COMP 322: Fundamentals of Parallel Programming

Lecture 21: Read-write Isolation, Atomic Variables

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

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

Worksheet #19 solution: Insertion of isolated for correctness

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.

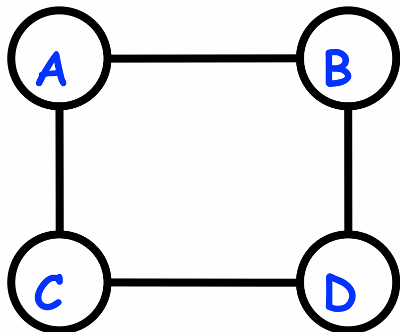
```
1.class IsolatedPRNG {
2. private int seed;
3. public int nextSeed() {
4. return isolatedWithReturn(()->{
5.   int retVal = seed;
6.   seed = nextInt(retVal);
7.   return retVal;
8.  });
9.} // nextSeed()
10.  . . .
11.} // IsolatedPRNG
```

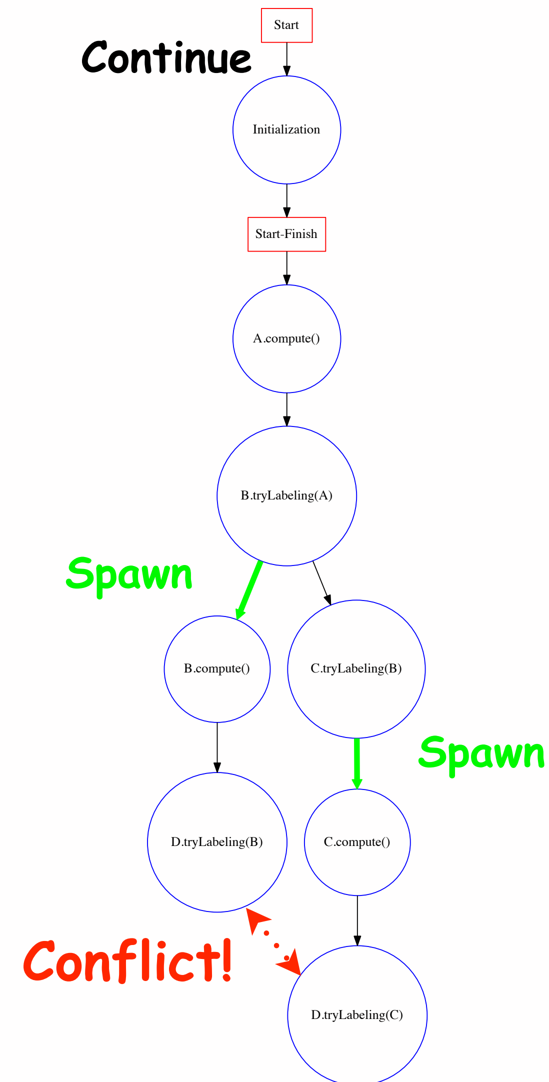
```
main() { // Pseudocode
  // Initial seed = 1
  IsolatedPRNG r = new IsolatedPRNG(1);
  async(() -> { print r.nextSeed(); ... });
  async(() -> { print r.nextSeed(); ... });
} // main()
```

Note that enclosing line 5 and line 6 in separate isolated constructs will avoid data races. But will it guarantee the semantics of a sequential Pseudo Random Number Generator?

Worksheet #20 Solution: Identifying conflicts in isolated constructs

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 using isolated construct

```
1.  class V  {
2.    V [] neighbors; // adjacency list for input graph
3.    V parent; // output value of parent in spanning tree
4.    boolean tryLabeling(final V n) {
5.      return isolatedWithReturn(() -> {
6.        if (parent == null) parent = n;
7.        return parent == n; // return true if n became parent
8.      });
9.    } // tryLabeling
10.   void compute() {
11.     for (int i=0; i<neighbors.length; i++) {
12.       final V child = neighbors[i];
13.       if (child.tryLabeling(this))
14.         async(() -> { child.compute(); }); // escaping async
15.      }
16.   } // compute
17. } // class V
18. . . .
19. root.parent = root; // Use self-cycle to identify root
20. finish(() -> { root.compute(); });
21. . . .
```

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

Object-based isolation in HJ (Recap)

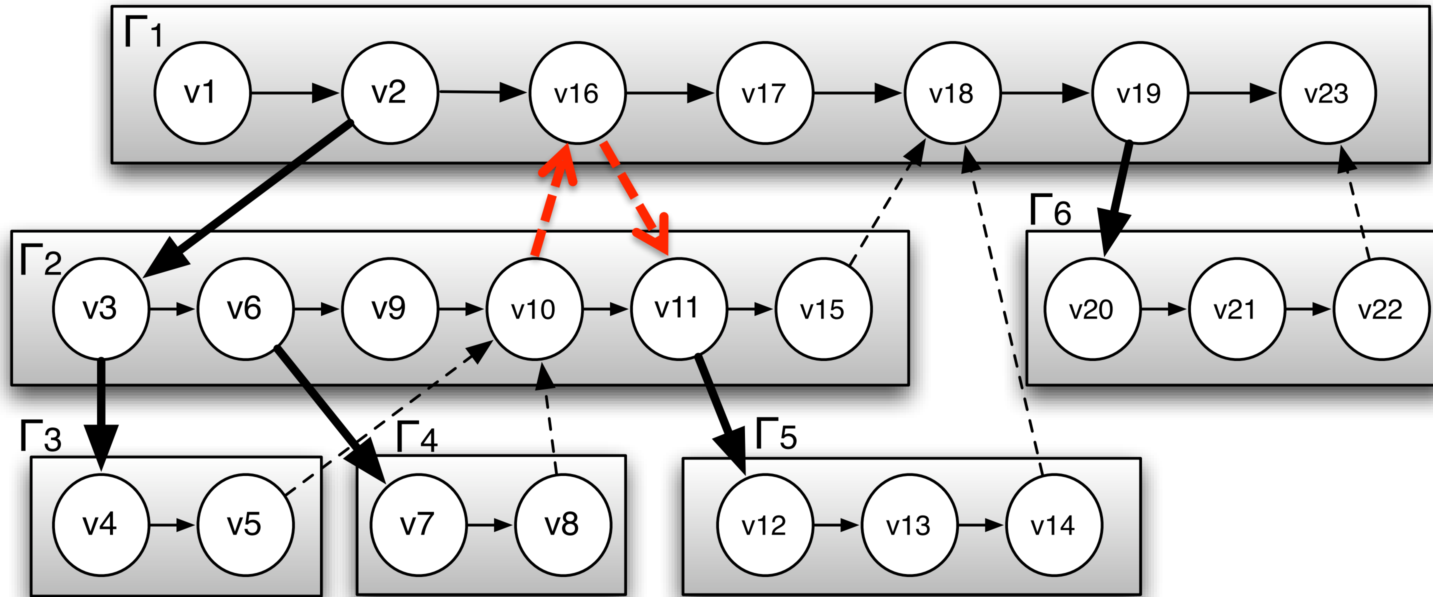
```
isolated(obj1, obj2, …, () -> <body>)
```

- **In this case, programmer specifies list of objects for which isolation is required**
- **Mutual exclusion is only guaranteed for instances of isolated constructs that have a non-empty intersection in their object lists**
  - **Standard isolated is equivalent to "isolated(*)" by default i.e., isolation across all objects**
- **Example:**
  - `isolated(a,b,()->{..})` **and** `isolated(c,d,()->{..})` **can execute in parallel**
  - `isolated(a,b,()->{..})` **and** `isolated(b,c,()->{..})` **cannot execute in parallel**

Parallel Spanning Tree Algorithm using object-based isolated construct (Lab 7)

```
1.  class V  {
2.    V [] neighbors; // adjacency list for input graph
3.    V parent; // output value of parent in spanning tree
4.    boolean tryLabeling(final V n) {
5.      return isolatedWithReturn(this, () -> {
6.                 if (parent == null) parent = n;
7.                 return parent == n; // return true if n became parent
8.               });
9.    } // tryLabeling
10.   void compute() {
11.     for (int i=0; i<neighbors.length; i++) {
12.       final V child = neighbors[i];
13.       if (child.tryLabeling(this))
14.         async(() -> { child.compute(); }); // escaping async
15.      }
16.   } // compute
17. } // class V
18. . . .
19. root.parent = root; // Use self-cycle to identify root
20. finish(() -> { root.compute(); });
21. . . .
```

Abstract Metrics with Isolated Constructs



Continue edge

Spawn edge

Join edge

Serialization edge

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

Abstract metrics considers one possible orderings of conflicting isolated constructs to compute the critical path length (CPL)

Read-Write Object-based isolation in HJ

```
isolated(readMode(obj1),writeMode(obj2), …, () -> <body> );
```

- Programmer specifies list of objects as well as their read-write modes for which isolation is required
- Not specifying a mode is the same as specifying a write mode
- Mutual exclusion is only guaranteed for instances of isolated statements that have a non-empty intersection in their object lists such that one of the accesses is in writeMode
  - Default mode = read + write
- Sorted List example

```
1.   public boolean contains(Object object) {
2.      return isolatedWithReturn( readMode(this), () -> {
3.        Entry pred, curr;
4.        ...
5.        return (key == curr.key);
6.     });
7.   }
8.
9.    public int add(Object object) {
10.    return isolatedWithReturn( writeMode(this), () -> {
11.      Entry pred, curr;
12.      ...
13.      if (...) return 1; else return 0;
14.    });
15.  }
```

java.util.concurrent library

- <u>Atomic variables</u>
  - 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
- <u>WARNING: only a small subset of the full java.util.concurrent library can safely be used in HJ programs</u>
  - 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.  int[] taskId = new int[X.length];
5.  AtomicInteger a = new AtomicInteger();
6.  . . .
7.  finish(() -> {
8.    for (int i=0; i<numTasks; i++ )
9.      async(() -> {
10.     do {
11.         int j = a.getAndAdd(1);
12.         // can also use a.getAndIncrement()
13.         if (j >= X.length) break;
14.         taskId[j] = i; // Task i processes string X[j]
15.         . . .
16.      } while (true);
17.    });
18.}); // finish-for-async
```

java.util.concurrent.AtomicInteger methods and their equivalent isolated constructs (pseudocode)

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.

java.util.concurrent.atomic.AtomicReference

- Constructors
  - `new AtomicReference()`
    - Creates a new AtomicReference with initial value `0`
  - `new AtomicReference(Object init)`
    - Creates a new AtomicReference with the given initial value
- Selected methods
  - `int getAndSet(Object newRef)`
    - Atomically get current value of the atomic variable, and set value to newRef
  - `int compareAndSet(Object expect, Object update)`
    - Atomically check if current value = expect. If so, replace the value of the atomic variable by update and return true. Otherwise, return false.

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 AtomicReference

```
1.  class V  {
2.    V [] neighbors; // adjacency list for input graph
3.    AtomicReference<V> parent; // output value of parent in spanning tree
4.    boolean tryLabeling(final V n) {
5.      return parent.compareAndSet(null, n);
6.  ;
7.    } // tryLabeling
8.    void compute() {
9.      for (int i=0; i<neighbors.length; i++) {
10.       final 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. . . .
```

Worksheet #21: Abstract Metrics with Isolated Constructs

Name: ______________________ Netid: ______________________

Compute the WORK and CPL metrics for this program. Indicate if your answer depends on the execution order of isolated constructs.

```
1.    finish(() -> {
2.         for (int i = 0; i < 5; i++) {
3.             async(() -> {
4.                     doWork(2);
5.                     isolated(() -> { doWork(1); });
6.                     doWork(2);
7.                 }); // async
8.         } // for
9.     }); // finish
```