COMP 322: Fundamentals of Parallel Programming

Lecture 20: Critical Sections and the Isolated Construct

Instructors: Vivek Sarkar, Mack Joyner
Department of Computer Science, Rice University
{vsarkar, mjoyner}@rice.edu

http://comp322.rice.edu/

COMP 322 Lecture 20 1 March 2017

Worksheet #19: Critical Path Length for Computation with Signal Statement

Name: ______________________ Netid: ______________________

Compute the WORK and CPL values for the program shown below. How would they be different if the signal() statement was removed? (Hint: draw a computation graph as in slide 11)

WORK = 204, CPL = 102. If the signal() is removed, CPL = 202.

```
finish(() -> {
  final HjPhaser ph = newPhaser(SIG_WAIT);
  asyncPhased(ph.inMode(SIG_WAIT), () -> { // Task T1
    A(0); doWork(1);   // Shared work in phase 0
    signal();
    B(0); doWork(100); // Local work in phase 0
    next(); // Wait for T2 to complete shared work in phase 0
    C(0); doWork(1);
  });
  asyncPhased(ph.inMode(SIG_WAIT), () -> { // Task T2
    A(1); doWork(1);   // Shared work in phase 0
    next(); // Wait for T1 to complete shared work in phase 0
    C(1); doWork(1);
    D(1); doWork(100); // Local work in phase 0
  });
}); // finish
```

Formal Definition of Data Races (Recap)

Formally, a data race occurs on location L in a program execution with computation graph CG if there exist steps (nodes) S1 and S2 in CG such that:

1. S1 does not depend on S2 and S2 does not depend on S1 i.e., there is no path of dependence edges from S1 to S2 or from S2 to S1 in CG, and
2. Both S1 and S2 read or write L, and at least one of the accesses is a write.

However, there are many cases in practice when two tasks may legitimately need to perform conflicting accesses to shared locations without incurring data races

- How should conflicting accesses be handled in general, when outcome may be nondeterministic?

⇒ Focus of Module 2: “Concurrency” (nondeterministic parallelism)

Example of two tasks performing conflicting accesses --- need for “mutual exclusion”

```
class DoublyLinkedListNode {
 DoublyLinkedListNode prev, next;
 . . .
 void delete() {
   { // start of desired mutual exclusion region
     this.prev.next = this.next;
     this.next.prev = this.prev;
   } // end of desired mutual exclusion region
   . . . // remaining code in delete() that does not need mutual exclusion
  }
} // DoublyLinkedListNode
. . .
static void deleteTwoNodes(final DoublyLinkedListNode L) {
  finish(() -> {
    DoublyLinkedListNode second = L.next;
    DoublyLinkedListNode third = second.next;
    async(() -> { second.delete(); });
    async(() -> { third.delete(); }); // conflicts with previous async
  });
}
```

How to enforce mutual exclusion?

- The predominant approach to ensure mutual exclusion proposed many years ago is to enclose the code region in a critical section.
  - "In concurrent programming a critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution. A critical section will usually terminate in fixed time, and a thread, task or process will have to wait a fixed time to enter it (aka bounded waiting). Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use, for example a semaphore."
  - Source: http://en.wikipedia.org/wiki/Critical_section

HJ isolated construct

`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 — which we will learn later) can lead to a deadlock, if used incorrectly

Use of isolated to fix previous example with conflicting accesses

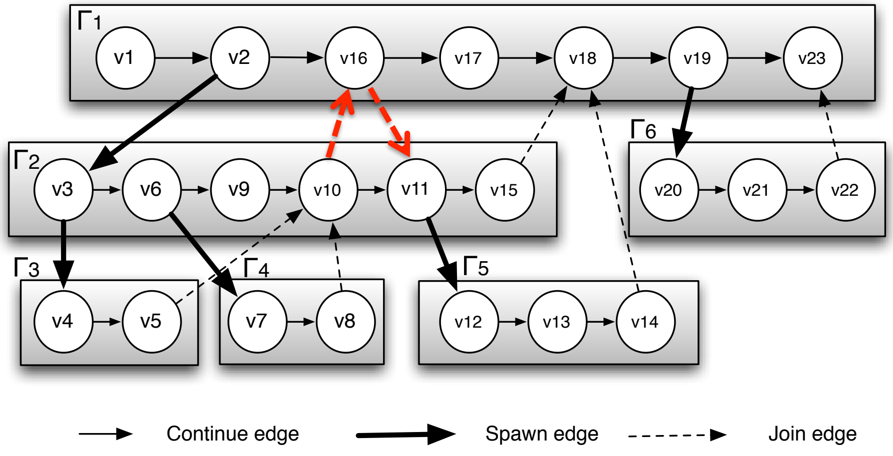
```
class DoublyLinkedListNode {
 DoublyLinkedListNode prev, next;
 . . .
 void delete() {
   isolated(() -> { // start of desired mutual exclusion region
     this.prev.next = this.next;
     this.next.prev = this.prev;
   }); // end of desired mutual exclusion region
   . . . // other code in delete() that does not need mutual exclusion
 }
} // DoublyLinkedListNode
. . .
static void deleteTwoNodes(final DoublyLinkedListNode L) {
 finish(() -> {
   DoublyLinkedListNode second = L.next;
   DoublyLinkedListNode third = second.next;
   async(() -> { second.delete(); });
   async(() -> { third.delete(); }); // conflicts with previous async
 });
}
```

Serialized Computation Graph for Isolated Constructs

- 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

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

---

Object-based isolation

```
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
  - —Serialization edges are only added between isolated steps with at least one common object (non-empty intersection of objstec lists)
  - —Standard isolated is equivalent to “isolated(*)” by default i.e., isolation across all objects
- Inner isolated constructs are redundant — they are not allowed to “add” new objects

Pros and Cons of Object-Based Isolation

- Pros
  - —Increases parallelism relative to critical section approach
  - —Simpler approach than “locks” (which we will learn later)
  - —Deadlock-freedom property is still guaranteed

- Cons
  - —Programmer needs to worry about getting the object list right
  - —Objects in object list can only be specified at start of the isolated construct (new objects cannot be added later on)
  - — Large object lists can contribute to large overheads

DoublyLinkedListNode Example revisited with Object-Based Isolation

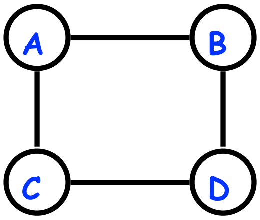
```
class DoublyLinkedListNode {
 DoublyLinkedListNode prev, next;
 . . .
 void delete() {
   isolated(this.prev, this, this.next, () -> { // object-based isolation
     this.prev.next = this.next;
     this.next.prev = this.prev;
   });
   . . .
 }
} // DoublyLinkedListNode
. . .
static void deleteTwoNodes(final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

Spanning Tree Definition

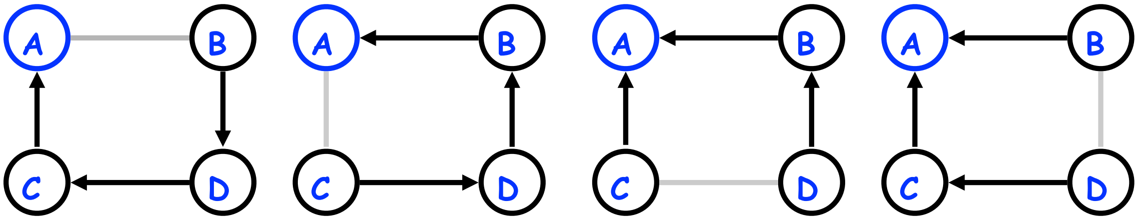
- A spanning tree, T, of a connected undirected graph G is
  - rooted at some vertex of G
  - defined by a parent map for each vertex
  - contains all the vertices of G, i.e. spans all vertices
  - contains exactly |v| - 1 edges
    - adding any other edge will create a cycle
  - contains no cycles (a tree!)
    - implies the edges involved in T is a subset of the edges in G

An Example Graph with 4 possible spanning trees rooted at vertex A

Example Undirected Graph:



Spanning Trees (edges are directed from child to parent):



Vertex	Parent
A	null
B	D
C	A
D	C

Vertex	Parent
A	null
B	A
C	D
D	B

Vertex	Parent
A	null
B	A
C	A
D	B

Vertex	Parent
A	null
B	A
C	A
D	C

Sequential Parallel Spanning Tree Algorithm

```
class V  {
  V [] neighbors; // adjacency list for input graph
  V parent; // output value of parent in spanning tree

  boolean makeParent(V n) {
    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))
        child.compute(); // recursive call
     }
  } // compute
} // class V
. . . // main program
root.parent = root; // Use self-cycle to identify root
root.compute();
. . .
```

---

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`

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.

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
```