Why compilers have failed and What we can do about it

Keshav Pingali

The University of Texas at Austin

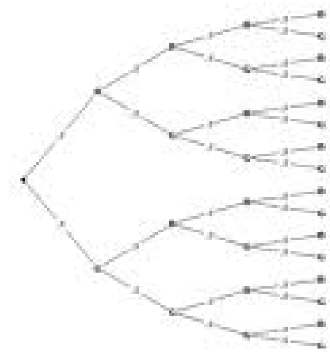
LCPC Keynote

October 7th, 2010

Organization

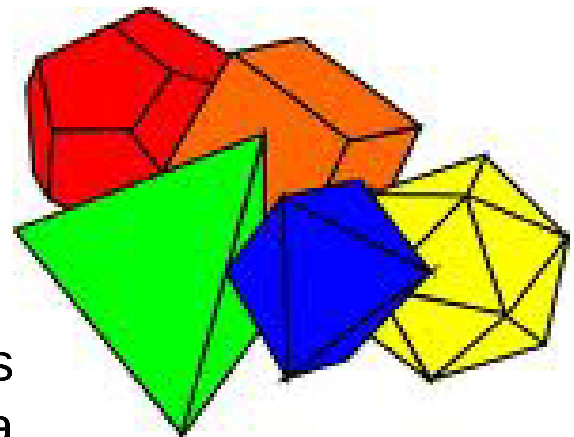
- 3 major accomplishments of compilers in the past 25 years
- 3 lessons from the failures of compilers in the past 25 years
- 2 lessons from the Galois project
- 1 challenge for the LCPC community

Accomplishments of past 25 years(I)



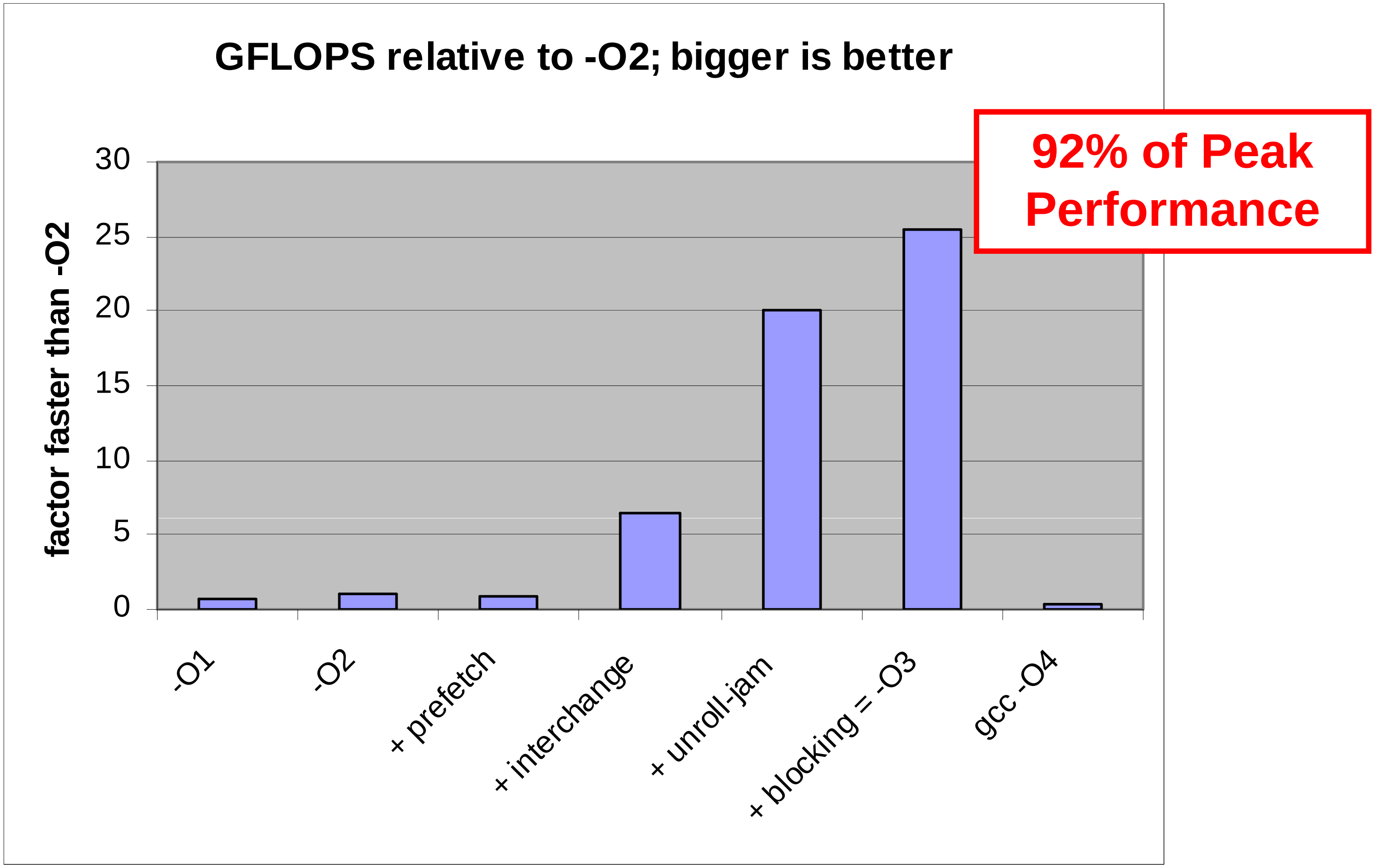
- Instruction-level parallelism (ILP)
  - Resources: processor pipeline
    - Functional units
    - Registers
  - Optimization scope:
    - Basic blocks (Hardware:IBM Stretch)
    - Instruction sequences: trace scheduling (Josh Fisher)
    - Innermost loops: software pipelining (Bob Rau)
    - Loops with conditionals (Bob Rau)
    - DAGs: super-blocks, hyper-blocks (Wen-Mei Hwu)
  - Key ideas:
    - Speculation: it's all about probabilities
    - Profile-driven optimization
    - 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,..), IBM (Fran Allen, Sarkar,..)
    - Program abstractions:
      - polyhedral methods (French school: Feautrier et al)

Itanium MMM (–O3)



From Wei Li (Intel)

Accomplishments of past 25 years (III)

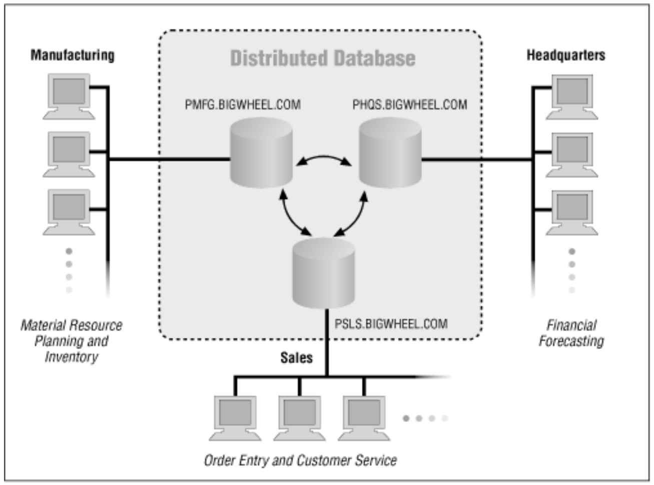
- Performance portability
  - Java: Gosling
    - byte-code interpretation +
    - just-in-time (JIT) compilation
  - FFTW,SPIRAL: Frigo,Johnson
    - codelets +
    - empirical search
  - ATLAS: Dongarra et al.
    - parameterized program +
    - empirical search

Bad news: we failed on the big one

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

Other communities

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



Organization

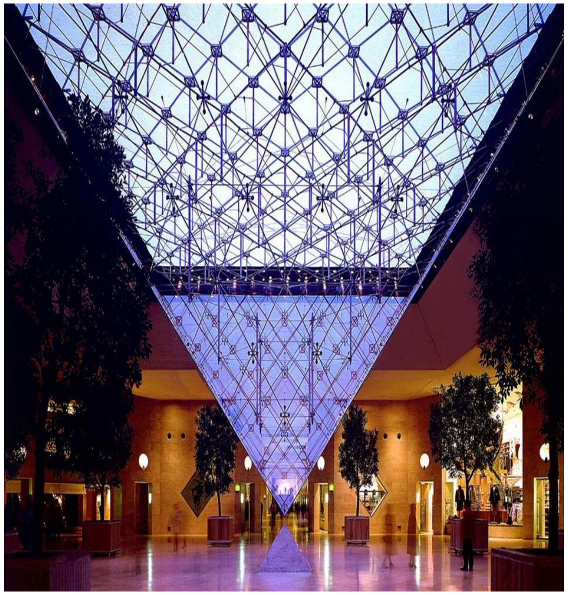
- 3 major accomplishments of compilers in the past 25 years
- 3 lessons from the failure of auto-parallelization
- 2 lessons from the Galois project
- 1 challenge for the LCPC community

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
  - Examples: databases, NA, FFTW
- 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 $\simeq$ 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

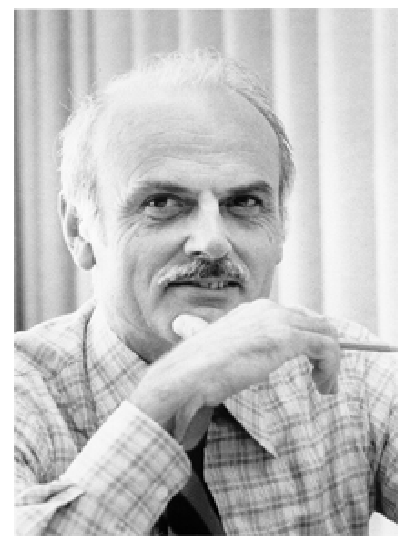
Lesson 3

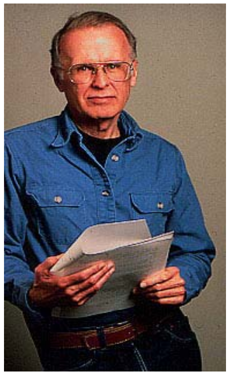
- Software contract between layers is more than an API
- Ontology or information model
  - Formal representation of entities that includes
    - properties of entities
    - relationships between entities
    - operations on entities
    - properties of operations
- Ontology examples
  - Computational algebras: (eg) Relational algebra in databases
  - BLAS interface in dense linear algebra
  - Machine language
- Motivation
  - Permits program at a given abstraction level to be optimized without knowledge of how lower layers are implemented (Kennedy: telescoping languages)
  - Permits application-specific selection of how lower layers are implemented
  - Portability: decompose program into codelets which are optimized for each architecture

Archetypal system that uses all 3 lessons

Relational databases: Codd’s 12 rules

- Rule 8: Physical data independence
  - The user should not be aware of where or upon which media data-files are stored.
- Rule 9: Logical data independence
  - User programs and the user should not be aware of any changes to the structure of the tables such as the addition of additional columns.
- Rule 11: Distribution independence
  - The RDBMS may be distributed across more than one system and across several networks, but to the end-user, the tables should appear no different than those that are local.





Contrast:general-purpose PL

- Monolithic view of programs
  - program are big, complex monoliths
  - optimized by other big, complex monoliths called compilers
  - optimization is “whole-program”
- No clear delineation of roles between
  - different classes of programmers
  - programmers and compilers
- Languages permit optimization by programmers and by compilers
  - no distinction between
    - abstraction and implementation: implicit array reshaping in FORTRAN
    - data and meta-data: pointers in C, representation exposure in OO languages

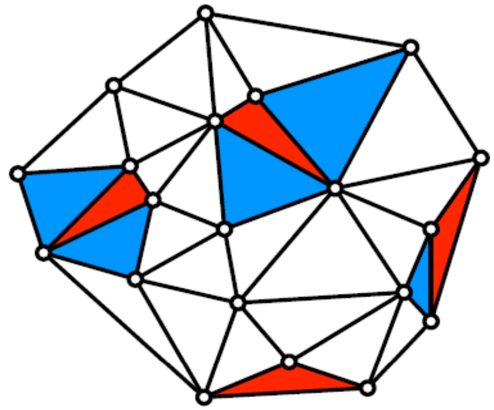
➔ Everyone and every system involved in the programming process is responsible for everything and nothing.

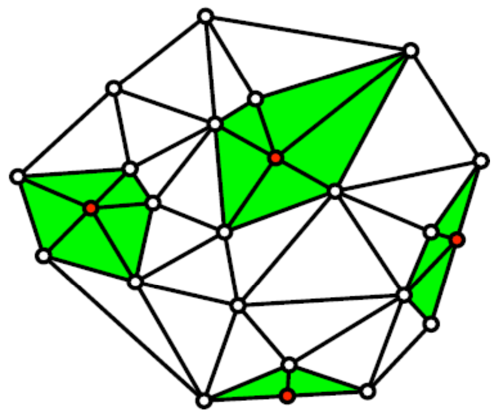
Organization

- 3 major accomplishments of compilers in the past 25 years
- 3 lessons from the failure of auto-parallelization
- 2 lessons from the Galois project
- 1 challenge for the LCPC community

Lesson 4

- 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:
  - auto-parallelization cannot mean just compile-time parallelization
  - must take a broader view of auto-parallelization in terms of binding time of scheduling decisions


Before


After

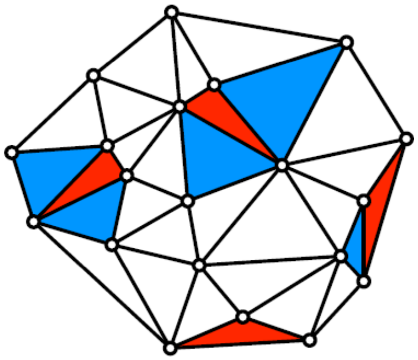
Delaunay mesh refinement

Binding time of scheduling decisions

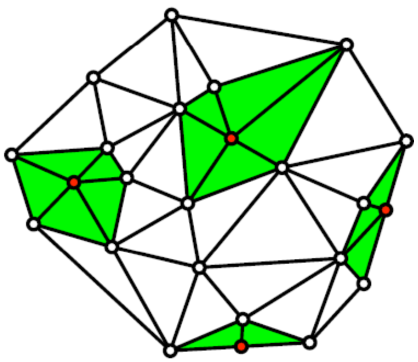
- Analogies:
  - Type checking
    - Compile-time: languages like Java
    - Runtime: languages like MATLAB and Python
  - Number of times a loop executes
    - Compile-time: “DO I = 1, 100”
    - Just-in-time: “DO I = 1, N”
    - Runtime: “while (true) do”
- Parallelization: when do we know dependences?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

Lesson 5

- Don't-care non-determinism is important for parallel performance
- Cannot be inferred by compiler analysis of programs
- Need language constructs to let programmer specify don't-care non-determinism wherever it is legal
  - Galois set iterator



Before



After

Organization

- 3 major accomplishments of compilers in the past 25 years
- 3 lessons from the failure of auto-parallelization
- 2 lessons from the Galois project
- 1 challenge for the LCPC community

Challenge for LCPC community

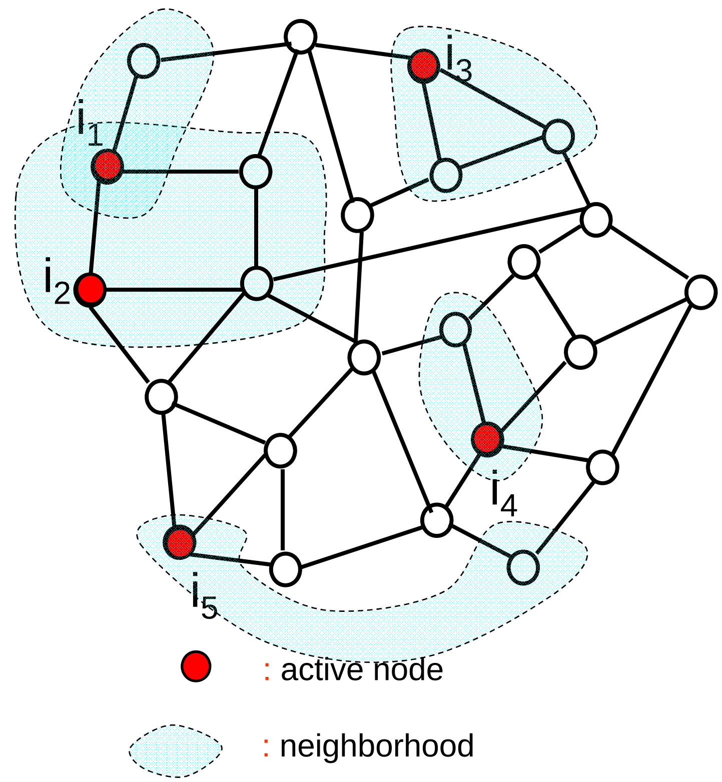
Build a general-purpose auto-parallelization system

for a 1K-core processor

First cut: Galois system

Operator formulation of algorithms

- 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:

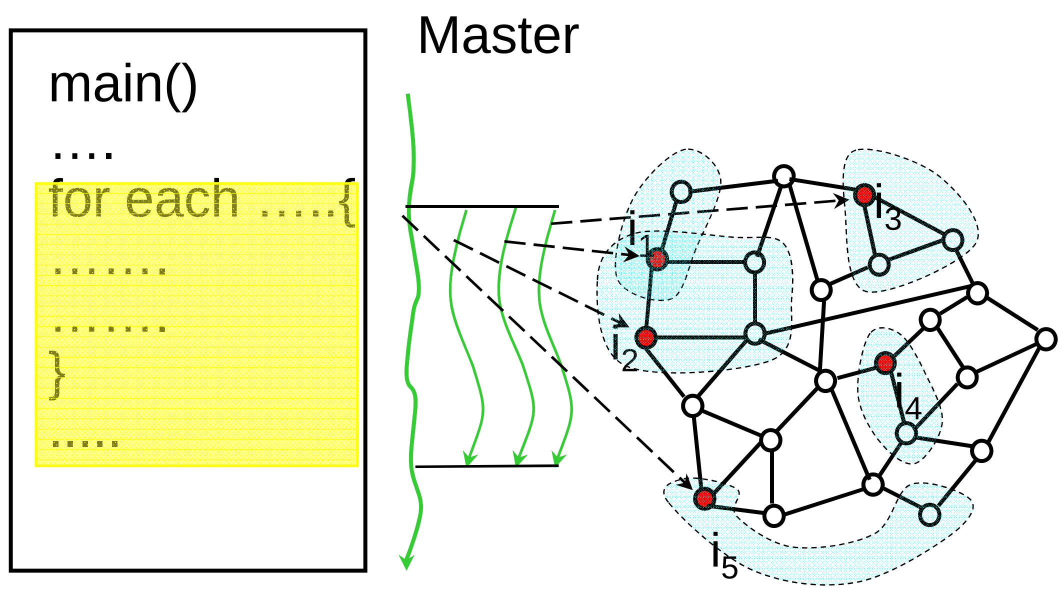
- master thread begins execution of program
- when it encounters iterator, worker threads help by executing iterations concurrently
- barrier synchronization at end of iterator

Independence of neighborhoods:

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

Ordering constraints for ordered set iterator:

- execute iterations out of order but commit in order
- cf. out-of-order CPUs

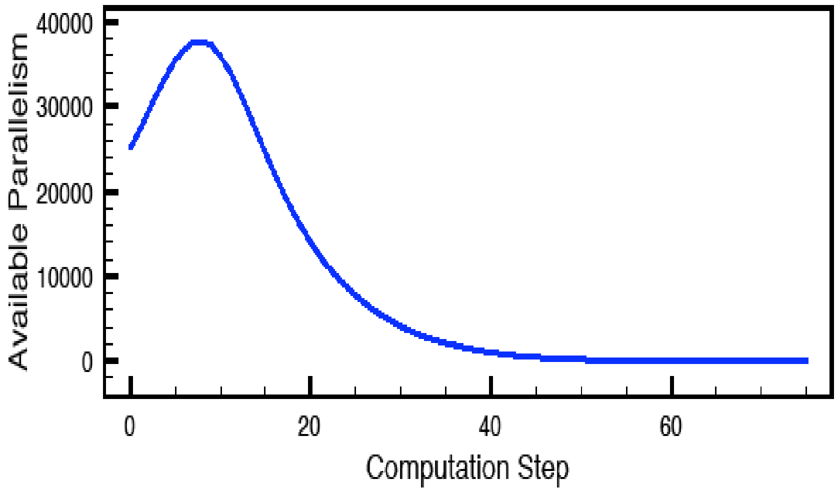


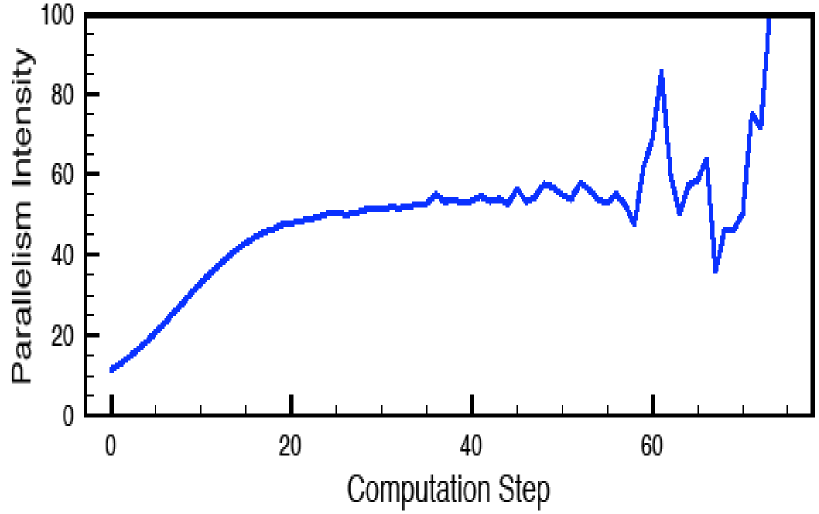
Joe Program

Concurrent Data structure

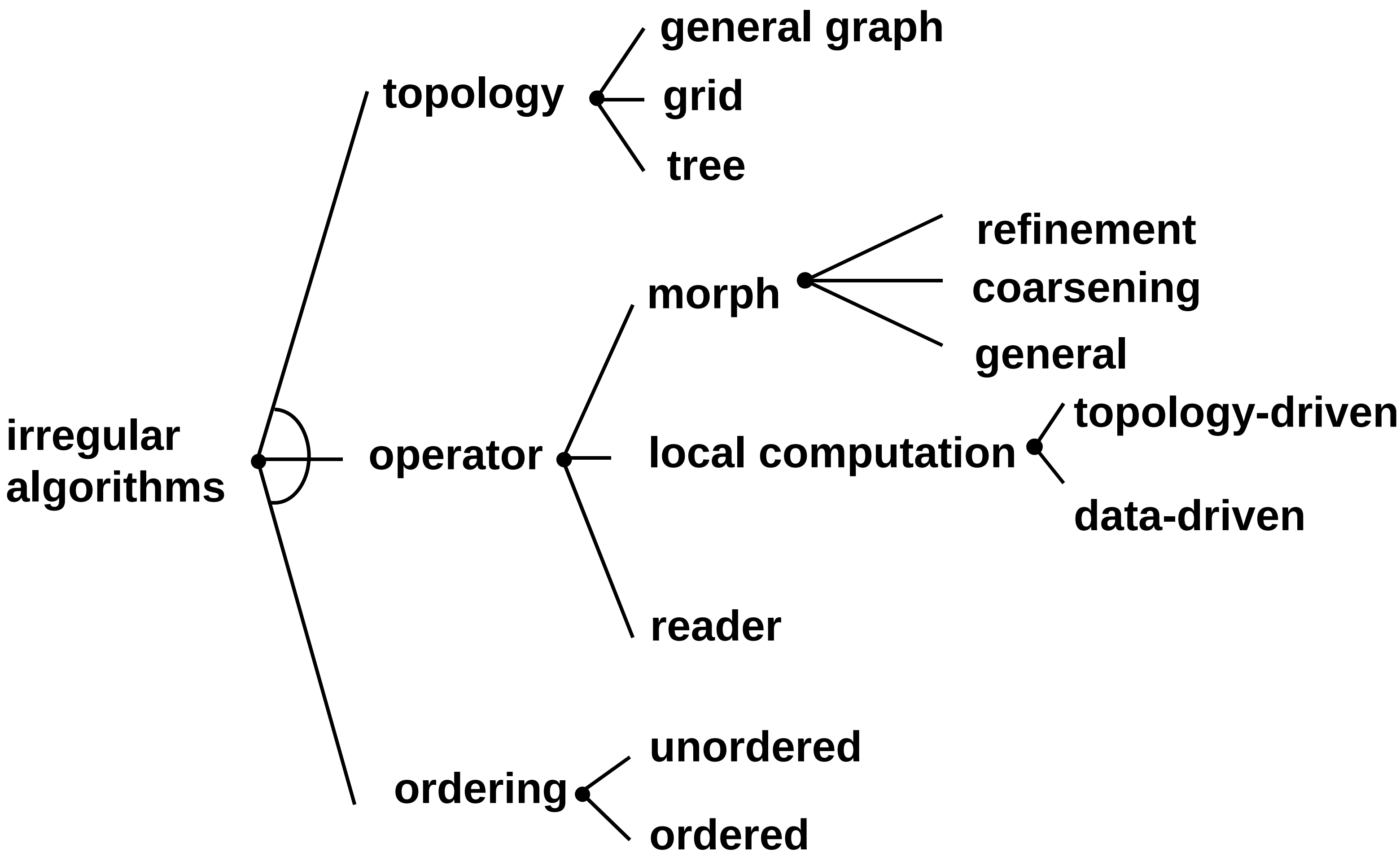
ParaMeter Parallelism Profiles (PPoPP 2009)

- 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?



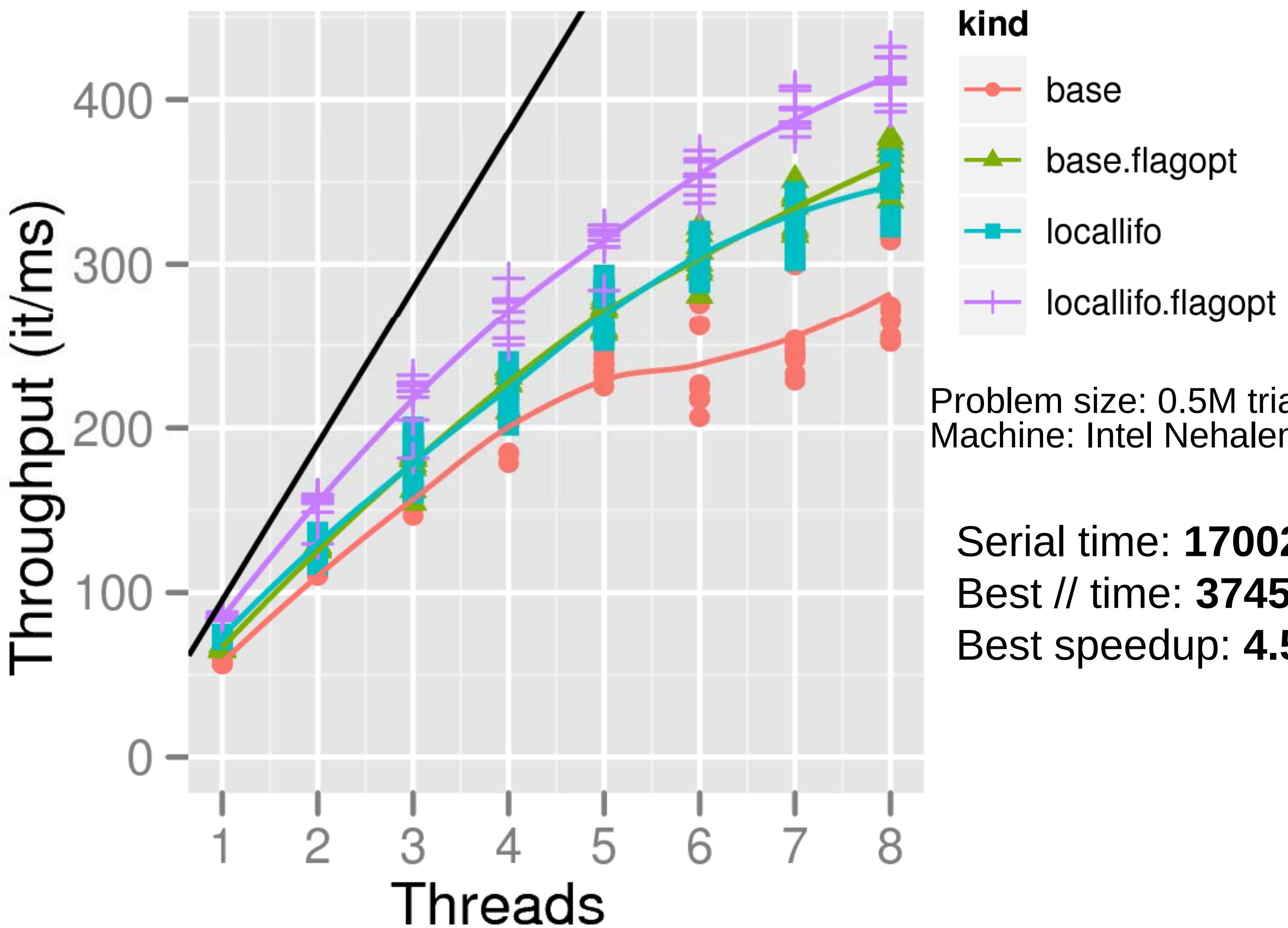


Algorithm abstractions



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

DMR Results

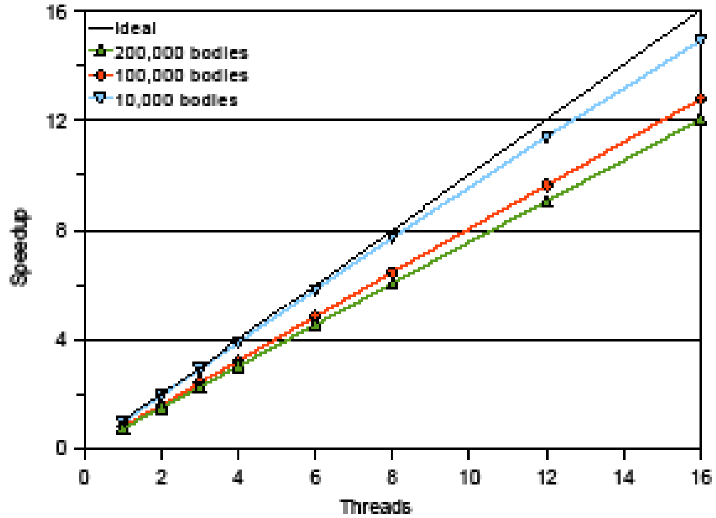


Problem size: 0.5M triangles, 0.25M bad triangles
Machine: Intel Nehalem, 2 Quad-core processors

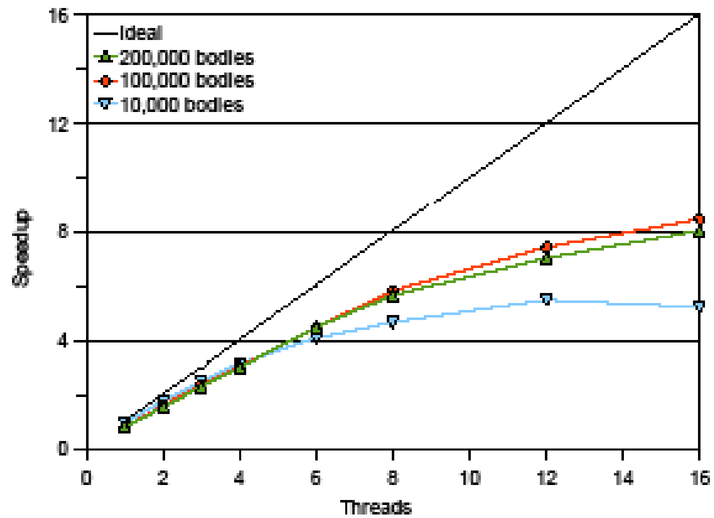
Serial time: **17002 ms**
Best // time: **3745 ms**
Best speedup: **4.5X**

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



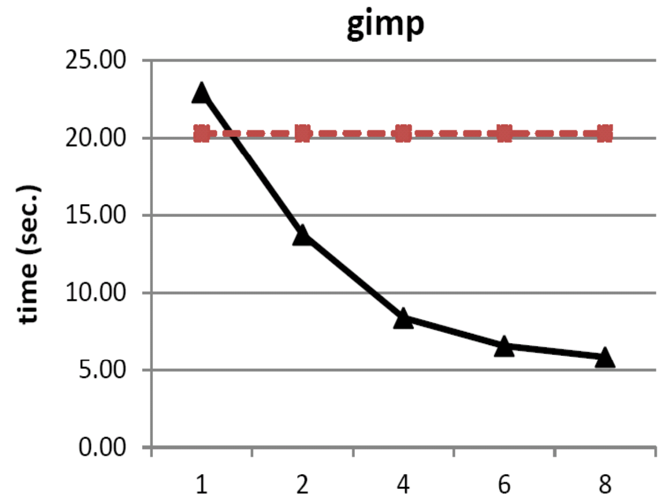
Sun Niagara-2

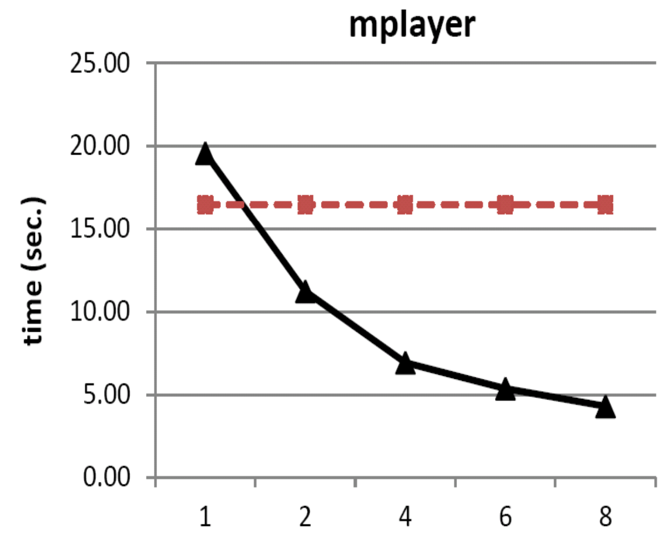


Nehalem

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)





Rising to the challenge

- Need an LCPC community-wide effort
  - too big for any one group
- Shared infrastructure
- Create a framework to identify and collect winning ideas for standardization and adoption
- Measuring progress
- Benchmarks and data-sets
- Change our research methodology
  - study algorithms and data structures, not just run benchmarks no one understands
  - reward carefully performed case studies of important kernels and applications

Patron saint of parallel programming

"Pessimism of the intellect, optimism of the will"

Antonio Gramsci (1891-1937)