COMP 322: Fundamentals of Parallel Programming

Lecture 39: Course Review

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

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
**Places in HJ (Lectures 17, 18)**

*here* = place at which current task is executing

*place.MAX_PLACES* = total number of places (runtime constant)
  
  Specified by value of *p* in runtime option, `-places p:w`

*place.factory.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

*async at(P) S*
  
  • Creates new task to execute statement *S* at place *P*
  
  • *async S* is equivalent to *async at(here) S*
  
  • Main program task starts at *place.factory.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.
Distributions --- hj.lang.dist

• A distribution maps points in a rectangular index space (region) to places e.g.,
  — \( i \rightarrow \text{place.factory.place}(i \% \text{place.MAX_PLACES}-1) \)

• Programmers are free to create any data structure they choose to store and compute these mappings

• For convenience, the HJ language provides a predefined type, hj.lang.dist, to simplify working with distributions

• Some public members available in an instance \( d \) of hj.lang.dist are:
  — \( d\text{.rank} = \) number of dimensions in the input region for distribution \( d \)
  — \( d\text{.get}(p) = \) place for point \( p \) mapped by distribution \( d \). It is an error to call \( d\text{.get}(p) \) if \( p\text{.rank} \neq d\text{.rank} \).
  — \( d\text{.places}() = \) set of places in the range of distribution \( d \)
  — \( d\text{.restrictToRegion}(pl) = \) region of points mapped to place \( pl \) by distribution \( d \)
Block Distribution

- `dist.factory.block([lo:hi])` creates a block distribution over the one-dimensional region, `lo:hi`.

- A block distribution splits the 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.

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place id</td>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cyclic Distribution

- dist.factory.cyclic([lo:hi]) creates a cyclic distribution over the one-dimensional region, lo:hi.

- 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 in Table 3: dist.factory.cyclic([0:15]) for 4 places

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

- Example in Table 4: dist.factory.cyclic([0:7,0:1]) for 4 places

<table>
<thead>
<tr>
<th>Index</th>
<th>[0,0]</th>
<th>[0,1]</th>
<th>[1,0]</th>
<th>[1,1]</th>
<th>[2,0]</th>
<th>[2,1]</th>
<th>[3,0]</th>
<th>[3,1]</th>
<th>[4,0]</th>
<th>[4,1]</th>
<th>[5,0]</th>
<th>[5,1]</th>
<th>[6,0]</th>
<th>[6,1]</th>
<th>[7,0]</th>
<th>[7,1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place id</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Homework 5: Solution to Problem 1a

1. \texttt{dist d = dist.factory.block([1:N]);}
2. \texttt{for (point [iter] : [0:M-1]) {}
3. \texttt{\quad finish for(int j=1; j<=N; j++)}
4. \texttt{\quad async at(d[j]) {
5. \texttt{\quad myNew[j] = (myVal[j-1] + myVal[j+1]) / 2.0;}
6. \texttt{\quad } //finish-for-async-at
7. \texttt{\quad double[] temp = myNew; myNew = myVal; myVal = temp;}
8. \texttt{\quad } } // for

Number of remote reads for block distribution ~ 2*M*P

Number of remote reads for cyclic distribution ~ 2*M*N
HJ isolated statement (Lectures 20, 21, 37)

isolated <body>

- Two tasks executing isolated statements with interfering accesses must perform the isolated statement in mutual exclusion
  - Two instances of isolated statements, ⟨stmt1⟩ and ⟨stmt2⟩, are said to interfere with each other if both access a shared location, such that at least one of the accesses is a write.

⇒ Weak isolation guarantee: no mutual exclusion applies to non-isolated statements i.e., to (isolated, non-isolated) and (non-isolated, non-isolated) pairs of statement instances

- Isolated statements may be nested (redundant)

- Isolated statements must not contain any other parallel statement that performs a blocking operation: finish, get, next
  - Non-blocking operations (e.g., async) are fine
Object-based isolation in HJ

\texttt{isolated(<object-list>) <body>}

- In this case, programmer specifies list of objects for which isolation is required
- Mutual exclusion is only guaranteed for instances of isolated statements that have a non-empty intersection in their object lists
  - Standard isolated is equivalent to isolated(\*) by default i.e., isolation across all objects
- Implementation can choose to distinguish between read/write accesses for further parallelism
  - Current HJ implementation supports object-based isolation, does not exploit read/write distinction
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`.

<table>
<thead>
<tr>
<th>j.u.c.atomic Class and Constructors</th>
<th>j.u.c.atomic Methods</th>
<th>Equivalent HJ isolated statements</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>AtomicInteger</code></td>
<td>int j = v.get();</td>
<td>int j; isolated (v) j = v.val;</td>
</tr>
<tr>
<td></td>
<td>v.set(newVal);</td>
<td>isolated (v) v.val = newVal;</td>
</tr>
<tr>
<td></td>
<td>int j = v.getAndSet(newVal);</td>
<td>int j; isolated (v) { j = v.val; v.val = newVal; }</td>
</tr>
<tr>
<td></td>
<td>int j = v.addAndGet(delta);</td>
<td>isolated (v) { v.val += delta; j = v.val; }</td>
</tr>
<tr>
<td></td>
<td>int j = v.getAndAdd(delta);</td>
<td>isolated (v) { j = v.val; v.val += delta; }</td>
</tr>
<tr>
<td><code>AtomicInteger()</code></td>
<td>boolean b =</td>
<td>boolean b;</td>
</tr>
<tr>
<td></td>
<td>v.compareAndSet</td>
<td>isolated (v)</td>
</tr>
<tr>
<td></td>
<td>(expect,update);</td>
<td>if (v.val==expect) {v.val=update; b=true;}</td>
</tr>
<tr>
<td>// init = 0</td>
<td></td>
<td>else b = false;</td>
</tr>
</tbody>
</table>
Parallel Spanning Tree Algorithm using isolated statement

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

Parallel Spanning Tree Algorithm using object-based isolation

1. class V {
2.     V [] neighbors; // adjacency list for input graph
3.     AtomicReference parent; // output value of parent in spanning tree
4.     boolean tryLabeling(V n) {
5.         isolated(this) if (parent == null) parent=n;
6.         return parent == n;
7.     } // tryLabeling
8.     void compute() {
9.         for (int i=0; i<neighbors.length; i++) {
10.            V child = neighbors[i];
11.            if (child.tryLabeling(this))
12.                async child.compute(); //escaping async
13.         }
14.     } // compute
15. } // class V
16. ...
17. root.parent = root; // Use self-cycle to identify root
18. finish root.compute();
19. ...
Parallel Spanning Tree Algorithm using java.util.concurrent.atomic.AtomicReference

1. class V {
2.     V [] neighbors; // adjacency list for input graph
3.     AtomicReference parent; // output value of parent in spanning tree
4.     boolean tryLabeling(V n) {
5.         return parent.compareAndSet(null , n);
6.     }
7. } // tryLabeling
8. void compute() {
9.     for (int i=0; i<neighbors.length; i++) {
10.        V child = neighbors[i];
11.        if (child.tryLabeling(this))
12.            async child.compute(); // escaping async
13.    }
14. } // compute
15.} // class V
16. . . .
17. root.parent = root; // Use self-cycle to identify root
18. finish root.compute();
19. . . .
The Actor Model (Lectures 21, 22, 23)

- An actor may:
  - process messages
  - read/write local state
  - create a new actor
  - start a new actor
  - send messages to other actors
  - terminate

- An actor processes messages sequentially
  - guaranteed mutual exclusion on accesses to local state
Actor Life Cycle

Actor states

- **New**: Actor has been created
  - e.g., email account has been created
- **Started**: Actor can receive and process messages
  - e.g., email account has been activated
- **Terminated**: Actor will no longer processes messages
  - e.g., termination of email account after graduation
Using Actors in HJ

- Create your custom class which extends hj.lang.Actor<Object> , and implement the void process() method

```java
class MyActor extends Actor<Object> {
   protected void process(Object message) {
      System.out.println("Processing " + message);
   }
}
```

- Instantiate and start your actor
`
Actor<Object> anActor = new MyActor(); anActor.start();
```

- Send messages to the actor
`
anActor.send(aMessage); //aMessage can be any object in general
```

- Use a special message to terminate an actor
`
protected void process(Object message) {
   if (message.someCondition()) exit();
}
```

- Actor execution implemented as async tasks in HJ

- Can use finish to await their completion
Simple Pipeline

A Simple pipeline with 3 stages

Stage-1: Filter even length strings

Stage-2: Filter lowercase strings

Stage-3: Print results

Simple pipeline with stages
Simple Pipeline using HJ Actors

1. // Main program
2. finish {
3. Actor<Object> firstStage =
4. new EvenLengthFilter(
5. new LowerCaseFilter(
6. new LastStage()));
7. firstStage.start(); // starts others
8. firstStage.send("pipeline");
9. firstStage.send(new StopMessage());
10. }
11. 
12. class LastStage extends Actor {
13. protected void process(Object msg) {
14. if (msg instanceof StopMessage) {
15. exit();
16. } else if (msg instanceof String) {
17. System.out.println(msg);
18. } } } 

Sends are asynchronous in actor model, but HJ Actor library preserves order of messages between same sender and receiver
Simple Pipeline using HJ Actors (contd)

19. class LowerCaseFilter extends Actor {
20.     protected void process(Object msg) {
21.         if (msg instanceof StopMessage) {
22.             exit(); nextStage.send(msg);
23.         } else if (msg instanceof String) {
24.             String str = (String) msg;
25.             if (str.toLowerCase().equals(str)) {
26.                 nextStage.send(str);
27.             }
28.         }
29.     }
30. }
31. }
32. 
33. class EvenLengthFilter extends Actor {
34.     protected void process(Object msg) {
35.         if (msg instanceof StopMessage) {
36.             nextStage.send(msg);
37.             exit();
38.         } else if (msg instanceof String) {
39.             String msgStr = (String) msg;
40.             if (msgStr.length() % 2 == 0) {
41.                 nextStage.send(msgStr);
42.             }
43.         }
44.     }
45. }
46. }
Adding support for places in HJ actors

• Basic approach: include an optional place parameter in the start() method

```java
Actor<Object> anActor = new MyActor();

anActor.start(p);  // Start actor at place p
```

• Example:

```java
SievePlaceActor nextActor = new SievePlaceActor(...);

// Start actor at next place, relative to current place
nextActor.start(here.next());
```
Summary of Mutual Exclusion approaches in HJ

• Isolated --- analogous to critical sections
• Object-based isolation, isolated(a, b, ...)
• Single object in list --- like monitor operations on object
• Multiple objects in list --- deadlock-free mutual exclusion on sets of objects
• Java atomic variables --- optimized implementation of object-based isolation
• Java concurrent collections --- optimized implementation of monitors
• Actors --- different paradigm from task parallelism (mutual exclusion by default)
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 1

Is this execution linearizable?

Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt
Example 2

Is this execution linearizable?

- $q.enq(x)$
- $q.enq(y)$
- $q.deq():y$
- $q.deq():x$

- time
import java.util.concurrent.atomic.*;

class IQueue {
    AtomicInteger head = new AtomicInteger(0);
    AtomicInteger tail = new AtomicInteger(0);
    Object[] items = new Object[Integer.MAX_VALUE];

    public void enq(Object x) {
        int slot;
        // Loop till enqueue slot is found
        do {
            slot = tail.get();
            if (!tail.compareAndSet(slot, slot + 1)) {
                throw new EmptyException();
            }
        } while (!tail.compareAndSet(slot, slot + 1));
    }

    public Object deq() throws EmptyException {
        Object value; int slot;
        // Loop till dequeue slot is found
        do {
            slot = head.get(); value = items[slot];
            if (value == null) throw new EmptyException();
        } while (!head.compareAndSet(slot, slot + 1));
        return value;
    }
}

Not linearizable. Consider { async enq(A); enq(B); deq(); }

Assume that enq(A) pauses between lines 9 and 10
Safety vs. Liveness (Lecture 25)

• In a concurrent setting, we need to specify both the safety and the liveness properties of an object
• Need a way to define
  — Safety: when an implementation is correct
  — Liveness: the conditions under which it guarantees progress
• Linearizability is a safety property for concurrent objects
Desirable Properties of Parallel Program Executions

- Data-race freedom
- Termination
  - But some applications are designed to be non-terminating
- Liveness = a program’s ability to make progress in a timely manner
- Different levels of liveness guarantees (from weaker to stronger)
  - Deadlock freedom
  - Livelock freedom
  - Starvation freedom
- Today’s lecture discusses progress guarantees for HJ programs
  - We will revisit progress guarantees for Java concurrency later
Deadlock-Free Parallel Program Executions

• A parallel program execution is deadlock-free if no task’s execution remains incomplete due to it being blocked awaiting some condition.

• Example of a program with a deadlocking execution:

```java
DataDrivenFuture left = new DataDrivenFuture();
DataDrivenFuture right = new DataDrivenFuture();
finish {
    async await (left) right.put(rightBuilder()); // Task1
    async await (right) left.put(leftBuilder()); // Task2
}
```

• In this case, Task1 and Task2 are in a deadlock cycle.
  - Only two constructs can lead to deadlock in HJ: async await or explicit phaser wait (instead of next)
  - There are many mechanisms that can lead to deadlock cycles in other programming models (e.g., locks)
Livelock-Free Parallel Program Executions

• A parallel program execution exhibits livelock if two or more tasks repeat the same interactions without making any progress (special case of nontermination)

• Livelock example:
  
  // Task 1
  incrToTwo(AtomicInteger ai) {
    // increment ai till it reaches 2
    while (ai.incrementAndGet() < 2);
  }

  // Task 2
  decrToNegativeTwo(AtomicInteger ai) {
    // decrement ai till it reaches -2
    while (a.decrementAndGet() > -2);
  }

• Many well-intended approaches to avoid deadlock result in livelock instead

• Any data-race-free HJ program without isolated/atomic-variables/actors is guaranteed to be livelock-free (may be nonterminating in a single task, however)
Starvation-Free Parallel Program Executions

- A parallel program execution exhibits starvation if some task is repeatedly denied the opportunity to make progress
  - Starvation-freedom is sometimes referred to as “lock-out freedom”
  - Starvation is possible in HJ programs, since all tasks in the same program are assumed to be cooperating, rather than competing
    - If starvation occurs in a deadlock-free HJ program, the “equivalent” sequential program must have been non-terminating

- Classic source of starvation: “Priority Inversion” problem for OS threads (usually from different processes)
  - Thread A is at high priority, waiting for result or resource from Thread C at low priority
  - Thread B at intermediate priority is CPU-bound
  - Thread C never runs, hence thread A never runs
  - Fix: when a high priority thread waits for a low priority thread, boost the priority of the low-priority thread
### Selecting the Right Pattern (Lecture 25)
(adapted from page 9, Parallel Programming w/ Microsoft .Net)

<table>
<thead>
<tr>
<th>Application characteristics</th>
<th>Algorithmic pattern</th>
<th>Relevant HJ constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential loop with independent iterations</td>
<td>1) Parallel Loop</td>
<td>forall, forasync</td>
</tr>
<tr>
<td>Independent operations with well-defined control flow</td>
<td>2) Parallel Task</td>
<td>async, finish</td>
</tr>
<tr>
<td>Aggregating data from independent tasks/iterations</td>
<td>3) Parallel Aggregation (reductions)</td>
<td>finish accumulators, atomic variables</td>
</tr>
<tr>
<td>Ordering of steps based on data flow constraints</td>
<td>4) Futures</td>
<td>futures, data-driven tasks</td>
</tr>
<tr>
<td>Divide-and-conquer algorithms with recursive data structures</td>
<td>5) Dynamic Task Parallelism</td>
<td>async, finish</td>
</tr>
<tr>
<td>Repetitive operations on data streams</td>
<td>6) Pipelines</td>
<td>streaming phasers (deterministic), actors (non-deterministic)</td>
</tr>
</tbody>
</table>
Supporting Patterns

1) Master-worker
   - A process or thread (the master) sets up a task queue and manages other threads (the workers) as they grab a task from the queue, carry out the computation, and then return for their next task. This continues until the master detects that a termination condition has been met, at which point the master ends the computation.

2) Single Instruction Multiple Data (SIMD)
   - A supporting pattern for data parallelism, in which a single instruction stream is applied to multiple data elements in parallel.

3) Single Program Multiple Data (SPMD)
   - Multiple copies of a single program are launched typically with their own view of the data. The path through the program for each copy is determined in part based on a unique ID (a rank).
3) SPMD Supporting Pattern

- **SPMD**: Single Program Multiple Data
- Run the same program on \( P \) processing elements (PEs)
- Use the “rank” ... an ID ranging from 0 to \((P-1)\) ... to determine what computation is performed on what data by a given PE
- Different PEs can follow different paths through the same code (unlike the SIMD pattern)
- Convenient pattern for hardware platforms that are not amenable to efficient forms of dynamic task parallelism
  - General-Purpose Graphics Processing Units (GPGPUs)
  - Distributed-memory parallel machines
- Key design decisions --- what data and computation should be replicated or partitioned across PEs?
SPMD Example #2: Iterative Averaging Example (Slide 9, Lecture 13)

1. double[] gVal=new double[n+2]; double[] gNew=new double[n+2];
2. gVal[n+1] = 1; // Boundary condition
3. int Cj = Runtime.getNumOfWorkers();
4. forall (point [jj]:[0:Cj-1]) { // SPMD computation
5.   double[] myVal = gVal; double[] myNew = gNew; // Local copy
6.   for (point [iter] : [0:numIters-1]) {
7.     // Compute MyNew as function of input array MyVal
8.       for (point [j]:getChunk([1:n],[Cj],[jj]))
9.         myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
10.    next; // Barrier before executing next iteration of iter loop
11.   } // for
12.  } // forall

java.lang.Thread class (Lecture 27)

- Execution of a Java program begins with an instance of Thread created by the Java Virtual Machine (JVM) that executes the program’s main() method.

- Parallelism can be introduced by creating additional instances of class Thread that execute as parallel threads.

```java
public class Thread extends Object implements Runnable {
    Thread() { ... } // Creates a new Thread
    Thread(Runnable r) { ... } // Creates a new Thread with Runnable object r
    void run() { ... } // Code to be executed by thread
    // Case 1: If this thread was created using a Runnable object,
    // then that object’s run method is called
    // Case 2: If this class is subclassed, then the run() method
    // in the subclass is called
    void start() { ... } // Causes this thread to start execution
    void join() { ... } // Wait for this thread to die
    void join(long m) // Wait at most m milliseconds for thread to die
    static Thread currentThread() // Returns currently executing thread
    ...
}
```

Listing 3: java.lang.Thread class
Listing 4: Two-way Parallel ArraySum using Java threads

```java
// Start of Task T1 (main program)
sum1 = 0; sum2 = 0; // Assume that sum1 & sum2 are fields (not local vars)
// Compute sum1 (lower half) and sum2 (upper half) in parallel
final int len = X.length;
Runnable r1 = new Runnable() {
    public void run(){
        for(int i=0; i < len/2; i++) sum1 += X[i];
    }
};
Thread t1 = new Thread(r1);
t1.start();
Runnable r2 = new Runnable() {
    public void run(){
        for(int i=len/2; i < len; i++) sum2 += X[i];
    }
};
Thread t2 = new Thread(r2);
t2.start();
// Wait for threads t1 and t2 to complete
t1.join(); t2.join();
int sum = sum1 + sum2;
```
Every Java object has an associated lock acquired via:
- **synchronized statements**
  ```java
synchronized( foo ) { // acquire foo's lock
    // execute code while holding foo's lock
  } // release foo's lock
```
- **synchronized methods**
  ```java
  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

Java's synchronized is related to “mutex” locks in POSIX thread library.
Implementation of Java synchronized statements/methods

- Every object has an associated lock
- “synchronized” is translated to matching monitorenter and monitorexit bytecode instructions for the Java virtual machine
  - monitorenter requests “ownership” of the object’s lock
  - monitorexit releases “ownership” of the object’s lock
- If a thread performing monitorenter does not own the lock (because another thread already owns it), it is placed in an unordered “entry set” for the object’s lock
The Java wait() Method

- A thread can perform a `wait()` method on an object that it owns:
  1. the thread releases the object lock
  2. thread state is set to blocked
  3. thread is placed in the wait set

- Causes thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object.

- Since interrupts and spurious wake-ups are possible, this method should always be used in a loop e.g.,
  ```java
  synchronized (obj) {
      while (<condition does not hold>)
          obj.wait();
      ... // Perform action appropriate to condition
  }
  ```

- Java’s `wait-notify` is related to “condition variables” in POSIX threads
Entry and Wait Sets

Diagram:
- Entry set: circles connected by arrows indicating the flow of acquire lock actions.
- Object lock: a central node labeled "owner".
- Wait set: circles connected by arrows indicating the flow of wait actions.

Legend:
- Acquire lock: Arrows pointing to the object lock from the entry set.
- Wait: Arrows pointing from the object lock to the wait set.
The notify() Method

When a thread calls `notify()`, the following occurs:

1. selects an arbitrary thread \( T \) from the wait set
2. moves \( T \) to the entry set
3. sets \( T \) to Runnable

\( T \) can now compete for the object’s lock again
The `java.util.concurrent.locks.Lock` interface is implemented by the `java.util.concurrent.locks.ReentrantLock` class.

```java
interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    boolean tryLock();
    boolean tryLock(long timeout, TimeUnit unit)
        throws InterruptedException;
    void unlock();
    Condition newCondition();
    // can associate multiple condition vars with lock
}
```

- `java.util.concurrent.locks.Lock` interface is implemented by `java.util.concurrent.locks.ReentrantLock` class
Simple ReentrantLock() example

- Used extensively within `java.util.concurrent`

```java
final Lock lock = new ReentrantLock();

...  
lock.lock();
try {
    // perform operations protected by lock
} // restore invariants & rethrow
catch(Exception ex) {
    finally {
        lock.unlock();
    }
}  
```

- Must manually ensure lock is released
Reading vs. writing

• Recall that the use of synchronization is to protect interfering accesses
  — Multiple concurrent reads of same memory: Not a problem
  — Multiple concurrent writes of same memory: Problem
  — Multiple concurrent read & write of same memory: Problem

So far:
  — If concurrent write/write or read/write might occur, use synchronization to ensure one-thread-at-a-time

But:
  — This is unnecessarily conservative: we could still allow multiple simultaneous readers

Consider a hashtable with one coarse-grained lock
  — So only one thread can perform operations at a time

But suppose:
  — There are many simultaneous lookup operations
  — insert operations are very rare
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.ReentrantLock class
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. }
```
Example of Send and Recv

Send() and Recv() calls are blocking operations by default.
Announcements

• Homework 6 due due by 11:55pm today
  – An automatic 7-day penalty-free extension can be used till April 27th

• Homeworks 4 and 5 will be returned by end of Monday, April 23rd

• Exam 2 is a take-home exam
  – Maximum duration = 2 hours
  – Closed-book, closed-notes, closed-computer
  – Pick up exam from Amanda Nokleby's office (Duncan Hall 3137) any time starting 2pm today
  – Return exam to Amanda's office by 4pm on Friday, April 27th
  – Written exam --- no penalty for minor syntactic errors in program text, so long as the meaning of the program is unambiguous.
  – If you believe there is any ambiguity or inconsistency in a question, you should state the ambiguity or inconsistency that you see, and any assumptions that you make to resolve it.
  – Scope of exam includes Lectures 17-34, excluding Lecture 19 (midterm review) and Lecture 28 (guest lecture)
Acknowledgments

• Graduate TAs
  – Sanjay Chatterjee
  – Deepak Majeti
  – Dragos Sbirlea

• Undergraduate TAs
  – Max Grossman
  – Damien Stone
  – Yunming Zhang

• Research Programmer
  – Vincent Cave

• Additional HJ expert
  – Shams Imam

• Administrative assistant
  – Amanda Nokleby

Have a great summer!!