

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

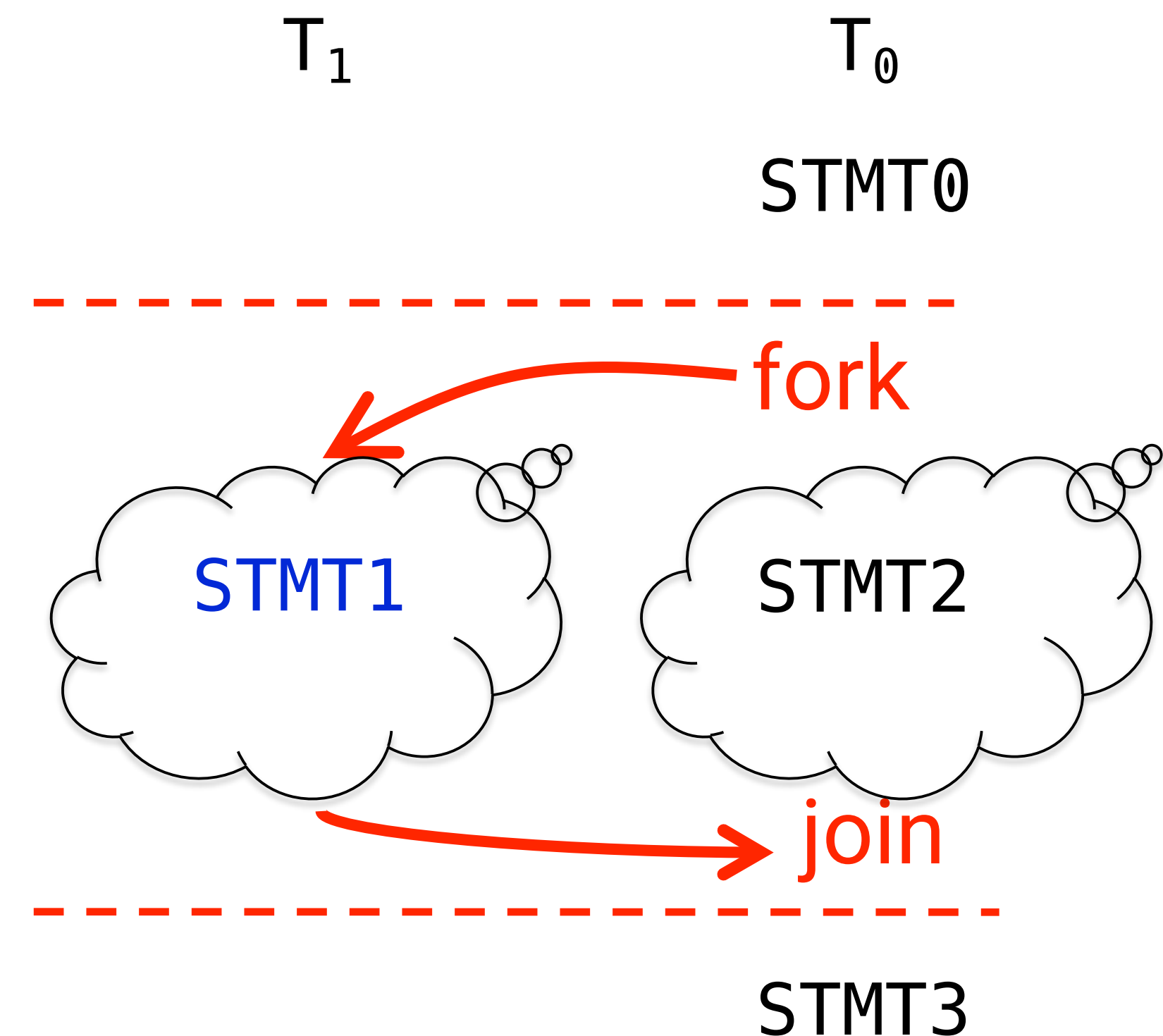
async S

- Creates a new child task that executes statement S

```
// 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.



One possible solution to Worksheet 1 (without statement reordering)

1. `finish {`
2. `async { Watch COMP 322 video for topic 1.2 by 1pm on Wednesday`
3. `Watch COMP 322 video for topic 1.3 by 1pm on Wednesday`
4. `}`
5. `async Make your bed`
6. `async { Clean out your fridge`
7. `Buy food supplies and store them in fridge }`
8. `finish { async Run load 1 in washer`
9. `Run load 2 in washer }`
10. `async Run load 1 in dryer`
11. `async Run load 2 in dryer`
12. `async Call your family`
13. `}`
14. `Post on Facebook that you're done with all your tasks!`



Another possible solution to Worksheet 1 (with statement reordering)

1. finish {
2. async Call your family
3. async Make your bed
4. async { Clean out your fridge
5. Buy food supplies and store them in fridge }
6. async { Run load 1 in washer
7. Run load 1 in dryer }
8. async { Run load 2 in washer
9. Run load 2 in dryer }
10. Watch COMP 322 video for topic 1.2 by 1pm on Wednesday
11. Watch COMP 322 video for topic 1.3 by 1pm on Wednesday
12. }
13. 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)

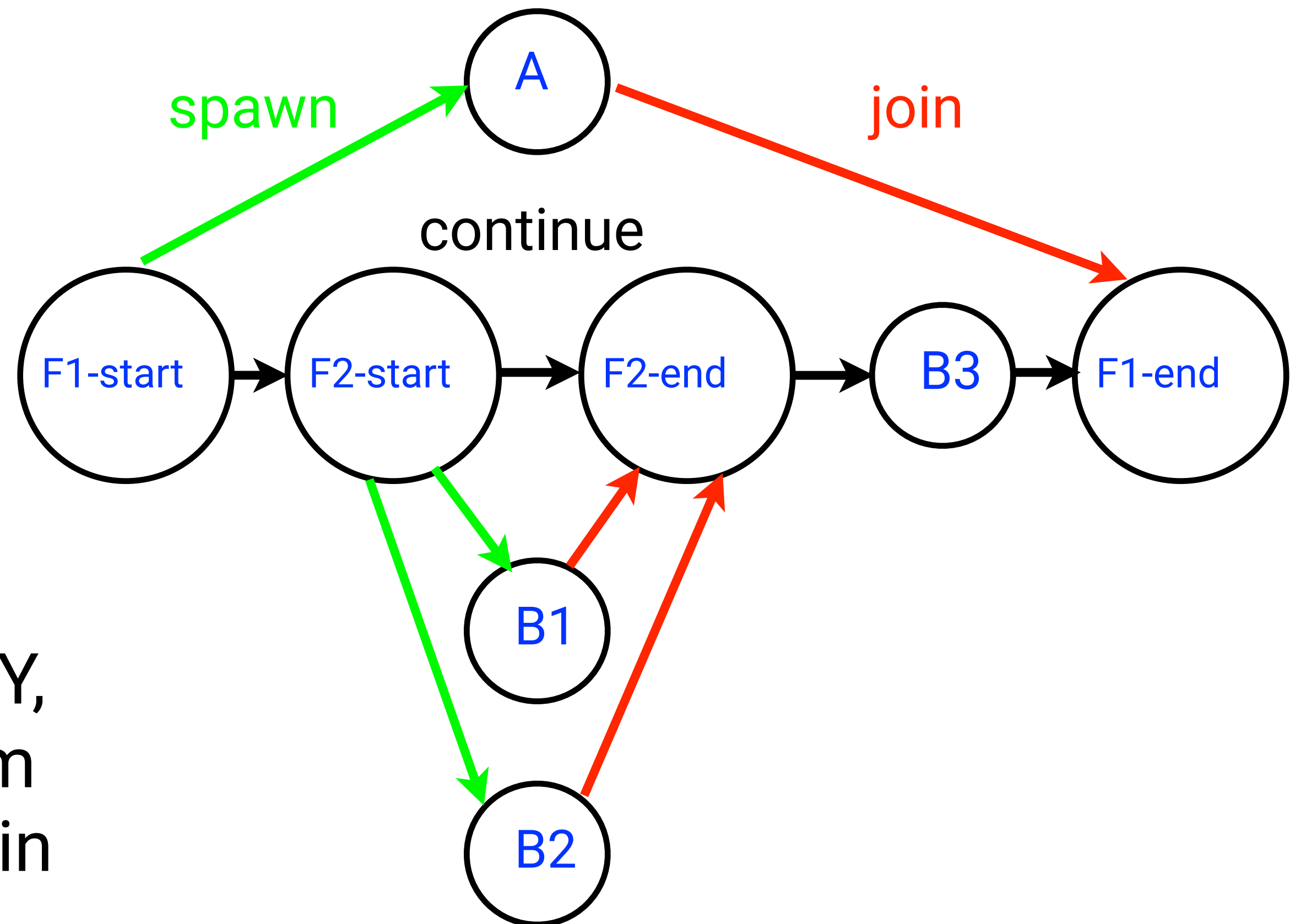


Which statements can potentially be executed in parallel with each other?

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

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.

Computation Graph



Complexity Measures for Computation Graphs

Define

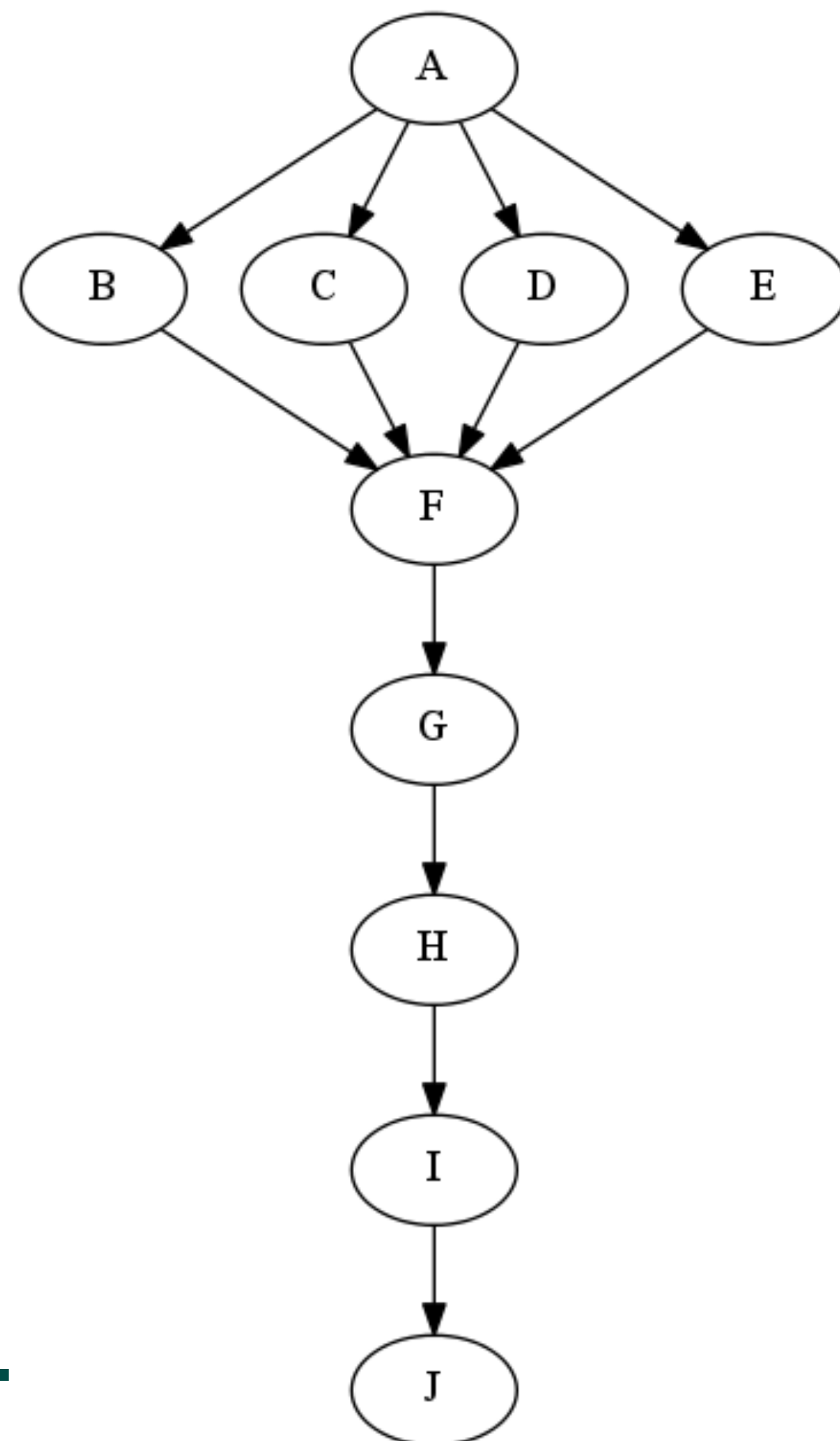
- $\text{TIME}(N)$ = execution time of node N
- $\text{WORK}(G)$ = sum of $\text{TIME}(N)$, for all nodes N in CG G
 - $\text{WORK}(G)$ is the total work to be performed in G
- $\text{CPL}(G)$ = length of a longest path in CG G , when adding up execution times of all nodes in the path
 - Such paths are 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



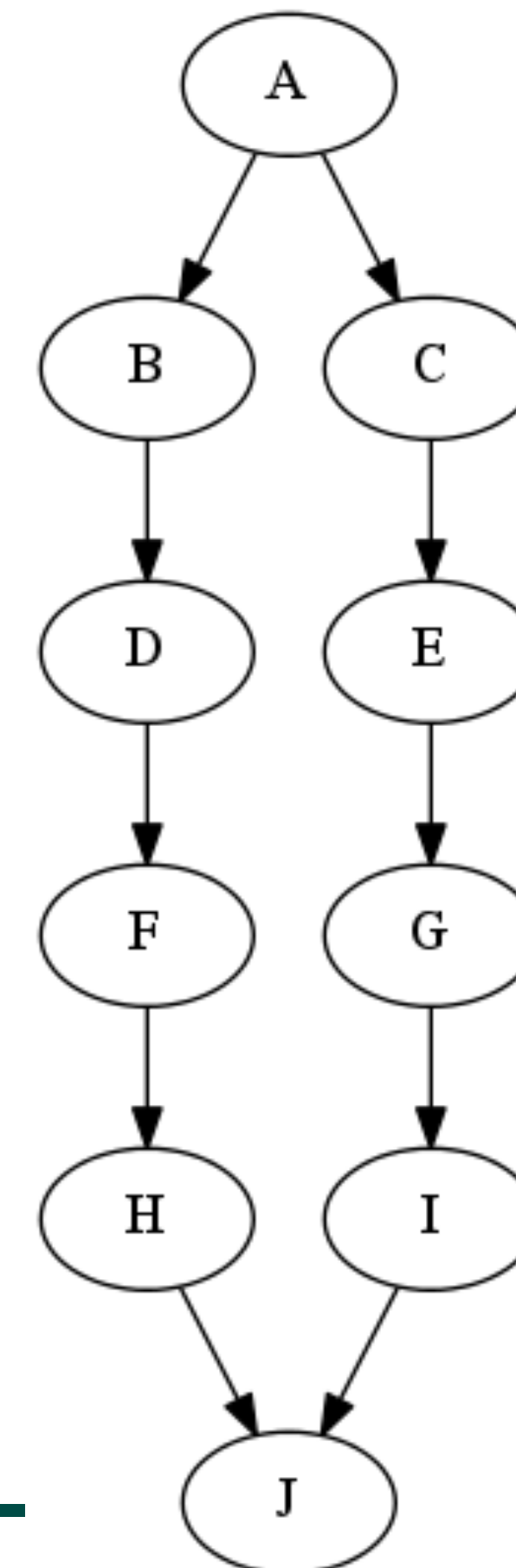
Which Computation Graph has more ideal parallelism?

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)

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

// 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 ;



Announcements & Reminders

- IMPORTANT:
 - This week's Thursday lab is at 4:50pm (Zoom)
 - Watch [videos](#) for topics 1.4 for next lecture
- HW1 will be available today and due on Feb 10th. (Homework is normally due on Wednesdays.)
- 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) is due by Feb 5th.
- See course web site for syllabus, work assignments, due dates, ...
 - <http://comp322.rice.edu>

