

---

# COMP 322: Fundamentals of Parallel Programming

## Lecture 11: Barrier Synchronization

**Vivek Sarkar, Eric Allen**  
**Department of Computer Science, Rice University**

**Contact email: [vsarkar@rice.edu](mailto:vsarkar@rice.edu)**

**<https://wiki.rice.edu/confluence/display/PARPROG/COMP322>**



# Solution to Worksheet #10: 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 “ $\text{myNew}[j] = (\text{myVal}[j-1] + \text{myVal}[j+1])/2.0;$ ”

| 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) If  $m$  is large enough, write the formula for the final value of `myNew[i]` as a function of  $i$  and  $n$ .

**After a large number of iterations, the iterated averaging code will converge with  $\text{myNew}[i] = \text{myVal}[i] = i / (n+1)$**



# Hello-Goodbye Forall Example (Pseudocode)

---

```
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);  
});
```

- **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)

```
1. forall (0, m - 1, (i) -> {
2.   int sq = i*i;
3.   System.out.println("Hello from task with square = " + sq);
4.   System.out.println("Goodbye from task with square = " + sq);
5. });
```

- **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

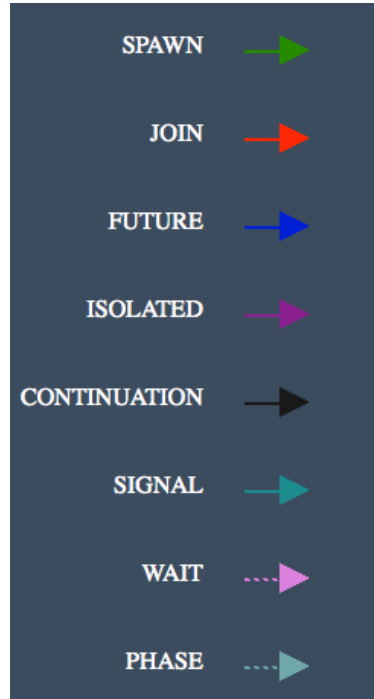
—**Problem:** Need to communicate local sq values from first forall to the second

```
1. // APPROACH 1
2. forall (0, m - 1, (i) -> {
3.   int sq = i*i;
4.   System.out.println("Hello from task with square = " + sq);
5. });
6. forall (0, m - 1, (i) -> {
7.   System.out.println("Goodbye from task with square = " + sq);
8. });
```

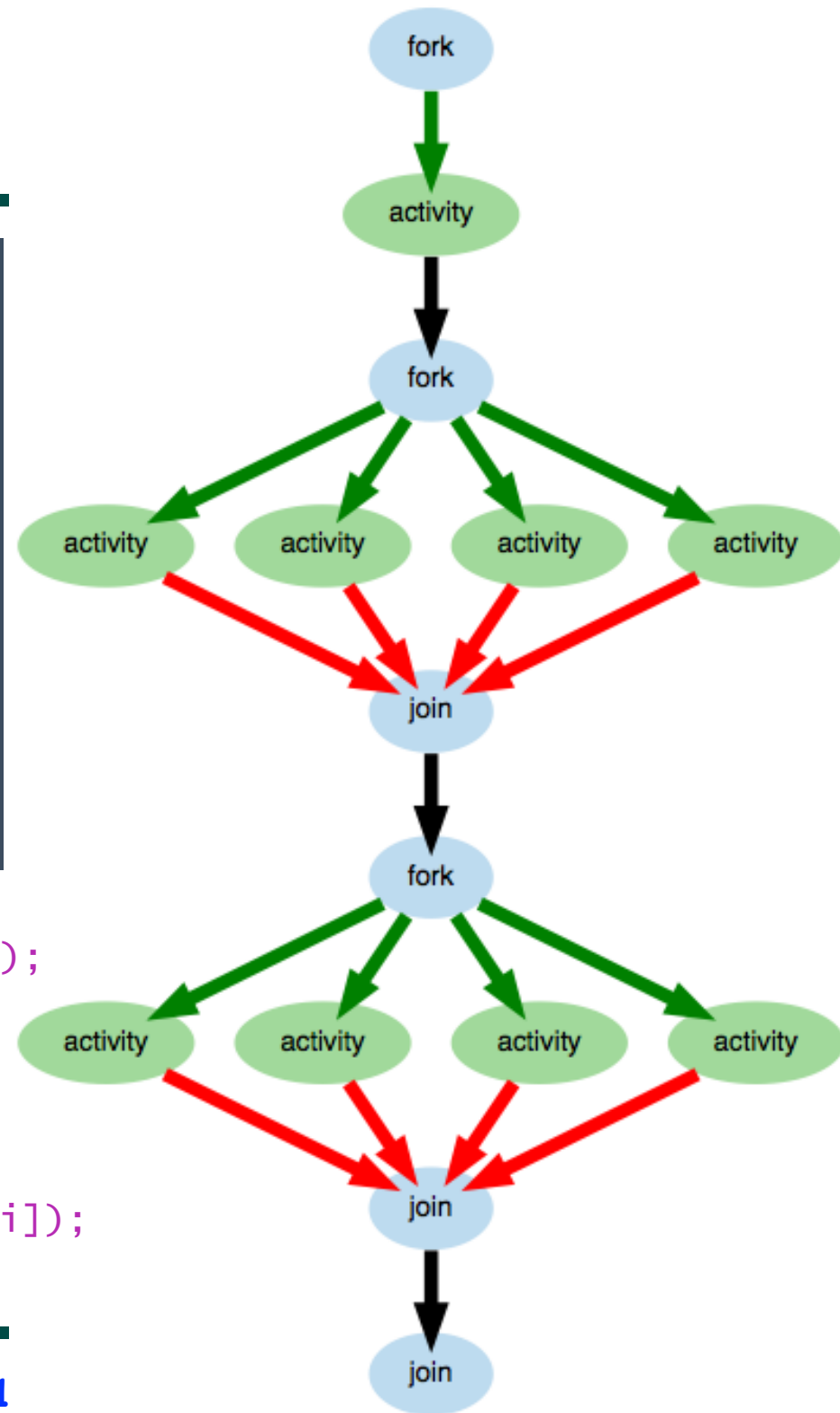


# Computation Graph for Approach 1 (m=4)

Images courtesy of HJ-Viz tool!



```
1. // APPROACH 1
2. forall (0, m - 1, (i) -> {
3.   int sq[i] = i*i;
4.   system.out.println(
5.     "Hello from task with square = " + sq[i]);
6. });
7. forall (0, m - 1, (i) -> {
8.   system.out.println(
9.     "Goodbye from task with square = " + sq[i]);
10. });
```



# 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

```
1. // APPROACH 2
2. forallPhased (0, m - 1, (i) -> {
3.   int sq = i*i;
4.   System.out.println("Hello from task with square = " + sq);
5.   next(); // Barrier
6.   System.out.println("Goodbye from task with square = " + sq);
7. });
```

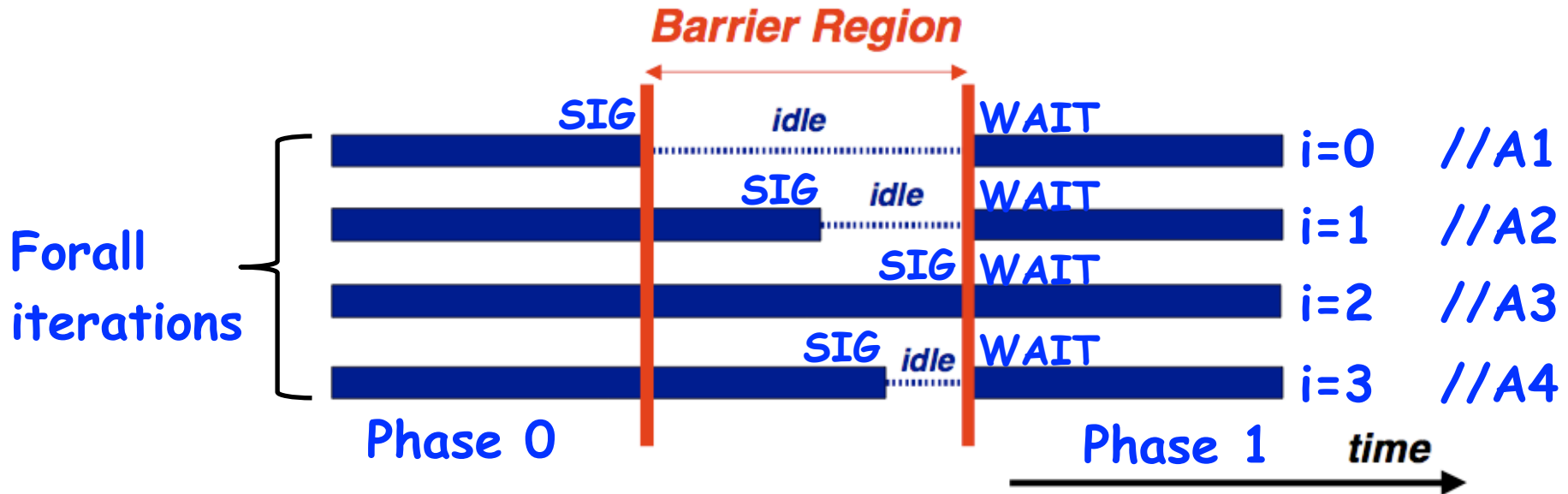
} Phase 0

} Phase 1

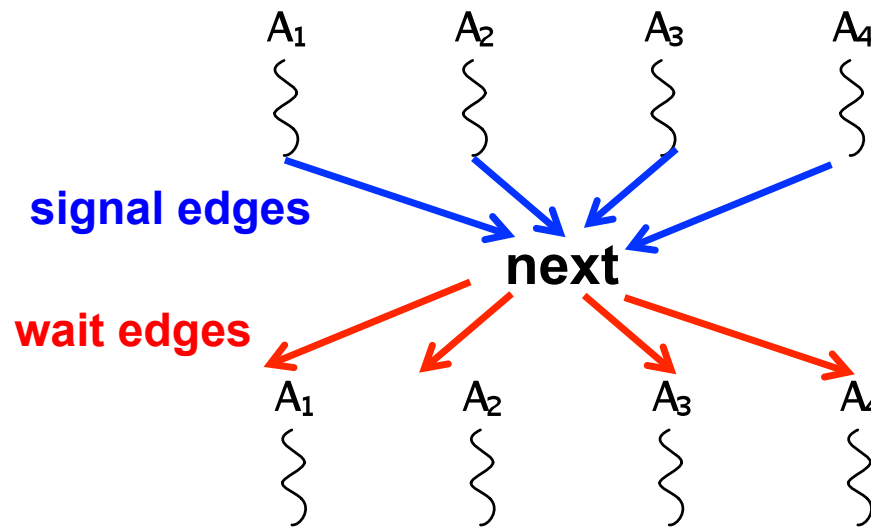
- **next** → each forall iteration waits at barrier until all iterations arrive (previous phase is completed), after which the next phase can start
  - Scope of next is the closest enclosing forall statement
  - If a forall iteration terminates before executing “next”, then the other iterations don’t wait for it
  - Special case of “phaser” construct (will be discussed later in class)



# Impact of barrier on scheduling forall iterations

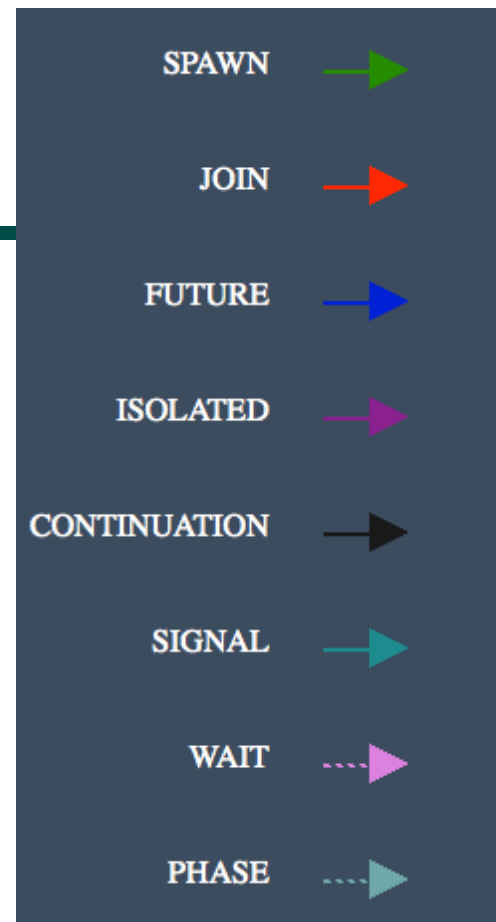


Modeling next operations in the computation graph with signal and wait edges

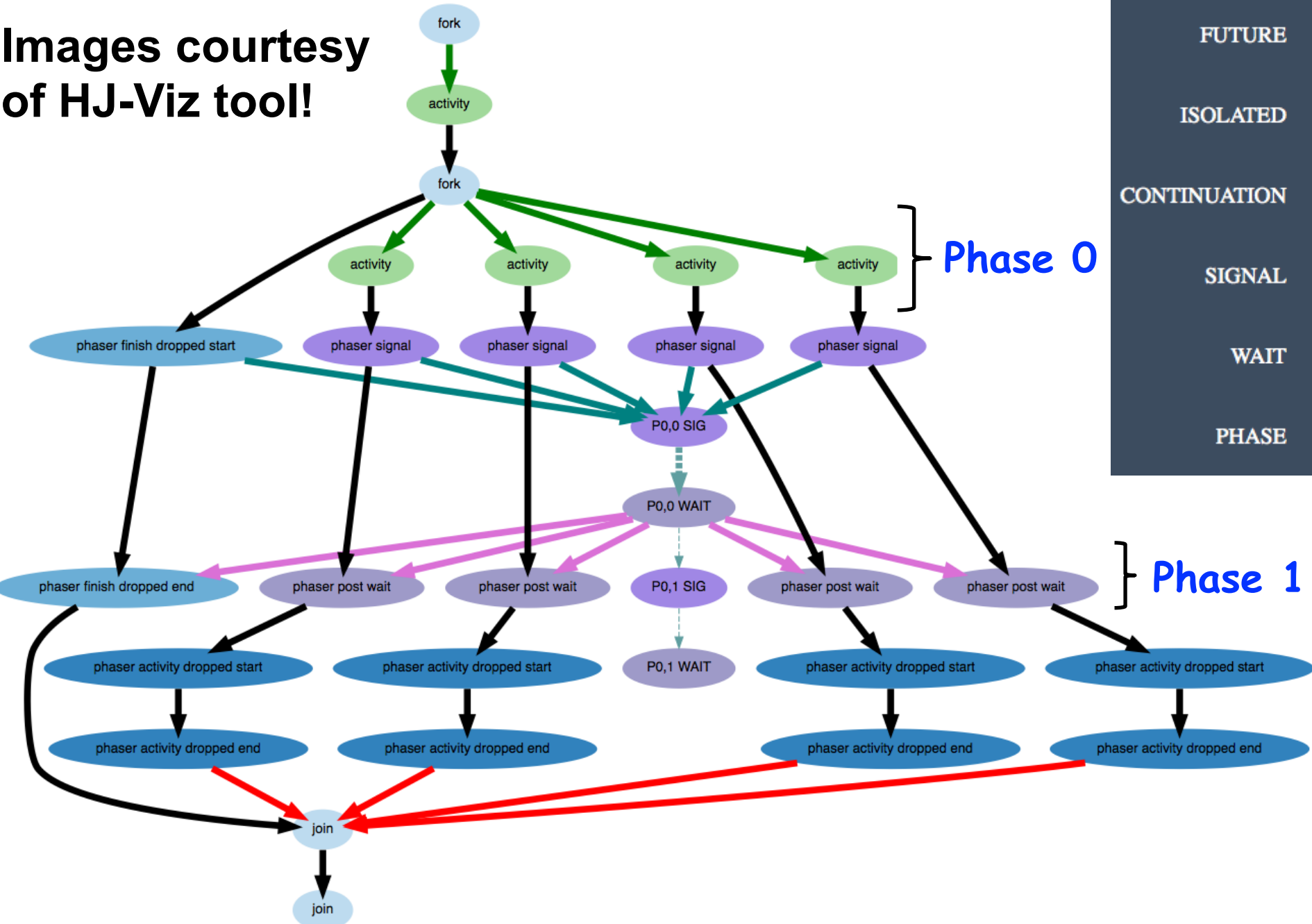




# Computation Graph for Approach 2 with Barriers (m=4)



Images courtesy of HJ-Viz tool!



# 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 forall statement

---

```
1. forallPhased (0, m - 1, (i) -> {
2.   println("Starting forall iteration " + i);
3.   next(); // Acts as barrier for forall-i
4.   forallPhased (0, n - 1, (j) -> {
5.     println("Hello from task (" + i + "," + j + ")");
6.     next(); // Acts as barrier for forall-j
7.     println("Goodbye from task (" + i + "," + j + ")");
8.   } // forall-j
9.   next(); // Acts as barrier for forall-i
10.  println("Ending forall iteration " + i);
11.}); // forall-i
```



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

---

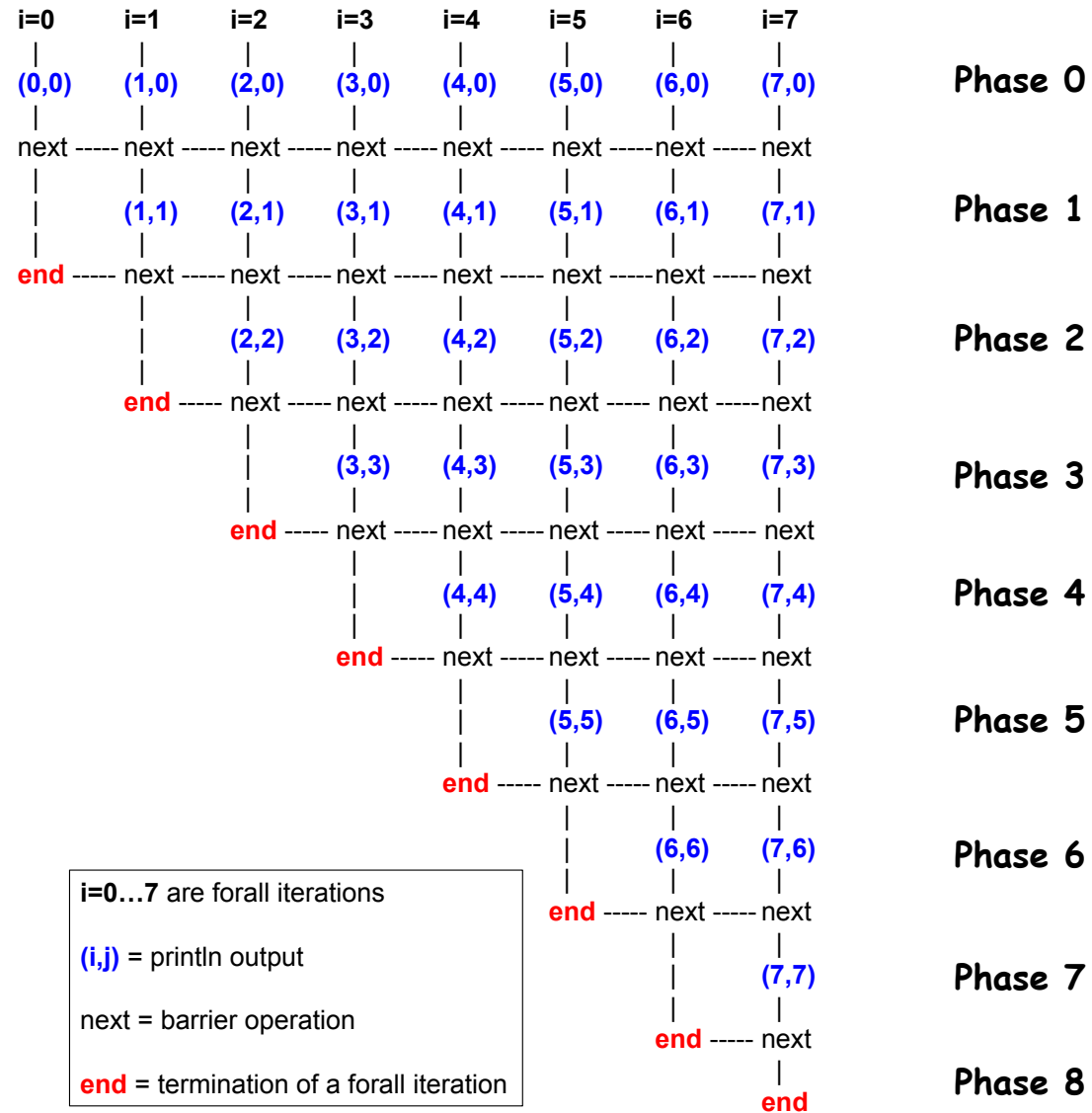
```
1. forallPhased (0, m - 1, (i) -> {
2.   forseq (0, i, (j) -> {
3.     // forall iteration i is executing phase j
4.     System.out.println("(" + i + ", " + j + ")");
5.     next();
6.   });
7. });
```

- 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.



# Barrier Matching for 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



## Observation 3: Different forall 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

