COMP 322: Fundamentals of Parallel Programming

Lecture 2: Computation Graphs, Ideal Parallelism

Instructors: Vivek Sarkar, Shams Iman
Department of Computer Science, Rice University
{vsarkar, shams}@rice.edu

http://comp322.rice.edu

Async and Finish Statements for Task Creation and Termination (Recap)

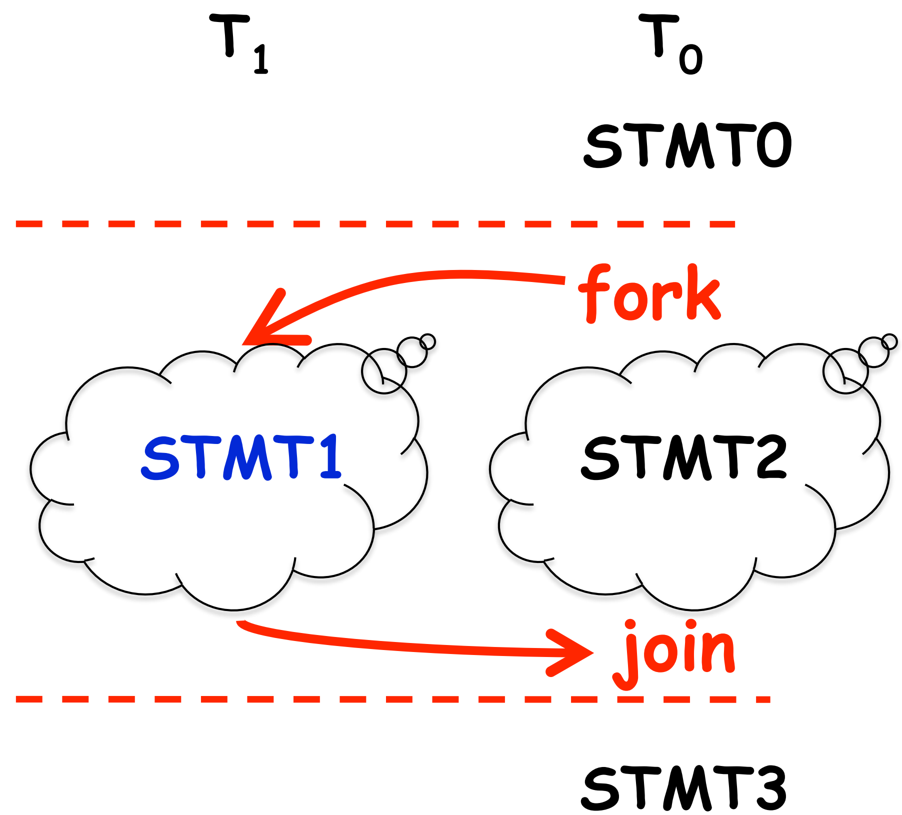
async S

- 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
              //Wait for T1
}             //End finish
STMT3;        //Continue in T0
```



One possible solution to Problem #1 in 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
            async 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 Problem #1 in Worksheet 1 (with statement reordering)

```
finish {
  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
  Call your family
}
Post on Facebook that you're done with all your tasks!
```

Is this a correct solution for Problem #2 in Worksheet 1?

```
finish {
  for (int i = 0 ; i < N ; i++)
    for (int j = 0 ; j < N ; j++)
        for (int k = 0 ; k < N ; k++)
          async {
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
          } // async
} // finish
```

Data race bug! Reads and writes can occur in parallel on the same C[i][j] location, in this example!

One Possible Solution to Problem #2 in Worksheet 1 (Parallel Matrix Multiplication)

```
finish {
  for (int i = 0 ; i < N ; i++)
    for (int j = 0 ; j < N ; j++)
      async {
        for (int k = 0 ; k < N ; k++)
          C[i][j] = C[i][j] + A[i][k] * B[k][j];
      } // async
} // finish
```

This program generates N^2 parallel async tasks, one to compute each C[i][j] element of the output array. Additional parallelism can be exploited within the inner k loop, but that would require more changes than inserting async & finish.

Another Possible Solution to Problem #2 in Worksheet 1 (Parallel Matrix Multiplication)

```
1.finish {
2.  for (int i = 0 ; i < N ; i++)
3.    async finish for (int j = 0 ; j < N ; j++)
4.      async finish for (int k = 0 ; k < N ; k++)
5.          C[i][j] = C[i][j] + A[i][k] * B[k][j];
6. } // finish
```

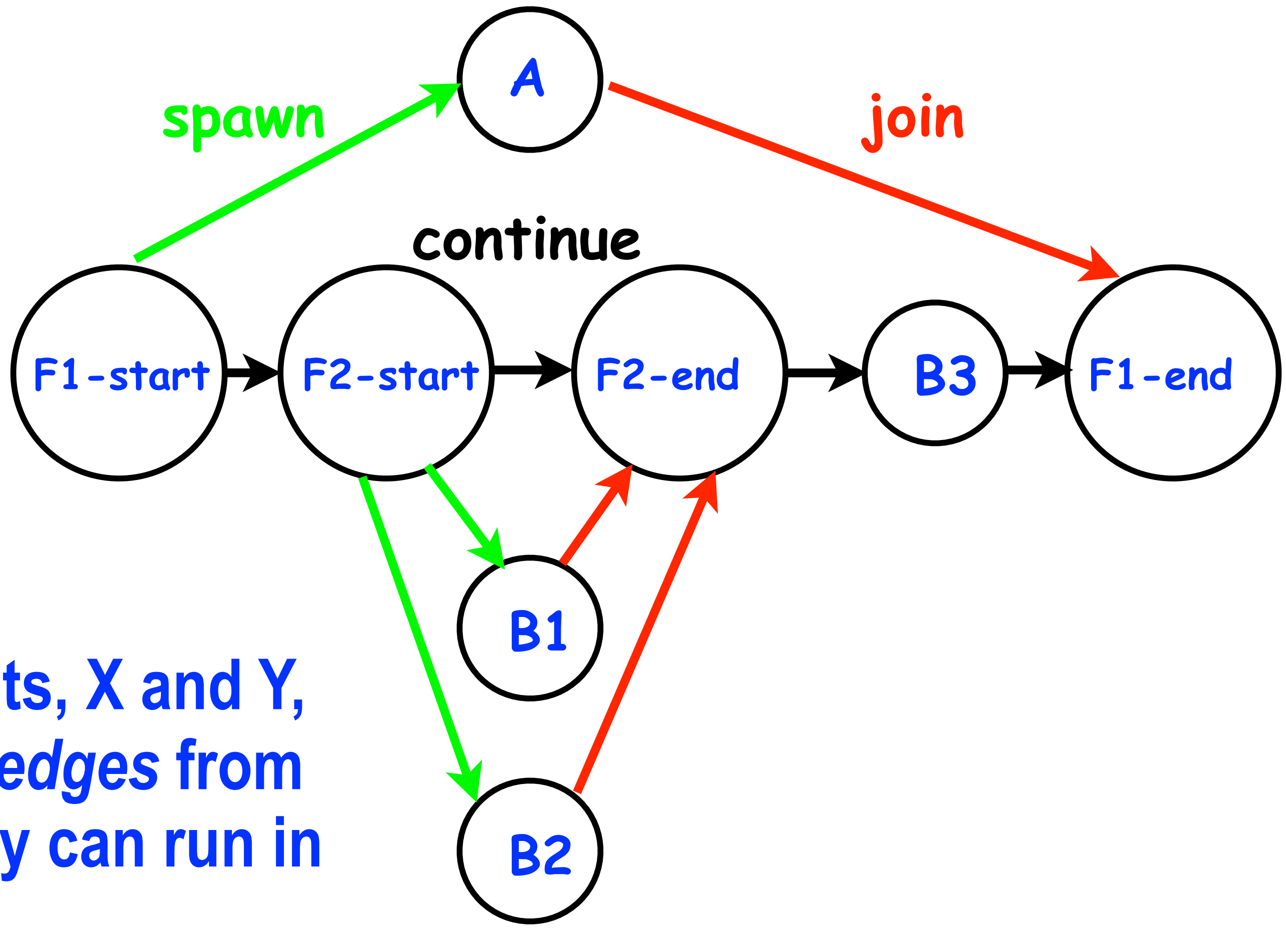
What is the impact of finish in lines 3 and 4? Compare with:

```
7.finish {
8.  for (int i = 0 ; i < N ; i++)
9.    async for (int j = 0 ; j < N ; j++)
10.      async for (int k = 0 ; k < N ; k++)
11.          C[i][j] = C[i][j] + A[i][k] * B[k][j];
12. } // finish
```

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
```

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.

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 and 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)

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 smallest possible execution time for the computation graph

What is the critical path length of this parallel computation?

```
finish { // F1
  async A; // Boil water & pasta (20)
  finish { // F2
    async B1; // Chop veggies (5)
    async B2; // Brown meat (10)
  } // F2
  B3; // Make pasta sauce (5)
} // F1
```

Step B1

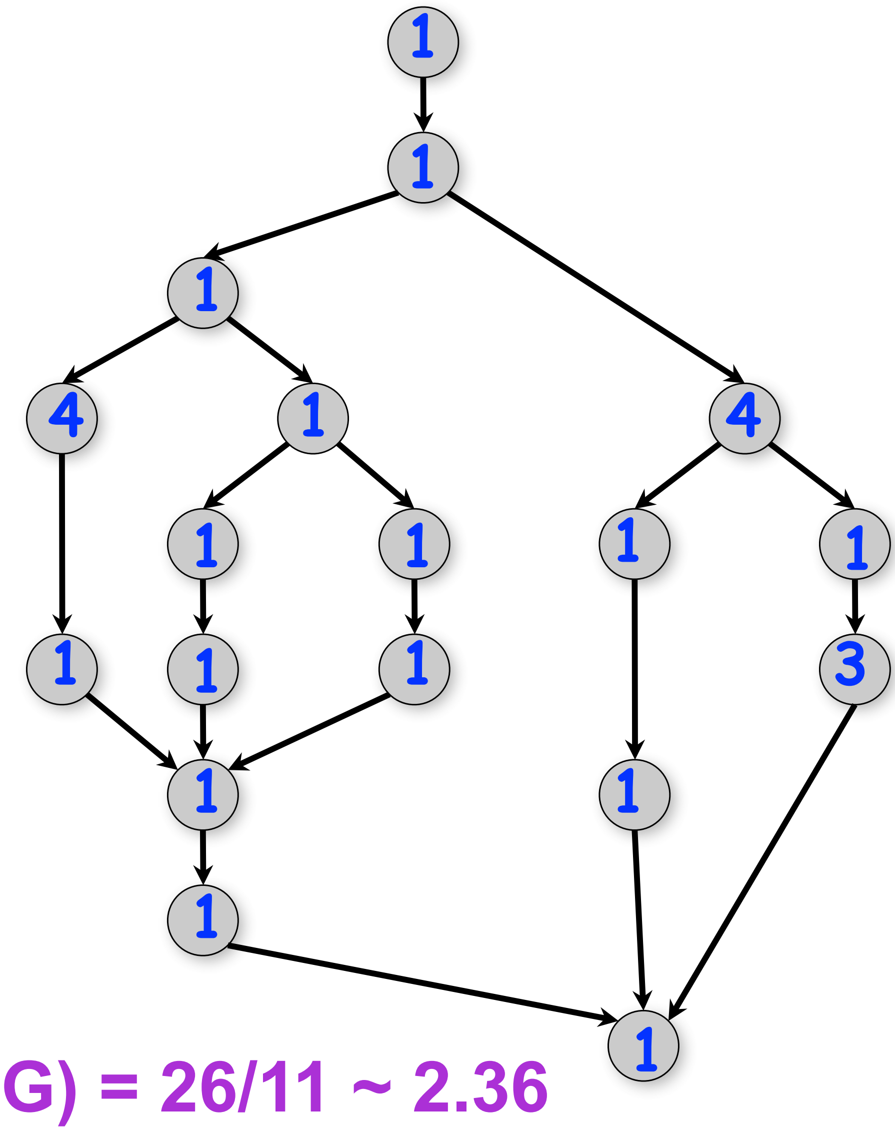
Step B2

Step B3

Step A

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



<u>Example</u>:

WORK(G) = 26

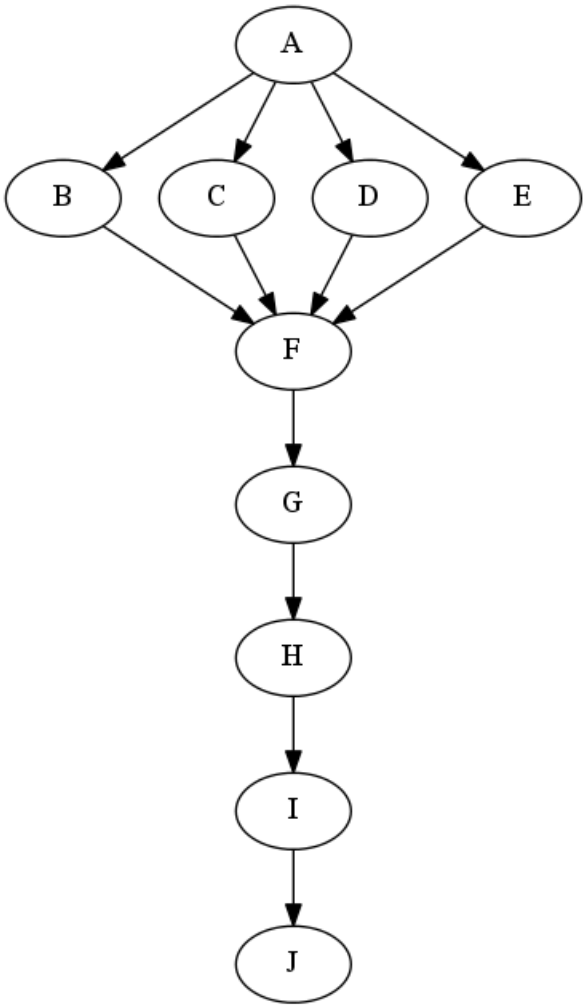
CPL(G) = 11

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

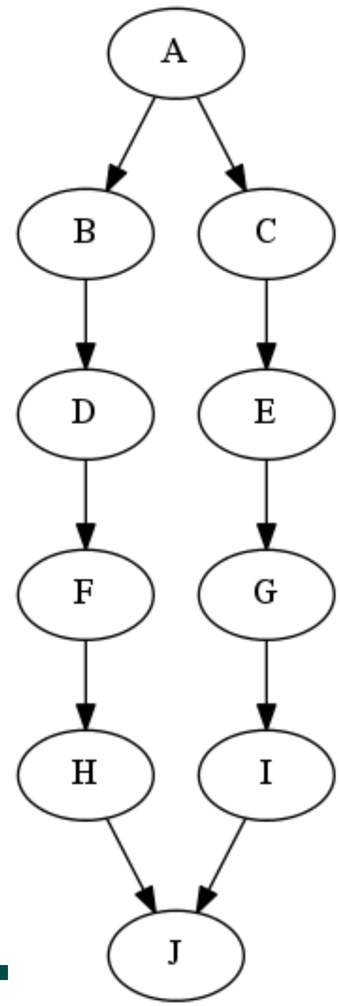
Which Computation Graph has more ideal parallelism?

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

Computation Graph 1



Computation Graph 2



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 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.
- 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.

Data Race Example: Buggy Matrix Multiply with N = 2

```
1.finish {
2.   for (int i = 0 ; i < N ; i++)
3.     for (int j = 0 ; j < N ; j++)
4.         for (int k = 0 ; k < N ; k++)
5.           async {
6.             C[i][j] = C[i][j] + A[i][k] * B[k][j];
7.           } // async
8.} // finish
```

No directed edge in computation graph between S6(i=0,j=0,k=0) and S6(i=0,j=0,k=1), but both read and write C[0][0].

Reminders

- IMPORTANT:
  - —Send email to comp322-staff@rice.edu if you do not have access to Piazza site (otherwise use Piazza for class communications, as far as possible)
  - —Bring your laptop to today's lab at 7pm on Wednesday (Section A01: DH 1064, Section A02: DH 1070)
  - —Watch videos for topic 1.4 for next lecture on Friday
- Complete each week's assigned quizzes on edX by 11:59pm that Friday. This week, you should submit quizzes for lecture & demonstration videos for topics 1.1, 1.2, 1.3, 1.4
- HW1 will be assigned on Jan 15th and be due on Jan 28th
- See course web site for syllabus, work assignments, due dates, …
  - http://comp322.rice.edu