Intro to Parallelism

Data Structures and Parallelism

Logistics

Project 1 Grades are out
- Everyone did well!
- Code feedback is in feedback.md in your p1 repo
- Writeup feedback is on gradescope
- Both are also on Canvas, along with all other grades up to this point
- A sanity check to make sure your p1 code is working for p2
- Above and beyond points still pending

Lots of stuff due this week, don't let it catch you off guard
- Ex 5,6, 7 (Wed)
- Project 2 (Thurs)

As usual let me know if you plan to change partners for Project 3

Parallelism & Concurrency

All of your programs have made the same assumption

One thing happens at a time

Usually called “sequential programming”

Over the next two weeks we’ll remove this assumption

- Divide work between multiple **threads** and **synchronize** their behavior
- Design new kinds of algorithms to provide a speedup
  - More **throughput**
- Decide how to allow **concurrent access** to data among all the threads.

Why are we doing this?

Parallelism is where computation is heading.

From 1980-2005 (ish) desktop computers got twice as fast every 18 months or so.

- Moore's Law. Not an immutable law of nature. Business decision.
- How? Keep making everything smaller

Code not running fast enough? It'll be four times as fast if you just buy a new computer.

Why are we doing this?

End of Moore's Law

We're at the limit of our ability to shrink processors.

- Transistors are really small (much smaller and quantum mechanics kicks in)
- and get really hot.

Computer Architects are working very hard to still speed up processors just a little bit more.

- Take an architecture class to get a taste.

But to really achieve a speedup, the solution has been more processors.

Why are we doing this?

Parallelism is where the world is heading.

Our computers are still getting faster by adding more processors

- Rather than just making each new one twice as fast.

If we want to solve new, bigger problems, we're going to need to take advantage of more than one processor.

We won't forget about sequential/single processor programming.

- It will still be simpler and good enough most of the time.

But understanding parallelism is more important than ever.

New Text

Parallelism and Concurrency topic resource is by Dan Grossman

Available in full as a PDF on the website (no excuse not to read it!)

Parallelism vs. Concurrency

Parallelism: Use extra resources (i.e. processors) to solve your problem faster

Concurrency: Correctly and efficiently sharing a single resource among multiple threads.

Terms aren’t completely standard.

They overlap somewhat.

Analogies

Cooking:

Parallelism:

I have hundreds of potatoes to slice.

Get 20 extra cooks (and knives)

Hand them all a bunch of potatoes

Concurrency:

10 cooks are trying to share 4 burners

And one oven

Examples

Parallelism:

I want to sum up all the elements in an array

Divide the array in 4, sum up each piece in a different thread

Add together the threads' answers for the final answer

Concurrency:
Two users are trying to add an entry to a hash table at the same time.

What if the hashes collide? What if they're the same key and different values?

Sharing Memory with Threads

Our parallelism model will be shared memory with threads.

- There are other models (see Grossman), we won’t use them.

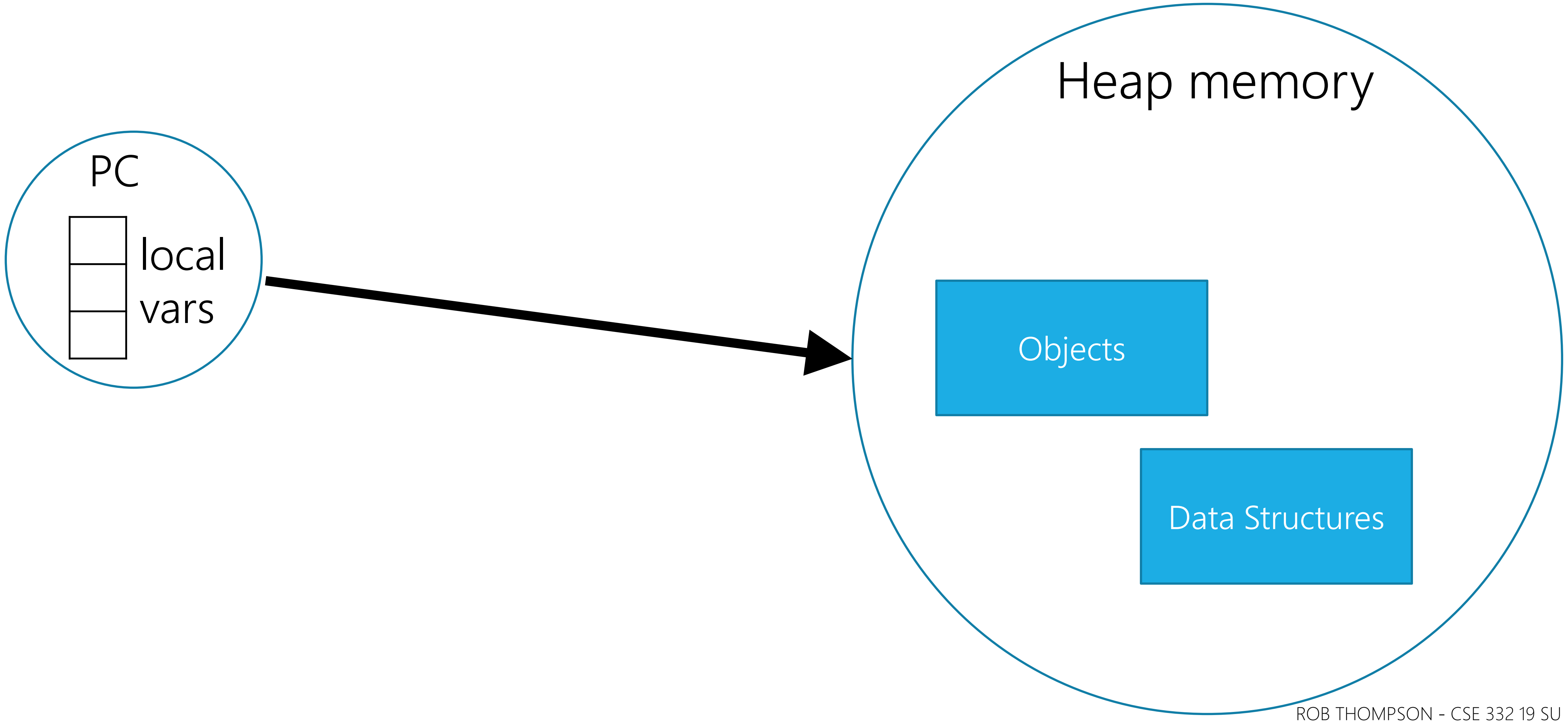
Sequential Story:

- One program counter
- One call stack
- `new` Objects go in the heap

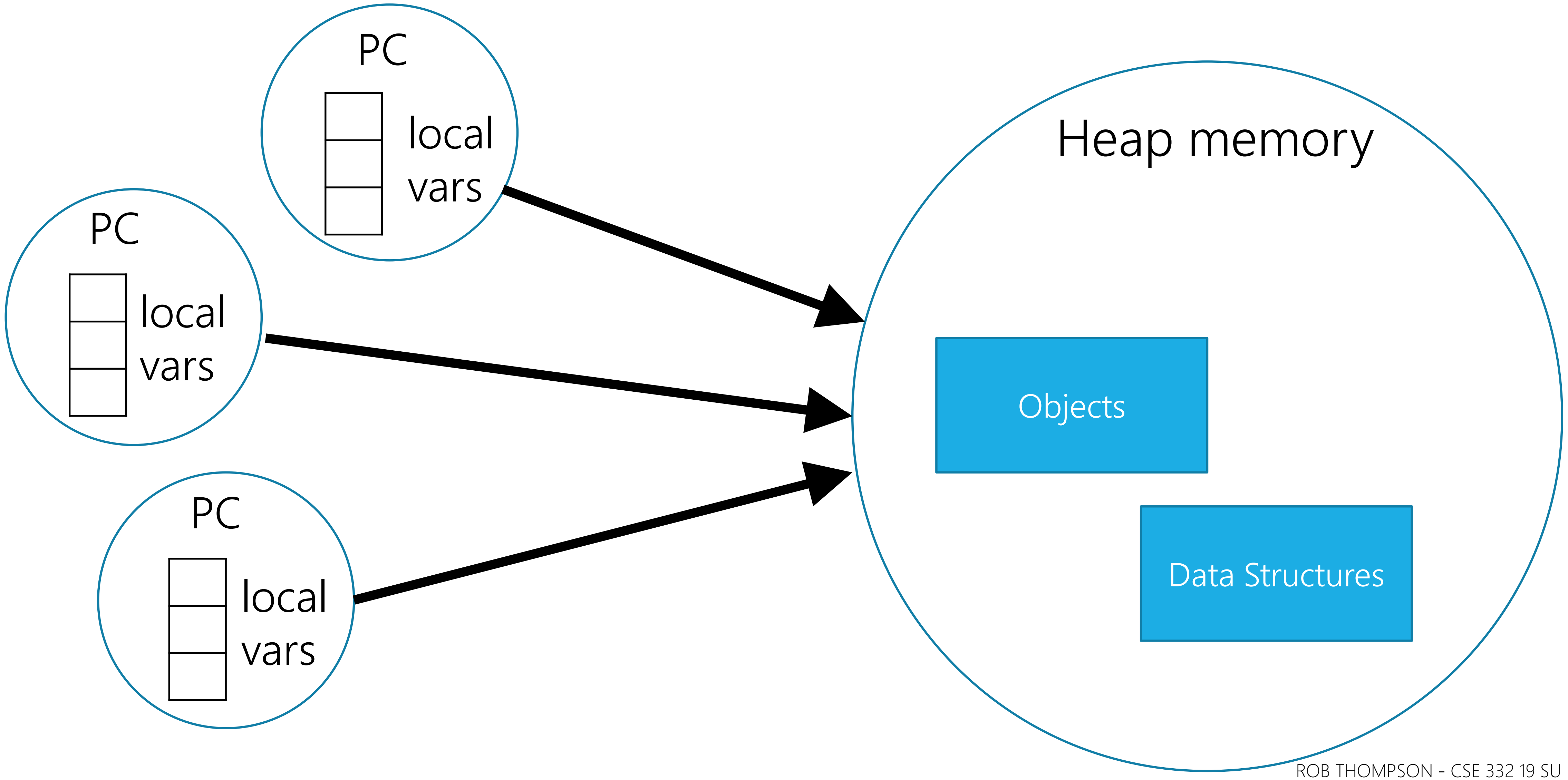
Parallel Story

- Set of threads. Each has its own program counter and its own stack
- Threads will (implicitly) share objects and static fields
- Threads communicate by altering memory.

Sequential Code



Parallel Code



We need new primitives

To write parallel programs we need a library with:

Ways to create and run multiple things at once
- i.e. threads

Ways for threads to share memory
- Usually just having the same references

Ways for threads to coordinate
- This week: A way for threads to wait for others to finish
- Next week: prevent others from accessing memory until we're done

For Today

We’ll only write pseudocode (we’ll introduce the library on Wednesday)

Parallelism requires a different mode of thinking

Just going to practice that on an example problem

A Simple Problem

Goal: Given an array, sum up all the elements.

First idea: Start up 4 threads. Each sums ¼ of the array.

Then add together those answers.

ParallelSum: Take 1

```
Class SumThread extends SomeThreadObject{
     int lo; int hi; int[] arr;
     int ans = 0; //result
     SumThread(int[] a, int l, int h){
          lo = l; hi=h; arr=a;
     }
     void compute(){
          for(i=lo; i<hi; i++)
               ans+=arr[i];
     }
}
```

ParallelSum: Take 1

There are major bugs with this code. Find some of them!

```
int sum(int[] arr){
	int len = arr.length;
	int ans = 0;
	SumThread[] ts = new SumThread[4];
	for(int i=0; i<4; i++)
		ts[i] = new SumThread(arr, i*len/4 (i+1)*len/4);
	for(int i=0; i<4; i++)
		ans += ts[i].ans;
	return ans;
}
```

Bugs

We made some Thread objects…

- but we never actually started them. They're just sitting there.
- Be careful what method you call!
- Libraries will have different methods for
  - "look at this thread object, run the code IN YOURSELF not in that thread."
  - "look at this object, tell THAT THREAD to run its code."

`threadInstance.compute`	(Run this code sequentially in the current thread)
`threadInstance.fork`	(Split off a new thread and run the code there)

ParallelSum: Take 2

```
int sum(int[] arr){
    int len = arr.length;
    int ans = 0;
    SumThread[] ts = new SumThread[4];
    for(int i=0; i<4; i++)
        ts[i] = new SumThread(arr, i*len/4 (i+1)*len/4);
        ts[i].fork();
    for(int i=0; i<4; i++)
        ans += ts[i].ans;
    return ans;
}
```

Bugs

The current thread is still running.

Will each thread update its ans field in time?

Need to tell original thread to WAIT for its children to finish.

ParallelSum: Take 3

```
int sum(int[] arr){
      int len = arr.length;
      int ans = 0;
      SumThread[] ts = new SumThread[4];
      for(int i=0; i<4; i++)
            ts[i] = new SumThread(arr, i*len/4 (i+1)*len/4);
            ts[i].fork()
      for(int i=0; i<4; i++)
            ts[i].join()
            ans += ts[i].ans;
      return ans;
}
```

Join

Parallelism libraries will define methods you can't implement on your own.

- E.g. whatever method starts a new thread isn't something you can do yourself.

`Join` is our first taste of coordinating computation

- Calling thread blocks (just sits there doing nothing) until receiver returns
- Avoids **race condition** in our original code.

- This style of programming is called "fork/join"
- Java note: `join` can throw exceptions. May not compile unless you catch a java.lang.InterruptedException
- A simple try-catch block should be fine for simple code.

(Almost) No Shared Memory!

Fork/join programs like these don’t really share memory

We divided up the array – no one tried to access the same locations.

Lo, hi, arr fields weren’t shared. Each helper thread had those values written by the main thread.

Main thread gets data back – doesn’t let the helper threads alter any shared data themselves.

To avoid race conditions, we’ll use `join`

Next week, we’ll see other ways to synchronize.

Optimizing: Number of Threads

The last version of ParallelSum will work.

I.e. it will always get the right answer

And it will use 4 threads.

But…

What if we get a new computer with 6 processors?

- We'll have to rewrite our code

What if the OS decides "no, you only get 2 processors right now."

What if our threads take wildly different amounts of time?

Optimizing: Number of Threads

The counter-intuitive solution:

Even more parallelism!

Divide the work into more smaller pieces.
If you get more processors, you take advantage of all of them.
If one thread finishes super fast, throw the next thread to that processor.

Engineering Question:

- Let’s say we change our ParallelSum code so each thread adds 10 elements.
- Is that a good idea? What’s the running time of the code going to be?

Divide and Conquer: Parallelism

What if we want a bunch of threads, but don't want to spend a bunch of time making threads?

Parallelize thread creation too!

Remember merge-sort?

Divide and Conquer SumThread

```
Class SumThread extends SomeThreadObject{
      //constructor, fields unchanged.
      void run(){
            if(hi-lo == 1)
                  ans = arr[lo]
            else{
                  SumThread left = new SumThread(arr, lo, (hi+lo)/2);
                  SumThread right = new SumThread(arr, (hi+lo)/2, hi);
                  left.fork(); right.fork();
                  left.join(); right.join();
                  ans = left.ans + right.ans;
} } }
```

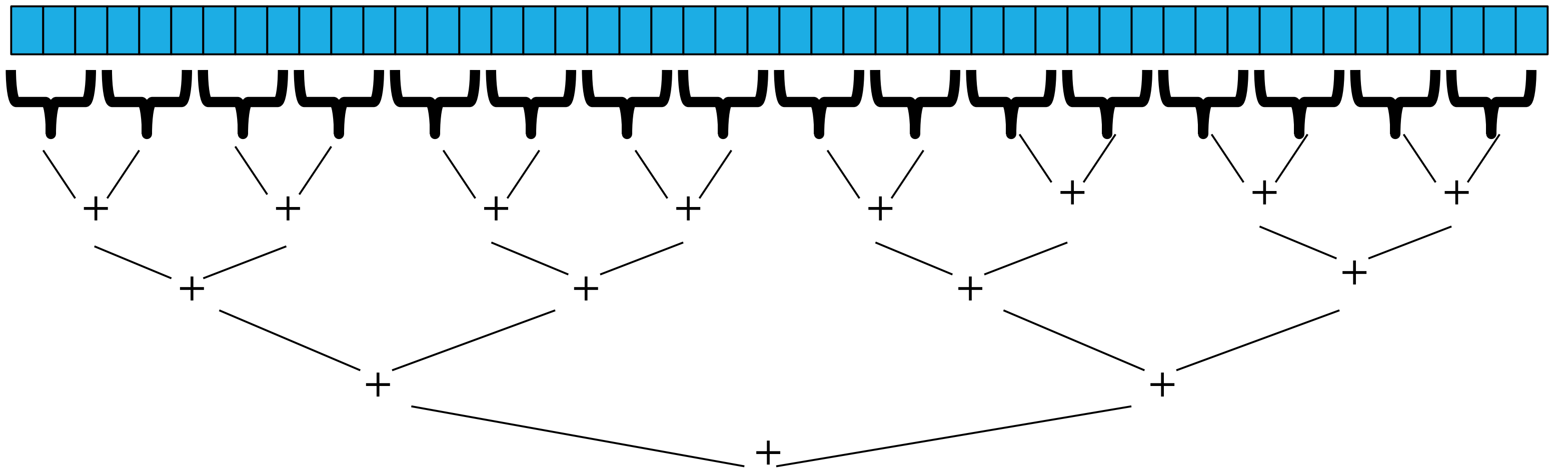
Divide And Conqure SumThread

```
int sum(int[] arr){
  SumThread t = new SumThread(arr, 0, arr.length);
  t.compute(); //this first call isn’t making a new
thread
  return t.ans;
}
```

Divide and Conquer Runtime

Dividing the work by 2 each time reduces it logarithmically

What is the height of our thread tree?



Divide And Conquer Optimization

Imagine calling our current algorithm on an array of size 4.

How many threads does it make

6

It shouldn't take that many threads to add a few numbers.

And every thread introduces A LOT of overhead.

We'll want to **cut-off** the parallelism when the threads cause too much overhead.

Similar optimizations can be used for (sequential) merge and quick sort

Cut-offs

Are we really saving that much?

Suppose we're summing an array of size 2^{30}

And we set a cut-off of size-100

- i.e. subarrays of size 100 are summed without making any new threads.

What fraction of the threads have we just eliminated?

99.9% !!!! (for fun you should check the math)

Divide and Conquer SumThread

```
Class SumThread extends SomeThreadObject{
    //constructor, fields unchanged.
    void run(){
        if(hi-lo <= 100)
            for(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.fork(); right.fork();
            left.join(); right.join();
            ans = left.ans + right.ans;
} } }
```

One more optimization

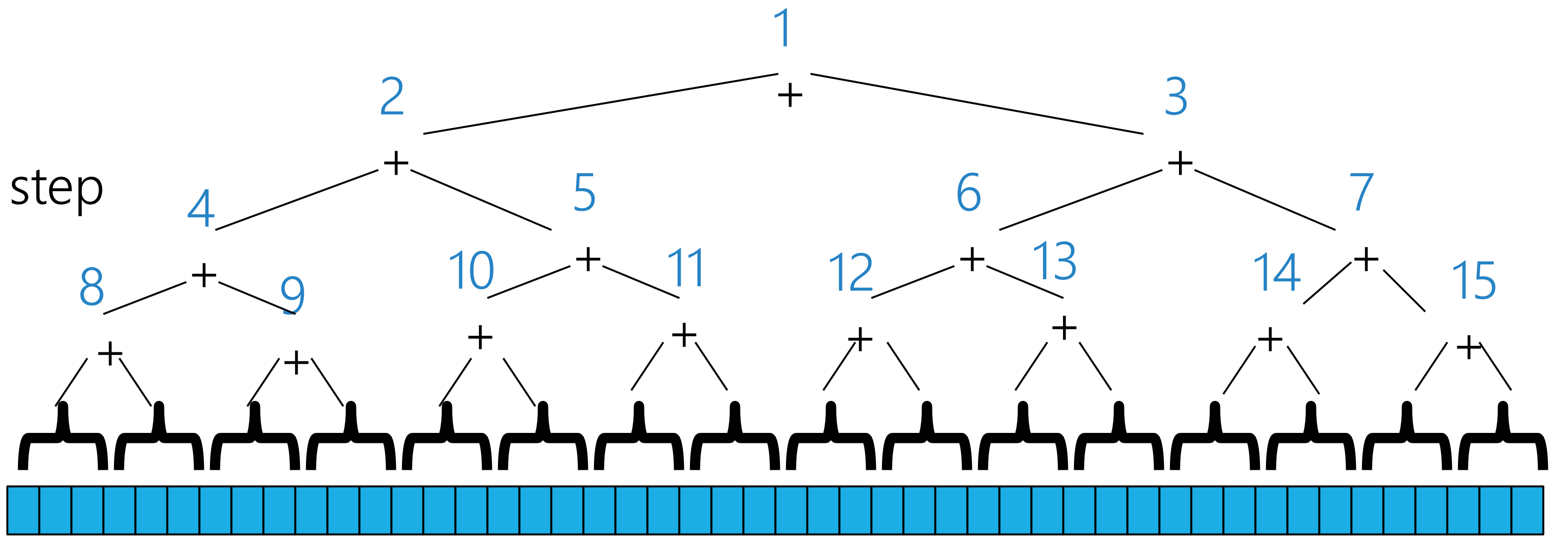
A small tweak to our code will eliminate half of our threads

```
SumThread left = new
SumThread(arr, lo, (hi+lo)/2);
SumThread right = new
SumThread(arr, (hi+lo)/2, hi);
left.fork(); right.fork();
left.join(); right.join();
```

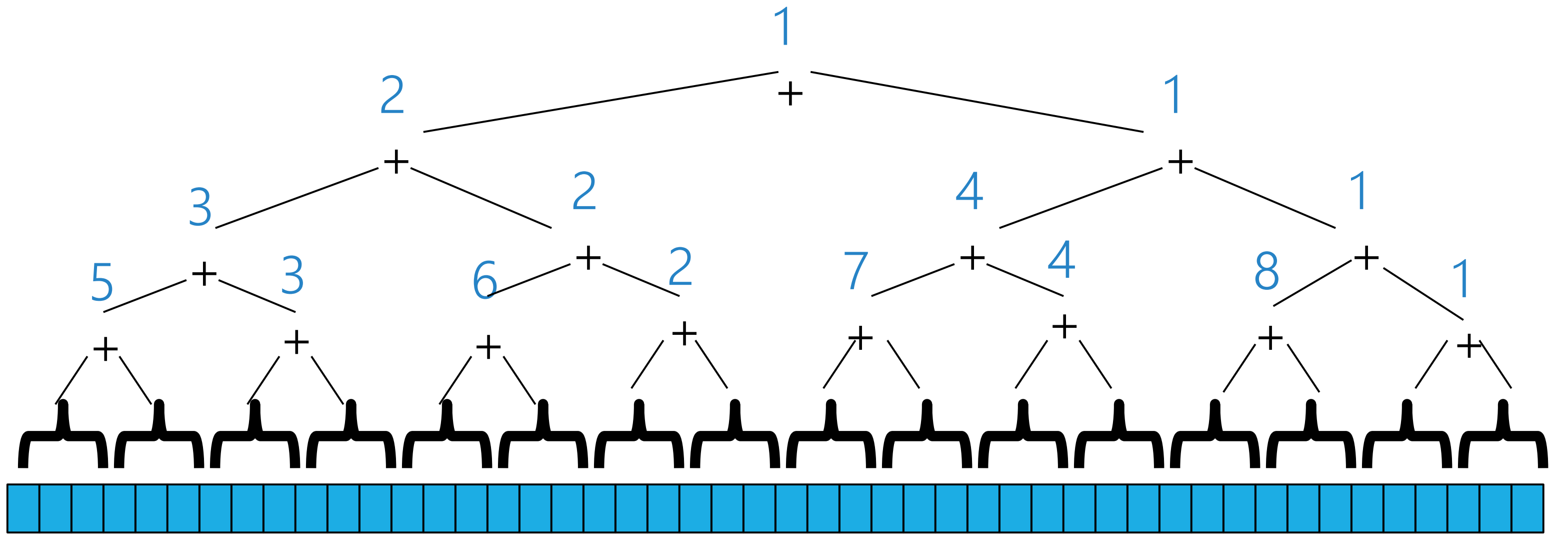
```
SumThread left = new
SumThread(arr, lo, (hi+lo)/2);
SumThread right = new
SumThread(arr, (hi+lo)/2, hi);
left.fork();
right.compute();
left.join();
```

Creating Fewer threads pictorially

2 new threads at each step (and only leaf threads do much work)
Total =
15 threads



1 new thread at each step
Total =
8 threads



Wrap Up

None of our optimizations will make a difference in the O() analysis

- Which we'll see on Wednesday

But they will make a difference in practice.

Wednesday:
Using a real library

Analyzing parallel programs.