COMP 322: Fundamentals of Parallel Programming

Lecture 13: Forall Statements & Barriers (contd)

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

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
AtomicInteger rank = new AtomicInteger();
forall (point[i] : [0:m-1]) {
    int r = rank.getAndIncrement();
    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
  - If a forall iteration terminates before executing “next”, then the other iterations do not wait for it
  - Scope of synchronization is the closest enclosing forall statement
  - Special case of “phaser” construct (will be covered in following lectures)
Recap of Observation 2 (Lecture 12): 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 Observation 2

- 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.

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

$(i,j) = \text{println output}$

next = barrier operation

end = termination of a forall iteration
Recap of Observation 3 (Lecture 12): 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()
One-Dimensional Iterative Averaging Example (Lecture 10)

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

- In each iteration, each interior element \(\text{myVal}[i]\) 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 \(\text{myVal}[i] = i/(n+1)\)
  - In this case, \(\text{myVal}[i] = (\text{myVal}[i-1] + \text{myVal}[i+1])/2\), for all \(i\) in \(1..n\)

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

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.34</td>
<td>0.21</td>
<td>0.86</td>
<td>0.65</td>
<td>0.11</td>
<td>0.43</td>
</tr>
</tbody>
</table>

- Boundary value
- Interior values
- Boundary value
HJ code for One-Dimensional Iterative Averaging with nested for-forall structure

1. double[] myVal=new double[n+2]; double[] myNew=new double[n+2];
2. myVal[n+1] = 1; // Boundary condition
3. for (point [iter] : [0:numIters-1]) {
4.   // Compute MyNew as function of input array MyVal
5.     forall (point [j] : [1:n]) { // Create n tasks
6.       myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
7.     } // forall
8.   // Swap myVal and myNew
9.     double[] temp=myVal; myVal=myNew; myNew=temp;
10.  // myNew becomes input array for next iteration
11.} // for

• Replace “finish async” from Lecture 10 by “forall”
• Overhead issue --- this version creates (numIters * n) async tasks
HJ code for One-Dimensional Iterative Averaging with chunked for-forall-for structure

1. double[] myVal=new double[n+2]; double[] myNew=new double[n+2];
2. myVal[n+1] = 1; // Boundary condition
3. // Set desired number of chunks for j loop to total number of workers
4. int Cj = Runtime.getNumOfWorkers();
5. for (point [iter]: [0:numIters-1]) {
6.   // Compute MyNew as function of input array MyVal
7.   forall (point [jj]:[0:Cj-1]) // Iterate over chunks
8.     for (point [j]:getChunk([1:n],[Cj],[jj])) // Iterate within chunk
9.       myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
10.  // Swap myVal and myNew
11.   double[] temp=myVal; myVal=myNew; myNew=temp;
12.  // myNew becomes input array for next iteration
13. } // for iter

- Chunked forall version creates numIters*Cj async tasks
- Can we do better with barriers?
One-Dimensional Iterative Averaging with Barrier Synchronization

1. double[] gVal = new double[n+2]; double[] gNew = new double[n+2]; gVal[n+1] = 1;
2. int Cj = Runtime.getNumOfWorkers();
3. forall (point [jj]:[0:Cj-1]) { // Chunked forall is now the outermost loop
4.   double[] myVal = gVal; double[] myNew = gNew; // Local copy of myVal/myNew pointers
5.   for (point [iter] : [0:numIters-1]) {
6.     // Compute MyNew as function of input array MyVal
7.     for (point [j]:getChunk([1:n],[Cj],[jj])) // Iterate within chunk
8.       myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
9.   } // Barrier before executing next iteration of iter loop
10.  // Swap myVal and myNew (each forall iterations swaps its pointers in local vars)
11.   double[] temp=myVal; myVal=myNew; myNew=temp;
12.  // myNew becomes input array for next iter
13. } // forall
14.} // forall

- Use of barrier reduces number of async tasks created to just Cj
- However, these Cj tasks perform Cj*numIters barrier operations
  - Good trade-off since, barrier operations have lower overhead than task creation if number of chunks <= number of workers
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

A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow A_4

signal edges

A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow A_4

wait edges
Use of next-with-single to print a log message between Hello and Goodbye phases

1. AtomicInteger rank = new AtomicInteger();
2. forall (point[i] : [0:m-1]) {
3.     // Start of Hello phase
4.     int r = rank.getAndIncrement();
5.     System.out.println("Hello from task ranked " + r);
6.     next single {
7.         System.out.println("LOG: Between Hello & Goodbye Phases");
8.     }
9.     // Start of Goodbye phase
10.    System.out.println("Goodbye from task ranked " + r);
11.} // forall