COMP 322: Fundamentals of Parallel Programming

Lecture 39: Review of Lectures 18 - 38 (Scope of Exam 2)

Instructors: Vivek Sarkar, Mack Joyner
Department of Computer Science, Rice University
{jmg3, vsarkar, mjoyner}@rice.edu

http://comp322.rice.edu/

HJ isolated construct
(Lecture 20 — Start of Module 2, Concurrency)

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

- **Nondeterminism — two isolated constructs may be executed in either order without a data race, but with a nondeterministic outcome**

- **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

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**

Worksheet #20: 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. AtomicReference methods and their equivalent isolated statements

j.u.c.atomic Class and Constructors	j.u.c.atomic Methods	Equivalent HJ isolated statements
AtomicReference	Object o = v.**get**();	Object o; isolated (v) o = v.ref;
	v.**set**(newRef);	isolated (v) v.ref = newRef;
AtomicReference() // init = null	Object o = v.**getAndSet**(newRef);	Object o; isolated (v) { o = v.ref; v.ref = newRef; }

AtomicReference(init)	boolean b =
	    v.**compareAndSet**
	    (expect,update);
		boolean b;
		isolated (v)
		    if (v.ref==expect) {v.ref=update; b=true;}
		    else b = false;

Methods in java.util.concurrent.AtomicReference class and their equivalent HJ object-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 for the reference.

Atomic Variables represent a special case of Object-based isolation

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

Read-Write Object-based isolation in HJ (Lecture 21)

`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
- <u>Sorted List example</u>

```
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. }
```

Worksheet #21a solution: Abstract Metrics with Isolated Construct

Q: 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.

```
1.    finish(() -> {
2.        for (int i = 0; i < 5; i++) {
3.            async(() -> {
4.                    doWork(2);
5.                    isolated(() -> { doWork(1); });
6.                    doWork(2);
7.               }); // async
8.        } // for
9.    }); // finish
```

Answer: WORK = 25, CPL = 9. These metrics do not depend on the execution order of isolated constructs.

Worksheet #21b solution:
Abstract Metrics with Object-Based Isolated Construct

Q: Compute the WORK and CPL metrics for this program with an <u>object-based isolated</u> construct. Indicate if your answer depends on the execution order of isolated constructs.

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

Answer: WORK = 25, CPL depends on execution order. Best-case CPL = 6, worst-case CPL = 7. (Worst-case example: if A_1, A_4 execute in parallel first, then the isolated sections in A_2, A_3 must be serialized thereafter.)
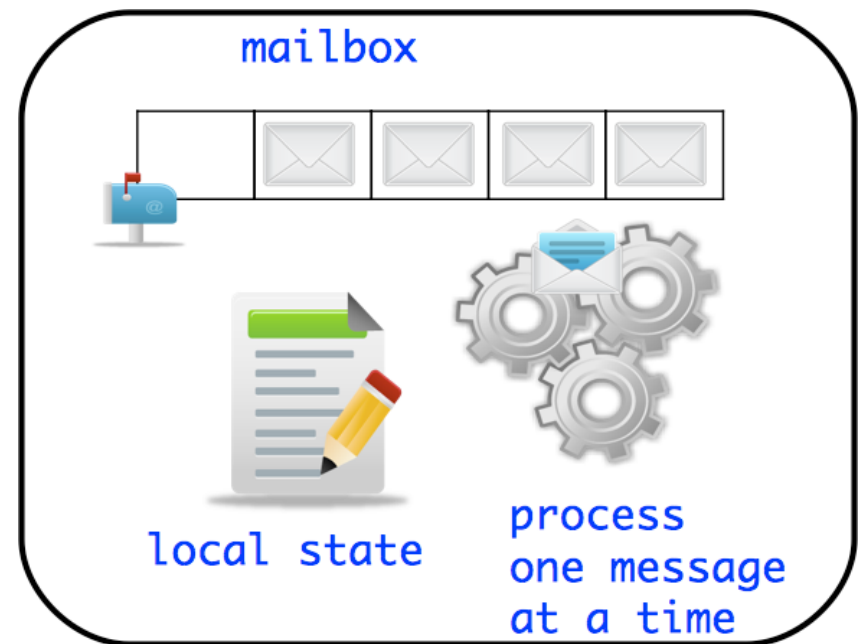
# Actor Life Cycle (Lecture 22)

Actor states

- **New: Actor has been created**

  - **e.g., email account has been created, messages can be received**

- **Started: Actor can process messages**

  - **e.g., email account has been activated**

- **Terminated: Actor will no longer processes messages**

  - **e.g., termination of email account after graduation**

ThreadRing (Coordination) Example

```
1.   finish(() -> {
2.     int threads = 4;
3.     int numberOfHops = 10;
4.     ThreadRingActor[] ring =
          new ThreadRingActor[threads];
5.     for(int i=threads-1;i>=0; i--) {
6.       ring[i] = new ThreadRingActor(i);
7.       ring[i].start();
8.       if (i < threads - 1) {
9.         ring[i].nextActor(ring[i + 1]);
10.    } }
11.    ring[threads-1].nextActor(ring[0]);
12.    ring[0].send(numberOfHops);
13. }); // finish
```
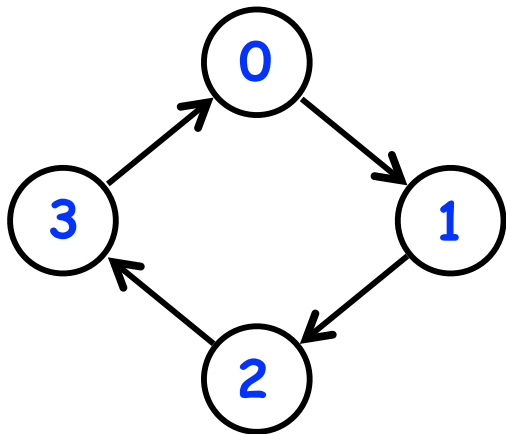
```
14. class ThreadRingActor
15.       extends Actor<Integer> {
16.   private Actor<Integer> nextActor;
17.   private final int id;
18.   ...
19.   public void nextActor(
         Actor<Object> nextActor) {...}

21.   protected void process(Integer n) {
22.     if (n > 0) {
23.       println("Thread-" + id +
24.         " active, remaining = " + n);
25.       nextActor.send(n - 1);
26.     } else {
27.       println("Exiting Thread-"+ id);
28.       nextActor.send(-1);
29.       exit();
30. } } }
```

Worksheet #22 solution: Interaction between `finish` and actors

What output will be printed if the end-finish operation from slide 13 is moved from line 13 to line 11 as shown below?

```
1.  finish(() -> {
2.      int numThreads = 4;
3.      int numberOfHops = 10;
4.      ThreadRingActor[] ring = new ThreadRingActor[numThreads];
5.      for(int i=numThreads-1;i>=0; i--) {
6.        ring[i] = new ThreadRingActor(i);
7.        ring[i].start(); // like an async
8.        if (i < numThreads - 1) {
9.          ring[i].nextActor(ring[i + 1]);
10.       } }
11. }); // finish
12.ring[numThreads-1].nextActor(ring[0]);
13.ring[0].send(numberOfHops);
```

Deadlock (no output): the end-finish operation in line 11 waits for all the actors started in line 7 to terminate, but the actors are waiting for the message sequence initiated in line 13 before they call exit().

Synchronous Reply using Pause/Resume (Lecture 23)

- Actors are asynchronous, sync. replies require blocking operations

- We need notifications from recipient actor on when to resume

- Resumption needs to be triggered on sender actor

- Use DDFs and `asyncAwait`

```
1.  class SynchronousSenderActor
2.      extends Actor<Message> {
3.   void process(Msg msg) {
4.     ...
5.     DDF<T> ddf = newDDF();
6.     otherActor.send(ddf);
7.     pause(); // non-blocking
8.     asyncAwait(ddf, () -> {
9.        T synchronousReply = ddf.get();
10.       println("Response received");
11.       resume(); // non-blocking
12.     });
13.     ...
14. } }
```

```
1.  class SynchronousReplyActor
2.      extends Actor<DDF> {
3.    void process(DDF msg) {
4.      ...
5.      println("Message received");
6.      // process message
7.      T responseResult = ...;
8.      msg.put(responseResult);
9.      ...
10. } }
```

Worksheet #23:
Analyzing Parallelism in an Actor Pipeline

Consider a three-stage pipeline of actors (as in slide 5), set up so that P0.nextStage = P1, P1.nextStage = P2, and P2.nextStage = null. The process() method for each actor is shown below. Assume that 100 non-null messages are sent to actor P0 after all three actors are started, followed by a null message. What will the total WORK and CPL be for this execution? Recall that each actor has a sequential thread.

Solution: WORK = 300, CPL = 102

...   Input sequence

$d_9d_8d_7d_6d_5d_4d_3d_2d_1d_0$ → $\boxed{P_0}$ ▸ $\boxed{P_1}$ ▸ $\boxed{P_2}$
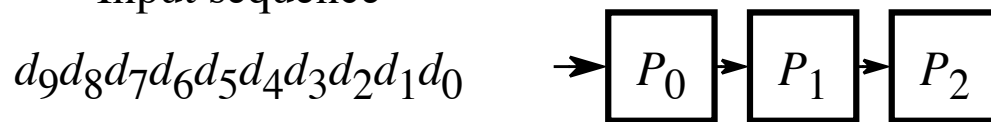
```
1.     protected void process(final Object msg) {
2.         if (msg == null) {
3.             exit(); //actor will exit after returning from process()
4.         } else {
5.             doWork(1); // unit work
6.         }
7.         if (nextStage != null) {
8.             nextStage.send(msg);
9.         }
10.    } // process()
```

Synchronized statements and methods in Java (Lecture 24)

- **Every Java object has an *implicit lock* acquired via:**

  - **`synchronized` statements**

    - synchronized(foo) { // acquire foo's lock
      // execute code while holding foo's lock
      } // release foo's lock

  - **`synchronized` methods**

    - public synchronized void op1() { // acquire 'this' lock
      // execute method while holding 'this' lock
      } // release 'this' lock

- **Java language does not enforce any relationship between object used for locking and objects accessed in isolated code**

  - **If same object is used for locking and data access, then the object behaves like a monitor**

- **Locking and unlocking are automatic**

  - **Locks are released when a synchronized block exits**

    - **By normal means: end of block reached, return, break**

    - **When an exception is thrown and not caught**

Dynamic Order Deadlocks

- **There are even more subtle ways for threads to deadlock due to inconsistent lock ordering**
  - **Consider a method to transfer a balance from one account to another:**

```
public class SubtleDeadlock {
    public void transferFunds(Account from,
                              Account to,
                              int amount) {
        synchronized (from) {
            synchronized (to) {
                from.subtractFromBalance(amount);
                to.addToBalance(amount);
            }
        }
    }
}
```

  - **What if one thread tries to transfer from A to B while another tries to transfer from B to A ?**
    - Inconsistent lock order again – Deadlock!

Avoiding Dynamic Order Deadlocks

- The solution is to **induce** a lock ordering

— **Here, uses an existing unique numeric key, acctId, to establish an order**

```
public class SafeTransfer {

    public void transferFunds(Account from, Account to, int amount) {
        Account firstLock, secondLock;
        if (fromAccount.acctId == toAccount.acctId)
            throw new Exception("Cannot self-transfer");
        else if (fromAccount.acctId < toAccount.acctId) {
            firstLock = fromAccount;
            secondLock = toAccount;
        }
        else {
            firstLock = toAccount;
            secondLock = fromAccount;
        }
        synchronized (firstLock) {

          synchronized (secondLock) {

            from.subtractFromBalance(amount);

            to.addToBalance(amount);

          }

        }

    }
}
```

Deadlock avoidance in HJ with object-based isolation

- HJ implementation ensures that any locks are acquired in the same order (HJ's use of locks to implement isolated is hidden from the user)

- ==> no deadlock possible with isolated construct

```
public class NoDeadlock1 {
   . . .
   public void leftHand() {
       isolated(lock1, lock2) {
           for (int i=0; i<10000; i++)
               sum += random.nextInt(100);


       }
   }
   public void rightHand() {
       isolated(lock2,lock1) {
           for (int i=0; i<10000; i++)
               sum += random.nextInt(100);
       }
   }
   }
}
```

One possible solution to Worksheet #24

1) Write a sketch of the pseudocode for a Java threads program that exhibits a data race using start() and join() operations.

```
1.  // Start of thread t0 (main program)
2.  sum1 = 0; sum2 = 0; // Assume that sum1 & sum2 are fields
3.  // Compute sum1 (lower half) and sum2 (upper half) in parallel
4.  final int len = X.length;
5.  Thread t1 = new Thread(() -> {
6.              for(int i=0 ; i < len/2 ; i++) sum1+=X[i];});
7.  t1.start();
8.  Thread t2 = new Thread(() -> {
9.              for(int i=len/2 ; i < len ; i++) sum2+=X[i];});
10. t2.start();
11. int sum = sum1 + sum2; //data race between t0 & t1, and t0 & t2
12. t1.join(); t2.join();
```

2) Write a sketch of the pseudocode for a Java threads program that exhibits a data race using synchronized statements.

```
1.  // Start of thread t0 (main program)
2.  sum = 0; // static int field
3.  Object a = new ... ;
4.  Object b = new ... ;
5.  Thread t1 = new Thread(() ->
6.                          { synchronized(a) { sum++; } });
7.  Thread t2 = new Thread(() ->
8.                          { synchronized(b) { sum++; } });
9.  t1.start();
10. t2.start(); // data race between t1 & t2
11. t1.join(); t2.join();
```

java.util.concurrent.locks.Lock interface (Lecture 26)

```
1.   interface Lock {
2.      // key methods
3.      void lock(); // acquire lock
4.      void unlock(); // release lock
5.      boolean tryLock();
6.      // Either acquire lock and return true, or return false if lock is
7.      /// not obtained.  A call to tryLock() never blocks!
8.      Condition newCondition();  // associate a new condition
9.                                 // variable with the lock
     }
```

- **java.util.concurrent.locks.Lock interface is implemented by java.util.concurrent.locks.ReentrantLock class**

Worksheet #26a solution: use of tryLock()

Rewrite the transferFunds() method below to use j.u.c. locks with calls to tryLock (see slide 8) instead of synchronized. Your goal is to write a correct implementation that never deadlocks, unlike the buggy version below (which can deadlock). Assume that each Account object already contains a reference to a ReentrantLock object dedicated to that object e.g., from.lock() returns the lock for the from object. Sketch your answer below using pseudocode.

```
1.  public void transferFunds(Account from, Account to, int amount) {
2.    while (true) {
3.      // assume that trylock() does not throw an exception
4.      boolean fromFlag = from.lock.trylock();
5.      if (!fromFlag) continue;
6.      boolean toFlag = to.lock.trylock();
7.      if (!toFlag) { from.lock.unlock(); continue; }
8.      try { from.subtractFromBalance(amount);
9.            to.addToBalance(amount); break; }
10.     finally { from.lock.unlock(); to.lock.unlock(); }
11.   } // while
12.  }
```

java.util.concurrent.locks.ReadWriteLock interface

```
interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
```

- **Even though the interface appears to just define a pair of locks, the semantics of the pair of locks is coupled as follows**
  - **Case 1: a thread has successfully acquired writeLock().lock()**
    - **No other thread can acquire readLock() or writeLock()**
  - **Case 2: no thread has acquired writeLock().lock()**
    - **Multiple threads can acquire readLock()**
    - **No other thread can acquire writeLock()**
- **java.util.concurrent.locks.ReadWriteLock interface is implemented by java.util.concurrent.locks.ReadWriteReentrantLock class**

Example code

```
class Hashtable<K,V> {
  …
  // coarse-grained, one lock for table
  ReadWriteLock lk = new ReentrantReadWriteLock();
  V lookup(K key) {
    int bucket = hasher(key);
    lk.readLock().lock(); // only blocks writers
    … read array[bucket] …
    lk.readLock().unlock();
  }
  void insert(K key, V val) {
    int bucket = hasher(key);
    lk.writeLock().lock(); // blocks readers and writers
    … write array[bucket] …
    lk.writeLock().unlock();
  }
}
```

Linearizability of Concurrent Objects
(Lecture 26)
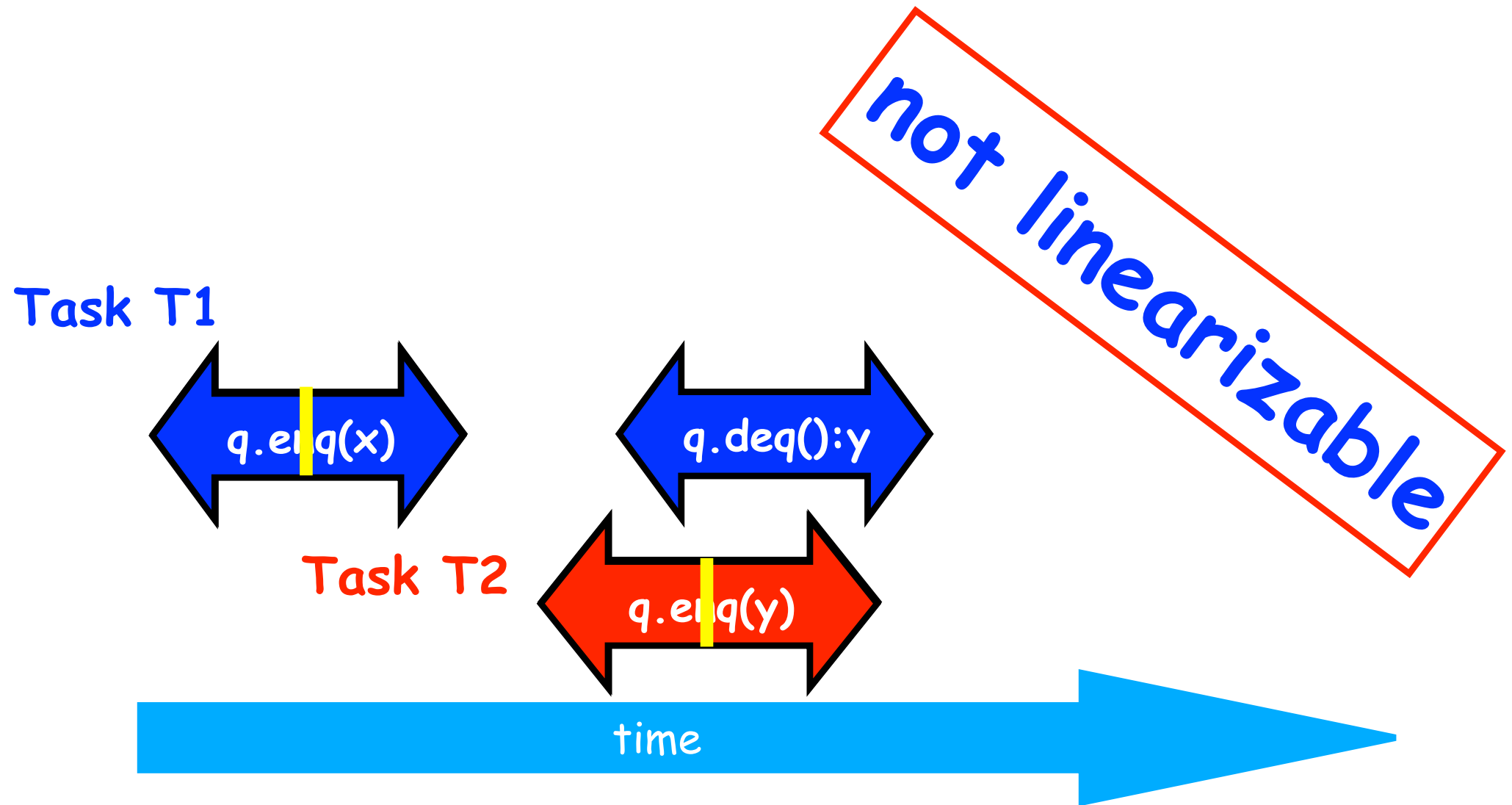
Concurrent object

- A concurrent object is an object that can correctly handle methods invoked in parallel by different tasks or threads

  — **Examples: Concurrent Queue, AtomicInteger**

Linearizability

- Assume that each method call takes effect "instantaneously" at some distinct point in time between its invocation and return.

- An <u>execution</u> is linearizable if we can choose instantaneous points that are consistent with a sequential execution in which methods are executed at those points

- An <u>object</u> is linearizable if all its possible executions are linearizable

Example 2: is this execution linearizable?

not linearizable

Task T1

q.enq(x)

q.deq():y

Task T2

q.enq(y)

time

Worksheet #26b solution:
Linearizability of method calls on a concurrent object

Is this a linearizable execution for a FIFO queue, q?

Time	Task A	Task B
0	Invoke q.enq(x)	
1	Return from q.enq(x)	
2		Invoke q.enq(y)
3	Invoke q.deq()	Work on q.enq(y)
4	Work on q.deq()	Return from q.enq(y)
5	Return y from q.deq()	

No! q.enq(x) must precede q.enq(y) in all linear sequences of method calls invoked on q. It is illegal for the q.deq() operation to return y.

Figure (a)

- Host node (P_c) connected to a cluster of processor nodes ($P_0 \ldots P_m$)

- Processors $P_0 \ldots P_m$ communicate via an interconnection network which could be standard TCP/IP (e.g., for Map-Reduce) or specialized for high performance communication (e.g., for scientific computing)

Figure (b)

- Each processor node consists of a processor, memory, and a Network Interface Card (NIC) connected to a router node (R) in the interconnect

Processors communicate by sending messages via an interconnect
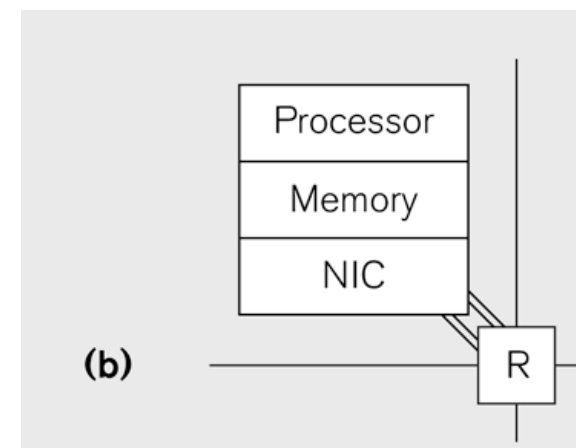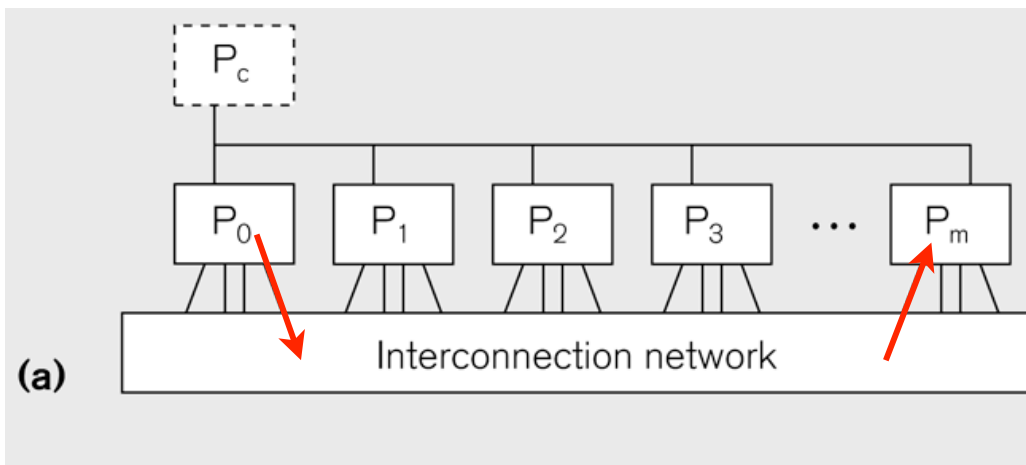
Our First MPI Program (mpiJava version)

> main() is enclosed in an implicit "forall" --- each process runs a separate instance of main() with "index variable" = myrank

```
1.  import mpi.*;

2.  class Hello {
3.      static public void main(String[] args) {
4.          // Init() be called before other MPI calls
5.          MPI.Init(args);
6.          int npes = MPI.COMM_WORLD.Size()
7.          int myrank = MPI.COMM_WORLD.Rank() ;
8.          System.out.println("My process number is " + myrank);
9.          MPI.Finalize(); // Shutdown and clean-up
10.     }
11. }
```

Example of Send and Recv

```
1.   import mpi.*;
2.   class myProg {
3.     public static void main( String[] args ) {
4.       int tag0 = 0; int tag1 = 1;
5.       MPI.Init( args );                        // Start MPI computation
6.       if ( MPI.COMM_WORLD.rank() == 0 ) { // rank 0 = sender
7.         int loop[] = new int[1]; loop[0] = 3;
8.         MPI.COMM_WORLD.Send( "Hello World!", 0, 12, MPI.CHAR, 1, tag0 );
9.         MPI.COMM_WORLD.Send( loop, 0, 1, MPI.INT, 1, tag1 );
10.      } else {                                 // rank 1 = receiver
11.        int loop[] = new int[1]; char msg[] = new char[12];
12.        MPI.COMM_WORLD.Recv( msg, 0, 12, MPI.CHAR, 0, tag0 );
13.        MPI.COMM_WORLD.Recv( loop, 0, 1, MPI.INT, 0, tag1 );
14.        for ( int i = 0; i < loop[0]; i++ )
15.          System.out.println( msg );
16.      }
17.      MPI.Finalize( );                         // Finish MPI computation
18.    }
19. }
```

Send() and Recv() calls are blocking operations

Worksheet #28 solution: MPI send and receive

```
1. int a[], b[];
2. ...
3. if (MPI.COMM_WORLD.rank() == 0) {
4.     MPI.COMM_WORLD.Send(a, 0, 10, MPI.INT, 1, 1);
5.     MPI.COMM_WORLD.Send(b, 0, 10, MPI.INT, 1, 2);
6. }
7. else {
8.     Status s2 = MPI.COMM_WORLD.Recv(b, 0, 10, MPI.INT, 0, 2);
9.     Status s1 = MPI.COMM_WORLD.Recv(a, 0, 10, MPI_INT, 0, 1);
10.    System.out.println("a = " + a + " ; b = " + b);
11.}
12. ...
```

Question: In the space below, indicate what values you expect the print statement in line 10 to output (assuming the program is invoked with 2 processes).

Answer: Nothing! The program will deadlock due to mismatched tags, with process 0 blocked at line 4, and process 1 blocked at line 8.

Collective Communications (Lecture 29)

- A popular feature of MPI is its family of collective communication operations.

- Each collective operation is defined over a communicator (most often, MPI.COMM_WORLD)

— Each collective operation contains an *implicit barrier*. The operation completes and execution continues when all processes in the communicator perform the *same* collective operation.

— A mismatch in operations results in *deadlock* e.g.,

Process 0: .... MPI.Bcast(...) ....

Process 1: .... MPI.Bcast(...) ....

Process 2: .... MPI.Gather(...) ....

- A simple example is the broadcast operation: all processes invoke the operation, all agreeing on one root process. Data is broadcast from that root.

   void Bcast(Object buf, int offset, int count, Datatype type, int root)

MPI Reduce

```
void MPI.COMM_WORLD.Reduce(
                Object          sendbuf         /* in */,
                int             sendoffset      /* in */,
                Object          recvbuf         /* out */,
                int             recvoffset      /* in */,
                int             count           /* in */,
                MPI.Datatype    datatype        /* in */,
                MPI.Op          operator        /* in */,
                int             root            /* in */ )
```
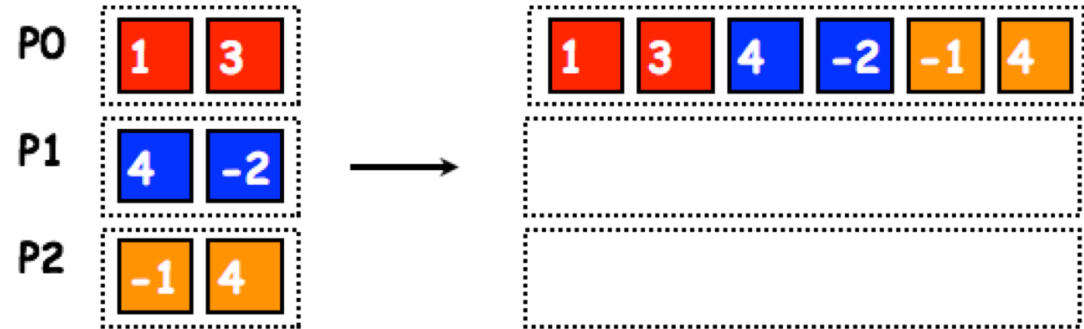


```
MPI.COMM_WORLD.Reduce(msg, 0, result, 0, 1, MPI.INT, MPI.SUM, 2);
```

Worksheet #29 solution: MPI Gather

Indicate what value should be provided instead of ??? in line 6 to minimize space, and how it should depend on `myrank`.


```
1.     MPI.Init(args) ;
2.     int myrank = MPI.COMM_WORLD.Rank() ;
3.     int numProcs = MPI.COMM_WORLD.Size() ;
4.     int size = ...;
5.     int[] sendbuf = new int[size];
6.     int[] recvbuf = new int[???];
7.     . . . // Each process initializes sendbuf
8.     MPI.COMM_WORLD.Gather(sendbuf, 0, size, MPI.INT,
9.                           recvbuf, 0, size, MPI.INT,
10.                          0 /*root*/);
11.    . . .
12.    MPI.Finalize();
```

Solution: `myrank == 0 ? (size * numProcs) : 0`

Worksheet #32 solution:
UPC data distributions

In the following example (which is similar to slide 17, but without the blocking), assume that each UPC array is distributed by default across threads with a cyclic distribution. In the space below, a) identify an iteration of the upc_forall construct for which all array accesses are local, and b) an iteration for which all array accesses are non-local (remote). You can assume any values for THREADS in the 2…99 range that you choose for parts a) and b). Explain your answer in each case.

Note that each shared array's distribution always starts with the first element assigned to thread 0 (not where the previous array may have ended).

```
1.   shared int a[100],b[100], c[100];
2.   int i;
3.   upc_forall (i=0; i<100; i++; (i*THREADS)/100)
4.       a[i] = b[i] * c[i];
```

index   0   1   2   3   4   5   6   .   .   .

index owner in 2-thread case
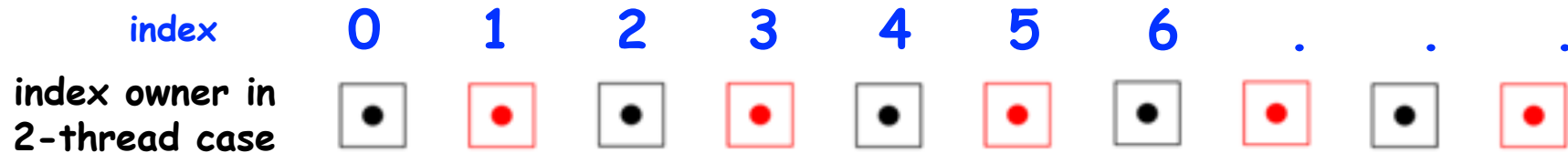
Solution:
- **Iteration 0 has affinity with thread 0, and accesses a[0], b[0], c[0], all of which are located locally at thread 0**
- **Iteration 1 has affinity with thread 0, and accesses a[1], b[1], c[1], all of which are located remotely at thread 1**

Worksheet #33: Combining Task and MPI parallelism

Compute the critical path length for the MPI program shown on the right in pseudocode, assuming that it is executed with 2 processes/ranks. (Assume that the send/recv calls in lines 5 & 10 match with each other.)

Solution: CPL = 2, because lines 6 and 9 can execute in parallel with each other

```
1. main() {
2.    if (my rank == 0)
3.       finish { // F1
4.          async await(req) doWork(1);
5.          MPI_Irecv(rank 1, … , req);
6.          doWork(1);
7.       }
8.    else {
9.       doWork(1);
10.      MPI_Send(rank 0, …);
11.   }
12. } // main
```
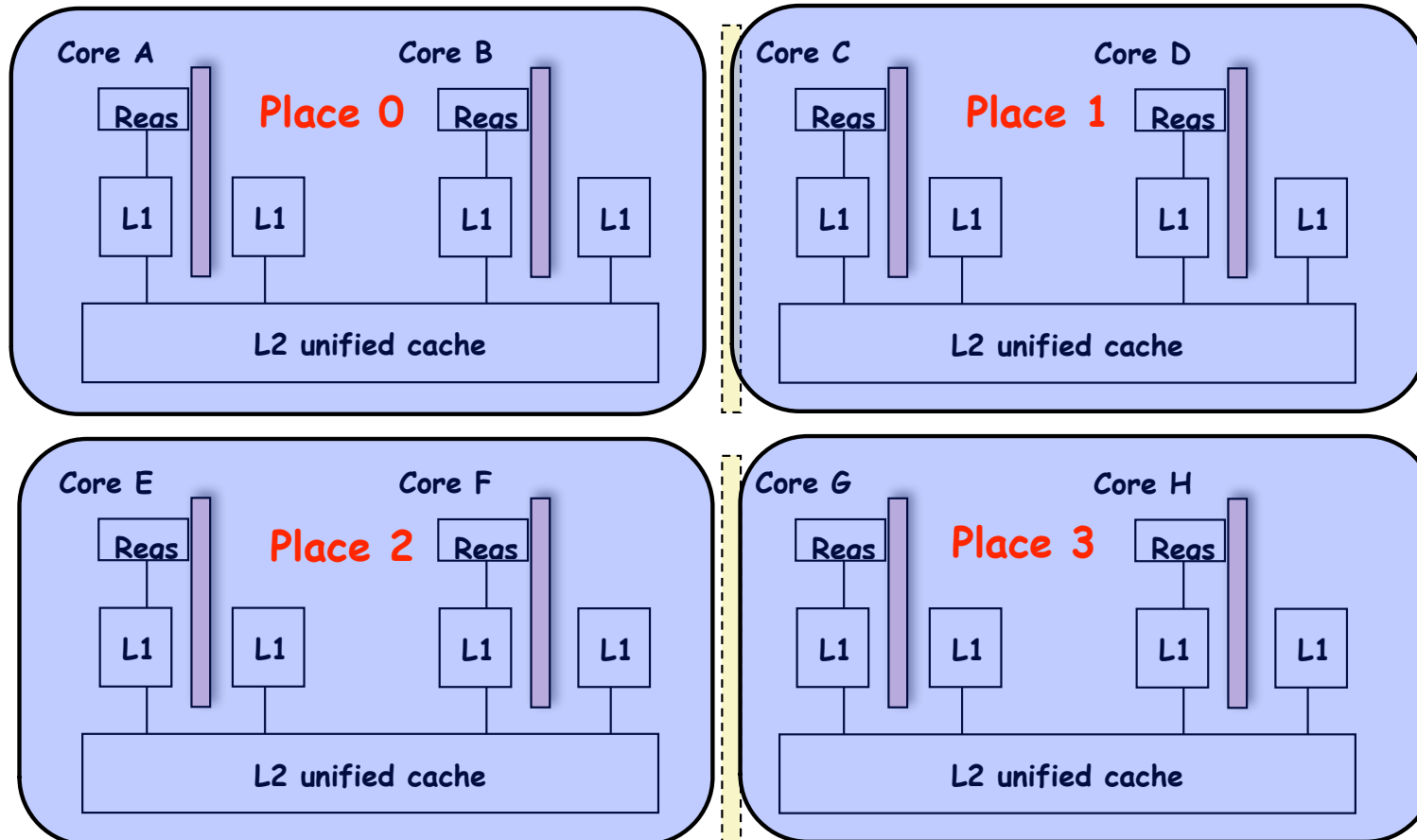
Co-locating async tasks in "places" (Lecture 34)

```
// Main program starts at place 0
asyncAt(place(0), () -> S1);
asyncAt(place(0), () -> S2);
```

```
asyncAt(place(1), () -> S3);
asyncAt(place(1), () -> S4);
asyncAt(place(1), () -> S5);
```



```
asyncAt(place(2), () -> S6);
asyncAt(place(2), () -> S7);
asyncAt(place(2), () -> S8);
```

```
asyncAt(place(3), () -> S9);
asyncAt(place(3), () -> S10);
```

Worksheet #34 solution: impact of distribution on parallel completion time (rather than locality)

```
1.   public void sampleKernel(
2.       int iterations, int numChunks, Distribution dist) {
3.     for (int iter = 0; iter < iterations; iter++) {
4.       finish(() -> {
5.         forseq (0, numChunks - 1, (jj) -> {
6.           asyncAt(dist.get(jj), () -> {
7.             doWork(jj);
8.             // Assume that time to process chunk jj = jj units
9.           });
10.        });
11.      });
12.    } // for iter
13. } // sample kernel
```

- Assume an execution with n places, each place with one worker thread
- Will a block or cyclic distribution for `dist` have a smaller abstract completion time, assuming that all tasks on the same place are serialized with one worker per place?

Answer: Cyclic distribution because it leads to better load balance (locality was not a consideration in this problem)

Worksheet #35 Solution: Finding maximal index of goal in matrix

Below is a code fragment intended to find the max ~~occurs multiple times in the input matrix. What log~~

	0	…	10	…	15	…
…						
5			M			
…						
10	M				M	

```
1.   class AsyncFinishEurekaSearchMaxIndexOf
2.     HjEureka eurekaFactory() {
3.       comparator = (cur, newVal) -> { //
             (cur.x==newVal.x) ? (cur.y - newV
4.       return new MaximaEureka([-1, -1], c
5.     }
6.     int[] doWork(matrix, goal) {
7.       val eu = eurekaFactory()
8.       finish (eu, () -> { // eureka registration
9.         forasync (0, matrix.length - 1, (r) ->
10.          procRow(matrix(r), r, goal));
11.      });
12.      return eu.get()
13.    }
14.    void procRow(array, r, goal) {
15.      for (int c = array.length() - 1; c >= 0; c--)
16.        check([r, c]) // terminate if comparator returns negative
17.        if goal.match(array(c)) offer([r, c]) // updates cur in eureka
18.    } }
```

The task terminates when check([r,c]) is called and the comparator has cur smaller than [r,c]. We need to ensure the iteration order in our code is such that the comparator returning negative means we cannot produce an offer([r',c']) where [r', c'] is greater than the value of cur.

Worksheet #36 problem statement: Parallelizing the Split step in Radix Sort

The Radix Sort algorithm loops over the bits in the binary representation of the keys, starting at the lowest bit, and executes a split operation for each bit as shown below. The split operation packs the keys with a 0 in the corresponding bit to the bottom of a vector, and packs the keys with a 1 to the top of the same vector. It maintains the order within both groups. The sort works because each split operation sorts the keys with respect to the current bit and maintains the sorted order of all the lower bits. Your task is to show how the split operation can be performed in parallel using scan operations, and to explain your answer.

```
                                     [101 111 011 001 100 010 111 010]
1.A =                                [5 7 3 1 4 2 7 2]
2.A⟨0⟩ =                             [1 1 1 1 0 0 1 0] //lowest bit
3.A←split(A,A⟨0⟩) = [4 2 2 5 7 3 1 7]
4.A⟨1⟩ =                             [0 1 1 0 1 1 0 1] // middle bit
5.A←split(A,A⟨1⟩) = [4 5 1 2 2 7 3 7]
6.A⟨2⟩ =                             [1 1 0 0 0 1 0 1] // highest bit
7.A←split(A,A⟨2⟩) = [1 2 2 3 4 5 7 7]
```

```
procedure split(A, Flags)
    I-down ← prescan(+, not(Flags)) // prescan = exclusive prefix sum
    I-up   ← rev(n - scan(+, rev(Flags)) // rev = reverse
    in parallel for each index i
      if (Flags[i])
        Index[i] ← I-up[i]
      else
        Index[i] ← I-down[i]
    result ← permute(A, Index)
```

|									
A	=	[5	7	3	1	4	2	7	2]
Flags	=	[1	1	1	1	0	0	1	0]
I-down	=	[0	0	0	0	0	1	2	2]
I-up	=	[3	4	5	6	7	7	7	8	
Index	=	[3	4	5	6	0	1	7	2]
permute(A, Index)	=	[4	2	2	5	7	3	1	7]

FIGURE 1.9

The split operation packs the elements with a 0 in the corresponding flag position to the bottom of a vector, and packs the elements with a 1 to the top of the same vector. The permute writes each element of A to the index specified by the corresponding position in Index.

Worksheet #37 (one possible solution): Creating a Circuit for Parallel Prefix Sums

Assume that you have a full adder cell, $\oplus$, that can be used as a building block for circuits (no need to worry about carry's). Create a circuit that generates the prefix sums for 1, … 6, by adding at most 5 more cells to the sketch shown below, while ensuring that the CPL is at most 3 cells long. Assume that you can duplicate any value (fan-out) to whatever degree you like without any penalty.
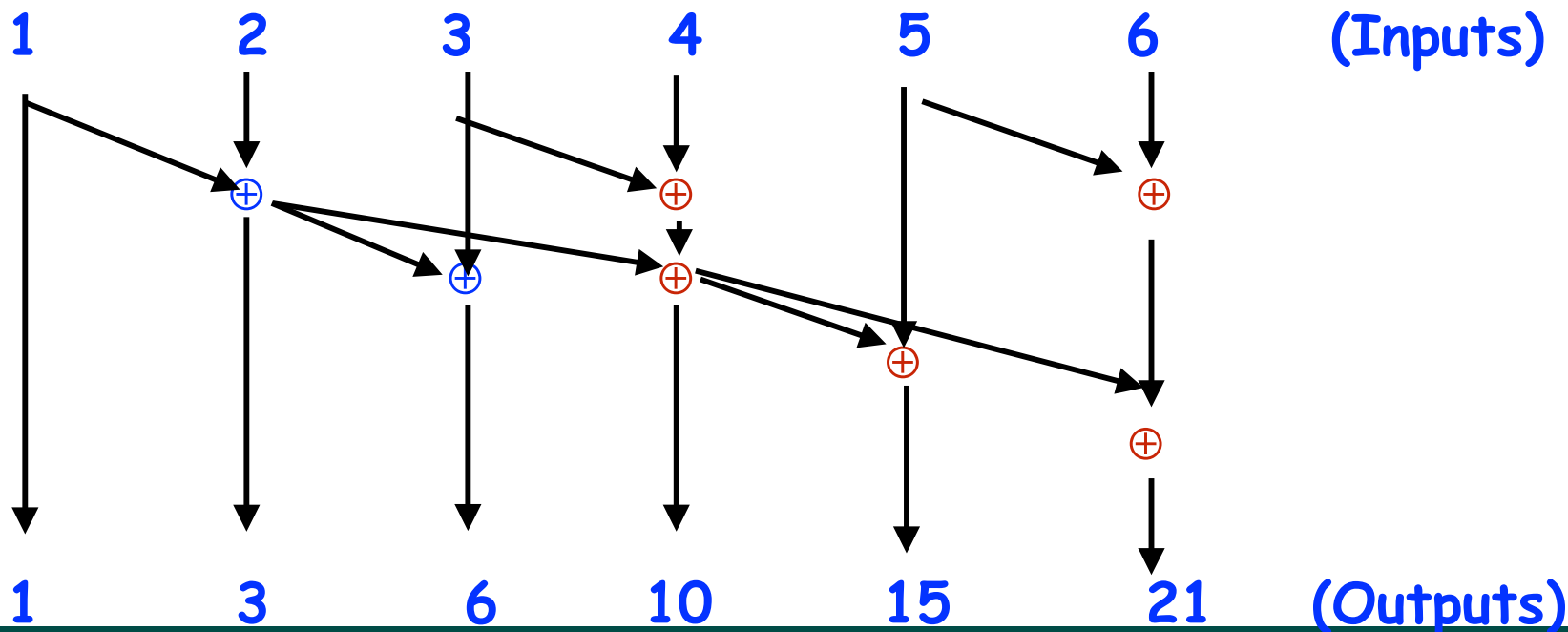
GPU Design Idea #2: lock stepping w/ branching (Lecture 38)
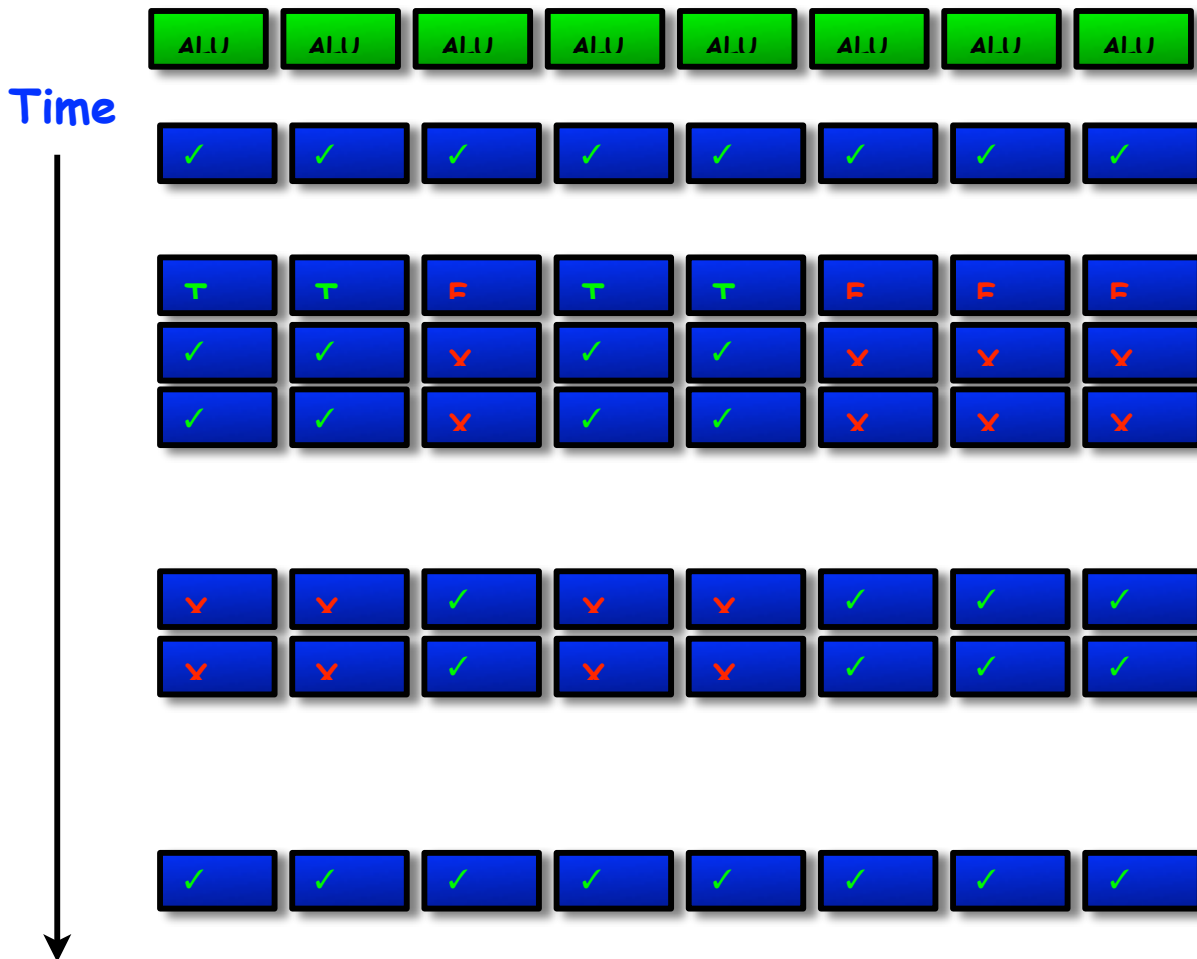
Non branching code;

if(flag > 0){ /* branch */
  x = exp(y);
  y = 2.3*x;
}
else{
  x = sin(y);
  y = 2.1*x;
}

Non branching code;

The cheap branching approach means that some ALUs are idle as all ALUs traverse all branches [executing NOPs if necessary]

In the worst possible case we could see 1/8 of maximum performance.

Worksheet #38 solution:
Branching in SIMD code

Consider SIMD execution of the following pseudocode with 8 threads in a block. Assume that each call to doWork(x) takes x units of time, and ignore all other costs. How long will this program take when executed on 8 GPU cores, taking into consideration the branching issues discussed in Slide 9?

```
1. int tx = threadIdx.x; // ranges from 0 to 7
2. if (tx % 2 = 0) {
3.    S1: doWork(1); // Computation S1 takes 1 unit of time
4. }
5. else {
6.    S2: doWork(2); // Computation S2 takes 2 units of time
7. }
```

Solution: 3 units of time (WORK=24, CPL=3)

Announcements

- Homework 5 due today (officially) with penalty-free extension until May 1st
  - —Any remaining slip days can be applied past May 1st

- Exam 2 is a scheduled final exam to be held during 9am - 12noon on Tuesday, May 2nd, in KCK 100
  - —Final exam will cover material from Lectures 18 - 38

- Group office hours will be held next week in Duncan Hall at the following times

  - 2pm - 3pm, Monday, April 24th (DH 3092)

  - 2pm - 3pm, Wednesday, April 26th (DH 3092)

  - 2pm - 3pm, Friday, April 28th (DH 3076)

- *COMP 322 grand finale at 3pm in DH 3092 TODAY (group office hours time slot, but cake will be served!)*

Acknowledgments

- **Co-instructor**
  - —Mack Joyner

- **Graduate TAs**
  - —Max Grossman (Head TA), Jonathan Sharman, Ryan Spring, Bing Xue, Lechen Yu

- **Undergraduate TAs**
  - —Marc Canby, Anna Chi, Peter Elmers, Joseph Hungate, Cary Jiang, Gloria Kim, Kevin Mullin, Victoria Nazari, Ashok Sankaran, Sujay Tadwalkar, Anant Tibrewal, Eugene Wang, Yufeng Zhou

- **Administrative Staff**
  - —Annepha Hurlock, Bel Martinez

Have a great summer!!

"Education is what survives when what has been learned has been forgotten"
B.F. Skinner