COMP 322: Fundamentals of Parallel Programming

Lecture 11: Iteration Grouping, Barrier Synchronization

Mack Joyner
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)

Announcements & Reminders

- Quiz for Unit 2 (topics 2.1 - 2.8) is available on Canvas, due by 11:59pm on Monday, Feb. 10th
- Midterm Exam on Thursday, Feb. 27th at TBD

HJ code for One-Dimensional Iterative Averaging

```
1.// Intialize m, n, myVal, newVal
2.m = … ; n = … ;
3.float[] myVal = new float[n+2];
4.float[] myNew = new float[n+2];
5.forseq(0, m-1, (iter) -> {
6.  // Compute MyNew as function of input array MyVal
7.    forall(1, n, (j) -> { // Create n tasks
8.        myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
9.    }); // forall
10.  // What is the purpose of line 11 below?
11.  float[] temp=myVal; myVal=myNew; myNew=temp;
12.}); // forseq
```

What about Overheads?

- It is inefficient to create `forall` iterations in which each iteration (`async` task) does very little work
- An alternate approach is “iteration grouping” or “loop chunking”
  - —e.g., replace

    ```
    forall(0, 99, (i) -> BODY(i)); // 100 tasks
    ```

  - —by

    ```
      forall(0, 3, (ii) -> { // 4 tasks
      // Each task executes a “chunk” of 25 iterations
        forseq(25*ii, 25*(ii+1)-1, (i) -> BODY(i));

      }); // forall
    ```

  - —This is better, but it’s still inconvenient for the programmer to do the “iteration grouping” or “loop chunking” explicitly

forallChunked APIs

- forallChunked(int s0, int e0, int chunkSize, edu.rice.hj.api.HjProcedure<Integer> body)
- Like forall(int s0, int e0, edu.rice.hj.api.HjProcedure<Integer> body)
- but `forallChunked` includes chunkSize as the third parameter!
  - e.g., replace

    forall(0, 99, (i) -> BODY(i)); // 100 tasks
  - by

    forallChunked(0, 99, 100/4, (i)->BODY(i));

```
int nc = numWorkerThreads();
... // Initializations
forseq(0, m-1, (iter) -> {
  // Compute MyNew as function of input array MyVal
    forallChunked(1, n, n/nc, (j) -> { // Create n/nc tasks
      myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
    }); // forallChunked
  // Swap myVal & myNew;
  float[] temp=myVal; myVal=myNew; myNew=temp;
  // myNew becomes input array for next iteration
}); // forseq
```

Barrier Synchronization: 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*
  - What’s the problem here?

```
// 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);
});
```

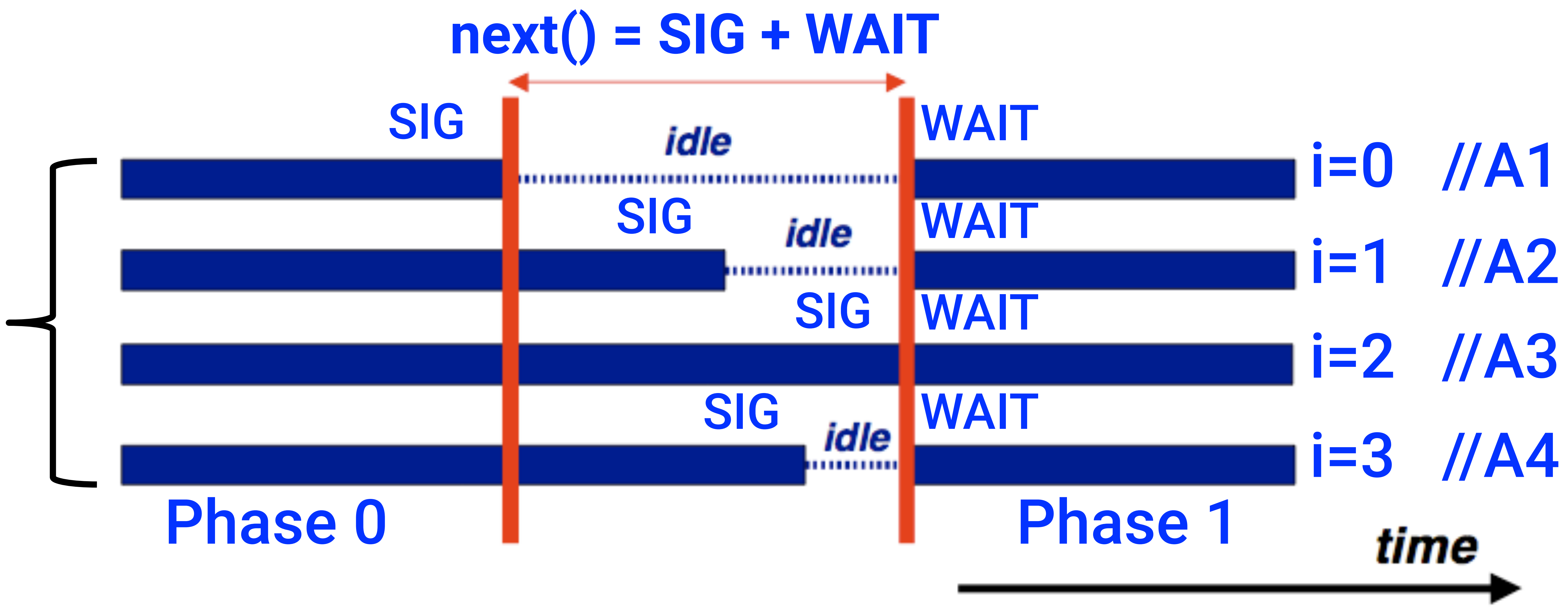
Phase 0

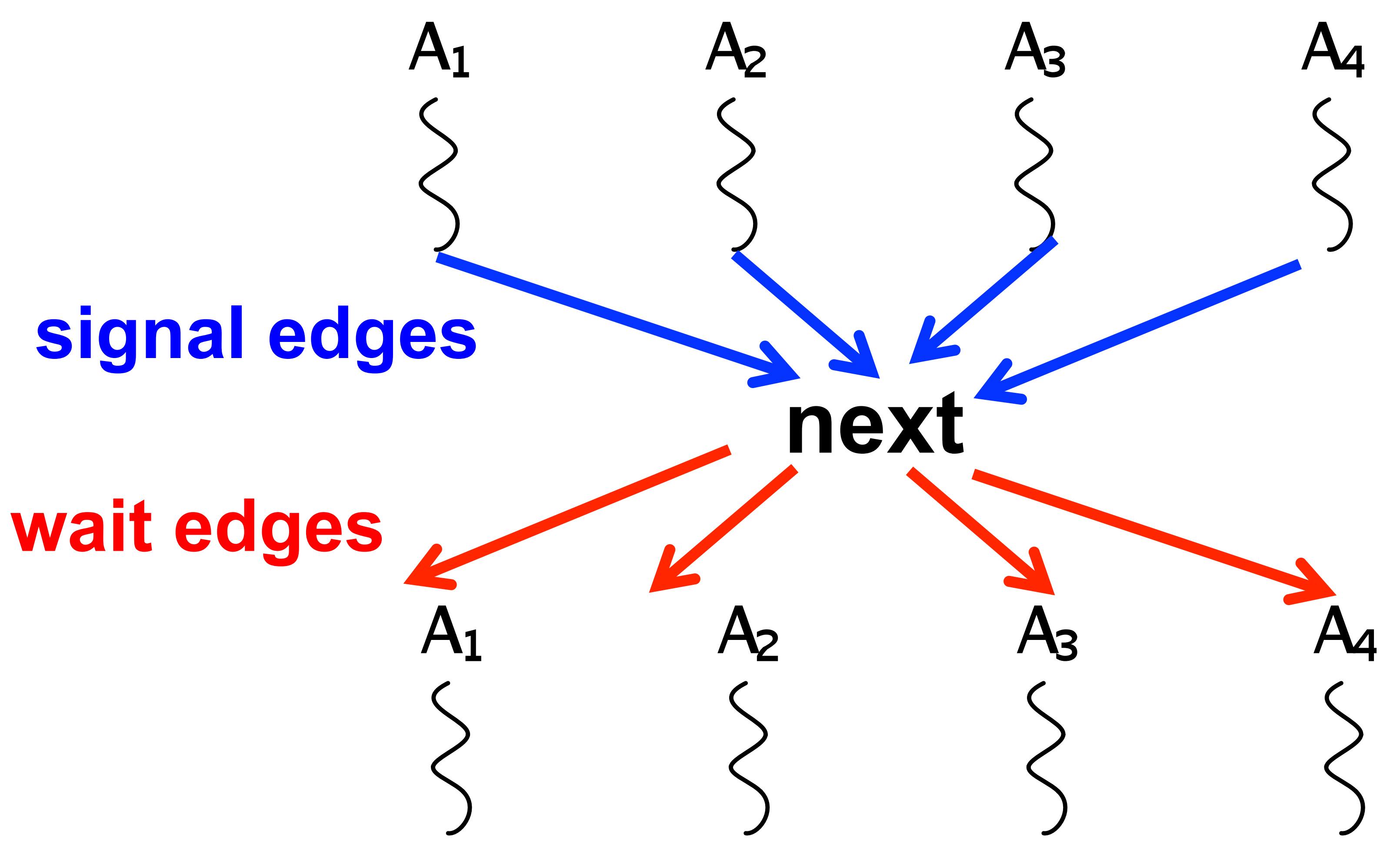
Phase 1

- 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
```

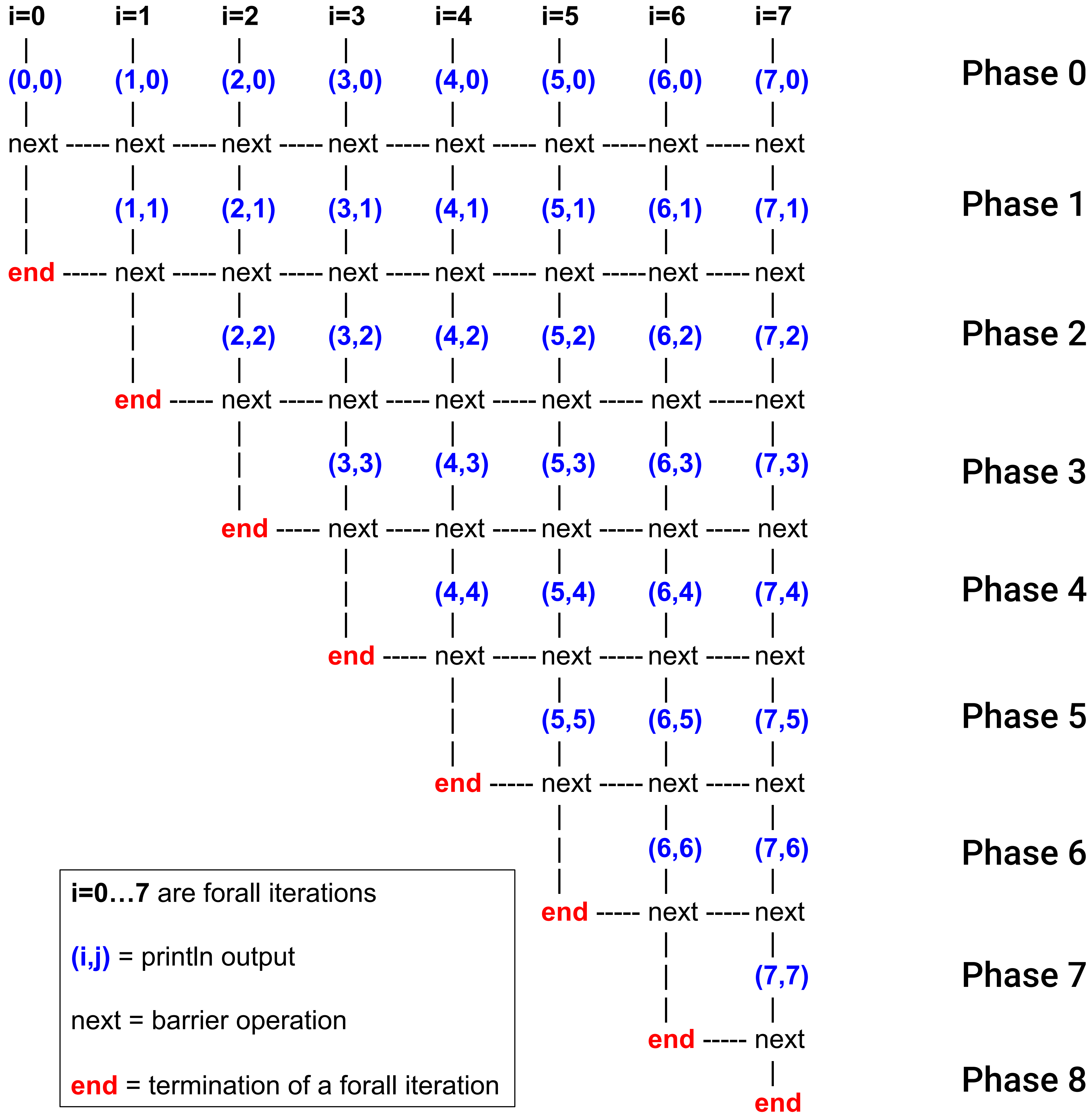
Observation 2: If a forall iteration terminates before “next”, then other iterations do not wait for it

```
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



```
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 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

Parallelizing loops in Matrix Multiplication example using forall

```
// Parallel version using forall
forall(0, n-1, 0, n-1, (i, j) -> {
    c[i][j] = 0;
  });
forall(0, n-1, 0, n-1, (i, j) -> {
   forseq(0, n-1, (k) -> {
       c[i][j] += a[i][k] * b[k][j];
    });
 });
// Print first element of output matrix
println(c[0][0]);
```

$$c[i,j] = \sum_{0 \leq k < n} a[i,k] * b[k,j]$$