A Sophomoric Introduction to Shared-Memory Parallelism and Concurrency

Lecture 2
Analysis of Fork-Join Parallel Programs

Steve Wolfman, based on work by Dan Grossman
(with small tweaks by Alan Hu)

Learning Goals

- Define work—the time it would take one processor to complete a parallelizable computation; span—the time it would take an infinite number of processors to complete the same computation; and Amdahl's Law—which relates the speedup in a program to the proportion of the program that is parallelizable.
- Use work, span, and Amdahl's Law to analyse the speedup available for a particular approach to parallelizing a computation.
- Judge appropriate contexts for and apply the parallel map, parallel reduce, and parallel prefix computation patterns.

Outline

Done:

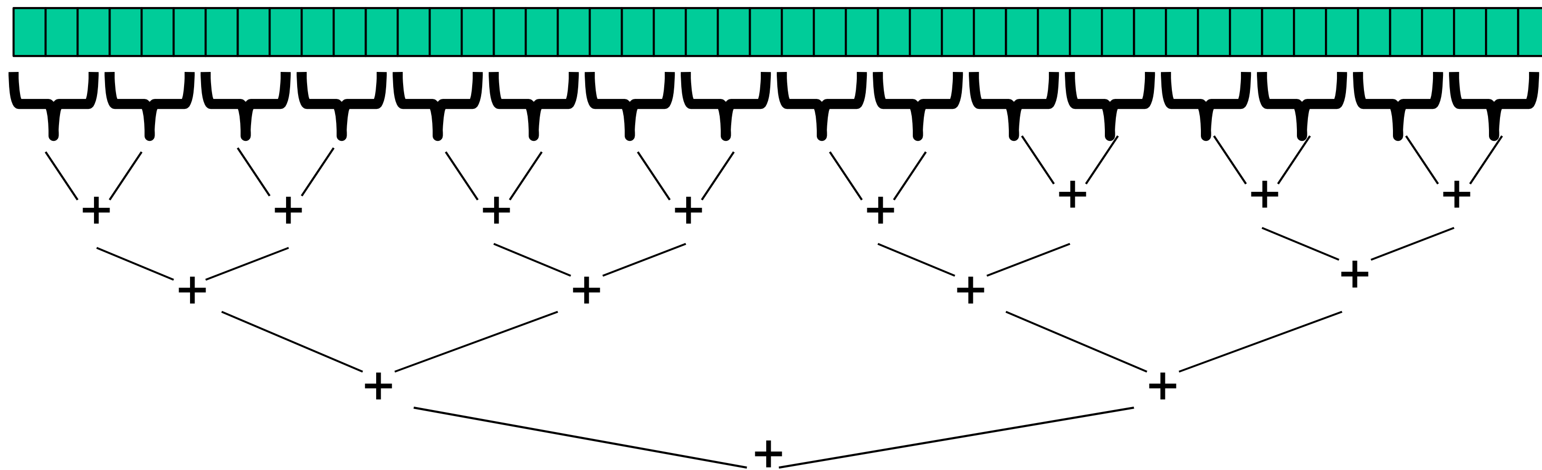
- How to use **`fork`** and **`join`** to write a parallel algorithm
- Why using divide-and-conquer with lots of small tasks is best
  - Combines results in parallel
- Some C++11 and OpenMP specifics
  - More pragmatics (e.g., installation) in separate notes

Now:

- More examples of simple parallel programs
- Other data structures that support parallelism (or not)
- Asymptotic analysis for fork-join parallelism
- Amdahl's Law

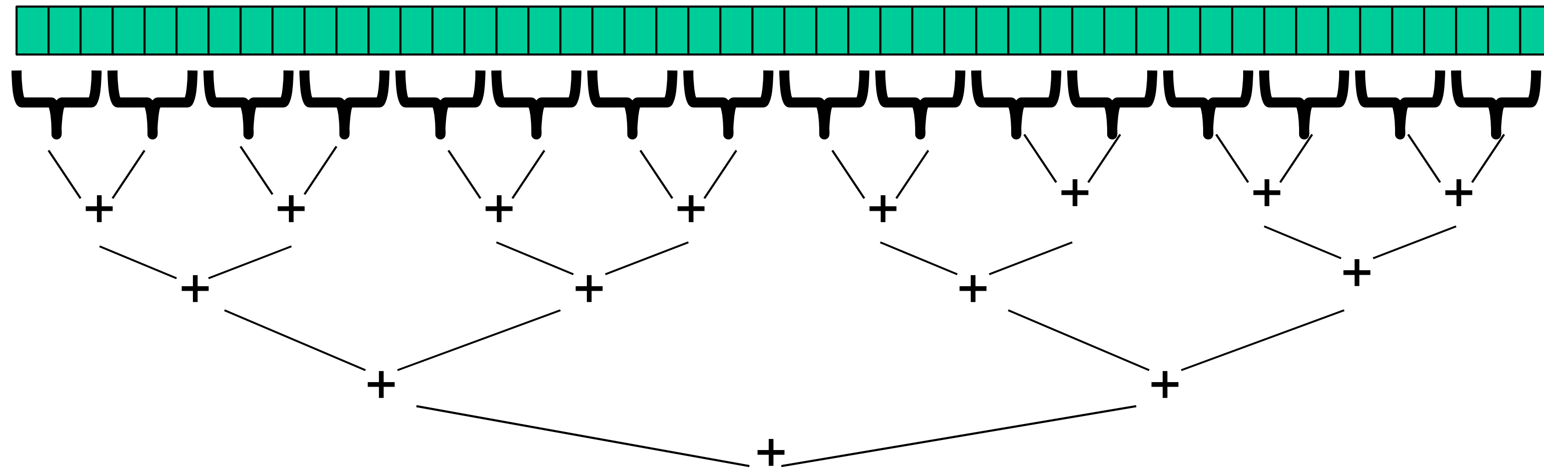
“Exponential speed-up” using Divide-and-Conquer

- Counting matches (lecture) and summing (reading) went from $O(n)$ sequential to $O(\texttt{log}\ n)$ parallel (*assuming **lots** of processors!*)
  - An exponential speed-up in theory (*in what sense?*)



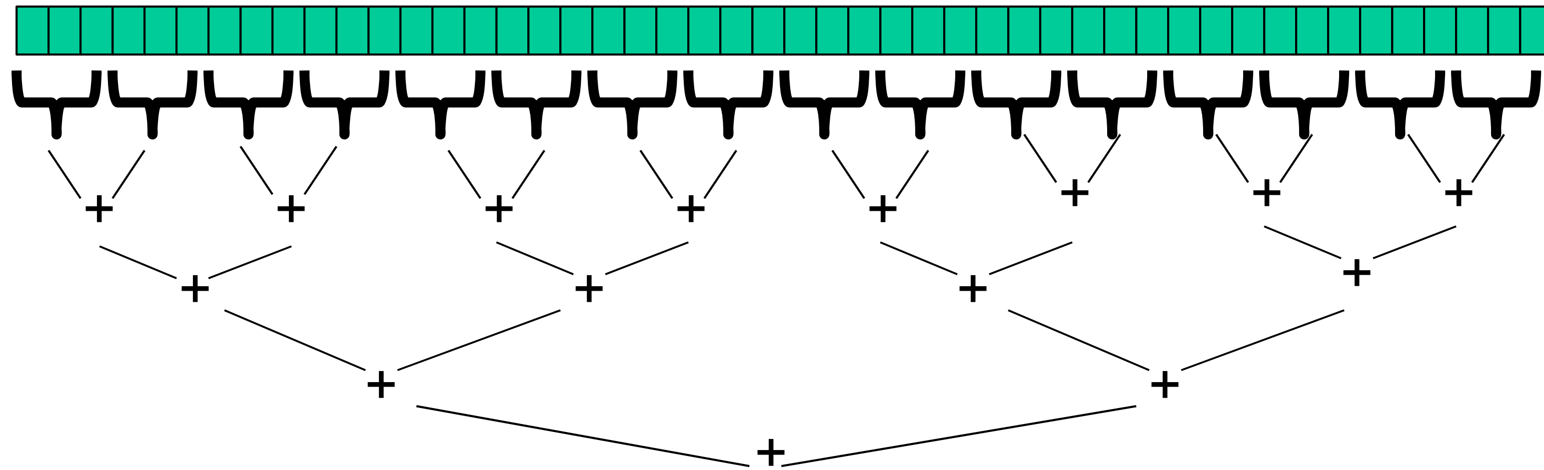
- Many other operations can also use this structure…

Other Operations?



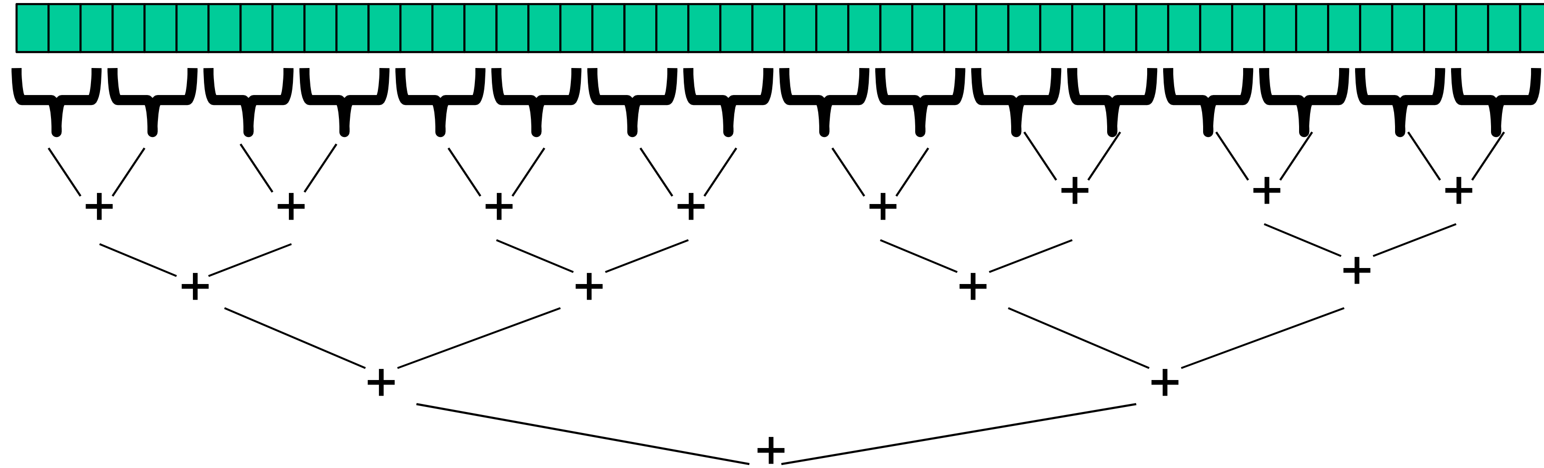
What's an example of something else we can put at the "+" marks?

What else looks like this?



What's an example of something we ***cannot*** put there (and have it work the same as a for loop would)?

Reduction:
a single answer aggregated from a list



What are the basic requirements for the reduction operator?

Note: The “single” answer can be a list or other collection.

Is Counting Matches Really a Reduction?

Count matches:

```
FORALL array elements:
  score = (if element == target then 1 else 0)
  total_score += score
```

Is this “really” a reduction?

Even easier parallel operation: Maps (AKA "Data Parallelism")

- A map operates on each element of a collection independently to create a new collection of the same size
  - No combining results
  - For arrays, this is so trivial some hardware has direct support!
  - You've also seen this in CPSC 110 in Racket
- Typical example: Vector addition

```
void vector_add(int result[], int left[], int right[], int len) {
  FORALL(i=0; i < len; i++) {
    result[i] = left[i] + right[i];
  }
}
```

Even easier parallel operation: Maps (AKA "Data Parallelism")

- A map operates on each element of a collection independently to create a new collection of the same size
  - No combining results
  - For arrays, this is so trivial some hardware has direct support!
  - You've also seen this in CPSC 110 in Racket
- Typical example: Vector addition

```
void vector_add(int result[], int left[], int right[], int len) {
  FORALL(i=0; i < len; i++) {
    result[i] = arr1[i] + arr2[i];
  }
}
```

This is **pseudocode** in the notes for a for loop where the iterations can go in parallel.

Maps in OpenMP *(w/explicit Divide & Conquer)*

```
void vector_add(int result[], int left[], int right[],
                int lo, int hi)
{
  const int SEQUENTIAL_CUTOFF = 1000;
  if (hi - lo <= SEQUENTIAL_CUTOFF) {
    for (int i = lo; i < hi; i++)
      result[i] = left[i] + right[i];
    return;
  }

#pragma omp task untied shared(result, left, right)
  vector_add(result, left, right, lo, lo + (hi-lo)/2);

  vector_add(result, left, right, lo + (hi-lo)/2, hi);
#pragma omp taskwait
}
```

Aside: Maps in OpenMP (w/ parallel for)

```
void vector_add(int result[], int left[], int right[],
                  int len) {
  #pragma omp parallel for
  for (int i=0; i < len; i++) {
    result[i] = left[i] + right[i];
  }
}
```

Maps are so common, OpenMP has built-in support for them.
(But the point of this class is to learn how the algorithms work.)

Even easier: Maps (Data Parallelism)

- A map operates on each element of a collection independently to create a new collection of the same size
  - No combining results
  - For arrays, this is so trivial some hardware has direct support
- **One we already did**: counting matches becomes mapping "number $\rightarrow$ 1 if it matches, else 0" and then reducing with +

```
void equals_map(int result[], int array[], int len, int target) {
  FORALL(i=0; i < len; i++) {
    result[i] = (array[i] == target) ? 1 : 0;
  }
}
```

Maps and reductions

These are by far the two most important and common patterns.

You should learn to recognize when an algorithm can be written in terms of maps and reductions because they make parallel programming simple…

Exercise: find the ten largest numbers

Given an array of positive integers, return the ten largest in the list.

How is this a map and/or reduce?

Exercise: count prime numbers

Given an array of positive integers, count the number of prime numbers.

How is this a map and/or reduce?

Exercise: find first substring match

Given a (small) substring and a (large) text, find the index where the first occurrence of the substring starts in the text.

How is this a map and/or reduce?

Digression: MapReduce on clusters

You may have heard of Google's "map/reduce" or the open-source version Hadoop

- Idea: Perform maps/reduces on data using many machines
  - The system takes care of distributing the data and managing fault tolerance
  - You just write code to map one element and reduce elements to a combined result
- Separates how to do recursive divide-and-conquer from what computation to perform
  - Old idea in higher-order functional programming transferred to large-scale distributed computing
  - Complementary approach to declarative queries for databases

Outline

Done:

- How to use `fork` and `join` to write a parallel algorithm
- Why using divide-and-conquer with lots of small tasks is best
  - Combines results in parallel
- Some C++11 and OpenMP specifics
  - More pragmatics (e.g., installation) in separate notes

Now:

- More examples of simple parallel programs
- Other data structures that support parallelism (or not)
- Asymptotic analysis for fork-join parallelism
- Amdahl's Law

On What Other Structures Can We Use Divide-and-Conquer Map/Reduce?

- A linked list?
- A binary tree?
  - Any?
  - Heap?
  - Binary search tree?
    - AVL?
    - B+?
- A hash table?

Outline

Done:

- How to use `fork` and `join` to write a parallel algorithm
- Why using divide-and-conquer with lots of small tasks is best
  - Combines results in parallel
- Some C++11 and OpenMP specifics
  - More pragmatics (e.g., installation) in separate notes

Now:

- More examples of simple parallel programs
- Other data structures that support parallelism (or not)
- Asymptotic analysis for fork-join parallelism
- Amdahl's Law

Analyzing algorithms

We'll set aside analyzing for correctness for now.
(Maps are obvious? Reductions are correct if the operator is associative?)

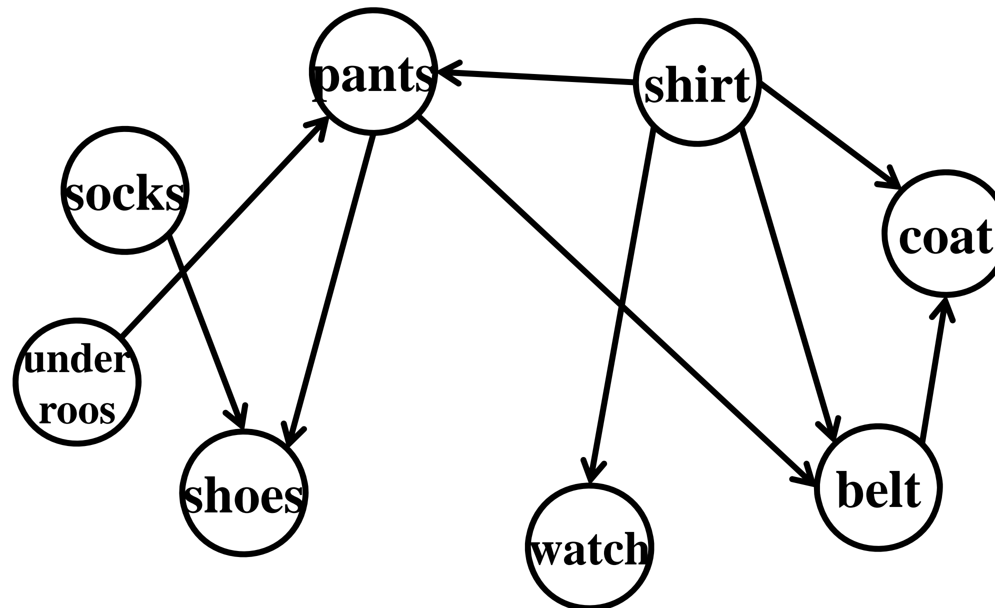
How do we analyze the *efficiency* of our parallel algorithms?

– We want asymptotic bounds
– We want to analyze the algorithm without regard to a specific number of processors

Note: a good OpenMP implementation does some "magic" to get expected run-time performance asymptotically optimal for the available number of processors.

So, we get to assume this guarantee.

Digression, Getting Dressed (1)

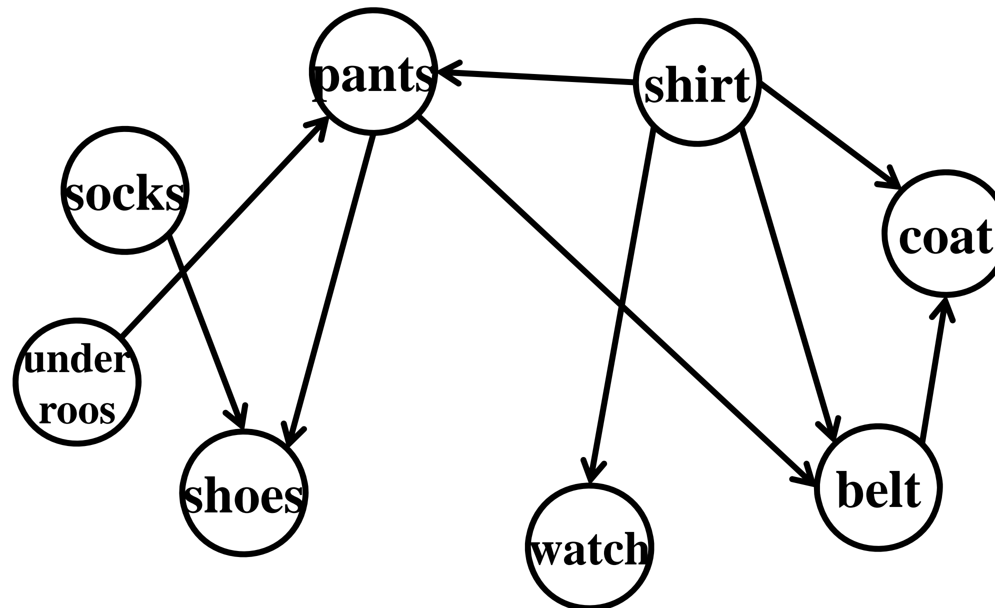


Assume it takes me 5 seconds to put on each item, and I cannot put on more than one item at a time:

How long does it take me to get dressed?

(Note: costs are on nodes, not edges.)

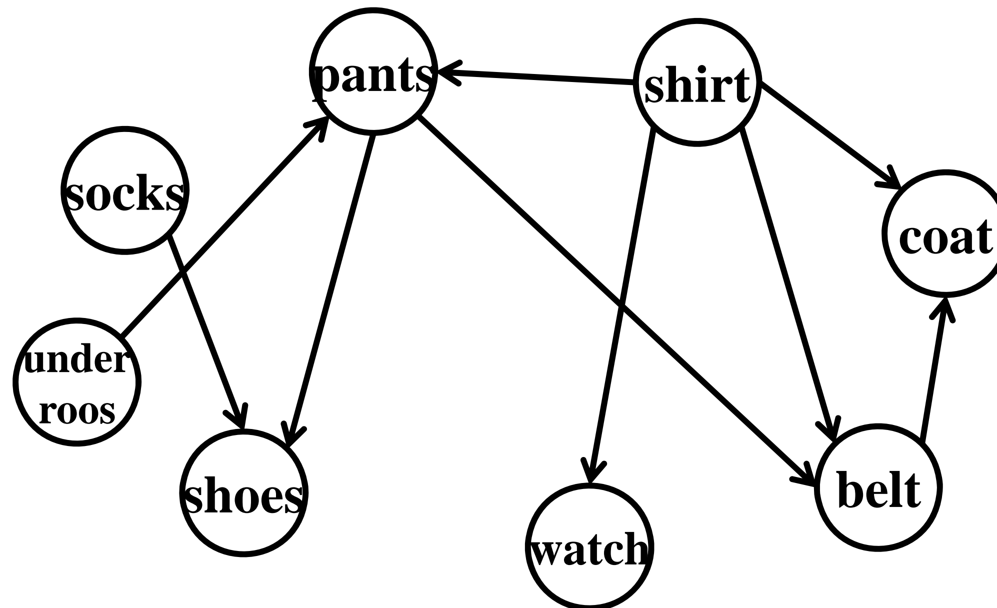
Digression, Getting Dressed (∞)



Assume it takes my robotic wardrobe 5 seconds to put me into each item, and it can put on up to 20 items at a time.

How long does it take me to get dressed?

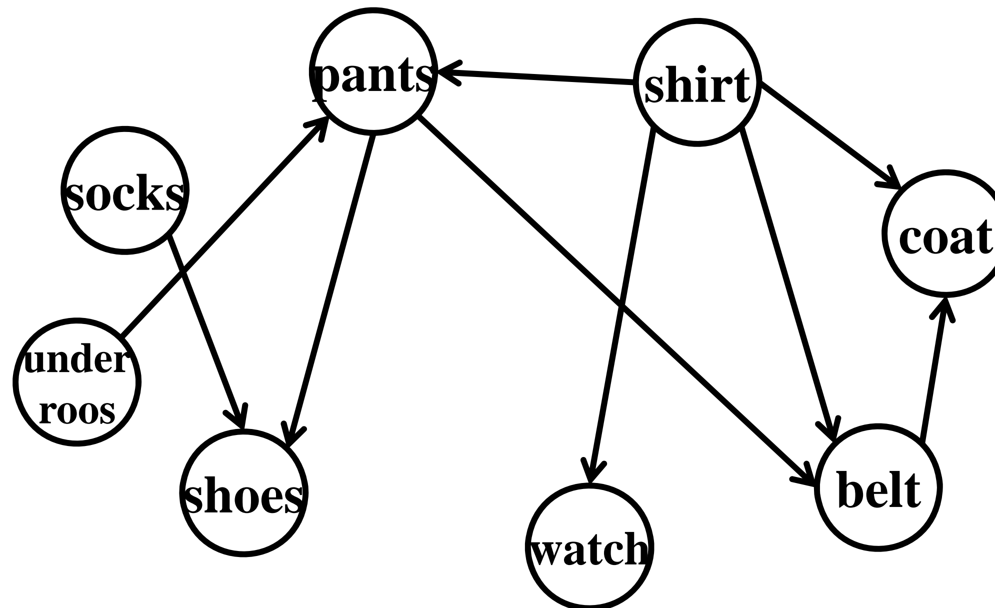
Digression, Getting Dressed (2)



Assume it takes me 5 seconds to put on each item, and I can use my two hands to put on 2 items at a time.

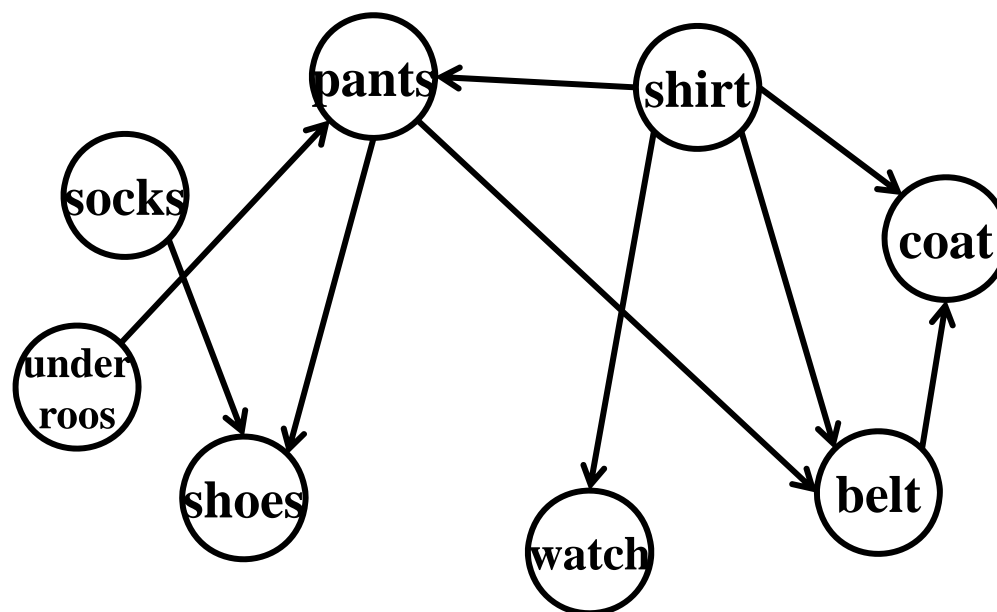
How long does it take me to get dressed?

Un-Digression, Getting Dressed: "Work"



What mattered when I could put only one item on at a time?
How do we count it?

Un-Digression, Getting Dressed: "Span"



What mattered when I could an infinite number of items on at a time?
How do we count it?

Work and Span

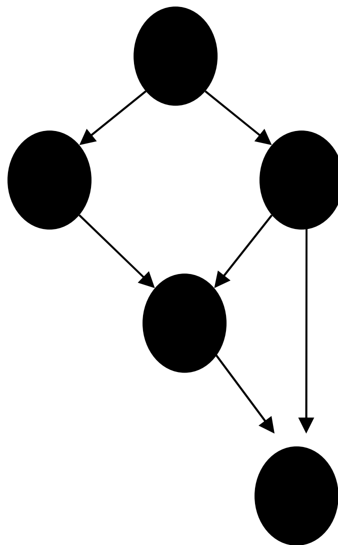
Let $\mathbf{T_P}$ be the running time if there are **P** processors available

Two key measures of run-time:

- Work: How long it would take 1 processor = $\mathbf{T_1}$
  - Just "sequentialize" the recursive forking
- Span: How long it would take infinity processors = $\mathbf{T_\infty}$
  - Example: $O(\texttt{log}\ n)$ for summing an array
    - Notice having > $n/2$ processors is no additional help

The DAG

- A program execution using **`fork`** and **`join`** can be seen as a DAG
  - Nodes: Pieces of work
  - Edges: Source must finish before destination starts



- A **`fork`** "ends a node" and makes two outgoing edges
  - New thread
  - Continuation of current thread
- A **`join`** "ends a node" and makes a node with two incoming edges
  - Node just ended
  - Last node of thread joined on

So, what is T_∞ in the graph?

Our simple examples

- **`fork`** and **`join`** are very flexible, but divide-and-conquer maps and reductions use them in a very basic way:
  - A tree on top of an upside-down tree

divide

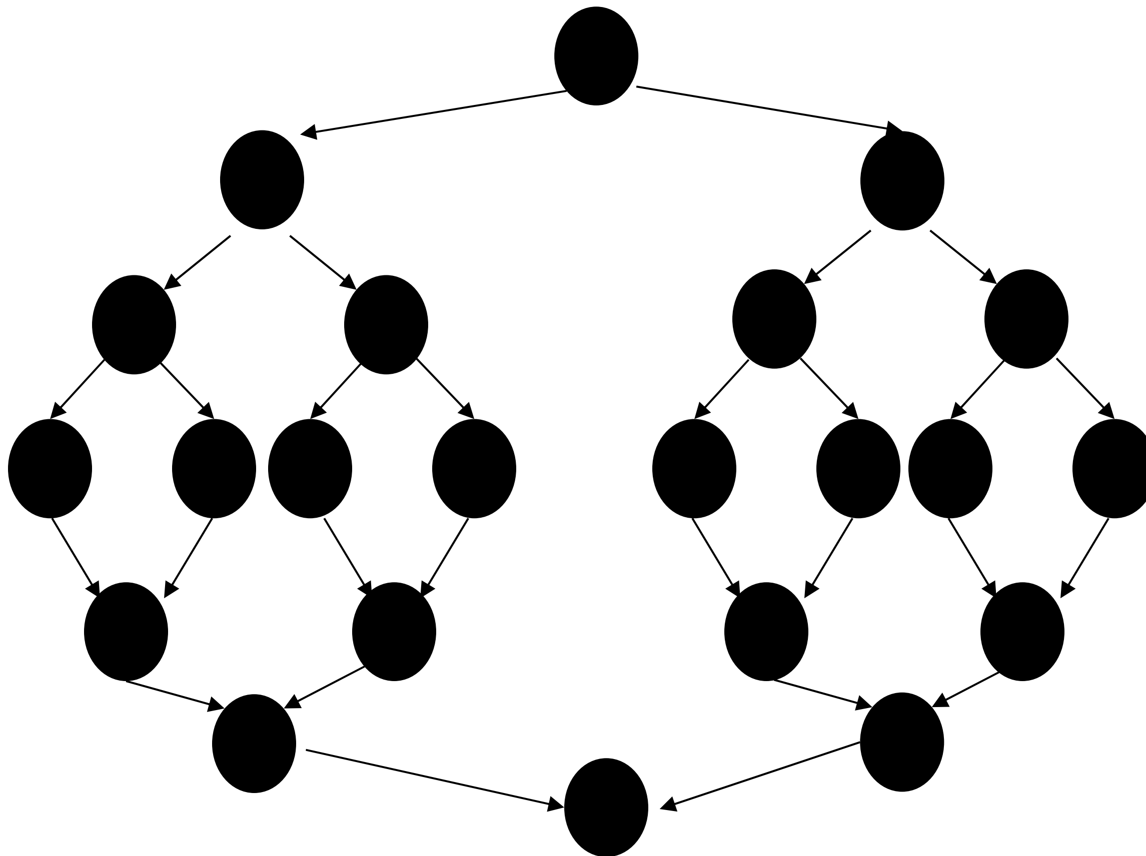
base cases

combine results

(Note: costs are on nodes, not edges.)

Map/Reduce DAG: Work?

What's the *work* in this DAG?



Map/Reduce DAG: Work?

What's the *span* in this DAG?

More interesting DAGs?

- The DAGs are not always this simple

- Example:
  - Suppose combining two results might be expensive enough that we want to parallelize each one
  - Then each node in the inverted tree on the previous slide would itself expand into another set of nodes for that parallel computation

Connecting to performance

- Recall: $\mathbf{T_P}$ = running time if there are **P** processors available
- Work = $\mathbf{T_1}$ = sum of run-time of all nodes in the DAG
  - That lonely processor does everything
  - Any topological sort is a legal execution
  - $O(n)$ for simple maps and reductions
- Span = $\mathbf{T_\infty}$ = sum of run-time of all nodes on the most-expensive path in the DAG
  - Note: costs are on the nodes not the edges
  - Our infinite army can do everything that is ready to be done, but still has to wait for earlier results
  - $O(\texttt{log}\ n)$ for simple maps and reductions

Definitions

A couple more terms:

- Speed-up on **P** processors: $\mathbf{T_1 / T_P}$
- If speed-up is **P** as we vary **P**, we call it perfect linear speed-up
  - Perfect linear speed-up means doubling **P** halves running time
  - Usually our goal; hard to get in practice
- Parallelism is the maximum possible speed-up: $\mathbf{T_1 / T_\infty}$
  - At some point, adding processors won't help
  - What that point is depends on the span

Parallel algorithms is about decreasing span without increasing work too much

Asymptotically Optimal T_P

- So we know $\mathbf{T_1}$ and $\mathbf{T_\infty}$ but we want $\mathbf{T_P}$ (e.g., **P**=4)
- Ignoring memory-hierarchy issues (caching), $\mathbf{T_P}$ can't beat
  - $\mathbf{T_1 / P}$ why not?
  - $\mathbf{T_\infty}$ why not?

Asymptotically Optimal T_P

- So we know $\mathbf{T_1}$ and $\mathbf{T_\infty}$ but we want $\mathbf{T_P}$ (e.g., **P**=4)
- Ignoring memory-hierarchy issues (caching), $\mathbf{T_P}$ can't beat
  - $\mathbf{T_1 / P}$
  - $\mathbf{T_\infty}$
- So an ***asymptotically*** optimal execution would be:

$$\mathbf{T_P = \mathit{O}((T_1 / P) + T_\infty)}$$

  - First term dominates for small **P**, second for large **P**
- Good OpenMP implementations give an *expected-time guarantee* of asymptotically optimal!
  - Expected time because it flips coins when *scheduling*
  - How? Beyond our current scope
  - Guarantee requires a few assumptions about your code…

Division of responsibility

- Our job as OpenMP users:
  - Pick a good algorithm
  - Write a program. When run, it creates a DAG of things to do
  - *Make all the nodes small-ish and (very) approximately equal amount of work*

- The framework-implementer's job:
  - Assign work to available processors to avoid idling
  - Keep constant factors low
  - Give the expected-time optimal guarantee assuming framework-user did their job

$$T_P = O((T_1 / P) + T_\infty)$$

Examples

$$T_P = O((T_1 / P) + T_\infty)$$

- In the algorithms seen so far (e.g., sum an array):
  - $T_1 = O(n)$
  - $T_\infty = O(\texttt{log}\ n)$
  - So expect (ignoring overheads): $T_P = O(n/P + \texttt{log}\ n)$

- Suppose instead:
  - $T_1 = O(n^2)$
  - $T_\infty = O(n)$
  - So expect (ignoring overheads): $T_P = O(n^2/P + n)$

Examples

$$T_P = O((T_1 / P) + T_\infty)$$

- Let's say we did our first-stab loop version (a main loop that forks off a bunch of workers). What would be the best choice of number of pieces? Think in terms of…
  - T_1 =
  - T_∞=
  - So expect (ignoring overheads): T_P =

Outline

Done:

- How to use `fork` and `join` to write a parallel algorithm
- Why using divide-and-conquer with lots of small tasks is best
  - Combines results in parallel
- Some C++11 and OpenMP specifics
  - More pragmatics (e.g., installation) in separate notes

Now:

- More examples of simple parallel programs
- Other data structures that support parallelism (or not)
- Asymptotic analysis for fork-join parallelism
- Amdahl's Law

Amdahl's Law (mostly bad news)

- So far: analyze parallel programs in terms of work and span
- In practice, typically have parts of programs that parallelize well…
  - Such as maps/reductions over arrays and trees

…and parts that don't parallelize at all

  - Such as reading a linked list, getting input, doing computations where each needs the previous step, etc.

"Nine women can't make a baby in one month"

Amdahl's Law (mostly bad news)

Let the ***work*** (time to run on 1 processor) be 1 unit time

Let **S** be the portion of the execution that can't be parallelized

Then: $\mathbf{T_1 = S + (1-S) = 1}$

Suppose we get perfect linear speedup *on the parallel portion*

Then: $\mathbf{T_P = S + (1-S)/P}$

So the overall speedup with **P** processors is (Amdahl's Law):

$\mathbf{T_1 / T_P =}$

How good/bad is this?

Amdahl's Law (mostly bad news)

Let the ***work*** (time to run on 1 processor) be 1 unit time

Let **S** be the portion of the execution that can't be parallelized

Then: $\mathbf{T_1 = S + (1-S) = 1}$

Suppose we get perfect linear speedup *on the parallel portion*

Then: $\mathbf{T_P = S + (1-S)/P}$

So the overall speedup with **P** processors is (Amdahl's Law):

$\mathbf{T_1 / T_P =}$

And the parallelism (infinite processors) is:

$\mathbf{T_1 / T_\infty =}$

Why such bad news

$T_1 / T_P = 1 / (S + (1\text{-}S)/P)$ $T_1 / T_\infty = 1 / S$

- Suppose 33% of a program is sequential
  - How much speed-up do you get from 2 processors?
  - How much speep-up do you get from 1,000,000 processors?

Why such bad news

$T_1 / T_P = 1 / (S + (1\text{-}S)/P)$ $T_1 / T_\infty = 1 / S$

- Suppose 33% of a program is sequential
  - How much speed-up do you get from 2 processors?
  - How much speep-up do you get from 1,000,000 processors?
- Suppose you miss the good old days (1980-2005) where 12ish years was long enough to get 100x speedup
  - Now suppose in 12 years, clock speed is the same but you get 256 processors instead of 1
  - For 256 processors to get at least 100x speedup, we need

$$100 \leq 1 / (\mathbf{S} + (1\text{-}\mathbf{S})/256)$$

What do we need for **S**?

Plots you have to see

1. Assume 256 processors
   - x-axis: sequential portion **S**, ranging from .01 to .25
   - y-axis: speedup $\mathbf{T_1 / T_P}$ (will go down as **S** increases)

2. Assume **S** = .01 or .1 or .25 (three separate lines)
   - x-axis: number of processors **P**, ranging from 2 to 32
   - y-axis: speedup $\mathbf{T_1 / T_P}$ (will go up as **P** increases)

Do this!

- Chance to use a spreadsheet or other graphing program
- Compare against your intuition
- A picture is worth 1000 words, especially if you made it

All is not lost

Amdahl's Law is a bummer!

- But it doesn't mean additional processors are worthless

- We can find new parallel algorithms
  - Some things that seem sequential are actually parallelizable

- Can change the problem we're solving or do new things
  - Example: Video games use tons of parallel processors
    - They are not rendering 10-year-old graphics faster
    - They are rendering more beautiful(?) monsters

Moore and Amdahl

- Moore's "Law" is an observation (based on physics, engineering, economics, and technology trends) about the progress of the semiconductor industry
  - Transistors per chip double roughly every 18 months
- Amdahl's Law is a mathematical theorem
  - Diminishing returns of adding more processors
- Both are incredibly important in designing computer systems

Learning Goals

- Define work—the time it would take one processor to complete a parallelizable computation; span—the time it would take an infinite number of processors to complete the same computation; and Amdahl's Law—which relates the speedup in a program to the proportion of the program that is parallelizable.
- Use work, span, and Amdahl's Law to analyse the speedup available for a particular approach to parallelizing a computation.
- Judge appropriate contexts for and apply the parallel map, parallel reduce, and parallel prefix computation patterns.