COMP 322: Fundamentals of Parallel Programming

Lecture 2: Computation Graphs, Ideal Parallelism

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu

Async and Finish Statements for Task Creation and Termination

async S

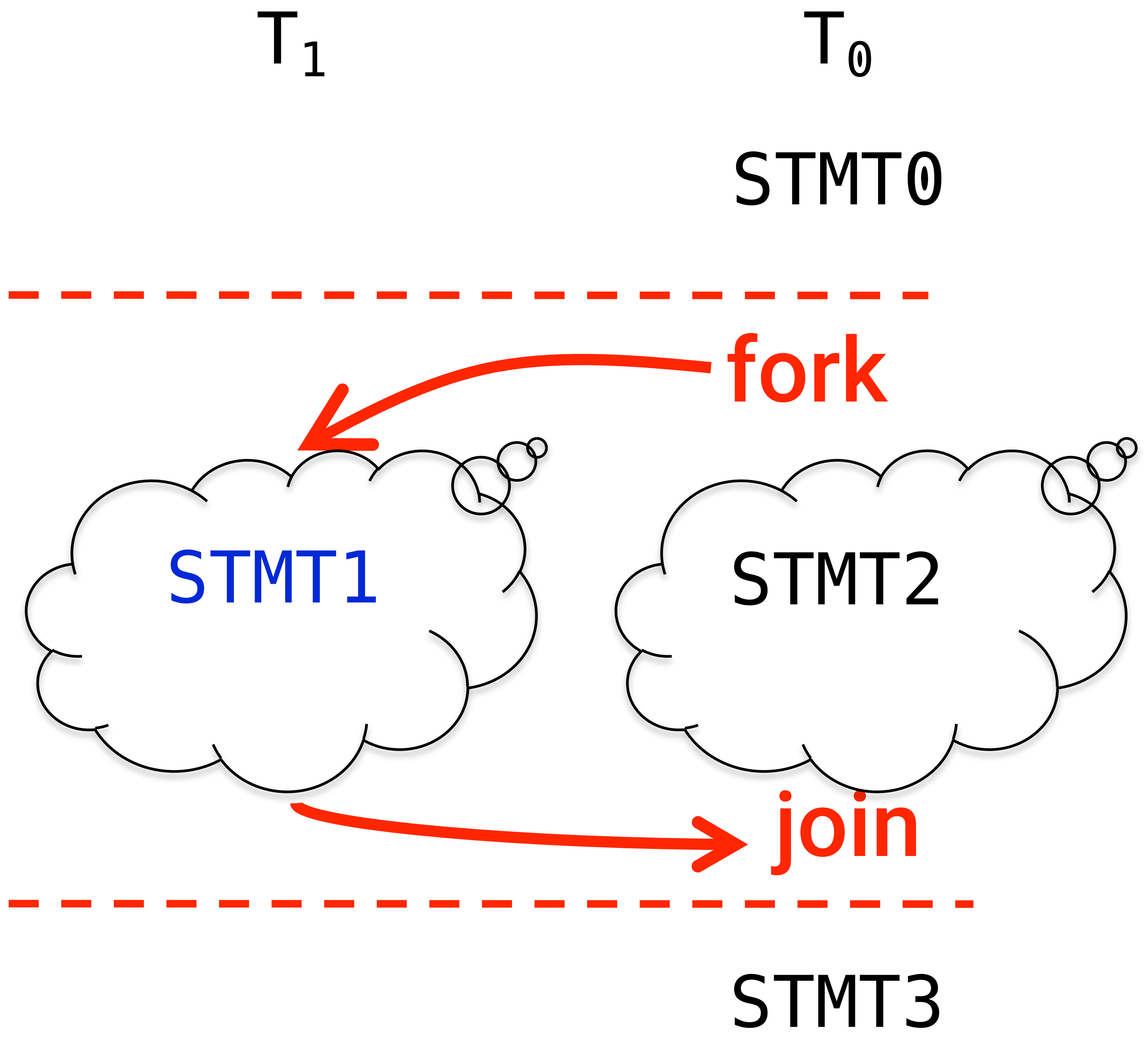
- Creates a new child task that executes statement S

finish S

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

```
// 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
```



One possible solution to Worksheet 1 (without statement reordering)

```
finish {
  async { Watch COMP 322 video for topic 1.2 by 1pm on Wednesday
          Watch COMP 322 video for topic 1.3 by 1pm on Wednesday
         }
  async Make your bed
  async { Clean out your fridge
          Buy food supplies and store them in fridge }
  finish { async Run load 1 in washer
           Run load 2 in washer }
  async Run load 1 in dryer
  async Run load 2 in dryer
  async Call your family
}
Post on Facebook that you're done with all your tasks!
```

Another possible solution to Worksheet 1 (with statement reordering)

```
finish {
  async Call your family
  async Make your bed
  async { Clean out your fridge
          Buy food supplies and store them in fridge }
  async { Run load 1 in washer
          Run load 1 in dryer }
  async { Run load 2 in washer
          Run load 2 in dryer }
  Watch COMP 322 video for topic 1.2 by 1pm on Wednesday
  Watch COMP 322 video for topic 1.3 by 1pm on Wednesday
}
Post on Facebook that you're done with all your tasks!
```

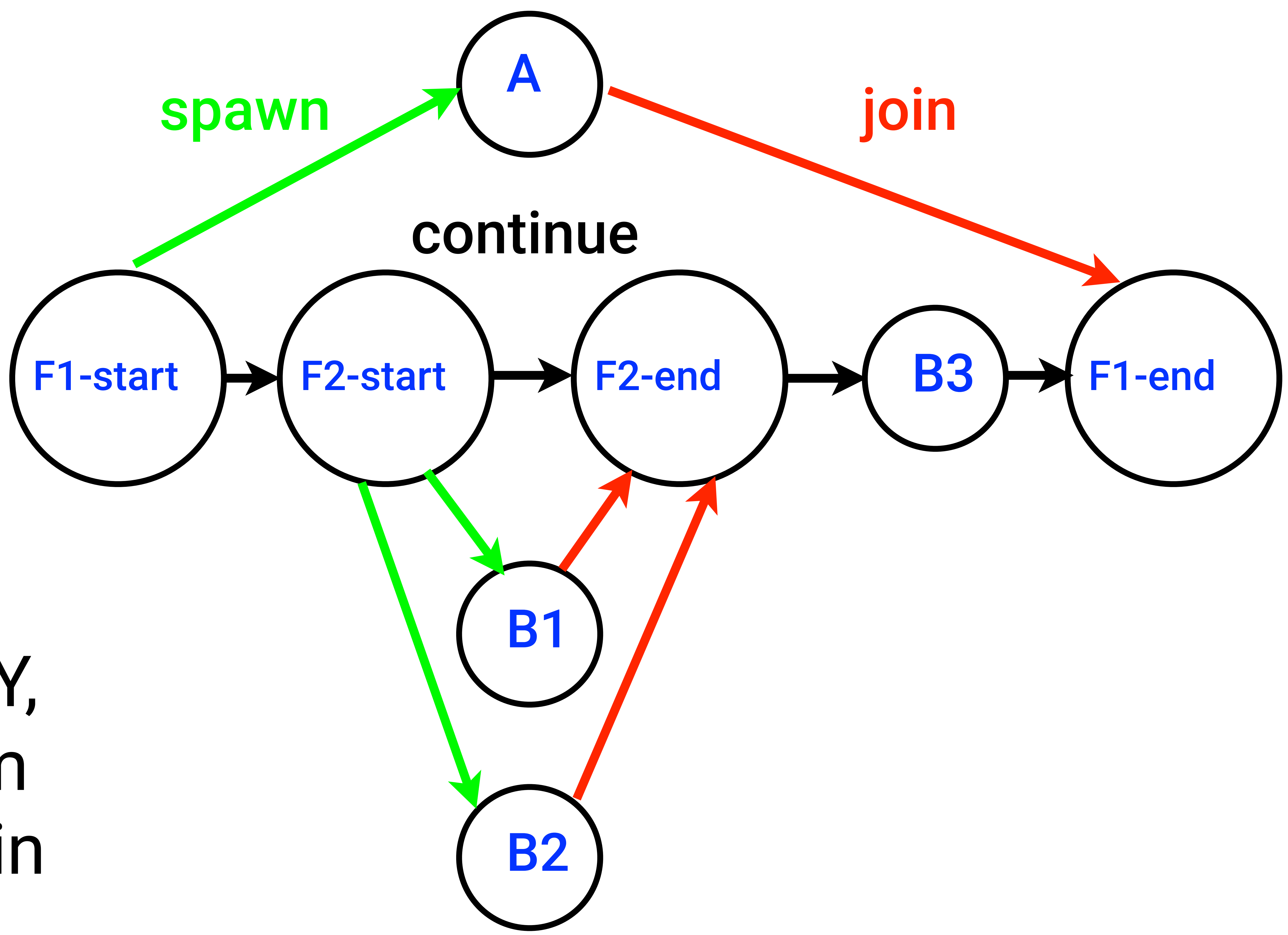
Computation Graphs

- 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 async, 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 async tasks
  - "Join" edges connect the end of each async task to its IEF's end-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?

```
finish { // F1
  async A;
  finish { // F2
    async B1;
    async B2;
  } // F2
  B3;
} // 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

```
finish { (F1)
  async  (WV) { Watch COMP 322 video for topic 1.2 by 1pm on Wednesday
                Watch COMP 322 video for topic 1.3 by 1pm on Wednesday
              }
  async (MB)   Make your bed
  async (SF) { Clean out your fridge
               Buy food supplies and store them in fridge }
  finish (F2) { async  Run load 1 in washer (LW1)
                Run load 2 in washer (LW2) }
  async Run load 1 in dryer (LD1)
  async Run load 2 in dryer (LD2)
  async Call your family (CF)
}
Post on Facebook that you're done with all your tasks! (PF)
```

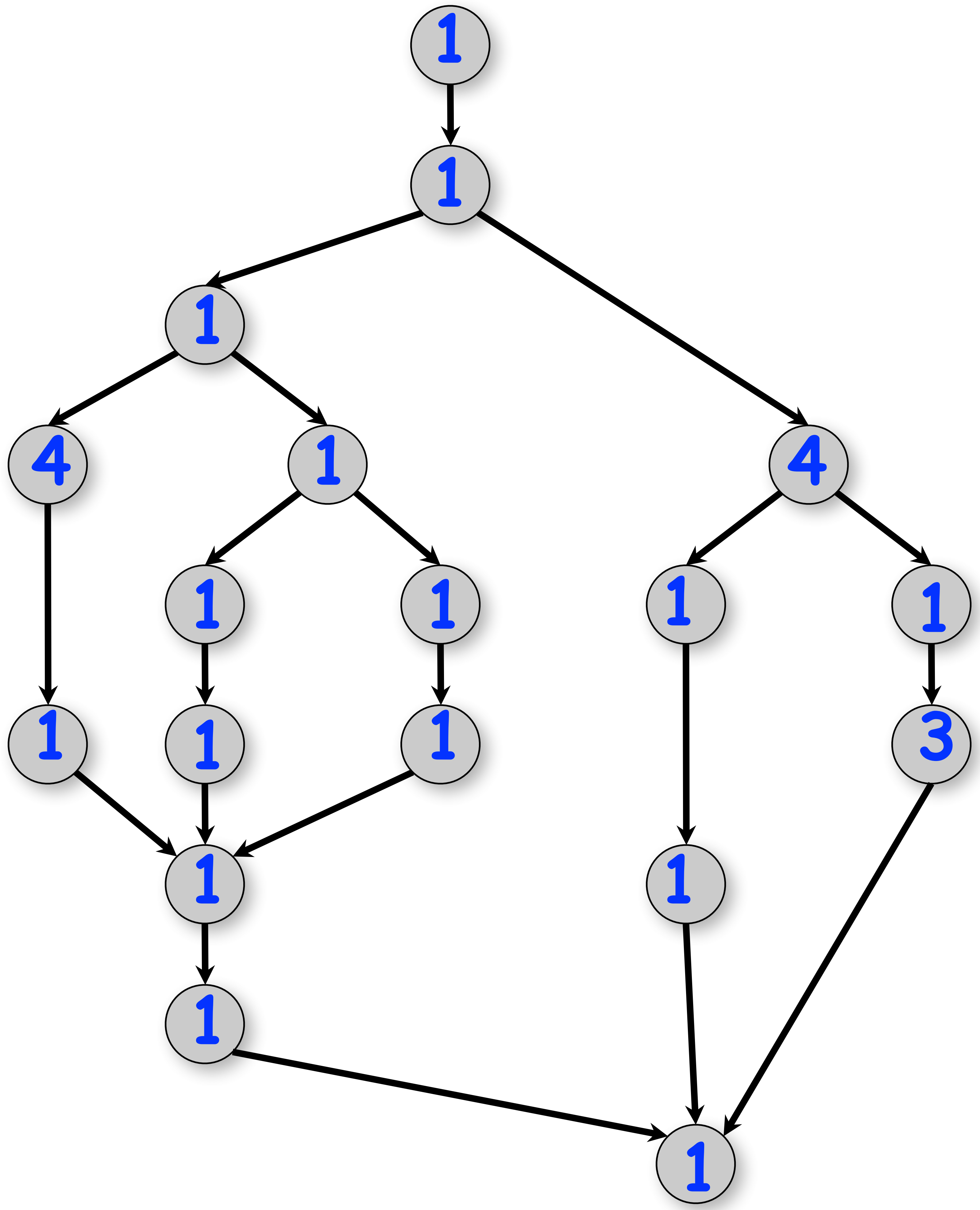
Complexity Measures for Computation Graphs

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



Example:

WORK(G) = 26

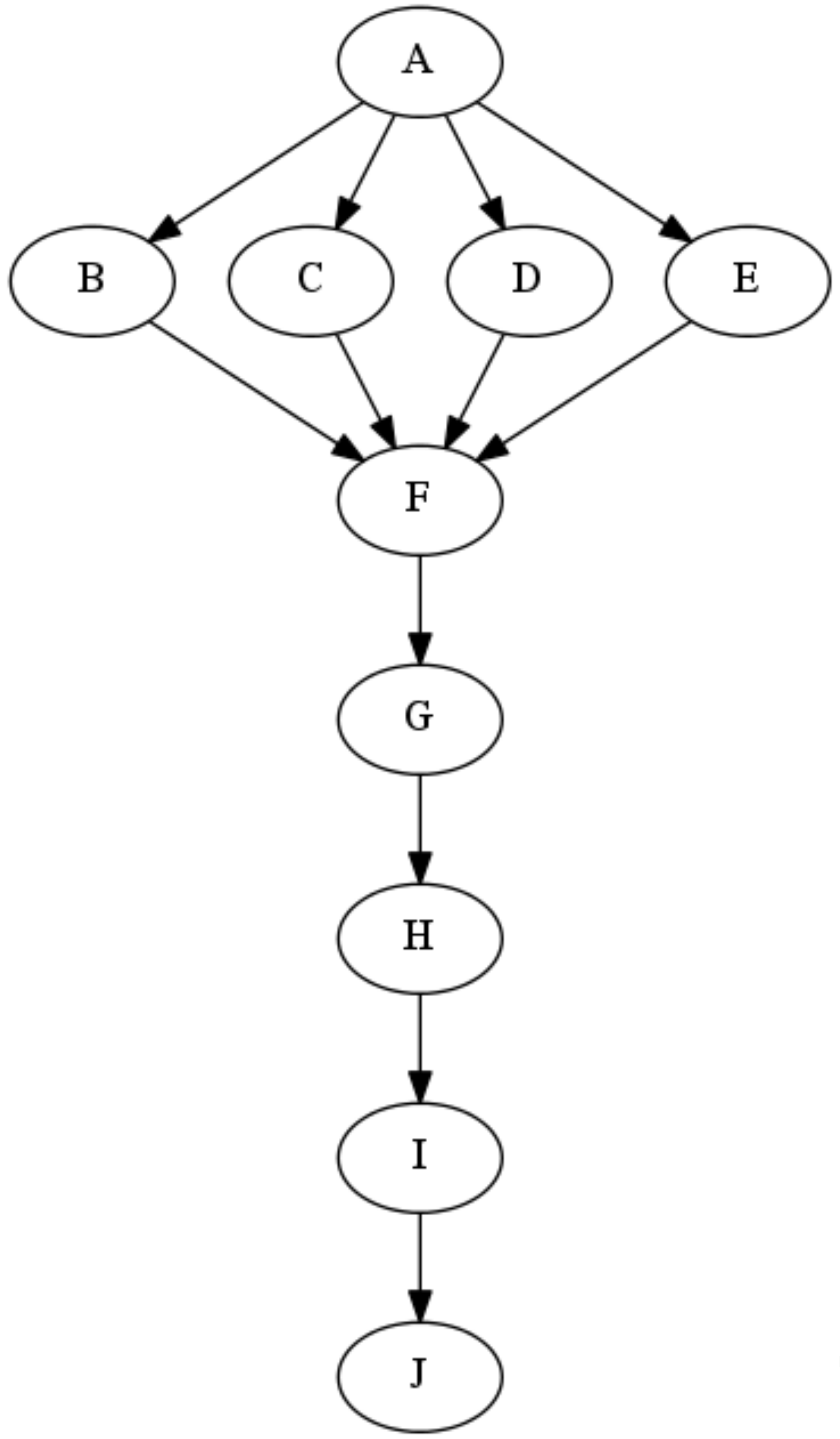
CPL(G) = 11

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

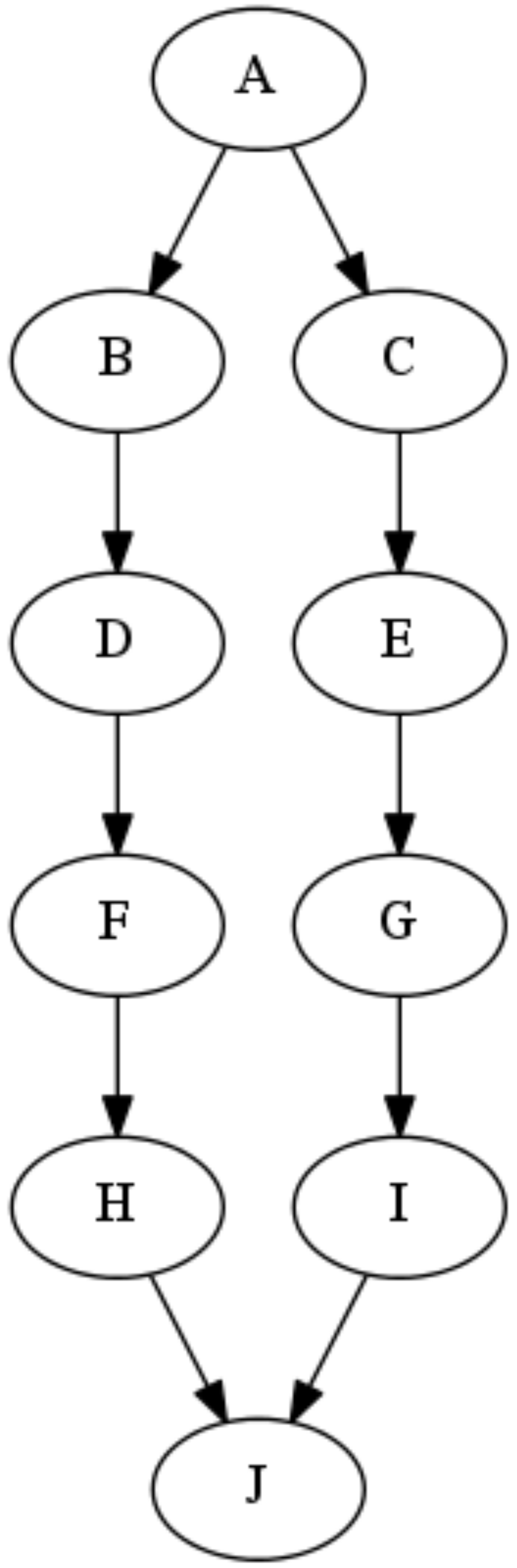
Which Computation Graph has more ideal parallelism?

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

Computation Graph 1



Computation Graph 2



Data Races

A data race occurs on location L in a program execution with computation graph CG if there exist steps (nodes) S1 and S2 in CG such that:

1. S1 does not depend on S2 and S2 does not depend on S1, i.e., S1 and S2 can potentially execute in parallel, and
2. Both S1 and S2 read or write L, and at least one of the accesses is a write.

- A data-race is usually considered an error. The result of a read operation in a data race is undefined. The result of a write operation is undefined if there are two or more writes to the same location.
  - Note that our definition of data race includes the case that both S1 and S2 write the same value in location L, even if that may not be considered an error.
- Above definition includes all "potential" data races i.e., we consider it to be a data race even if S1 and S2 end up executing on the same processor.

Example of a Sequential Program: Computing the 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;

By definition, an async inside the for loop would create a data race

Two-way Parallel Array Sum using async & finish constructs

Algorithm 2: Two-way Parallel ArraySum

Input: Array of numbers, X.
Output: sum = sum of elements in array X.

```
// Start of Task T1 (main program)
sum1 ← 0; sum2 ← 0;
// Compute sum1 (lower half) and sum2 (upper half) in parallel.
finish{
    async{
        // Task T2
        for i ← 0 to X.length/2 − 1 do
            sum1 ← sum1 + X[i];
    };
    async{
        // Task T3
        for i ← X.length/2 to X.length − 1 do
            sum2 ← sum2 + X[i];
    };
};
// Task T1 waits for Tasks T2 and T3 to complete
// Continuation of Task T1
sum ← sum1 + sum2;
return sum;
```

Announcements & Reminders

- IMPORTANT:
  - Bring your laptop to tomorrow's lab at 1pm or 4pm on Thursday (Sewall 301)
  - Watch videos for topic 1.4 for next lecture on Friday
- HW1 will be assigned today and be due on Jan 29th
- Each quiz (to be taken online on Canvas) will be due on the Friday after the unit is covered in class. The first quiz for Unit 1 (topics 1.1 - 1.5) will be assigned on next Wednesday and is due Friday, Jan 31st.
- See course web site for syllabus, work assignments, due dates, …
  - http://comp322.rice.edu