Parallel Programming with the Galois System

Andrew Lenharth
The University of Texas at Austin
• Thinking about algorithms using the amorphous parallelism framework.
• Implementing them using the Galois runtime.
• Presented with a focus on graph analytics and big data.
Outline

• Current State of Parallel programming
• Amorphous data parallelism / Operator formulation
• High Level Galois
• Current state of the system
• Implementing Graph Analytic DSLs on Galois
• Practical Galois
• Extended Example: SSSP
• Scheduling
• Active research
Intel Study: Galois vs. Graph Frameworks

“Navigating the maze of graph analytics frameworks” Nadathur et al SIGMOD 2014
FROM:
COMPUTATION CENTRIC
TO:
DATA CENTRIC

PROGRAMMING MODELS
Parallelism is everywhere

Texas Advanced Computing Center

Laptops

Cell-phones
Parallel programming?

- 40-50 years of work on parallel programming in HPC domain
- Focused mostly on “regular” dense matrix/vector algorithms
  - Stencil computations, FFT, etc.
  - Mature theory and tools
- Not useful for “irregular” algorithms that use graphs, sets, and other complex data structures
  - Most algorithms are irregular 😞
- Galois project:
  - New data-centric abstractions for parallelism and locality
  - Galois system for multicores and GPUs

“The Alchemist”
Cornelius Bega (1663)
HPC example

- **Finite-difference computation**
- **Algorithm**
  - Operator: five-point stencil
  - Different schedules have different locality
- **Regular application**
  - Application can be parallelized at compile-time

//Jacobi iteration with 5-point stencil
//initialize array A
for time = 1, nsteps
  for <i,j> in [2,n-1]x[2,n-1]
    temp(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))
  for <i,j> in [2,n-1]x[2,n-1]:
    A(i,j) = temp(i,j)
Irregular example

Mesh m = /* read in mesh */
WorkList wl;
wl.add(m.badTriangles());
while (true) {
  if (wl.empty()) break;
  Element e = wl.get();
  if (e no longer in mesh)
    continue;
  Cavity c = new Cavity();
  c.expand();
  c.retriangulate();
  m.update(c);//update mesh
  wl.add(c.badTriangles());
}

• Where is parallelism in
  program?
    – Loop: do static analysis to find dependence graph
• Static analysis fails to find parallelism.
    – May be there is no parallelism in program?
Data-centric view of algorithm

- **Algorithm**
  - composition of unitary *actions* on data structures

- **Actions:** *operator*
  - DMR: {find cavity, retriangulate, update mesh}

- **Composition of actions:**
  - specified by a *schedule*

- **Parallelism**
  - disjoint actions can be performed in parallel

- **Parallel data structures**
  - graph
  - worklist of bad triangles
Operator formulation of algorithms

• **Active element**
  – Site where computation is needed

• **Operator**
  – Computation at active element
  – Activity: application of operator to active element

• **Neighborhood**
  – Set of nodes/edges read/written by activity
  – Distinct usually from neighbors in graph

• **Ordering**: scheduling constraints on execution order of activities
  – Unordered algorithms: no semantic constraints but performance may depend on schedule
  – Ordered algorithms: problem-dependent order

• **Amorphous data-parallelism**
  – Multiple active nodes can be processed in parallel subject to neighborhood and ordering constraints

Parallel program = Operator + Schedule + Parallel data structure
Parallelization strategies: Binding Time

When do you know the active nodes and neighborhoods?

Compile-time

Static parallelization  (stencil codes, FFT, dense linear algebra)

After input is given

Inspector-executor (Bellman-Ford)

During program execution

Interference graph (DMR, chaotic SSSP)

After program is finished

Optimistic Parallelization  (Time-warp)

“The TAO of parallelism in algorithms” Pingali et al, PLDI 2011
TAO analysis: Structure in algorithms (PLDI 2011)

Algorithms
  - Topology
    - Structured (grid, clique, set,..)
    - Semi-structured (tree)
    - Unstructured (general graph)
  - Active Nodes
    - Location
      - Topology-driven
      - Data-driven
    - Ordering
      - Unordered
      - Ordered
  - Operator
    - Morph
    - Local computation
    - Reader

: active node
: neighborhood
Outline

• Current State of Parallel programming
• Amorphous data parallelism / Operator formulation
• **High Level Galois**
• Current state of the system
• Implementing Graph Analytic DSLs on Galois
• Practical Galois
• Extended Example: SSSP
• Scheduling
• Active research
Galois System

Parallel Program = Operator + Schedule + Parallel Data Structure

User Program

<table>
<thead>
<tr>
<th>Operators</th>
<th>Schedules</th>
<th>Data Structure API Calls</th>
</tr>
</thead>
</table>

Galois System

<table>
<thead>
<tr>
<th>Schedulers</th>
<th>Data Structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread Primitives</td>
<td>Allocator</td>
</tr>
</tbody>
</table>

Multicore
Multi-level Programming Model

Parallel program = Operator + Schedule + Parallel data structures

- **Ubiquitous parallelism:**
  - small number of expert programmers (Stephanies) must support large number of application programmers (Joes)
  - cf. SQL

- **Galois system:**
  - Stephanie: library of concurrent data structures and runtime system
    - Provides serializable, atomic execution of activities
  - Joe: application code in sequential C++
    - Galois set iterator for highlighting opportunities for exploiting ADP
Parallel Program = Operator + Schedule + Parallel Data Structure

Algorithm

• What is the operator?
  – Other graph analytics frameworks: only vertex programs
  – Galois: Unrestricted, may even morph graph by adding/removing nodes and edges

• Where/When does it execute?
  – Autonomous scheduling: activities execute asynchronously and transactionally
  – Coordinated scheduling: activities execute in rounds
    • Read values refer to previous rounds
    • Multiple updates to the same location are resolved with reduction, etc.
Galois Parallel Execution Model

Parallel execution model:
- Shared-memory
- Optimistic execution of Galois iterators

Implementation:
- Master thread begins execution of program
- When it encounters iterator, worker threads help by executing iterations concurrently
- Iterations may enqueue new tasks
- Barrier synchronization at end of iterator

Independence of neighborhoods:
- Concurrency managed by data structure library
- Logical locks on nodes and edges
- Implemented using CAS operations
#include "Galois/Galois.h"
#include "Galois/Graphs/LCGraph.h"

struct Data { int value; float f; };

typedef Galois::Graph::LC_CSR_Graph<Data, void> Graph;
typedef Galois::Graph::GraphNode Node;

Graph graph;

struct P {
    void operator()(Node n, Galois::UserContext<Node>& ctx) {
        graph.getData(n).value += 1;
    }
};

int main(int argc, char** argv) {
    graph.structureFromGraph(argv[1]);
    Galois::for_each(graph.begin(), graph.end(), P());
    return 0;
}
Lonestar

• Collection irregular algorithms
Outline

- Current State of Parallel programming
- Amorphous data parallelism / Operator formulation
- High Level Galois
- **Current state of the system**
- Implementing Graph Analytic DSLs on Galois
- Practical Galois
- Extended Example: SSSP
- Scheduling
- Active research
Galois: Performance on SGI Ultraviolet

![Graph showing performance comparison of different implementations on SGI Ultraviolet.](image)

<table>
<thead>
<tr>
<th>App</th>
<th>Implementation</th>
<th>Threads</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dmr</td>
<td>Galois</td>
<td>1</td>
<td>155.7</td>
</tr>
<tr>
<td></td>
<td>Galois</td>
<td>512</td>
<td>0.37</td>
</tr>
<tr>
<td>dt</td>
<td>triangle</td>
<td>1</td>
<td>1185</td>
</tr>
<tr>
<td></td>
<td>Galois</td>
<td>512</td>
<td>56.6</td>
</tr>
<tr>
<td></td>
<td>Galois</td>
<td>512</td>
<td>0.18</td>
</tr>
<tr>
<td>bh</td>
<td>splash2</td>
<td>1</td>
<td>&gt;6000</td>
</tr>
<tr>
<td></td>
<td>Galois</td>
<td>1</td>
<td>1386</td>
</tr>
<tr>
<td></td>
<td>Galois</td>
<td>512</td>
<td>3.55</td>
</tr>
<tr>
<td>bc</td>
<td>HPCS SSCA</td>
<td>1</td>
<td>6720</td>
</tr>
<tr>
<td></td>
<td>Galois</td>
<td>1</td>
<td>5394</td>
</tr>
<tr>
<td></td>
<td>Galois</td>
<td>512</td>
<td>21.6</td>
</tr>
<tr>
<td>tri</td>
<td>graphlab</td>
<td>2</td>
<td>531</td>
</tr>
<tr>
<td></td>
<td>Galois</td>
<td>1</td>
<td>7.03</td>
</tr>
<tr>
<td></td>
<td>Galois</td>
<td>512</td>
<td>0.028</td>
</tr>
</tbody>
</table>

Table 2: Serial runtime comparisons to other implementations rounded to the nearest second. Included are runtimes for Galois algorithms at 512 threads. The splash2 implementation of bh timed out after 100 minutes.
## GPU implementation

<table>
<thead>
<tr>
<th></th>
<th>Single Source Shortest Path</th>
<th>Survey Propagation</th>
<th>Delaunay Mesh Refinement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image1" alt="Graph" /></td>
<td><img src="image2" alt="Graph" /></td>
<td><img src="image3" alt="Graph" /></td>
</tr>
</tbody>
</table>

- **Barnes Hut**
- **Points-to Analysis**

<table>
<thead>
<tr>
<th></th>
<th>Barnes Hut</th>
<th>Points-to Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image4" alt="Graph" /></td>
<td><img src="image5" alt="Graph" /></td>
</tr>
</tbody>
</table>

**Inputs:**

- **SSSP:** 23M nodes, 57M edges
- **SP:** 1M literals, 4.2M clauses
- **DMR:** 10M triangles
- **BH:** 5M stars
- **PTA:** 1.5M variables, 0.4M constraints

**Multicore:** 24 core Xeon
**GPU:** NVIDIA Tesla
nomad with 40 threads on bgg does not converge
“Navigating the maze of graph analytics frameworks” Nadathur et al SIGMOD 2014
Moctar & Brisk, “Parallel FPGA Routing based on the Operator Formulation”
DAC 2014
Outline

• Current State of Parallel programming
• Amorphous data parallelism / Operator formulation
• High Level Galois
• Current state of the system
• Implementing Graph Analytic DSLs on Galois
• Practical Galois
• Extended Example: SSSP
• Scheduling
• Active research
What is Graph Analytics?

• Algorithms to compute properties of graphs
  – Connected components, shortest paths, centrality measures, diameter, PageRank, ...

• Many applications
  – Google, path routing, friend recommendations, network analysis

• Difficult to implement on a large scale
  – Data sets are large, data accesses are irregular
  – Need parallelism and efficient runtimes
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

“A lightweight infrastructure for graph analytics”
Nguyen, Lenharth, Pingali (SOSP 2013)
• 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
Outline

• Current State of Parallel programming
• Amorphous data parallelism / Operator formulation
• High Level Galois
• Current state of the system
• Implementing Graph Analytic DSLs on Galois
• **Practical Galois**
• Extended Example: SSSP
• Scheduling
• Active research
Galois in Practice

• C++ library
  – Galois::for_each(begin, end, functor)
  – Galois::Graph::*, Galois::Bag, ...

• Currently supports
  – Cautious operators (i.e., no undos)
  – No static analysis (e.g., POPL 2011)
Building Galois Programs

• Requirements
  – Linux, modern compiler, Boost headers
    • Partial support for Solaris, Windows
    • Partial support for Intel MIC, Arm, Power
  – Hugepages (optional)

• As easy as gcc...
  g++ -I${GDIR}/include -L${GDIR}/lib *.cpp -lgalois

• Galois distribution uses CMake to simplify build
  cmake ${GDIR}; make
Baseline Runtime System

• Speculative execution-based runtime

• Provides hooks (Joe++) to allow user to optimize performance

• Once program works correctly in parallel, then optimize
#include "Galois/Galois.h"
#include "Galois/Graphs/LCGraph.h"

using namespace Galois;

struct Data { int value; float f; }

typedef Graph::LC_Linear_Graph<Data,void> Graph;
typedef Graph::GraphNode Node;

Graph graph;

struct P {
    void operator()(const Node& n, UserContext<Node>& ctx) {
        graph.getData(n).value += 1;
    }
};

int main(int argc, char** argv) {
    graph.structureFromGraph(argv[1]);
    for_each(graph.begin(), graph.end(), P());
    return 0;
}
A Galois Program

• Operator
  – The Context

• Iterator
  – Topology-Driven
  – Data-Driven

• Data Structures
  – Api for graphs, etc

• Scheduling
  – Priorities, etc

• Miscellaneous directives
Example Operator

//Operators are any valid C++ functor with the correct signature

```cpp
struct P {
    Graph& g;
    P(Graph& g) : g(g) {}
    void operator()(const Node& n, UserContext<Node>& ctx) {
        graph.getData(n).value += 1;
    }
};
```
The Operator Context

```c
void operator()(const Node& n, UserContext<Node>& ctx);
```

- Context is a handle to the loop-runtime
- UserContext<WorkItemType> has
  - breakLoop(); //Break out of the current parallel loop (eventually)
  - PerIterAllocTy& getPerIterAlloc(); //A per-iteration region allocator
  - void push(Args&&... args); //Add a new item to the worklist (forwards args to WorkItemType constructor)
Fast Local Memory

```cpp
void operator()(const Node& n, UserContext<Node>& ctx) {
    // This vector uses scalable allocation
    std::vector<Node, Galois::PerIterAllocTy::rebind<Node>::other> vec(ctx.getPerIterAlloc());
    for (...) { vec.push_back(graph.getEdgeDst(ii)); }
}
```
Applying an Operator: Topology

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

//Standard Topology driven initialization
Galois::for_each(graph.begin(), graph.end(),
    [&graph] (const Node& n, UserContext<Node>& ctx) {
        graph.getData(n).value = 0;
    });
Applying an Operator: Data-driven

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

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

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

• Local Computation, Compressed Sparse Row
• Key Typedefs:
  – GraphNode: node handle
  – edge_iterator
  – iterator
• Key Functions:
  – nodeData& getData(GraphNode)
  – edgeData& getEdgeData(edge_iterator)
  – GraphNode getEdgeDst(edge_iterator)
  – iterator begin()
  – iterator end()
  – edge_iterator edge_begin(GraphNode)
  – edge_iterator edge_end(GraphNode)
LC_CSR_Graph Example

//sum values on edges and nodes
LC_CSR_Graph<double, double> graph;
...
double sum;
for (auto N : graph) {
    sum = graph.getData(N);
    for (auto ii = graph.edge_begin(N), ee = graph.edge_end(N);
         ii != ee; ++ii) {
        sum += graph.getEdgeData(ii);
    }
}
Scheduling

• In Galois/WorkList/*
• Scheduling specified by mini language in optional argument to for_each loops

```cpp
using namespace Galois::WorkList;
typedef dChunkedLIFO<256> Sched;
Galois::for_each(g.begin(), g.end, Op(),
    Galois::wl<Sched>());
```
Standard Scheduling Options

*Most have options (including sub-schedulers)*

- Lifo (Fifo) Like:
  - LIFO, ChunkedLIFO, dChunkedLIFO
  - AltChunkedLIFO
- No worklist pushes:
  - StableIterator
- Round Based:
  - BulkSynchronous
- New Work stays local:
  - LocalQueue
- Priority Scheduling:
  - OrderedByIntegerMetric
Useful Directives

• Loopname: report statistics by loop
  – for_each(..., loopname(“name”));
• Timers: Galois::StatTimer
  – May be named
• PAPI measurements
• reportpageAlloc: report pages allocated
• setActiveThreads(n) : limit threads to n
Outline

• Current State of Parallel programming
• Amorphous data parallelism / Operator formulation
• High Level Galois
• Current state of the system
• Implementing Graph Analytic DSLs on Galois
• Practical Galois

**Extended Example: SSSP**
• Scheduling
• Active research
Example: 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 $O(2^V)$
  – Bellman-Ford $O(VE)$
  – Dijkstra’s algorithm $O(E \log V)$
    • Uses priority queue
  – $\Delta$-stepping
    • Uses sequence of bags to prioritize work
    • $\Delta=1$, $O(E \log V)$
    • $\Delta=\infty$, $O(VE)$

• Different algorithms are different schedules for applying relaxations
  – SSSP needs priority scheduling for work efficiency
Algorithmic Variants == Scheduling

• Chaotic Relaxation:
  – Specify a non-priority scheduler
  • E.g. dChunkedFIFO

• Dijkstra:
  – Use Ordered Executor

• Delta-Stepping Like:
  – Specify OBIM priority scheduler

• Bellman-Ford
  – Push every edge in non-priority scheduler
  – Execute
  – Repeat #nodes times
Simple (PUSH) SSSP in Galois

```cpp
struct SSSP {
    void operator()(UpdateRequest& req,
        Galois::UserContext<UpdateRequest>& ctx) const {
        unsigned& data = graph.getData(req.second);
        if (req.first > data) return;

        for (Graph::edge_iterator ii = graph.edge_begin(req.second),
            ee = graph.edge_end(req.second); ii != ee; ++ii)
            relax_edge(data, ii, ctx);
    }
};
```
void relax_edge(unsigned src_data, Graph::edge_iterator ii,
   Galois::UserContext<UpdateRequest>& ctx) {
  GNode dst = graph.getEdgeDst(ii);
  unsigned int edge_data = graph.getEdgeData(ii);
  unsigned& dst_data = graph.getData(dst);
  unsigned int newDist = dst_data + edge_data;
  if (newDist < dst_data) {
    dst_data = newDist;
    ctx.push(std::make_pair(newDist, dst));
  }
}

Relax Edge (PUSH)
Specifying Schedule and Running

Galois::Graph::readGraph(graph, filename);
Galois::for_each(graph.begin(), graph.end(), Init());

using namespace Galois::WorkList;
typedef dChunkedLIFO<16> dChunk;
typedef OrderedByIntegerMetric<UpdateRequestIndexer,dChunk> OBIM;

graph.getData(*graph.begin()) = 0;
Galois::for_each(std::make_pair(0U, *graph.begin()), SSSP(),
    Galois::wl<OBIM>();

Load

WorkList

SSSP
Implementation Variants: Push V.S. Pull

- Simple optimization to control concurrency costs, locks, etc.
- Push: Look at node and update neighbors
- Pull: Look at neighbors and update self
- Pull seems “obviously” better, but in practice it depends on algorithm, scheduling, and data
struct SSSP {
    void operator()(GNode req, Galois::UserContext<UpdateRequest>& ctx) {

        // update self
        for (auto ii = graph.edge_begin(req), ee = graph.edge_end(req); ii != ee; ++ii) {
            auto edist = graph.getEdgeData(ii), ndist = graph.getData(graph.getEdgeDst(ii));
            if (edist + ndist < data)
                data = edist + ndist;
        }

        // push higher neighbors
        for (auto ii = graph.edge_begin(req), ee = graph.edge_end(req); ii != ee; ++ii) {
            auto edist = graph.getEdgeData(ii), ndist = graph.getData(graph.getEdgeDst(ii));
            if (ndist > data + edist)
                ctx.push(graph.getEdgeDst(ii));
        }
    }
};
SSSP Demo

• Start with chaotic algorithm and vary scheduling policy
  – Different policies give different amounts of work and scalability but all policies produce correct executions

• Policies
  – FIFO
  – ChunkedFIFO
    • FIFO of fixed size chunks of items
  – dChunkedFIFO
    • A ChunkedFIFO per package with stealing between ChunkedFIFOs
  – OBIM
    • Generalization of sequence of bags when sequence is sparse
Demo SSSP variants
Outline

• Current State of Parallel programming
• Amorphous data parallelism / Operator formulation
• High Level Galois
• Current state of the system
• Implementing Graph Analytic DSLs on Galois
• Practical Galois
• Extended Example: SSSP
• **Scheduling**
• Active research
Best Scheduling Policies

1. Exploit locality
2. Control the total amount of work
3. Use architecture-aware concurrent data structures that must scale to many threads
4. Vary according to application

Require sophisticated implementations
Contribution

• A language for scheduling policies
  – *Declarative*: sophisticated schedulers w/o writing code
  – *Effective*: performance comparable to hand-written and often better than previous schedulers

Get good performance without users writing (serial or concurrent) scheduling code
Rules and their composition

- FIFO
- LIFO
- ChunkedLIFO(k)
- Ordered(f)
- Random
- OrderedByMetric(g)
- ChunkedFIFO(k)

OrderedByMetric(g) → ChunkedFIFO(2) → FIFO

<table>
<thead>
<tr>
<th>Order</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
# Application-specific Policies

<table>
<thead>
<tr>
<th>App</th>
<th>Order</th>
<th>Scheduling Policy</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFP</td>
<td>FIFO</td>
<td>FIFO</td>
<td>[Goldberg88]</td>
</tr>
<tr>
<td>PFP</td>
<td>HL</td>
<td>OrderedByMetric($n. \cdot n.height$) FIFO</td>
<td>[Cherkassy95]</td>
</tr>
<tr>
<td>SSSP</td>
<td>D-stepping</td>
<td>OrderedByMetric($n. \cdot n.w / D \cdot r + \ldots$) FIFO</td>
<td>[Meyer98]</td>
</tr>
<tr>
<td>SSSP</td>
<td>Dijkstra</td>
<td>Ordered($a, b. a.w \cdot b.w$)</td>
<td>[Dijkstra59]</td>
</tr>
<tr>
<td>DMR</td>
<td>Local stack</td>
<td>ChunkedFIFO($k$) Local: LIFO</td>
<td>[Kulkarni08]</td>
</tr>
<tr>
<td>DT</td>
<td>BRIO</td>
<td>OrderedByMetric($p. p.rnd$) ChunkedFIFO($k$)</td>
<td>[Amenta03]</td>
</tr>
<tr>
<td>MATCHING</td>
<td>ABMP</td>
<td>OrderedByMetric($n. n.lv$) FIFO</td>
<td>[ABMP91]</td>
</tr>
<tr>
<td>BP</td>
<td>RBP</td>
<td>Ordered($a, b. a.old-a.new \cdot b. old-b.new$)</td>
<td>[Elidan06]</td>
</tr>
</tbody>
</table>
Synthesis

• Generate scheduler implementation from specification

• Assemble atoms that implement individual rules into final implementation
  – Tricky depending on overall behavior that needs to be maintained
Outline

• Current State of Parallel programming
• Amorphous data parallelism / Operator formulation
• High Level Galois
• Current state of the system
• Implementing Graph Analytic DSLs on Galois
• Practical Galois
• Extended Example: SSSP
• Scheduling
• **Active research**
Interesting Problems

• Algorithm implementation synthesis
• GPU execution
• Hybrid execution
• Hardware mapping
• Development tools
• Distributed memory
Elixir: Synthesizing parallel graph algorithms

- **Elxir DSL:**
  - Relational data-structure spec
  - Operators as rewrite rules
  - Schedule specified declaratively

- Compiler synthesizes fixpoint computation

- Inserts synchronization automatically

- Allows quick experimentation with many algorithm variants

```
Graph [ nodes(node : Node, dist : int )
        edges(src : Node, dst : Node, wt : int ) ]

source : Node

initDist = [ nodes(node a, dist d) ] →
           [ d = if (a == source) 0 else ∞ ]

relaxEdge = [ nodes(node a, dist ad)
              nodes(node b, dist bd)
              edges(src a, dst b, wt w)
              ad + w < bd ] →
             [ bd = ad + w ]

init = foreach initDist
sssp = iterate relaxEdge ∪ sched
main = init ; sssp
```

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Schedule specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra</td>
<td>sched = metric ad ≫ group b</td>
</tr>
<tr>
<td>Label-correcting</td>
<td>sched = group b ≫ unroll 2 ≫ approx metric ad</td>
</tr>
<tr>
<td>Δ-stepping-style</td>
<td>DELTA : unsigned int () sched = metric () (ad + w) / DELTA</td>
</tr>
<tr>
<td>Bellman-Ford</td>
<td>NUM_NODES : unsigned int</td>
</tr>
<tr>
<td></td>
<td>// override sssp</td>
</tr>
<tr>
<td></td>
<td>sssp = for i = 1..(NUM_NODES - 1)</td>
</tr>
<tr>
<td></td>
<td>step</td>
</tr>
<tr>
<td></td>
<td>step = foreach relaxEdge</td>
</tr>
</tbody>
</table>
SSSP: synthesized vs. handwritten

(a) FLA runtimes

(c) FLA runtime distribution

- Input graph: Florida road network, 1M nodes, 2.7M edges
Irregular Algorithms on the GPU

- GPUs offer hundreds of concurrent threads for computation
- Discrete GPUs possess higher memory bandwidths and allow more throughput than CPUs
- GPUs can be used solely or share work with the CPU
Key Challenges

- GPU hardware is optimized for *regular* code
- Dynamic scheduling is hard because GPU threads are hardware-scheduled
- Limited synchronization primitives with little to no communication allowed between threads
- No standard library, code reuse is hard
- Autotuning necessary for performance portability
The LonestarGPU Suite and beyond

- LonestarGPU 2.0 Suite contains fast implementations of many irregular algorithms
  - BFS, SSSP, MST, BH, PTA, SP, DMR
- LSG-next contains more algorithms and autotuning support
- Written by hand currently
- Working on a code generator for irregular algorithms
Heterogeneous execution

• Distribute work between multiple devices; CPU (host) and accelerator (GPU, Xeon-Phi, FPGA etc.)

• Challenges:
  – Work division – how to divide workload between devices to minimize communication
  – Communication – reduce communication overhead by combining/overlapping
  – Data representation – preferred layouts different on different devices
Integrated GPUs.

- Simpler problem:
  - Low communication overhead, Atomics b/w CPU-GPU
- Constrained:
  - GPUs not as powerful as discrete GPUs
  - Memory limit (less than 1G)
- Use runtime-profiling to determine work-distribution
  - Adaptive execution addresses load imbalance
Misc

• Hardware transactional memory
  – How does it compare to Galois’s conflict checking
  – How can it be improved to be used as basis for non-trivial runtimes

• Performance Prediction
  – Can we measure scaling without having to first write code?
Distributed Memory

• Source compatible DSM Galois
• Handles non-vertex programs
  – Add remove nodes and edges
Conclusions

• Yesterday:
  – Computation-centric view of parallelism

• Today:
  – Data-centric view of parallelism
  – Operator formulation of algorithms
  – Permits a unified view of parallelism and locality in algorithms
  – Joe/Stephanie programming model
  – Galois system is an implementation

• Tomorrow:
  – DSLs for different applications
  – Layer on top of Galois

Parallel program = Operator + Schedule + Parallel data structure
More information

• Website
  – http://iss.ices.utexas.edu

• Download
  – Galois system for multicores
  – Lonestar benchmarks (CPU and GPU)
  – All our papers