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

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

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

NBody Simulator

p0, v0, a0

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

p1, v1, a1

p2, v2, a2
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. }
8.
9. void updateAccel(Point other) {
10.   this.acceleration = ...
11. }
12.
13. void updatePosition() {
14.   this.position += ...;
15. }

1. forall (me : particles) {
2.   for (otherParticle : particles) {
3.     otherParticle.updateAccel(me)
4.   }
5.   me.updateVelocity()
6.   me.updatePosition()
7. }
8.
9. void updateAccel(Point other) {
10.   this.acceleration = ...
11. }
12.
13. void updatePosition() {
14.   this.position += ...
15. }

Datarace!
Motivation for Read-Write Object-based isolation

NBody Simulator

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

What if there are many points calling updateAccel() on the same object?
Motivation for Read-Write Object-based isolation

NBody Simulator

1. void updateAccel(Point other) {
2.   isolated (this) {
3.     this.acceleration = …
4.   }
5. }
6.
7. void updatePosition() {
8.   isolated (this) {
9.     this.position += …
10. }
11. }

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

Demo!
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**

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

- **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

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
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.

<table>
<thead>
<tr>
<th>j.u.c.atomic Class and Constructors</th>
<th>j.u.c.atomic Methods</th>
<th>Equivalent HJ isolated statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>AtomicInteger</td>
<td>int j = v.get();</td>
<td>int j; isolated (v) j = v.val;</td>
</tr>
<tr>
<td></td>
<td>v.set(newVal);</td>
<td>isolated (v) v.val = newVal;</td>
</tr>
<tr>
<td>AtomicInteger()</td>
<td>int j = v.getAndSet(newVal);</td>
<td>int j; isolated (v) { j = v.val; v.val = newVal; }</td>
</tr>
<tr>
<td>// init = 0</td>
<td>int j = v.addAndGet(delta);</td>
<td>isolated (v) { v.val += delta; j = v.val; }</td>
</tr>
<tr>
<td>AtomicInteger(init)</td>
<td>int j = v.getAndAdd(delta);</td>
<td>isolated (v) { j = v.val; v.val += delta; }</td>
</tr>
<tr>
<td></td>
<td>boolean b =</td>
<td>boolean b;</td>
</tr>
<tr>
<td></td>
<td>v.compareAndSet(</td>
<td>isolated (v)</td>
</tr>
<tr>
<td></td>
<td>(expect,update);</td>
<td>if (v.val==expect) {v.val=update; b=true;}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else b = false;</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>j.u.c.atomic Class and Constructors</th>
<th>j.u.c.atomic Methods</th>
<th>Equivalent HJ isolated statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>AtomicReference</td>
<td>Object o = v.get(); v.set(newRef);</td>
<td>Object o; isolated (v) o = v.ref;</td>
</tr>
<tr>
<td>AtomicReference()</td>
<td></td>
<td>isolated (v) v.ref = newRef;</td>
</tr>
<tr>
<td>AtomicReference(init)</td>
<td>Object o = v.getAndSet(newRef);</td>
<td>Object o; isolated (v) { o = v.ref; v.ref = newRef; }</td>
</tr>
<tr>
<td></td>
<td>boolean b = v.compareAndSet</td>
<td>boolean b; isolated (v)</td>
</tr>
<tr>
<td></td>
<td>(expect,update);</td>
<td>if (v.ref==expect) {v.ref=update; b=true;}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else b = false;</td>
</tr>
</tbody>
</table>

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.
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(); });