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

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.

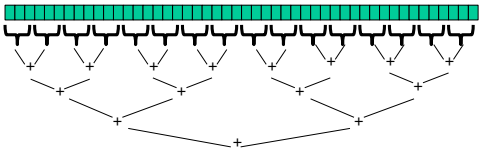
Outline

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

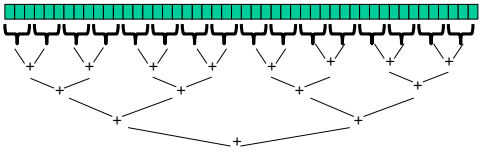
"Exponential speed-up" using Divide-and-Conquer

- Counting matches (lecture) and summing (reading) went from $O(n)$ sequential to $O(\log n)$ parallel (*assuming **lots** of processors!*)
  - An exponential speed-up (or more like: the sequential version represents an exponential ***slow-down***)



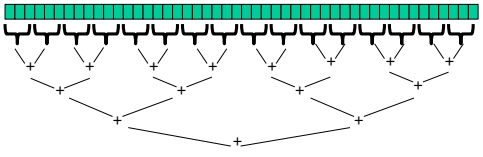
- Many other operations can also use this structure…

Other Operations?



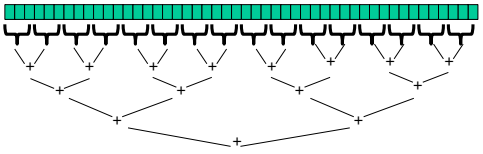
What an example of something else we can put at the "+" marks?

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: 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 → 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;
  }
}
```


Another Map Example: Vector Addition

Note: if <1, 2, 3, 4, 5> and <2, 5, 3, 3, 1> are vectors, then their "vector sum" is the sum of corresponding elements: <3, 7, 6, 7, 6>.

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


Maps in OpenMP (w/explicit Divide & Conquer)

```
void vector_add(int result[], int arr1[], int arr2[],
                int lo, int hi)
{
  const int SEQUENTIAL_CUTOFF = 1000;
  if (hi - lo <= SEQUENTIAL_CUTOFF) {
    for (int i = lo; i < hi; i++)
      result[i] = arr1[i] + arr2[i];
    return;
  }

#pragma omp task untied
{
  vector_add(result, arr1, arr2, lo, lo + (hi-lo)/2);
}

  vector_add(result, arr1, arr2, lo + (hi-lo)/2, hi);
#pragma omp taskwait   // (Not totally necessary here.)
}
```


Maps and reductions

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

Learn to recognize when an algorithm can be written in terms of maps and reductions! They make parallel programming simple…

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

- system distributes the data and manages fault tolerance
- your code just operates on one element (map) or combines two elements (reduce)
- old functional programming idea → big data/distributed computing

What is specifically possible in a Hadoop "map/reduce" is more general than the examples we've so far seen.

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 largest

Given an array of positive integers, find the largest number.

How is this a map and/or reduce?

*Exercise: find largest **AND** smallest*

Given an array of positive integers, find the largest and the smallest number.

How is this a map and/or reduce?

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: 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?

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

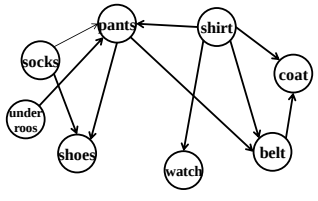
Analyzing Parallel Algorithms

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 our bound to incorporate the number of processors (so we know how our algorithm will improve as chips do!)

But… how hard is it to reason about the number of processors?

Digression, Getting Dressed

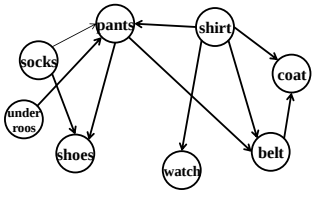


Here's a graph representation for parallelism.

Nodes: (small) tasks that are *potentially* executable in parallel
Edges: dependencies (the target of the arrow depends on its source)

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

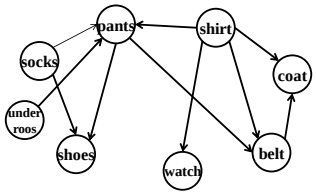
Digression, Getting Dressed (1)



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?

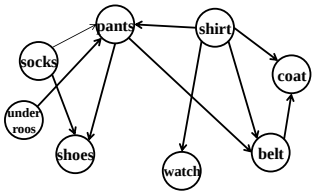
Digression, Getting Dressed (∞)



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?

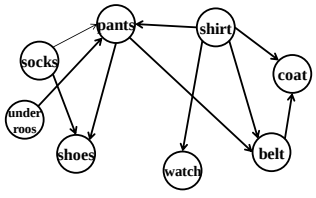
Digression, Getting Dressed (2)



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.

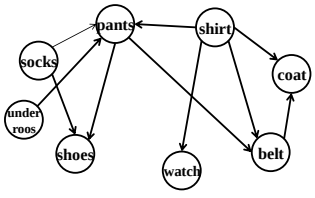
How long does it take me to get dressed?

Un-Digression, Getting Dressed: "Work", AKA T_1



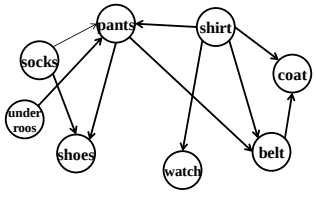
What mattered when I could put only one item on at a time?
How do we count it?

Un-Digression, Getting Dressed: "Span", AKA T_∞



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

Un-Digression, Getting Dressed: Performance for P processors, AKA T_P



What mattered when I could put on 2 items on at a time?
Was it as easy as work or span to calculate?

Asymptotically Optimal T_P

T_1 and T_∞ are easy, but we want to understand T_P in terms of **P**

But… can T_P beat:

- T_1 / P why or why not?
- T_∞ why or why not?

Various issues (e.g., memory-hierarchy) can throw this off. We'll ignore that!

Asymptotically Optimal T_P

T_1 and T_∞ are easy, but we want to understand T_P in terms of **P**

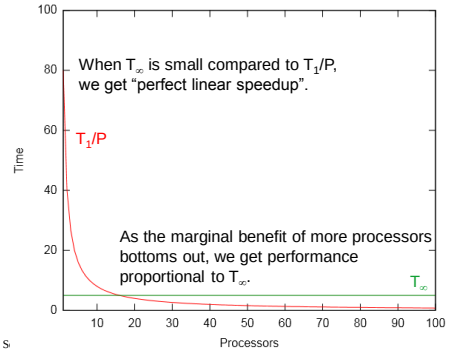
But… T_P cannot beat:
- T_1 / P because otherwise we didn't do all the work!
- T_∞ because we still don't have have ∞ processors!

So an ***asymptotically*** optimal execution would be:

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

- First term dominates for small **P**, second for large **P**

Asymptotically Optimal T_P



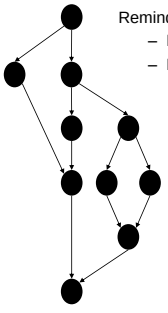
Getting an Asymptotically Optimal Bound

Good OpenMP implementations *guarantee* $O((T_1 / P) + T_\infty)$ as their *expected* bound. (Expected b/c of use of randomness.)

Or they do ***if*** we do our job as OpenMP users:
- Pick a good algorithm
- Make each node's work "small", constant, and *very roughly* equal

(Better yet… use OpenMP's **built-in** reductions, parallel for loops, etc.
But… understand how they work from this course!)

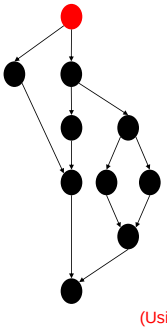
Analyzing Code, Not Clothes



Reminder, in our DAG representation:
- Each node: one piece of *constant-sized* work
- Each edge: source must finish before destination starts

Cost is per *node* not edge.
Say each node has 1 unit cost.
What is T_∞ in this graph?

Where the DAG Comes From



pseudocode

```
main:
  a = fork task1
  b = fork task2
  O(1) work
  join a
  join b

task1:
  O(1) work

task2:
  c = fork task1
  O(1) work
  join c
```

C++11

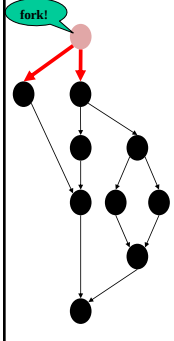
```
int main(..) {
  std::thread t1(&task1);
  std::thread t2(&task2);
  // O(1) work
  t1.join();
  t2.join();
  return 0;
}

void task1() {
  // O(1) work
}

void task2() {
  std::thread t(&task1);
  // O(1) work
  t.join();
}
```

We start with just one thread.
(Using C++11 not OpenMP syntax to make things cleaner.)

Where the DAG Comes From




```
main:
  a = fork task1
  b = fork task2
  O(1) work
  join a
  join b

task1:
  O(1) work

task2:
  c = fork task1
  O(1) work
  join c
```

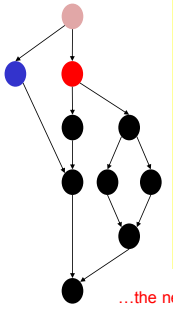
```
int main(..) {
  std::thread t1(&task1);
  std::thread t2(&task2);
  // O(1) work
  t1.join();
  t2.join();
  return 0;
}

void task1() {
  // O(1) work
}

void task2() {
  std::thread t(&task1);
  // O(1) work
  t.join();
}
```

A fork ends a node and generates two new ones…

Where the DAG Comes From



```
main:
  a = fork task1
  b = fork task2
  O(1) work
  join a
  join b

task1:
  O(1) work

task2:
  c = fork task1
  O(1) work
  join c
```

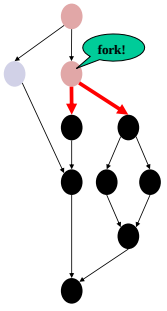
```
int main(..) {
  std::thread t1(&task1);
  std::thread t2(&task2);
  // O(1) work
  t1.join();
  t2.join();
  return 0;
}

void task1() {
  // O(1) work
}

void task2() {
  std::thread t(&task1);
  // O(1) work
  t.join();
}
```

…the new task/thread and the continuation of the current one.

Where the DAG Comes From




```
main:
  a = fork task1
  b = fork task2
  O(1) work
  join a
  join b

task1:
  O(1) work

task2:
  c = fork task1
  O(1) work
  join c
```

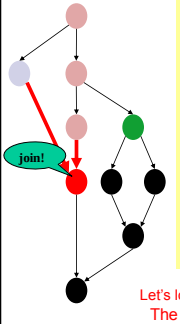
```
int main(..) {
  std::thread t1(&task1);
  std::thread t2(&task2);
  // O(1) work
  t1.join();
  t2.join();
  return 0;
}

void task1() {
  // O(1) work
}

void task2() {
  std::thread t(&task1);
  // O(1) work
  t.join();
}
```

Again, we fork off a task/thread.
Meanwhile, the left (blue) task finished.

Where the DAG Comes From




```
main:
  a = fork task1
  b = fork task2
  O(1) work
  join a
  join b

task1:
  O(1) work

task2:
  c = fork task1
  O(1) work
  join c
```

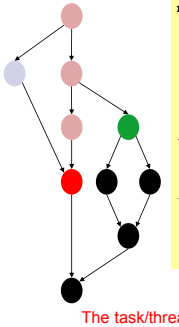
```
int main(..) {
  std::thread t1(&task1);
  std::thread t2(&task2);
  // O(1) work
  t1.join();
  t2.join();
  return 0;
}

void task1() {
  // O(1) work
}

void task2() {
  std::thread t(&task1);
  // O(1) work
  t.join();
}
```

Let's look at `main`'s join next. (Either way we get the same DAG.)
The join makes a node that "waits on" (has an arrow from) the last node of the joining task and of the joined one.

Where the DAG Comes From



```
main:
  a = fork task1
  b = fork task2
  O(1) work
  join a
  join b

task1:
  O(1) work

task2:
  c = fork task1
  O(1) work
  join c
```

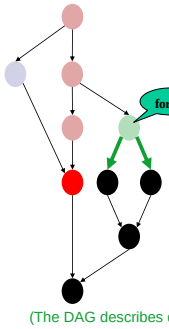
```
int main(..) {
  std::thread t1(&task1);
  std::thread t2(&task2);
  // O(1) work
  t1.join();
  t2.join();
  return 0;
}

void task1() {
  // O(1) work
}

void task2() {
  std::thread t(&task1);
  // O(1) work
  t.join();
}
```

The next join isn't ready to go yet.
The task/thread it's joining isn't finished. So, it waits and so do we.

Where the DAG Comes From




```
main:
  a = fork task1
  b = fork task2
  O(1) work
  join a
  join b

task1:
  O(1) work

task2:
  c = fork task1
  O(1) work
  join c
```

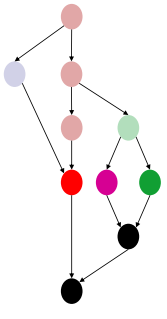
```
int main(..) {
  std::thread t1(&task1);
  std::thread t2(&task2);
  // O(1) work
  t1.join();
  t2.join();
  return 0;
}

void task1() {
  // O(1) work
}

void task2() {
  std::thread t(&task1);
  // O(1) work
  t.join();
}
```

Meanwhile, task2 also forks a task1.
(The DAG describes dynamic execution. We can run the same code many times!)

Where the DAG Comes From



```
main:
  a = fork task1
  b = fork task2
  O(1) work
  join a
  join b

task1:
  O(1) work

task2:
  c = fork task1
  O(1) work
  join c
```

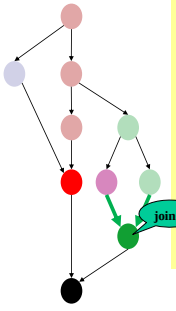
```
int main(..) {
  std::thread t1(&task1);
  std::thread t2(&task2);
  // O(1) work
  t1.join();
  t2.join();
  return 0;
}

void task1() {
  // O(1) work
}

void task2() {
  std::thread t(&task1);
  // O(1) work
  t.join();
}
```

task1 and task2 both chugging along.

Where the DAG Comes From




```
main:
  a = fork task1
  b = fork task2
  O(1) work
  join a
  join b

task1:
  O(1) work

task2:
  c = fork task1
  O(1) work
  join c
```

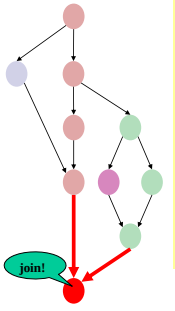
```
int main(..) {
  std::thread t1(&task1);
  std::thread t2(&task2);
  // O(1) work
  t1.join();
  t2.join();
  return 0;
}

void task1() {
  // O(1) work
}

void task2() {
  std::thread t(&task1);
  // O(1) work
  t.join();
}
```

task2 joins task1.

Where the DAG Comes From




```
main:
  a = fork task1
  b = fork task2
  O(1) work
  join a
  join b

task1:
  O(1) work

task2:
  c = fork task1
  O(1) work
  join c
```

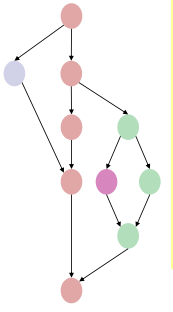
```
int main(..) {
  std::thread t1(&task1);
  std::thread t2(&task2);
  // O(1) work
  t1.join();
  t2.join();
  return 0;
}

void task1() {
  // O(1) work
}

void task2() {
  std::thread t(&task1);
  // O(1) work
  t.join();
}
```

Task2 (the right, green task) is finally done.
So, the main task joins with it.
(Arrow from the last node of the joining task and of the joined one.)

Where the DAG Comes From



```
main:
  a = fork task1
  b = fork task2
  O(1) work
  join a
  join b

task1:
  O(1) work

task2:
  c = fork task1
  O(1) work
  join c
```

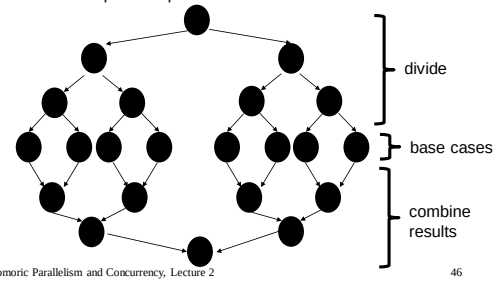
```
int main(..) {
  std::thread t1(&task1);
  std::thread t2(&task2);
  // O(1) work
  t1.join();
  t2.join();
  return 0;
}

void task1() {
  // O(1) work
}

void task2() {
  std::thread t(&task1);
  // O(1) work
  t.join();
}
```

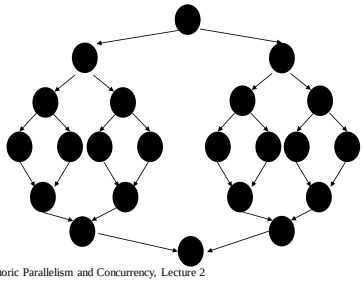

Analyzing Real Code (Not Fake Code or Clothes)

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



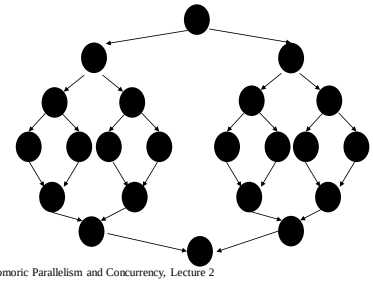
Map/Reduce DAG: Work?

Asymptotically, what's the **work** in this DAG?



Map/Reduce DAG: Span?

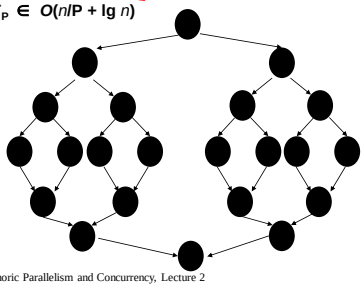
Asymptotically, what's the **span** in this DAG?



Map/Reduce DAG

Work: $O(n)$
Span: $O(\lg n)$
So: $T_P \in O(n/P + \lg n)$

LOTS of room for perfect linear speedup with large n!



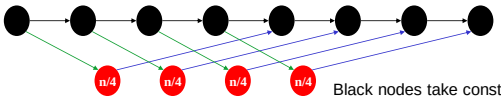
Loop (**not** Divide-and-Conquer) DAG: Work/Span?

```
int divs = 4;  /* some number of divisions */

std::thread workers[divs];
int results[divs];
for (int d = 0; d < divs; d++)
  // count matches in 1/divs sized part of the array
  workers[d] = std::thread(&cm_helper_seq1, ...);

int matches = 0;
for (int d = 0; d < divs; d++) {
  workers[d].join();
  matches += results[d];
}

return matches;
```



Black nodes take constant time.
Red nodes take ***non-constant*** time!

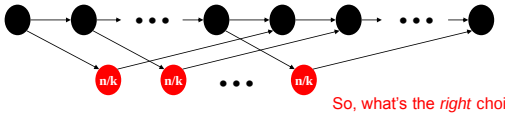
Loop (**not** Divide-and-Conquer) DAG: Work/Span?

```
int divs = k;  /* some number of divisions */

std::thread workers[divs];
int results[divs];
for (int d = 0; d < divs; d++)
  // count matches in 1/divs sized part of the array
  workers[d] = std::thread(&cm_helper_seq1, ...);

int matches = 0;
for (int d = 0; d < divs; d++) {
  workers[d].join();
  matches += results[d];
}

return matches;
```



So, what's the *right* choice of `k`?
What work/span/expected performance does it give us?

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)

Work/span is great, but real programs typically have:
- parts that parallelize well like maps/reduces over arrays/trees
- parts that don't parallelize at all like 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 $T_1 = 1$ (measured in weird but handy units)

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

Then: $T_1 = S + (1-S) = 1$

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

Then: $T_P = S + (1-S)/P$

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

$T_1 / T_P =$

How good/bad is this?

Amdahl's Law (mostly bad news)

Let $\mathbf{T_1 = 1}$ (measured in weird but handy units)

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):

$T_1 / T_P =$

And the parallelism (infinite processors) is:

$T_1 / T_\infty =$

Why such bad news

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

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

Why such bad news

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

- 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

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

    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. Parallelism can still help!

In our maps/reduces, the sequential part is ***O*(1)** and so becomes trivially small as ***n*** scales up. **(This is tremendously important!)**

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

We 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

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.

Moore and Amdahl

- Moore's "Law" is an observation about the progress of the semiconductor industry
  - Transistor density doubles 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