COMP 322: Fundamentals of Parallel Programming

Lecture 21: Read-write Isolation, Atomic Variables

Vivek Sarkar, Shams Imam, Max Grossman
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu, shams.imam@twosigma.com, jmg3@rice.edu

http://comp322.rice.edu/

Worksheet #20 solution: Parallel Spanning Tree Algorithm

1. Insert finish, async, and isolated constructs (pseudocode is fine) to convert the sequential spanning tree algorithm below into a parallel algorithm

See slide 3, as well as the isolatedWithReturn() API in slide 4 for convenience in implementing the pseudocode.

2. Is it better to use a global isolated or an object-based isolated construct for the parallelization in question 1? If object-based is better, which object(s) should be included in the isolated list?

Object-based isolation should be better with a singleton object list containing the "this" object for the makeParent() method.

Parallel Spanning Tree Algorithm using isolated construct

```
class V  {
  V [] neighbors; // adjacency list for input graph
  V parent; // output value of parent in spanning tree
  boolean makeParent(final V n) {
    return isolatedWithReturn(this, () -> {
      if (parent == null) { parent = n; return true; }
      else return false; // return true if n became parent
    });
  } // makeParent
  void compute() {
    for (int i=0; i<neighbors.length; i++) {
      final V child = neighbors[i];
      if (child.makeParent(this))
        async(() -> { child.compute(); });
     }
  } // compute
} // class V
. . .
root.parent = root; // Use self-cycle to identify root
finish(() -> { root.compute(); });
. . .
```

HJ isolatedWithReturn construct

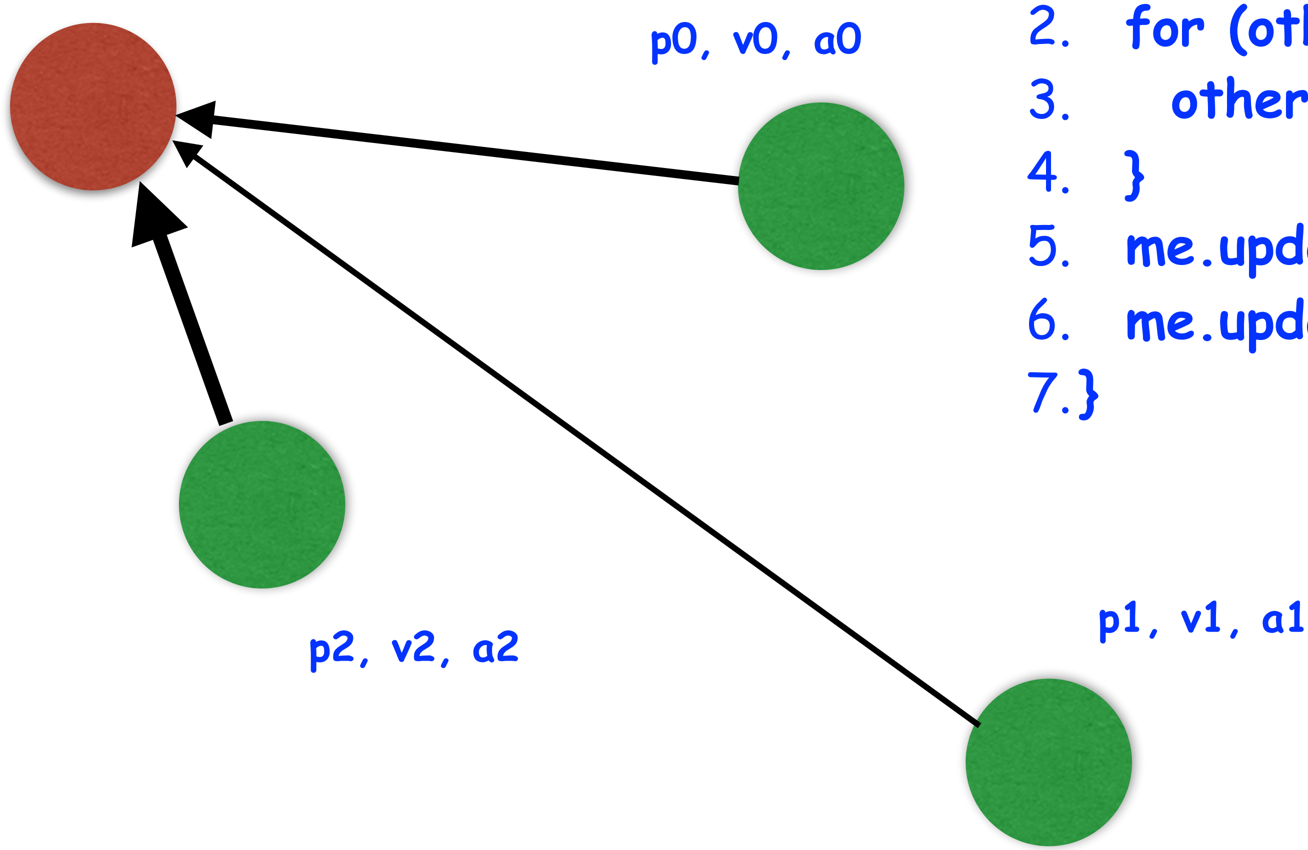
```
// <body> must contain return statement
isolatedWithReturn (obj1, obj2, …, () -> <body> );
```

Motivation: isolated() construct cannot modify local variables due to restrictions imposed by Java 8 lambdas

- Workaround 1: use isolated() and modify objects rather than local variables
  - Pro: code can be easier to understand than modifying local variables
  - Con: source of errors if multiple tasks read/write same object
- Workaround 2: use isolatedWithReturn()
  - Pro: cleaner than modifying local variables
  - Con: can only return one value

Motivation for Read-Write Object-based isolation

NBody Simulator




```
1. for (me : particles) {
2.   for (otherParticle : particles) {
3.     otherParticle.updateAccel(me)
4.   }
5.   me.updateVelocity()
6.   me.updatePosition()
7.}
```

Motivation for Read-Write Object-based isolation

<u>**NBody Simulator**</u>

```
for (me : particles) {
  for (otherParticle : particles) {
    otherParticle.updateAccel(me)
  }
  me.updateVelocity()
  me.updatePosition()
}

void updateAccel(Point other) {
  this.acceleration = …
}

void updatePosition() {
  this.position += …;
}
```

```
forall (me : particles) {
  for (otherParticle : particles) {
    otherParticle.updateAccel(me)
  }
  me.updateVelocity()
  me.updatePosition()
}

void updateAccel(Point other) {
  this.acceleration = …
}

void updatePosition() {
  this.position += …
}
```

Datarace!

Motivation for Read-Write Object-based isolation

NBody Simulator

```
forall (me : particles) {
  for (otherParticle : particles) {
    otherParticle.updateAccel(me)
  }
  me.updateVelocity()
  me.updatePosition()
}

void updateAccel(Point other) {
  this.acceleration = …
}

void updatePosition() {
  this.position += …
}
```

Datarace!

```
void updateAccel(Point other) {
  isolated (this) {
    this.acceleration = …
  }
}

void updatePosition() {
  isolated (this) {
    this.position += …
  }
}
```

What if there are many points calling updateAccel() on the same object?

Motivation for Read-Write Object-based isolation

NBody Simulator

```
void updateAccel(Point other) {
   isolated (this) {
      this.acceleration = …
   }
}

void updatePosition() {
   isolated (this) {
      this.position += …
   }
}
```

Demo!

What if there are many points calling updateAccel() on the same object?

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()	int j = v.**getAndSet**(newVal);	int j; **isolated** (v) { j = v.val; v.val = newVal; }
// init = 0	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 makeParent(final V n) {
    // compareAndSet() is a more efficient implementation of
    // object-based isolation
    return parent.compareAndSet(null, n);
  } // makeParent
  void compute() {
    for (int i=0; i<neighbors.length; i++) {
      final V child = neighbors[i];
      if (child.makeParent(this))
        async(() -> { child.compute(); }); // escaping async
     }
  } // compute
} // class V
. . .
root.parent = root; // Use self-cycle to identify root
finish(() -> { root.compute(); });
. . .
```