Why compilers have failed and What we can do about it

Keshav Pingali

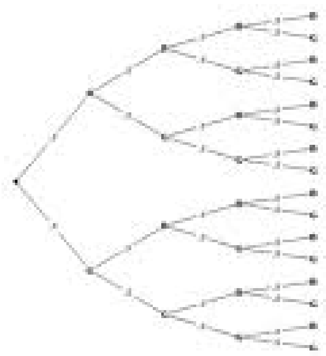
The University of Texas at Austin

Organization

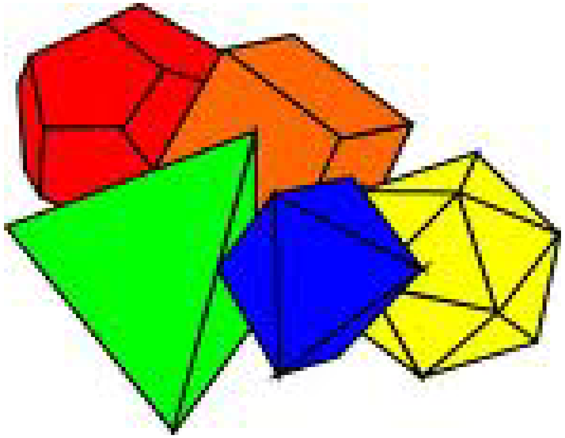
- Two successes of compilers
- Two failures of compilers
- Three lessons
- Learning from failures

Successes of past 25 years(I)

- Instruction-level parallelism (ILP)
  - Resources: processor pipeline
    - Functional units
    - Registers
  - Optimization scope:
    - Basic blocks (Hardware:IBM Stretch)
    - Instruction sequences: trace scheduling (Josh Fisher)
    - Innermost loops: software pipelining (Bob Rau)
    - Loops with conditionals (Bob Rau)
    - DAGs: super-blocks, hyper-blocks (Wen-Mei Hwu, Scott Mahlke)
  - Key ideas:
    - Speculation: it's all about probabilities
    - Dynamic branch prediction

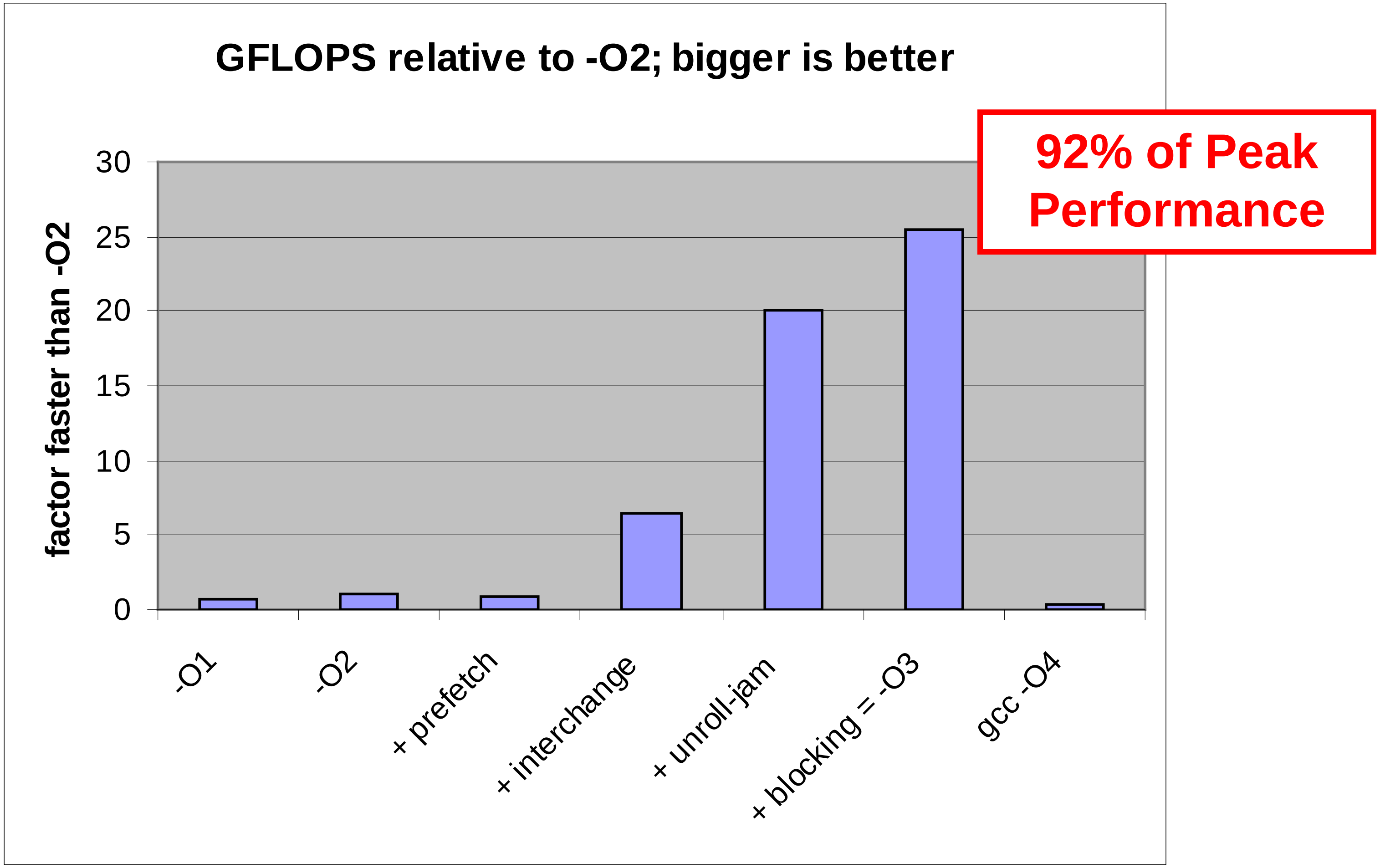


Accomplishments of past 25 years (II)



- Memory-hierarchy optimization
  - Resources:
    - Caches and registers
    - Functional units
  - Optimization scope:
    - Perfectly nested DO-loops + dense arrays
    - Imperfectly nested DO-loops + dense arrays
  - Key ideas:
    - Loop transformations:
      - UIUC (Kuck, Padua,..), Rice (Kennedy,Cooper,Hall..), IBM (Fran Allen, Sarkar,..)
    - Program abstractions:
      - polyhedral methods (French school, Sanjay, Saday, Reservoir,..)

Itanium MMM (–O3)



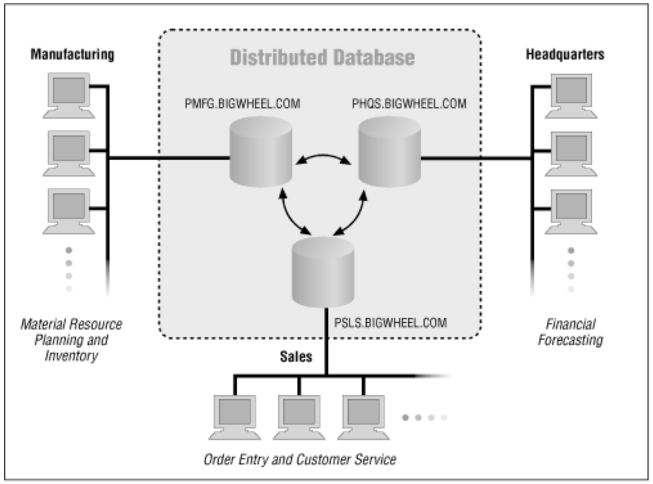
From Wei Li (Intel)

Bad news: we failed on the big ones

- Auto-parallelization
  - Some success with vectorization of dense matrix programs
- Dusty-deck rejuvenation

Other communities

- Although we have failed with parallelism, other communities have succeeded
  - Databases: (Codd)
    - SQL
  - Numerical linear algebra: (Dongarra, Demmel, Gropp,…)
    - ScaLAPACK, PetSc, etc.

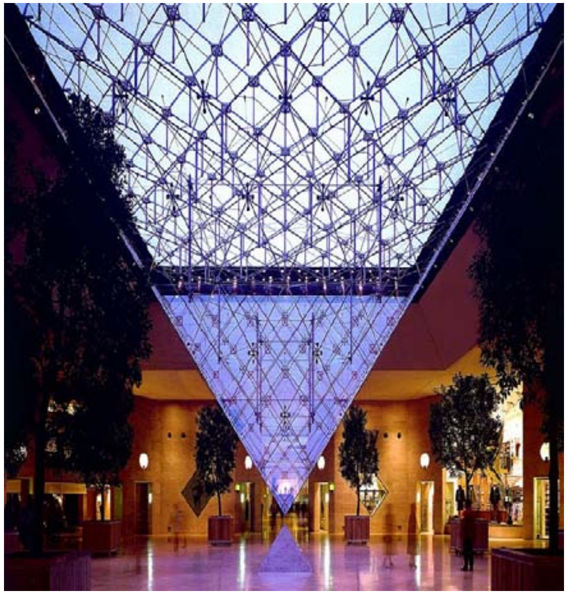


Lesson 1

- Compilers
  - Good at lowering abstraction level of program
    - conventional code generation from HLL programs
    - ILP exploitation
  - Bad at raising abstraction level
    - dusty-deck rejuvenation
    - auto-parallelization
- Lesson
  - Solution to auto-parallelization problem must not require compiler to raise abstraction level to uncover high level structure
- Wrong question:
  - Can dusty-deck program written in FORTRAN or C be parallelized?
- Right question:
  - Given the state of the art of program analysis and runtime systems, can we invent
    - sequential descriptions of algorithms + minimal amount of explicitly parallel code/annotations/directives such that
    - performance of the resulting program $\approx$ performance of explicitly parallel program for the same algorithm?

Lesson 2

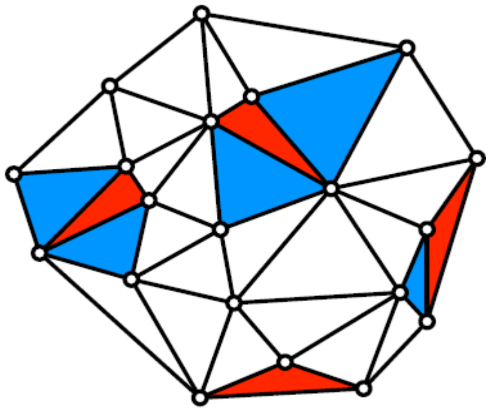
- Domains that have harnessed parallelism successfully have at least two distinct classes of programmers
  - Databases:
    - SQL programmers: Joe programmers
    - DBMS implementers: Stephanie programmers
  - Numerical linear algebra:
    - MATLAB users: Joe programmers
    - LAPACK implementers: Stephanie programmers
    - BLAS implementers: Kazushige Goto programmer



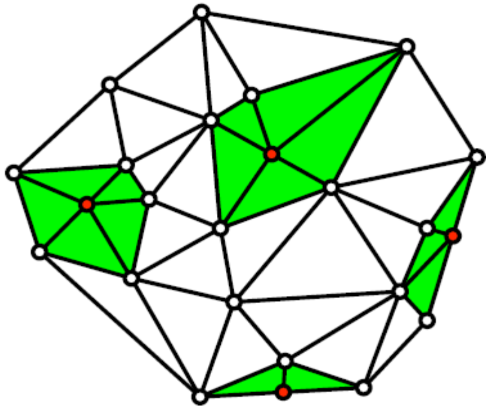
- Strategy
  - Small number of Stephanies to support large number of Joes
  - Software contract between Joes and Stephanies
- Lesson:
  - Multicore programs and programmers will not be monolithic
  - Languages and tools for Joe may be very different from those for Stephanie or Goto
  - Need to figure out levels and software contracts between levels

Lesson 3

- Static dependence graphs are not useful abstractions for many algorithms
  - In many algorithms, dependences are functions of runtime values
- For these algorithms, compile-time parallelization and scheduling is not possible
  - Much if not most of the work for parallelization must be done at runtime
    - Inspector-executor approach
    - Interference graph approach
    - Speculative or optimistic execution
- Lesson:
  - parallelization cannot mean just compile-time parallelization
  - must think in terms of binding time



Before



After

Delaunay mesh refinement

Binding time of scheduling decisions

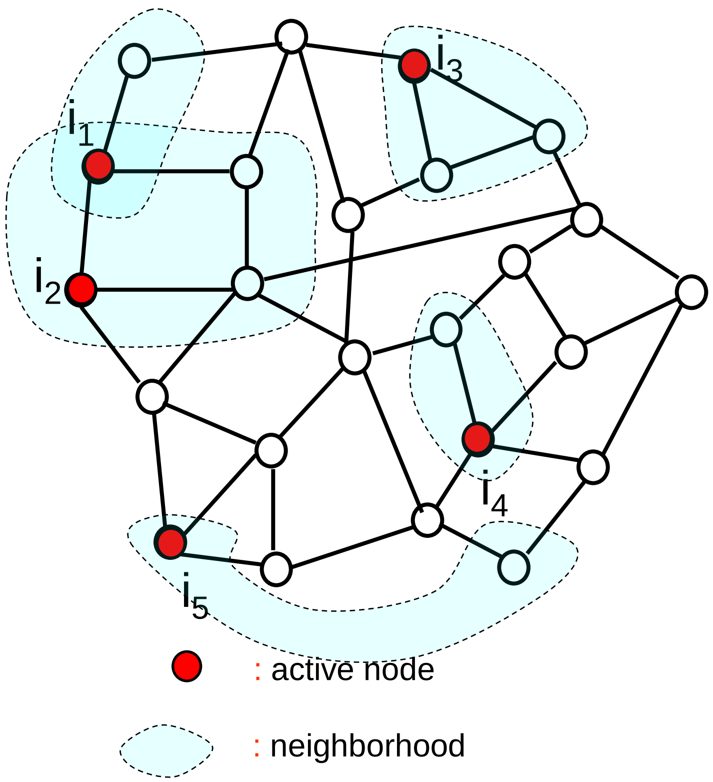
- Analogies:
  - Type checking
    - Compile-time: languages like Java
    - Runtime: languages like MATLAB and Python
  - Number of times a loop executes
    - Compile-time: “DO I = 1, 100”
    - Just-in-time: “DO I = 1, N”
    - Runtime: “while (true) do”
- Parallelization: when do we know dependences?
  - Compile-time: dense matrix codes, FFT, stencils,Barnes-Hut,..
  - Just-in-time (inspector-executor): sparse MVM, tree walks
  - Runtime: irregular codes like DMR, event-driven simulation
- Lesson:
  - parallelization requires fusion of compiler and runtime systems

Galois system

- Focus: irregular applications
  - solve the A(B(I)) = ..A((CI)) problem
- Abstract data types:
  - set, priorityQ, graph
- Parallel program = Algorithm + Parallel data structure
  - algorithm: written in C++ by Joe
  - parallel data structures: written by Stephanie
- Finding parallelism
  - speculation
  - interference graphs
- Compiler optimization to reduce parallel overheads

Galois approach

- Algorithm = repeated application of operator to graph
  - active element:
    - node or edge where computation is needed
  - neighborhood:
    - set of nodes and edges read/written to perform activity
    - distinct usually from neighbors in graph
  - ordering:
    - order in which active elements must be executed in a sequential implementation
      - any order
      - problem-dependent order
- Amorphous data-parallelism
  - parallel execution of activities, subject to neighborhood and ordering constraints



Galois programming model (PLDI 2007)

- Layered architecture
- 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) SQL and database programming

Galois parallel execution model

Parallel execution model:

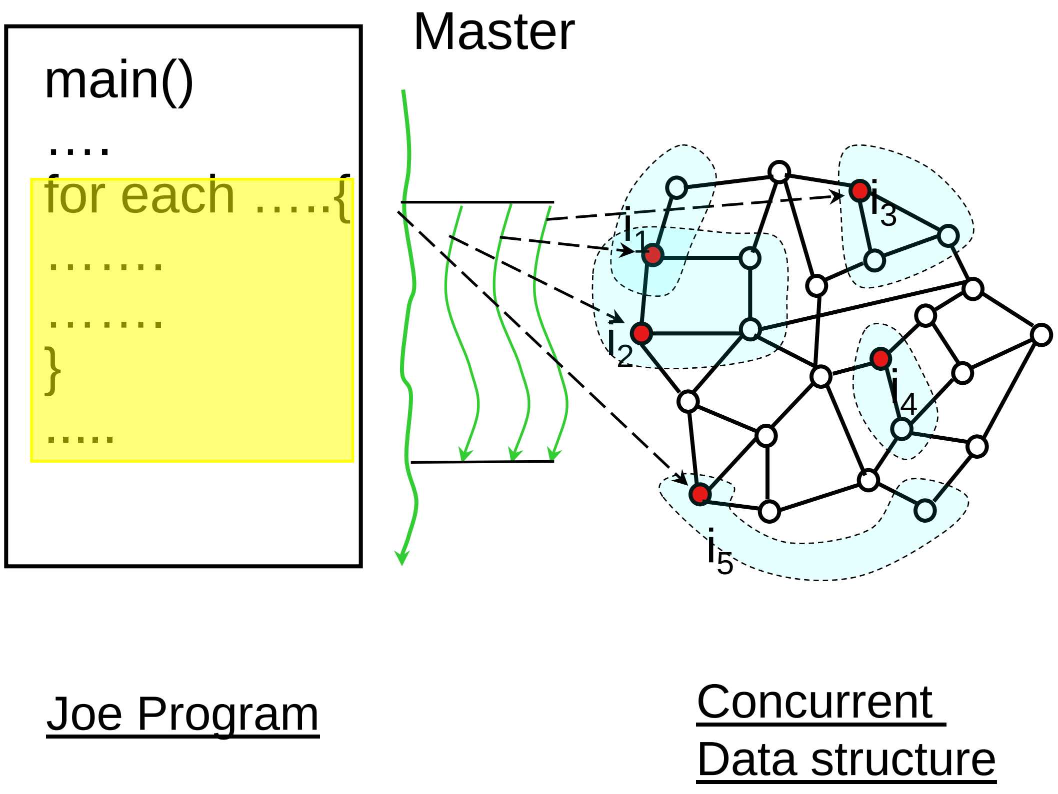
- shared-memory
- optimistic execution of Galois iterators

Implementation:

- master thread begins execution of program
- when it encounters iterator, worker threads help by executing iterations concurrently
- barrier synchronization at end of iterator

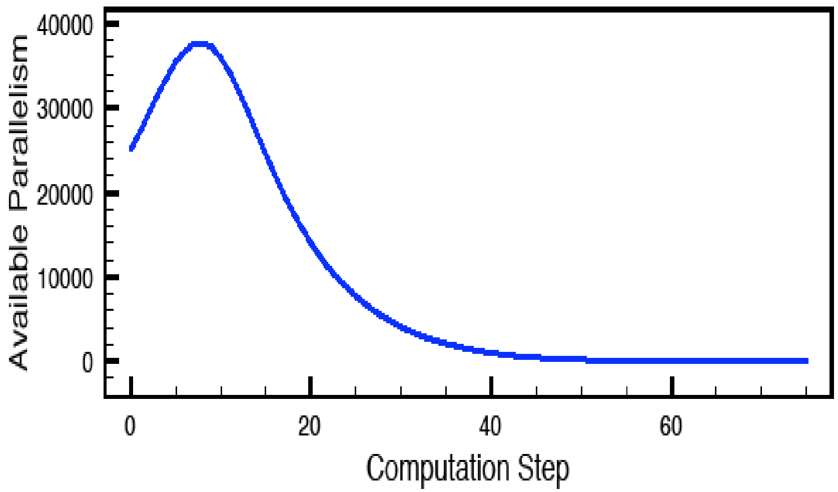
Independence of neighborhoods:

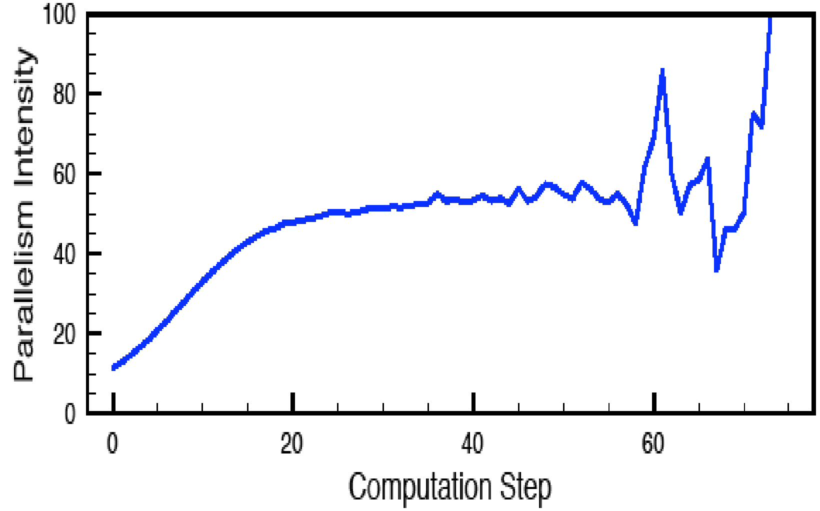
- software TLS/TM variety
- logical locks on nodes and edges



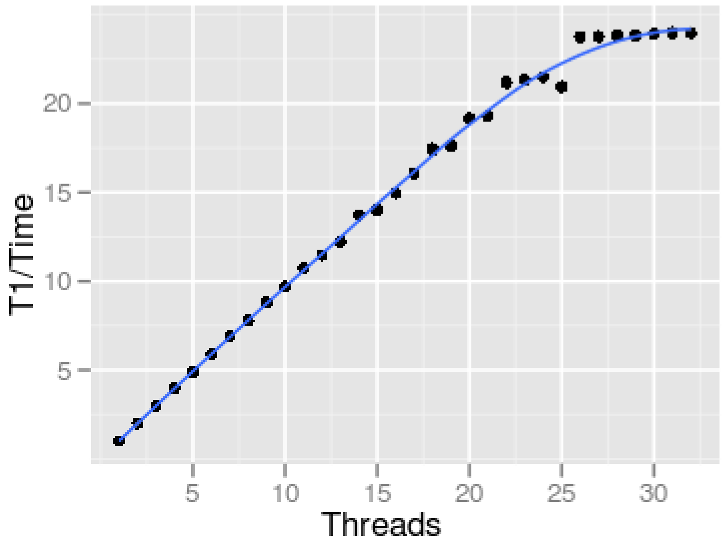
ParaMeter Parallelism Profiles

- 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?

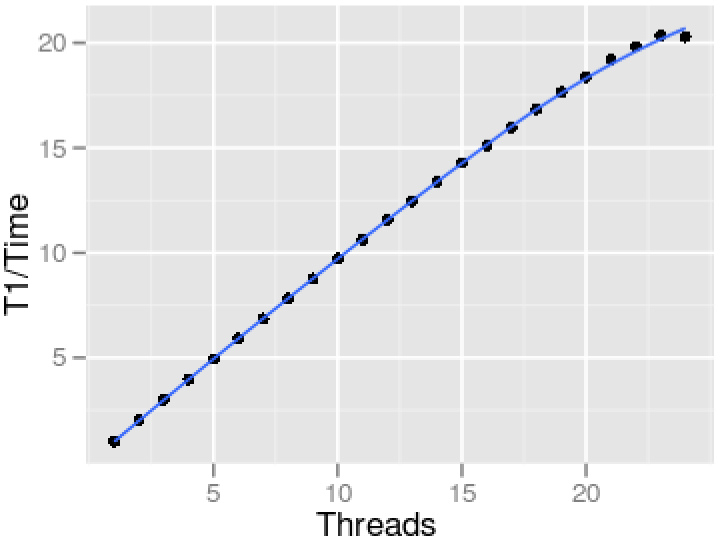




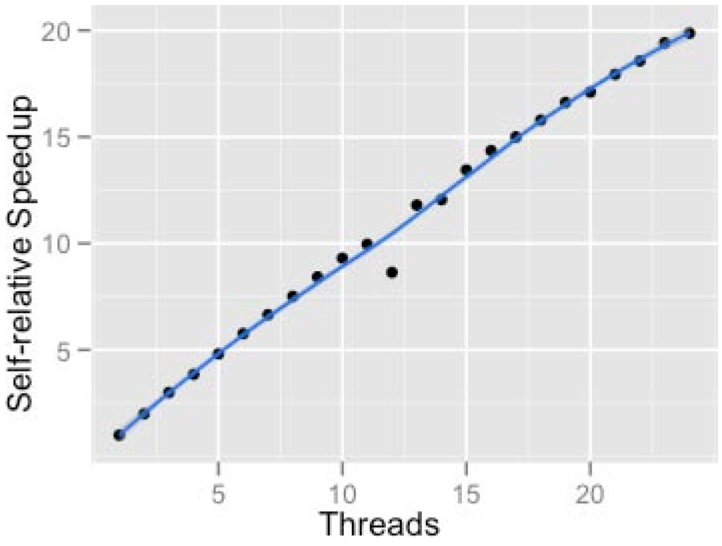
Performance of Galois system



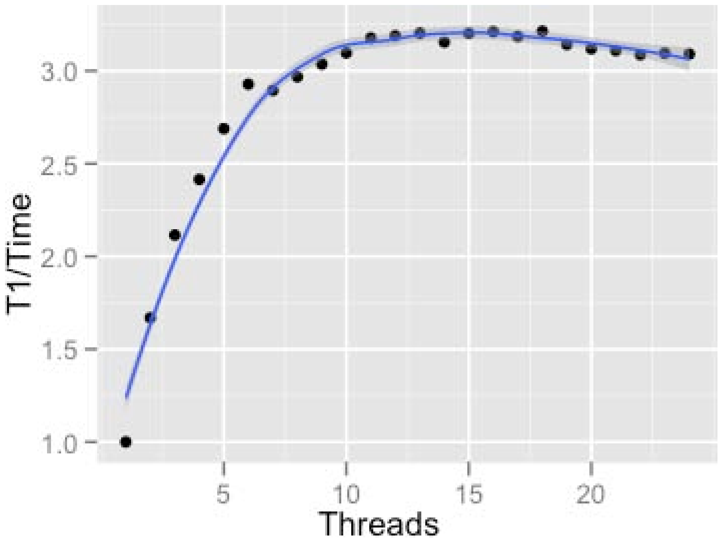
Barnes-Hut



Delaunay Mesh Refinement



Asynchronous Variational Integrator



Metis

Lesson 4

- Do not try to be all things to all application communities (yet).
- Our focus:
  - Irregular applications
    - No dense matrix applications
    - Postpone sparse matrix applications
  - Node-level parallelism
    - No distributed-memory platforms

Summary

- Compilers
  - good for optimizing programs while lowering abstraction level
  - bad at raising abstraction level
- Abstraction is your friend. Use it.
- There are several level of abstraction, each with its own programmers.
- Compile-time parallelism is a special case of parallelism.
  - static dependence graphs are not a good foundation for all parallel programming.

Patron saint of parallel programming

"Pessimism of the intellect, optimism of the will"

Antonio Gramsci (1891-1937)