COMP 322: Fundamentals of Parallel Programming

Lecture 38: Review of Lectures 20 - 37 (Scope of Exam 2)

Mack Joyner and Zoran Budimlić
{mjoyner, zoran}@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

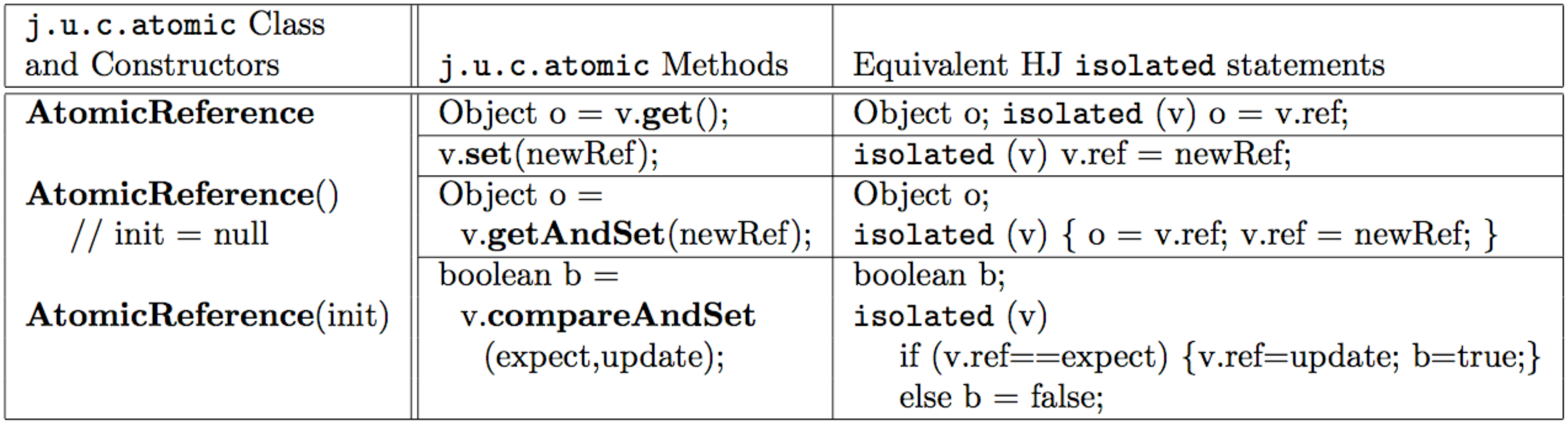
`isolated(obj1, obj2, …, () -> <body>)`

- 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

```
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. 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

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

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
- 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;
   });
 }
```

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.

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

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

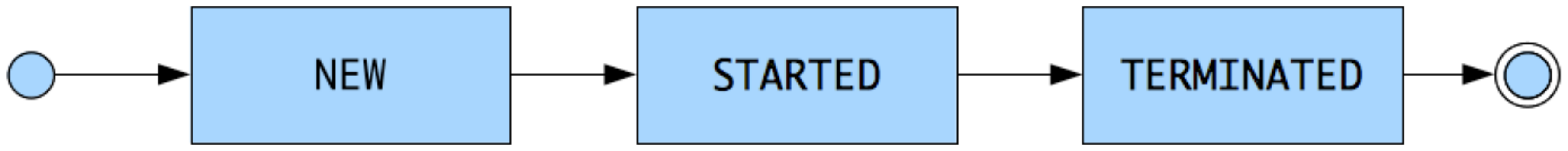
Worksheet #21b solution: Abstract Metrics with Object-Based Isolated Construct

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.

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

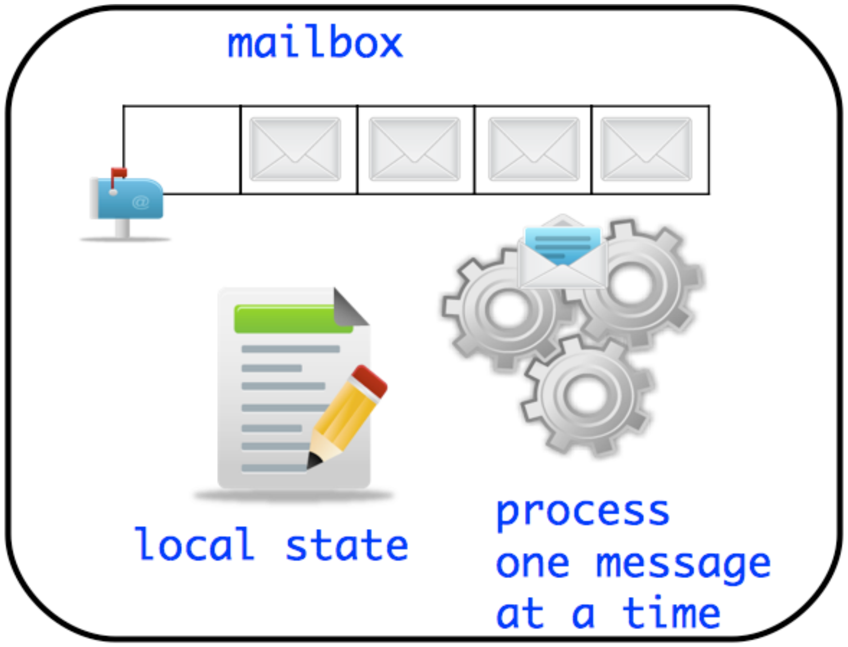
Answer: WORK = 25, CPL depends on execution order. Best-case CPL = 6, worst-case CPL = 7. (Worst-case example: if A_1, A_4 execute in parallel first, then the isolated sections in A_2, A_3 must be serialized thereafter.)

Actor Life Cycle (Lecture 22)



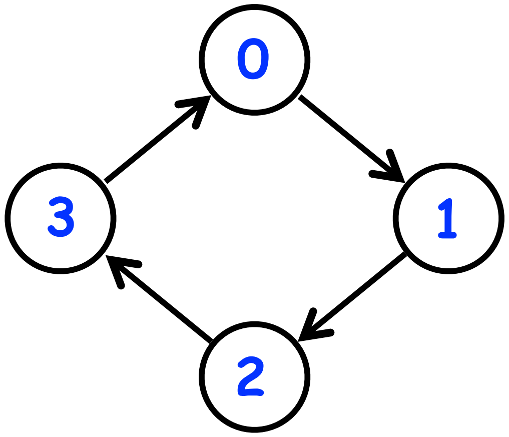
Actor states

- **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

```
finish(() -> {
  int threads = 4;
  int numberOfHops = 10;
  ThreadRingActor[] ring =
    new ThreadRingActor[threads];
  for(int i=threads-1;i>=0; i--) {
    ring[i] = new ThreadRingActor(i);
    ring[i].start();
    if (i < threads - 1) {
      ring[i].nextActor(ring[i + 1]);
  } }
  ring[threads-1].nextActor(ring[0]);
  ring[0].send(numberOfHops);
}); // finish
```



```
class ThreadRingActor
    extends Actor<Integer> {
  private Actor<Integer> nextActor;
  private final int id;
  ...
  public void nextActor(
    Actor<Object> nextActor) {...}

  protected void process(Integer n) {
    if (n > 0) {
      println("Thread-" + id +
        " active, remaining = " + n);
      nextActor.send(n - 1);
    } else {
      println("Exiting Thread-"+ id);
      nextActor.send(-1);
      exit();
} } }
```

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`

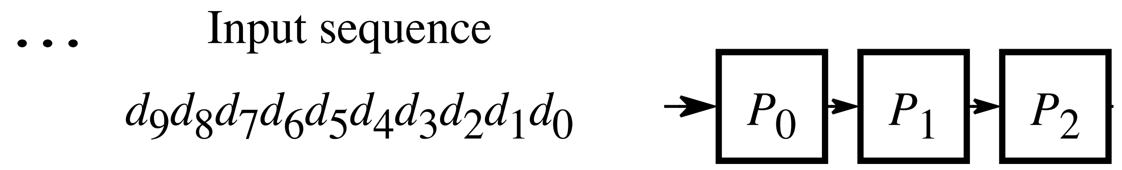
```
class SynchronousSenderActor
    extends Actor<Message> {
  void process(Msg msg) {
    ...
    DDF<T> ddf = newDDF();
    otherActor.send(ddf);
    pause(); // non-blocking
    asyncAwait(ddf, () -> {
      T synchronousReply = ddf.get();
      println("Response received");
      resume(); // non-blocking
    });
    ...
} }
```

```
class SynchronousReplyActor
    extends Actor<DDF> {
  void process(DDF msg) {
    ...
    println("Message received");
    // process message
    T responseResult = ...;
    msg.put(responseResult);
    ...
} }
```

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



```
protected void process(final Object msg) {
   if (msg == null) {
     exit(); //actor will exit after returning from process()
   } else {
     doWork(1); // unit work
   }
   if (nextStage != null) {
     nextStage.send(msg);
   }
} // 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.

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

One possible solution to Worksheet #24 (contd)

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

```
// Start of thread t0 (main program)
sum = 0; // static int field
Object a = new ... ;
Object b = new ... ;
Thread t1 = new Thread(() ->
                    { synchronized(a) { sum++; } });
Thread t2 = new Thread(() ->
                    { synchronized(b) { sum++; } });
t1.start();
t2.start(); // data race between t1 & t2
t1.join(); t2.join();
```

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

```
interface Lock {
  // key methods
  void lock(); // acquire lock
  void unlock(); // release lock
  boolean tryLock();
  // Either acquire lock and return true, or return false if lock is
  /// not obtained.  A call to tryLock() never blocks!
  Condition newCondition();  // associate a new condition
                             // 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.

```
public void transferFunds(Account from, Account to, int amount) {
  while (true) {
    // assume that trylock() does not throw an exception
    boolean fromFlag = from.lock.trylock();
    if (!fromFlag) continue;
    boolean toFlag = to.lock.trylock();
    if (!toFlag) { from.lock.unlock(); continue; }
    try { from.subtractFromBalance(amount);
          to.addToBalance(amount); break; }
     finally { from.lock.unlock(); to.lock.unlock(); }
  } // while
}
```

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)

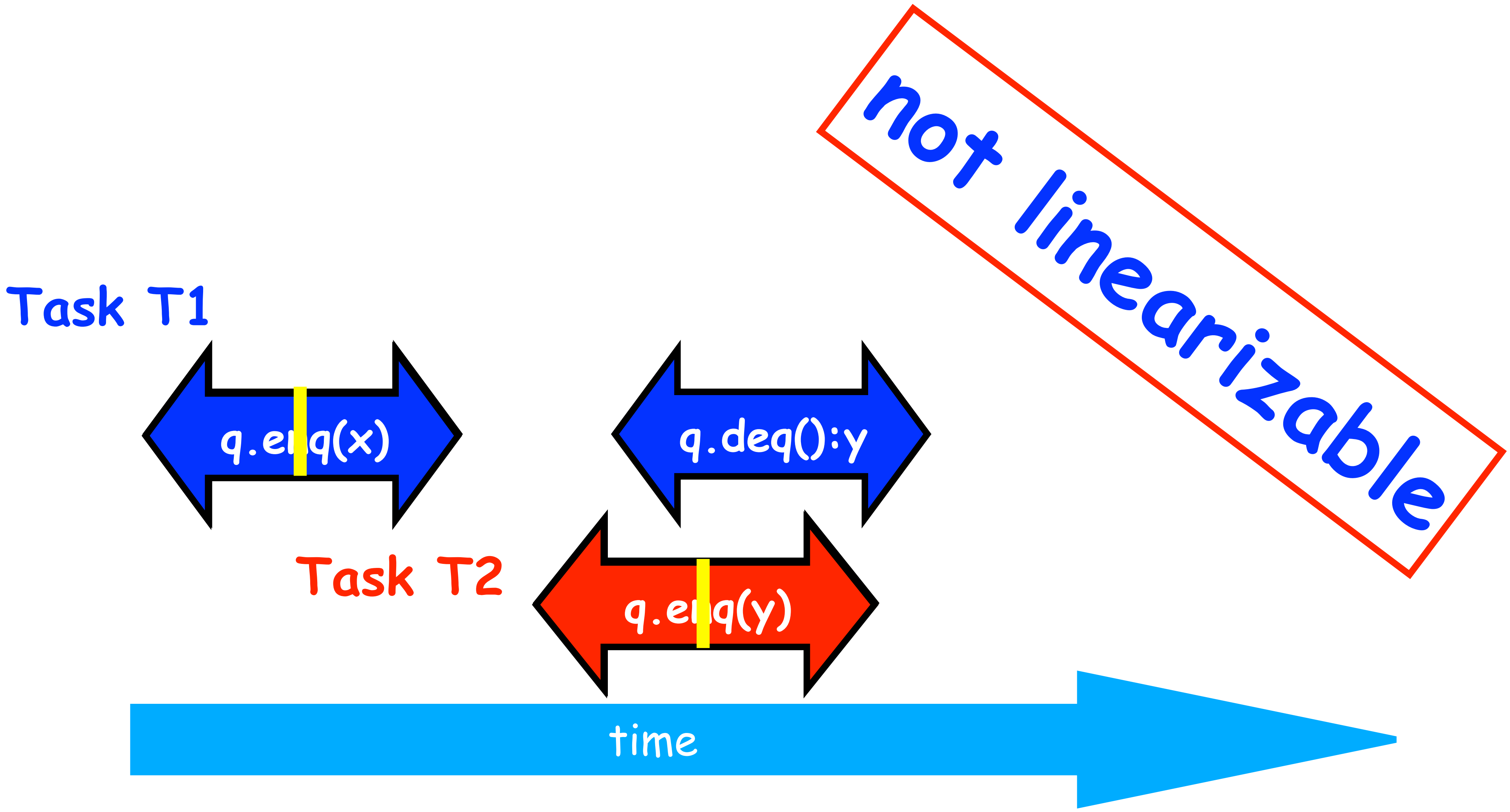
Concurrent object

- 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 execution is linearizable if we can choose instantaneous points that are consistent with a sequential execution in which methods are executed at those points
- An object is linearizable if all its possible executions are linearizable

Example 2: is this execution linearizable?



Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

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.

Organization of a Distributed-Memory Multiprocessor (Lecture 28 — Start of Module 3)

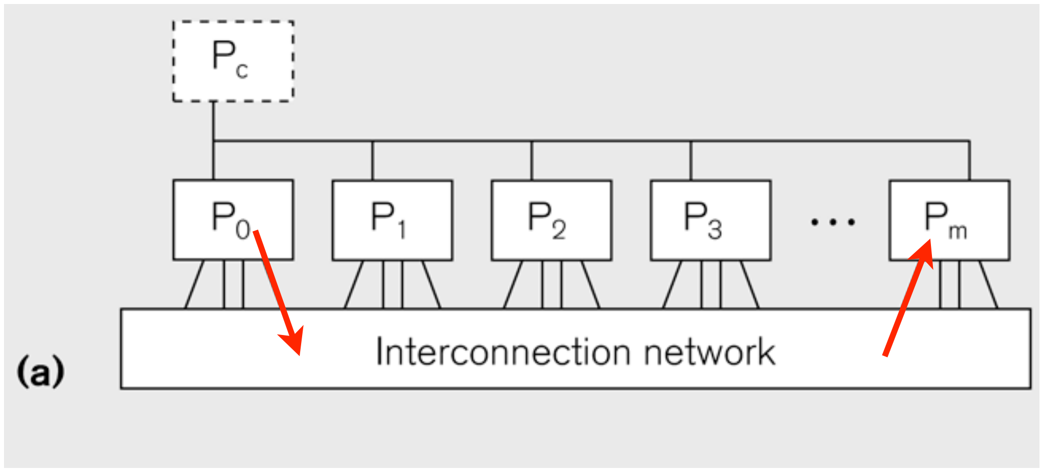
Figure (a)

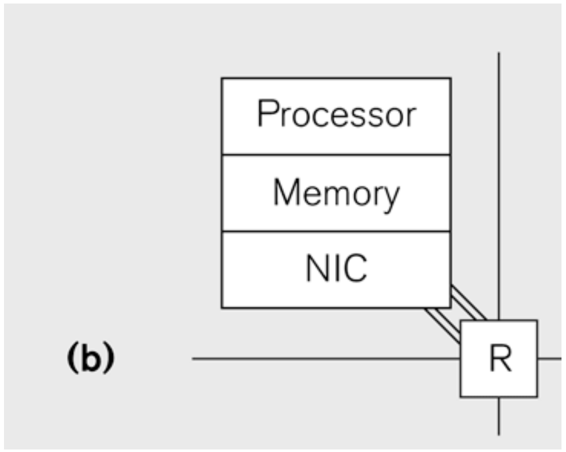
- Host node (P_c) connected to a cluster of processor nodes (P_0 … P_m)
- Processors P_0 … P_m communicate via an interconnection network which could be standard TCP/IP (e.g., for Map-Reduce) or specialized for high performance communication (e.g., for scientific computing)

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

```
import mpi.*;

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

Example of Send and Recv

```
import mpi.*;
class myProg {
  public static void main( String[] args ) {
    int tag0 = 0; int tag1 = 1;
    MPI.Init( args );                   // Start MPI computation
    if ( MPI.COMM_WORLD.rank() == 0 ) { // rank 0 = sender
      int loop[] = new int[1]; loop[0] = 3;
      MPI.COMM_WORLD.Send( "Hello World!", 0, 12, MPI.CHAR, 1, tag0 );
      MPI.COMM_WORLD.Send( loop, 0, 1, MPI.INT, 1, tag1 );
    } else {                           // rank 1 = receiver
      int loop[] = new int[1]; char msg[] = new char[12];
      MPI.COMM_WORLD.Recv( msg, 0, 12, MPI.CHAR, 0, tag0 );
      MPI.COMM_WORLD.Recv( loop, 0, 1, MPI.INT, 0, tag1 );
      for ( int i = 0; i < loop[0]; i++ )
        System.out.println( msg );
    }
    MPI.Finalize( );                   // Finish MPI computation
  }
}
```

`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(...) ....

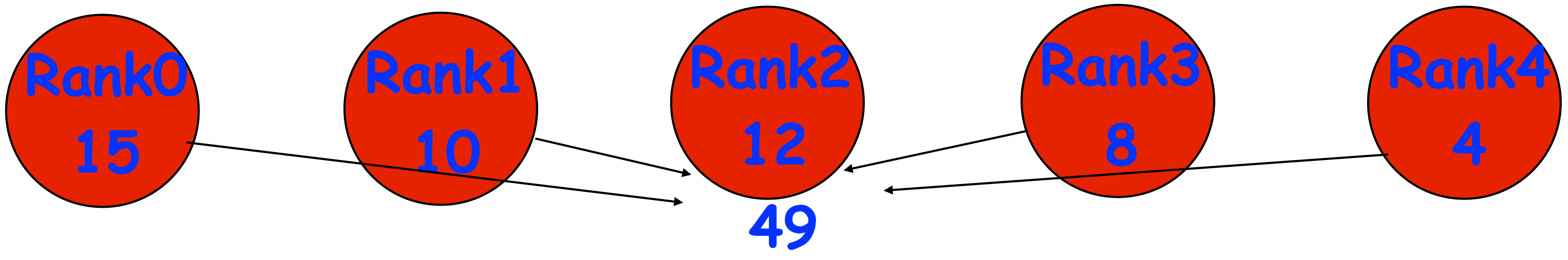
Process 2: .... MPI.Gather(...) ….

- 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

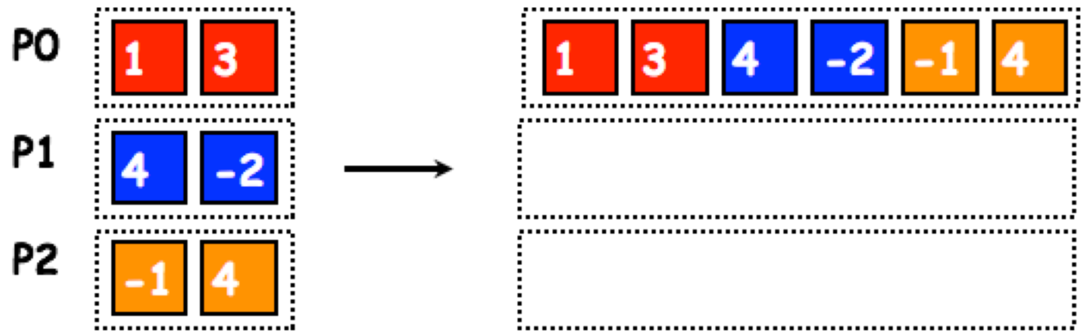
```
void MPI.COMM_WORLD.Reduce(
        Object  sendbuf         /* in */,
        int           sendoffset      /* in */,
        Object  recvbuf         /* out */,
        int           recvoffset      /* in */,
        int           count           /* in */,
        MPI.Datatype  datatype        /*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

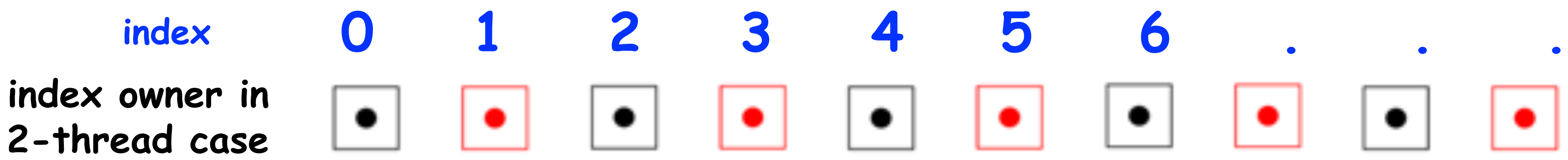
Solution: `myrank == 0 ? (size * numProcs) : 0`

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).

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



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.)

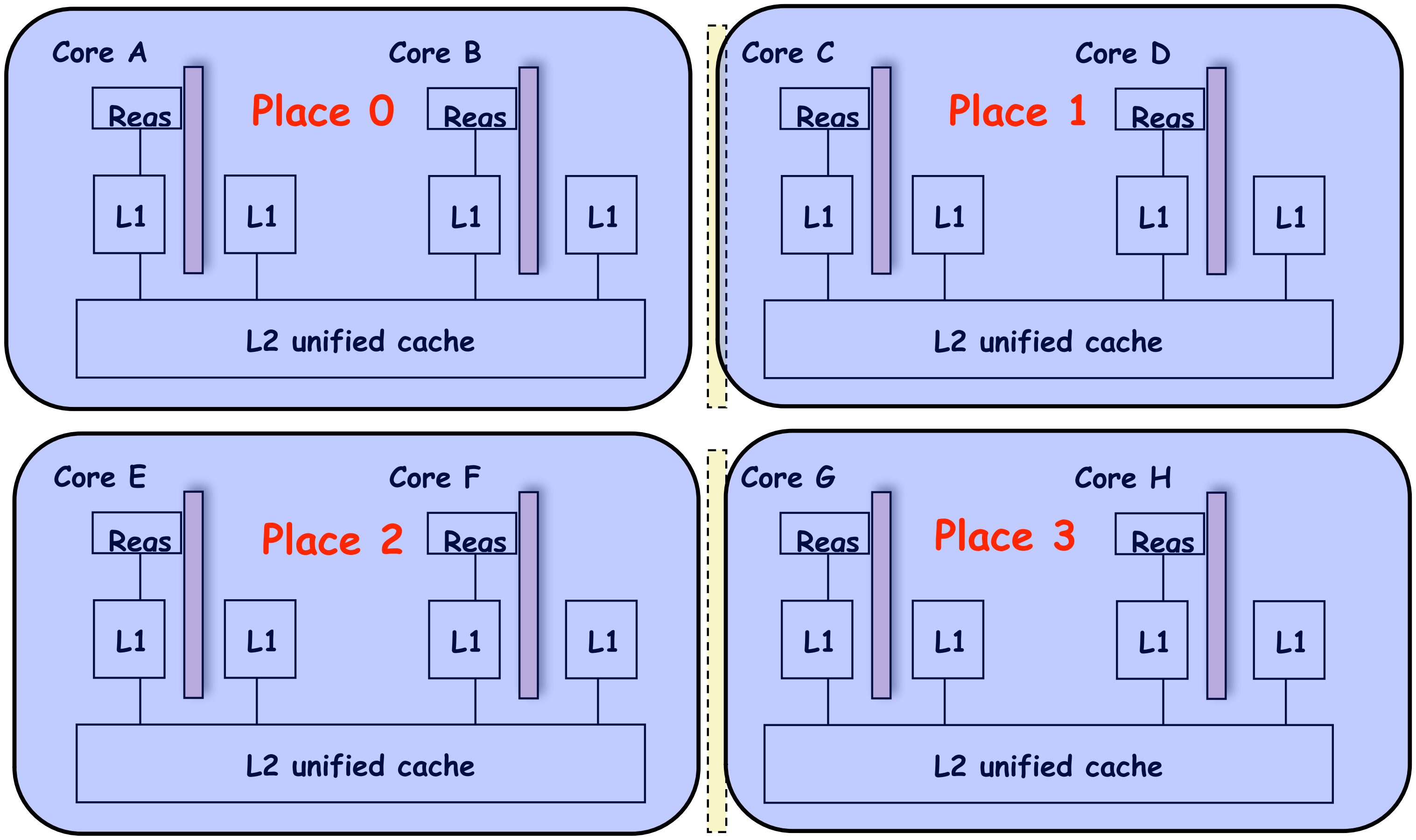
```
1.  main() {
2.    if (my rank == 0)
3.      finish { // F1
4.        async await(req) doWork(1);
5.        MPI_Irecv(rank 1, … , req);
6.        doWork(1);
7.      }
8.    else {
9.      doWork(1);
10.     MPI_Send(rank 0, …);
11.   }
12. } // main
```

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

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)

```
public void sampleKernel(
      int iterations, int numChunks, Distribution dist) {
    for (int iter = 0; iter < iterations; iter++) {
      finish(() -> {
        forseq (0, numChunks - 1, (jj) -> {
          asyncAt(dist.get(jj), () -> {
            doWork(jj);
            // Assume that time to process chunk jj = jj units
         });
        });
      });
    } // for iter
} // 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	

```
class AsyncFinishEurekaSearchMaxIndexOf
  HjEureka eurekaFactory() {
    comparator = (cur, newVal) -> { //
      (cur.x==newVal.x) ? (cur.y - newV
    return new MaximaEureka([-1, -1], c
  }
  int[] doWork(matrix, goal) {
    val eu = eurekaFactory()
    finish (eu, () -> { // eureka registration
      forasync (0, matrix.length - 1, (r) ->
        procRow(matrix(r), r, goal));
    });
    return eu.get()
  }
  void procRow(array, r, goal) {
    for (int c = array.length() - 1; c >= 0; c--)
      check([r, c]) // terminate if comparator returns negative
      if goal.match(array(c)) offer([r, c]) // updates cur in eureka
  } }
```

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]
```

Worksheet #36 solution: Parallelizing the Split step in Radix Sort

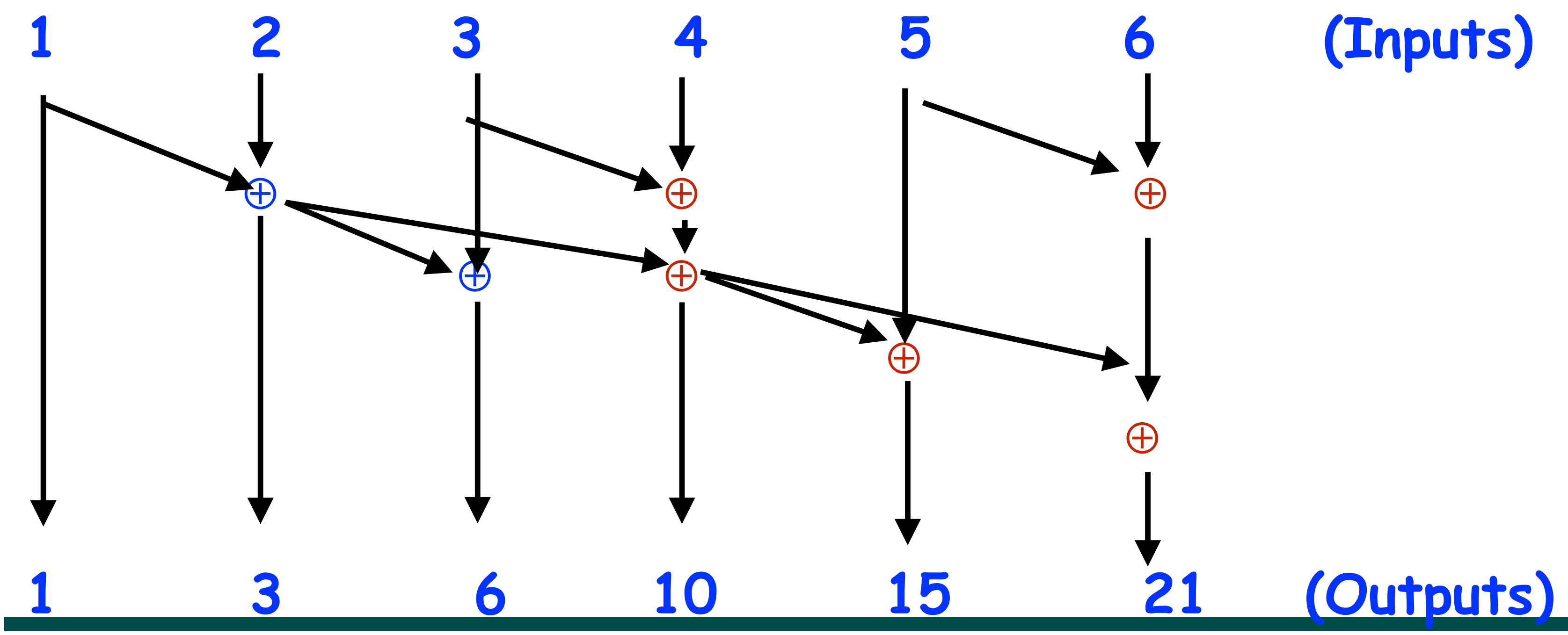
```
procedure split(A, Flags)
  I-down ← prescan(+, not(Flags)) // prescan = exclusive prefix sum
  I-up   ← rev(n - scan(+, rev(Flags)) // rev = reverse
  in parallel for each index $i$
    if (Flags[$i$])
      Index[$i$] ← I-up[$i$]
    else
      Index[$i$] ← I-down[$i$]
  result ← permute(A, Index)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

A	=	[5	7	3	1	4	2	7	2]
Flags	=	[1	1	1	1	0	0	1	0]
I-down	=	[0	0	0	0	$\boxed{0}$	$\boxed{1}$	2	$\boxed{2}$]
I-up	=	[$\boxed{3}$	$\boxed{4}$	$\boxed{5}$	$\boxed{6}$	7	7	$\boxed{7}$	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.



Announcements

- Quiz 10 (Optional) is due today at 11:59pm.
- Homework 5 due today (officially) with penalty-free extension until Apr 21st
  - —Any remaining slip days can be applied past Apr 21st
- Exam 2 is a scheduled final exam to be held during 9am - 12noon on Wednesday, May 1st, in Duncan Hall (McMurtry Auditorium)
  - — Final exam will cover material from Lectures 20 - 37
  - — It will be an online Canvas exam

Acknowledgments

- Course Design
  - —Vivek Sarkar
- Infrastructure
  - —Max Grossman, Shams Imam
- Head TA
  - —Srđan Milaković
- Graduate TAs
  - —Jonathan Sharman
- Undergraduate TAs
  - —Liam Bonnage, Harrison Brown, Mustafa El-Gamal, Krishna Goel, Ryan Green, Ryan Han, Rishu Harpavat, Namanh Kapur, Tian Lan, Tam Le, Eva Ma, Hamza Nauman, Rutvik Patel, Aryan Sefidi, Tory Songyang, Jiaqi Wang, Erik Yamada, Yifan Yang
- Administrative Staff
  - —Annepha Hurlock

Have a great summer!!

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