COMP 322: Fundamentals of Parallel Programming

Lecture 37: Review of Lectures 19-34 (Scope of Exam 2)

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu

Announcements & Reminders

- Quiz for Unit 8 is due today at 11:59pm

- The Final exam (in Canvas) is Wednesday, May 6th from at 9am - 12pm (CST).
  - Final exam is optional
    - If final exam is not taken or score is lower than midterm, midterm will be exam grade (40%)
    - If final exam is higher than midterm, we'll average the scores for exam grade
    - It is an open book (slides, module handout, videos) and open notes exam
  - You may reschedule the exam time if your current time zone is not CST

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
- 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

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 object 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

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

Worksheet #19 Abstract Metrics with Object-based Isolated Constructs

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. 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.

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

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.

Worksheet #20: Atomic Variables represent a special (and more efficient) 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

```
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;
  });
}
```

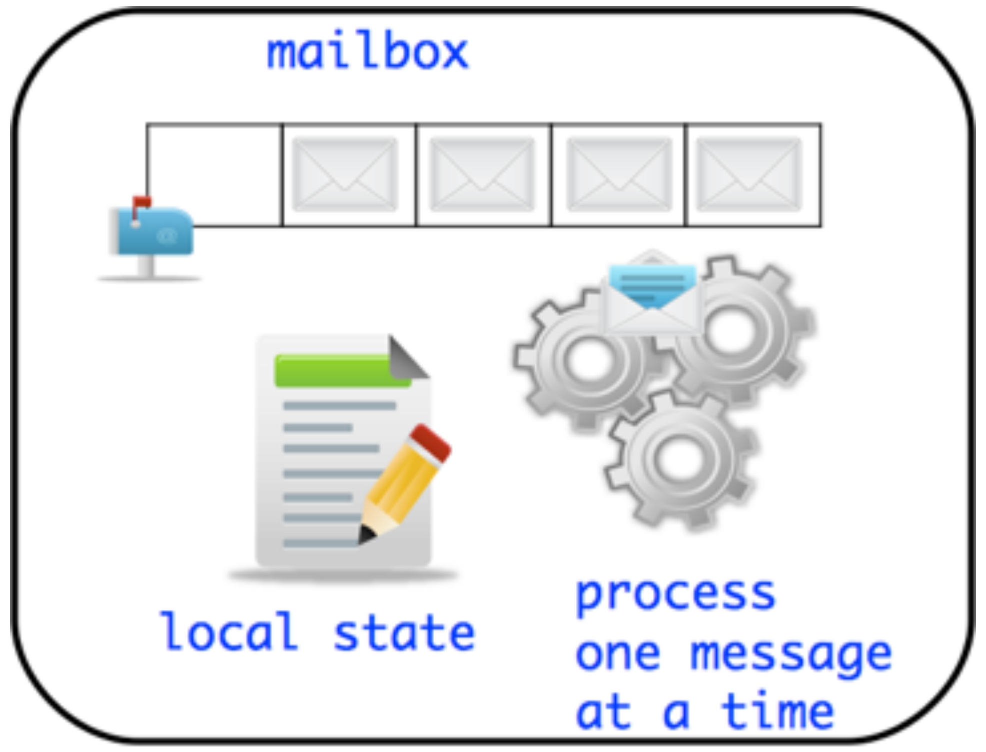
Actor Life Cycle (Lecture 21)

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



Worksheet #21: Interaction between finish and actors

What output will be printed if the end-finish operation from slide 15 is moved from line 13 to line 11 as shown below?

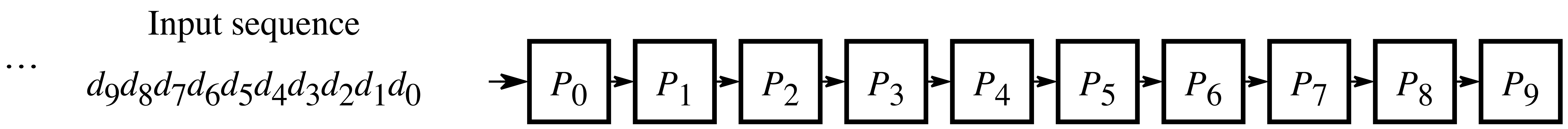
```
1. finish(() -> {
2.    int threads = 4;
3.    int numberOfHops = 10;
4.    ThreadRingActor[] ring = new ThreadRingActor[threads];
5.    for(int i=threads-1;i>=0; i--) {
6.      ring[i] = new ThreadRingActor(i);
7.      ring[i].start(); // like an async
8.      if (i < threads - 1) {
9.        ring[i].nextActor(ring[i + 1]);
10.      } }
11. }); // finish
12.ring[threads-1].nextActor(ring[0]);
13.ring[0].send(numberOfHops);
14.
```

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().

Worksheet #22: 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.




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

WORK = 300, CPL = 102

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);
      ...
} }
```

Synchronized statements and methods in Java (Lecture 24)

- Every Java object has an associated 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 the 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!

Deadlock avoidance in HJ with object-based isolation

- HJ implementation ensures that all locks are acquired in the same order
- ==> no deadlock

```
public class ObviousDeadlock {
   . . .
   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 27)

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

    Condition newCondition();  // associate a new condition
}
```

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

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

Hashtable Example

```
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();
  }
}
```

Worksheet #27 Solution: Use of trylock()

Rewrite the transferFunds() method below to use j.u.c. locks with calls to tryLock (see slide 4) 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 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
 }
```

Linearizability of Concurrent Objects (Lecture 28)

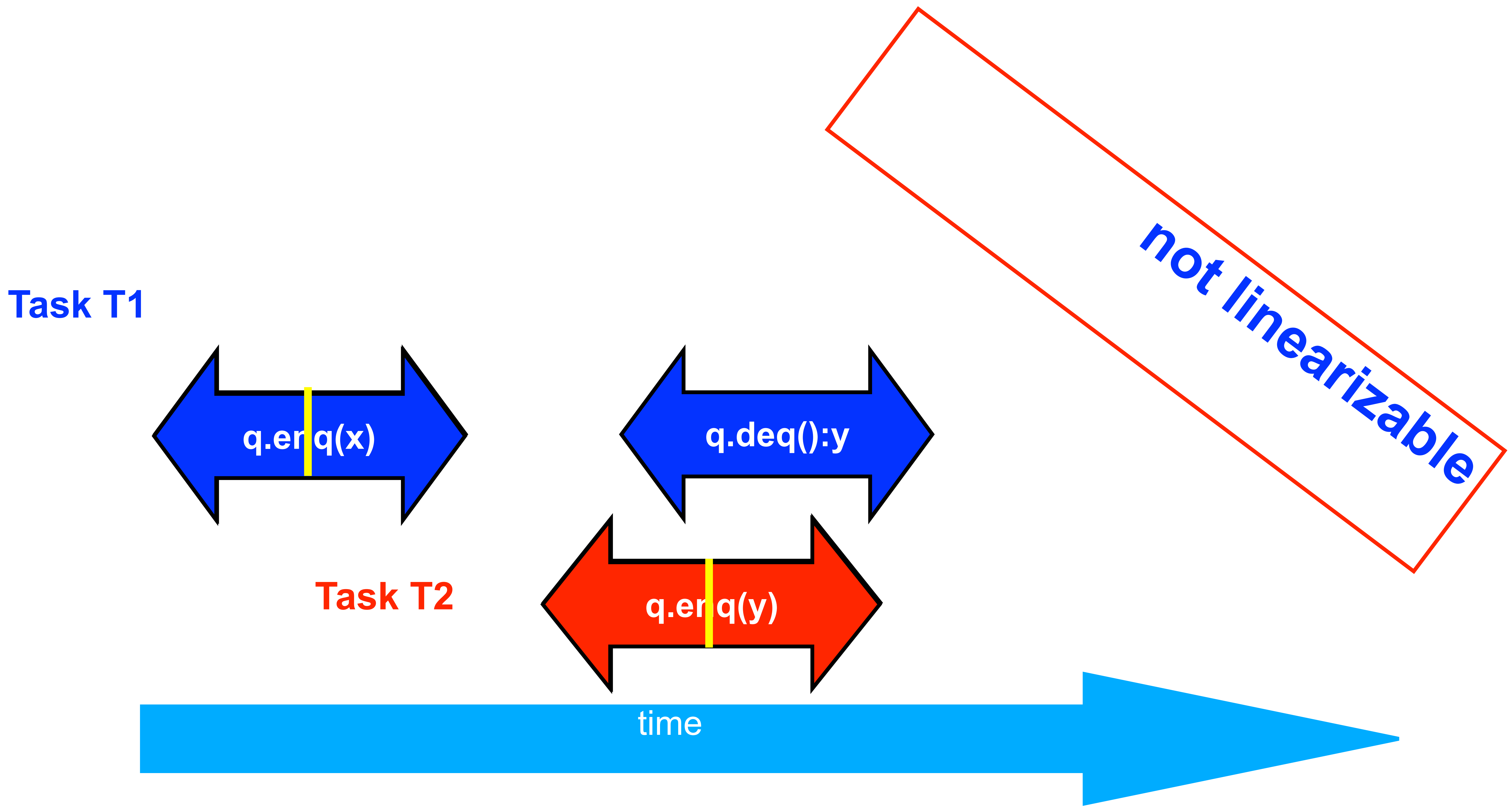
Concurrent object

- 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

Example 5: execution of a concurrent implementation of a FIFO queue q

Is this a linearizable execution?

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

Yes! Can be linearized as "q.enq(x) ; q.enq(y) ; q.deq():x"

Organization of a Distributed-Memory Multiprocessor (Lecture 31 - Start of Module 3)

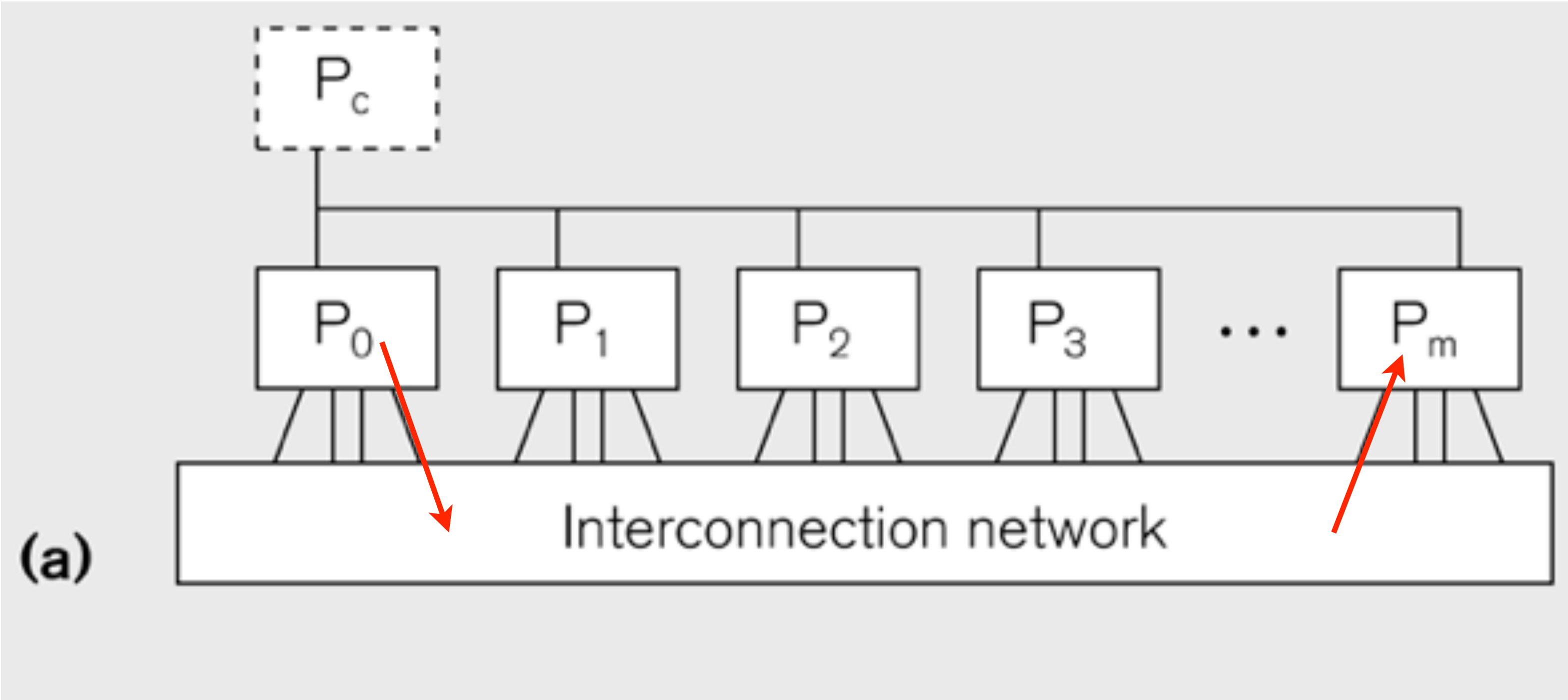
Figure (a)

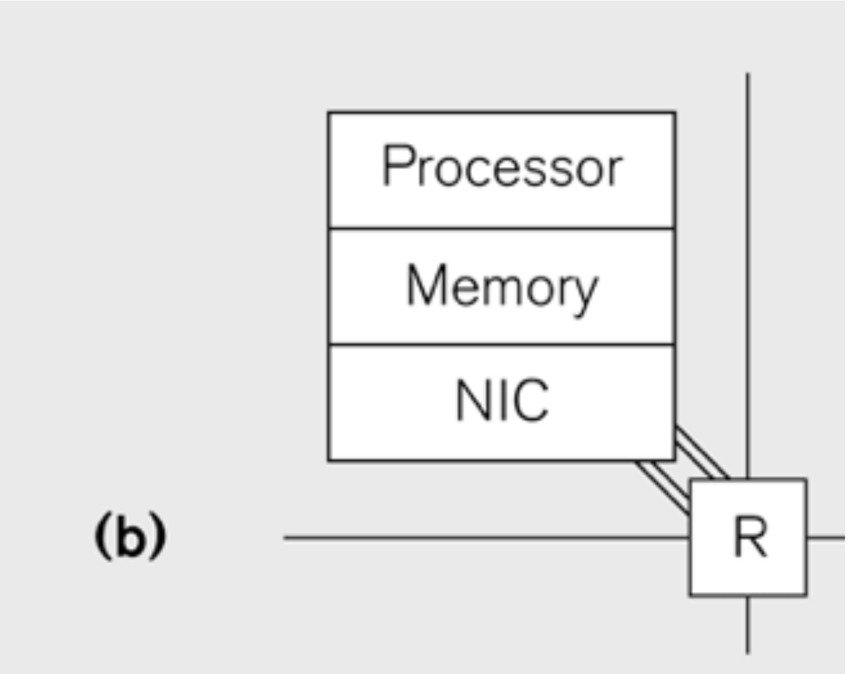
- Host node (P_c) connected to a cluster of processor nodes (P_0 … P_m)
- Processors P_0 … P_m communicate via an interconnection network which could be standard TCP/IP (e.g., for Map-Reduce) or specialized for high performance communication (e.g., for scientific computing)

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)

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

```java
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 #31: MPI send and receive

In the space below, indicate what values you expect the print statement in line 10 to output, assuming that the program is executed with two MPI processes.

```
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. …
```

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

Non-Blocking Send and Receive Operations (Lecture 32)

- In order to overlap communication with computation, MPI provides a pair of functions for performing non-blocking send and receive operations ("I" stands for "Immediate")

```
Request Isend(Object buf, int offset, int count, Datatype type, int dst, int tag) ;
Request Irecv(Object buf, int offset, int count, Datatype type, int src, int tag) ;
```

- Use Wait() to wait for operation to complete (like future get).

```
Status Wait(Request request)
```

- The Wait() operation is declared to return a Status object. In the case of a non-blocking receive operation, this object has the same interpretation as the Status object returned by a blocking Recv() operation.

Collective Communications (Lecture 33)

- 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(...) ....

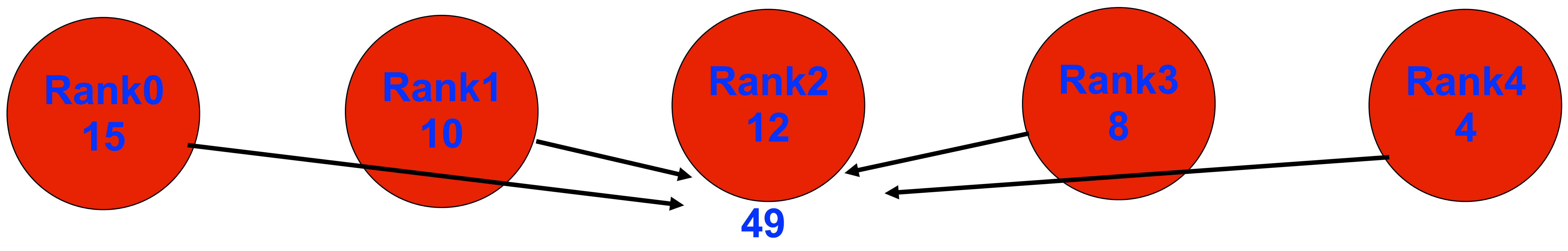
    Process 2: .... MPI.Gather(...) ....

- 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 */ )
```

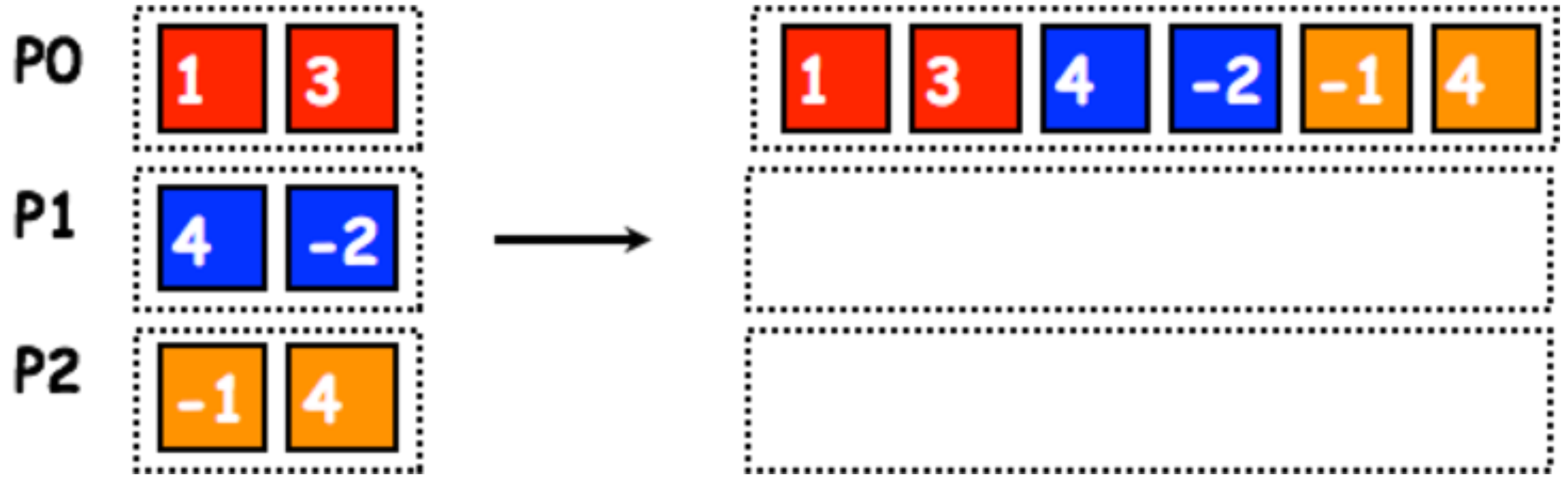


MPI.COMM_WORLD.Reduce(msg, 0, result, 0, 1, MPI.INT, MPI.SUM, 2);

Worksheet #33 Solution: MPI_Gather

In the space below, indicate what value should be provided instead of ??? in line 6, and how it should depend on myrank.

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

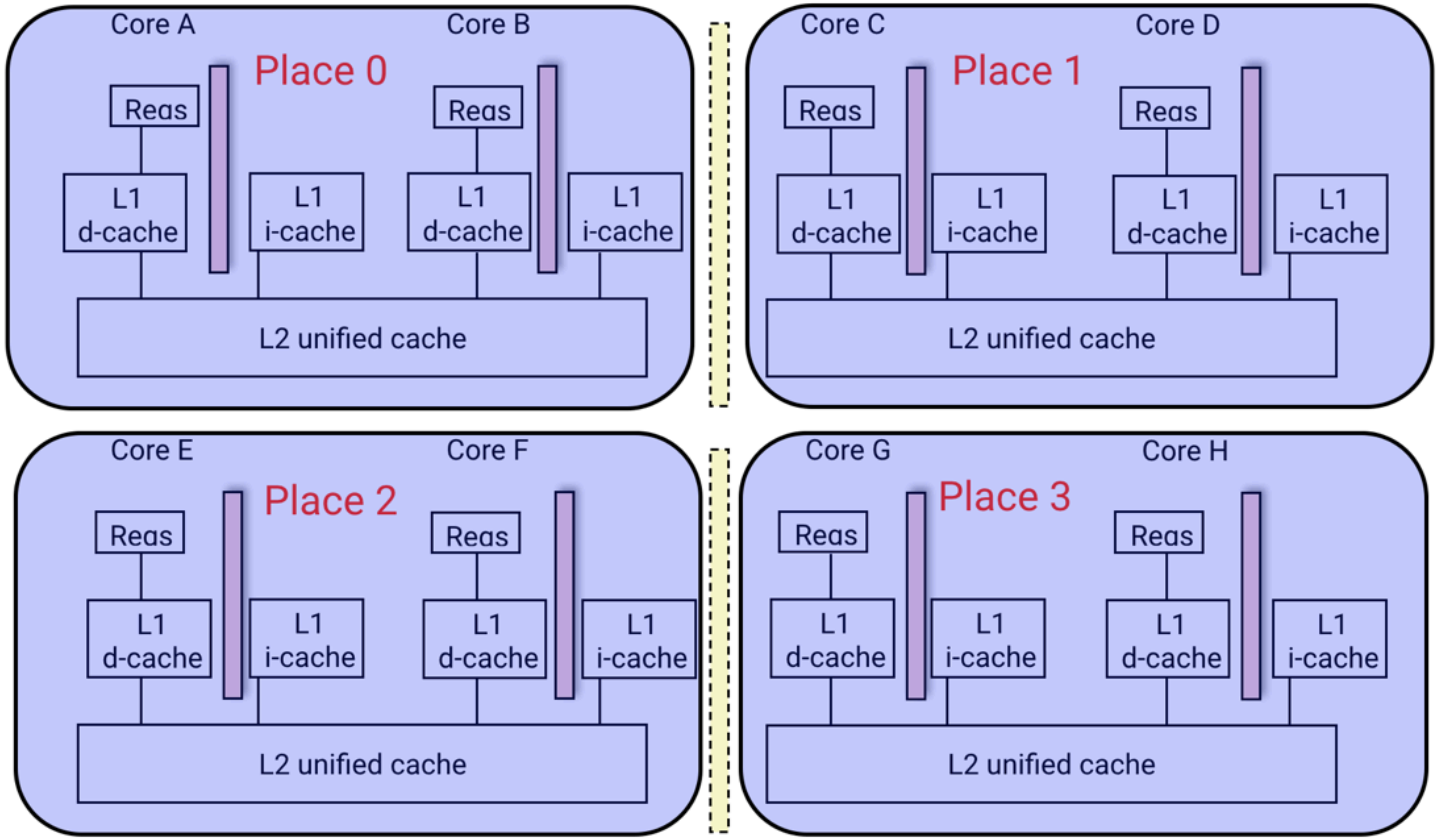


Solution: myrank == 0 ? (size * numProcs) : 0

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: impact of distribution on parallel completion time

```
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 #36: 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, reverse, not(Flags) operations, and to explain your answer.

```
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]
```

Just showing solution to last worksheet. Parallel prefix sum (scan) will not be on final exam.

Worksheet #36: 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	[0]	[1]	2	[2]]
I-up	=	[[3]	[4]	[5]	[6]	6	6	[7]	7]
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.