COMP 322: Fundamentals of Parallel Programming

Lecture 12: Iteration Grouping (Chunking), Barrier Synchronization

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	?	0.2	?	0.4	?	0.6	?	0.8	?	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).

Outline of Today's Lecture

- **Iteration Grouping (Chunking)**
- Barrier Synchronization

forall vs. forallChunked APIs (Recap)

- `forall(int s0, int e0, HjProcedure<java.lang.Integer> body)`
- `forall(0, 7, (i) -> BODY(i)); // 8 tasks`

Task 0	Task 1	Task 2	Task 3	Task 4	Task 5	Task 6	Task 7
BODY(0)	BODY(1)	BODY(2)	BODY(3)	BODY(4)	BODY(5)	BODY(6)	BODY(7)

- `forallChunked(int s0, int e0, int chunkSize, edu.rice.hj.api.HjProcedure<java.lang.Integer> body)`
- `forallChunked(0, 7, 2, (i) -> BODY(i)); // 4 tasks`

Task 0		Task 1		Task 2		Task 3	
BODY(0)	BODY(1)	BODY(2)	BODY(3)	BODY(4)	BODY(5)	BODY(6)	BODY(7)

- **Chunking is a special case of "iteration grouping"**

One-Dimensional Grouping (Pseudocode)

```
forall (i : [0:n-1]) S1   ⇒

forall (g : [0:ng-1])
   for(i : myGroup(g,[0:n-1],ng)) S1
```

where

- `ng` **= number of groups**
- `g` **= group id (index variable of outer forall)**
- `myGroup(g,[0:n-1],ng)` **= region corresponding to group g**
  - **No requirement that iterations in a group be contiguous!**

Two Common Groupings

- Block grouping (a.k.a. "chunking")

Task 0		Task 1		Task 2		Task 3	
BODY(0)	BODY(1)	BODY(2)	BODY(3)	BODY(4)	BODY(5)	BODY(6)	BODY(7)

- Cyclic grouping

Task 0		Task 1		Task 2		Task 3	
BODY(0)	BODY(4)	BODY(1)	BODY(5)	BODY(2)	BODY(6)	BODY(3)	BODY(7)

HJlib myGroup() method for 1D Grouping

```
public static HjRegion1D myGroup(final int groupId,
  final HjRegion1D hjRegion1D, final int groupSize) {
    final int lower0 = hjRegion1D.lowerLimit(0);
    final int upper0 = hjRegion1D.upperLimit(0);
    final int range0 = upper0 - lower0 + 1;
    final int increment0 = (range0 / groupSize) +
                             (range0 % groupSize == 0 ? 0 : 1);
    final int start0 = (groupId * increment0) + lower0;
    final int end0 = Math.min(upper0,
                                      start0 + increment0 - 1);
     return newRectangularRegion1D(start0, end0);
 }
```

Example Use of 1D myGroup API

```
HjRegion1D iterSpace = newRectangularRegion1D(0,N-1);
forall(0, tasks - 1, (t) -> {
  HjRegion.HjRegion1D myGroup =
    myGroup(t, iterSpace, tasks);
  forseq(myGroup, (i) -> {
      for (int j = 0; j < N; j++)
        for (int k = 0; k < N; k++)
          C[i][j] += A[i][k] * B[k][j];
  }); // forseq
}); // forall
```

Pros & Cons of Chunked vs. Group APIs

- forallChunked
  - Pro: efficient performance (no call to forseq)
  - Con: grouping policy is determined by library
- myGroup (1D version):
  - Pro: grouping interface exposed to programmer
  - Con: explicit forseq is source of complexity and inefficiency
- myGroup (2D version):
  - Pro: programmer can control number of groups in each dimension
  - Con: explicit forseq is source of complexity and inefficiency

Two-Dimensional Grouping (Pseudocode)

```
forall ([i,j] : [0:n-1,0:n-1]) S1   ⇒

forall ([g0,g1] : [0:ng0-1,0:ng1-1])
  for([i,j] : myGroup([g0,g1], [0:n-1,0:n-1],
                          ng0, ng1) S1
```

where

- `ng0*ng1` **= totalnumber of groups**
- `g0,g1` **= two-dimensional group id**
- `myGroup([g0,g1],[0:n-1,0:n-1],`
  `ng0,ng1)` **= region corresponding to group g0,g1**

HJlib myGroup() method for 2D Grouping

```
public static RectangularRegion2D myGroup(
        final int groupId0, final int groupId1,
        final HjRegion2D hjRegion2D,
        final int groupSize0, final int groupSize1) {

    final int[] lowerLimits = hjRegion2D.lowerLimits();
    final int[] upperLimits = hjRegion2D.upperLimits();

    final int lower0 = lowerLimits[0];
    final int upper0 = upperLimits[0];
    final int range0 = upper0 - lower0 + 1;
    final int increment0 = (range0 / groupSize0) + (range0 % groupSize0 == 0 ? 0 : 1);

    final int lower1 = lowerLimits[1];
    final int upper1 = upperLimits[1];
    final int range1 = upper1 - lower1 + 1;
    final int increment1 = (range1 / groupSize1) + (range1 % groupSize1 == 0 ? 0 : 1);

    final int start0 = (groupId0 * increment0) + lower0;
    final int end0 = Math.min(upper0, start0 + increment0 - 1);

    final int start1 = (groupId1 * increment1) + lower1;
    final int end1 = Math.min(upper1, start1 + increment1 - 1);

    return newRectangularRegion2D(start0, end0, start1, end1);
}
```

Example Use of 2D myGroup API

```
HjRegion2D hjRegion = newRectangularRegion2D(0, N-1, 0, N-1);
final int grid1 = Math.sqrt(tasks);
final int grid2 = grid1;
assert(grid1*grid2==tasks,“tasks is not a perfect square?”)
forall(0, tasks - 1, (t) -> {
  final int id1 = t / grid1;
  final int id2 = t % grid1;
  final RectangularRegion2D myGroup =
        myGroup(id1, id2, hjRegion, grid1, grid2);
    forseq(myGroup, (i, j) -> {
              computationKernel(A, B, C, N, i, j);
   }); // forseq
 }); // forall

```

k-dimensional Iteration Grouping (General approach in pseudocode)

- Assume that the forasync loop originally iterates over region r

```
forall(r, (p) -> BODY(p)); // No. of tasks = r.size()
```

- Assume that we have an int array, nc = {nc0, nc1, ...}, for the desired number of chunks in each dimension
  - A good choice is to choose these values such that the product of nc[0]*nc[1]*... = Runtime.getNumOfWorkers()
- Assume that we have a helper method, *getChunk(id, r, nc)* that returns the iteration range for chunk pp as an HJ region
  - e.g., getChunk([0,0], [0:99,0:99], {2,2}) = [0:49,0:49]
-

Outline of Today's Lecture

- Iteration Grouping (Chunking)

- <u>Barrier Synchronization</u>

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)

```
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 one forall to the next**

```
// 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?
- Approach 2: insert a "barrier" between the hello's and goodbye's
  - "next" statement in HJ's forall loops

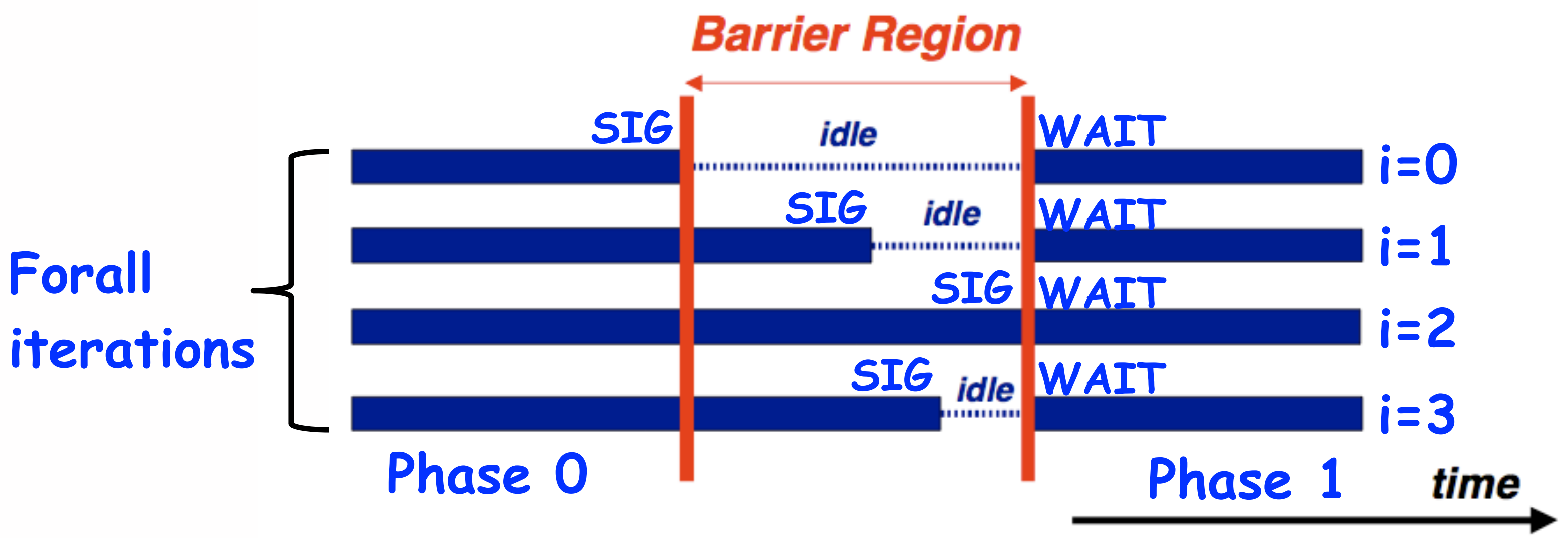
```
// APPROACH 2
forallPhased (0, m - 1, (i) -> {
 int sq = i*i;                                                  } Phase 0
 System.out.println("Hello from task with square = " + sq);     }
 next(); // Barrier
 System.out.println("Goodbye from task with square = " + sq);   } Phase 1
});
```

- 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)

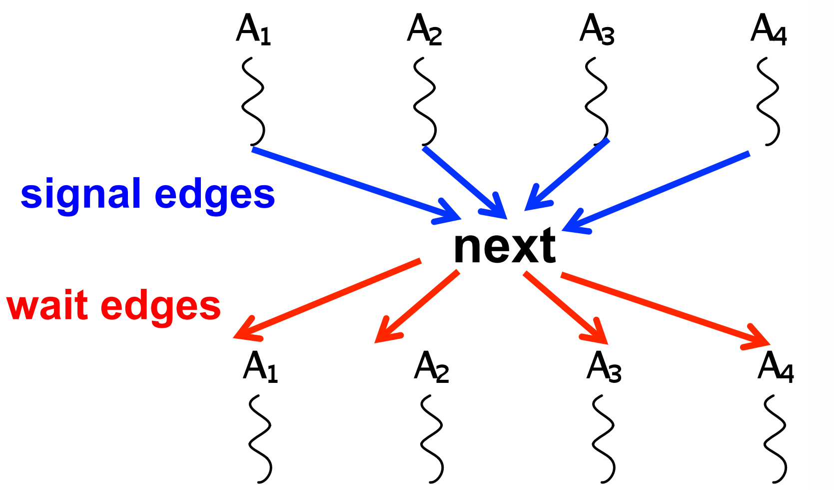
forallPhased API's in HJlib
(http://www.cs.rice.edu/~vs3/hjlib/doc/edu/rice/hj/Module1.html)

- `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.
  - Calls to next() are only permitted in forallPhased() not in forall()

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

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

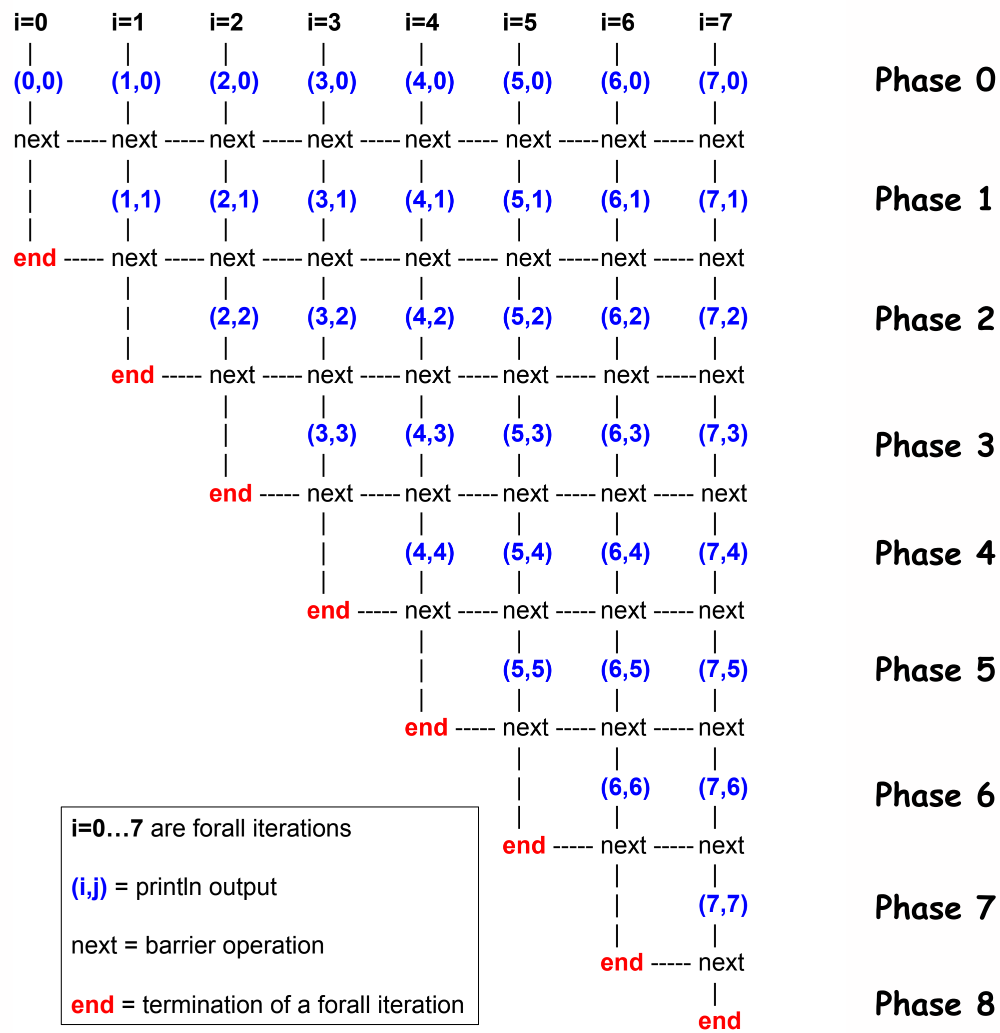
Observation 2: If a forall iteration terminates before "next", then other iterations do not wait for it

```
1. 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.

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



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

```
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
```

- 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()

Worksheet #12: Forall Loops and Barriers

Name: ______________________ Netid: ______________________

Draw a "barrier matching" figure similar to slide 23 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++) {
    // forall iteration i is executing phase j
    System.out.println("(" + i + "," + j + ")");
    next();
  }
});
```