COMP 322: Fundamentals of Parallel Programming

Lecture 20: Parallel Spanning Tree, Atomic Variables

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu

Worksheet #19 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.

Answer: WORK = 25, CPL = 7.

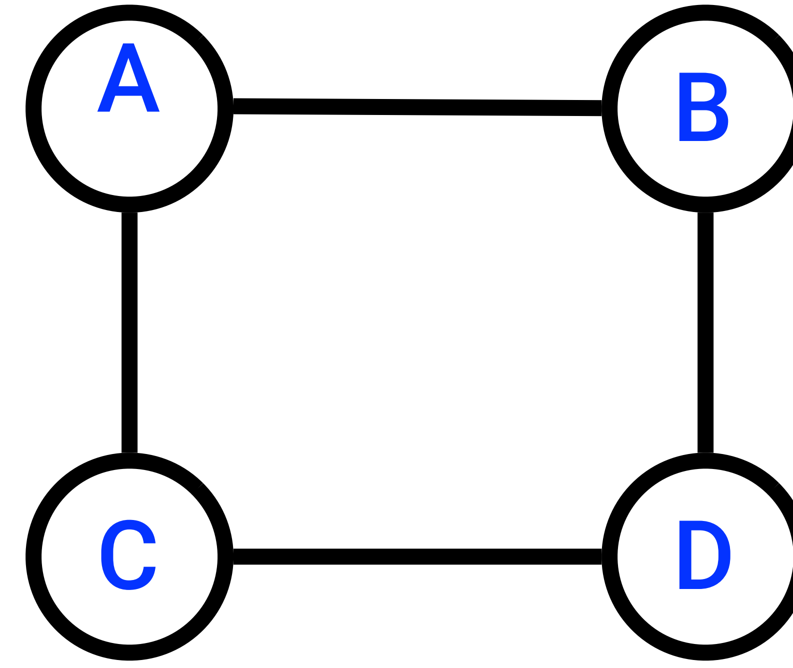
```
1.    finish(() -> {
2.          // Assume X is an array of distinct objects
3.          for (int i = 0; i < 5; i++) {
4.            async(() -> {
5.              doWork(2);
6.              isolated(X[i], X[i+1],
7.                       () -> { doWork(1); });
8.              doWork(2);
9.            }); // async
10.         } // for
11.     }); // finish
```

Spanning Tree Definition

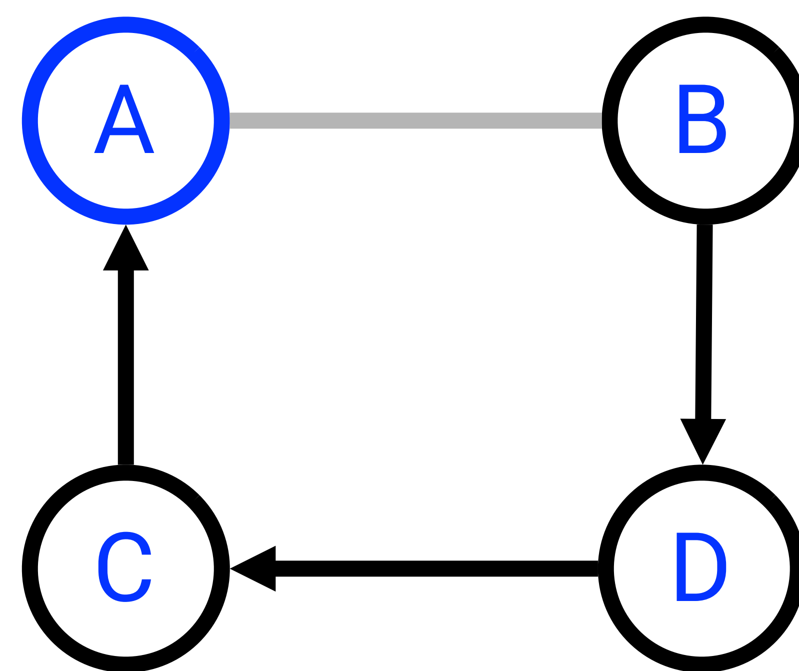
- A spanning tree, T, of a connected undirected graph G is
  - rooted at some vertex of G
  - defined by a parent map for each vertex
  - contains all the vertices of G, i.e. spans all vertices
  - contains exactly |v| - 1 edges
    - adding any other edge will create a cycle
  - contains no cycles (a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

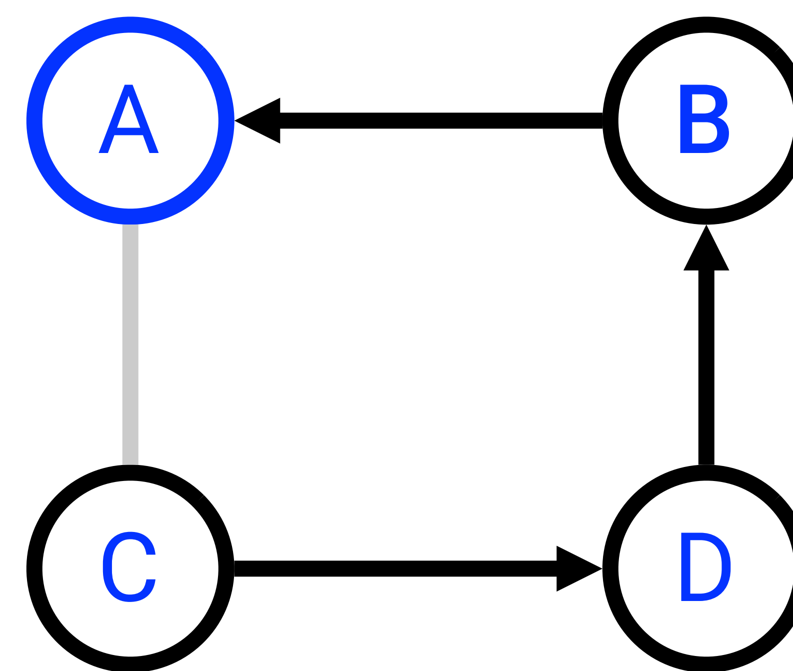
An Example Graph with 4 possible spanning trees rooted at vertex A

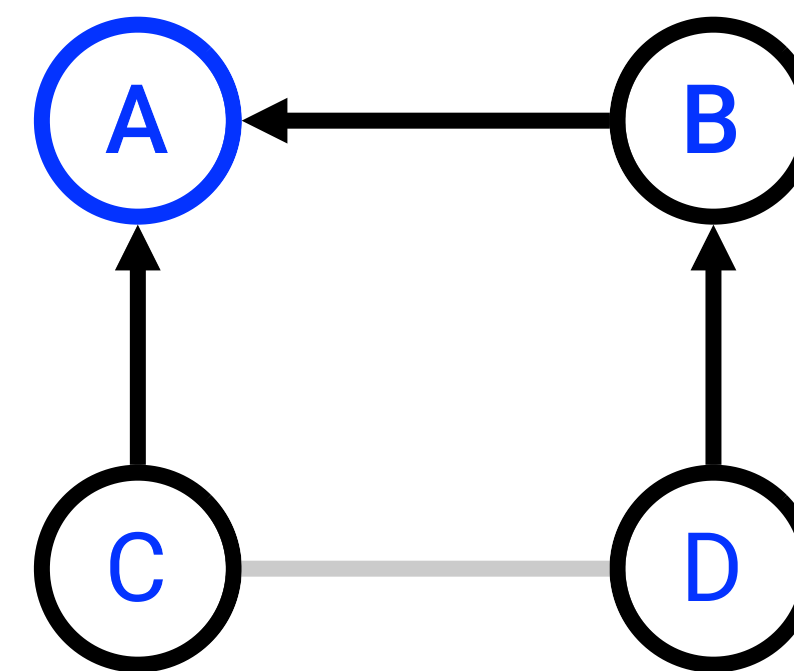
Example Undirected Graph:

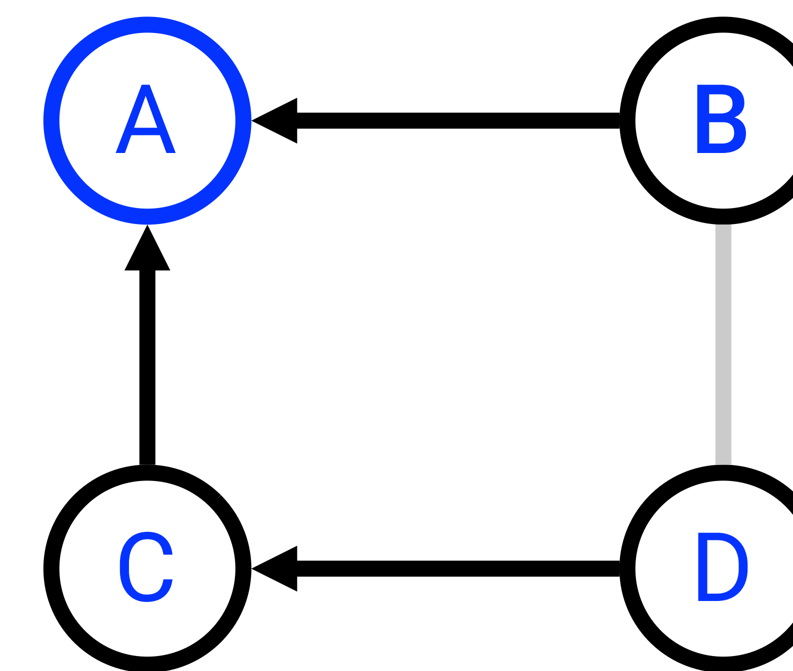


Spanning Trees (edges are directed from child to parent):









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

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

4.    boolean makeParent(V n) {
5.      if (parent == null) { parent = n; return true; }
6.      else return false; // return true if n became parent
7.    } // makeParent

8.    void compute() {
9.      for (int i=0; i<neighbors.length; i++) {
10.       final V child = neighbors[i];
11.       if (child.makeParent(this))
12.         child.compute(); // recursive call
13.      }
14.   } // compute
15. } // class V
16. . . . // main program
17. root.parent = root; // Use self-cycle to identify root
18. root.compute();
19. . . .
```

Exercise: Parallel Spanning Tree Algorithm using object-based 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 makeParent(V n) {
5.      if (parent == null) { parent = n; return true; }
6.      else return false; // return true if n became parent
7.    } // makeParent

8.    void compute() {
9.      for (int i=0; i<neighbors.length; i++) {
10.       final V child = neighbors[i];
11.       if (child.makeParent(this))
12.         child.compute(); // recursive call
13.      }
14.   } // compute
15. } // class V
16. . . . // main program
17. root.parent = root; // Use self-cycle to identify root
18. root.compute();
19. . . .
```

Exercise: Parallel Spanning Tree Algorithm using object-based 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 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. . . .
```

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.

Exercise: Work-Sharing Pattern using AtomicInteger

```
1.  import java.util.concurrent.atomic.AtomicInteger;
2.  . . .
3.  String[] X = ... ; int numTasks = …; int j;
4.  int[] taskId = new int[X.length];
5.. . .
6.  finish(() -> {
7.    for (int i=0; i<numTasks; i++ )
8.      async(() -> {
9.      do {
10.         j = j + 1;
11.         // check if at end of X
12.         if (j >= X.length) break;
13.         taskId[j] = i; // Task i processes string X[j]
14.         . . .
15.       } while (true);
16.     });
17.}); // finish-for-async
```

Exercise: Work-Sharing Pattern using AtomicInteger

```
1.  import java.util.concurrent.atomic.AtomicInteger;
2.  . . .
3.  String[] X = ... ; int numTasks = …; int j;
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.         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
```

Exercise: 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
```

Motivation for Read-Write Object-based isolation

```
1. Sorted List example
2.  public boolean contains(Object object) {
3.     // Observation: multiple calls to contains() should not
4.     // interfere with each other
5.     return isolatedWithReturn(this, () -> {
6.       Entry pred, curr;
7.       ...
8.       return (key == curr.key);
9.    });
10. }
11.
12.  public int add(Object object) {
13.   return isolatedWithReturn(this, () -> {
14.     Entry pred, curr;
15.     ...
16.     if (...) return 1; else return 0;
17.   });
18. }
```

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

```
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. }
```

Announcements & Reminders

- Quiz for Unit 4 is due Friday, March 6th at 11:59pm

Worksheet #20: Sequential->Parallel Spanning Tree Algorithm

Insert finish, async, and atomic (includes a compareAndSet) constructs (pseudocode is fine) to convert the sequential spanning tree algorithm to a parallel algorithm

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