Some Lessons from Fifty Years of Parallel Programming

Keshav Pingali
The University of Texas at Austin

Overview

Parallel programming research: 1970-

Productivity for Joe programmers
Abstractions to hide complexity of parallel hardware

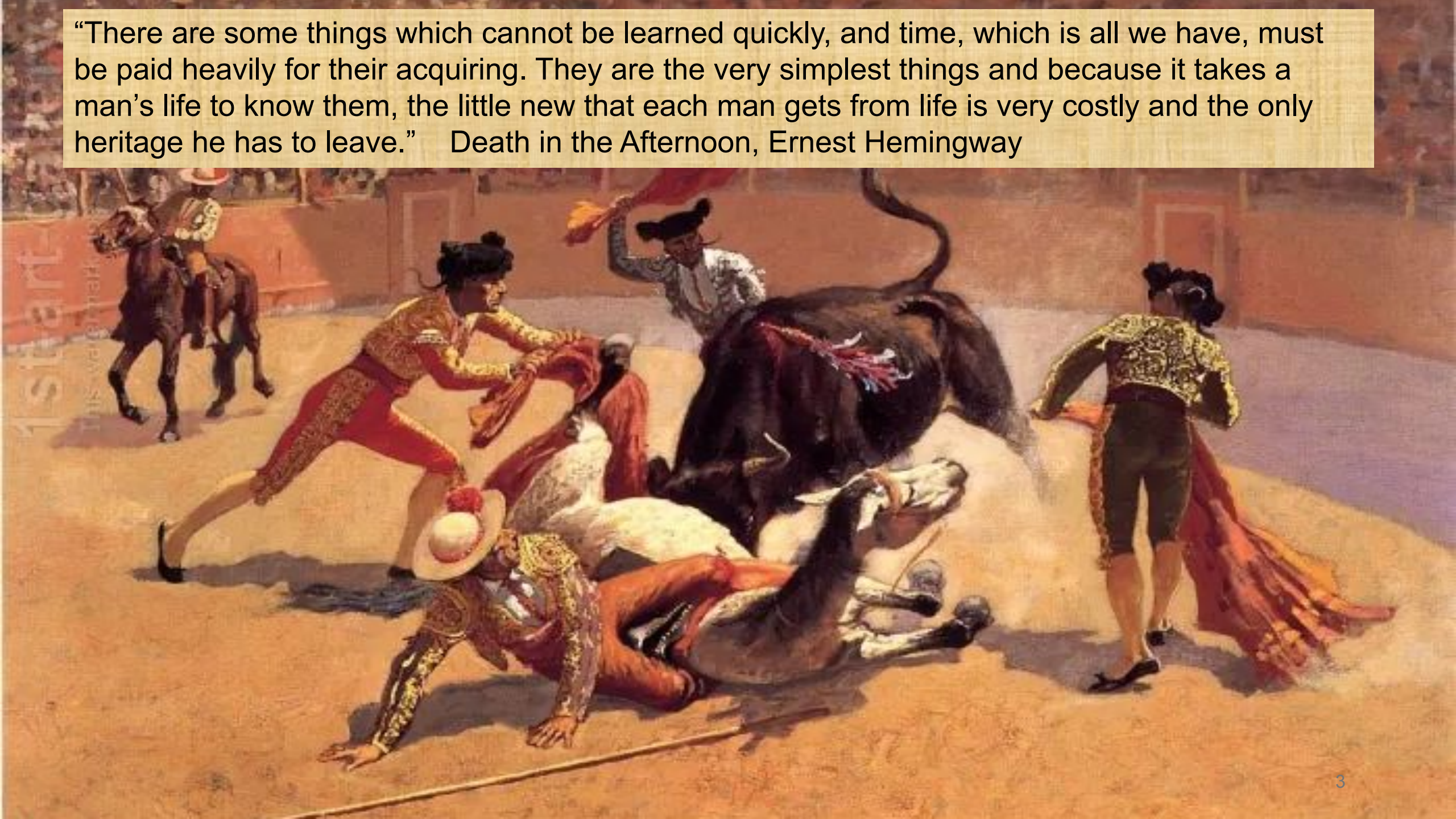
Performance from Stephanie programmers
Implement abstractions efficiently

What should these abstractions be?

How are they implemented efficiently?

“Scalable” parallel programming

“There are some things which cannot be learned quickly, and time, which is all we have, must be paid heavily for their acquiring. They are the very simplest things and because it takes a man’s life to know them, the little new that each man gets from life is very costly and the only heritage he has to leave.” Death in the Afternoon, Ernest Hemingway



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

Instruction-level Parallelism (1960-)

Execute basic block instructions in parallel if there are no dependences between them

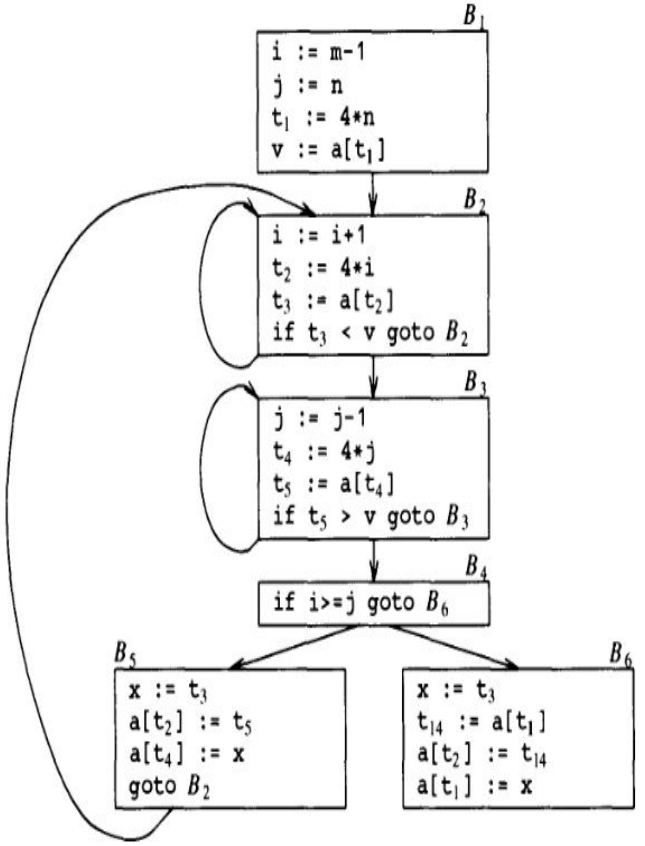
- CDC 6600, IBM 360/91,…

Problem

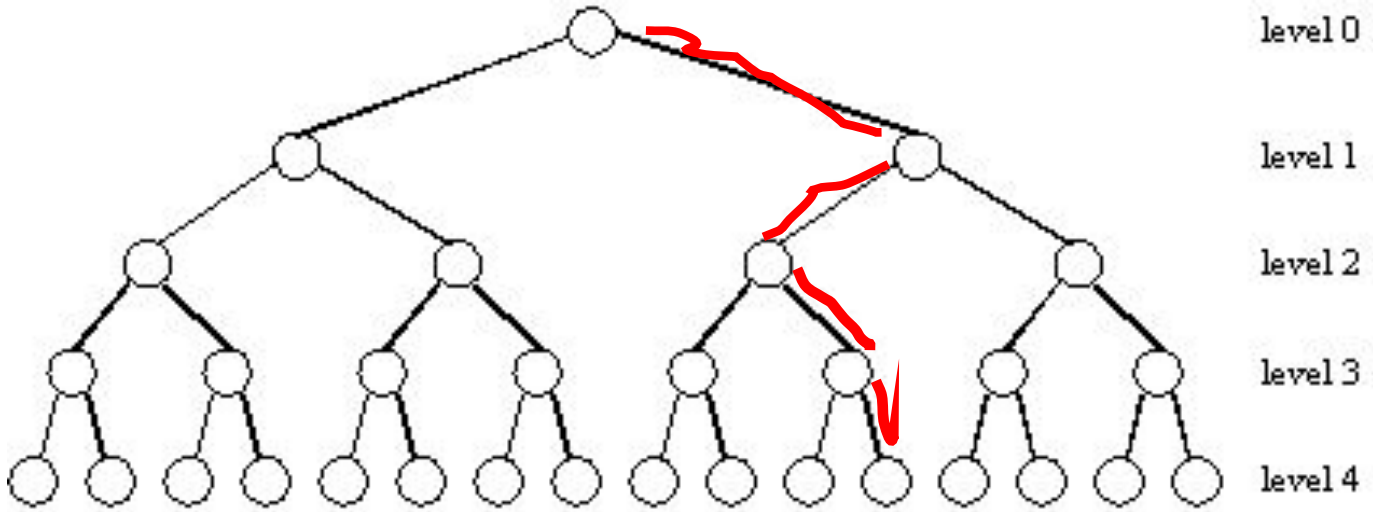
- Average (RISC) basic block has 5 instructions
- May not know which side of branch will be taken until basic block instructions are completed

Solution

- Fetch (and conditionally decode/execute) instructions from both sides of the branch



Impossibility of exploiting ILP: [c. 1972]



"..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

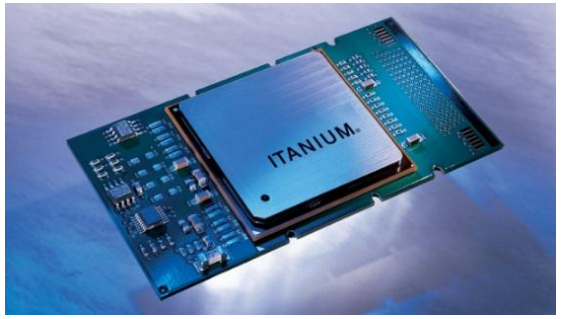
Key insight

Branch speculation

- *Guess* which way branch will go and execute instructions only from that side
- Backup/re-execute (if guess is wrong)
- Dynamic branch prediction (to guess right usually)
- Used in all superscalar processors today

Speculative execution

- Essential mechanism in *parallel* computing
- Not needed in sequential computing



Infallibility is for popes, not parallel computing

(2) (Static) dependence graphs are not the right foundation for parallelism.

Auto-vectorization

SIMD/Vector machines: ILLIAC IV [1974],Cray-1[1975],..

Vector code from array programs: Kuck,Kennedy, Fran Allen

- Compute dependence graph of innermost loop
- If iterations independent, generate vector code

Dependence tests for affine programs

- GCD, Banerjee, Lambda, Omega,…
- Integer Linear Programming (ILP): Feautrier [1972]

Dependence graph successes

- Dense LA w/o pivoting (compiler)
  - pivoting (fractal symbolic analysis, Menon and P. 2003]
- Stencil codes (compiler)
- Sparse direct methods: Duff/Reid[77] (library)

```
for i = 1,N
   A[i] = A[i]+1

for i = 2,N
   A[i] = A[i-1]+1

for i = 1,N
  A[2i] =
A[2i-1]+1
```

$1 \le iw < ir \le N$

$2*iw = 2*ir - 1$

Beyond matrix programs

Irregular programs

- Pointer-based data structures: lists, trees, graphs

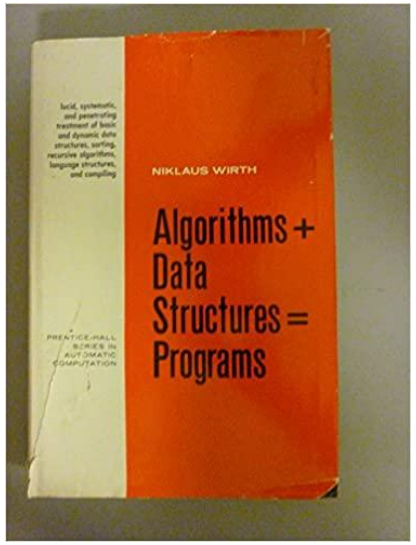
Dependence-based parallelization

- Points-to & shape analysis [1980-

Fails to find parallelism in irregular programs

- No parallelism in these applications?
- More sophisticated dependence analyses needed?

```
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());
}
```



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

The trouble with benchmarks

Systems research: large benchmark suites

- SPEC, Perfect Club, …

Benchmark suites necessary, not sufficient

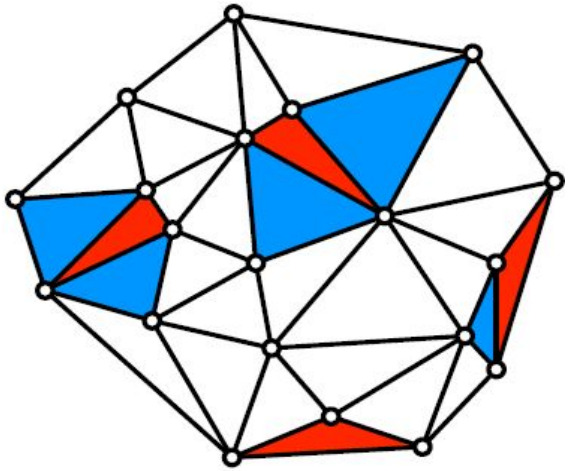
- Encode programming patterns, data structures and algorithms not central to the problem being solved

Understand underlying algorithms and data structures

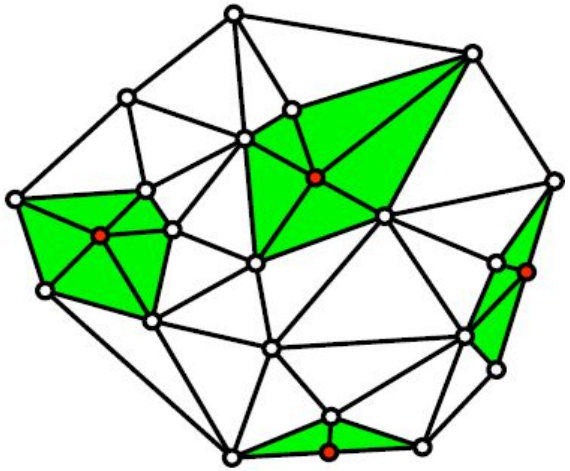
- Knowing what can be changed is essential

```
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());
}
```

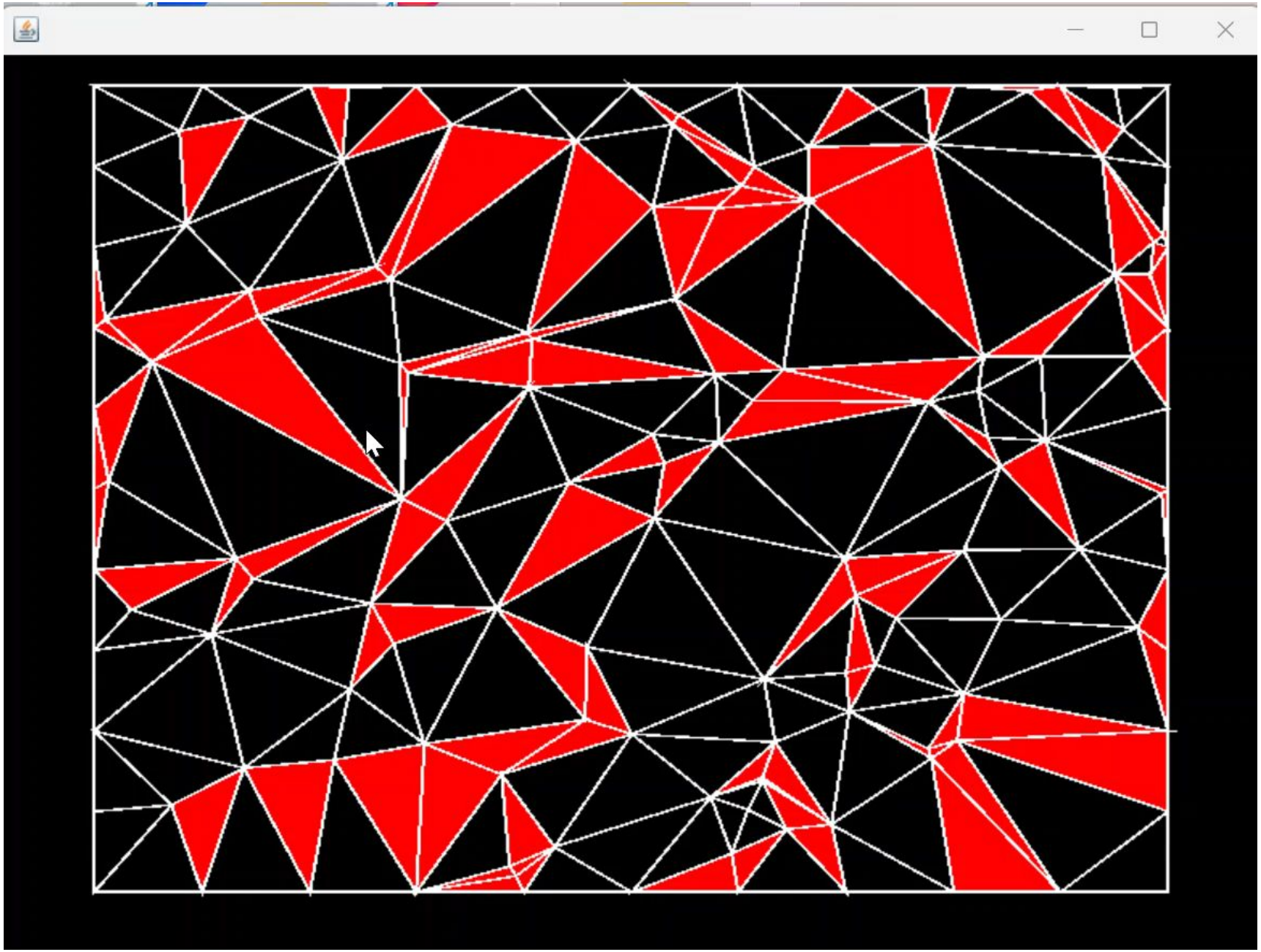
Program for



Before



After



(4) Nondeterminism is your enemy, don’t-care nondeterminism is your friend

Don’t care nondeterminism

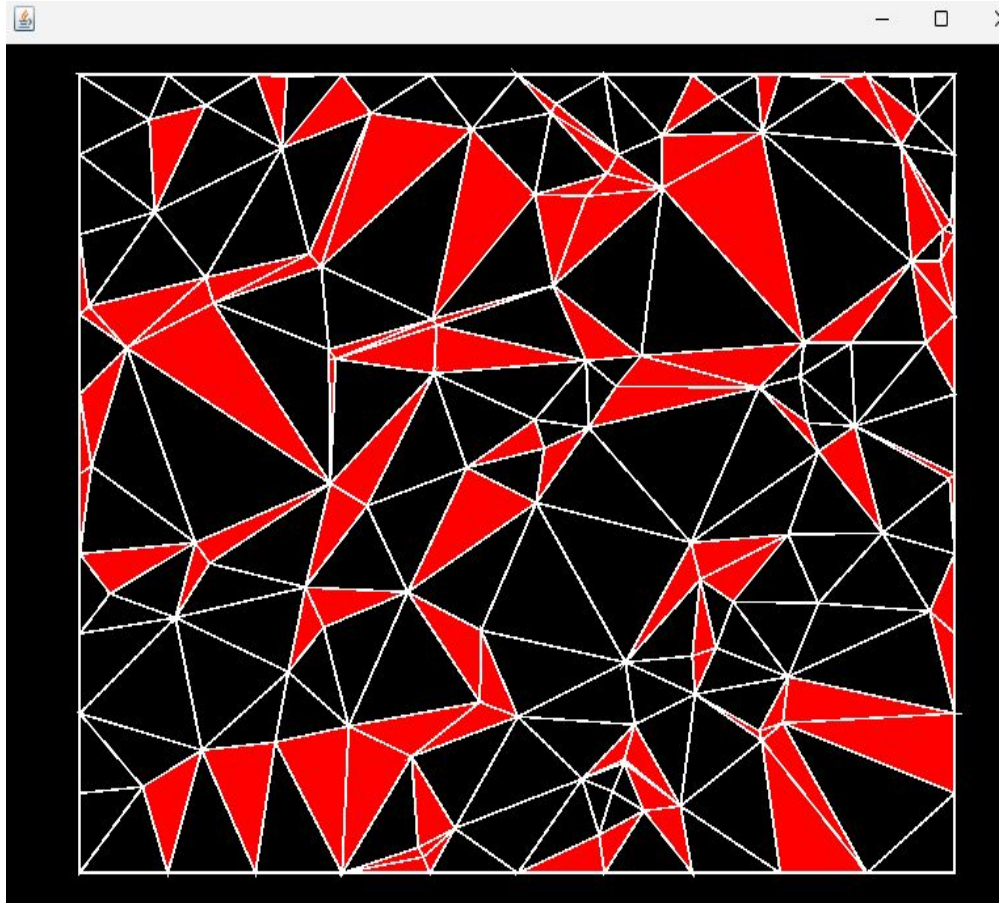
Final mesh in DMR

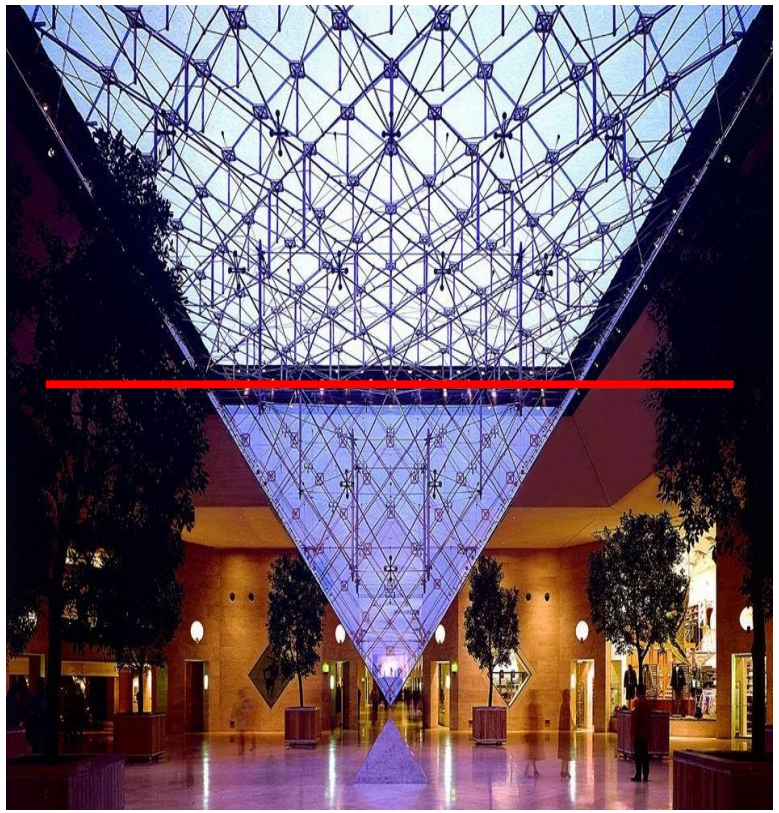
- Depends on order of processing bad triangles
- *Nondeterministic* output even with sequential execution

However, all output meshes are acceptable

Don’t-care nondeterminism

- Freedom of choice is important for parallelism
- Guarded commands [Dijkstra75, Unity88]





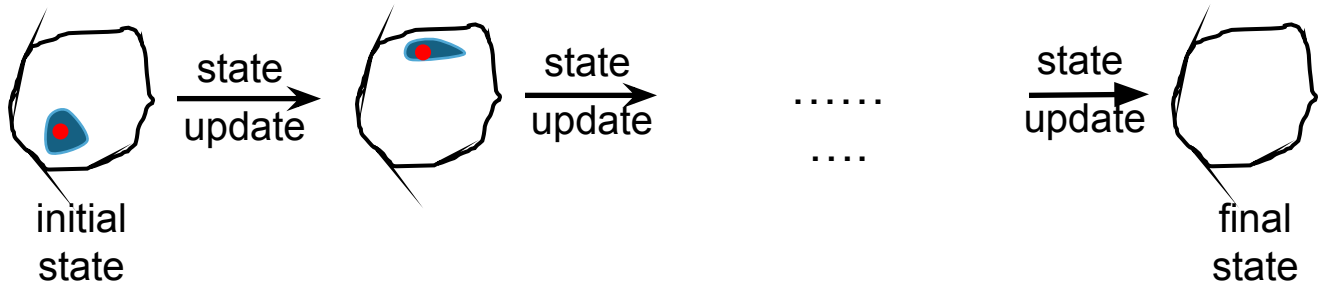
Parallel Program

Operator +Schedule

Parallel data structures

(5) Express algorithms using data-centric abstractions.

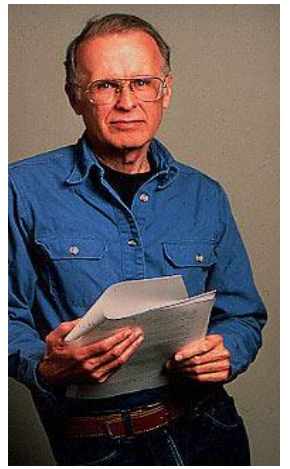
von Neumann programming model (control-centric)



Algorithm:
- State update: Assignment statement
- Schedule: Control-flow constructs

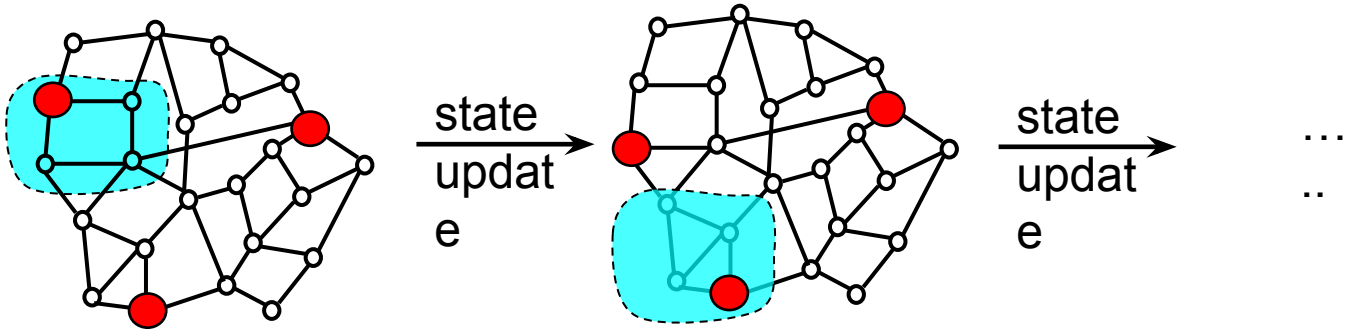
Runtime: Program counter

von Neumann bottleneck [Backus 79]



"Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs" Turing Award lecture (1978)

Operator formulation (data-centric)



: active node

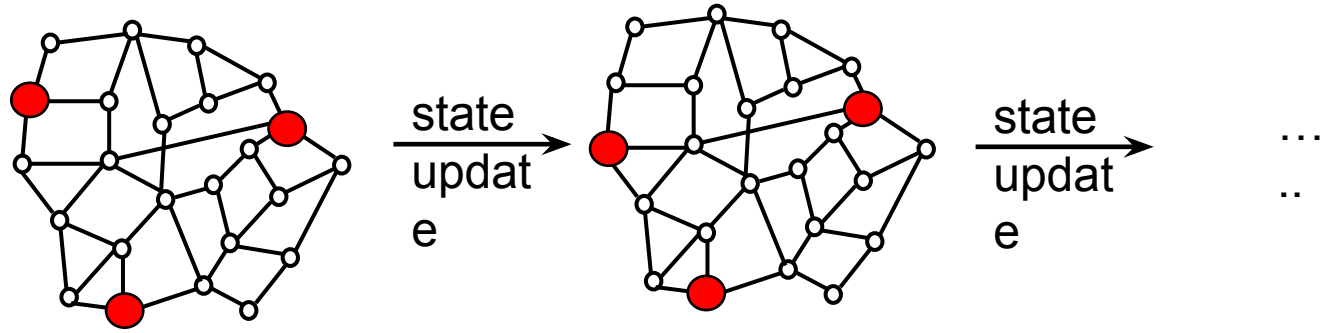
: neighborhood

Algorithm:

- State update: Operator
- Schedule: Ordering between active nodes

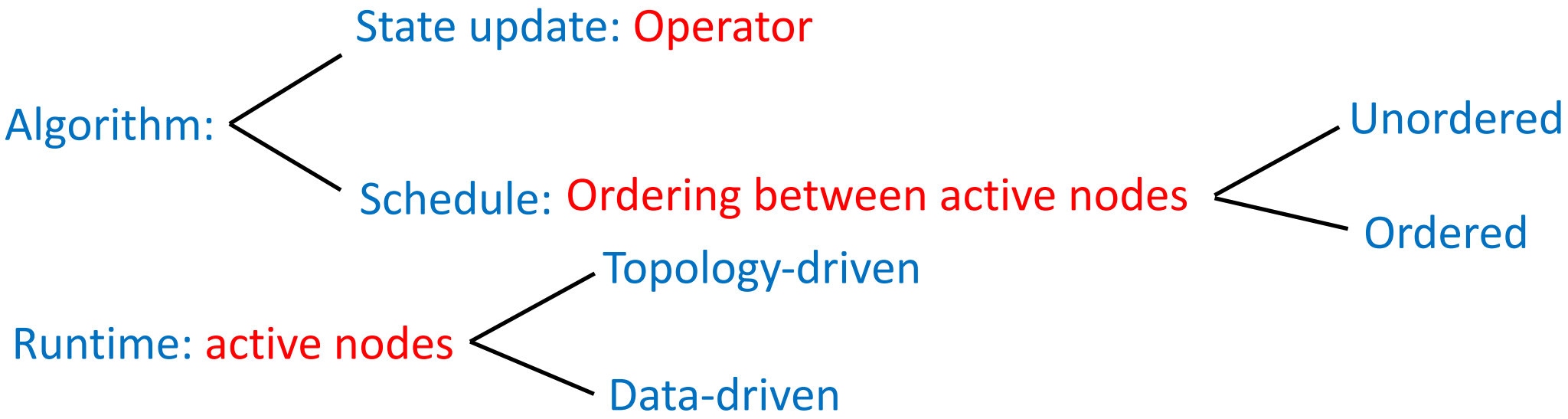
Runtime: active nodes

Operator formulation (data-centric)



● : active node

⬭ : neighborhood



Programming model (Joe)

Unordered algorithms

- Set iterators: [SETL,Schwartz70]

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

- Don't-care non-determinism: iterate over set in any order
- Optional soft priorities on elements
- New elements can be added to set during iteration

Ordered algorithms

- Ordered set iterators: [Schwartz70]
- New elements can be added to ordered set during iteration
- Implementation: priority queue

```
Mesh m = /* read in mesh */
WorkList wl;
wl.add(m.badTriangles());
for each e in wl do{
   if (e no longer in mesh)
      continue;
   Cavity c = new Cavity();
   c.expand();
   c.retriangulate();
   m.update(c);//update mesh
   wl.add(c.badTriangles());
}
```

Parallel Implementation (Stephanie)

Unordered algorithms

Activities must be executed atomically (transactional semantics)

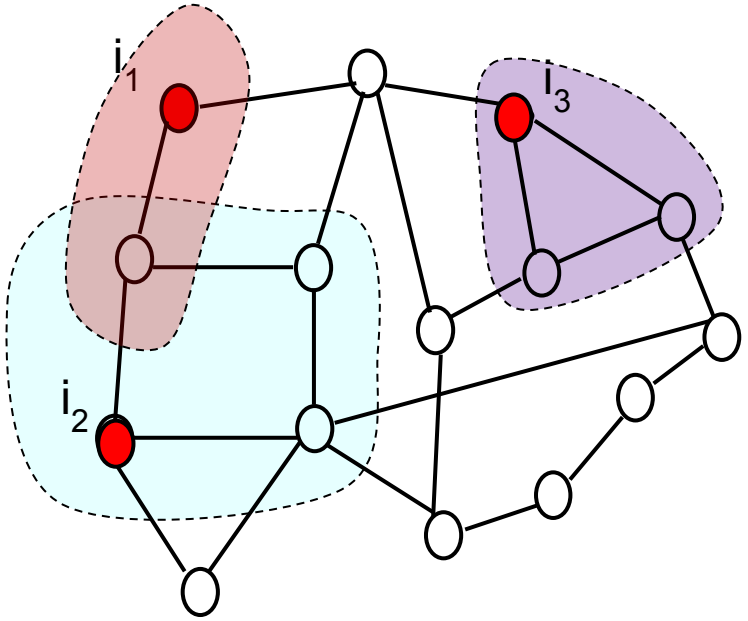
One approach: execute activities with disjoint neighborhoods in parallel

Ordered algorithms

Algorithmic order must also be respected

Key question: how do we find activities with disjoint neighborhoods?

Baseline approach: speculative execution



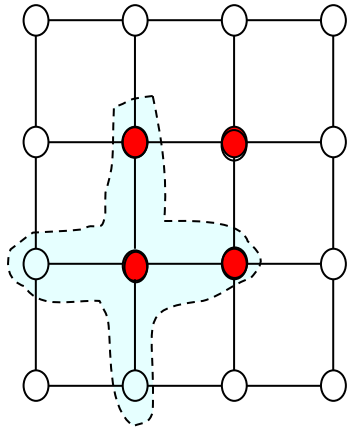
Unordered algorithms

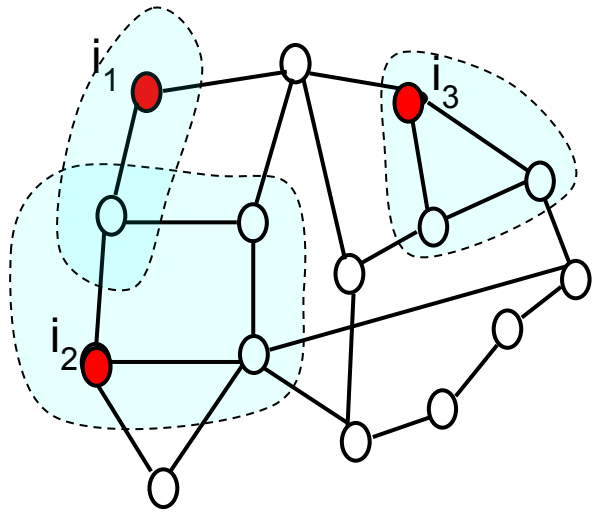
- Thread picks active node from worklist
  - Executes operator to compute neighborhood incrementally, marking state elements with unique color and buffering writes
  - If neighborhood expansion completes, perform state update (commit)
  - Otherwise, conflict so put active node back on worklist (rollback)

(6) Exploit context and structure for efficiency.

Optimizing transactional semantics: exploit context

When are active nodes and neighborhoods known?





Compile-time

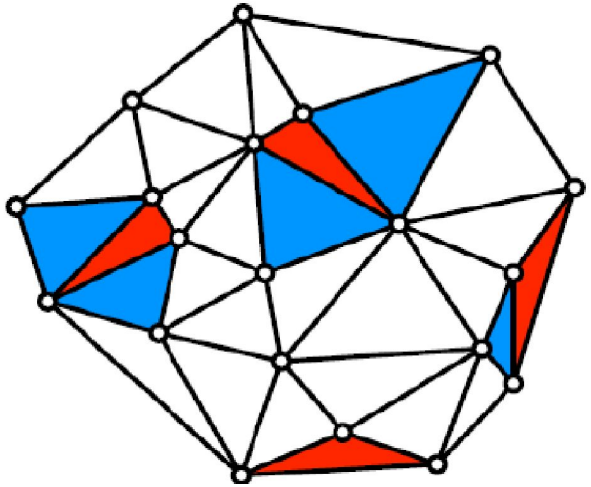
After input is given

During program execution

After program is finished

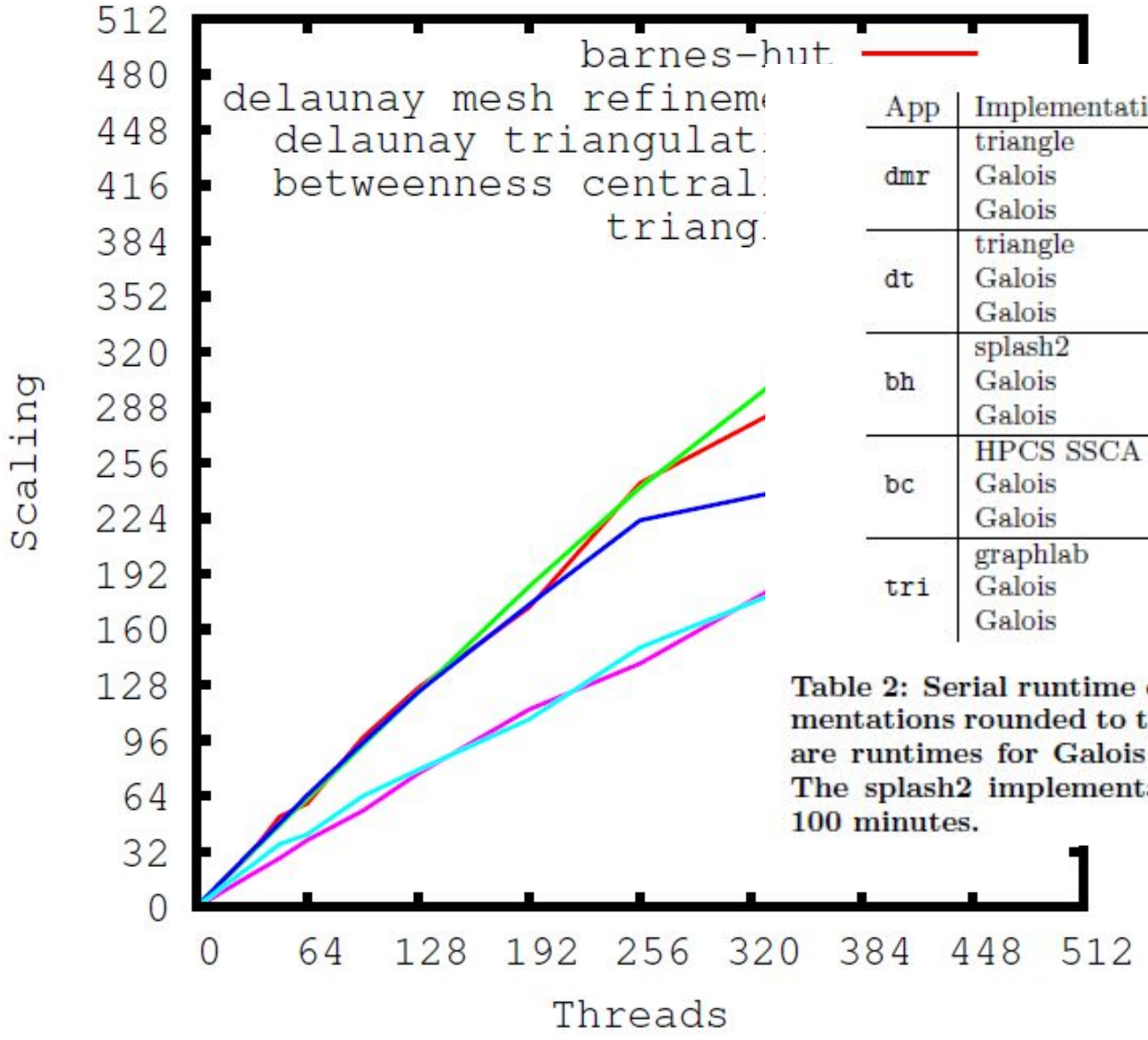
- **Static dependence graphs (stencils, dense LA)**
- **Inspector-executor (sparse LA)**
- **Interference graph (unordered algorithms)**
- **Speculative execution (ordered algorithms)**

Optimizing 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
  - Transactional memory is overkill

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.

Final remarks

Functional languages are neither necessary nor sufficient for parallel programming.

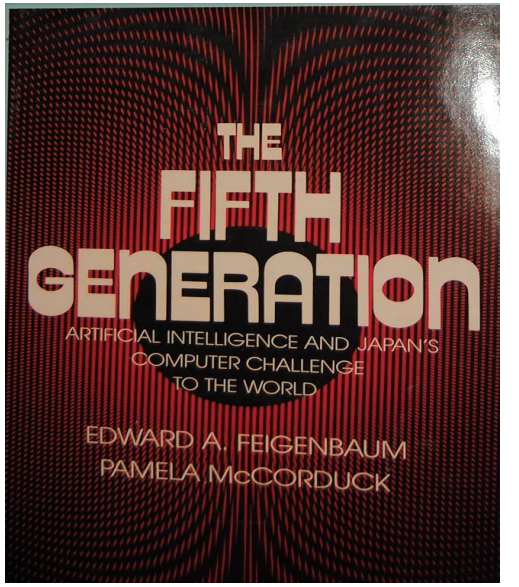
Where will we get millions of threads from?

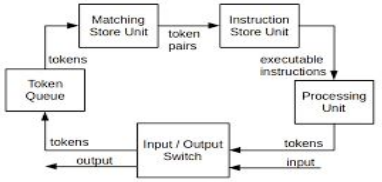
Think data-centric, not control-centric

Languages can be a problem, but they are rarely a solution (David Kuck).

Domain-specific languages: great idea

- Compilers are like skiers: downhill is lot easier than uphill





Parallel programming lessons

1. It's better to be wrong once in a while than to be right all the time.
   Speculation is essential in parallel program execution
2. (Static) Dependence graphs are not the right foundation for parallelism.
   In many programs, parallelism depends on runtime values
3. Study algorithms and data structures, not programs.
   Understand what is essential and what can be changed in programs
4. Nondeterminism is your enemy, don't-care nondeterminism is your friend.
   Convey freedom of choice to implementation.
5. Express algorithms using data-centric abstractions.
   Parallel program = Operator + Schedule + Parallel data structures
   Parallelism without explicitly parallel constructs
6. Exploit context and structure for efficiency.
   Lesson from CISC vs RISC debate
7. Functional languages are neither necessary nor sufficient for parallel programming.
   Languages can be a problem, but they are rarely a solution

Further reading

The Tao of Parallelism in Algorithms *

Keshav Pingali[1,3], Donald Nguyen[1], Milind Kulkarni[5],
Martin Burtscher[4], M. Amber Hassaan[2], Rashid Kaleem[1], Tsung-Hsien Lee[2], Andrew Lenharth[3],
Roman Manevich[3], Mario Méndez-Lojo[3], Dimitrios Prountzos[1], Xin Sui[1]

[1]Department of Computer Science, [2]Electrical and Computer Engineering and [3]Institute for Computational Engineering and Sciences
The University of Texas at Austin
[4]Department of Computer Science, Texas State University–San Marcos
[5]School of Electrical and Computer Engineering, Purdue University

Abstract

For more than thirty years, the parallel programming community has used the *dependence graph* as the main abstraction for reasoning about and exploiting parallelism in "regular" algorithms that use dense arrays, such as finite-differences and FFTs. In this paper, we argue that the dependence graph is not a suitable abstraction for algorithms in new application areas like machine learning and network analysis in which the key data structures are "irregular" data structures like graphs, trees, and sets.

To address the need for better abstractions, we introduce a data-centric formulation of algorithms called the *operator formulation* in which an algorithm is expressed in terms of its action on data structures. This formulation is the basis for a structural analysis of algorithms that we call *tao-analysis*. Tao-analysis can be viewed as an abstraction of algorithms that distills out algorithmic properties important for parallelization. It reveals that a generalized form of data-parallelism called *amorphous data-parallelism* is ubiquitous in algorithms, and that, depending on the tao-structure of the algorithm, this parallelism may be exploited by compile-time,

1. Introduction

道 n. /tau/, /dau/: **1. a.** the source and guiding principle of all reality as conceived by Taoists; **b.** process which is to be followed for a life of harmony. Origin: Chinese *dào*, literally, way. (adapted from Merriam-Webster)

Dense matrix computations arise naturally in high-performance implementations of important computational science methods like finite-differences and multigrid. Therefore, over the years, the parallel programming community has acquired a deep understanding of the patterns of parallelism and locality in these kinds of "regular" algorithms. In contrast, "irregular" data structures such as sparse graphs, trees and sets are the norm in most emerging problem domains such as the following.

- In social network analysis, the key data structures are extremely sparse graphs in which nodes represent people and edges represent relationships. Algorithms for betweenness-centrality, maxflow, etc. are used to extract network properties [10]

Thank you