COMP 322: Fundamentals of Parallel Programming

Lecture 12: Barrier Synchronization

Zoran Budimlić and Mack Joyner
{zoran, mjoyner}@rice.edu

http://comp322.rice.edu

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

1) Assuming n=9 and the input array below, perform a "half-iteration" of the iterative averaging example by only filling in the blanks for odd values of j in the myNew[] array (different from the real algorithm). 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?

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

3) Write the formula for the final value of myNew[i] as a function of i and n. In general, this is the value that we will get if m (= #iterations in sequential for-iter loop) is large enough.

After a sufficiently large number of iterations, the iterated averaging code will converge with myNew[i] = myVal[i] = i / (n+1)

Hello-Goodbye Forall Example (Pseudocode)

```
forall (0, m - 1, (i) -> {
  int sq = i*i;  // NOTE: video used lookup(i) instead
  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 (0, m - 1, (i) -> {
  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 (0, m - 1, (i) -> {
  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 first forall to the second

```
// APPROACH 1
forall (0, m - 1, (i) -> {
 int sq = i*i;
 System.out.println("Hello from task with square = " + sq);
});
forall (0, m - 1, (i) -> {
 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, without having to change local ?
- Approach 2: insert a “barrier” (“next” statement) between the hello’s and goodbye’s

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

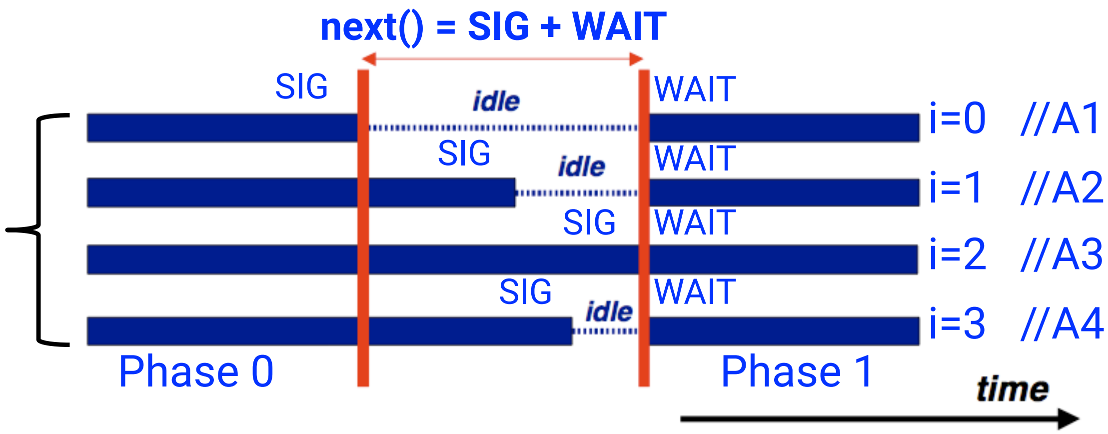
Phase 0 (lines `int sq = i*i;` and the Hello println)

Phase 1 (the Goodbye println)

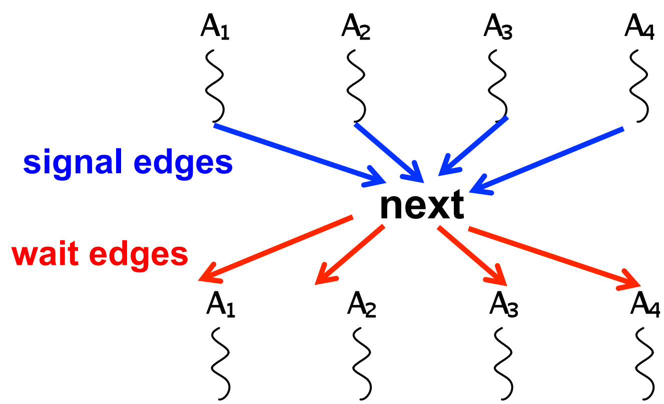
- next -> each forallPhased iteration waits at barrier until all iterations arrive (previous phase is completed), after which the next phase can start
  - Scope of next is the closest enclosing forallPhased statement
  - If a forallPhased iteration terminates before executing “next”, then the other iterations don’t wait for it

Impact of barrier on scheduling forallPhased iterations

Four forallPhased iterations, each with a next() barrier



next() operation is modeled in the Computation Graph using *signal* and *wait* edges



forallPhased API's in HJlib

http://www.cs.rice.edu/~vs3/hjlib/doc/edu/rice/hj/Module1.html

- `static void forallPhased(int s0, int e0, edu.rice.hj.api.HjProcedure<java.lang.Integer> body)`
- `static <T> void forallPhased(java.lang.Iterable<T> iterable, edu.rice.hj.api.HjProcedure<T> body)`
- `static void next()`

- NOTE:
  - All forallPhased API's include an implicit finish at the end (just like a regular forall)
  - Calls to next() are only permitted in forallPhased(), not in forall()

Observation 1: Scope of synchronization for “next” barrier is its closest enclosing forallPhased statement

```
forallPhased (0, m - 1, (i) -> {
 println(“Starting forall iteration ” + i);
 next(); // Acts as barrier for forallPhased-i
 forallPhased (0, n - 1, (j) -> {
   println(“Hello from task (“ + i + “,” + j + “)”);
   next(); // Acts as barrier for forallPhased-j
   println(“Goodbye from task (“ + i + “,” + j + “)”);
 } // forallPhased-j
 next(); // Acts as barrier for forallPhased-i
 println(“Ending forallPhased iteration ” + i);
}); // forallPhased-i
```

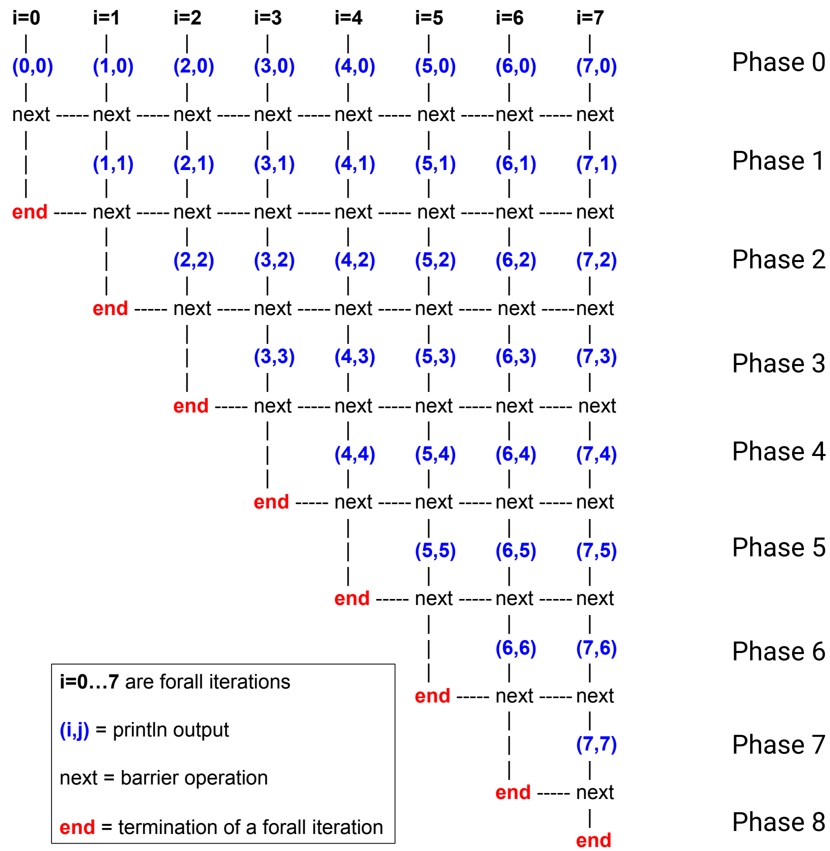
Observation 2: If a forall iteration terminates before “next”, then other iterations do not wait for it

```
1. forallPhased (0, m - 1, (i) -> {
2.   forseq (0, i, (j) -> {
3.     // forall iteration i is executing phase j
4.     System.out.println("(" + i + "," + j + ")");
5.     next();
6.   }); //forseq-j
7. }); //forall-i
```

- 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.

Barrier Matching for previous example

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



Observation 3: Different forallPhased iterations may perform "next" at different program points

```
1.  forallPhased (0, m-1, (i) -> {
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
```

Barriers are not statically scoped — matching barriers may come from different program points, and may even be in different methods!

- Barrier operation synchronizes odd-numbered iterations at line 4 with even-numbered iterations in line 8
- One reason why barriers are "less structured" than finish, async, future

Announcements & Reminders

- HW2 is available and due by 11:59pm on Wednesday
- Quiz for Unit 2 (topics 2.1 - 2.6) is available on Canvas, and due by 11:59pm on Monday
- No class on Friday (spring recess)
- See course web site for all work assignments and due dates
- Use Piazza (public or private posts, as appropriate) for all communications re. COMP 322
- See Office Hours link on course web site for latest office hours schedule.

Worksheet #12: Forall Loops and Barriers

Name: ______________________　　**Net ID: ______________**

Draw a “barrier matching” figure similar to slide 11 for the code fragment below.

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

BACKUP SLIDES START HERE

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

```
forseq (0, m - 1, (iter) -> {
  // Compute MyNew as function of input array MyVal
  finish(() -> {
    forasync (1, n, (j) -> { // Create n tasks
      myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
    }); // forasync
  }) // finish
  temp=
  // myN
}); // 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 = numWorkerThreads();
forseq (0, m - 1, (iter) -> {
  // Compute MyNew as function of input array MyVal
  finish (() -> {
   forasync (0, nc - 1, (jj) -> {
     HjRegion1D iterSpace = newRectangularRegion1D(1, n);
     forseq (getChunk(iterSpace, nc, jj), (j) -> {
       myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
     });
   }); // forasync
  }); // finis
  temp=my
  // myNew
}); // 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.

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

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();
forallPhased (0, nc - 1, (jj) -> { // Chunked forall is now the outermost loop
  double[] myVal = gVal; double[] myNew = gNew; // Local copy of myVal/myNew pointers
  forseq (0, m - 1, (iter) -> {
    // Compute MyNew as function of input array MyVal
    forseq (getChunk([1:n],nc,jj), (j) -> { // 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