
COMP 322: Fundamentals of Parallel Programming

Lecture 13: Forall and Barriers (contd), Data-driven tasks

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

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

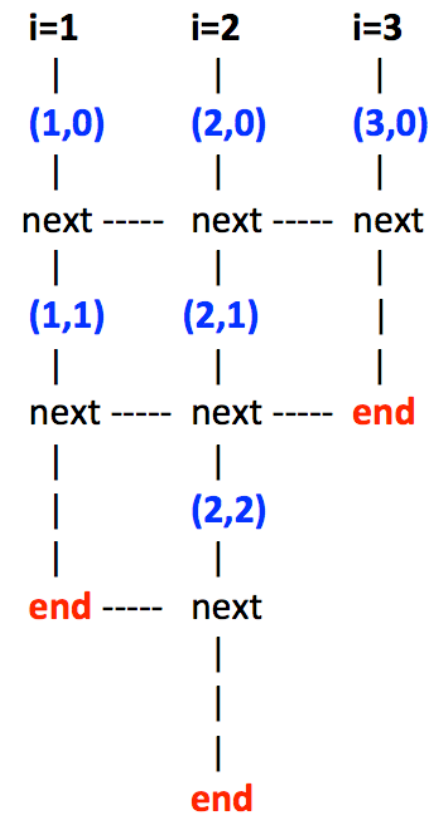


Worksheet #12: Forall Loops and Barriers

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

```
1. String[] a = { "ab", "cde", "f" };
2. . . . int m = a.length; . . .
3. forall (point[i] : [1:m]) {
4.     for (int j = 0; j < a[i-1].length(); j++) {
5.         // forall iteration i is executing phase j
6.         System.out.println("(" + i + "," + j + ")");
7.         next;
8.     }
9. }
```

Solution



Outline of Today's Lecture

- **Barrier Synchronization in Forall Loops (contd)**
- **Dataflow Computing, Data-Driven Futures (DDFs) and Data-Driven Tasks (DDTs)**

Acknowledgments

- COMP 322 Module 1 handout, Chapters 10, 11



One-Dimensional Iterative Averaging: chunkedForkJoin version with *chunked for-forall-for* structure (Recap)

```
1. double[] gVal=new double[n+2]; double[] gNew=new double[n+2];
2. gVal[0] = 0; gVal[n+1] = 1; // boundary condition
3. int nc = Runtime.getNumOfWorkers(); // number of chunks
4. double[] myVal = gVal; double[] myNew = gNew;
5. for (point [iter] : [0:m-1]) {
6.     // Compute MyNew as function of input array MyVal
7.     forall (point [jj] : [0:nc-1]) {
8.         for(point [j] : getChunk([1:n],nc,jj))
9.             myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
10.    } // forall
11.    temp=myVal; myVal=myNew; myNew=temp;// Swap myVal & myNew;
12.    // myNew becomes input array for next iteration
13.} // for
```

This program creates $m \cdot nc$ async tasks



One-Dimensional Iterative Averaging: Barrier version with *chunked forall-for-for+next* structure (Recap)

```
1. double[] gVal=new double[n+2]; double[] gNew=new double[n+2]; gVal[n+1] = 1;
2. int nc = Runtime.getNumWorkers();
3. forall (point [jj]:[0:nc-1]) { // Chunked forall is now the outermost loop
4.     double[] myVal = gVal; double[] myNew = gNew; // Copy of myVal/myNew pointers
5.     for (point [iter] : [0:m-1]) {
6.         // Compute MyNew as function of input array MyVal
7.         for (point [j]:getChunk([1:n],nc,jj)) // Iterate within chunk
8.             myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
9.         next; // Barrier before executing next iteration of iter loop
10.        // Swap local pointers, myVal and myNew
11.        double[] temp=myVal; myVal=myNew; myNew=temp;
12.        // myNew becomes input array for next iter
13.    } // for
14.} // forall
```

This program creates nc async tasks, and performs $m*nc$ barrier operations



What just happened?

chunkedForkJoin version:

```
5. for (point [iter] : [0:m-1])
7.   forall (point [jj] : [0:nc-1]) {
8,9.     for(point [j] : getChunk([1:n],nc,jj)) { ... }
10.  } // forall
    . . .
13. } // for
```

barrier version:

```
3. forall (point [jj]:[0:nc-1]))
5.   for (point [iter] : [0:m-1]) {
    . . .
7,8.   for (point [j]:getChunk([1:n],nc,jj)) { ... } // for
9.     next;
    . . .
13,   } // for
15. } // forall
```



Single Program Multiple Data (SPMD) Parallel Programming Model

Basic idea

- Run the same code (program) on P workers
- Use the “rank” --- an ID ranging from 0 to (P-1) --- to determine what data is processed by which worker
 - Hence, “single-program” and “multiple-data”
 - Rank is equivalent to index in a top-level “forall (point[i] : [0:P-1])” loop
- Lower-level programming model than dynamic async/finish parallelism
 - Programmer’s code is essentially at the worker level (each forall iteration is like a worker), and work distribution is managed by programmer by using barriers and other synchronization constructs
 - Harder to program but can be more efficient for restricted classes of applications (e.g. for OneDimAveraging, but not for nqueens)
- Convenient for hardware platforms that are not amenable to efficient dynamic task parallelism
 - General-Purpose Graphics Processing Unit (GPGPU) accelerators
 - Distributed-memory parallel machines



One-Dimensional Iterative Averaging: Barrier version with *chunked forall-for-for+next* structure is an SPMD program

```
1. double[] gval=new double[n+2]; double[] gNew=new double[n+2]; gval[n+1] = 1;
2. int nc = Runtime.getNumWorkers();
3. forall (point [jj]:[0:nc-1]) { // Chunked forall is now the outermost loop
4.     double[] myval = gval; double[] myNew = gNew; // Copy of myval/myNew pointers
5.     for (point [iter] : [0:m-1]) {
6.         // Compute MyNew as function of input array Myval
7.         for (point [j]:getChunk([1:n],nc,jj)) // Iterate within chunk
8.             myNew[j] = (myval[j-1] + myval[j+1])/2.0;
9.         next; // Barrier before executing next iteration of iter loop
10.        // Swap local pointers, myval and myNew
11.        double[] temp=myval; myval=myNew; myNew=temp;
12.        // myNew becomes input array for next iter
13.    } // for
14.} // forall
```

Instead of async-finish, this SPMD version of OneDimAveraging creates one task per worker, uses getChunk() to distribute work, and use barriers to synchronize workers.



Motivation for “single” statement with barriers --- Hello Goodbye Example revisited (Listing 36)

- **Goal:** rewrite Hello-Goodbye example so as to print a single log message in between phases
- **Simple solution:** add a second barrier and designate a specific forall task to print the log message between those two barriers

```
1. // Listing 36 in Module 1 handout
2. forall (point[i] : [0:m-1]) {
3.   int sq = i*i;
4.   System.out.println("Hello from task with square = " + sq);
5.   next; // Barrier
6.   if (i==0) System.out.println("LOG: Between Hello & Goodbye phases");
7.   next; // Barrier
8.   System.out.println("Goodbye from task with square = " + sq);
9. }
```

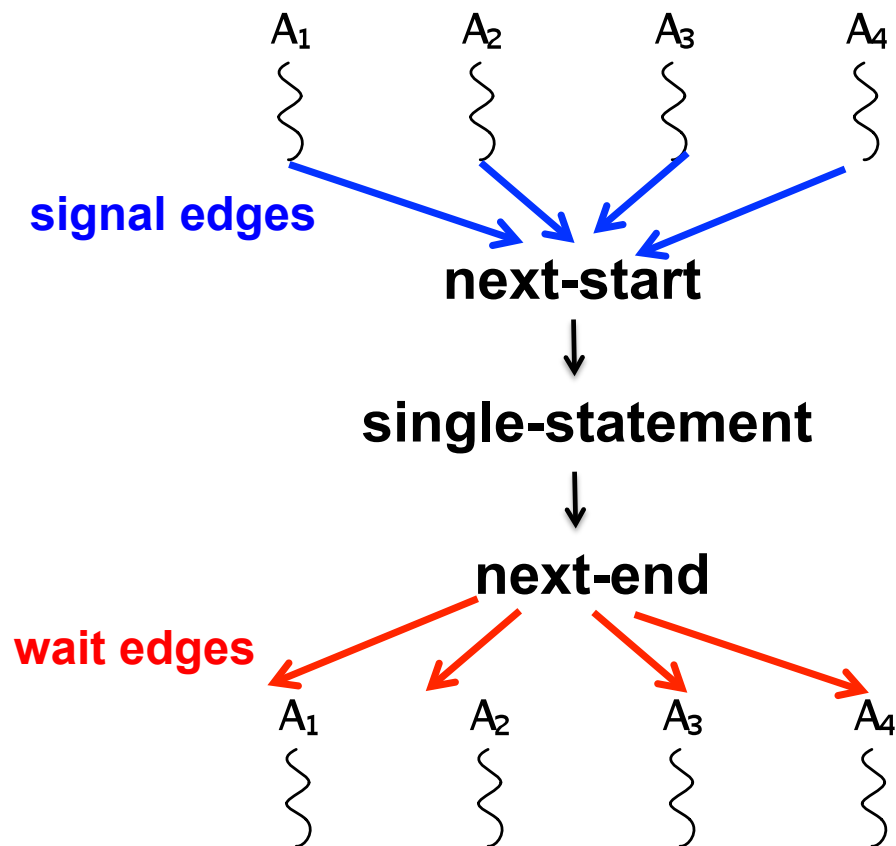
- **More efficient solution:** use next-with-single



Next-with-Single Statement

“next single-stmt;” is a barrier in which single-stmt is performed exactly once after all tasks have completed the previous phase and before any task begins its next phase.

Modeling next-with-single in the Computation Graph



Use of next-with-single to print a log message between Hello and Goodbye phases

```
1. // Listing 37 in Module 1 handout
2. forall (point[i] : [0:m-1]) {
3.   int sq = i*i;
4.   System.out.println("Hello from task with square = " + sq);
5.   next { // next-with-single statement
6.     System.out.println("LOG: Between Hello & Goodbye phases");
7.   }
8.   System.out.println("Goodbye from task with square = " + sq);
9. }
```



One-Dimensional Iterative Averaging with Single Statement and global gVal & gNew fields

```
1. static double[] gVal=new double[n+2];
2. static double[] gNew=new double[n+2];
3. . . .
4. gVal[n+1] = 1; // Boundary condition
5. int nc = Runtime.getNumWorkers();
6. forall (point [jj]:[0:nc-1]) { // forall is now outermost loop
7.     for (point [iter] : [0:m-1]) {
8.         // Compute Gnew as function of input array Gval
9.         for (point [j]:getChunk([1:n],nc,jj)) // Iterate within chunk
10.            gNew[j] = (gVal[j-1] + gVal[j+1])/2.0;
11.         // Use next-with-single
12.         next {double[] temp=gVal; gVal=gNew; gNew=temp;} // single
13.         // gNew becomes input array for next iter
14.     } // for
15.} // forall
```



Outline of Today's Lecture

- **Barrier Synchronization in Forall Loops (contd)**
- **Dataflow Computing, Data-Driven Futures (DDFs) and Data-Driven Tasks (DDTs)**

Acknowledgments

- COMP 322 Module 1 handout, Chapters 10, 11



Dataflow Computing

- Original idea: replace machine instructions by a small set of dataflow operators

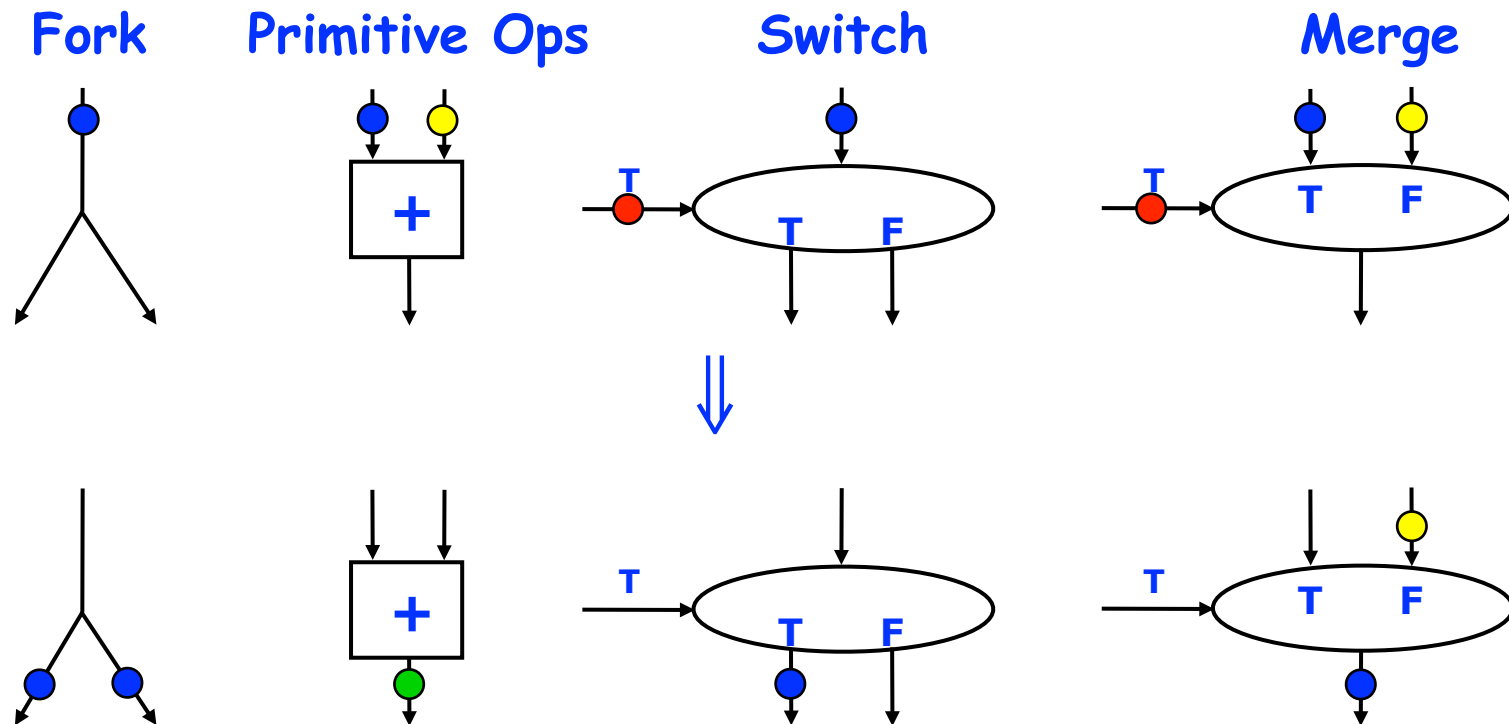
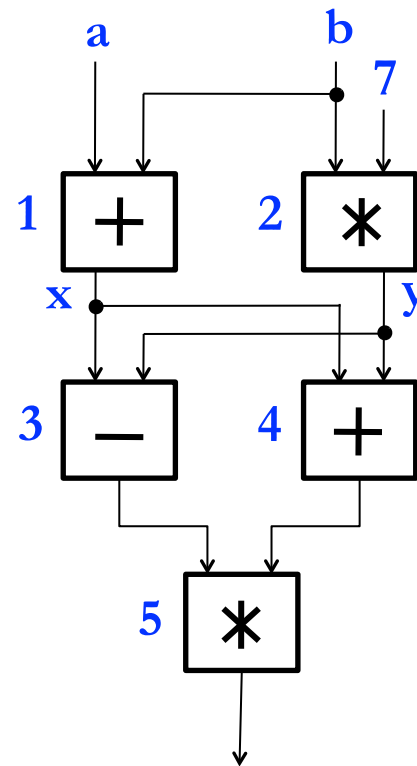


Figure 37: Example instruction sequence and its dataflow graph

```
x = a + b;  
y = b * 7;  
z = (x - y) * (x + y);
```

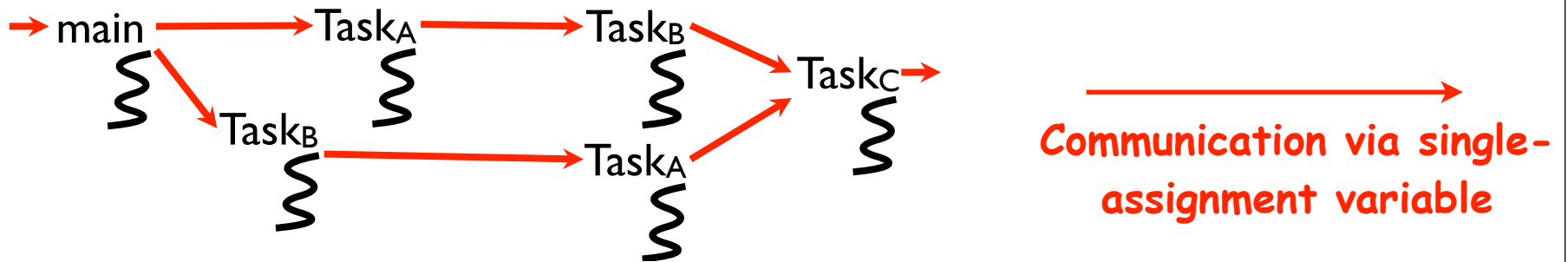


No separate control flow

An operator executes when all its input values are present; copies of the result value are distributed to the destination operators.



Macro-Dataflow Programming



- “Macro-dataflow” = extension of dataflow model from instruction-level to task-level operations
- General idea: build an arbitrary task graph, but restrict all inter-task communications to single-assignment variables
 - Static dataflow ==> graph fixed when program execution starts
 - Dynamic dataflow ==> graph can grow dynamically
- Semantic guarantees: race-freedom, determinism
 - Deadlocks are possible due to unavailable inputs (but they are deterministic)



Extending HJ Futures for Macro-Dataflow: Data-Driven Futures (DDFs) and Data-Driven Tasks (DDTs)

```
ddfA = new DataDrivenFuture<T1>();
```

- Allocate an instance of a data-driven-future object (container)
- Object in container must be of type T1

```
async await(ddfA, ddfB, ...) Stmt
```

- Create a new data-driven-task to start executing **Stmt** after all of **ddfA**, **ddfB**, ... become available (i.e., after task becomes “enabled”)

```
ddfA.put(V) ;
```

- Store object V (of type T1) in **ddfA**, thereby making **ddfA** available
- Single-assignment rule: at most one put is permitted on a given DDF

```
ddfA.get()
```

- Return value (of type T1) stored in **ddfA**
- Can only be performed by **async**'s that contain **ddfA** in their **await** clause (hence no blocking is necessary for DDF gets)



Implementing Future Tasks using DDFs

- **Future version**

1. `final future<T> f = async<T> { return g(); };`
2. `S1`
3. `... = f.get();`
4. `S2`
5. `S3`

- **DDF version**

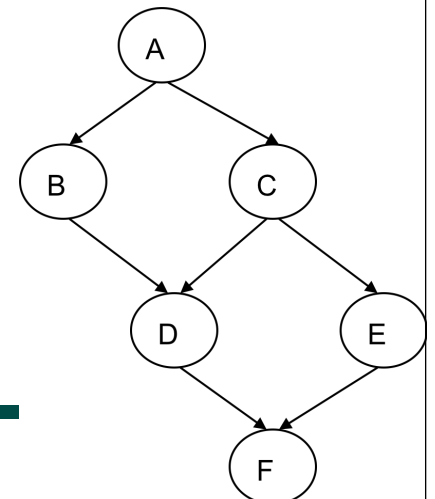
1. `DataDrivenFuture<T> f = new DataDrivenFuture<T>();`
2. `async { f.put(g()) };`
3. `S1`
4. `finish async await(f) {`
5. `... = f.get();`
6. `S2 // DDT must include full continuation starting`
7. `S3 // with S2`
8. `}`



Use of DDFs with dummy objects (like future<void>)

```
1. finish {
2.   DataDrivenFuture ddfA = new DataDrivenFuture();
3.   DataDrivenFuture ddfB = new DataDrivenFuture();
4.   DataDrivenFuture ddfC = new DataDrivenFuture();
5.   DataDrivenFuture ddfD = new DataDrivenFuture();
6.   DataDrivenFuture ddfE = new DataDrivenFuture();
7.   async { ... ; ddfA.put(""); } // Task A
8.   async await(ddfA) { ... ; ddfB.put(""); } // Task B
9.   async await(ddfA) { ... ; ddfC.put(""); } // Task C
10.  async await(ddfB,ddfC) { ... ; ddfD.put(""); } // Task D
11.  async await(ddfC) { ... ; ddfE.put(""); } // Task E
12.  async await(ddfD,ddfE) { ... } // Task F
13. }
```

- This example uses an empty string as a dummy object



Differences between Futures and DDFs/DDTs

- Consumer task blocks on `get()` for each future that it reads, whereas `async-await` does not start execution till all DDFs are available
- Future tasks cannot deadlock, but it is possible for a DDT to block indefinitely (“deadlock”) if one of its input DDFs never becomes available
- DDTs and DDFs are more general than futures
 - Producer task can only write to a single future object, where as a DDT can write to multiple DDF objects
 - The choice of which future object to write to is tied to a future task at creation time, where as the choice of output DDF can be deferred to any point with a DDT
- DDTs and DDFs can be more implemented more efficiently than futures
 - An “`async await`” statement does not block the worker, unlike a `future.get()`
 - You will never see the following message with “`async await`”
 - “ERROR: Maximum number of hj threads per place reached”



Two Exception (error) cases for DDFs that do not occur in futures

- **Case 1:** If two put's are attempted on the same DDF, an exception is thrown because of the violation of the single-assignment rule
 - There can be at most one value provided for a future object (since it comes from the producer task's return statement)
- **Case 2:** If a get is attempted by a task on a DDF that was not in the task's await list, then an exception is thrown because DDF's do not support blocking gets
 - Futures support blocking gets



Deadlock example with DDTs

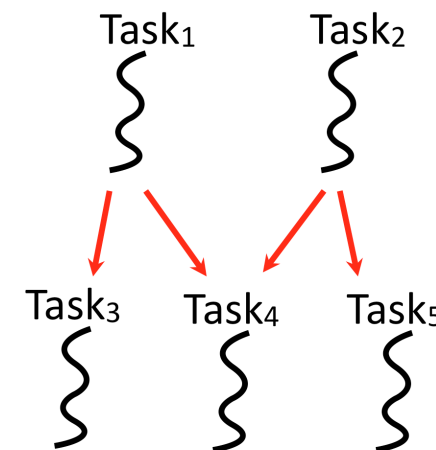
```
1. DataDrivenFuture left = new DataDrivenFuture();
2. DataDrivenFuture right = new DataDrivenFuture();
3. finish {
4.     async await(left) right.put(rightWriter());
5.     async await(right) left.put(leftWriter());
6. }
```



Another Example with DDTs and DDFs

```
1. DataDrivenFuture left = new DataDrivenFuture();
2. DataDrivenFuture right = new DataDrivenFuture();
3. finish {
4.   async await(left) leftReader(left); // Task3
5.   async await(right) rightReader(right); // Task5
6.   async await(left, right)
7.     bothReader(left, right); // Task4
8.   async left.put(leftWriter()); // Task1
9.   async right.put(rightWriter()); // Task2
10. }
```

- **await** clauses capture data flow relationships



Interesting example. Let's discuss it further in Worksheet 13!



Implementing DDFs/DDTs using Future tasks

Shown for completeness, but not recommend for performance ...

- DDF version

```
DataDrivenFuture f1 = new DataDrivenFuture();
DataDrivenFuture f2 = new DataDrivenFuture();
async { f1.put(g()) }; async { f2.put(h()) };
// async doesn't start till f1 & f2 are available
async await (f1, f2) {
    ... = f1.get() + f2.get(); };
```

- Future version

```
final future<int> f1 = async<int> { return g(); };
final future<int> f2 = async<int> { return h(); };
// Async may block at each get() operation
async { ... = f1.get() + f2.get(); };
```



Worksheet #13: Forall Loops and Barriers

Name 1: _____

Name 2: _____

For the example below, will reordering the five `async` statements change the meaning of the program? If so, show two orderings that exhibit different behaviors. If not, explain why not. (You can use the space below this slide for your answer.)

```
1. DataDrivenFuture left = new DataDrivenFuture();
2. DataDrivenFuture right = new DataDrivenFuture();
3. finish {
4.     async await(left) leftReader(left); // Task3
5.     async await(right) rightReader(right); // Task5
6.     async await(left, right)
7.         bothReader(left, right); // Task4
8.     async left.put(leftWriter()); // Task1
9.     async right.put(rightWriter()); // Task2
10. }
```

