COMP 322: Parallel and Concurrent Programming

Lecture 12: Scheduling

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

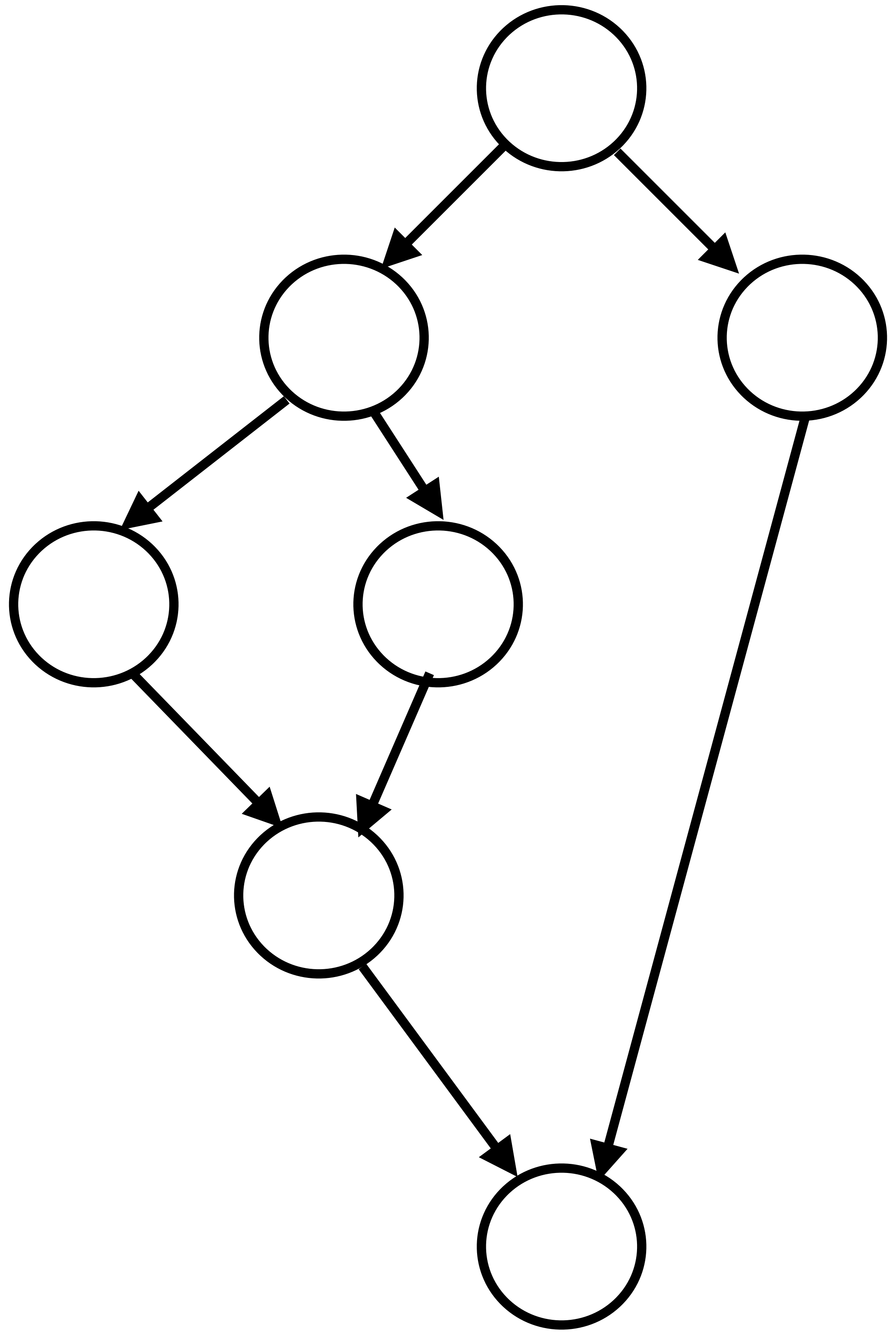
http://comp322.rice.edu

Computation Graphs

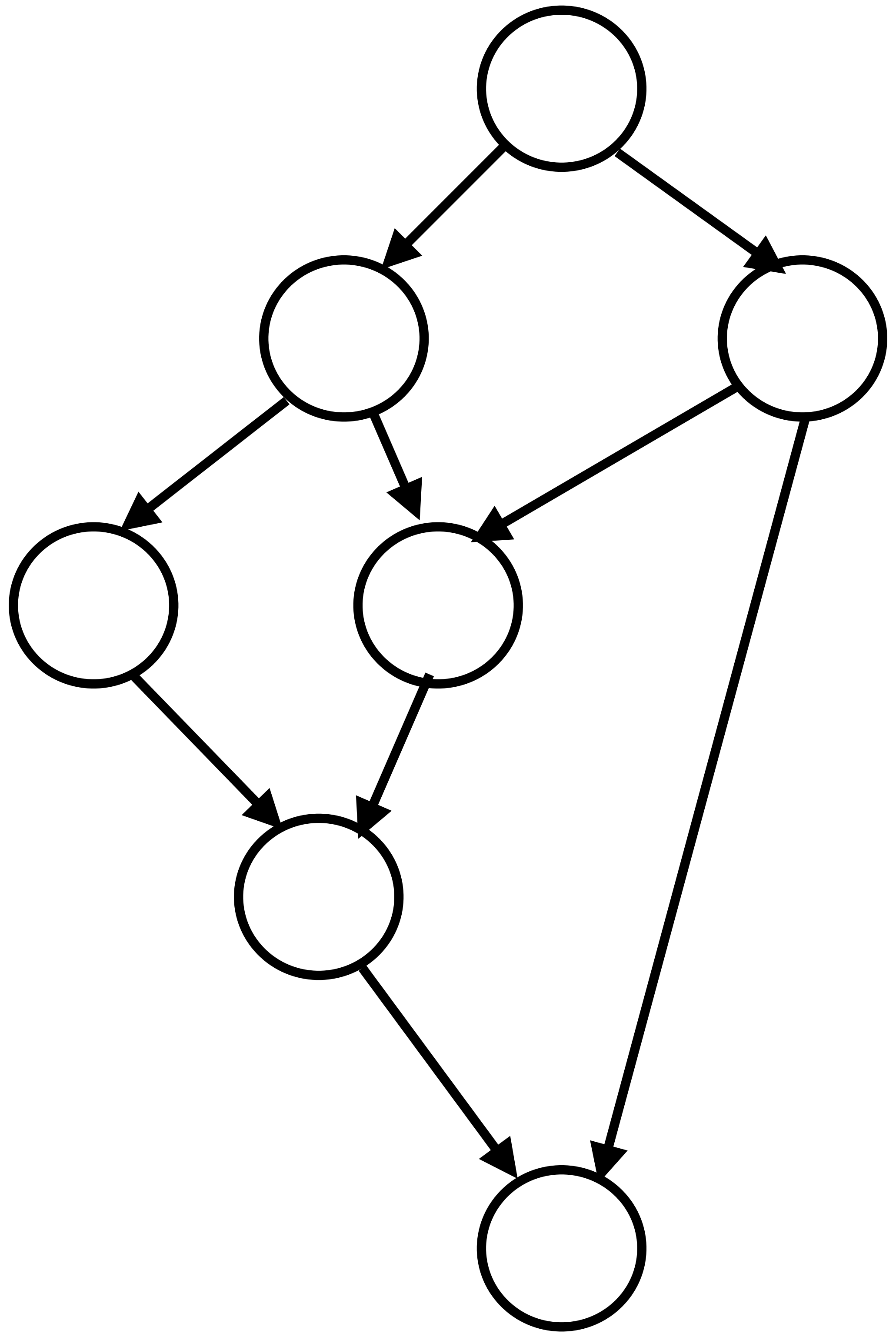
Structured Parallelism (Finish/async)

Futures and Future Tasks

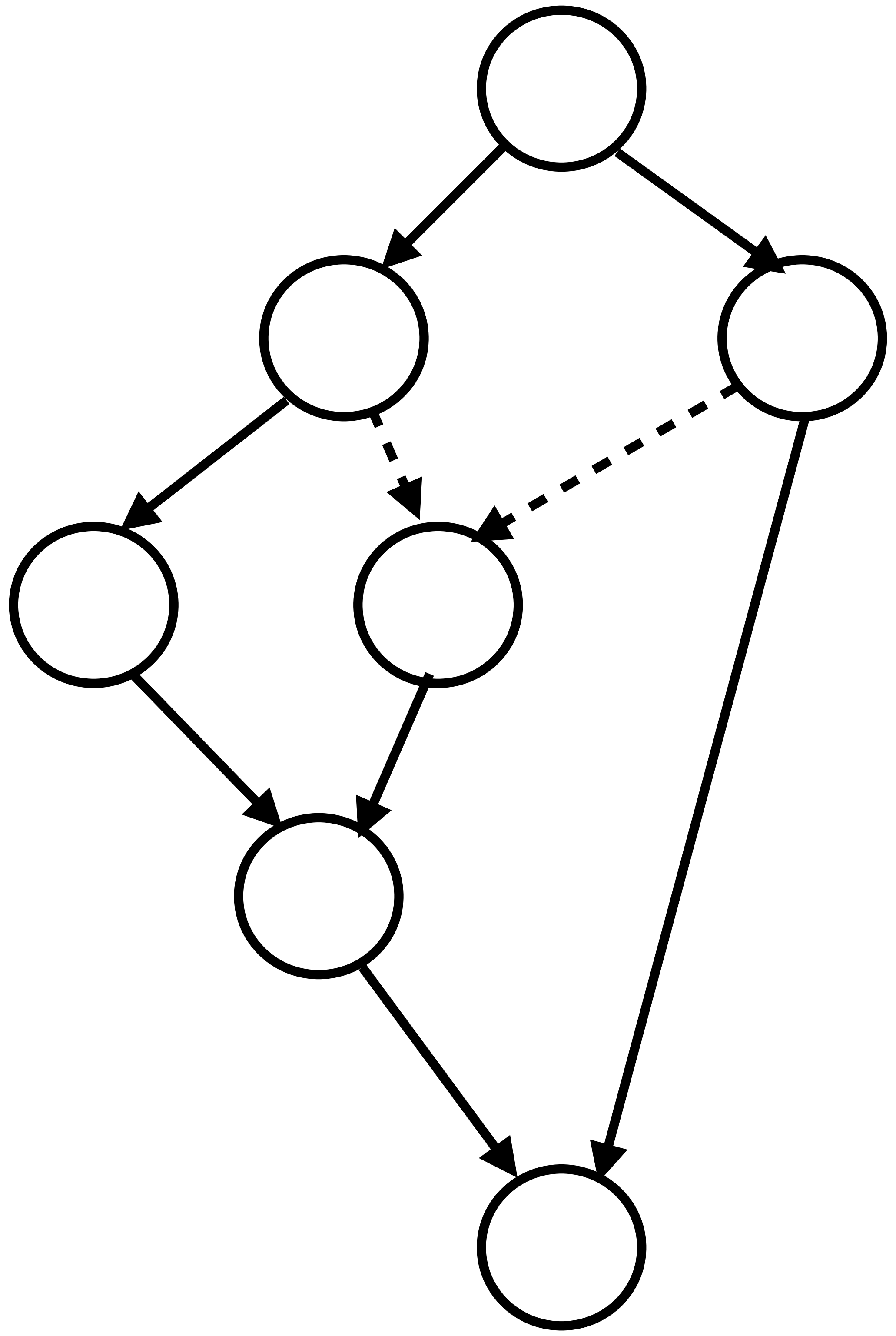
Promises and Data-Driven Tasks



<



<



Computation Graphs

- Structured parallelism (finish/async):
  Create structured graphs (similar to what structured programming can create)
  No high-level data representation: have to share data
  Fast implementation, easy to synchronize large # of tasks
- Futures and future tasks:
  Easy to construct unstructured, arbitrary graphs
  Elegant, functional high-level data representation: futures
  Functional, “push” model: “where is the data going to, create futures for those”
  Large overhead when handling large # of tasks
- Promises and data-driven tasks:
  Easy to construct unstructured, arbitrary graphs with unknown task-promise association
  Data-driven, “pull” model: “what data does this DDT depend on, create promises for those”
  Can have a faster implementation than futures
  Large overhead when handling large # of tasks

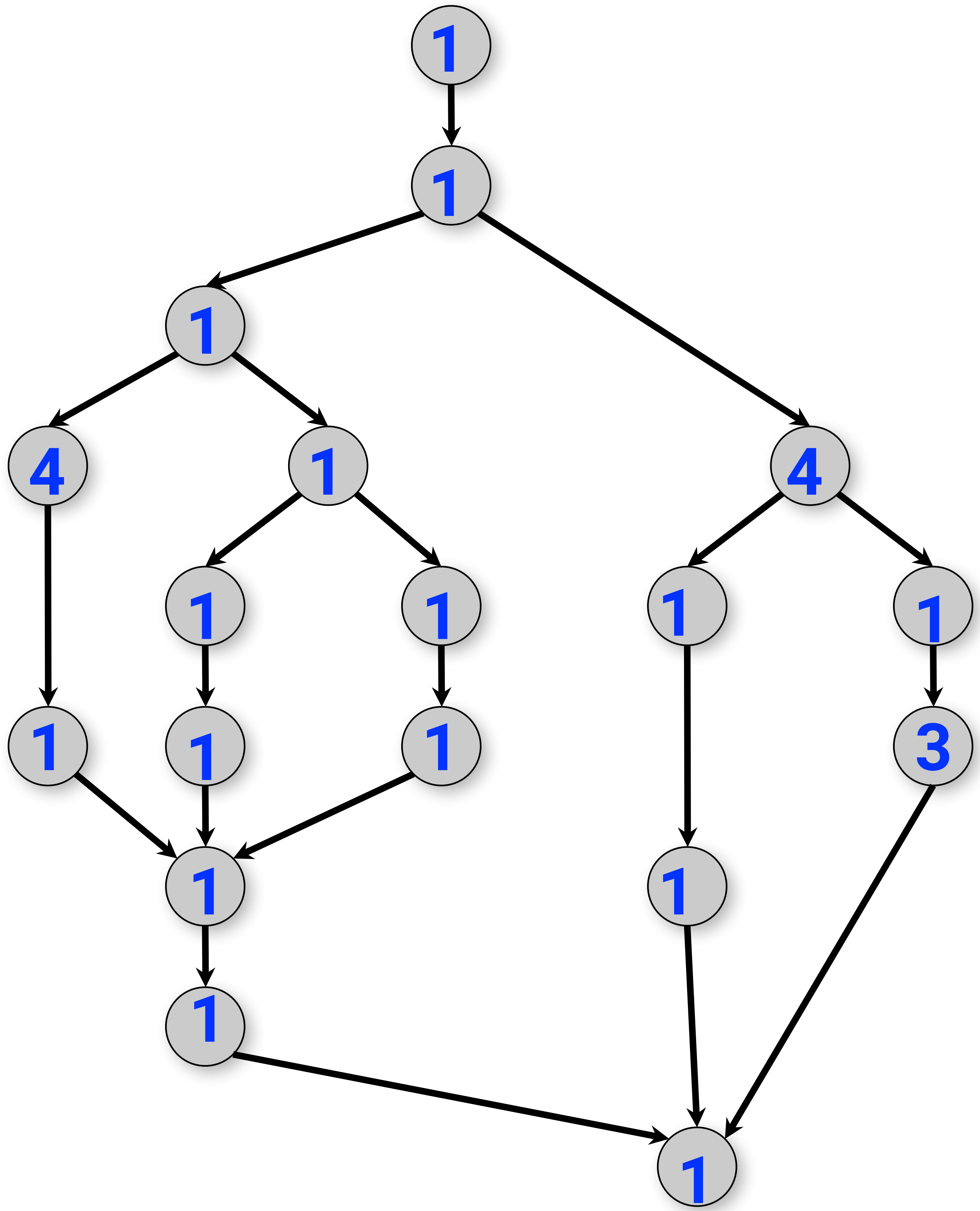
Ordering Constraints and Transitive Edges in a Computation Graph

- The primary purpose of a computation graph is to determine if an ordering constraint exists between two steps (nodes)
  - Observation: Node A must be performed before node B if there is a path of directed edges from A and B

- An edge, X →Y, in a computation graph is said to be transitive if there exists a path of directed edges from X to Y that does not include the X →Y edge
  - Observation: Adding or removing a transitive edge does not change the ordering constraints in a computation graph

Ideal Parallelism (Recap)

- 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

What is the critical path length of this parallel computation?

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

Step A

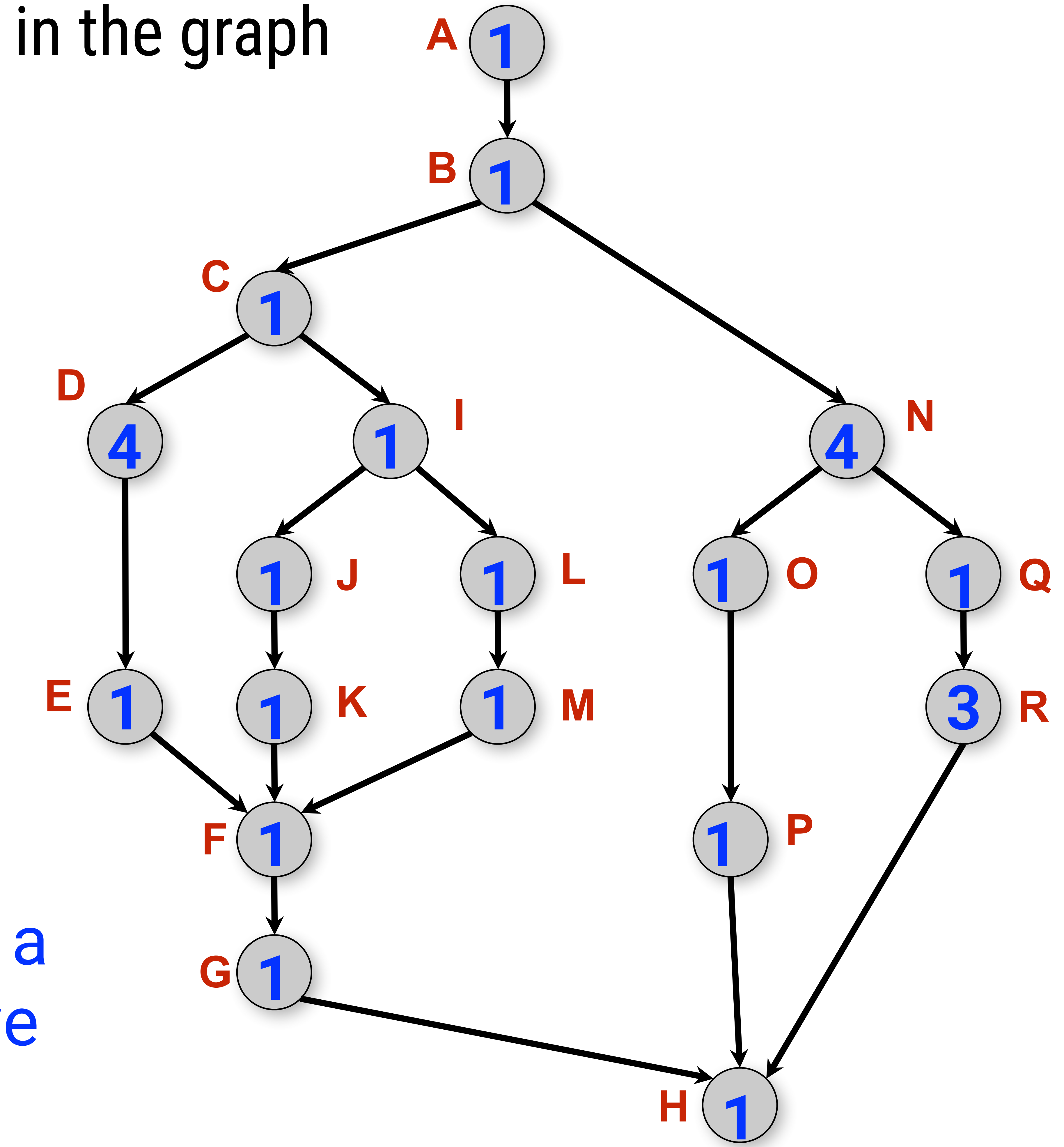
Step B1

Step B2

Step B3

Scheduling of a Computation Graph on a fixed number of processors

Node label = time(N), for all nodes N in the graph



Start time	Proc 1	Proc 2	Proc 3
0	A		
1	B		
2	C	N	
3	D	N	I
4	D	N	J
5	D	N	K
6	D	Q	L
7	E	R	M
8	F	R	O
9	G	R	P
10	H		
11	Completion time = 11		

NOTE: this schedule achieved a completion time of 11. Can we do better?

Scheduling of a Computation Graph on a fixed number of processors

- Assume that node N takes TIME(N) regardless of which processor it executes on, and that there is no overhead for creating parallel tasks

- A schedule specifies the following for each node
  - START(N) = start time
  - PROC(N) = index of processor in range 1...P

  such that
  - START(i) + TIME(i) <= START(j), for all CG edges from i to j (Precedence constraint)
  - A node occupies consecutive time slots in a processor (Non-preemption constraint)
  - All nodes assigned to the same processor occupy distinct time slots (Resource constraint)

Greedy Schedule

- A greedy schedule is one that never forces a processor to be idle when one or more nodes are ready for execution

- A node is ready for execution if all its predecessors have been executed
- Observations
  - T_1 = WORK(G), for all greedy schedules
  - T_∞ = CPL(G), for all greedy schedules

- $T_P(S)$ = execution time of schedule S for computation graph G on P processors

Lower Bounds on Execution Time of Schedules

- Let T_P = execution time of a schedule for computation graph G on P processors
  - T_P can be different for different schedules, for same values of G and P

- Lower bounds for all greedy schedules
  - Capacity bound: $T_P \geq WORK(G)/P$
  - Critical path bound: $T_P \geq CPL(G)$

- Putting them together
  - $T_P \geq \max(WORK(G)/P, CPL(G))$

Upper Bound on Execution Time of Greedy Schedules

Theorem [Graham ’66].
Any greedy scheduler achieves

$$T_P \leq WORK(G)/P + CPL(G)$$

Start time	Proc 1	Proc 2	Proc 3
0	A		
1	B		
2	C	N	
3	D	N	I
4	D	N	J
5	D	N	K
6	D	Q	L
7	E	R	M
8	F	R	O
9	G	R	P
10	H		
11			

Proof sketch:
Define a time step to be complete if P processors are scheduled at that time, or incomplete otherwise

complete time steps ≤ WORK(G)/P

incomplete time steps ≤ CPL(G)

Bounding the Performance of Greedy Schedulers

Combine lower and upper bounds to get

$\max(WORK(G)/P, CPL(G)) \leq T_P \leq WORK(G)/P + CPL(G)$

Corollary: Any greedy scheduler achieves execution time T_P that is within a factor of 2 of the optimal time (since $\max(a,b)$ and $(a+b)$ are within a factor of 2 of each other, for any $a \geq 0, b \geq 0$).

Corollary 2: Lower and upper bounds approach the same value whenever:

- There's lots of parallelism, $WORK(G)/CPL(G) \gg P$
- Or there's little parallelism, $WORK(G)/CPL(G) \ll P$

Abstract Performance Metrics

- Basic Idea
  - Count operations of interest, as in big-O analysis, to evaluate parallel algorithms
  - Abstraction ignores many overheads that occur on real systems
- Calls to doWork()
  - Programmer inserts calls of the form, doWork(N) within a task (async, future task or data-driven task) to indicate abstract execution of N application-specific abstract operation
    - e.g., in lab 4, we included one call to doWork(1) for each double addition, and ignore the cost of everything else
- Abstract metrics are enabled by calling HjSystemProperty.abstractMetrics.set(true) at start of program execution
- If an HJ program is executed with this option, abstract metrics can be printed at end of program execution with calls to abstractMetrics().totalWork(), abstractMetrics().criticalPathLength(), and abstractMetrics().idealParallelism()

Abstract Performance Metrics

- Pay attention where you put doWork() calls
- What does this mean?

```
var bottom = future(() → . . .);
var top = future(() → . . .)
doWork(1);
return bottom.get() + top.get();
```

- Correct:

```
var bottom = future(() → . . .);
var top = future(() → . . .);

var bottomVal = bottom.get();
var topVal = top.get();
doWork(1);
return bottomVal + topVal;
```