

# COMP 322: Parallel and Concurrent Programming

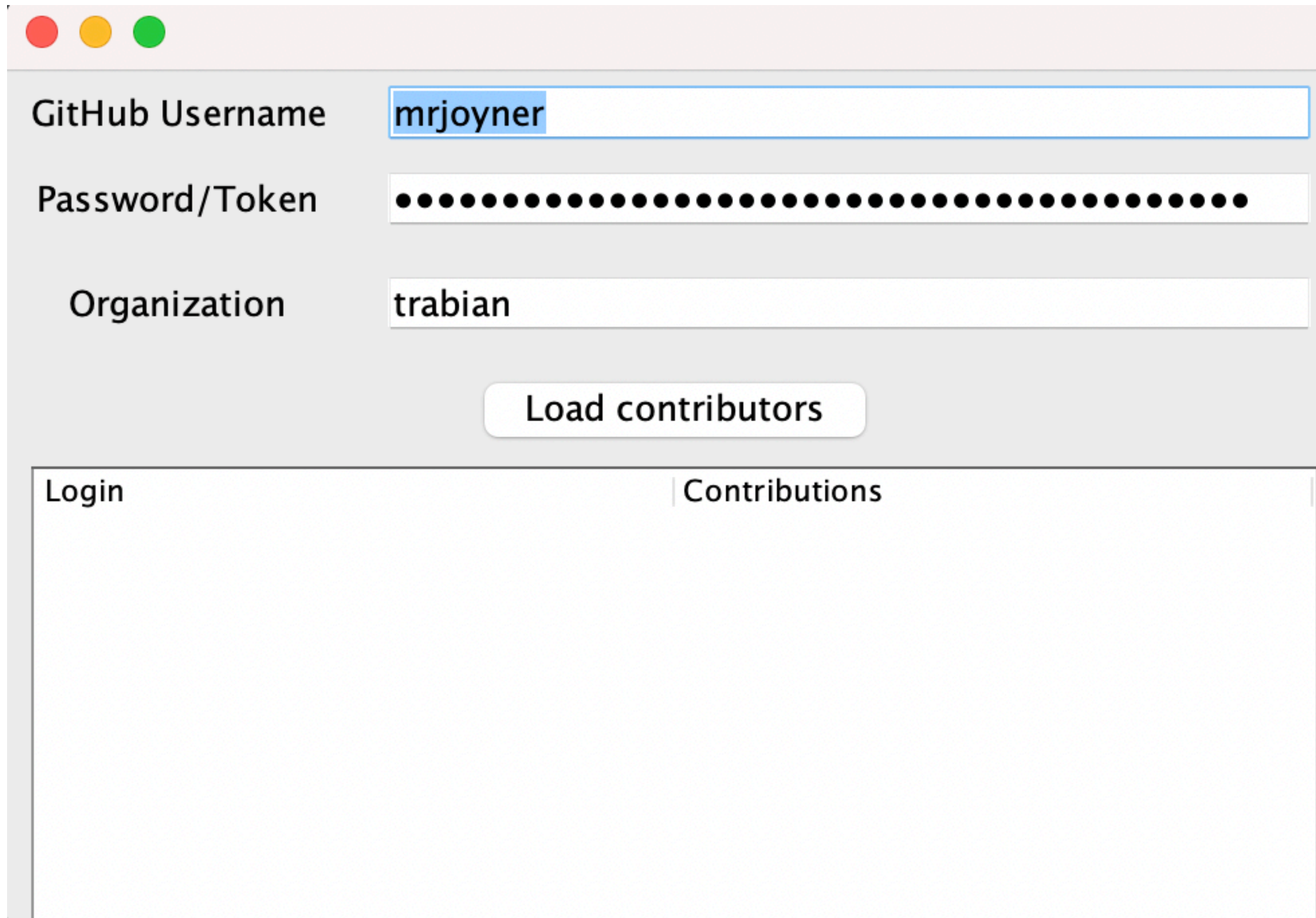
## Lecture 11: Scheduling

Mack Joyner  
mjoyner@rice.edu

<http://comp322.rice.edu>



# Homework 2: GitHub Contributors



The screenshot shows a web application window with a light gray background and a title bar with three colored buttons (red, yellow, green). The form contains three input fields: "GitHub Username" with the value "mrjoyner", "Password/Token" with a masked password of 20 dots, and "Organization" with the value "trabian". Below the inputs is a "Load contributors" button. At the bottom, there is a large white area with two tabs: "Login" and "Contributions".

GitHub Username:

Password/Token:

Organization:

Login | Contributions



# GitHub Contributors

The screenshot shows a web application window with a light gray background and a title bar with three colored buttons (red, yellow, green). The main content area is divided into several sections:

- GitHub Username:** A text input field containing the value "mrjoyner".
- Password/Token:** A password input field filled with black dots.
- Organization:** A text input field containing the value "trabian".
- Action:** A blue button with rounded corners labeled "Load contributors".
- Output Area:** A large white rectangular area below the form, divided into two tabs: "Login" (selected) and "Contributions". The "Login" tab is currently empty.



# GUI Events with Java Swing

---

- Swing enables you to build a GUI in Java and respond to user events
- Containers (e.g. JFrame)
- Components
  - JButton
  - JLabel
  - JTextField
- Users interact with the GUI and trigger actions (events)
- ActionListeners are setup for a component to respond to the event



# GitHub Contributors

Login	Contributions
dhh	3393
jeremy	3352
mpdehaan	3321
josevalim	2501
tenderlove	2268
billdawson	1823
fxn	1600
marshall	1498
vishalduggal	1494
trabianmatt	1421



# 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

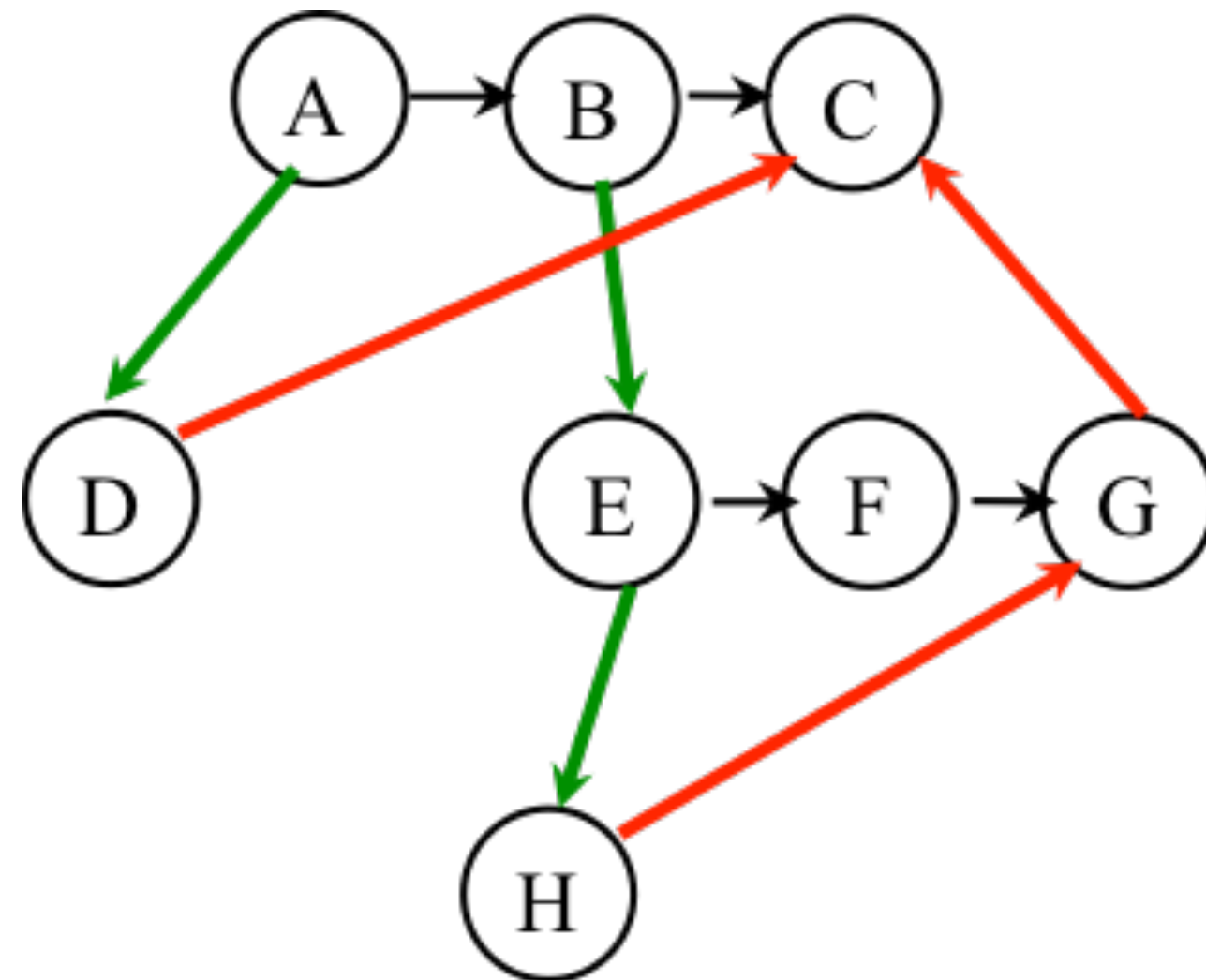


# Ordering Constraints and Transitive Edges in a Computation Graph

- 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,  $X \rightarrow Y$ , 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  $X \rightarrow Y$  edge
  - Observation: Adding or removing a transitive edge does not change the ordering constraints in a computation graph



# Reverse Engineering a Computation Graph



## Observations:

- Any node with out-degree  $> 1$  must be an async (must have an outgoing **spawn edge**)
- Any node with in-degree  $> 1$  must be an end-finish (must have an incoming **join edge**)
- Adding or removing transitive edges does not impact ordering constraints

```
1. A ();
2. finish { // F1
3.   async D ();
4.   B ();
5.   E ();
6.   finish { // F2
7.     async H ();
8.     F ();
9.   } // F2
10. G ();
11. } // F1
12. C ();
```





# Ideal Parallelism (Recap)

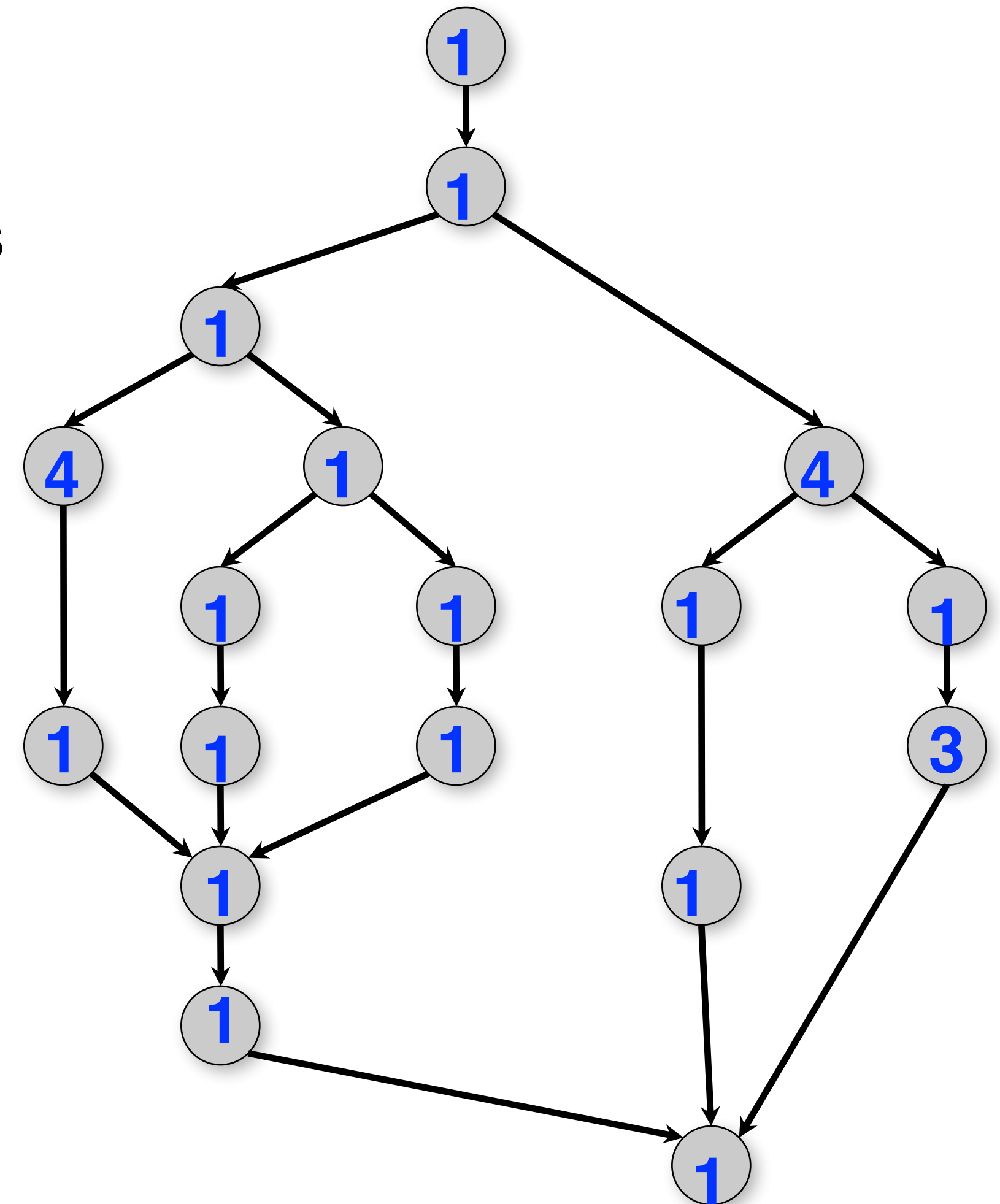
- Define **ideal parallelism** of Computation G Graph as the ratio,  $WORK(G)/CPL(G)$
- Ideal Parallelism only depends on the computation graph, and is the speedup that you can obtain with an unbounded number of processors

## Example:

$$WORK(G) = 26$$

$$CPL(G) = 11$$

$$\text{Ideal Parallelism} = WORK(G)/CPL(G) = 26/11 \sim 2.36$$



# What is the critical path length of this parallel computation?

1. `finish () -> {` // F1
2. `async () -> A);` // Boil water & pasta (10)
3. `finish () -> {` // F2
4. `async () -> B1);` // Chop veggies (5)
5. `async () -> B2);` // Brown meat (10)
6. `});` // F2
7. `B3;` // Make pasta sauce (5)
8. `})` // F1

**Step A**



**Step B3**



**Step B1**

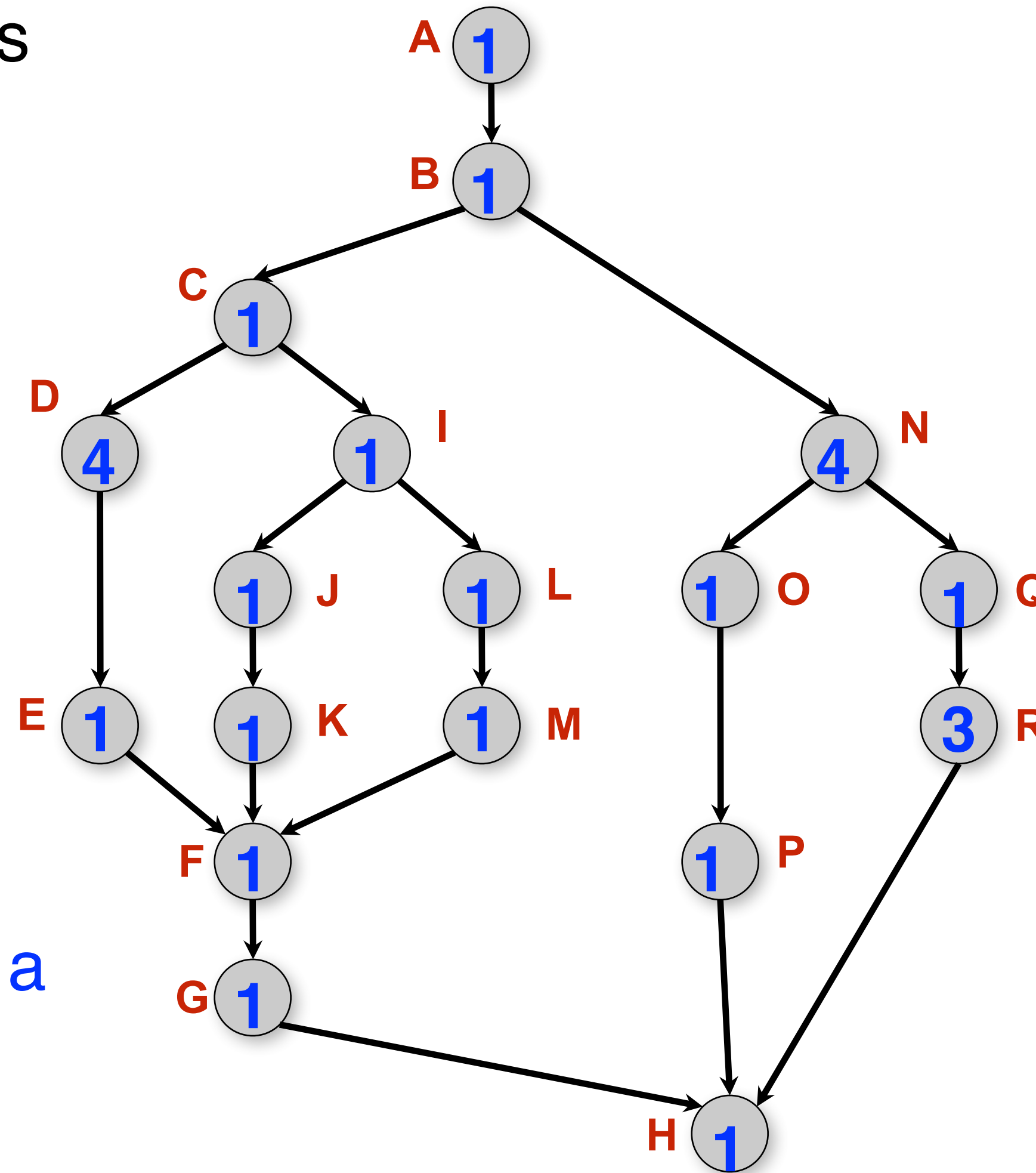


**Step B2**



# Scheduling of a Computation Graph on a fixed number of processors

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



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

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		



# Scheduling of a Computation Graph on a fixed number of processors

- Assume that node  $N$  takes  $\text{TIME}(N)$  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
  - $\text{START}(N)$  = start time
  - $\text{PROC}(N)$  = index of processor in range  $1 \dots P$

such that

- $\text{START}(i) + \text{TIME}(i) \leq \text{START}(j)$ , 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
  - $T_1 = \text{WORK}(G)$ , for all greedy schedules
  - $T_\infty = \text{CPL}(G)$ , 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:  $T_P \geq \text{WORK}(G)/P$
  - Critical path bound:  $T_P \geq \text{CPL}(G)$
- Putting them together
  - $T_P \geq \max(\text{WORK}(G)/P, \text{CPL}(G))$



# Upper Bound on Execution Time of Greedy Schedules

Theorem [Graham '66].  
Any greedy scheduler achieves

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

Proof sketch:

Define a time step to be **complete** if  $P$  processors are scheduled at that time, or **incomplete** otherwise

# complete time steps  $\leq \text{WORK}(G)/P$

# incomplete time steps  $\leq \text{CPL}(G)$

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			



# Bounding the Performance of Greedy Schedulers

Combine lower and upper bounds to get

$$\max(\text{WORK}(G)/P, \text{CPL}(G)) \leq T_p < \text{WORK}(G)/P + \text{CPL}(G)$$

**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,  $\text{WORK}(G)/\text{CPL}(G) \gg P$

Or there's little parallelism,  $\text{WORK}(G)/\text{CPL}(G) \ll P$

