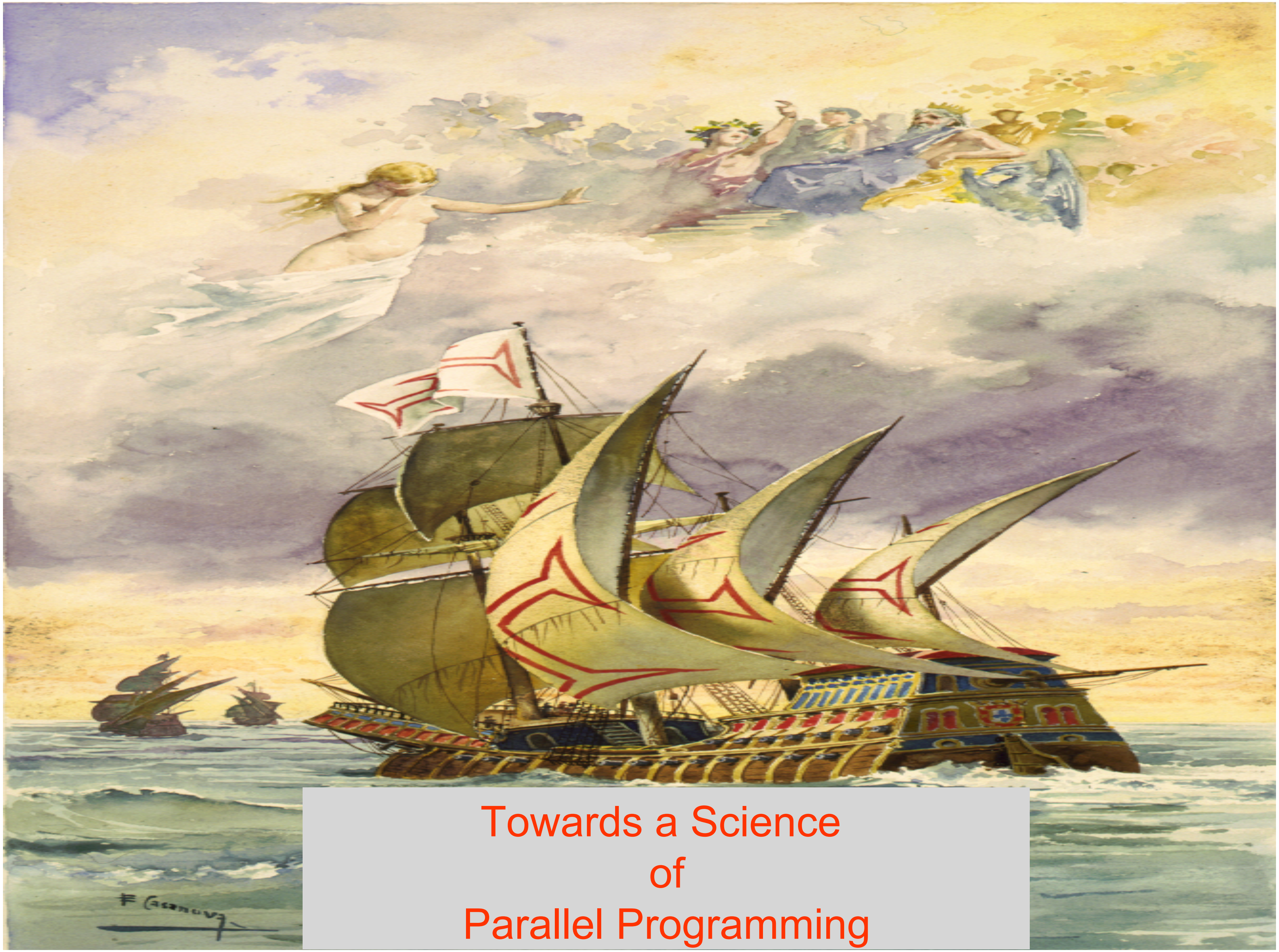


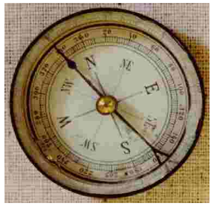
Towards a Science of Parallel Programming

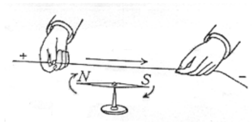
Problem Statement

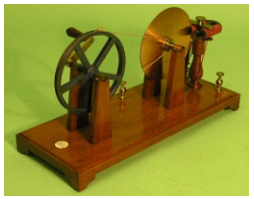
- Community has worked on parallel programming for more than 30 years
  - programming models
  - machine models
  - programming languages
  - ….
- However, parallel programming is still a research problem
  - matrix computations, stencil computations, FFTs etc. are well-understood
  - few insights for irregular applications
    - each new application is a "new phenomenon"
- Thesis: we need a science of parallel programming
  - analysis: framework for thinking about parallelism in application
  - synthesis: produce an efficient parallel implementation of application

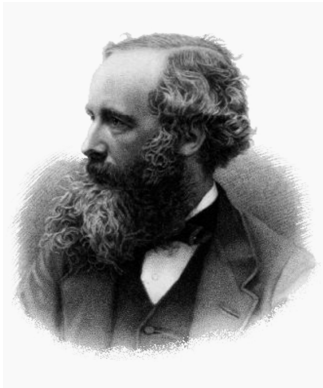
"The Alchemist" Cornelius Bega (1663)

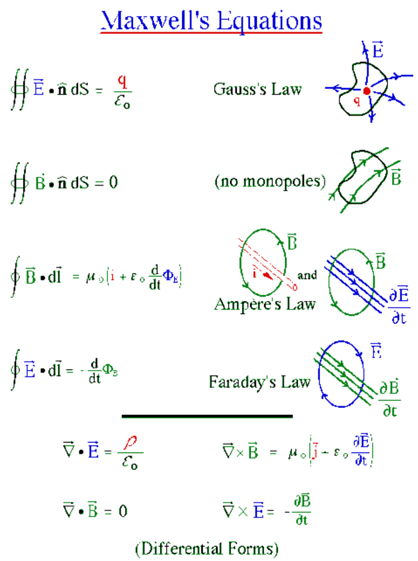
Analogy: science of electro-magnetism

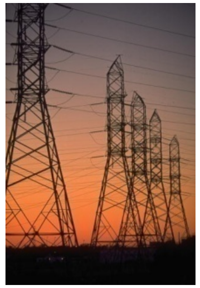


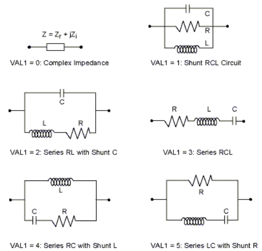












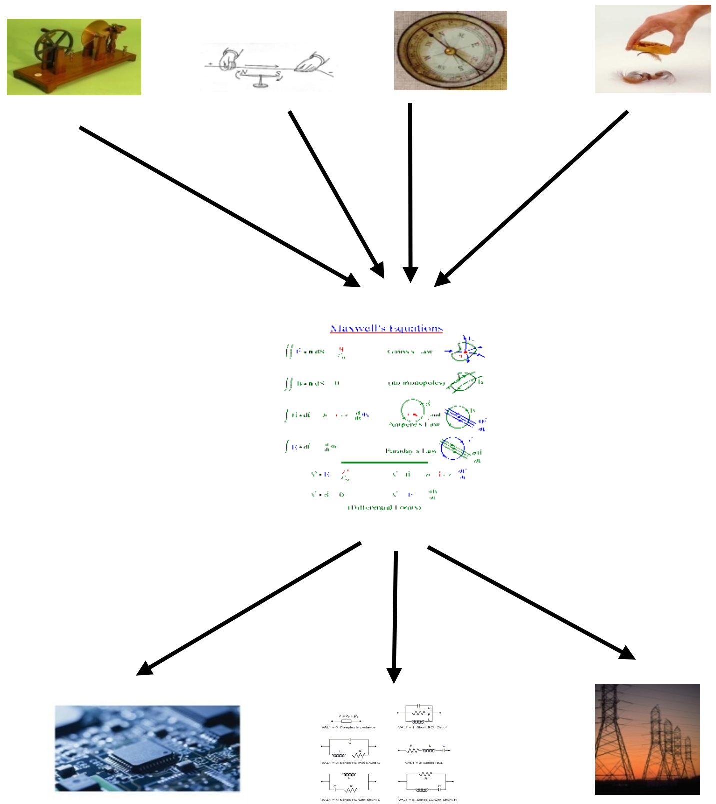
Seemingly unrelated phenomena

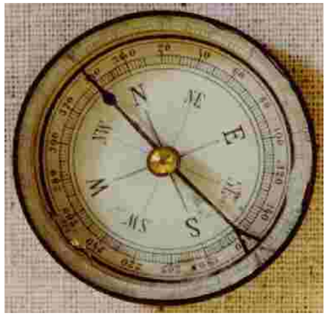
Unifying abstractions

Specialized models that exploit structure

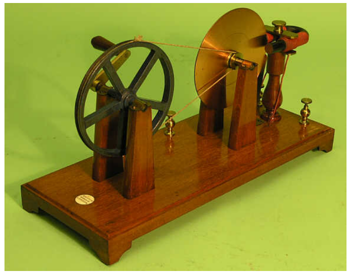
Organization of talk

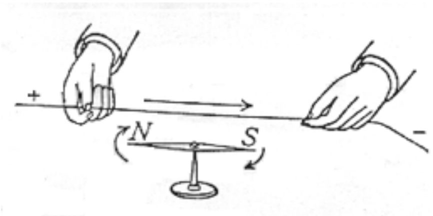
- Seemingly unrelated parallel algorithms and data structures
  - Stencil codes
  - Delaunay mesh refinement
  - Event-driven simulation
  - Graph reduction of functional languages
  - ………
- Unifying abstractions
  - Operator formulation of algorithms
  - Amorphous data-parallelism
  - Galois programming 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



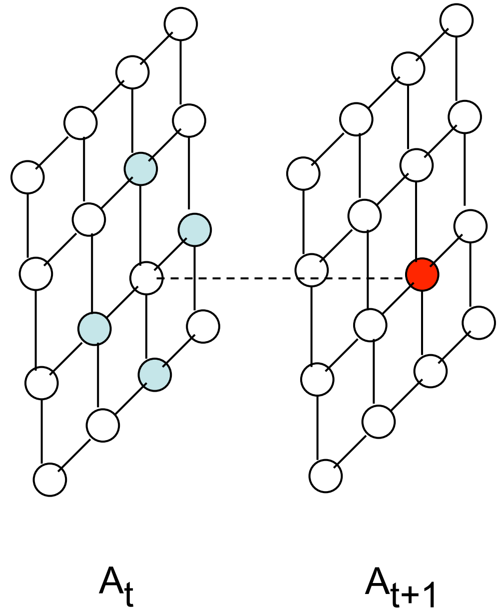


Examples

Application/domain		Algorithm
Meshing		Generation/refinement/partitioning
Compilers		Iterative and elimination-based dataflow algorithms
Functional interpreters		Graph reduction, static and dynamic dataflow
Maxflow		Preflow-push, augmenting paths
Minimal spanning trees		Prim, Kruskal, Boruvka
Event-driven simulation		Chandy-Misra-Bryant, Jefferson Timewarp
AI		Message-passing algorithms
Stencil computations		Jacobi, Gauss-Seidel, red-black ordering
Data-mining		Clustering

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

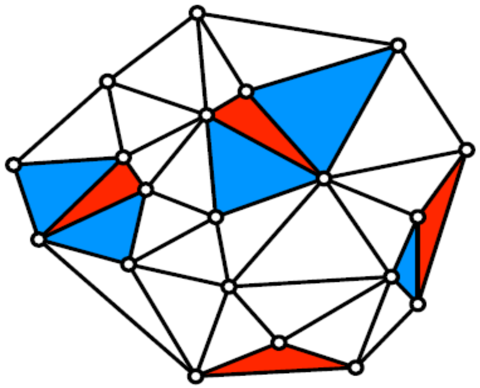


Jacobi iteration, 5-point stencil

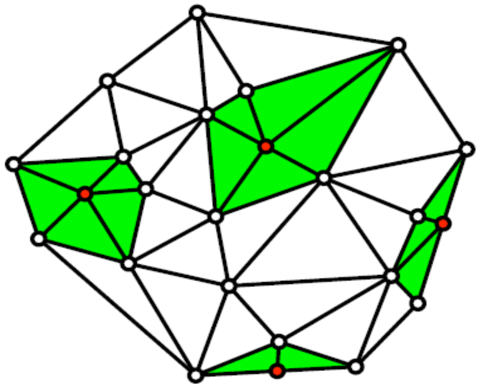
```
//Jacobi iteration with 5-point stencil
//initialize array A
for time = 1, nsteps
   for <i,j> in [2,n-1]x[2,n-1]
       temp(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))
   for <i,j> in [2,n-1]x[2,n-1]:
       A(i,j) = temp(i,j)
```

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();//re-triangulate
region
      m.update(c);//update mesh
      wl.add(c.badTriangles());
}
```



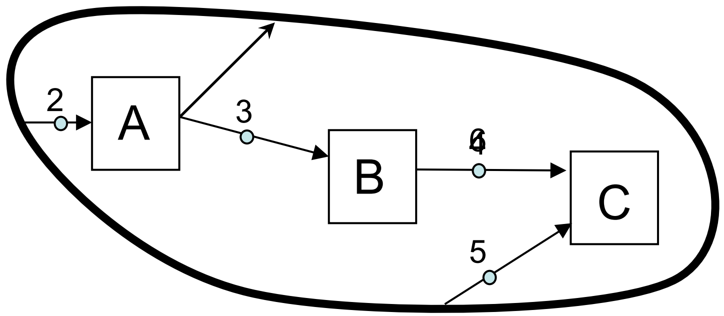
Before



After

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

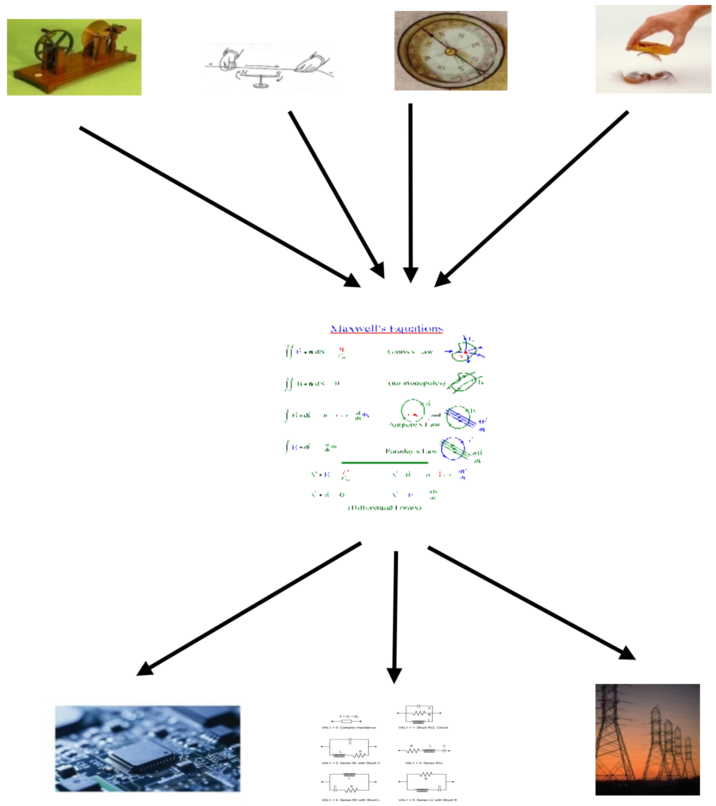


Remarks on algorithms

- 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
  - 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

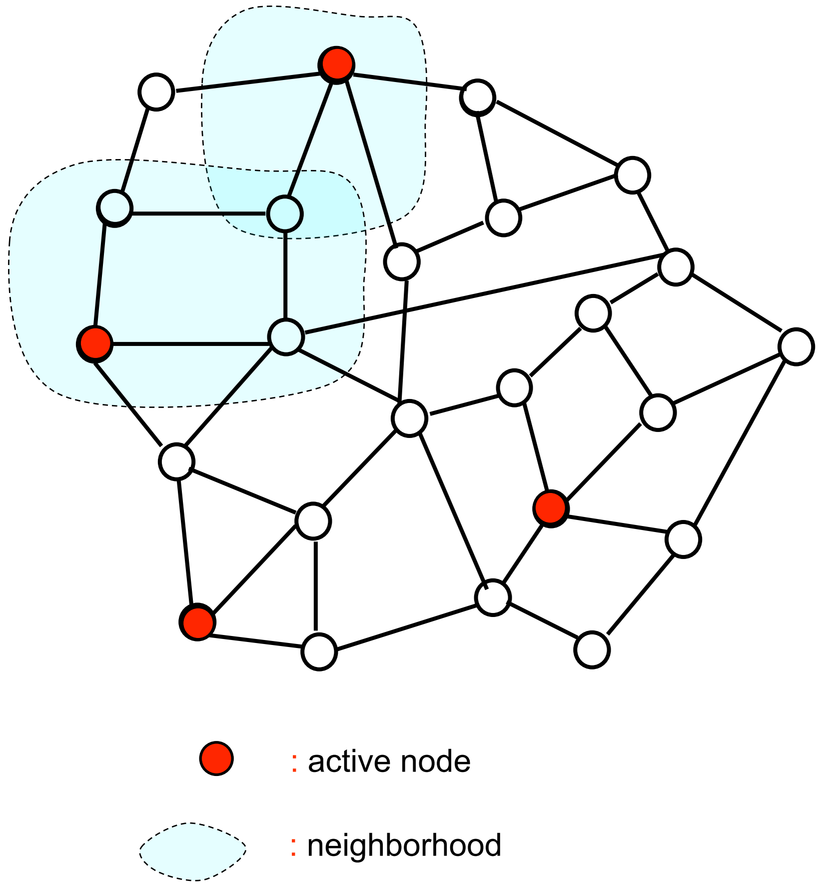


Unifying abstractions

- Should provide a model of parallelism in irregular algorithms
- Ideally, unified treatment of parallelism in regular and irregular algorithms
  - parallelism in regular algorithms should emerge as a special case of general model
  - (cf.) correspondence principles in Physics
- Abstractions should be effective
  - should be possible to write an interpreter to execute algorithms in parallel

Operator formulation of algorithms

- Algorithm formulated in data-centric terms
  - active element:
    - node or edge where computation is needed
      - DMR: nodes representing bad triangles
      - Event-driven simulation: station with incoming message
      - Jacobi: nodes of mesh
  - activity:
    - application of operator to active element
  - neighborhood:
    - set of nodes and edges read/written to perform computation
      - DMR: cavity of bad triangle
      - Event-driven simulation: station
      - Jacobi: nodes in stencil
    - 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
  - 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
- (Wirth) Algorithms + Data structures = Programs
  - (cf) database programming

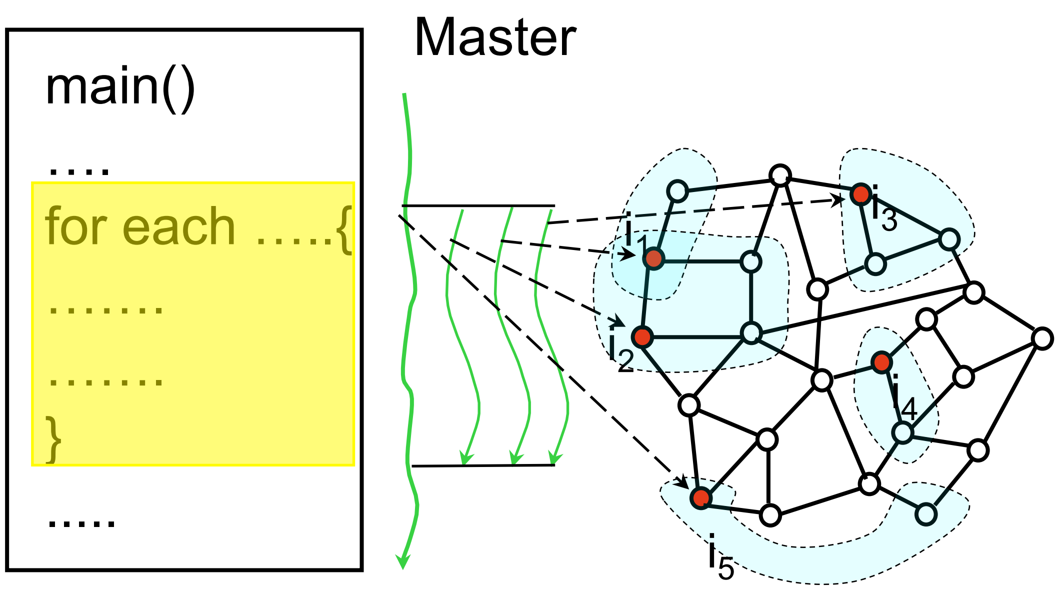
```
Mesh m = /* read in mesh */
Set ws;
ws.add(m.badTriangles()); //
initialize ws

for each tr in Set ws do { //unordered
Set iterator                    if (tr
no longer in mesh) continue;
       Cavity c = new Cavity(tr);
       c.expand();
       c.retriangulate();
       m.update(c);
       ws.add(c.badTriangles()); //bad
triangles
}
```

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

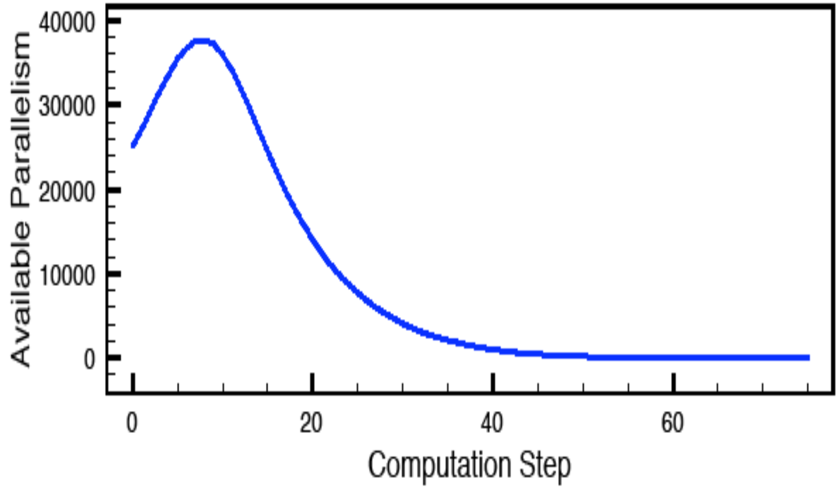
Concurrent
Data structure

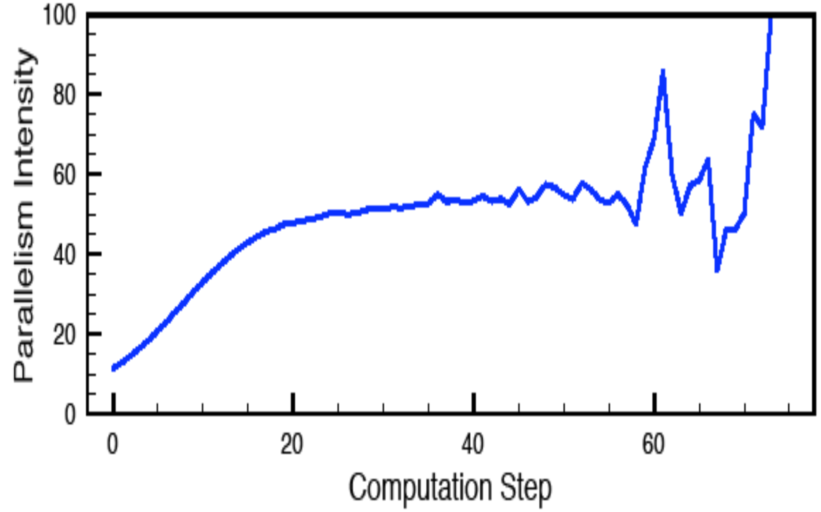
Parameter tool (PPoPP 2009)

- 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

Example: DMR

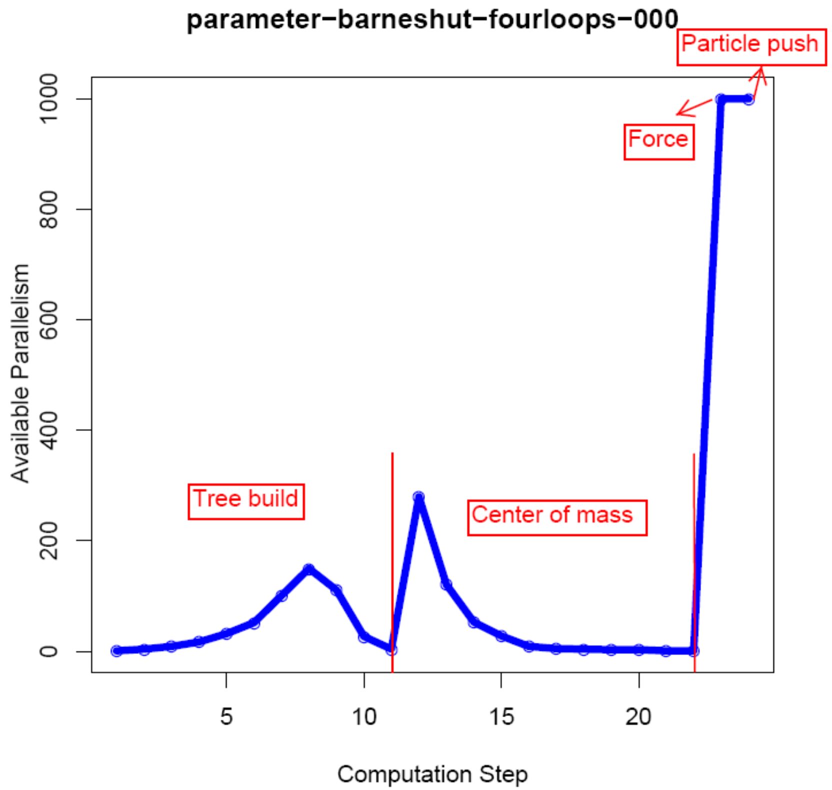
- Input mesh:
  - Produced by Triangle (Shewchuck)
  - 550K triangles
  - Roughly half are badly shaped
- Available parallelism:
  - How many non-conflicting triangles can be expanded at each time step?
- Parallelism intensity:
  - What fraction of the total number of bad triangles can be expanded at each step?





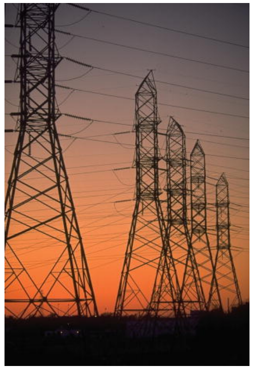
Example:Barnes-Hut

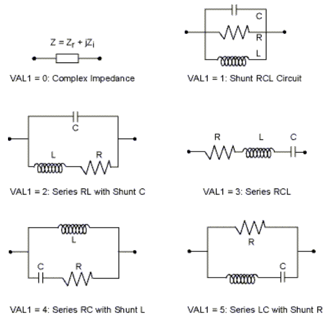
- Four phases:
  - build tree
  - center-of-mass
  - force computation
  - push particles
- Problem size:
  - 1000 particles
- Parallelism profile of tree build phase similar to that of DMR
  - why?



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

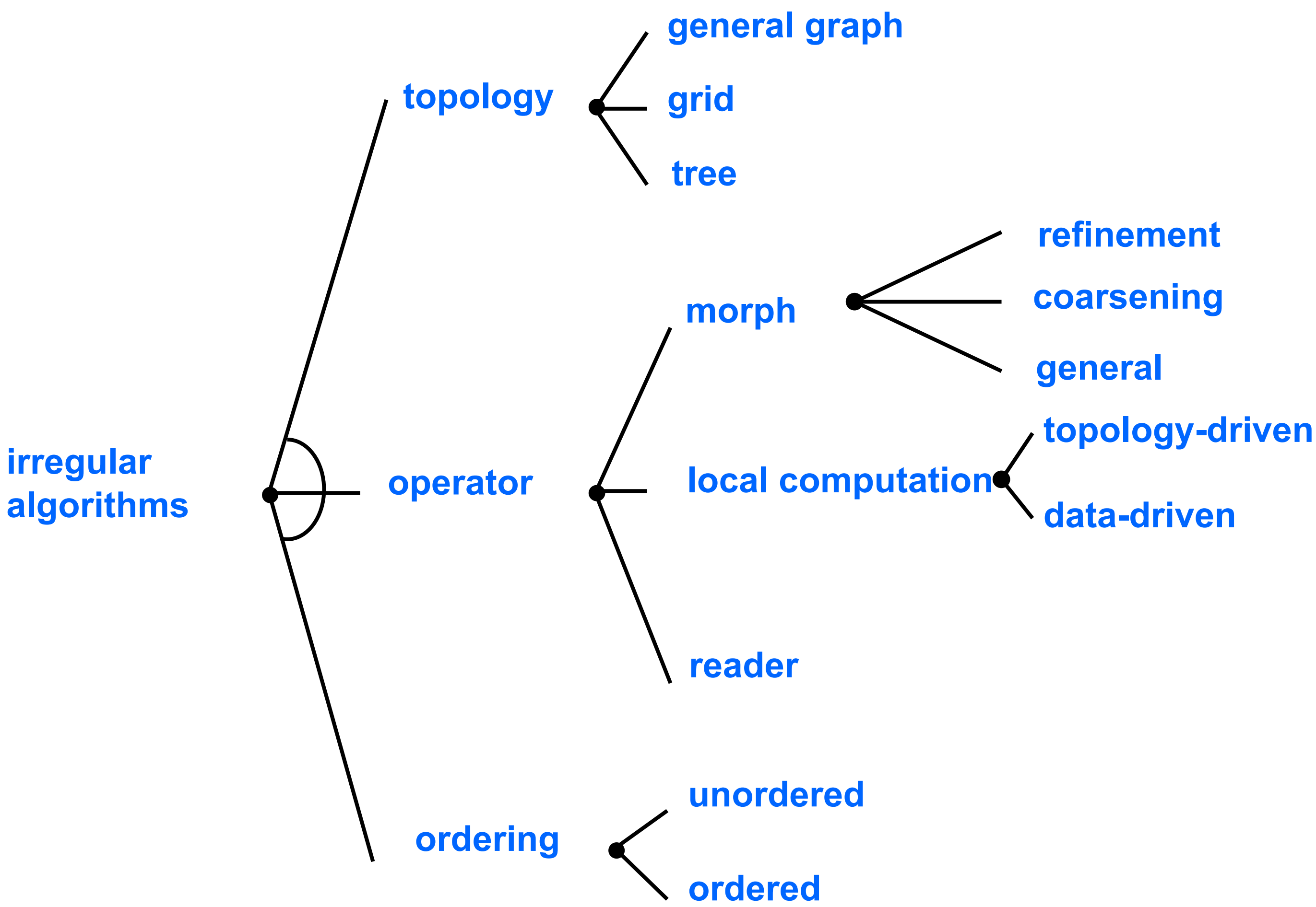




Structure in irregular algorithms

- Baseline implementation is general but usually inefficient
  - (eg) dynamic scheduling of iterations is not needed for stencil codes since grid structure is known at compile-time
  - (eg) hand-written parallel implementations of DMR do not buffer updates to neighborhood until commit point
- 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

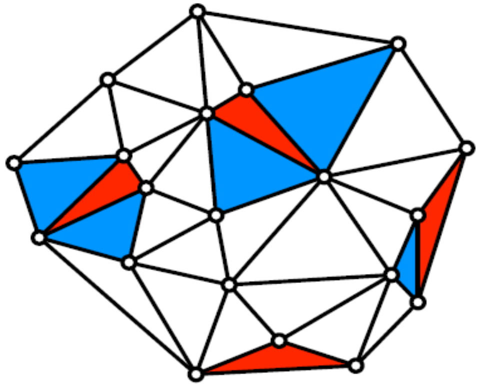
Structural analysis of irregular algorithms



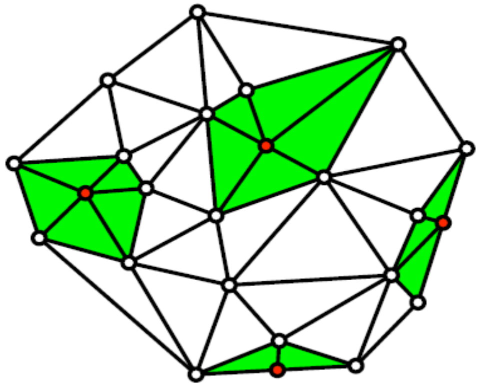
Jacobi: topology: grid, operator: local computation, ordering: unordered
DMR, graph reduction: topology: graph, operator: morph, ordering: unordered
Event-driven simulation: topology: graph, operator: local computation, ordering: ordered

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

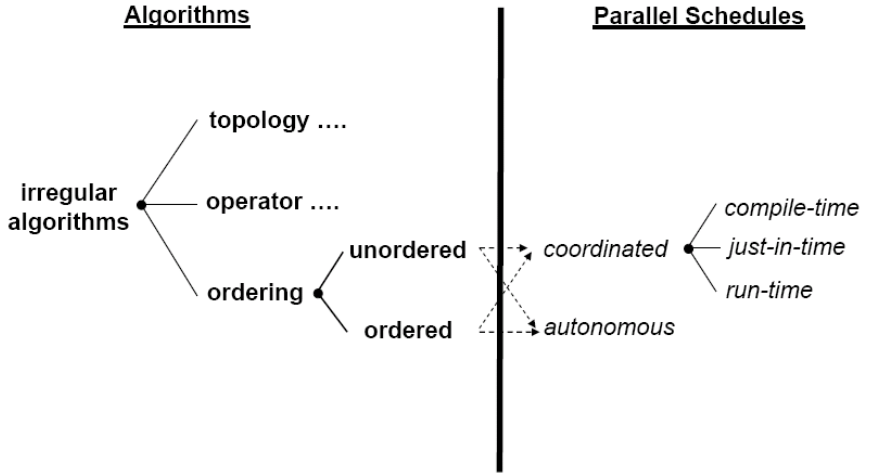


Before



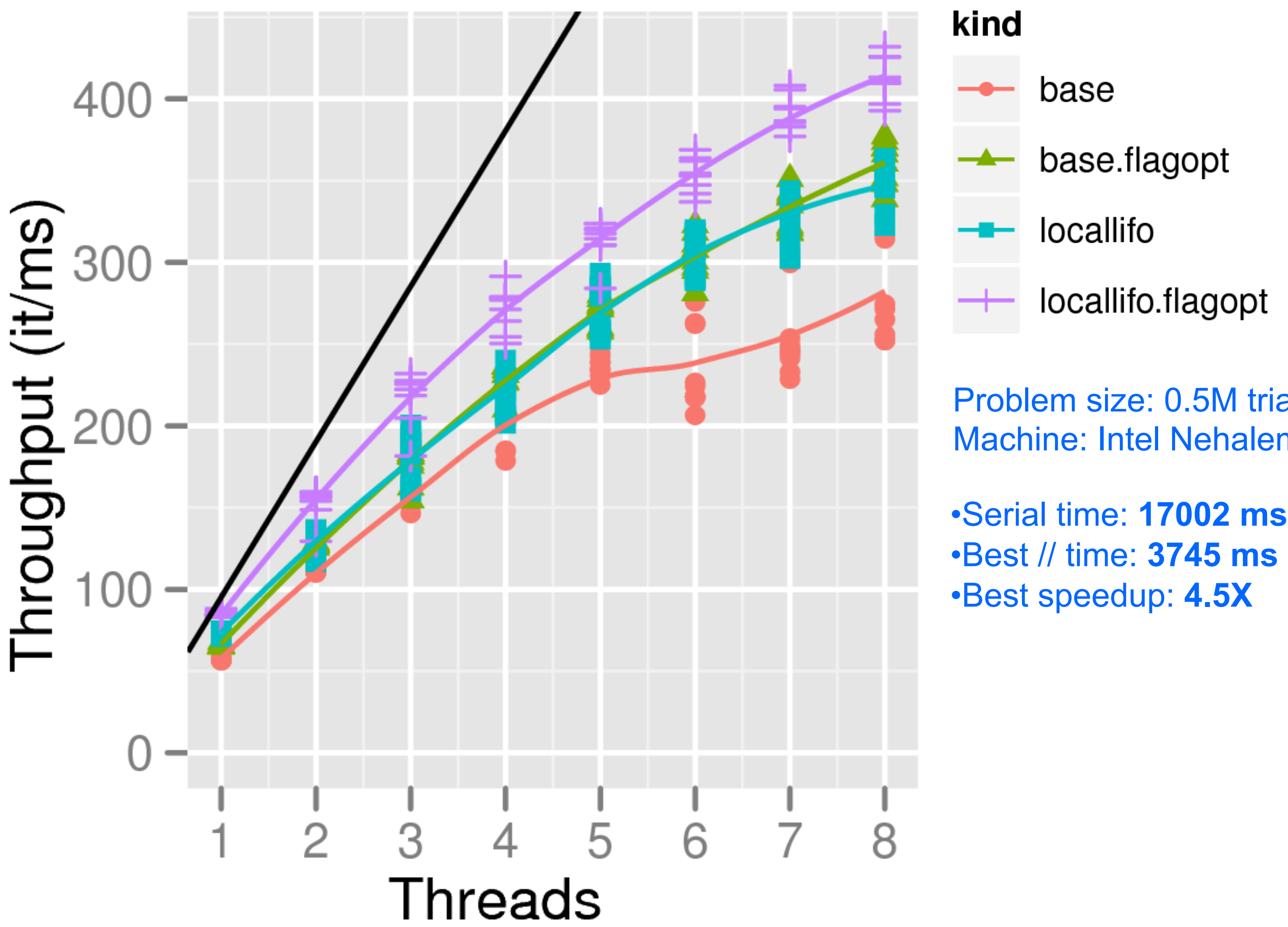
After

Eliminating speculation



- Coordinated execution of activities:
  - if we can build dependence graph
  - early binding of scheduling decisions
- Binding times
  - Run-time scheduling:
    - cautious operator + unordered active elements
    - execute all activities partially to determine neighborhoods
    - create interference graph and find independent set of activities
    - execute independent set of activities in parallel w/o synchronization
  - Just-in-time scheduling:
    - local computation + topology-driven (eg) tree walks, sparse MVM
    - inspector-executor approach
  - Compile-time scheduling:
    - previous case + graph is known at compile-time (eg) Jacobi
    - make all scheduling decisions at compile-time time

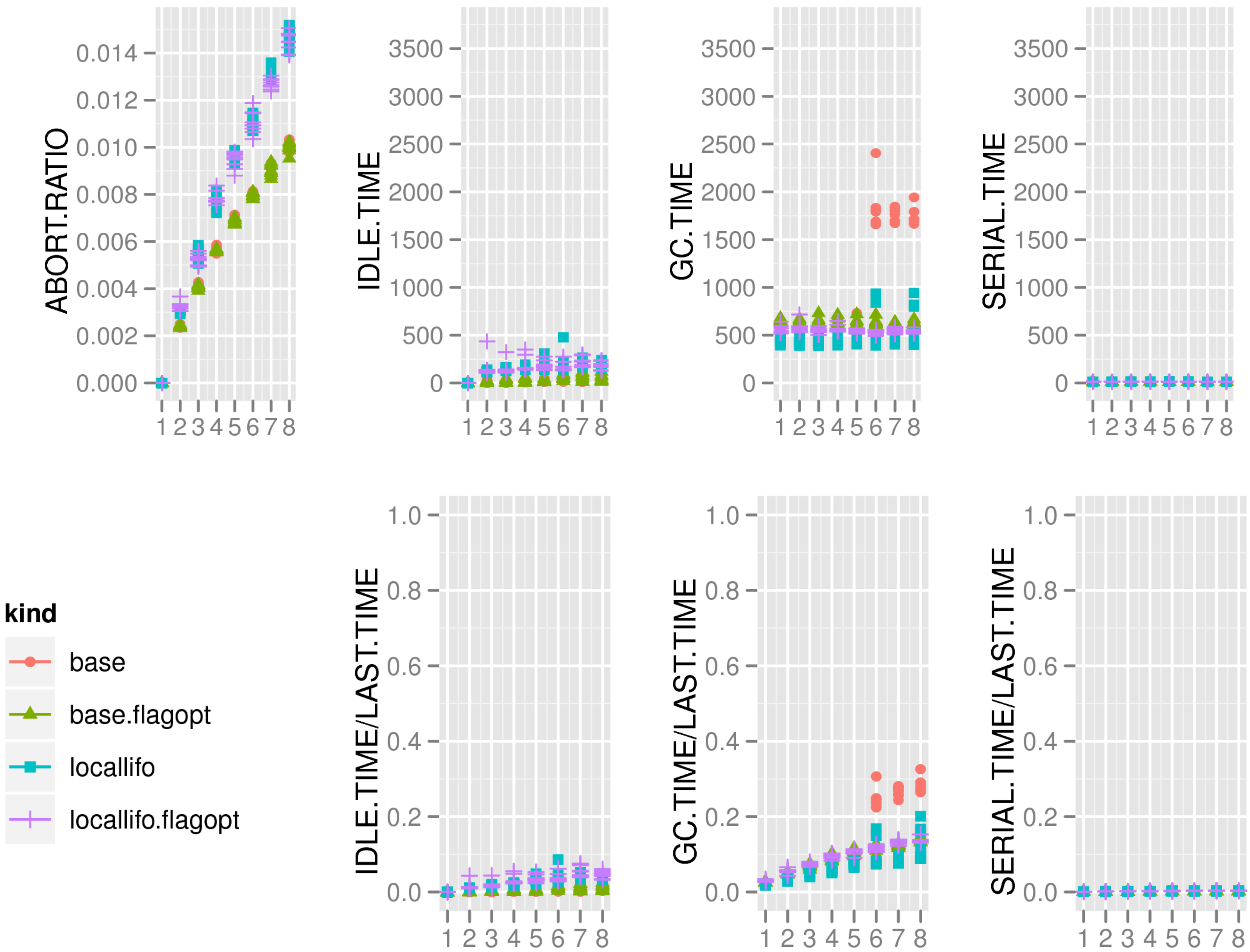
DMR Results



Problem size: 0.5M triangles, 0.25M bad triangles
Machine: Intel Nehalem, 2 Quad-core processors

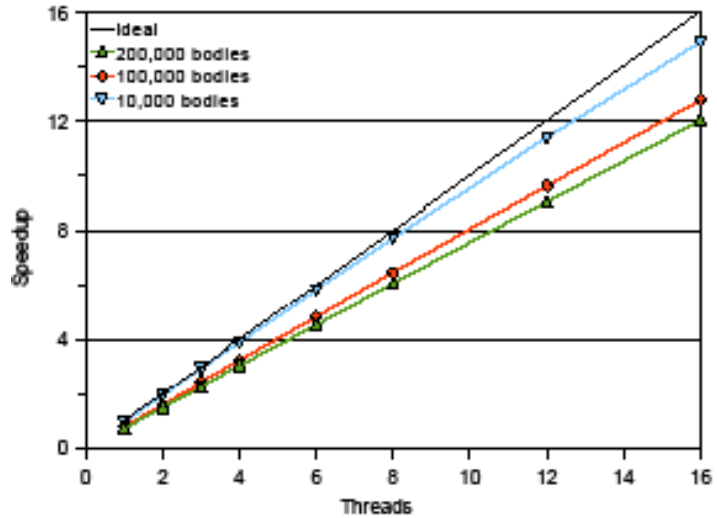
- Serial time: **17002 ms**
- Best // time: **3745 ms**
- Best speedup: **4.5X**

DMR Statistics

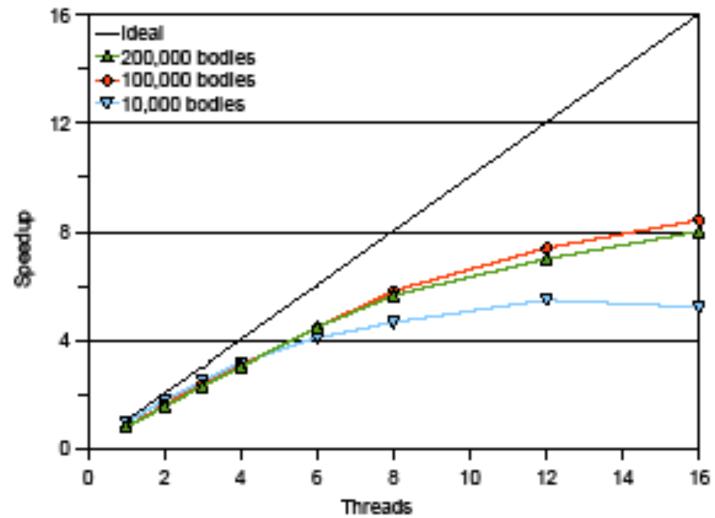


Barnes-Hut

- Optimization
  - static parallelization of particle-pushing
    - 90+ % of execution time
  - Galois runtime system but conflict-checking is turned off
- SPLASH-2 C implementation:
    - same scaling
    - roughly twice as fast (Java vs. C)
- Shows that static parallelization can be viewed as early-binding of scheduling decisions



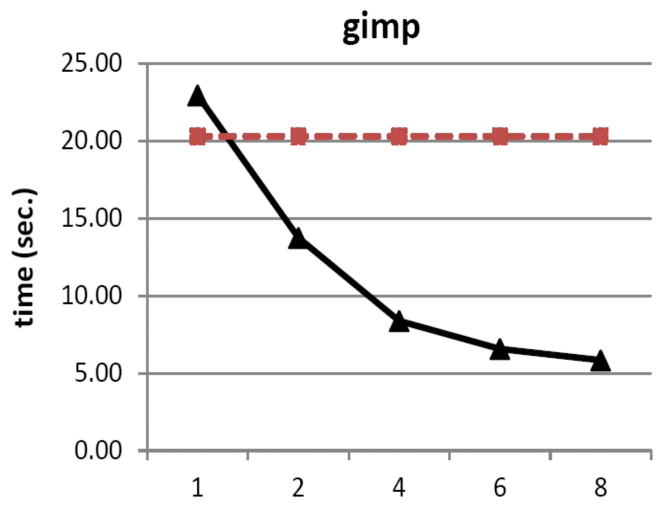
Sun Niagara-2

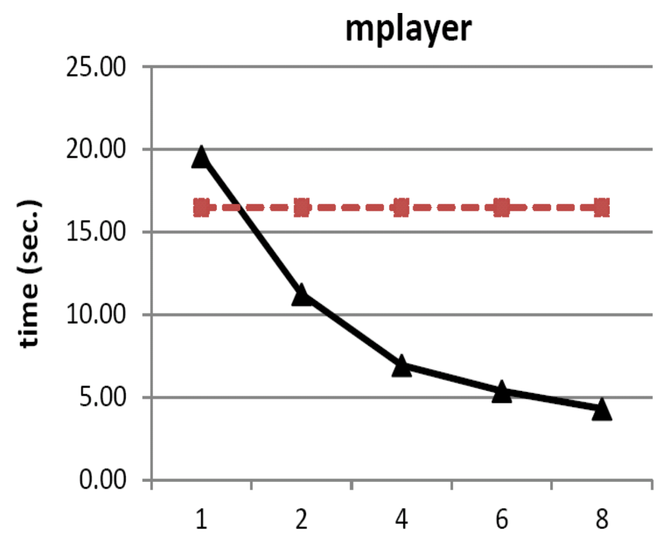


Nehalem

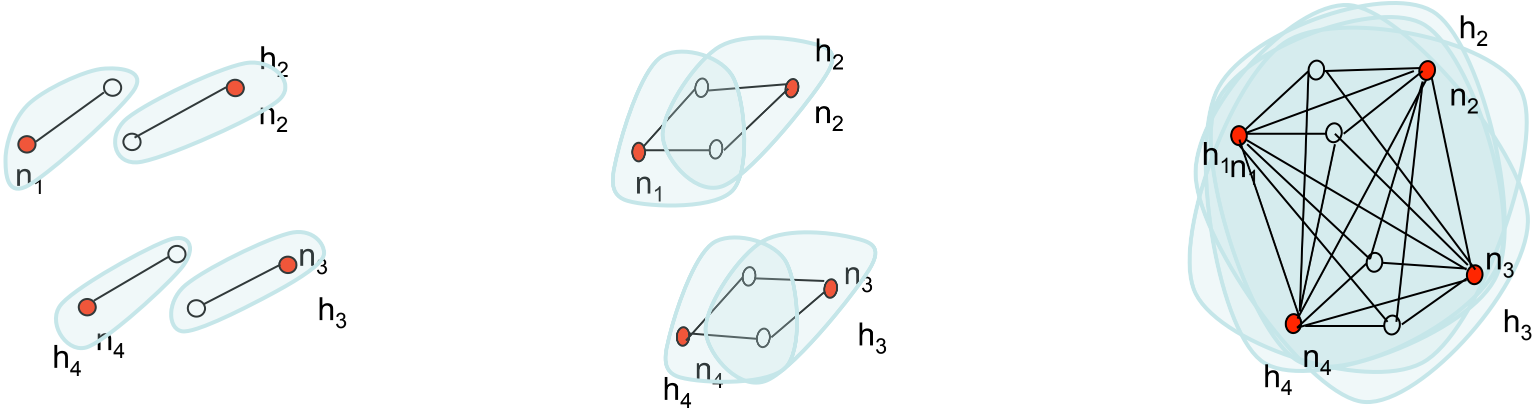
Andersen-style points-to analysis

- Algorithm formulation
  - solution to system of set constraints
  - 3 graph rewrite rules
  - speedup algorithm by collapsing cycles in constraint graph
- State of the art C++ implementation
  - Hardekopf & Lin
  - red lines in graphs
- "Parallel Andersen-style points-to analysis" Mendez-Lojo et al (OOPSLA 2010)





Ongoing work



- System building
  - current version of Galois, Lonestar, ParaMeter: http://iss.ices.utexas.edu/galois
  - ordered algorithms
- Algorithm studies:
  - other kinds of structure
  - intra-operator parallelism
  - locality
- Application studies
  - case studies of hand-optimized codes
- Compiler analysis
  - analyze and optimize code for operators
- Specializing data structure implementations to particular algorithms
  - can this be done semi-automatically?

Ongoing work (contd.)

- Kali project (with David Padua, UIUC)
  - system for exploiting
    - conventional data-parallelism
    - amorphous data-parallelism

Related work

- Transactional memory (TM)
  - Programming model:
    - TM: explicitly parallel (threads)
      - transactions: synchronization mechanism for threads
      - mostly memory-level conflict detection
    - Galois: Joe programs are sequential OO programs
      - ADT-level conflict detection
  - Where do threads come from?
    - TM: someone else's problem
    - Galois project: focus on sources of parallelism in algorithm
- Thread-level speculation
  - Programming model:
    - Galois: separation between ADT and its implementation is critical
      - permits separation of Joe and Stephanie layers (cf. relational databases)
      - permits more aggressive conflict detection schemes like commutativity relations
    - TLS: FORTRAN/C, so no separation between ADT and implementation
  - Programming model:
    - Galois: don't-care non-determinism plays a central role
    - TLS: FORTRAN/C, so only ordered algorithm

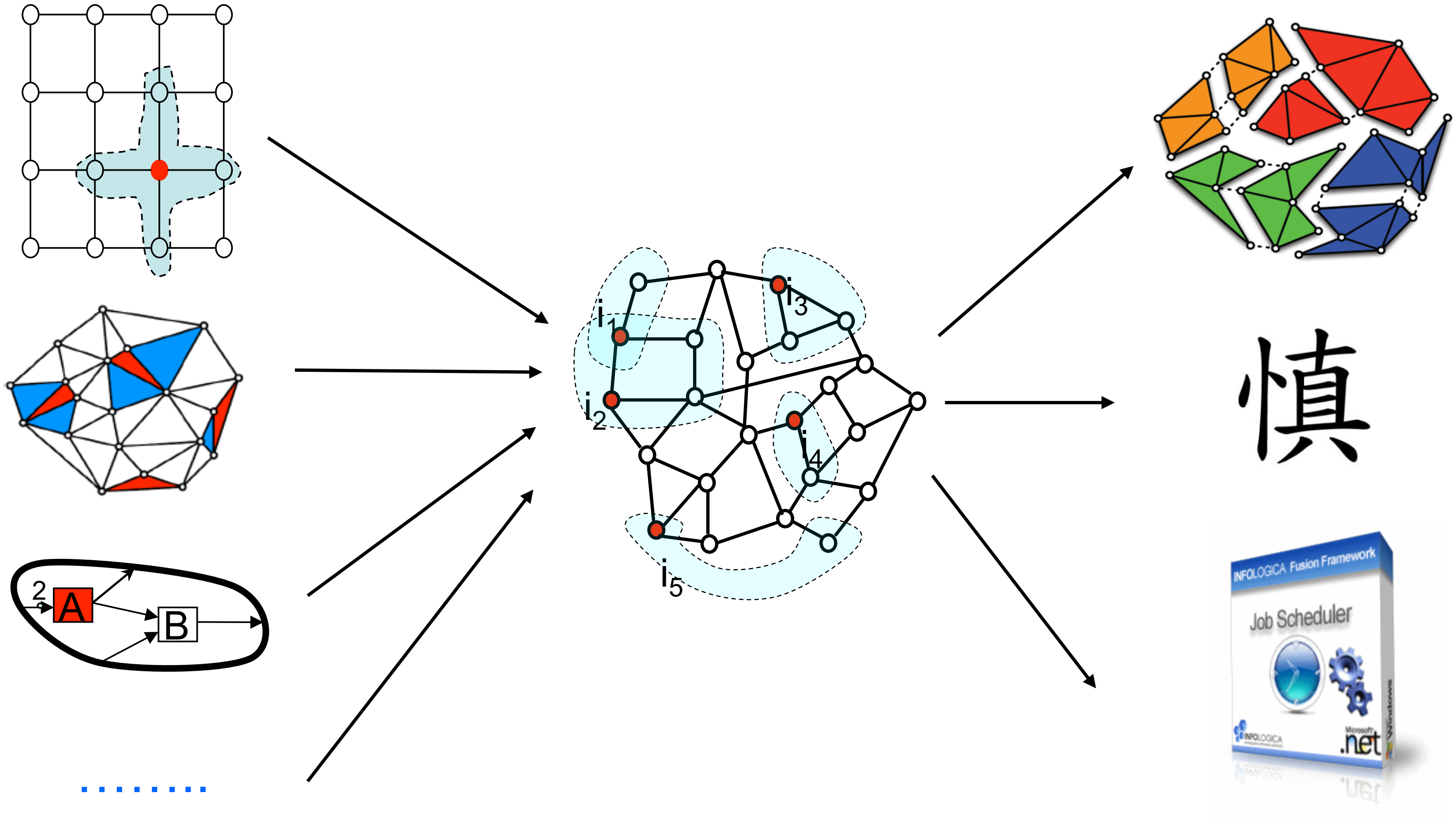
Summary

- Current 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

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

structure

Science of Parallel Programming



Seemingly unrelated algorithms

Unifying abstractions

Specialized models that exploit structure