Lecture 13: Barrier Synchronization (cont'd)
Announcements

• Homework 4 due by 5pm on Wednesday, Feb 16\(^{th}\)
  — We will try and return graded homeworks by Feb 23\(^{rd}\)

• Guest lecture on Bitonic Sort by John Mellor-Crummey on Friday, Feb 18\(^{th}\)

• Feb 23\(^{rd}\) lecture will be a Midterm Review

• No lecture on Friday, Feb 25\(^{th}\) since midterm is due that day
  — Midterm will be a 2-hour take-home written exam
  — Will be given out at lecture on Wed, Feb 23\(^{rd}\)
  — Must be handed in by 5pm on Friday, Feb 25\(^{th}\)
Acknowledgments for Today’s Lecture

• “Principles of Parallel Programming”, Calvin Lin & Lawrence Snyder, Addison-Wesley, 2009
  — Includes resources available at http://www.pearsonhighered.com/educator/academic/product/0,3110,0321487907,00.html

• Handout for Lectures 12 and 13
Barrier Synchronization using HJ’s “next” statement (recap of Hello-Goodbye example)

```java
rank.count = 0; // rank object contains an int field, count
forall (point [i] : [0:m-1]) {
    int r;
    isolated {r = rank.count++;}
    System.out.println("Hello from task ranked " + r);
    next; // Acts as barrier between phases 0 and 1
    System.out.println("Goodbye from task ranked " + r);
}
```

- `next` — each forall iteration suspends at `next` until all iterations arrive (complete previous phase), after which the phase can be advanced

Observation 1: Scope of synchronization is the closest enclosing forall statement

Observation 2: If a forall iteration terminates before executing “next”, then the other iterations do not wait for it

Observation 3: Different forall iterations may perform “next” at different program points e.g., consider a conditional based on the forall index value
Impact of barrier on scheduling forall iterations

Modeling a next operation in the computation graph
Observation 1: Scope of synchronization for “next” is closest enclosing forall statement

```java
forall (point [i] : [0:m-1]) {
    System.out.println("Starting forall iteration " + i);
    next; // Acts as barrier for forall-i
forall (point [j] : [0:n-1]) {
    System.out.println("Hello from task (" + i + ","
                       + j + ")");
    next; // Acts as barrier for forall-j
    System.out.println("Goodbye from task (" + i + ","
                       + j + ")");
} // forall-j
next; // Acts as barrier for forall-i
    System.out.println("Ending forall iteration " + i);
} // forall-i
```
Observation 2: If a forall iteration terminates before “next”, then other iterations do not wait for it

1. `forall (point[i] : [0:m-1]) {`
2. `for (point[j] : [0:i]) {`
3. `   // Forall iteration i is executing phase j`
4. `   System.out.println("(" + i + "," + j + ")");`
5. `   next;`
6. `}`
7. `}

- Outer forall-i loop has m iterations, 0...m-1
- Inner sequential j loop has i+1 iterations, 0...i
- Line 4 prints (task,phase) = (i, j) before performing a next operation.
- Iteration i = 0 of the forall-i loop prints (0, 0), performs a next, and then terminates. Iteration i = 1 of the forall-i loop prints (1,0), performs a next, prints (1,1), performs a next, and then terminates. And so on.
### Illustration of previous example

- Iteration $i=0$ of the `forall-i` loop prints $(0, 0)$ in Phase 0, performs a `next`, and then ends Phase 1 by terminating.

- Iteration $i=1$ of the `forall-i` loop prints $(1, 0)$ in Phase 0, performs a `next`, prints $(1,1)$ in Phase 1, performs a `next`, and then ends Phase 2 by terminating.

- And so on until iteration $i=8$ ends an empty Phase 8 by terminating.

<table>
<thead>
<tr>
<th>$i=0$</th>
<th>$i=1$</th>
<th>$i=2$</th>
<th>$i=3$</th>
<th>$i=4$</th>
<th>$i=5$</th>
<th>$i=6$</th>
<th>$i=7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(0,0)$</td>
<td>$(1,0)$</td>
<td>$(2,0)$</td>
<td>$(3,0)$</td>
<td>$(4,0)$</td>
<td>$(5,0)$</td>
<td>$(6,0)$</td>
<td>$(7,0)$</td>
</tr>
</tbody>
</table>

Phase 0

| $i=0...7$ are forall iterations |

| $(i,j) =$ println output |

| `next` = barrier operation |

| `end` = termination of a forall iteration |
Observation 3: Different forall iterations may perform “next” at different program points

1. `forall (point[i] : [0:m-1]) {`
2. `if (i % 2 == 1) { // i is odd`
3. `oddPhase0(i);`
4. `next;`
5. `oddPhase1(i);`
6. `} else { // i is even`
7. `evenPhase0(i);`
8. `next;`
9. `evenPhase1(i);`
10. `} // if-else`
11. `} // forall`

- Barrier operation synchronizes odd-numbered iterations at line 4 with even-numbered iterations in line 8
- next statement may even be in a method such as `oddPhase1()`
Recap: incorrect translation of PRAM Array sum algorithm to task-parallel program

1. `forall (point[i] : [0:n/2-1]) {`
2. `for (point[j] : [0:ceilLog2(n)-1]) {`
3. `int exp2j = 1<<j;`
4. `if (i % exp2j == 0 && 2*i+exp2j < n)`
6. `}` // for
7. `}` // forall
8. `static int ceilLog2(int n) { // returns 0 if n <= 0`
9. `int r=0; while (n > 1) { r++; n = n >> 1; } return r;`
10. `}`

Is there a data race in this program?
If so, why was the PRAM algorithm correct?
Correct translation of PRAM Array sum algorithm to HJ using for-forall structure

1. for (point[j] : [0:ceilLog2(n)-1]) {
2.   forall (point[i] : [0:n/2-1]) {
3.     int exp2j = 1<<j;
4.     if (i % exp2j == 0 && 2*i+exp2j < n)
6.   } // forall
7. } // for

- Moving the forall loop inside the for loop inserts implicit finish after each step (lines 3, 4, 5)
- Think of a PRAM program as sequential at the outer level, while executing each step as a forall loop across all processors
Correct translation of PRAM Array sum algorithm to HJ using forall-for-next

1. forall (point[i] : [0:n/2-1]) {
2. for (point[j] : [0:ceilLog2(n)-1]) {
3. int exp2j = 1<<j;
4. if (i \% exp2j == 0 && 2*i+exp2j < n)
6. next; // barrier ensures lock-step semantics
7. } // for
8. } // forall

- You can also think of a PRAM program as parallel at the outer level with a barrier (next) operation at each step to synchronize all processors
Next-with-Single Statement

next <single-stmt> is a barrier in which single-stmt is performed exactly once after all tasks have completed the previous phase and before any task begins its next phase.

Modeling next-with-single in the Computation Graph

- **next-start**
- **single-statement**
- **next-end**

**Signal edges**
- From A1, A2, A3, A4 to next-start

**Wait edges**
- From A1, A2, A3, A4 to next-end
Use of next-with-single to print a log message between Hello and Goodbye phases (Listing 6)

1. rank.count = 0; // rank object contains an int field, count
2. forall (point[i] : [0:m-1]) {
3.   // Start of Hello phase
4.   int r;
5.   isolated {r = rank.count++;}
6.   System.out.println("Hello from task ranked " + r);
7.   next { // single statement
8.     System.out.println("LOG: Between Hello & Goodbye Phases");
9.   }
10. } // Start of Goodbye phase
11. System.out.println("Goodbye from task ranked " + r);
12.} // forall
One-Dimensional Iterative Averaging Example

- Initialize a one-dimensional array of \((n+2)\) double's with boundary conditions, \(myVal[0] = 0\) and \(myVal[n+1] = 1\).

- In each iteration, each interior element \(myVal[i]\) in \(1..n\) is replaced by the average of its left and right neighbors.
  - Two separate arrays are used in each iteration, one for old values and the other for the new values.

- After a sufficient number of iterations, we expect each element of the array to converge to \(myVal[i] = i/(n+1)\)
  - In this case, \(myVal[i] = (myVal[i-1] + myVal[i+1])/2\), for all \(i\) in \(1..n\)

Illustration of an intermediate step for \(n = 8\) (source: Figure 6.19 in Lin-Snyder book)

```
<table>
<thead>
<tr>
<th></th>
<th>0.00</th>
<th>0.34</th>
<th>0.21</th>
<th>0.86</th>
<th>0.65</th>
<th>0.11</th>
<th>0.43</th>
<th>0.97</th>
<th>0.51</th>
<th>1.00</th>
</tr>
</thead>
</table>
```

- Boundary value
- Interior values
- Boundary value
HJ code for One-Dimensional Iterative Averaging using nested for-forall structure (Listing 8)

1. double[] myVal = new double[n]; myVal[0] = 0; myVal[n+1] = 1;
2. for (point [iter] : [0:iterations-1]) {
3.   // Output array MyNew is computed as function of  
4.   // input array MyVal from previous iteration  
5.   double[] myNew = new double[n]; myNew[0] = 0; myNew[n+1] = 1;
6.   forall (point [j] : [1:n]) { // Create n tasks  
7.     myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;  
8.   } // forall  
9.   myVal = myNew; // myNew becomes input array for next iteration
10. } // for

• How many tasks does this version create?
• This is an idealized version with no batching of forall iterations and a new array allocation in each iteration of the for-iter loop
HJ code for One-Dimensional Iterative Averaging using nested forall-for-all-next structure (Listing 9)

1. // Assume that myVal and myNew are mutable fields of type double[]
2. myNew = new double[n]; myNew[0] = 0; myNew[n+1] = 1;
3. forall (point [j] : [1:n]) { // Create n tasks
4.   for (point [iter] : [0:iterations-1]) {
5.     next { // single statement
6.       myVal = myNew; // myNew becomes input array for next iteration
7.       myNew = new double[n]; myNew[0] = 0; myNew[n+1] = 1;
8.     }
9.   myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
10. } // for
11.} // forall

• How many tasks does this version create?
• This version uses next-with-single to synchronize array allocation in each iteration of the for-iter loop