COMP 322: Fundamentals of Parallel Programming

Lecture 22: Read-write Isolation, Atomic Variables

Vivek Sarkar, Eric Allen
Department of Computer Science, Rice University

Contact email: 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 are guaranteed to 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 (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 common object 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

DoublyLinkedListNode Example revisited with Object-Based Isolation

```
class DoublyLinkedListNode {
 DoublyLinkedListNode prev, next;
 . . .
 void delete() {
   isolated(this.prev, this, this.next, () -> { // object-based isolation
     this.prev.next = this.next;
     this.next.prev = this.prev;
   });
   . . .
 }
} // DoublyLinkedListNode
. . .
static void deleteTwoNodes(final DoublyLinkedListNode L) {
 finish(() -> {
   DoublyLinkedListNode second = L.next;
   DoublyLinkedListNode third = second.next;
   async(() -> { second.delete(); });
   async(() -> { third.delete(); });
 });
}
```

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 (default mode = read + write)
- 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
- Sorted List example

```
public boolean contains(Object object) {
   return isolatedWithReturn( readMode(this), () -> {
     Entry pred, curr;
     ...
     return (key == curr.key);
  });
}

 public int add(Object object) {
   return isolatedWithReturn( writeMode(this), () -> {
     Entry pred, curr;
     ...
     if (...) return 1; else return 0;
   });
}
```


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 with HJlib
  - 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

```
import java.util.concurrent.atomic.AtomicInteger;
. . .
String[] X = ... ; int numTasks = ...;
int[] taskId = new int[X.length];
AtomicInteger a = new AtomicInteger();
. . .
finish(() -> {
  for (int i=0; i<numTasks; i++ )
    async(() -> {
     do {
        int j = a.getAndAdd(1);
        // can also use a.getAndIncrement()
        if (j >= X.length) break;
        taskId[j] = i; // Task i processes string X[j]
        . . .
      } while (true);
    });
}); // 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

```
class V  {
  V [] neighbors; // adjacency list for input graph
  AtomicReference<V> parent; // output value of parent in spanning tree
  boolean tryLabeling(final V n) {
    return parent.compareAndSet(null, n);
;
  } // 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(); });
. . .
```

COMP 322 Worksheet 21 solution: Eureka-style Speculative Parallelism.

The code snippet below performs a eureka-style search on a 2-D array with a fixed number of tasks. Each task uses the `next()` operation to ensure the computation progresses in a lock step manner where all tasks execute one call to `check()` before performing the equality (==) comparison. What does the program print when it completes execution? Also, print the number of == comparisons performed by the program. Remember that once the search eureka has been resolved (via a call to `offer()`), subsequent calls to `check()` will cause the task to terminate.

```
final int numRows = 10;
final int numCols = 100;
final int[][] dataArray = new int[numRows][numCols];
for (int i = 0; i < numRows; i++) {
    for (int j = 0; j < numCols; j++) {
        dataArray[i][j] = 100 * i + j;
    }
}

final int searchElement = 625;
final HjSearchEureka<int[]> eureka = newSearchEureka();
finish(eureka, ()-> {
       forasyncPhased(0, numRows - 1, (i) -> {
         for (int j = 0; j < numCols; j++) {
             final int[] elemIndex = {i, j};
             eureka.check(elemIndex);
             next(); // barrier
             if (dataArray[i][j] == searchElement) {
                 eureka.offer(elemIndex);
             }
             next(); // barrier
        } // for
       }); // forasyncPhased
}); // finish
final int[] index = eureka.get();
System.out.println("Result = " + Arrays.toString(index));
```

Answer:

```
Result: [6, 25] (found by task i=6 in iteration j=25)
Number of == comparisons: 260
```

Due to the presence of the barriers, each of the 10 tasks performs a comparison operation before the next comparison operation is performed by any of the other tasks. Since the call to offer() occurs at the 26th iteration (j=25), the total number of comparisons is 10x26 = 260.