COMP 322: Fundamentals of Parallel Programming

Lecture 8: Computation Graphs, Ideal Parallelism

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

http://comp322.rice.edu

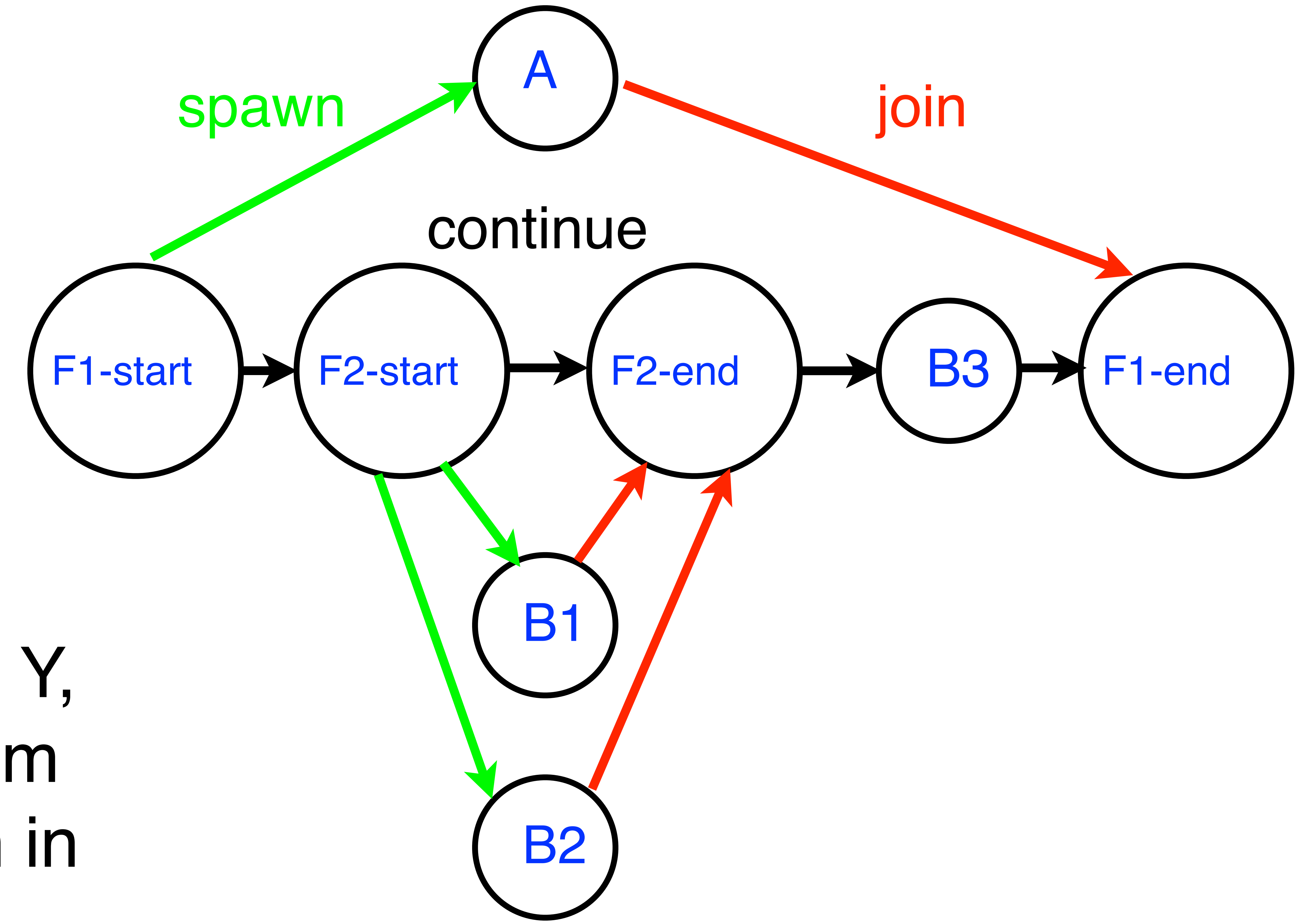
Computation Graphs

- A Computation Graph (CG) captures the dynamic execution of a parallel program, for a specific input
- CG nodes are "steps" in the program's execution
  - A step is a sequential subcomputation without any spawned, begin-finish or end-finish operations
- CG edges represent ordering constraints
  - "Continue" edges define sequencing of steps within a task
  - "Spawn" edges connect parent tasks to child spawned tasks
  - "Join" edges connect the end of each spawned task to its IEF's end-must finish operations
- All computation graphs must be acyclic
  - It is not possible for a node to depend on itself
- Computation graphs are examples of "directed acyclic graphs" (DAGs)

Which statements can potentially be executed in parallel with each other?

```
1.  must finish { // F1
2.     spawn { A; }
3.     must finish { // F2
4.        spawn { B1; }
5.        spawn { B2; }
6.     } // F2
7.     B3;
8.  } // F1
```

Computation Graph



Key idea: If two statements, X and Y, have *no path of directed edges* from one to the other, then they can run in parallel with each other.

Computational Graph Exercise

Assume you have 2 washers and 2 dryers. Assume there's 0 cost to spawn a task.

Place "must finish" blocks and "spawn" blocks around the following tasks:

1. Run load 1 in washer (LW1)
2. Run load 2 in washer (LW2)
3. Run load 1 in dryer (LD1)
4. Run load 2 in dryer (LD2)

Computational Graph Exercise (Solution #1)

Assume you have 2 washers and 2 dryers. Assume there's 0 cost to spawn a task.

Place "must finish" blocks and "spawn" around the following tasks:

1. must finish { // F1
2.   spawn { Run load 1 in washer (LW1) }
3.   spawn { Run load 2 in washer (LW2) }
4.} // F1
5. spawn { Run load 1 in dryer (LD1) }
6. spawn { Run load 2 in dryer (LD2) }

Computational Graph Exercise (Solution #2)

Assume you have 2 washers and 2 dryers. Assume there's 0 cost to spawn a task.

Place "must finish" blocks and "spawn" around the following tasks:

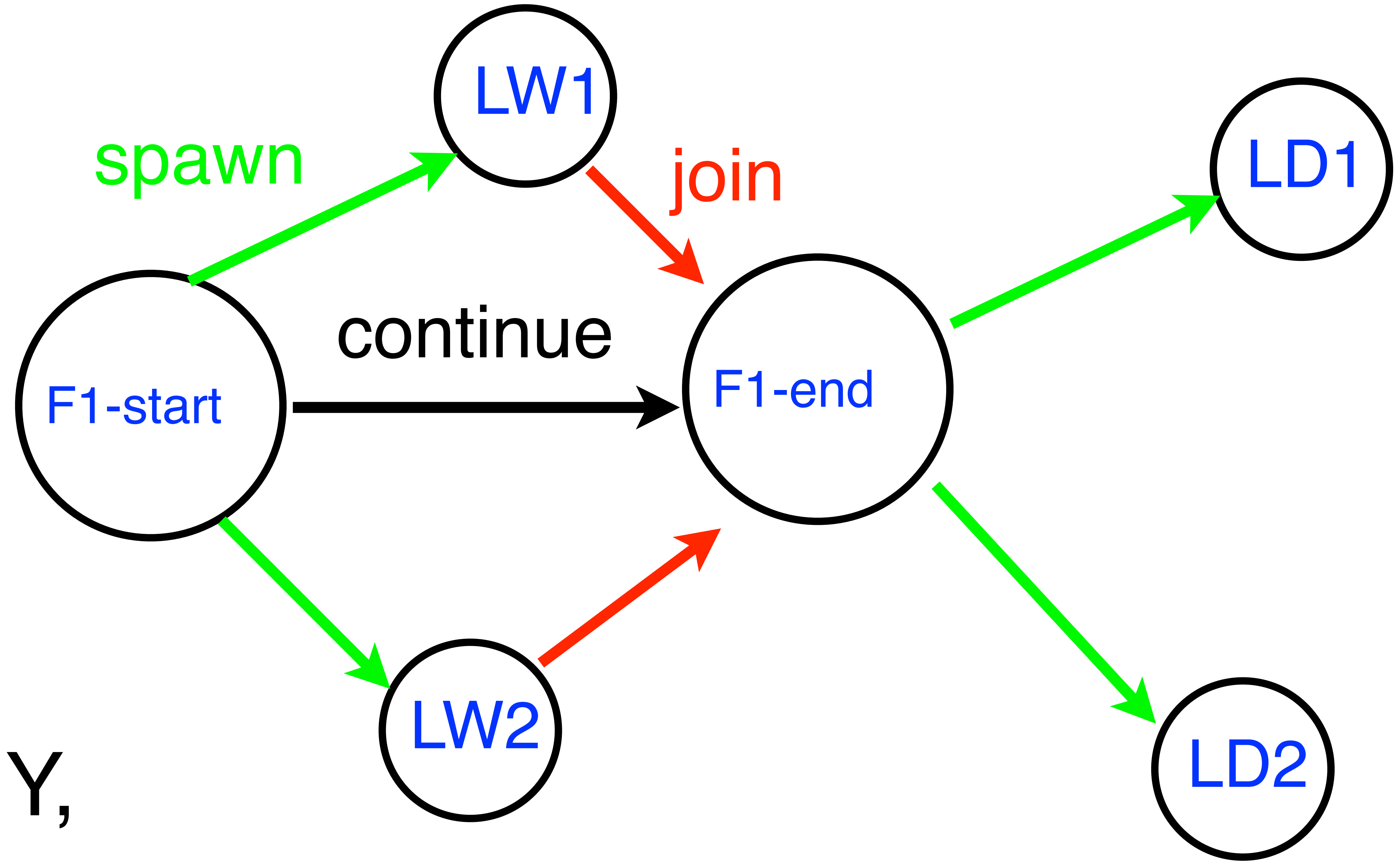
1. must finish { // F1
2. spawn { Run load 1 in washer (LW1); Run load 1 in dryer (LD1) }
3. spawn { Run load 2 in washer (LW2); Run load 2 in dryer (LD2) }
4. } // F1

Draw Computation Graph for Solution

Draw Computation Graph for Solution #1

1. must finish { // F1
2.   spawn LW1;
3.   spawn LW2;
4. } // F1
5. spawn LD1;
6. spawn LD2;

Computation Graph

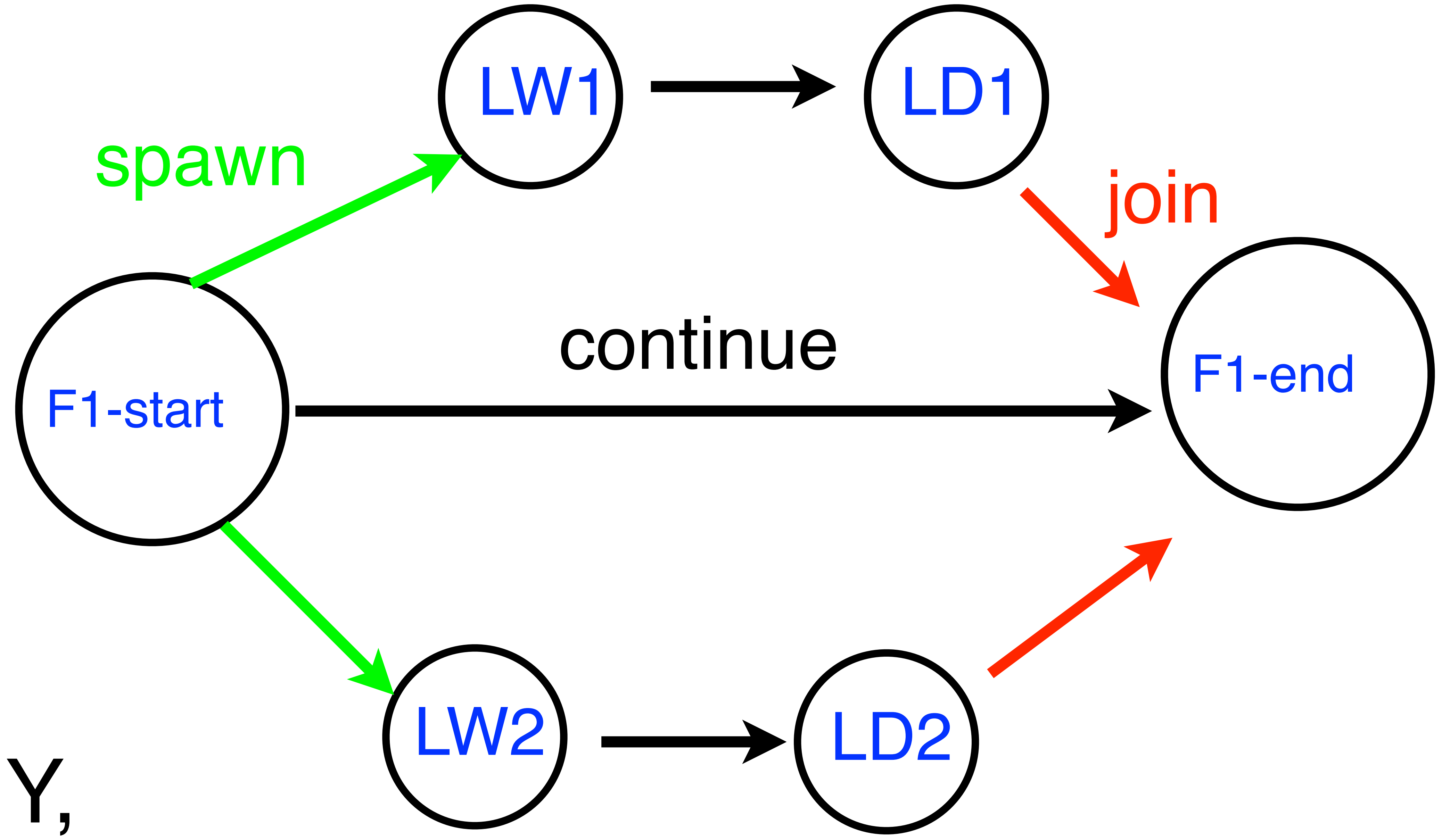


<u>Key idea</u>: If two statements, X and Y, have *no path of directed edges* from one to the other, then they can run in parallel with each other.

Draw Computation Graph for Solution #2

```
1. must finish { // F1
2.     spawn { LW1; LD1 }
3.     spawn { LW2; LD2 }
4.} // F1
```

Computation Graph



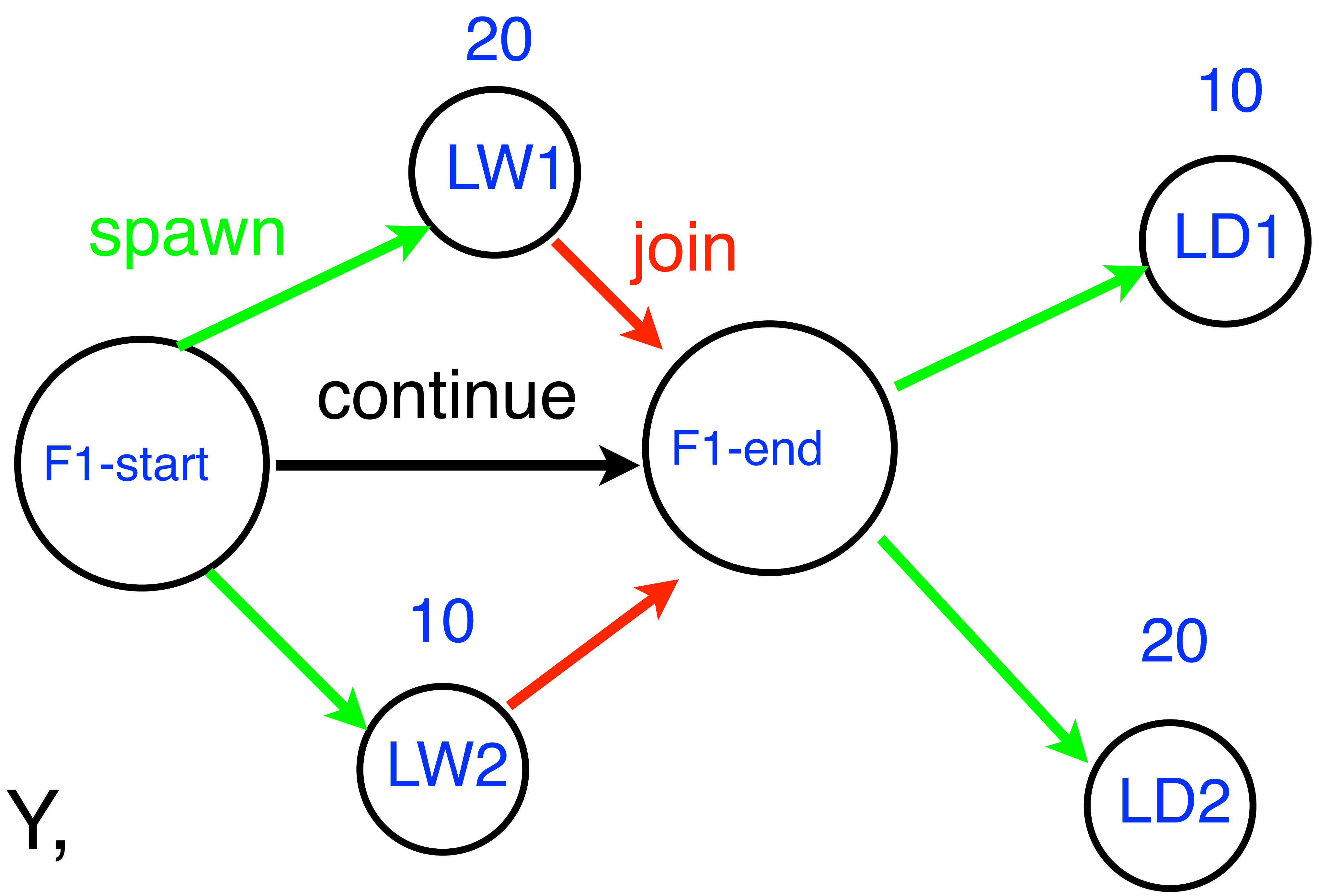
Key idea: If two statements, X and Y, have *no path of directed edges* from one to the other, then they can run in parallel with each other.

Which solution is better?

Draw Computation Graph for Solution #1

1. must finish { // F1
2.   spawn LW1;
3.   spawn LW2;
4.} // F1
5. spawn LD1;
6. spawn LD2;

Computation Graph

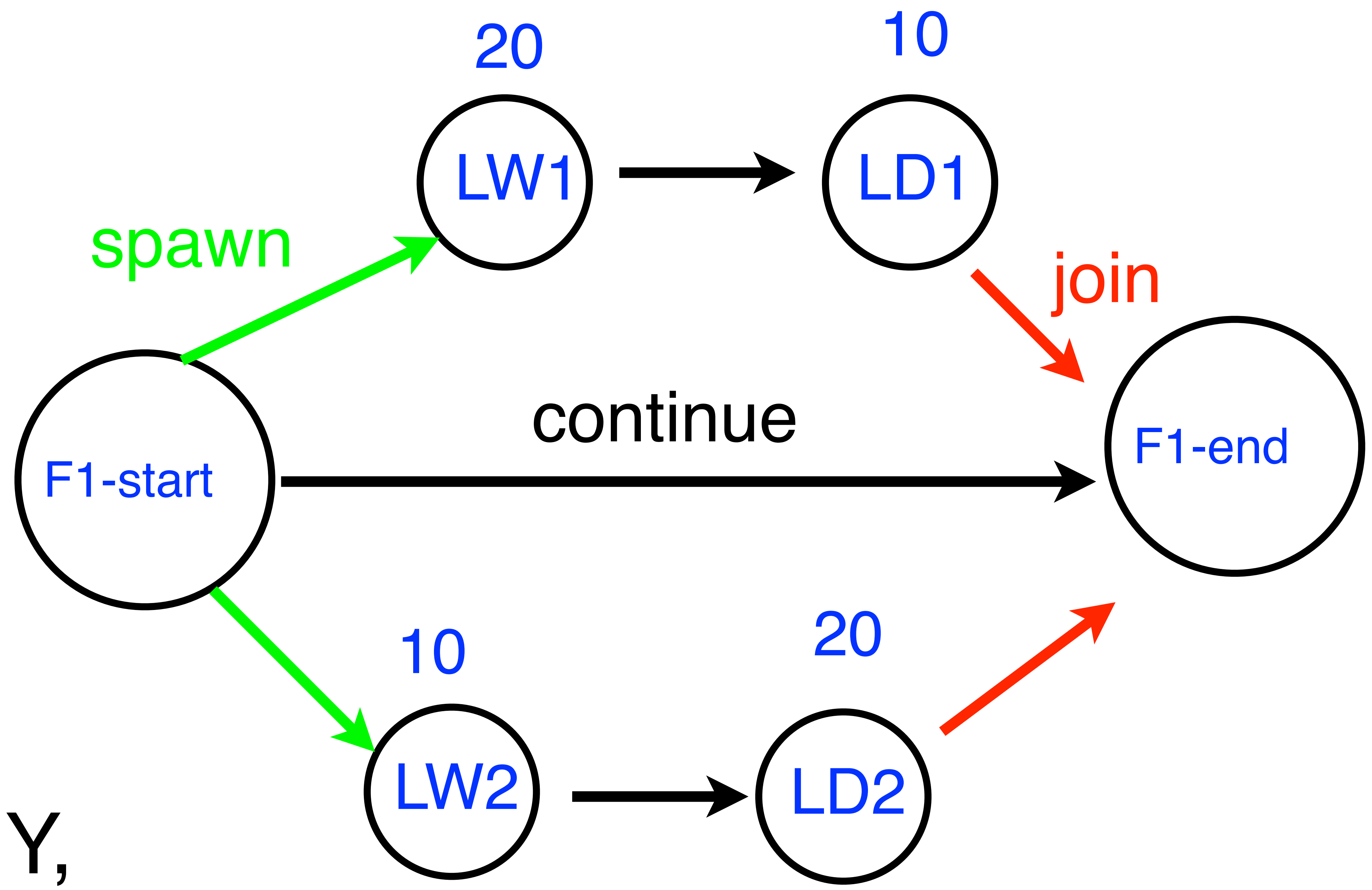


Key idea: If two statements, X and Y, have *no path of directed edges* from one to the other, then they can run in parallel with each other.

Draw Computation Graph for Solution #2

1. must finish { // F1
2. spawn { LW1; LD1 }
3. spawn { LW2; LD2 }
4. } // F1

Computation Graph



Key idea: If two statements, X and Y, have *no path of directed edges* from one to the other, then they can run in parallel with each other.

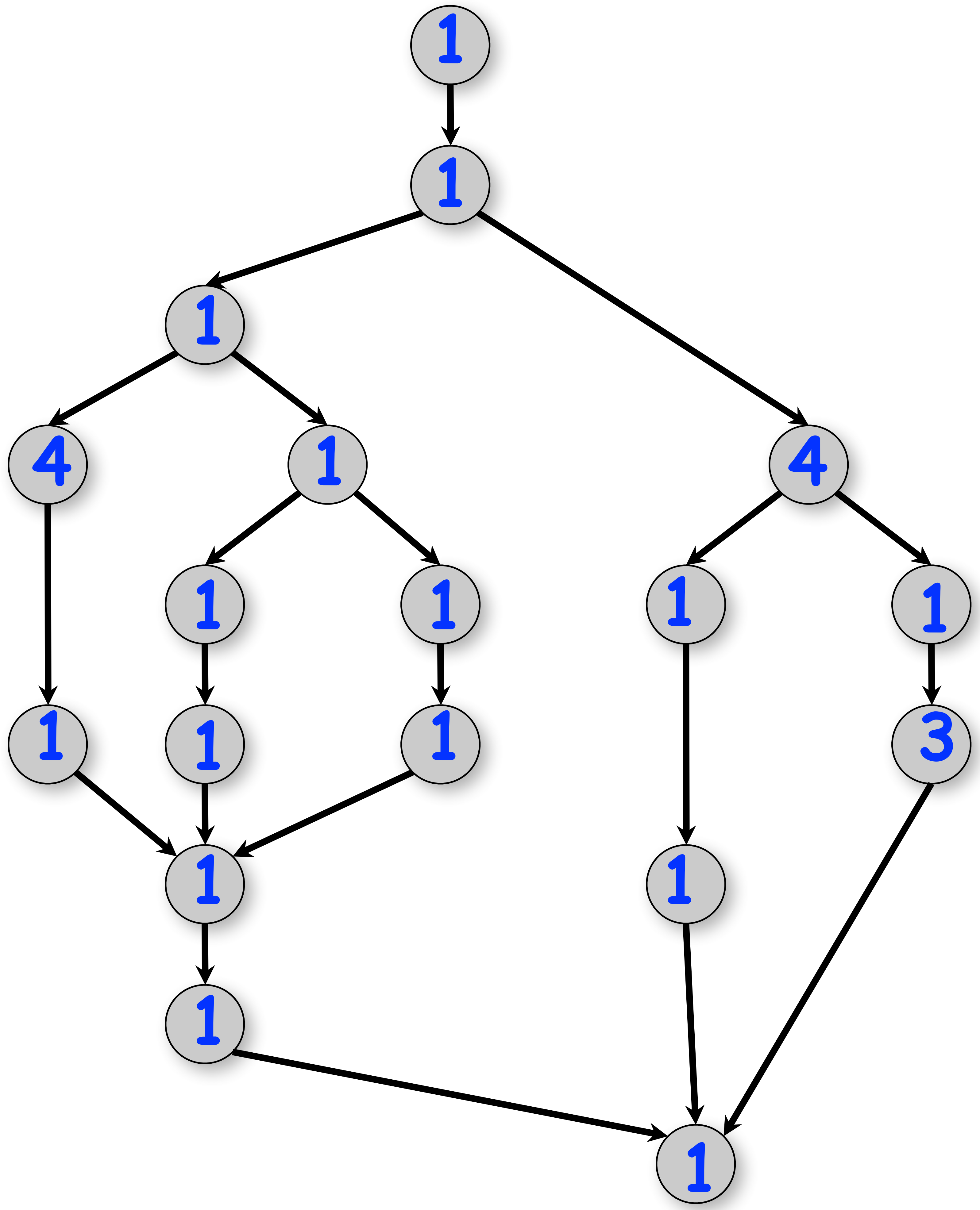
Complexity Measures for Computation Graphs

Define

- TIME(N) = execution time of node N
- WORK(G) = sum of TIME(N), for all nodes N in CG G
  - WORK(G) is the total work to be performed in G
- 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*
  - CPL(G) is the length of these paths (critical path length, also referred to as the *span* of the graph)
  - 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



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

Example:

WORK(G) = 26

CPL(G) = 11

Ideal Parallelism = WORK(G)/CPL(G) = 26/11 ~ 2.36

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

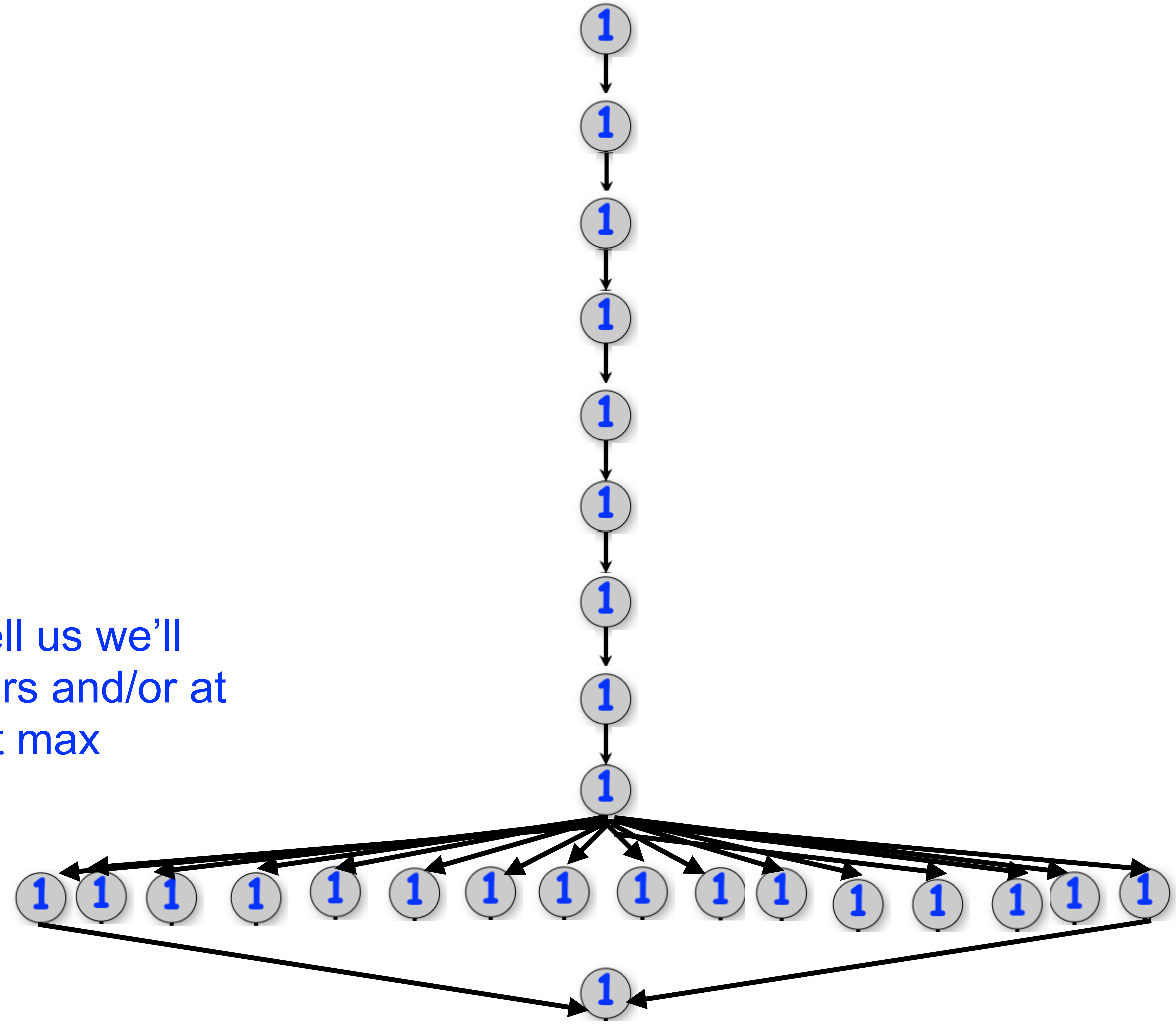
Does ideal parallelism tell us we'll need at least x processors and/or at most y processors to get max speedup?

<u>Example</u>:

WORK(G) = 26

CPL(G) = 11

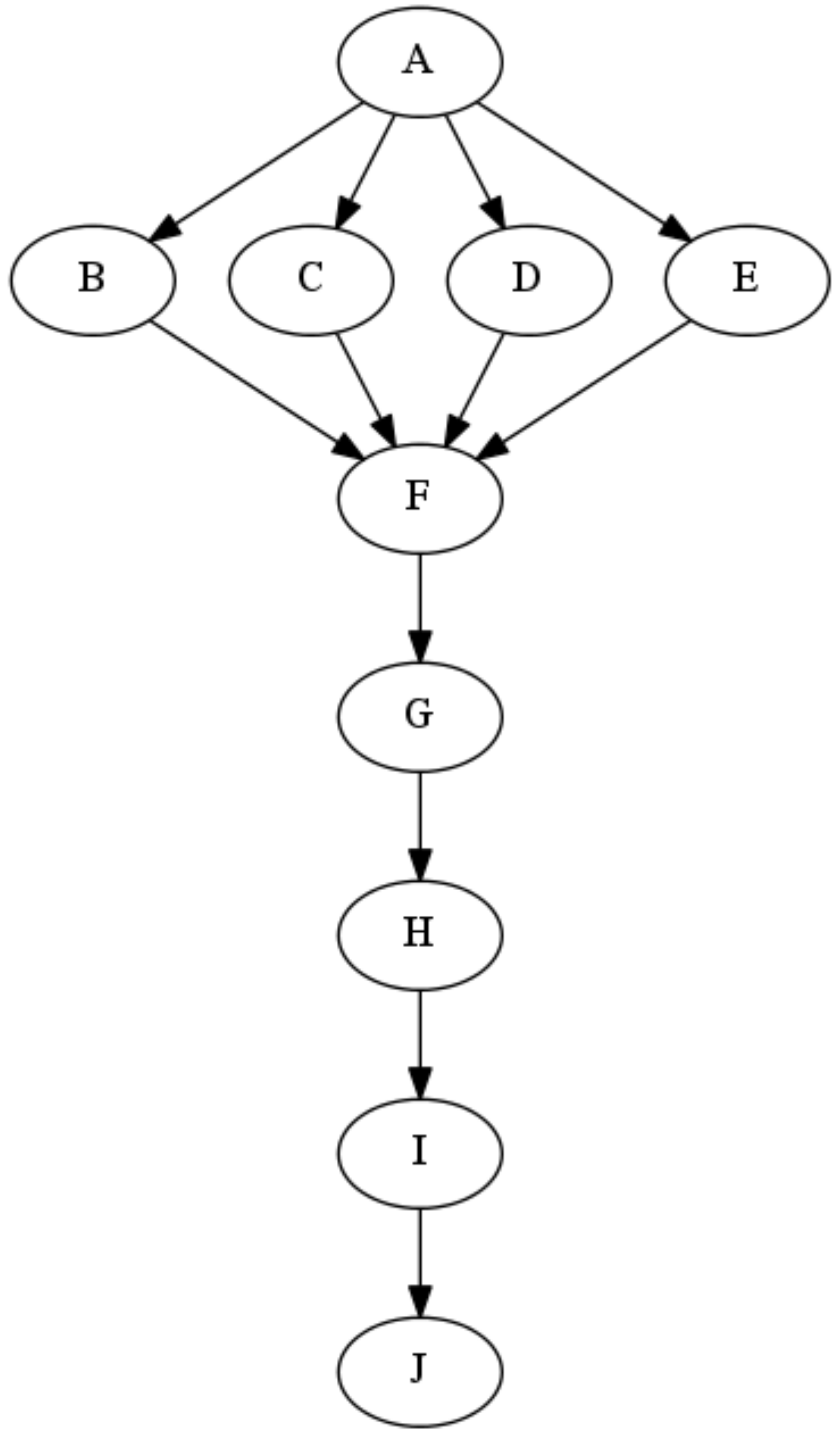
Ideal Parallelism = WORK(G)/CPL(G) = 26/11 ~ 2.36



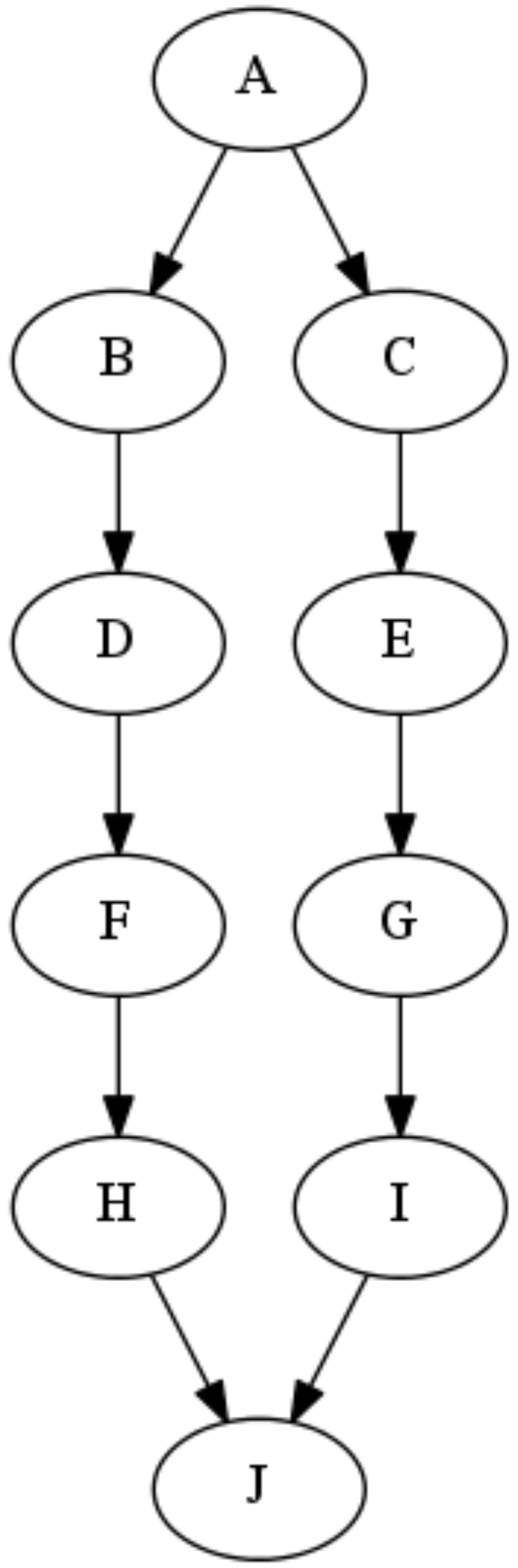
Which Computation Graph has more ideal parallelism?

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

Computation Graph 1



Computation Graph 2



Announcements & Reminders

- IMPORTANT:
  - Watch videos for topics 1.1, 4.5 for next lecture
- HW 1 is due on Friday, Feb 4th
- Quiz 2 is due on Sunday, Feb 6th
- Worksheets due same day by 11:59pm for full credit, before next class for partial credit (0.5)
- Module 1 handout is available
- See course web site for syllabus, work assignments, due dates, …
  - http://comp322.rice.edu