

COMP 322: Fundamentals of Parallel Programming

Lecture 9: Async, Finish, Data-Driven Tasks

Mack Joyner and Zoran Budimlić
{mjoyner, zoran}@rice.edu

<http://comp322.rice.edu>



Async and Finish Statements for Task Creation and Termination

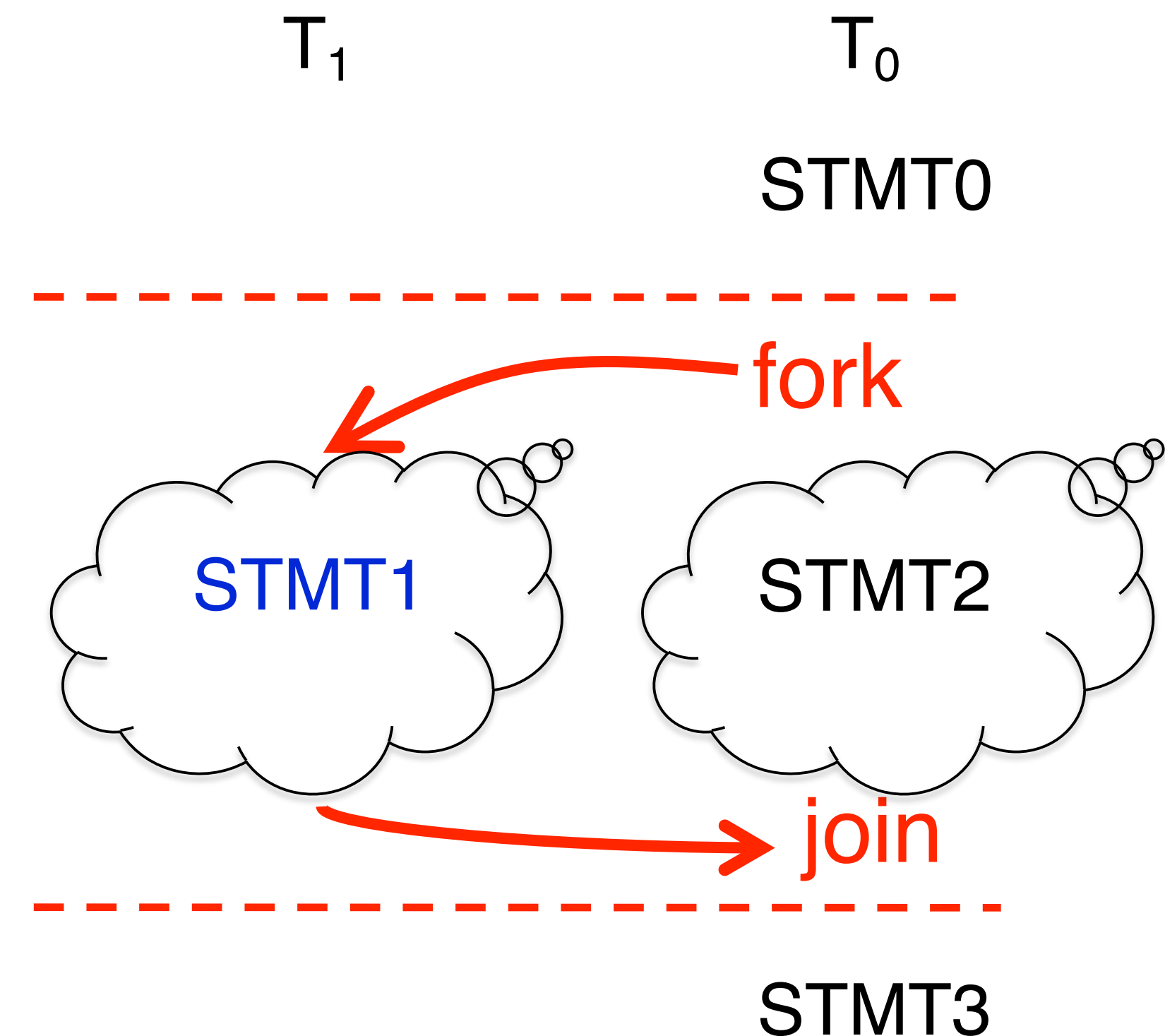
async S

- Creates a new child task that executes statement **S**

```
// T0(Parent task)
STMT0;
finish { //Begin finish
  async {
    STMT1; //T1(Child task)
  }
  STMT2; //Continue in T0
} //End finish (wait for T1)
STMT3; //Continue in T0
```

finish S

- Execute **S**, but wait until *all* asyncs in **S**'s scope have terminated.



Example of a Sequential Program: Computing sum of array elements

Algorithm 1: Sequential ArraySum

Input: Array of numbers, X .

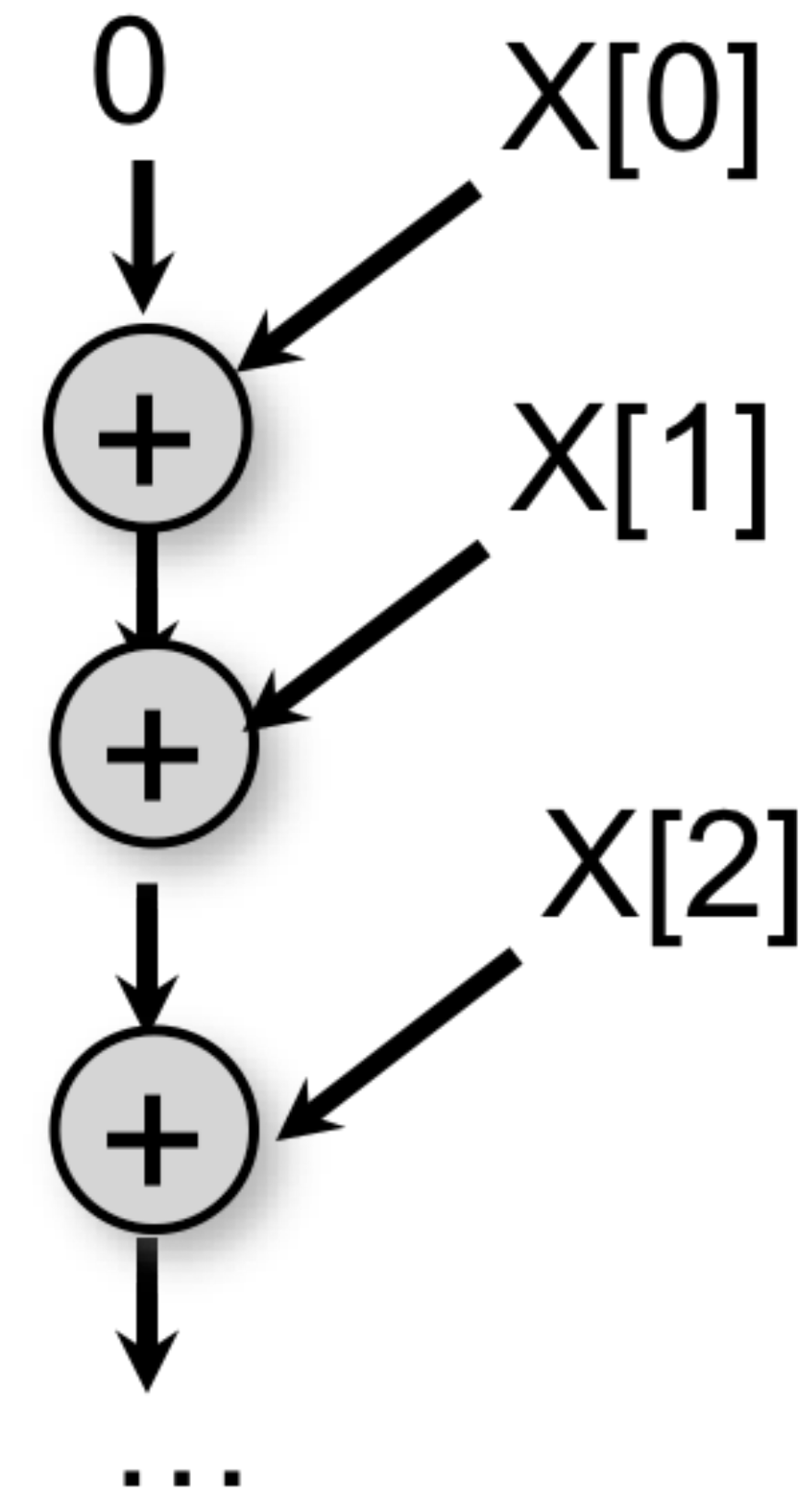
Output: sum = sum of elements in array X .

$sum \leftarrow 0$;

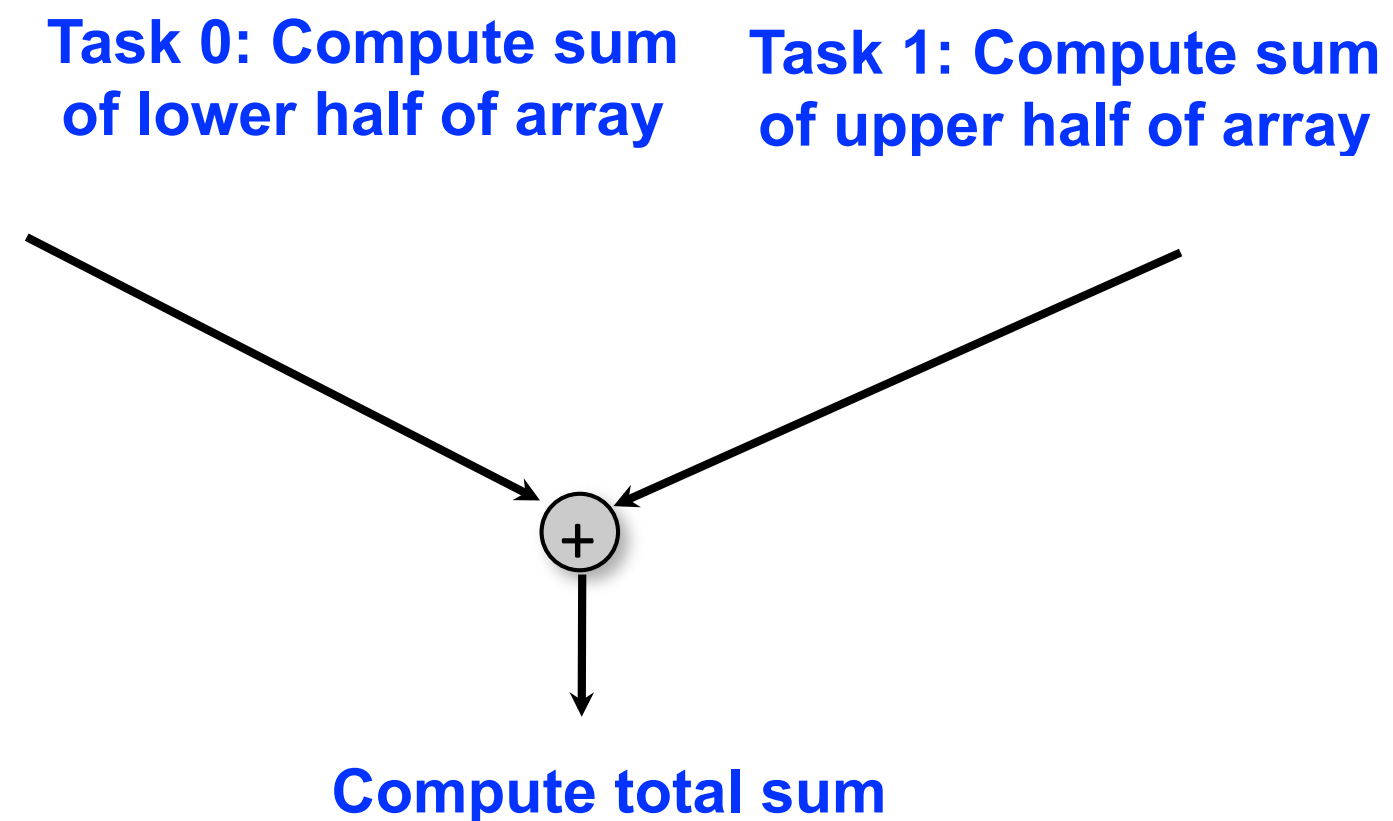
for $i \leftarrow 0$ **to** $X.length - 1$ **do**

$sum \leftarrow sum + X[i]$;

return sum ;



Parallelization Strategy for 2 cores (Two-way Parallel Array Sum)



Basic idea:

- Decompose problem into two tasks for partial sums
- Combine results to obtain final answer
- Parallel divide-and-conquer pattern



Two-way Parallel Array Sum using async & finish constructs

Algorithm 2: Two-way Parallel ArraySum

Input: Array of numbers, X .

Output: $sum = \text{sum of elements in array } X$.

// Start of Task T1 (main program)

$sum1 \leftarrow 0$; $sum2 \leftarrow 0$;

// Compute $sum1$ (lower half) and $sum2$ (upper half) in parallel.

finish{

async{

 // Task T2

for $i \leftarrow 0$ **to** $X.length/2 - 1$ **do**

$sum1 \leftarrow sum1 + X[i]$;

 };

async{

 // Task T3

for $i \leftarrow X.length/2$ **to** $X.length - 1$ **do**

$sum2 \leftarrow sum2 + X[i]$;

 };

};

// Task T1 waits for Tasks T2 and T3 to complete

// Continuation of Task T1

$sum \leftarrow sum1 + sum2$;

return sum ;



Two-way Parallel Array Sum using async & finish constructs

Algorithm 2: Two-way Parallel ArraySum

Input: Array of numbers, X .

Output: $sum = \text{sum of elements in array } X$.

// Start of Task T1 (main program)

$sum1 \leftarrow 0$; $sum2 \leftarrow 0$;

// Compute $sum1$ (lower half) and $sum2$ (upper half) in parallel.

finish{

async{

 // Task T2

for $i \leftarrow 0$ **to** $X.length/2 - 1$ **do**

$sum1 \leftarrow sum1 + X[i]$;

 };

async{

 // Task T3

for $i \leftarrow X.length/2$ **to** $X.length - 1$ **do**

$sum2 \leftarrow sum2 + X[i]$;

 };

};

foo();

$sum \leftarrow sum1 + sum2$;

return sum ;



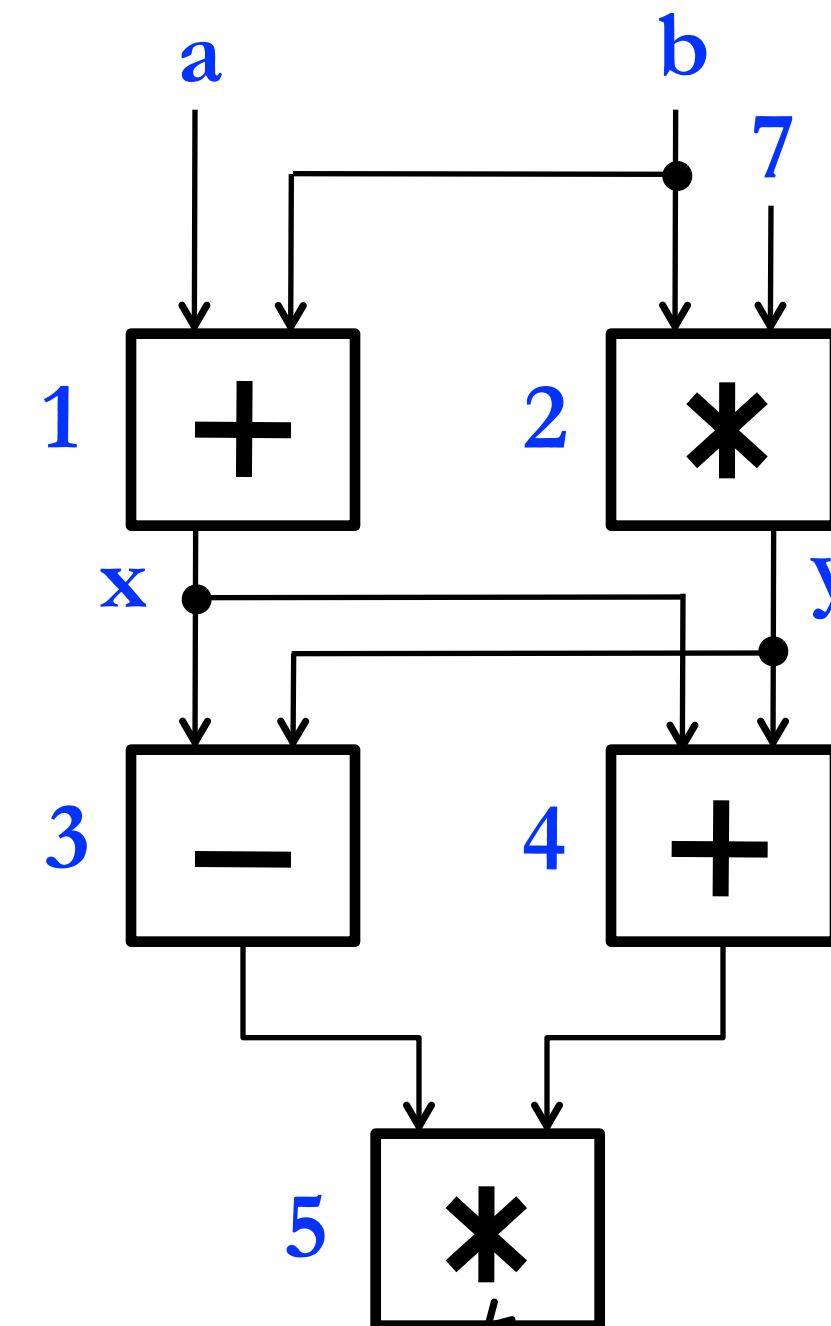
Two-way Parallel Array Sum using futures

```
// Parent Task T1 (main program)
// Compute sum1 (lower half) & sum2 (upper half) in parallel
var sum1 = future(() -> { // Future Task T2
    int sum = 0;
    for (int i = 0; i < X.length / 2; i++) sum += X[i];
    return sum;
});
var sum2 = future(() -> { // Future Task T3
    int sum = 0;
    for (int i = X.length / 2; i < X.length; i++) sum += X[i];
    return sum;
});
foo();
int total = sum1.get() + sum2.get();
```



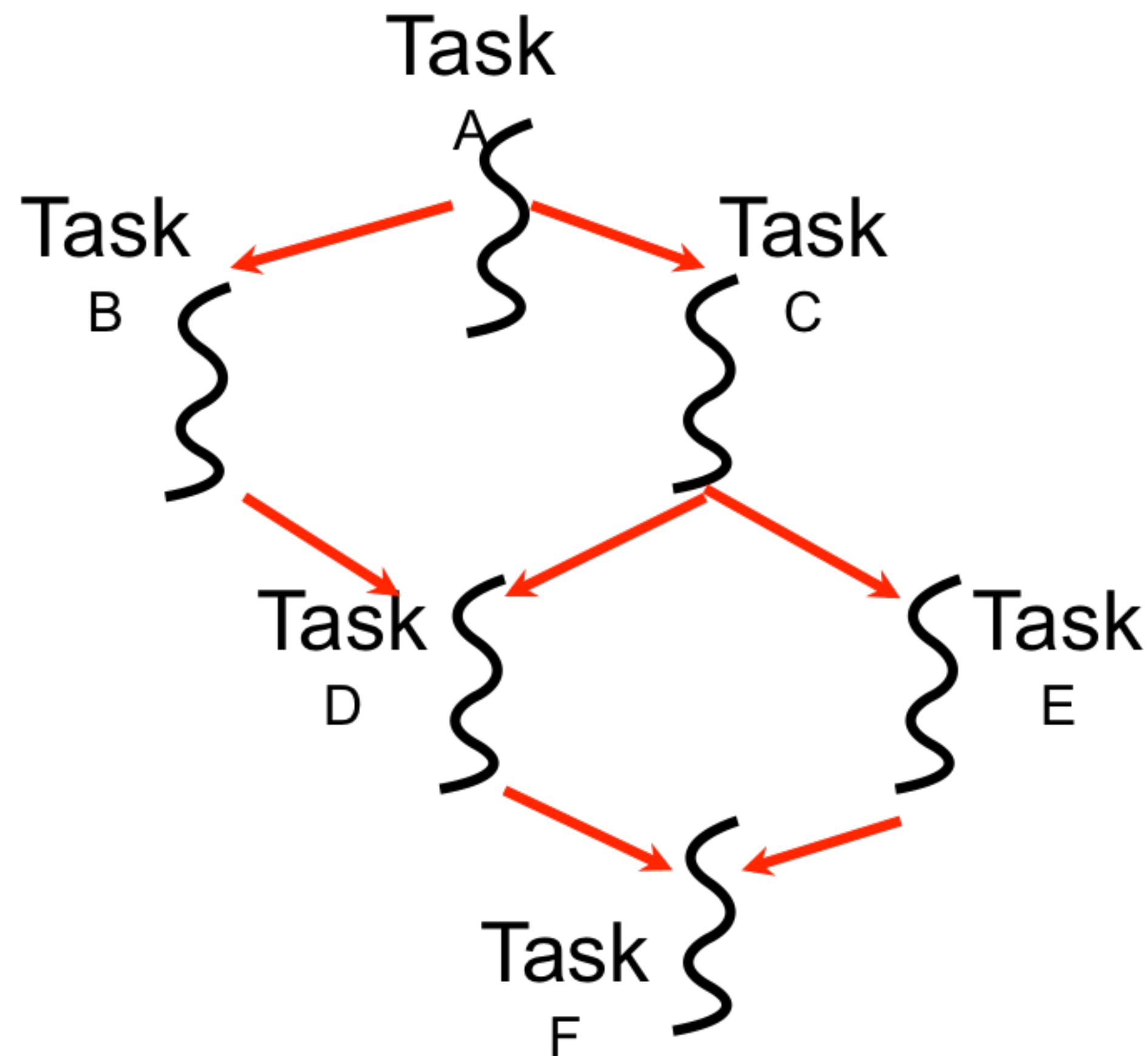
Example instruction sequence and its dataflow graph

```
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)

```
final 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”)
- Alternatively, you can pass a list to `asyncAwait`
- 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

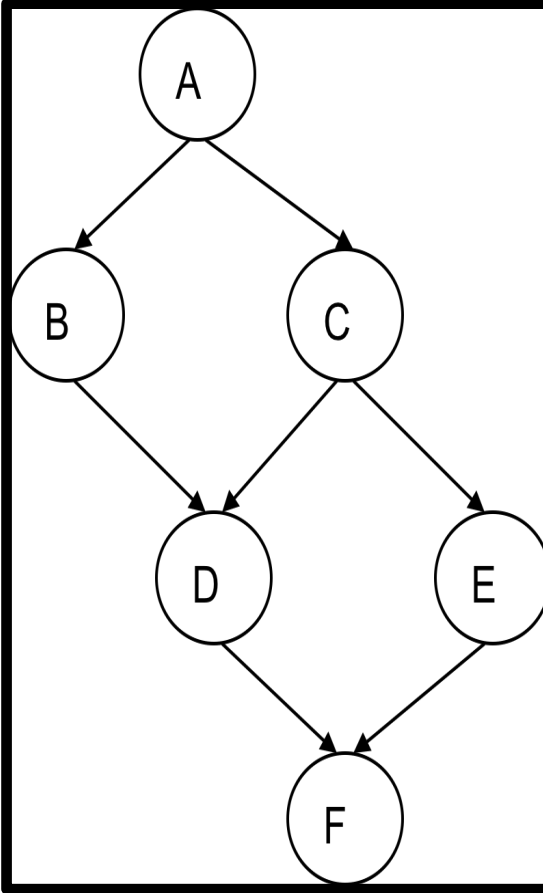
`ddfA.safeGet()`

- Doesn't throw an exception
 - 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.   final HjDataDrivenFuture<Void> ddfA = newDataDrivenFuture();
3.   final HjDataDrivenFuture<Void> ddfB = newDataDrivenFuture();
4.   final HjDataDrivenFuture<Void> ddfC = newDataDrivenFuture();
5.   final HjDataDrivenFuture<Void> ddfD = newDataDrivenFuture();
6.   final 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

```
final HJDataDrivenFuture<Object> left = newDataDrivenFuture();  
final HJDataDrivenFuture<Object> right = newDataDrivenFuture();  
finish {  
    asyncAwait ( left ) right.put(rightBuilder()); // Task1  
    asyncAwait ( 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. final HjFuture<T> f = future(() -> { return g(); });
2. S1
3. async(() -> {
4.     ... = f.get(); // blocks if needed
5.     S2;
6.     S3;
7. });
```

- DDF version

```
1. final HjDataDrivenFuture<T> f = newDataDrivenFuture();
2. async(() -> { f.put(g()) });
3. S1
4. asyncAwait(f, () -> {
5.     ... = f.safeGet(); // does not need to 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. final HjDataDrivenFuture<Object> left = newDataDrivenFuture();
2. final HjDataDrivenFuture<Object> right = newDataDrivenFuture();
3. finish(() -> {
4.     asyncAwait(left, () -> {
5.         right.put(rightWriter()); });
6.     asyncAwait(right, () -> {
7.         left.put(leftWriter()); });
8. });
```

Can you think of a deadlock example or explain why it can't happen?

- 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

- Regular office hour schedule can be found at Office Hours link on course web site
- Hw #1 is due Friday, Feb. 4th by 11:59pm
- Lab #4 is this week

