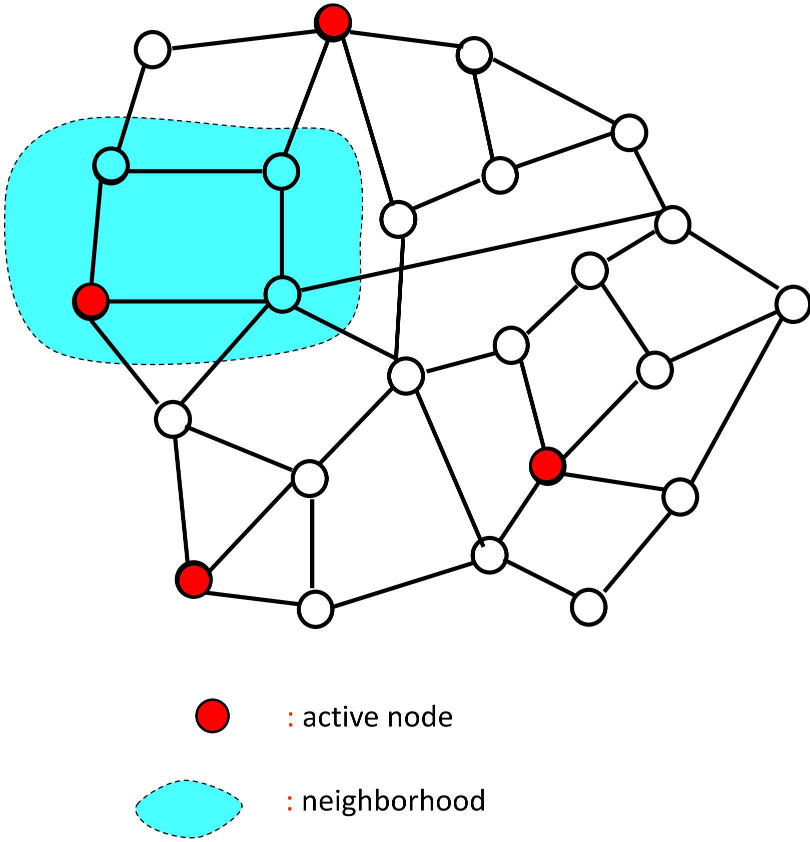
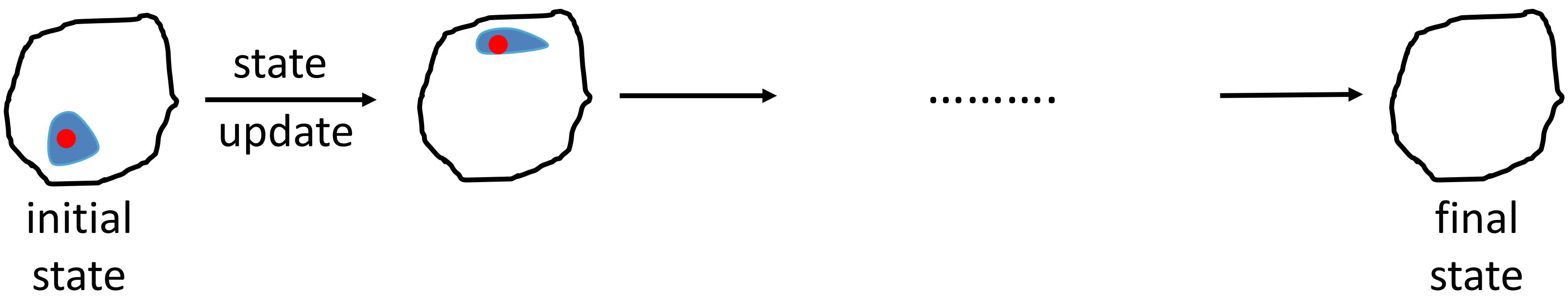
Implementing the Operator Formulation

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

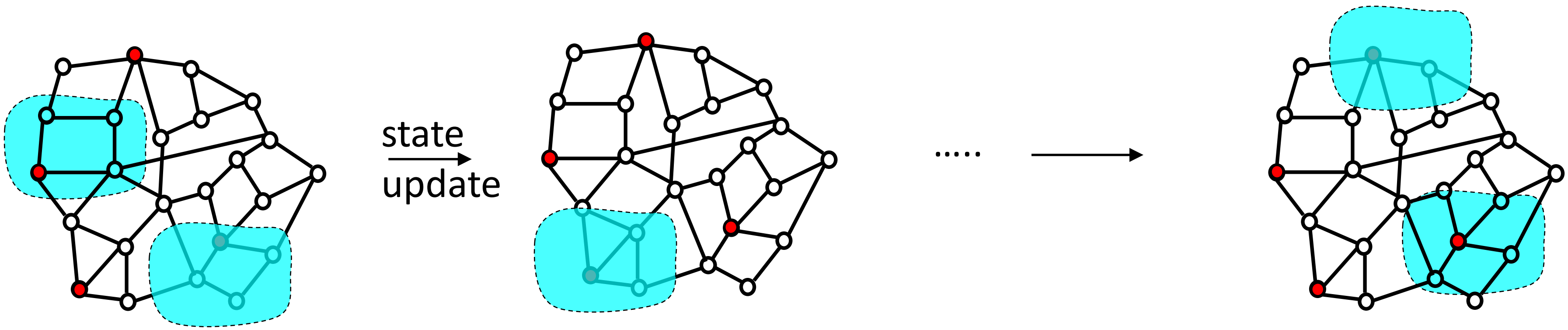


Program Execution

- where → Program counter
- what → State update: assignment statement (local view)
- when → Schedule: control-flow constructs (global view)

von Neumann bottleneck [Backus 79]

Data-centric programming model



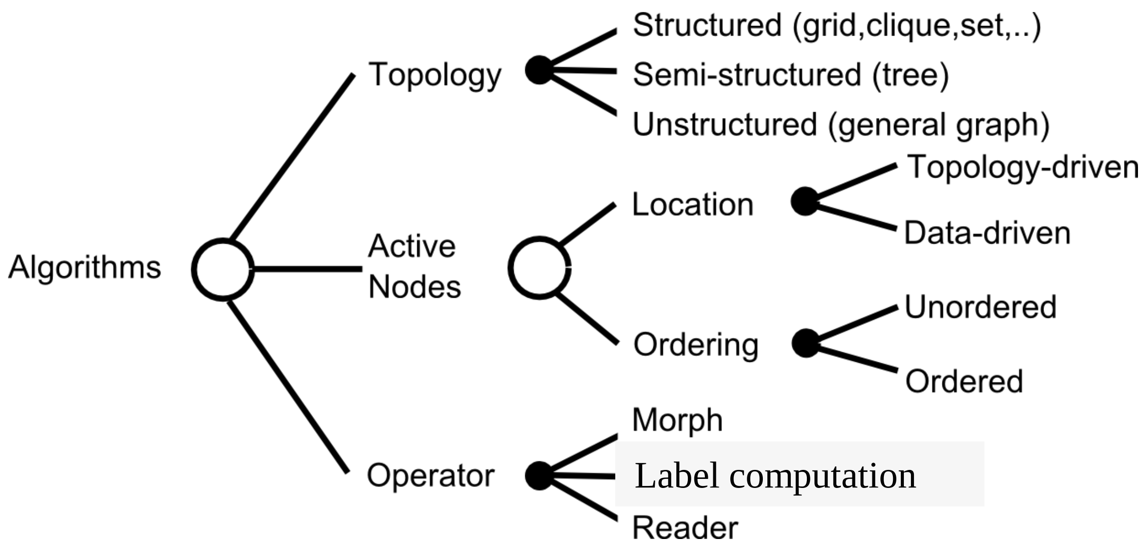
Program Execution

where — Active nodes

what — State update: operator
(local view)

when — Schedule: ordering between active nodes
(global view)

TAO terminology for algorithms

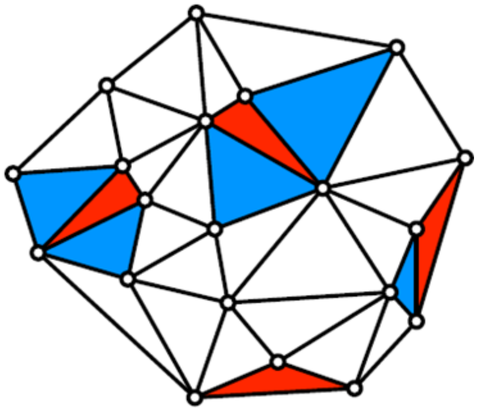


- Active nodes
  - Topology-driven algorithms
    - Algorithm is executed in rounds
    - In each round, all nodes/edges are initially active
    - Iterate till convergence
  - Data-driven algorithms
    - Some nodes/edges initially active
    - Applying operator to active node may create new active 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

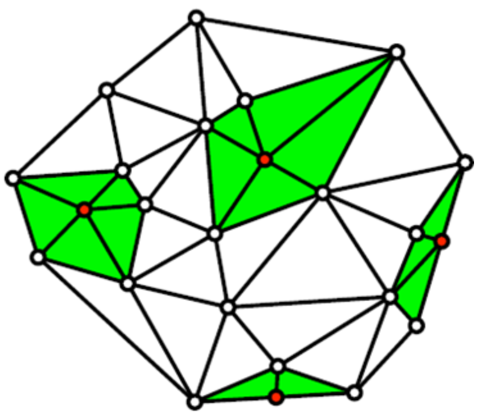
Algorithms we have studied

- 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



Before

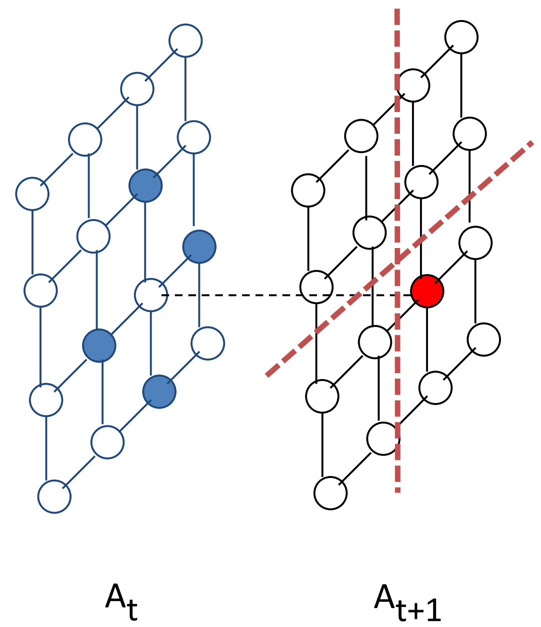


After

- Each mutable data structure element (node, triangle,..) has an ID and a mark
- What threads do:
  - nhoodElements ← {}
  - Get active element from worklist, acquire its mark and add element nhoodElements
  - Iteratively expand neighborhood, and for each data structure element in neighborhood, acquire its mark and add element to 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 nhoodElements set
  - If any mark acquisition fails, release marks of all elements in 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



Jacobi iteration, 5-point stencil

```
//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)
```

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?

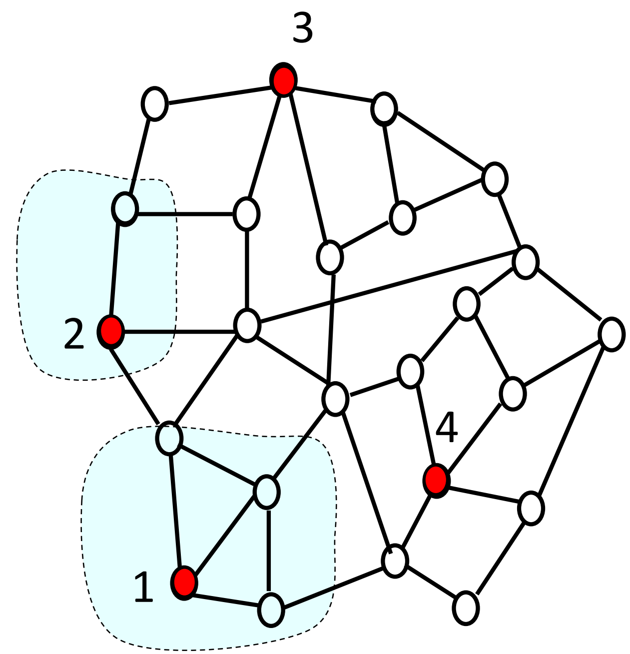
Binding time

- 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

Compile-time — **Static parallelization (stencil codes, FFT, dense linear algebra)**

After input is given — **Inspector-executor (sparse Cholesky, Bellman-Ford)**



During program execution — **Interference graph (DMR, chaotic SSSP)**

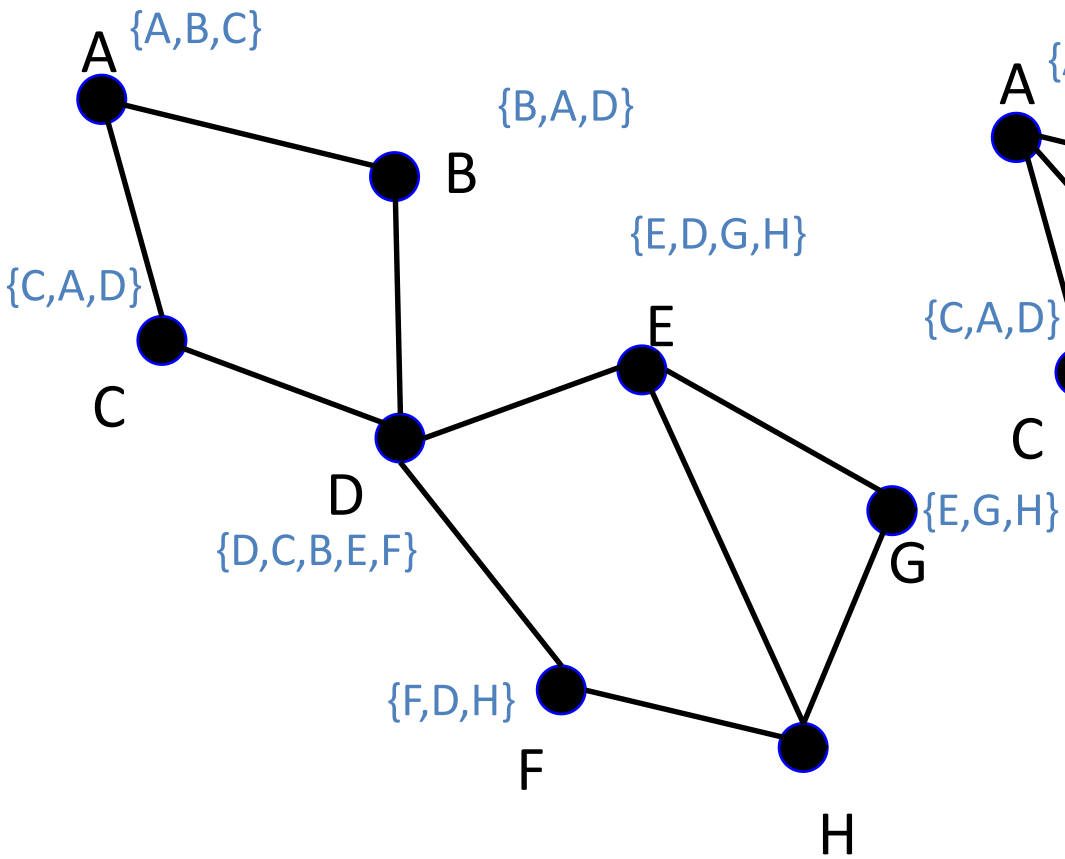
After program is finished — **Optimistic Parallelization (Time-warp)**

"The TAO of parallelism in algorithms" Pingali et al, PLDI 2011

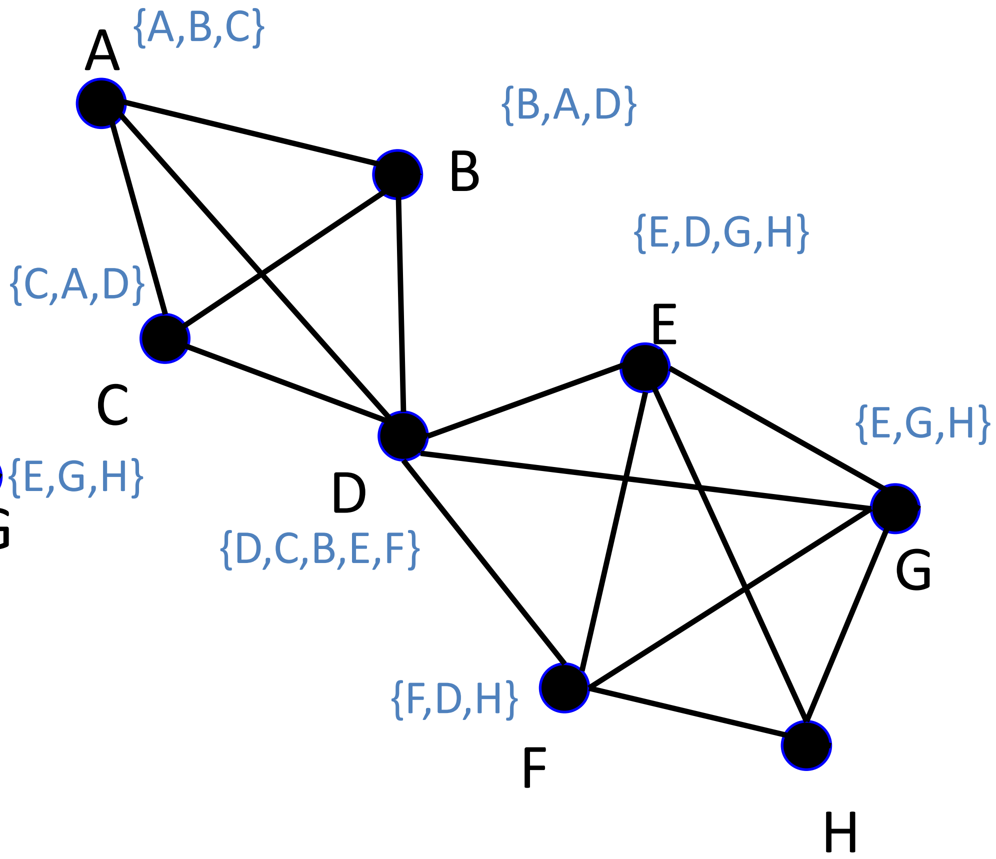
Inspector-Executor

- 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

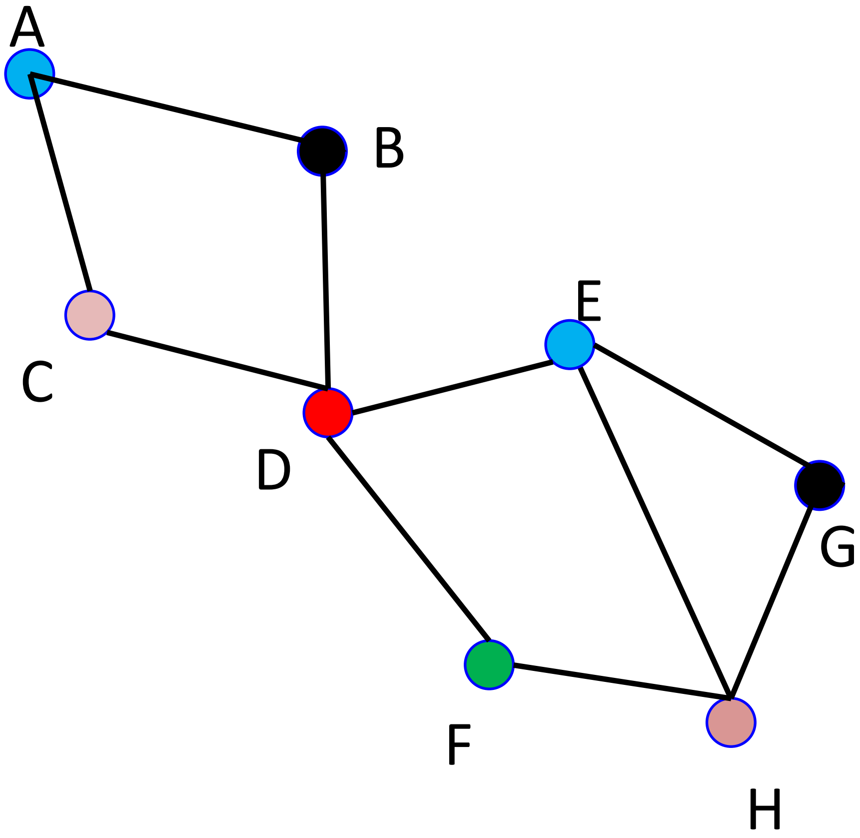


Neighborhoods of activities



Interference graph

Inspector-Executor



{D}
{E,A}
{F}
{G,B}
{H,C}

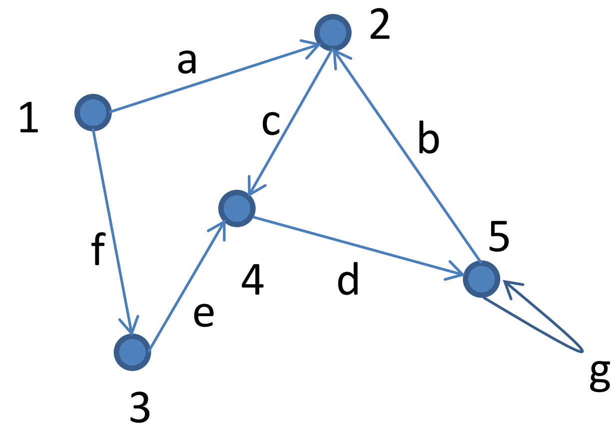
Neighborhoods of activities

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

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|x|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 $\leftarrow\rightarrow$ undirected graph
  - Lower triangular matrix $\leftarrow\rightarrow$ no edges from lower numbered nodes to higher numbered nodes
  - Dense matrix $\leftarrow\rightarrow$ clique (edge between every pair of nodes)



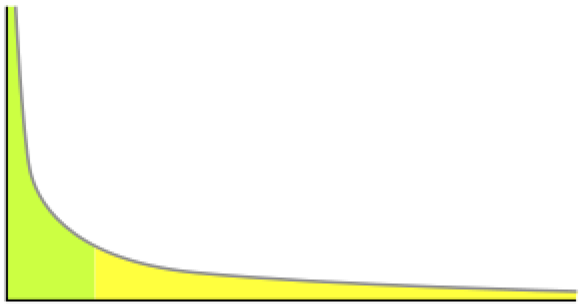
to

from

$$
\begin{array}{cc}
 & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} 0 & a & f & 0 & 0 \\ 0 & 0 & 0 & c & 0 \\ 0 & 0 & 0 & e & 0 \\ 0 & 0 & 0 & 0 & d \\ 0 & b & 0 & 0 & g \end{bmatrix}
\end{array}
$$

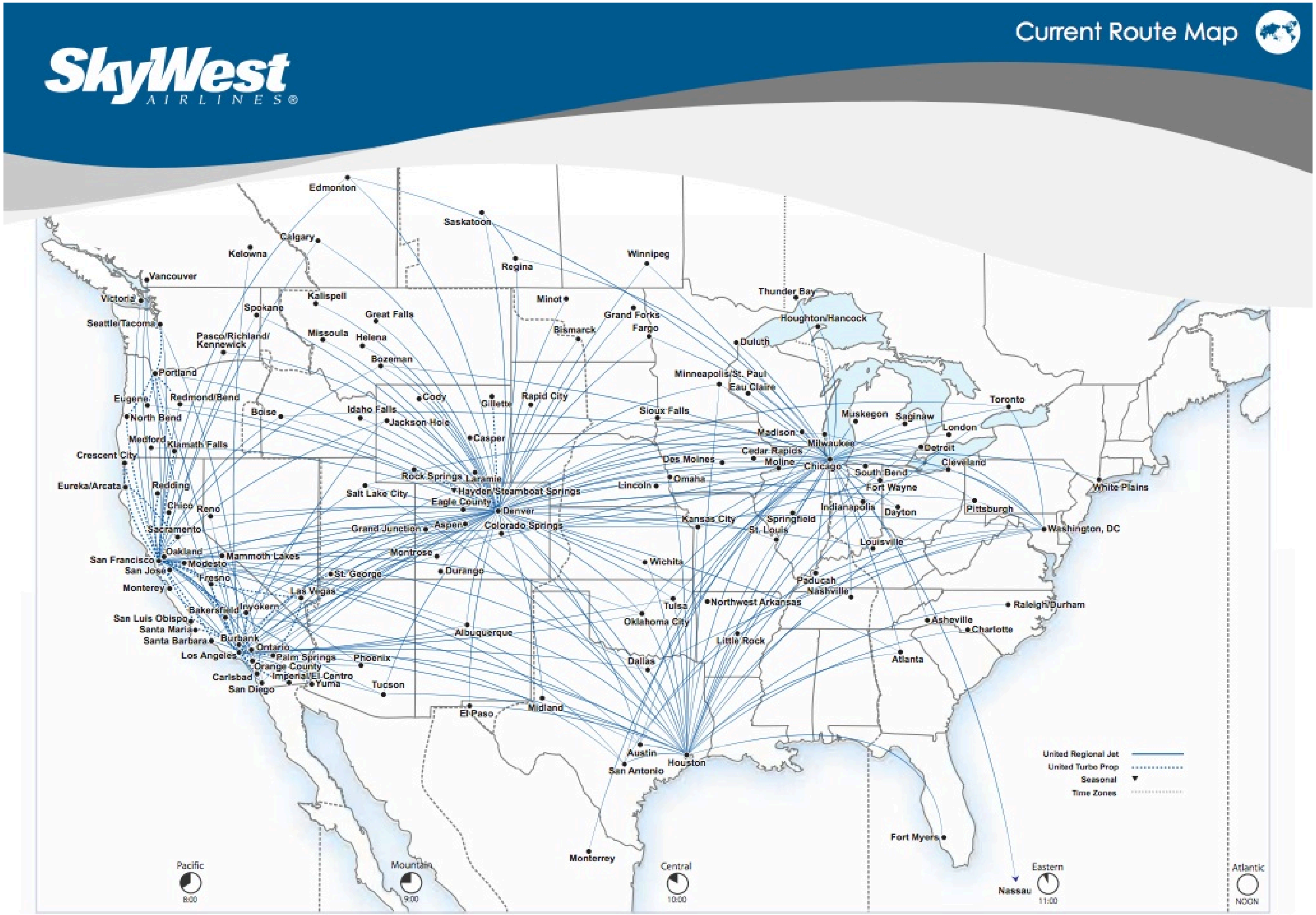
Sparse graphs

- 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-Rènyi) graphs
  - constructed by random insertion of edges
  - mathematically interesting but few real-life examples



Node degree distribution of power-law graphs

Airline route map: power-law graph



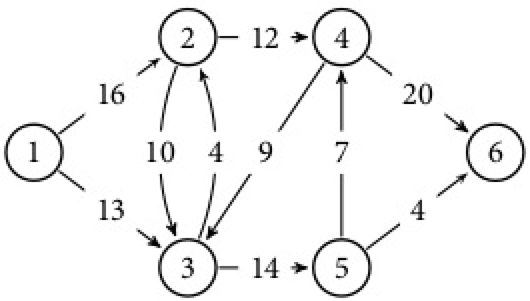
UNITED EXPRESS

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

rp	1	3	5	7	9	11	11			
ci	2	3	3	4	2	5	3	6	4	6	∅
ai	16	13	10	12	4	14	9	20	7	4	

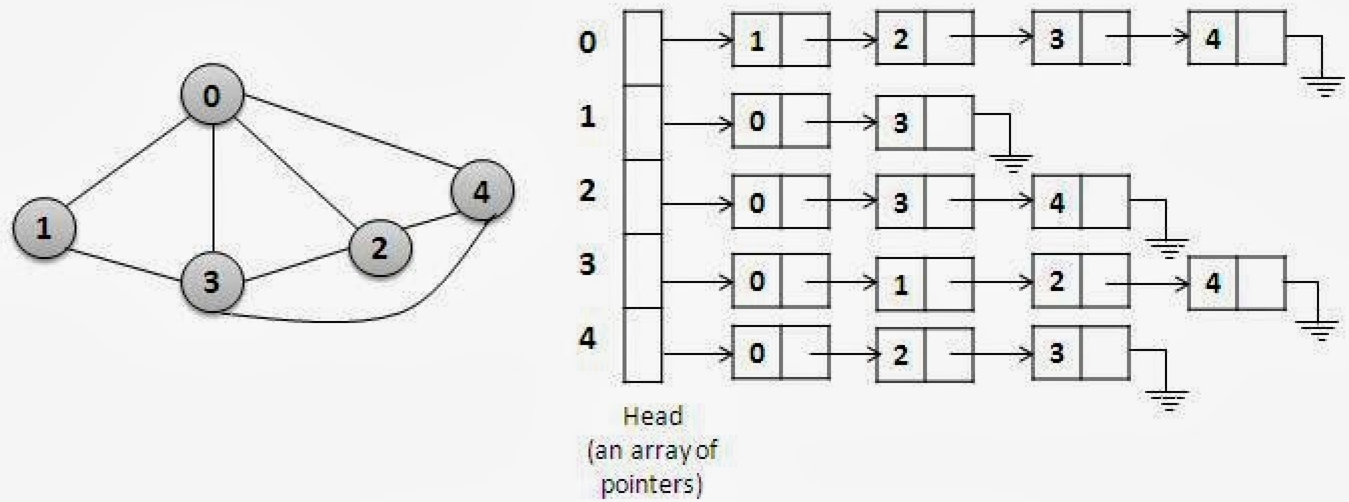
$$\begin{bmatrix} 0 & 16 & 13 & 0 & 0 & 0 \\ 0 & 0 & 10 & 12 & 0 & 0 \\ 0 & 4 & 0 & 0 & 14 & 0 \\ 0 & 0 & 9 & 0 & 0 & 20 \\ 0 & 0 & 0 & 7 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Compressed sparse column

cp	1	1	3	6	8	9	11				
ri	1	3	1	2	4	2	5	3	4	5	∅
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



Adjacency List Representation of Graph

From: https://www.thecrazyprogrammer.com

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

Graph sizes

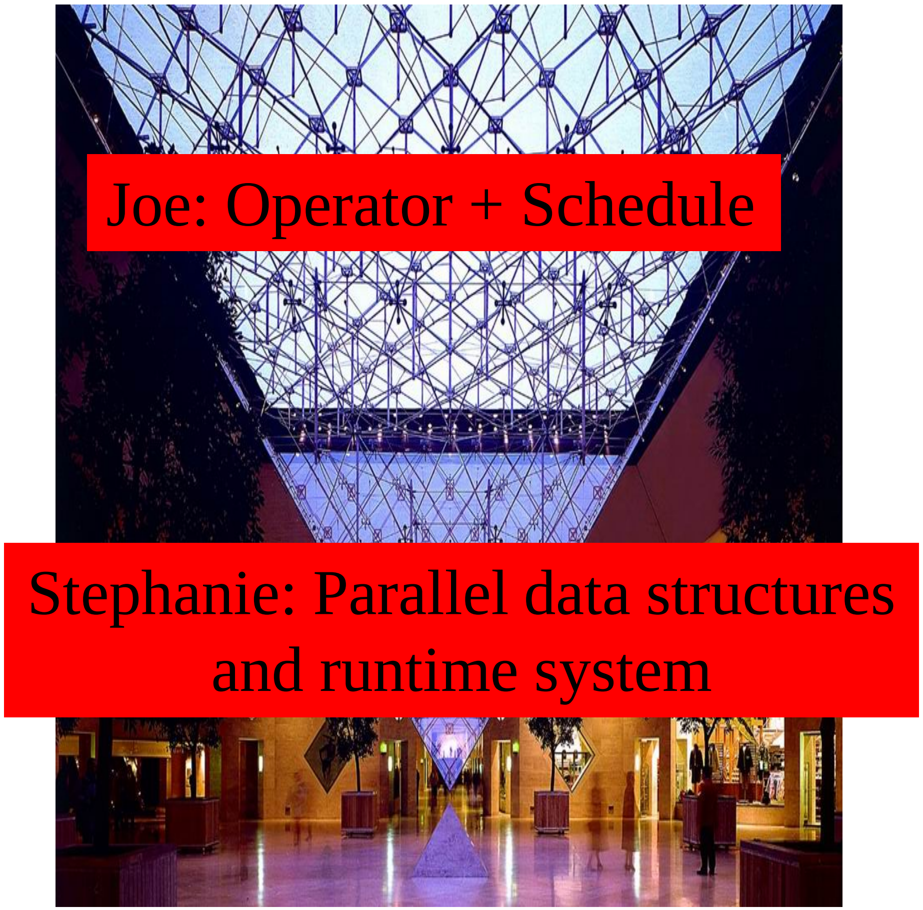
Inputs	rmat28	kron30	clueweb12	wdc12
|V|	268M	1073M	978M	3,563M
|E|	4B	11B	42B	129B
|E|/|V|	16	16	44	36
Size (CSR)	35GB	136GB	325GB	**986GB**

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

```
#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;
}
```

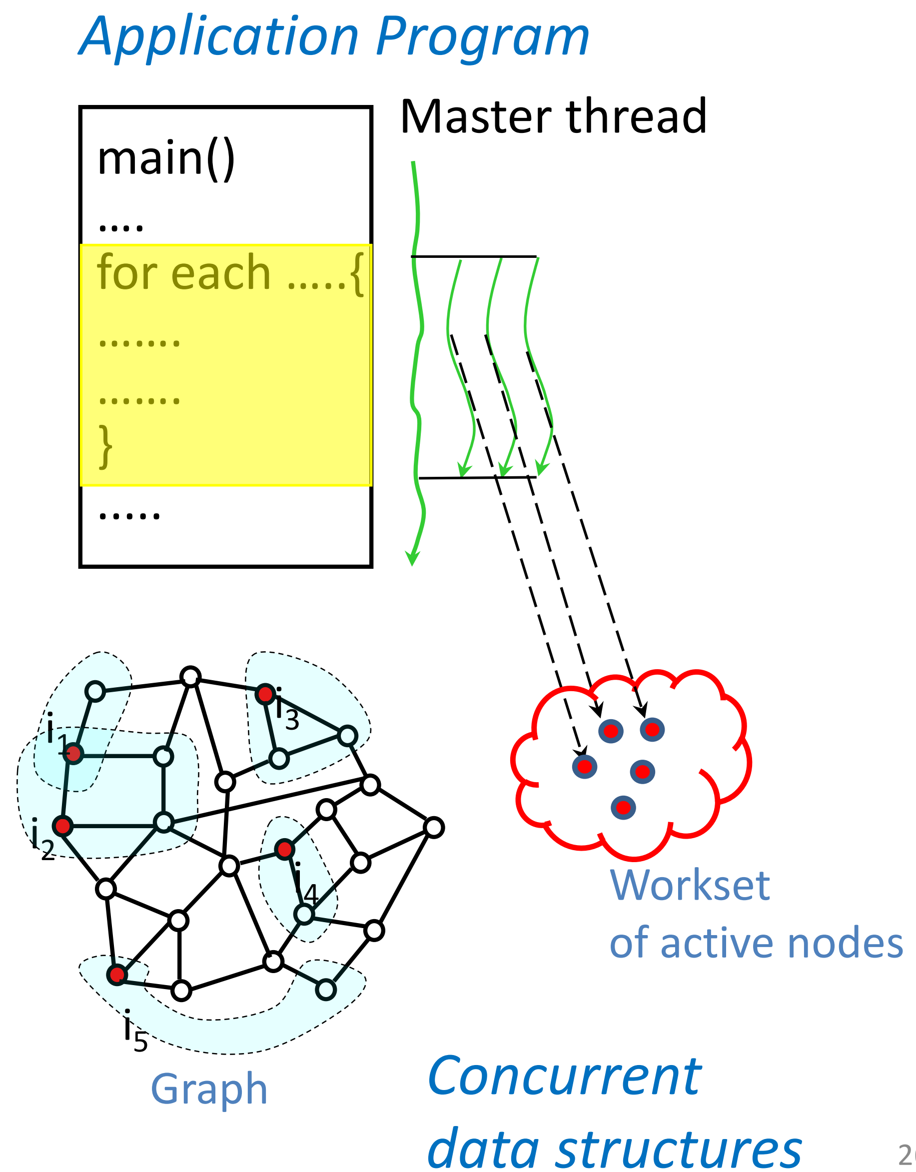
Data structure Declarations

Operator

Galois Iterator

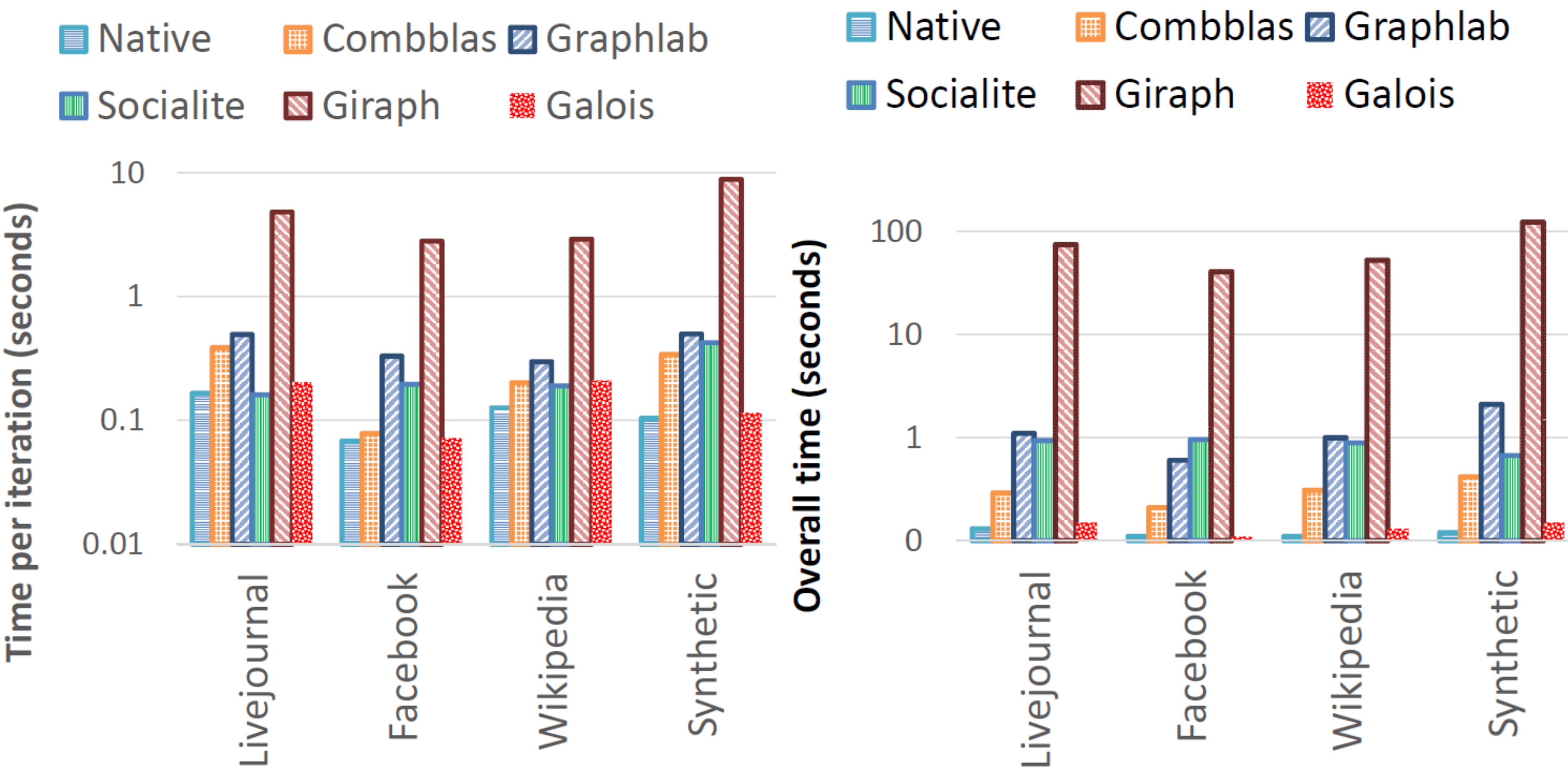
Parallel execution of Galois programs

- 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

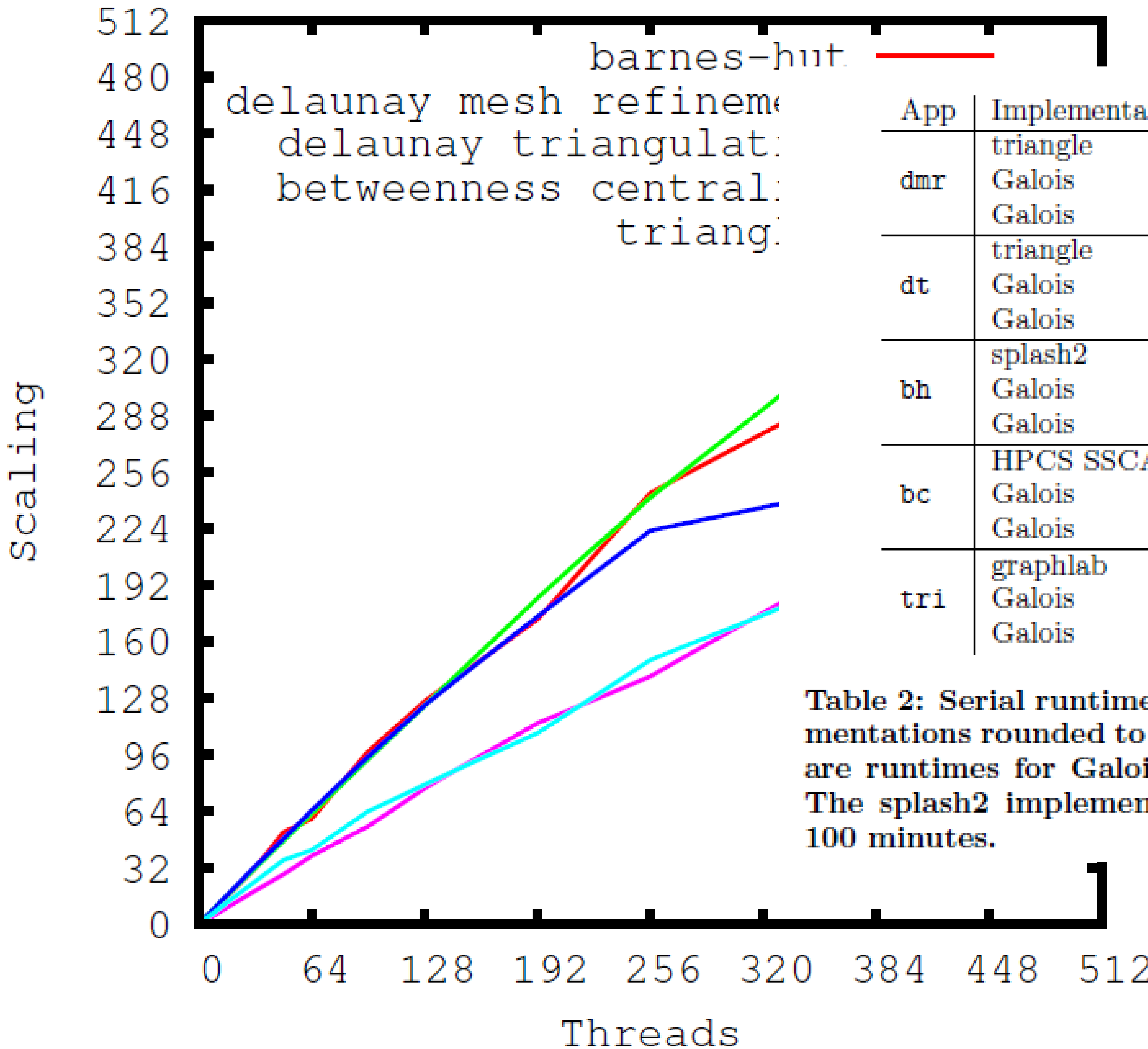
Intel Study: Galois vs. Graph Frameworks



(a) PageRank (b) Breadth-First Search

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

Galois: Performance on SGI Ultraviolet

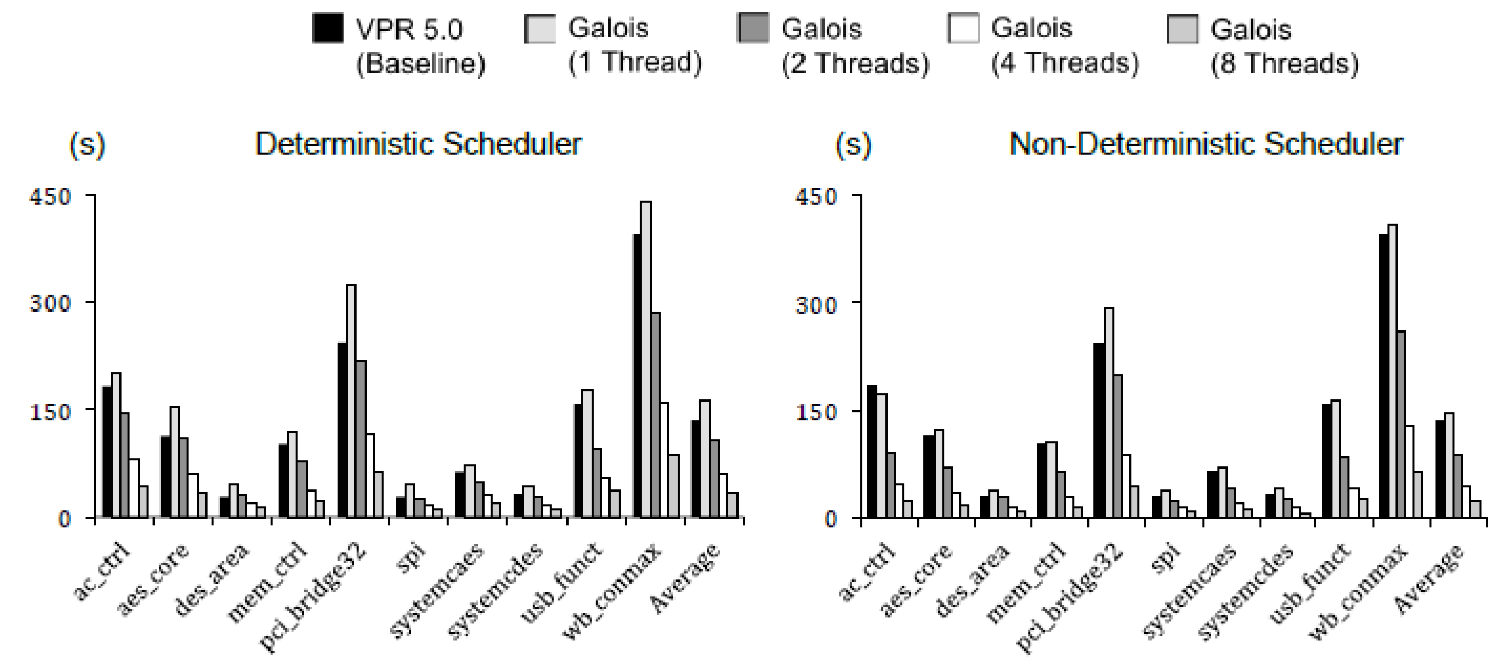


App	Implementation	Threads	Time (s)
dmr	triangle	1	96
	Galois	1	155.7
	Galois	512	0.37
dt	triangle	1	1185
	Galois	1	56.6
	Galois	512	0.18
bh	splash2	1	>6000
	Galois	1	1386
	Galois	512	3.55
bc	HPCS SSCA	1	6720
	Galois	1	5394
	Galois	512	21.6
tri	graphlab	2	531
	Galois	1	7.03
	Galois	512	0.028

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.

FPGA Tools

Maze Router Execution Time



Averages				
	VPR 5.0	134.6 seconds	VPR 5.0	134.6 seconds
	Galois (1 Thread)	162.4 seconds	Galois (1 Thread)	145.3 seconds
	Galois (2 Threads)	106.6 seconds	Galois (2 Threads)	88.8 seconds
	Galois (4 Threads)	59.2 seconds	Galois (4 Threads)	43.0 seconds
	Galois (8 Threads)	33.7 seconds	Galois (8 Threads)	22.6 seconds

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