Implementing the Operator Formulation
Operator formulation of algorithms

- **Active node/edge:**
  - site where computation is needed

- **Operator:**
  - local view of algorithm
  - computation at active node/edge
  - neighborhood: data structure elements read and written by operator

- **Schedule:**
  - global view of algorithm
  - unordered algorithms:
    - active nodes can be processed in any order
    - all schedules produce the same answer but performance may vary
  - ordered algorithms:
    - problem-dependent order on active nodes
von Neumann programming model

Program Execution

Program counter

State update: assignment statement
(local view)

Schedule: control-flow constructs
(global view)

von Neumann bottleneck [Backus 79]
Data-centric programming model

Active nodes

Program Execution

State update: operator
(local view)

Schedule: ordering between active nodes
(global view)
TAO terminology for algorithms

- **Active nodes**
  - Topology-driven algorithms
    - Algorithm is executed in rounds
    - In each round, all nodes/edges are initially active
    - Iterate till convergence
  - Data-driven algorithms
    - Some nodes/edges initially active
    - Applying operator to active node may create new active nodes
    - Terminate when no more active nodes/edges in graph

- **Operator**
  - Morph: may change the graph structure by adding/removing nodes/edges
  - Label computation: updates labels on nodes/edges w/o changing graph structure
  - Reader: makes no modification to graph
Algorithms we have studied

• Mesh generation
  – Delaunay mesh refinement: data-driven, unordered

• SSSP
  – Chaotic relaxation: data-driven, unordered
  – Dijkstra: data-driven, ordered
  – Delta-stepping: data-driven, ordered
  – Bellman-Ford: topology-driven, unordered

• Machine learning
  – Page-rank: topology-driven, unordered
  – Matrix completion using SGD: topology-driven, unordered

• Computational science
  – Stencil computations: topology-driven, unordered
Parallelization of Delaunay mesh refinement

- Each mutable data structure element (node, triangle,..) has an ID and a mark
- What threads do:
  - $\text{nhoodElements} \leftarrow \{\}$
  - Get active element from worklist, acquire its mark and add element to $\text{nhoodElements}$
  - Iteratively expand neighborhood, and for each data structure element in neighborhood, acquire its mark and add element to $\text{nHoodElement}$
  - When neighborhood expansion is complete, apply operator
  - If there are newly created active elements, add them to the worklist
  - Release marks of elements in $\text{nhoodElements}$ set
  - If any mark acquisition fails, release marks of all elements in $\text{nhoodElements}$ and put active element back on worklist
- Optimistic (speculative) parallelization
Parallelization of stencil computation

• What threads do:
  – there are no conflicts so each thread just applies operator to its active nodes

• Good policy for assigning active nodes to threads:
  – divide grid into 2D blocks and assign one block to each thread
  – this promotes locality

• Static parallelization: no need for speculation
Questions

• Why can we parallelize some algorithms statically while other algorithms have to parallelized at run time using optimistic parallelization?

• Are there parallelization strategies other than static and optimistic parallelization?

• What is the big picture?
Binding time

- Useful concept in programming languages
  - When do you have the information you need to make some decision?
- Example: type-checking
  - Static type-checking: Java, ML
    - type information is available in the program
    - type correctness can be checked at compile-time
  - Dynamic type-checking: Python, Matlab
    - types of objects are known only during execution
    - type correctness must be checked at runtime
- Binding time for parallelization
  - When do we know the active nodes and neighborhoods?
Parallelization strategies: Binding Time

<table>
<thead>
<tr>
<th>Compile-time</th>
<th>Static parallelization (stencil codes, FFT, dense linear algebra)</th>
</tr>
</thead>
<tbody>
<tr>
<td>After input is given</td>
<td>Inspector-executor (sparse Cholesky, Bellman-Ford)</td>
</tr>
<tr>
<td>During program execution</td>
<td>Interference graph (DMR, chaotic SSSP)</td>
</tr>
<tr>
<td>After program is finished</td>
<td>Optimistic Parallelization (Time-warp)</td>
</tr>
</tbody>
</table>

“The TAO of parallelism in algorithms” Pingali et al, PLDI 2011
Inspector-Executor

- Figure out what can be done in parallel
  - after input has been given, but
  - before executing the actual algorithm
- Useful for topology-driven algorithms on graphs
  - algorithm is executed in many rounds
  - overhead of preprocessing can be amortized over many rounds
- Basic idea:
  - determine neighborhoods at each node
  - build interference graph
  - use graph coloring to find sets of nodes that can be processed in parallel without synchronization
- Example:
  - sparse Cholesky factorization
  - we will use Bellman-Ford (in practice Bellman-Ford is implemented differently)
Inspector-Executor

Neighborhoods of activities

Interference graph
Inspector-Executor

Neighborhoods of activities

- Nodes in a set can be done in parallel
- Use barrier synchronization between sets

\{D\}
\{E, A\}
\{F\}
\{G, B\}
\{H, C\}
Graph representations: how to store graphs in memory
Graph-matrix duality

**Graph (V,E) as a matrix**
- Choose an ordering of vertices
- Number them sequentially
- Fill in \(|V| \times |V|\) matrix
  - \(A(i,j)\) is \(w\) if graph has edge from node \(i\) to node \(j\) with label \(w\)
- Called *adjacency matrix* of graph
- Edge \((u \rightarrow v)\):
  - \(v\) is *out-neighbor* of \(u\)
  - \(u\) is *in-neighbor* of \(v\)

**Observations:**
- Diagonal entries: weights on self-loops
- Symmetric matrix \(\leftrightarrow\) undirected graph
- Lower triangular matrix \(\leftrightarrow\) no edges from lower numbered nodes to higher numbered nodes
- Dense matrix \(\leftrightarrow\) clique (edge between every pair of nodes)
Sparse graphs

- **Terminology:**
  - Degree of node: number of edges connected to it
  - (Average) diameter of graph: average number of hops between two nodes

- **Power-law graphs**
  - small number of very high degree nodes (see next slide for example)
  - low diameter
    - “six degrees of separation” (Karinthy 1929, Milgram 1967), on Facebook, it is 4.74
  - typical of social network graphs like the Internet graph or the Facebook graph

- **Uniform-degree graphs**
  - nodes have roughly same degree
  - high diameter
  - road networks, IC circuits, finite-element meshes

- **Random (Erdös-Rényi) graphs**
  - constructed by random insertion of edges
  - mathematically interesting but few real-life examples
Airline route map: power-law graph
Road map: uniform-degree graph
Three storage formats: CSR, CSC, COO

Coordinate storage

```
1 2 4 5 3 1 2 3 4 5
2 4 6 6 5 3 3 2 3 4
16 12 20 4 15 13 10 4 9 7
```

Compressed sparse row

```plaintext
rp: [1 3 5 7 9 11 11]
ci: [2 3 3 4 2 5 3 6 4 6 0]
ai: [16 13 10 12 4 14 9 20 7 4]
```

Compressed sparse column

```plaintext
cp: [1 1 3 6 8 9 11]
ri: [1 3 1 2 4 2 5 3 4 5 0]
ai: [16 4 13 10 9 12 7 14 20 4]
```

Labels on nodes are stored in a separate vector (not shown)
Adjacency list representation

Permits you to add and remove edges from graph
Deleting edges: often it is more efficient to just to mark an edge as deleted rather than delete it physically from the list

From: https://www.thecrazyprogrammer.com
## Graph sizes

<table>
<thead>
<tr>
<th>Inputs</th>
<th>rmat28</th>
<th>kron30</th>
<th>clueweb12</th>
<th>wdc12</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>V</td>
<td>$</td>
<td>268M</td>
<td>1073M</td>
</tr>
<tr>
<td>$</td>
<td>E</td>
<td>$</td>
<td>4B</td>
<td>11B</td>
</tr>
<tr>
<td>$</td>
<td>E</td>
<td>/</td>
<td>V</td>
<td>$</td>
</tr>
<tr>
<td>Size (CSR)</td>
<td>35GB</td>
<td>136GB</td>
<td>325GB</td>
<td>986GB</td>
</tr>
</tbody>
</table>
Shared-memory Galois System
Galois system

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
  – Joe: application code in sequential C++
    • Galois set iterator for highlighting opportunities for exploiting ADP
Hello graph Galois Program

```cpp
#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;
}  ```
Parallel execution of Galois programs

- **Application (Joe) program**
  - Sequential C++
  - Galois set iterator: for each
    - New elements can be added to set during iteration
    - Optional scheduling specification (cf. OpenMP)
    - Highlights opportunities in program for exploiting amorphous data-parallelism

- **Runtime system**
  - Ensures serializability of iterations
  - Execution strategies
    - Optimistic parallelization
    - Interference graphs
PERFORMANCE STUDIES
"Navigating the maze of graph analytics frameworks" Nadathur et al SIGMOD 2014
### Galois: Performance on SGI Ultraviolet

#### 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.

<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>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>
FPGA Tools

Maze Router Execution Time

Moctar & Brisk, “Parallel FPGA Routing based on the Operator Formulation”
DAC 2014
Summary

• Finding parallelism in programs
  – binding time: when do you know the active nodes and neighborhoods
  – range of possibilities from static to optimistic
  – optimistic parallelization can be used for all algorithms but in general, early binding is better

• Shared-memory Galois implements some of these parallelization strategies
  – focus: irregular programs