

COMP 322: Fundamentals of Parallel Programming

Lecture 12: Data-Driven Tasks

Mack Joyner
mjoyner@rice.edu

<http://comp322.rice.edu>

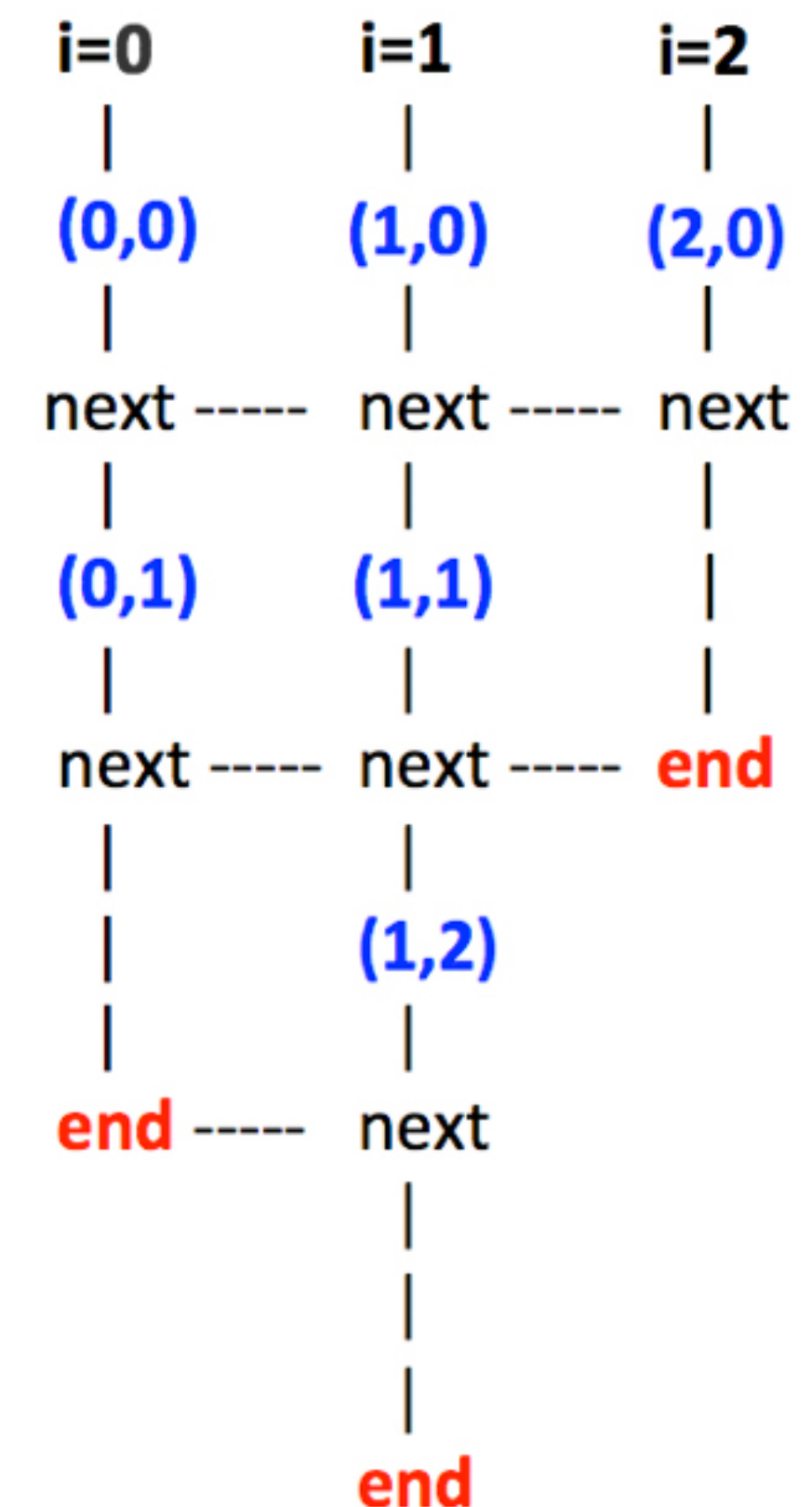


Worksheet #11: Forall Loops and Barriers

Draw a “barrier matching” figure similar to lecture 12 slide 11 for the code fragment below.

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

Solution



HJ code for One-Dimensional Iterative Averaging using nested `forseqforall` structure

```
1. double[] myVal=new double[n+2]; myVal[n+1] = 1;
2. double[] myNew=new double[n+2]; myNew[n+1] = myVal[n+1];
3. forseq(0, m-1, (iter) -> {
4. // Compute MyNew as function of input array MyVal
5. forall(1, n, (j) -> { // Create n tasks
6.     myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
7. }); // forall
8. // Swap myVal & myNew;
9. temp=myVal; myVal=myNew; myNew=temp;
10. // myNew becomes input array for next iteration
11.}); // for
```



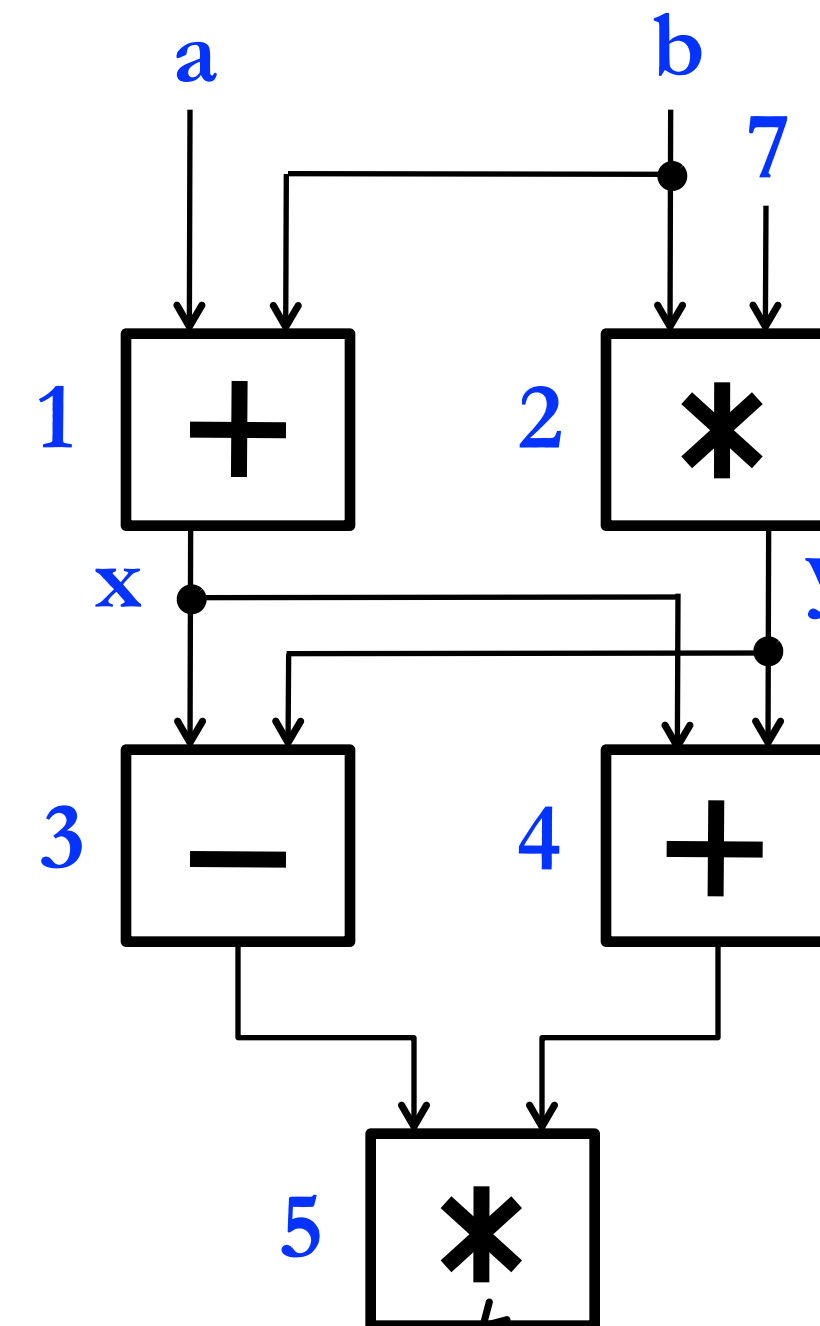
Example: HJ for One-Dimensional Iterative Averaging with chunking

```
1. double[] myVal=new double[n+2]; myVal[n+1] = 1;
2. double[] myNew=new double[n+2]; myNew[n+1] = myVal[n+1];
3. int nc = numWorkerThreads();
4. forseq(0, m-1, (iter) -> {
5.     // Compute MyNew as function of input array MyVal
6.     forallChunked(1, n, n/nc, (j) -> { // Create nc tasks
7.         myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
8.     }); // forallChunked
9.     temp=myVal; myVal=myNew; myNew=temp;// Swap myVal & myNew;
10. // myNew becomes input array for next iteration
11.}); // for
```



Example instruction sequence and its dataflow graph

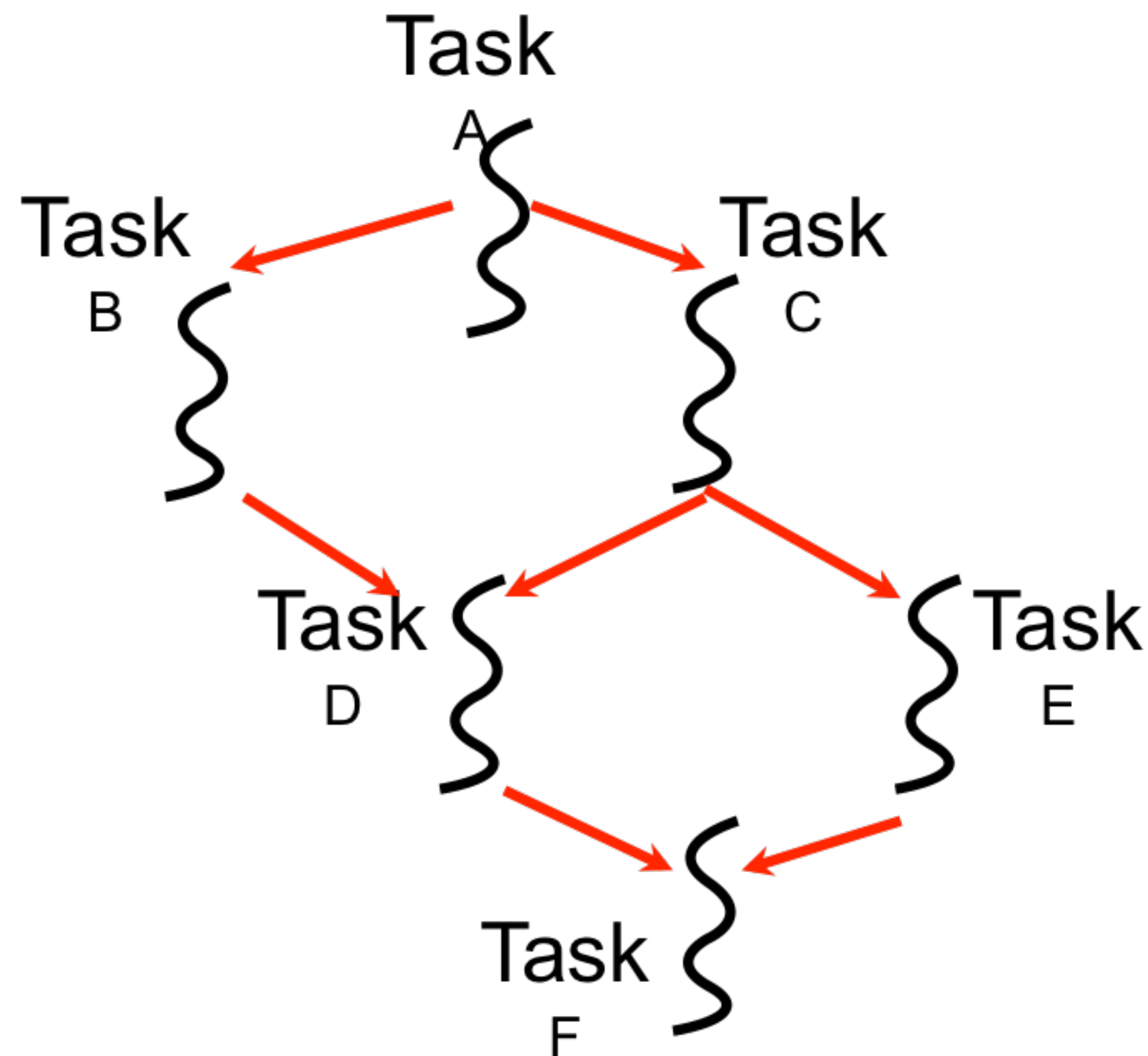
```
x = a + b;  
y = b * 7;  
z = (x - y) * (x + y);
```



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



Macro-Dataflow Programming



→
Communication via “single-assignment” variables

- “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 (like futures)
 - 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)

```
HjDataDrivenFuture<T1> ddfA = newDataDrivenFuture( );
```

- Allocate an instance of a data-driven-future object (container)
- Object in container must be of type T1, and can only be assigned once via put() operations
- HjDataDrivenFuture extends the HjFuture interface

```
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



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

```
asyncAwait(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”)
- Await clause can be used to implement “nodes” and “edges” in a computation graph

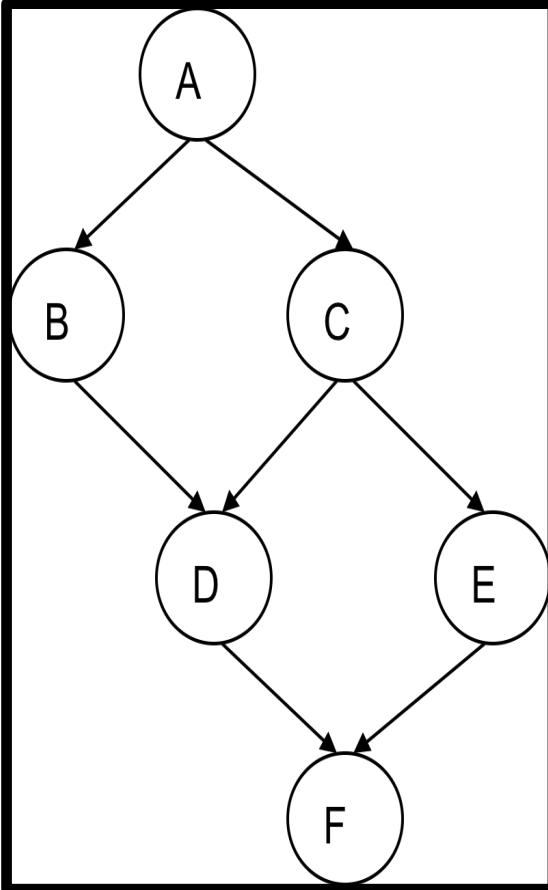
```
ddfA.get()
```

- Return value (of type T1) stored in `ddfA`
- Throws an exception if `put()` has not been performed
 - Should be performed by `async`'s that contain `ddfA` in their await clause, or if there's some other synchronization to guarantee that the `put()` was performed



Converting previous Future example to Data-Driven Futures and AsyncAwait Tasks

```
1. finish(() -> {
2.   HjDataDrivenFuture<Void> ddfA = newDataDrivenFuture();
3.   HjDataDrivenFuture<Void> ddfB = newDataDrivenFuture();
4.   HjDataDrivenFuture<Void> ddfC = newDataDrivenFuture();
5.   HjDataDrivenFuture<Void> ddfD = newDataDrivenFuture();
6.   HjDataDrivenFuture<Void> ddfE = newDataDrivenFuture();
7.   asyncAwait(ddfA, () -> { ... ; ddfB.put(...); }); // Task B
8.   asyncAwait(ddfA, () -> { ... ; ddfC.put(...); }); // Task C
9.   asyncAwait(ddfB, ddfC, ()->{ ... ; ddfD.put(...); }); // Task D
10.  asyncAwait(ddfC, () -> { ... ; ddfE.put(...); }); // Task E
11.  asyncAwait(ddfD, ddfE, () -> { ... }); // Task F
12.  // Note that creating a “producer” task after its “consumer”
13.  // task is permitted with DDFs & DDTs, but not with futures
14.  async(() -> { ... ; ddfA.put(...); }); // Task A
15. }); // finish
```



What is Deadlock?

- A parallel program execution contains a deadlock if some task's execution remains incomplete due to it being *blocked indefinitely* awaiting some condition
- Example of a program with a deadlocking execution

```
DataDrivenFuture left = new DataDrivenFuture();
DataDrivenFuture right = new DataDrivenFuture();
finish {
    async await ( left ) right.put(rightBuilder()); // Task1
    async await ( right ) left.put(leftBuilder()); // Task2
}
```

- In this case, Task1 and Task2 are in a deadlock cycle.
- HJ-Lib has a deadlock detection debug option, which can be enabled as follows:
 - `System.setProperty(HjSystemProperty.trackDeadlocks.propertyKey(), "true");`
 - Throws an `edu.rice.hj.runtime.util.DeadlockException` when deadlock detected



Implementing Future Tasks using DDFs

- Future version

```
1. HjFuture<T> f = future(() -> { return g(); });
2. S1
3. async(() -> {
4.     ... = f.get(); // blocks if needed
5.     S2;
6.     S3;
7. });
```

- DDF version

```
1. HjDataDrivenFuture<T> f = newDataDrivenFuture();
2. async(() -> { f.put(g()) });
3. S1
4. asyncAwait(f, () -> {
5.     ... = f.get(); // does not block -- why?
6.     S2;
7.     S3;
8. });
```



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, whereas 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
 - Consumer DDTs can be created before the producer tasks
- DDTs and DDFs can be implemented more efficiently than futures
 - An “`asyncAwait`” statement does not block the worker, unlike a `future.get()`



Two Exception (error) cases for DDFs that cannot occur with 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 (cannot be reproduced with futures)

- A parallel program execution contains a deadlock if some task's execution remains incomplete due to it being *blocked indefinitely* awaiting some condition

```
1. HjDataDrivenFuture left = newDataDrivenFuture();
2. HjDataDrivenFuture right = newDataDrivenFuture();
3. finish(() -> {
4.     asyncAwait(left, () -> {
5.         right.put(rightWriter()); });
6.     asyncAwait(right, () -> {
7.         left.put(leftWriter()); });
8. });
```

- HJ-Lib has deadlock detection mode
- Enabled using:
 - `System.setProperty(HjSystemProperty.trackDeadlocks.propertyKey(), "true");`
 - Throws an `edu.rice.hj.runtime.util.DeadlockException` when deadlock detected



Announcements & Reminders

- Quiz for Unit 2 (topics 2.1 - 2.8) is due **today** by 11:59pm
- HW #2 is due Wednesday, March 3rd by 11:59pm
- No class on Monday (“Sprinkle” day)
- Lab #4 on Tuesday/Thursday
- Lab #3 due Thursday, March 4th by 2pm
- Midterm Exam on Thursday, Mar. 11th from 7-9pm in Canvas



Worksheet 12: Data Driven Futures

For the example below, will reordering the five `async` statements change the meaning of the program (assuming that the semantics of the reader/writer methods depends only on their parameters) ? If so, show two orderings that exhibit different behaviors. If not, explain why not.

```
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. }
```

