# COMP 322: Fundamentals of Parallel Programming

# Lecture 27: Speculative parallelization of isolated blocks

Swarat Chaudhuri
Vivek Sarkar
Department of Computer Science, Rice University
swarat@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

# HJ isolated statement (Recap)

isolated <body>

- Two tasks executing isolated statements with interfering accesses must perform the isolated statement in mutual exclusion
  - Two instances of isolated statements, ⟨stmt1⟩ and ⟨stmt2⟩, are said to interfere with each other if both access a shared location, such that at least one of the accesses is a write.
  - ➔ Weak isolation guarantee: no mutual exclusion applies to non-isolated statements i.e., to (isolated, non-isolated) and (non-isolated, non-isolated) pairs of statement instances
- Isolated statements may be nested (redundant)
- Isolated statements must not contain any other parallel statement that performs a blocking operation: finish, get, next
  - Non-blocking operations (e.g., async) are fine

# Implementations of isolated statement

- isolated statements are convenient for the programmer but pose significant challenges for the language implementation
  - Implementation does not know ahead of time if two dynamic instances of isolated statements will interfere or not
- HJ implementation used in COMP 322 takes a simple single-lock approach to implementing isolated statements
  - Entry to isolated statement is treated as an acquire() operation on the lock
  - Exit from isolated statement is treated as a release() operation on the lock
  - Though correct, this approach essentially implements isolated statements as critical sections, thereby serializing all interfering and non-interfering isolated statement instances.
- How can we do better?

# Research Idea 1: Transactional Memory

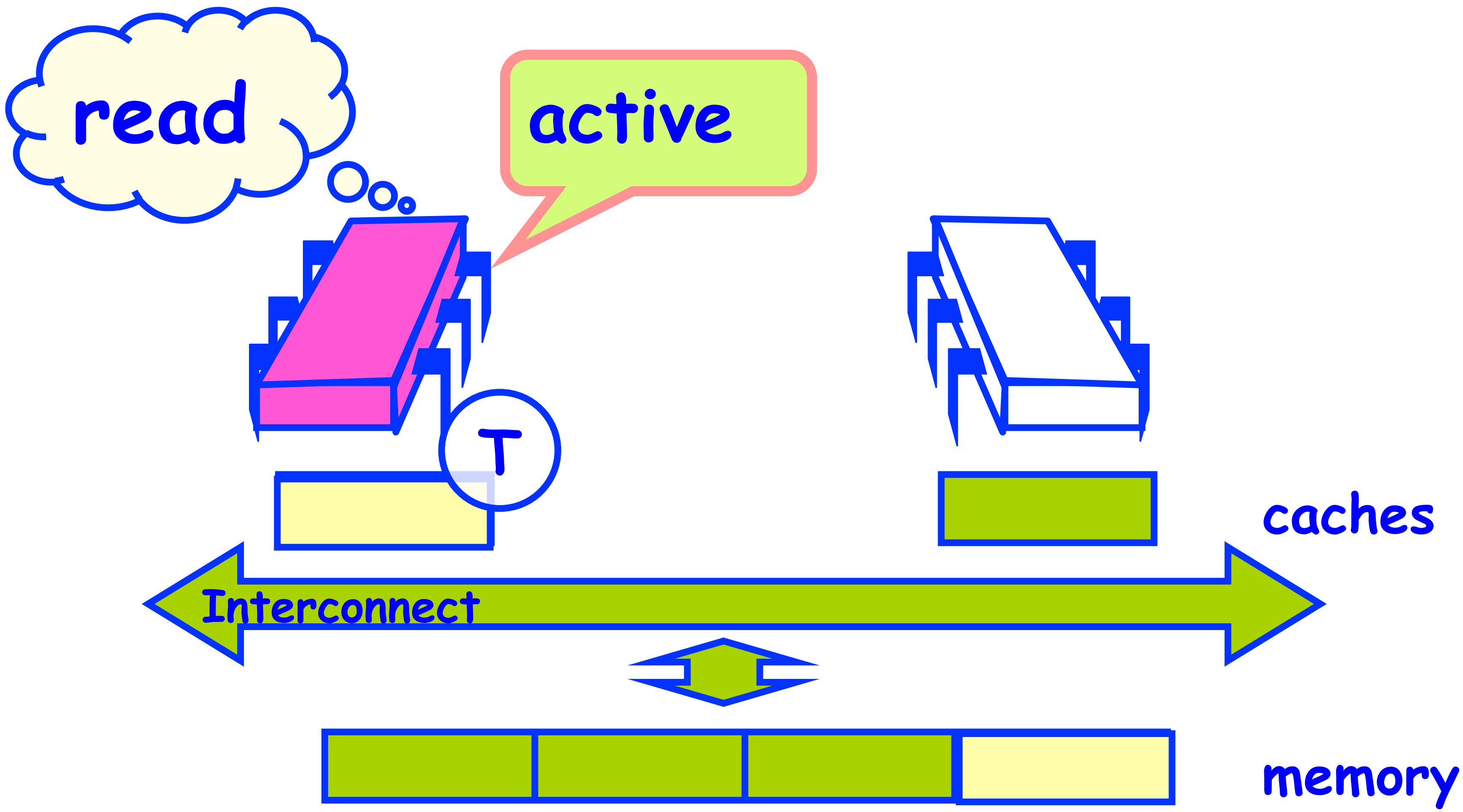
- Execution of an isolated statement is treated as a transaction
  - In database systems, a transaction refers to a "unit of work" that has "all-or-nothing" semantics. Each unit of work must either complete in its entirety or have no visible effect.
- A TM system optimistically permits transactions to run in parallel, speculating that there won't be interference
- At the end of a transaction, a TM system checks if interference occurred with another transaction
  - If not, the transaction can be committed
  - If so, the transaction fails and has to be "retried"
- Both software and hardware implementations of TM have been explored extensively by the research community, but no implementation has proved suitable for mainstream use as yet.

# Hardware Transactional Memory

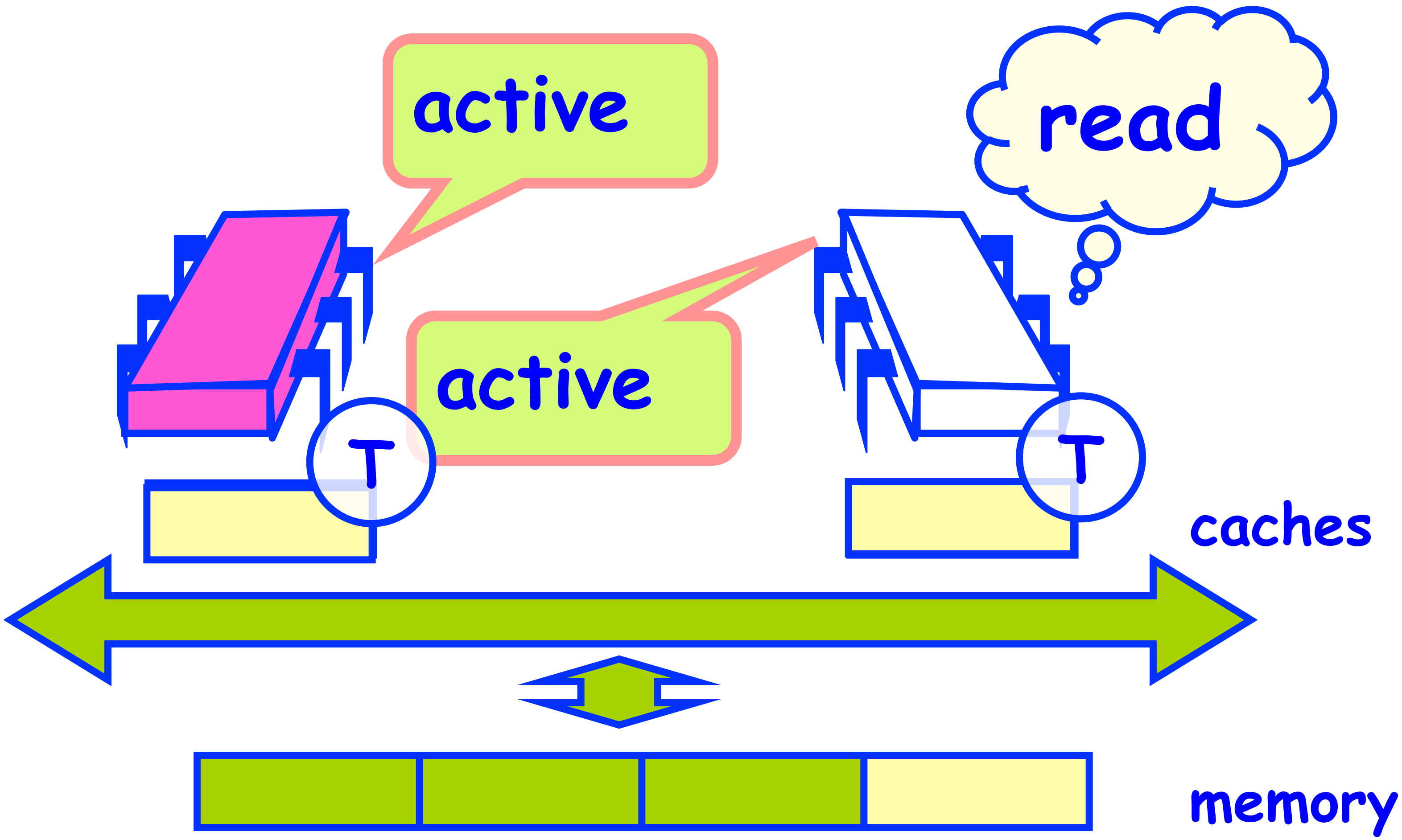
isolated <body>

- Exploit Cache coherence protocols
- Already do almost what we need
  - Invalidation
  - Consistency checking
- Exploit Speculative execution
  - Branch prediction = optimistic synch
- Related work:
  - First wave: Herlihy&Moss 93, Stone et al. 93
  - Second wave: Rajwar&Goodman 02, Martinez&Torellas 02, Oplinger&Lam 02, TCC 04, VTM 05, …

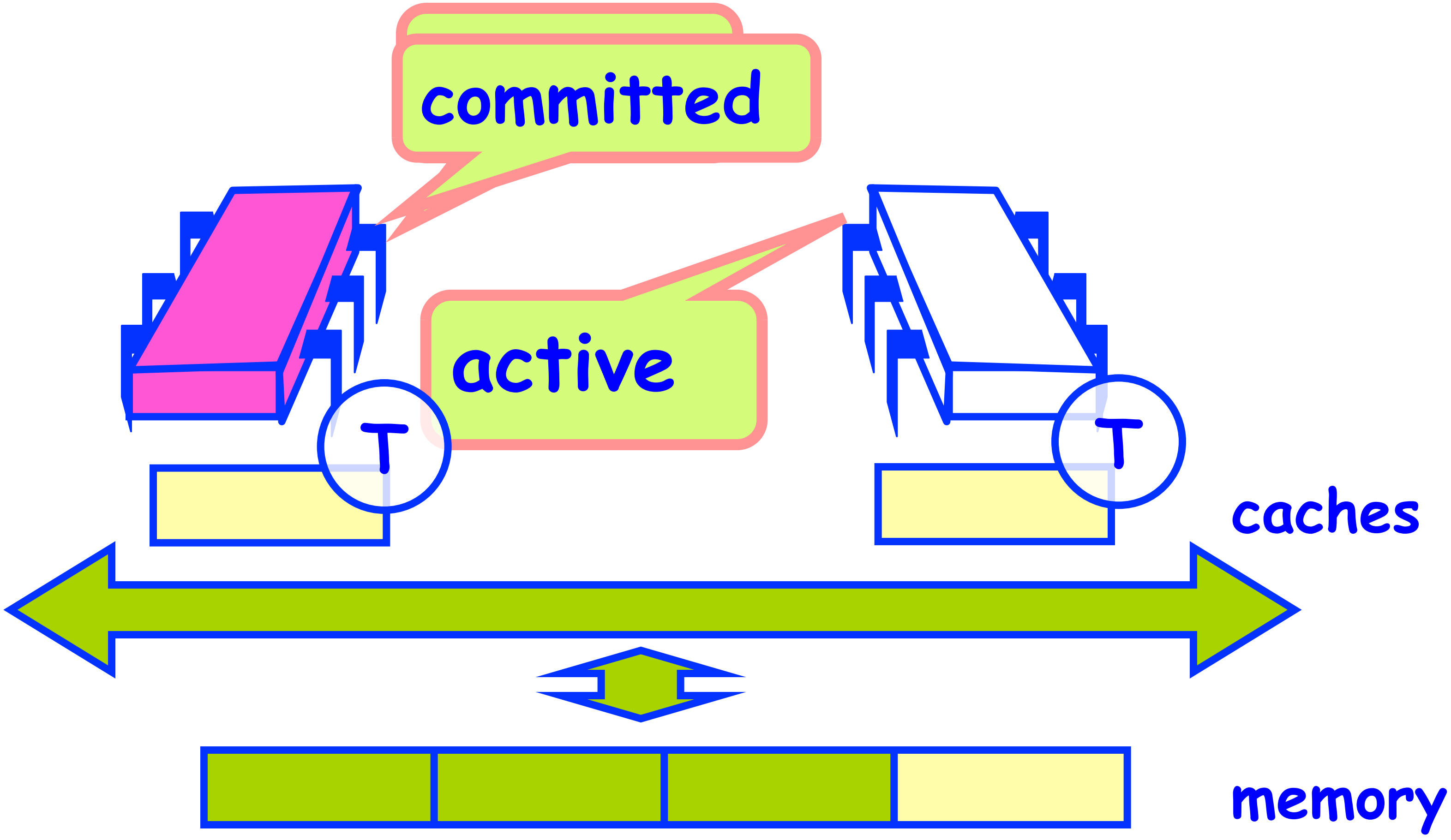
# HW Transactional Memory



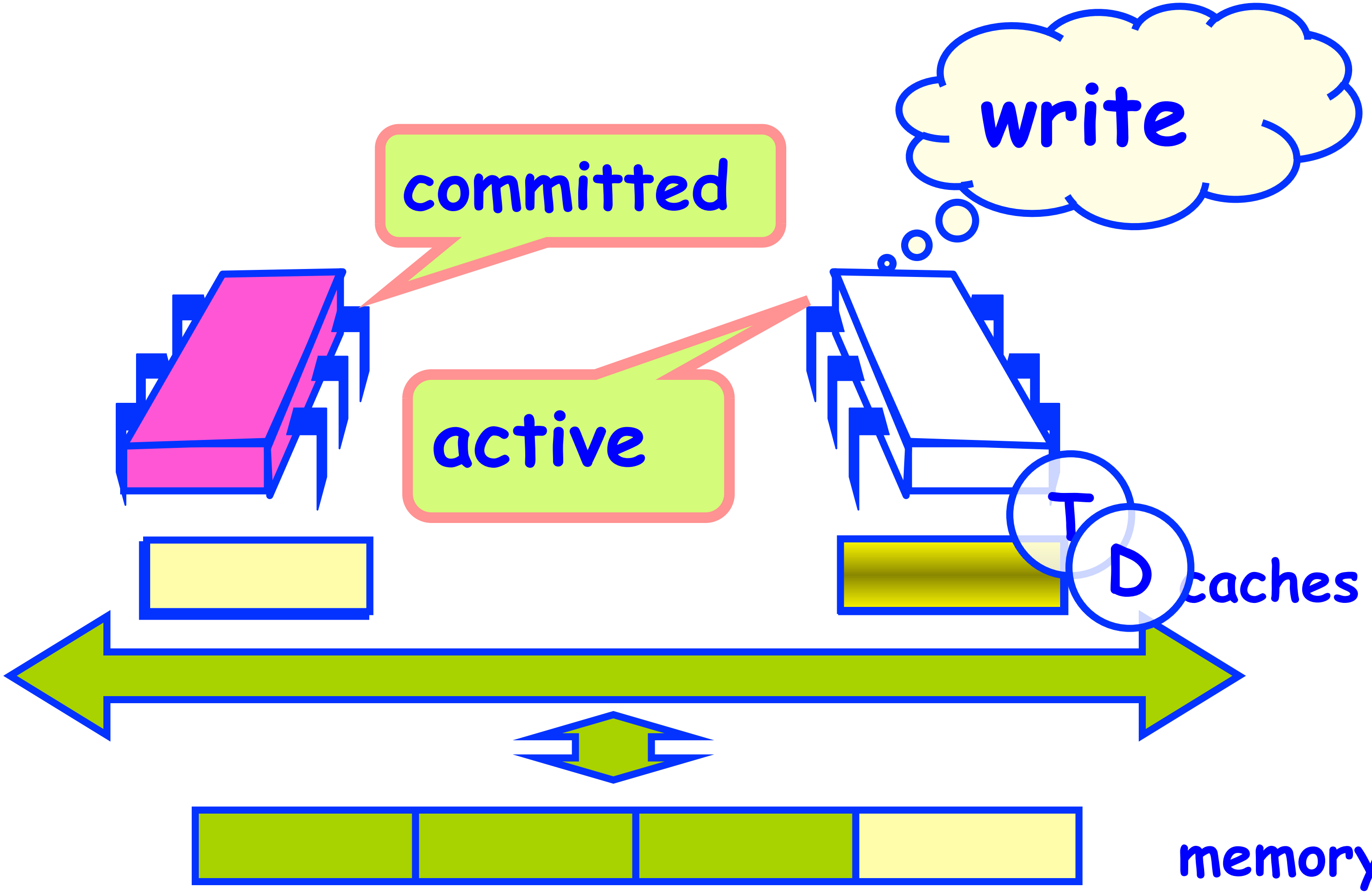
# Transactional Memory



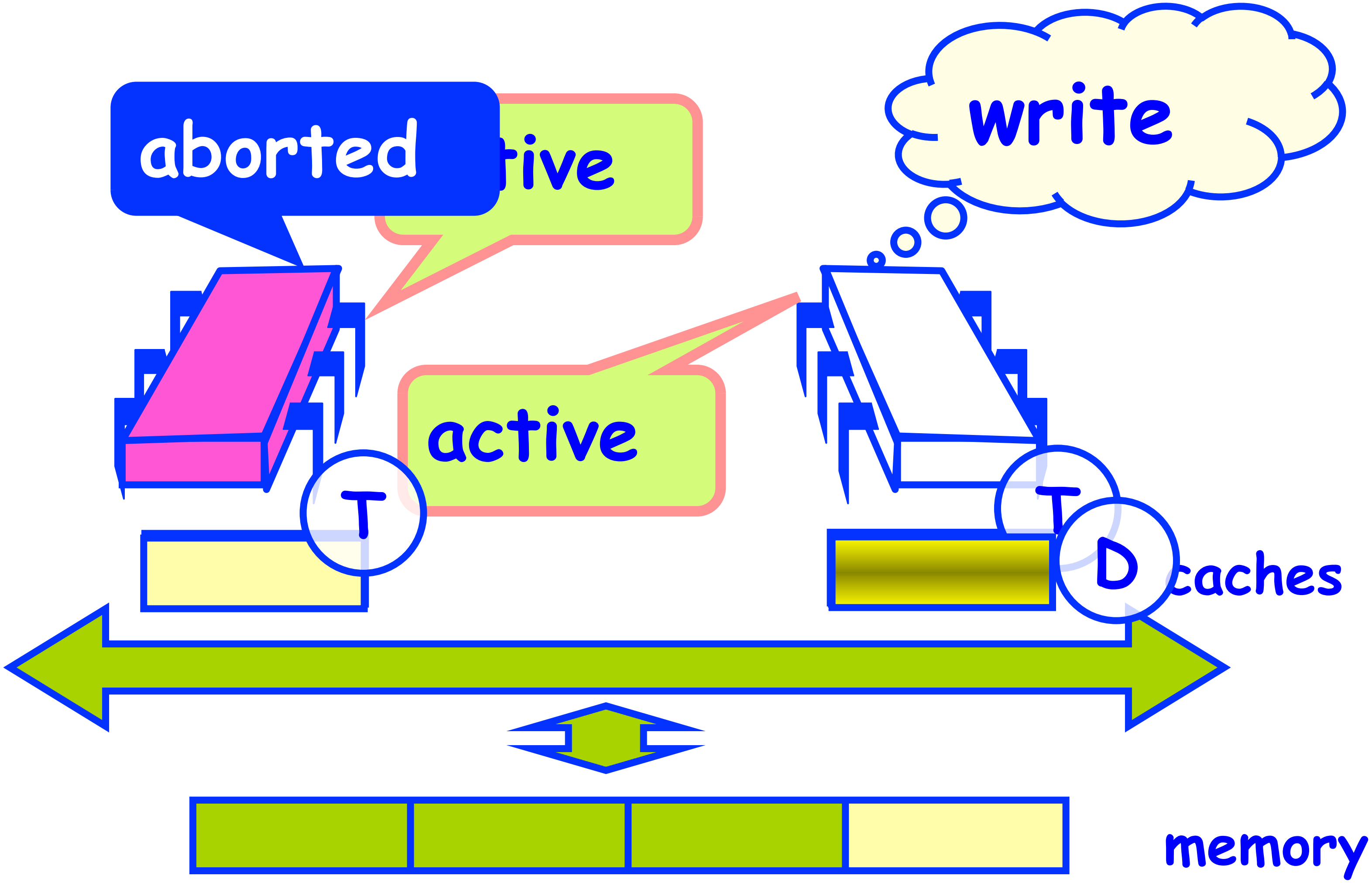
# Transactional Memory



# Transactional Memory



# Rewind



# Transaction Commit

- At commit point
  - If no cache conflicts, we win.
- Mark transactional entries
  - Read-only: valid
  - Modified: dirty (eventually written back)
- Challenges:
  - Limits to
    - Transactional cache size
    - Scheduling quantum
  - Transaction cannot commit if it is
    - Too big
    - Too slow
    - Actual limits platform-dependent

# Software TMs (e.g., DSTM)

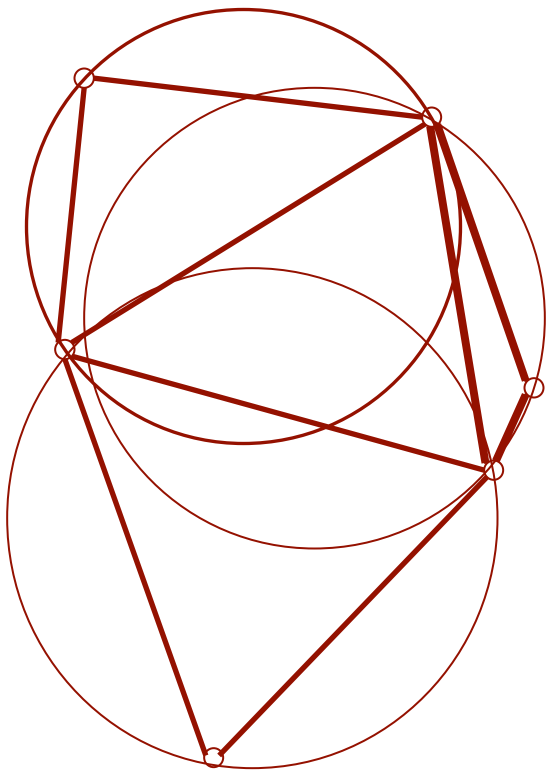
- Logs all read and write operations performed in a transaction. Implements conflict detection and aborts in software
- Minimal hardware support: compare-and-swap is enough
- Example implementation questions:
  - Do zombie (orphan) transactions see consistent states?
  - Undo or redo?
    - Undo logs

      Update in place; Reads are fast; Rolling back wedged transaction complex
    - Redo logs

      Apply changes on commit; Reads require look-aside; Rolling back wedged transaction easy
  - Does interference detection need a global view of the heap?
- Especially challenging: irregular applications, where parallelism depends heavily on the input

# Irregular parallelism: Delaunay Mesh Refinement

- Input: a 2d triangle mesh that satisfies:
  - the Delaunay property: no point is contained in the circumcircle of a triangle
- Output: a 2d triangle mesh that
  - —satisfies the Delaunay property
  - —contains all points in the original mesh
  - —satisfies an extra quality constraint
    - – no triangle can have an angle $< 25°$
- Algorithm (Ruppert's algorithm)
  - —iteratively select a triangle that violates the quality constraint and refine the mesh around it.



# DMR Algorithm (Sequential and HJ)

```
Mesh m = /* read input mesh */
Worklist wl = new worklist(m.getBad());
foreach triangle t in wl {
   if (t in m) {
      Cavity c = new Cavity(t)
       c.expand()
       c.retriangulate(m)
       wl.add(c.getBad()); } }
```

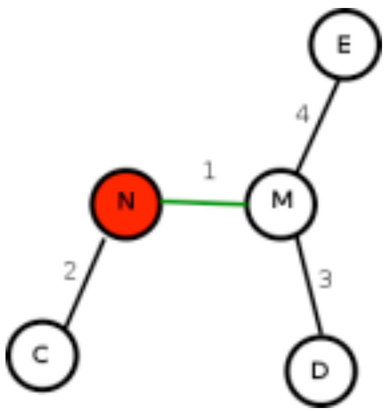
*Sequential*

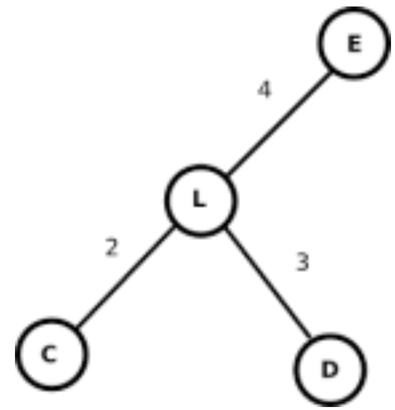
```
...
foreach triangle t in wl {
  isolated {
     if (t in m) {
     Cavity c = new Cavity(t);
      c.expand();
      c.retriangulate(m);
      wl.add(c.getBad());} }}
```

*With isolated construct*

# Another example: Boruvka's MST algorithm

Before contraction




After contraction

```
Graph g = ...
Forest mst = g.getNodes();
Workset ws = g.getNodes();
foreach Node n in ws {
    Node m = minWeight(n, g.getOutEdges(n));
    Node l = edgeContract(n, m);
    mst.addEdge(n, m);
    ws.add(l);
}
```

# Research Idea 2: Delegated Isolation

- Challenge: scalable implementation of isolated without using a single global lock and without incurring transactional memory overheads
- Delegated isolation:
  - Restrict attention to "async isolated" case
    - replace non-async "isolated" by "finish async isolated"
  - Task dynamically acquires ownership of each object accessed in isolated block (optimistic parallelism)
  - On conflict, task A transfers all ownerships to worker executing conflicting task B and delegates execution of isolated block to B (Chorus execution model)
  - Deadlock-freedom and livelock-freedom guarantees

  - Reference: "Delegated Isolation", R. Lublinerman, J. Zhao, Z. Budimlic, S. Chaudhuri, V. Sarkar, OOPSLA 2011

# The Aida execution model

Heap =
  directed graph

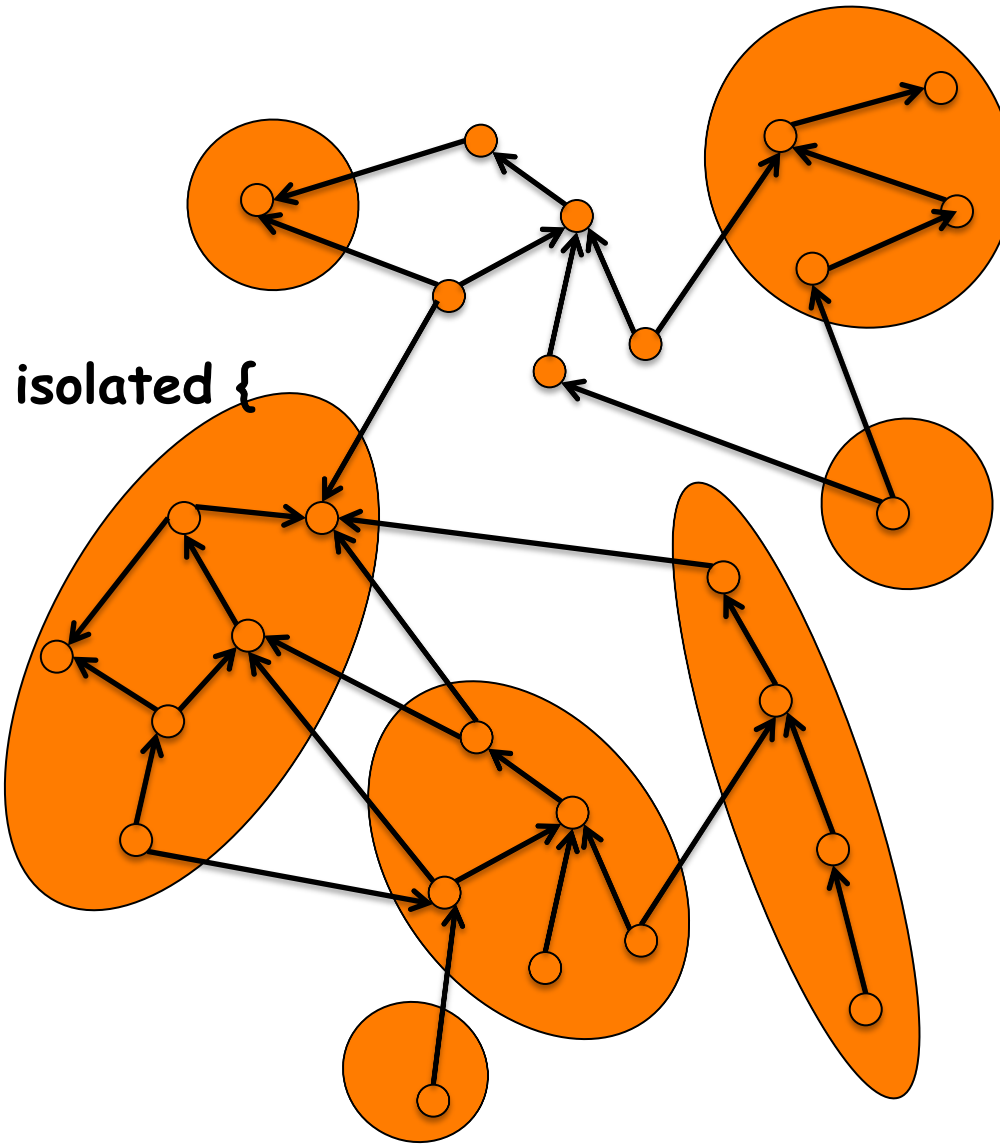
Nodes =
  memory locations

Labeled edges =
  pointers

Regions =
  subgraphs induced by a partitioning

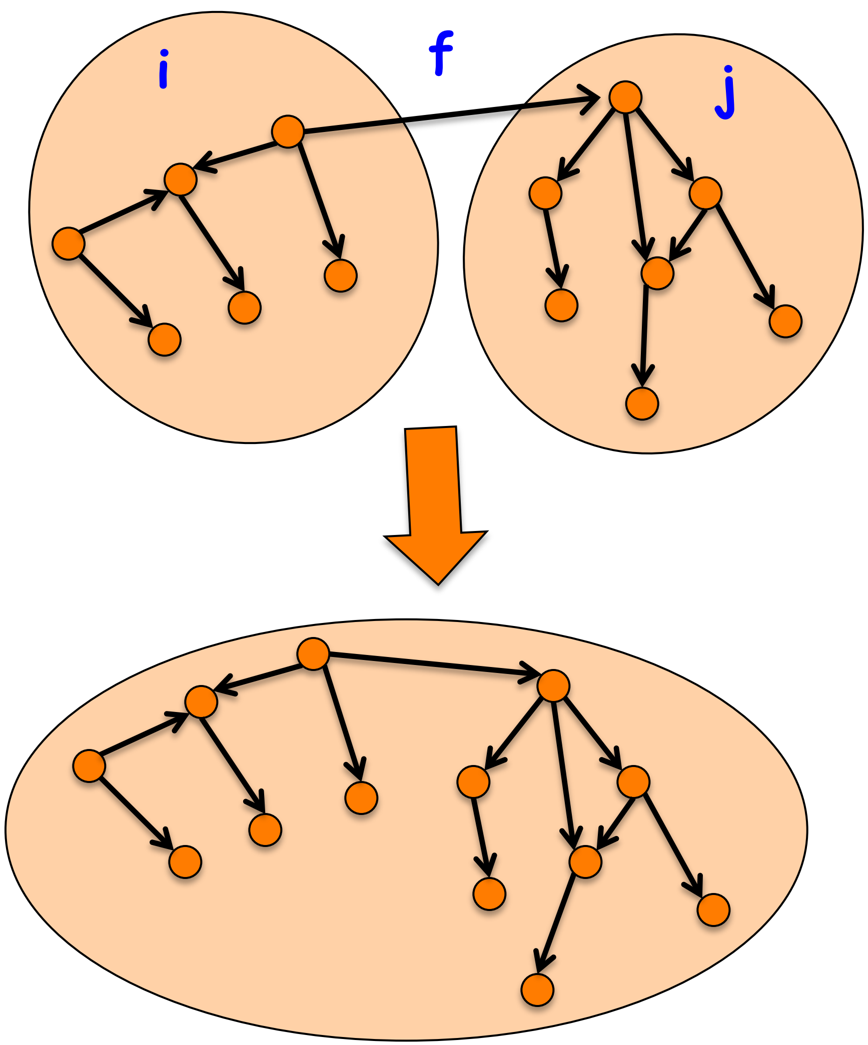
Assembly =
  task + owned region

```
async isolated {
  ...
}
```



An assembly can only access objects that it owns

# Conflict management: merging




- Assembly i merges with assembly j along an edge f
- Delegation:
  - j keeps local state
  - i dies passing closure to j. Effects of i rolled back
- Alternative: preemption (i keeps local state, j gets killed. More difficult to implement.
- Guarantees aside from isolation:
  - Deadlock-freedom
  - Progress: For each conflict, at least one commit

# DMR Algorithm (Delegated isolation)

```
processTriangle (Triangle t) {
   async isolated {
     if (t in m) {
       Cavity c = new Cavity(t);
        c.expand();
        c.retriangulate();
        for (s in c.badTriangles());
          processTriangle (s); } } }

main () {
 finish {
    for (t in initial set of bad triangles)
       processTriangle (t);
 }
}
```

# Delauney Mesh Refinement in Habanero-Java using Delegated Isolation

```
void doCavity(Triangle start) {
  async isolated {
    if (start.isActive()) {
      Cavity c = new Cavity(start);
      c.initialize(start);
      c.retriangulate();

      // launch retriagnulation on new bad triangles.
      Iterator bad = c.getBad().iterator();
      while (bad.hasNext()) {
        final Triangle b = (Triangle)bad.next();
        doCavity(b);
      }

      // if original bad triangle was NOT retriangulated,
      // launch its retriangulation again
      if (start.isActive())
        doCavity(start);
    }
  } // end isolated
}

void main() {
  mesh = ... ; // Load from file
  initialBadTriangles = mesh.badTriangles();
  Iterator it = initialBadTriangles.iterator();
  finish {
    while (it.hasNext()) {
      final Triangle t = (Triangle) it.next();
      if (t.isBad())
        Cavity.doCavity(t);
    }
  }
}
```

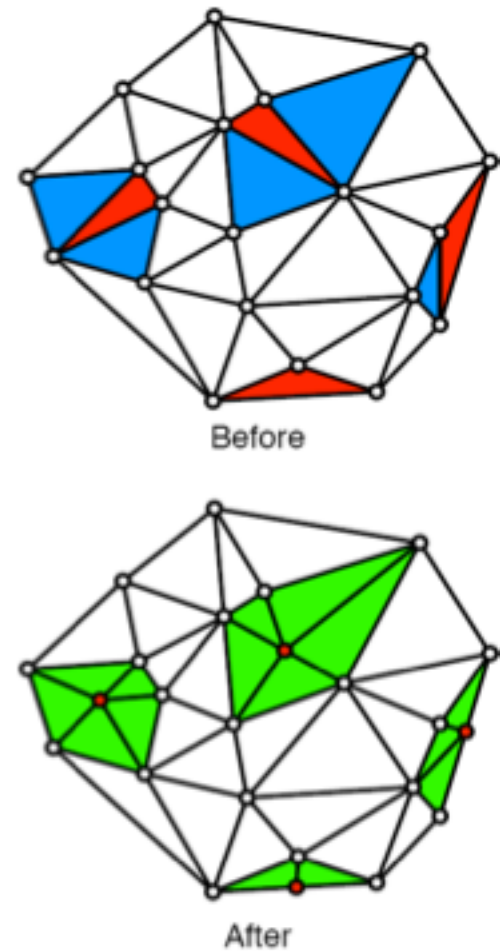
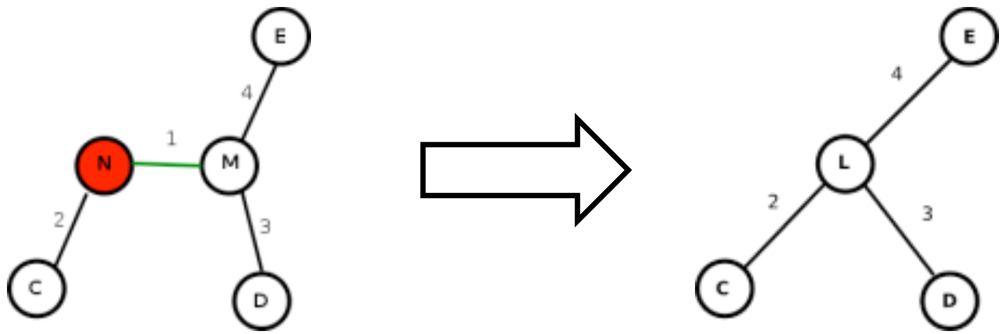



Figure source:
http://lcpc10.rice.edu/Keynote_Speakers_files/PingaliKeynote.pdf

# Boruvka's MST algorithm

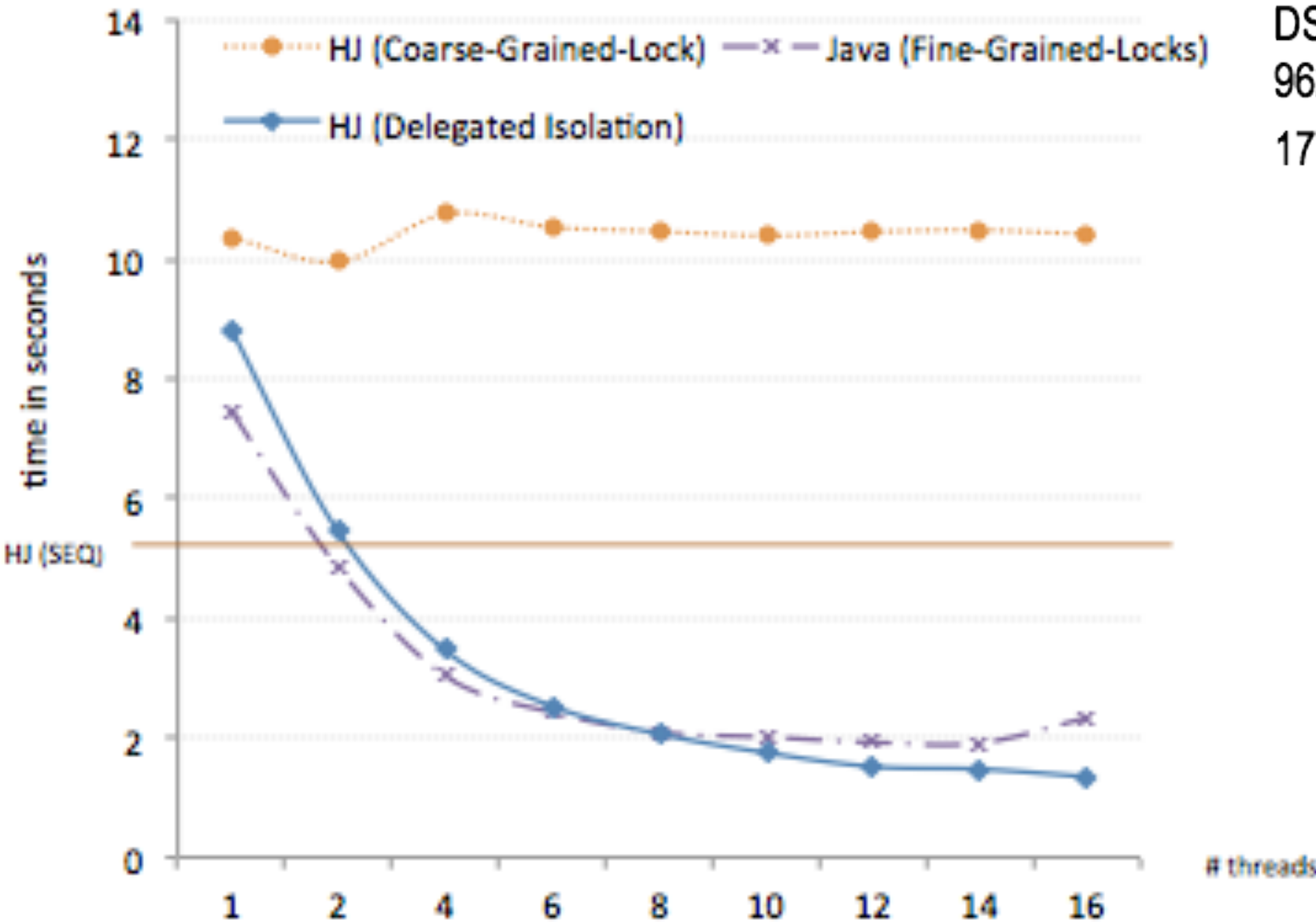



```
processTree (Node n) {
   async isolated {
        Node m = minWeight(n, g.getOutEdges(n));
        Node l = edgeContract(n, m);
        l.mst.addEdge(n, m);
        processTree(l); }

main () {
   finish {
     for nodes n
          processTree(n); } }
```

# Performance: DMR benchmark on 16-core Xeon SMP

(100,770 initial triangles of which 47,768 are “bad”; average # retriangulations is ~ 130,000)




DSTM2 performance:

962s w/ 1 thread

177s w/ 16 threads

# Properties of isolated statements

## How small or big should an isolated statement be?

- Too small → may lose invariants desired from mutual exclusion
- Too big → limits parallelism

## Deadlock freedom guarantees

- Observation: no combination of the following HJ constructs can create a deadlock cycle among tasks
  - finish, async, get, forall, next, isolated
- There are only two HJ constructs that can lead to deadlock
  - async await (data-driven tasks)
  - explicit phaser wait operation (instead of next)

# Three cases of contention among isolated statements

1. Low contention: when isolated statements are executed infrequently
   – A single-lock approach as in HJ is often the best solution. No visible benefit from other techniques because they incur overhead that is not needed since contention is low.
2. Moderate contention: when the serialization of all isolated statements in a single-lock approach limits the performance of the parallel program due to Amdahl's Law, but a finer-grained approach that only serializes interfering isolated statements results in good scalability
   – Atomic variables usually do well in this scenario since the benefit obtained from reduced serialization far outweighs any extra overhead incurred.
3. High contention: when interfering isolated statements dominate the program execution time in certain phases
   – Best approach in such cases is to find an alternative algorithm to using isolated