Data-parallel Abstractions for Irregular Applications

Keshav Pingali
University of Texas, Austin

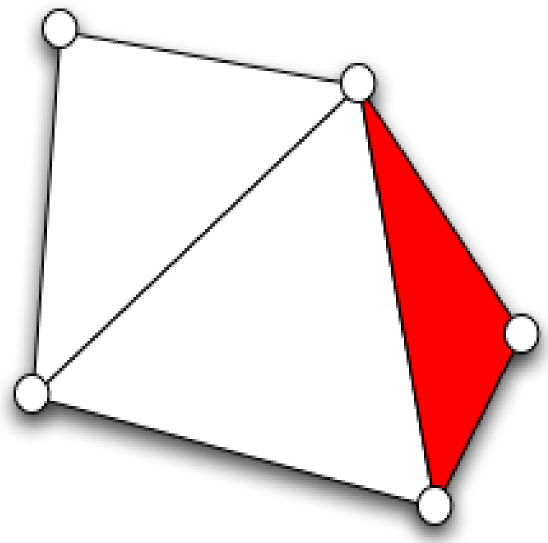
Joint work with Milind Kulkarni (UT Austin), and
Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, Paul Chew (Cornell)

Summary

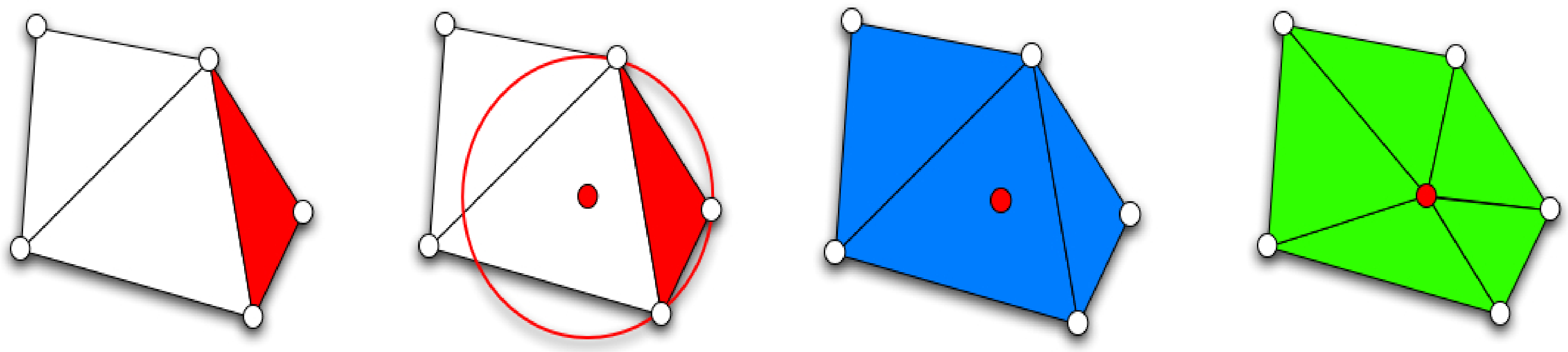
- Characteristics of irregular applications:
  - data structures like stacks/queues/trees/graphs
  - few if any dense arrays
- Optimistic (speculative) parallelization is essential
  - pointer/shape analysis cannot work
- Current thread-level speculation (TLS) implementations will not work
  - crucial to exploit abstractions provided by object-oriented languages
  - in particular, distinction between abstract data type and its implementation type
- Concurrency can be packaged within natural syntactic constructs
- Benchmark *programs* are useless
  - Wirth: Program = Algorithm + Data structure

Delaunay Mesh Refinement

- Delaunay meshes (2-D)
  - Triangulation of a surface, given vertices
  - Delaunay property: circumcircle of any triangle does not contain another point in the mesh
- In practice, want all triangles in mesh to meet certain quality constraints
  - (e.g.) no angle > 120°
- Mesh refinement:
  - fix bad triangles through iterative refinement



Refinement Algorithm

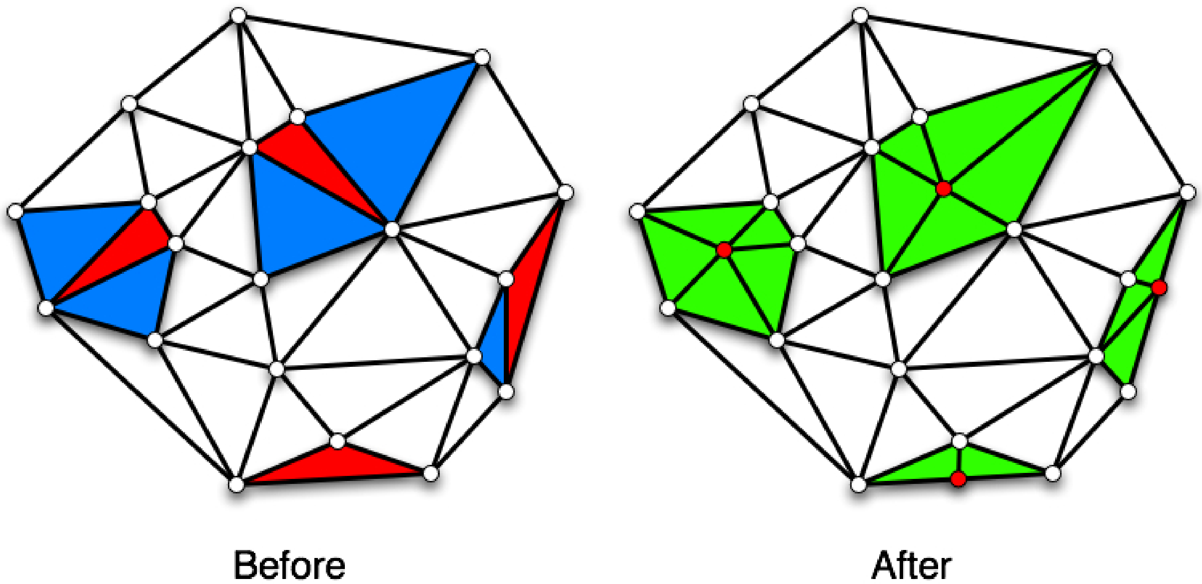


while there are bad triangles
  { pick a bad triangle
    add new vertex at center of circumcircle
    gather all triangles that no longer satisfy Delaunay property into cavity
    re-triangulate affected region, including new point
      // some new triangles may be bad themselves
  }

Sequential Algorithm

```
Mesh m = /* read in mesh */
WorkList wl;
wl.add(mesh.badTriangles()); // non-deterministic order

while (true) {
        if ( wl.empty() ) break;

        Element e = wl.get();//non-deterministic choice

         if (e no longer in mesh) continue;

        Cavity c = new Cavity(e);      //determine new cavity
        c.expand();                    //determine affected triangles
        c.retriangulate();             //re-triangulate region

        m.update(c);                   //update mesh

        wl.add(c.badTriangles());      //add new bad triangles to queue in some order
}
```

Parallelization Opportunities



- Unit of work: fixing a bad triangle
- Bad triangles ***with non-overlapping cavities*** can be processed in parallel.
- Cannot tell if cavities of two bad triangles will overlap without actually building cavities → must detect conflicts dynamically

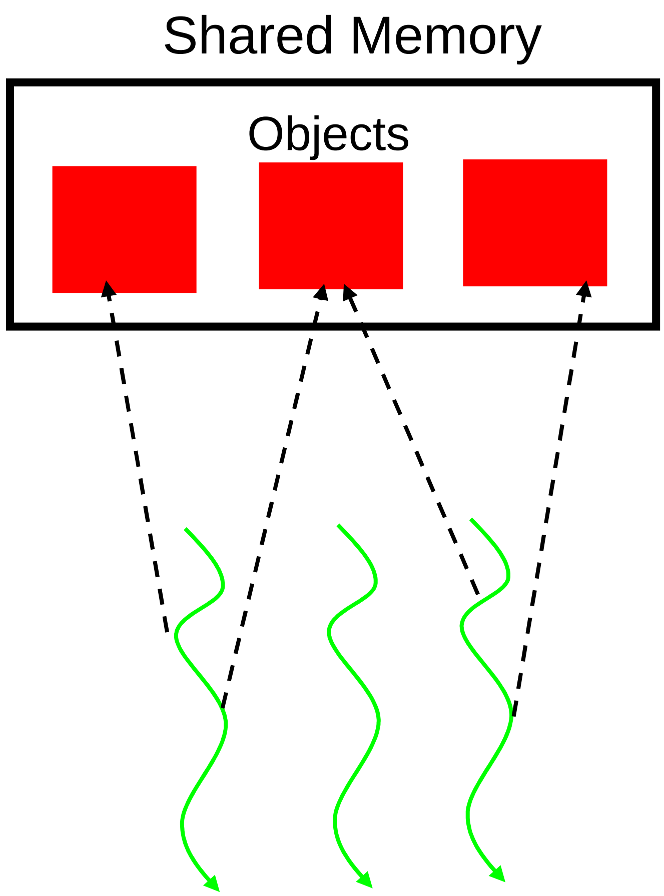
Take-away lessons

- Parallelism in irregular apps depends on "data values"
  - →purely compile-time approach cannot find parallelism
  - →inspector-executor approach (Saltz) cannot find parallelism
    - →optimistic parallelization is the only solution
- Parallelism is data "parallelism" of some kind
  - – but on irregular data structure elements, not arrays
  - – computations with different data items may conflict
  - – how conflicts must be handled depends on app
    - Delaunay: abort all but one conflicting computations
    - Agglomerative clustering: must ensure sequential order is respected

Galois programming model and implementation

Computational model

- Object-based shared-memory model
- Computation performed by some number of threads
  - but programs do not mention threads
- Threads can have their own local memory
- Threads must invoke methods to access internal state of objects
  - mesh refinement:shared objects are
    - worklist
    - mesh



Components of Galois approach

1) Two syntactic constructs for packaging optimistic parallelism as iteration over sets
2) Assertions about methods in class libraries
3) Runtime system for detecting and recovering from potentially unsafe accesses by optimistic computations

(1) Concurrency constructs: two iterators

- *for each e in Set S do B(e)*
  - evaluate block B(e) for each element in set S
  - sequential implementation
    - 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 PoSet S do B(e)*
  - evaluate block B(e) for each element in set S
  - sequential implementation
    - perform iterations in order specified by poSet
    - there may be dependences between iterations
  - set S may get new elements during execution

Galois version of mesh refinement

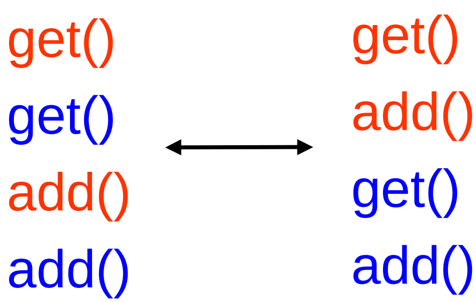
```
Mesh m = /* read in mesh */
Set wl;
wl.add(mesh.badTriangles()); // non-deterministic order

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

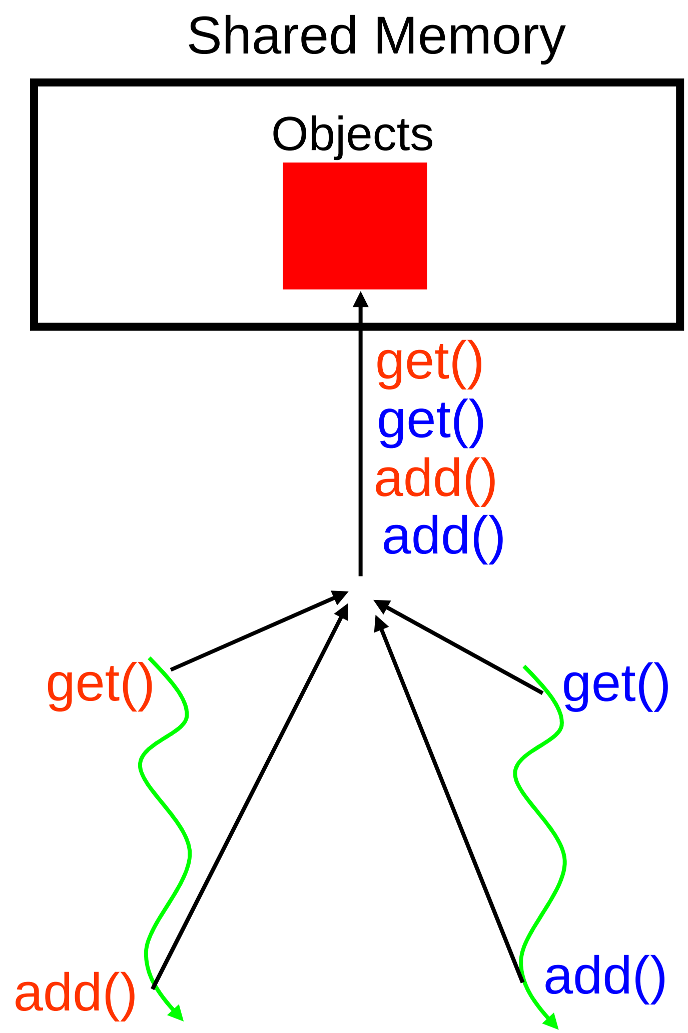
Parallel execution of iterators

- 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 some iterations concurrently with master
  - threads synchronize by barrier synchronization at end of iterator
- Key technical problem: semantics of iterators
  - serializability: result of parallel execution must appear as though iterations were performed in some interleaved order
  - ordering: for poSet iterator, this order must correspond to poSet order

(II) Assertions on methods

- Concurrent accesses to a mutable object by multiple threads are OK provided method invocations commute





Assertions on methods (contd.)

- Semantic commutativity vs. concrete commutativity
  - (e.g.) workset representation may be different for different method invocation orders
  - for client program, this is not relevant
- Information provided by class implementer
  - commutativity of method invocations
  - undo methods
    - (e.g.) add(x) is inverse of remove(x)

(III) Runtime system

- Detect conflicts in method invocations on objects and roll back appropriate iteration
  - maintain logs of method invocations from ongoing iterations
- For PoSet iterator, ensure that iterations commit in order
  - similar to reorder buffer in speculative execution processors

Experiments

Experimental Setup

- Machines
  - 4-processor 1.5 GHz Itanium 2
    - 16 KB L1, 256 KB L2, 3MB L3 cache
    - no shared cache between processors
    - Red Hat Linux
  - Dual processor, dual core 3.0 GHz Xeon
    - 32 KB L1, 4 MB L2 cache
    - dual cores share L2
    - Red Hat Linux

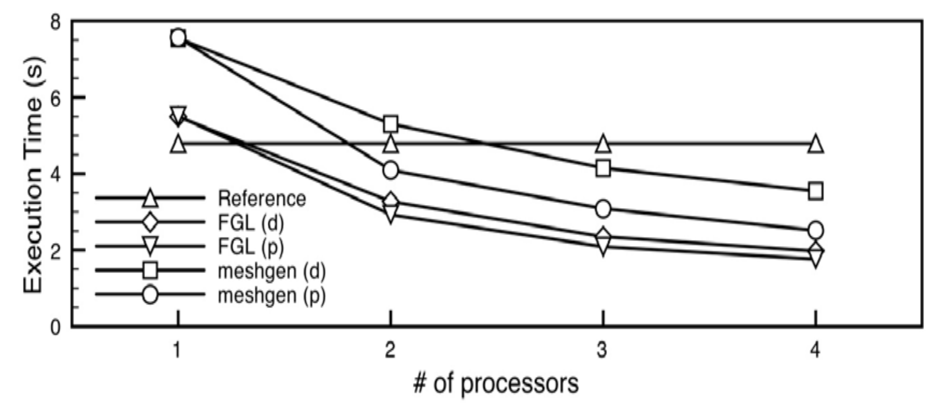
Delaunay mesh generation

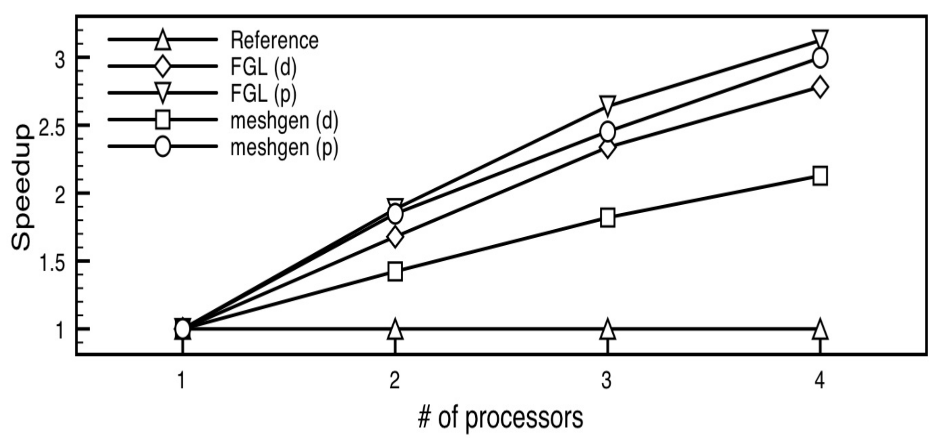
- Workset: implemented using STL queue
- Mesh: implemented as a graph
  - each triangle is a node
  - edges in graph represent triangle adjacencies
  - used adjacency list representation of graph
- Input mesh:
  - from Shewchuck's Triangle program
  - 10,156 triangles of which 4,837 were bad

Code versions

- Three “default” versions
  - reference: sequential version w/o locks/threads/etc.
  - FGL(d): handwritten code that uses fine-grain locks on triangles
  - meshgen(d): Galois version
- Experiments showed high abort ratio
  - fixing a bad triangle creates a cluster of new bad triangles in the cavity
  - if workset is queue, these are co-scheduled with high probability
  - one solution: get() does random selection from workset
- Two other codes with randomized workset:
  - FGL(p), meshgen(p)

Speedups

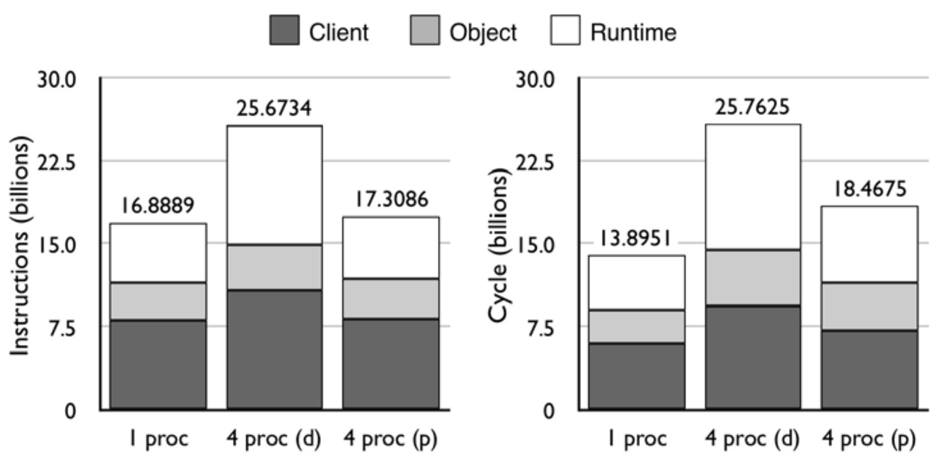




- sequential version is best on 1 processor
- meshgen(p) performs better than meshgen(d)
  - smaller abort ratio
- FGL(d) and FGL(p) perform almost equally well
  - cost of aborts is small in FGL
- FGL(p) and meshgen(p) perform almost equally well

Abort ratios and CPI

	Committed iterations	Aborted iterations
1 proc	21918	n/a
4 proc meshgen(d)	21736	28290
4 proc meshgen(p)	21908	49



- Abort ratio is high for meshgen(d)
- Sequential and meshgen(p) perform almost same number of instructions
- However, cycles/instruction is higher for meshgen(p) mainly because of L3 cache misses

Related Work

- Weihl, 1988 – Concurrency control using commutativity properties of ADTs
- Rinard & Diniz, 1996 – Static commutativity analysis for parallelization
- Wu & Padua, 1998 – Exploiting semantic properties of containers for parallelization
- Hosking & Moss – Open nesting using data structure semantics

Take-away message

- Benchmark programs are bad
  - Programs ☹
  - Algorithms+data structures ☺
- Parallelism in many irregular apps is inherently data-dependent
  - Pointer/shape analysis cannot work for these apps
- Optimistic parallelization is essential for such apps
  - Analysis might be useful though to optimize parallel program execution
- Exploiting abstractions provided by OO is critical
  - Only CS people still worry about F77 and C anyway….
- Exploiting high-level semantic information about programs is critical
  - Galois knows about priority queues, sets, etc.
- Support for ordering speculative computations important
- Good scheduling may require domain-specific knowledge
- These beliefs are basis for Galois project

Thank you!