

Optimistic Parallelization using the Galois System

Milind Kulkarni, Keshav Pingali – University of Texas

Bruce Walter, Ganesh Ramanarayanan, L. Paul Chew and Kavita Bala – Cornell University

Optimistic Parallelization using the Galois System

Milind Kulkarni, Keshav Pingali – University of Texas

Bruce Walter, Ganesh Ramanarayanan, L. Paul Chew and Kavita Bala – Cornell University

Motivation

- ◆ Multicore processors increasingly prevalent
 - ◆ Parallel programming very important
- ◆ Parallel programming is hard!
 - ◆ Must be accessible to more programmers
 - ◆ Irregular applications especially hard
 - ◆ Pointer-based data structures make synchronization difficult

Core Beliefs

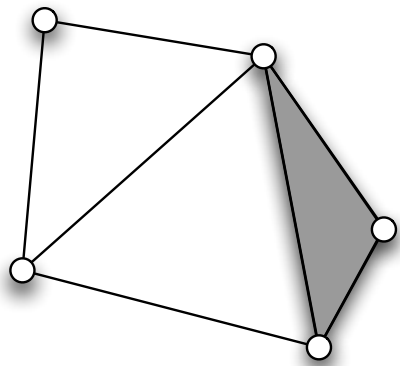
- ✦ Optimistic parallelization is crucial
- ✦ Exploit the semantics of object-oriented applications
- ✦ Expose parallelism through simple syntactic constructs
- ✦ Allow concurrent access to mutable shared data

Outline

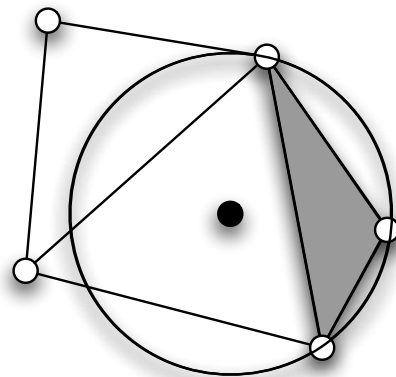
- ◆ Applications
 - ◆ Delaunay mesh refinement
 - ◆ Agglomerative clustering
- ◆ Galois model
- ◆ Runtime system
- ◆ Evaluation
- ◆ Conclusions
- ◆ Related Work

Delaunay Mesh Refinement

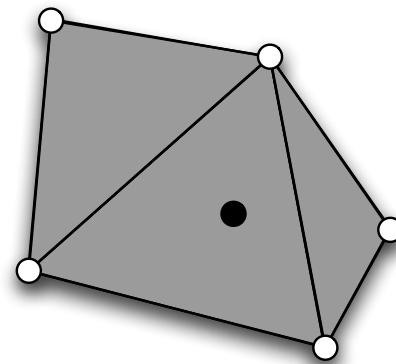
- ✦ Iterative refinement procedure to produce guaranteed quality meshes



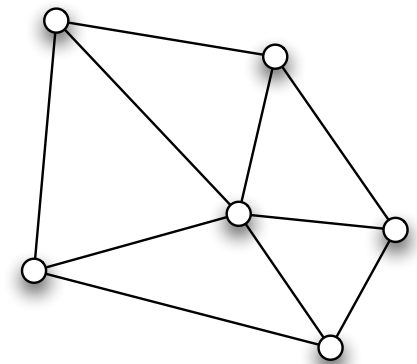
(a)



(b)



(c)



(d)

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());
}
```

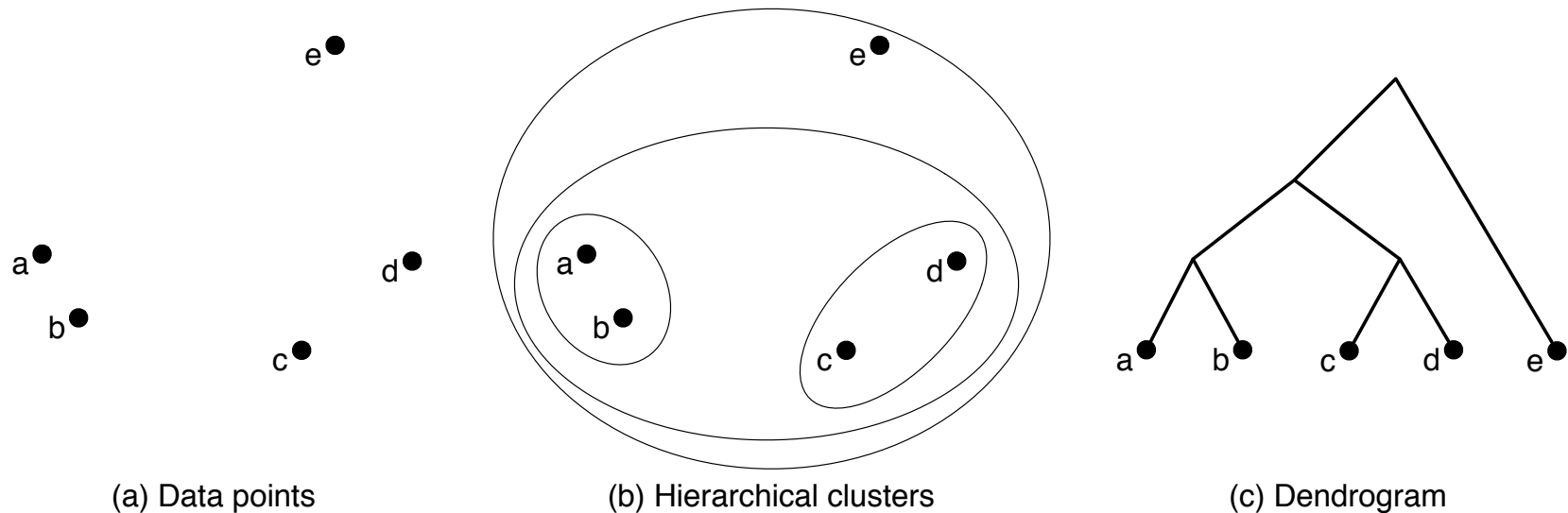
non-deterministic
choice

Finding Parallelism

- ✦ Can expand multiple cavities in parallel
 - ✦ While-loop iterations can execute in parallel
 - ✦ Provided cavities do not overlap
- ✦ Determining this *a priori* is impossible
 - ✦ Optimistic Parallelism!

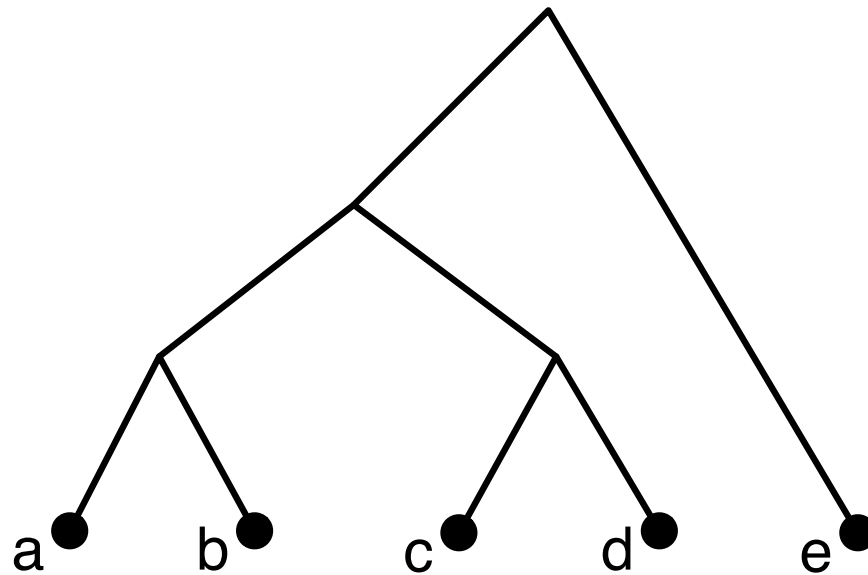
Agglomerative Clustering

- ◆ Create binary tree of points in a space in bottom-up fashion
- ◆ Always choose two closest points to cluster
 - ◆ Using kd-tree & priority queue



Finding Parallelism

- ◆ Also a worklist algorithm
 - ◆ But seems sequential!
- ◆ If clusters are independent, can be done in parallel



Lessons Learned

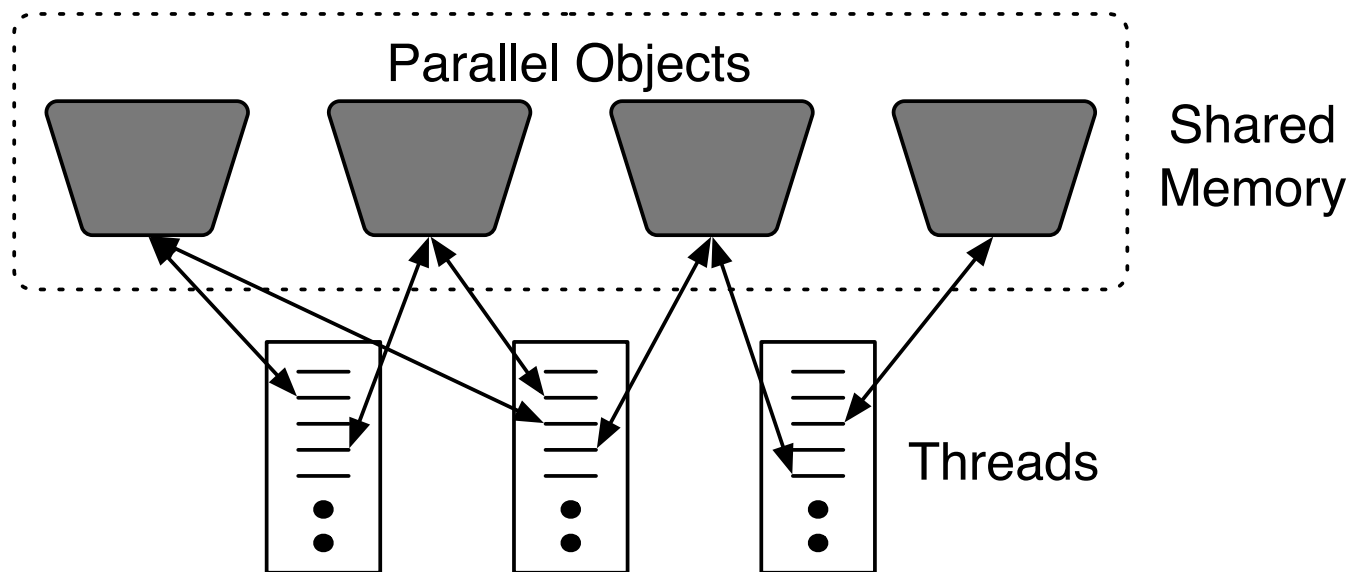
- ◆ Worklist-style data parallelism
 - ◆ But may be dependences between iterations
- ◆ Concurrent access to mutable objects a must
 - ◆ worklist, priority queue, kd-tree

Galois Model

- ◆ Memory model
- ◆ Concurrency constructs
- ◆ Execution model
- ◆ Object model (Galois classes)

Memory Model

- ◆ Shared memory
 - ◆ Collection of shared objects
 - ◆ Only accessed through object interfaces



Concurrency Constructs

- ◆ Iterators over collections
 - ◆ `foreach e in set S do B(e)`
 - ◆ Iterations can execute in any order
 - ◆ As in Delaunay mesh generation
 - ◆ `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());
}
```

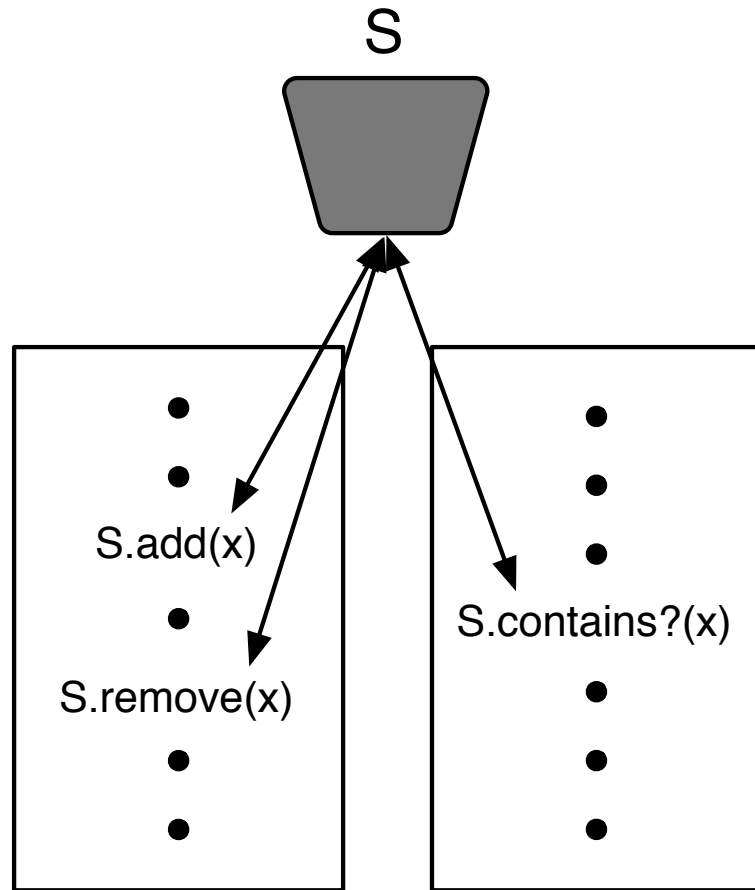
Delaunay Example

```
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());
}
```

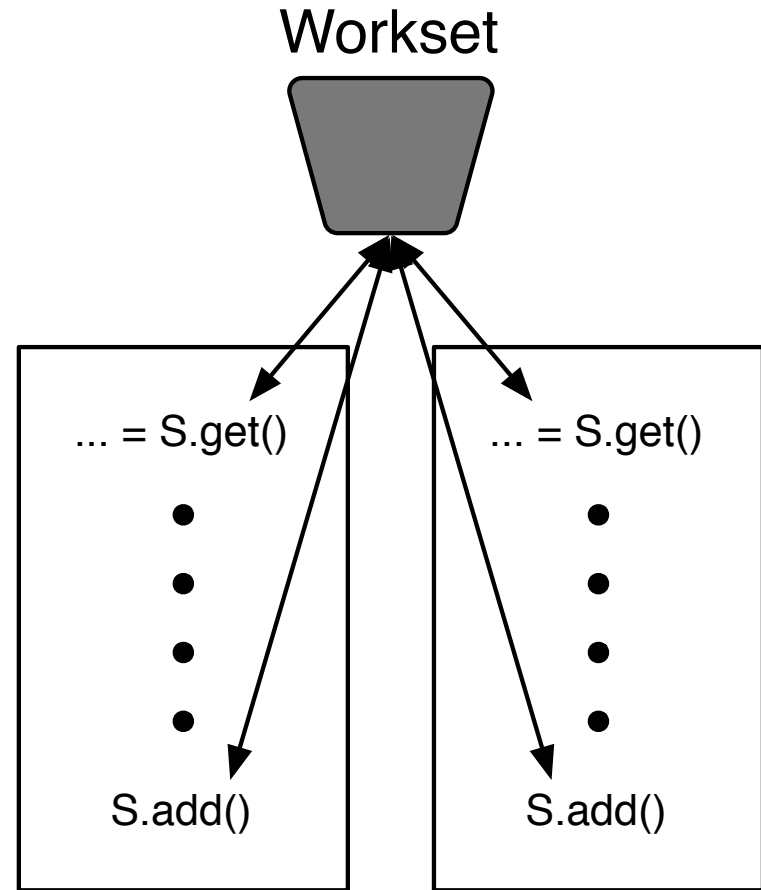
Execution Model

- ✦ Master thread begins execution and encounters iterator
- ✦ Spawns 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 match sequential semantics of iterator

Concurrent Access to Mutable Objects



(a) Interleaving is illegal



(b) Interleaving is legal

Galois Classes

- ◆ Semantic commutativity
 - ◆ Method calls which commute can be interleaved
 - ◆ Else, commutativity violation
- ◆ Property of interface
 - ◆ Implementation independent

Galois Classes

- ◆ Inverse methods
 - ◆ Allow for rollback when commutativity violated
- ◆ Specification through annotation of interface

```
class SetInterface {  
    void add(T x);  
        [conflicts]  
        add(x)  
        remove(x)  
        contains(x)  
    [inverse]  
        remove(x)  
  
    bool contains(T x);  
        [conflicts]  
        add(x)  
        remove(x)  
  
    ...  
}
```

Galois Classes

specify conflicts

- ◆ Inverse methods
 - ◆ Allow for rollback when commutativity violated
- ◆ Specification through annotation of interface

```
class SetInterface {  
    void add(T x);  
        [conflicts]  
            add(x)  
            remove(x)  
            contains(x)  
        [inverse]  
            remove(x)  
  
    bool contains(T x);  
        [conflicts]  
            add(x)  
            remove(x)  
  
    ...  
}
```

Galois Classes

specify inverses

- ◆ Inverse methods
 - ◆ Allow for rollback when commutativity violated
- ◆ Specification through annotation of interface

```
class SetInterface {  
    void add(T x);  
    [conflicts]  
    add(x)  
    remove(x)  
    contains(x)  
    [inverse]  
    remove(x)  
  
    bool contains(T x);  
    [conflicts]  
    add(x)  
    remove(x)  
  
    ...  
}
```

Runtime System

- ◆ Two main components:
 - ◆ Global commit pool
 - ◆ Manages iterations
 - ◆ Similar to reorder buffer in OOE processors
 - ◆ Each iteration has associated iteration record which maintains execution status and priority
 - ◆ Per object conflict logs
 - ◆ Detects commutativity violations

Conflict Logs

- ◆ Detects commutativity violations
- ◆ For each method in class, maintains conflict set
- ◆ Methods invoked by currently executing iterations placed in appropriate conflict set
 - ◆ Entries contain arguments and return values
- ◆ When new method called, conflict sets checked to check for violations

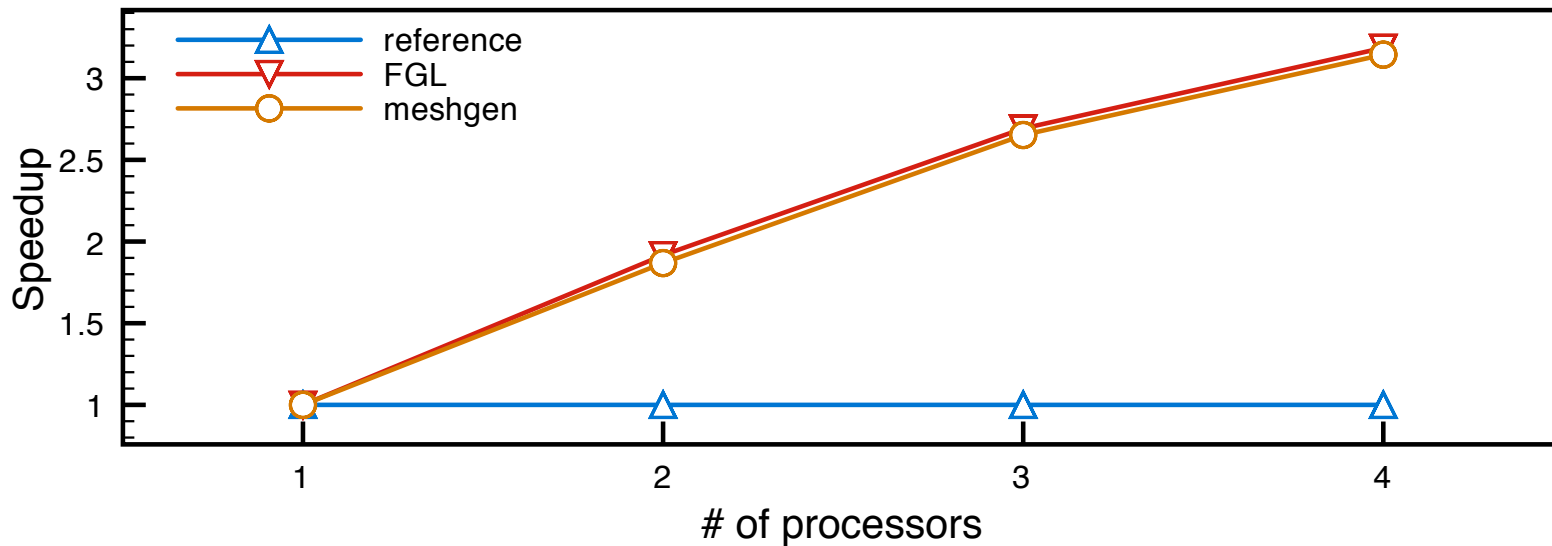
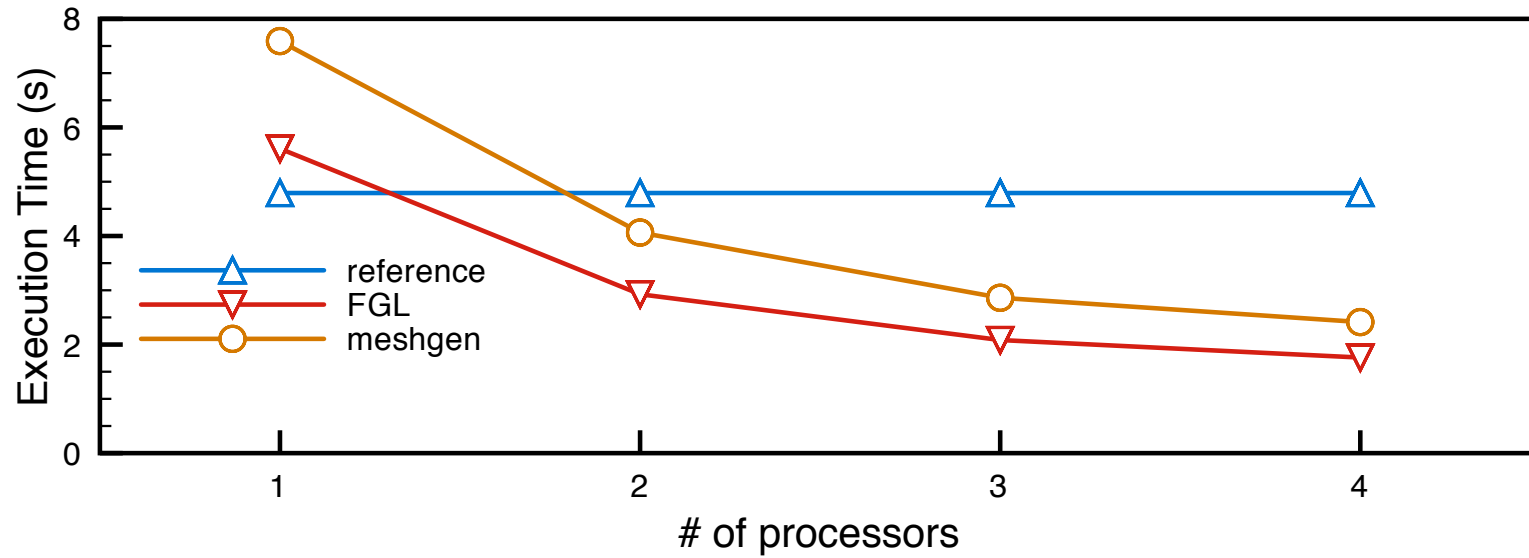
Evaluation

- ◆ Evaluation platform:
 - ◆ 4 processor, shared memory system
 - ◆ Itanium 2 @ 1.5 GHz
 - ◆ Red Hat Linux & gcc compiler

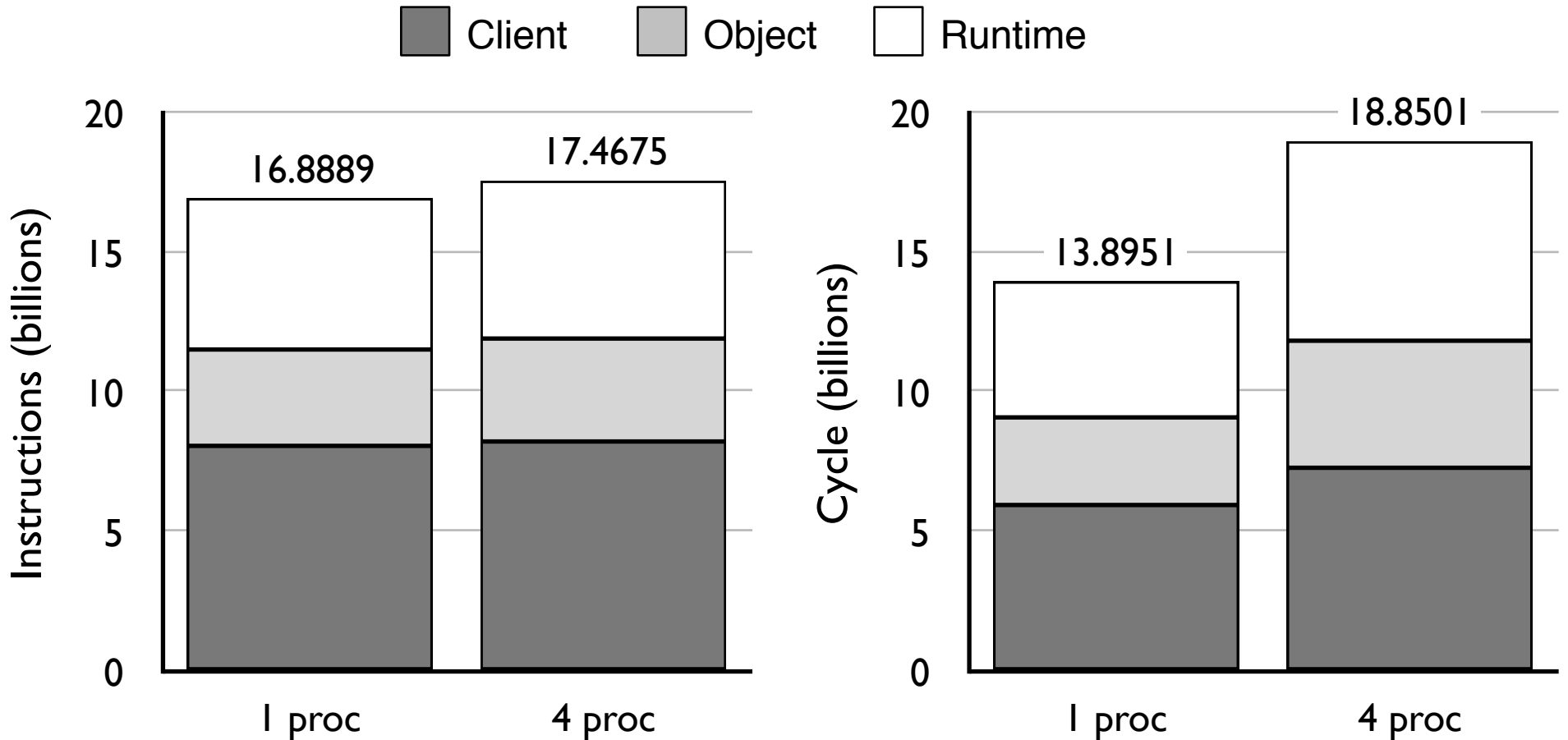
Evaluation – Delaunay

- ♦ Three different versions of benchmark
 - ♦ **reference** – purely sequential code
 - ♦ **FGL** – hand-written parallel code using fine-grained locking
 - ♦ **meshgen** – Galois version of code

Results



Performance Breakdown for meshgen

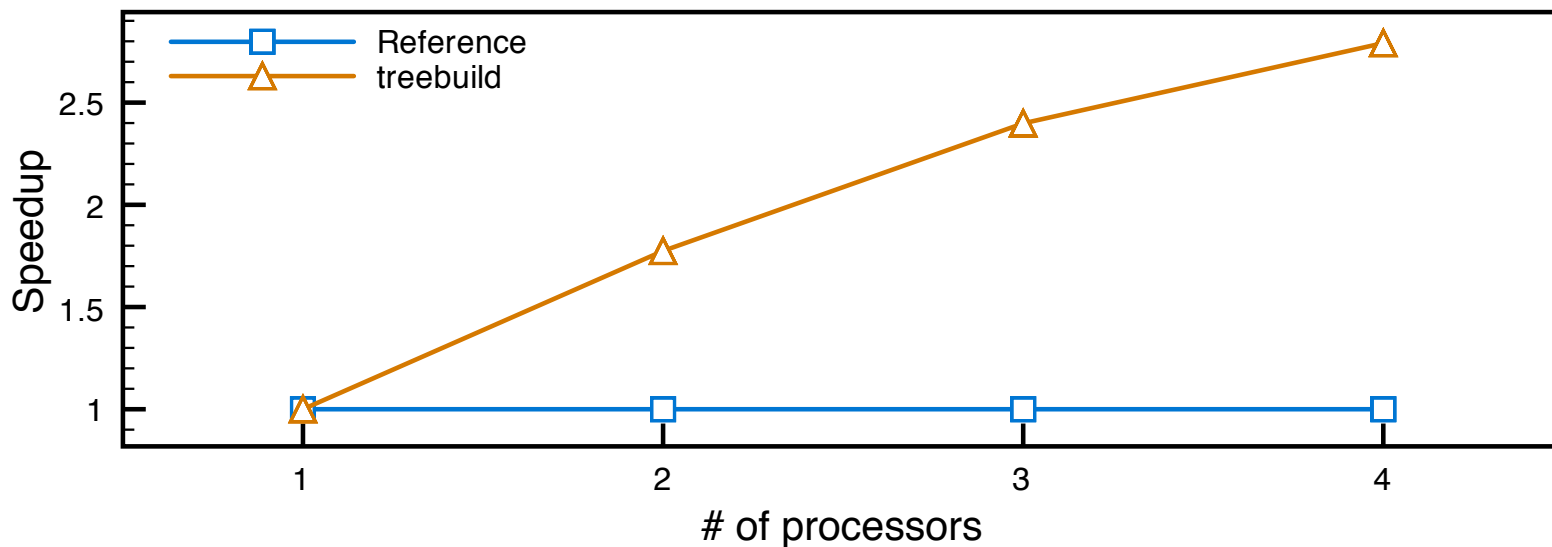
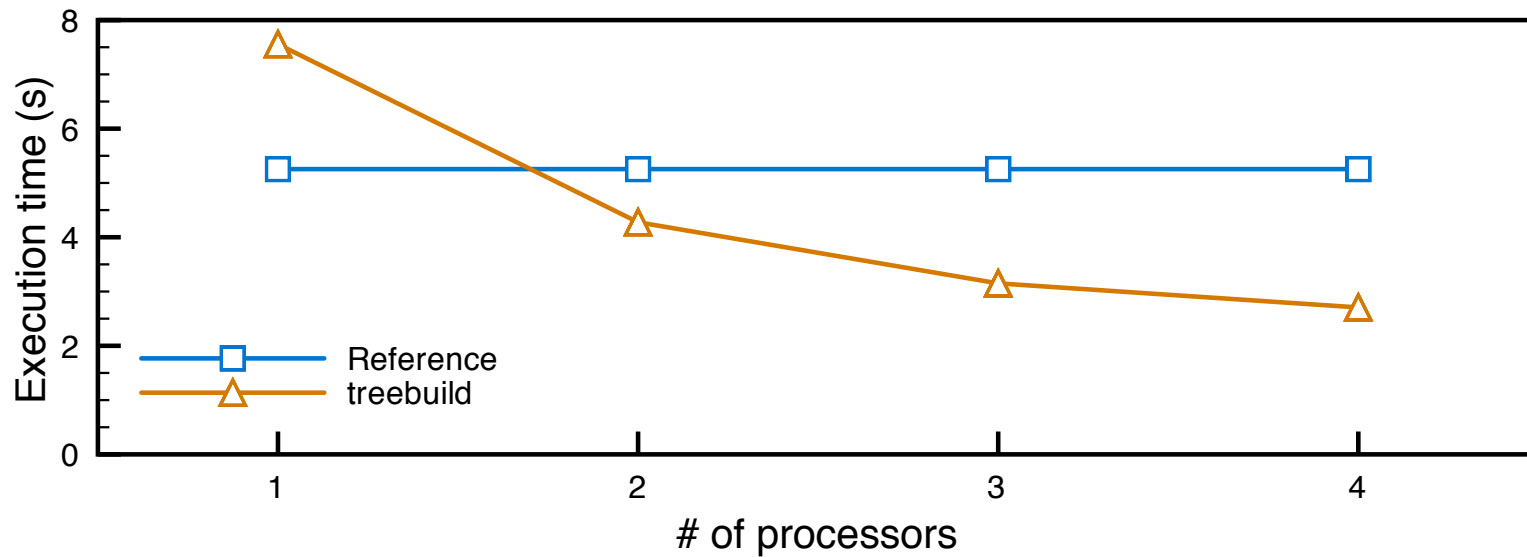


*4 processor numbers are summed over all processors

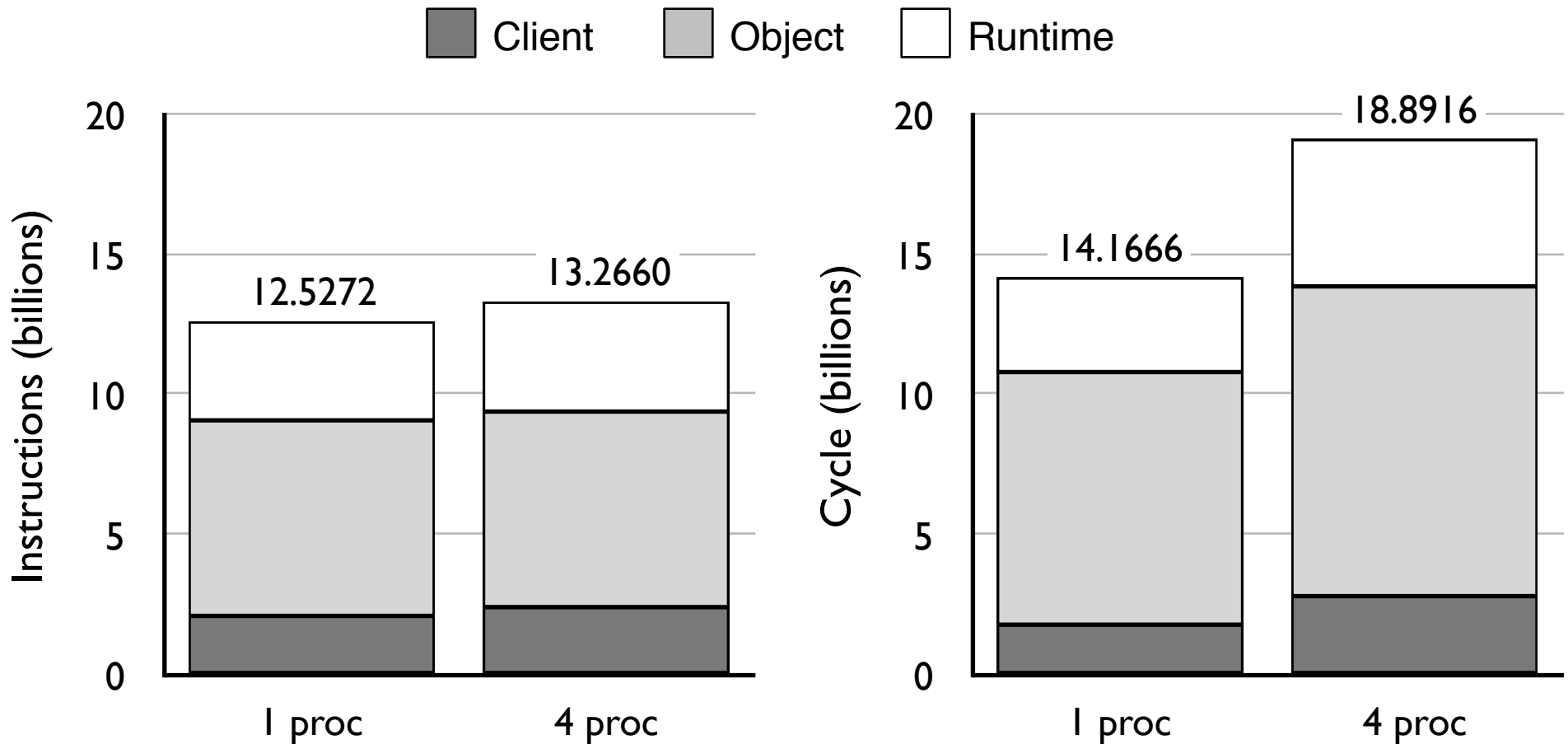
Evaluation – Clustering

- ♦ Two different versions of benchmark
 - ♦ **reference** – purely sequential code
 - ♦ **treebuild** – Galois version of code

Evaluation – Clustering



Performance Breakdown



Core Beliefs Validated

- ✦ Optimistic parallelization is crucial
- ✦ Exploit the semantics of object-oriented applications
- ✦ Expose parallelism through simple syntactic constructs
- ✦ Allow concurrent access to mutable shared data
- ✦ Galois system achieves significant speedup on two real-world irregular applications

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

Questions/Comments?