

# COMP 322: Fundamentals of Parallel Programming

## Lecture 9: Ideal Parallelism, Data-Driven Tasks

Mack Joyner  
mjoyner@rice.edu

<http://comp322.rice.edu>



# Complexity Measures for Computation Graphs

## Define

- $\text{TIME}(N)$  = execution time of node  $N$
- $\text{WORK}(G)$  = sum of  $\text{TIME}(N)$ , for all nodes  $N$  in CG  $G$ 
  - $\text{WORK}(G)$  is the total work to be performed in  $G$
- $\text{CPL}(G)$  = length of a longest path in CG  $G$ , when adding up execution times of all nodes in the path
  - Such paths are called *critical paths*
  - $\text{CPL}(G)$  is the length of these paths (critical path length, also referred to as the *span* of the graph)
  - $\text{CPL}(G)$  is also the shortest possible execution time for the computation graph





# Ideal Parallelism

- Define **ideal parallelism** of Computation G Graph as the ratio,  $WORK(G)/CPL(G)$
- Ideal Parallelism only depends on the computation graph, and is the speedup that you can obtain with an unbounded number of processors

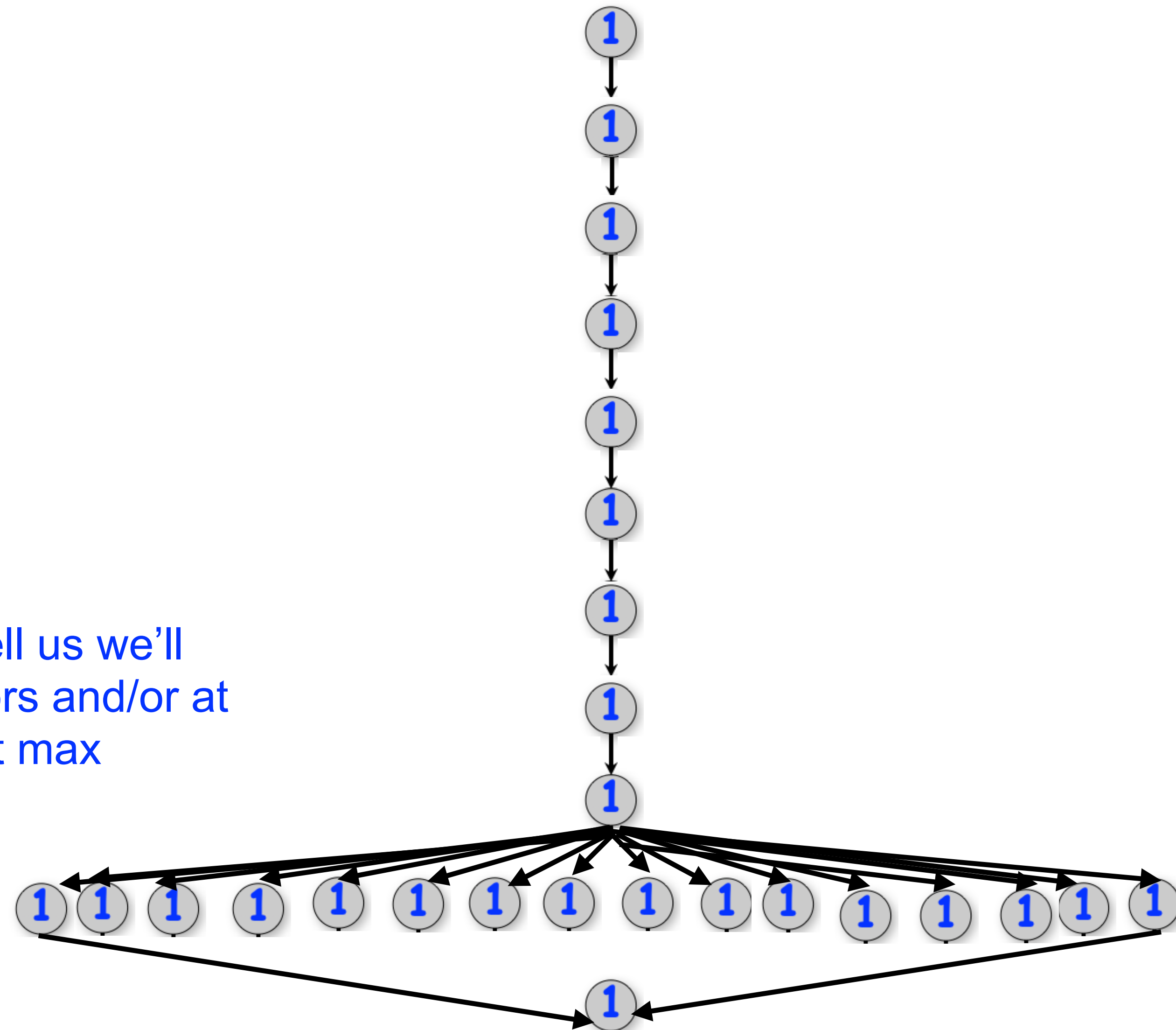
## Example:

$$WORK(G) = 26$$

$$CPL(G) = 11$$

$$\text{Ideal Parallelism} = WORK(G)/CPL(G) = 26/11 \sim 2.36$$

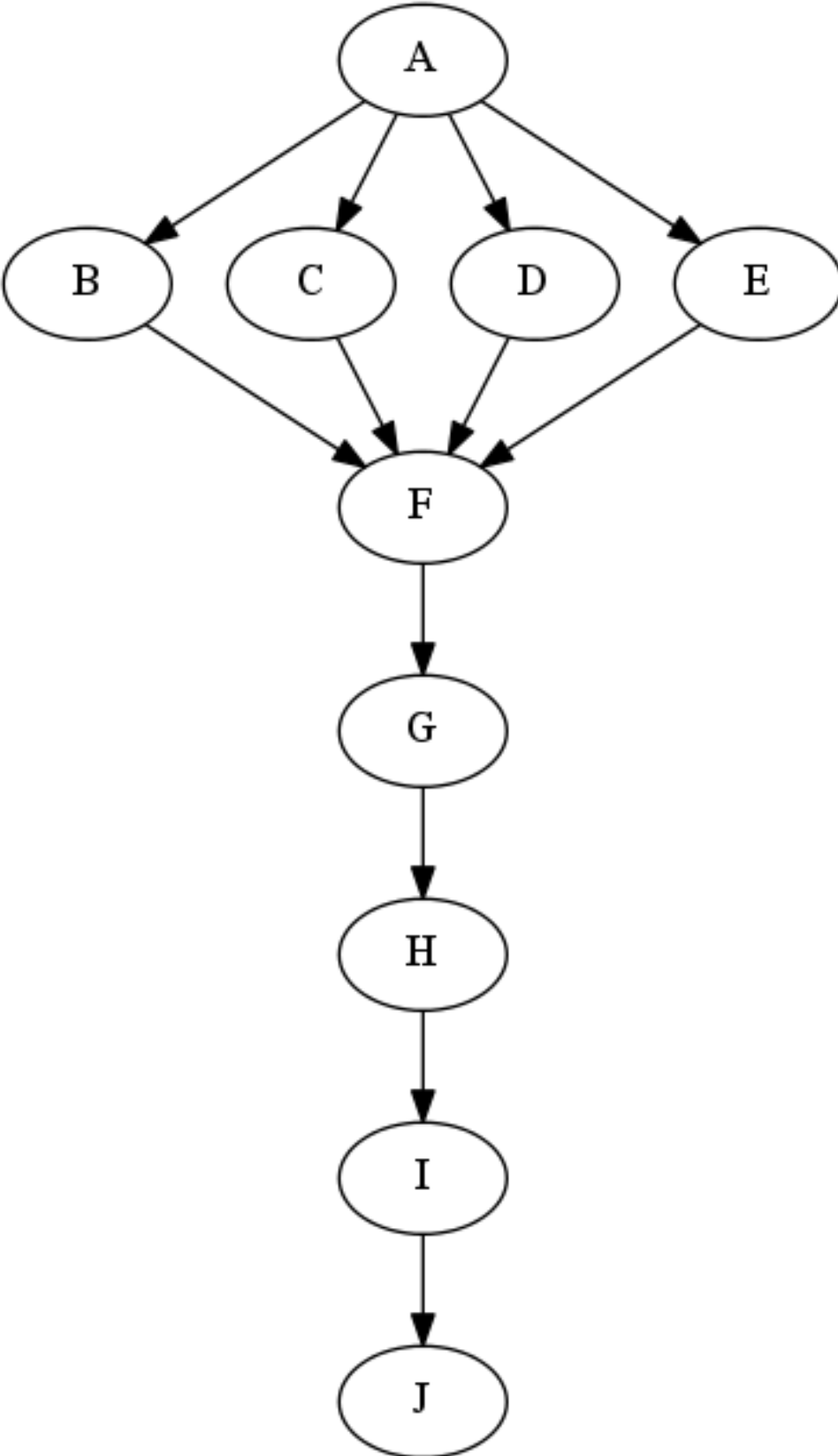
Does ideal parallelism tell us we'll need at least x processors and/or at most y processors to get max speedup?



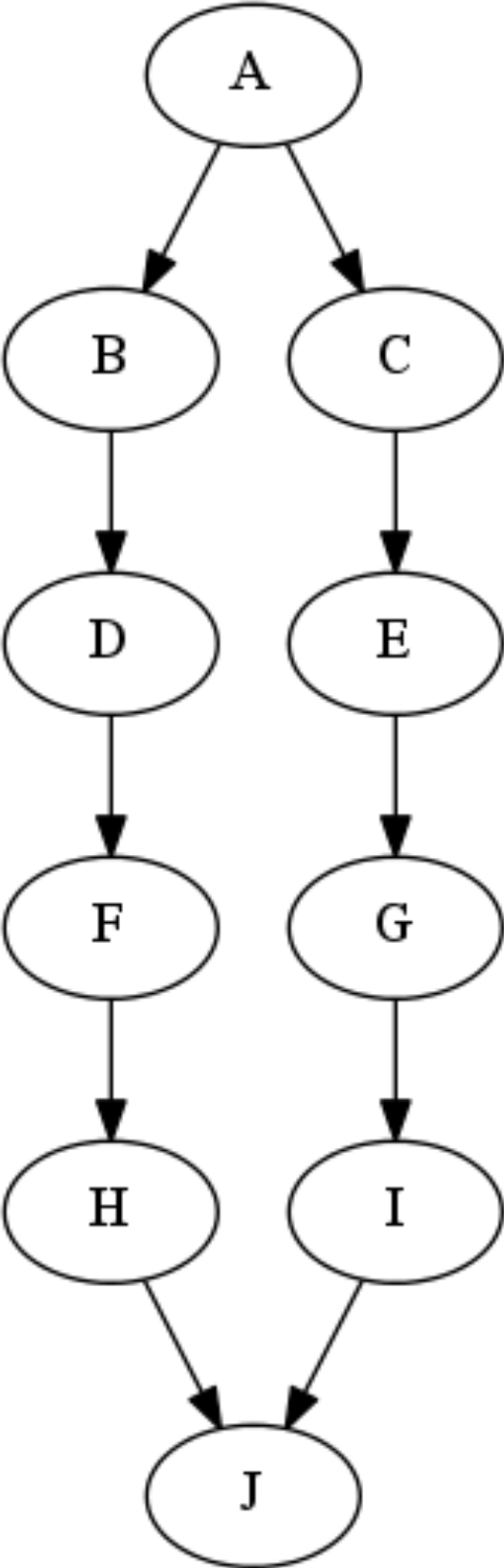
# Which Computation Graph has more ideal parallelism?

Assume that all nodes have TIME = 1, so WORK = 10 for both graphs.

Computation Graph 1

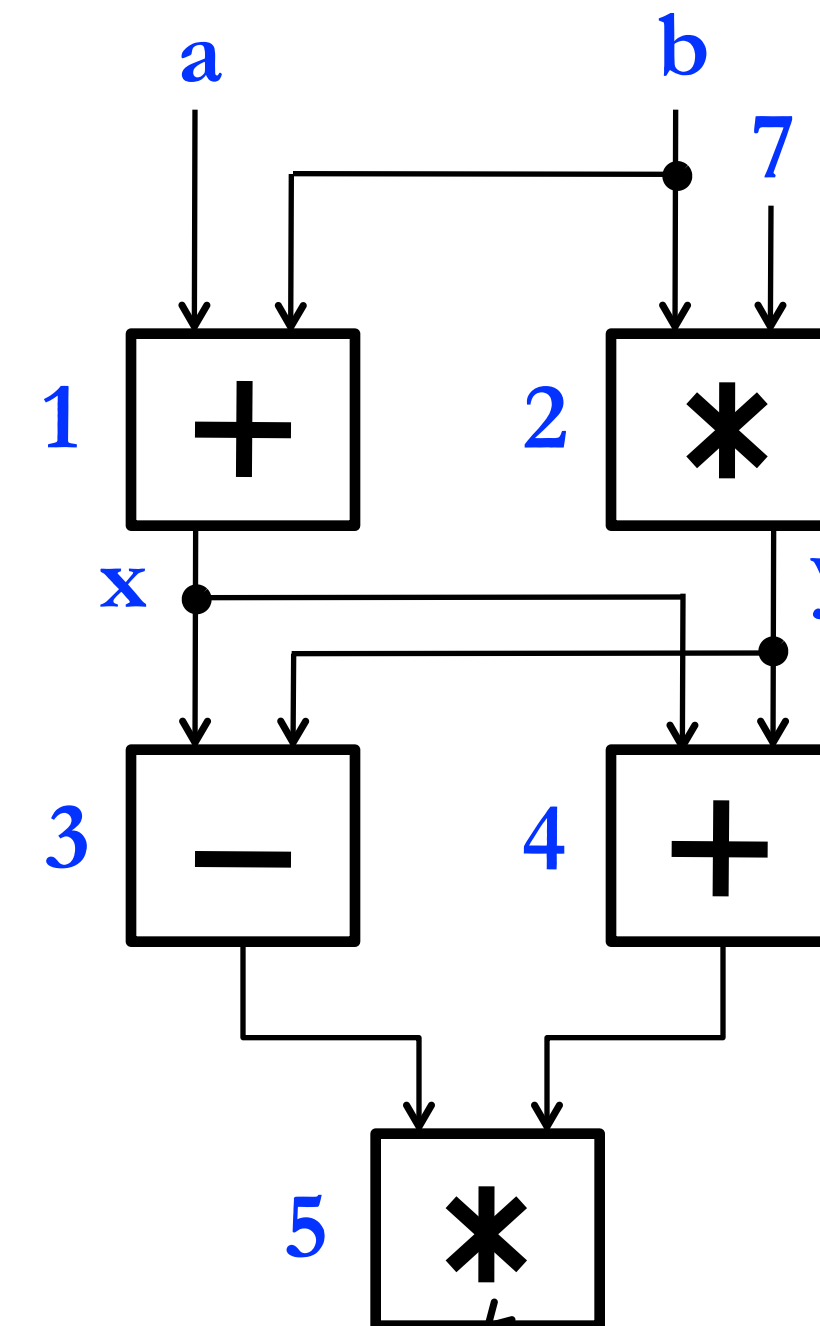


Computation Graph 2



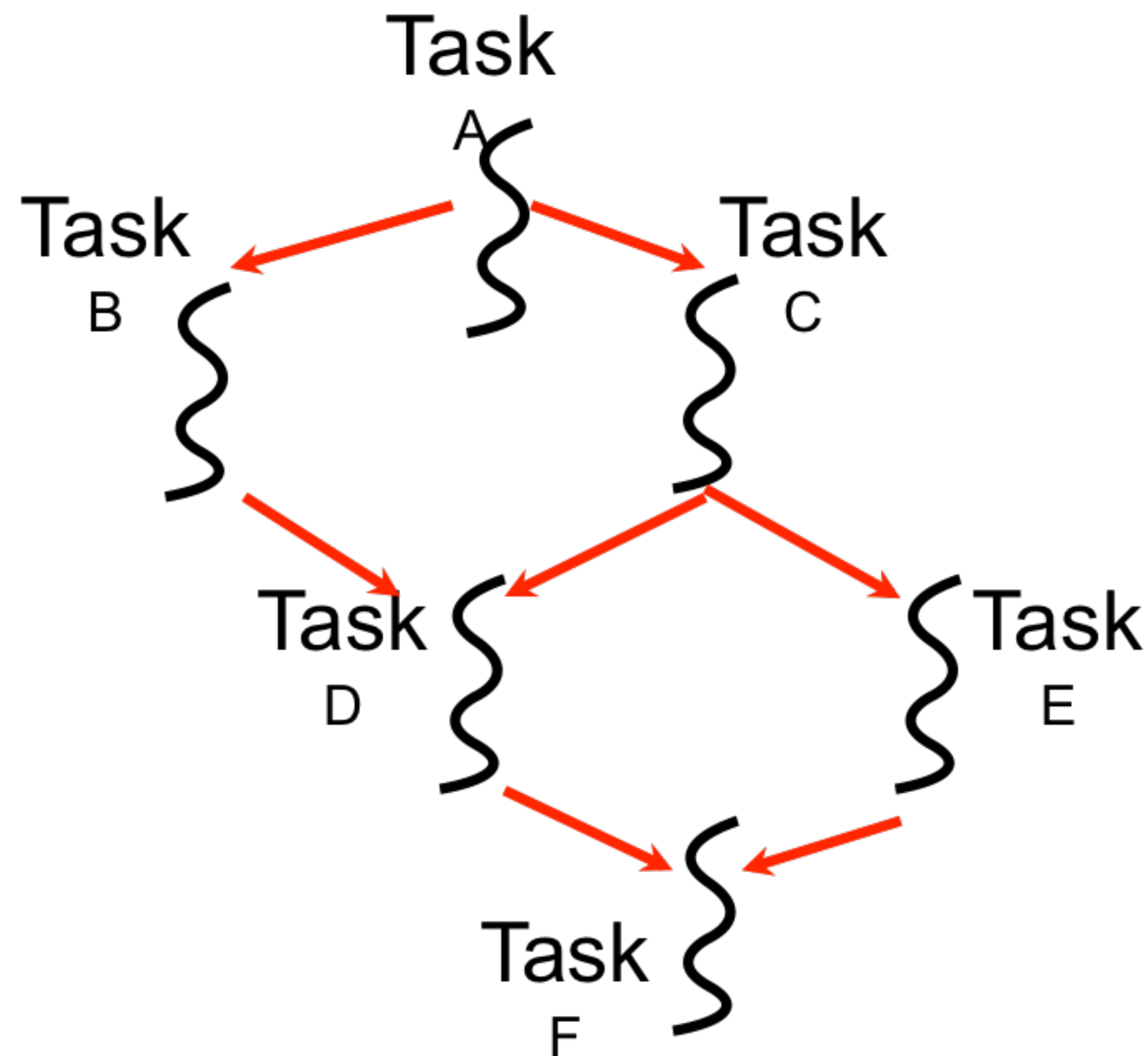
# Example instruction sequence and its dataflow graph

```
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



# 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.



# Implementing Future Tasks using DDTs

- Future version

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

- DDT version

```
1. var 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 DDTs

- Consumer task blocks on `get()` for each future that it reads, whereas `async-await` **does not start execution** till all futures are available
- Future tasks cannot deadlock, but it is possible for a DDT to block indefinitely (“deadlock”) if one of its input futures 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. var left = newDataDrivenFuture();
2. var 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 with futures or explain why it can't happen?

