COMP 322: Fundamentals of Parallel Programming

Lecture 21: Read-Write Isolation, Review of Phasers

Zoran Budimlić and Mack Joyner
{zoran, mjoyner}@rice.edu

http://comp322.rice.edu

Worksheet #20 solution: Sequential->Parallel Spanning Tree Algorithm

1. Insert finish, async, and isolated constructs (pseudocode is fine) to convert the sequential spanning tree algorithm on the other side 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.

Worksheet #20: Sequential->Parallel Spanning Tree Algorithm using object-based 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) {
    final boolean[] result = new boolean[1];
    isolated(this, () -> {
      if (parent == null) { parent = n; result[0] = true; }
      else result[0] = false; // return true if n became parent
    });
    return result[0];
  } // 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

Worksheet #20: Sequential->Parallel Spanning Tree Algorithm using object-based 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(); });
. . .
```

java.util.concurrent.AtomicInteger methods and their equivalent object-based isolated constructs (Lecture 20)

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.

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
```

Atomic Variables represent a special (and more efficient) case of Object-based isolation

```
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(); });
. . .
```

Motivation for Read-Write Object-based isolation

Sorted List example

```
public boolean contains(Object object) {
    // Observation: multiple calls to contains() should not
    // interfere with each other
    return isolatedWithReturn(this, () -> {
      Entry pred, curr;
      ...
      return (key == curr.key);
   });
}

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

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;
  });
}
```

The world according to Module 1 without & with Phasers

- All the non-phaser parallel constructs that we learned focused on task creation and termination
  - async creates a task
    - forasync creates a set of tasks specified by an iteration region
  - finish waits for a set of tasks to terminate
    - forall (like "finish forasync") creates and waits for a set of tasks specified by an iteration region
  - future get() waits for a specific task to terminate
  - asyncAwait() waits for a set of DataDrivenFuture values before starting
- Motivation for phasers
  - Deterministic directed synchronization within tasks for barriers, point-to-point synchronization, pipelining
  - Separate from synchronization associated with task creation and termination
  - next operations are much more efficient than task creation/termination (async/finish), but they *only help reduce overhead if you perform multiple next operations in a single task*

Pipeline Parallelism: Another Example of Point-to-point Synchronization (Recap)

- Medical imaging pipeline with three stages
  1. Denoising stage generates a sequence of results, one per image.
  2. Registration stage's input is Denoising stage's output.
  3. Segmentation stage's input is Registration stage's output.
- Even though the processing is sequential for a single image, *pipeline parallelism* can be exploited via point-to-point synchronization between neighboring stages

Implementation of Medical Imaging Pipeline

```
final List<PhaserPair> phList1 = Arrays.asList(ph0.inMode(PhaserMode.SIG));
final List<PhaserPair> phList2 = Arrays.asList(ph0.inMode(PhaserMode.WAIT), ph1.inMode(PhaserMode.SIG));
final List<PhaserPair> phList3 = Arrays.asList(ph1.inMode(PhaserMode.WAIT));

asyncPhased(phList1, () -> { // DENOISE stage
    for (int i = 0; i < n; i++) {
          doWork(1);
          signal(); // same as ph0.signal(); as only ph0 is registered in this async
    }
 });

 asyncPhased(phList2, () -> { // REGISTER stage
     for (int i = 0; i < n; i++) {
           ph0.doWait();  // WARNING: Explicit calls to doWait() can lead to deadlock in general
           doWork(1);
           ph1.signal();
     }
 });

 asyncPhased(phList3, () -> { // SEGMENT stage
     for (int i = 0; i < n; i++) {
           ph1.doWait();
           doWork(1);
```

Announcements

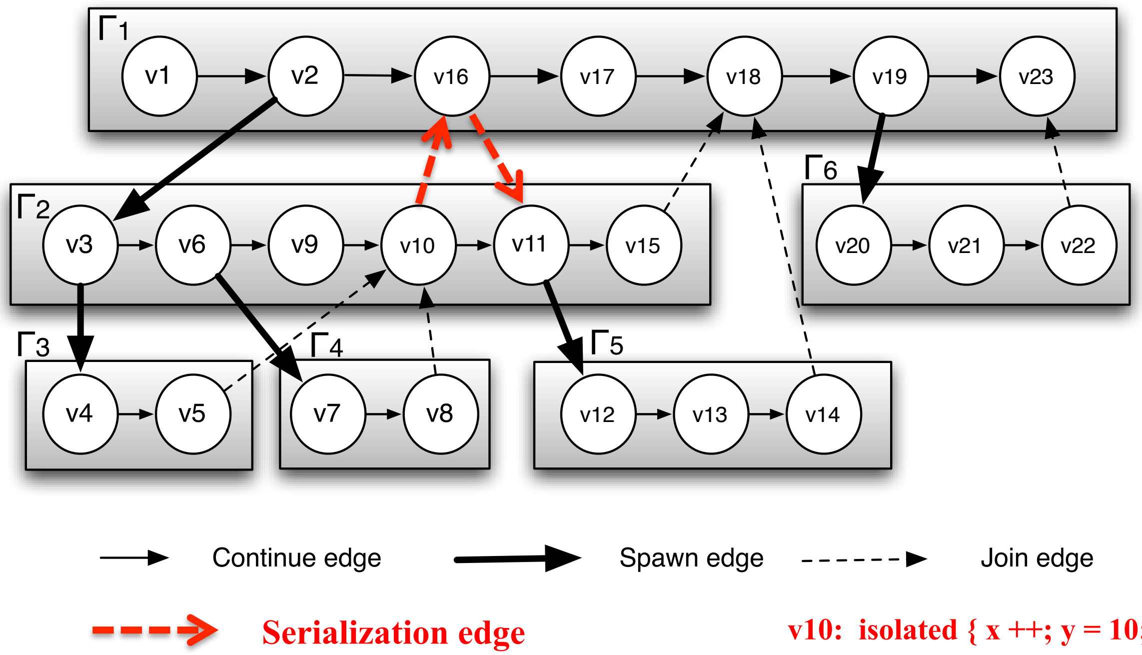
- Reminder: Quiz for Unit 4 is due today!
- Quiz for Unit 5 will be out today
- Checkpoint #2 for Homework 3 will be due by Wednesday, March 6th, and the entire homework is due by March 20th
- Scope of the final exam (Exam 2) will be limited to Lectures 19 - 38

Serialized Computation Graph for Isolated Constructs (Recap)

- Model each instance of an isolated construct as a distinct step (node) in the CG.
- Need to reason about the *order* in which interfering isolated constructs are executed
  - Complicated because the order of isolated constructs may vary from execution to execution
- Introduce Serialized Computation Graph (SCG) that includes a specific ordering of all interfering isolated constructs.
  - SCG consists of a CG with additional serialization edges.
  - Each time an isolated step, S', is executed, we add a serialization edge from S to S' for each prior "interfering" isolated step, S
    - Two isolated constructs always interfere with each other
    - Interference of "object-based isolated" constructs depends on intersection of object sets
    - Serialization edge is not needed if S and S' are already ordered in CG
  - An SCG represents a set of schedules in which all interfering isolated constructs execute in the same order.

Example of Serialized Computation Graph with Serialization Edges for v10-v16-v11 order (Recap)

Data race definition can be applied to Serialized Computation Graphs (SCGs) just like regular CGs



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

– Need to consider all possible orderings of interfering isolated constructs to establish data race freedom

Worksheet #21a: Abstract Metrics with Isolated Constructs

Name: ______________________________ Netid: _________________

Compute the WORK and CPL metrics for this program with a global isolated construct. Indicate if your answer depends on the execution order of isolated constructs. Since there may be multiple possible computation graphs (based on serialization edges), try and pick the worst-case CPL value across all computation graphs.

```
finish(() -> {
     for (int i = 0; i < 5; i++) {
       async(() -> {
         doWork(2);
         isolated(() -> { doWork(1); });
         doWork(2);
       }); // async
     } // for
 }); // finish
```

Worksheet #21b: Abstract Metrics with Object-based Isolated Constructs

Compute the WORK and CPL metrics for this program with an object-based isolated construct. Indicate if your answer depends on the execution order of isolated constructs. Since there may be multiple possible computation graphs (based on serialization edges), try and pick the worst-case CPL value across all computation graphs.

```
finish(() -> {
     // Assume X is an array of distinct objects
     for (int i = 0; i < 5; i++) {
       async(() -> {
         doWork(2);
         isolated(X[i], X[i+1],
                  () -> { doWork(1); });
         doWork(2);
       }); // async
     } // for
 }); // finish
```