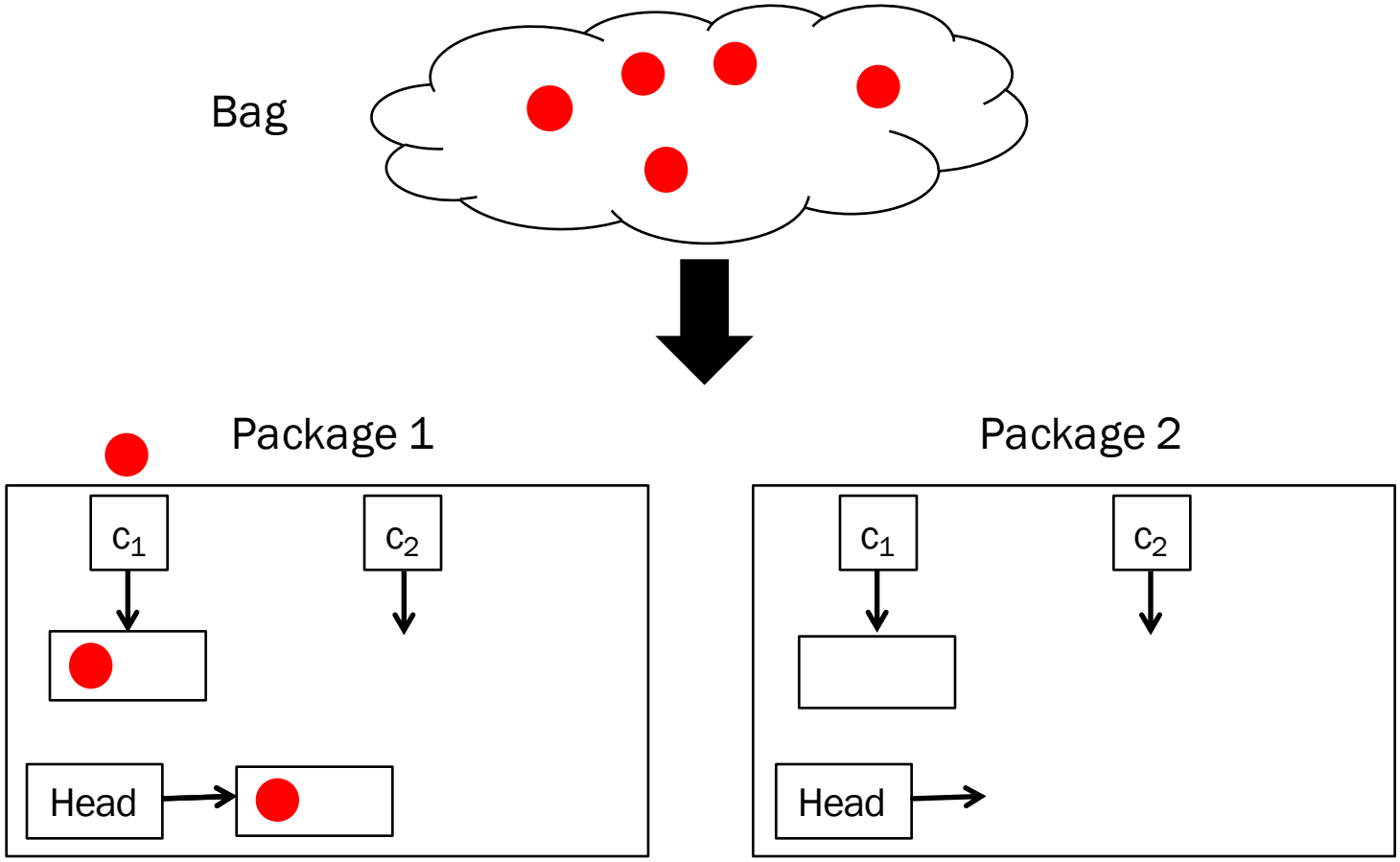
Per-Package Bag: Andrew Lenharth

Scheduling Bag



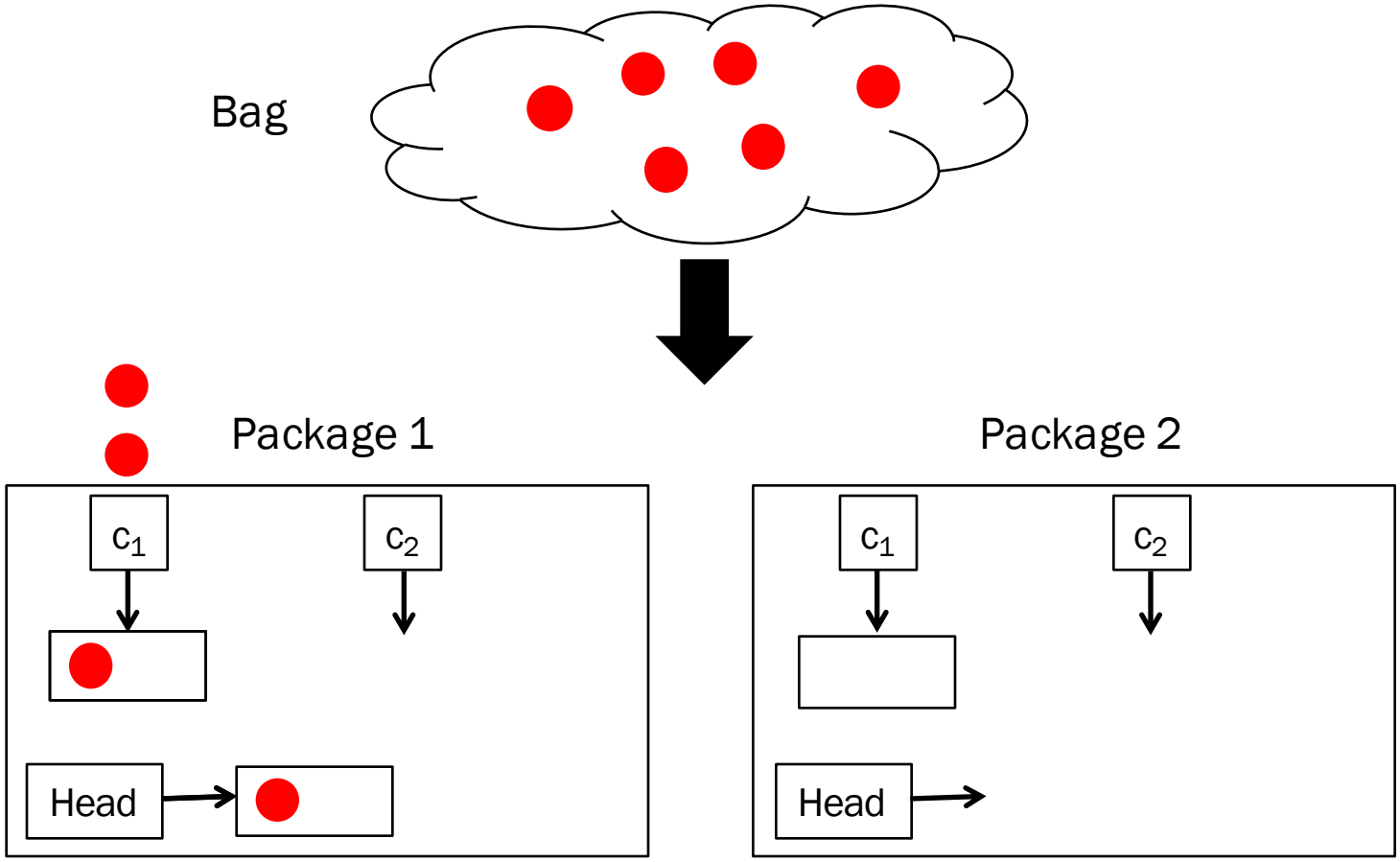
Per-Package Bag: Andrew Lenharth

Scheduling Bag



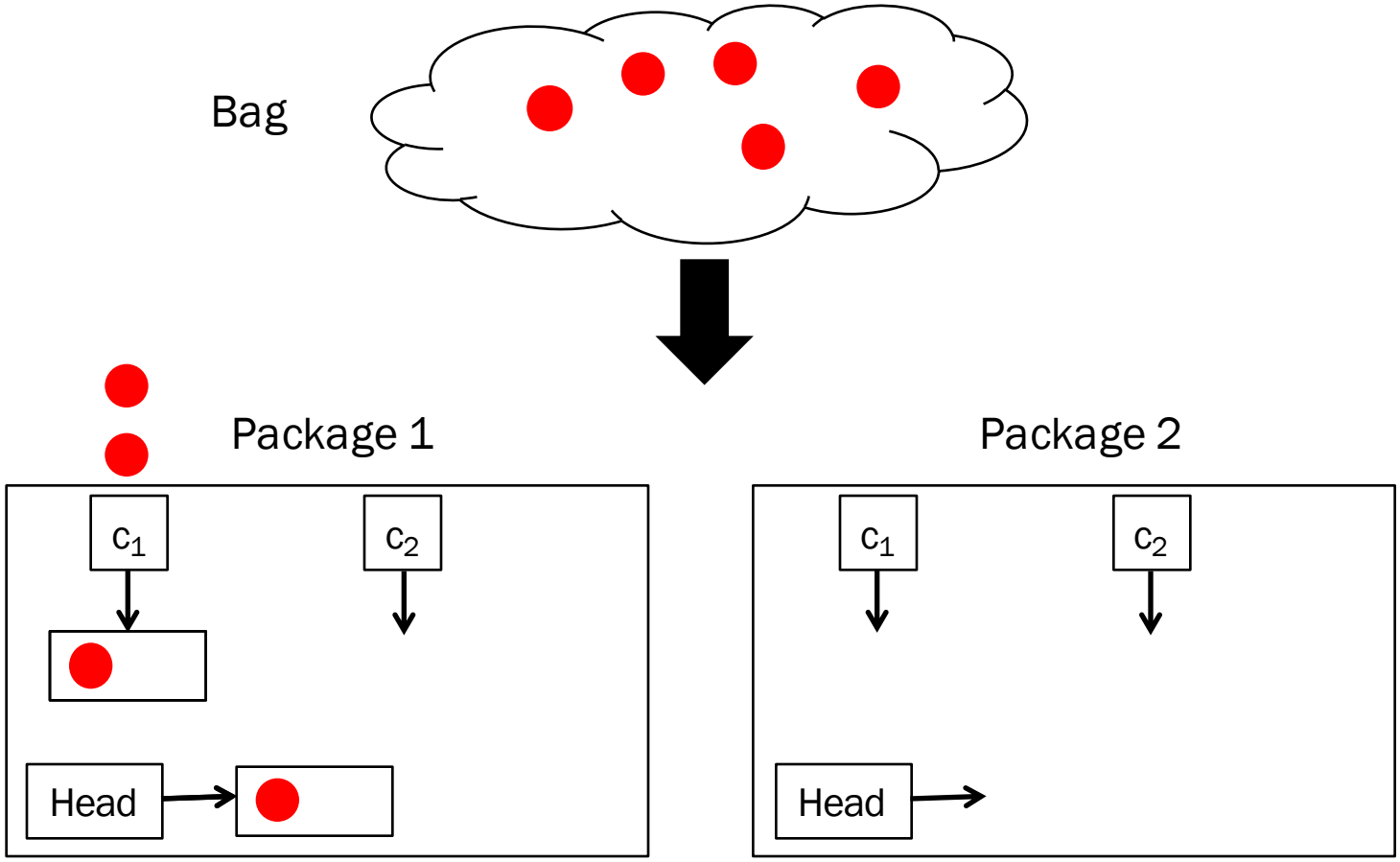
Per-Package Bag: Andrew Lenharth

Scheduling Bag



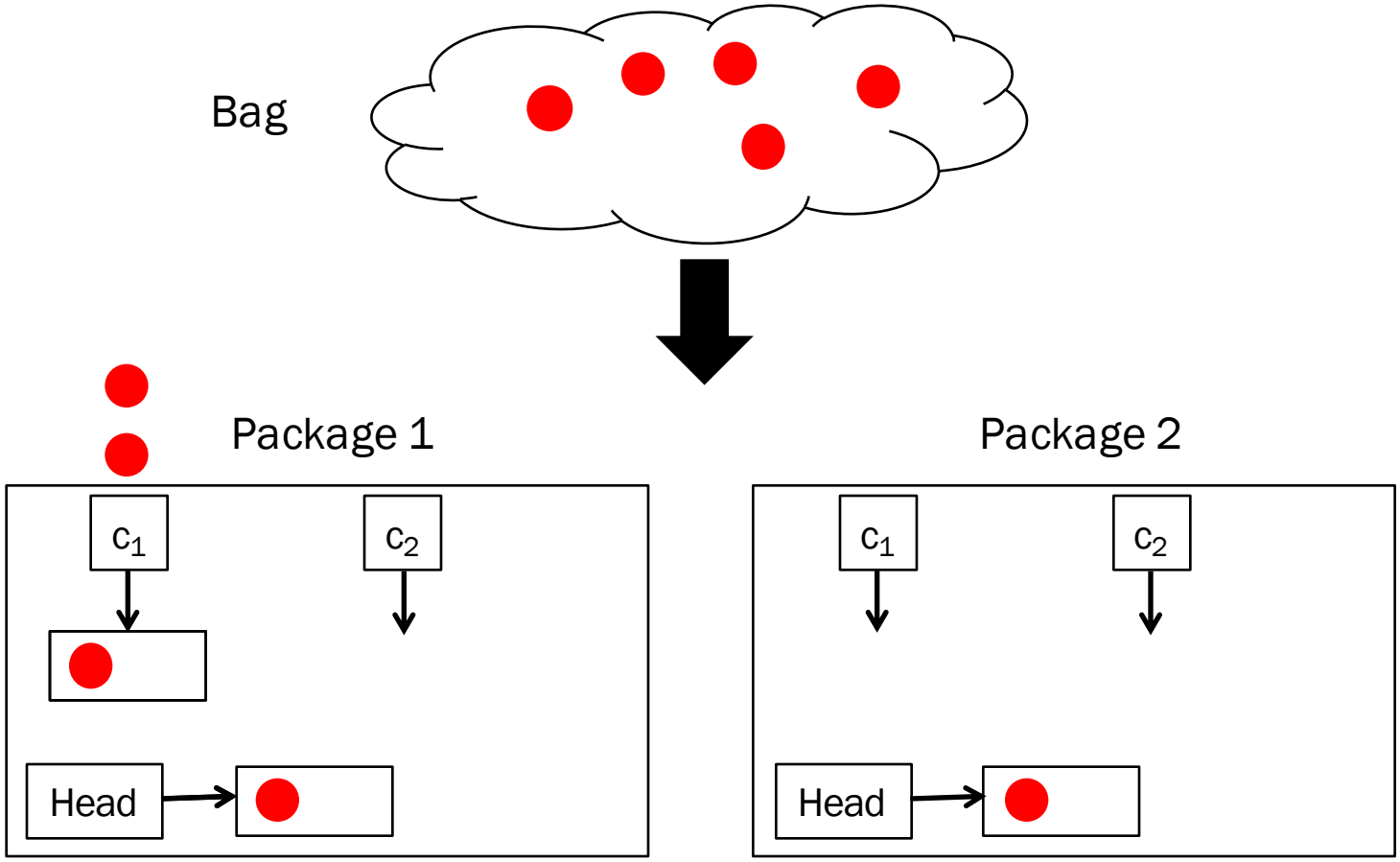
Per-Package Bag: Andrew Lenharth

Scheduling Bag



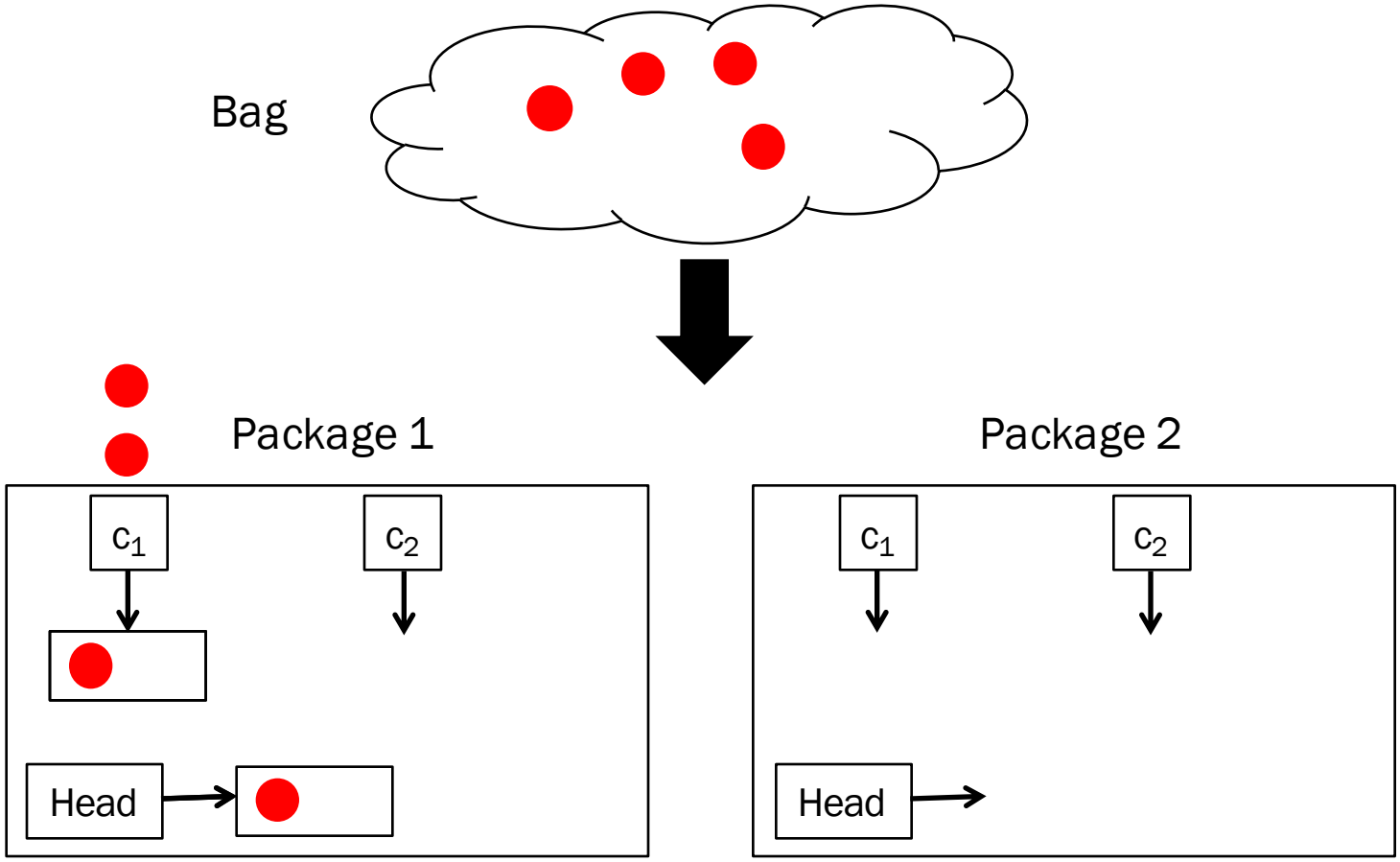
Per-Package Bag: Andrew Lenharth

Scheduling Bag



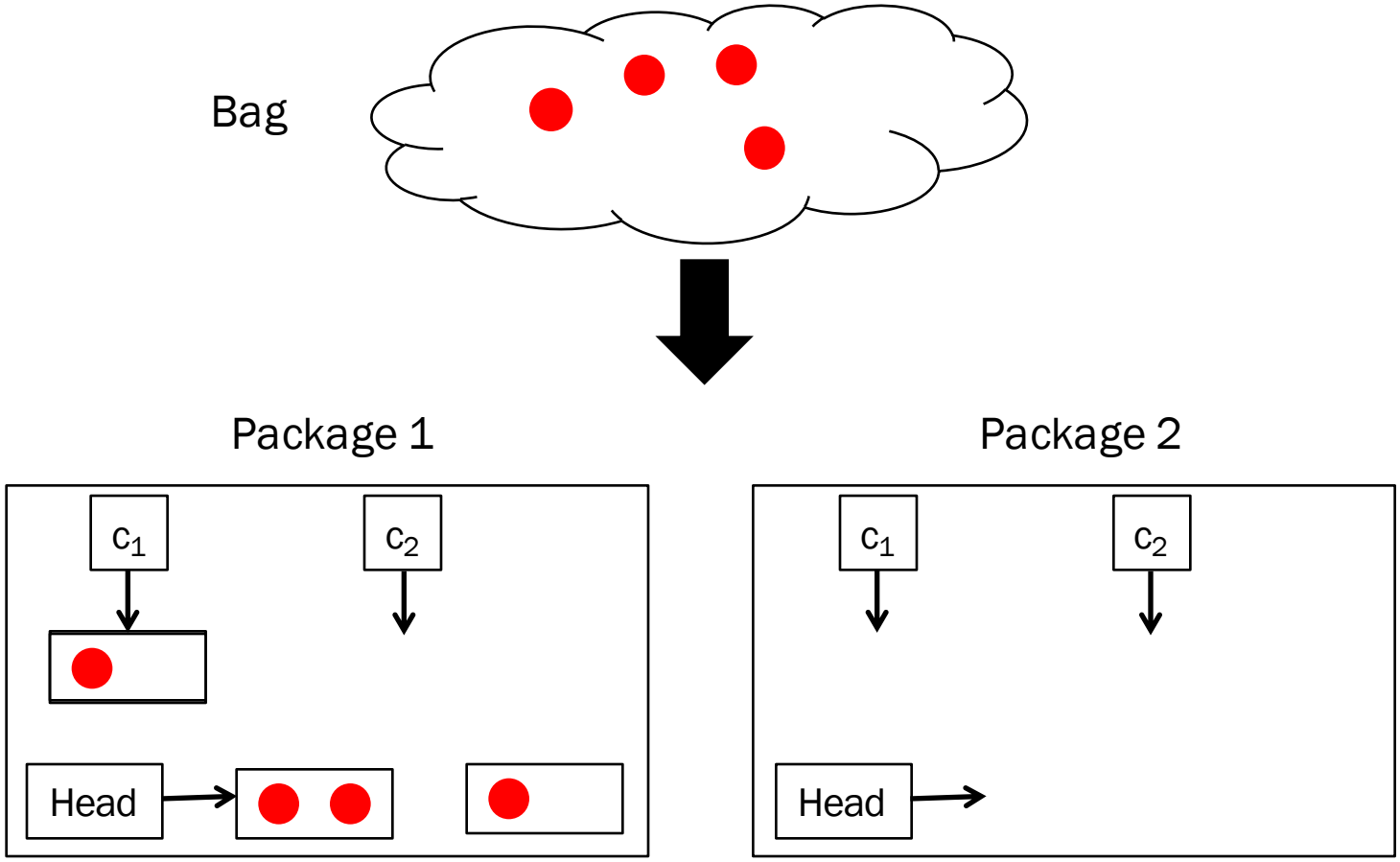
Per-Package Bag: Andrew Lenharth

Scheduling Bag

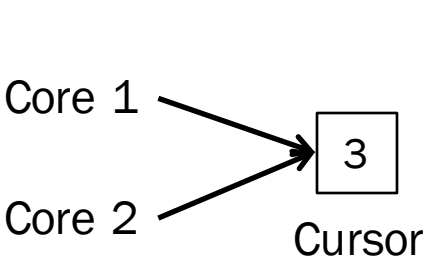


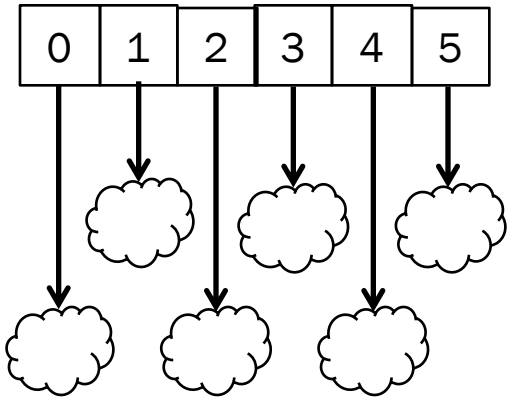
Per-Package Bag: Andrew Lenharth

Scheduling Bag

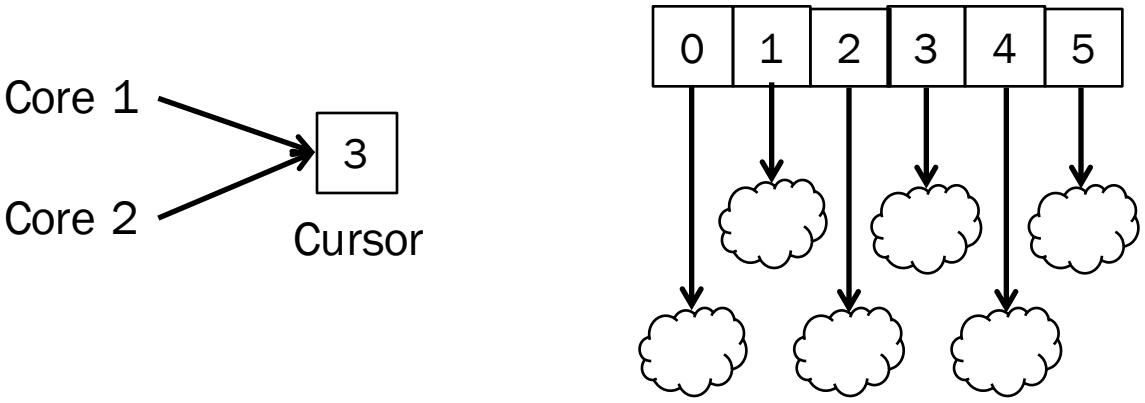


OrderedByIntegerMetric



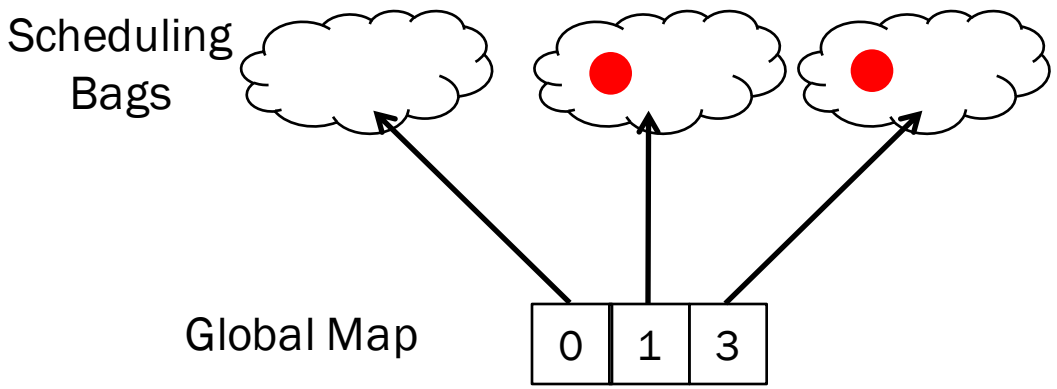


OrderedByIntegerMetric



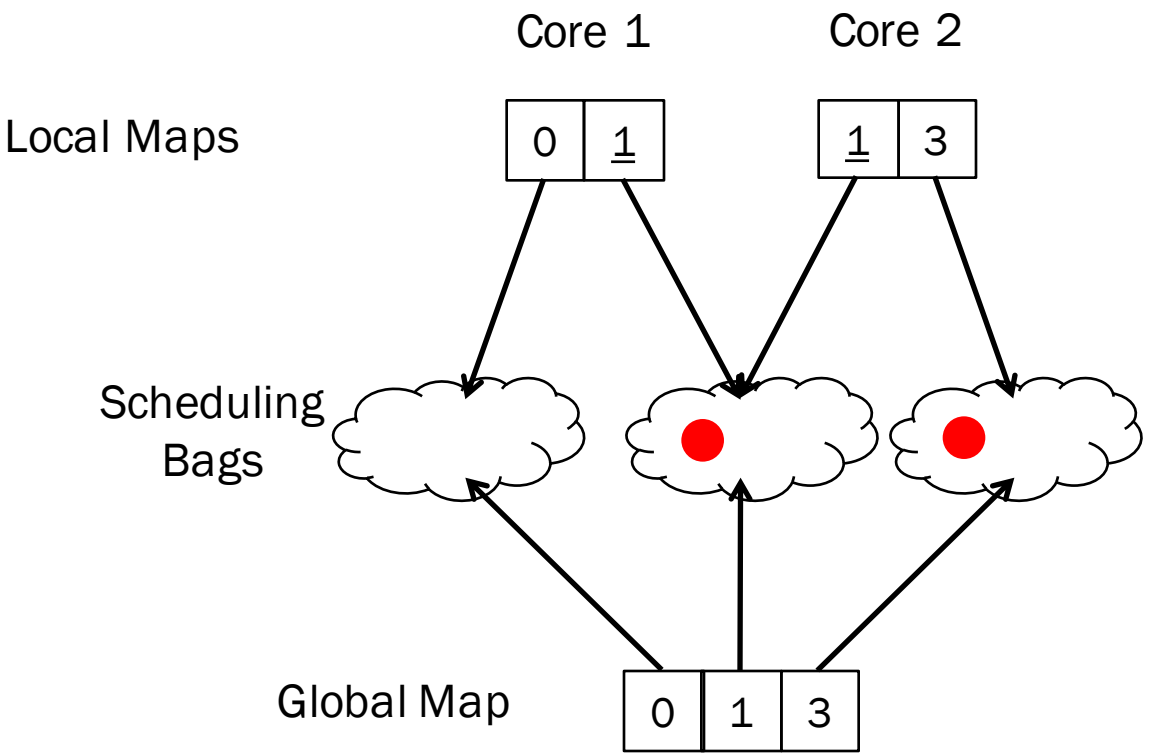
- Implements soft priorities
- Drawbacks
  - Dense range of priorities
  - Contention on cursor location

Improved OrderedByIntegerMetric



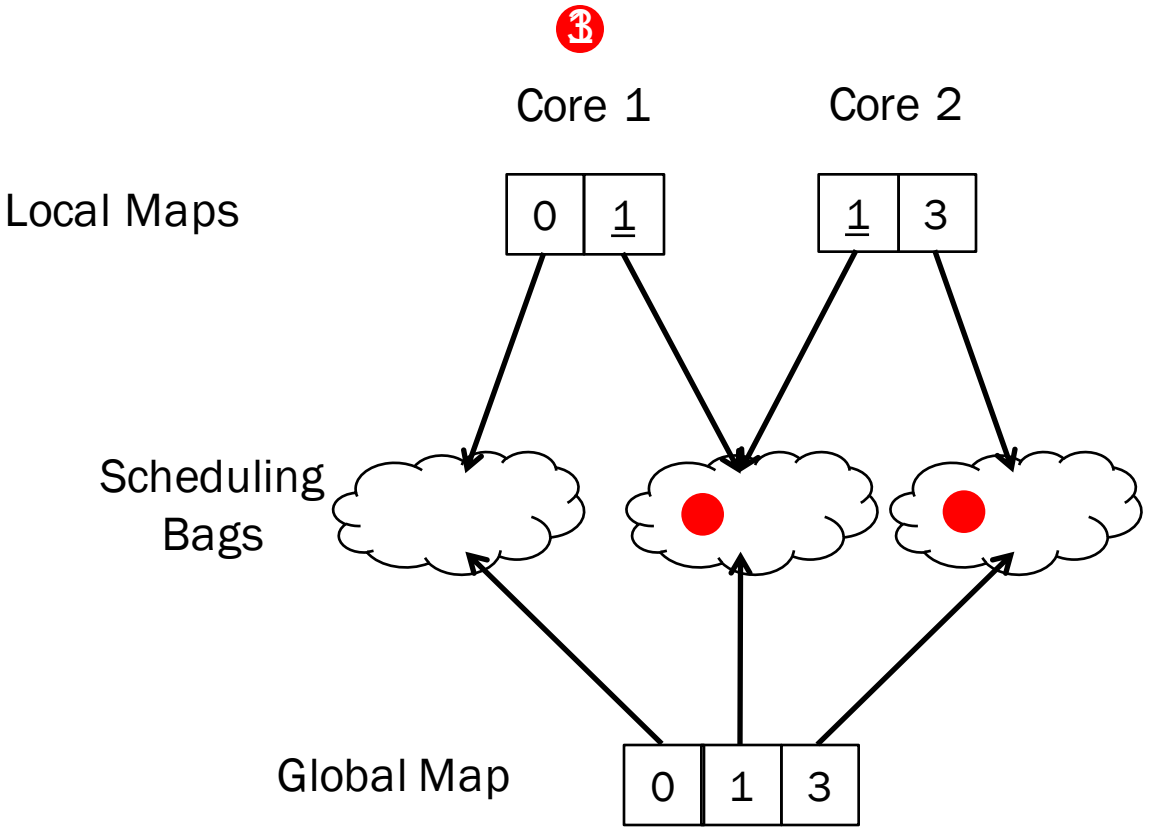
Implementation by Andrew Lenharth

Improved OrderedByIntegerMetric



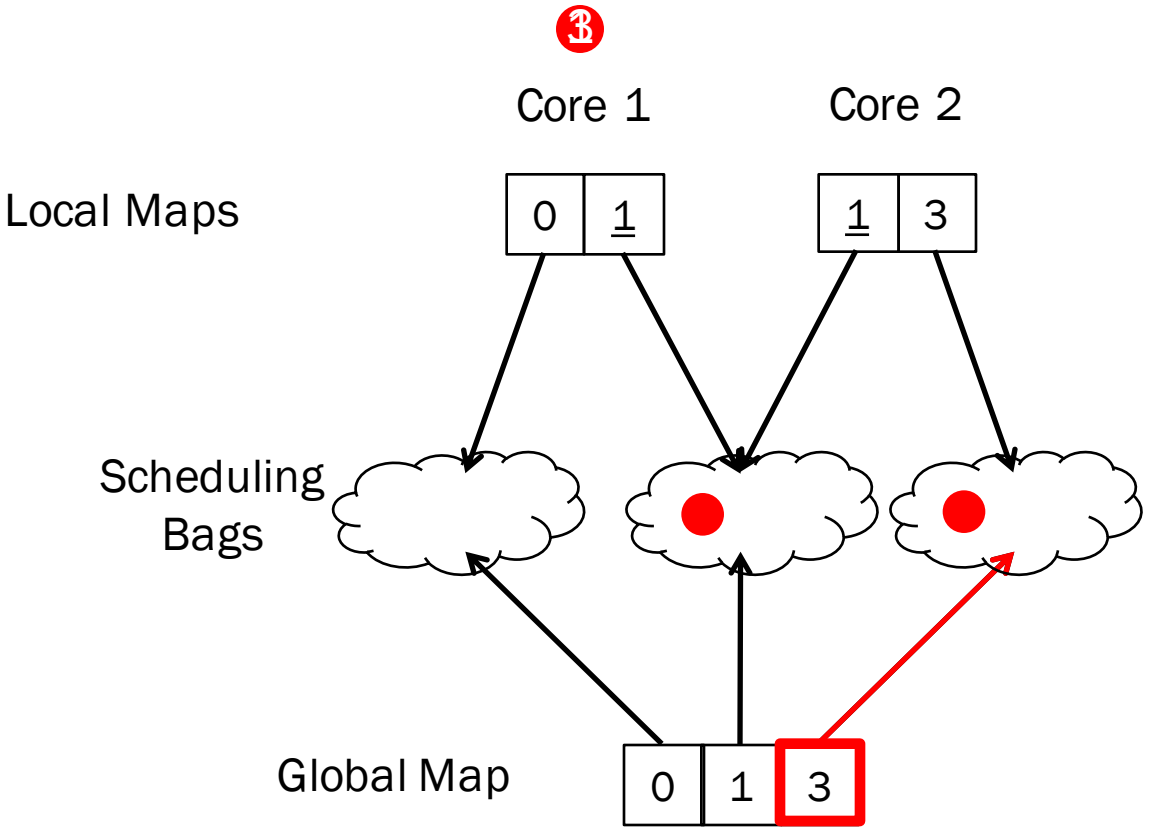
Implementation by Andrew Lenharth

Improved OrderedByIntegerMetric



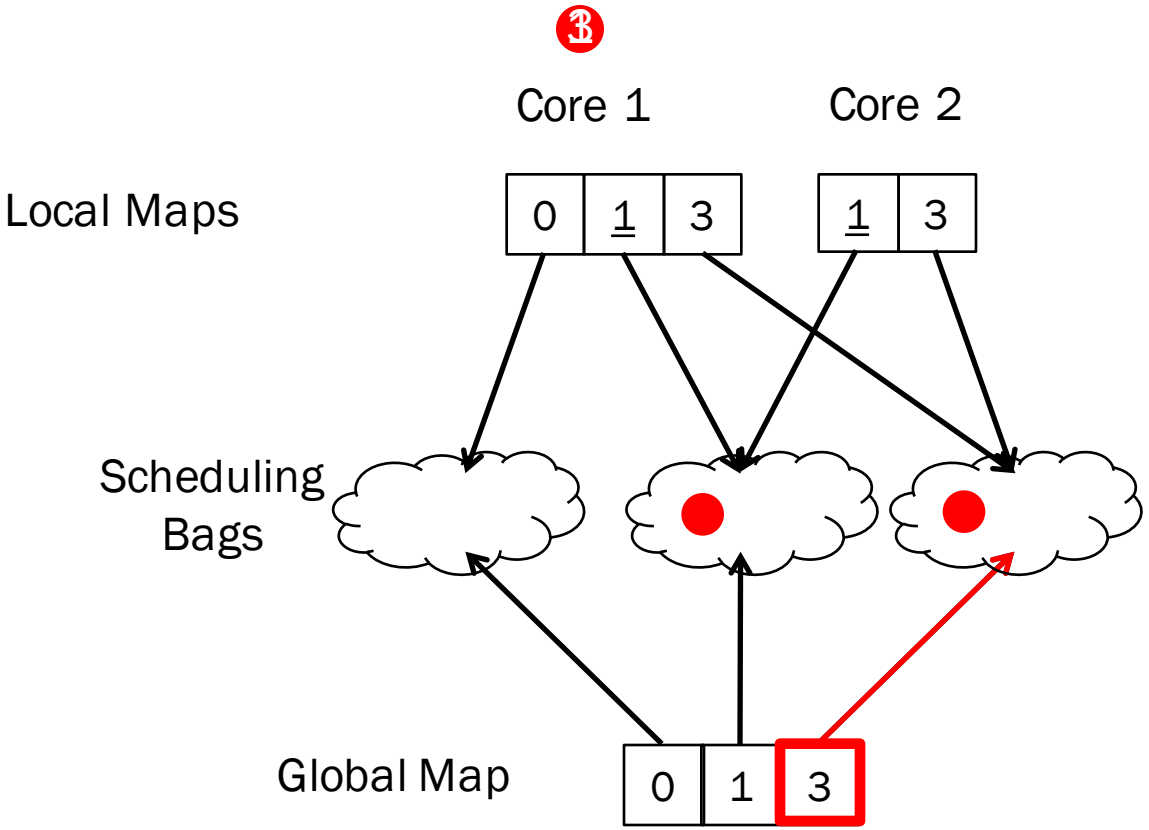
Implementation by Andrew Lenharth

Improved OrderedByIntegerMetric



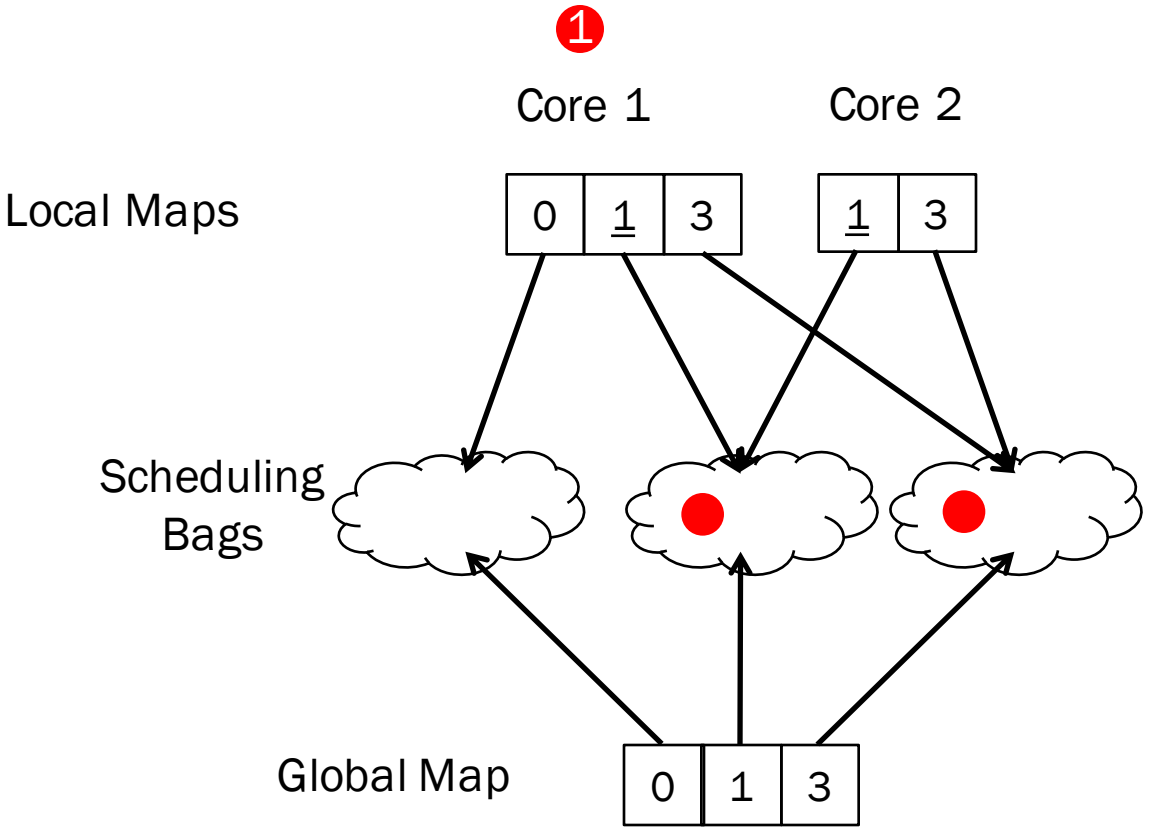
Implementation by Andrew Lenharth

Improved OrderedByIntegerMetric



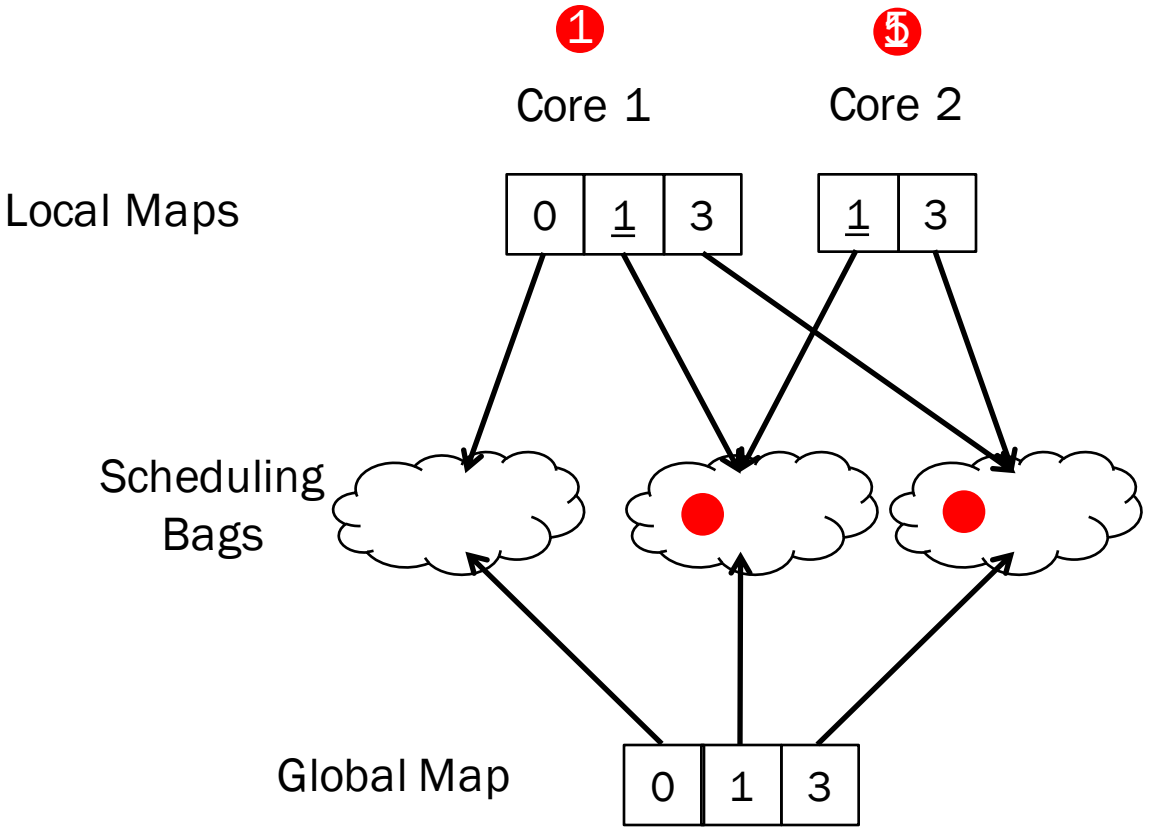
Implementation by Andrew Lenharth

Improved OrderedByIntegerMetric



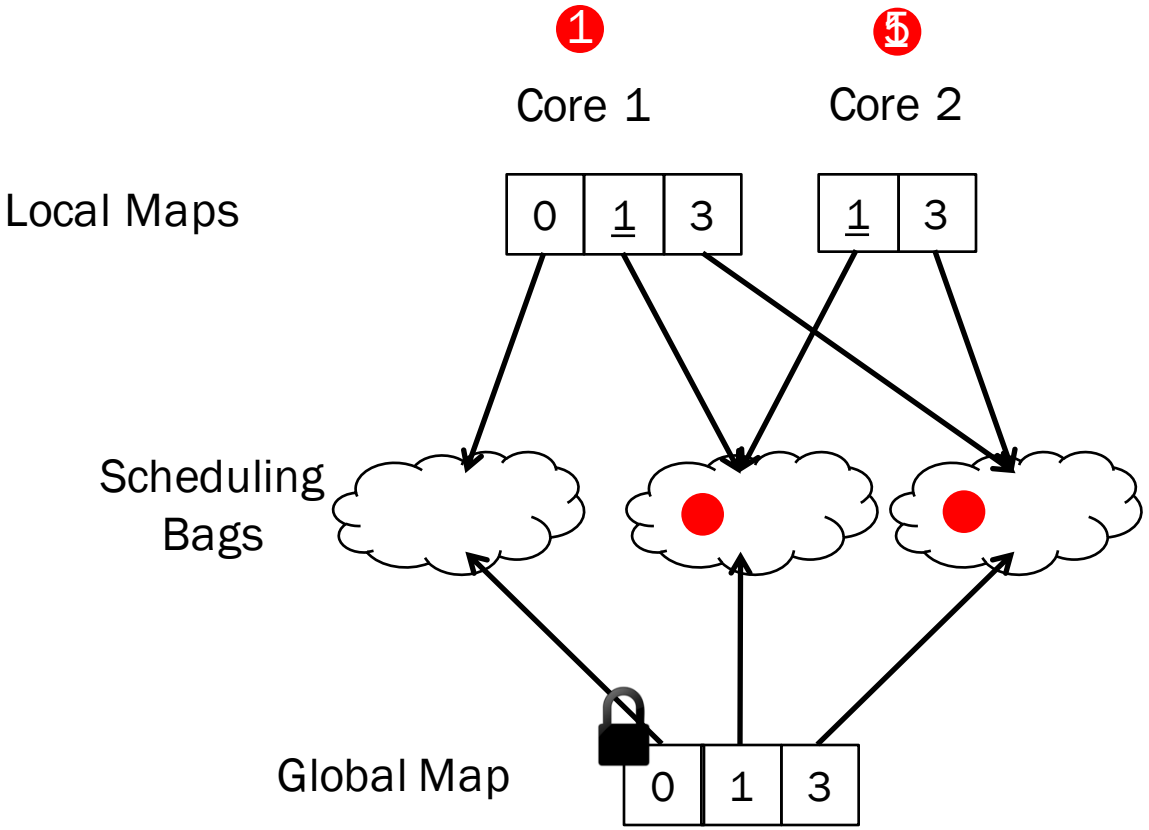
Implementation by Andrew Lenharth

Improved OrderedByIntegerMetric



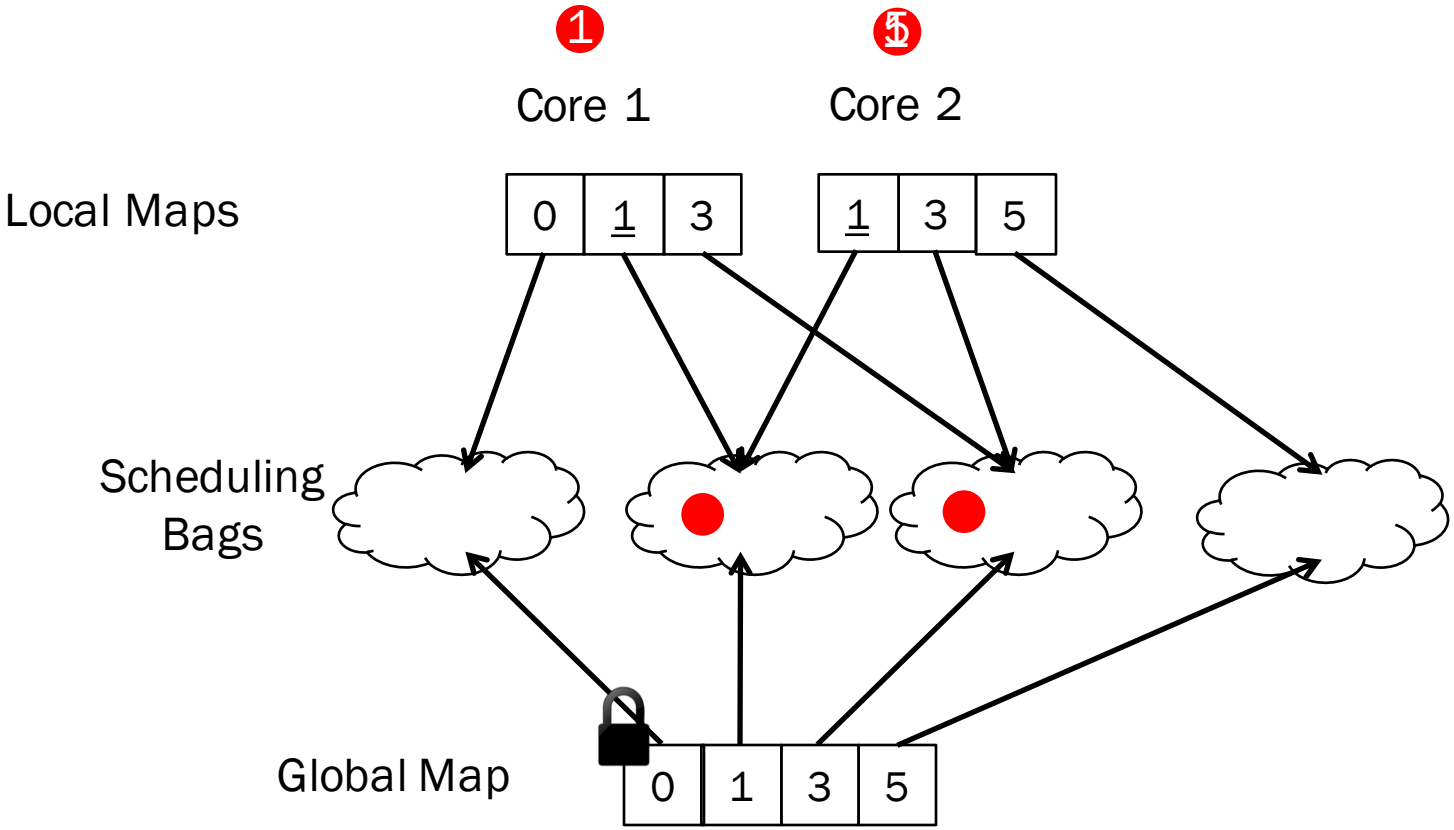
Implementation by Andrew Lenharth

Improved OrderedByIntegerMetric



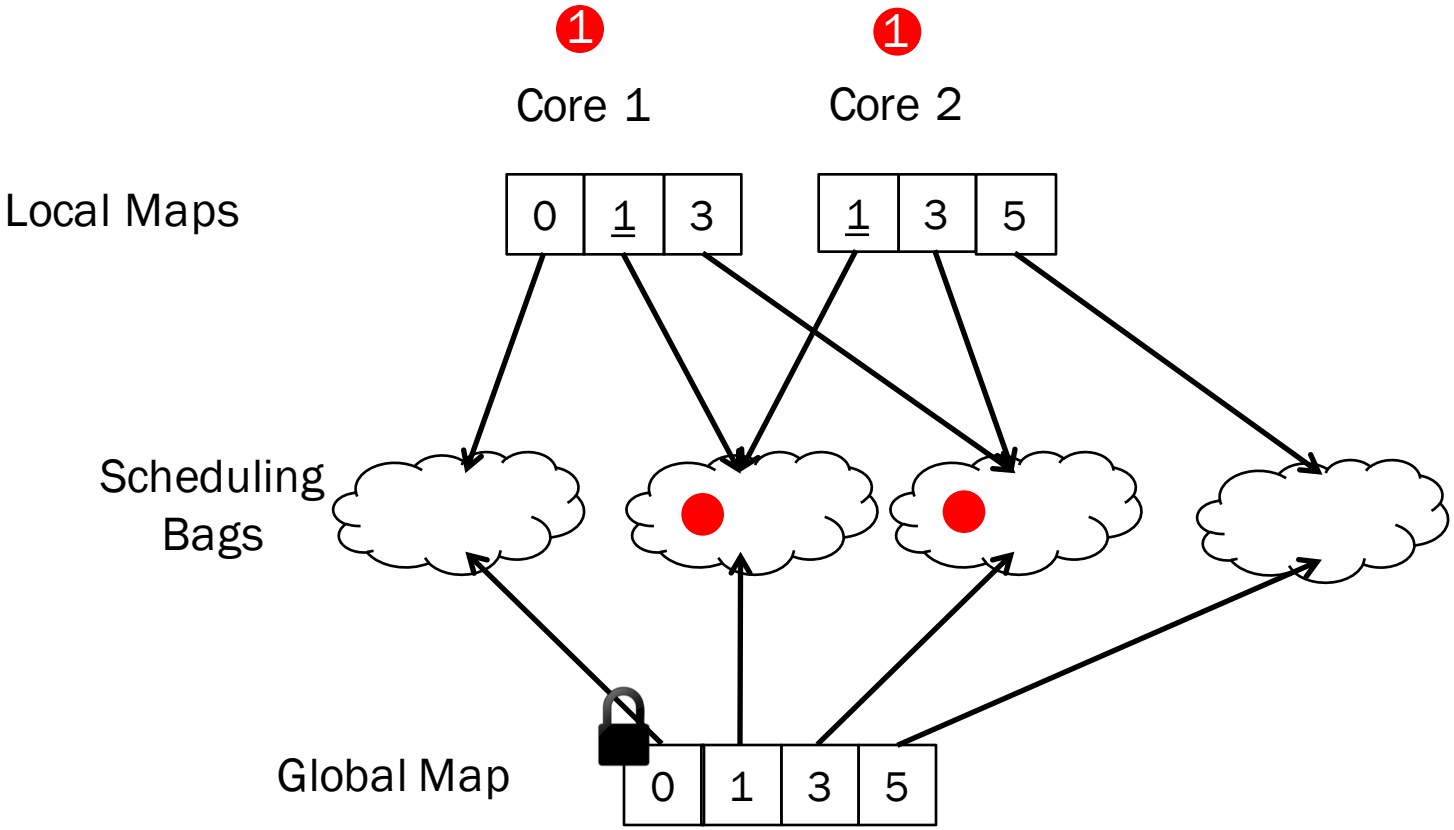
Implementation by Andrew Lenharth

Improved OrderedByIntegerMetric



Implementation by Andrew Lenharth

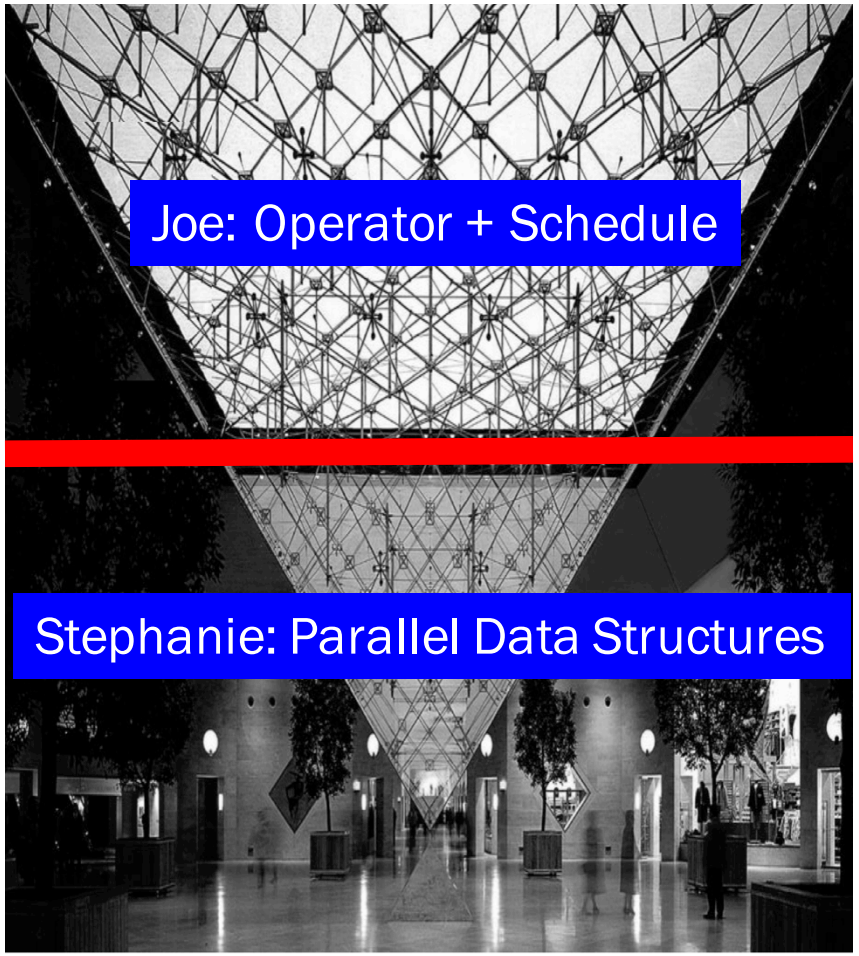
Improved OrderedByIntegerMetric



Implementation by Andrew Lenharth

Galois System

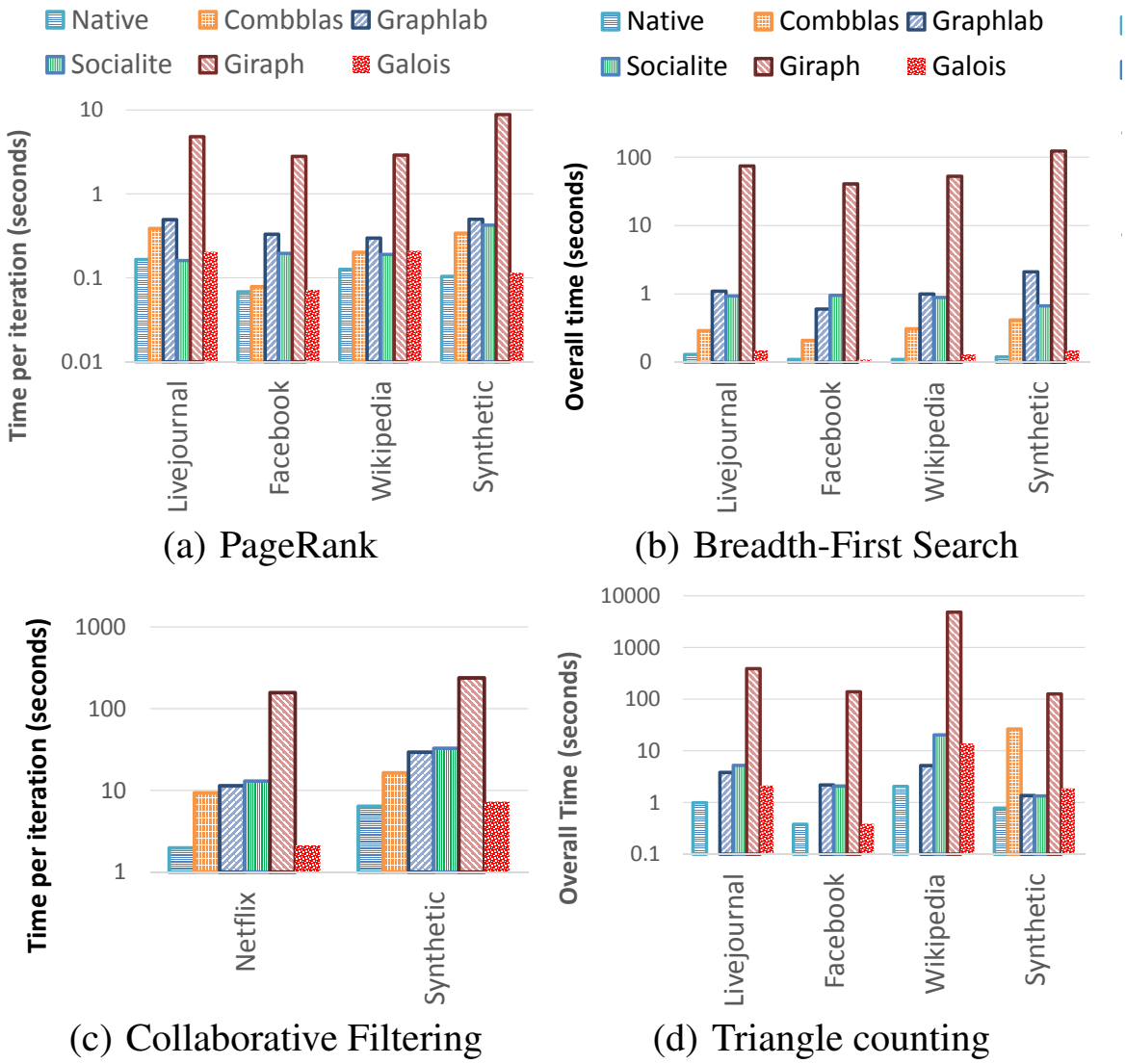
- Joe
  - Writes sequential code
  - Uses Galois data structures and scheduling hints
- Stephanie
  - Writes concurrent code
  - Provides variety of data structures and scheduling policies for Joe to choose from
    - cf., Frameworks that only support one scheduler or one graph implementation



Outline

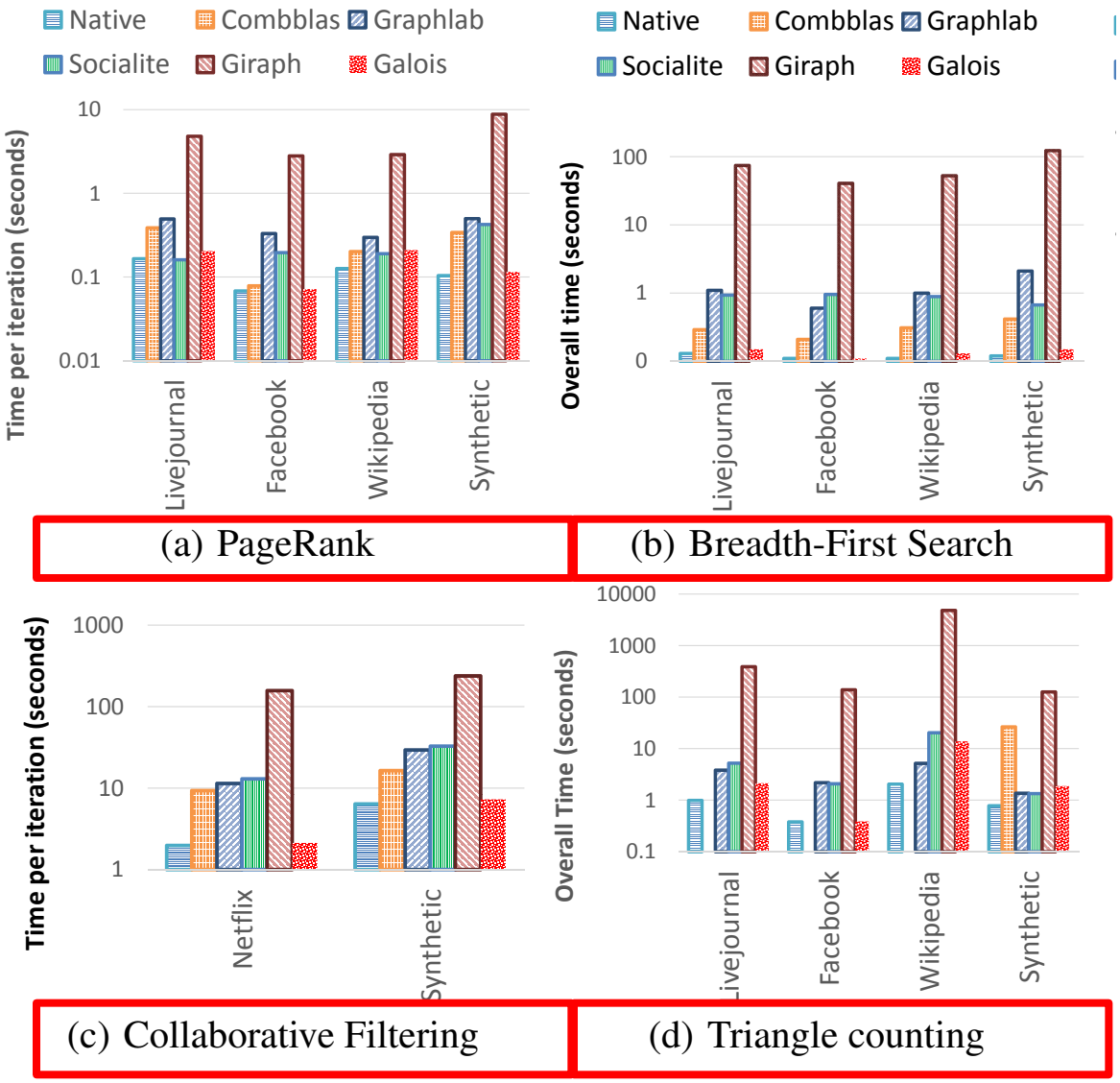
- Parallel Program = Operator + Schedule + Parallel Data Structure
- Galois system implementation
  - Operator, data structures, scheduling
- Galois studies
  - Intel study
  - Deterministic scheduling
  - Adding a scheduler: sparse tiling
  - Comparison with transactional memory
  - Implementing other programming models

Intel Study



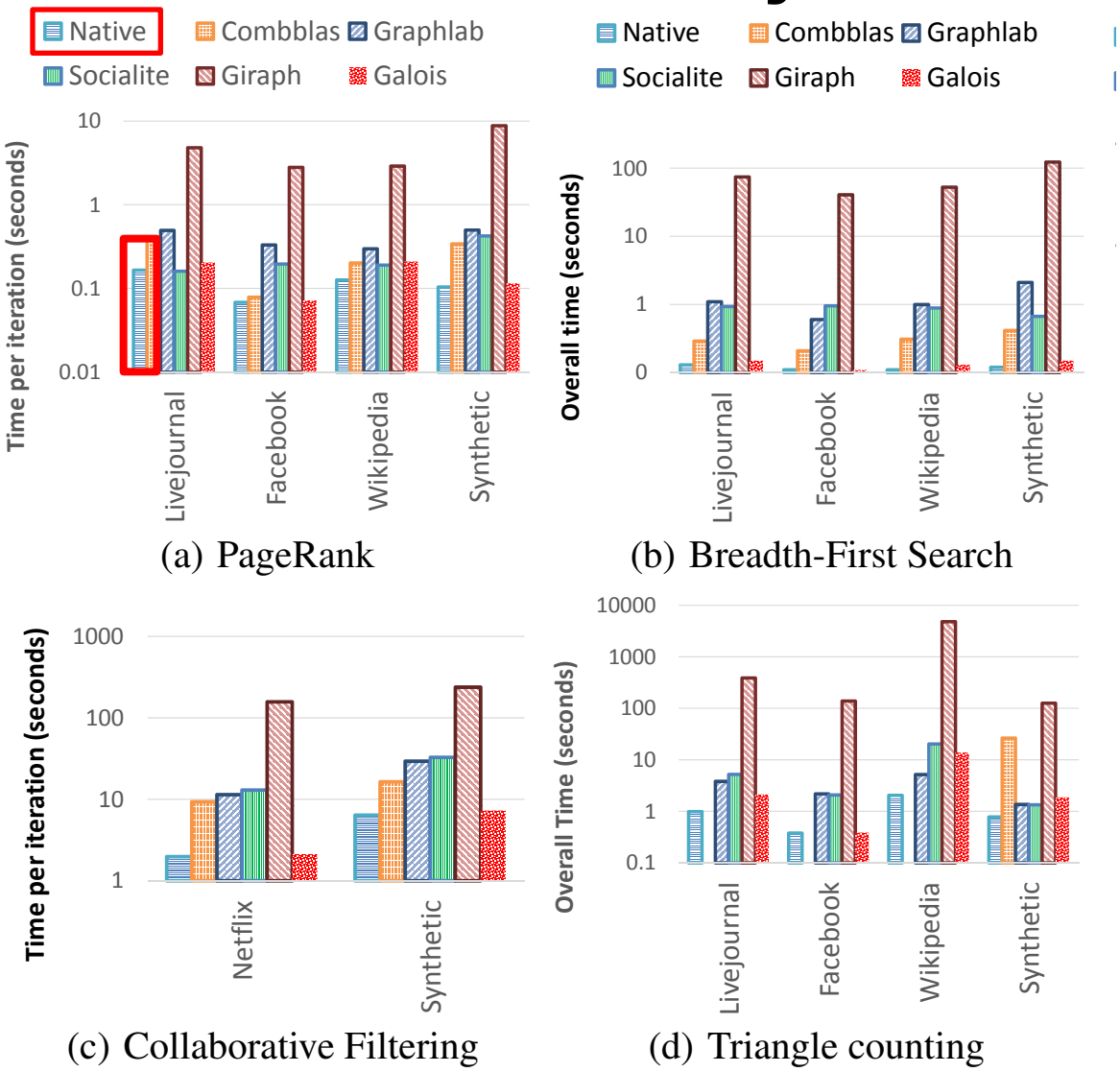
Nadathur et al. Navigating the maze of graph analytics frameworks. SIGMOD 2014.

Intel Study



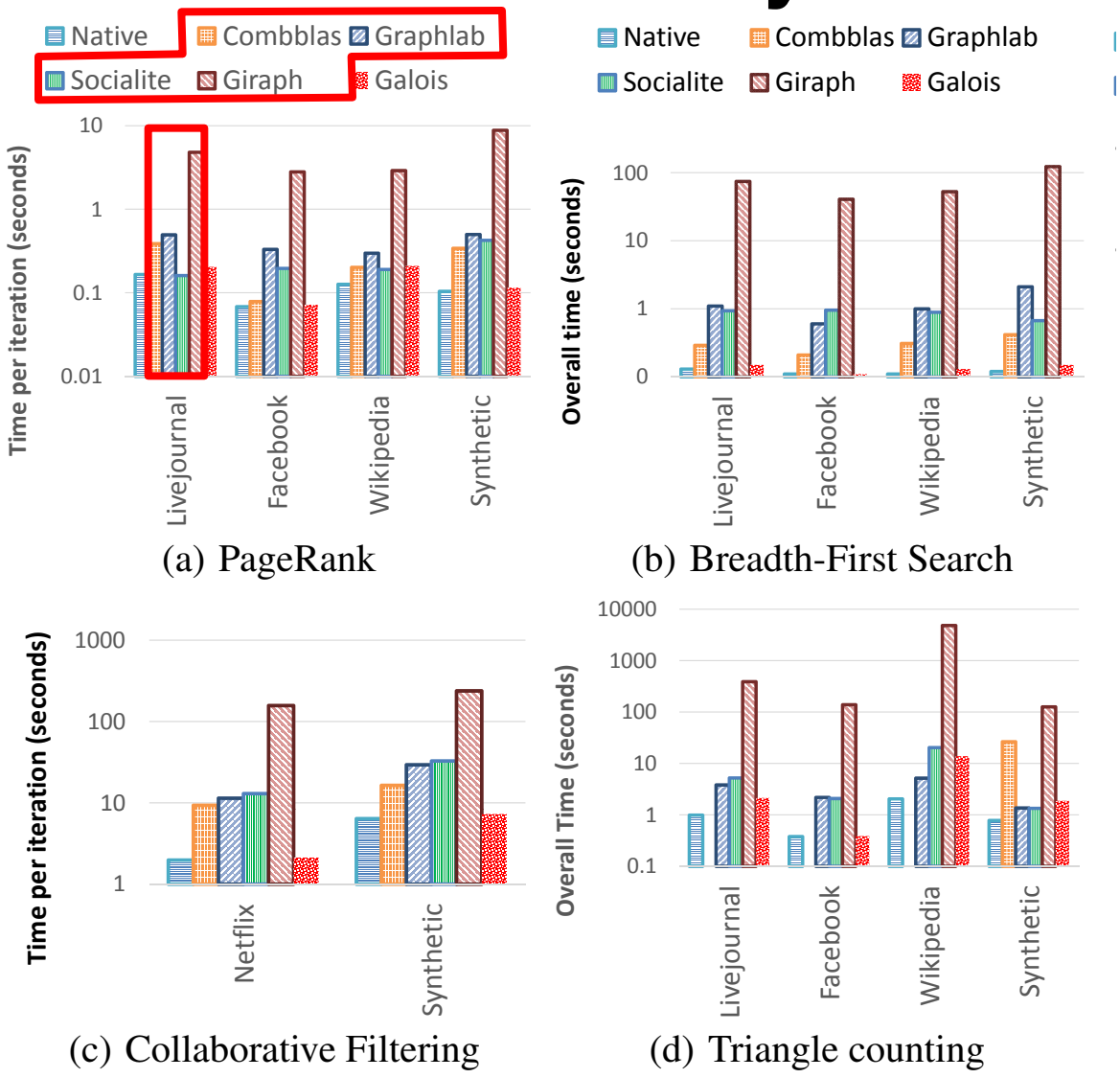
Nadathur et al. Navigating the maze of graph analytics frameworks. SIGMOD 2014.

Intel Study



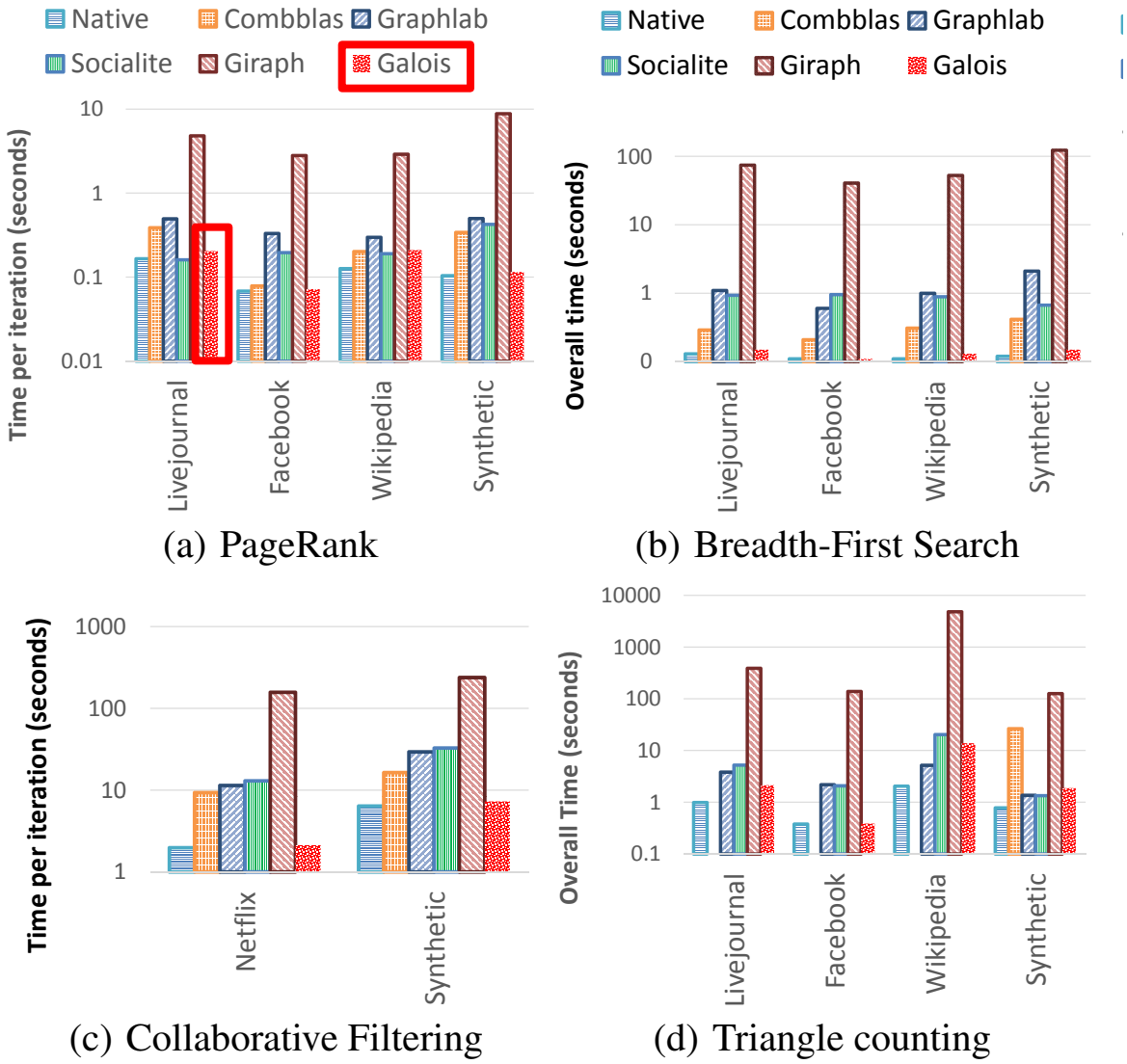
Nadathur et al. Navigating the maze of graph analytics frameworks. SIGMOD 2014.

Intel Study



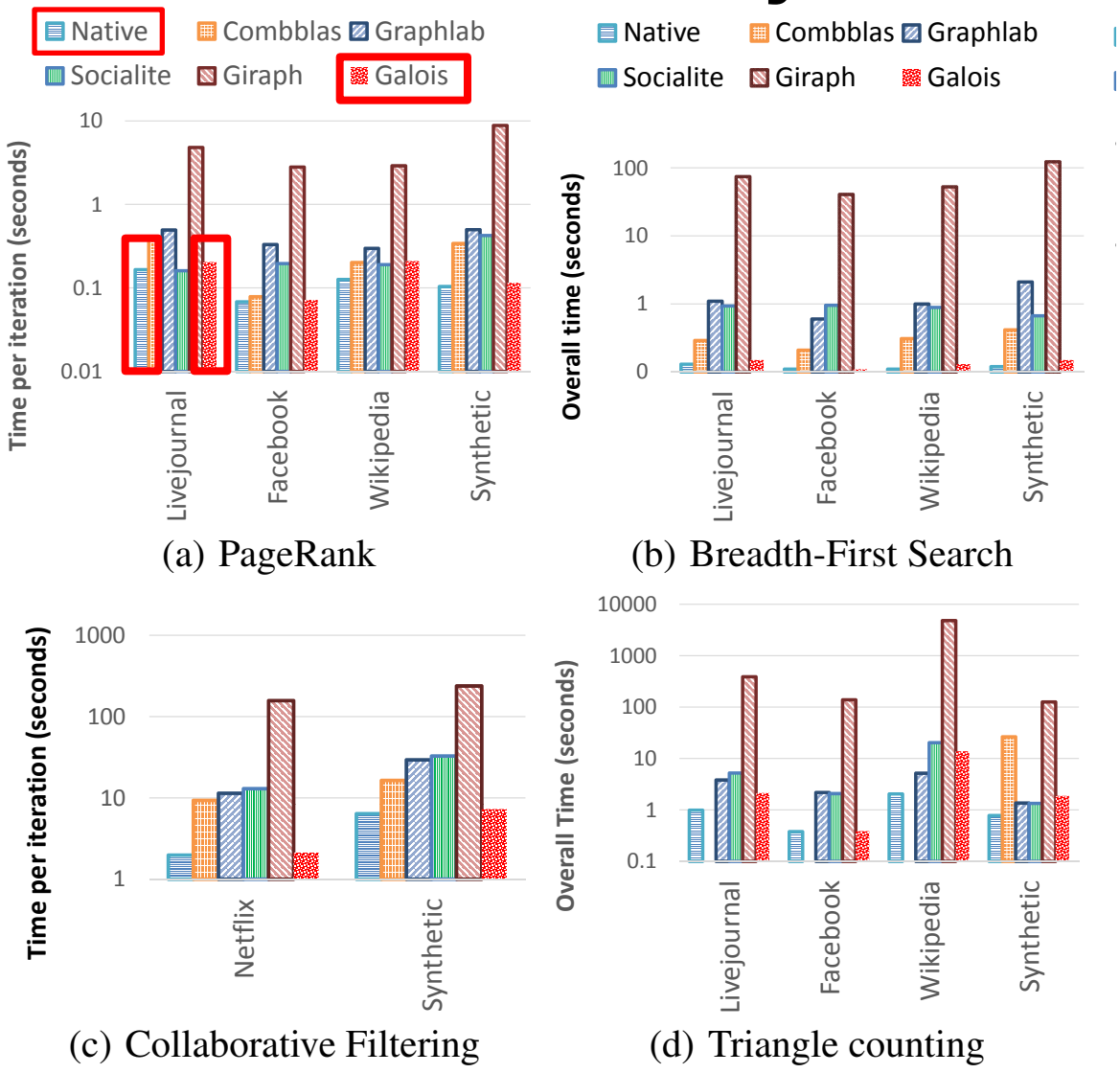
Nadathur et al. Navigating the maze of graph analytics frameworks. SIGMOD 2014.

Intel Study



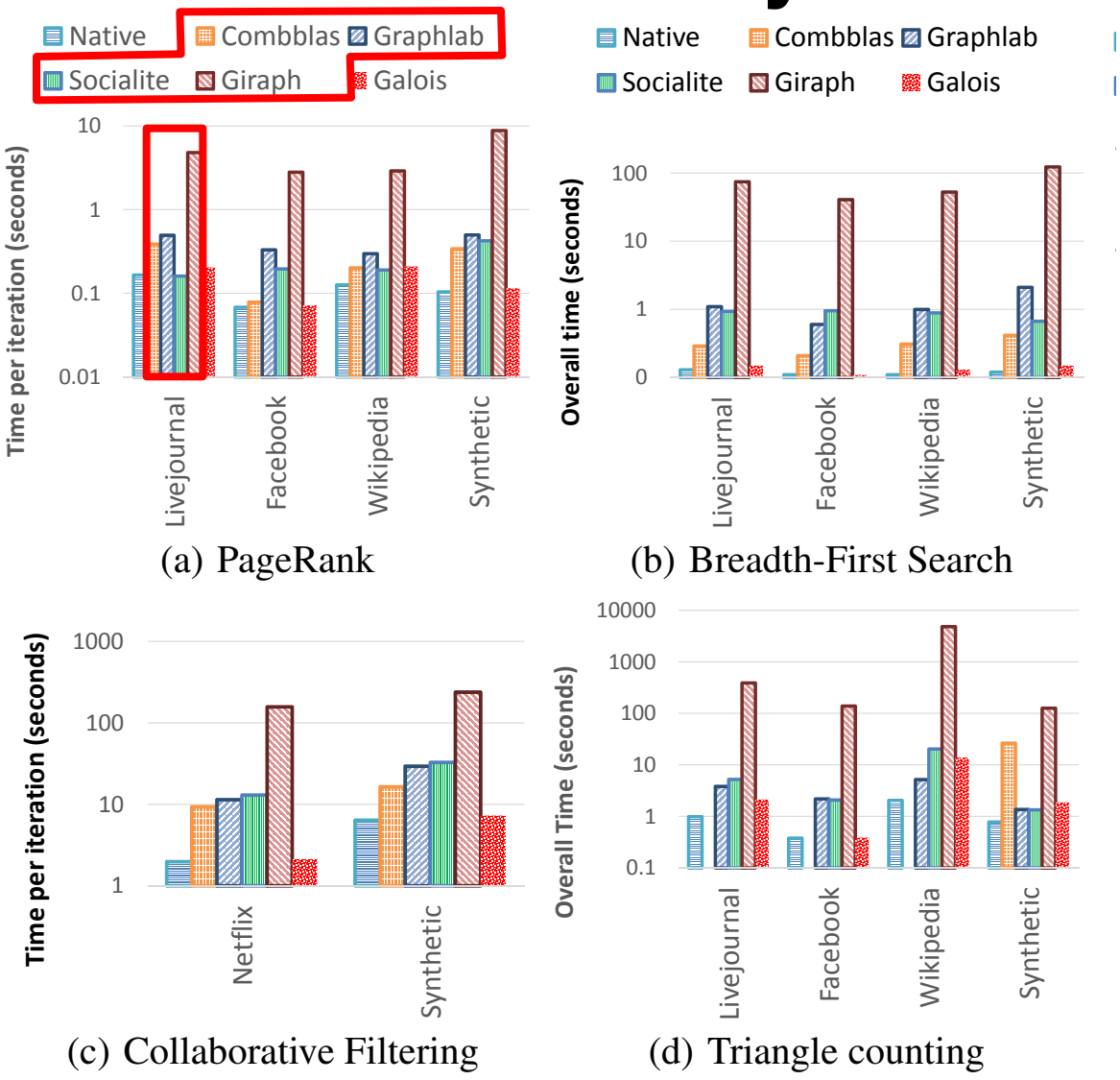
Nadathur et al. Navigating the maze of graph analytics frameworks. SIGMOD 2014.

Intel Study



Nadathur et al. Navigating the maze of graph analytics frameworks. SIGMOD 2014.

Intel Study



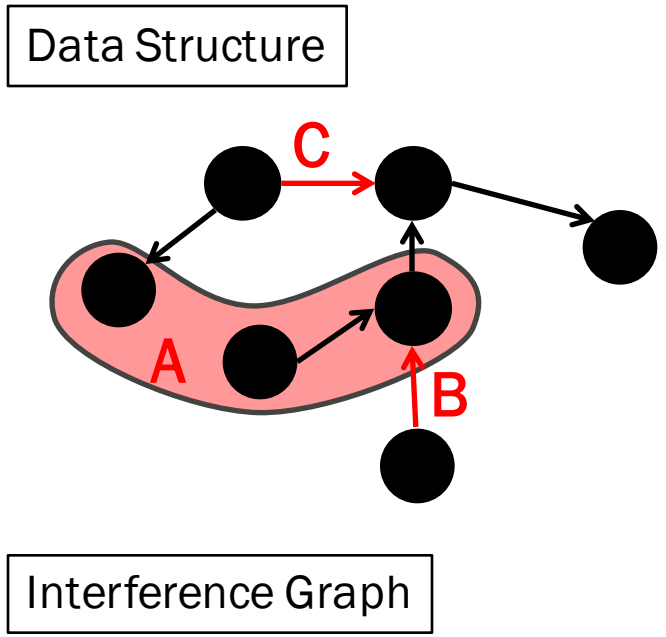
Nadathur et al. Navigating the maze of graph analytics frameworks. SIGMOD 2014.

Outline

- Parallel Program = Operator + Schedule + Parallel Data Structure
- Galois system implementation
  - Operator, data structures, scheduling
- Galois studies
  - Intel study
  - Deterministic scheduling
  - Adding a scheduler: sparse tiling
  - Comparison with transactional memory
  - Implementing other programming models

Deterministic Interference Graph (DIG) Scheduling

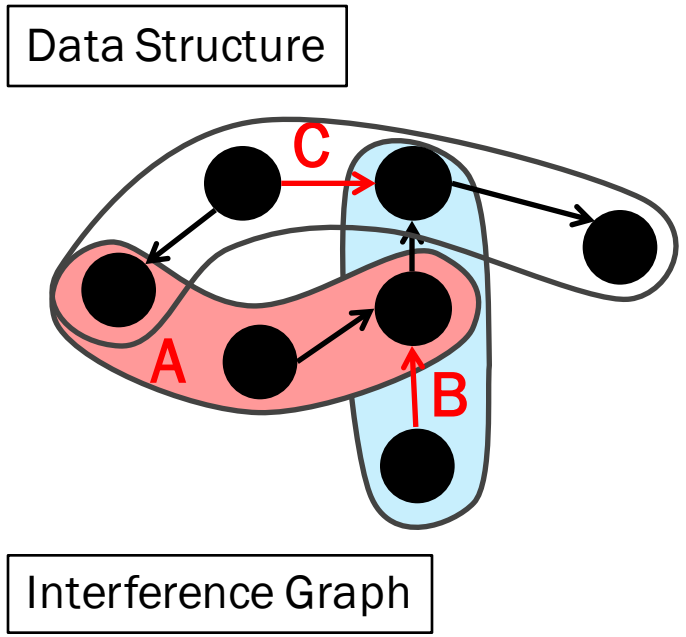
- Traditional Galois execution
  - Asynchronous, non-deterministic
- Deterministic Galois execution
  - Round-based execution
  - Construct an interference graph
  - Execute independent set
  - Repeat



Nguyen, Lenharth and Pingali. Deterministic Galois: on-demand, parameterless and portable. ASPLOS 2014.

Deterministic Interference Graph (DIG) Scheduling

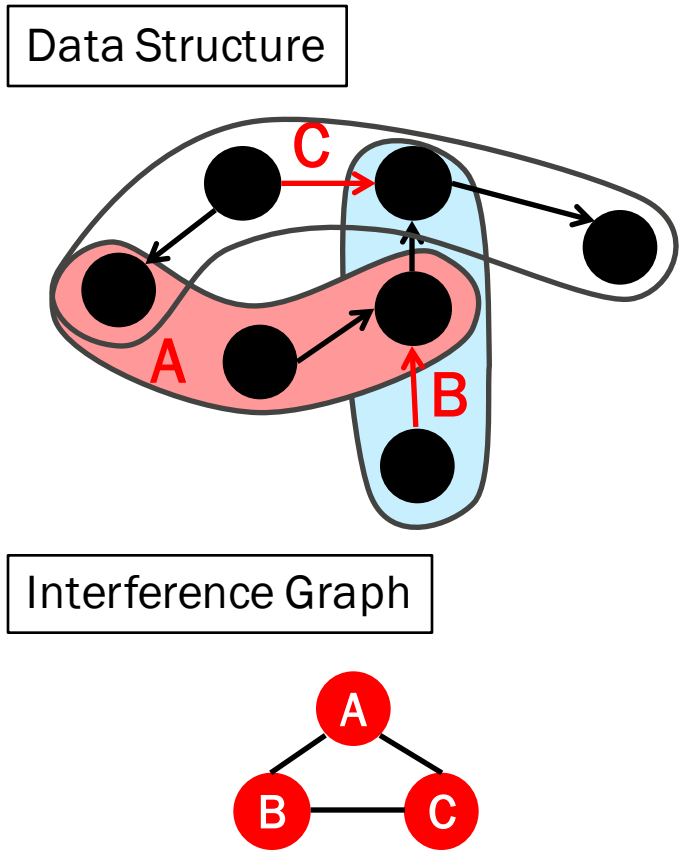
- Traditional Galois execution
  - Asynchronous, non-deterministic
- Deterministic Galois execution
  - Round-based execution
  - Construct an interference graph
  - Execute independent set
  - Repeat



Nguyen, Lenharth and Pingali. Deterministic Galois: on-demand, parameterless and portable. ASPLOS 2014.

Deterministic Interference Graph (DIG) Scheduling

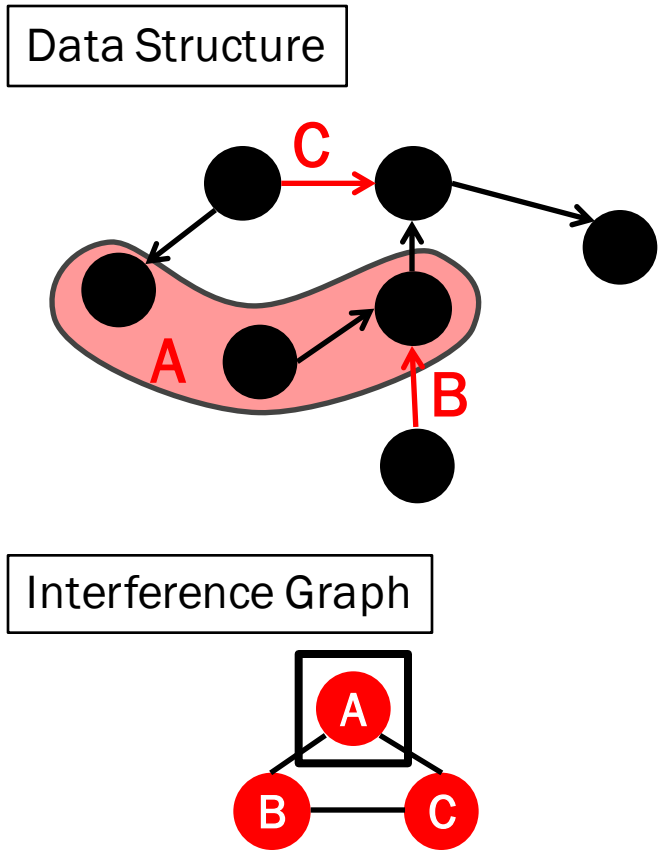
- Traditional Galois execution
  - Asynchronous, non-deterministic
- Deterministic Galois execution
  - Round-based execution
  - Construct an interference graph
  - Execute independent set
  - Repeat



Nguyen, Lenharth and Pingali. Deterministic Galois: on-demand, parameterless and portable. ASPLOS 2014.

Deterministic Interference Graph (DIG) Scheduling

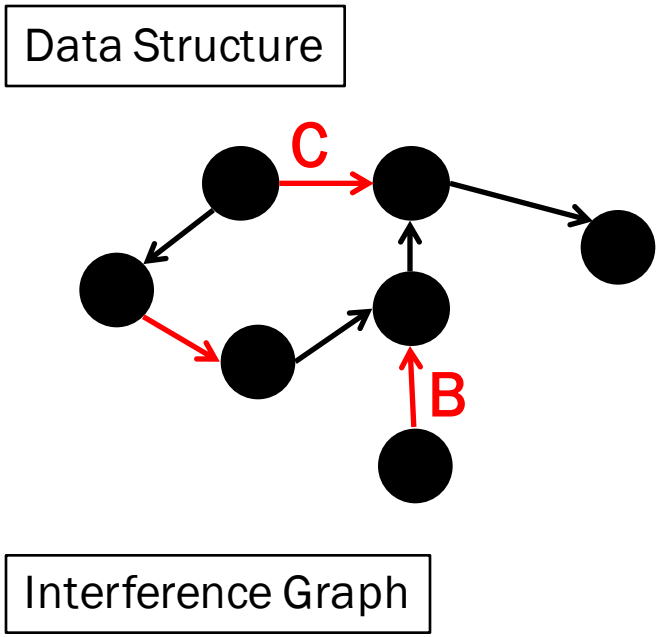
- Traditional Galois execution
  - Asynchronous, non-deterministic
- Deterministic Galois execution
  - Round-based execution
  - Construct an interference graph
  - Execute independent set
  - Repeat



Nguyen, Lenharth and Pingali. Deterministic Galois: on-demand, parameterless and portable. ASPLOS 2014.

Deterministic Interference Graph (DIG) Scheduling

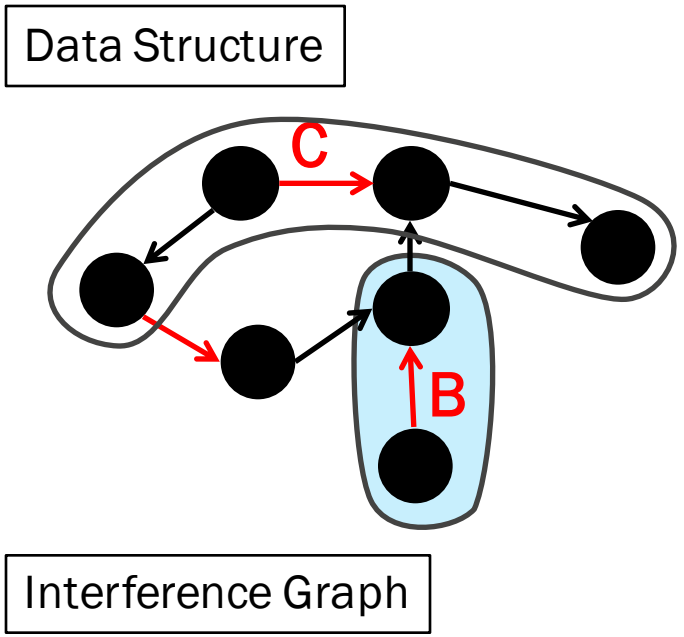
- Traditional Galois execution
  - Asynchronous, non-deterministic
- Deterministic Galois execution
  - Round-based execution
  - Construct an interference graph
  - Execute independent set
  - Repeat



Nguyen, Lenharth and Pingali. Deterministic Galois: on-demand, parameterless and portable. ASPLOS 2014.

Deterministic Interference Graph (DIG) Scheduling

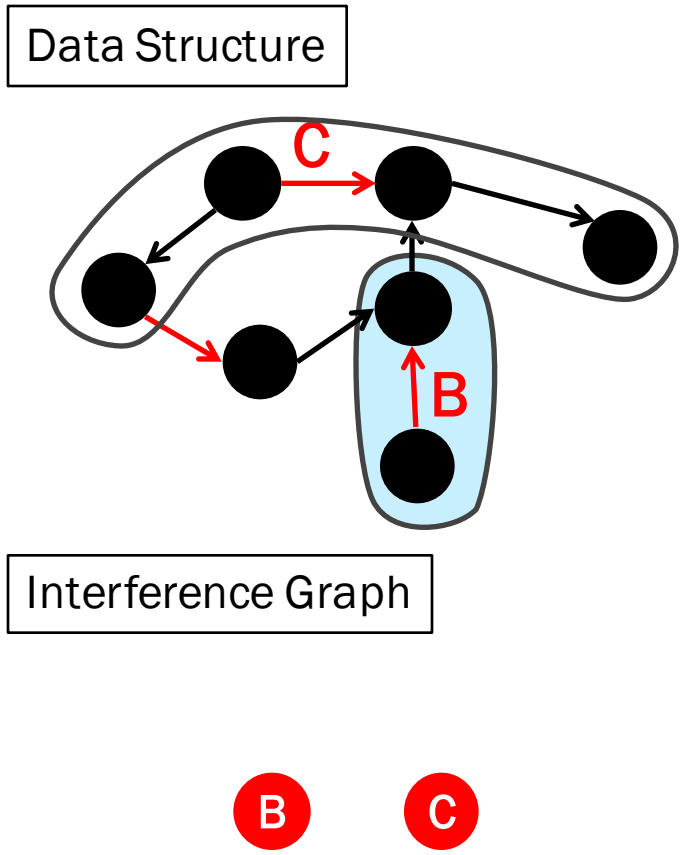
- Traditional Galois execution
  - Asynchronous, non-deterministic
- Deterministic Galois execution
  - Round-based execution
  - Construct an interference graph
  - Execute independent set
  - Repeat



Nguyen, Lenharth and Pingali. Deterministic Galois: on-demand, parameterless and portable. ASPLOS 2014.

Deterministic Interference Graph (DIG) Scheduling

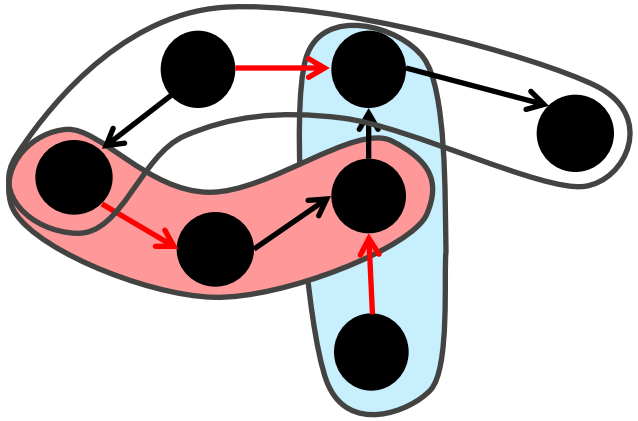
- Traditional Galois execution
  - Asynchronous, non-deterministic
- Deterministic Galois execution
  - Round-based execution
  - Construct an interference graph
  - Execute independent set
  - Repeat



Nguyen, Lenharth and Pingali. Deterministic Galois: on-demand, parameterless and portable. ASPLOS 2014.

DIG Scheduling in Practice

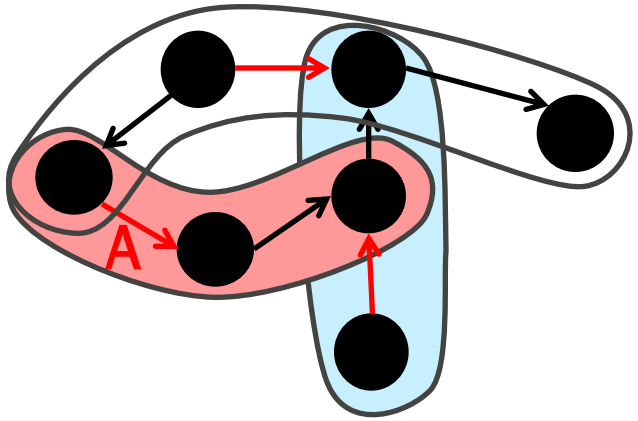
- Just find roots in interference graph
- Sequence of rounds
  - Round has two phases
- Phase 1: mark neighborhood
  - Acquire marks by writing task id atomically until failsafe point
  - Overwriting an id only if replacing greater value
  - Final mark values are deterministic
- Phase 2: execute roots
  - Execute tasks whose neighborhood only contains their own marks



A < B < C

DIG Scheduling in Practice

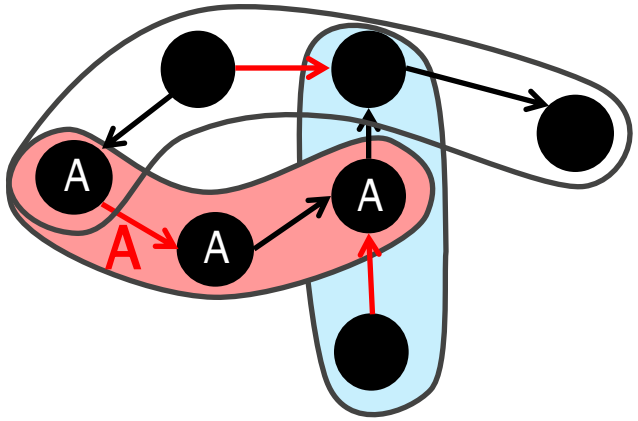
- Just find roots in interference graph
- Sequence of rounds
  - Round has two phases
- Phase 1: mark neighborhood
  - Acquire marks by writing task id atomically until failsafe point
  - Overwriting an id only if replacing greater value
  - Final mark values are deterministic
- Phase 2: execute roots
  - Execute tasks whose neighborhood only contains their own marks



A < B < C

DIG Scheduling in Practice

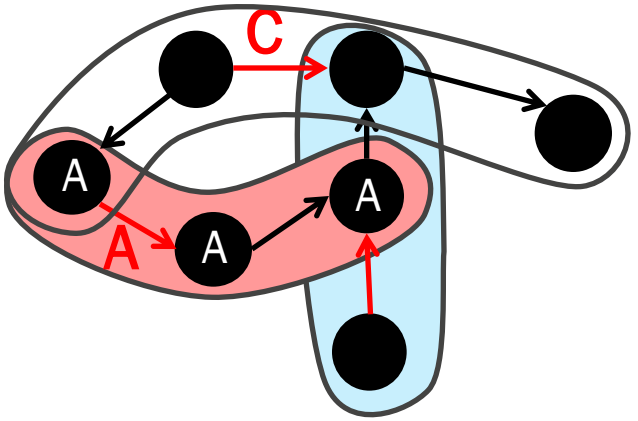
- Just find roots in interference graph
- Sequence of rounds
  - Round has two phases
- Phase 1: mark neighborhood
  - Acquire marks by writing task id atomically until failsafe point
  - Overwriting an id only if replacing greater value
  - Final mark values are deterministic
- Phase 2: execute roots
  - Execute tasks whose neighborhood only contains their own marks



A < B < C

DIG Scheduling in Practice

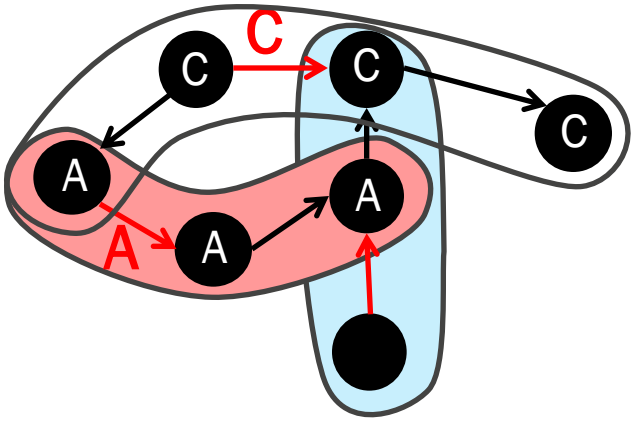
- Just find roots in interference graph
- Sequence of rounds
  - Round has two phases
- Phase 1: mark neighborhood
  - Acquire marks by writing task id atomically until failsafe point
  - Overwriting an id only if replacing greater value
  - Final mark values are deterministic
- Phase 2: execute roots
  - Execute tasks whose neighborhood only contains their own marks



A < B < C

DIG Scheduling in Practice

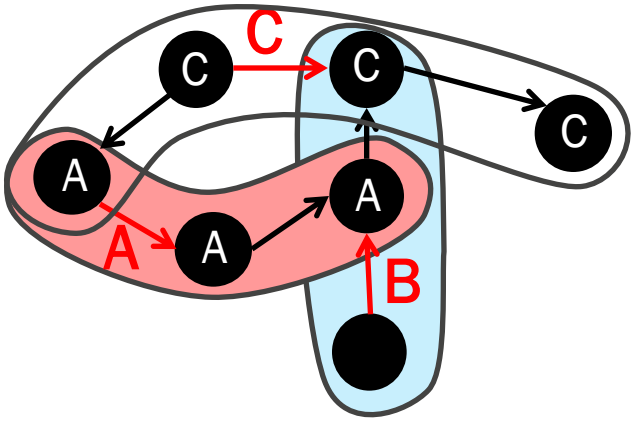
- Just find roots in interference graph
- Sequence of rounds
  - Round has two phases
- Phase 1: mark neighborhood
  - Acquire marks by writing task id atomically until failsafe point
  - Overwriting an id only if replacing greater value
  - Final mark values are deterministic
- Phase 2: execute roots
  - Execute tasks whose neighborhood only contains their own marks



A < B < C

DIG Scheduling in Practice

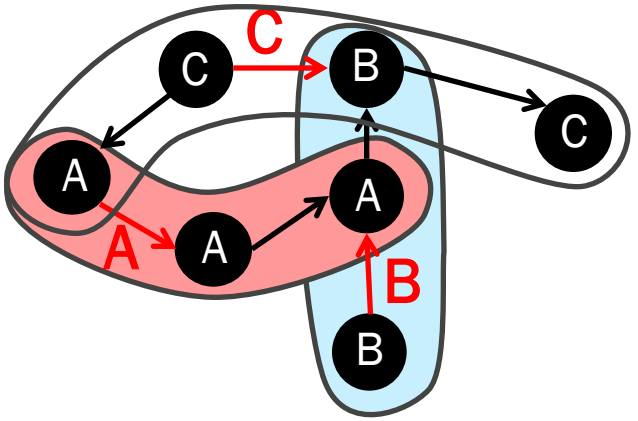
- Just find roots in interference graph
- Sequence of rounds
  - Round has two phases
- Phase 1: mark neighborhood
  - Acquire marks by writing task id atomically until failsafe point
  - Overwriting an id only if replacing greater value
  - Final mark values are deterministic
- Phase 2: execute roots
  - Execute tasks whose neighborhood only contains their own marks



A < B < C

DIG Scheduling in Practice

- Just find roots in interference graph
- Sequence of rounds
  - Round has two phases
- Phase 1: mark neighborhood
  - Acquire marks by writing task id atomically until failsafe point
  - Overwriting an id only if replacing greater value
  - Final mark values are deterministic
- Phase 2: execute roots
  - Execute tasks whose neighborhood only contains their own marks



A < B < C

Optimizations

- Resuming tasks
  - Avoid re-executing tasks
  - Suspend and resume execution at failsafe point
  - Provide buffers to save local state
- Windowing
  - Inspect only a subset of tasks at a time
  - Adaptive algorithm varies window size

Evaluation of Deterministic Scheduling

Platform

- Intel 4x10 core (2.27 GHz)

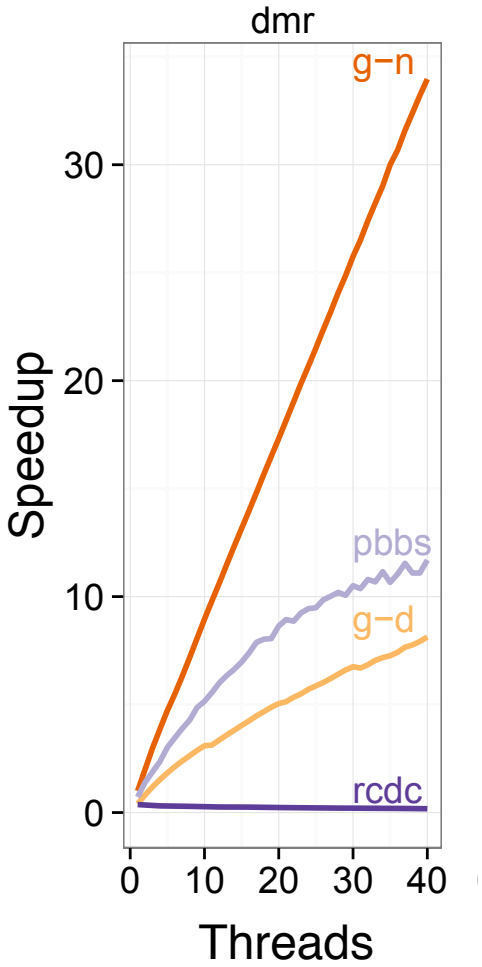
Applications

- PBBS [Blelloch11]
  - Breadth-first search (bfs)
  - Delaunay mesh refinement (dmr)
  - Delaunay triangulation (dt)
  - Maximal independent set (mis)

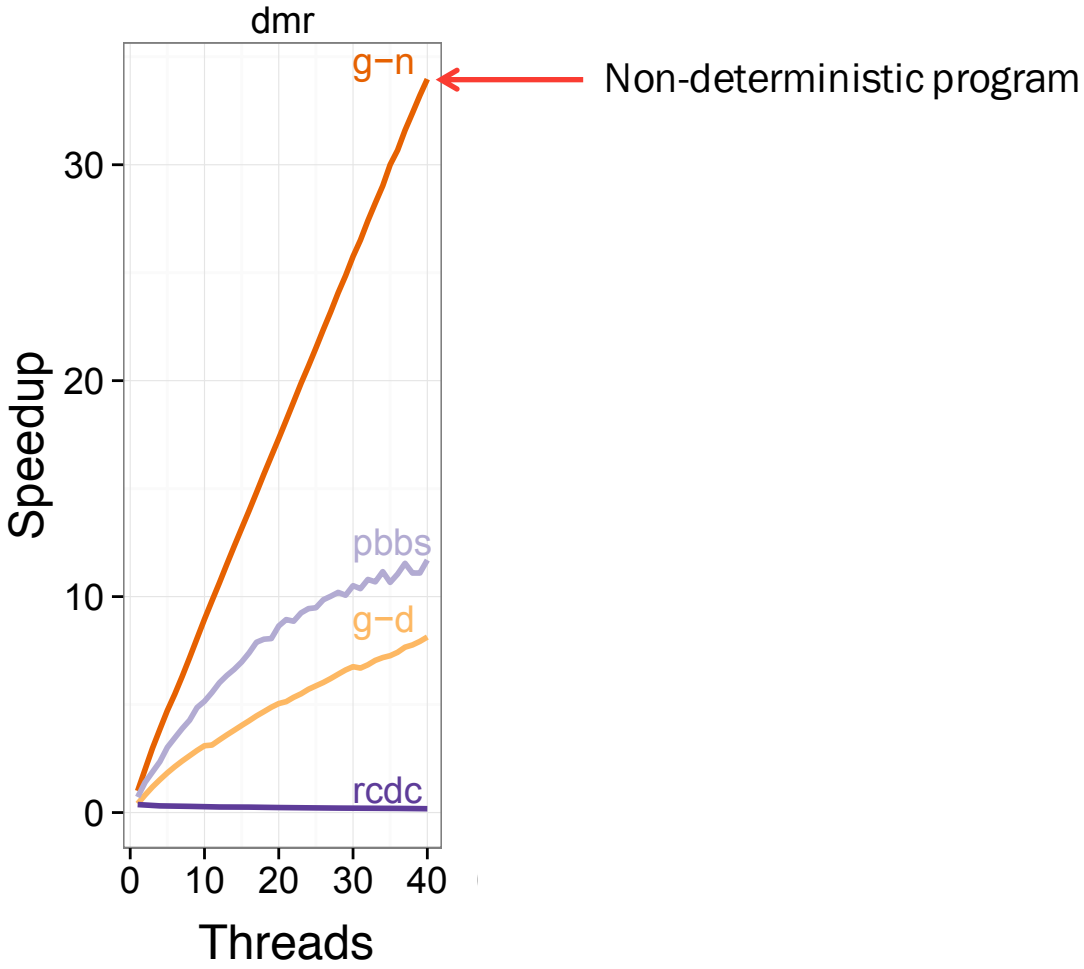
Deterministic Systems

- RCDC [Devietti11]
  - Deterministic by scheduling
  - Implementation of Kendo algorithm [Olszewski09]
- PBBS
  - Deterministic by construction
  - Handwritten deterministic implementations
- Deterministic Galois
  - Non-deterministic programs (g-n) automatically made deterministic (g-d)

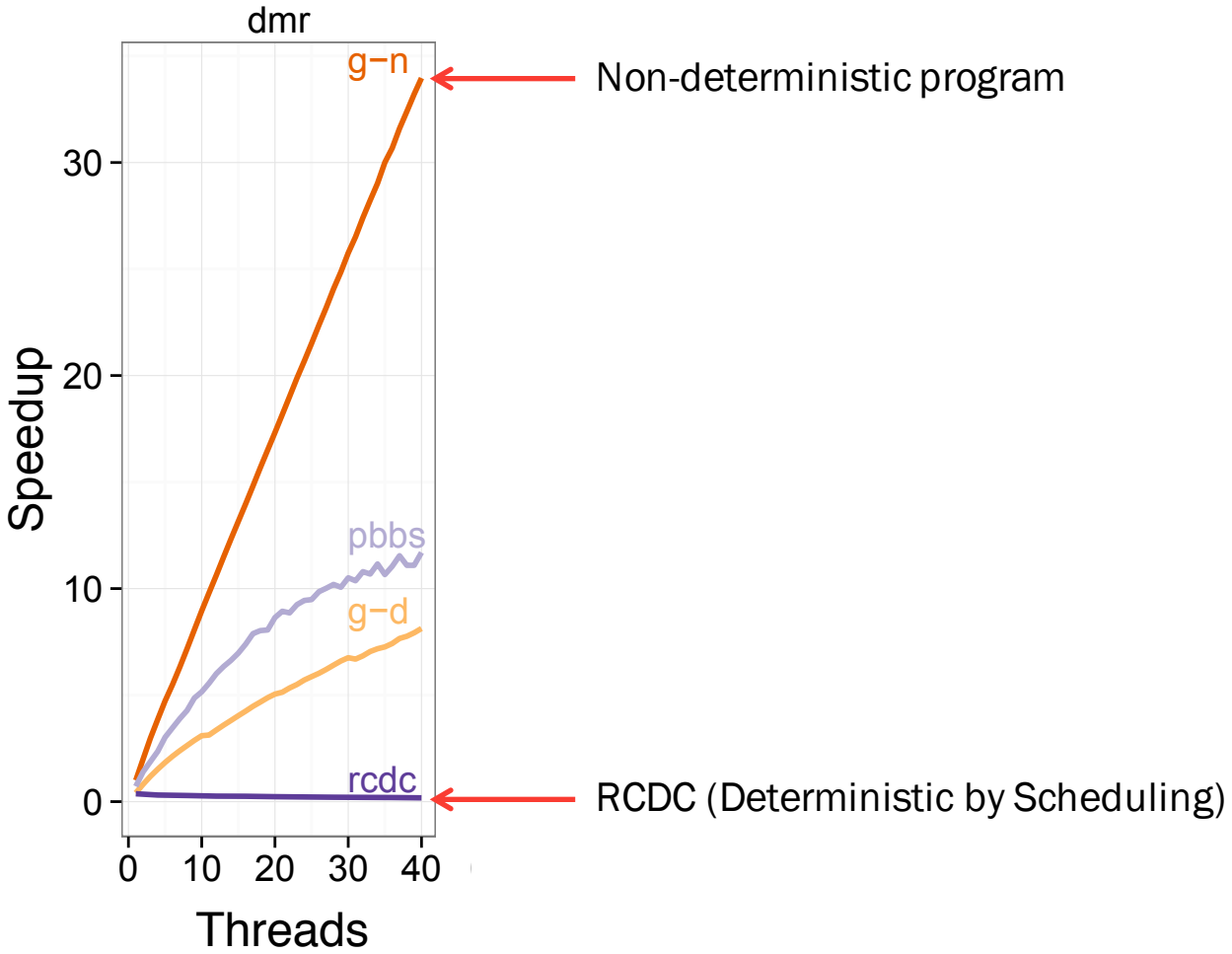
Evaluation



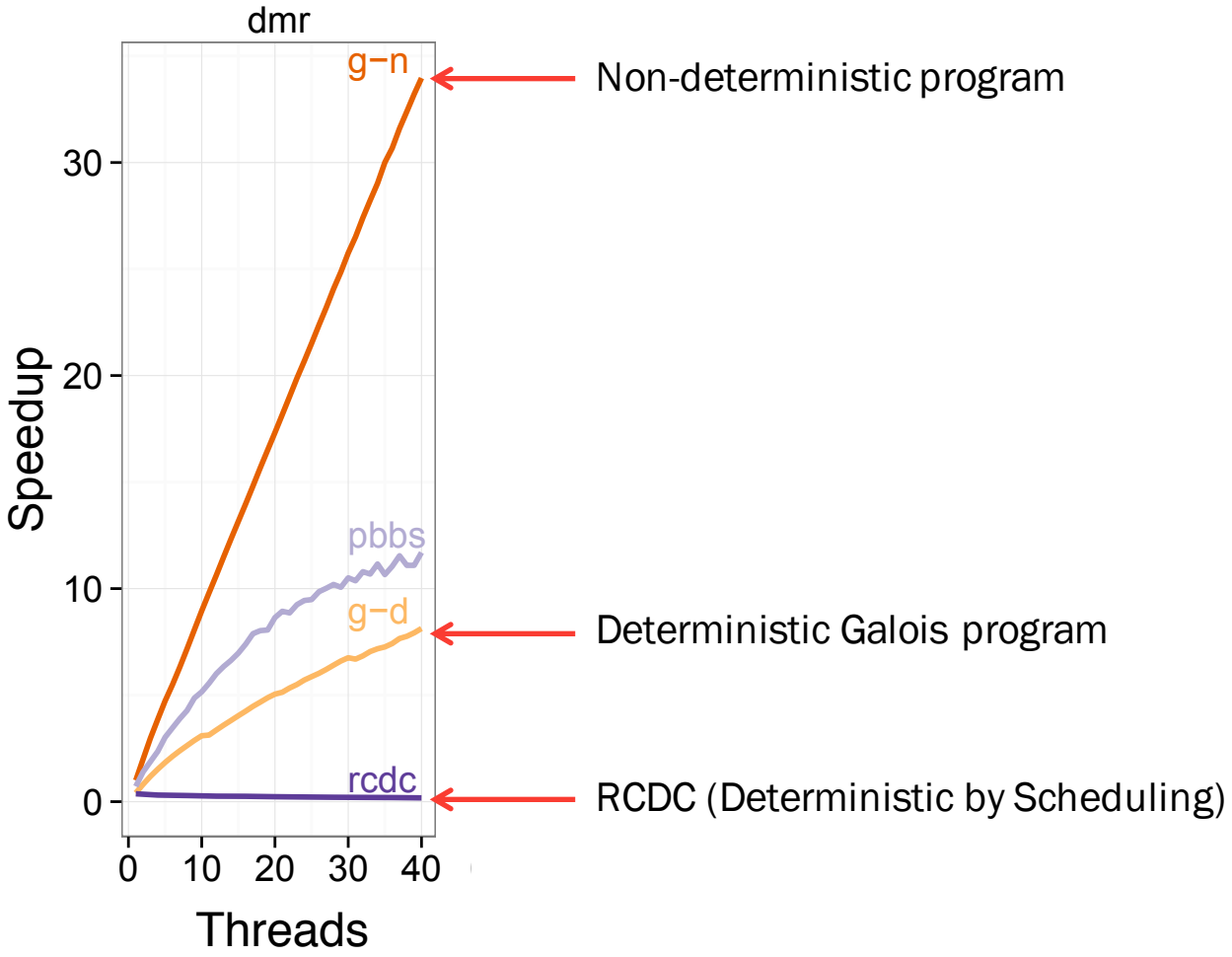
Evaluation


dmr
g−n
Non-deterministic program
30
20
10
0
Speedup
pbbs
g−d
rcdc
0
10
20
30
40
Threads

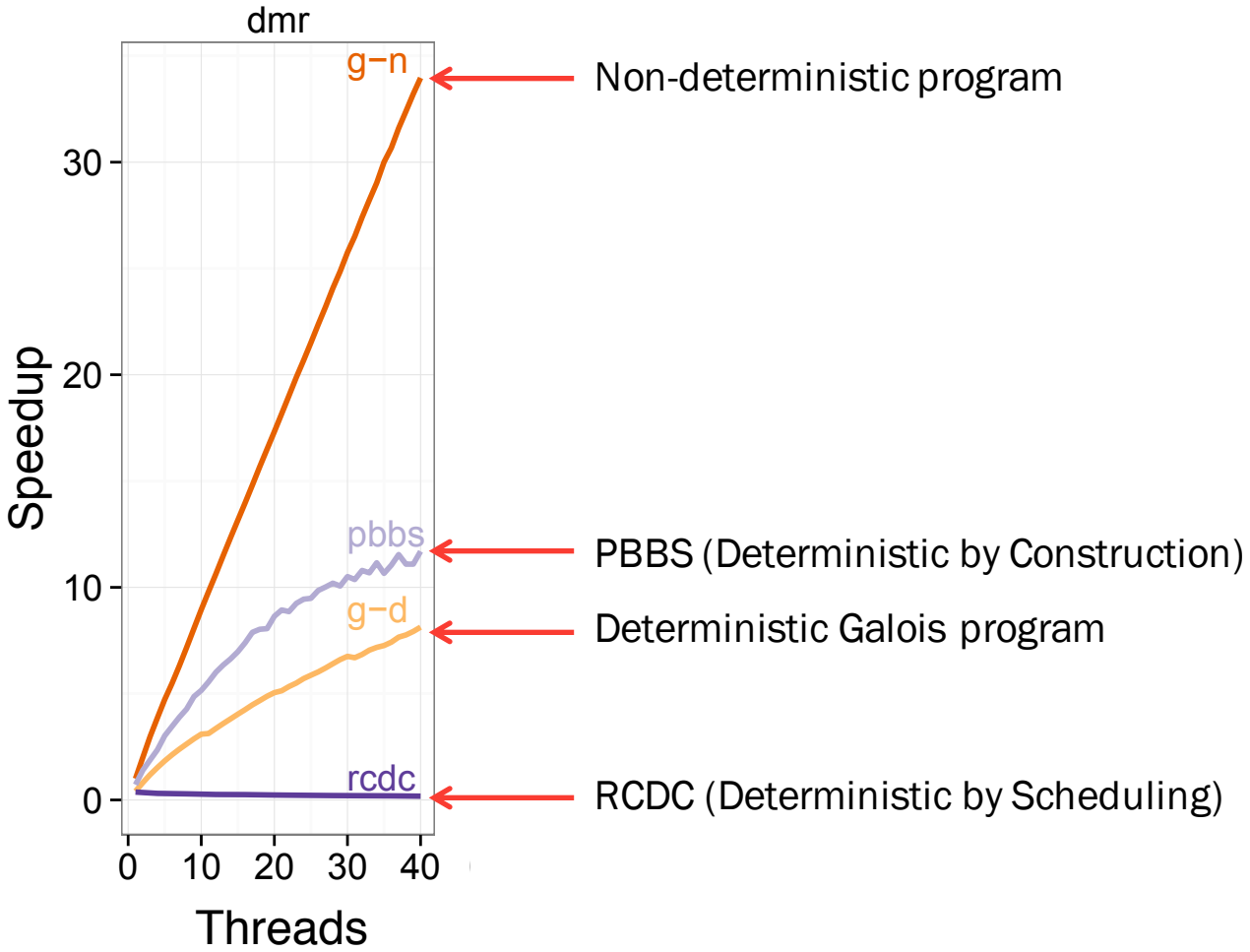
Evaluation


dmr
g−n
pbbs
g−d
rcdc
Speedup
0
10
20
30
0
10
20
30
40
Threads
Non-deterministic program
RCDC (Deterministic by Scheduling)

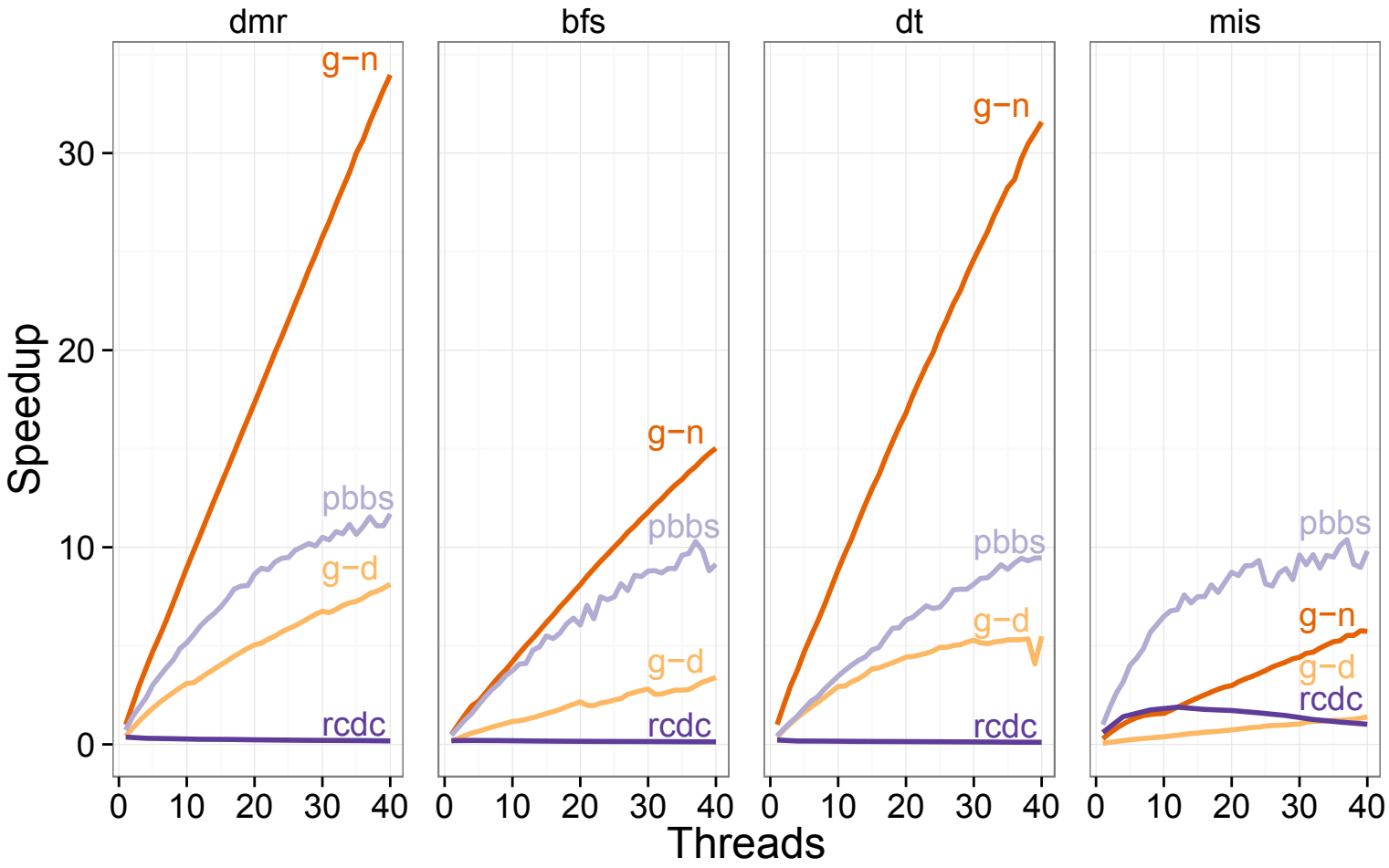
Evaluation


dmr
g−n
pbbs
g−d
rcdc
Speedup
0
10
20
30
0
10
20
30
40
Threads
Non-deterministic program
Deterministic Galois program
RCDC (Deterministic by Scheduling)

Evaluation

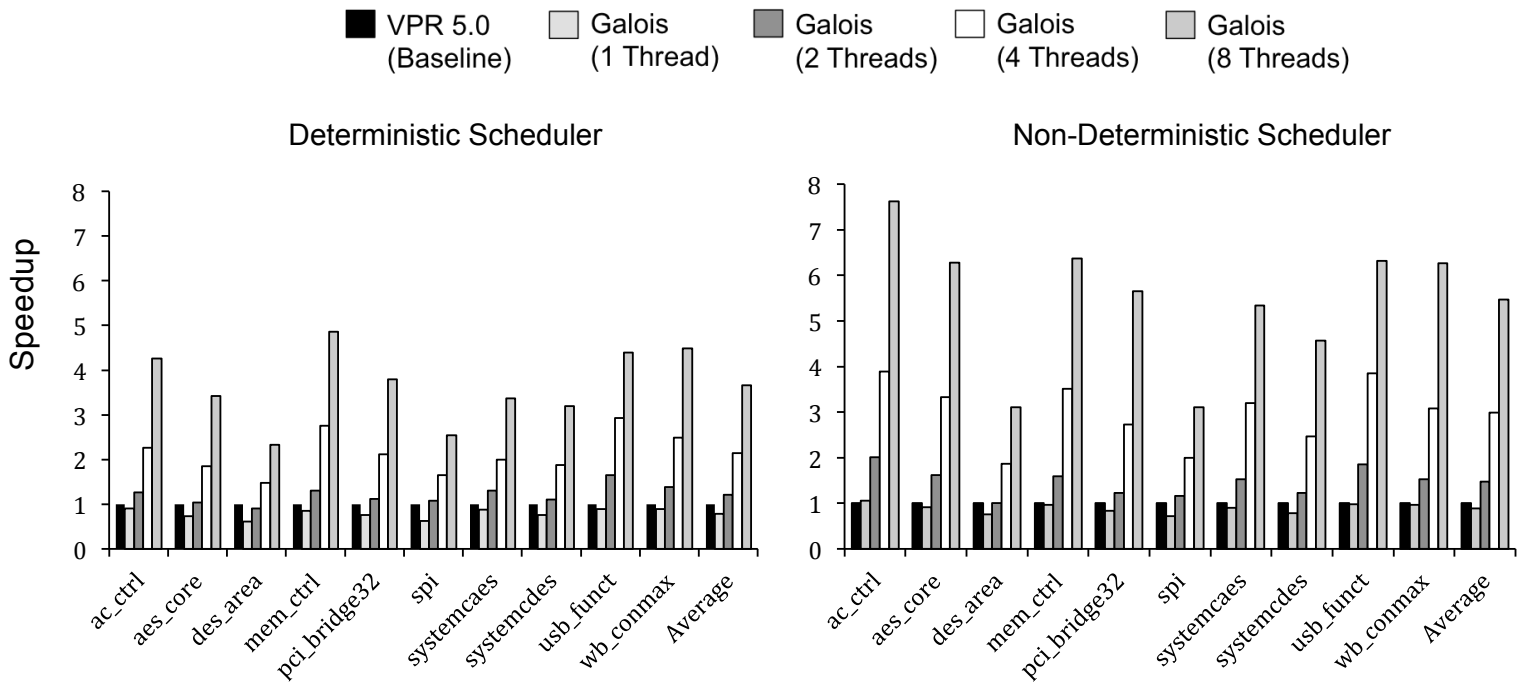

dmr
g−n
pbbs
g−d
rcdc
Speedup
0
10
20
30
0
10
20
30
40
Threads
Non-deterministic program
PBBS (Deterministic by Construction)
Deterministic Galois program
RCDC (Deterministic by Scheduling)

Evaluation



Third-party Evaluation

- In FPGA routing useful to have deterministic results



Averages		
	VPR 5.0	1.00x
	Galois (1 Thread)	0.79x
	Galois (2 Threads)	1.21x
	Galois (4 Threads)	2.14x
	Galois (8 Threads)	3.67x

VPR 5.0	1.00x
Galois (1 Thread)	0.89x
Galois (2 Threads)	1.47x
Galois (4 Threads)	2.99x
Galois (8 Threads)	5.46x

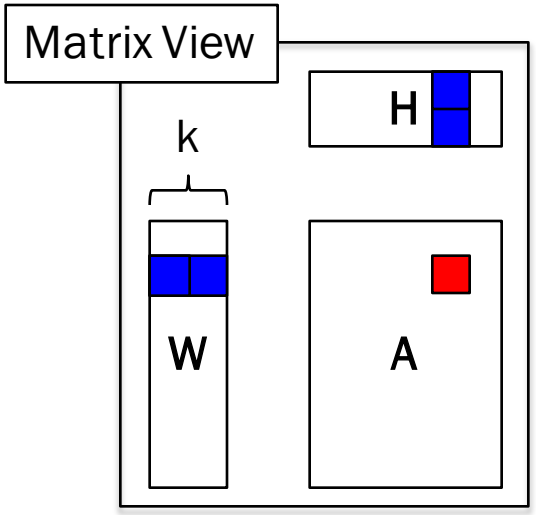
Extended results from: Moctar and Brisk. Parallel FPGA Routing based on the Operator Formulation. DAC 2014.

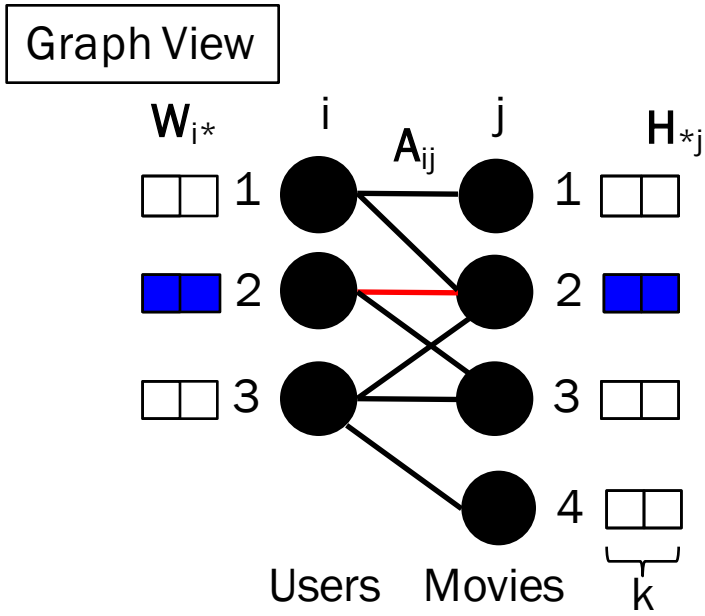
Outline

- Parallel Program = Operator + Schedule + Parallel Data Structure
- Galois system implementation
  - Operator, data structures, scheduling
- Galois studies
  - Intel study
  - Deterministic scheduling
  - Adding a scheduler: sparse tiling
  - Comparison with transactional memory
  - Implementing other programming models

Matrix Completion

- Matrix completion problem
  - Netflix problem
- An optimization problem
  - Find m×k matrix **W** and k×n matrix **H** (k << min(m,n)) such that **A** ≈ **WH**
  - Low-rank approximation
- Algorithms
  - Stochastic gradient descent (SGD)
  - …

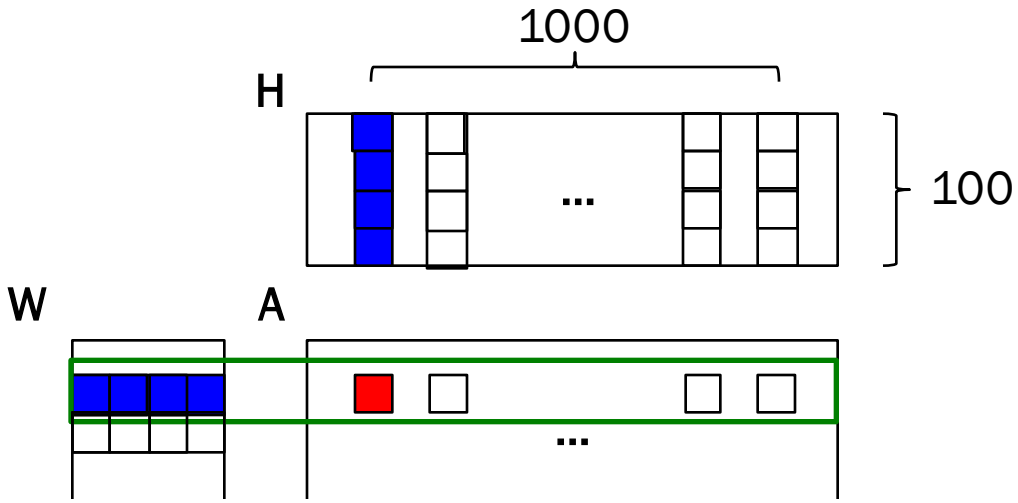




Naïve SGD Implementation

- Row-wise operator

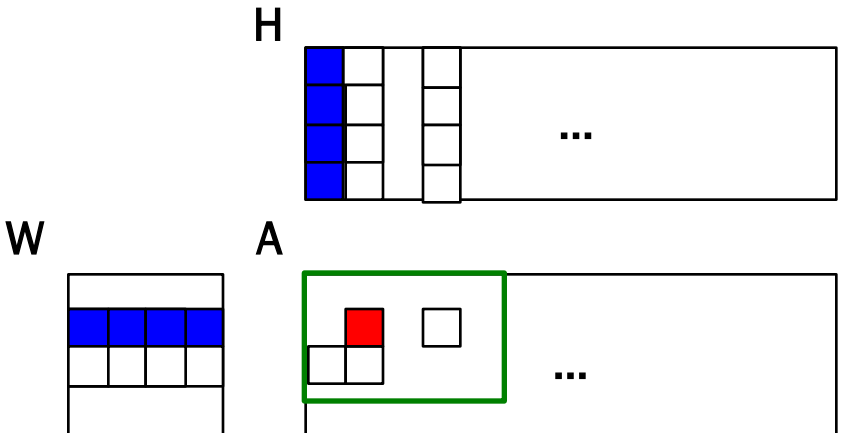
```
for (User u : users)
  for (Item m : items(u))
    op(u, m, Aij);
```



- Poor cache behavior
  - With k = 100, $W_{i*} \approx$ 1KB
  - Degree(user) > 1000

2D Tiled SGD

- Apply operator to small 2D tiles



- Additional concerns
  - Conflict-free scheduling of tiles
- Future optimizations
  - Adaptive, non-uniform tiling

Evaluation of Tiling

Implementations

- Galois
  - 2D scheduling
    - 2 weeks to implement
  - Synchronized updates
- GraphLab
  - Standard GraphLab only supports GD
  - Implement 1D SGD with unsynchronized updates
- Nomad
  - From scratch distributed code
    - ~1 year to implement
  - 2D scheduling
    - Tile size is function of #threads
  - Synchronized updates

Datasets

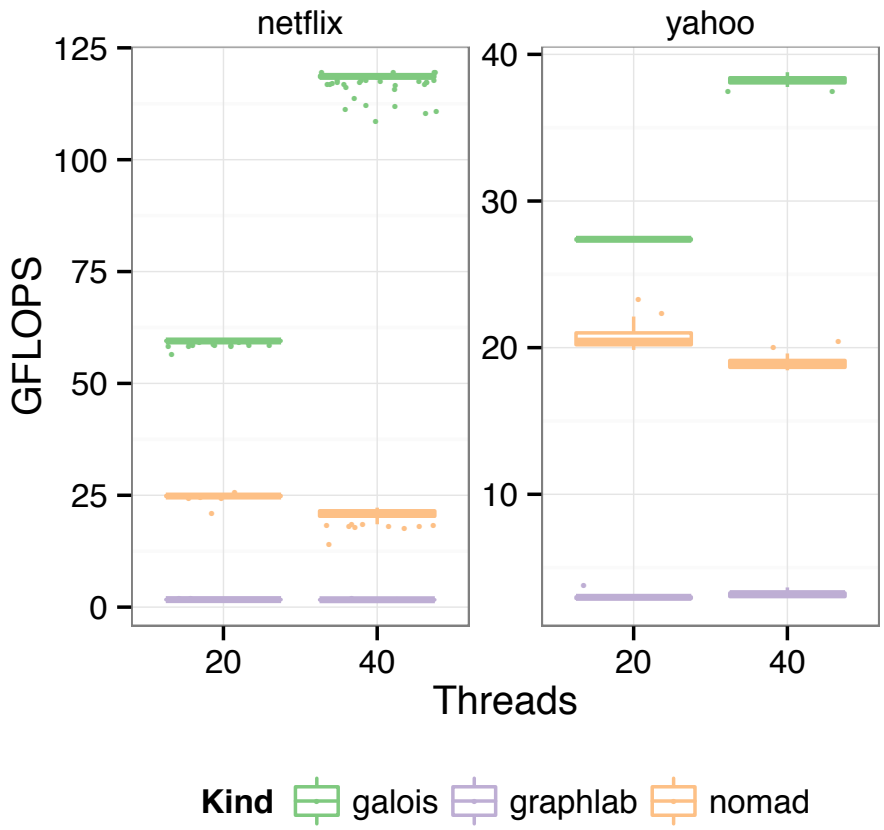
	Items	Users	Ratings	Sparsity
netflix	17K	480K	99M	1e-2
yahoo	624K	1M	253M	4e-4

Parameters

- Same SGD parameters, initial latent vectors between implementations
- $\varepsilon(n) = \text{alpha} / (1 + \text{beta} * n^{1.5})$
- Handtuned tile size

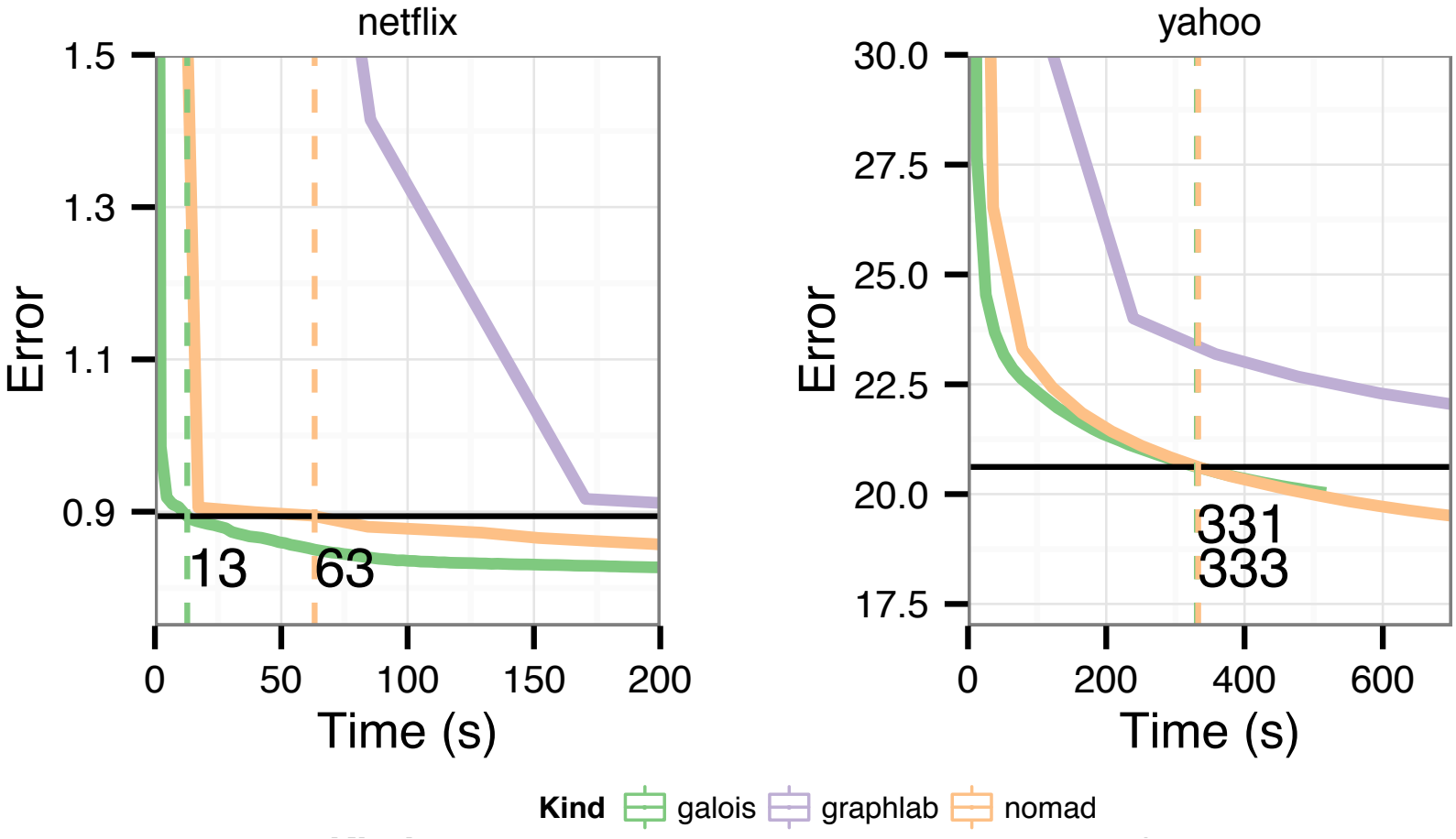
Yun, Yu, Hsieh, Vishwanathan, Dhillon. NOMAD: Non-locking stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion. VLDB 2013.

Throughput



Theoretical Peak: 363.2 GFLOPS

Convergence



20 Threads

Outline

- Parallel Program = Operator + Schedule + Parallel Data Structure
- Galois system implementation
  - Operator, data structures, scheduling
- Galois studies
  - Intel study
  - Deterministic scheduling
  - Adding a scheduler: sparse tiling
  - Comparison with transactional memory
  - Implementing other programming models

Transactional Memory

- Alternative to adopting a new programming model
  - Parallelization reusing existing sequential programs
- Transactional memory
  - Add threads and transactions but keep existing sequential code
  - TM system ensures correct execution of atomic regions
    - Software and hardware impl.
  - Speculative execution

```
while (begin != end)
  begin += 1
  while (head)
    ...
    head = next
  root.size += 1
```

Sequential Program

```
while (begin != end)

    begin += 1

    while (head)
      ...
      head = next
    root.size += 1
```

TM Program

Transactional Memory

- Alternative to adopting a new programming model
  - Parallelization reusing existing sequential programs
- Transactional memory
  - Add threads and transactions but keep existing sequential code
  - TM system ensures correct execution of atomic regions
    - Software and hardware impl.
  - Speculative execution

```
while (begin != end)
  begin += 1
  while (head)
    ...
    head = next
  root.size += 1
```

Sequential Program

```
fork()
while (begin != end)

    begin += 1

    while (head)
      ...
      head = next
    root.size += 1
join()
```

TM Program

Transactional Memory

- Alternative to adopting a new programming model
  - Parallelization reusing existing sequential programs
- Transactional memory
  - Add threads and transactions but keep existing sequential code
  - TM system ensures correct execution of atomic regions
    - Software and hardware impl.
  - Speculative execution

```
while (begin != end)
  begin += 1
  while (head)
    ...
    head = next
  root.size += 1
```

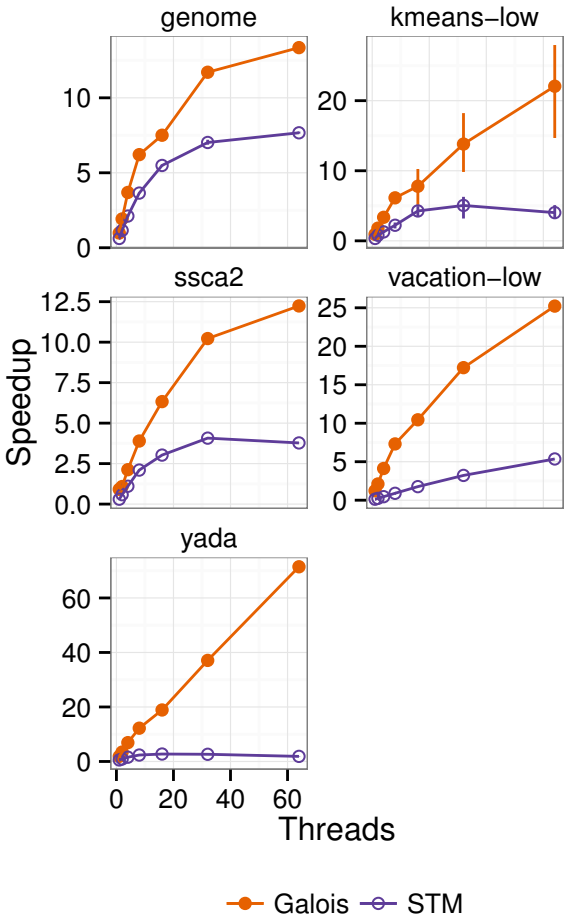
Sequential Program

```
fork()
while (begin != end)
  atomic
    begin += 1
  atomic
    while (head)
      ...
      head = next
    root.size += 1
join()
```

TM Program

Stampede

- STAMP [CaoMinh08]
  - Set of real-world applications using TM
  - Historically, performance has been poor
- Stampede benchmarks
  - Re-implemented STAMP in Galois
  - Compared to TinySTM
  - Median improvement ratio over STAMP of 4.4X at 64 threads (max: 37X, geomean: 3.9X)



- Blue Gene/Q
- Median of 5 runs

Nguyen and Pingali. What scalable transactional programs need from transactional memory. In submission to PLDI 2015.

Stampede Performance

```
fork()
while (begin != end)
  atomic
    begin += 1
  atomic
    while (head)
      ...
      head = next
    root.size += 1
join()
```

TM Program

Stampede Performance

- Controlling communication is paramount
  - Chief performance cost is moving bits around
- Scalable principle 1: disjoint access
  - Recall implementation of Bag and OrderedByIntegerMetric scheduler
- Scalable principle 2: virtualize tasks
  - Be as flexible as possible to dynamic conditions
- These changes are beyond capability of TM
  - Use Galois data structures (disjoint access)
  - Use Galois scheduler (virtualized tasks)

```
fork()
while (begin != end)
  atomic
    begin += 1
  atomic
    while (head)
      ...
      head = next
    root.size += 1
join()
```

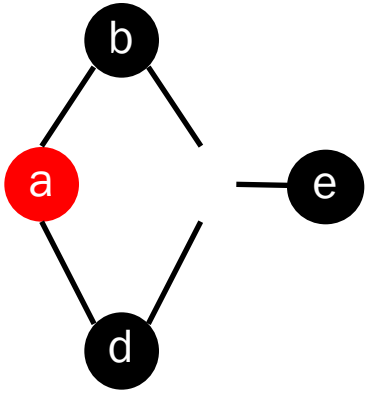
TM Program

Outline

- Parallel Program = Operator + Schedule + Parallel Data Structure
- Galois system implementation
  - Operator, data structures, scheduling
- Galois studies
  - Intel study
  - Deterministic scheduling
  - Adding a scheduler: sparse tiling
  - Comparison with transactional memory
  - Implementing other programming models

DSLs

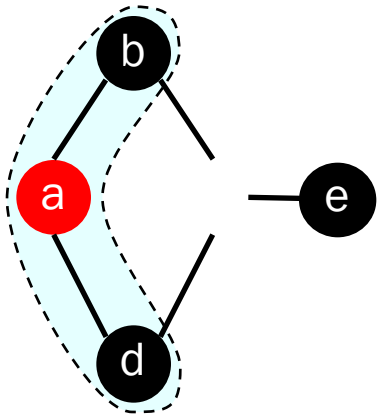
- Layer other DSLs on top of Galois
  - Other programming models are more restrictive than operator formulation
- Bulk-Synchronous Vertex Programs
  - Nodes are active
  - Operator is over node and its neighbors
  - Execution is in rounds
    - Read values taken from previous round
    - Overlapping writes reduced to single value
  - Ex: Pregel [Malewicz10], Ligra [Shun13]



- Gather-Apply-Scatter
  - Refinement of bulk-synchronous into subrounds
  - Ex: GraphLab [Low10] / PowerGraph [Gonzalez12]

DSLs

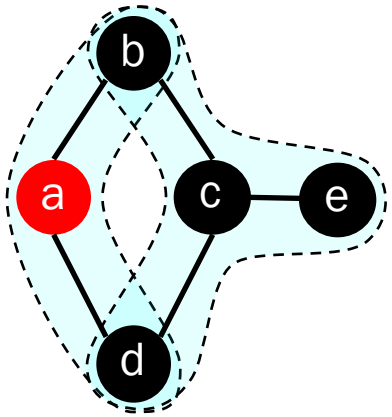
- Layer other DSLs on top of Galois
  - Other programming models are more restrictive than operator formulation
- Bulk-Synchronous Vertex Programs
  - Nodes are active
  - Operator is over node and its neighbors
  - Execution is in rounds
    - Read values taken from previous round
    - Overlapping writes reduced to single value
  - Ex: Pregel [Malewicz10], Ligra [Shun13]



- Gather-Apply-Scatter
  - Refinement of bulk-synchronous into subrounds
  - Ex: GraphLab [Low10] / PowerGraph [Gonzalez12]

DSLs

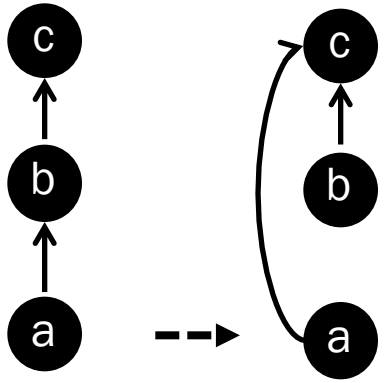
- Layer other DSLs on top of Galois
  - Other programming models are more restrictive than operator formulation
- Bulk-Synchronous Vertex Programs
  - Nodes are active
  - Operator is over node and its neighbors
  - Execution is in rounds
    - Read values taken from previous round
    - Overlapping writes reduced to single value
  - Ex: Pregel [Malewicz10], Ligra [Shun13]



- Gather-Apply-Scatter
  - Refinement of bulk-synchronous into subrounds
  - Ex: GraphLab [Low10] / PowerGraph [Gonzalez12]

Insufficiency of Vertex Programs

- Connected components problem
  - Construct a labeling function such that color(*x*) = color(neighbor(*x*))
- Algorithms
  - Union-Find-based
  - Label propagation
  - Hybrids



Pointer Jumping

Cost to Implement Other Models

Feature	LoC
GraphLab [Low10]	0
PowerGraph [Gonzalez12] (synchronous engine)	200
PowerGraph (asynchronous engine)	200
GraphChi [Kyrola12]	400
Ligra [Shun13]	300
GraphChi + Ligra (additional)	100

Cost to Implement Other Models

Feature	LoC
GraphLab [Low10]	0
PowerGraph [Gonzalez12] (synchronous engine)	200
PowerGraph (asynchronous engine)	200
GraphChi [Kyrola12]	400
Ligra [Shun13]	300
GraphChi + Ligra (additional)	100

PowerGraph-g

Comparison with Other Models

Applications

- Breadth-first search (bfs)
- Connected components (cc)
- Approximate diameter (dia)
- PageRank (pr)
- Single-source shortest paths (sssp)

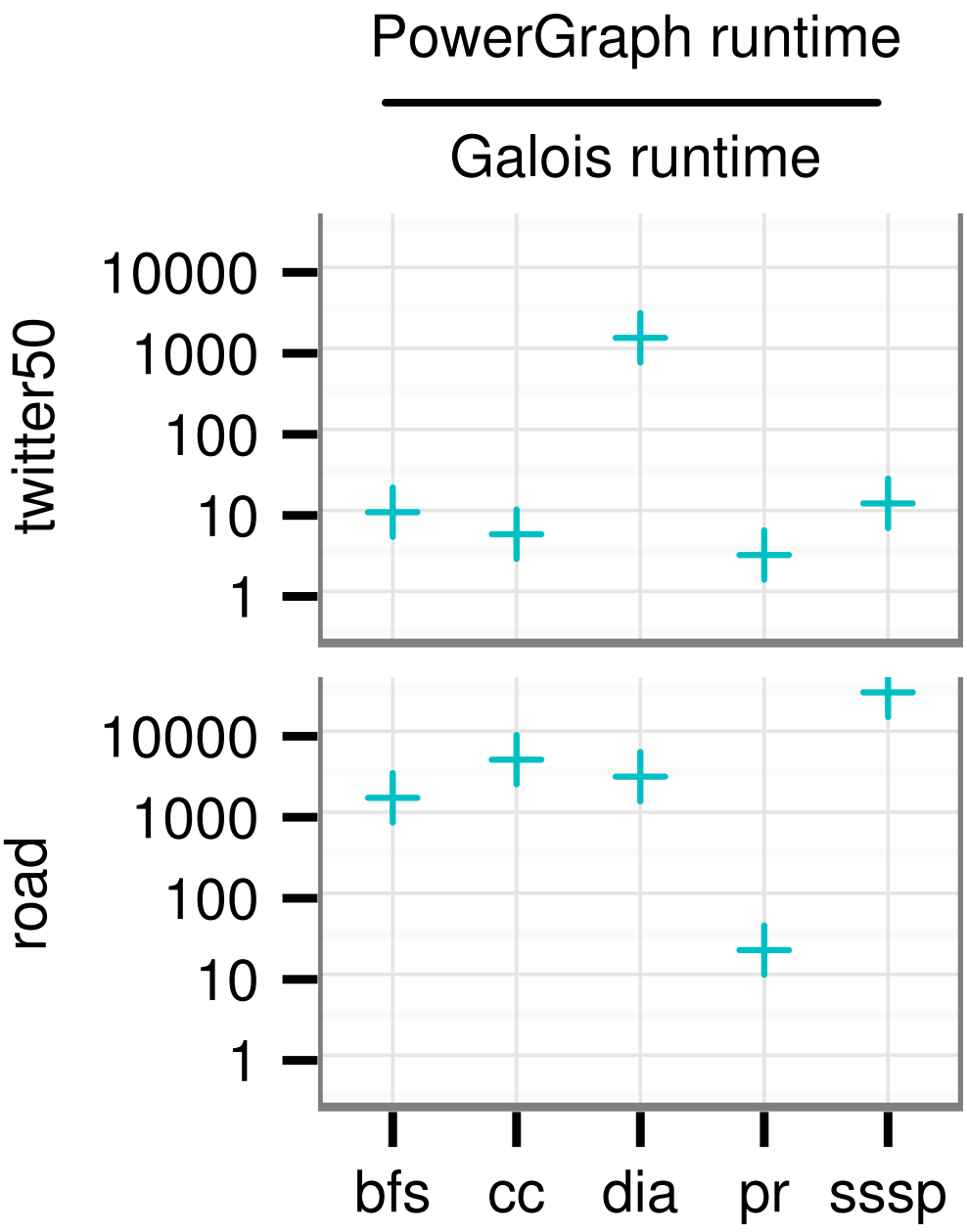
Inputs

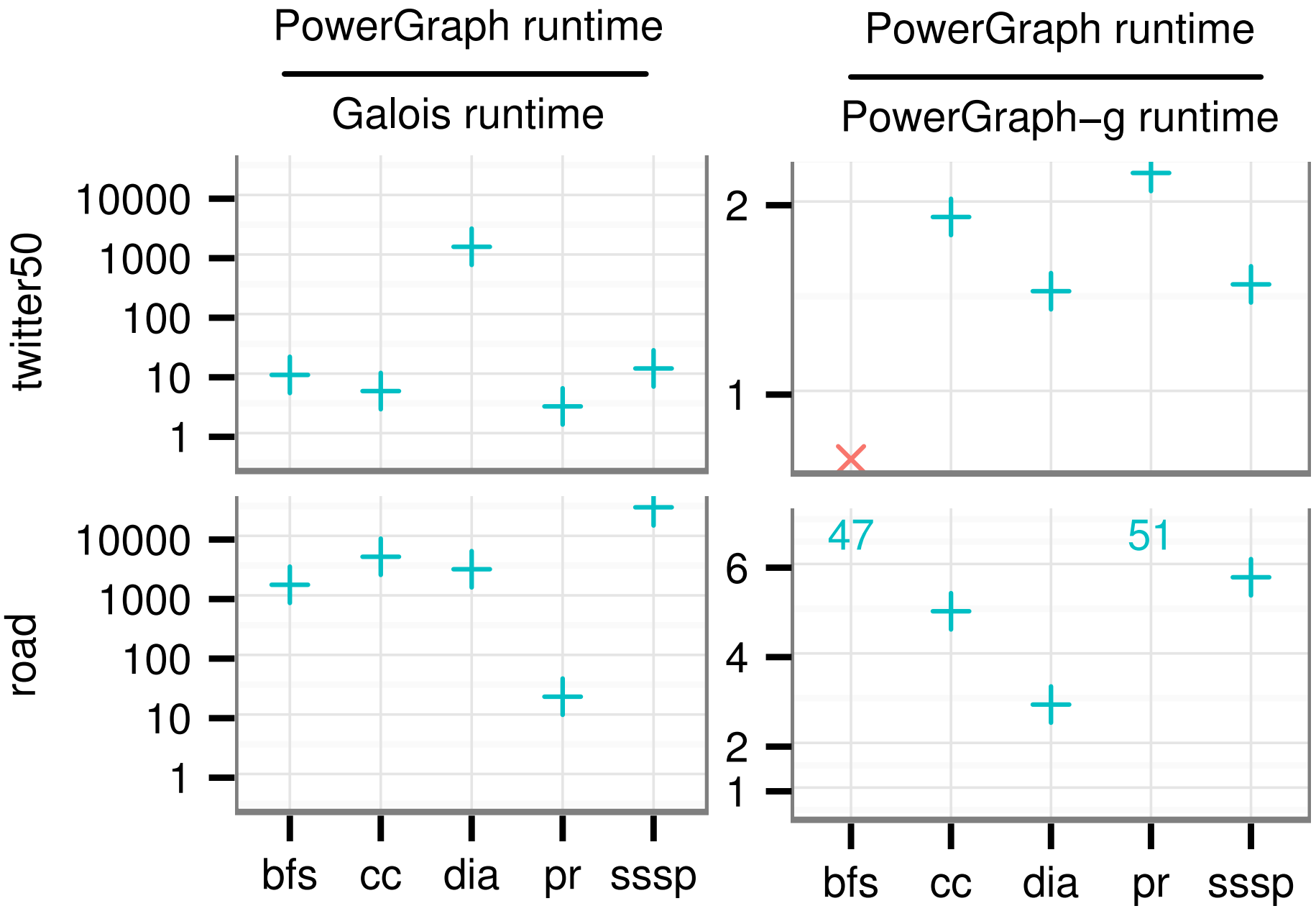
- twitter50 (50 M nodes, 2 B edges, low-diameter)
- road (20 M nodes, 60 M edges, high-diameter)

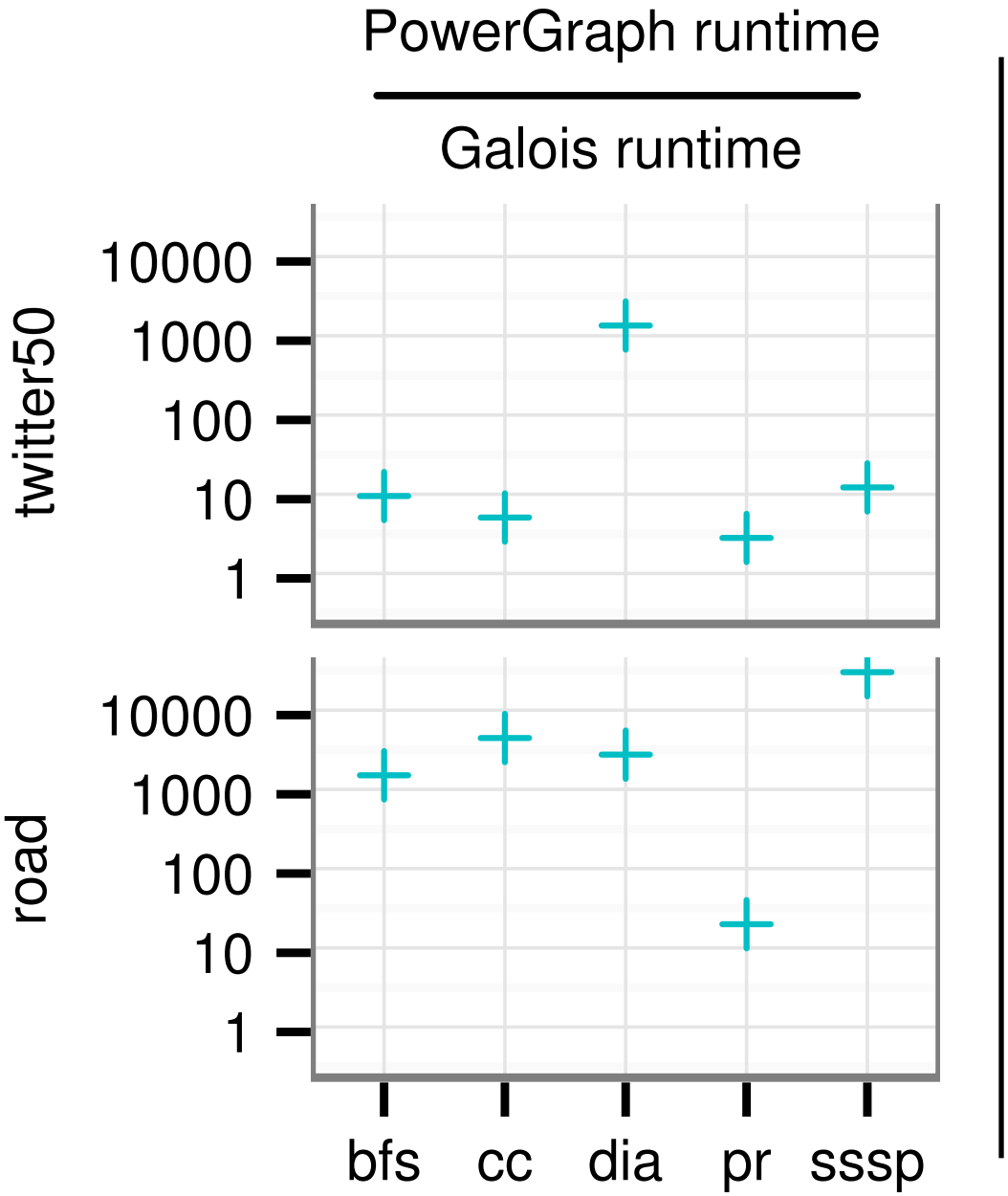
Comparisons

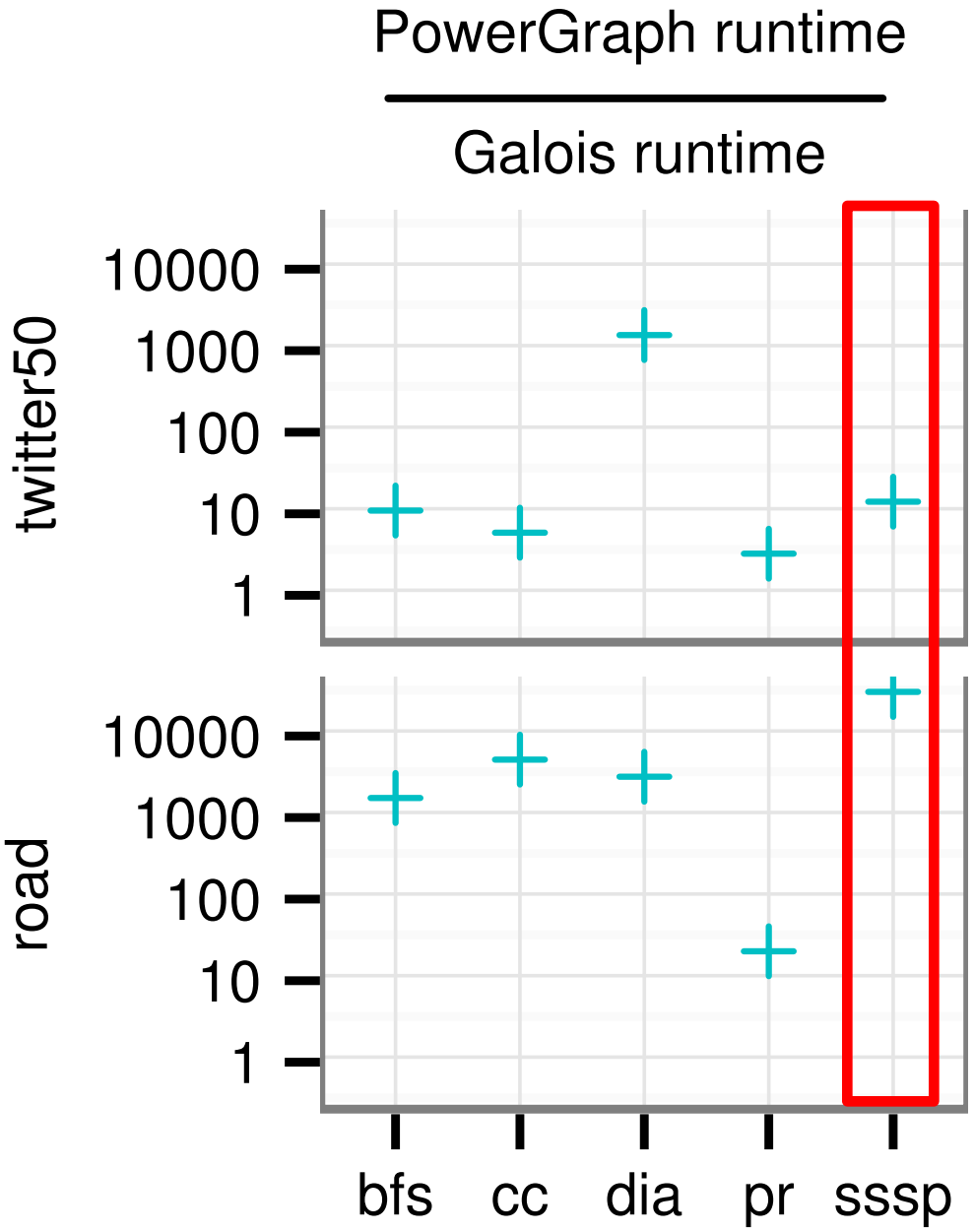
- PowerGraph (distributed)
  - Runtimes with 64 16-core machines (1024 cores) does not beat one 40-core machine using Galois

Nguyen, Lenharth and Pingali. A lightweight infrastructure for graph analytics. SOSP 2013.

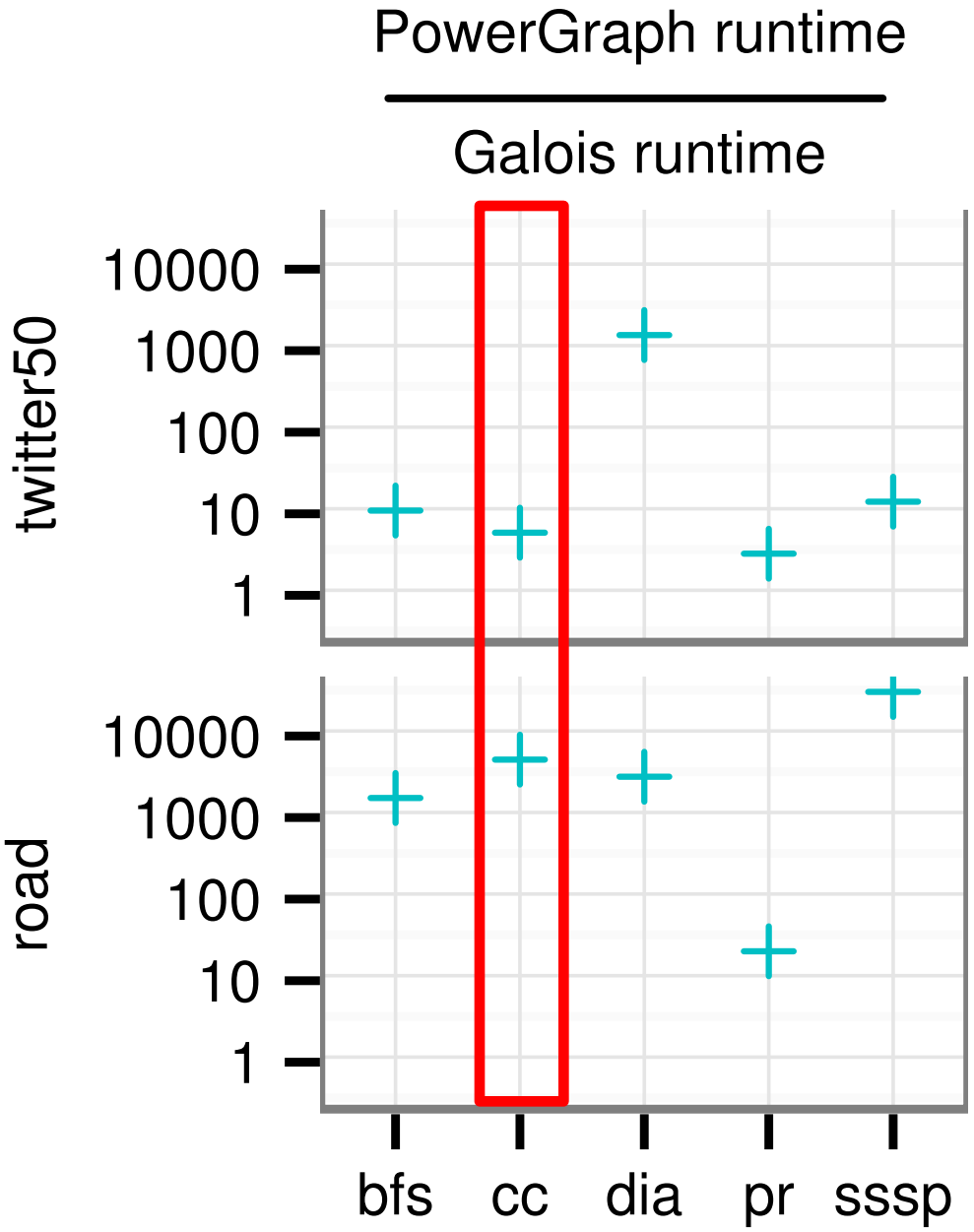

PowerGraph runtime
Galois runtime
twitter50
road
10000
1000
100
10
1
bfs
cc
dia
pr
sssp


PowerGraph runtime
Galois runtime
PowerGraph runtime
PowerGraph–g runtime
twitter50
road
10000
1000
100
10
1
2
1
6
4
2
1
47
51
bfs
cc
dia
pr
sssp

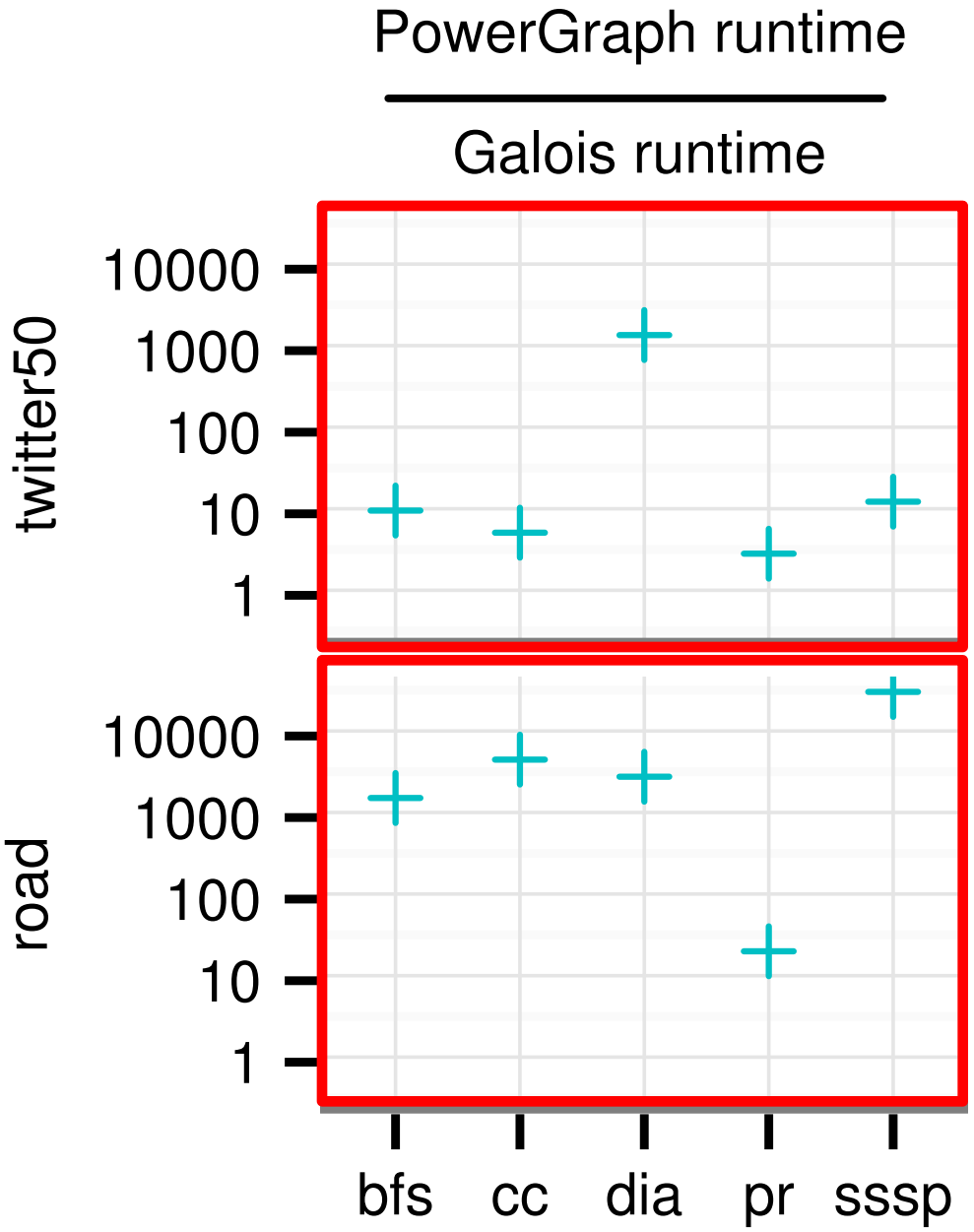

PowerGraph runtime
Galois runtime
twitter50
road
10000
1000
100
10
1
bfs
cc
dia
pr
sssp



- The best algorithm may require application-specific scheduling
  - Priority scheduling for SSSP
- The best algorithm may not be expressible as a vertex program
  - Connected components with union-find
- Speculative execution required for high-diameter graphs
  - Bulk-synchronous scheduling uses many rounds and has too much overhead



- The best algorithm may require application-specific scheduling
  - Priority scheduling for SSSP
- The best algorithm may not be expressible as a vertex program
  - Connected components with union-find
- Speculative execution required for high-diameter graphs
  - Bulk-synchronous scheduling uses many rounds and has too much overhead



- The best algorithm may require application-specific scheduling
  - Priority scheduling for SSSP
- The best algorithm may not be expressible as a vertex program
  - Connected components with union-find
- Speculative execution required for high-diameter graphs
  - Bulk-synchronous scheduling uses many rounds and has too much overhead

Outline

- Parallel Program = Operator + Schedule + Parallel Data Structure
- Galois system implementation
  - Operator, data structures, scheduling
- Galois studies
  - Intel study
  - Deterministic scheduling
  - Adding a scheduler: sparse tiling
  - Comparison with transactional memory
  - Implementing other programming models

Third-party Evaluations

"Galois performs better than other frameworks, and is close to native performance"

Nadathur et al. Navigating the maze of graph analytics frameworks. SIGMOD 2014.

"This paper demonstrates that speculative parallelization and the operator formalism, central to Galois' programming model and philosophy, *is the best choice* for irregular CAD algorithms that operate on graph-based data structures."

Moctar and Brisk. Parallel FPGA Routing based on the Operator Formulation. DAC 2014.