The Galois Project

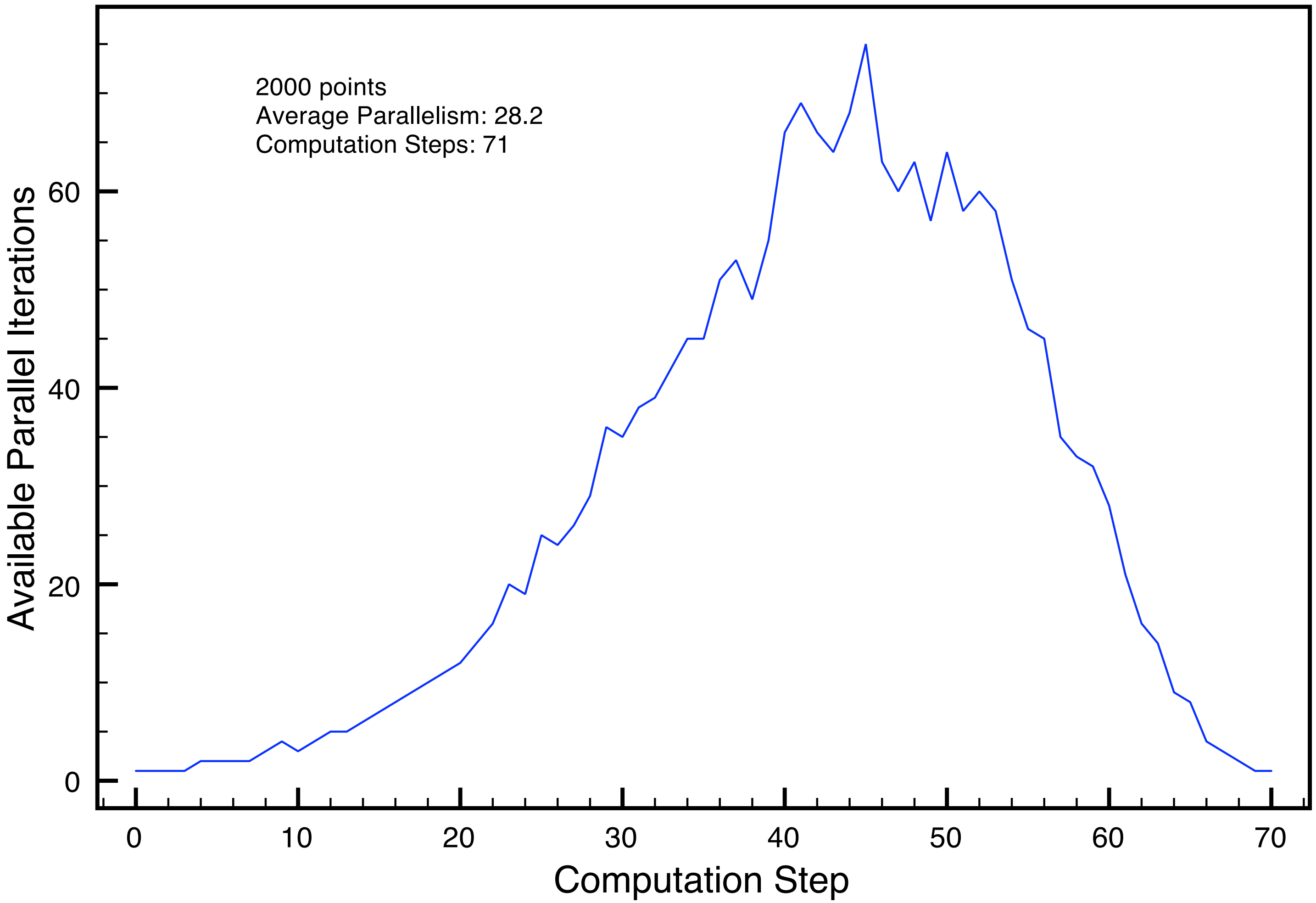
Milind Kulkarni
University of Texas, Austin

Joint work with Keshav Pingali, Martin Burtscher, Patrick Carribault, Donald Nguyen, Dimitrios Prountzos, Zifei Zhong

Proposition

- Autotuning research should broaden its scope
  - Look at irregular, pointer-based applications
    - Current focus: linear algebra, FFT, etc.
  - Look at more tuning parameters
    - Parameters related to parallel execution
  - Perform online tuning
    - Not enough information at compile-time
    - Tuning parameters can change during execution

Example: Parallelizing Delaunay Triangulation



Overview of Galois project

- Focus of Galois project:
  - parallel execution of irregular programs
    - pointer-based data structures like graphs and trees
  - raise abstraction level for “Joe programmers”
    - explicit parallelism is too difficult for most programmers
    - performance penalty for abstraction should be small

- Research approach:

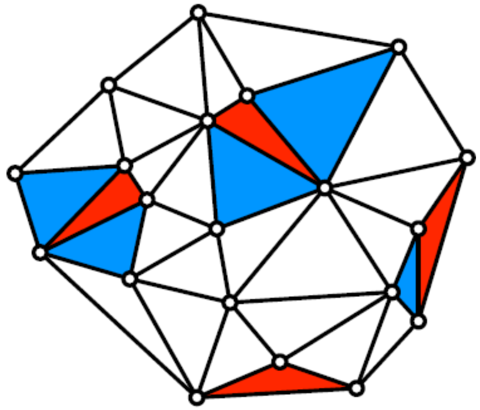
  a) study algorithms to find common patterns of parallelism and locality

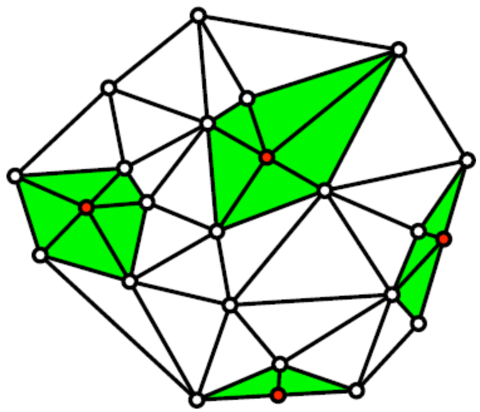
  b) design abstractions for expressing these patterns

  c) implement these abstractions efficiently

- For more information
  - papers in PLDI 2007, ASPLOS 2008, SPAA 2008
  - website: http://iss.ices.utexas.edu

Delaunay Mesh Refinement



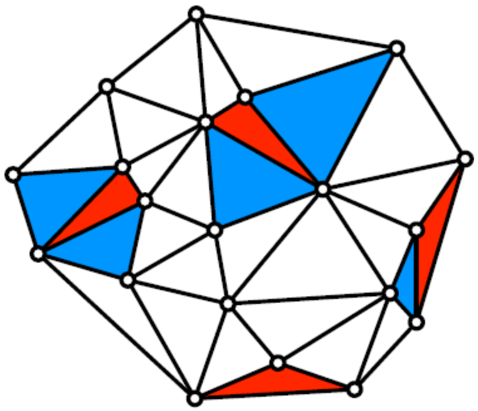


- Iterative refinement to remove badly shaped triangles:

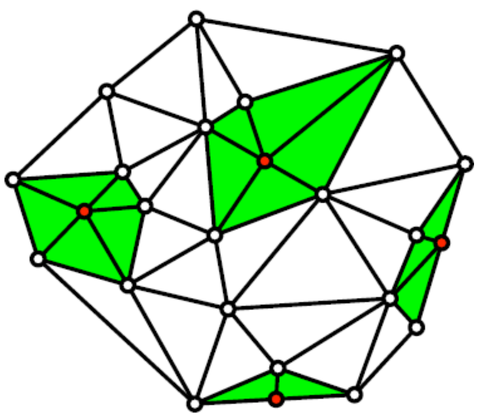
```
while there are bad triangles do {
        Pick a bad triangle;
        Find its cavity;
        Retriangulate cavity;
            // may create new bad triangles
}
```

- Order in which bad triangles should be refined:
  - final mesh depends on order in which bad triangles are processed
  - but all bad triangles will be eliminated ultimately regardless of order

Delaunay Mesh Refinement



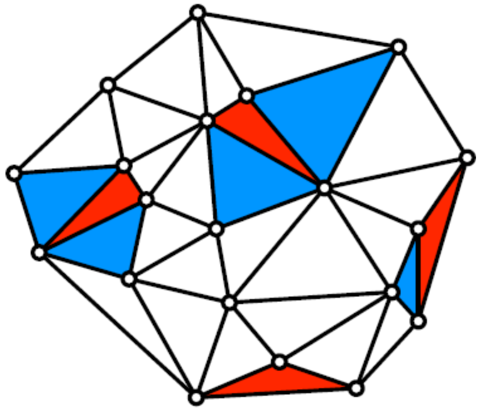
Before



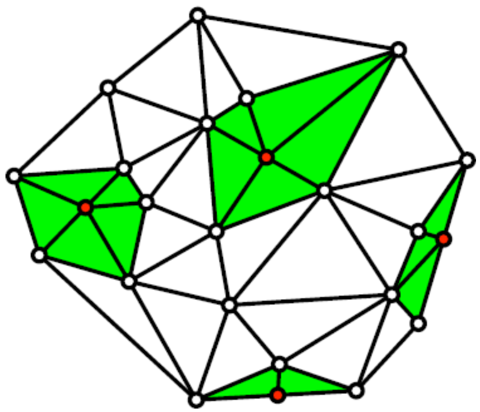
After

```
Mesh m = /* read in mesh */
WorkList wl;
wl.add(mesh.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());
}
```

Delaunay Mesh Refinement



Before



After

- Parallelism:
  - triangles with non-overlapping cavities can be processed in parallel
  - if cavities of two triangles overlap, they must be done serially
  - in practice, lots of parallelism
- Exploiting this parallelism
  - compile-time parallelization techniques like points-to and shape analysis cannot expose this parallelism (property of algorithm, not program)
  - runtime dependence checking is needed
    - Galois approach: optimistic parallelization

Take-away lessons

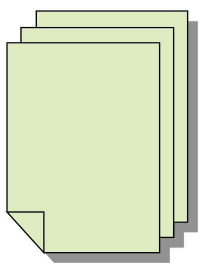
- Amorphous data-parallelism
  - iterative algorithm over ordered or unordered work-list
  - elements can be added to work-list during computation
  - complex patterns of dependences between computations on different work-list elements
  - but many of these computations can be done in parallel
- Amorphous data-parallelism is ubiquitous
  - Delaunay mesh generation: points to be inserted into mesh
  - Delaunay mesh refinement: list of bad triangles
  - Reduction-based interpreters for λ-calculus
  - Agglomerative clustering: priority queue of pairs of points
  - Boykov-Kolmogorov algorithm for image segmentation
  - Iterative dataflow analysis algorithms in compilers
  - Approximate SAT solvers: survey propagation, WalkSAT
  - ……

Take-away lessons (contd.)

- Amorphous data-parallelism is obscured within while loops, exit conditions, etc. in conventional languages
  - Need transparent syntax similar to FOR loops for regular data-parallelism
- Optimistic parallelization is necessary in general
  - Compile-time approaches using points-to analysis or shape analysis may be adequate for some cases
  - In general, runtime dependence checking is needed
  - Property of algorithms, not programs

Galois system

- Application program
  - Has well-defined sequential semantics
    - current implementation: sequential Java
  - Uses *optimistic iterators* to highlight for the runtime system opportunities for exploiting parallelism



- Class libraries
  - Like Java collections library but with additional information for concurrency control

- Runtime system
  - Managing optimistic parallelism

Optimistic set iterators

- *for each e in Set S do B(e)*
  - evaluate block B(e) for each element in set S
  - sequential semantics
    - set elements are unordered, so no a priori order on iterations
    - there may be dependences between iterations
  - set S may get new elements during execution
- *for each e in OrderedSet S do B(e)*
  - evaluate block B(e) for each element in set S
  - sequential semantics
    - perform iterations in order specified by OrderedSet
    - there may be dependences between iterations
  - set S may get new elements during execution

Galois version of mesh refinement

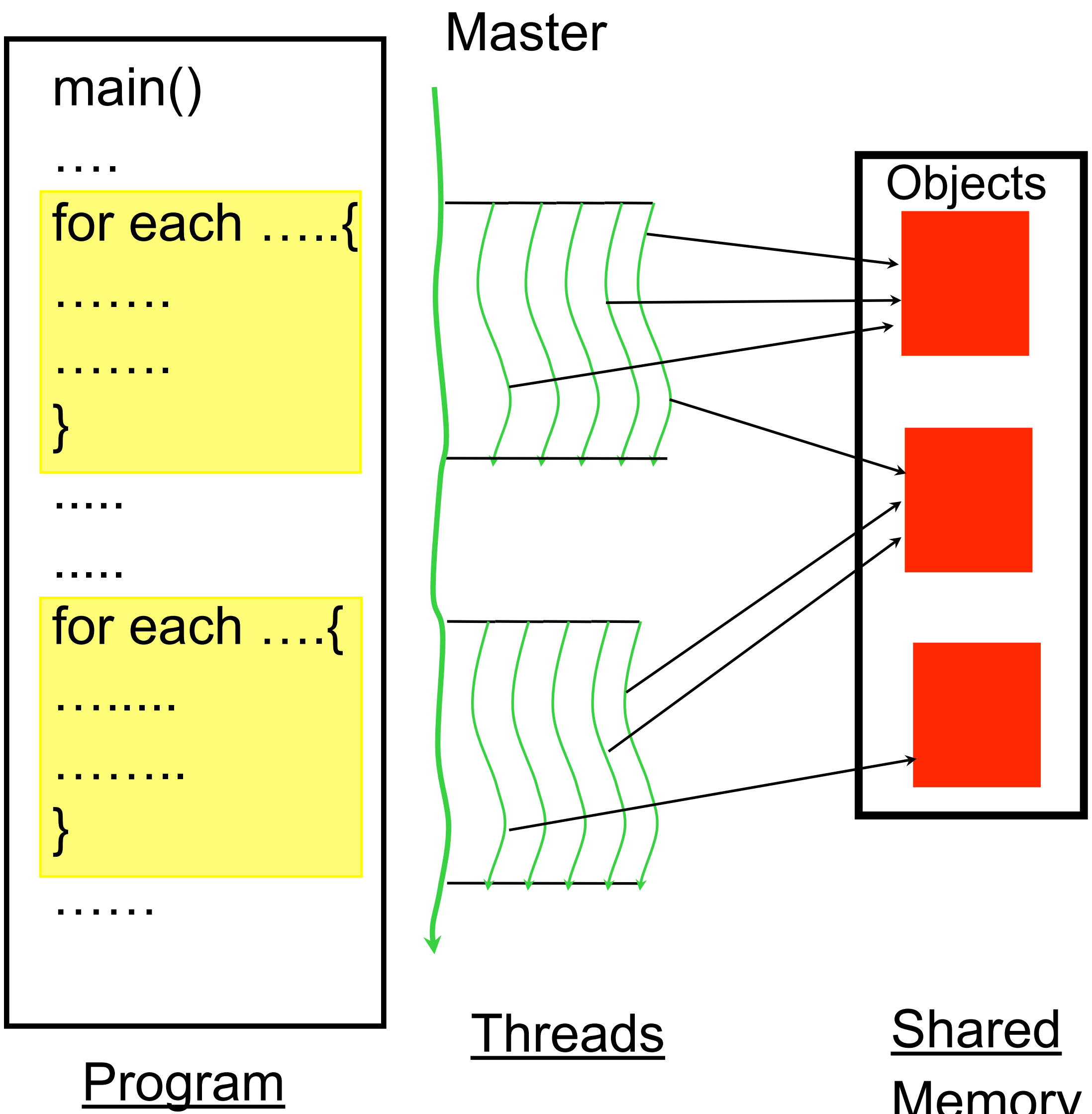
```
Mesh m = /* read in mesh */
Set wl;
wl.add(mesh.badTriangles()); // initialize the Set wl

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

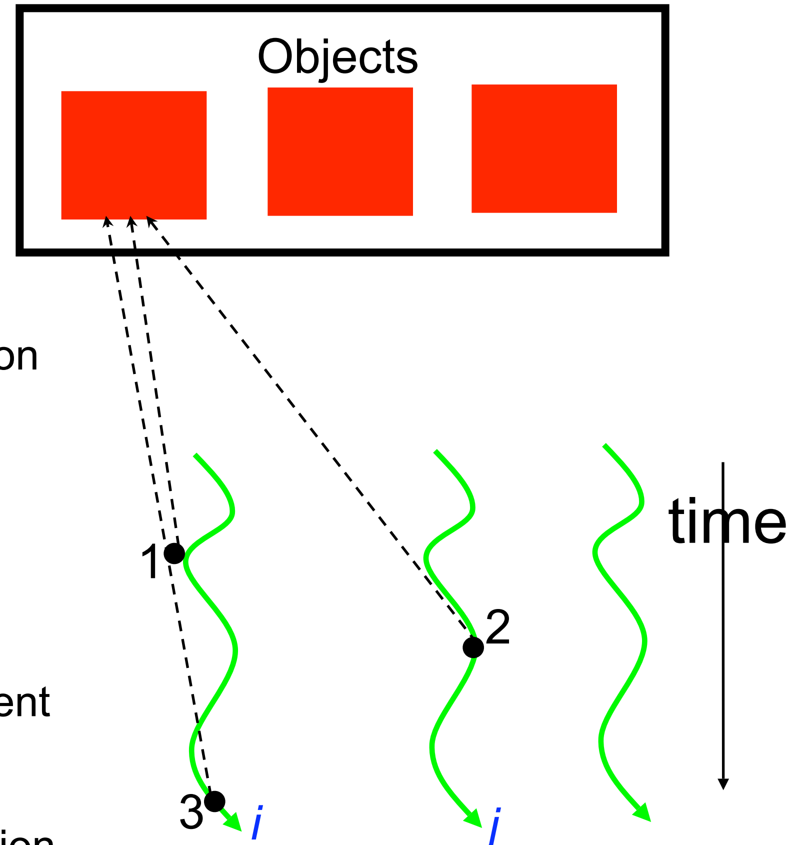
- Scheduling policy for iterator:
    - controlled by implementation of Set class
    - good choice for temporal locality: stack

Parallel execution model

- Object-based shared-memory model
- Master thread and some number of worker threads
  - master thread begins execution of program and executes code between iterators
  - when it encounters iterator, worker threads help by executing iterations concurrently with master
  - threads synchronize by barrier synchronization at end of iterator
- Threads invoke methods to access internal state of objects
  - how do we ensure sequential semantics of program are respected?



Baseline solution: PLDI 2007

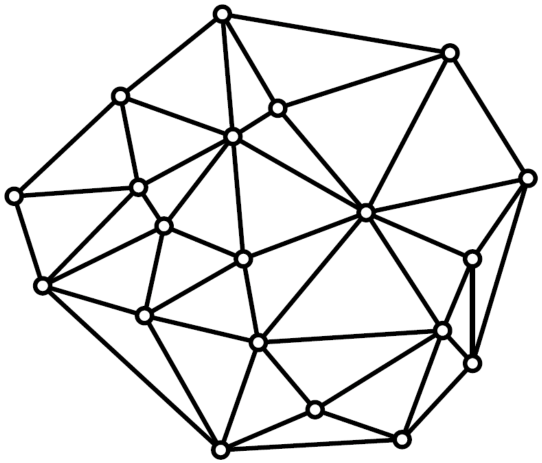


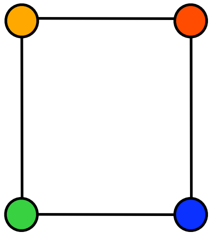
- Iteration must lock object to invoke method
- Two types of objects:
  - catch and keep policy
    - lock is held even after method invocation completes
    - locks released at end of iteration
    - this is often inefficient!
  - catch and release policy
    - like Java locking policy
    - permits method invocations from different concurrent iterations to be interleaved, provided it is safe
    - safety: requires commutativity information from class implementer
    - crucial for collections and accumulators

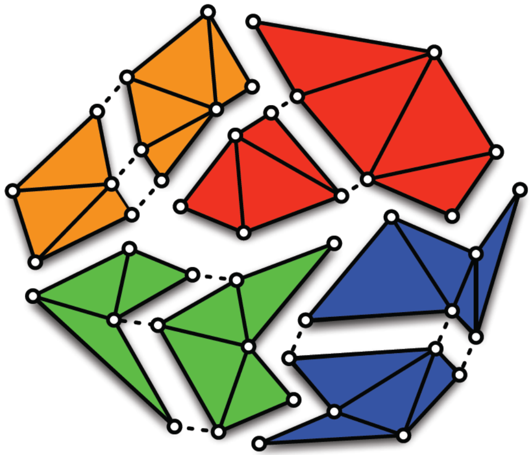
Scheduling iterators (SPAA 2008)

- Control scheduling by changing implementation of work-set class
  - stack/queue/etc.
- Can have a profound effect on abort rates and locality
- Example: Delaunay mesh refinement
  - input mesh from Shewchuck's Triangle
  - 10,156 triangles of which 4,837 were bad
  - sequential code, work-set is stack:
    - 21,918 completed iterations+0 aborted
  - 4-processor, with different work-set implementations:
    - stack: 21,736 iterations completed+28,290 aborted
    - array+random choice: 21,908 iterations completed+49 aborted
- Developed framework that generalizes Open-MP style schedules

Data Partitioning (ASPLOS 2008)



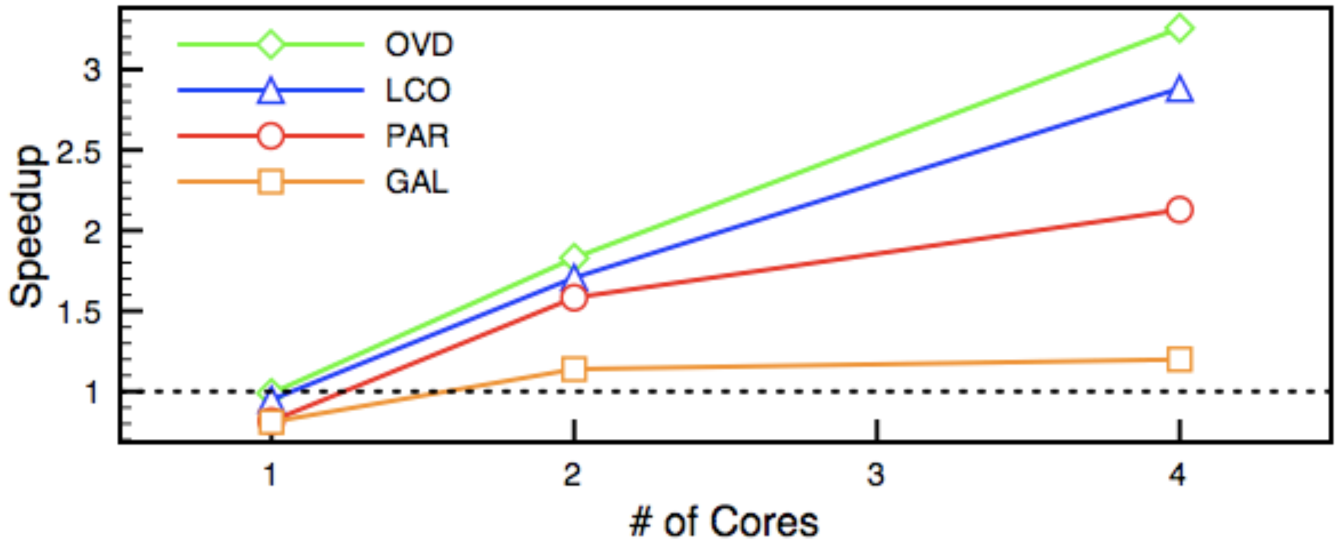




- Partition the graph between cores
- Data-centric assignment of work:
  - core gets bad triangles from its own partitions
  - improves locality
  - can dramatically reduce conflicts
- Lock coarsening:
  - associate locks with partitions, lock partitions to enforce correctness
- Over-decomposition
  - improves core utilization

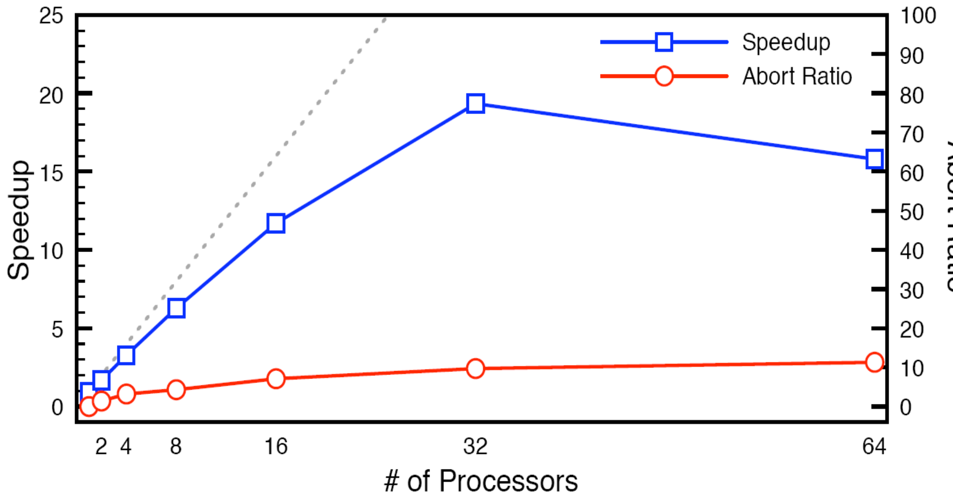
Small-scale multiprocessor results

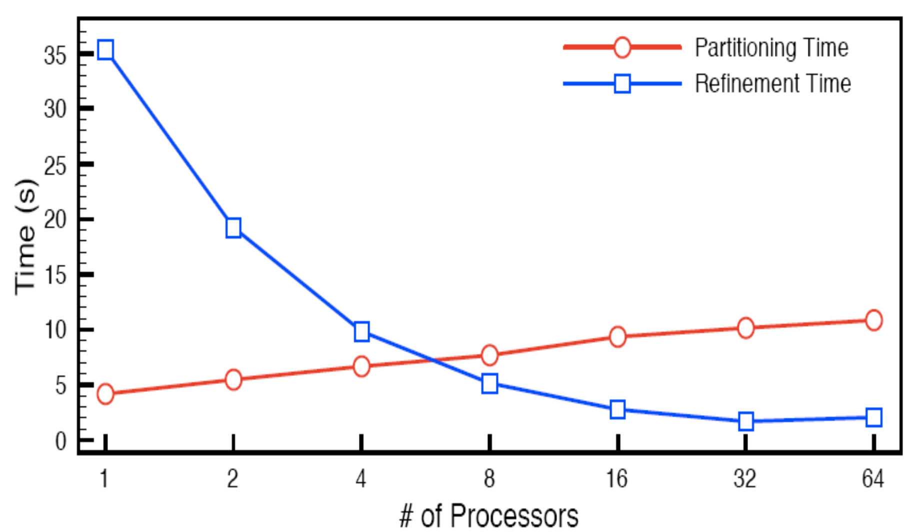
- 2x2 Xeon @ 3GHz
- Versions:
  - GAL: using stack as worklist
  - PAR: partitioned mesh + data-centric work assignment
  - LCO: locks on partitions
  - OVD: over-decomposed version (factor of 4)



Large-scale multiprocessor results

- Maverick@TACC
  - 128-core Sun Fire E25K 1 GHz
  - 64 dual-core processors
  - Sun Solaris
- First “out-of-the-box” results
- Speed-up of 20 on 32 cores for refinement
  - New results in LCPC’08
- Mesh partitioning is still sequential
  - time for mesh partitioning starts to dominate after 8 processors (32 partitions)
- Need parallel mesh partitioning





Related work

- Transactions
  - programming model is explicitly parallel
  - assumes someone else is responsible for parallelism, locality, load-balancing, and scheduling, and focuses only on synchronization
  - Galois: main concerns are parallelism, locality, load-balancing, and scheduling
- Thread level speculation
  - not clear where to speculate in C programs
    - wastes power in useless speculation
  - many schemes require extensive hardware support
  - unable to exploit commutativity at abstract data type level
  - no analogs of data partitioning or scheduling
  - overall results are disappointing

Opportunities for Auto-tuning

- On-line feedback from run-time system
  - Dynamically change amount of parallelism
    - Perhaps based on mis-speculation statistics
  - Dynamically change overdecomposition level
    - Use finer-grained partitions if mis-speculation too high
- Schedule tuning
  - Choosing which schedule to run
    - Based on properties of input data
  - Tuning particular schedule
    - Which cores should do which work

Summary

- Irregular applications have amorphous data-parallelism
  - Work-list based iterative algorithms over unordered and ordered sets
- Amorphous data-parallelism may be inherently data-dependent
  - Pointer/shape analysis cannot work for these apps
- Optimistic parallelization is essential for such apps
  - Analysis might be useful to optimize parallel program execution
- Exploiting abstractions and high-level semantics is critical
  - Galois knows about sets, ordered sets, accumulators…
- Galois approach provides unified view of data-parallelism in regular and irregular programs
  - Baseline is optimistic parallelism
  - Use compiler analysis to make decisions at compile-time whenever possible
  - Autotuning can “fill in the gaps”