# Parallel And Sequential Data Structures and Algorithms

## Parallel Scheduling

# Announcements

- **SandwichLab** due on Wednesday (not today!)
- **Final Exam** next week on Friday at 1pm

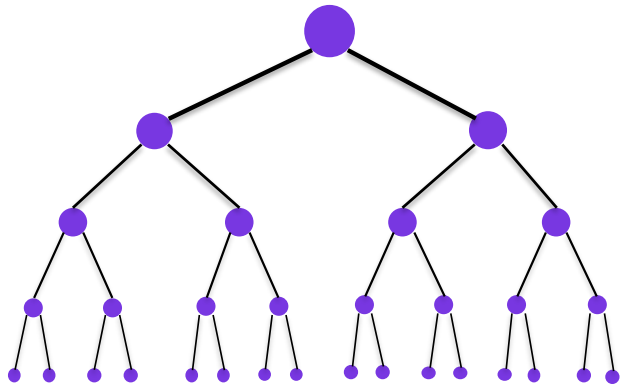
# Learning Objectives

- Review of the **Nested Parallelism Model**, **computation graphs**, and **Brent's Theorem**
- Scheduling parallel machines via the **pebble game**
- Nearly optimal scheduling via **greedy scheduling**
- Work, Span and **Parallelism**

# Nested Parallelism Model & Brent's Theorem

# Nested Parallelism as an Abstraction

- Think of nested parallelism as having ***infinitely many processors***!
- A process can always fork into more, and this can continue recursively (e.g., divide-and-conquer style). No limit.



**Remark:** Deciding which instructions run on which CPU core is called *scheduling*. Your OS also has a scheduler for running processes.

- Nested parallelism allows algorithms to express parallel structure without specifying *how* computations are assigned to processors

# We've been using Work and Span. Why?

**Definition (Work):** The **work** of a parallel algorithm is sum of the costs of all instructions it executes, across all parallel branches.

***Intuition:*** Work is the cost of the algorithm on one processor (i.e. no parallelism)

**Definition (Span):** The **span** of a parallel algorithm is the cost of the longest chain of dependent computations

***Intuition:*** Span is the cost of the algorithm with infinitely many processors!

**Question:** most CPUs have neither one nor infinitely many processors… so why are these quantities useful?

# Why Work and Span?

**Question:** most CPUs have neither one nor infinitely many processors… so why are these quantities useful?

**Answer:** It turns out that by measuring the work and span, we can derive the cost of the algorithm for any number of processors!

**Theorem (Brent's Theorem):** An algorithm with work $W$ and span $S$ can be scheduled on a $P$-processor machine in $O\big(\max(W/P, S)\big)$ time

*Brent's Theorem essentially proves that nested parallelism is a **good abstraction**!*

# Realizing Brent's Theorem

**The Greedy Scheduling Theorem:** There exists a greedy scheduling algorithm that achieves the bound of Brent's Theorem.

# Scheduling

# Scheduling

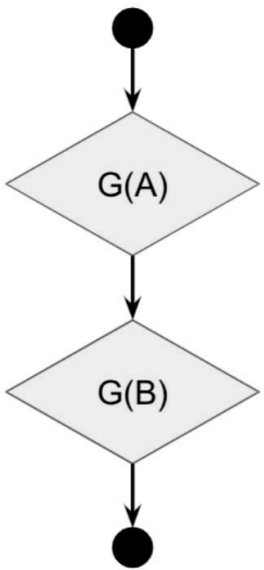
The topic of today's lecture is scheduling.

- We will show how to schedule a parallel machine.
- In the process we "prove" Brent's Theorem, and the Greedy Scheduling Theorem.
- We begin by defining the **computation graph** and **the pebble game**.

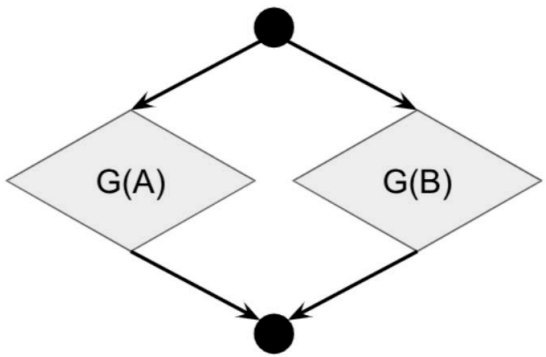
# The Computation Graph

A **DAG** that is generated by examining the code for an algorithm.

A; B



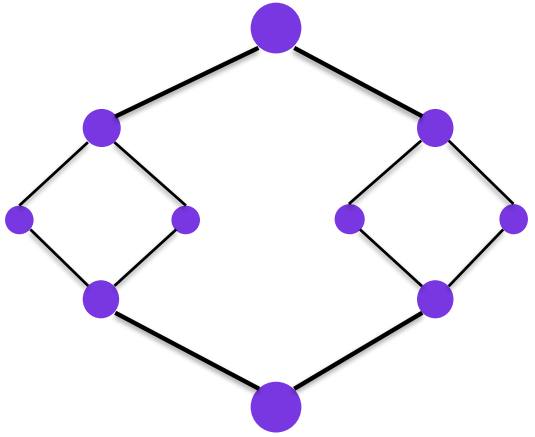

A || B



# Example: Summing A Sequence

```
fun sum(S : sequence<int>) -> int:
  match length(S) with:
    case 0: return 0
    case 1: return S[0]
    case _:
      L, R = split_mid(S)
      Lsum, Rsum = parallel (sum(L), sum(R))
      return Lsum + Rsum
```

For a sequence of length 4
This is the computation DAG



# The Computation Graph

Each node represents a constant time computation.

An edge from $x$ to y indicates that in order to compute $y$, $x$ must have already been computed. $x$ is a **predecessor** of $y$.

The number of nodes in the computation graph is the WORK.

The longest path in from START to END in the computation graph is the SPAN

# A Pebble Game

Now we turn to the problem of scheduling $p$ processors on a computation graph.

We convert the problem into what we call the pebble game.

# A Pebble Game

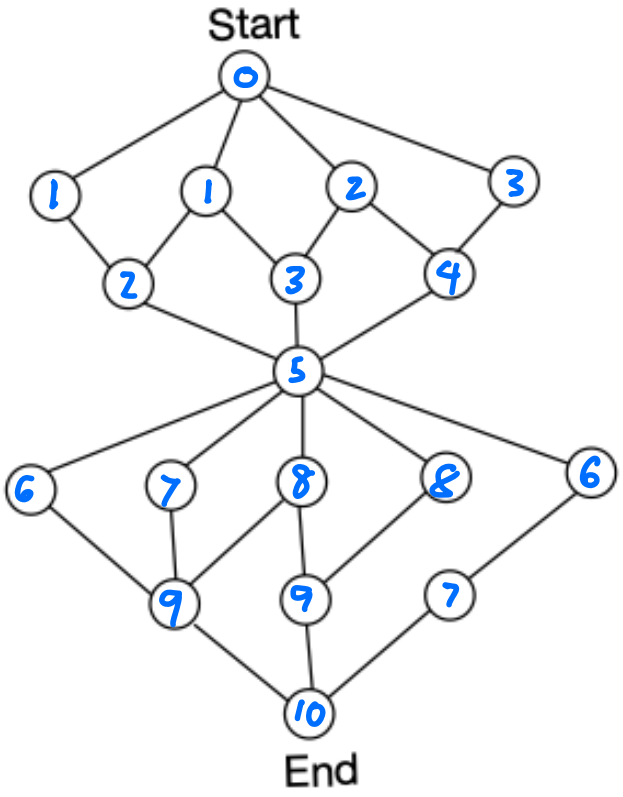
We place pebbles on the nodes in the graph until the entire graph is pebbled. A pebble represents doing the computation of the node. Placement works in **rounds,** as follows:

- Initially there is a pebble on the start node. That is round 0.
- In each subsequent round up to $p$ pebbles are simultaneously placed on nodes of the graph.
- But -- A pebble may be placed on any node in round $i$ only if **all of its predecessors were pebbled in an earlier round**.

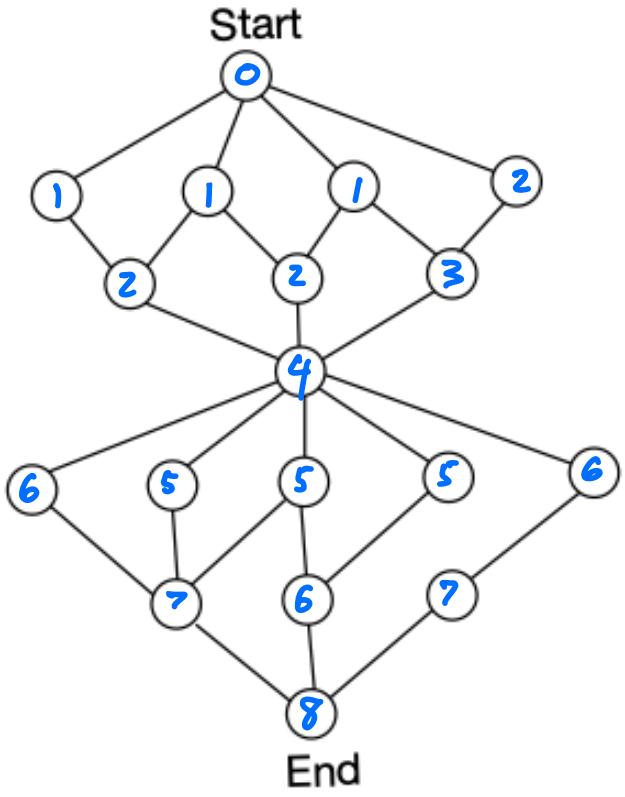
The time to do the parallel computation is represented by the number of rounds.

# A Pebble Game: Examples

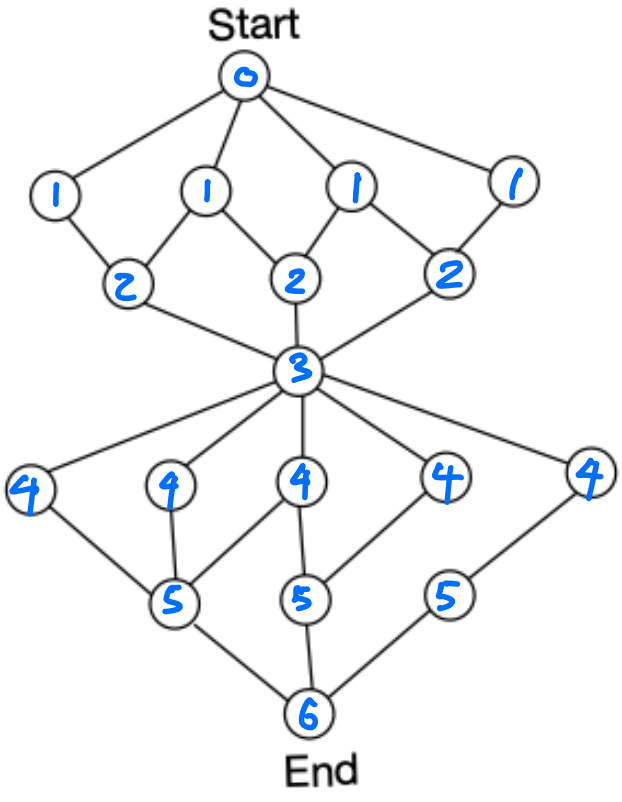
$P = 2$




$P = 3$




$P \geq 5$



# Greedy Pebbling

- For each round, the set of nodes that can receive a pebble are said to be **ready**. Let $r$ be the number of ready nodes.
- The maximum number of nodes that can be pebbled in the round is $\min(r, p)$.
- An algorithm is **greedy** if it places $\min(r, p)$ in each round.

**Theorem (You Might as well be Greedy):** Let $P$ be a pebbling process for a graph with $p$ processors that completes in $R$ rounds. Then there is a greedy pebbling process that completes in at most $R$ rounds.

**Theorem (You Might as well be Greedy):** Let $P$ be a pebbling process for a graph with $p$ processors that completes in $R$ rounds. Then there is a greedy pebbling process that completes in at most $R$ rounds.

*Proof: We can convert P into a greedy algorithm which finishes in equal or fewer rounds. We run P and the modified process M in parallel. We preserve the invariant that at any time the vertices pebbled in P is a subset of those pebbled in M.*

*Whenever P is non-greedy and places fewer pebbles than greedy would, we pebble the same nodes that P did, and then a few more to be greedy. This is legal because of the invariant, and this preserves the invariant.*

*The modified algorithm terminates no later than P.*

# Properties of Greedy

**Corollary:** There exists an optimal greedy strategy.

**Challenge:** Give an example where any greedy algorithm is not optimal.

# Lower and Upper bounds on Pebbling

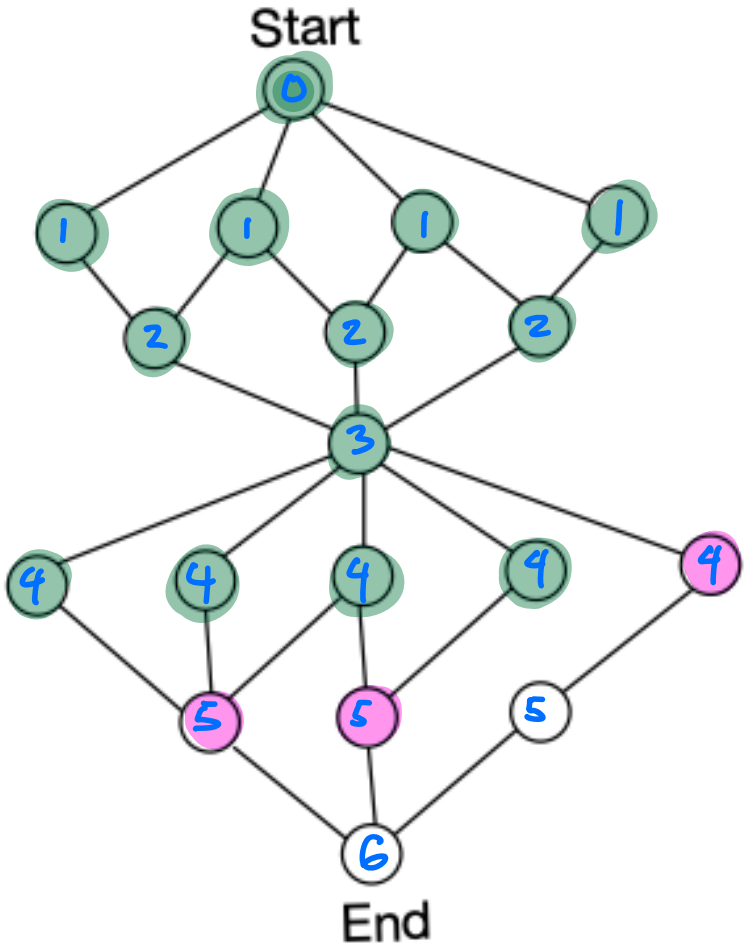
# Dividing the DAG into Layers

It's going to be useful to divide the computation graph into layers, based on what greedy pebbling would do if there were an infinite number of processors.

- $L_0$ is the usual **start** node.
- $L_i$ is the set of nodes not in layers $L_0, L_1, \ldots, L_{i-1}$ whose predecessors are all in those layers.
- Alternatively $L_i$ is those nodes $x$ such that the longest path from the start node to $x$ is of length $i$.
- This partitions the graph into a sequence of layers $L_0, L_1, \ldots, L_d$

# Dividing the DAG into Layers

Layer numbers shown in blue




● ← Pebbeling state with p = 4

j = 4 is the lowest incompletly Pebbled level.

● ← Next round. It only pebbles 3 < p nodes. And completes level 4.

# Lower Bound on Pebbling

**Theorem (Lower Bound on Pebbling):** At least $\max\left(\left\lceil\frac{n}{p}\right\rceil, d\right)$ rounds are necessary to pebble a graph with $n$ nodes and $d$ layers

*Proof:* At least $\left\lceil\frac{n}{p}\right\rceil$ rounds are needed to have enough pebbles to cover the whole graph.

There exists a path of length $d$, therefore at least at least $d$ rounds are needed.

QED

# Upper Bound on Pebbling

**Theorem (Greedy is Good):** Any greedy strategy will take at most $\frac{n}{p} + d$ rounds.

*Proof:* Divide the graph into levels as above. On every round at least one of the following two events occur:

( 1 ) $p$ pebbles are placed

( 2 ) The last pebble on a level is placed.

(see the sample run on page 22.)

Let $j$ be the lowest numbered level with an unpebbled node in it. All nodes of level $j$ are ready. If ( 1 ) does not happen then greedy must have exhausted all the ready nodes. Therefore it must have completed level $j$.

(1) happens at most $n/p$ times and (2) happens at most $d$ times. QED

# Greedy is within a factor of 2

Note that $a + b \leq 2\max(a, b)$ for all positive $a$ and $b$.

Therefore $\frac{n}{p} + d \leq 2\max\left(\left\lceil\frac{n}{p}\right\rceil, d\right)$ So greedy is within a factor of 2 of the lower bound.

# Span, Work, and Parallelism

# Span, Work and Parallelism

The span of a computation $S$ corresponds to $d$ , and the work $W$ corresponds to $n$. So these results prove Brent's theorem, and the Greedy Scheduling Theorem.

**Theorem (Brent's Theorem):** An algorithm with work $W$ and span $S$ can be scheduled on a $P$-processor machine in $O\big(\max(W/P, S)\big)$ time

**The Greedy Scheduling Theorem:** There exists a greedy scheduling algorithm that achieves the bound of Brent's Theorem.

# Span, Work and Parallelism

So our results say that

$$\text{time} = \Theta\left(\frac{W}{p} + S\right)$$

At what point does adding more processors begin to have diminishing returns? When these two terms are about equal, or

$$p \simeq \frac{W}{S}$$

This quantity is therefore called the **parallelism** of the algorithm.

Roughly speaking it's about how many processors can be usefully utilized to speed up the algorithm.

This is why we strive to reduce SPAN!

# Summary

- We show how to reduce the scheduling problem for parallel algorithms to a **pebble game** on graphs.
- We show how efficient scheduling can be achieved by a simple greedy algorithm.