COMP 322: Fundamentals of Parallel Programming

Lecture 12: Barrier synchronization in forall loops

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

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Solution to Worksheet #11: One-dimensional Iterative Averaging Example

1) Assuming n=9 and the input array below, perform one iteration of the iterative averaging example by only filling in the blanks for odd values of j in the myNew[] array. Recall that the computation is "myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;"

index, j	0	1	2	3	4	5	6	7	8	9	10
myVal	0	0	0.2	0	0.4	0	0.6	0	0.8	0	1
myNew	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1

2) Will the contents of myVal[] and myNew[] change in further iterations, after myNew above in 1) becomes myVal[] in the next iteration?

No, this represents the converged value (equilibrium/fixpoint).

HJ code for One-Dimensional Iterative Averaging using nested for-finish-forasync structure (Recap)

```
for (point [iter] : [0:m-1]) {
  // Compute MyNew as function of input array MyVal
  finish forasync (point [j] : [1:n]) { // Create n tasks
     myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
  } // finish forasync
  temp=myVal; myVal=myNew; myNew=temp;// Swap myVal & myNew;
  // myNew becomes input array for next iteration
} // for
```

Question: How many async tasks does this program create as a function of m and n?

Answer: m*n. Can we do better with chunking?

Example: HJ code for One-Dimensional Iterative Averaging with chunked for-finish-forasync-for structure (Recap)

```
int nc = Runtime.getNumOfWorkers();
for (point [iter] : [0:m-1]) {
  // Compute MyNew as function of input array MyVal
  finish forasync (point [jj] : [0:nc-1]) {
    for(point [j] : getChunk([1:n],nc,jj)) {
      myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
  } // finish forasync
  temp=myVal; myVal=myNew; myNew=temp;// Swap myVal & myNew;
  // myNew becomes input array for next iteration
} // for
```

Question: How many async tasks does this program create as a function of m, n, and nc?

Answer: m*nc. But we can do even better with “forall” loops and “barrier” synchronization.

Outline of Today's Lecture

- **<u>Barrier Synchronization in Forall Loops</u>**

Acknowledgments

- COMP 322 Module 1 handout, Sections 10.1, 10.2, 10.4.

HJ’s forall statement = finish + forasync + barriers

Goal 1 (minor): replace common finish-forasync idiom by forall e.g., replace

```
finish forasync (point [I,J] : [0:N-1,0:N-1])
  for (point[K] : [0:N-1])
    C[I][J] += A[I][K] * B[K][J];
```

by

```
forall (point [I,J] : [0:N-1,0:N-1])
  for (point[K] : [0:N-1])
    C[I][J] += A[I][K] * B[K][J];
```

Goal 2 (major): Also support “barrier” synchronization

- Caveat: forall is only supported on the work-sharing runtime because of barrier synchronization

Hello-Goodbye Forall Example (Listing 33)

```
forall (point[i] : [0:m-1]) {
  int sq = i*i;
  System.out.println(“Hello from task with square = “ + sq);
  System.out.println(“Goodbye from task with square = “ + sq);
}
```

- **Sample output for m = 4**

```
Hello from task with square = 0
Hello from task with square = 1
Goodbye from task with square = 0
Hello from task with square = 4
Goodbye from task with square = 4
Goodbye from task with square = 1
Hello from task with square = 9
Goodbye from task with square = 9
```

Hello-Goodbye Forall Example (contd)

```
forall (point[i] : [0:m-1]) {
  int sq = i*i;
  System.out.println(“Hello from task with square = “ + sq);
  System.out.println(“Goodbye from task with square = “ + sq);
}
```

- Question: how can we transform this code so as to ensure that all tasks say hello before *any* tasks say goodbye?
- Statements in red below will need to be moved to solve this problem

```
Hello from task with square = 0
Hello from task with square = 1
Goodbye from task with square = 0
Hello from task with square = 4
Goodbye from task with square = 4
Goodbye from task with square = 1
Hello from task with square = 9
Goodbye from task with square = 9
```

Hello-Goodbye Forall Example (contd)

```
forall (point[i] : [0:m-1]) {
 int sq = i*i;
 System.out.println(“Hello from task with square = “ + sq);
 System.out.println(“Goodbye from task with square = “ + sq);
}
```

- **Question: how can we transform this code so as to ensure that all tasks say hello before any tasks say goodbye?**
- **Approach 1: Replace the forall loop by two forall loops, one for the hello’s and one for the goodbye’s**
  - **—Problem: Need to communicate local sq values from one forall to the next**

```
// APPROACH 1
forall (point[i] : [0:m-1]) {
 int sq = i*i;
 System.out.println(“Hello from task with square = “ + sq);
}
forall (point[i] : [0:m-1]) {
 System.out.println(“Goodbye from task with square = “ + sq);
}
```

Hello-Goodbye Forall Example (contd)

- Question: how can we transform this code so as to ensure that all tasks say hello before any tasks say goodbye?
- Approach 2: insert a “barrier” between the hello’s and goodbye’s
    - —“next” statement in HJ’s forall loops

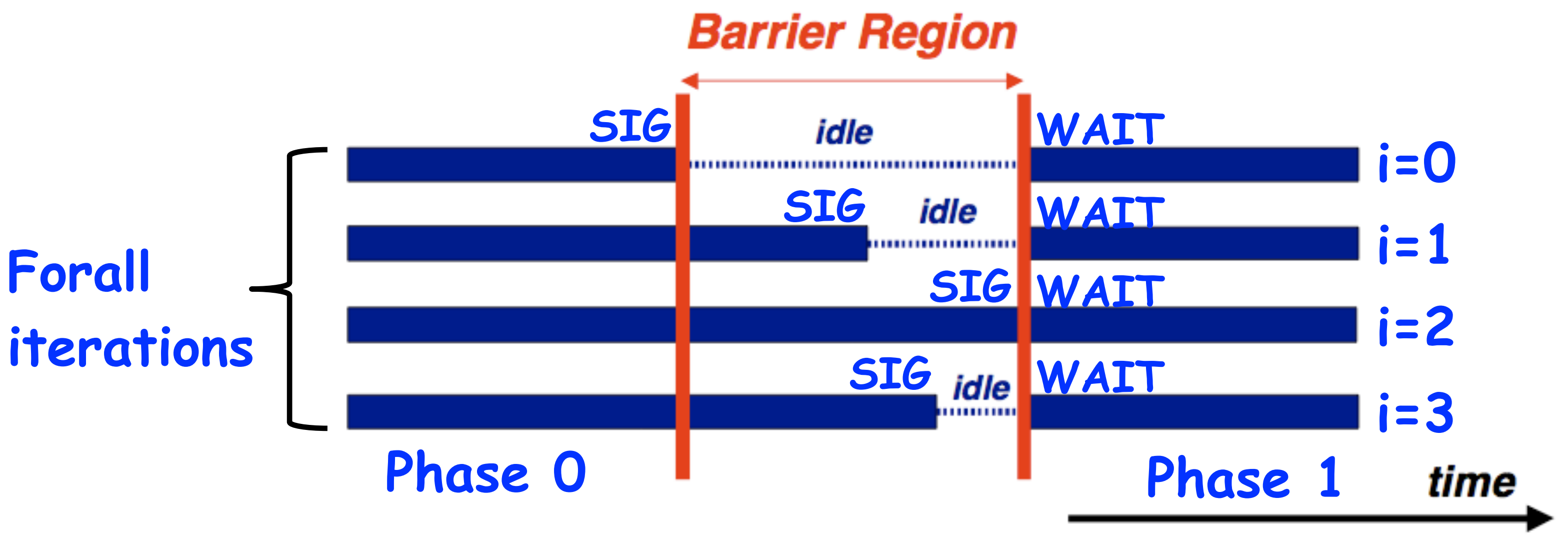
```
// APPROACH 2
forall (point[i] : [0:m-1]) {
 int sq = i*i;
 System.out.println(“Hello from task with square = “ + sq);
 next; // Barrier
 System.out.println(“Goodbye from task with square = “ + sq);
}
```

Phase 0

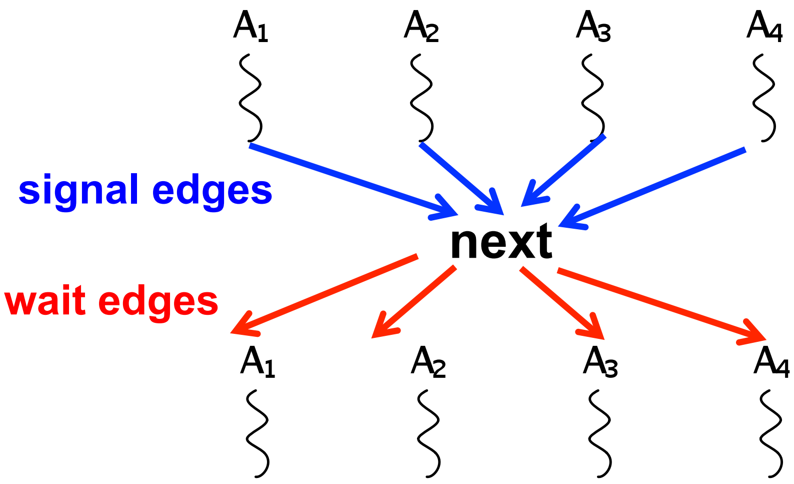
Phase 1

- 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 next is the closest enclosing forall statement
    - —Special case of “phaser” construct (will be covered later in class)

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

```
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

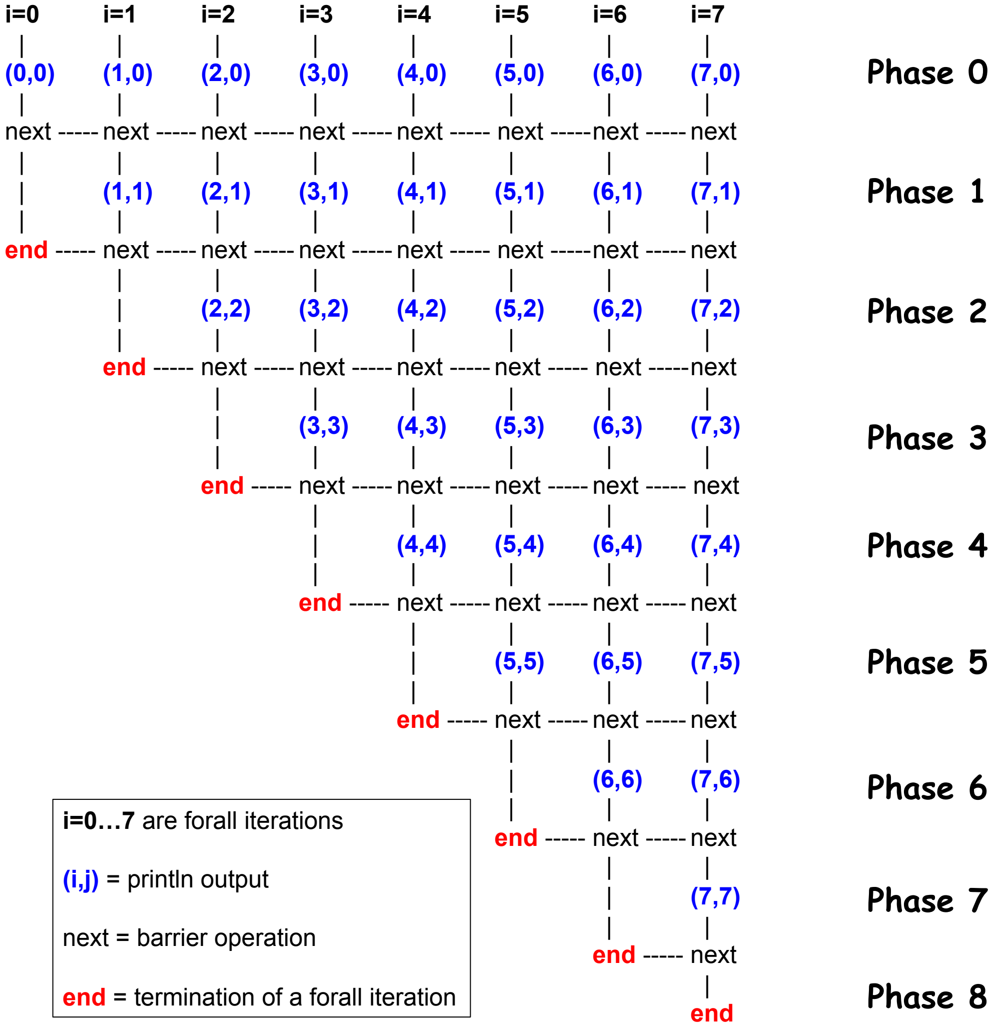
```
1.  forall (point[i] : [0:m-1]) {
2.    for (point[j] : [0:i]) {
3.      // Forall iteration i is executing phase j
4.      System.out.println("(" + i + "," + j + ")");
5.      next;
6.    }
7.  }
```

- 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

Interesting figure. Try out another one in Worksheet 12!



```
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

Observation 3: Different forall iterations may perform “next” at different program points (barrier matching problem)

```
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 with Barrier Synchronization

```
double[] gVal=new double[n+2]; double[] gNew=new double[n+2]; gVal[n+1] = 1;
int nc = Runtime.getNumWorkers();
forall (point [jj]:[0:nc-1]) { // Chunked forall is now the outermost loop
  double[] myVal = gVal; double[] myNew = gNew; // Local copy of myVal/myNew pointers
  for (point [iter] : [0:m-1]) {
    // Compute MyNew as function of input array MyVal
    for (point [j]:getChunk([1:n],nc,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
```

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

Worksheet #12: Forall Loops and Barriers

Name 1: ______________________ Name 2: ______________________

1) Draw a “barrier matching” figure similar to slide 14 for the code fragment below.

```
String[] a = { “ab”, “cde”, “f” };
. . . int m = a.length; . . .
forall (point[i] : [0:m-1]) {
   for (int j = 0; j < a[i].length(); j++) {
     // forall iteration i is executing phase j
     System.out.println("(" + i + "," + j + ")");
     next;
   }
 }
```