

Optimistic Parallelism Requires Abstractions

Milind Kulkarni, Keshav Pingali – The University of Texas at Austin
Bruce Walter, Ganesh Ramanarayanan, Kavita Bala and L. Paul Chew – Cornell University

Optimistic Parallelism Requires Abstractions

Milind Kulkarni, Keshav Pingali – The University of Texas at Austin
Bruce Walter, Ganesh Ramanarayanan, Kavita Bala and L. Paul Chew – Cornell University

Motivation

- ◆ Parallel programming very important
 - ◆ Multicore processors
- ◆ Parallel programming is hard!
 - ◆ Limited success in domains which deal with structured data
 - ◆ Array programs
 - ◆ Database applications
- ◆ What about irregular applications which deal with unstructured data?
 - ◆ Compile time techniques have failed

Galois System: Core Beliefs

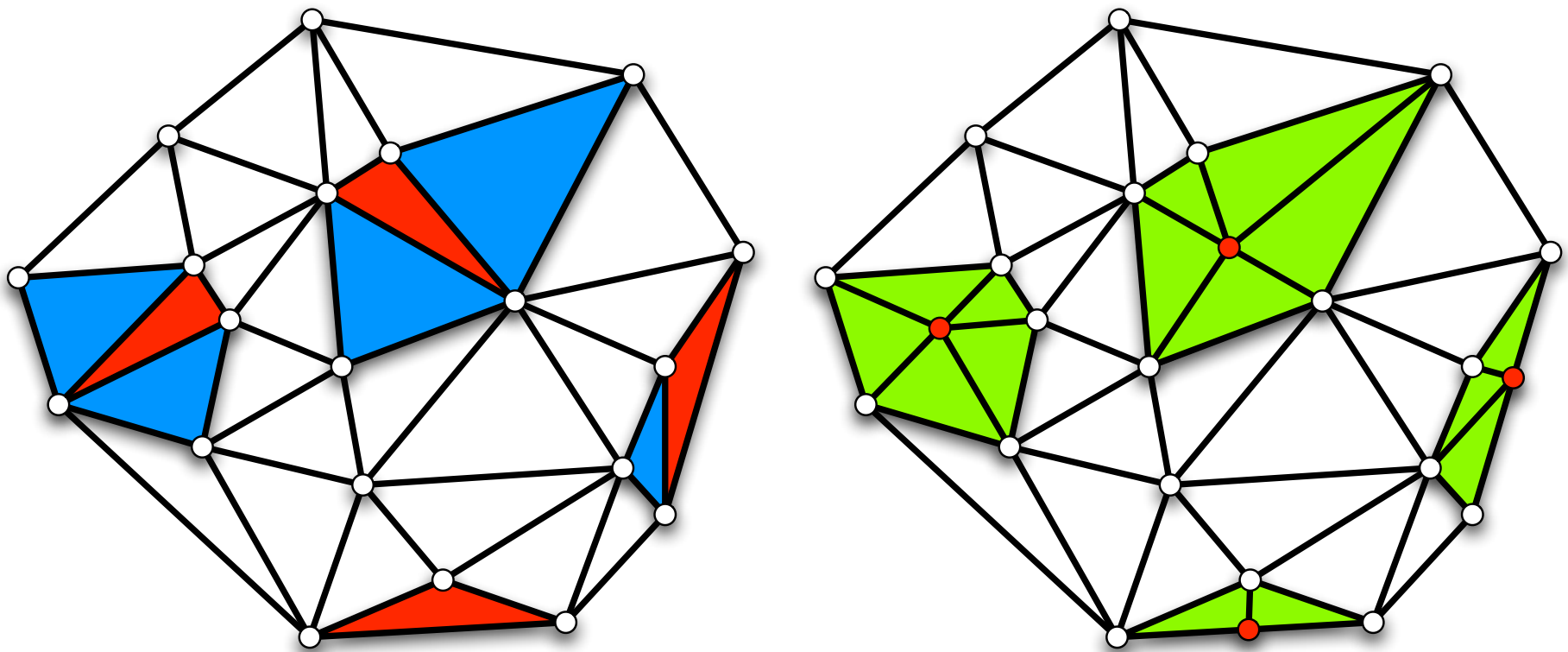
- ✦ Irregular applications have worklist-style data parallelism
- ✦ Optimistic parallelization is crucial
- ✦ Parallelism should be hidden within natural syntactic constructs
- ✦ High level application semantics are critical for parallelization

Outline

- ◆ Two challenge problems
- ◆ Galois programming model and implementation
- ◆ Evaluation
- ◆ Related Work
- ◆ Conclusions

Delaunay Mesh Refinement

- ✦ Iterative refinement procedure to produce guaranteed quality meshes



Delaunay Pseudo-code

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

while (wl.size() != 0) {
    Element e = wl.get();
    if (e no longer in mesh)
        continue;
    Cavity c = new Cavity(e);
    c.expand();
    c.retriangulate();
    mesh.update(c);
    wl.add(c.badTriangles());
}
```

Delaunay Pseudo-code

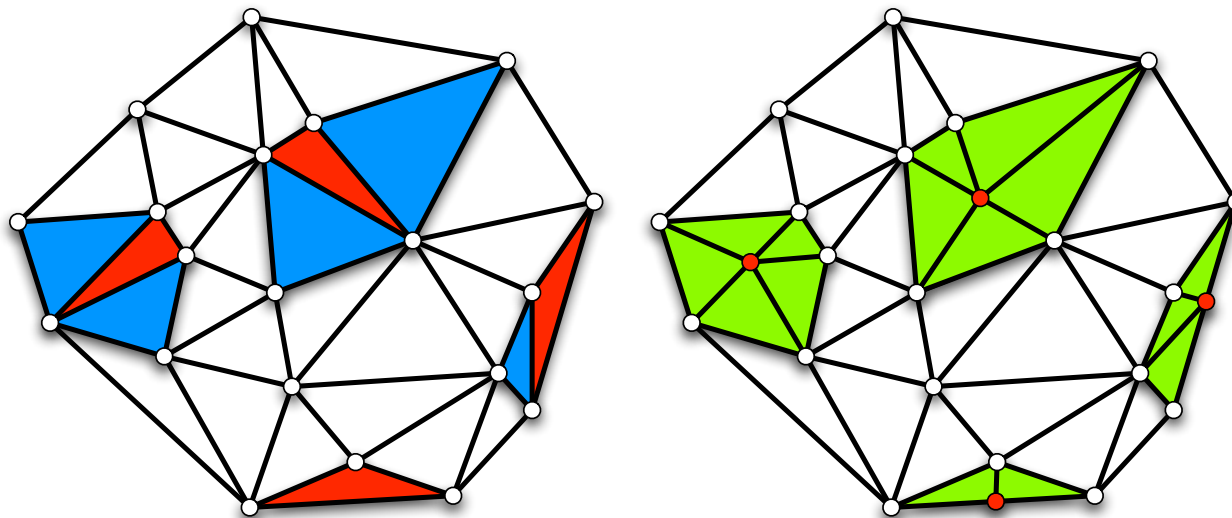
```
Mesh m = /* read in mesh */  
WorkList wl;  
wl.add(mesh.badTriangles());
```

```
while (wl.size() != 0) {  
    Element e = wl.get();  
    if (e no longer in mesh)  
        continue;  
    Cavity c = new Cavity(e);  
    c.expand();  
    c.retriangulate();  
    mesh.update(c);  
    wl.add(c.badTriangles());  
}
```

Worklist idiom

Finding Parallelism

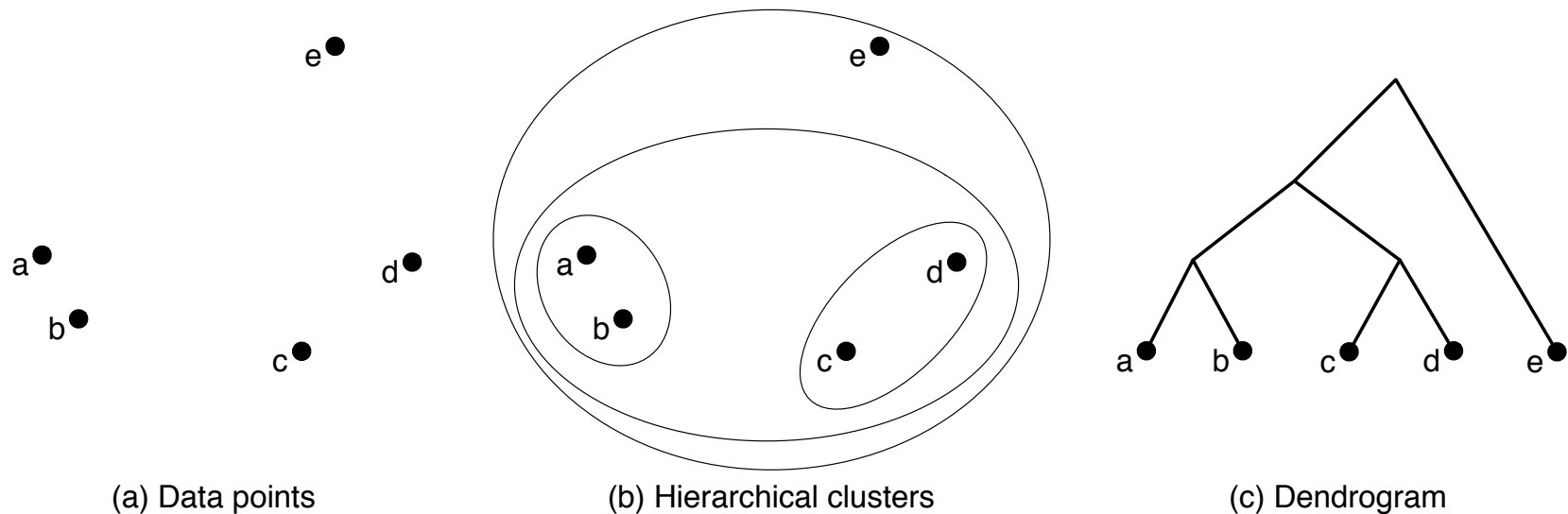
- ✦ Can expand multiple cavities in parallel
 - ✦ Provided cavities do not overlap



- ✦ Determining this statically is impossible
 - ✦ Solution: Optimistic parallel execution

Agglomerative Clustering

- ◆ Create binary tree of points in a space in bottom-up fashion
- ◆ Always choose two closest points to cluster

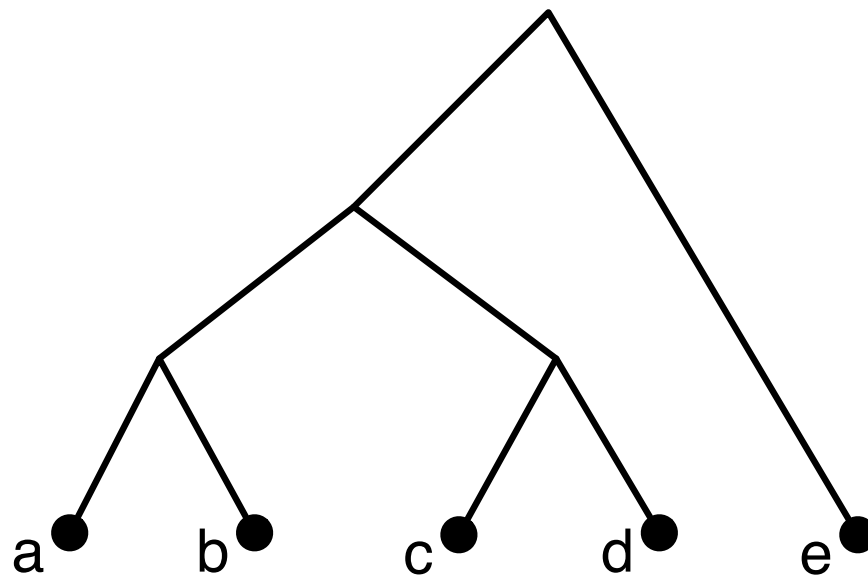


Agglomerative Clustering

- ◆ Two key data structures
 - ◆ Priority Queue – Keeps pairs of points $\langle p, n \rangle$ where n is the nearest neighbor of p
 - ◆ Ordered by distance
 - ◆ KD-tree – Spatial structure to find nearest neighbors

Finding Parallelism

- ◆ Priority queue functions as a worklist
 - ◆ Seems to be completely sequential
- ◆ If clusters are independent, can be done in parallel



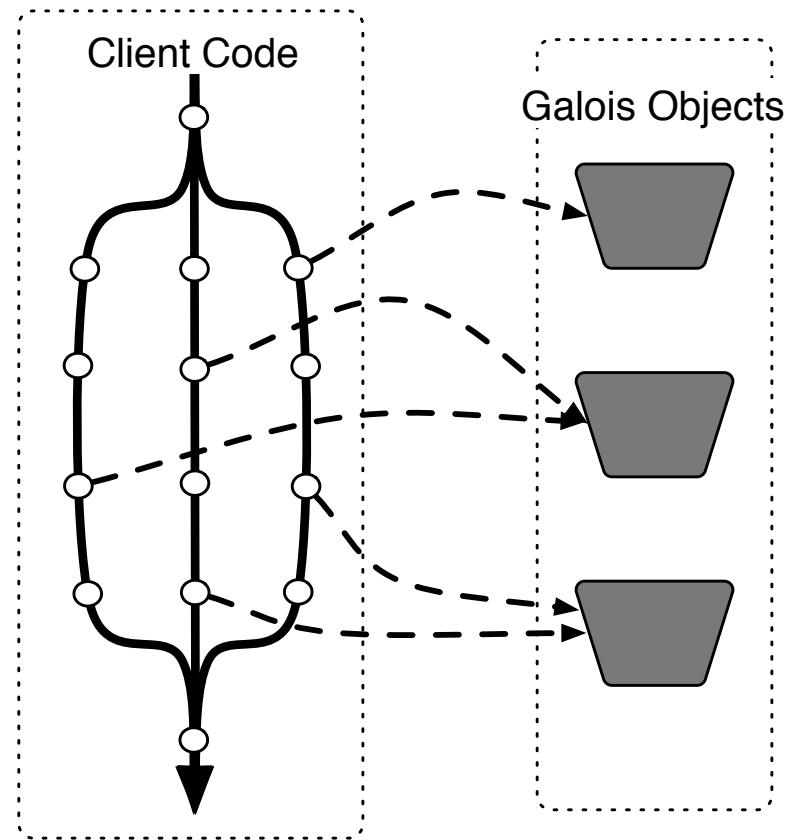
Lessons Learned

- ◆ Worklist-style data parallelism
 - ◆ May be dependences between iterations
- ◆ However, worklist abstractions are missing from the code
- ◆ Concurrent access to shared objects a must
 - ◆ worklist, priority queue, kd-tree

Galois Programming Model and Implementation

Programming Model

- ◆ Object-based shared memory model
 - ◆ Client code must invoke methods to access object state
- ◆ Client code has sequential semantics
 - ◆ But runtime system may execute code in parallel



Worklist Abstractions

- ◆ Iterators over collections
 - ◆ `foreach e in set S do B(e)`
 - ◆ Iterations can execute in any order
 - ◆ As in Delaunay mesh refinement
 - ◆ `foreach e in poSet S do B(e)`
 - ◆ Iterations must respect ordering of S
 - ◆ As in agglomerative clustering
- ◆ May be dependences between iterations
- ◆ Sets can change during execution

Delaunay Example

```
Mesh m = /* read in mesh */
WorkList wl;
wl.add(mesh.badTriangles());
while (wl.size() != 0) {
    Element e = wl.get();
    if (e no longer in mesh)
        continue;
    Cavity c = new Cavity(e);
    c.expand();
    c.retriangulate();
    mesh.update(c);
    wl.add(c.badTriangles());
}
```

Delaunay Example

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

rest of code unchanged

Delaunay Example

```
Mesh m = /* read in mesh */  
WorkList wl;  
wl.add(mesh.badTriangles());  
foreach Element e in wl {
```

Iterators expose worklist abstraction
to runtime system

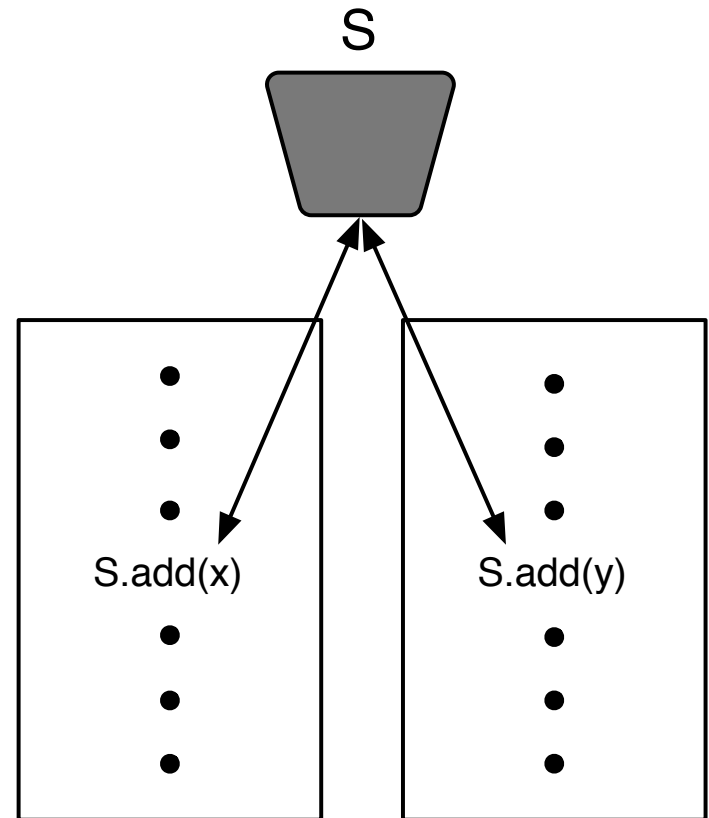
```
    c.expand();  
    c.retriangulate();  
    mesh.update(c);  
    wl.add(c.badTriangles());  
}
```

Execution Model

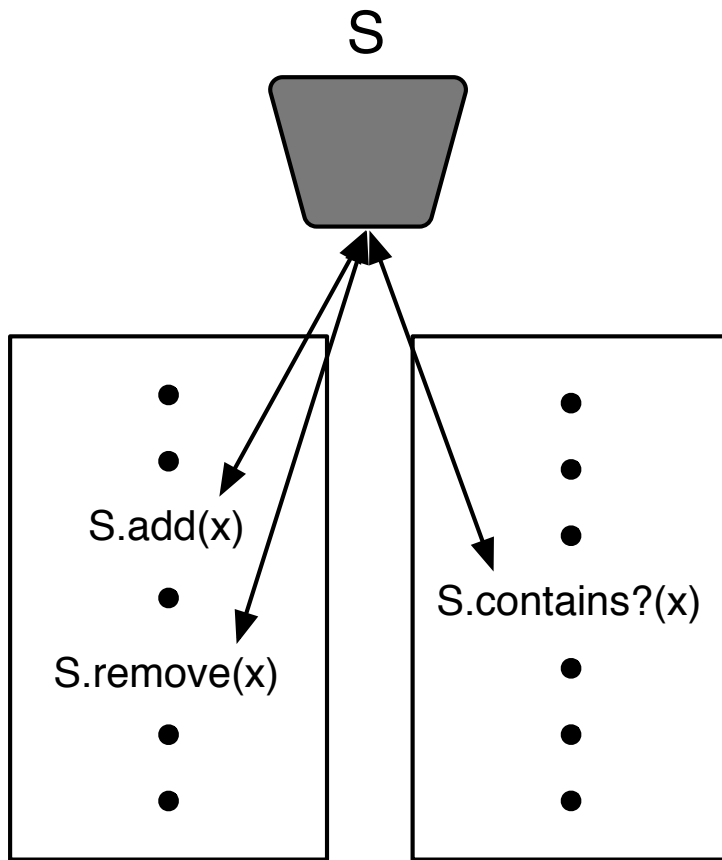
- ◆ Master thread begins execution
- ◆ When it encounters an iterator, it uses helper threads to aid in execution of iterations
 - ◆ Iterations assigned to thread according to scheduling policy (for now, dynamic to ensure load balance)
- ◆ Parallel execution of iterator must respect sequential semantics of iterator
 - ◆ Concurrent access control
 - ◆ Serializability of iterations

Concurrent Access

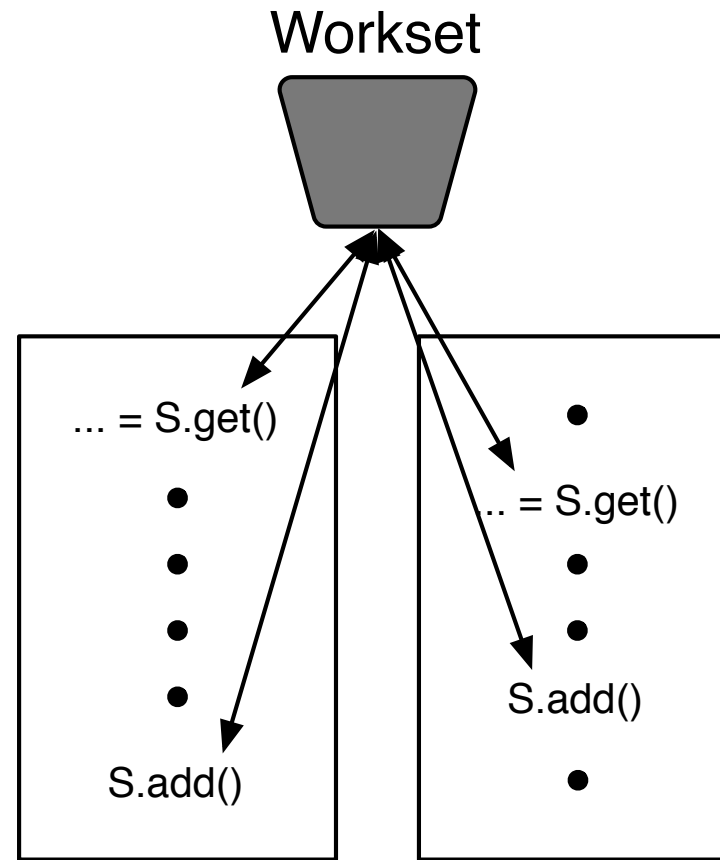
- ◆ Concurrent invocations to a shared object must not interfere
- ◆ Our current implementation uses locks
- ◆ Can use other techniques such as TM



Serializability



(a) Interleaving is illegal



(b) Interleaving is legal
(and necessary)

Semantic Commutativity

- ◆ Method calls which commute can be interleaved
 - ◆ Else, commutativity violation
- ◆ Property of abstract data type
 - ◆ Implementation independent

Galois Classes

- ◆ Inverse methods
 - ◆ Allow for rollback when commutativity violated
- ◆ Commutativity and inverse specified through interface annotation

```
class SetInterface {
    void add(T x);
    [commutes]
        add(y) {y != x}
        remove(y) {y != x}
        contains(y) {y != x}
    [inverse]
        remove(x)

    bool contains(T x);
    [commutes]
        add(y) {y != x}
        remove(y) {y != x}

    ...
}
```


Galois Classes

- ◆ Inverse methods

- ◆ Allow for rollback when commutativity

- ▼ Galois Classes expose abstractions to the runtime system

- ◆ Commutativity and inverse specified through interface annotation

```
class SetInterface {
    void add(T x);
    [commutes]
    add(y) {y != x}
    remove(y) {y != x}
    contains(y) {y != x}
    ...
}
```

Runtime System

- ◆ Two main components:
 - ◆ Global commit pool
 - ◆ Manages iterations
 - ◆ Similar to reorder buffer in OOE processors
 - ◆ Per object conflict logs
 - ◆ Detects commutativity violations
 - ◆ Triggers aborts if commutativity violated

Evaluation

- ◆ Evaluation platform:
 - ◆ Implementation in C++
 - ◆ gcc compiler on Red Hat Linux
 - ◆ 4 processor, shared memory system
 - ◆ Itanium 2 @ 1.5 GHz

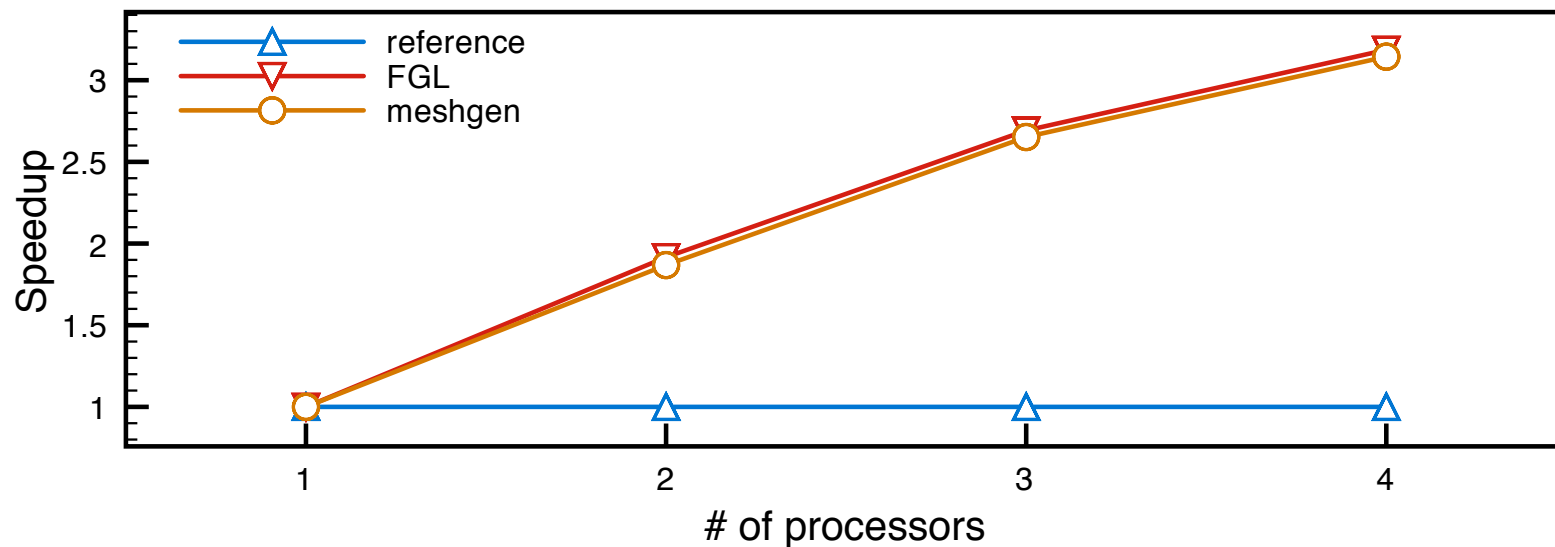
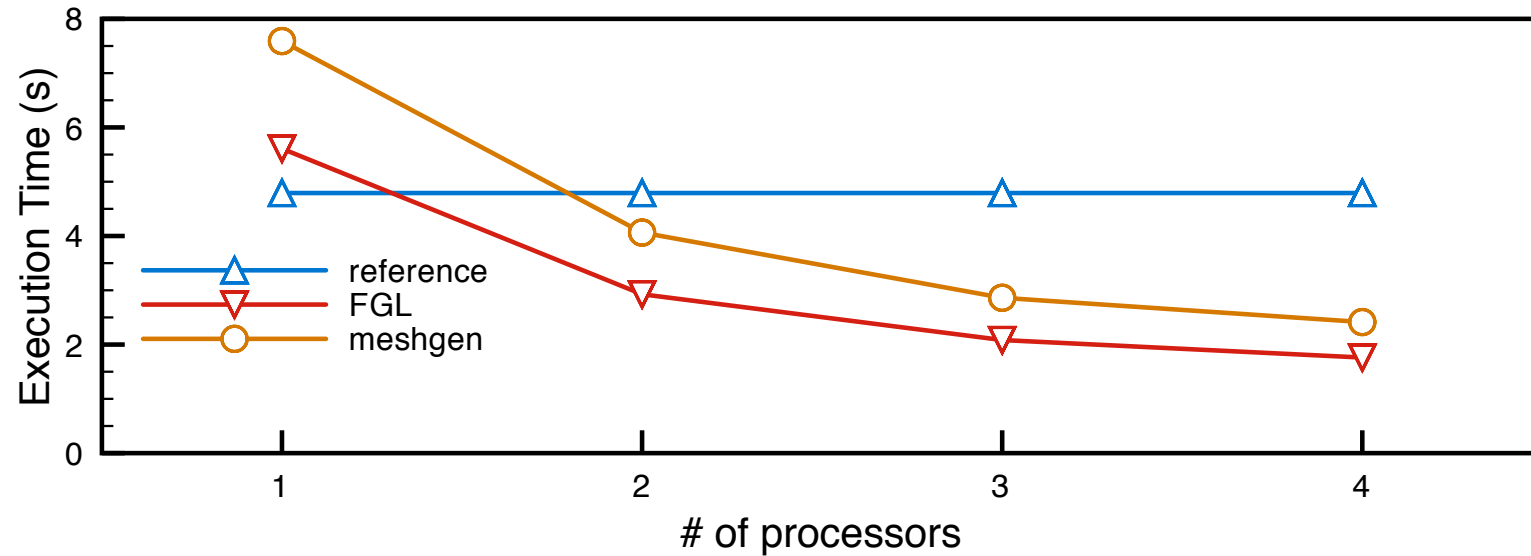
Evaluation – Delaunay

- ◆ Three different versions of benchmark
 - ◆ **reference** – purely sequential code
 - ◆ **FGL** – hand-written, optimistic parallel code using fine-grained locking
 - ◆ **meshgen** – Galois version of code
- ◆ Input mesh generated using Triangle
 - ◆ ~10K triangles
 - ◆ ~4K bad triangles

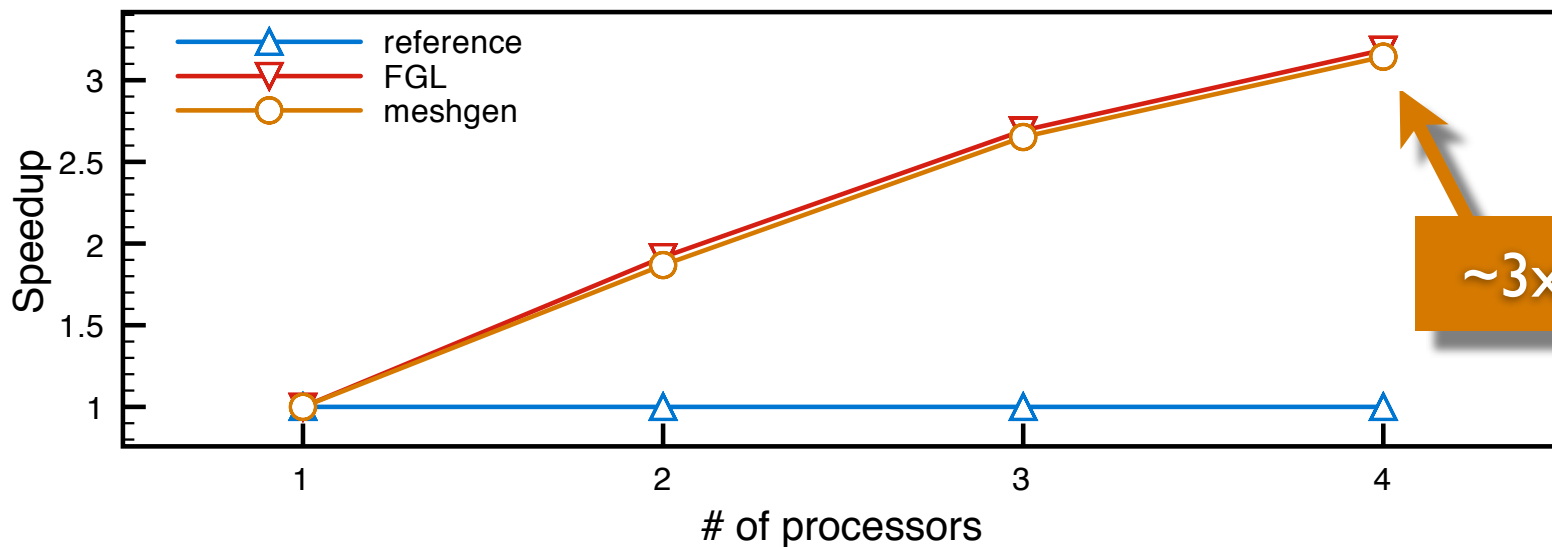
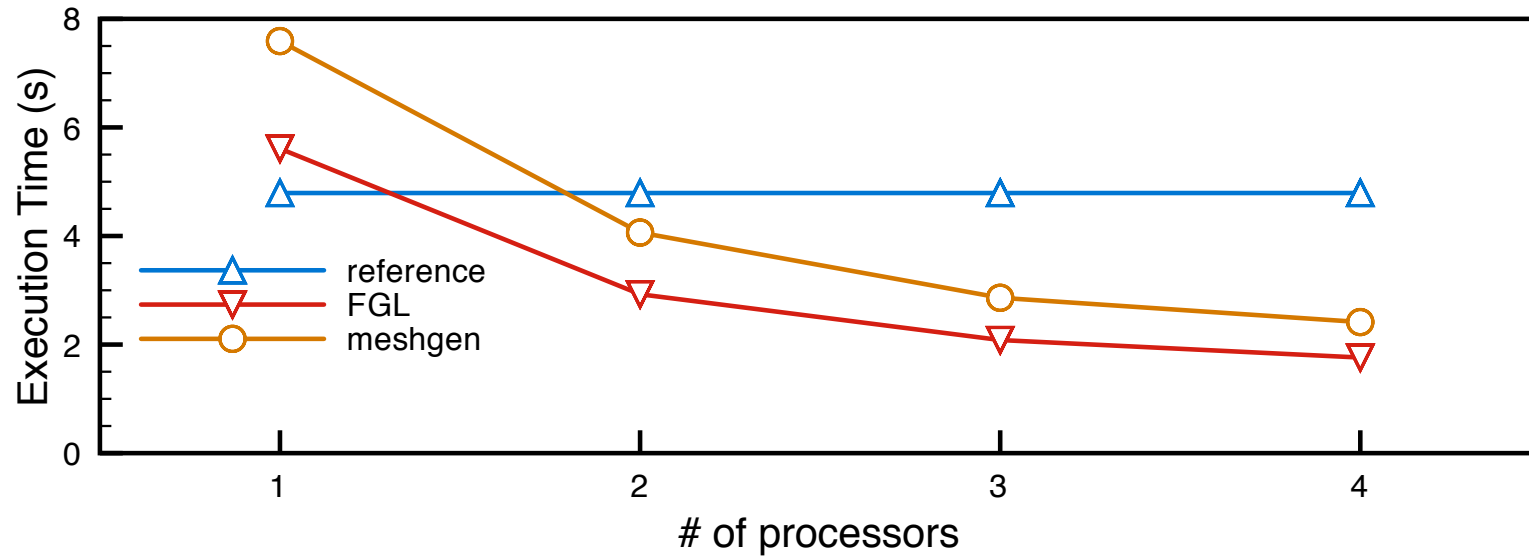
Abort Ratios

- ◆ Optimism must be warranted
 - ◆ Conflicts lead to rollbacks, which waste work
- ◆ FGL and meshgen have abort ratios $< 1\%$ on 4 processors
- ◆ Closely tied to scheduling policy
 - ◆ Choice of proper scheduling policy is crucial for good performance

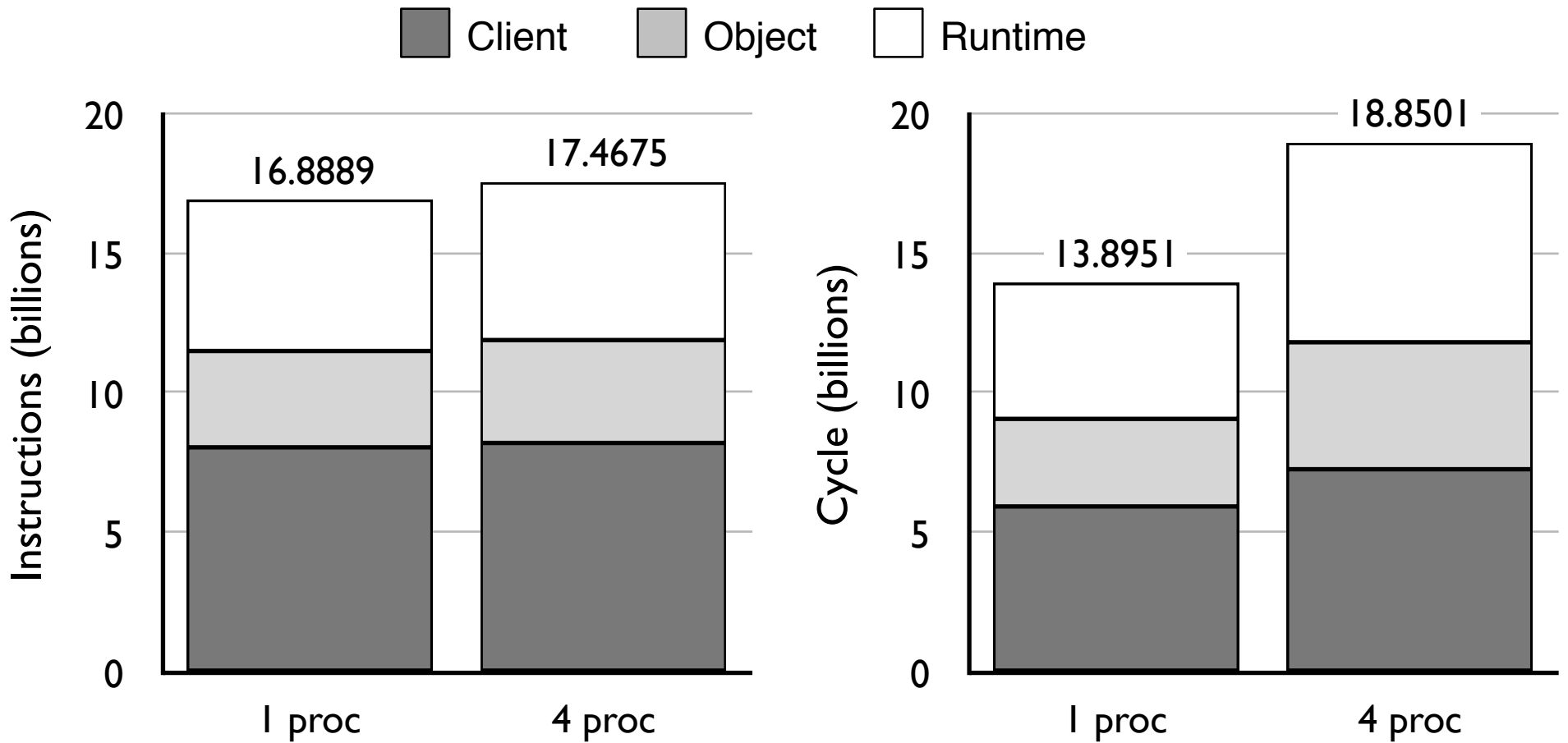
Evaluation – Delaunay



Evaluation – Delaunay



Performance Breakdown



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 in compilation
- ✦ [Ni et al, 2007](#) – Open nesting using abstract locks

Conclusions

- ◆ Optimistic parallelism necessary to parallelize irregular, worklist-based applications
- ◆ Need to exploit high-level semantics
 - ◆ Iterators to expose parallelism
 - ◆ Galois classes to expose semantics of objects

Thank You!

Email: milind@cs.utexas.edu