COMP 322: Fundamentals of Parallel Programming

Lecture 38: Review of Modules 2 & 3 (Lectures 20-37)

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

comp322.rice.edu

HJ isolated construct (Lecture 20)

```
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 (Lecture 20)

```
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

Parallel Spanning Tree Algorithm using object-based isolation (Worksheet 20)

```
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. . . .
```

Parallel Spanning Tree Algorithm using AtomicReference (Lecture 21)

```
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. . . .
```

java.util.concurrent.AtomicInteger methods and their equivalent isolated constructs (Lecture 21)

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 object-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.

java.util.concurrent. AtomicReference methods and their equivalent isolated statements (Lecture 21)

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.

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. }
```

Prefix Sum (Scan) Problem Statement (Lecture 22)

Given input array A, compute output array X as follows

$$X[i] = \sum_{0 \leq j \leq i} A[j]$$

- The above is an inclusive prefix sum since X[i] includes A[i]
- For an exclusive prefix sum, perform the summation for 0 <=j <i
- It is easy to see that inclusive prefix sums can be computed sequentially in O(n) time …

```
// Copy input array A into output array X
X = new int[A.length]; System.arraycopy(A,0,X,0,A.length);
// Update array X with prefix sums
for (int i=1 ; i < X.length ; i++ ) X[i] += X[i-1];
```

- … and so can exclusive prefix sums

Summary of Parallel Prefix Sum Algorithm (Lecture 22)

- Critical path length, CPL = O(log n)
- Total number of add operations, WORK = O(n)
- Optimal algorithm for P = O(n/log n) processors
  — Adding more processors does not help
- Parallel Prefix Sum has several applications that go beyond computing the sum of array elements
  - Parallel Prefix Sum can be used for any operation that is associative (need not be commutative)
    — In contrast, finish accumulators required the operator to be both associative and commutative

Implementing Parallel Filter using Parallel Prefix Sum (Lecture 22)

1. Parallel map to compute a bit-vector for true elements (can use Java streams)

```
input  [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]
bits   [1,  0, 0, 0,  1, 0,  1,  1, 0,  1]
```

2. Parallel-prefix sum on the bit-vector (not available in Java streams)

```
bitsum [1,  1, 1, 1,  2, 2,  3,  4, 4,  5]
```

3. Parallel map to produce the output (can use Java streams)

```
output [17, 11, 13, 19, 24]
```

```
output = new array of size bitsum[n-1]
FORALL(i=0; i < input.length; i++){
  if(bits[i]==1)
    output[bitsum[i]-1] = input[i];
}
```

Two-way Parallel Array Sum using Java Threads (Lecture 23)

```
1.  // Start of main thread
2.  sum1 = 0 sum2 = 0; // sum1 & sum2 are static fields
3.  Thread t1 = new Thread(() -> {
4.      // Child task computes sum of lower half of array
5.      for(int i=0; i < X.length/2; i++) sum1 += X[i];
6.    });
7.  t1.start();
8.  // Parent task computes sum of upper half of array
9.  for(int i=X.length/2; i < X.length; i++) sum2 += X[i];
10. // Parent task waits for child task to complete (join)
11. t1.join();
12. return sum1 + sum2;
```

Deadlock example with Java synchronized statement (Lecture 23)

- The code below can deadlock if `leftHand()` and `rightHand()` are called concurrently from different threads
  - Because the locks are not acquired in the same order

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

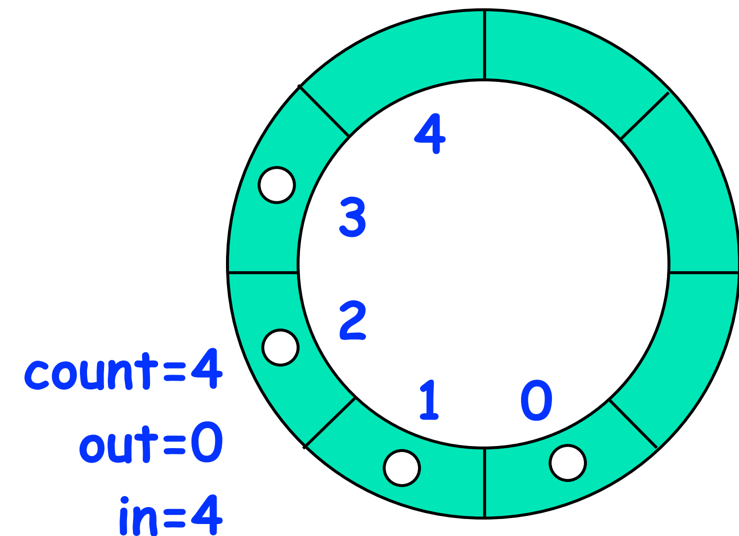
Avoiding Dynamic Order Deadlocks (Lecture 23)

- The solution is to induce a lock ordering
  - For example, use 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);
         }
      }
    }
  }
```

What if you want to wait for shared state to satisfy a desired property? (Circular Bounded Buffer Example. Lecture 24)

```
1. public synchronized void insert(Object item) { // producer
2.      // TODO: wait till count < BUFFER SIZE
3.      ++count;
4.      buffer[in] = item;
5.      in = (in + 1) % BUFFER SIZE;
6.      // TODO: notify consumers
7. }
8.
9. public synchronized Object remove() { // consumer
10.     Object item;
11.    // TODO: wait till count > 0
12.     --count;
13.     item = buffer[out];
14.     out = (out + 1) % BUFFER SIZE;
15.     // TODO: notify producers
16.     return item;
17.}
```



insert() & remove() with wait/notify methods for Circular Bounded Buffer (Lecture 24)

```
1. public synchronized void insert(Object item) {
2.     while (count == BUFFER SIZE) wait();
3.     ++count;
4.     buffer[in] = item;
5.     in = (in + 1) % BUFFER SIZE;
6.     notify();
7. }
8.
9. public synchronized Object remove() {
10.     Object item;
11.     while (count == 0) wait();
12.     --count;
13.     item = buffer[out];
14.     out = (out + 1) % BUFFER SIZE;
15.     notify();
16.     return item;
17.}
```

Linearizability of Concurrent Objects (Summary, Lecture 25)

<u>Concurrent object</u>

- A concurrent object is an object that can correctly handle methods invoked in parallel by different tasks or threads
  — Examples: concurrent queue, AtomicInteger

<u>Linearizability</u>

- 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? (Lecture 25)

Task T1

q.enq(x)

q.deq():y

Task T2

q.enq(y)

time

not linearizable

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

Worksheet #26 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; //acquire from.lock
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.  }
```

Liveness (Lecture 27)

- Liveness = a program's ability to make progress in a timely manner
- Is termination a requirement for liveness?
  - some applications are designed to be non-terminating
- Different levels of liveness guarantees (from weaker to stronger)
  1. Deadlock freedom (can't have all threads blocked)
  2. Livelock freedom (can't have all threads doing "busy work" with no progress)
  3. Starvation freedom (can't have any thread blocked forever)
  4. Bounded wait (can't have any thread blocked for an unbounded time)

Worksheet #27 solution: Liveness Guarantees

```
/** Atomically adds delta to the current value.
 *
 * @param delta the value to add
 * @return the previous value
 */
public final int getAndAdd(int delta) {
    for (;;) {
        int current = get();
        int next = current + delta;
        if (compareAndSet(current, next))
             return current;
    }
}
```

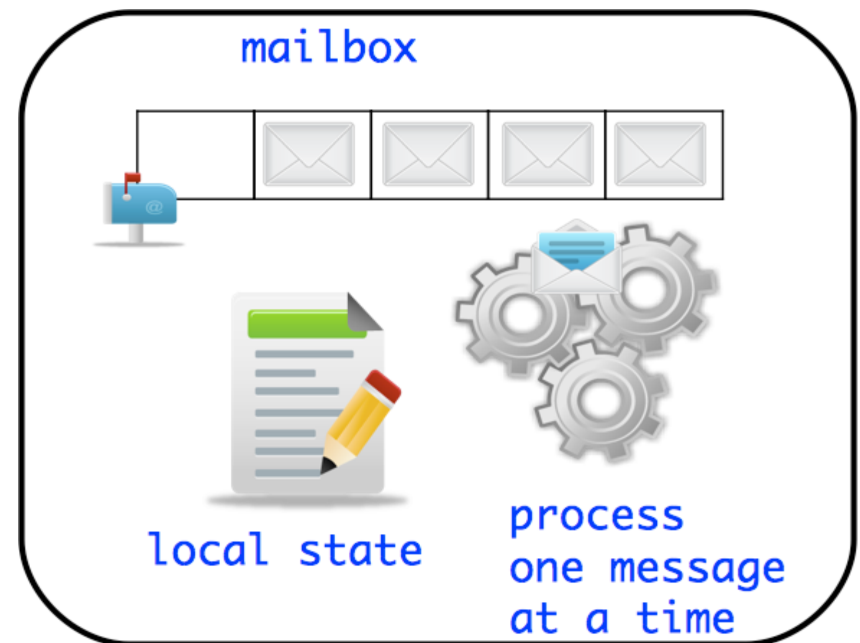
Assume that multiple tasks call getAndAdd() repeatedly in parallel. Can this implementation of getAndAdd() lead to a) deadlock, b) livelock, or c) starvation? Write and explain your answer below.

SOLUTION: c) starvation is possible, but a) deadlock and b) livelock are not possible
NOTE 1: a terminating parallel program execution exhibits none of a), b), or c).

Actor Life Cycle (Lecture 28)

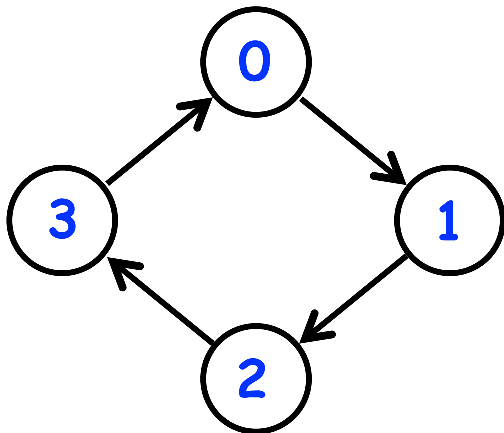
Actor states

- **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 Example (Lecture 28)

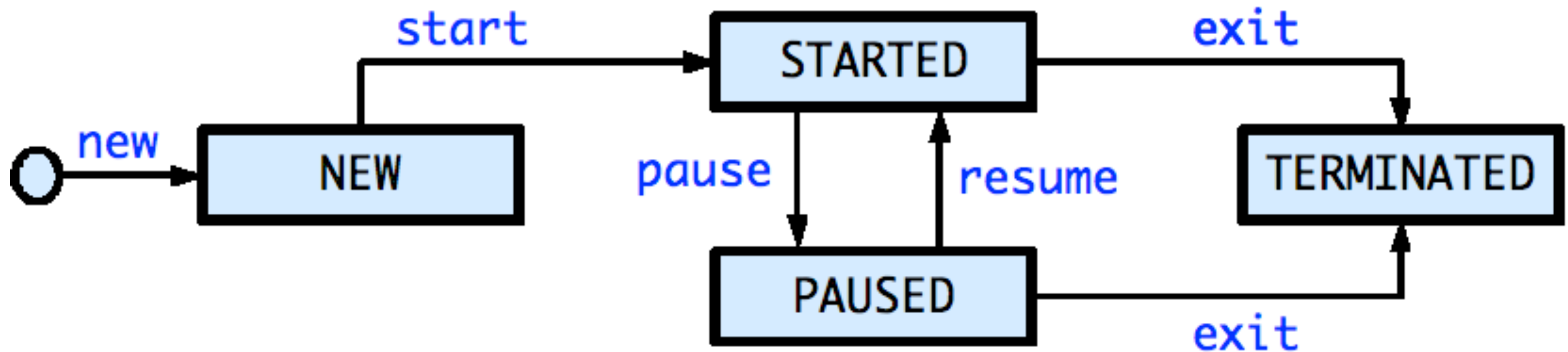
```
1.  finish(() -> {
2.    int threads = 4;
3.    int numberOfHops = 10;
4.    ThreadRingActor[] ring =
        new ThreadRingActor[threads];
5.    for(int i=threads-1;i>=0; i--) {
6.      ring[i] = new ThreadRingActor(i);
7.      ring[i].start();
8.      if (i < threads - 1) {
9.        ring[i].nextActor(ring[i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

State Diagram for Extended Actors with Pause-Resume (Lecture 29)

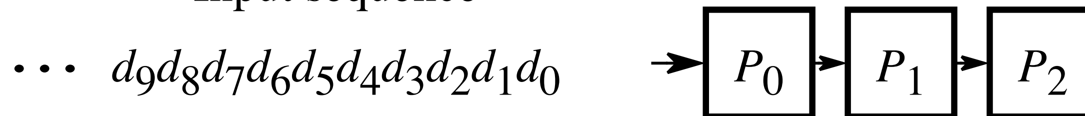


- Paused state: actor will not process subsequent messages until it is resumed
- Resume actor when it is safe to process the next message
- Messages can accumulate in mailbox when actor is in PAUSED state (s in NEW state)

Worksheet #29: 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




```
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()
```

Worksheet #30: Characterizing Solutions to the Dining Philosophers Problem

For the five solutions studied in today's lecture, indicate in the table below which of the following conditions are possible and why:

1. Deadlock: when all philosopher tasks are blocked (neither thinking nor eating)
2. Livelock: when all philosopher tasks are executing but ALL philosophers are starved (never get to eat)
3. Starvation: when one or more philosophers are starved (never get to eat)
4. Non-Concurrency: when more than one philosopher cannot eat at the same time, even when resources are available

	Deadlock	Livelock	Starvation	Non-concurrency
Solution 1: synchronized	Yes (72/73)	No (68/73)	Yes (50/73)	Yes (22/73)
Solution 2: tryLock/unLock	No (73/73)	Yes (45/73)	Yes (67/73)	Yes (15/73)
Solution 3: isolated	No (71/73)	No (72/73)	Yes (26/73)	Yes (67/73)
Solution 4: object-based isolation	No (71/73)	No (67/73)	Yes (64/73)	No (64/73)
Solution 5: semaphores w/ FIFO queues	No (71/73)	No (71/73)	No (57/73)	No (71/73)

Places in HJlib (Lecture 32)

`here()` = place at which current task is executing

`numPlaces()` = total number of places (runtime constant)

Specified by value of **p** in runtime option:

```
HjSystemProperty.numPlaces.set(p);
```

`place(i)` = place corresponding to index i

`<place-expr>.toString()` returns a string of the form "place(id=0)"

`<place-expr>.id()` returns the id of the place as an int

`asyncAt(P, () -> S)`

- Creates new task to execute statement S at place P
- `async(() -> S)` is equivalent to `asyncAt(here(), () -> S)`
- Main program task starts at `place(0)`

Note that `here()` in a child task refers to the place P at which the child task is executing, not the place where the parent task is executing

Example of 4:2 option on an 8-core node (4 places w/ 2 workers per place, Lecture 32)

```
// 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);
```

Core A Core B
Place 0
Regs Regs
L1 L1 L1 L1
L2 unified cache

Core C Core D
Place 1
Regs Regs
L1 L1 L1 L1
L2 unified cache

Core E Core F
Place 2
Regs Regs
L1 L1 L1 L1
L2 unified cache

Core G Core H
Place 3
Regs Regs
L1 L1 L1 L1
L2 unified cache

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

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

Block Distribution (Lecture 32)

- A block distribution splits the index region into contiguous subregions, one per place, while trying to keep the subregions as close to equal in size as possible.
- Block distributions can improve the performance of parallel loops that exhibit spatial locality across contiguous iterations.
- Example: dist.get(index) for a block distribution on 4 places, when index is in the range, 0…15

<table>
<tr><td>Index</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td></tr>
<tr><td>Place id</td><td colspan="4">0</td><td colspan="4">1</td><td colspan="4">2</td><td colspan="4">3</td></tr>
</table>

Cyclic Distribution (Lecture 32)

- A cyclic distribution "cycles" through places 0 … place.MAX PLACES − 1 when spanning the input region
- Cyclic distributions can improve the performance of parallel loops that exhibit load imbalance
- Example: dist.get(index) for a cyclic distribution on 4 places, when index is in the range, 0…15

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Place id	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

Worksheet #32 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)

Our First MPI Program (mpiJava version, Lecture 33)

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. }
```

Worksheet #33 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.

Simple Irecv() example (Lecture 34)

- **The simplest way of waiting for completion of a single non-blocking operation is to use the instance method Wait() in the Request class, e.g:**

```
// Post a receive (like a “communication async”)
Request request = Irecv(intBuf, 0, n, MPI.INT,
                        MPI.ANY_SOURCE,  0) ;
// Do some work while the receive is in progress
…
// Wait for message to arrive (like a future get)
Status status = request.Wait() ;
// Do something with data received in intBuf
…
```

- **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 34)

- 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(...) ....

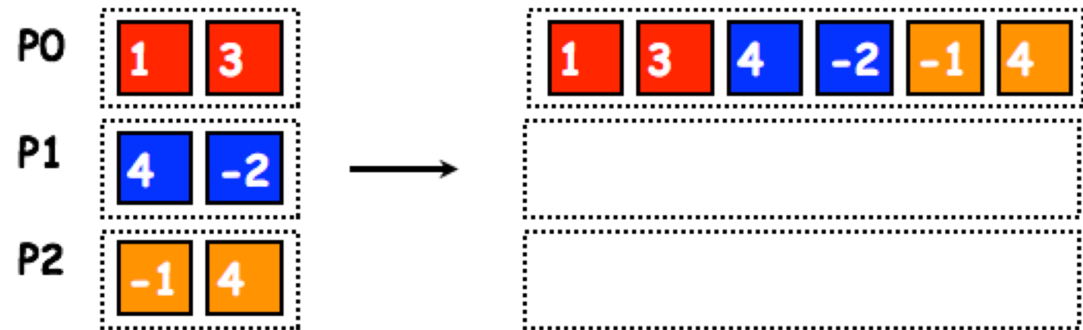
- A simple example is the broadcast operation: all processes invoke the operation, all agreeing on one root process. Data is broadcast from that root.

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

  - Broadcast a message from the process with rank root to all processes of the group

Worksheet #34: 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`

Execution of a CUDA program (Lecture 35)

Host Code
(small number of threads)

Explicit host-device communication

SIMD SIMD

Device Kernel
(large number of threads)

Block 0 Block 1 . . .

. . .

Explicit host-device communication

Host Code
(small number of threads)

Explicit host-device communication

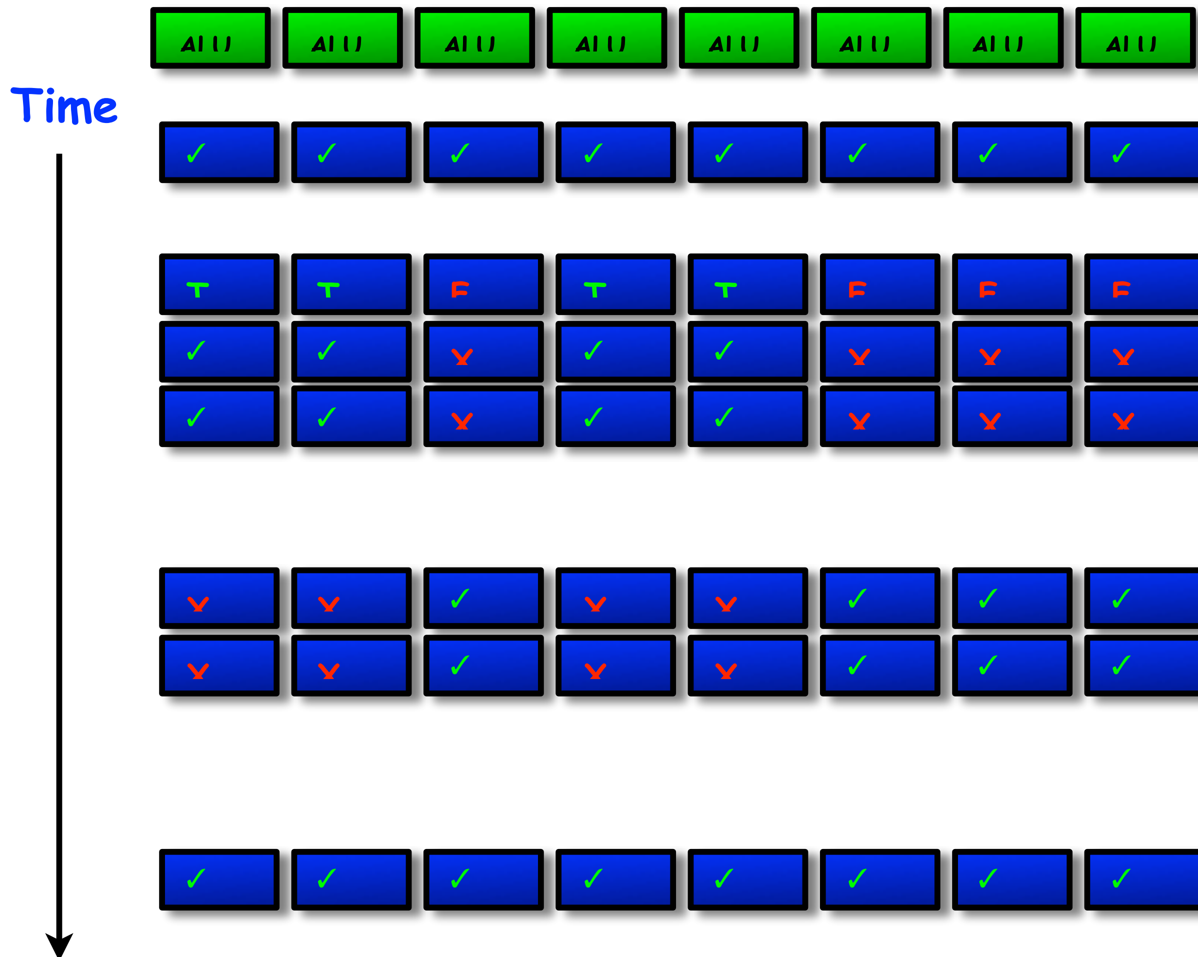
Device Kernel
(large number of threads)

. . .

Explicit host-device communication

Host Code
(small number of threads)

SIMD "lock-step"execution for threads in the same block (Lecture 35)




```
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 #35: Branching in SIMD code

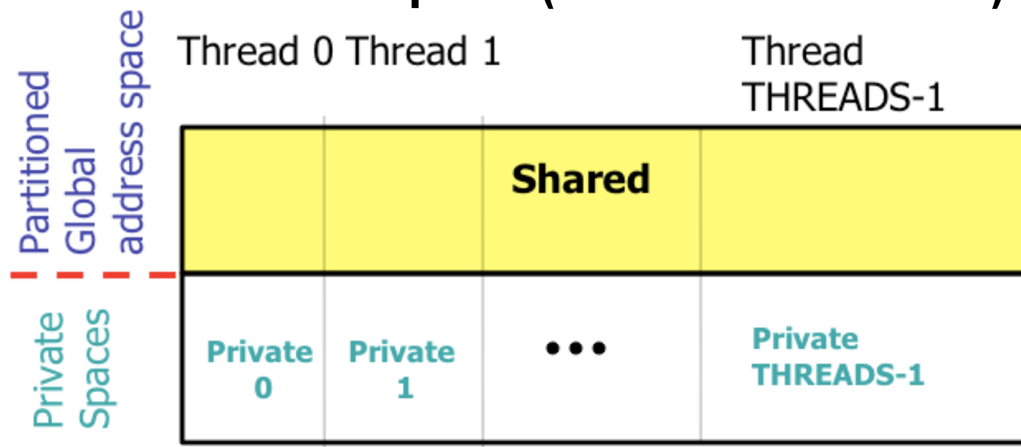
Consider SIMD execution of the following pseudocode with 8 threads in a block. Assume that each call to doWork(x) takes x units of time, and ignore all other costs. How long will this program take when executed on 8 GPU cores, taking into consideration the branching issues discussed in Slide 9?

```
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)

Unified Parallel C (UPC) Execution Model (Lecture 36)

- Multiple threads working independently in a SPMD fashion
  - MYTHREAD specifies thread index (0..THREADS-1)
    - Like MPI processes and ranks
  - # threads specified at compile-time or program launch time
- Partitioned Global Address Space (different from MPI)



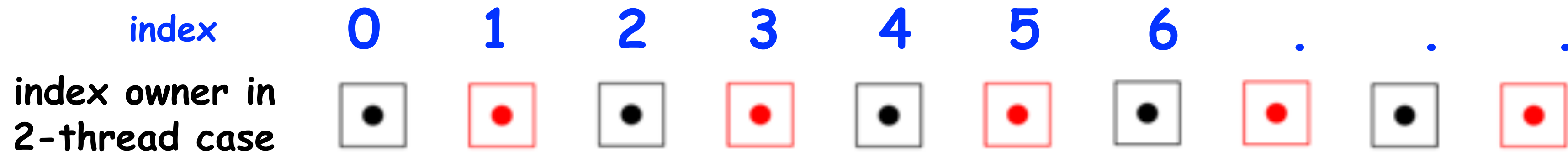
- Threads synchronize as necessary using
  - synchronization primitives
  - shared variables

Worksheet #36: UPC data distributions

In the following example from Lecture 36 slide 20, assume that each UPC array is distributed by default across threads with a *cyclic* distribution. In the space below, identify an iteration of the upc_forall construct for which all array accesses are local, and an iteration for which all array accesses are non-local (remote).

Assume 2 <= THREADS < 100. Explain your answer in each case.

```
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];
```



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**

How did COMP 322 work out this semester?

- What worked (relatively) well
  - Course software: Java 8, HJlib, AutoGrader, Abstract Metrics
  - Course material: Worksheets, labs, videos, quizzes, lecture handouts
  - Organization: Piazza, reduced grading delays compared to previous years
- What was challenging
  - Performance variability for Java on your laptops vs. NOTS vs. AutoGrader
- What we would like to improve in the future
  - Extend lecture handouts
  - New programming examples for labs and homeworks
  - Improved debugging in AutoGrader e.g., automatic datarace detection
- Help us improve COMP 322 in the future!
  - Send us your suggestions for improvement
  - Serve as a TA next year
  - Sign up (and get paid!) to work on improving course material and software

Announcements

- Homework 5 due today (officially) with penalty-free extension until 12noon on May 2nd
  - Any remaining slip days can be applied past May 2nd
- Exam 2 is a scheduled final exam to be held during 9am - 12noon on Tuesday, May 3rd, in Herzstein Hall Auditorium
  - Final exam will cover material from Lectures 20 - 37
  - A practice exam & solution will be made available this weekend
- Group office hours will be held next week in Herzstein 212 at the following times
  - 1pm - 3pm, Monday, April 25th
  - 1pm - 3pm, Wednesday, April 27th
  - 1pm - 3pm, Friday, April 29th

Acknowledgments

- Co-instructor
  - Shams Imam
- Graduate TAs
  - Max Grossman (Head TA), Prasanth Chatarasi, Arghya Chatterjee, Yuhan Peng, Jonathan Sharman
- Undergraduate TAs
  - Prudhvi Boyapalli, Peter Elmers, Nicholas Hanson-Holtry, Ayush Narayan, Timothy Newton, Alitha Partono, Tom Roush, Hunter Tidwell, Bing Xue
- Administrative Staff
  - Annepha Hurlock, Bel Martinez

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