COMP 322: Fundamentals of Parallel Programming

Lecture 19: Java Atomic Variables — a special case of isolated

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

Acknowledgments for Today's Lecture

- **Lecture 19 handout**

HJ isolated statement (Recap)

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: async, finish, get, forall
- In case of exception, all updates performed by <body> before throwing the exception will be observable after exiting <body>

DoublyLinkedListNode example (Recap)

```
class DoublyLinkedListNode {
  DoublyLinkedListNode prev, next;
  . . .
  void delete() {
    isolated { // start of mutual exclusion region (critical section)
      if (this.prev != null) this.prev.next = this.next;
      if (this.next != null) this.next.prev = this.prev
    } // end of mutual exclusion region (critical section)
    . . .
  }
  . . .
}
. . .
static void deleteTwoNodes(DoublyLinkedListNode n1, n2) {
  finish {
    async n1.delete();
    async n2.delete();
  }
}
```

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?

Transactional Memory (TM)

- 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 is suitable for mainstream use as yet

Three cases of contention among isolated statements

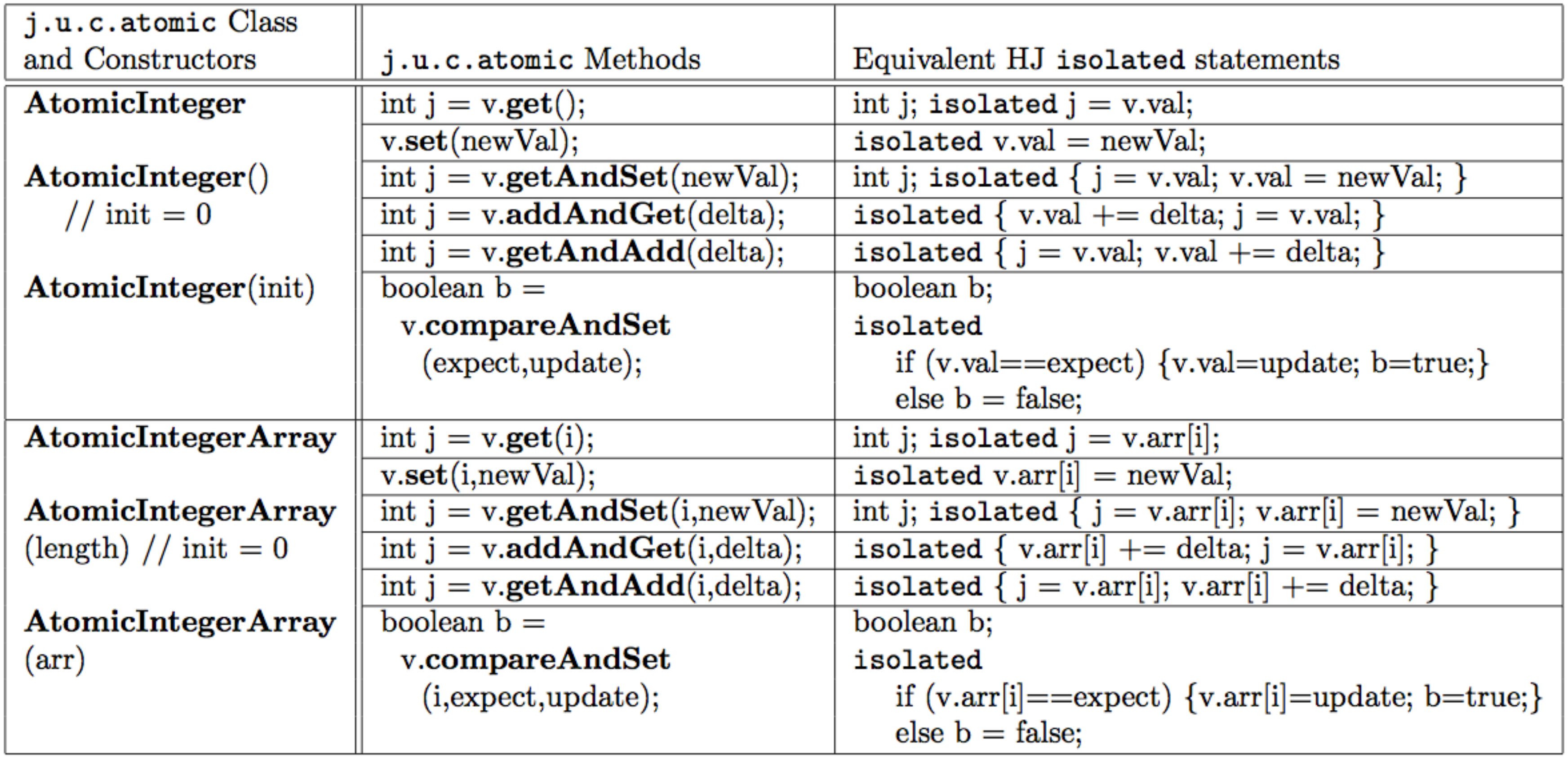
1. Low contention: when isolated statements are executed infrequently
   - A single-lock approach as in HJ is often the best solution. No visible benefit from other techniques because they incur overhead that is not needed since contention is low.
2. Moderate contention: when the serialization of all 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

`java.util.concurrent`

Sub-packages include

- 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

Table 1: Methods in java.util.concurrent atomic classes AtomicInteger and AtomicIntegerArray



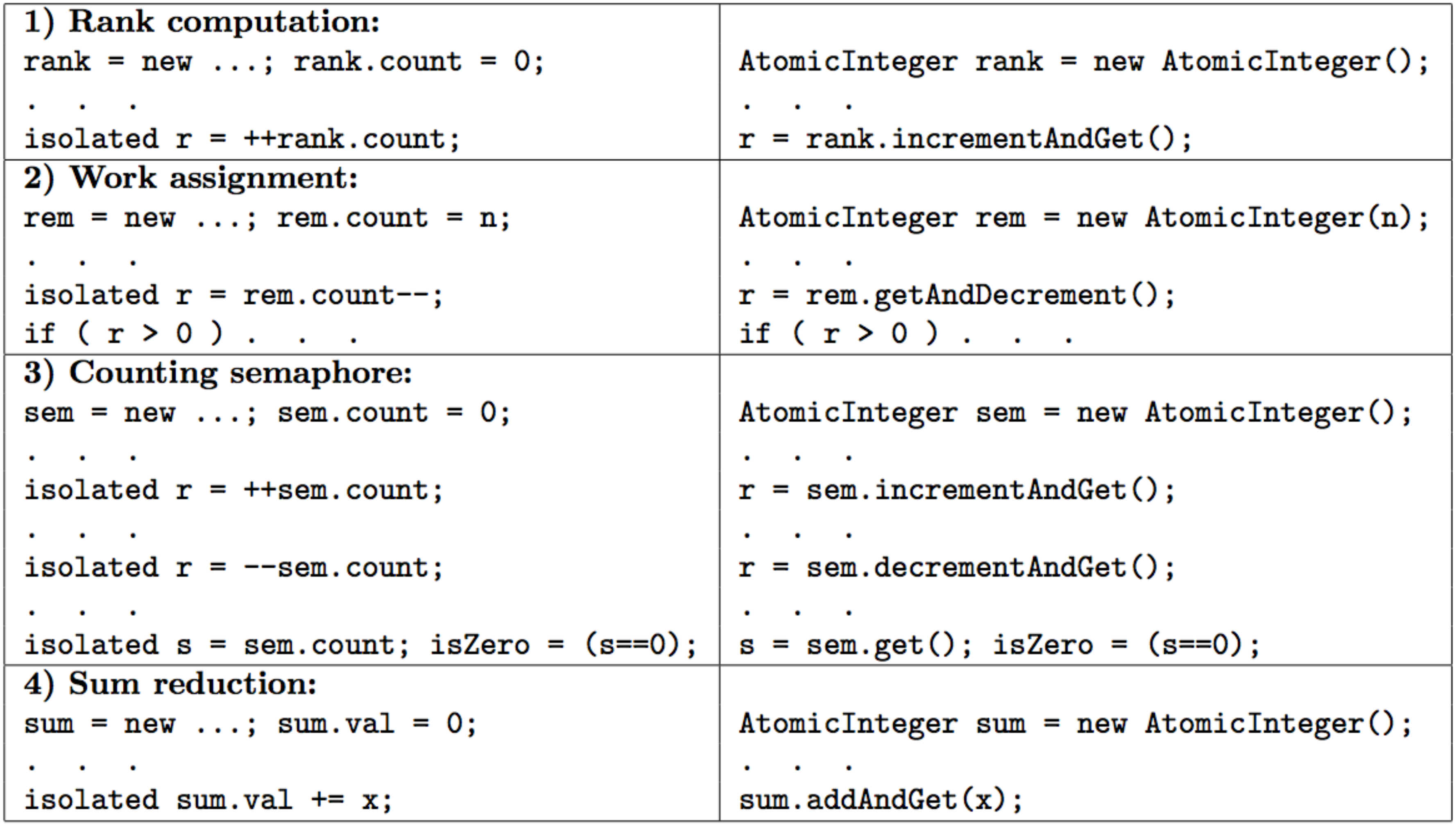
`j.u.c.atomic` Class and Constructors	`j.u.c.atomic` Methods	Equivalent HJ `isolated` statements
AtomicInteger **AtomicInteger**() // init = 0 **AtomicInteger**(init)	int j = v.**get**();	int j; `isolated` j = v.val;
	v.**set**(newVal);	`isolated` v.val = newVal;
	int j = v.**getAndSet**(newVal);	int j; `isolated` { j = v.val; v.val = newVal; }
	int j = v.**addAndGet**(delta);	`isolated` { v.val += delta; j = v.val; }
	int j = v.**getAndAdd**(delta);	`isolated` { j = v.val; v.val += delta; }

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

AtomicIntegerArray **AtomicIntegerArray** (length) // init = 0 **AtomicIntegerArray** (arr)	int j = v.**get**(i);	int j; `isolated` j = v.arr[i];
	v.**set**(i,newVal);	`isolated` v.arr[i] = newVal;
	int j = v.**getAndSet**(i,newVal);	int j; `isolated` { j = v.arr[i]; v.arr[i] = newVal; }
	int j = v.**addAndGet**(i,delta);	`isolated` { v.arr[i] += delta; j = v.arr[i]; }
	int j = v.**getAndAdd**(i,delta);	`isolated` { j = v.arr[i]; v.arr[i] += delta; }

	boolean b =
	v.**compareAndSet**
	(i,expect,update);
		boolean b;
		`isolated`
		if (v.arr[i]==expect) {v.arr[i]=update; b=true;}
		else b = false;

Table 2: Examples of common isolated statement idioms and their equivalent AtomicInteger implementations



1) Rank computation:
`rank = new ...; rank.count = 0;`
`. . .`
`isolated r = ++rank.count;`
	`AtomicInteger rank = new AtomicInteger();`
	`. . .`
	`r = rank.incrementAndGet();`

2) Work assignment:
`rem = new ...; rem.count = n;`
`. . .`
`isolated r = rem.count--;`
`if ( r > 0 ) . . .`
	`AtomicInteger rem = new AtomicInteger(n);`
	`. . .`
	`r = rem.getAndDecrement();`
	`if ( r > 0 ) . . .`

3) Counting semaphore:
`sem = new ...; sem.count = 0;`
`. . .`
`isolated r = ++sem.count;`
`. . .`
`isolated r = --sem.count;`
`. . .`
`isolated s = sem.count; isZero = (s==0);`
	`AtomicInteger sem = new AtomicInteger();`
	`. . .`
	`r = sem.incrementAndGet();`
	`. . .`
	`r = sem.decrementAndGet();`
	`. . .`
	`s = sem.get(); isZero = (s==0);`

4) Sum reduction:
`sum = new ...; sum.val = 0;`
`. . .`
`isolated sum.val += x;`
	`AtomicInteger sum = new AtomicInteger();`
	`. . .`
	`sum.addAndGet(x);`

Table 3: Methods in java.util.concurrent atomic classes AtomicReference and AtomicReferenceArray

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

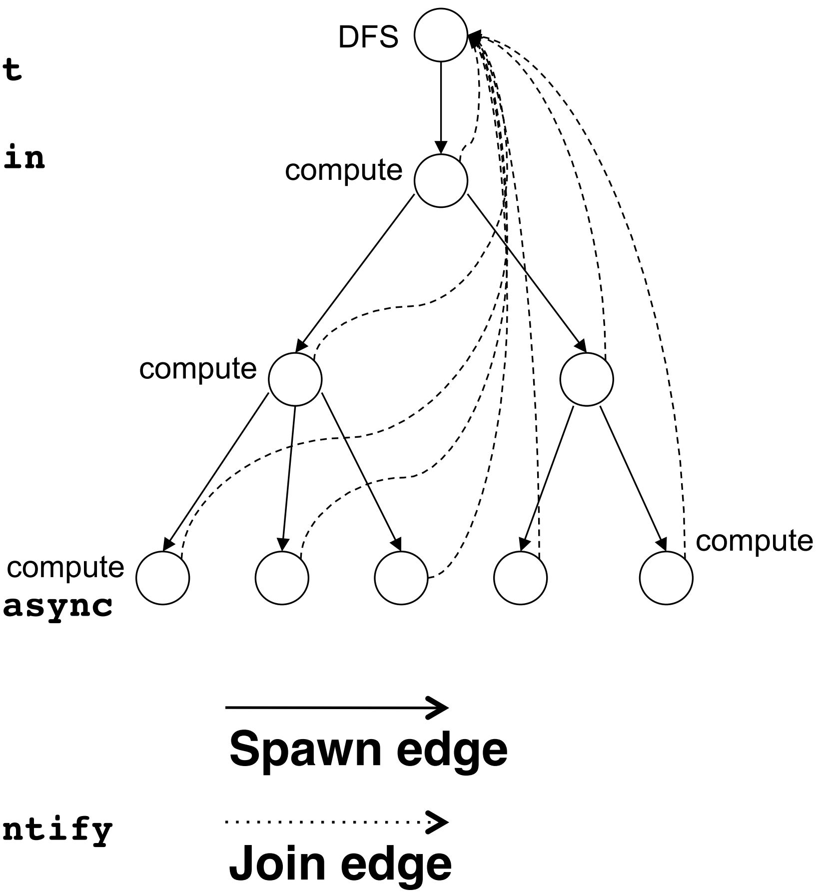
	boolean b =
	v.**compareAndSet**
	(expect,update);
		boolean b;
		`isolated`
		if (v.ref==expect) {v.ref=update; b=true;}
		else b = false;

AtomicReferenceArray **AtomicReferenceArray** (length) // init = null **AtomicIntegerArray** (arr)	Object o = v.**get**(i);	Object o; `isolated` o = v.arr[i];
	v.**set**(i,newRef);	`isolated` v.arr[i] = newRef;
	Object o = v.**getAndSet**(i,newRef);	Object o; `isolated` { o = v.arr[i]; v.arr[i] = newRef; }

	boolean b =
	v.**compareAndSet**
	(i,expect,update);
		boolean b;
		`isolated`
		if (v.arr[i]==expect) {v.arr[i]=update; b=true;}
		else b = false;

Parallel Depth-First Search Spanning Tree Example revisited

```
class V  {
  V [] neighbors; // adjacency list for input graph
  V parent;       // output value of parent in spanning tree
  boolean tryLabeling(V n) {
    isolated if (parent == null) parent=n;
    return parent == n;
  } // tryLabeling
  void compute() {
    for (int i=0; i<neighbors.length; i++) {
      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();
. . .
```



Parallel Depth-First Search Spanning Tree Example revisited

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

  } // tryLabeling
  void compute() {
    for (int i=0; i<neighbors.length; i++) {
      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();
. . .
```