# COMP 322: Fundamentals of Parallel Programming

# Lecture 20: Speculative parallelization of isolated constructs


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


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

# HJ isolated construct (Recap)

isolated (() -> <body> );

- Isolated construct identifies a critical section
- Two tasks executing isolated constructs must perform them in mutual exclusion
  - ➔Isolation guarantee applies to (isolated, isolated) pairs of constructs, not to (isolated, non-isolated) pairs of constructs
- Isolated constructs may be nested
  - — An inner isolated construct is redundant
- Blocking parallel constructs are forbidden inside isolated constructs
  - —Isolated constructs must not contain any parallel construct that performs a blocking operation e.g., finish, future get, next
  - —Non-blocking async operations are permitted, but isolation guarantee only applies to creation of async, not to its execution
- Isolated constructs can never cause a deadlock
  - — Other techniques used to enforce mutual exclusion (e.g., locks) can lead to a deadlock, if used incorrectly

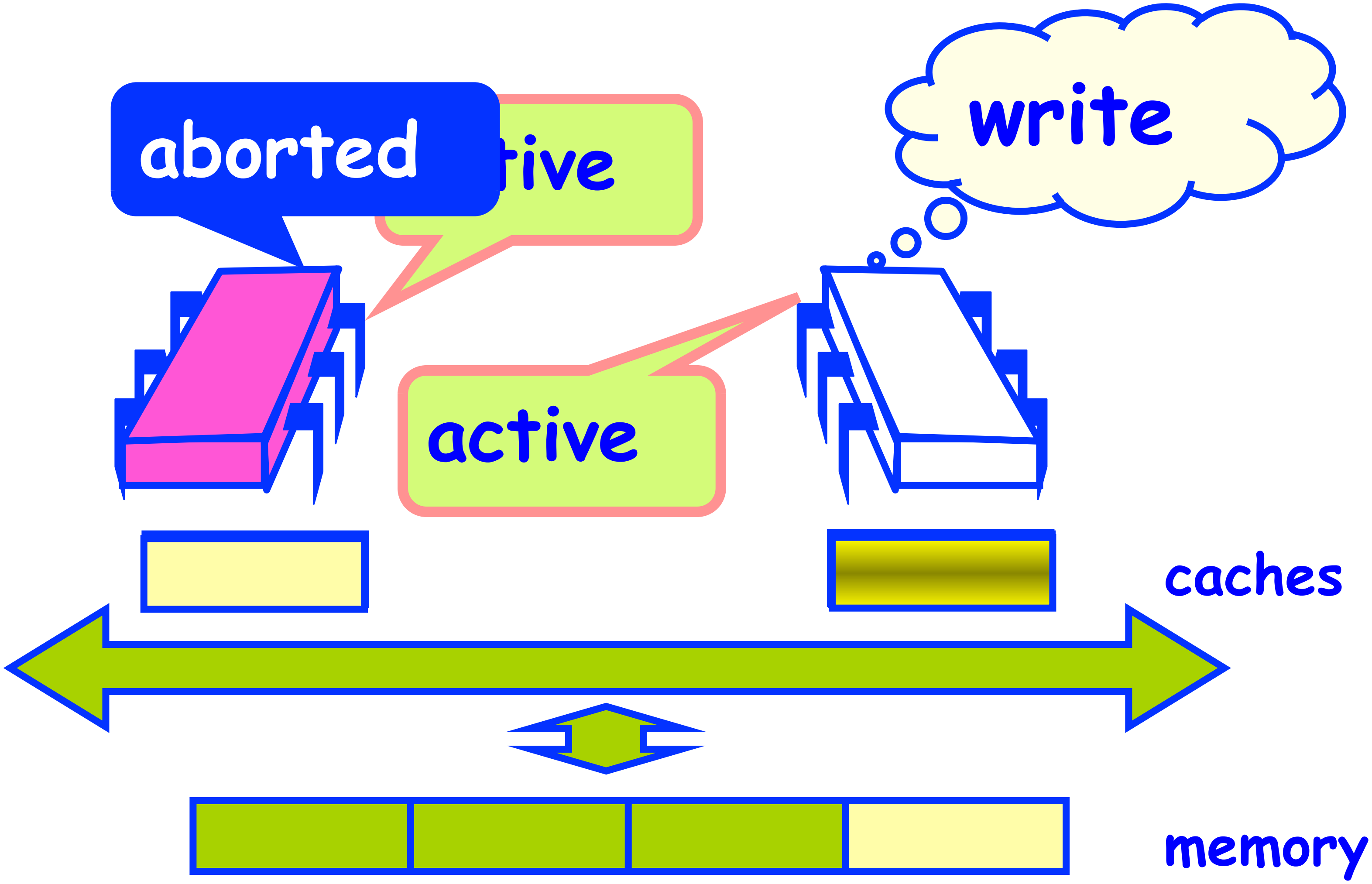
# Implementations of isolated construct

- isolated constructs are convenient for the programmer but pose significant challenges for the language implementation
  - Implementation does not know ahead of time if two parallel instances of isolated constructs will perform conflicting accesses on a shared location
- Naive implementation: allocated a single "lock"
  - Only one async can enter an isolated construct at a time
  - No differentiation between isolated and object-based isolated
- HJ-lib implementation: allocate a set of "locks"
  - Use hashcode to map from objects to locks
  - Global isolated construct waits to acquire all locks
  - Object-based isolated construct only acquires locks corresponding to the objects in its list
- How can we do better?

# Research Idea 1: Transactional Memory

- Execution of an isolated construct 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 any conflicts
- At the end of a transaction, a TM system checks if a conflict occurred with another transaction
  - If not, the transaction can be committed
  - If so, the transaction fails (aborts) and has to be "retried"
- Both software and hardware implementations of TM have been explored extensively by the research community, but no implementation has achieved mainstream success as yet.

# Transactional Memory Scenario

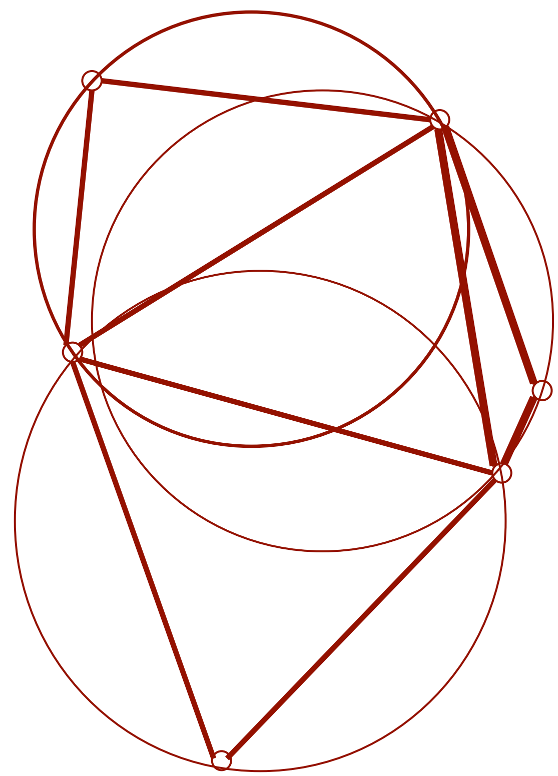


# Irregular parallelism: Delaunay Mesh Refinement (DMR)

- 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 pseudocode)

```
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()); } }
```

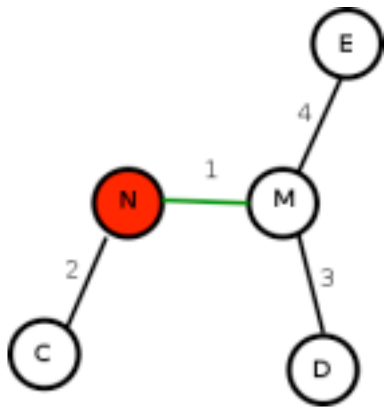
*Sequential version*

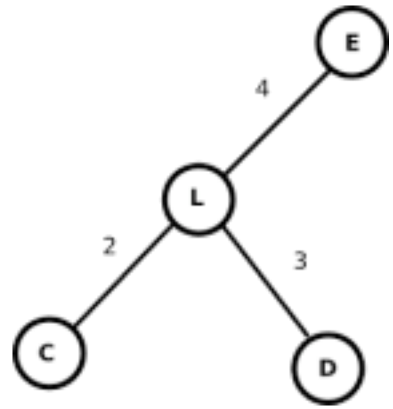
```
...
finish foreach triangle t in wl {
   async isolated {
      if (t in m) {
      Cavity c = new Cavity(t);
       c.expand();
       c.retriangulate(m);
       wl.add(c.getBad());}
   }}
```

*Parallel version with isolated construct*

# Another example: Boruvka's Minimum Spanning Tree (MST) algorithm

Before contraction




After contraction

```
Graph g = ...
Forest mst = g.getNodes();
Workset ws = g.getNodes();
finish foreach Node n in ws
  async isolated {
    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

  - References:
    - "Delegated Isolation", R. Lublinerman, J. Zhao, Z. Budimlic, S. Chaudhuri, V. Sarkar, OOPSLA 2011
    - "Isolation for Nested Task Parallelism" J. Zhao, R. Lublinerman, Z. Budimlic, S. Chaudhuri, V. Sarkar, OOPSLA 2013.

# The Aida execution model

Heap =
  directed graph

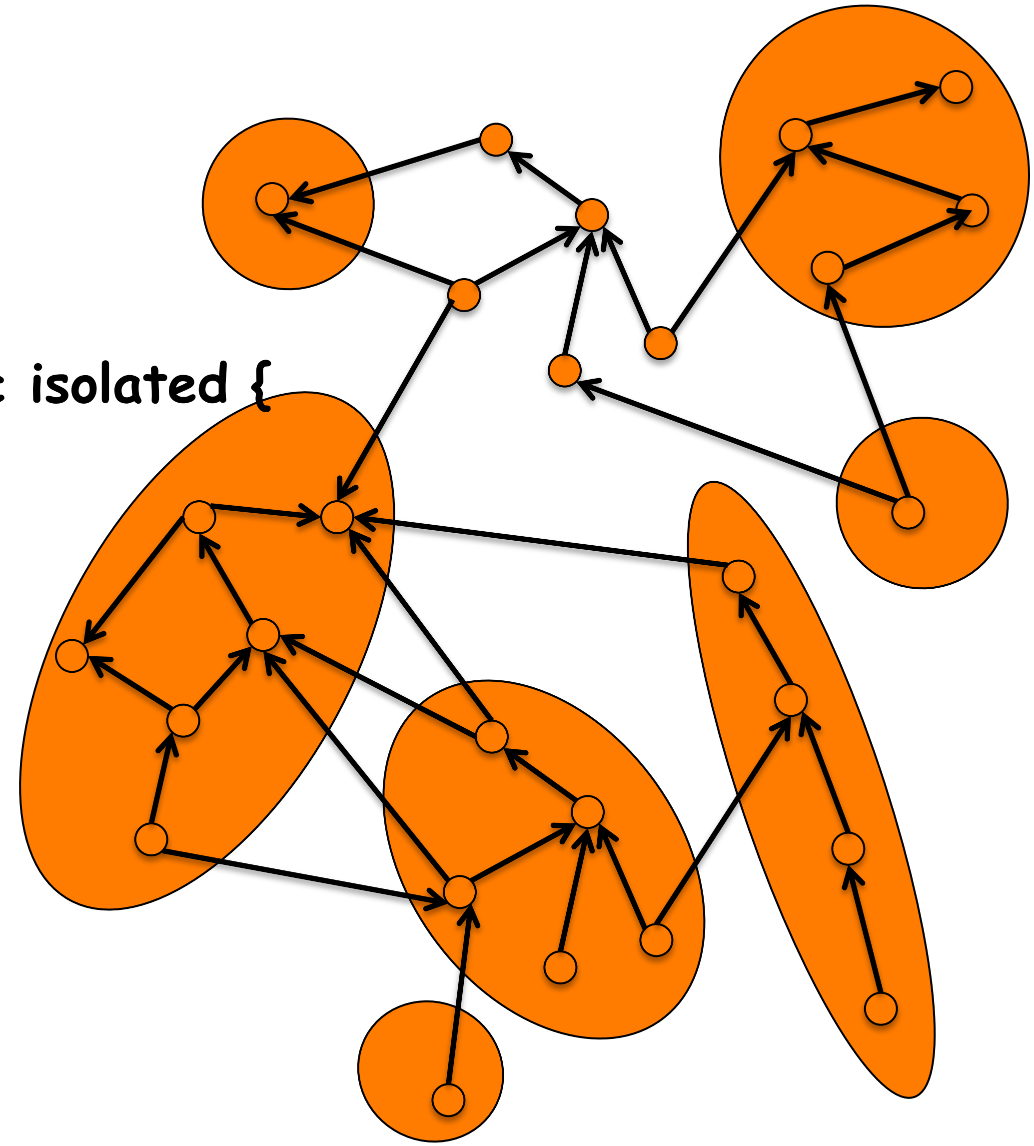
Nodes =
  memory locations

Labeled edges =
  pointers

Regions =
  subgraphs induced by a partitioning

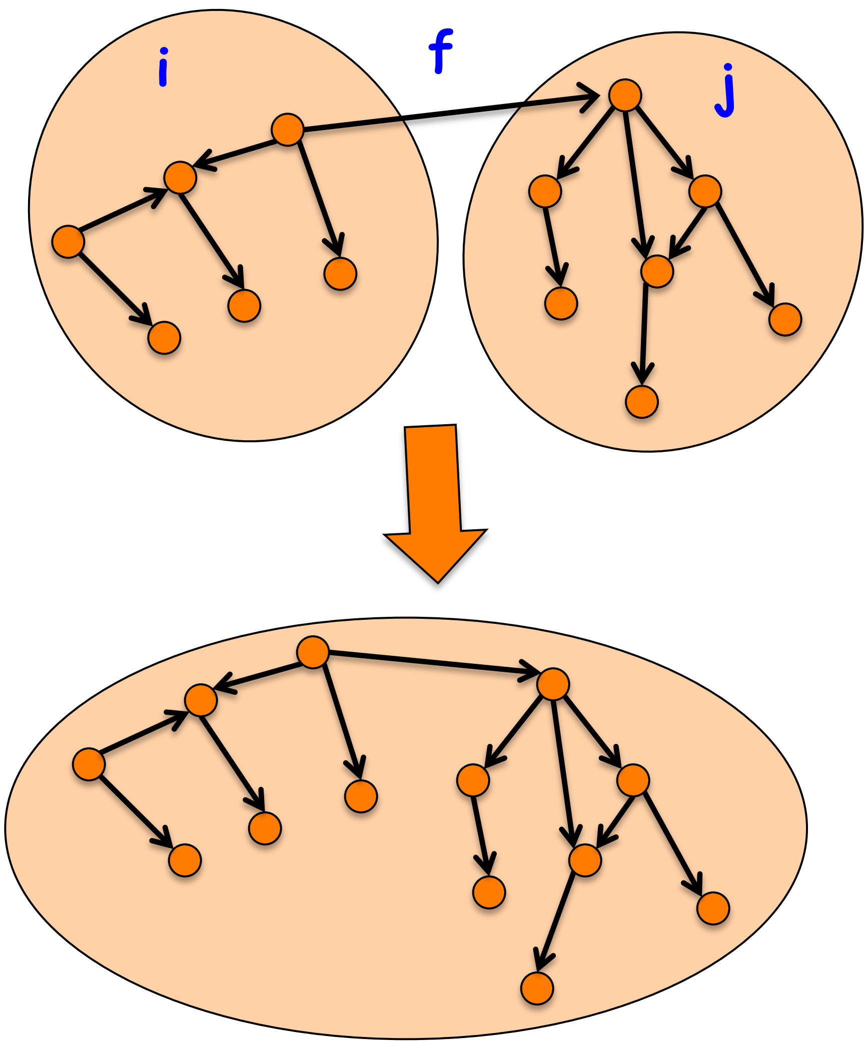
Assembly =
  task + owned region

```
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);
    }
  }
}
```

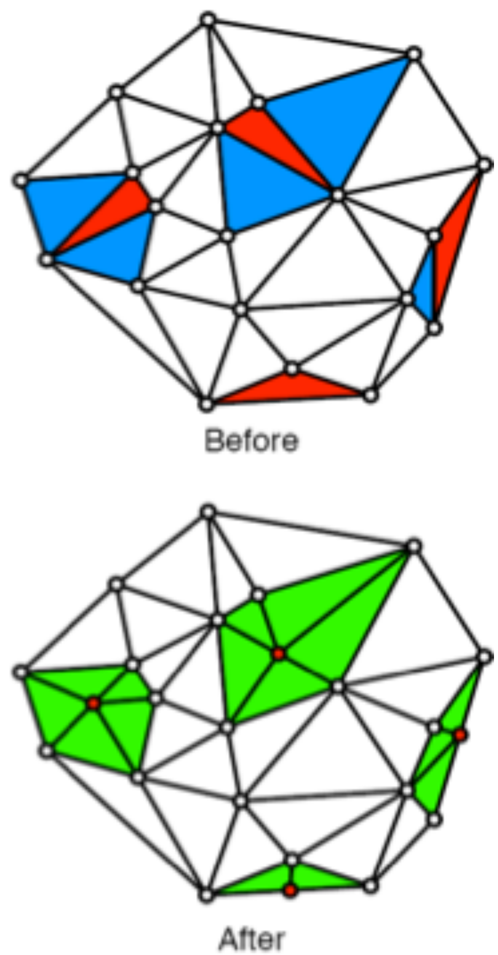
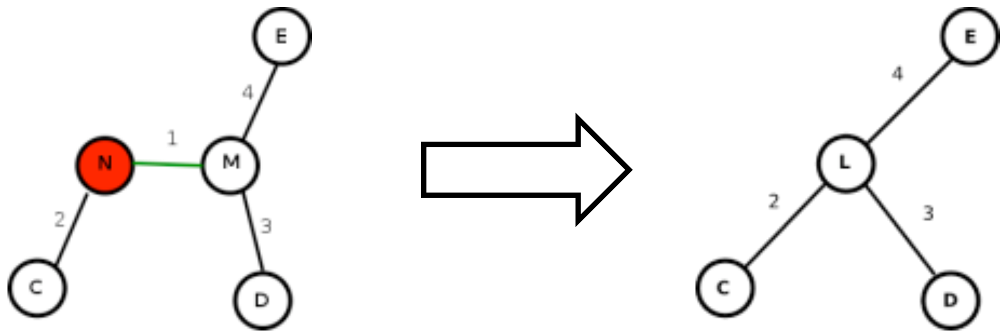



Figure source:
http://lcpc10.rice.edu/Keynote_Speakers_files/PingaliKeynote.pdf

# Boruvka's MST algorithm

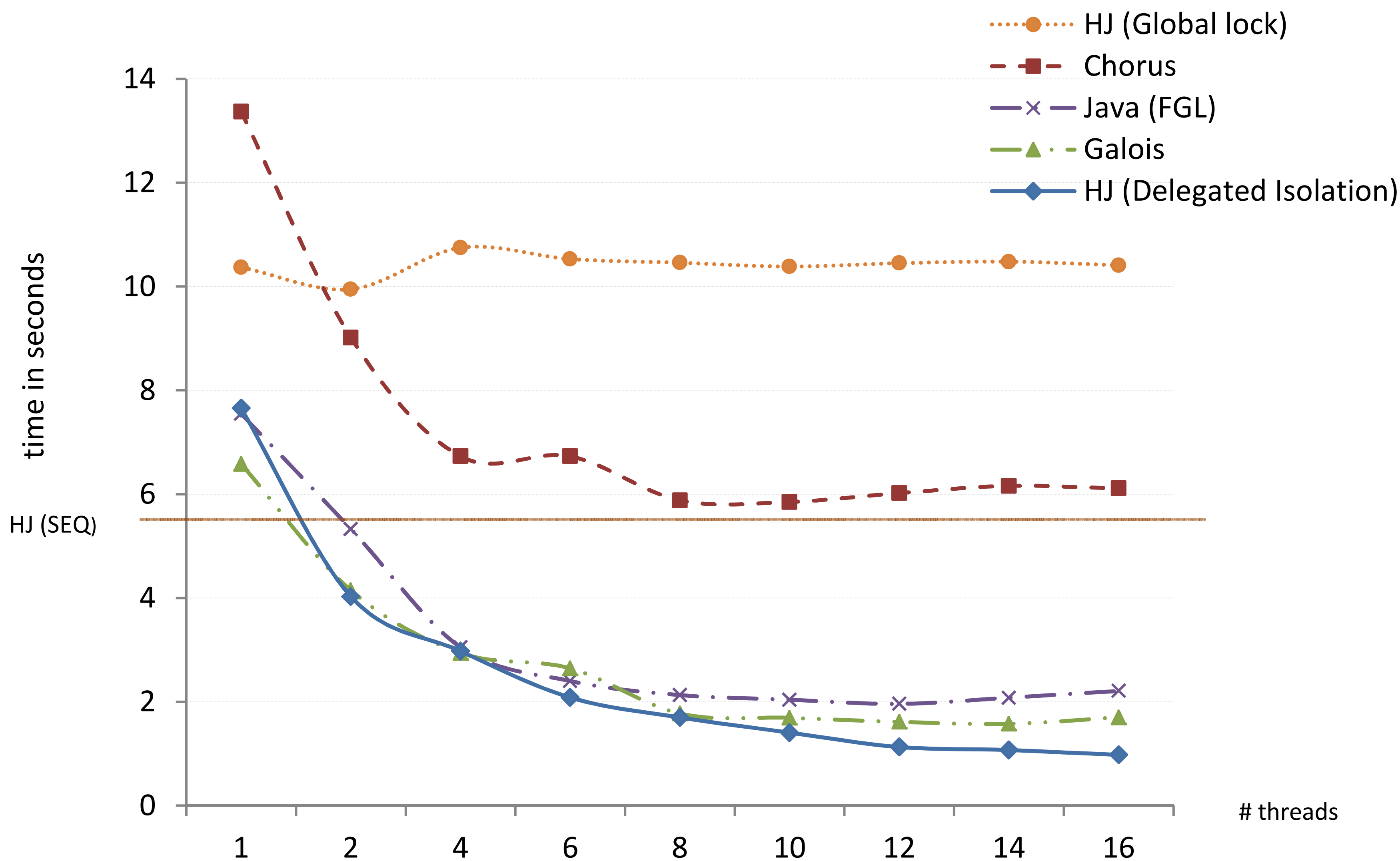



```
processTree (Node n) {
  async isolated {
      Node m = minWeight(n, g.getOutEdges(n));
      Node l = edgeContract(n, m);
      l.mst.addEdge(n, m);
      processTree(l); }

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)



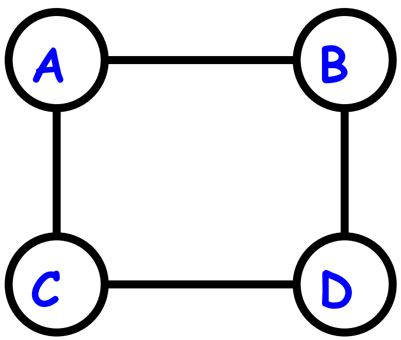
# Three cases of contention among isolated constructs

1. **Low contention: when isolated constructs 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 constructs 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 conflicting isolated constructs results in good scalability**
   – Object-based isolation and "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 conflicting isolated constructs dominate the program execution time in certain phases**
   – Best approach in such cases is to find an alternative approach or algorithm to using isolated e.g., use of accumulators or parallel prefix sum algorithm for reductions

# Worksheet #20: Identifying conflicts in isolated constructs

Name: ______________________ Netid: ______________________

Consider the Parallel Spanning Tree algorithm discussed in the last lecture (and shown below in slide 18). Assume that the isolated construct is implemented using a Transactional Memory mechanism. Outline a parallel execution scenario for the input graph below that could lead to a conflict between isolated constructs.



# Parallel Spanning Tree Algorithm

```
class V  {
  V [] neighbors; // adjacency list for input graph
  V parent; // output value of parent in spanning tree
  boolean tryLabeling(final V n) {
    return isolatedWithReturn(() -> {
             if (parent == null) parent = n;
             return parent == n; // return true if n became parent
           });
  } // tryLabeling
  void compute() {
    for (int i=0; i<neighbors.length; i++) {
      final V child = neighbors[i];
      if (child.tryLabeling(this))
        async(() -> { child.compute(); }); // escaping async
     }
  } // compute
} // class V
. . .
root.parent = root; // Use self-cycle to identify root
finish(() -> { root.compute(); });
. . .
```