Introduction to the Galois System

Donald Nguyen

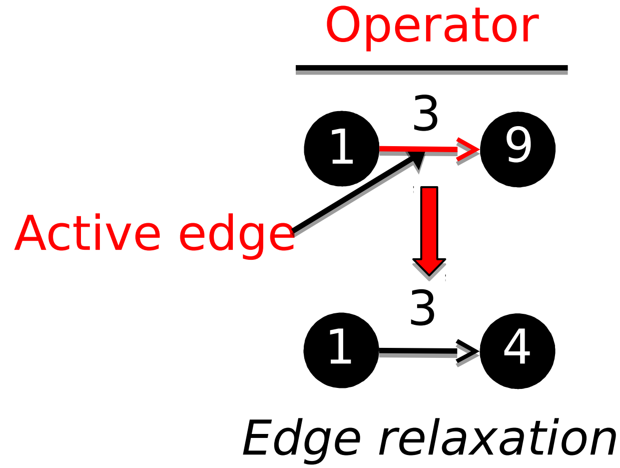
University of Texas at Austin

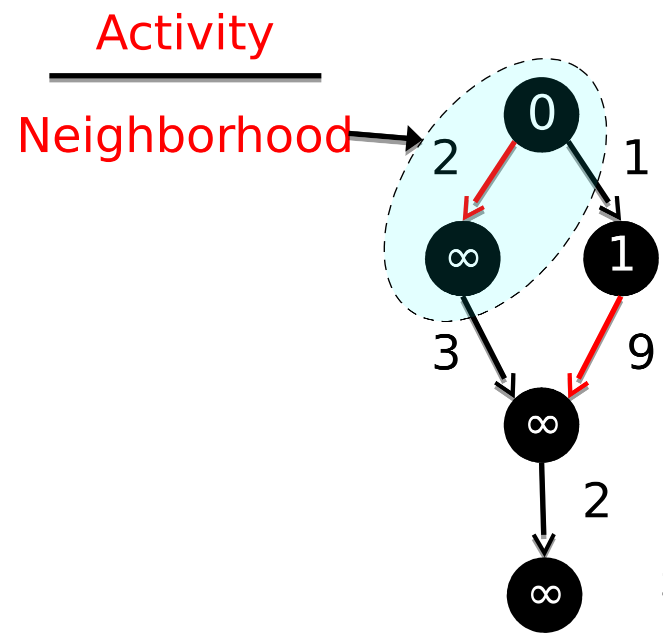
Overview

- Review
  - SSSP, Operator Formulation, Connected Components
- Other Programming Models
  - Bulk-synchronous, Gather-Apply-Scatter
- Example Programs
- Using Galois
  - Iterators, Graphs, Building Programs, Tuning

SSSP

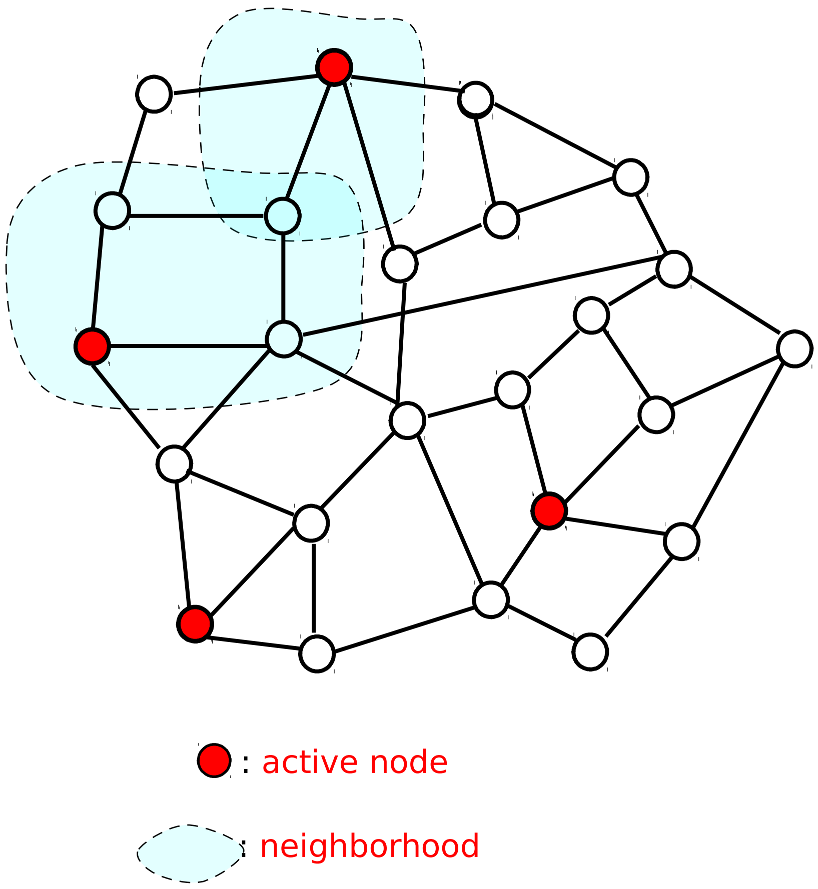
- Find the shortest distance from source node to all other nodes in a graph
  - Label nodes with tentative distance
  - Assume non-negative edge weights
- Algorithms
  - Chaotic relaxation O(2V)
  - Bellman-Ford O(VE)
  - Dijkstra’s algorithm O(E log V)
    - Uses priority queue
  - Δ-stepping
    - Uses sequence of bags to prioritize work
    - Δ=1, O(E log V)
    - Δ=∞, O(VE)
- Different algorithms are different schedules for applying relaxations
  - SSSP needs priority scheduling for work efficiency





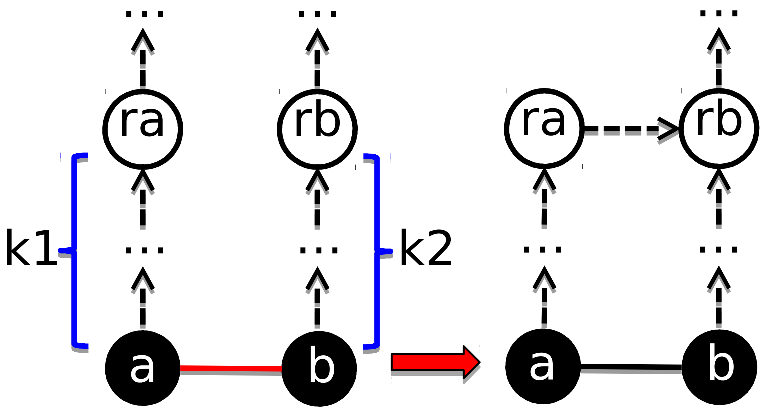
Abstraction of Algorithms

- 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 (chaotic relaxation, Jacobi, …)
    - Ordered algorithms: algorithm-specific order (Dijkstra, …)

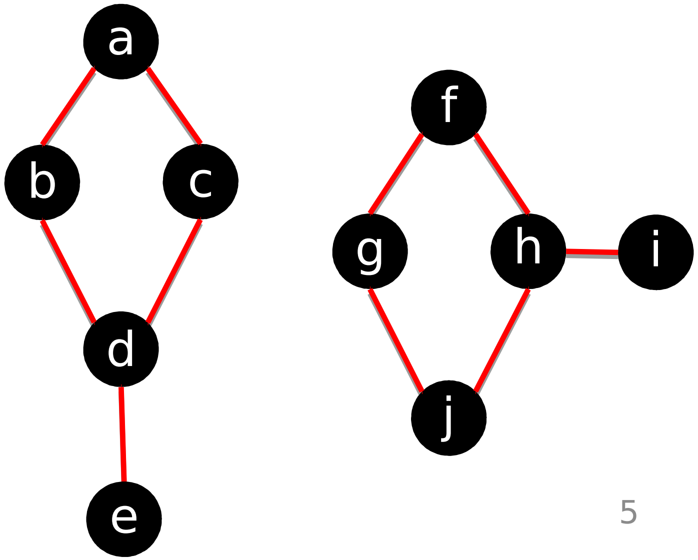


Connected Components

- Construct a labeling function such that color(x) = color(neighbor(x))
- Algorithms
  - Label propagation
  - Union-Find
  - Hybrids
- Different algorithms are different instantiations of generic operator
  - k*i* = 1, 2, …
  - edge => subgraph

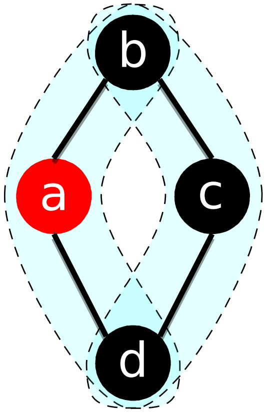


Generic Operator



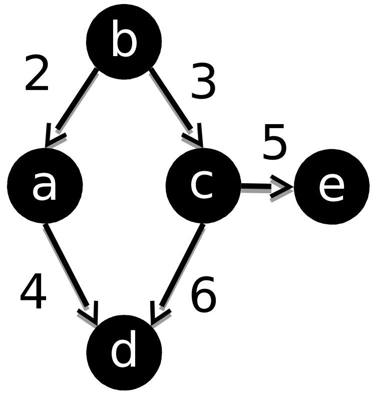
Programming Models (1)

- 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
- Gather-Apply-Scatter
  - Refinement of above into subrounds
  - Ex: GraphLab / PowerGraph



Programming Models (2)

- MapReduce
  - Graph-Table duality
  - Bulk operations on tables
    - Map, join, …
  - Ex: Hadoop, Spark GraphX

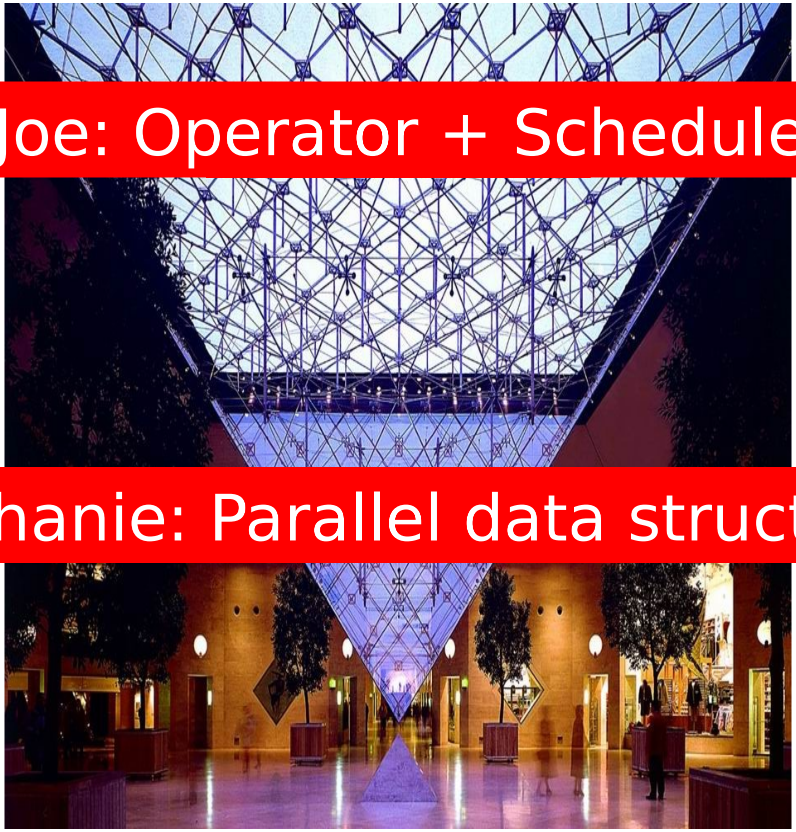


eid	src	dst
1	b	a
2	b	c
3	a	d
4	c	d
5	c	e

eid	value
1	2
2	3
3	4
4	5
5	6

Galois System: Two-Level Infrastructure

- Small number of expert programmers must support a large number of application programmers
  - cf. SQL
- Galois project
  - Program = Algorithm + Data structure (Wirth)
  - Library of concurrent data structures and runtime system written by expert programmers
  - Application programmers code in sequential C++
    - All concurrency control is in data structure library and runtime system



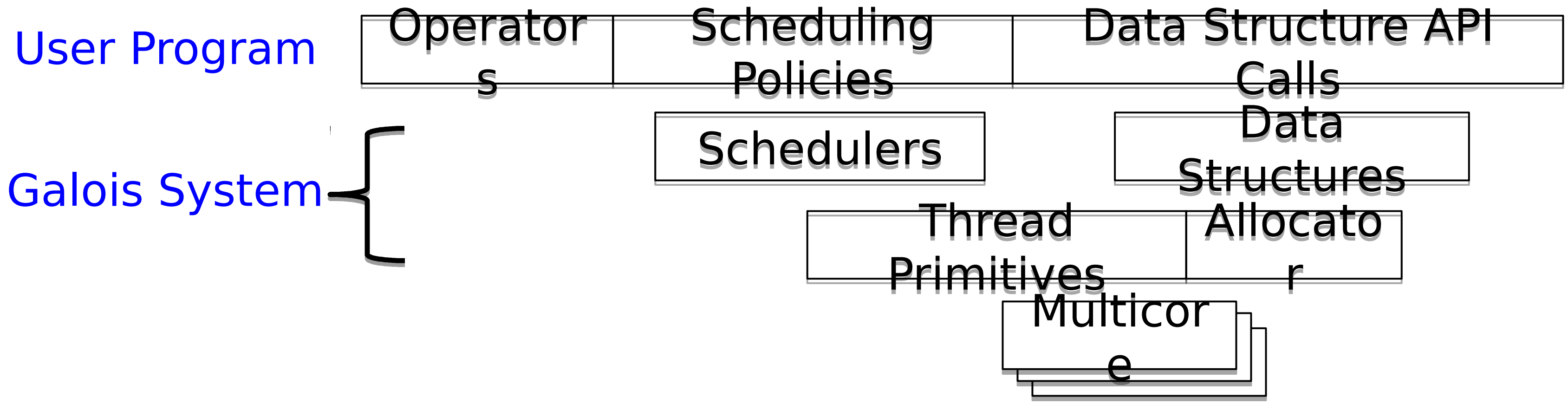
Joe: Operator + Schedule

Stephanie: Parallel data structure

allel program = Operator + Schedule + Parallel data structu

Galois System

Parallel Program = Operator + Schedule + Parallel Data Structure



Galois SSSP

```
#include "Galois/Galois.h"
#include "Galois/Graph/Graph.h"

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

Graph graph;

struct P {
 void operator()(Task& t, Galois::UserContext<Task>& c) {
  int srcDist = graph.getData(t.second);
  if (t.first > srcDist) return;
  for (auto edge : graph.out_edges(t.second)) {
   GNode dst = graph.getEdgeDst(edge);
   int& dstDist = graph.getData(dst);
   int newDist = srcDist + graph.getEdgeData(edge);
   if (newDist < dstDist) {
    dstDist = newDist;
    ctx.push(std::make_pair(newDist, dst));
   }
  }
 }
};

int main(… ) {
 Galois::Graph::readGraph(graph, … );
 …
 Galois::for_each(initial, P(), Galois::wl<WL>());
 return 0;
}
```

```
struct TaskIndexer {
 int operator()(Task& t) {
  return t.first >> shift;
 }
};

using namespace Galois::WorkList;

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

Data structure

Operator

Galois Iterator

Galois Connected Components

```
using namespace Galois::Graph;
typedef LC_CSR_Graph<int,void> Graph;
typedef Graph::GraphNode GNode;

Graph graph;

struct P {
 void operator()(GNode& n,
    Galois::UserContext<GNode>& c)
 {
  int srcData = graph.getData(n);
  for (auto edge : graph.out_edges(n)) {
   GNode dst = graph.getEdgeDst(edge);
   int& dstData = graph.getData(dst);
   if (srcData < dstData) {
    dstData = srcData;
    c.push(dst);
   }
  }
 }
};

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

Label Propagation (Push)

```
struct Node: Galois::UnionFindNode<Node> { }

using namespace Galois::Graph;
typedef LC_CSR_Graph<Node,void> Graph;
typedef Graph::GraphNode GNode;

Graph graph;

struct P {
 void operator()(GNode& n,
    Galois::UserContext<GNode>&)
 {
  Node& srcData = graph.getData(n);
  for (auto edge : graph.out_edges(n)) {
   GNode dst = graph.getEdgeDst(edge);
   Node dstData = graph.getData(dst);
   srcData.merge(dstData);
  }
 }
};

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

Union-Find

GraphLab SSSP

```
#include <graphlab.hpp>
#include <graphlab/macros_def.hpp>
using namespace graphlab;

struct node_data : IS_POD_TYPE { int dist; }; struct edge_data : IS_POD_TYPE { int dist; };
struct scatter_msg : IS_POD_TYPE { int dist; … }; struct gather_msg : IS_POD_TYPE { };
typedef distributed_graph<node_data, edge_data> graph;

struct P : ivertex_program <graph, gather_msg, scatter_msg> , IS_POD_TYPE {
 int v;
 bool changed;

 void init(icontext_type&, vertex_type&, scatter_msg& m) { v = m.dist; }
 edge_dir_type gather_edges(icontext_type&, vertex_type&) { return NO_EDGES; }
 gather_data gather(icontext_type&, vertex_type&, edge_type&) { … }
 void apply(icontext_type&, vertex_type& v, gather_msg&) {
  changed = (v.data().dist < v);
  v.data().dist = v;
 }
 edge_dir_type scatter_edges(icontext_type&, vertex_type&) {
  if (changed) return OUT_EDGES;
  else return NO_EDGES;
 }
 void scatter(icontext_type& c, vertex_type& v, edge_type& e) {
  int newdist = v.data().dist + edge.data().dist;
  if (e.target().data().dist > newdist)
   c.signal(e.target(), scatter_msg(newdist));
 }
};

graph g; … ; omni_engine<P> engine(…, g, …); engine.signal(… scatter_msg(…)); engine.start();
```

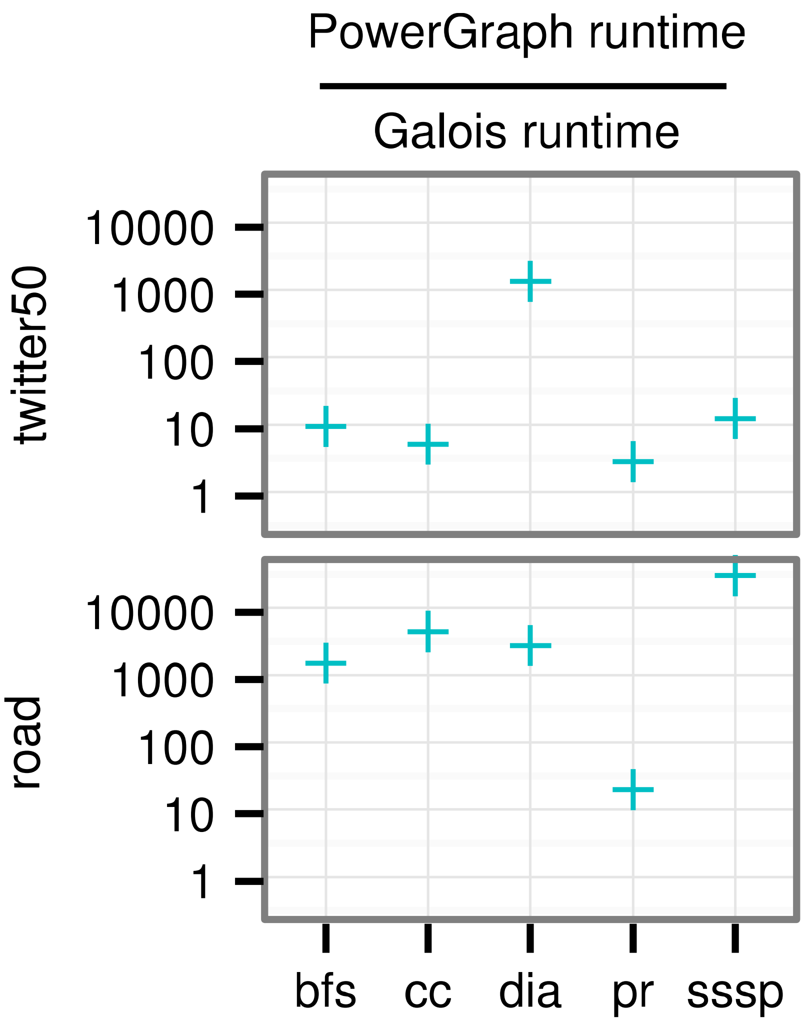
GraphX Connected Components

```
import scala.reflect.ClassTag
import org.apache.spark.graphx._

object ConnectedComponents {
  def run[VD: ClassTag, ED: ClassTag](graph: Graph[VD, ED]): Graph[VertexId, ED] = {
    val ccGraph = graph.mapVertices { case (vid, _) => vid }
    def sendMessage(edge: EdgeTriplet[VertexId, ED]) = {
      if (edge.srcAttr < edge.dstAttr) {
        Iterator((edge.dstId, edge.srcAttr))
      } else if (edge.srcAttr > edge.dstAttr) {
        Iterator((edge.srcId, edge.dstAttr))
      } else {
        Iterator.empty
      }
    }
    val initialMessage = Long.MaxValue
    Pregel(ccGraph, initialMessage, activeDirection = EdgeDirection.Either)(
      vprog = (id, attr, msg) => math.min(attr, msg),
      sendMsg = sendMessage,
      mergeMsg = (a, b) => math.min(a, b))
  }
}
```

Limitations of Vertex Programs

- 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
  - Bulk-synchronous scheduling has too many rounds and has too much overhead



Galois System

- include/Galois/*.h
  - Galois.h
  - Bag.h
- include/Galois/WorkList/*.h
  - WorkList.h
  - FIFO.h, LIFO.h, Chunked.h, …
- include/Galois/Graph/*.h
  - Graph.h
  - FirstGraph.h
  - LC_CSR_Graph.h, …

Galois.h

```
// Galois unordered iterator over iterator range
template<typename I, typename P, typename Args... >
  void for_each(I begin, I end, P p, Args... );

// Galois unordered iterator over Galois container
template<typename C, typename P, typename Args... >
  void for_each_local(C& gcontainer, P p, Args... );

// Simple data parallel loop
template<typename I, typename P, typename Args... >
  void do_all(I begin, I end, P p, Args... );
```

Operator Context

```
//Operator argument of for_each implements this
void operator()(T n, Galois::UserContext<T>& ctx);

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();
};

//Operator for do_all implements this
void operator()(T n);
```

WorkList.h

- Various scheduling policies available

```
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 a policy

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


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

LC_CSR_Graph.h

- 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);

// C++
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);
 }
}
```

FirstGraph.h

- Morph Computation

```
template<typename NodeData,typename EdgeData,bool Dir>
struct FirstGraph {
 template<typename… Args>
  GraphNode createNode(Args&&… args);

 void addNode(GraphNode);
 void removeNode(GraphNode);
 edge_iterator addEdge(GraphNode,GraphNode);
 void removeEdge(GraphNode,edge_iterator);
};
```

Building Galois Programs

1. Download

```
wget http://iss.ices.utexas.edu/projects/galois/downloads/Galois-2.2.1.tar.gz
tar xzvf Galois-2.2.1.tar.gz
SDIR=`pwd`/Galois-2.1.1
```

2. Build

```
mkdir $BDIR; cd $BDIR
cmake $SDIR –DCMAKE_INSTALL_PREFIX=$IDIR
make  # alternatives: make –j8; make –C apps/sssp –j
```

3. Install

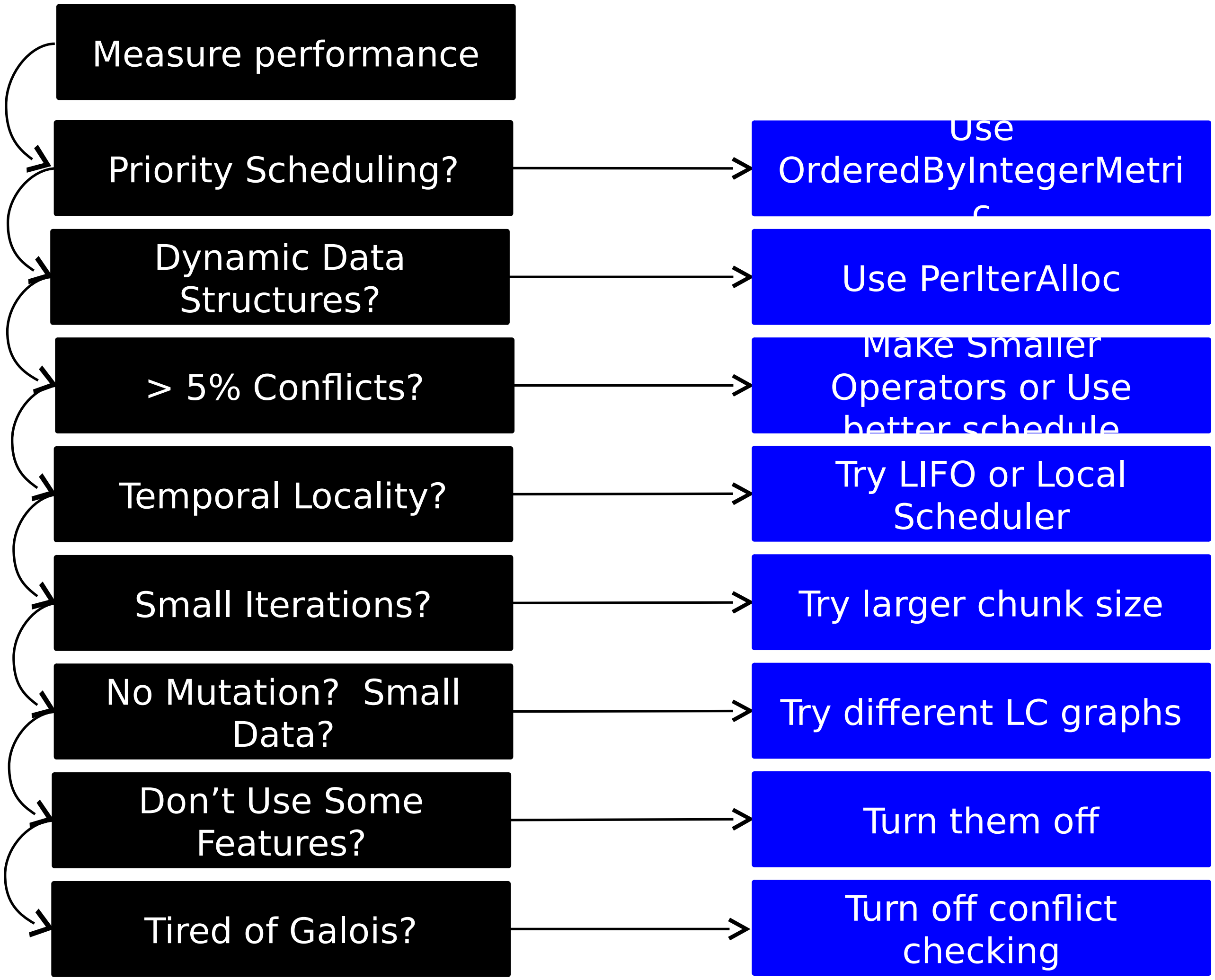
```
make install
g++ -std=c++11 \
 –L$IDIR/lib –I$IDIR/include \
 myapp.cpp –lgalois
```

or

3. In-source builds

```
vim $SDIR/apps/CMakeLists.txt
vim $SDIR/apps/myapp/CMakeLists.txt
make –C $BDIR/apps/myapp
```

Tuning Galois Programs



Measuring Performance

- include/Galois/Statistic.h

```
class StatTimer {
 void start();
 void stop();
};

class StatManager { … };
```

- In action

```
int main(… ) {
 Galois::StatManager sm;

 Galois::StatTimer totalTime;
 totalTime.start();
 Galois::for_each(… , Galois::loopname("first"));

 Galois::StatTimer secondTime("secondTime");
 secondTime.start();
 Galois::for_each(… , Galois::loopname("second"));
 secondTime.stop();
 totalTime.stop();

 return 0;
}
```

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
- Programs that fail 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

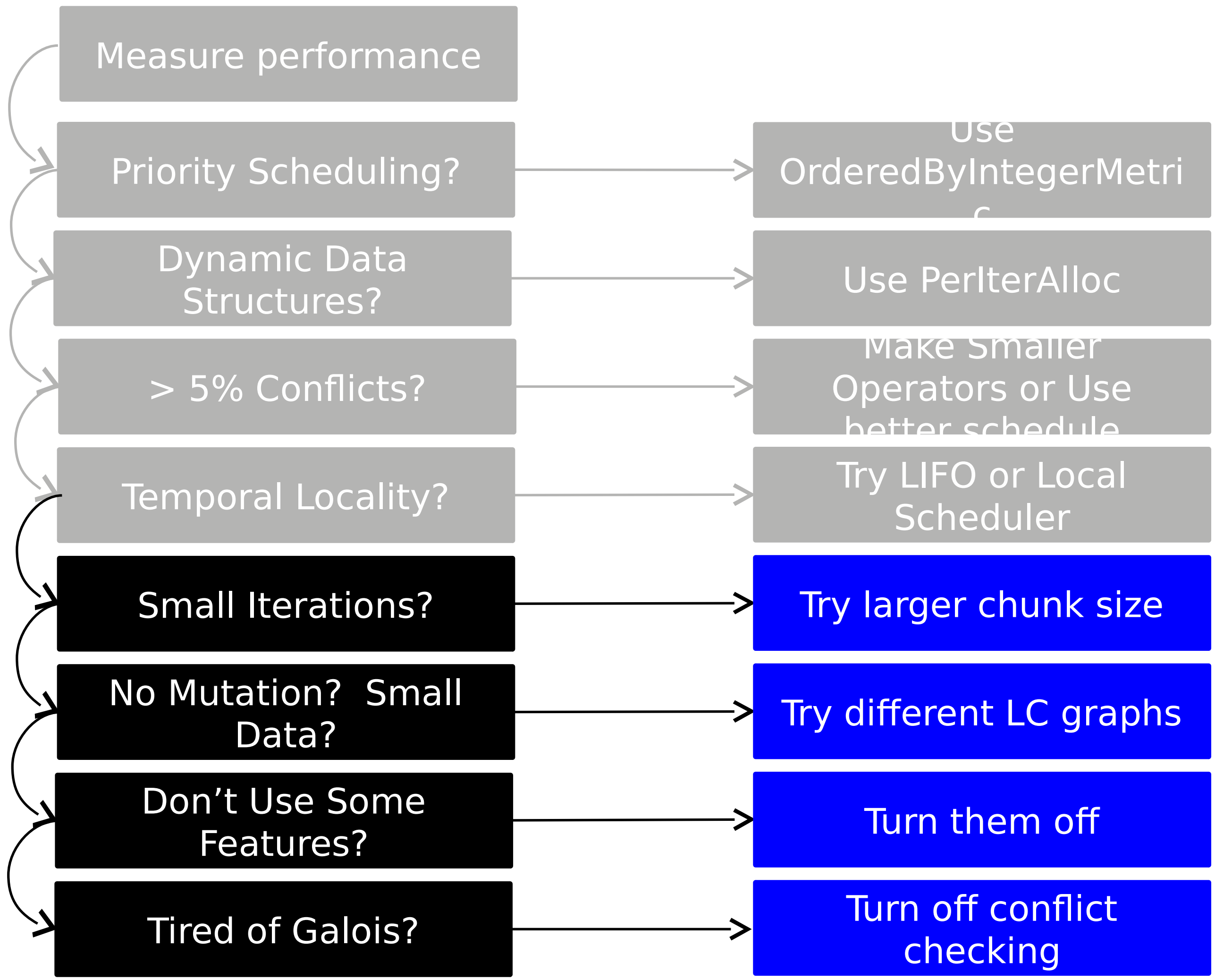
Bonus: Spatial Locality!

- Use locality maintaining and preserving for_each_local
  - 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 Galois Programs



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
    }
  }
}
```

Summary

- Operator formulation of algorithms
  - Program = Algorithm + Data Structure
  - Parallel Program = Parallel Algorithm + Parallel Data Structure
  - Parallel Algorithm = Operator + Schedule
- Galois system supports a wide variety of programs
  - Compare with GraphLab, Spark GraphX
- High-performance achieved through program refinement
  - Tuning scheduler, adjusting data structures, …