COMP 322: Fundamentals of Parallel Programming

Lecture 20: Critical Sections and the Isolated Statement

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

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

Formal Definition of Data Races (Recap from Lecture 5)

Formally, a data race occurs on location L in a program execution with computation graph CG if there exist steps (nodes) S1 and S2 in CG such that:

1. S1 does not depend on S2 and S2 does not depend on S1 i.e., there is no path of dependence edges from S1 to S2 or from S2 to S1 in CG, and
2. Both S1 and S2 read or write L, and at least one of the accesses is a write.

However, there are many cases in practice when two tasks may legitimately need to perform conflicting accesses to shared locations without incurring data races

- Special cases: finish/phaser accumulators, atomic variables
- How should conflicting accesses be handled in general?

Example of two tasks performing conflicting accesses

```
1.  class DoublyLinkedList {
2.   DoublyLinkedList prev, next;
3.   . . .
4.   void delete() {
5.     isolated { // start of mutual exclusion region (critical section)
6.       this.prev.next = this.next;
7.       this.next.prev = this.prev;
8.     } // end of mutual exclusion region (critical section)
9.     . . .
10.  }
11.  . . .
12. } // DoublyLinkedList
13. . . .
14. static void deleteTwoNodes(DoublyLinkedList L) {
15.  finish {
16.    async L.delete();
17.    async L.next.delete();
18.  }
19. }
```

How to enforce mutual exclusion?

- The predominant approach to ensure mutual exclusion proposed many years ago is to enclose the code region in a critical section.
  - "In concurrent programming a critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution. A critical section will usually terminate in fixed time, and a thread, task or process will have to wait a fixed time to enter it (aka bounded waiting). Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use, for example a semaphore."
  - Source: http://en.wikipedia.org/wiki/Critical_section

HJ isolated statement

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

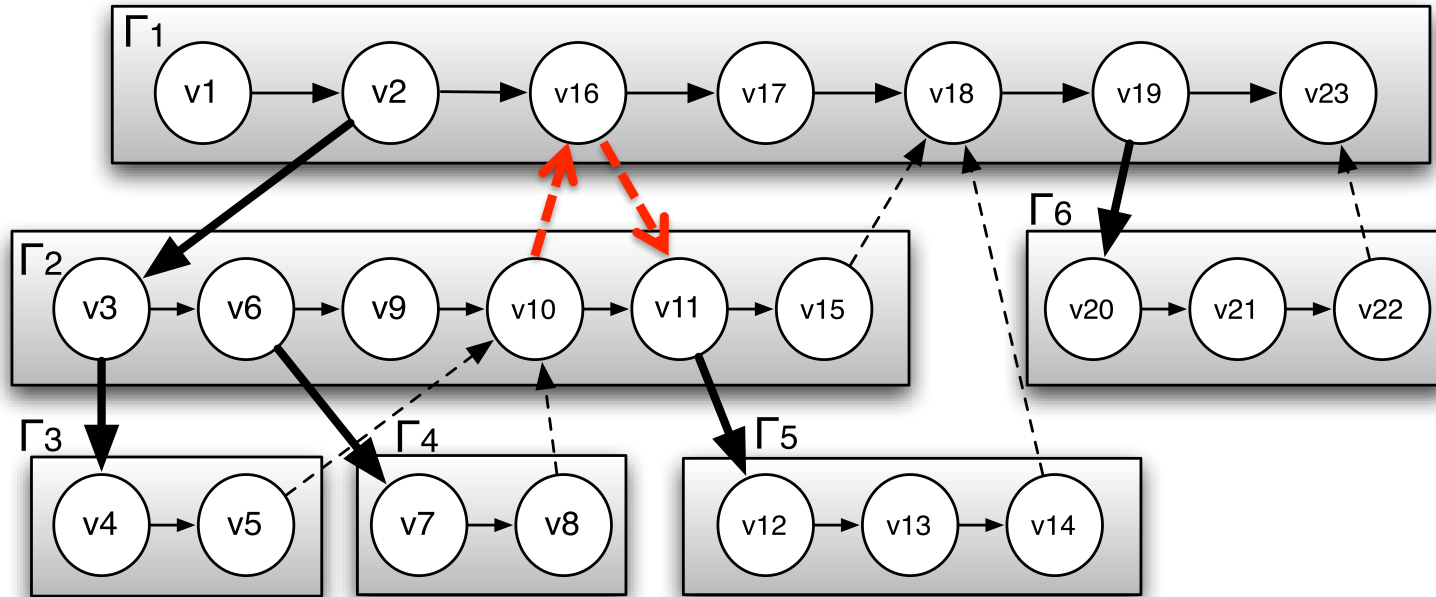
Semantics of Exceptions and Async's within an Isolated Statement

```
1.  isolated {
2.   int t1 = p.x;
3.   p.x++;
4.   // Task execution terminates with NullPointerException
5.   // if q==null (as in non-isolated case)
6.   int t2 = q.x;
7.   q.x--;
8.   // Async creation (but not execution) is part of mutual
9.   // exclusion construct.  Async can logically be executed
10.  // at end of isolated statement.
11.  async { ... t1 ... t2 ... }
12.  . . .
13. } // isolated
```

Serialized Computation Graph for Isolated Statements

- Model each instance of an isolated statement as a distinct step (node) in the CG.
- Need to reason about the order in which interfering isolated statements are executed
  - Complicated because the order may vary from execution to execution
- Introduce Serialized Computation Graph (SCG) that includes a specific ordering of all interfering isolated statements.
  - SCG consists of a CG with additional serialization edges.
  - Each time an isolated step, S', is executed, we add a serialization edge from S to S' for each isolated step, S, that has already executed such that S and S' have interfering accesses.
  - An SCG represents a set of executions in which all interfering isolated statements execute in the same order.

Example of Serialized Computation Graph with Serialization Edges for v10-v16-v11 order



Continue edge   Spawn edge   Join edge

Serialization edge

v10: isolated { x ++; y = 10; }
v11: isolated { x++; y = 11; }
v16: isolated { x++; y = 16; }

- Data race definition can be applied to Serialized Computation Graphs (SCGs) just like regular CGs
  - Need to consider all possible orderings of interfering isolated statements to establish data race freedom

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 logs all read and write operations performed in a transaction and 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
- Example of Software TM system for Java: DSTM2

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

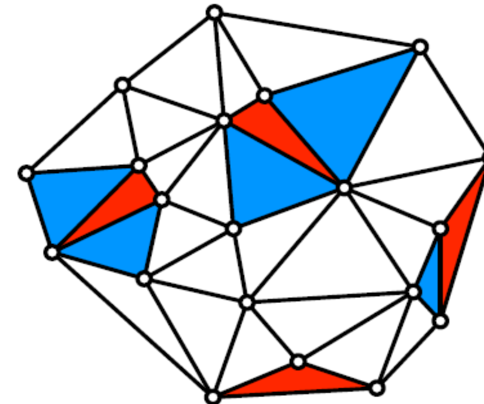
Delauney Mesh Refinement in Habanero-Java using Delegated Isolation

```
 1: void doCavity(Triangle start) {
 2:   async isolated {
 3:     if (start.isActive()) {
 4:       Cavity c = new Cavity(start);
 5:       c.initialize(start);
 6:       c.retriangulate();

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

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

13: void main() {
14:   mesh = ... ; // Load from file
15:   initialBadTriangles = mesh.badTriangles();
16:   Iterator it = initialBadTriangles.iterator();
17:   finish {
18:     while (it.hasNext()) {
19:       final Triangle t = (Triangle) it.next();
20:       if (t.isBad())
21:         Cavity.doCavity(t);
22:     }
19:   }
20: }
```


Before

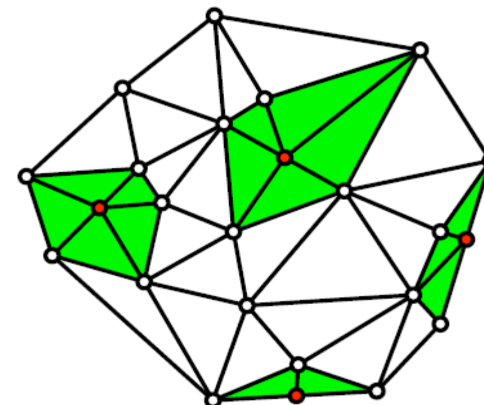

After

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

Performance: DMR benchmark on 16-core Xeon SMP

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

HJ (Coarse-Grained-Lock)
Java (Fine-Grained-Locks)
HJ (Delegated Isolation)

time in seconds

14
12
10
8
6
HJ (SEQ)
4
2
0

1 2 4 6 8 10 12 14 16

threads

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

Object-based isolation in HJ

```
isolated(<object-list>) <body>
```

- In this case, programmer specifies list of objects for which isolation is required
- Mutual exclusion is only guaranteed for instances of isolated statements that have a non-empty intersection in their object lists
  - Standard isolated is equivalent to isolated(*) by default i.e., isolation across all objects
- Implementation can choose to distinguish between read/write accesses for further parallelism
  - Current HJ implementation supports object-based isolation, does not exploit read/write distinction

DoublyLinkedList Example revisited with Object-Based Isolation

```
1.  class DoublyLinkedList {
2.   DoublyLinkedList prev, next;
3.   . . .
4.   void delete() {
5.     isolated (this.prev, this.next) { // start of object-based isolation
6.       this.prev.next = this.next;
7.       this.next.prev = this.prev
8.     } // end of object-based isolation
9.     . . .
10.  }
11.  . . .
12. } // DoublyLinkedList
13. . . .
14. static void deleteTwoNodes(DoublyLinkedList L) {
15.  finish {
16.    async L.delete();
17.    async L.next.delete();
18.  }
19. }
```

java.util.concurrent.AtomicInteger methods and their equivalent isolated statements

j.u.c.atomic Class and Constructors	j.u.c.atomic Methods	Equivalent HJ isolated statements
AtomicInteger	int j = v.**get**();	int j; isolated (v) j = v.val;
	v.**set**(newVal);	isolated (v) v.val = newVal;
AtomicInteger() // init = 0	int j = v.**getAndSet**(newVal);	int j; isolated (v) { j = v.val; v.val = newVal; }
	int j = v.**addAndGet**(delta);	isolated (v) { v.val += delta; j = v.val; }
	int j = v.**getAndAdd**(delta);	isolated (v) { j = v.val; v.val += delta; }
AtomicInteger(init)	boolean b = v.**compareAndSet**(expect,update);	boolean b; isolated (v) if (v.val==expect) {v.val=update; b=true;} else b = false;

Methods in java.util.concurrent.AtomicInteger class and their equivalent HJ isolated statements. Variable v refers to an AtomicInteger object in column 2 and to a standard non-atomic Java object in column 3. val refers to a field of type int.

Implementing AtomicInteger.getAndAdd() using compareAndSet()

```
        /** Atomically adds delta to the current value.
1.       *
2.       * @param delta the value to add
3.       * @return the previous value
4.       */
5.      public final int getAndAdd(int delta) {
6.          for (;;) {
7.              int current = get();
8.              int next = current + delta;
9.              if (compareAndSet(current, next))
10.                 return current;
11.          }
12.     }
```

- **Source: http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/main/java/util/concurrent/atomic/AtomicInteger.java**

java.util.concurrent. AtomicReference methods and their equivalent isolated statements

j.u.c.atomic Class and Constructors	j.u.c.atomic Methods	Equivalent HJ isolated statements
AtomicReference	Object o = v.**get**();	Object o; isolated (v) o = v.ref;
	v.**set**(newRef);	isolated (v) v.ref = newRef;
AtomicReference() // init = null	Object o = v.**getAndSet**(newRef);	Object o; isolated (v) { o = v.ref; v.ref = newRef; }

AtomicReference(init)	boolean b =
	v.**compareAndSet**
	(expect,update);
		boolean b;
		isolated (v)
		if (v.ref==expect) {v.ref=update; b=true;}
		else b = false;

Methods in java.util.concurrent.AtomicReference class and their equivalent HJ isolated statements. Variable v refers to an AtomicReference object in column 2 and to a standard non-atomic Java object in column 3. ref refers to a field of type Object.

AtomicReference<T> can be used to specify a type parameter.

Parallel Spanning Tree Algorithm using isolated statement

```
1. class V  {
2.   V [] neighbors; // adjacency list for input graph
3.   AtomicReference parent; // output value of parent in spanning tree
4.   boolean tryLabeling(V n) {
5.     isolated(this) if (parent == null) parent=n;
6.     return parent == n;
7.   } // tryLabeling
8.   void compute() {
9.     for (int i=0; i<neighbors.length; i++) {
10.      V child = neighbors[i];
11.      if (child.tryLabeling(this))
12.          async child.compute(); //escaping async
13.     }
14.  } // compute
15.} // class V
16.. . .
17.root.parent = root; // Use self-cycle to identify root
18.finish root.compute();
19.. . .
```

1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16

Figure source: http://en.wikipedia.org/wiki/Spanning_tree