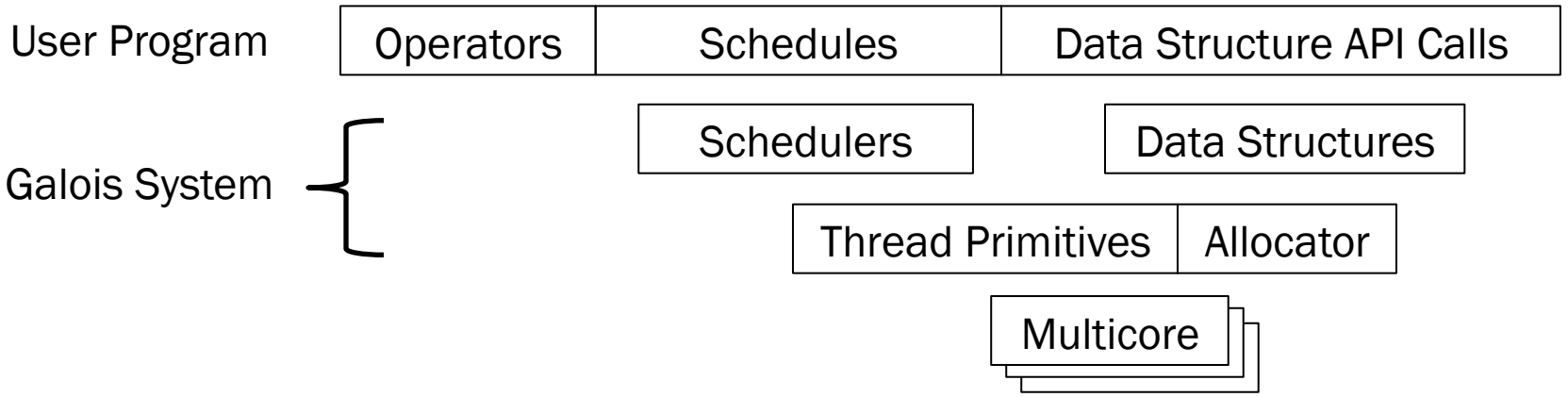
Galois, Practically

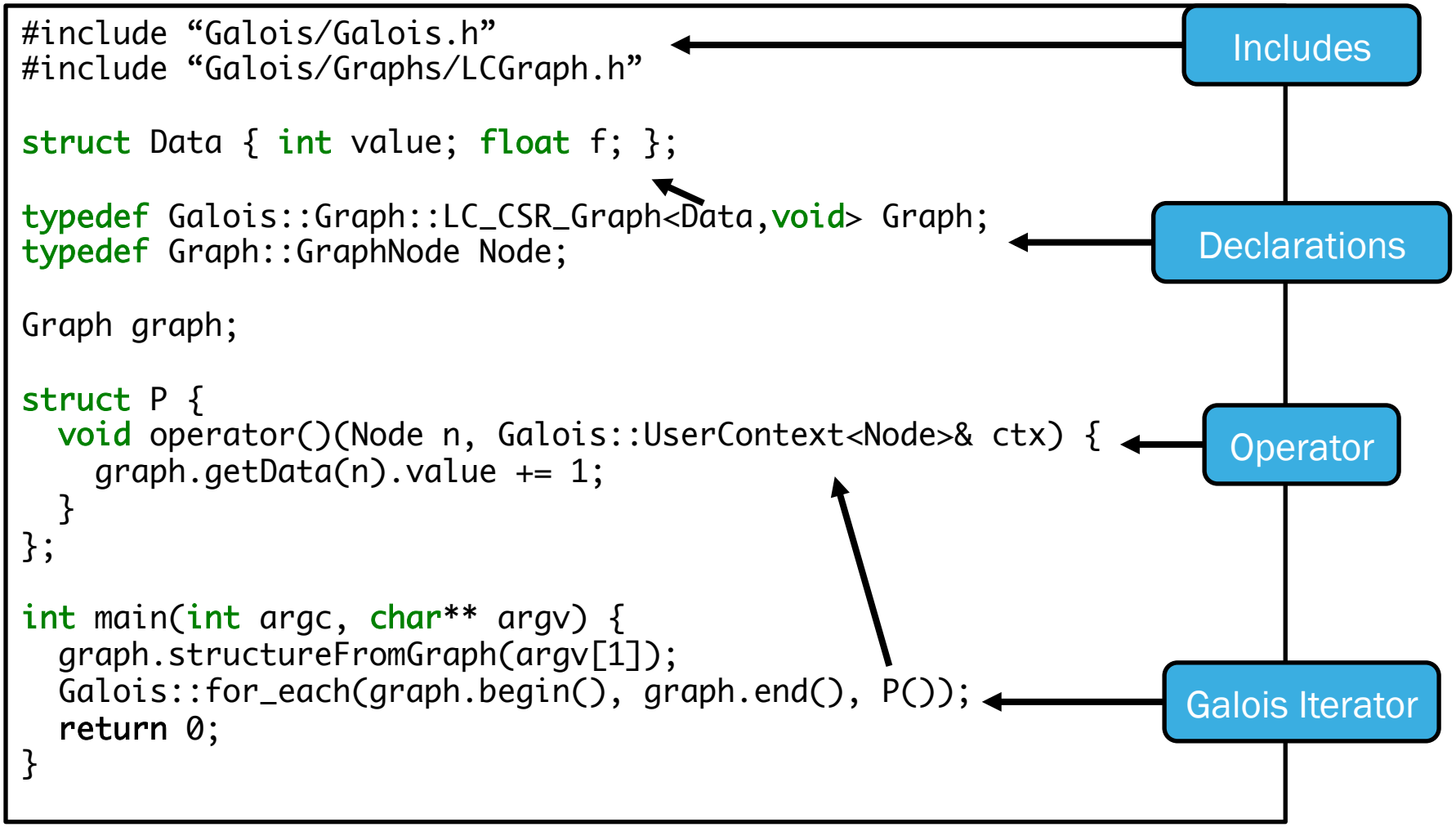
Andrew Lenharth

Galois System

Parallel Program = Operator + Schedule + Parallel Data Structure



A Very Short Galois Program



A Galois Program

- Operator
- Iterator
  - Topology-Driven
  - Data-Driven
- Data structures
  - Graphs, ...
- Scheduling
  - Priorities, ...
- Utility functions
  - Performance metrics

```
typedef Galois::Graph::LC_CSR_Graph<Data,void> Graph;

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

Galois::for_each(graph.begin(), graph.end(), P());
```

Operators

```
//Operators are any valid C++ functor with the correct signature
struct P {
  Graph& graph;
  P(Graph& g): graph(g) { }
  void operator()(Node n, Galois::UserContext<Node>& ctx) {
    graph.getData(n).value += 1;
  }
};
Galois::for_each(graph.begin(), graph.end(), P(graph));

//Or as a lambda
Galois::for_each(graph.begin(), graph.end(),
  [&graph] (Node n, Galois::UserContext<Node>& ctx) {
    graph.getData(n).value += 1;
});
```

Operator Context

```
void operator()(T n, Galois::UserContext<T>& ctx);

typedef … PerIterAllocTy;

template<typename T>
struct UserContext {
  // Add a new item to the worklist
  template<typename Args…>
  void push(Args&&… args);

  // Get per-iteration region allocator
  PerIterAllocTy& getPerIterAlloc();
};
```

Fast Local Memory

```
void operator()(Node n, Galois::UserContext<Node>& ctx) {
  //This vector uses scalable allocation
  typedef PerIterAllocTy::rebind<Node>::other Alloc;
  std::vector<Node,Alloc> v(ctx.getPerIterAlloc());

  auto& d = graph.getData(n).data;
  std::copy(d.begin(), d.end(), std::back_inserter(v));
}
```

A Galois Program

- Operator
- Iterator
  - Topology-Driven
  - Data-Driven
- Data structures
  - Graphs, …
- Scheduling
  - Priorities, …
- Utility functions
  - Performance metrics

```
typedef Galois::Graph::LC_CSR_Graph<Data,void> Graph;

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

Galois::for_each(graph.begin(), graph.end(), P());
```

Topology-driven Iteration

```
//Topology-driven iteration
Galois::for_each(graph.begin(), graph.end(),
    [&graph] (Node n, Galois::UserContext<Node>& ctx) {
      graph.getData(n).value = 0;
    });

//Topology-driven fixedpoint
while (!converged()) {
  //Apply op to each node in the graph
  Galois::for_each(graph.begin(), graph.end(), P(graph));
}
```

Data-driven Iteration

```
struct P {
  void operator()(int n, Galois::UserContext<int>& ctx) {
    if (n < 100) {
      ctx.push(n + 1);
      ctx.push(n + 2);
    }
  }
};

//for_each has overload for a single work item
//1 is the initial work item
//Yes, you can work on abstract iteration spaces
Galois::for_each(1, P());
```

A Galois Program

- Operator
- Iterator
  - Topology-Driven
  - Data-Driven
- Data structures
  - Graphs, ...
- Scheduling
  - Priorities, ...
- Utility functions
  - Performance metrics

```
typedef Galois::Graph::LC_CSR_Graph<Data,void> Graph;

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

Galois::for_each(graph.begin(), graph.end(), P());
```

Data Structures

- Graphs
  - In namespace Galois::Graph
  - In include/Galois/Graph/*
  - General Graph: FirstGraph.h
- Specialized graphs: LC_*.h
  - No edge/node creation/removal
  - Variants for different memory layouts
  - Except LC_Morph_Graph: allows new nodes with declared number of edges
- Others: Trees, Bags, Reducers

LC_CSR_Graph

- Local Computation, Compressed Sparse Row

```
template<typename NodeData, typename EdgeData>
struct LC_CSR_Graph {
  typedef … GraphNode;
  typedef … edge_iterator;
  typedef … iterator;

  iterator begin();
  iterator end();
  edge_iterator edge_begin(GraphNode);
  edge_iterator edge_end(GraphNode);
  NodeData& getData(GraphNode);
  EdgeData& getEdgeData(edge_iterator);
  GraphNode getEdgeDst(edge_iterator);
};
```

LC_CSR_Graph Example

```
//Sum values on edges and nodes
typedef LC_CSR_Graph<double, double> Graph;
typedef Graph::iterator iterator;
typedef Graph::edge_iterator edge_iterator;

Graph g;
Galois::Graph::readGraph(graph, filename);
for (iterator ii = g.begin(),
              ei = g.end(); ii != ei; ++ii) {
  double sum = g.getData(*ii);
  for (edge_iterator jj = g.edge_begin(*ii),
                     ej = g.edge_end(*ii);
       jj != ej; ++jj) {
    sum += g.getEdgeData(jj);
  }
}

//C++11
for (auto n : g) {
  double sum = g.getData(n);
  for (auto edge : g.out_edges(n)) {
    sum += graph.getEdgeData(edge);
  }
}
```

A Galois Program

- Operator
- Iterator
  - Topology-Driven
  - Data-Driven
- Data structures
  - Graphs, ...
- Scheduling
  - Priorities, ...
- Utility functions
  - Performance metrics

```
typedef Galois::Graph::LC_CSR_Graph<Data,void> Graph;

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

Galois::for_each(graph.begin(), graph.end(), P());
```

Scheduling

- Abstractly, iterations of for_each loop are placed in an unordered collection of tasks
- Often, programmers do not need to worry about the scheduling of tasks to threads
- But, more explicit control is available through scheduling interface

Scheduling

- Various scheduling policies available
  - In namespace Galois::WorkList
  - In include/Galois/WorkList/*

```
template<…> struct LIFO {};
template<…> struct FIFO {};
template<int ChunkSize, …> struct ChunkedLIFO {};
template<int ChunkSize, …> struct dChunkedLIFO {};
template<int ChunkSize, …> struct AltChunkedLIFO {};

template<…> struct StableIterator {};

template<…> struct BulkSynchronous {};

template<typename GlobalWL, typename LocalWL, …>
  struct LocalQueue {};

template<typename Indexer, typename WL, …>
  struct OrderedByIntegerMetric {};
```

Using Schedulers

```
typedef Galois::WorkList::dChunkedLIFO<256> WL;

Galois::for_each(g.begin(), g.end(), P(), Galois::wl<WL>());
```

A Galois Program

- Operator
- Iterator
  - Topology-Driven
  - Data-Driven
- Data structures
  - Graphs, ...
- Scheduling
  - Priorities, ...
- Utility functions
  - Performance metrics

```
typedef Galois::Graph::LC_CSR_Graph<Data,void> Graph;

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

Galois::for_each(graph.begin(), graph.end(), P());
```

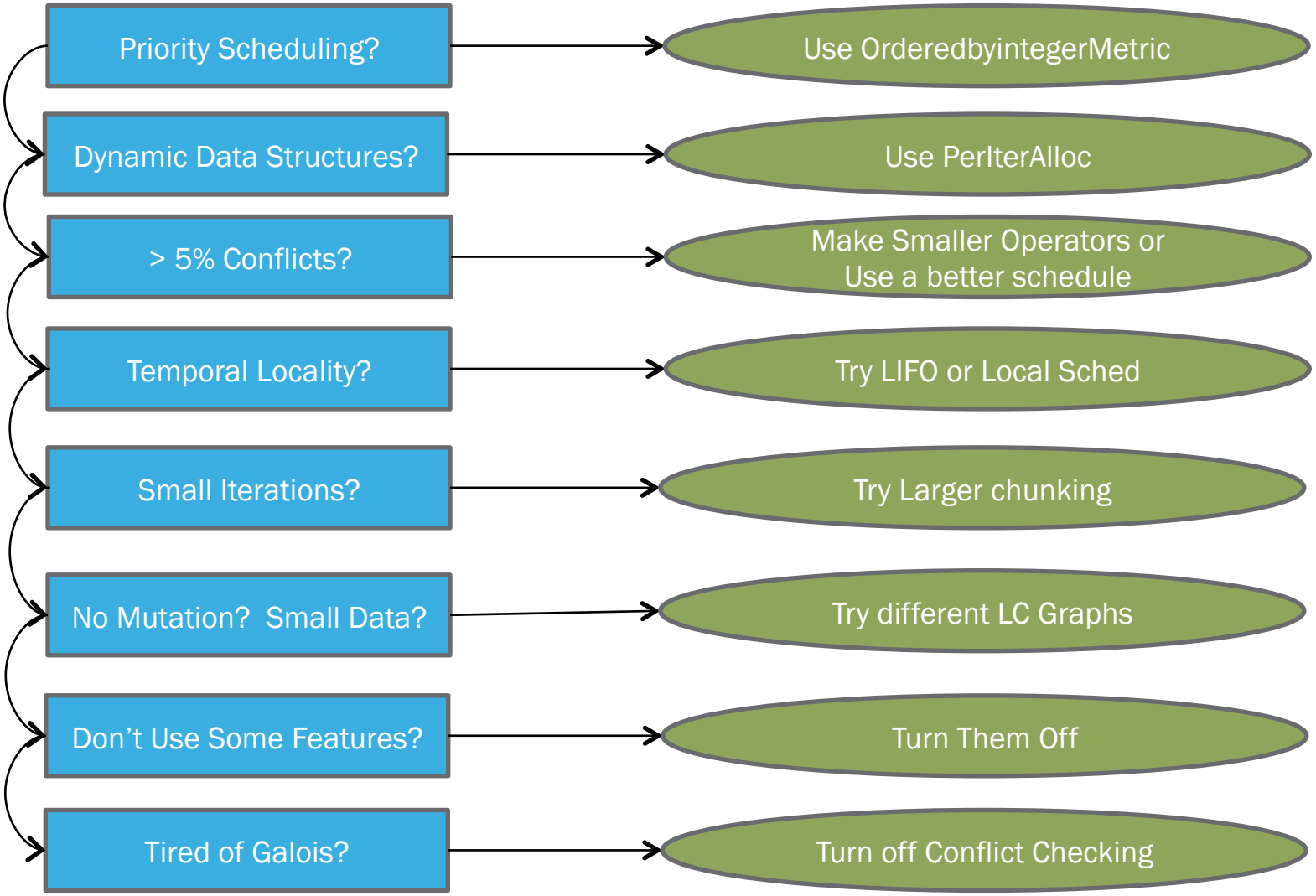
Performance Metrics

```
#include “Galois/Galois.h”
#include “Galois/Statistics.h”
#include <iostream>

int main(int argc, char** argv) {
  Galois::StatManager stats;

  //Set number of threads
  Galois::setActiveThreads(4);

  //Report statistics by loop name
  Galois::for_each(…, Galois::loopname(“MyLoop”));

  //Insert own timers
  Galois::StatTimer timer(“Phase2”);
  timer.start();
  …
  timer.stop();
  std::cout << “Phase 2 took “ << timer.get() << “ milliseconds\n”;

  //Report on memory activity
  Galois::reportPageAlloc(“AfterPhase2”);

  return 0;
}
```

Tuning Galois Programs

Andrew Lenharth

Tuning a Galois Program



Priority Scheduling

- An algorithm prefers a particular order for algorithmic reasons, but is correct in any order
  - SSSP: Dijkstra vs. Chaotic Relaxation
- Use OrderedByIntegerMetric scheduler

```
struct Indexer { int operator()(GraphNode n); };

typedef Galois::WorkList::OrderedByIntegerMetric<Indexer> WL;

Galois::for_each(g.begin(), g.end(), P(), Galois::wl<WL>());
```

PerIterAlloc

- If you use local, dynamic data-structures in an iteration
  - E.g., keep track of a variable sized set
- Use PerIterAlloc as the backing allocator for your container
  - Fast and scalable
- Failure to do so WILL NOT SCALE

```
void operator()(Node n, Galois::UserContext<Node>& ctx) {
  //This vector uses scalable allocation
  typedef PerIterAllocTy::rebind<Node>::other Alloc;
  std::vector<Node,Alloc> v(ctx.getPerIterAlloc());

  auto& d = graph.getData(n).data;
  std::copy(d.begin(), d.end(), std::back_inserter(v));
}
```

High Conflict (Abort) Rate

- High abort rates will hurt parallelism
  - Galois partially orders aborted work to ensure forward progress
- Option 1: rework operator to touch less data
  - E.g., in DT we replaced a (usually) short mesh walk with an acceleration tree
- Option 2: Schedule to keep threads apart
  - E.g., DMR starts threads at random locations in the mesh, but processes nearby items next (thus keeping threads apart)

Temporal Locality

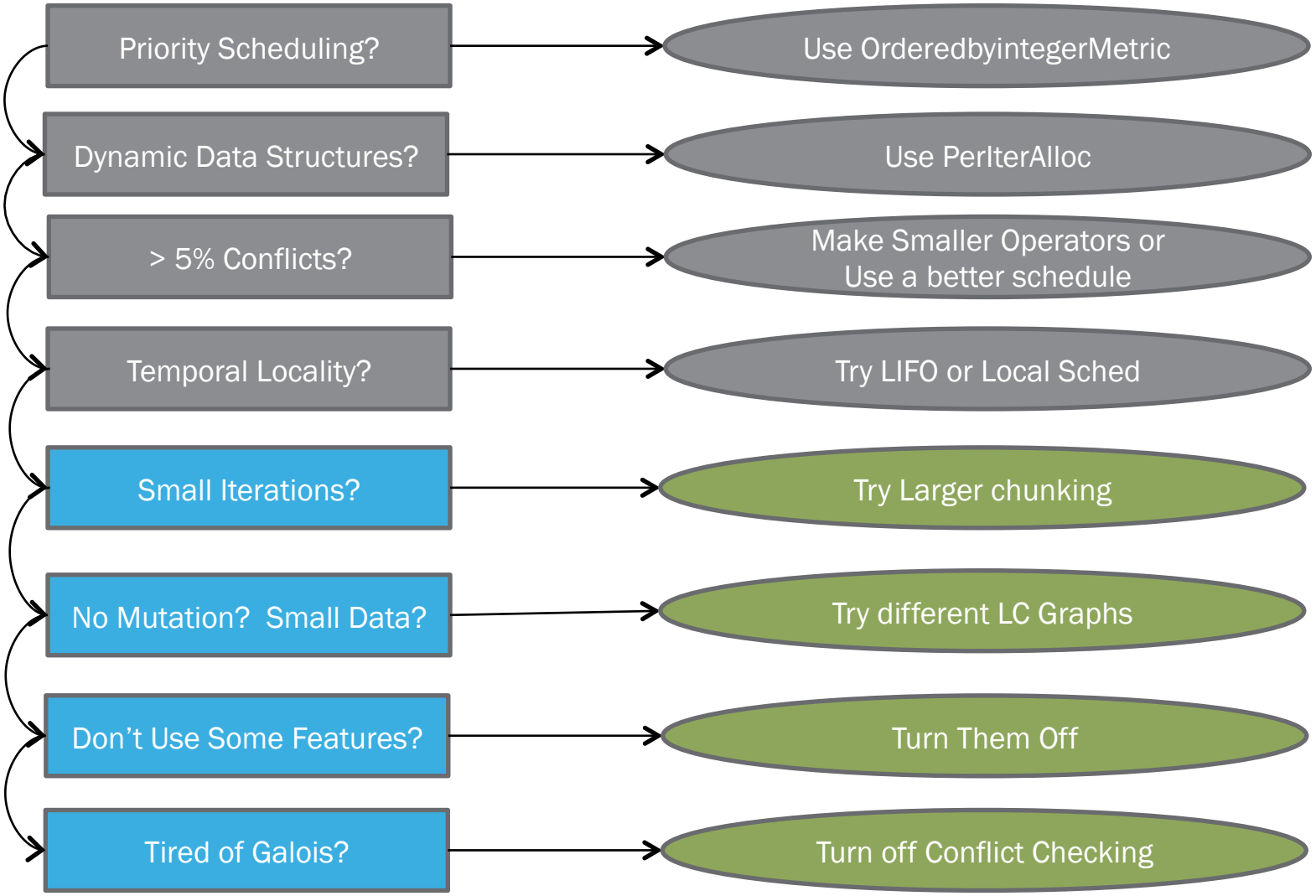
- Schedule new work first (LIFO-like schedule)
- Limit stealing of new work

Bonus: Spatial Locality!

- Use locality maintaining and preserving for_each
  - Supported by most Galois data structures
- Nodes are owned be each thread, process them on that thread
  - At least until load balance issues
- NUMA friendly (and any non-trivial machine is enough NUMA that this helps)

```
//Standard for_each
Galois::for_each(g.begin(), g.end(), P());

//Locality maintaining and preserving for_each
Galois::for_each_local(g, P());
```

Tuning a Galois Program



Scheduling Overhead

- If iterations are small and plentiful, use a larger chunking in the worklist to reduce communication and synchronization
- Large chunks hurt load balance
- Large chunks hurt how closely priority is followed in priority scheduling

```
typedef Galois::WorkList::dChunkedLIFO<256> LargeChunks;
typedef Galois::WorkList::dChunkedLIFO<4> SmallChunks;

Galois::for_each(…, Galois::wl<SmallChunks>());
```

Data Layout

- If graph structure is not being mutated in a loop, use an LC_* graph
- Many LC_* graphs exist with different data layouts, try them all

```
typedef Galois::Graph::LC_CSR_Graph<Data,void> Graph;

typedef Galois::Graph::LC_InlineEdge_Graph<Data,void> Graph;
typedef Galois::Graph::LC_InlineEdge_Graph<Data,void>
  ::with_compressed_node_ptr<true>::type Graph;

typedef Galois::Graph::LC_Linear_Graph<Data,void> Graph;
```

Feature Removal

- Features of the runtime can be disabled via type-traits on the operator or arguments to the for_each
  - Reduces size of runtime for that loop
  - See include/Galois/TypeTraits.h
- Disabling features:
  - Reduces code size
  - Reduces dynamic branches
  - Removes unnecessary runtime checks and overhead

```
struct P {
  typedef int tt_does_not_need_push;
  typedef int tt_does_not_need_aborts;
};

Galois::for_each(…, P());
```

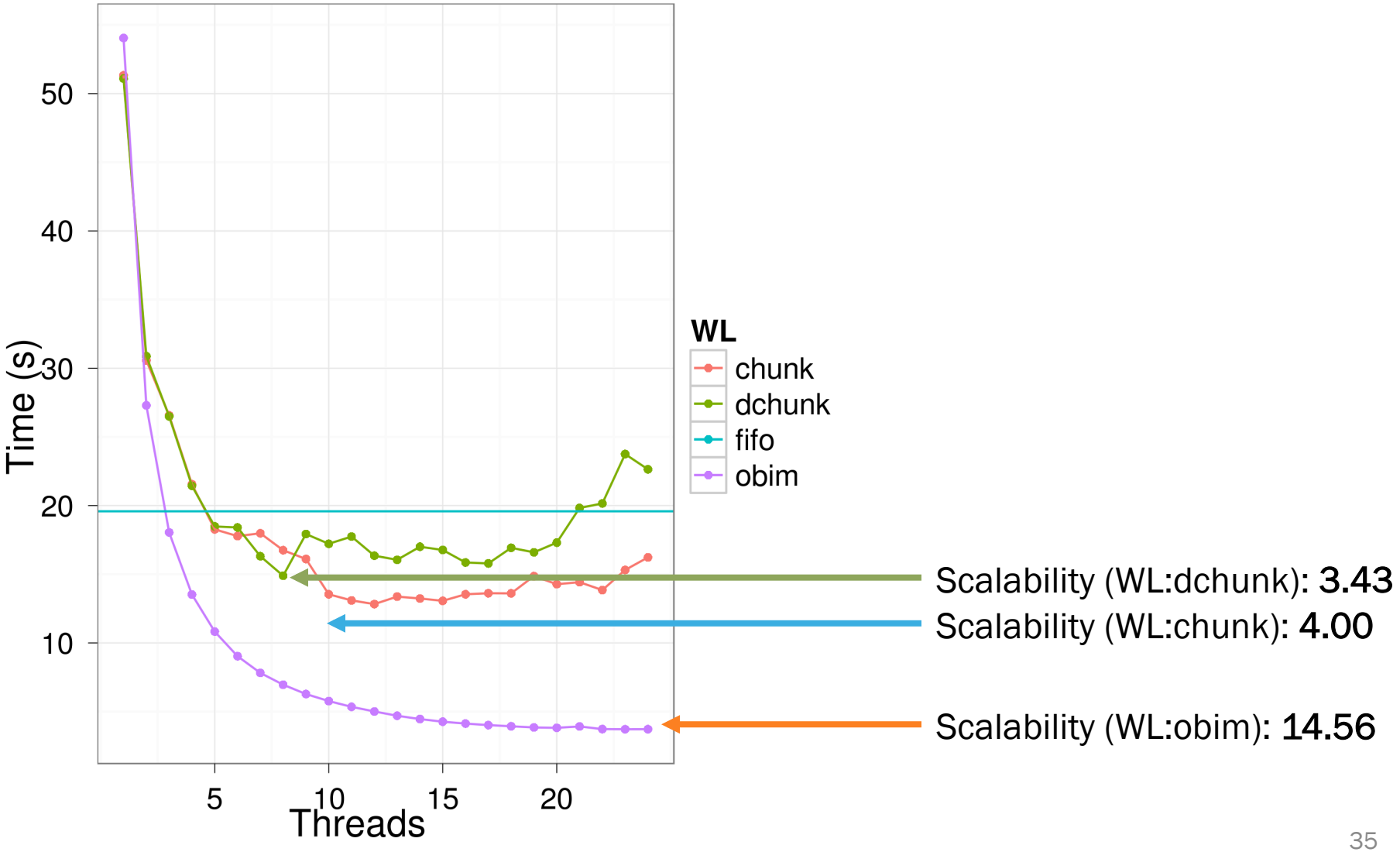
Flag Optimization

- Do you know better than Galois?
- Are you sure a data-race would be acceptable?
- You don't modify any hidden state?
- You can selectively disable conflict detection
  - E.g., in SSSP we update the node with a CAS, and use racy reads rather than lock neighborhoods

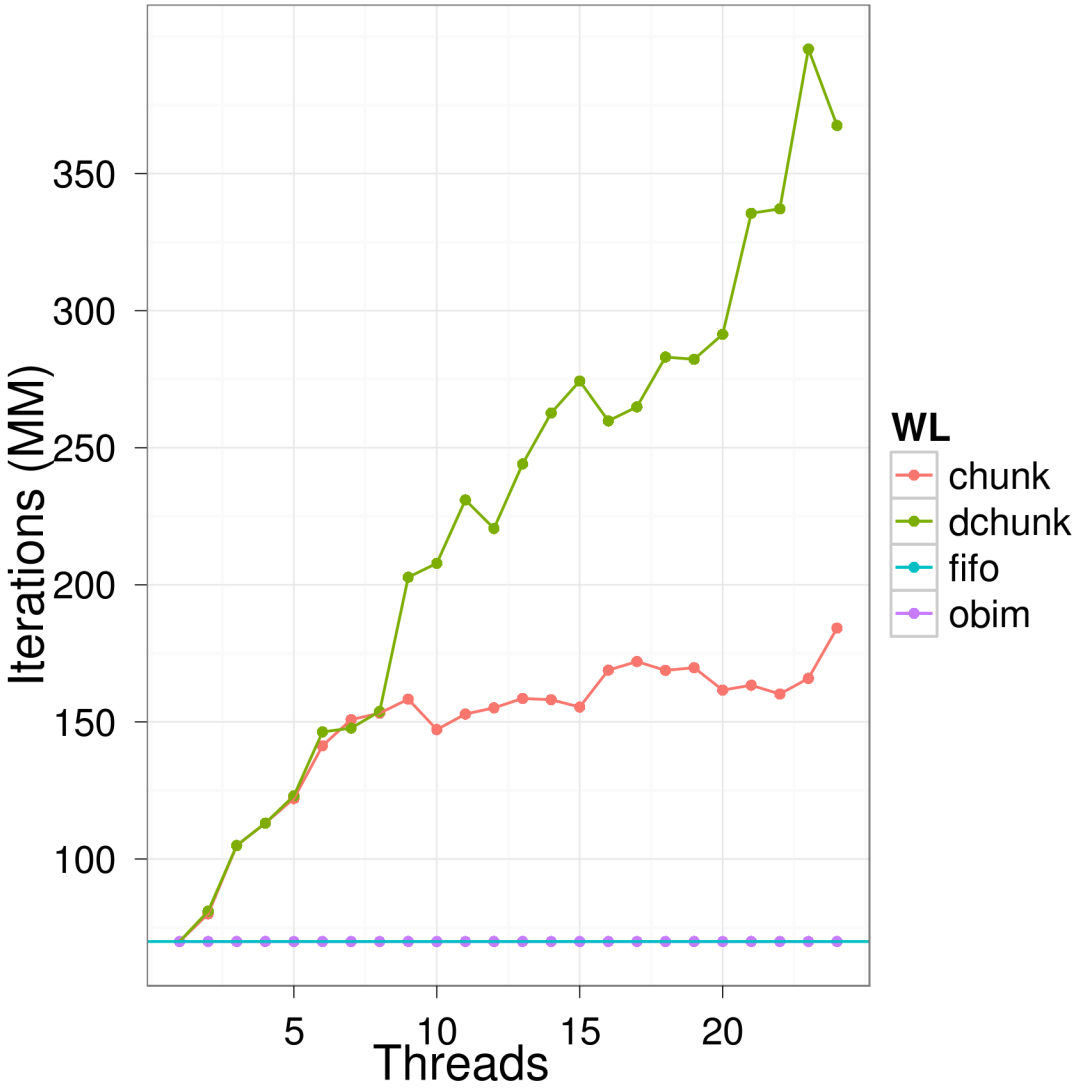
```
void relaxEdge(Node dst, int newDist) {
  Data& ddata = g.getData(dst, Galois::NONE));
  int oldDist;
  while (newDist < (oldDist = ddata.data)) {
    if (CAS(ddata.dist, oldDist, newDist)) {
      // updated to new dist
    }
  }
}
```

What's the Practical Impact?

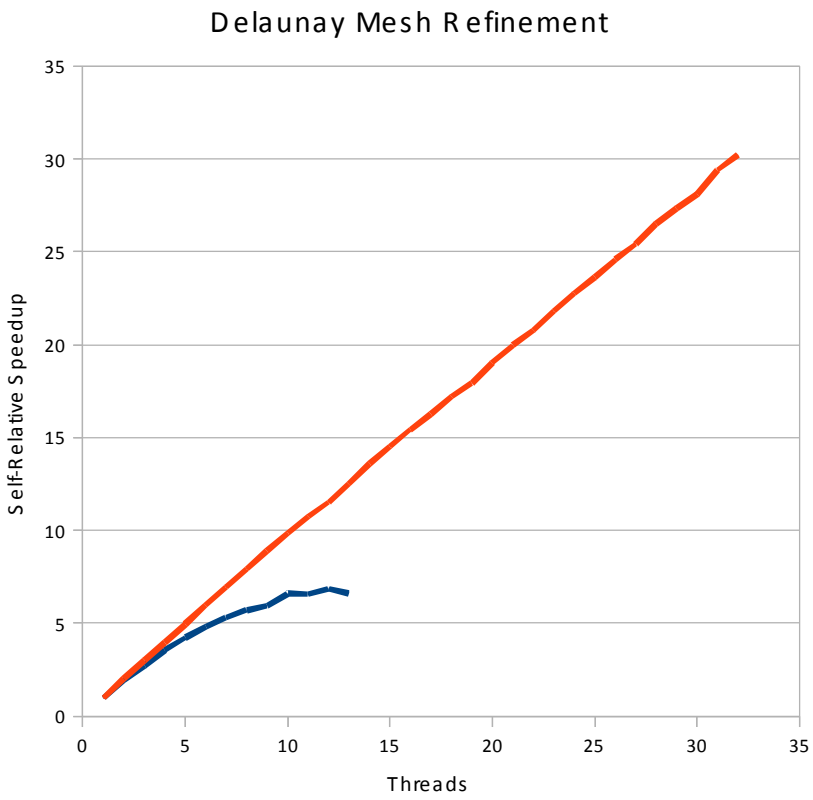
Varying Schedulers for SSSP

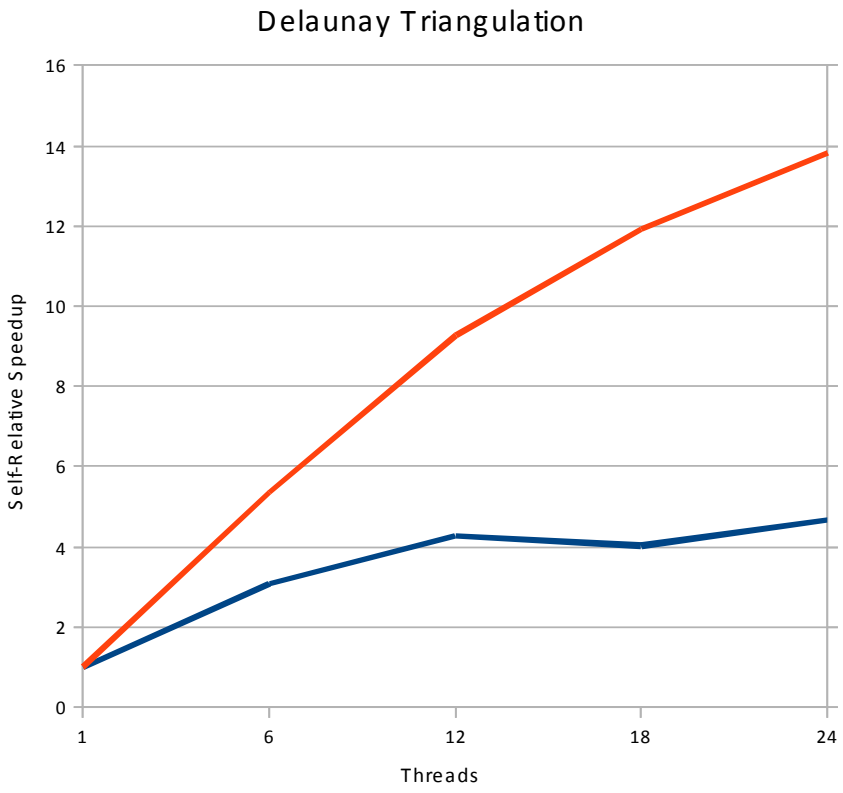


SSSP: Amount of Work



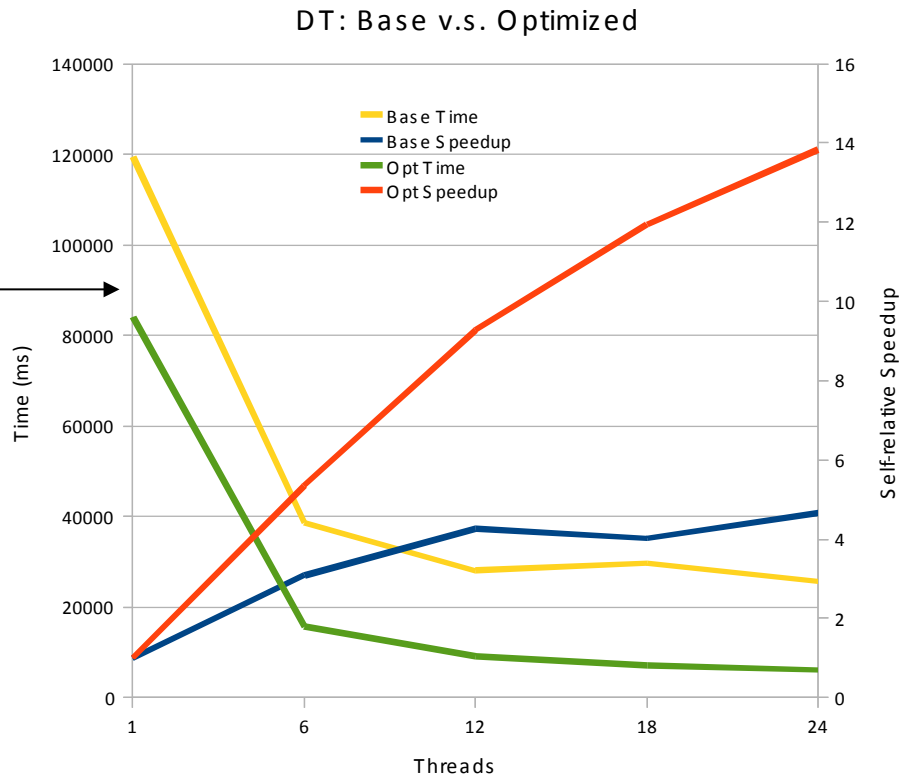
DMR and DT


Delaunay Mesh Refinement
Self-Relative Speedup
Threads
0
5
10
15
20
25
30
35


Delaunay Triangulation
Self-Relative Speedup
Threads
0
2
4
6
8
10
12
14
16
1
6
12
18
24

DT

~30% faster serial code too



Implementing Stochastic Gradient Descent in Galois

Donald Nguyen

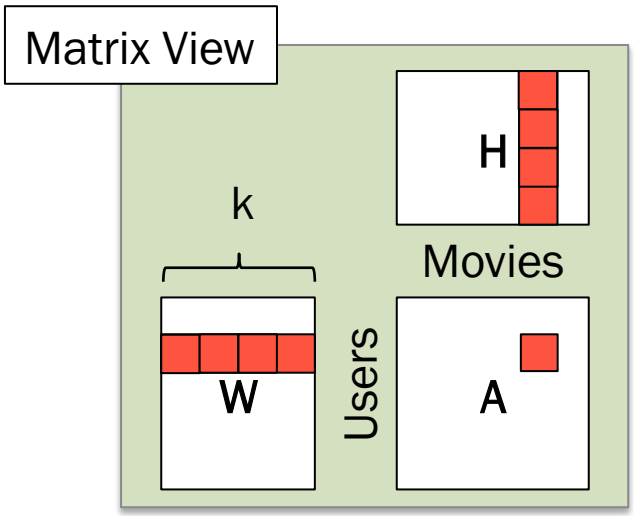
Recommender Systems

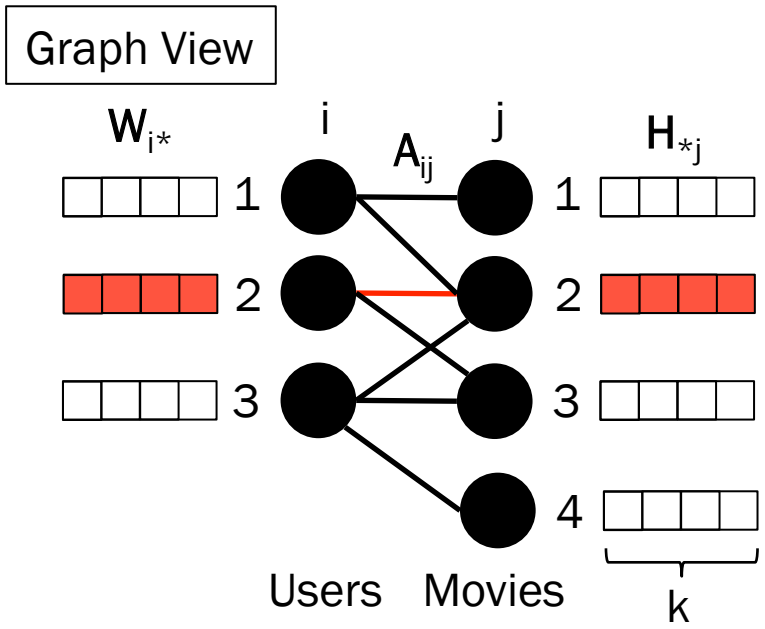
- Input
  - Set of users
  - Set of items (e.g., movies, songs, …)
  - Subset of ratings or (user, item, value) tuples

- Output
  - Predicted values for unseen ratings

Matrix Completion Problem

- Given a partially observed m×n matrix **A**, predict the unobserved entries
- As optimization problem
  - Find m×k matrix **W** and k×n matrix **H** (k << min(m,n)) such that **A** ≈ **WH**
- Algorithms
  - Stochastic gradient descent (SGD)
  - Coordinate descent
  - Alternating least squares





Gradient Descent

- Given
  - Φ: parameters (e.g., W, H)
  - $L(\Phi)$: loss function (objective function)
- Iteratively take small steps ε in direction of negative gradient $L'(\Phi)$ of loss function

$$\Phi(n+1) = \Phi(n) - \varepsilon(n)\, L'(\Phi)$$

- Stochastic GD
  - Update parameters individually for each row or column (based on noisy estimate of Φ)

Naïve SGD Implementation

- Sequential row-wise operator

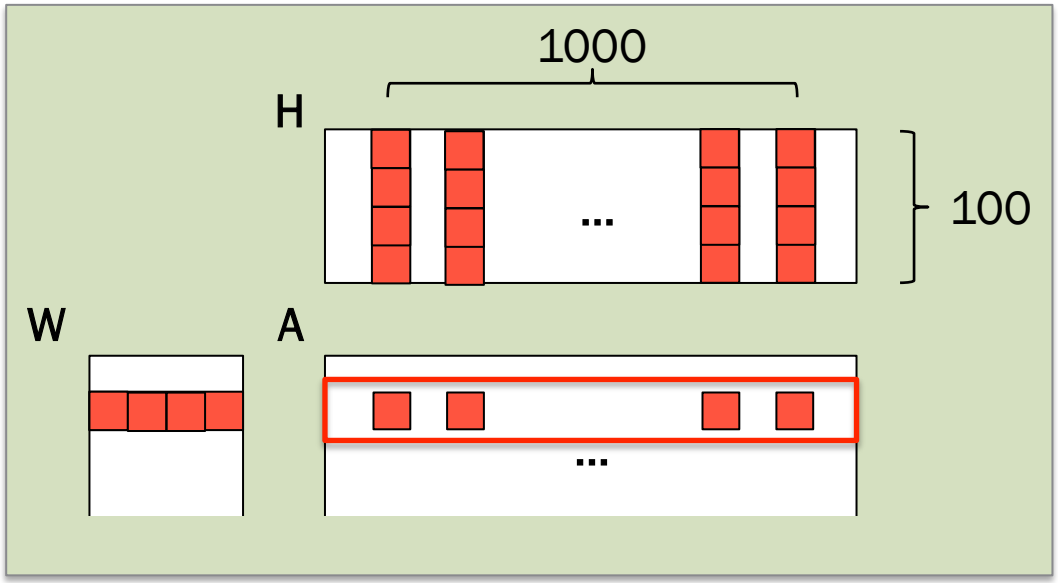
```
for (User u : users)
  for (Item m : items(u))
    op(u, m, A_ij);
```

- In Galois

```
Galois::for_each(users.begin(), users.end(),
  [&](Node u, Galois::UserContext<Node>& ctx) {
    for (auto e : g.out_edges(u))
      op(u, g.getEdgeDst(e), g.getEdgeData(e));
  }
);
```

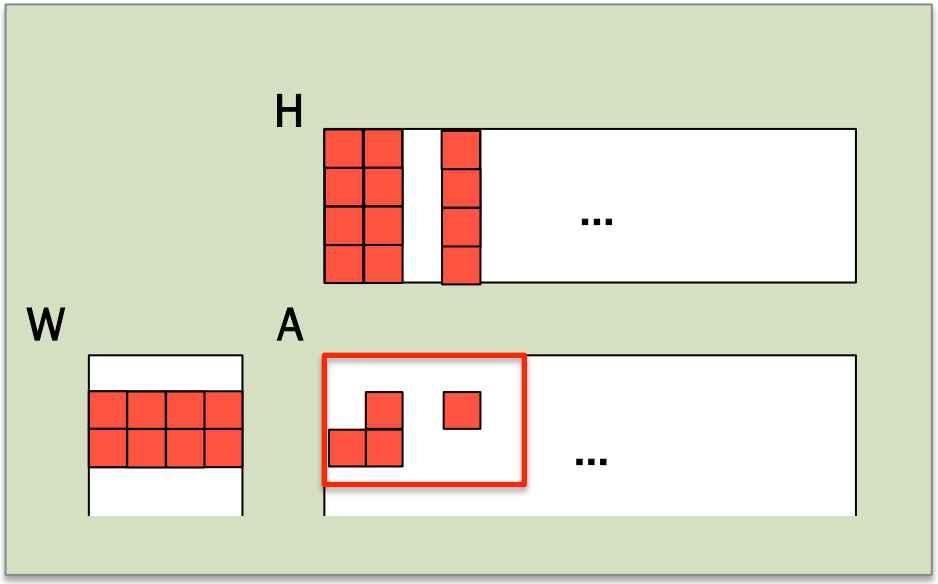
Shortcomings of Naïve Implementation

- Poor cache behavior
  - With k = 100 and double-precision floats, $W_{i*} \approx$ 1KB
  - Number of ratings for user or item can be large (e.g., scale-free), 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

2D Tiling in Galois (Experimental)

```
Galois::Exp::for_each_2d(
  users.begin(), users.end(),
  items.begin(), items.end(),
  blockSizeX, blockSizeY,
  [&](Node u, Node m, edge_iterator edge) {
    op(u, m, g.getEdgeData(edge));
  }
);
```

Evaluation

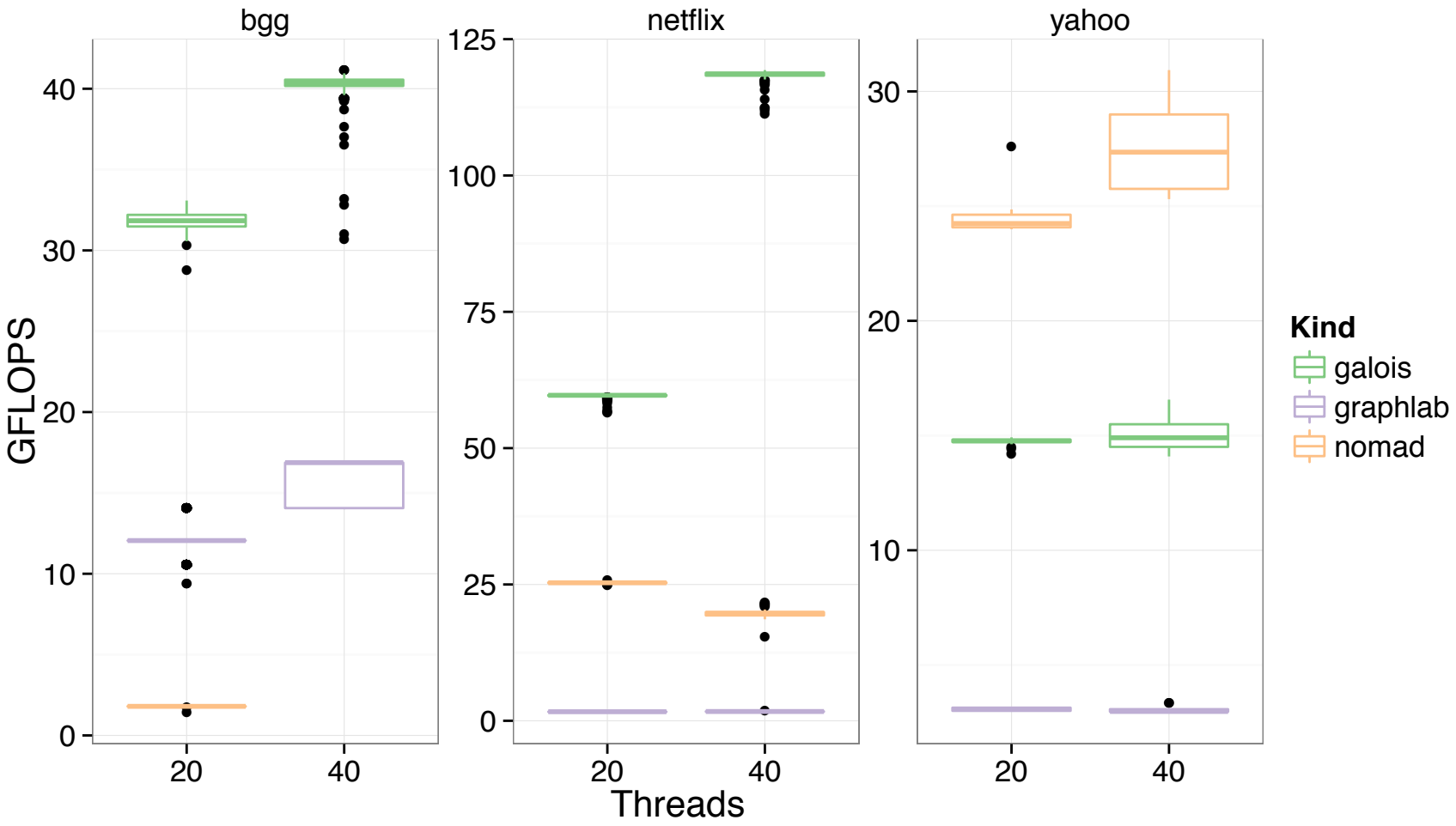
- Implementations
  - Galois
    - 2D scheduling
    - Synchronized updates
  - GraphLab
    - Standard GraphLab only supports GD
    - Implement 1D SGD with unsynchronized updates
  - Nomad
    - From scratch distributed code
    - 2D scheduling
      - Tile size is function of #threads
    - Synchronized updates
- Machine
  - 4 x 10 core (Xeon E7-4860) "Westmere"
  - 2.27 GHz

- Datasets

	Items	Users	Ratings	Sparsity
bgg	47K	109K	6M	1e-3
netflix	17K	480K	99M	1e-2
yahoo	624K	1M	253M	4e-4

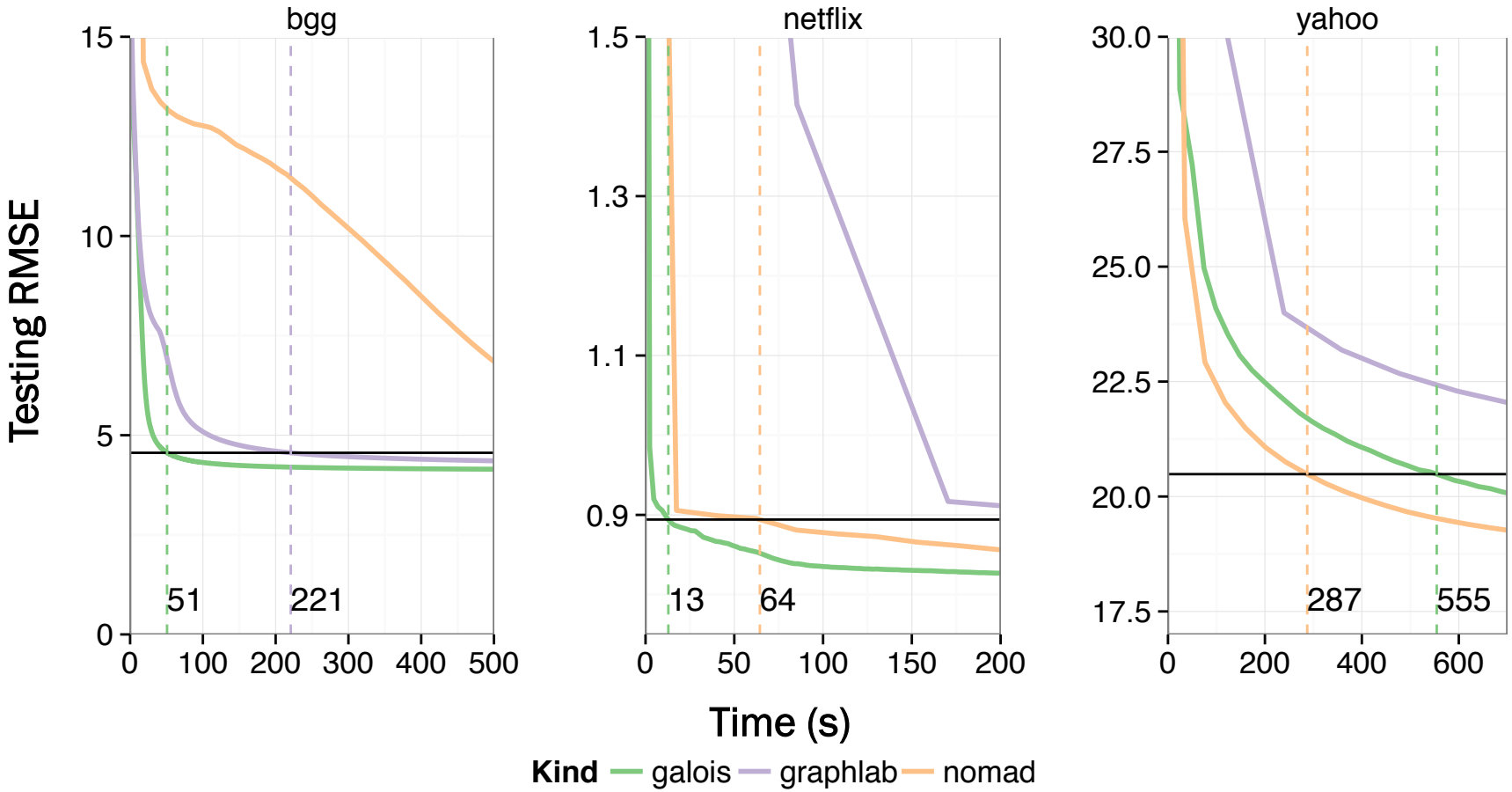
- Parameters
  - Same SGD parameters, initial latent vectors between implementations
  - $\varepsilon(n) = \text{alpha} / (1 + \text{beta} * n^{1.5})$
  - Handtuned tile size

Evaluation of Throughput



nomad with 40 threads on bgg does not converge

Evaluation of Convergence



#Threads = 20

Training over entire dataset

Lessons

- When the node data is large and the graph is relatively dense, simple row-wise iteration can be inefficient
  - Large number of cache misses
- Solution: apply operator across smaller 2D regions of the graph

A Lightweight Infrastructure for Graph Analytics

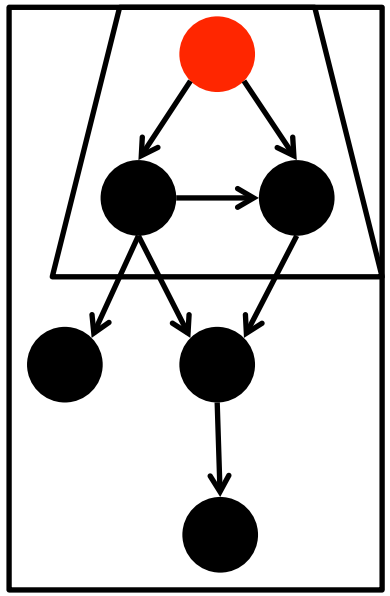
Donald Nguyen

Andrew Lenharth and Keshav Pingali

Appeared at SOSP ‘13

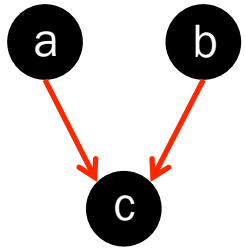
Parallel Program = Operator + Schedule + Parallel Data Structure

Algorithm



- What is the operator?
  - Ligra, PowerGraph: only vertex programs
  - Galois: Unrestricted, may even morph graph by adding/removing nodes and edges

- Where/When does it execute?
  - Autonomous scheduling: activities execute transactionally
  - Coordinated scheduling: activities execute in rounds
    - Read values refer to previous rounds
    - Multiple updates to the same location are resolved with reduction, etc.



Graph Analytics DSLs

- GraphLab Low et al. (UAI ’10)
- PowerGraph Gonzalez et al. (OSDI ’12)
- GraphChi Kyrola et al. (OSDI ’12)
- Ligra Shun and Blelloch (PPoPP ’13)
- Pregel Malewicz et al. (SIGMOD ‘10)
- …

- Easy to implement their APIs on top of Galois system
  - Galois implementations called PowerGraph-g, Ligra-g, etc.
  - About 200-300 lines of code each

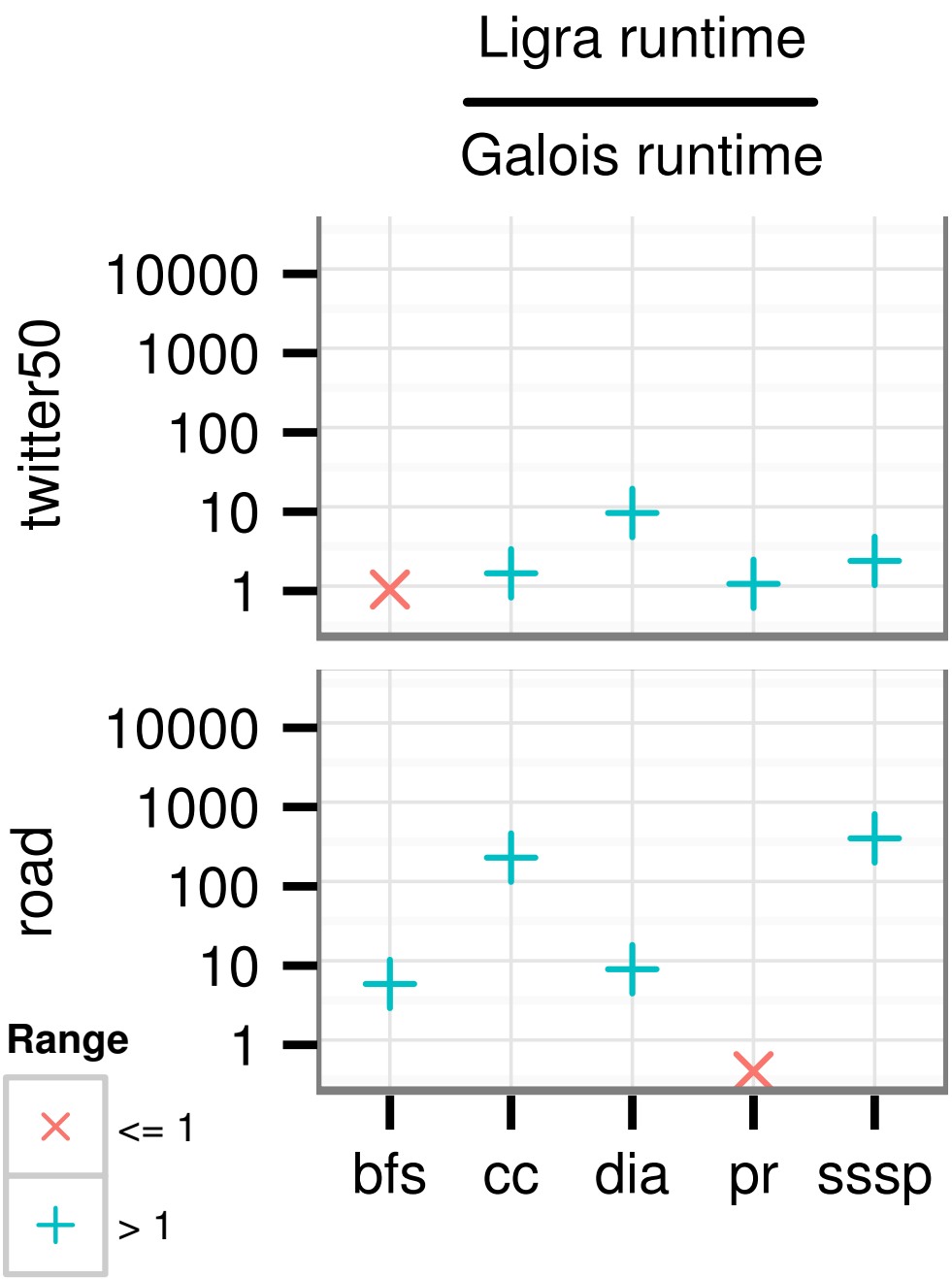
Evaluation

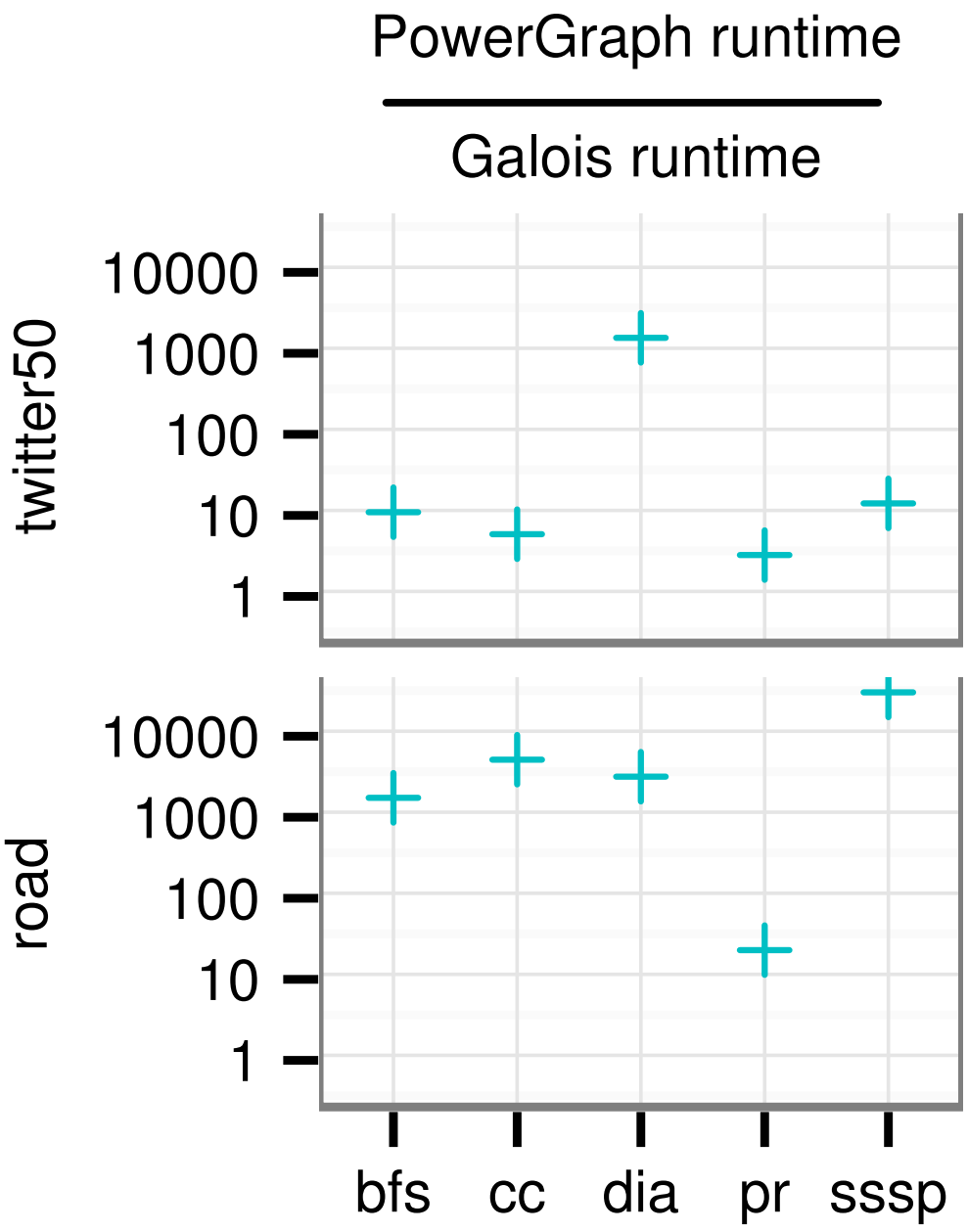
- Platform
  - 40-core system
    - 4 socket, Xeon E7-4860 (Westmere)
  - 128 GB RAM

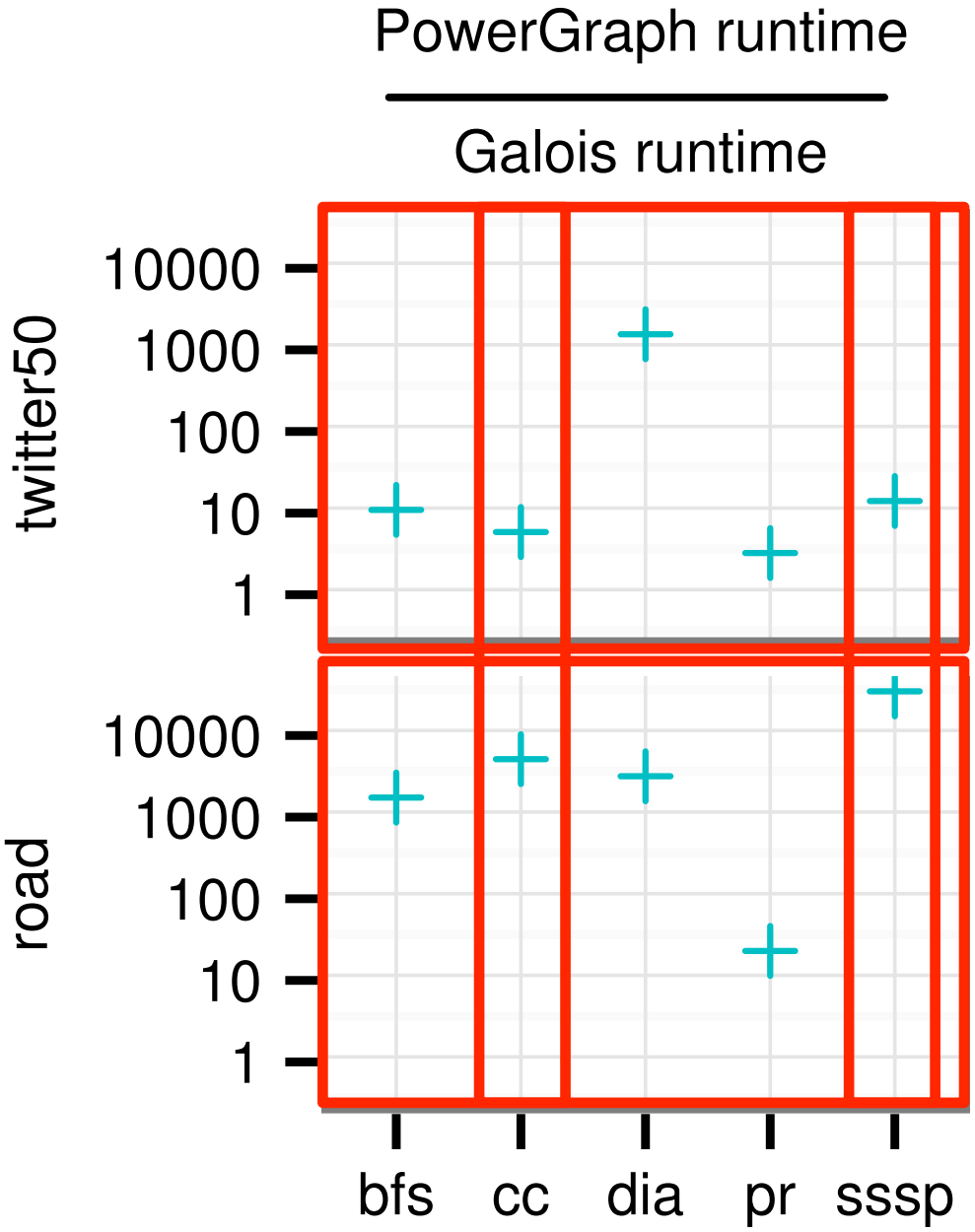
- Applications
  - Breadth-first search (bfs)
  - Connected components (cc)
  - Approximate diameter (dia)
  - PageRank (pr)
  - Single-source shortest paths (sssp)

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

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


Ligra runtime
Galois runtime
twitter50
road
10000
1000
100
10
1
bfs
cc
dia
pr
sssp
Range
<= 1
> 1


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
- Autonomous scheduling required for high-diameter graphs
  - Coordinated scheduling uses many rounds and has too much overhead

Summary

- Galois programming model is general
  - Permits efficient algorithms to be written
- Galois infrastructure is lightweight
- Simpler graph DSLs can be layered on top
  - Can perform better than existing systems