
COMP 322: Fundamentals of Parallel Programming

Lecture 19: Critical Sections and the Isolated Construct

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

<https://wiki.rice.edu/confluence/display/PARPROG/COMP322>



Summary of Module 1: Deterministic Shared-Memory Parallelism

- **Serializable subset of HJ**
 - { async, finish, future, forasync }
 - Erasure property: any HJ program written using the above constructs can be converted to an equivalent sequential program by “erasing” all parallel constructs i.e., by removing async & finish, and replacing future & forasync by equivalent sequential constructs
- **Deadlock-free subset of HJ**
 - { next, barriers, phasers, forall, async phased } + Serializable subset
 - Deadlock-freedom property: any HJ program written using the above constructs is guaranteed to never deadlock
- **Deterministic subset of HJ**
 - { data driven futures, async await } + Deadlock-free subset
 - Data-race-free determinism property: if any HJ program written using the above constructs is guaranteed to be data-race-free for a given input, then it must also be functionally deterministic and structurally deterministic for that input i.e., all executions with the same input must generate the same output AND the same computation graph



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

- Special cases with determinism guarantees: finish accumulators, phaser accumulators
- How should conflicting accesses be handled in general, when outcome may be nondeterministic?



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

```
1. class DoublyLinkedListNode {
2.     DoublyLinkedListNode prev, next;
3.     . . .
4.     void delete() {
5.         { // start of desired mutual exclusion region
6.             this.prev.next = this.next;
7.             this.next.prev = this.prev;
8.         } // end of desired mutual exclusion region
9.         . . . // remaining code in delete() that does not need mutual exclusion
10.    }
11. } // DoublyLinkedListNode
12. . . .
13. static void deleteTwoNodes(final DoublyLinkedListNode L) {
14.     finish(() -> {
15.         DoublyLinkedListNode second = L.next;
16.         DoublyLinkedListNode third = second.next;
17.         async(() -> { second.delete(); });
18.         async(() -> { third.delete(); }); // conflicts with previous async
19.     });
20. }
```



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 must 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) can lead to a deadlock, if used incorrectly



Use of isolated to fix previous example with conflicting accesses

```
1. class DoublyLinkedListNode {
2.     DoublyLinkedListNode prev, next;
3.     . . .
4.     void delete() {
5.         isolated(() -> { // start of desired mutual exclusion region
6.             this.prev.next = this.next;
7.             this.next.prev = this.prev;
8.         }); // end of desired mutual exclusion region
9.         . . . // other code in delete() that does not need mutual exclusion
10.    }
11. } // DoublyLinkedListNode
12. . . .
13. static void deleteTwoNodes(final DoublyLinkedListNode L) {
14.     finish(() -> {
15.         DoublyLinkedListNode second = L.next;
16.         DoublyLinkedListNode third = second.next;
17.         async(() -> { second.delete(); });
18.         async(() -> { third.delete(); }); // conflicts with previous async
19.     });
20. }
```



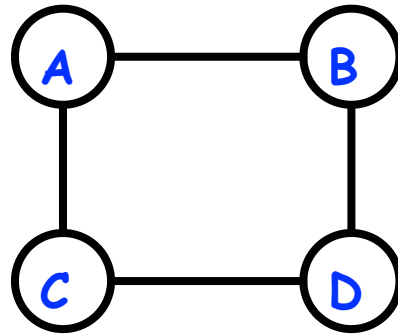
Spanning Tree Definition

- 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

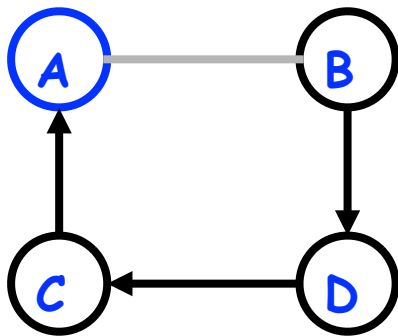


An Example Graph with 4 possible spanning trees rooted at vertex A

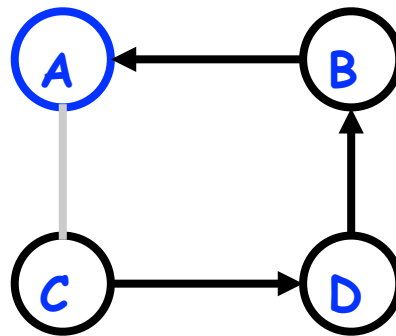
Example Graph:



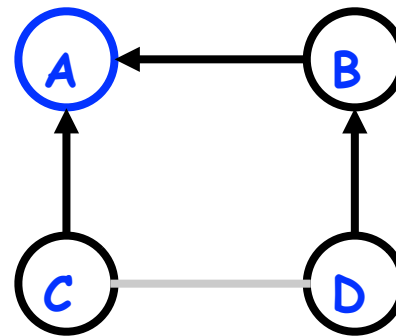
Spanning Trees:



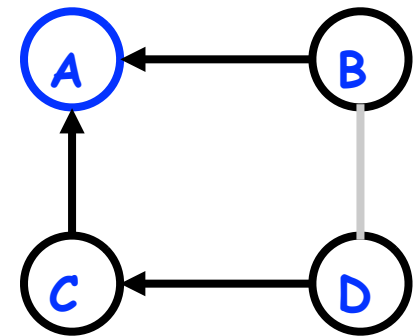
Vertex	Parent
A	null
B	D
C	A
D	C



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



Parallel Spanning Tree Algorithm using isolated construct

```
1. class V {
2.     V [] neighbors; // adjacency list for input graph
3.     V parent; // output value of parent in spanning tree
4.     boolean tryLabeling(final V n) {
5.         return isolatedWithReturn(() -> {
6.             if (parent == null) parent = n;
7.             return parent == n; // return true if n became parent
8.         });
9.     } // tryLabeling
10.    void compute() {
11.        for (int i=0; i<neighbors.length; i++) {
12.            final V child = neighbors[i];
13.            if (child.tryLabeling(this))
14.                async(() -> { child.compute(); }); // escaping async
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 (() -> <body>);

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: many potential errors if multiple tasks read/write same object
- **Workaround 2: use isolatedWithReturn()**
 - Pro: cleaner than modifying local variables
 - Con: can only return one value

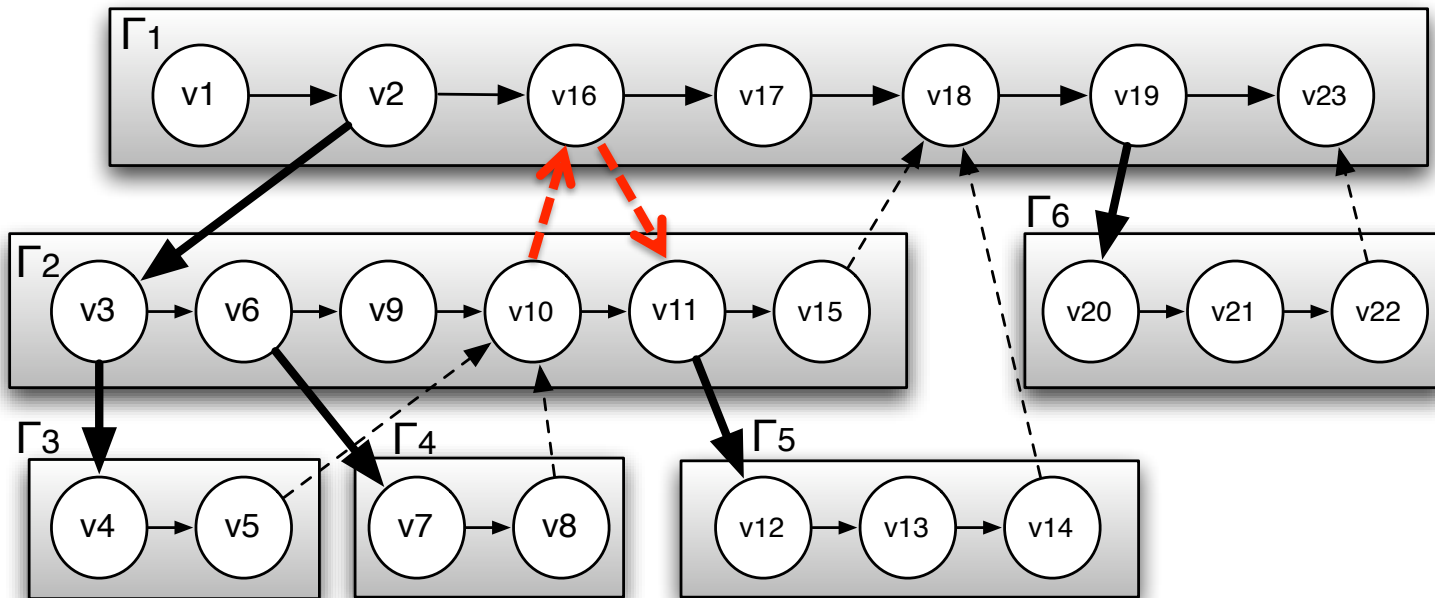


Serialized Computation Graph for Isolated Constructs

- 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 executions in which all interfering isolated constructs execute in the same order.



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



→ Continue edge → Spawn edge - - - - - Join edge

- - - - - → **Serialization edge**

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

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

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



Object-based isolation in HJ

`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 non-empty intersection in their object lists
 - **Standard isolated is equivalent to “isolated(*)” by default i.e., isolation across all objects**
- Implementation can choose to distinguish between read/write accesses for further parallelism
 - **Current HJ implementation supports object-based isolation, but does not exploit read/write distinction**



DoublyLinkedListNode Example revisited with Object-Based Isolation

```
1. class DoublyLinkedListNode {
2.     DoublyLinkedListNode prev, next;
3.     . . .
4.     void delete() {
5.         isolated(this.prev, this, this.next, () -> { // object-based isolation
6.             this.prev.next = this.next;
7.             this.next.prev = this.prev;
8.         });
9.         . . .
10.    }
11. } // DoublyLinkedListNode
12. . . .
13. static void deleteTwoNodes(final DoublyLinkedListNode L) {
14.     finish(() -> {
15.         DoublyLinkedListNode second = L.next;
16.         DoublyLinkedListNode third = second.next;
17.         async(() -> { second.delete(); });
18.         async(() -> { third.delete(); });
19.     });
```



Announcements

- **Homework 3 is due by 11:59pm on Wednesday, March 12, 2014**
 - **Help session today at 8pm in Duncan Hall 1049**
- **Addition of slip days (as in COMP 221)**
 - **All students will be given 3 slip days to use for the rest of the semester. When you use a slip day, you will receive up to 24 additional hours to complete the assignment. You may use these slip days in any way you see fit (3 days on one assignment, 1 day each on 3 assignments, etc.). The only requirement for use of your slip days is that you e-mail the instructors prior to the time the assignment is due.**
 - **When slip days are used, you should clearly indicate so at the beginning of the assignment writeup.**



Worksheet #19:

Insertion of isolated for correctness

Name 1: _____

Netid: _____

The goal of `IsolatedPRNG` is to implement a single Pseudo Random Number Generator object that can be shared by multiple tasks. Show the isolated construct(s) that you can insert in method `nextSeed()` to avoid data races and guarantee proper semantics.

```
class IsolatedPRNG {
  private int seed;
  public int nextSeed() {
    int retVal;

    retVal = seed;

    seed = nextInt(retVal);

    return retVal;
  } // nextSeed()
  . . .
} // IsolatedPRNG
```

```
main() { // Pseudocode
  // Initial seed = 1
  IsolatedPRNG r = new IsolatedPRNG(1);
  async(() -> { print r.nextSeed(); ... });
  async(() -> { print r.nextSeed(); ... });
} // main()
```

