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

HJ's forall statement = finish + forasync + barriers (next)

```
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);
}
```

Phase 0

Phase 1

- 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     i=1     i=2     i=3     i=4     i=5     i=6     i=7
 |       |       |       |       |       |       |       |
(0,0)   (1,0)   (2,0)   (3,0)   (4,0)   (5,0)   (6,0)   (7,0)          Phase 0
 |       |       |       |       |       |       |       |
next ----- next ----- next ----- next ----- next ----- next -----next ----- next
 |       |       |       |       |       |       |       |
 |      (1,1)   (2,1)   (3,1)   (4,1)   (5,1)   (6,1)   (7,1)          Phase 1
 |       |       |       |       |       |       |       |
end ----- next ----- next ----- next ----- next ----- next -----next ----- next
         |       |       |       |       |       |       |
         |      (2,2)   (3,2)   (4,2)   (5,2)   (6,2)   (7,2)          Phase 2
         |       |       |       |       |       |       |
        end ---- next ----- next ----- next ----- next ----- next -----next
                 |       |       |       |       |       |
                 |      (3,3)   (4,3)   (5,3)   (6,3)   (7,3)          Phase 3
                 |       |       |       |       |       |
                end ----- next ----- next ----- next ----- next ----- next
                         |       |       |       |       |
                         |      (4,4)   (5,4)   (6,4)   (7,4)          Phase 4
                         |       |       |       |       |
                        end ----- next ----- next ----- next ----- next
                                 |       |       |       |
                                 |      (5,5)   (6,5)   (7,5)          Phase 5
                                 |       |       |       |
                                end ----- next ----- next ----- next
                                         |       |       |
                                         |      (6,6)   (7,6)          Phase 6
                                         |       |       |
                                        end ----- next ----- next
                                                 |       |
                                                 |      (7,7)          Phase 7
                                                 |       |
                                                end ----- next
                                                         |
                                                        end            Phase 8
```

i=0…7 are forall iterations

(i,j) = 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

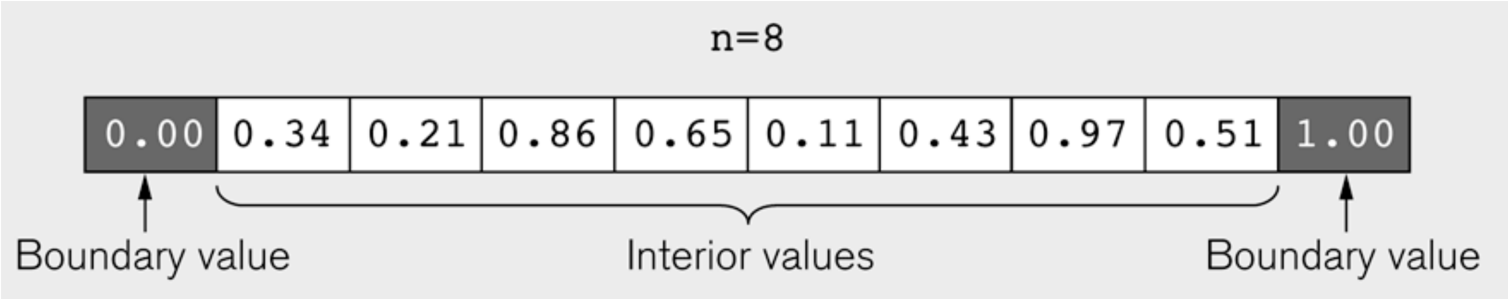
```
1.  forall (point[i] : [0:m-1]) {
2.    if (i % 2 == 1) { // i is odd
3.      oddPhase0(i);
4.      next;
5.      oddPhase1(i);
6.    } else { // i is even
7.      evenPhase0(i);
8.      next;
9.      evenPhase1(i);
10.   } // if-else
11. } // forall
```

- 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, 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

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



HJ code for One-Dimensional Iterative Averaging with nested for-forall structure

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

- Replace “finish async” from Lecture 10 by “forall”
- Overhead issue --- this version creates (numIters * n) async tasks

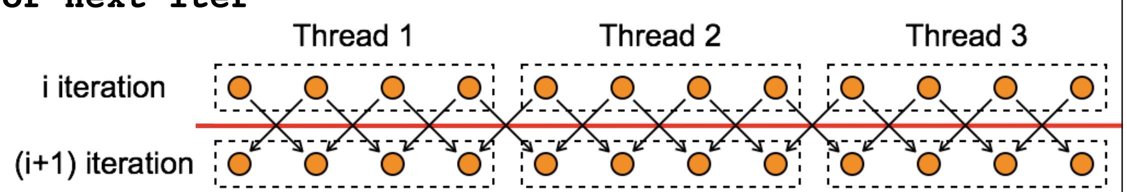
HJ code for One-Dimensional Iterative Averaging with chunked for-forall-for structure

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

- Chunked forall version creates numIters*Cj async tasks
- Can we do better with barriers?

One-Dimensional Iterative Averaging with Barrier Synchronization

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



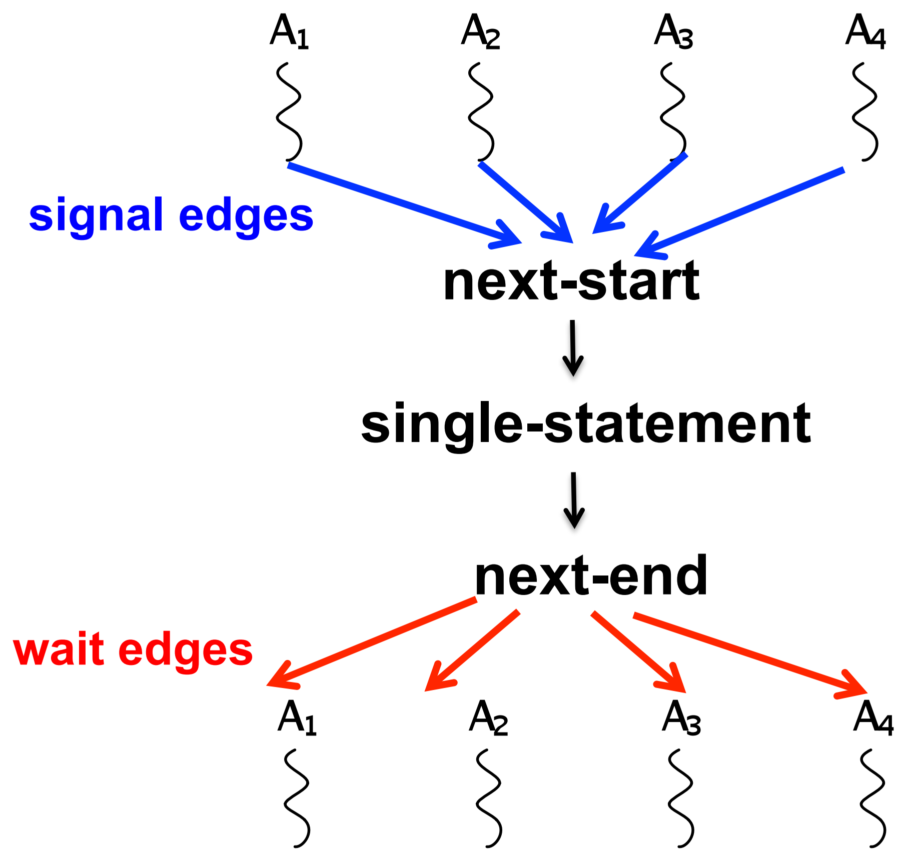
(b) Barrier synchronization

- 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



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

```
AtomicInteger rank = new AtomicInteger();
forall (point[i] : [0:m-1]) {
  // Start of Hello phase
  int r = rank.getAndIncrement();
  System.out.println("Hello from task ranked " + r);
  next single {
  System.out.println("LOG: Between Hello & Goodbye Phases");
  }
  // Start of Goodbye phase
   System.out.println("Goodbye from task ranked " + r);
} // forall
```