Fifty Years of Parallel Programming: Ieri, Oggi, Domani

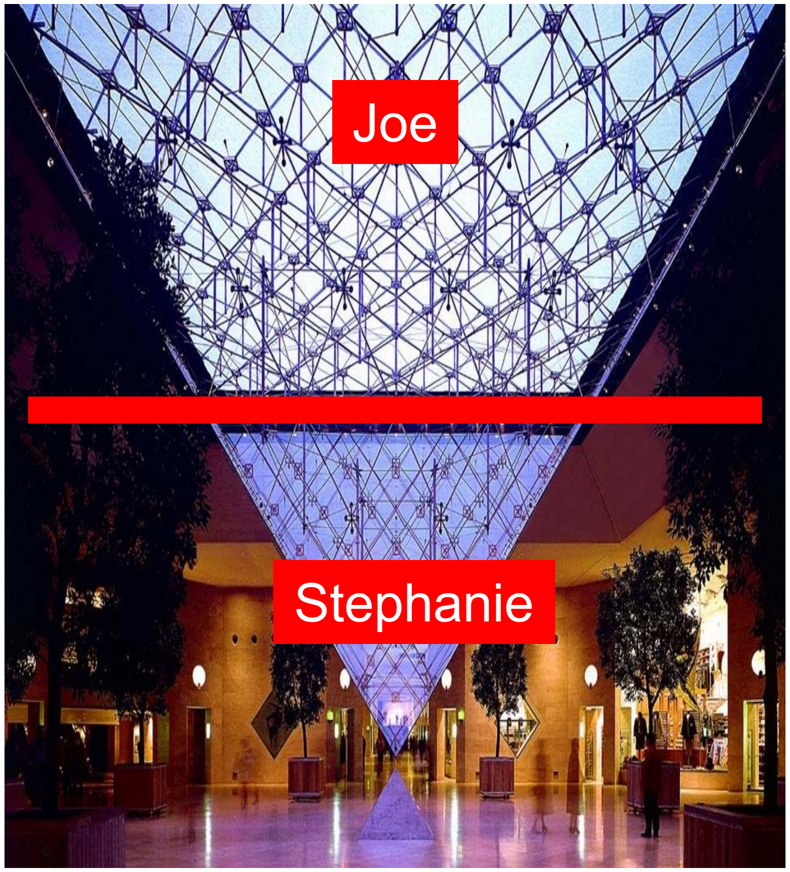
Keshav Pingali
The University of Texas at Austin

Overview

- Parallel programming research started in mid-60's
- Goal:
  - Productivity for Joe: abstractions to hide complexity of parallel hardware
  - Performance from Stephanie: implement abstractions efficiently

What should these abstractions be and how are they implemented?

- Yesterday:
  - Six lessons from the past
- Today:
  - Model for parallelism and locality
- Tomorrow:
  - Research challenges

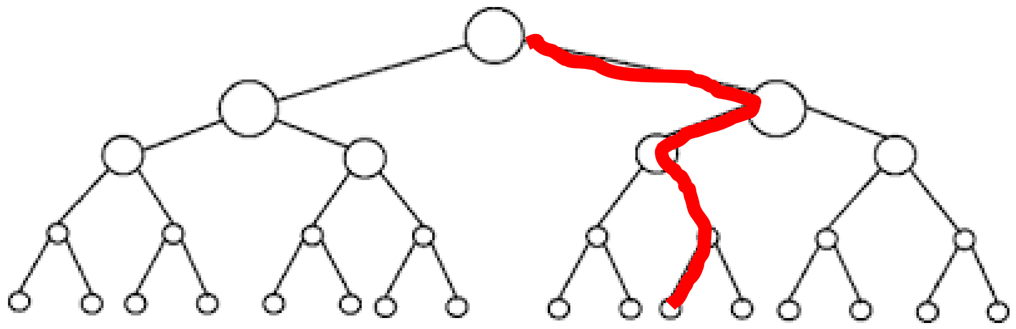


"Scalable" parallel programming:

few Stephanies, many Joes

(1) It’s better to be wrong once in a while than to be right all the time.

Impossibility of exploiting ILP: [c. 1972]



Flynn bottleneck

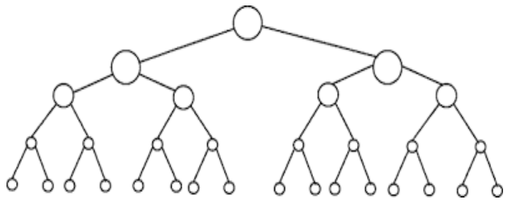
"..Therefore, we must reject the possibility of bypassing conditional jumps as being of substantial help in speeding up execution of programs.

In fact, our results seem to indicate that even very large amounts of hardware applied to programs at runtime do not generate hemibel (> 3x) improvements in execution speed."

Riseman and Foster, IEEE Trans. Computers, 1972

Exploiting ILP [Fisher, Rau c.1982]

- Key idea:
  - Branch speculation
  - Dynamic branch prediction [Smith,Patt]
  - Backup/re-execute if prediction is wrong



- Infallibility is for popes, not parallel computing

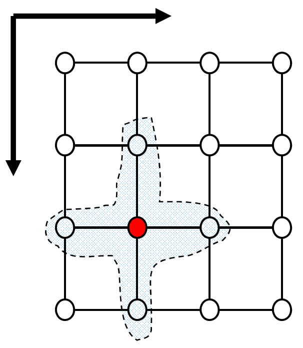
- Broader lesson:
  - Runtime parallelization: essential in spite of overhead and wasted work
  - Compilers: only *part* of the solution to exploiting parallelism

(2) Aunque la mona se vista de seda, mona se queda.

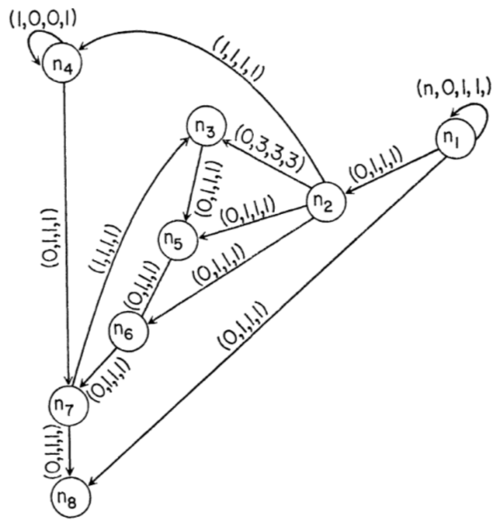
Dependence graphs are not the right foundation for parallel programming

Thread-level parallelism

- Dependence graph [Karp/Miller66,Dennis 68,Kuck72]
  - Nodes: tasks, edges: ordering of tasks
  - Independent operations: execute in parallel
- Dependence-based parallelization
  - Program analysis [Kuck72,Feautrier92]: stencils, FFT, dense linear algebra
  - Inspector-executor [Duff/Reid77,Saltz90]: sparse linear algebra
  - Thread-level speculation [Jefferson81,Rauchwerger/Padua95]: executor-inspector
- Works well for HPC programs
- Key assumptions:
  - Gold standard is a sequential program
  - Dependences must be removed/respected by parallel execution



Gauss-Seidel: 5-point stencil



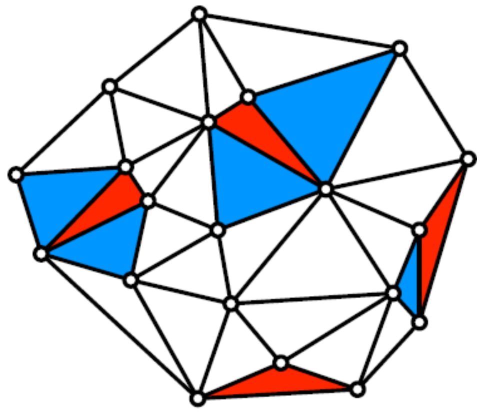
Computation graph for G-S: [Karp and Miller, 1966]

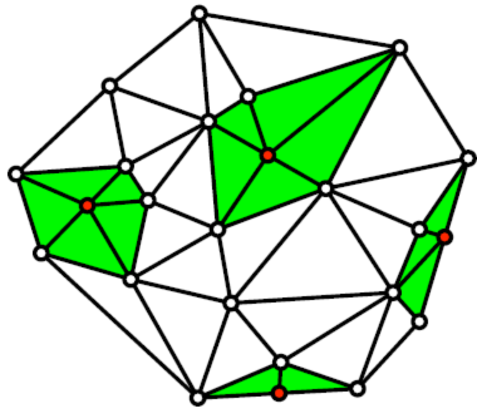
Beyond HPC

- Many graph algorithms
  - Tasks can generate and kill other tasks
  - Unordered: tasks can be executed in any order in spite of conflicts
  - Output may be different for different execution orders, all acceptable

Don’t-care non-determinism
- Arises from under-specification of execution order

- My opinion:
  - Dependence graphs are not right abstraction for such algorithms
  - No gold standard sequential program
- Questions:
  - What is the right abstraction?
  - Relation to dependence graphs?


Before


After

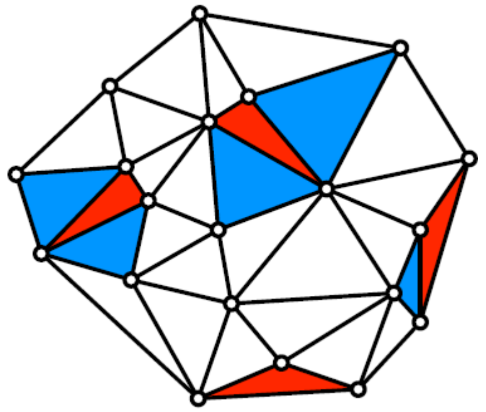
Delaunay mesh refinement
Red Triangle: badly shaped triangle
Blue triangles: cavity of bad triangle

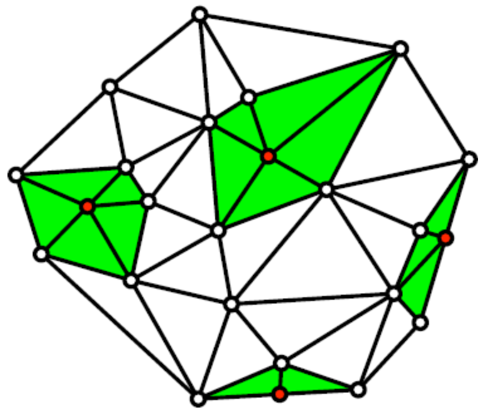
(3) Study algorithms and data structures, not programs*.

* Wirth: Algorithm + Data structure = Program

Programs vs. Algorithms + Data structures

```
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());
}
```


Before


After

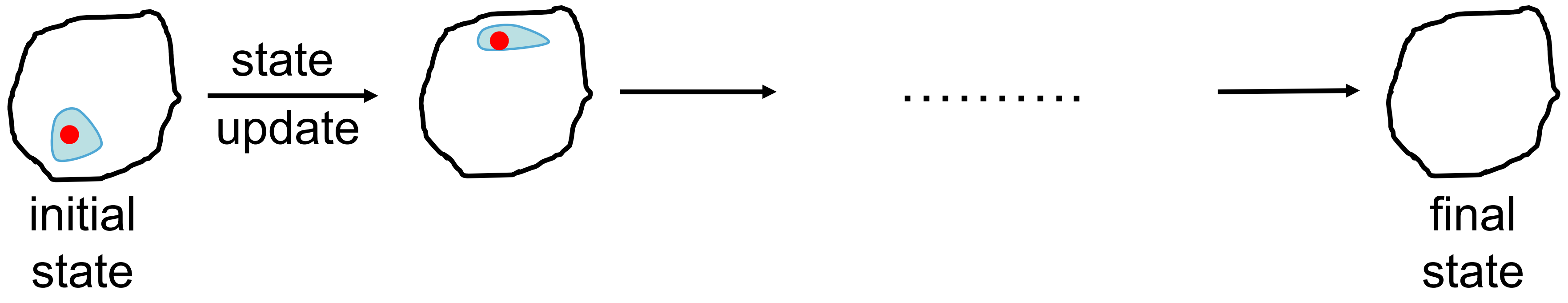
Program for DMR

Algorithm + Data structure

(4) Algorithms should be expressed using data-centric abstractions.

Operator formulation of algorithms

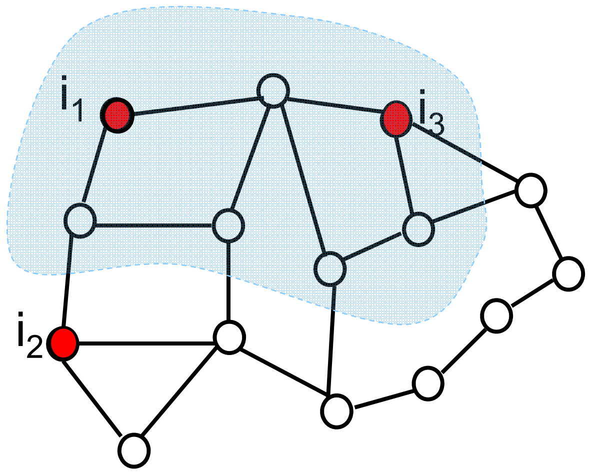
von Neumann programming model

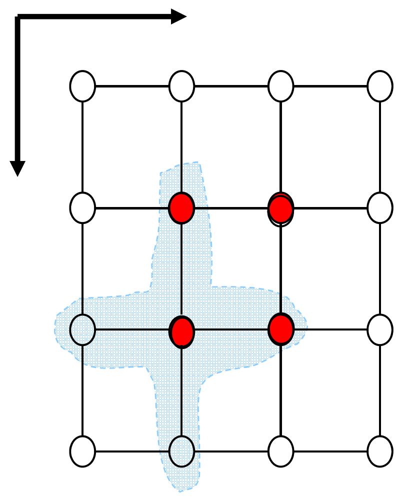


Algorithm
- State update: assignment statement (local view)
- Schedule: control-flow constructs (global view)

von Neumann bottleneck [Backus 79]

Operator formulation





● : active node

▨ : neighborhood

- Algorithm
  - State update: Operator (local view)
  - Schedule (global view)
    - Location (where?)
      - Topology-driven
      - Data-driven
    - Ordering (when?)
      - Unordered
      - Ordered

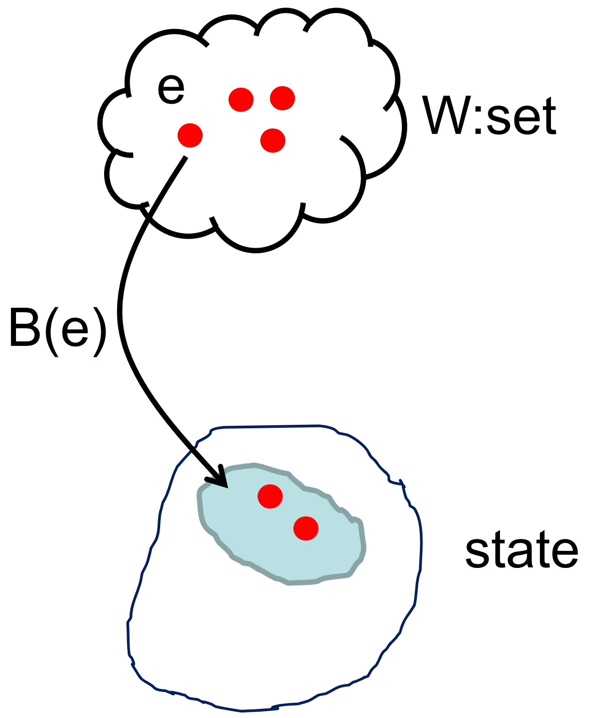
No distinction between sequential/parallel, regular/irregular algorithms
Unifies seemingly different algorithms for same problem

Joe: specifying unordered algorithms

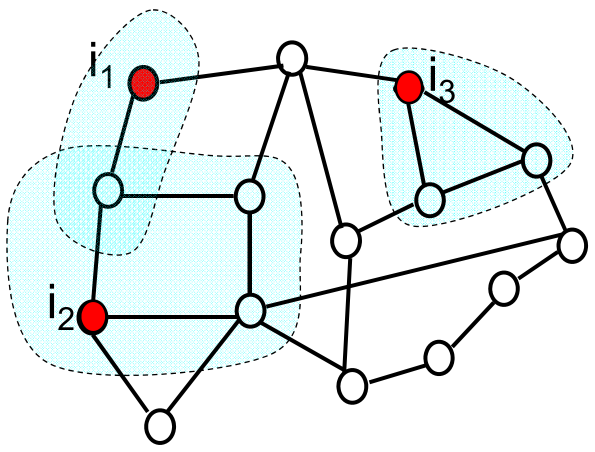
- Set iterator: [Schwartz70]

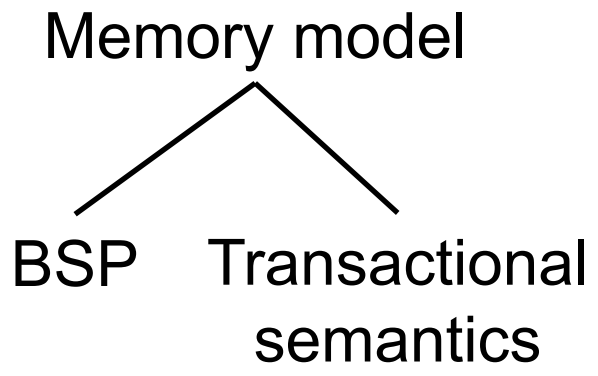
```
for each e in W:set  do
     B(e)  //state update
```

  - don't-care non-determinism: implementation free to iterate over set in any order
  - optional soft priorities on elements (cf. OpenMP)
- Captures the "freedom" in unordered algorithms



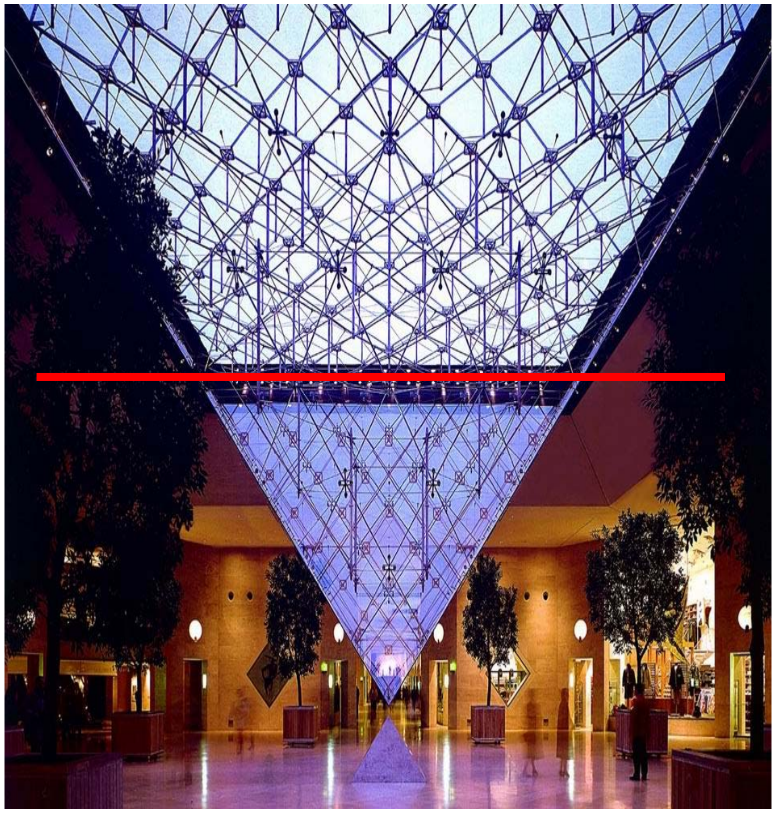
Parallelism





- Memory model:
  - When do writes by one activity become visible to other activities?
- Two popular models:
  - Bulk-synchronous Parallel(BSP) [Valiant 90]
  - Transactional semantics [everyone else]
- How should transactional semantics for operators be implemented by Stephanie?
  - One possibility: Transactional Memory(TM) [Herlihy/Moss, Harris]

$$\begin{tikzcd}\text{Construct} \arrow[d, "?"] \\ \text{Implementation}\end{tikzcd}$$



(5) Exploit context and structure for efficiency.

Tailor-made solutions are better than ready-made solutions.

RISC vs. CISC [c. 80’s-90’s]

- CISC philosophy:
  - Map high-level language (HLL) idioms directly to instructions and addressing modes
  - Makes compiler’s job easier
- RISC philosophy:
  - Minimalist ISA
  - Sophisticated compiler generated code for HLL constructs tailored to
    - *program context*
    - *structure*

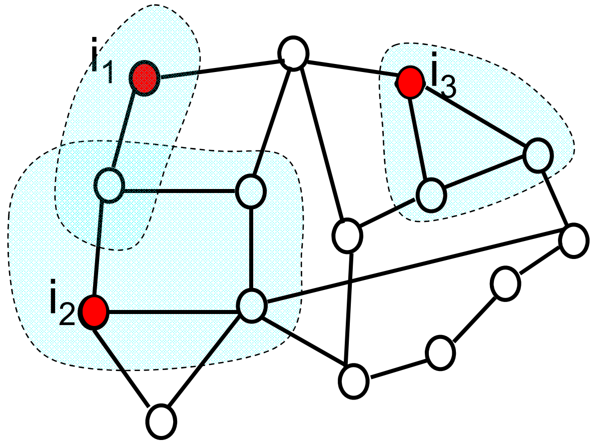
```
for  (int i=0; i<N; i++) {

                …..a[i]…..
}
```

Exploiting context for efficiency

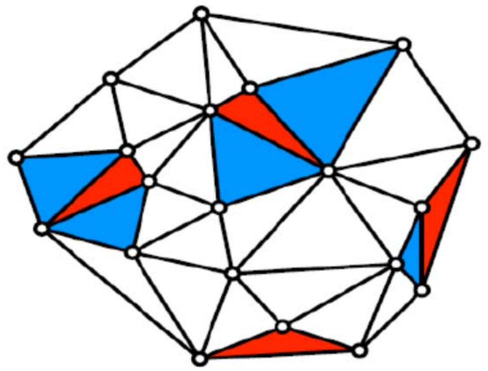
Transactional semantics: exploiting context

Binding time: when are active nodes and neighborhoods known?

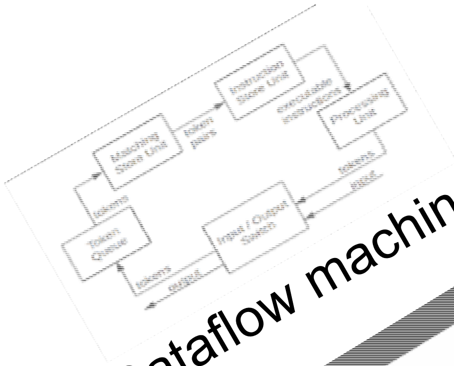


Compile-time	Dependence graphs (stencils,dense LA)
After input is given	Inspector-executor (SGD,sparse LA)
During program execution	Interference graph (DMR, chaotic SSSP)
After program is finished	Optimistic parallelization (Time-warp)

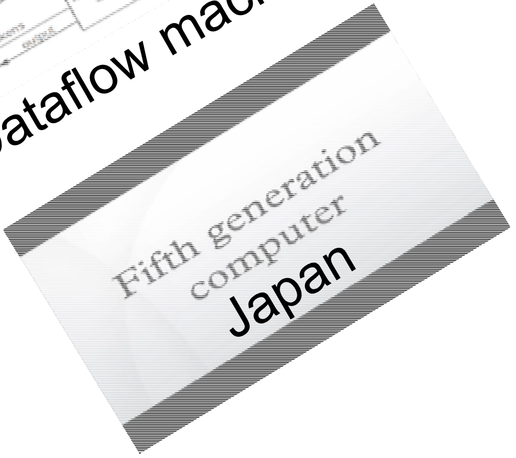
Transactional semantics: exploiting structure



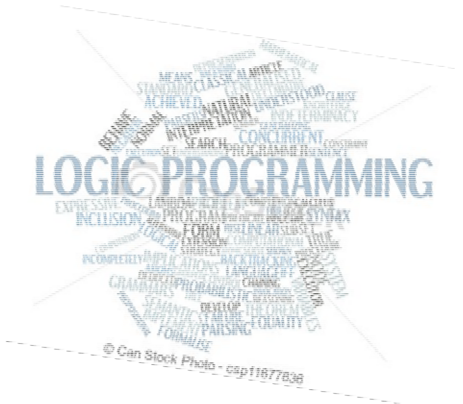
- Operators have structure
  - Cautious operators: read entire neighborhood before any write, so no need to track writes
  - Detect conflicts at ADT level, not memory level
- Generate customized code using atomic instructions
  - RISC-like approach to ensuring transactional semantics



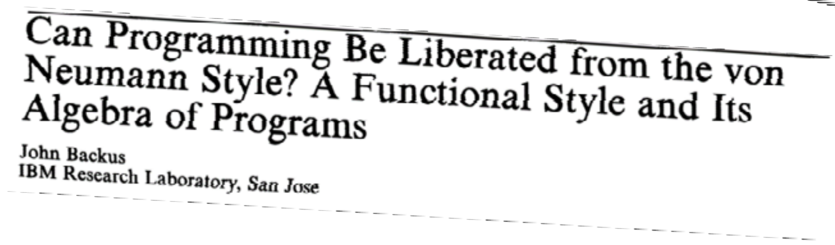
Dataflow machine


Fifth generation computer

Japan



CODE WRITTEN IN HASKELL IS GUARANTEED TO HAVE NO SIDE EFFECTS.
...BECAUSE NO ONE WILL EVER RUN IT?

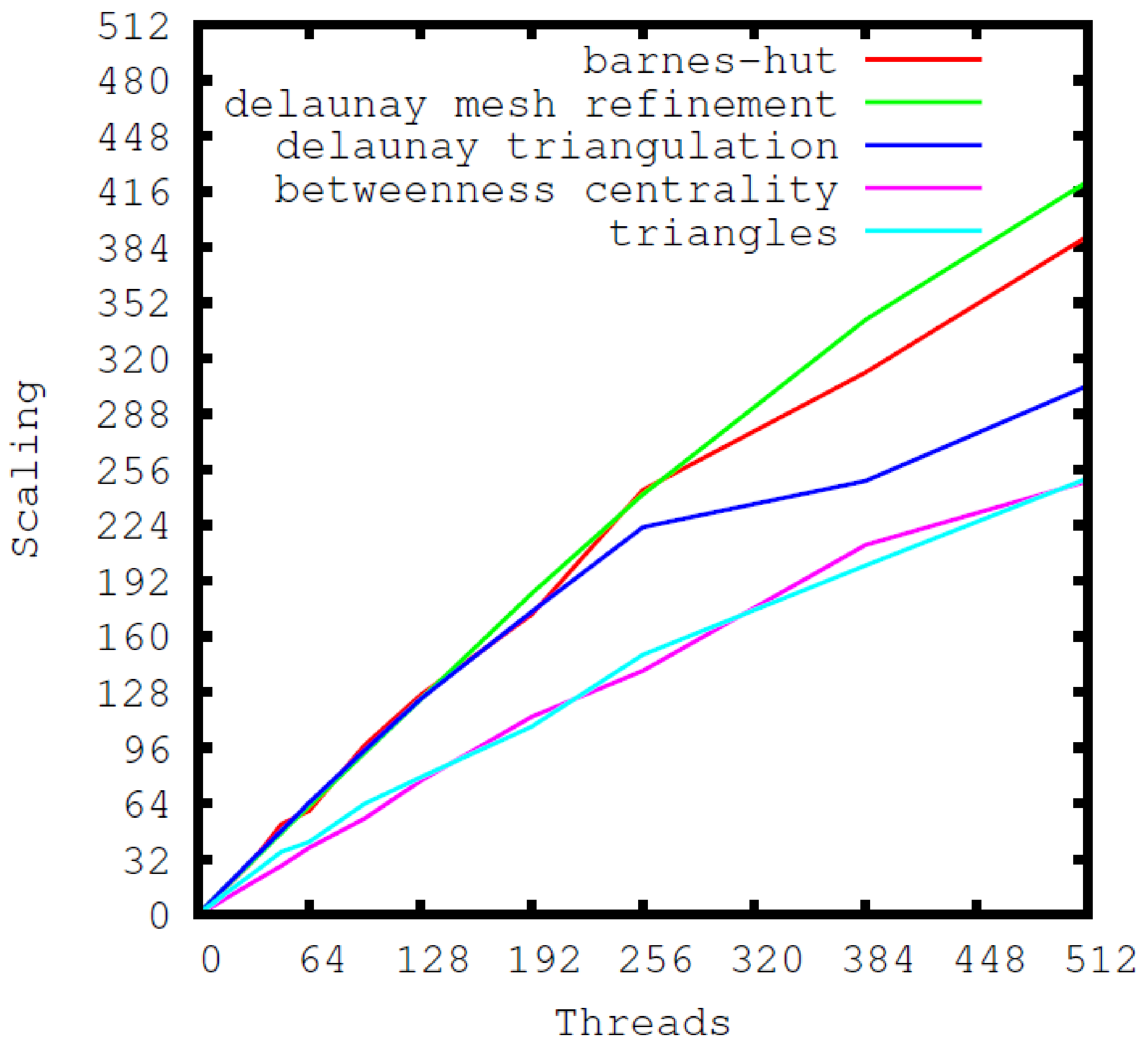

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose

(6) The difference between theory and practice is smaller in theory than in practice.

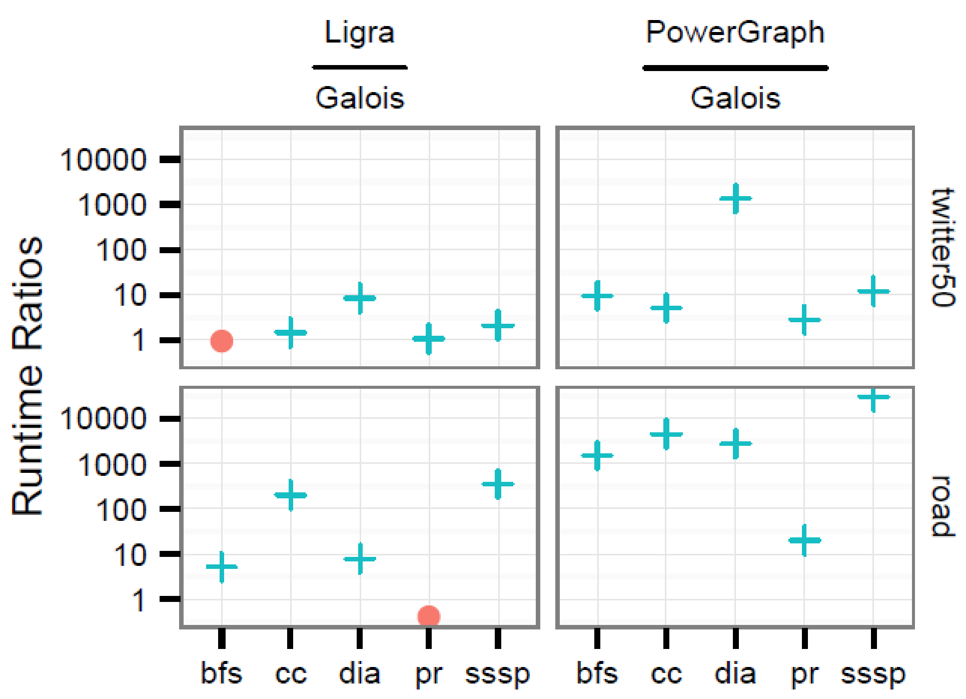
McKinsey & Co: “So what?”

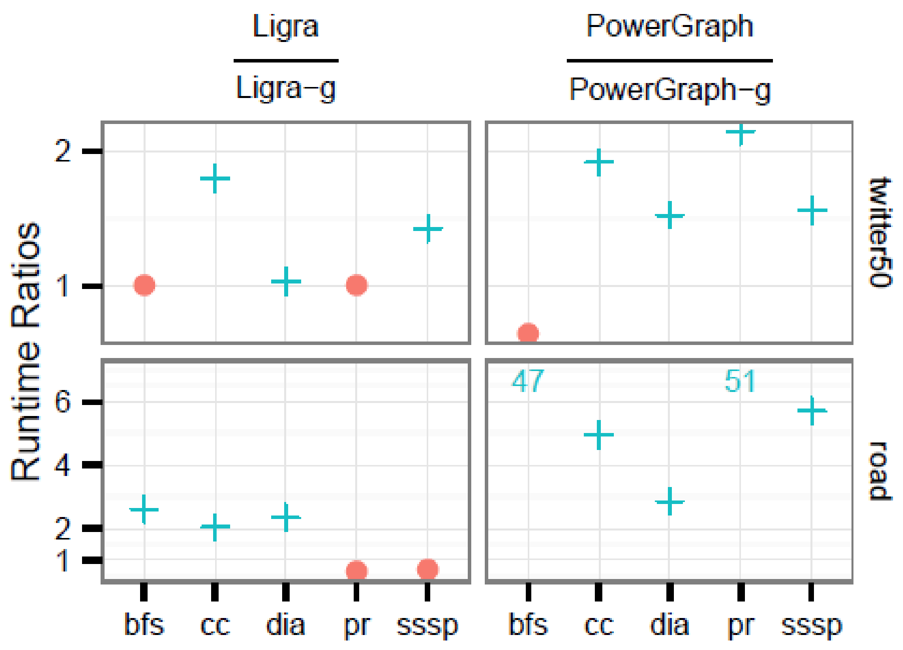
Galois: Performance on SGI Ultraviolet



Lenharth et al. : IEEE Computer Aug 2015

Galois: Graph analytics



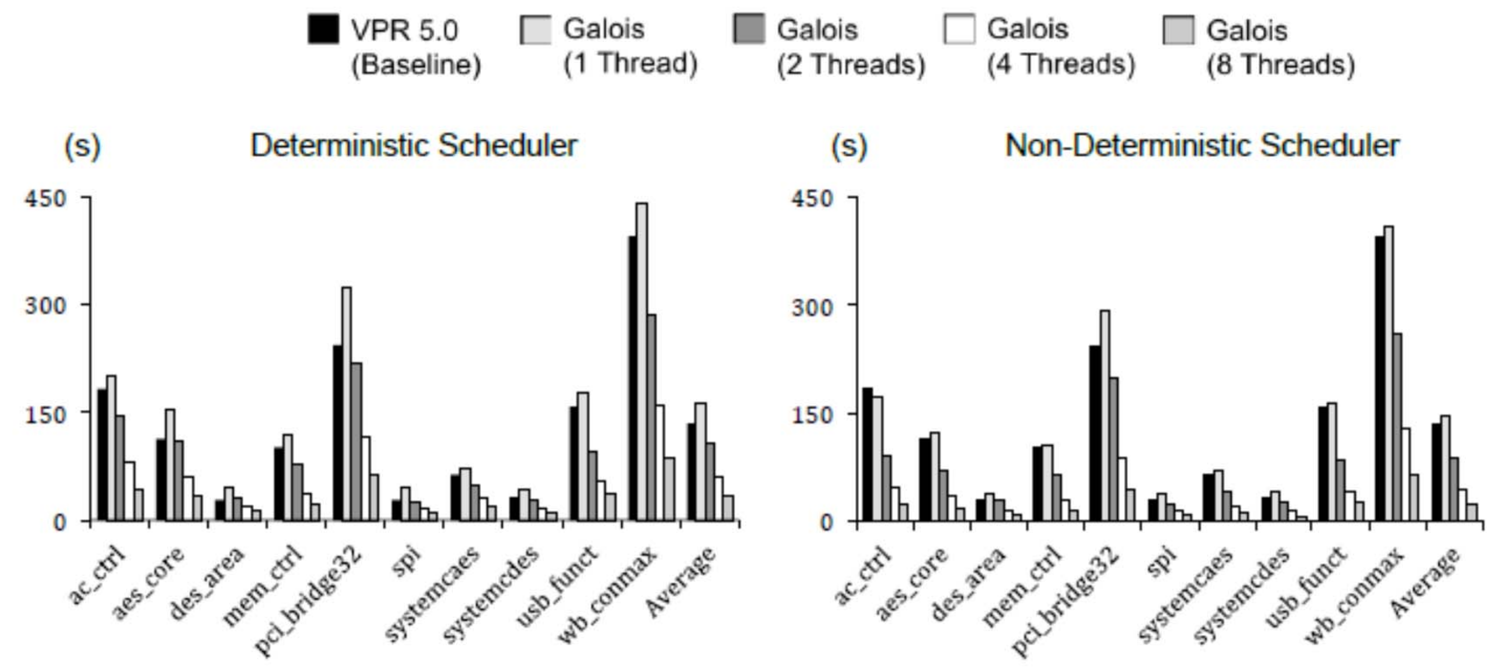


- Galois lets you code more effective algorithms for graph analytics than DSLs like PowerGraph (left figure)
- Easy to implement APIs for graph DSLs on top on Galois and exploit better infrastructure (few hundred lines of code for PowerGraph and Ligra) (right figure)

"A lightweight infrastructure for graph analytics" Nguyen, Lenharth, Pingali (SOSP 2013)

FPGA Tools

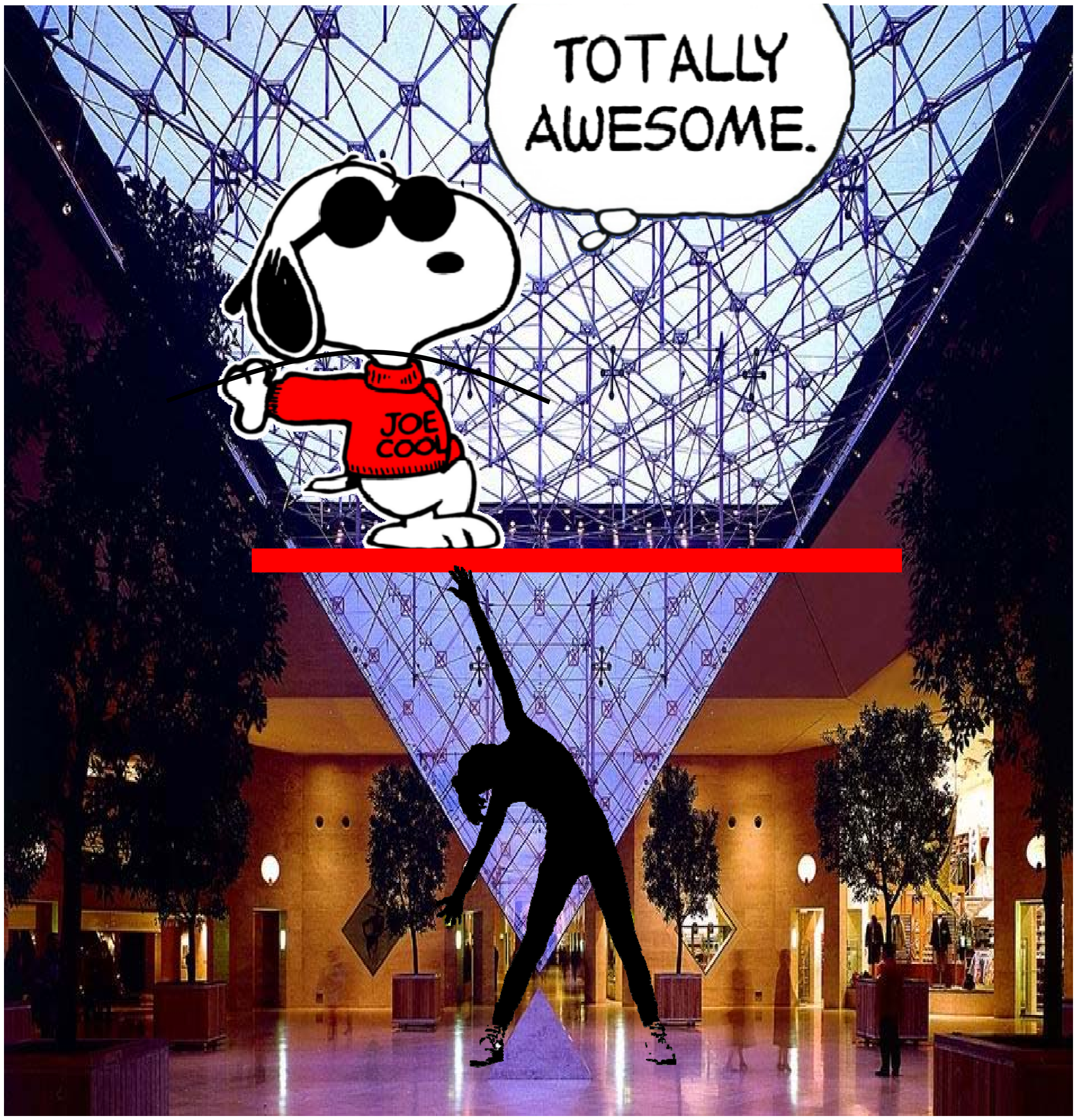
Maze Router Execution Time



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

Domani



Research problems

- Heterogeneity/energy/etc.
  - Multicores/GPUs/FPGAs
- Synthesize parallel implementations from specifications
  - SMT solvers [Gulwani], planning [Prountzos15]
- Fault tolerance
  - Contract between hardware and software?
  - Need more sophisticated techniques than CPR [Spark]
  - Exploit program structure to tailor fault tolerance?
- Correctness
  - Formally verified compilers [Hoare/Misra, Coq]
  - Proofs are programs: what does this mean for us?
- Inexact computing
  - Customized consistency models [parameter server in ML]
  - Principled approximate computing [Rinard,Demmel]

Patron saint of parallel programming

“Pessimism of the intellect, optimism of the will”

Antonio Gramsci (1891-1937)

Lessons

- It’s better to be wrong once in a while than to be right all the time.
  - Runtime parallelization essential in spite of overheads and wasted work.
- Aunque la mona se vista de seda, mona se queda.
  - Dependence graphs are not the right foundation for parallel programming.
- Study algorithms and data structures, not programs.
  - Leads to a deeper understanding of program behavior
- Algorithms should be structured using data-centric abstractions.
  - Parallel program = Operator + Schedule + Parallel data structure
- Exploit context and structure for efficiency.
  - Tailor-made solutions are usually better than ready-made solutions
- The difference between theory and practice is smaller in theory than in practice.
  - Always ask yourself “So what?”