

---

# COMP 322: Fundamentals of Parallel Programming

## Lecture 19: Critical Sections and the Isolated Statement

Vivek Sarkar  
Department of Computer Science, Rice University  
[vsarkar@rice.edu](mailto:vsarkar@rice.edu)

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



# Summary of Module 1: Deterministic Shared-Memory Parallelism

---

- **Serializable subset of HJ**
  - { async, finish, future, forasync }
  - Erasure property: any HJ program written using the above constructs can be converted to an equivalent sequential program by “erasing” all parallel constructs i.e., by removing async & finish, and replacing future & forasync by equivalent sequential constructs
- **Deadlock-free subset of HJ**
  - { next, barriers, phasers, forall, async phased } + Serializable subset
  - Deadlock-freedom property: any HJ program written using the above constructs is guaranteed to never deadlock
- **Deterministic subset of HJ**
  - { data driven futures, async await } + Deadlock-free subset
  - Data-race-free determinism property: if any HJ program written using the above constructs is guaranteed to be data-race-free for a given input, then it must also be deterministic and structurally deterministic for that input i.e., all executions with the same input must generate the same output AND the same computation graph



# Outline of Today's Lecture

---

## Start of Module 2: Nondeterministic Shared-Memory Parallelism and Concurrency

- Critical Sections and the Isolated Statement
- Atomic Variables



# Formal Definition of Data Races (Recap)

---

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 with determinism guarantees: finish accumulators, phaser accumulators
- How should conflicting accesses be handled in general, when outcome may be nondeterministic?



# Example of two tasks performing conflicting accesses --- need for “mutual exclusion”

```
1. class DoublyLinkedListNode {
2.     DoublyLinkedListNode prev, next;
3.     . . .
4.     void delete() {
5.         { // start of desired mutual exclusion region
6.             this.prev.next = this.next;
7.             this.next.prev = this.prev;
8.         } // end of desired mutual exclusion region
9.         . . . // other code in delete() that does not need mutual exclusion
10.    }
11. } // DoublyLinkedListNode
12. . . .
13. static void deleteTwoNodes(DoublyLinkedListNode L) {
14.     finish {
15.         DoublyLinkedListNode second = L.next;
16.         DoublyLinkedListNode third = second.next;
17.         async second.delete();
18.         async third.delete(); // conflicts with previous async
19.     }
20. }
```



# 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](http://en.wikipedia.org/wiki/Critical_section)



# HJ isolated statement

---

## isolated <body>

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



# Use of isolated to fix previous example with conflicting accesses

```
1. class DoublyLinkedListNode {
2.     DoublyLinkedListNode prev, next;
3.     . . .
4.     void delete() {
5.         isolated { // start of desired mutual exclusion region
6.             this.prev.next = this.next;
7.             this.next.prev = this.prev;
8.         } // end of desired mutual exclusion region
9.         . . . // other code in delete() that does not need mutual exclusion
10.    }
11. } // DoublyLinkedListNode
12. . . .
13. static void deleteTwoNodes(DoublyLinkedListNode L) {
14.     finish {
15.         DoublyLinkedListNode second = L.next;
16.         DoublyLinkedListNode third = second.next;
17.         async second.delete();
18.         async third.delete(); // conflicts with previous async
19.     }
20. }
```

Let's try another isolated example in Worksheet 19!





# Parallel Spanning Tree Algorithm using isolated statement

```
1. class V {
2.   V [] neighbors; // adjacency list for input graph
3.   V parent; // output value of parent in spanning tree
4.   boolean tryLabeling(V n) {
5.     isolated if (parent == null) parent=n;
6.     return parent == n; // return true for success
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. . . .
```

Example graph  
(root=1, spanning  
tree edge shown  
as arrow from  
child to parent)

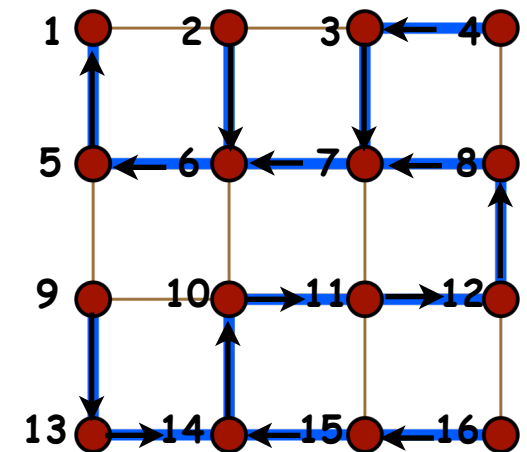


Figure source:

[http://en.wikipedia.org/wiki/Spanning\\_tree](http://en.wikipedia.org/wiki/Spanning_tree)



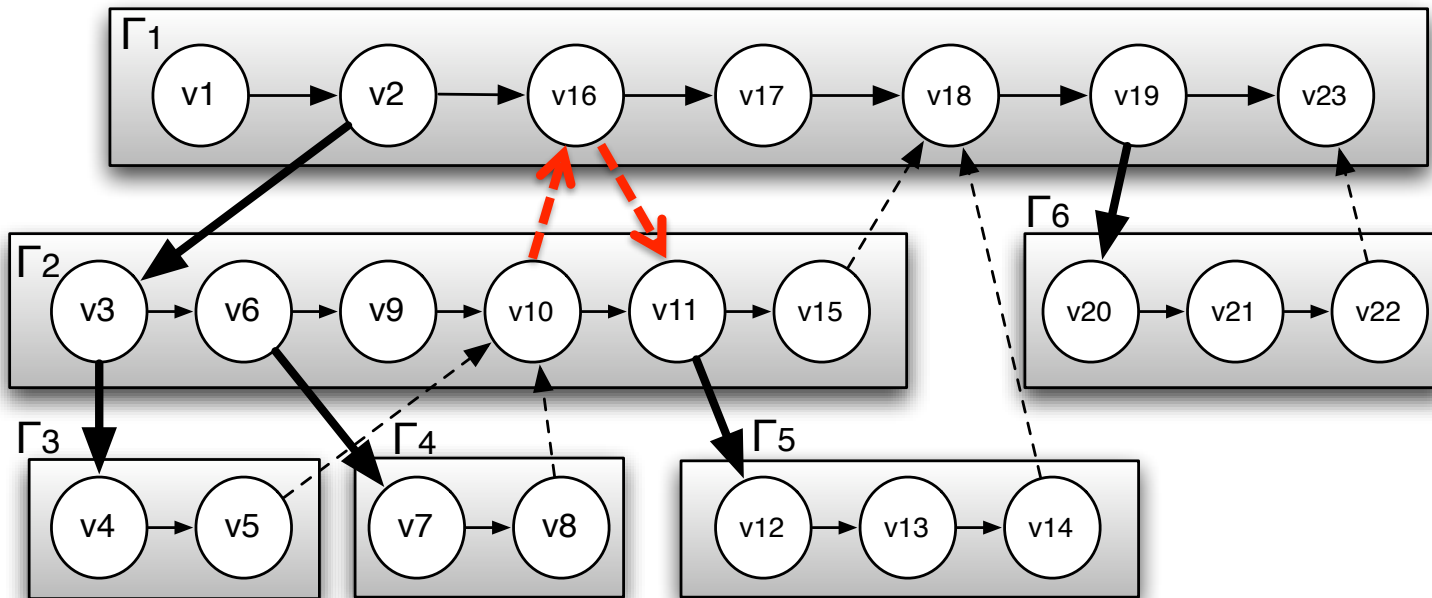
# Serialized Computation Graph for Isolated Statements

---

- 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 of isolated statements 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 prior “interfering” isolated step,  $S$ 
    - Two isolated statements always interfere with each other
    - Interference of “object-based isolated” statements depends on intersection of object sets
    - Serialization edge is not needed if  $S$  and  $S'$  are already ordered in CG
  - 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**



# Object-based isolation in HJ

---

`isolated(obj1, obj2, ...) <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, but does not exploit read/write distinction**



# DoublyLinkedListNode Example revisited with Object-Based Isolation

```
1. class DoublyLinkedListNode {
2.     DoublyLinkedListNode prev, next;
3.     . . .
4.     void delete() {
5.         isolated(this.prev, this, this.next) { // object-based isolation
6.             this.prev.next = this.next;
7.             this.next.prev = this.prev;
8.         }
9.         . . .
10.    }
11. } // DoublyLinkedListNode
12. . . .
13. static void deleteTwoNodes(DoublyLinkedListNode L) {
14.     finish {
15.         DoublyLinkedListNode second = L.next;
16.         DoublyLinkedListNode third = second.next;
17.         async second.delete();
18.         async third.delete();
19.     }
20. }
```



# Outline of Today's Lecture

---

## Start of Module 2: Nondeterministic Shared-Memory Parallelism and Concurrency

- **Critical Sections and the Isolated Statement**
- **Atomic Variables**



# Atomic Accesses in Java

---

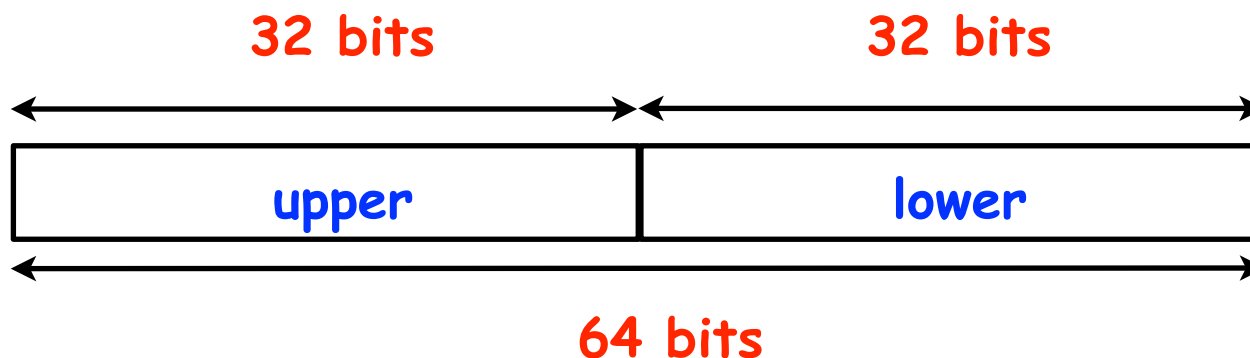
- **An atomic action happens all at once**
  - **Subcomponents of one atomic action cannot be interleaved with subcomponents of another atomic action**
- **Reads and write for reference variables and primitives (except long and double) are atomic**
- **Basic safety guarantee: a read always returns a value written by some task, some time in the past**
  - **No “out-of-thin-air” values for references and primitives (except for long and double)**



# Why reads and writes on long/double values may be non-atomic

---

1. `long x; // upper = lower = 0`
2. `async { x = 1L << 32 + 1L; } // lower=1; upper=1;`
3. `async { x = 2L << 32 + 2L; } // lower=2; upper=2;`
4. `async { System.out.println(x); }`
5. `// Possible output value includes`
6. `// 1L << 32 + 2L (lower=2, upper=1)`





# java.util.concurrent library

---

- **Atomic variables**
  - Efficient implementations of special-case patterns of isolated statements
- **Concurrent Collections:**
  - Queues, blocking queues, concurrent hash map, ...
  - Data structures designed for concurrent environments
- **Executors, Thread pools and Futures**
  - Execution frameworks for asynchronous tasking
- **Locks and Conditions**
  - More flexible synchronization control
  - Read/write locks
- **Synchronizers: Semaphore, Latch, Barrier, Exchanger, Phaser**
  - Tools for thread coordination
- **WARNING: only a small subset of the full java.util.concurrent library can safely be used in HJ programs**
  - Atomic variables are part of the safe subset
  - We will study the full library later this semester as part of Java Concurrency



# java.util.concurrent.atomic.AtomicInteger

---

- **Constructors**
  - `new AtomicInteger()`
    - Creates a new `AtomicInteger` with initial value 0
  - `new AtomicInteger(int initialValue)`
    - Creates a new `AtomicInteger` with the given initial value
- **Selected methods**
  - `int addAndGet(int delta)`
    - Atomically adds delta to the current value of the atomic variable, and returns the new value
  - `int getAndAdd(int delta)`
    - Atomically returns the current value of the atomic variable, and adds delta to the current value
- **Similar interfaces available for `LongInteger`**



# Work-Sharing Pattern using AtomicInteger

---

```
1. import java.util.concurrent.atomic.AtomicInteger;
2. . . .
3. String[] X = ... ; int numTasks = ...;
4. AtomicInteger a = new AtomicInteger();
5. . . .
6. finish for (int i=0; i<numTasks; i++ )
7.     async {
8.         do {
9.             int j = a.getAndAdd(1);
10.            // can also use a.getAndIncrement()
11.            if (j >= X.length) break;
12.            . . . // Process X[j]
13.        } while (true);
14.    } // finish-for-async
```



# Atomicity in standard vs. atomic variables

---

- Reads and write for reference variables and primitives (except long and double) are atomic
- Basic safety guarantee: a read always returns a value written by some task, some time in the past
  - No “out-of-thin-air” values for references and primitives (except for long and double)
- Atomic variables support compound atomic operations that go beyond single read/write accesses
  - Operations on atomic variables can be safely invoked by parallel tasks, but (like isolated statements) they may increase the critical path length of your parallel program
    - Not a problem if the remaining parallel (non-atomic) work is large



# 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
<b>AtomicInteger</b>	<code>int j = v.get();</code>	<code>int j; isolated (v) j = v.val;</code>
	<code>v.set(newVal);</code>	<code>isolated (v) v.val = newVal;</code>
<b>AtomicInteger()</b> // init = 0	<code>int j = v.getAndSet(newVal);</code>	<code>int j; isolated (v) { j = v.val; v.val = newVal; }</code>
	<code>int j = v.addAndGet(delta);</code>	<code>isolated (v) { v.val += delta; j = v.val; }</code>
	<code>int j = v.getAndAdd(delta);</code>	<code>isolated (v) { j = v.val; v.val += delta; }</code>
<b>AtomicInteger(init)</b>	<code>boolean b = v.compareAndSet (expect,update);</code>	<code>boolean b; isolated (v) if (v.val==expect) {v.val=update; b=true;} else b = false;</code>

**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.**



# 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
<b>AtomicReference</b>	Object o = v.get();	Object o; isolated (v) o = v.ref;
	v.set(newRef);	isolated (v) v.ref = newRef;
<b>AtomicReference()</b> // init = null	Object o = v.getAndSet(newRef);	Object o; isolated (v) { o = v.ref; v.ref = newRef; }
<b>AtomicReference(init)</b>	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 AtomicReference

---

```
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.         return parent.compareAndSet(null, n);
6.     } // tryLabeling
7.     void compute() {
8.         for (int i=0; i<neighbors.length; i++) {
9.             V child = neighbors[i];
10.            if (child.tryLabeling(this))
11.                async child.compute(); //escaping async
12.        }
13.    } // compute
14.} // class V
15. . . .
16.root.parent = root; // Use self-cycle to identify root
17.finish root.compute();
18. . . .
```



# Three cases of contention among isolated statements

---

- 1. Low contention: when isolated statements are executed infrequently**
  - Use of global isolated statements is usually the best approach. No visible benefit from other techniques because they incur overhead that is not needed since contention is low.
- 2. Moderate contention (no variable is a “hot spot”): when serialization of all isolated statements limits performance, but serializing only interfering isolated statements results in good scalability**
  - Atomic variables and object-based isolation usually do well in this scenario since the benefit obtained from reduced serialization outweighs any extra overhead incurred.
- 3. High contention (one or more variables are hot spots): when interfering isolated statements dominate the program execution time**
  - Best approach in such cases is to find an alternative approach to isolated e.g., use of finish/phaser accumulators





# Worksheet #19:

## Insertion of isolated for correctness

---

Name 1: \_\_\_\_\_

Name 2: \_\_\_\_\_

The goal of `IsolatedPRNG` is to implement a single Pseudo Random Number Generator object that can be shared by multiple tasks. Show the isolated statement(s) that you can insert in method `nextSeed()` to avoid data races and guarantee proper semantics.

```
class IsolatedPRNG {
  private int seed;
  public int nextSeed() {
    int retVal;

    retVal = seed;

    seed = nextInt(retVal);

    return retVal;
  } // nextSeed()
  . . .
} // IsolatedPRNG
```

```
main() { // Pseudocode
  // Initial seed = 1
  IsolatedPRNG r = new IsolatedPRNG(1);
  async { print r.nextSeed(); ... }
  async { print r.nextSeed(); ... }
} // main()
```

