

Parallel Programming Needs Data-centric Foundations

Keshav Pingali
The University of Texas at Austin

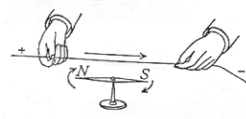
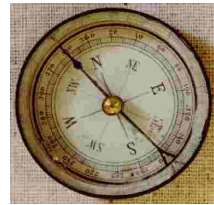
Problem Statement

- 40-50 years of work on parallel programming
 - programming models: PRAM, BSP,...
 - programming languages and libraries: MPI, OpenMP, HPF,...
- Parallel programming is still a research problem
 - case studies: matrix computations, stencil computations, FFTs etc.
 - few general insights, particularly for irregular applications
- Thesis: we need a science of parallel programming
 - analysis: framework for thinking about parallelism in applications
 - synthesis: produce an efficient parallel implementation of applications



“The Alchemist” Cornelius Bega (1663)

Analogy: science of electro-magnetism



Seemingly
unrelated phenomena



Maxwell's Equations

$$\oiint \mathbf{E} \cdot \mathbf{n} \, dS = \frac{q}{\epsilon_0}$$

Gauss's Law



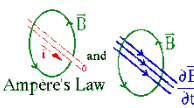
$$\oiint \mathbf{B} \cdot \mathbf{n} \, dS = 0$$

(no monopoles)



$$\oint \mathbf{B} \cdot d\mathbf{l} = \mu_0 \left[i + \epsilon_0 \frac{d\Phi_E}{dt} \right]$$

Ampere's Law



$$\oint \mathbf{E} \cdot d\mathbf{l} = -\frac{d\Phi_B}{dt}$$

Faraday's Law



$$\nabla \cdot \mathbf{E} = \frac{\rho}{\epsilon_0}$$

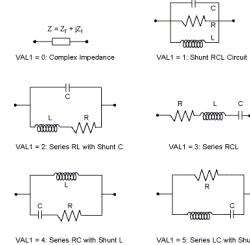
$$\nabla \times \mathbf{B} = \mu_0 \left[\mathbf{j} - \epsilon_0 \frac{\partial \mathbf{E}}{\partial t} \right]$$

$$\nabla \cdot \mathbf{B} = 0$$

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}$$

(Differential Forms)

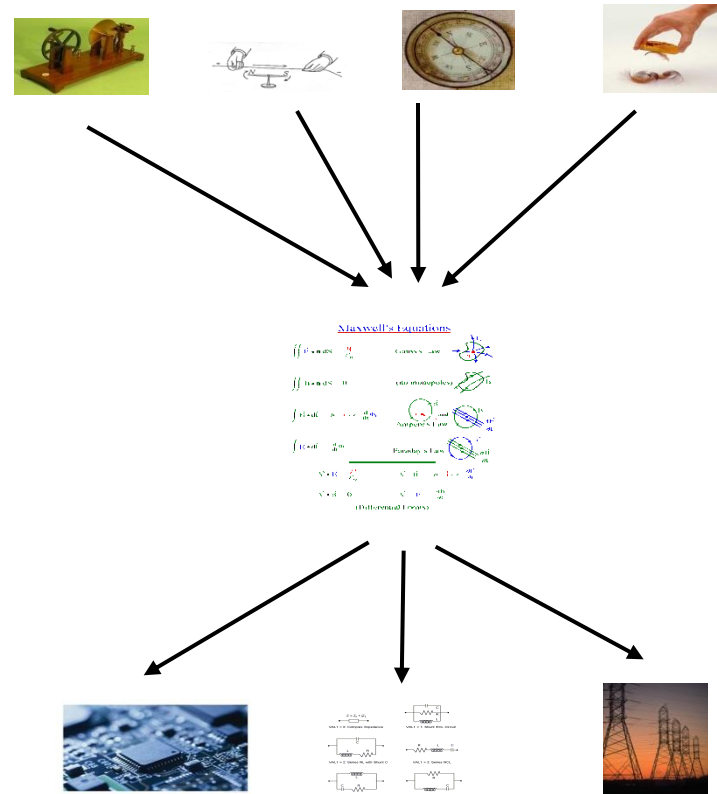
Unifying abstractions

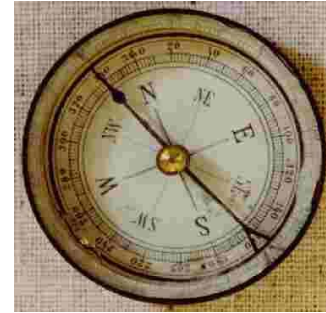


Specialized models
that exploit structure

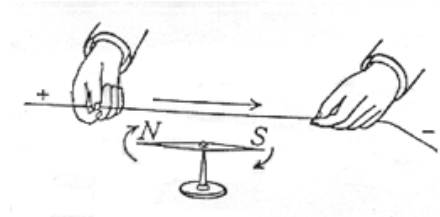
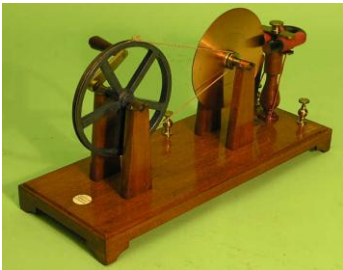
Organization of talk

- **Seemingly unrelated parallel algorithms and data structures**
 - Stencil codes
 - Delaunay mesh refinement
 - Event-driven simulation
 - Graph reduction of functional languages
 -
- **Unifying abstractions**
 - Operator formulation of algorithms
 - Amorphous data-parallelism
 - Galois programming model
 - Baseline parallel implementation
- **Specialized implementations that exploit structure**
 - Structure of algorithms
 - Optimized compiler and runtime system support for different kinds of structure
- **Ongoing work**





Seemingly unrelated algorithms

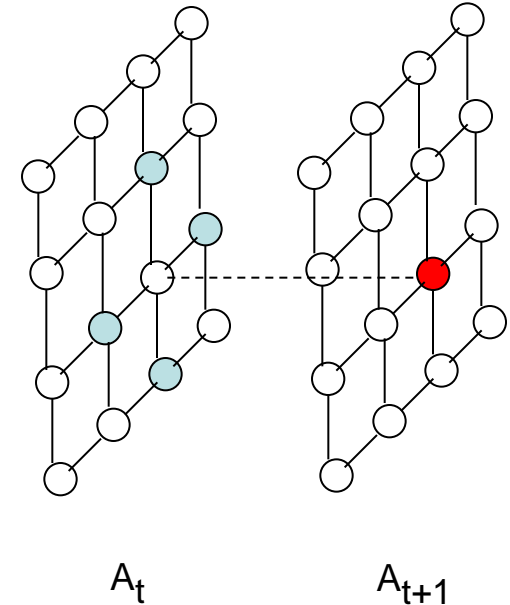


Examples

Application/domain	Algorithm
Graphics	Mesh Generation/refinement/partitioning
Compilers	Iterative and elimination-based dataflow algorithms
Functional interpreters	Graph reduction, static and dynamic dataflow
Graph algorithms	Maxflow, MST
Event-driven simulation	Chandy-Misra-Bryant, Jefferson Timewarp
AI	Message-passing algorithms
PDE solvers	Finite-differences, finite-elements
Social network analysis	Clustering, betweenness-centrality

Stencil computation: Jacobi iteration

- Finite-difference method for solving pde's
 - discrete representation of domain: grid
- Values at interior points are updated using values at neighbors
 - values at boundary points are fixed
- Data structure:
 - dense arrays
- Parallelism:
 - values at next time step can be computed simultaneously
 - parallelism is not dependent on runtime values
- Compiler can find the parallelism
 - spatial loops are DO-ALL loops

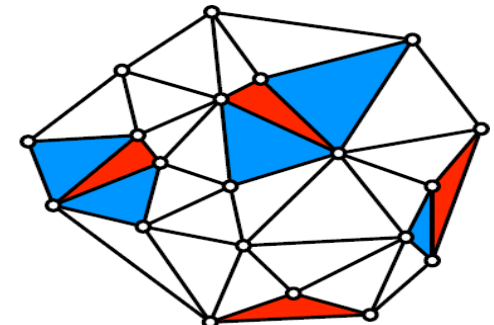


Jacobi iteration, 5-point stencil

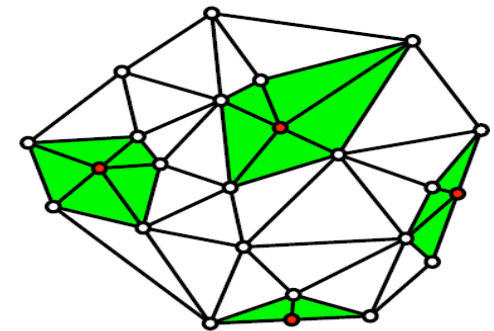
```
//Jacobi iteration with 5-point stencil
//initialize array A
for time = 1, nsteps
  for <i,j> in [2,n-1]x[2,n-1]
    temp(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))
  for <i,j> in [2,n-1]x[2,n-1]:
    A(i,j) = temp(i,j)
```

Delaunay Mesh Refinement

```
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(e); //determine new cavity
    c.expand();
    c.retriangulate();
    m.update(c); //update mesh
    wl.add(c.badTriangles());
}
```

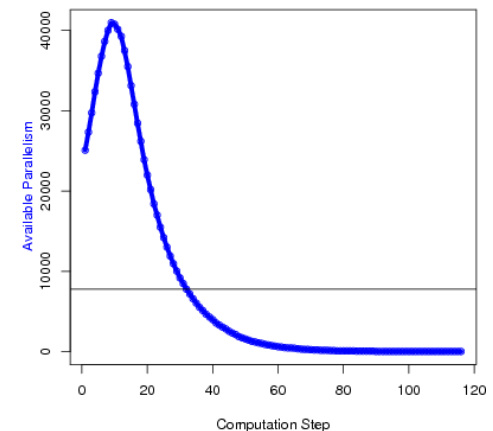


Before



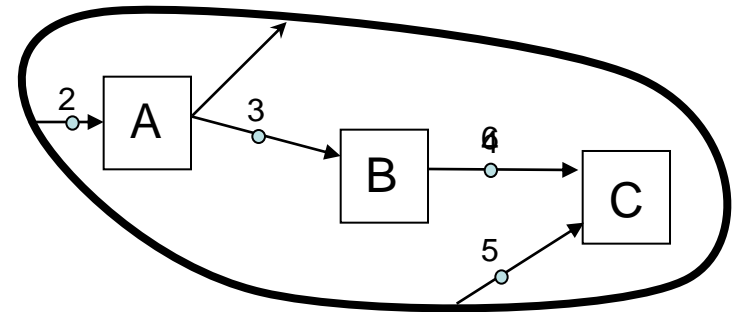
After

Available parallelism



Event-driven simulation

- Stations communicate by sending messages with time-stamps on FIFO channels
- Stations have internal state that is updated when a message is processed
- Messages must be processed in time-order at each station
- Data structure:
 - Messages in event-queue, sorted in time-order
- Parallelism:
 - activities created in future may interfere with current activities
 - static parallelization and interference graph technique will not work
 - Jefferson time-warp
 - station can fire when it has an incoming message on *any* edge
 - requires roll-back if speculative conflict is detected
 - Chandy-Misra-Bryant
 - conservative event-driven simulation
 - requires null messages to avoid deadlock

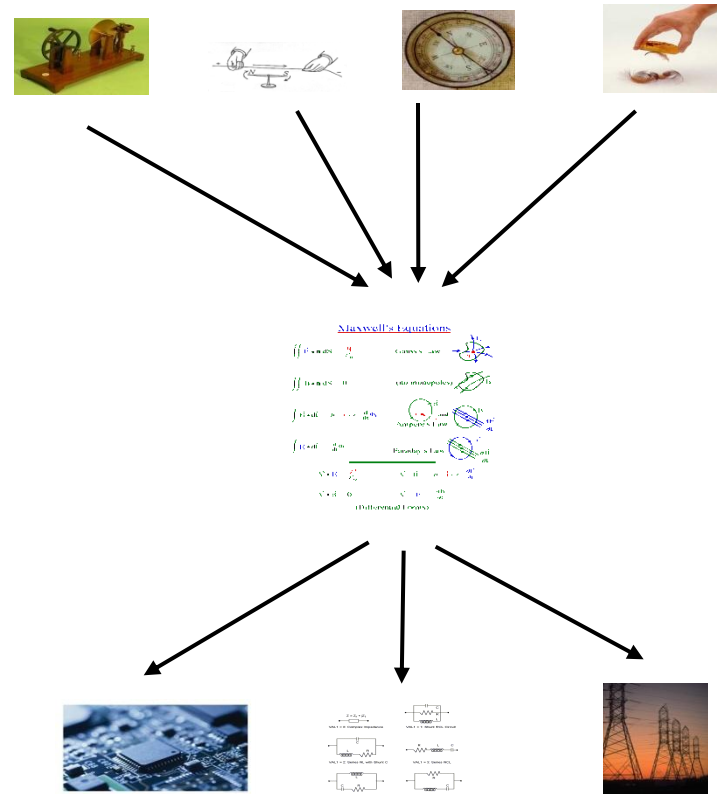


Remarks on algorithms

- Algorithms:
 - parallelism can be dependent on runtime values
 - DMR, event-driven simulation, graph reduction,....
 - don't-care non-determinism
 - nothing to do with concurrency
 - DMR, graph reduction
 - activities created in the future may interfere with current activities
 - event-driven simulation...
- Data structures:
 - relatively few algorithms use dense arrays
 - more common: graphs, trees, lists, priority queues,...
- Parallelism in irregular algorithms is very complex
 - static parallelization usually does not work
 - static dependence graphs are the wrong abstraction
 - finding parallelism: most of the work must be done at runtime

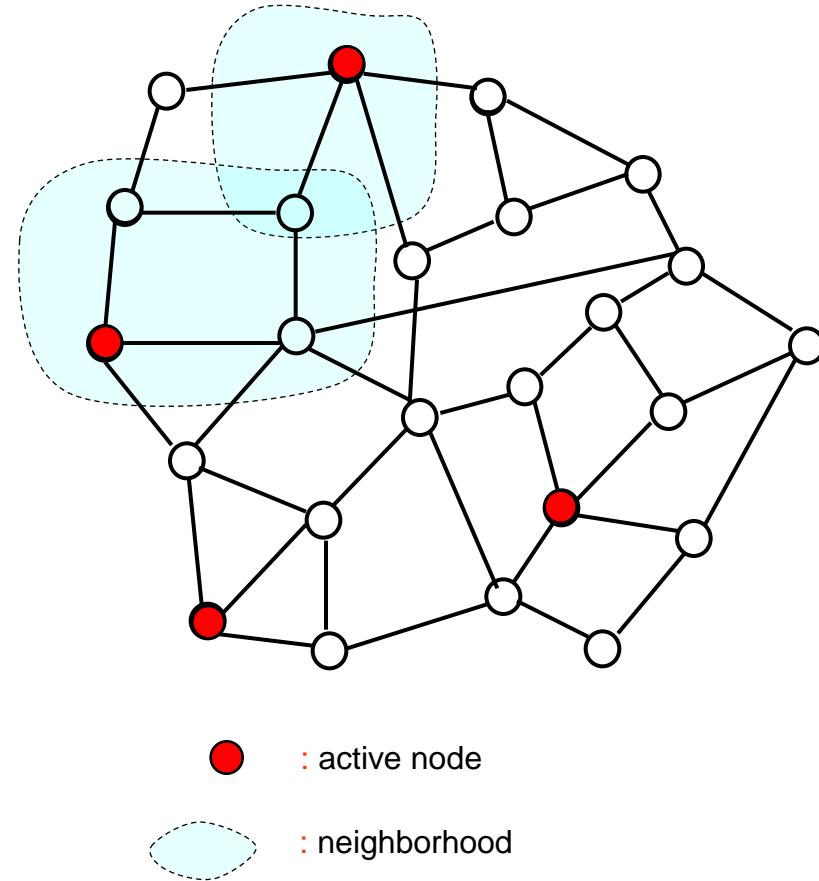
Organization of talk

- **Seemingly unrelated parallel algorithms and data structures**
 - Stencil codes
 - Delaunay mesh refinement
 - Event-driven simulation
 - Graph reduction of functional languages
 -
- **Unifying abstractions**
 - Operator formulation of algorithms
 - Amorphous data-parallelism
 - Baseline parallel implementation for exploiting amorphous data-parallelism
- **Specialized implementations that exploit structure**
 - Structure of algorithms
 - Optimized compiler and runtime system support for different kinds of structure
- **Ongoing work**



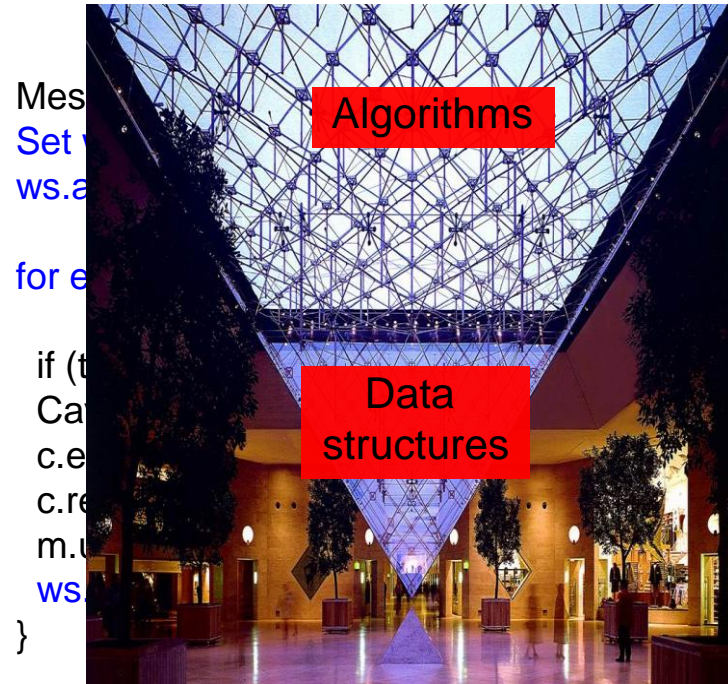
Operator formulation of algorithms

- Algorithm formulated in data-centric terms
 - **active element:**
 - node or edge where computation is needed
 - DMR: nodes representing bad triangles
 - **activity:**
 - application of operator to active element
 - **neighborhood:**
 - set of nodes and edges read/written to perform computation
 - DMR: cavity of bad triangle
 - distinct usually from neighbors in graph
 - **ordering:**
 - order in which active elements must be executed in a **sequential implementation**
 - any order (Jacobi, DMR, graph reduction)
 - some problem-dependent order (event-driven simulation)
- **Amorphous data-parallelism**
 - active nodes can be processed in parallel, subject to
 - neighborhood constraints
 - ordering constraints



Galois programming model (PLDI 2007)

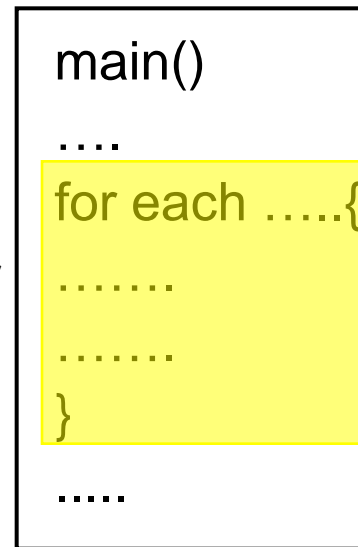
- **Joe programmers**
 - **sequential**, OO model:C++
 - Galois set iterators: for iterating over unordered and ordered sets of active elements
 - *for each e in Set S do B(e)*
 - evaluate B(e) for each element in set S
 - no a priori order on iterations
 - set S may get new elements during execution
 - *for each e in OrderedSet S do B(e)*
 - evaluate B(e) for each element in set S
 - perform iterations in order specified by OrderedSet
 - set S may get new elements during execution
- **Stephanie programmers**
 - Galois concurrent data structure library



DMR using Galois iterators

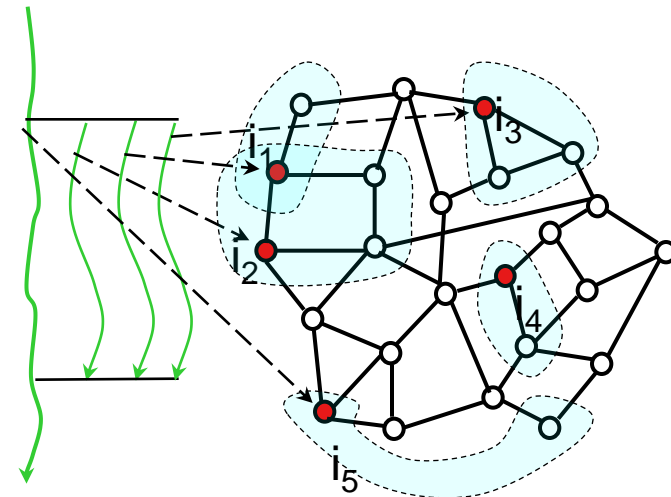
Galois parallel execution model

- **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
 - barrier synchronization at end of iterator
- **Independence of neighborhoods:**
 - logical locks on nodes and edges
 - implemented using CAS operations
- **Ordering constraints for ordered set iterator:**
 - execute iterations out of order but commit in order
 - cf. out-of-order CPUs



Joe Program

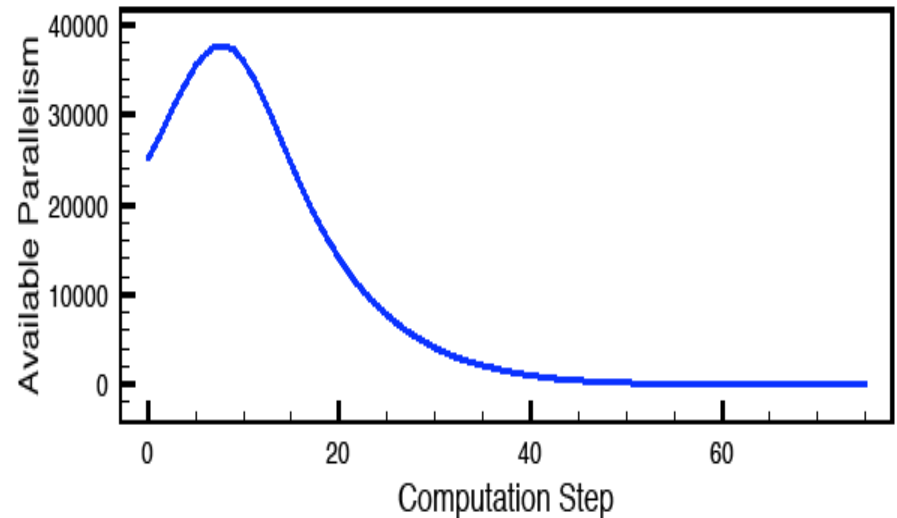
Master



Concurrent
Data structure

Parameter tool

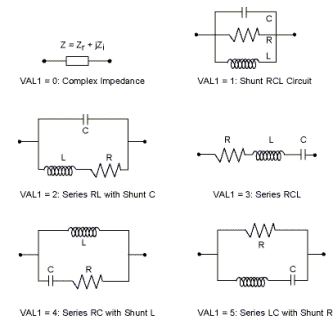
- Measures amorphous data-parallelism in irregular program execution
- Idealized execution model:
 - unbounded number of processors
 - applying operator at active node takes one time step
 - execute a maximal set of active nodes
 - perfect knowledge of neighborhood and ordering constraints
- Useful as an analysis tool



DMR: Input mesh: 550K triangles
Produced by Triangle

Organization of talk

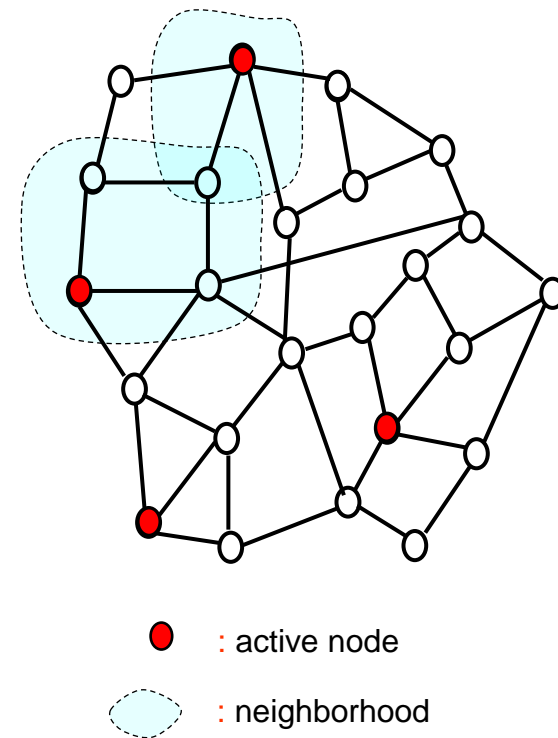
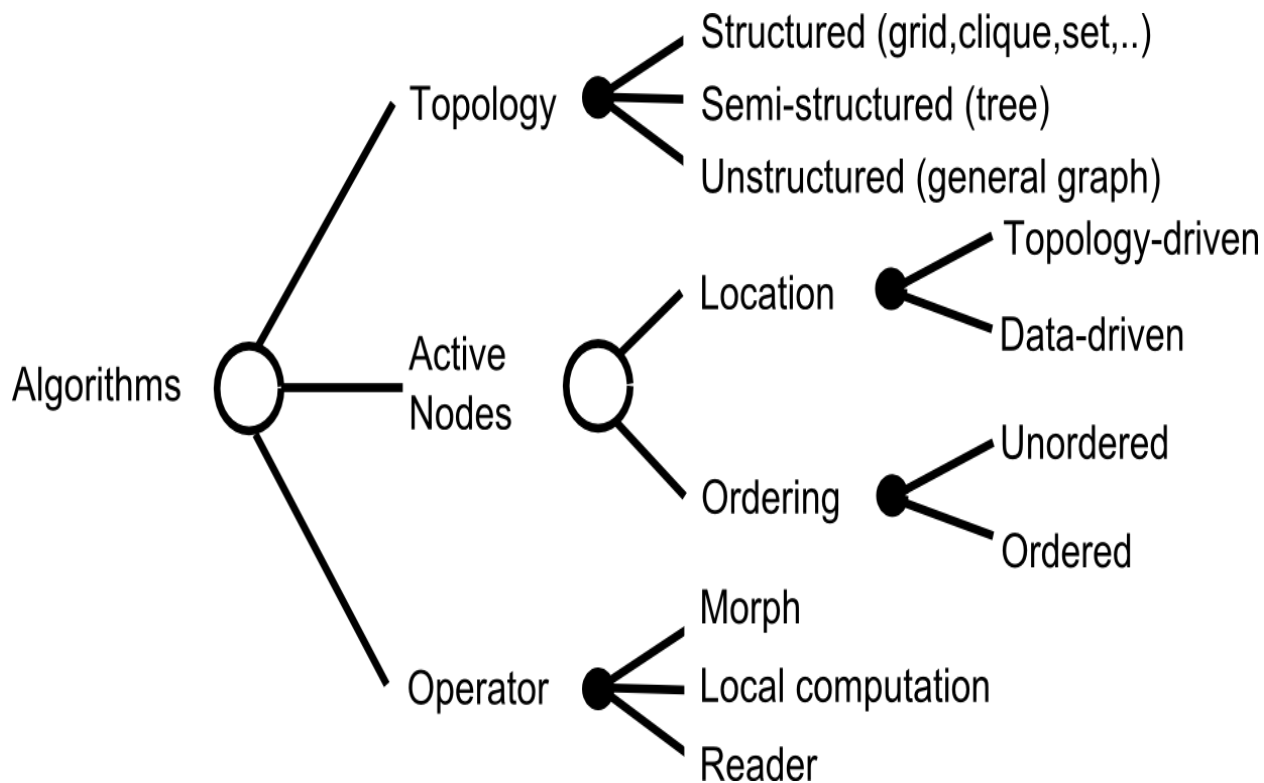
- Seemingly unrelated parallel algorithms and data structures
 - Stencil codes
 - Delaunay mesh refinement
 - Event-driven simulation
 - Graph reduction of functional languages
 -
- Unifying abstractions
 - Operator formulation of algorithms
 - Amorphous data-parallelism
 - Galois programming model
 - Baseline parallel implementation
- Specialized implementations that exploit structure
 - Structure in algorithms
 - Optimized compiler and runtime system support for different kinds of structure
- Ongoing work



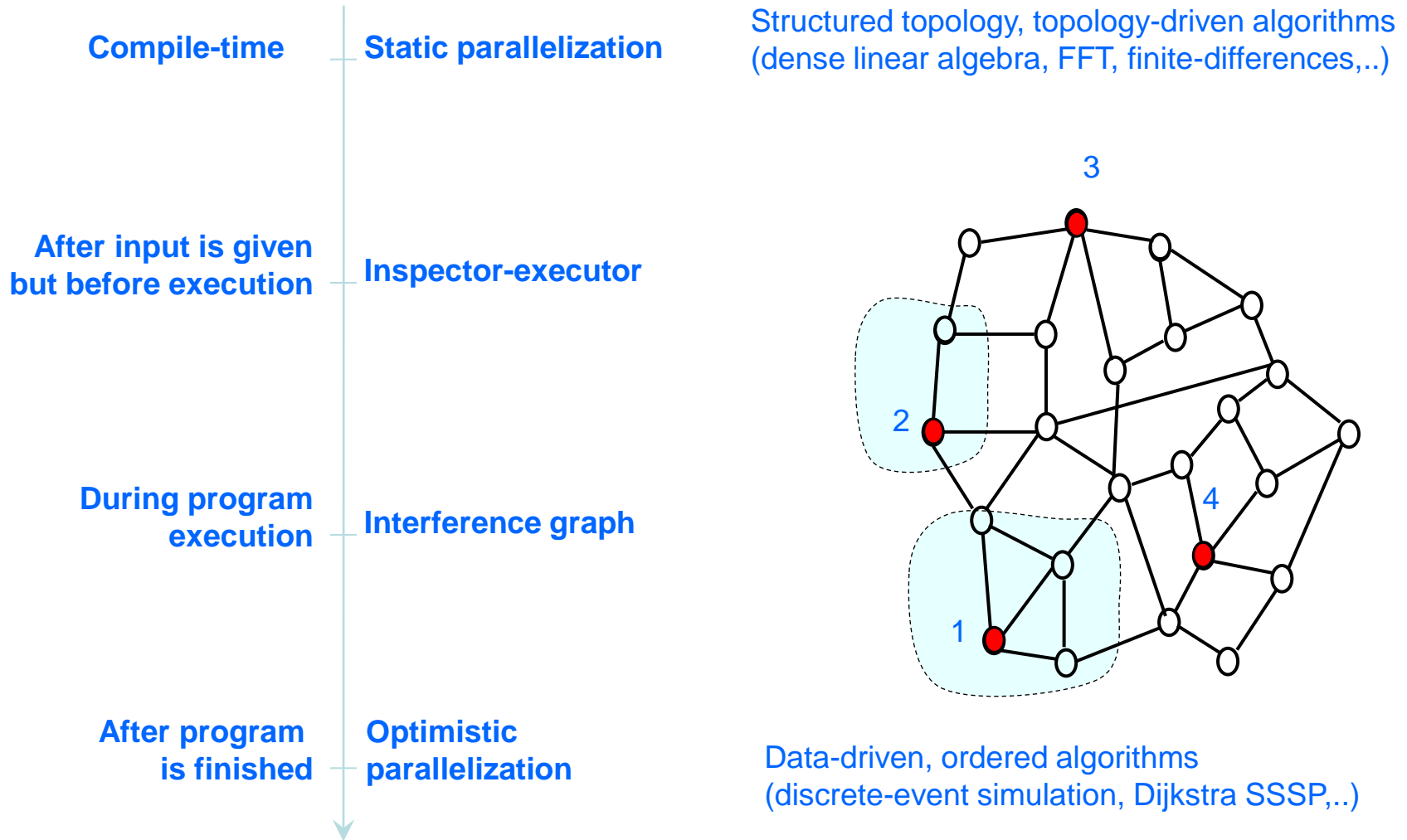
Structure in irregular algorithms

- Baseline implementation is general but usually inefficient
 - (eg) optimistic parallelization is not needed for stencil codes since graph structure, active nodes and neighborhoods are known statically
- Efficient execution requires exploiting structure in algorithms and data structures
- How do we talk about structure in algorithms?
 - Previous approaches: like descriptive biology
 - Mattson et al book
 - Parallel programming patterns (PPP): Snir et al
 - Berkeley motifs: Patterson, Yelick, et al
 - ...
 - Our approach: like molecular biology
 - structural analysis of algorithms
 - based on amorphous data-parallelism framework

TAO analysis:structure in algorithms (PLDI 2011)

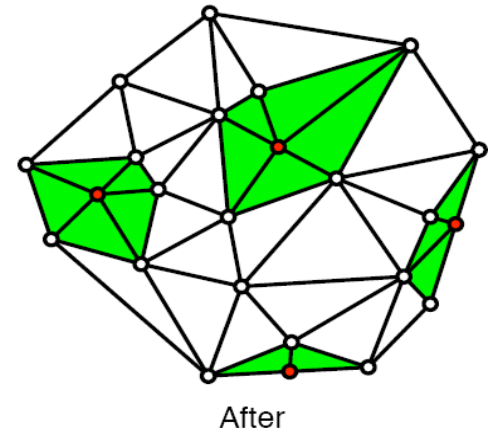
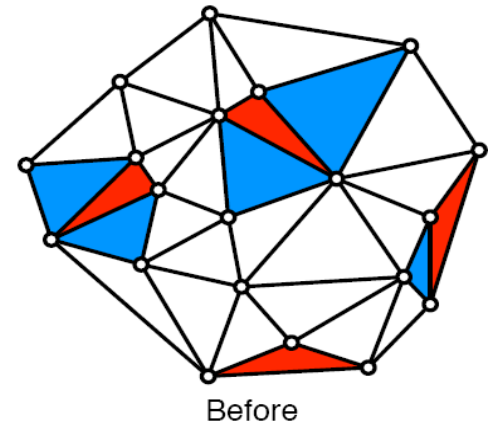


Exploiting TAO-structure to eliminate speculation: Binding Time



Cautious operators (PPoPP 2010)

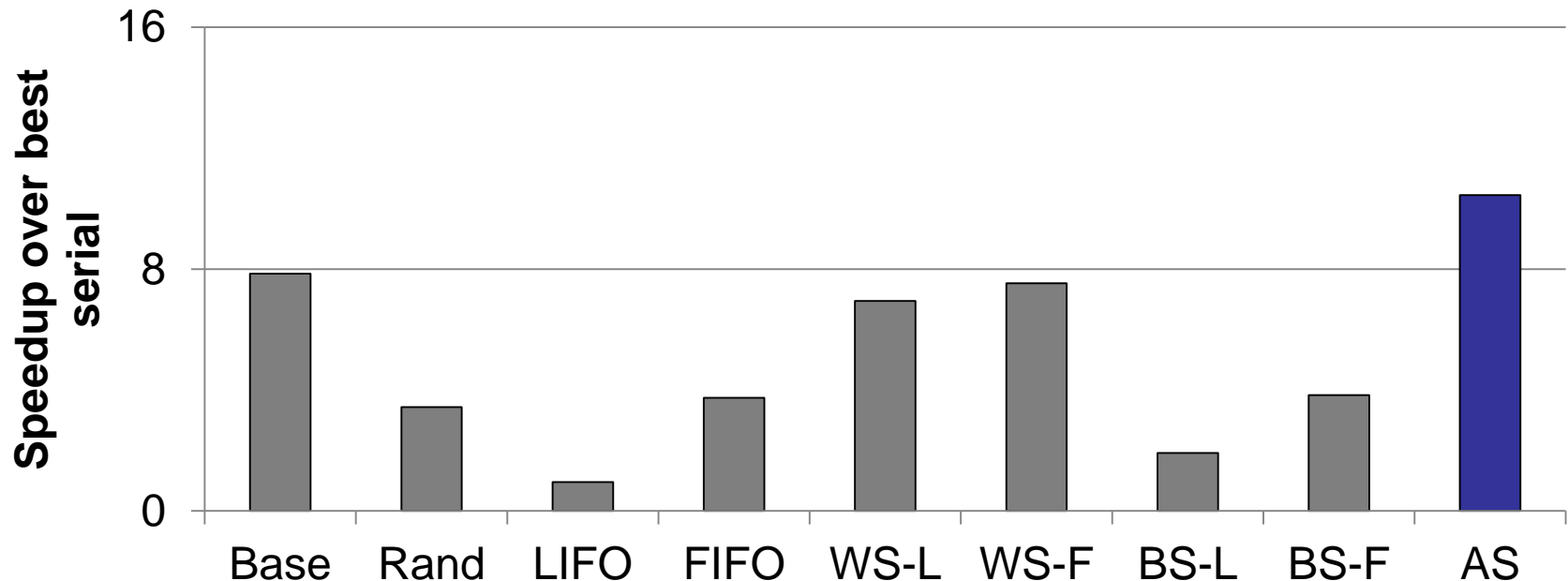
- Cautious operator implementation:
 - reads all the elements in its neighborhood before modifying any of them
 - (eg) Delaunay mesh refinement
- Algorithm structure:
 - cautious operator + unordered active elements
- Optimization: optimistic execution w/o buffering
 - grab locks on elements during read phase
 - conflict: someone else has lock, so release your locks
 - once update phase begins, no new locks will be acquired
 - update in-place w/o making copies
 - zero-buffering
 - note: this is not two-phase locking



Scheduling for unordered algorithms

- Best serial policy for DMR: LIFO
 - Exploit temporal (and potentially spatial) locality
- Best parallel policy for DMR: *not* LIFO
 - LIFO increases conflicts
 - Best policy: per thread LIFOs with initial work placed in global queue of chunks
 - New work placed on creating thread's LIFO
 - When a local LIFO is empty, steal a chunk from global queue
 - Application-specific policy: exploit locality while maintaining scalability and reducing conflicts
- Scheduler is a parallel program
 - can be harder to write than the application

Scheduler Sensitivity: DMR



- **Rand**
- **LIFO, FIFO**: Global queue or stack
- **WS-L, WS-F**: Work-stealing with queue or stack
- **BS-L, BS-F**
- **Base**: FIFO of chunks of at most 32 elements
- **AS**: Application-specific policy

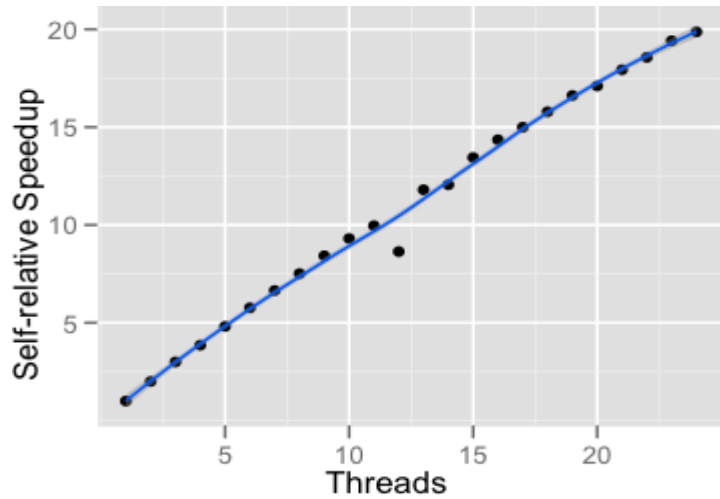
Scheduler Synthesis

(Nguyen et al, ASPLOS 2011)

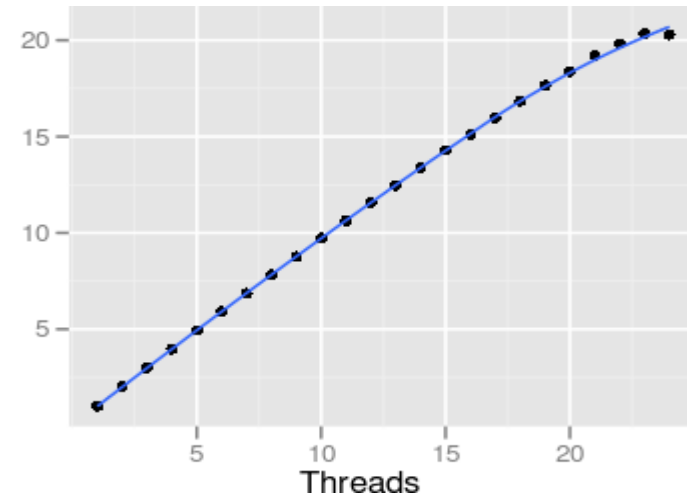
- Language for scheduling policies
 - *Declarative*: compositional specification w/o code
- Synthesized schedulers exploit hierarchical cache architecture of modern multicores
 - *Effective*: performance comparable to hand-written and often better than previous schedulers

Get good performance without writing
(serial or concurrent) scheduling code

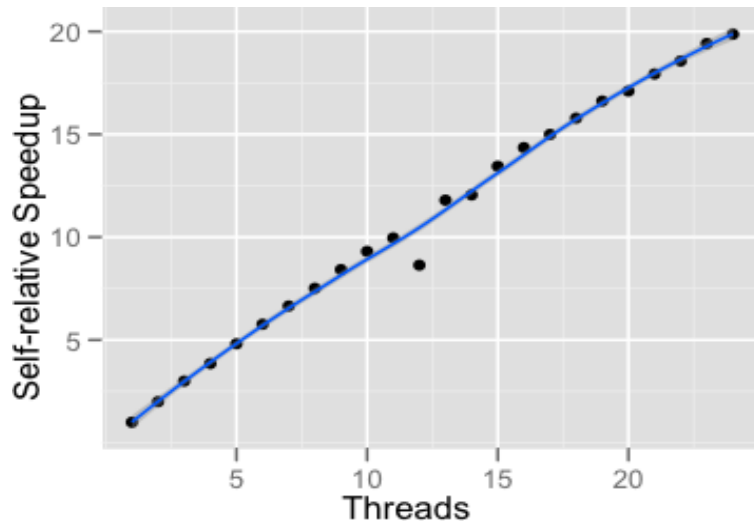
Performance of Galois system (I)



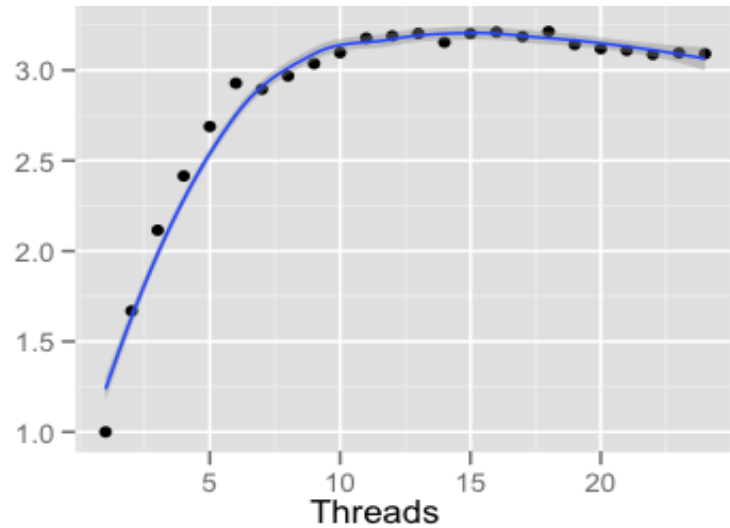
[Betweenness Centrality](#)



[Delaunay Mesh Refinement](#)



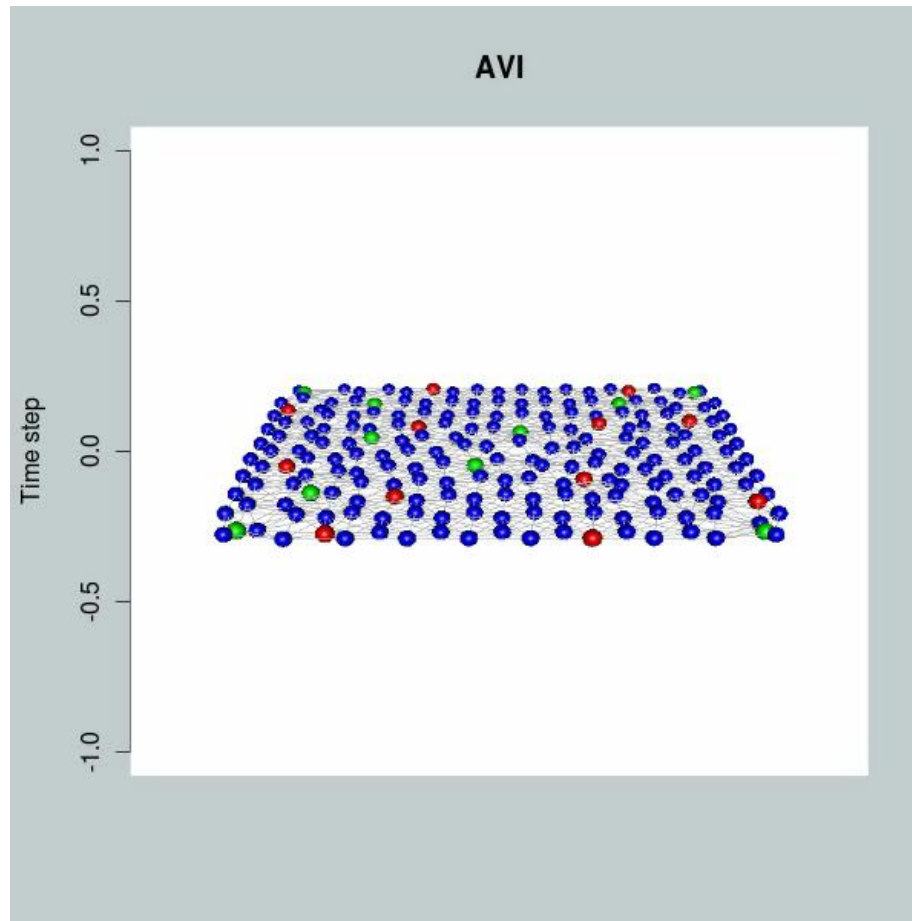
[Asynchronous Variational Integrator](#)



[Metis](#)

Intel 4x6 Xeon E7540 @ 2 GHz

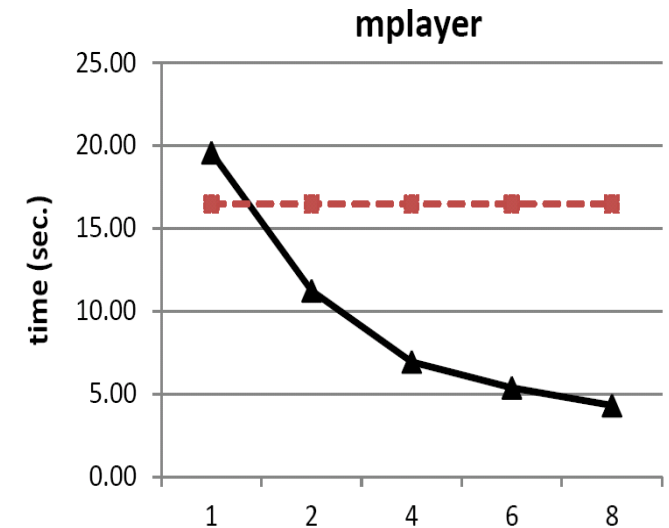
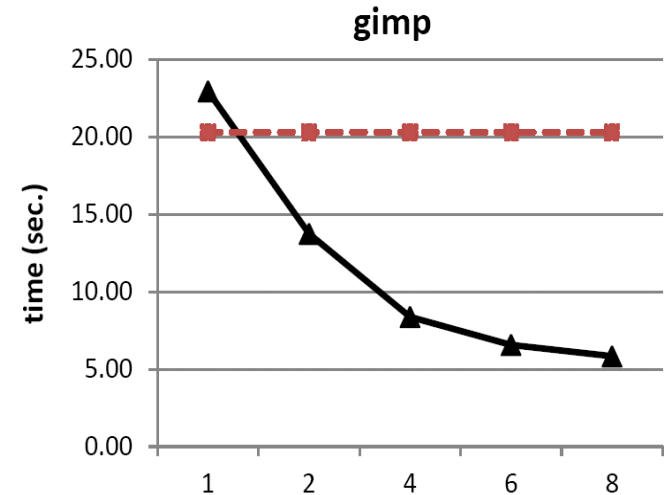
Performance of Galois system (II)



Asynchronous Variational Integrator (AVI)
Joint work with Professor Adrian Lew (Stanford)

Performance of Galois system (III)

- Andersen-style points-to analysis
- Algorithm formulation
 - 3 operators
 - speedup algorithm by collapsing cycles in constraint graph
- State of the art sequential C++ implementation
 - Hardekopf & Lin
 - red lines in graphs
- “Parallel Andersen-style points-to analysis” Mendez-Lojo et al (OOPSLA 2010)



Related work

- Transactional memory (TM)
 - Programming model:
 - explicitly parallel
 - no separation between ADT and implementation
 - Where do threads come from?
- Thread-level speculation (TLS)
 - Programming model:
 - no separation between ADT and implementation
 - ordered algorithms

Galois system =

Abstract Data Types (permit Joe/Stephanie separation)

+

Don't-care non-determinism (unordered set iterator)

+

Scheduling directives (synthesis)

+

Optimistic parallelization

+

Exploitation of structure in algorithms and data (compiler)

Summary of high-level message

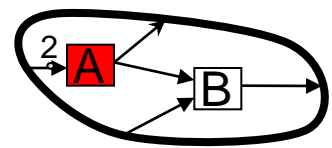
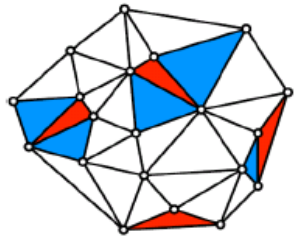
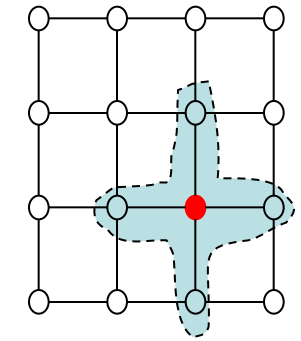
• Old approach

1. Static parallelization is the norm
2. Inspector-executor, optimistic parallelization, etc.
 - needed only for weird programs, crutch for dumb programmers
 - they are expensive: (eg) high abort ratio
3. Dependence graphs are the right abstraction for parallelism
 - program-centric abstraction

• New approach

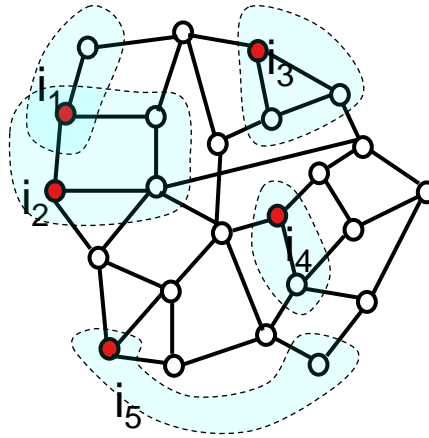
1. Optimistic parallelization is the baseline
2. Static parallelization, inspector-executor etc.
 - possible only for some programs
 - scheduling decisions, binding of threads
 - overheads of optimistic parallelization can be controlled
3. Operator formulation of algorithms is the right abstraction
 - data-centric abstraction

Science of Parallel Programming



.....

**Seemingly
unrelated algorithms**



Unifying abstractions



慎



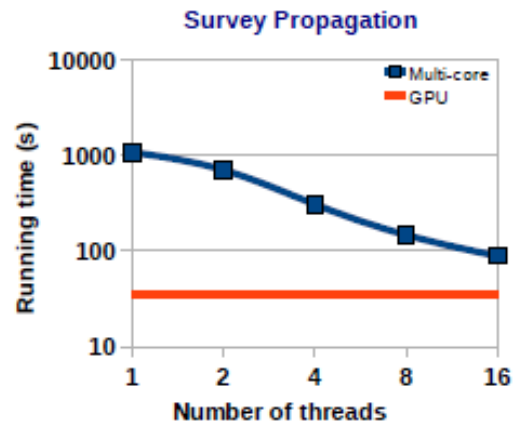
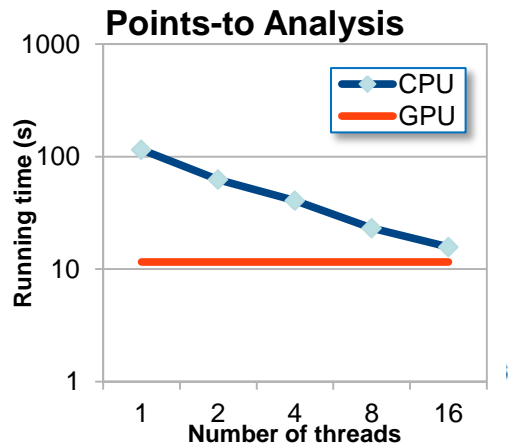
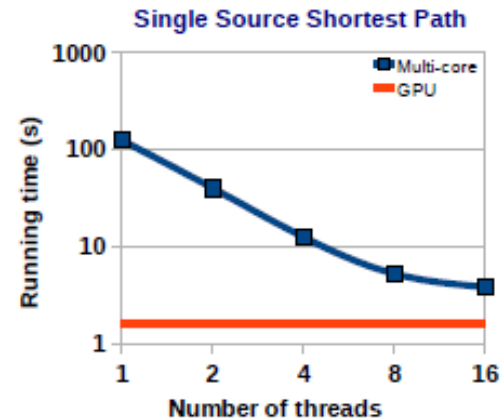
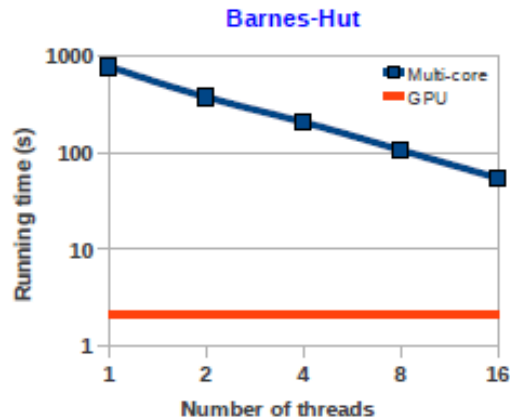
**Specialized models
that exploit structure**

Acknowledgements

- Martin Burtscher (Texas State)
- Patrick Carribault (France)
- Ambar Hassaan
- Rashid Kaleem
- Milind Kulkarni (Purdue)
- Andrew Lenharth
- Tsung-Hsien Lee
- Roman Manevich
- Mario Mendez-Lojo (AMD)
- Rupesh Nasre
- Donald Nguyen
- Dimitris Prountzos
- Xin Sui
- Akshatha Bhatt (UTSA)
- Qing Yi (UTSA)

Download: <http://iss.ices.utexas.edu/~galois>

Performance Results



Inputs: BH: 5M stars
PT: linux (1.5M variables, 0.4M constraints)

SSSP: 23M nodes, 57M edges,
SP: 1M literals, 4.2M clauses