Galois
Language assistance for runtime parallelization

Programming language problems

- Programming languages cause valuable information to be lost when expressing an algorithm
- The programmer is forced to specify a sequential order on the execution of a program
  - This order may be more restrictive than necessary
  - Thus, when processing elements of an unordered set, an iterator will specify an overly stringent order
- Methods may be commutative and this is not expressed in common languages
- Two approaches: discover information at runtime (discussed), or have languages express needed information

Programming language problems cause compiler problems

- Much of what compilers do is to decide if operations are independent and can be reordered
  - Reverse engineering what may already be known by programmers
- That some data structure is a graph, tree, singly linked list, etc., is generally known to a programmer
  - Incredibly hard for compiler to figure out
  - Shape analysis does this, but does not work well with large, realistic programs

Can languages overcome this?

- Note that Java, Pthreads and C++ have some or all of iterators, thread safe standard data structures, forks and joins, etc.
  - These are generally implemented as method calls
  - Are opaque to compilers
  - And are very large and complex
  - Do not have runtime support to allow speculative execution, which is necessary

Galois Project*

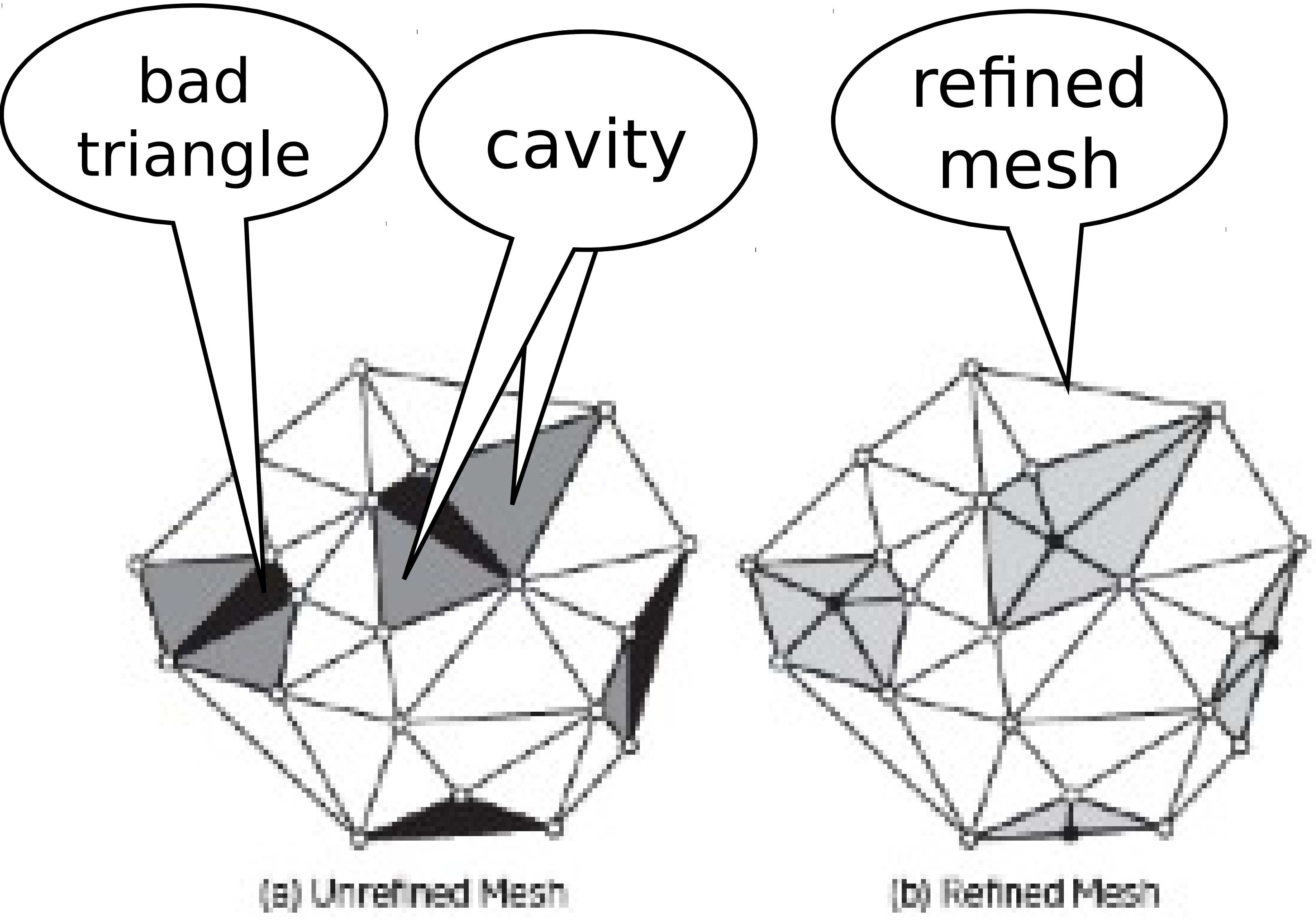
- Galois is one project that seeks to overcome these limits
- Provides abstractions to allow programmer to give information about ordering, commutativity
- Programmer writes a sequential program, compiler generates a parallel execution
- Similar to what databases provide

The Tao of Parallelism in Algorithms, Pingali et al., PLDI 2011

Optimistic parallelism requires abstractions, Kulkarni, Pingali et al., CACM September 2009

A running example
Delauney mesh refinement

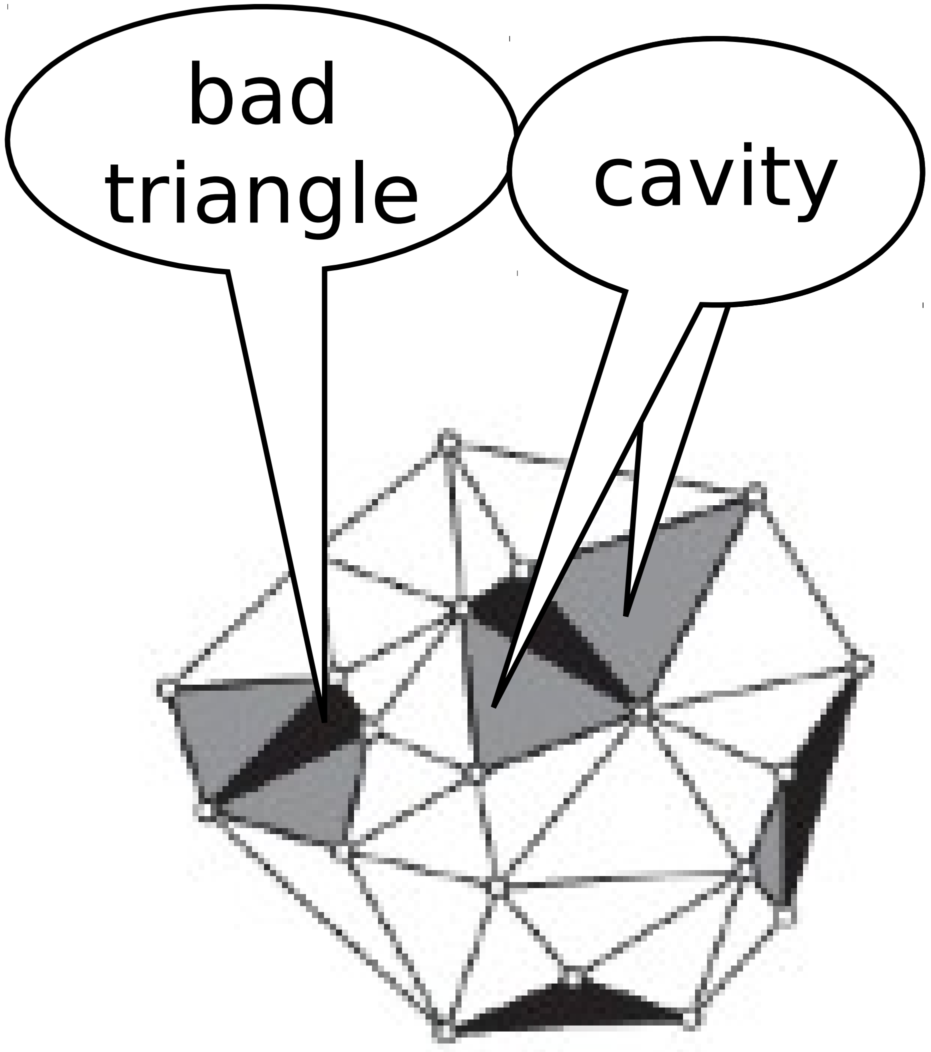
From Kulkarni, Pingali et al., CACM September 2009



(a) Unrefined Mesh (b) Refined Mesh

- Some triangles on the mesh are bad (e.g., too large, bad angles)
- Affects of refinement on *cavity* must be taken into account
- Refinement can often happen in parallel

Available parallelism



(a) Unrefined Mesh

- If two bad triangles do not have overlapping cavities, they can be processed in parallel
- Kulkarni, et al. measured
  - a mesh of 100,000 triangles,
  - ~50% bad
  - ~256 independent bad triangles for most of execution
- Data structure is a graph that is modified repeatedly during execution

Alternate solutions (1)

- Inspector/executor: traverse the structure and find independent work, then do the work
  - Works best if inspector can be done once and executor done many times which happens only if the structure does not change during work, not true here
  - Used for sparse matrix computations, but will not work here
- Shape analysis
  - Graph has no particular structure, shape analysis will not enable parallelization

Alternate solutions (2)

- Hudson's method*
  1. compute cavities of all bad triangles
  2. find maximal independent set of cavities
  3. fix those cavities
  4. repeat 1-3 until no bad triangles
- This works well, but appropriate only to this problem
- A more general technique is desirable - we want to solve lots of problems, not just mesh refinement

***Sparse parallel Delaunay mesh refinement*, Hudson, Miller, Phillips, SPAA 2007

Goals

- Allow programmer to naturally express:
  1. Operations that are ordered and unordered
  2. Operations that commute with one another because of the application
  3. Operations that commute with one another because of data structure semantics
     - In a linked list representation of a set, the order of insertion is irrelevant
     - Two different linked lists may result, but the set represented is identical
- 2 and 3 are instances of *semantic commutativity* - not strictly commutative, but commutative because of the task semantics

Semantic *vs.* concrete commutativity

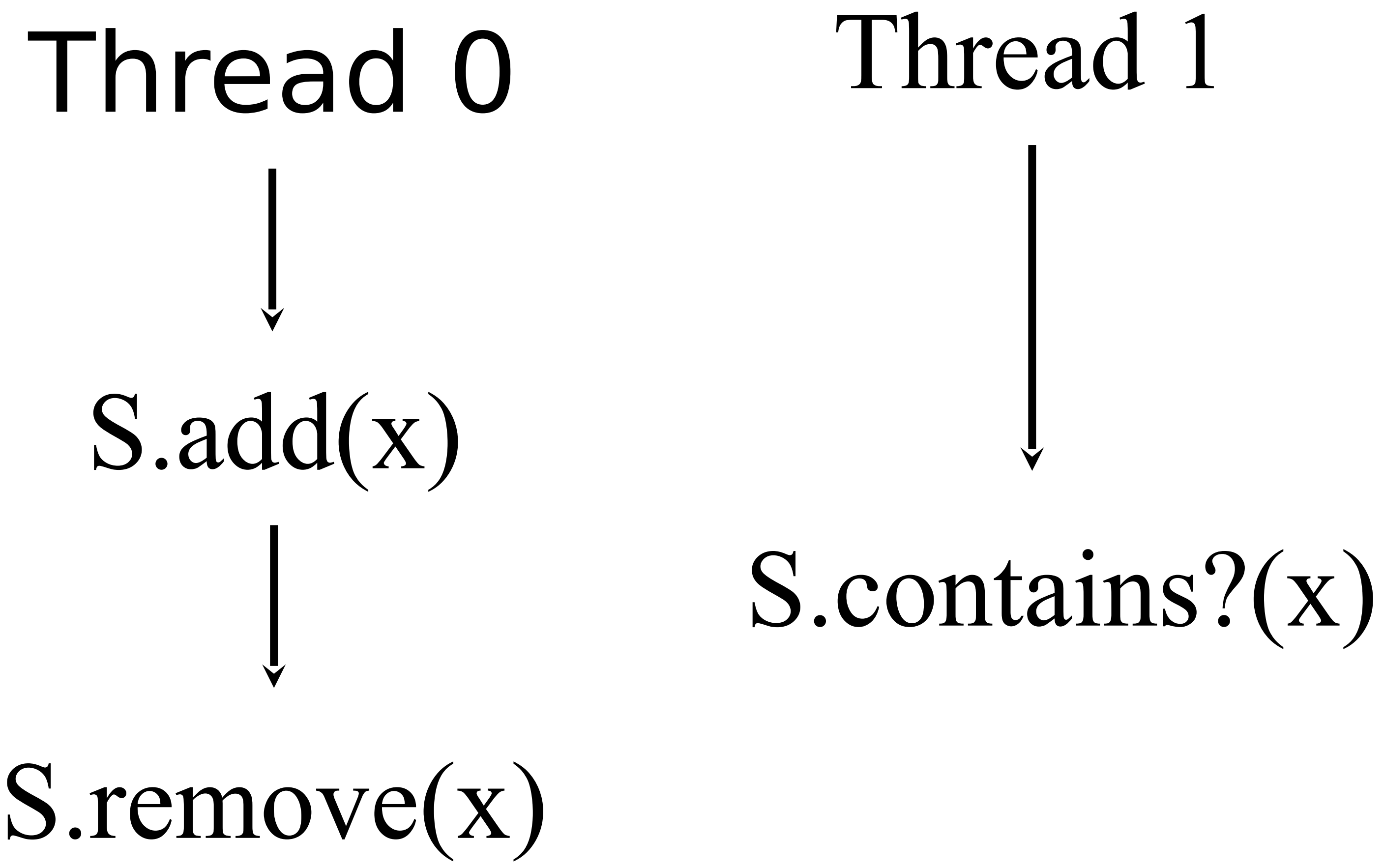
- Semantic commutativity means that when operations commute the meaning of the resulting state is correct even though the values may be different
  - Representation of a set by a linked list, mentioned previously, is an example of semantic commutativity
- Concrete commutativity means that when operations commute the result is the same
  - In the set representation example, the resulting linked list, not just the represented set, would be identical
- Programmers often make use of semantic commutativity
- Compilers can only find concrete commutativity
- Cannot intuit programmer's intention

Galois language

Two iterators over sets are supplied

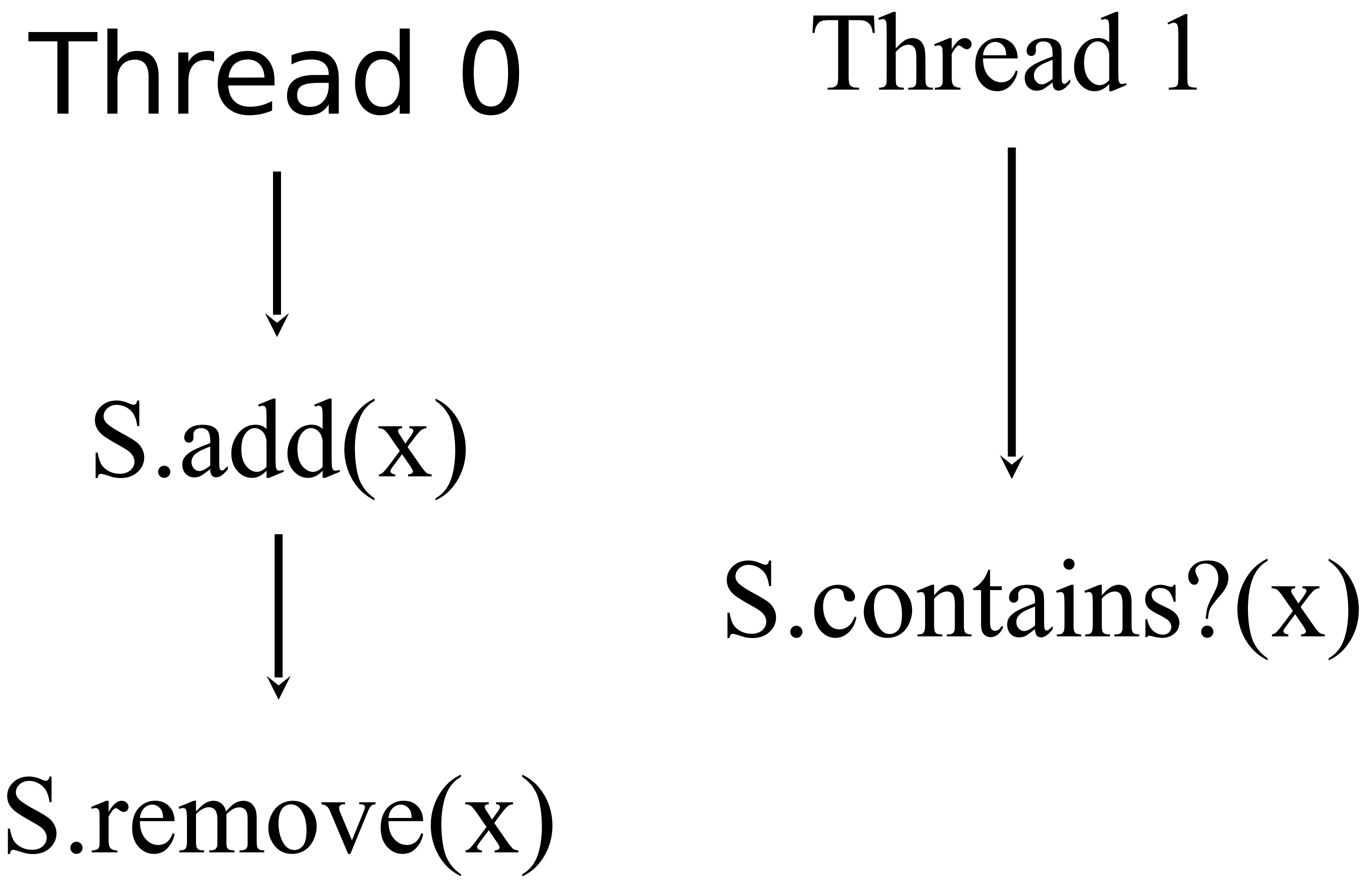
1. Unordered: *for each e in set S do B(e)*
   - Body *B* executed on each element *e*
   - Any *serial* order of executing iterations legal
   - Iterations can add elements to S
2. Ordered: *for each e in Poset S do B(e)*
   - like the unordered iterator, except it must respect orders specified by the partially ordered set *S*
   - Iterations can add elements to *S*

Parallel semantics



- Our old friend atomicity returns.
- Does S.contains?(x) ever return true?
- Not if add/remove are atomic, can if they are not

Parallel semantics



Thread 0

S.add(x)

S.remove(x)

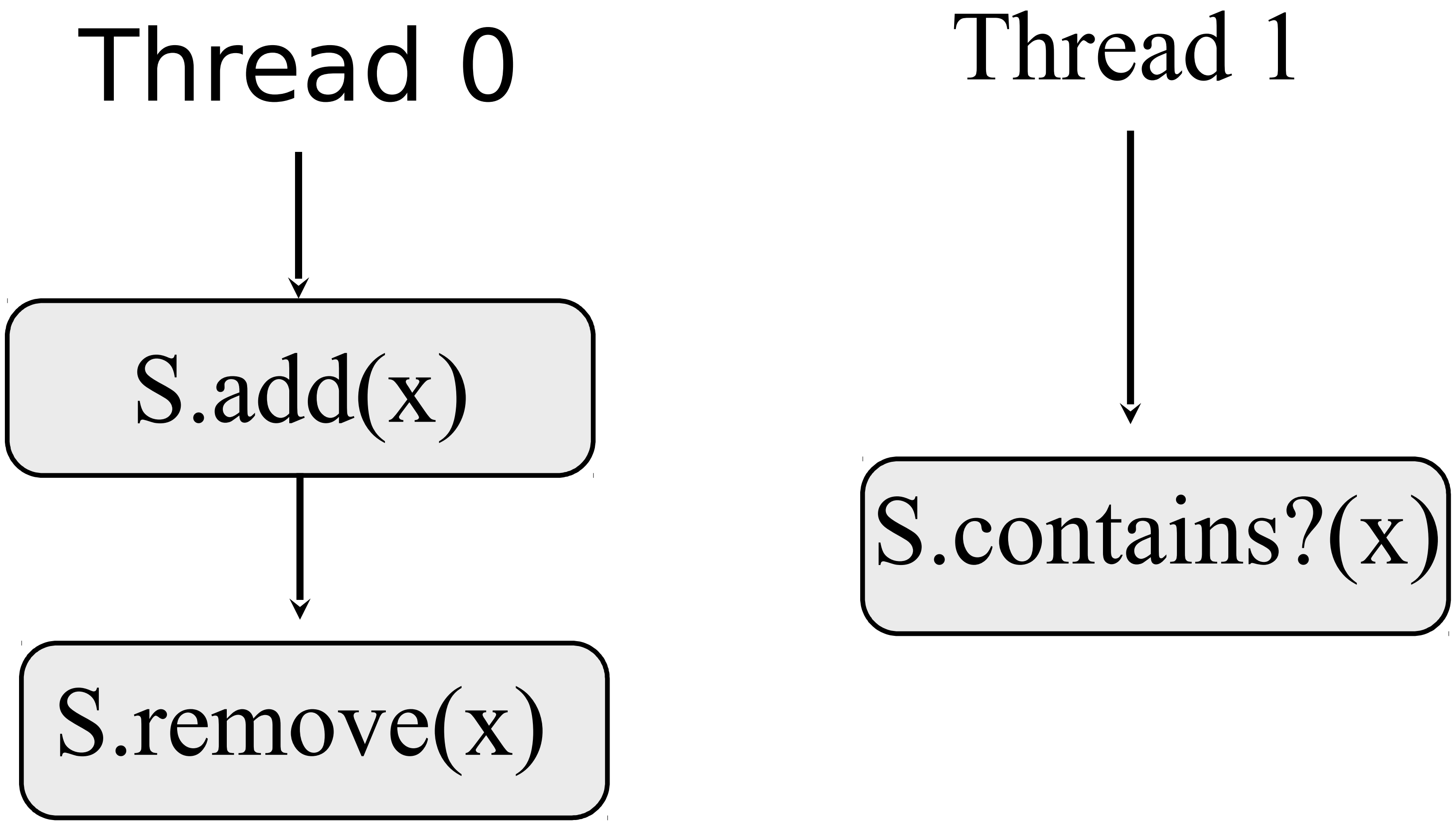
Thread 1

S.contains?(x)

- Galois requires iterators give serial semantics, i..e, outcome is as if iterations ran serially in some order allowed by the program semantics.

This requires atomicity, not fine grained locks.

Parallel semantics



Thread 0

S.add(x)

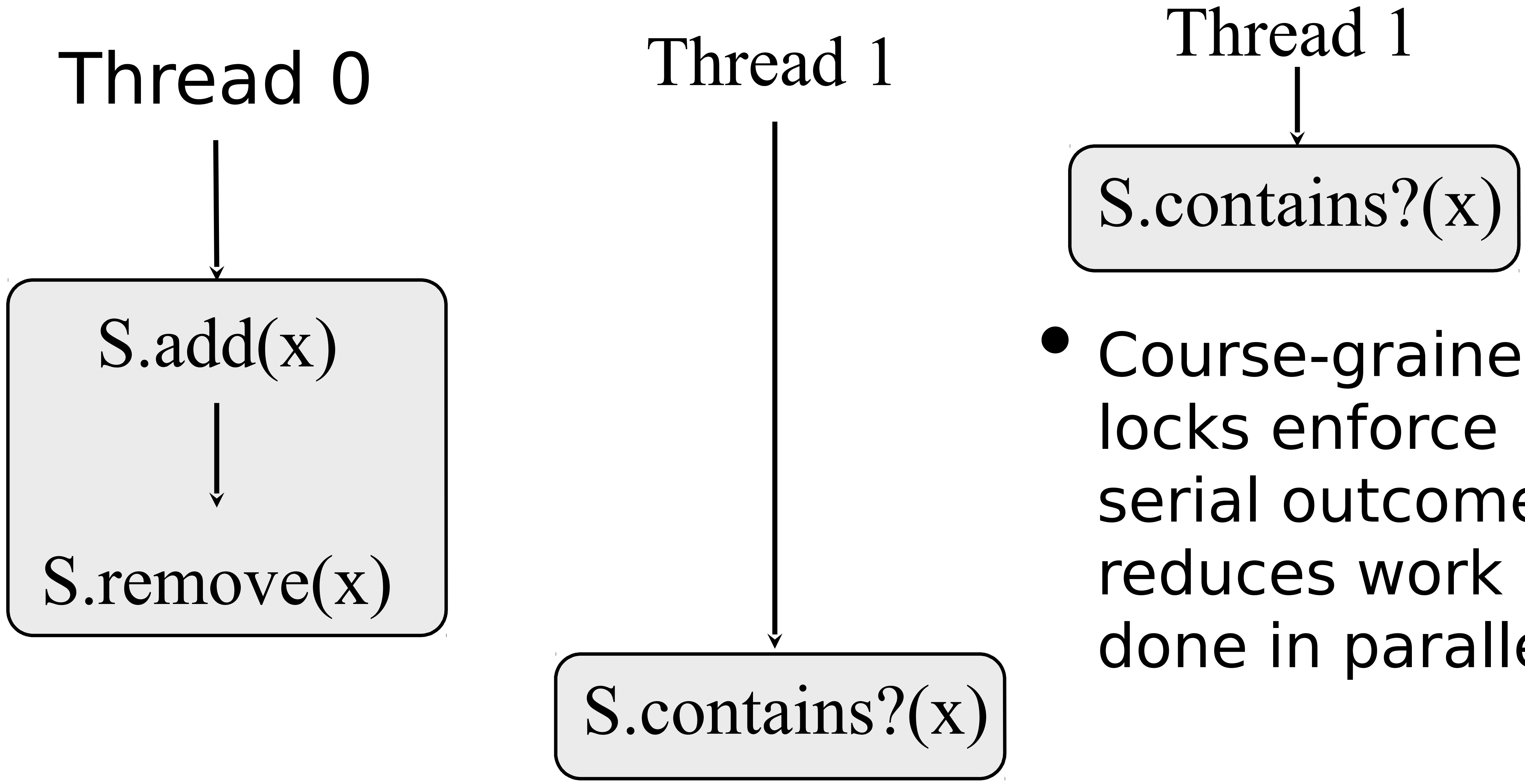
S.remove(x)

Thread 1

S.contains?(x)

- Fine-grained locks allow more work to happen in parallel
- Allows a non-serial outcome.

Parallel semantics



Thread 0

S.add(x)

S.remove(x)

Thread 1

S.contains?(x)

Thread 1

S.contains?(x)

- Course-grained locks enforce serial outcomes, reduces work done in parallel.

We want our cake and to eat it to -- concurrency + serial semantics

DeLaunay mesh w/set iterator

```
S1   Mesh m = /* read in initial mesh */
S2   Set w1;
S3   w1.add(mesh.badTriangles( ));
S4   for each e in w1 do {
S5       if (e no longer in mesh) continue;
S6       Cavity c = new Cavity(e);
S7       c.expand( );
S8       c.retriangulate( );
S9       m.update (c);
S10      w1.add(c.badTriangles( ));
     }
```

- Set elements can be picked by S4 in any order

1. Result must be as if body (S5 - S10) across different iterations executed serially *in some order*
2. Multiple loop bodies will likely execute in parallel

- Runtime forces 1 and 2 to be consistent

Specifying Abstract Data Type (ADT) properties

```
class Set {
  // interface methods
  void add (Element x);
    [commute]
      - add(y) {y != x}
      - remove(y) {y != x}
      - contains(y) {y != x}
   [inverse] remove(x);

  void remove (Element x);
   . . .
  void contains (Element x);
   [commute]
      . . .
      - contains(*) // any call to contains
   ...
 }
```

- Allows specification of *semantic* commutativity, and limitations
- *inverse* operation to be used when computation needs to be undone (discussed later

Galois library classes

- Galois objects, like Java objects, have a lock associated with them
- Galois uses these locks to support two kinds of classes:
  - Catch-and-keep (default)
  - Catch-and-release
- Different classes have different rollback policies
- We will explain these now

Catch-and-keep classes

- A form of two phase locking strategy
  - Phase 1 -- locks are acquired, and number of locks only increases or stays the same
  - Phase 2 -- locks are released, and number of locks only decreases or stays the same
  - Cannot, e.g., lock A, lock B, release B, lock C
  - Can do work between locks
- Objects copied before lock on that object is acquired
  - If a lock cannot be obtained, there is a conflict with another iteration, and the iteration is rolled back
  - Rollback accomplished by using copy of possibly modified objects

Catch-and-release classes

- Locking is *not* two-phase, locks can be acquired and released
  - Can, e.g., lock A, lock B, release B, lock C, release C, release A
  - Lock release allows interleaving of method executions in different threads
- Raises serializability issues -- which objects can be interleaved?
  - Commutative method calls are allowed to interleave
  - Conflicts among non-commutative methods force rollback
- Rollback *cannot* use a copy of the object before the method started
  - This would enforce concrete commutativity, but we need semantic commutativity
  - Use of *inverse* functions supports this rollback

Galois runtime

- Runtime maintains *commit pool*
- Commit pool
  - Creates new iteration records to start an iteration
  - Performs callbacks to inverse methods when necessary
  - Performs commits based on *priorities* assigned in set
  - Decides when it is legal to commit an iteration and who to roll back
    - When two iterations conflict, rolls back the lowest priority one.
    - When no conflicts and priority constraints met, commits the iteration

Galois runtime

- Runtime maintains *conflict logs*
- Conflict logs used to detect conflicts and there is one per catch/release object
- When iteration i attempts to execute method_1 on an object
  - Checks logs for conflicting methods (i.e. methods that don't commute) on the same object.
  - If one found, abort process begins. If ok, add call to log and invoke method
- When an iteration i aborts or commits all of its log entries are removed

Galois performance

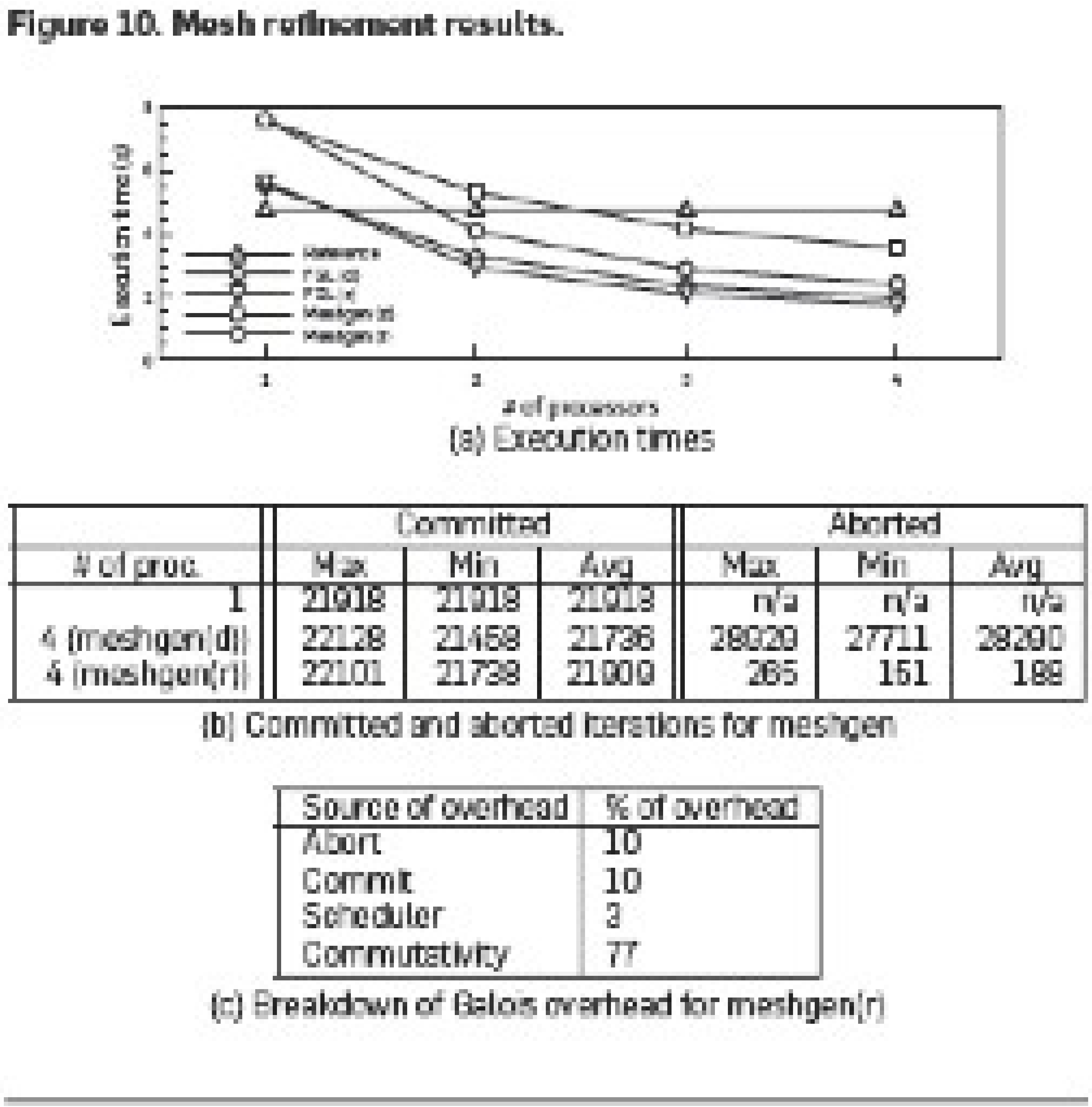


Figure 10. Mesh refinement results.

(a) Execution times

# of proc.	Committed Max	Committed Min	Committed Avg	Aborted Max	Aborted Min	Aborted Avg
1	21918	21918	21918	n/a	n/a	n/a
4 (meshgen(d))	22128	21458	21736	28929	27711	28290
4 (meshgen(r))	22101	21738	21909	265	151	188

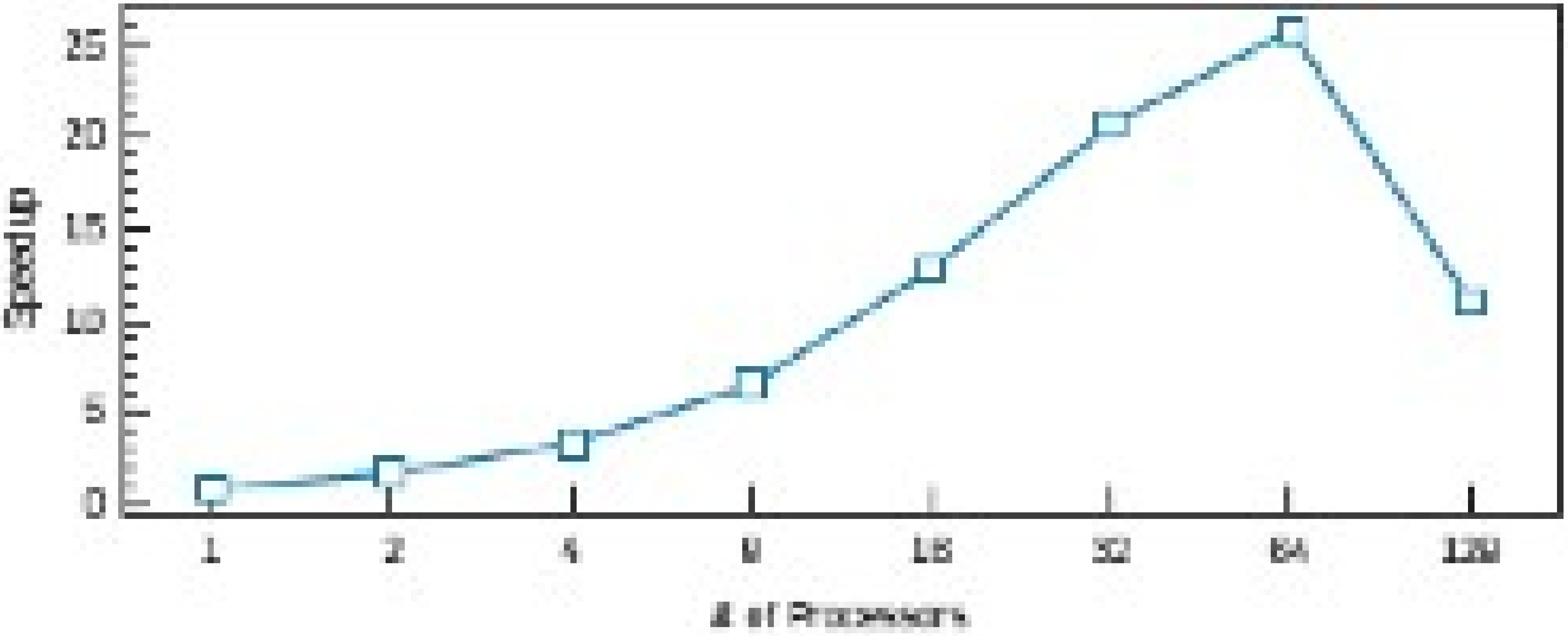
(b) Committed and aborted iterations for meshgen

Source of overhead	% of overhead
Abort	10
Commit	10
Scheduler	3
Commutativity	77

(c) Breakdown of Galois overhead for meshgen(r)

Galois performance

Figure 12. Speedup vs. # of processors for mesh refinement.



Summary

- Static compilation is insufficient for many programs
- Speculative techniques employing roll-back are useful
- Compiler/runtime and Language/compiler/runtime solutions are being studied
  - Both show promise
  - Language based solutions requires re-coding but has the potential to capture more information