CSE 332 Data Abstractions:

Introduction to Parallelism and Concurrency

Kate Deibel

Summer 2012

Midterm: Question 1d

What is the tightest bound that you can give for the summation $\sum_{i=0}^{n} i^k$?

This is an important summation to recognize

k=1 → $\sum_{i=1}^{n} i^1 = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{n^2}{2}$

k=2 → $\sum_{i=1}^{n} i^2 = 1 + 4 + 9 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{n^3}{3}$

k=3 → $\sum_{i=1}^{n} i^3 = 1 + 8 + 27 + \cdots + n^3 = \frac{n^2(n+1)^2}{4} \approx \frac{n^4}{4}$

k=4 → $\sum_{i=1}^{n} i^4 = 1 + 16 + 81 + \cdots + n^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30} \approx \frac{n^5}{5}$

In general, the sum of the first n integers to the kth power is always of the next power up

$$\sum_{i=1}^{n} i^k = 1^k + 2^k + 3^k \cdots + n^k \approx \frac{n^{k+1}}{k+1} = \Theta(n^{k+1})$$

Changing a Major Assumption

So far most or all of your study of computer science has assumed:

ONE THING HAPPENED AT A TIME

Called sequential programming—everything part of one sequence

Removing this assumption creates major challenges and opportunities

- Programming: Divide work among threads of execution and coordinate among them (i.e., synchronize their work)
- Algorithms: How can parallel activity provide speed-up (more throughput, more work done per unit time)
- Data structures: May need to support concurrent access (multiple threads operating on data at the same time)

A Simplified View of History

Writing correct and efficient multithreaded code is often much more difficult than single-threaded code

- Especially in typical languages like Java and C
- So we typically stay sequential whenever possible

From roughly 1980-2005, desktop computers got exponentially faster at running sequential programs

- About twice as fast every couple years

But nobody knows how to continue this

- Increasing clock rate generates too much heat
- Relative cost of memory access is too high

A Simplified View of History

We knew this was coming, so we looked at the idea of using multiple computers at once

- Computer clusters (e.g., Beowulfs)
- Distributed computing (e.g., SETI@Home)

These ideas work but are not practical for personal machines, but fortunately:

- We are still making "wires exponentially smaller" (per Moore's "Law")
- So why not put multiple processors on the same chip (i.e., "multicore")?

What to do with Multiple Processors?

Your next computer will likely have 4 processors

- Wait a few years and it will be 8, 16, 32, …
- Chip companies decided to do this (not a "law")

What can you do with them?

- Run multiple different programs at the same time?
  - We already do that with time-slicing with the OS
- Do multiple things at once in one program?
  - This will be our focus but it is far more difficult
  - We must rethink everything from asymptotic complexity to data structure implementations

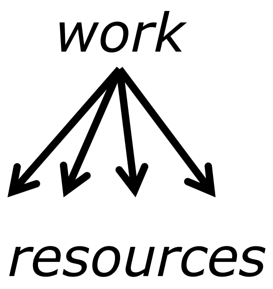
Definitions definitions definitions… are you sick of them yet?

BASIC DEFINITIONS: PARALLELISM & CONCURRENCY

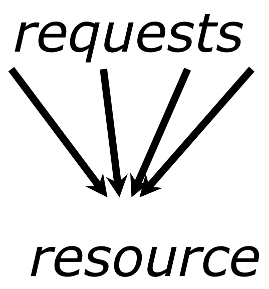
Parallelism vs. Concurrency

Note: These terms are not yet standard but the perspective is essential
Many programmers confuse these concepts

Parallelism:
Use extra resources to solve a problem faster



Concurrency:
Correctly and efficiently manage access to shared resources



These concepts are related but still different:

- Common to use threads for both
- If parallel computations need access to shared resources, then the concurrency needs to be managed

An Analogy

CS1 idea: A program is like a recipe for a cook

- One cook who does one thing at a time!

Parallelism:

- Have lots of potatoes to slice?
- Hire helpers, hand out potatoes and knives
- But too many chefs and you spend all your time coordinating

Concurrency:

- Lots of cooks making different things, but there are only 4 stove burners available in the kitchen
- We want to allow access to all 4 burners, but not cause spills or incorrect burner settings

Parallelism Example

Parallelism: Use extra resources to solve a problem faster (increasing throughput via simultaneous execution)

Pseudocode for array sum

- No ‘FORALL’ construct in Java, but we will see something similar
- Bad style for reasons we’ll see, but may get roughly 4x speedup

```
int sum(int[] arr){
  result = new int[4];
  len = arr.length;
  FORALL(i=0; i < 4; i++) { //parallel iterations
    result[i] = sumRange(arr,i*len/4,(i+1)*len/4);
  }
  return result[0]+result[1]+result[2]+result[3];
}

int sumRange(int[] arr, int lo, int hi) {
   result = 0;
   for(j=lo; j < hi; j++)
      result += arr[j];
   return result;
}
```

Concurrency Example

Concurrency: Correctly and efficiently manage access to shared resources (from multiple possibly-simultaneous clients)

Pseudocode for a shared chaining hashtable

- Prevent bad interleavings (critical ensure correctness)
- But allow some concurrent access (critical to preserve performance)

```
class Hashtable<K,V> {
   …
   void insert(K key, V value) {
      int bucket = …;
      prevent-other-inserts/lookups in table[bucket]
      do the insertion
      re-enable access to arr[bucket]
   }
   V lookup(K key) {
       (similar to insert,
       but can allow concurrent lookups to same bucket)
   }
}
```

Shared Memory with Threads

The model we will assume is shared memory with explicit threads

Old story: A running program has

- One program counter (the current statement that is executing)
- One call stack (each stack frame holding local variables)
- Objects in the heap created by memory allocation (i.e., new) (same name, but no relation to the heap data structure)
- Static fields in the class shared among objects

Shared Memory with Threads

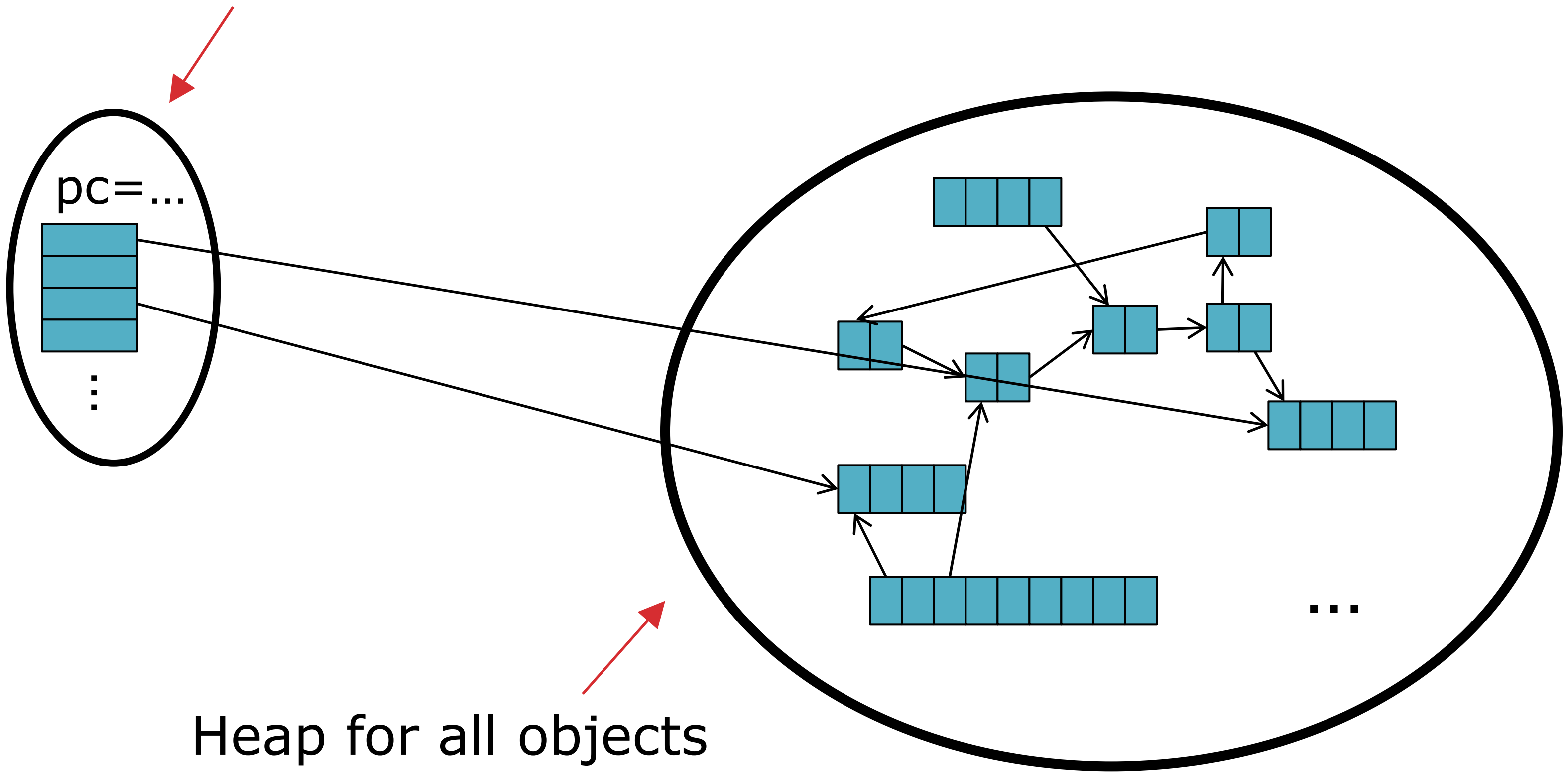
The model we will assume is shared memory with explicit threads

New story:

- A set of threads, each with a program and call stack but no access to another thread's local variables
- Threads can implicitly share objects and static fields
- Communication among threads occurs via writing values to a shared location that another thread reads

Old Story: Single-Threaded

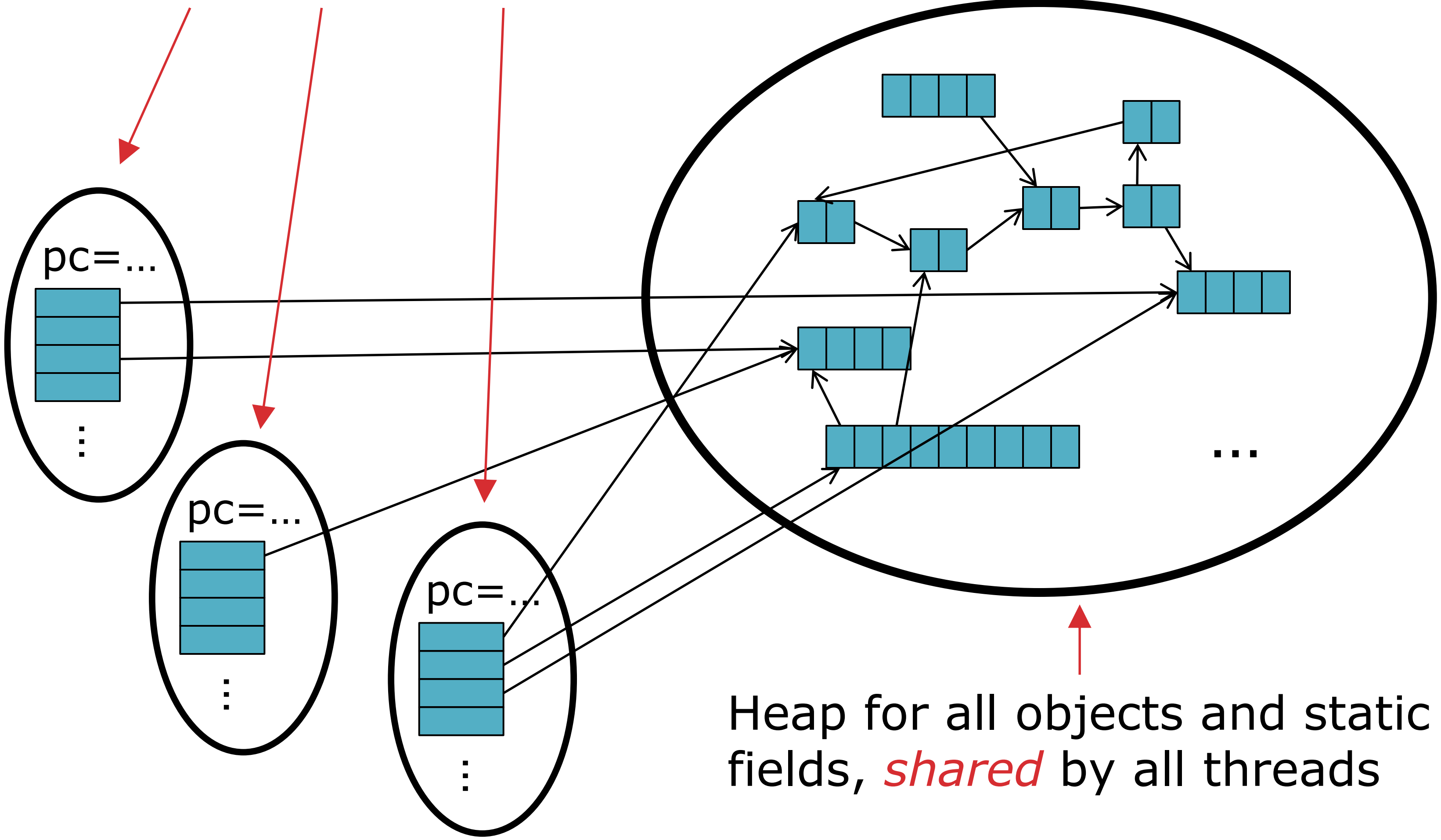
Call stack with local variables
Program counter for current statement
Local variables are primitives or heap references



Heap for all objects
and static fields

New Story: Threads & Shared Memory

Threads, each with own *unshared* call stack and "program counter"



Heap for all objects and static fields, *shared* by all threads

Other Parallelism/Concurrency Models

We will focus on shared memory, but you should know several other models exist and have their own advantages

Message-passing:

- Each thread has its own collection of objects
- Communication is via explicitly sending/receiving messages
- Cooks working in separate kitchens, mail around ingredients

Dataflow:

- Programmers write programs in terms of a DAG.
- A node executes after all of its predecessors in the graph
- Cooks wait to be handed results of previous steps

Data parallelism:

- Have primitives for things like "apply function to every element of an array in parallel"

Keep in mind that Java was first released in 1995

FIRST IMPLEMENTATION: SHARED MEMORY IN JAVA

Our Needs

To write a shared-memory parallel program, we need new primitives from a programming language or library

Ways to create and *run multiple things at once*

- We will call these things threads

Ways for threads to *share memory*

- Often just have threads with references to the same objects

Ways for threads to *coordinate* (*a.k.a. synchronize*)

- For now, a way for one thread to wait for another to finish
- Other primitives when we study concurrency

Java Basics

We will first learn some basics built into Java via the provided **`java.lang.Thread`** package

- We will learn a better library for parallel programming

To get a new thread running:

1. Define a subclass `C` of **`java.lang.Thread`**,
2. Override the **`run`** method
3. Create an object of class `C`
4. Call that object's **`start`** method

`start` sets off a new thread, using **`run`** as its "main"

What if we instead called the **`run`** method of `C`?

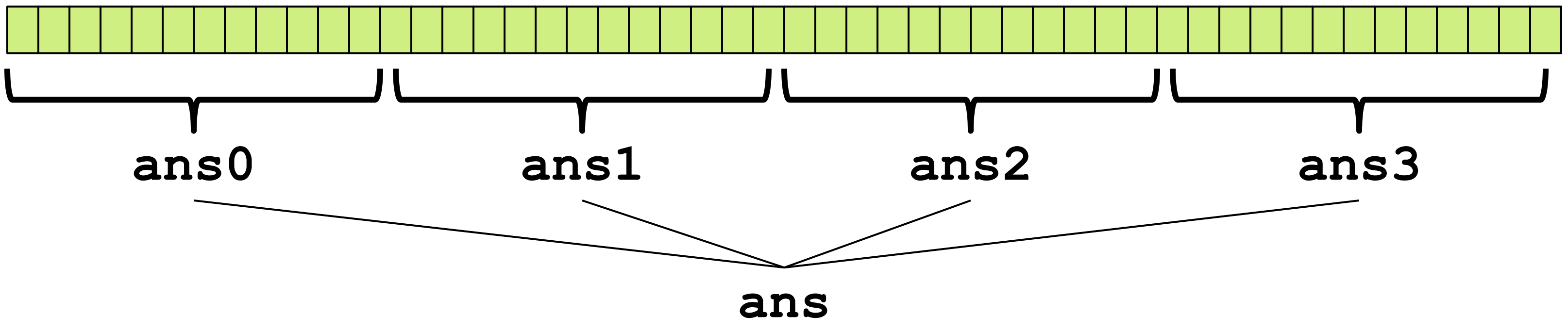
- Just a normal method call in the current thread

Parallelism Example: Sum an Array

Have 4 threads simultaneously sum 1/4 of the array

Approach:

- Create 4 thread objects, each given a portion of the work
- Call start() on each thread object to actually run it in parallel
- Somehow 'wait' for threads to finish
- Add together their 4 answers for the final result



Warning: This is the inferior first approach, do not do this

Creating the Thread Subclass

```
class SumThread extends java.lang.Thread {

  int lo; // arguments
  int hi;
  int[] arr;

  int ans = 0; // result

  SumThread(int[] a, int l, int h) {
    lo=l; hi=h; arr=a;
  }

  public void run() { //override must have this type
    for(int i=lo; i < hi; i++)
      ans += arr[i];
  }
}
```

We will ignore handling the case where:
arr.length % 4 != 0

Because we override a no-arguments/no-result run, we use fields to communicate data across threads

Creating the Threads Wrongly

```
class SumThread extends java.lang.Thread {
  int lo, int hi, int[] arr; // arguments
  int ans = 0; // result
  SumThread(int[] a, int l, int h) { … }
  public void run(){ … } // override
}
```

```
int sum(int[] arr){ // can be a static method
  int len = arr.length;
  int ans = 0;
  SumThread[] ts = new SumThread[4];
  for(int i=0; i < 4; i++) // do parallel computations
    ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
  for(int i=0; i < 4; i++) // combine results
    ans += ts[i].ans;
  return ans;
}
```

We forgot to start the threads!!!

Starting Threads but Still Wrong

```
int sum(int[] arr){ // can be a static method
  int len = arr.length;
  int ans = 0;
  SumThread[] ts = new SumThread[4];
  for(int i=0; i < 4; i++){// do parallel computations
    ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
    ts[i].start(); // start not run
  }
  for(int i=0; i < 4; i++) // combine results
    ans += ts[i].ans;
  return ans;
}
```

We start the threads and then assume they finish right away!!!

Join: The 'Wait for Thread' Method

The `Thread` class defines various methods that provide primitive operations you could not implement on your own

- For example: `start`, which calls `run` in a new thread

The `join` method is another such method, essential for coordination in this kind of computation

- Caller blocks until/unless the receiver is done executing (meaning its `run` method returns after its execution)
- Without join, we would have a 'race condition' on `ts[i].ans` in which the variable is read/written simultaneously

This style of parallel programming is called fork/join"

- If we write in this style, we avoid many concurrency issues
- But certainly not all of them

Third Attempt: Correct in Spirit

```
int sum(int[] arr){ // can be a static method
  int len = arr.length;
  int ans = 0;
  SumThread[] ts = new SumThread[4];
  for(int i=0; i < 4; i++){// do parallel computations
    ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
    ts[i].start();
  }
  for(int i=0; i < 4; i++) { // combine results
    ts[i].join(); // wait for helper to finish!
    ans += ts[i].ans;
  }
  return ans;
}
```

Note that there is no guarantee that ts[0] finishes before ts[1]

- Completion order is nondeterministic
- Not a concern as our threads do the same amount of work

Where is the Shared Memory?

Fork-join programs tend not to require [thankfully] a lot of focus on sharing memory among threads

- But in languages like Java, there is memory being shared

In our example:

- **`lo`**, **`hi`**, **`arr`** fields written by "main" thread, read by helper thread
- **`ans`** field written by helper thread, read by "main" thread

When using shared memory, the challenge and absolute requirement is to avoid race conditions

- While studying parallelism, we'll stick with **`join`**
- With concurrency, we'll learn other ways to synchronize

Keep in mind that Java was first released in 1995

BETTER ALGORITHMS: PARALLEL ARRAY SUM

A Poor Approach: Reasons

Our current array sum code is a poor usage of parallelism for several reasons

1. Code should be reusable and efficient across platforms
   - "Forward-portable" as core count grows
   - At the *very* least, we should parameterize the number of threads used by the algorithm

```
int sum(int[] arr, int numThreads){
  … // note: shows idea, but has integer-division bug
  int subLen = arr.length / numThreads;
  SumThread[] ts = new SumThread[numThreads];
  for(int i=0; i < numThreads; i++){
   ts[i] = new SumThread(arr,i*subLen,(i+1)*subLen);
   ts[i].start();
  }
  for(int i=0; i < numThreads; i++) {
    …
  }
  …
```

A Poor Approach: Reasons

Our current array sum code is a poor usage of parallelism for several reasons

2. We want to use only the processors "available now"
   - Not used by other programs or threads in your program
     - Maybe caller is also using parallelism
     - Available cores can change even while your threads run
   - If 3 processors available and 3 threads would take time **`x`**, creating 4 threads can have worst-case time of **`1.5x`**

```
// numThreads == numProcessors is bad
// if some are needed for other things
int sum(int[] arr, int numThreads){
  …
}
```

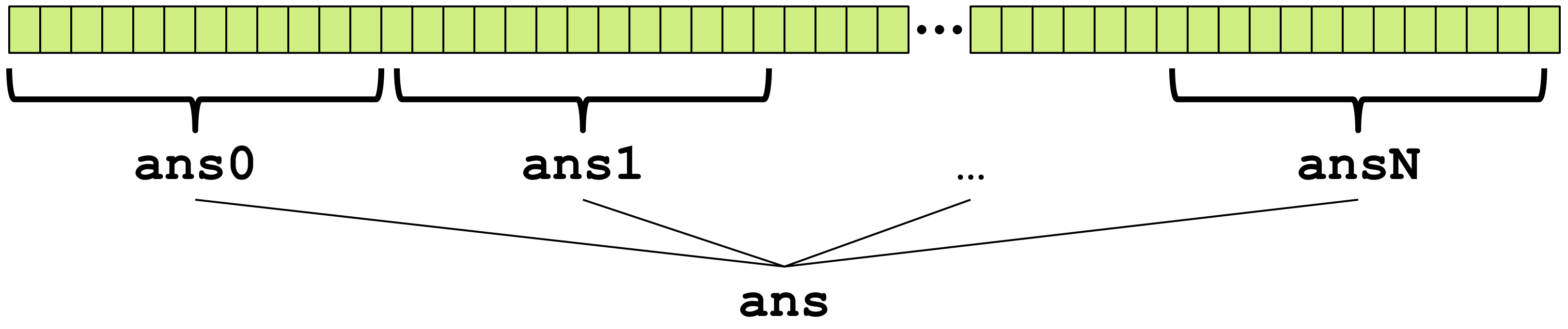
A Poor Approach: Reasons

Our current array sum code is a poor usage of parallelism for several reasons

3. Though unlikely for **`sum`**, subproblems may take significantly different amounts of time
   - Example: Apply method **`f`** to every array element, but maybe **`f`** is much slower for some data items
     - Example: Determine if a large integer is prime?
   - If we create 4 threads and all the slow data is processed by 1 of them, we won’t get nearly a 4x speedup
     - Example of a load imbalance

A Better Approach: Counterintuitive

Although counterintuitive, the better solution is to use a lot more threads beyond the number of processors



1. **Forward-Portable**: Lots of helpers each doing small work
2. **Processors Available**: Hand out "work chunks" as you go
   - If 3 processors available and have 100 threads, worst-case extra time is < 3% (if we ignore constant factors and load imbalance)
3. **Load Imbalance**: Problem "disappears"
   - Try to ensure that slow threads are scheduled early
   - Variation likely small if pieces of work are also small

But Do Not Be Naïve

This approach does not provide a free lunch:

Assume we create 1 thread to process every N elements

```
int sum(int[] arr, int N){
   …
   // How many pieces of size N do we have?
   int numThreads = arr.length / N;
   SumThread[] ts = new SumThread[numThreads];
   …
}
```

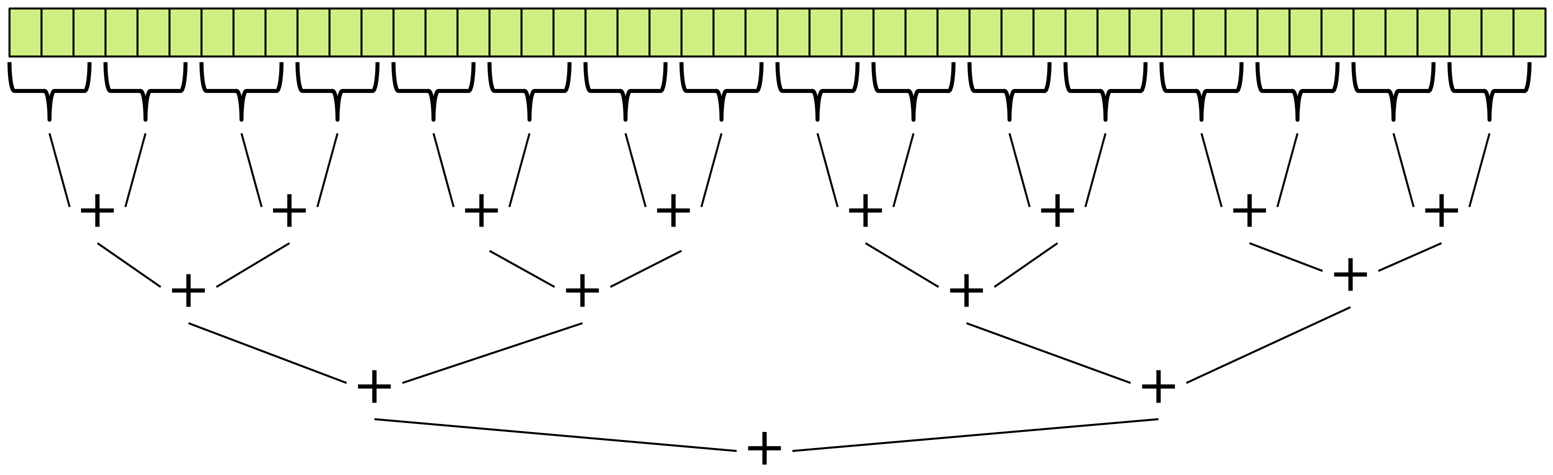
Combining results will require **`arr.length/N`** additions

- As **`N`** increases, this becomes linear in size of array
- Previously we only had 4 pieces, Θ(1) to combine

In the extreme, suppose we create one thread per element

- Using a loop to combine the results requires N iterations

A Better Idea: Divide-and-Conquer



Straightforward to implement

Use parallelism for the recursive calls

- Halve and make new thread until size is at some cutoff
- Combine answers in pairs as we return

This starts small but grows threads to fit the problem

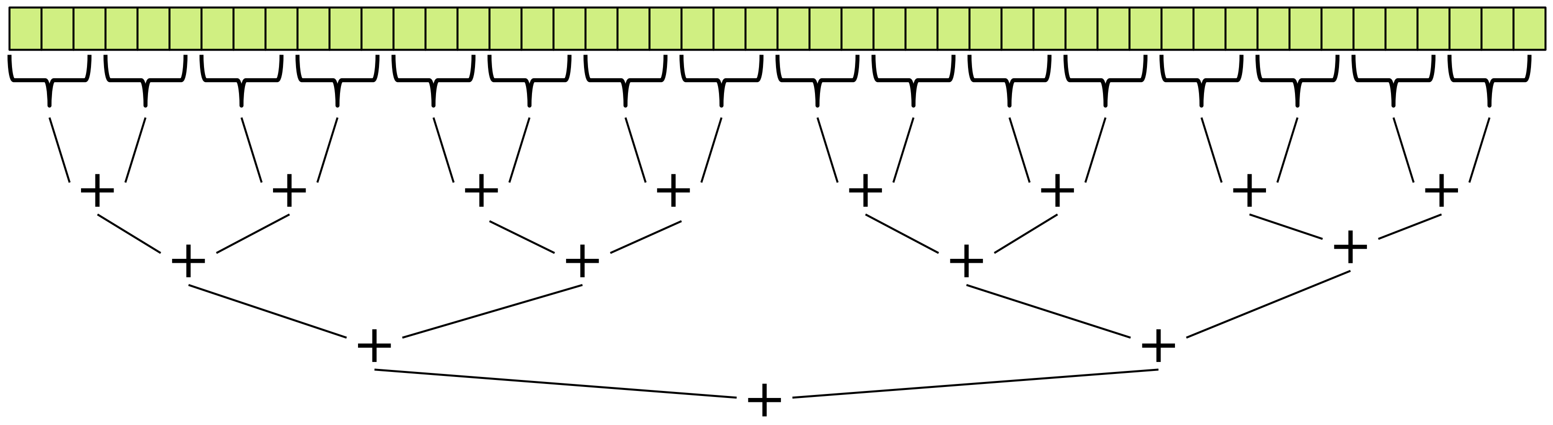
Divide-and-Conquer

```
public void run(){ // override
  if(hi - lo < SEQUENTIAL_CUTOFF)
      for(int i=lo; i < hi; i++)
       ans += arr[i];
  else {
    SumThread left = new SumThread(arr,lo,(hi+lo)/2);
    SumThread right= new SumThread(arr,(hi+lo)/2,hi);
    left.start();
    right.start();
    left.join(); // don't move this up a line – why?
    right.join();
    ans = left.ans + right.ans;
  }
 }
}

int sum(int[] arr){
  SumThread t = new SumThread(arr,0,arr.length);
  t.run();
  return t.ans;
}
```

Divide-and-Conquer Really Works

The key is to parallelize the result-combining

- With *enough* processors, total time is the tree height: $O(\log n)$
- This is optimal and exponentially faster than sequential $O(n)$)
- But the reality is that we usually have $P < O(n)$ processors



Still, we will write our parallel algorithms in this style

- Relies on operations being associative (as with +)
- But will use a special library engineered for this style
- It takes care of scheduling the computation well

Good movie… speaks to Generation Xers…

REALITY BITES

Being Realistic

In theory, you can divide down to single elements and then do all your result-combining in parallel and get optimal speedup

In practice, creating all those threads and communicating amongst them swamps the savings,

To gain better efficiency:

- Use a *sequential cutoff*, typically around 500-1000
  - Eliminates *almost all* of the recursive thread creation because it eliminates the bottom levels of the tree
  - This is *exactly* like quicksort switching to insertion sort for small subproblems, but even more important here
- Be clever and do not create unneeded threads
  - When creating a thread, you are already in another thread
  - Why not use the current thread to do half the work?
  - Cuts the number of threads created by another 2x

Halving the Number of Threads

```
// wasteful: don't
SumThread left  = …
SumThread right = …

// create two threads
left.start();
right.start();
left.join();
right.join();
ans=left.ans+right.ans;
```

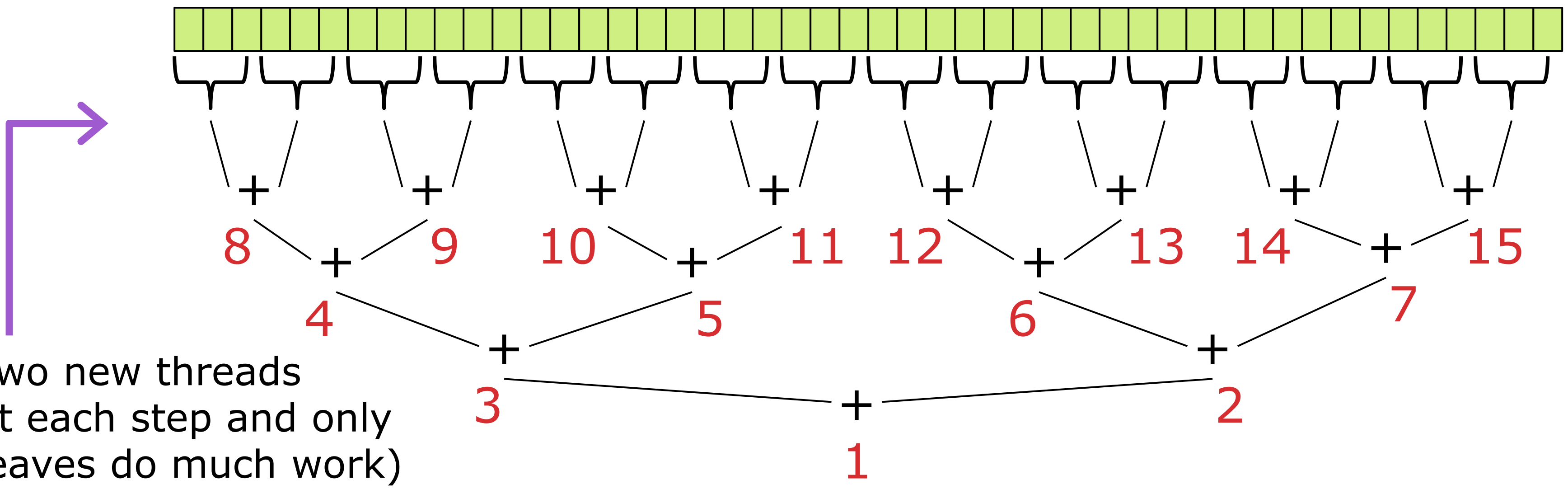
```
// better: do
SumThread left  = …
SumThread right = …

// order of next 4 lines
// essential – why?
left.start();
right.run();
left.join();
ans=left.ans+right.ans;
```

If a *language* had built-in support for fork-join parallelism, we would expect this hand-optimization to be unnecessary

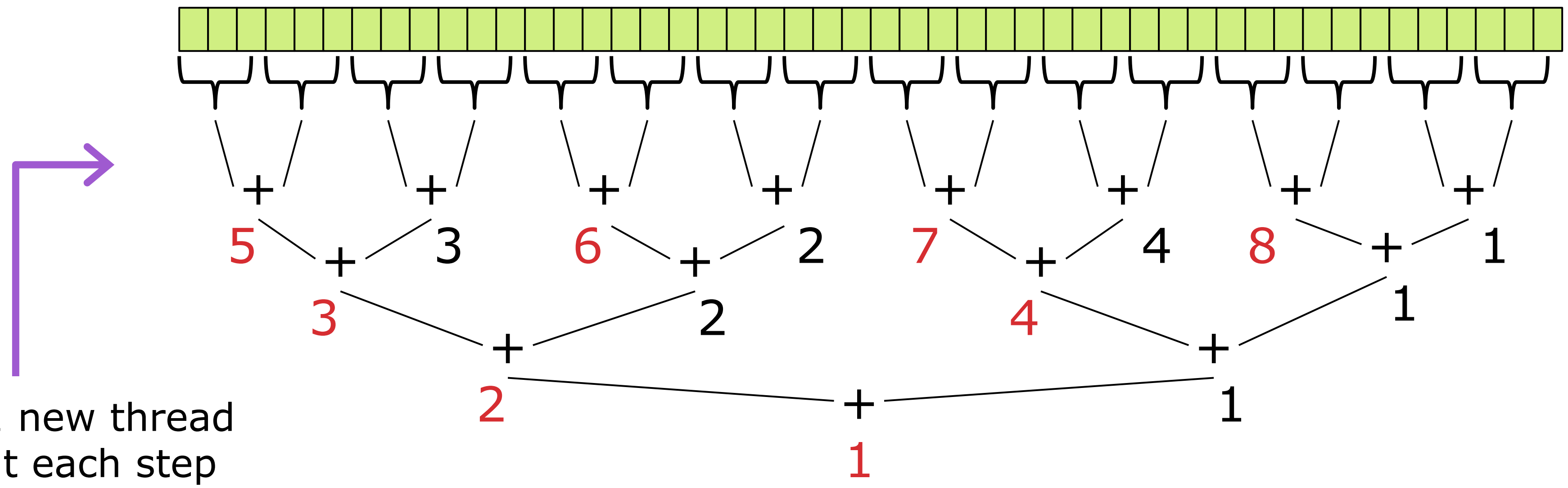
But the *library* we are using expects you to do it yourself

- And the difference is surprisingly substantial
- But no difference in theory

Illustration of Fewer Threads



Two new threads
at each step and only
leaves do much work)



1 new thread
at each step

Limits of The Java Thread Library

Even with all this care, Java's threads are too *heavyweight*

- Constant factors, especially space overhead
- Creating 20,000 Java threads just a bad idea

The ForkJoin Framework is designed/engineered to meet the needs of divide-and-conquer fork-join parallelism

- Included in the Java 7 standard libraries
- Also available as a downloaded **`.jar`** file for Java 6
- Section will discuss some pragmatics/logistics
- Similar libraries available for other languages
  - C/C++: Cilk, Intel's Thread Building Blocks
  - C#: Task Parallel Library
- Library implementation is an advanced topic

Different Terms / Same Basic Ideas

To use the ForkJoin Framework:

- A little standard set-up code (e.g., create a **`ForkJoinPool`**)

The Fundamental Differences:

Don't subclass **`Thread`**	Do subclass **`RecursiveTask<V>`**
Don't override **`run`**	Do override **`compute`**
Do not use an **`ans`** field	Do return a **`V`** from **`compute`**
Do not call **`start`**	Do call **`fork`**
Do not just call **`join`**	Do call **`join`** which returns answer
Do not call **`run`** to hand-optimize	Do call **`compute`** to hand-optimize
Do not have a topmost call to **`run`**	Do create a pool and call **`invoke`**

See the Dan Grossman's web page for

"A Beginner's Introduction to the ForkJoin Framework"

http://www.cs.washington.edu/homes/djg/teachingMaterials/spac/grossmanSPAC_forkJoinFramework.html

Final Version in ForkJoin Framework

```
class SumArray extends RecursiveTask<Integer> {
  int lo; int hi; int[] arr; // arguments
  SumArray(int[] a, int l, int h) { … }
  protected Integer compute(){// return answer
    if(hi - lo < SEQUENTIAL_CUTOFF) {
      int ans = 0;
      for(int i=lo; i < hi; i++)
        ans += arr[i];
      return ans;
    } else {
      SumArray left = new SumArray(arr,lo,(hi+lo)/2);
      SumArray right= new SumArray(arr,(hi+lo)/2,hi);
      left.fork();
      int rightAns = right.compute();
      int leftAns  = left.join();
      return leftAns + rightAns;
    }
  }
}

static final ForkJoinPool fjPool = new ForkJoinPool();

int sum(int[] arr){
  return fjPool.invoke(new SumArray(arr,0,arr.length));
}
```

For Comparison: Java Threads Version

```
class SumThread extends java.lang.Thread {
  int lo; int hi; int[] arr;//fields to know what to do
  int ans = 0; // for communicating result
  SumThread(int[] a, int l, int h) { … }
  public void run(){
    if(hi – lo < SEQUENTIAL_CUTOFF)
      for(int i=lo; i < hi; i++)
        ans += arr[i];
    else { // create 2 threads, each will do ½ the work
      SumThread left = new SumThread(arr,lo,(hi+lo)/2);
      SumThread right= new SumThread(arr,(hi+lo)/2,hi);
      left.start();
      right.start();
      left.join(); // don't move this up a line – why?
      right.join();
      ans = left.ans + right.ans;
    }
  }
}

class C {
 static int sum(int[] arr){
   SumThread t = new SumThread(arr,0,arr.length);
   t.run(); // only creates one thread
   return t.ans;
 }
}
```

Getting Good Results with ForkJoin

Sequential threshold

- Library documentation recommends doing approximately 100-5000 basic operations in each "piece" of your algorithm

Library needs to "warm up"

- May see slow results before the Java virtual machine re-optimizes the library internals
- When evaluating speed, loop computations to see the "long-term benefit" after these optimizations have occurred

Wait until your computer has more processors

- Seriously, overhead may dominate at 4 processors
- But parallel programming becoming much more important

Beware memory-hierarchy issues

- Will not focus on but can be crucial for parallel performance

Ah yes… the comfort of mathematics…

ENOUGH IMPLEMENTATION: ANALYZING PARALLEL CODE

Key Concepts: Work and Span

Analyzing parallel algorithms requires considering the full range of processors available

- We parameterize this by letting $\mathbf{T_P}$ be the running time if **P** processors are available
- We then calculate two extremes: work and span

Work: $\mathbf{T_1}$ → How long using only 1 processor

- Just "sequentialize" the recursive forking

Span: $\mathbf{T_\infty}$ → How long using infinity processors

- The longest dependence-chain
- Example: $O(\log n)$ for summing an array
  - Notice that having > $n/2$ processors is no additional help
- Also called "critical path length" or "computational depth"

The DAG

A program execution using **`fork`** and **`join`** can be seen as a DAG

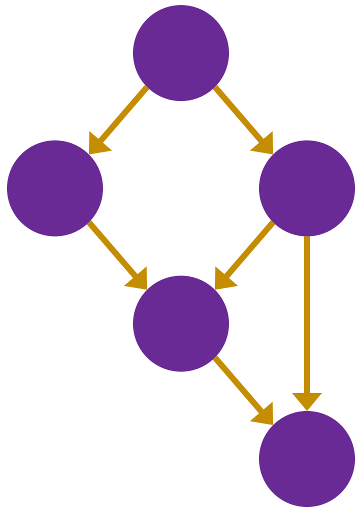
- Nodes: Pieces of work
- Edges: Source must finish before destination starts

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

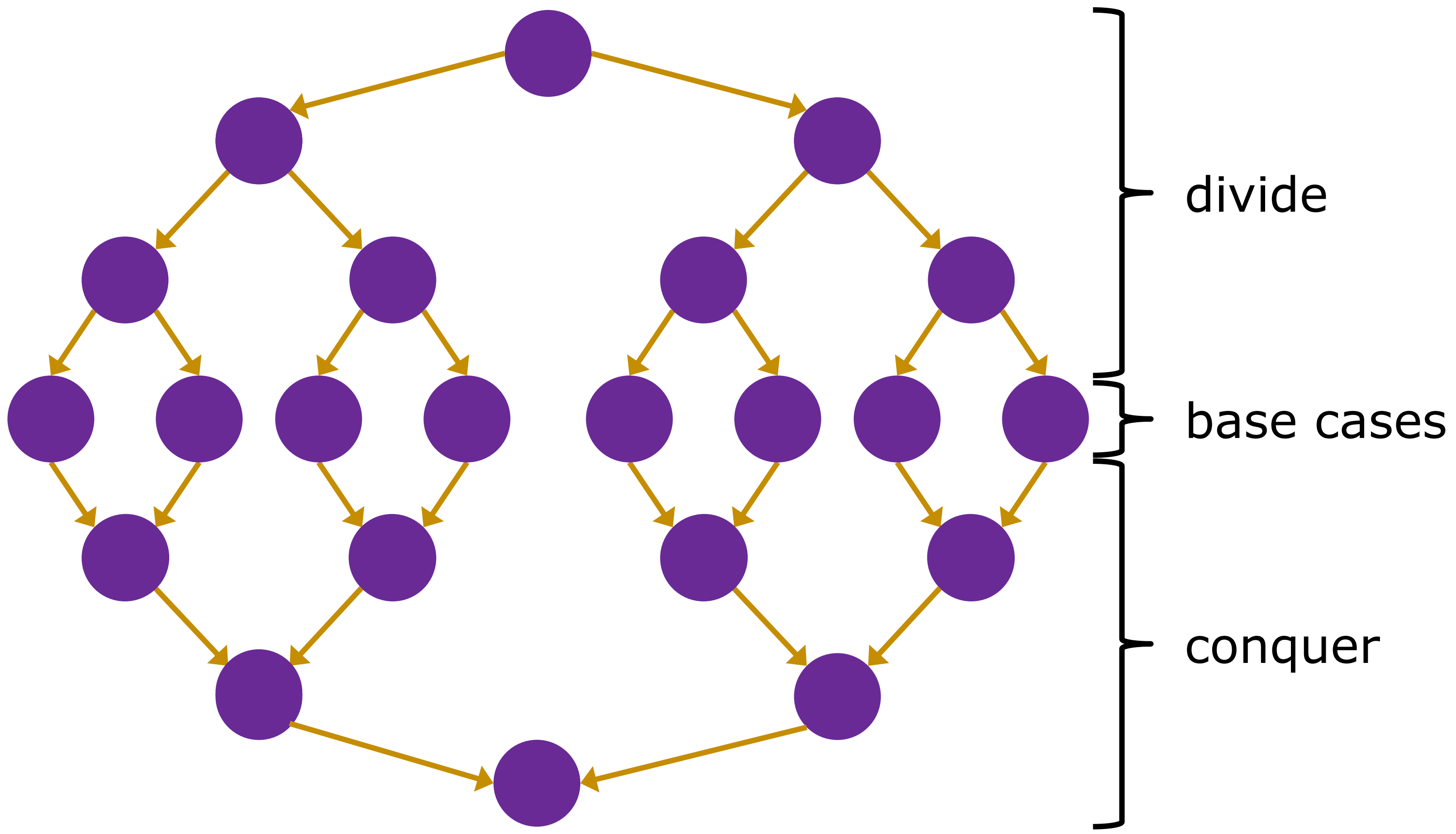
- Node just ended
- Last node of thread joined on



Our Simple Examples

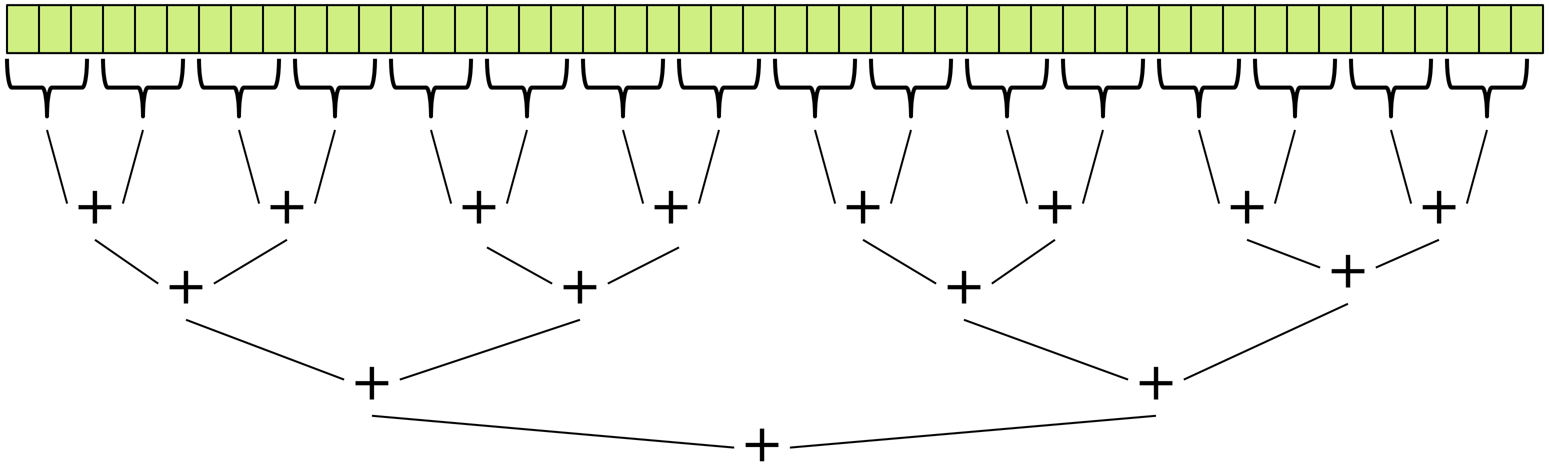
`fork` and **`join`** are very flexible, but divide-and-conquer use them in a very basic way:

- A tree on top of an upside-down tree



What Else Looks Like This?

Summing an array went from $O(n)$ sequential to $O(\texttt{log}\ n)$ parallel (*assuming* ***a lot*** *of processors and very large n*)



Anything that can use results from two halves and merge them in $O(1)$ time has the same properties and exponential speed-up (in theory)

Examples

- Maximum or minimum element
- Is there an element satisfying some property (e.g., is there a 17)?
- Left-most element satisfying some property (e.g., first 17)
  - What should the recursive tasks return?
  - How should we merge the results?
- Corners of a rectangle containing all points (a "bounding box")
- Counts (e.g., # of strings that start with a vowel)
  - This is just summing with a different base case

More Interesting DAGs?

Of course, the DAGs are not always so simple (and neither are the related parallel problems)

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

Reductions

Such computations of this simple form are common enough to have a name: reductions (or reduces?)

Produce single answer from collection via an associative operator

- Examples: max, count, leftmost, rightmost, sum, …
- Non-example: median

Recursive results don't have to be single numbers or strings and can be arrays or objects with fields

- Example: Histogram of test results

But some things are inherently sequential

- How we process `arr[i]` may depend entirely on the result of processing `arr[i-1]`

Maps and 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 (often in graphics cards)

Canonical example: Vector addition

```
int[] vector_add(int[] arr1, int[] arr2){
  assert (arr1.length == arr2.length);
  result = new int[arr1.length];
  FORALL(i=0; i < arr1.length; i++) {
    result[i] = arr1[i] + arr2[i];
  }
  return result;
}
```

Maps in ForkJoin Framework

```
class VecAdd extends RecursiveAction {
  int lo; int hi; int[] res; int[] arr1; int[] arr2;
  VecAdd(int l,int h,int[] r,int[] a1,int[] a2){ … }
  protected void compute(){
    if(hi – lo < SEQUENTIAL_CUTOFF) {
       for(int i=lo; i < hi; i++)
         res[i] = arr1[i] + arr2[i];
    } else {
      int mid = (hi+lo)/2;
      VecAdd left = new VecAdd(lo,mid,res,arr1,arr2);
      VecAdd right= new VecAdd(mid,hi,res,arr1,arr2);
      left.fork();
      right.compute();
      left.join();
    }
  }
}
static final ForkJoinPool fjPool = new ForkJoinPool();
int[] add(int[] arr1, int[] arr2){
  assert (arr1.length == arr2.length);
  int[] ans = new int[arr1.length];
  fjPool.invoke(new VecAdd(0,arr.length,ans,arr1,arr2);
  return ans;
}
```

Maps and Reductions

Maps and reductions are the "workhorses" of parallel programming

- By far the two most important and common patterns
- We will discuss two more advanced patterns later

We often use maps and reductions to describe parallel algorithms

- We will aim to learn to recognize when an algorithm can be written in terms of maps and reductions
- Programming them then becomes "trivial" with a little practice (like how for-loops are second-nature to you)

Digression: MapReduce on Clusters

You may have heard of Google's "map/reduce"

- Or the open-source version Hadoop

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 database declarative queries

Maps and Reductions on Trees

Work just fine on balanced trees

- Divide-and-conquer each child
- Example:
  Finding the minimum element in an unsorted but balanced binary tree takes $O(\log n)$ time given enough processors

How to do you implement the sequential cut-off?

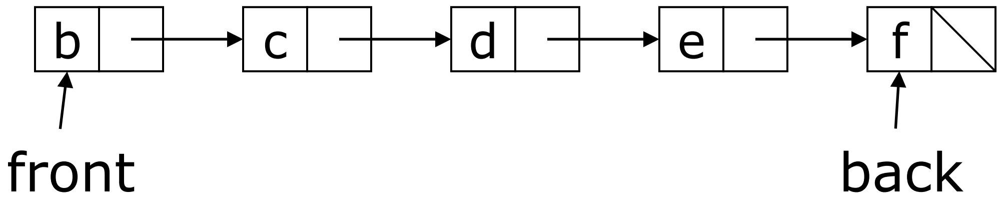
- Each node stores number-of-descendants (easy to maintain)
- Or approximate it (e.g., AVL tree height)

Parallelism also correct for unbalanced trees but you obviously do not get much speed-up

Linked Lists

Can you parallelize maps or reduces over linked lists?

- Example: Increment all elements of a linked list
- Example: Sum all elements of a linked list



Once again, data structures matter!

For parallelism, balanced trees generally better than lists so that we can get to all the data exponentially faster $O(\log n)$ vs. $O(n)$

- Trees have the same flexibility as lists compared to arrays (i.e., no shifting for insert or remove)

Analyzing algorithms

Like all algorithms, parallel algorithms should be:

- Correct
- Efficient

For our algorithms so far, their correctness is "obvious" so we'll focus on efficiency

- Want asymptotic bounds
- Want to analyze the algorithm without regard to a specific number of processors
- The key "magic" of the ForkJoin Framework is getting expected run-time performance asymptotically optimal for the available number of processors
    - Ergo we analyze algorithms assuming this guarantee

Connecting to Performance

Recall: $\mathbf{T_P}$ = run time if **P** processors are available

We can also think of this in terms of the program's DAG

Work = $\mathbf{T_1}$ = sum of run-time of all nodes in the DAG

- Note: costs are on the nodes not the edges
- That lonely processor does everything
- Any topological sort is a legal execution
- $O(n)$ for simple maps and reductions

Span = $\mathbf{T_\infty}$ = run-time of most-expensive path in 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
- $O(\log n)$ for simple maps and reductions

Some More Terms

Speed-up on **P** processors: $\mathbf{T_1 / T_P}$

Perfect linear speed-up: If speed-up is **P** as we vary **P**

- Means we get full benefit for each additional processor: as in doubling **P** halves running time
- Usually our goal
- Hard to get (sometimes impossible) 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

Optimal T_P: Thanks ForkJoin library

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), $\mathbf{T_P}$ cannot

- Less than $\mathbf{T_1 / P}$ why not?
- Less than $\mathbf{T_\infty}$ why not?

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**

The ForkJoin Framework gives an *expected-time guarantee* of asymptotically optimal!

- Expected time because it flips coins when *scheduling*
- How? For an advanced course (few need to know)
- Guarantee requires a few assumptions about your code…

Division of Responsibility

Our job as ForkJoin Framework users:

- Pick a good parallel algorithm and implement it
- Its execution creates a DAG of things to do
- *Make all the nodes small(ish) and approximately equal amount of work*

The framework-writer's job:

- Assign work to available processors to avoid idling
- Keep constant factors low
- Give the expected-time optimal guarantee assuming framework-user did his/her job

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

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

Algorithms seen so far (e.g., sum an array):

If $\mathbf{T_1} = O(n)$ and $\mathbf{T_\infty} = O(\mathbf{log}\ n)$

$\rightarrow \mathbf{T_P} = O(n/\mathbf{P} + \log n)$

Suppose instead:

If $\mathbf{T_1} = O(n^2)$ and $\mathbf{T_\infty} = O(n)$

$\rightarrow \mathbf{T_P} = O(n^2/\mathbf{P} + n)$

Of course, these expectations ignore any overhead or memory issues

Things are going so smoothly…

Parallelism is awesome…

Hello stranger, what's your name?

Murphy? Oh @!♪%★$☹*!!!

AMDAHL'S LAW

Amdahl’s Law (mostly bad news)

In practice, much of our programming typically has parts that parallelize well

- Maps/reductions over arrays and trees

And also parts that don’t parallelize at all

- Reading a linked list
- Getting/loading input
- Doing computations based on previous step

To understand the implications, consider this:

"Nine women cannot make a baby in one month"

Amdahl's Law (mostly bad news)

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

If **S** is the portion of execution that cannot be parallelized, then we can define T_1 as:

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

If we get perfect linear speedup on *the parallel portion*, then we can define T_P as:

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

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

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

And the parallelism (infinite processors) is:

$$\mathbf{T_1 / T_\infty = 1 / S}$$

Why this is such bad news

Amdahl's Law: $\mathbf{T_1 / T_P = 1 / (S + (1-S)/P)}$

$\mathbf{T_1 / T_\infty = 1 / S}$

Suppose 33% of a program is sequential

- Then a billion processors won't give a speedup over 3

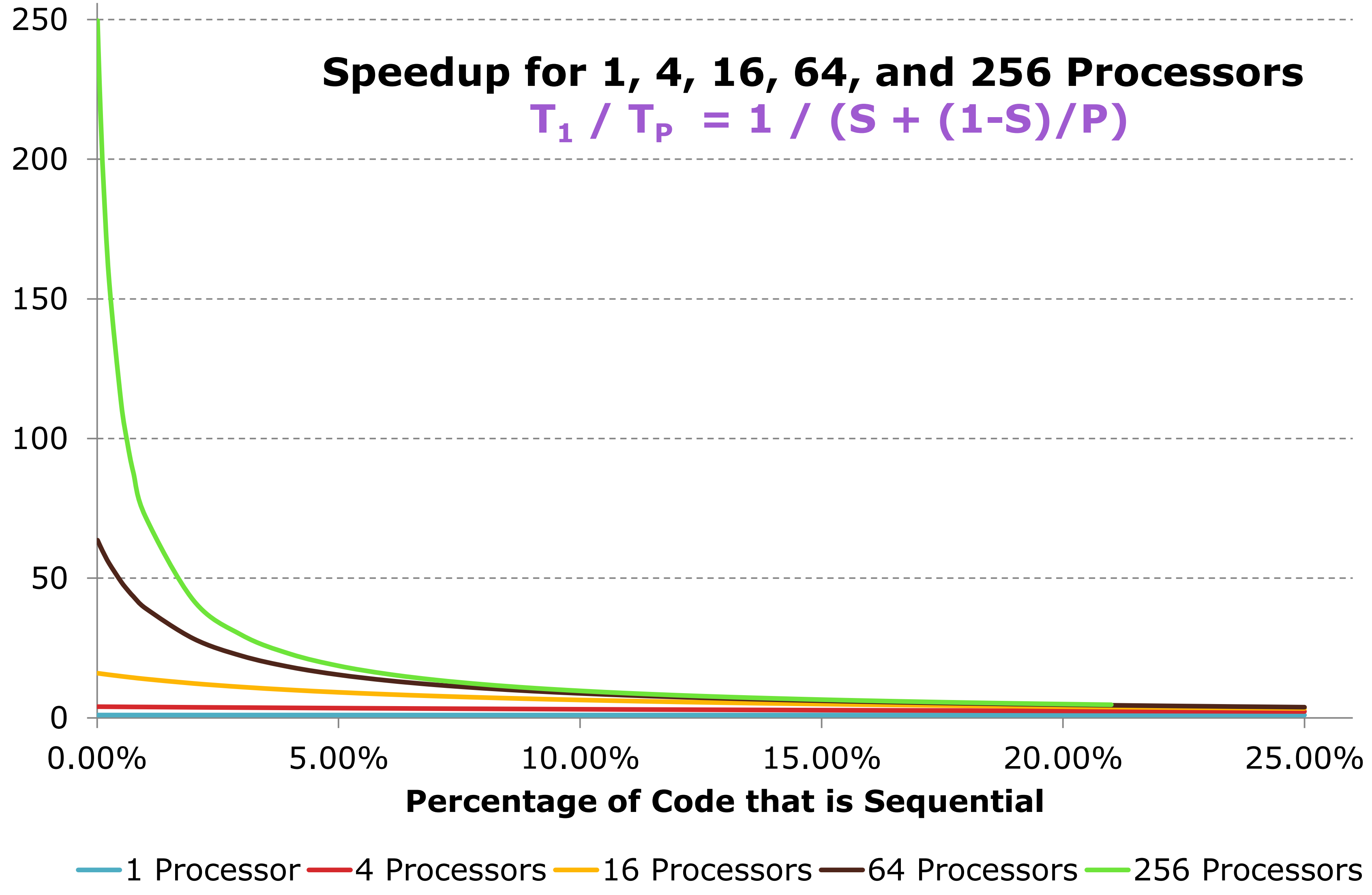
Suppose you miss the good old days (1980-2005) where 12 years or so was long enough to get 100x speedup

- Now suppose in 12 years, clock speed is the same but you get 256 processors instead of just 1
- For the 256 cores to gain ≥100x speedup, we need

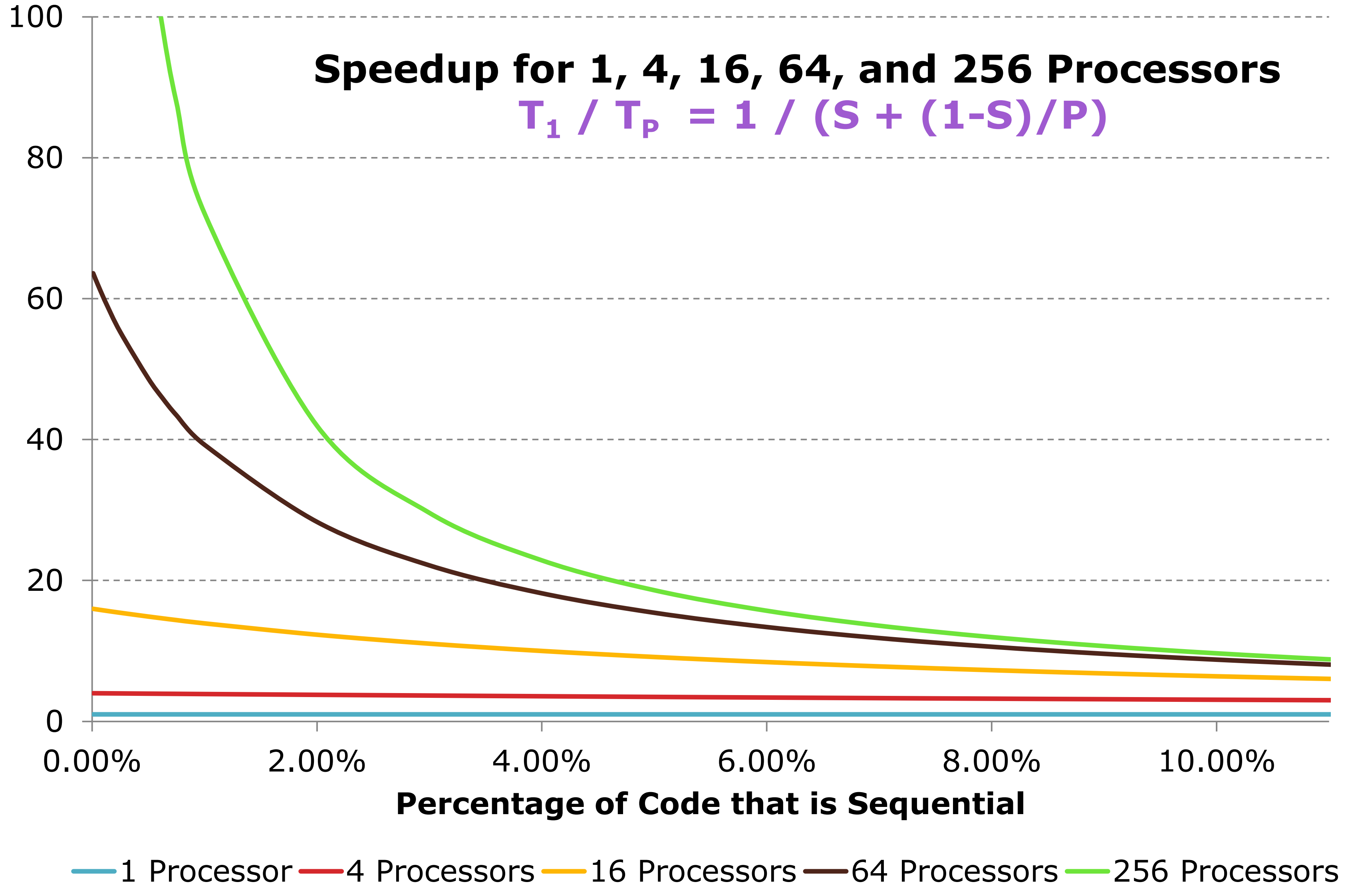
  $$100 \leq 1 / (\mathbf{S} + (1-\mathbf{S})/256)$$

  Which means **S** ≤ .0061 or 99.4% of the algorithm must be perfectly parallelizable!!

A Plot You Have To See



A Plot You Have To See (Zoomed In)



All is not lost

Amdahl’s Law is a bummer!

- Doesn’t mean additional processors are worthless!!

We can always search for new parallel algorithms

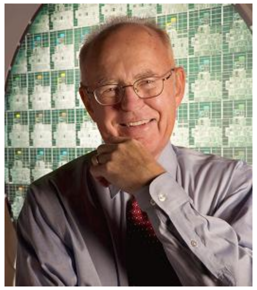
- We will see that some tasks may seem inherently sequential but can be parallelized

We can also change the problems we’re trying to solve or pursue new problems

- Example: Video games/CGI use parallelism
  - But not for rendering 10-year-old graphics faster
  - They are rendering more beautiful(?) monsters

A Final Word on Moore and Amdahl

Although we call both of their work laws, they are very different entities



Moore’s "Law" is an *observation* about the progress of the semiconductor industry:

- Transistor density doubles every ≈18 months

Amdahl’s Law is a mathematical theorem

- Diminishing returns of adding more processors

Very different but incredibly important in the design of computer systems

Welcome to the Parallel World

We will continue to explore this topic and its implications

In fact, the next class will consist of 16 lectures presented simultaneously

- I promise there are no concurrency issues with your brain
- It is up to you to parallelize your brain before then

The interpreters and captioner should attempt to grow more limbs as well