CSE 332: Data Structures & Parallelism
Lecture 13: Parallelism Intro

Arthur Liu

Summer 2022

Announcements

- Good news: We’re going to delay P2 deadline to Thursday
  - Writeup takes a lot of time!
- Good news: Your lowest 2 exercises will be dropped (and count towards “above & beyond” otherwise)
- EX10 released tonight
  - Will be due next Monday

Outline

- Finish QuickSort
- Comparison Sorting Bound
- Non-comparison Sorts

- Possibly, Parallelism Intro

The Parallelism Part of this class

- Introduction of Parallelism Ideas
  - Java’s Thread
  - ForkJoin Library
- General Parallelism Algorithms
  - Reduce, Map
  - Analysis (span, work)
- Clever Parallelism Ideas
  - Parallel Sorts
  - Parallel Prefix
- Synchronization
  - The need for locks (Concurrency)
- Other Synchronization Issues
  - Race Conditions: Data Races & Bad Interleavings

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.

Parallelism vs. Concurrency

(Read Grossman text!!! Very short and digestible pdf, and free)

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

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

There is some connection (confusion!) between them

Analogies

Parallelism:

I have hundreds of potatoes to slice.

Get 20 extra cooks (and knives)

Hand them all a bunch of potatoes

Concurrency:

The 20 cooks are trying to share four 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

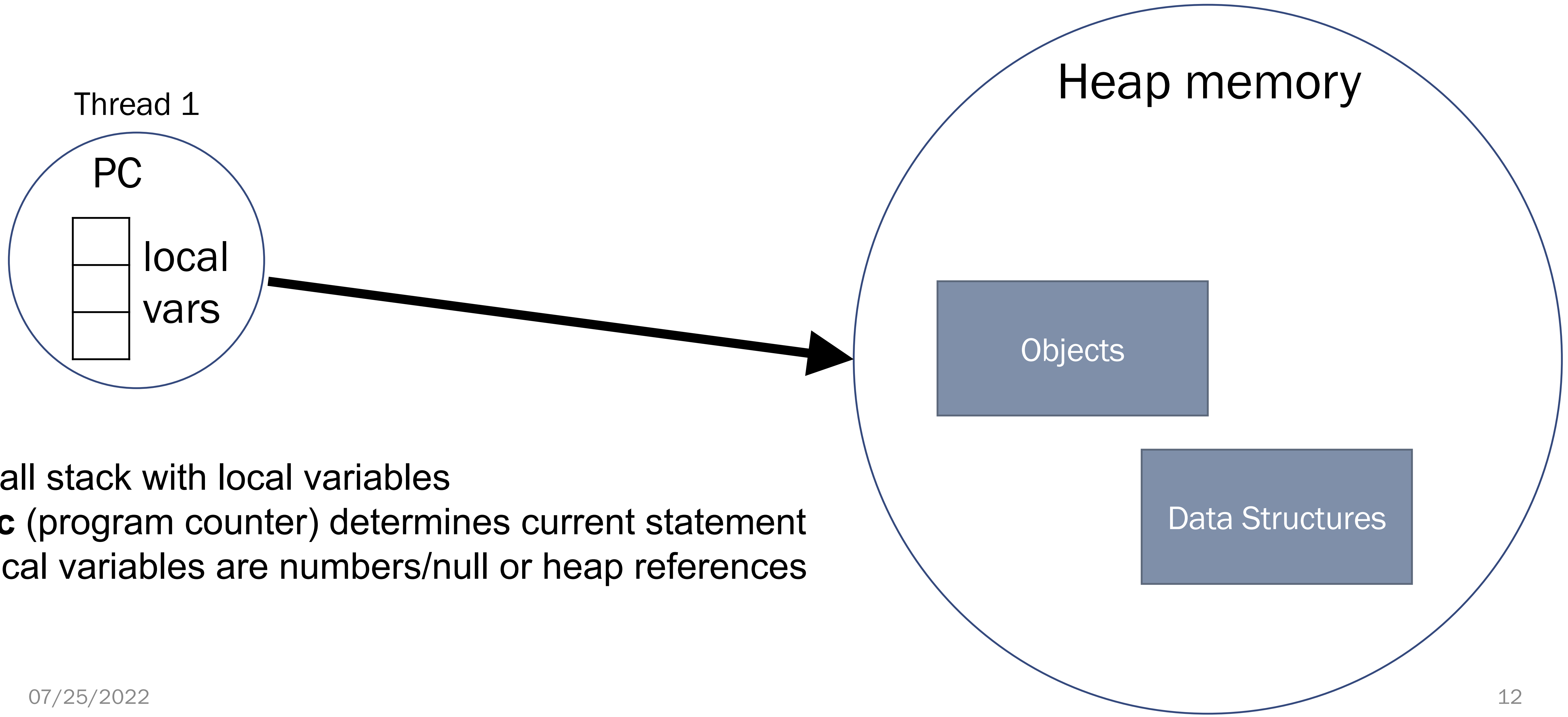
Concurrency:
Two users are trying to add an entry to a hash table at the same time.

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

Sharing Memory with Threads

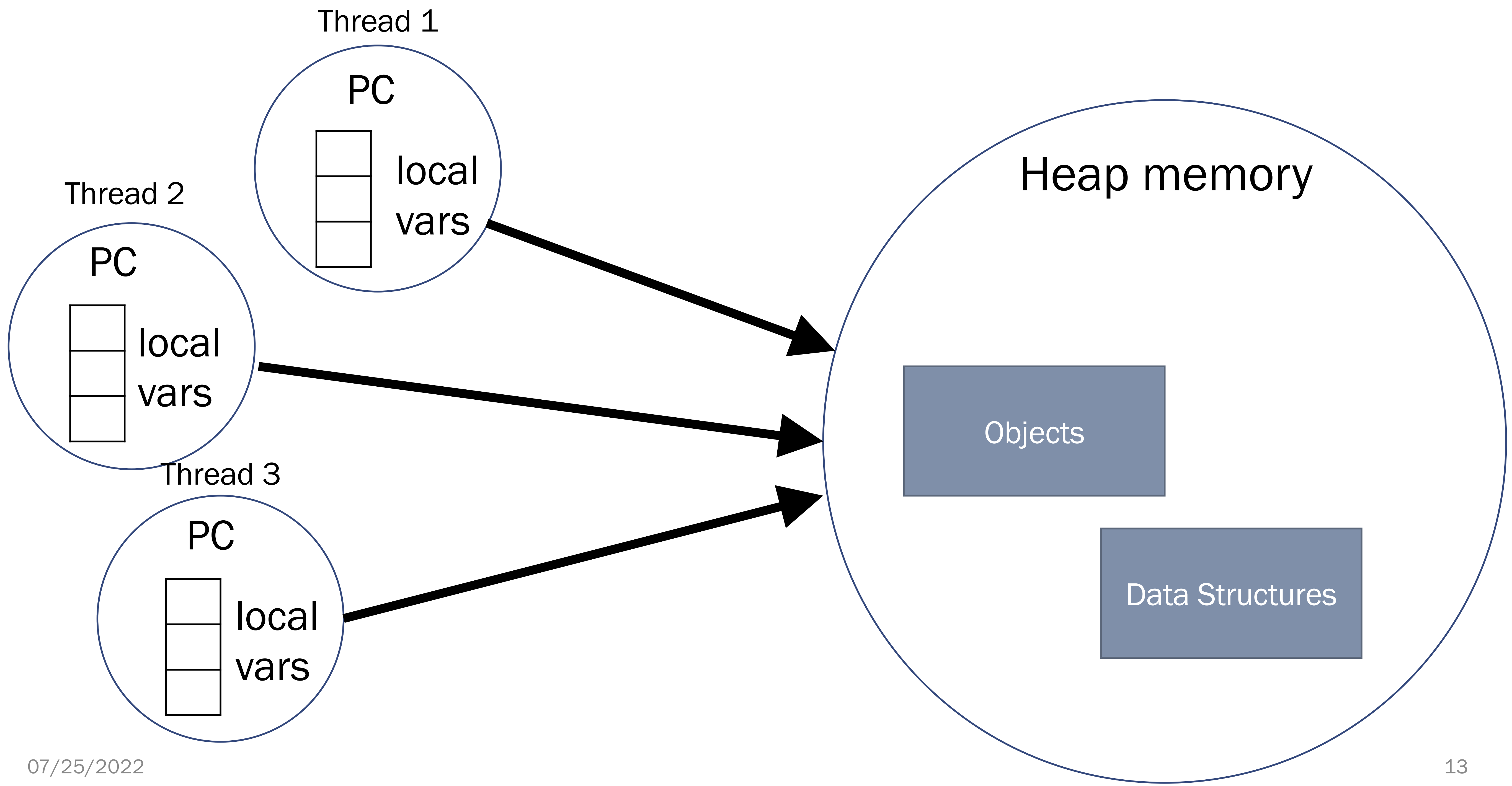
- Our parallelism model will be shared memory with threads.
  - There are other models (see Grossman), we won't use them.
- Sequential Story:
  - One program counter
  - One call stack
  - `new` Objects go in the heap
- Parallel Story
  - Set of threads. Each has its own program counter and its own stack
  - Threads will (implicitly) share objects and static fields
  - Threads communicate by altering memory.

Sequential Code



- Call stack with local variables
- **pc** (program counter) determines current statement
- local variables are numbers/null or heap references

Parallel Code



Other Models

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

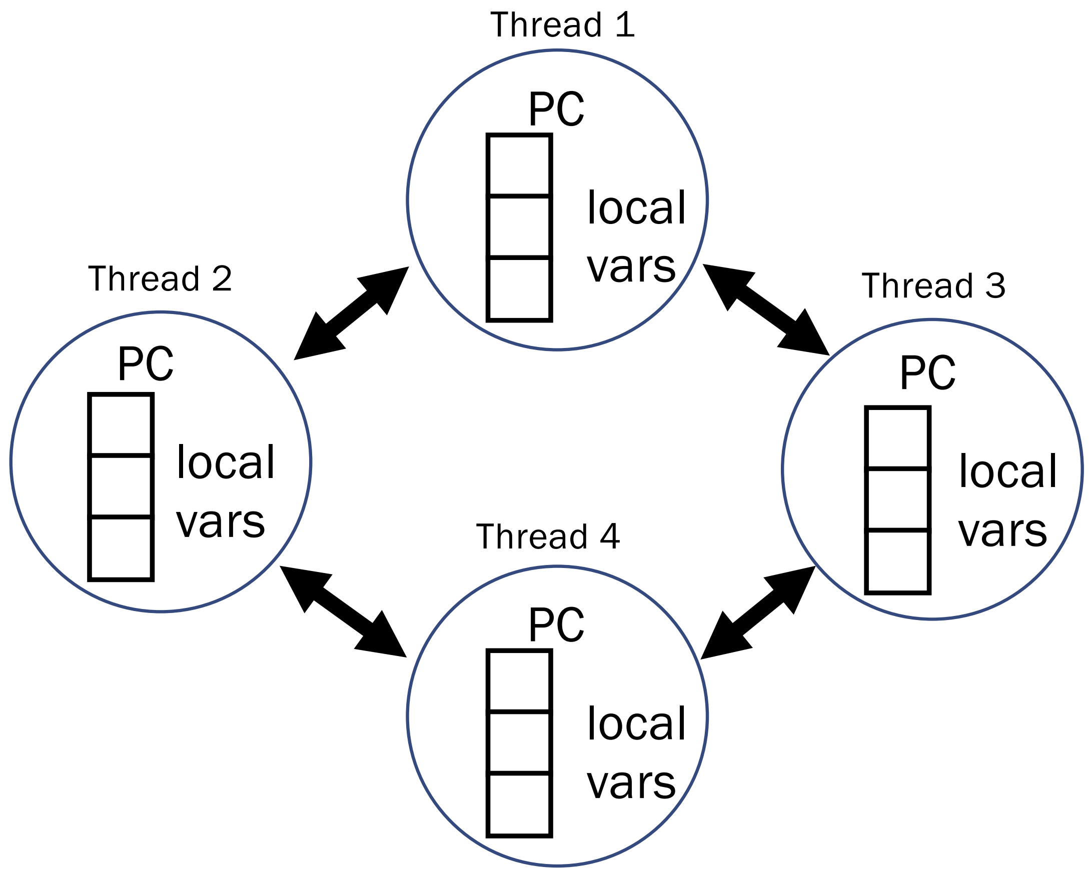
Message-passing: Each thread has its own collection of objects. Communication is via explicitly sending/receiving messages

Cooks working in separate kitchens, mail around ingredients

Dataflow: Programmers write programs in terms of a DAG. A node executes after all 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"



We need new primitives

To write parallel programs we need a library with:

- Ways to create and run multiple things at once
  - Let’s call these things 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

Java basics

First learn some basics built into Java via **`java.lang.Thread`**

- Then a better library for parallel programming

To get a new thread running:

1. Define a subclass **`C`** of **`java.lang.Thread`**, overriding **`run`**
2. Create an object of class **`C`**
3. Call that object's **`start`** method
    - **`start`** sets off a new thread, using **`run`** as its "main"

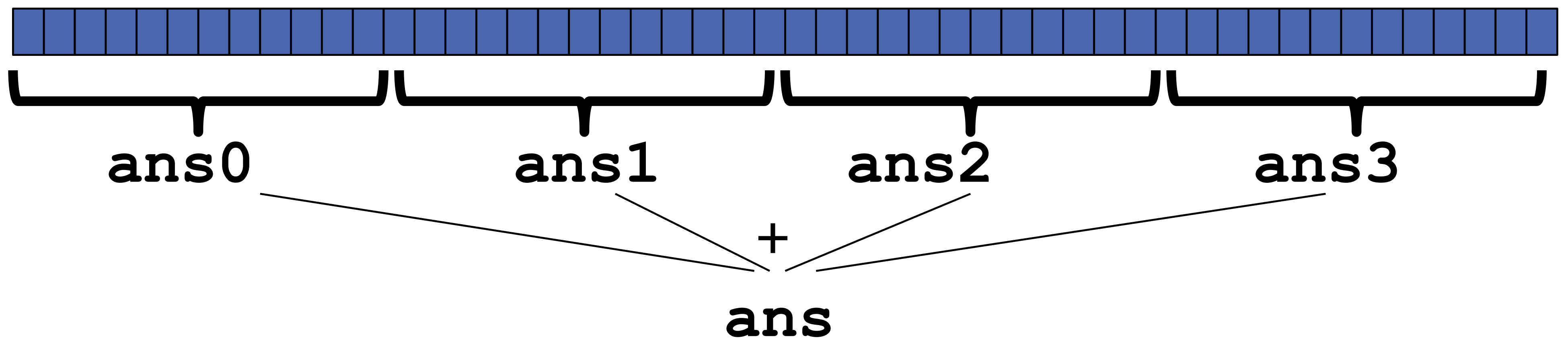
What if we instead called the **`run`** method of **`C`**?

- This would just be a normal method call, in the current thread

Let's see how to share memory and coordinate via an example…

Parallelism idea

- Example: Sum elements of a large array
- Idea: Have 4 threads simultaneously sum 1/4 of the array
  - Warning: This is an inferior first approach



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

First attempt, part 1

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

  int lo; // fields, assigned in the constructor
  int hi; // so threads know what to do.
  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];
  }
}
```

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

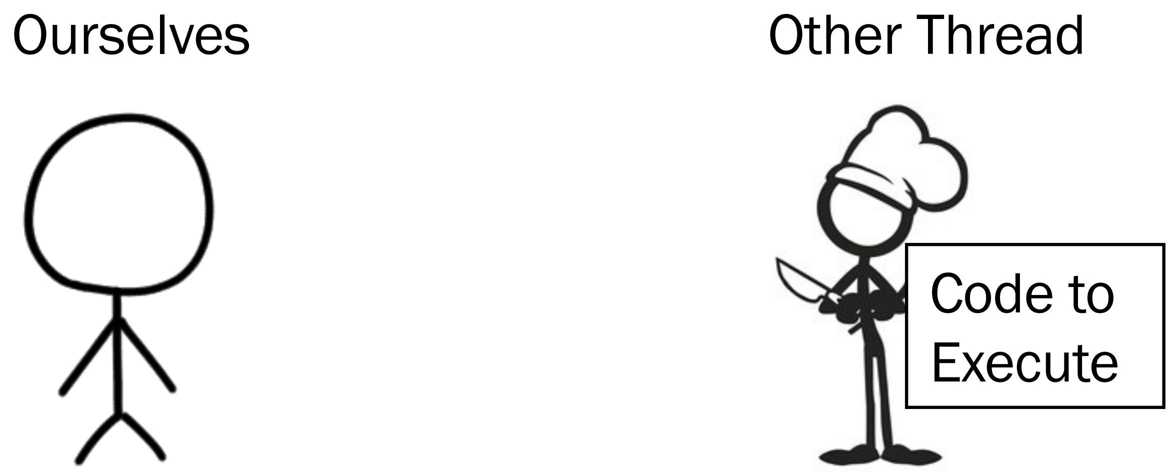
First attempt, continued (wrong)

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

```
int sum(int[] arr){
  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;
}
```

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
    - run() : "look at this thread object, run the code IN YOURSELF not in that thread."
    - start() : "look at this object, tell THAT THREAD to start and run its code."



Second attempt (still wrong)

```
class SumThread extends java.lang.Thread {
  int lo, int hi, int[] arr; // fields to know what to do
  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);
    ts[i].start(); // start not run
  }
  for(int i=0; i < 4; i++) // combine results
    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."~~
- 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.

Third attempt (correct in spirit)

```
class SumThread extends java.lang.Thread {
  int lo, int hi, int[] arr; // fields to know what to do
  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);
    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;
}
```

Join

- Parallelism libraries will define methods you can't implement on your own.
  - E.g. starting 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 on **`ts[i].ans`**

  - 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 (thankfully) do not require much 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, you must avoid race conditions
  - While studying parallelism, we'll stick with **`join`**
  - With concurrency, we will learn other ways to synchronize

A better approach

Several reasons why this is a poor parallel algorithm

1. Want code to be reusable and efficient across platforms
   - "Forward-portable" as core count grows
   - So at the *very* least, parameterize by the number of threads

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

A Better Approach

2. Want to use (only) processors "available to you *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 you have 3 processors available and using 3 threads would take time **`X`**, then creating 4 threads would take time **`1.5X`**
  - **Example: 12 units of work, 3 processors**
    - **Work divided into 3 parts will take 4 units of time**
    - **Work divided into 4 parts will take 3*2 units of time**

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

A Better Approach

3. Though unlikely for `sum`, in general subproblems may take significantly different amounts of time

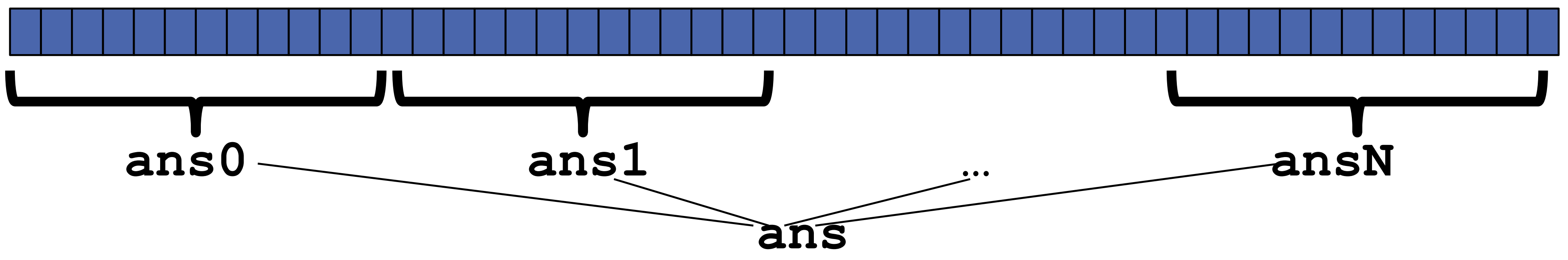
- Example: Apply method `f` to every array element, but maybe `f` is much slower for some data items
  - Example: Is a large integer prime?

- 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

The counterintuitive (?) solution to all these problems is to cut up our problem into *many* pieces, far more than the number of processors

- But this will require changing our algorithm
- And for constant-factor reasons, abandoning Java's threads



1. **Forward-portable:** Lots of helpers each doing a small piece
2. **Processors available:** Hand out "work chunks" as you go
   - If 3 processors available and have 100 threads, then ignoring constant-factor overheads, extra time is < 3%
3. **Load imbalance:** No problem if slow thread scheduled early enough
   - Variation probably small anyway if pieces of work are small

Naïve algorithm is poor

Suppose we create 1 thread to process every 1000 elements

```
int sum(int[] arr){
   …
   int numThreads = arr.length / 1000;
   SumThread[] ts = new SumThread[numThreads];
   …
}
```

Then the "combining of results" part of the code will have `arr.length / 1000` additions

- Linear in size of array (with constant factor 1/1000)
- Previous we had only 4 pieces (Θ(1) to combine)

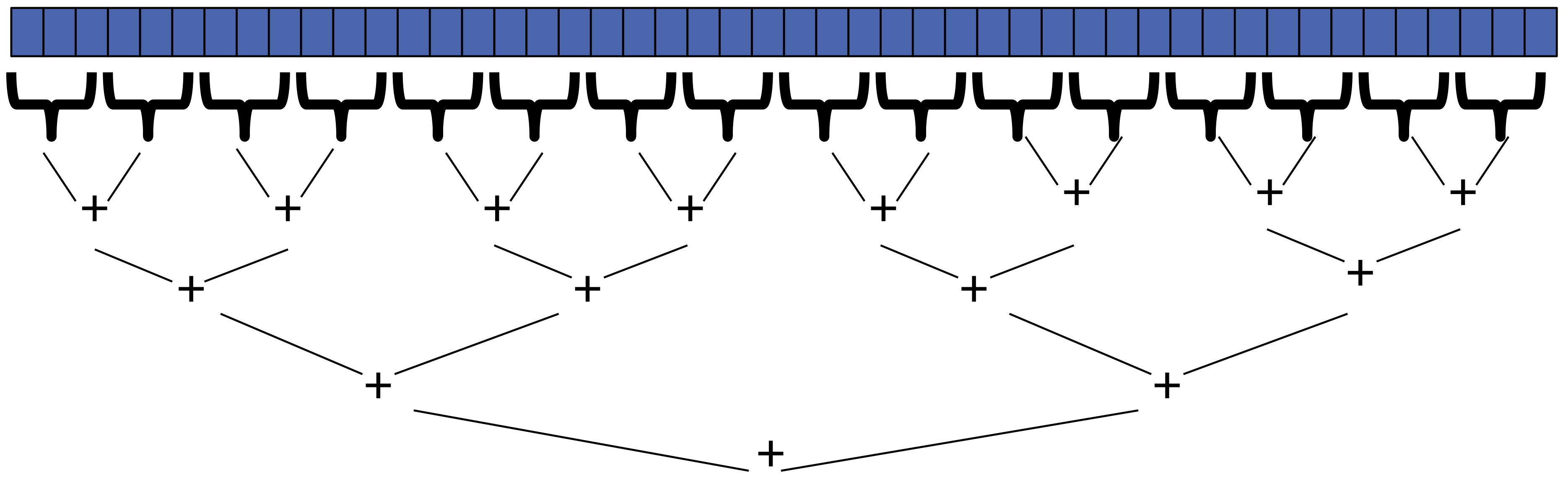
- In the extreme, suppose we create one thread per element – If we use a for loop to combine the results, we have N iterations
- In either case we get a Θ(N) algorithm with the combining of results as the bottleneck….

A better idea: Divide and Conquer!

1) Divide problem into pieces recursively:

- Start with full problem at root
- Halve and make new thread until size is at some cutoff

2) Combine answers in pairs as we return from recursion (see diagram)



This will start small, and 'grow' threads to fit the problem

This is straightforward to implement using divide-and-conquer

- Parallelism for the recursive calls

Code looks something like this (<u>still using Java Threads</u>)

```
class SumThread extends java.lang.Thread {
  int lo; int hi; int[] arr; // fields to know what to do
  int ans = 0; // result
  SumThread(int[] a, int l, int h) { … }
  public void run(){ // override
    if(hi - lo == 1)
      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){ // just make one thread!
   SumThread t = new SumThread(arr,0,arr.length);
   t.run();
   return t.ans;
}
```

Divide And Conquer Optimization

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

How many threads does it make?

7

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

Code looks something like this (still using Java Threads)

```
class SumThread extends java.lang.Thread {
  int lo; int hi; int[] arr; // fields to know what to do
  int ans = 0; // result
  SumThread(int[] a, int l, int h) { … }
  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){ // just make one thread!
   SumThread t = new SumThread(arr,0,arr.length);
   t.run();
   return t.ans;
}
```

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)

One more optimization

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

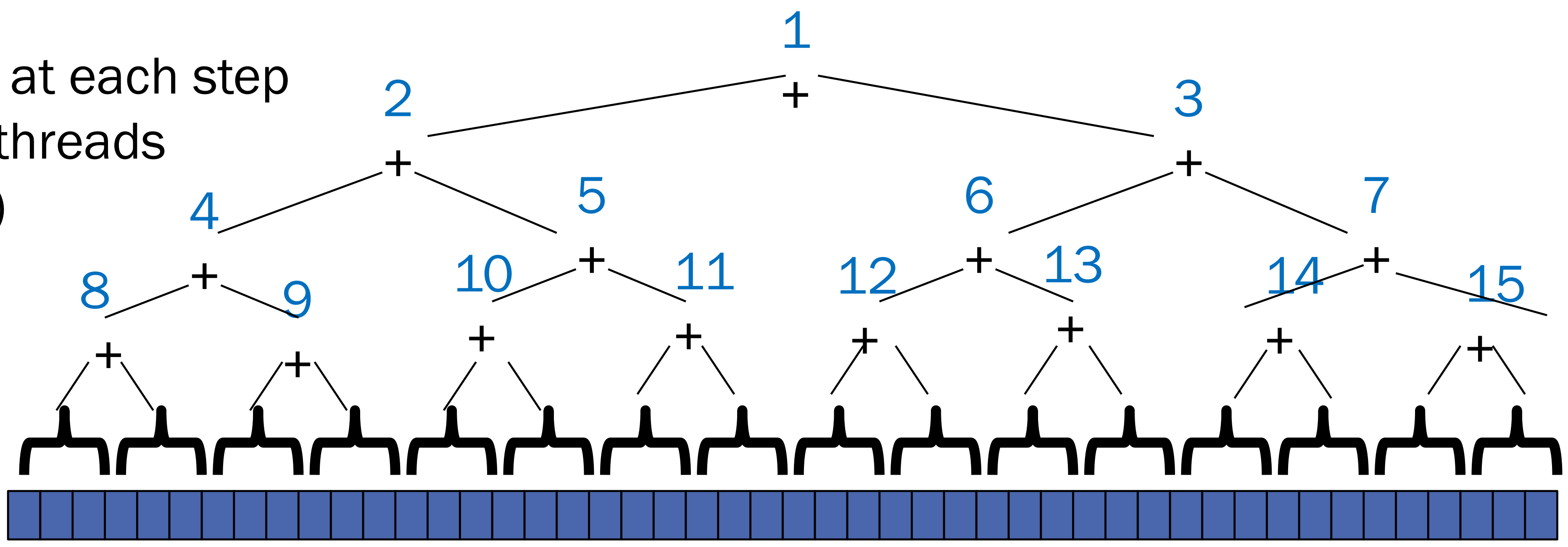
```
// wasteful: don't
SumThread left  = …
SumThread right = …

left.start();
right.start();



left.join();
right.join();
ans=left.ans+right.ans;
```

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

left.start();
right.run();



left.join();
// no right.join needed!
ans=left.ans+right.ans;
```

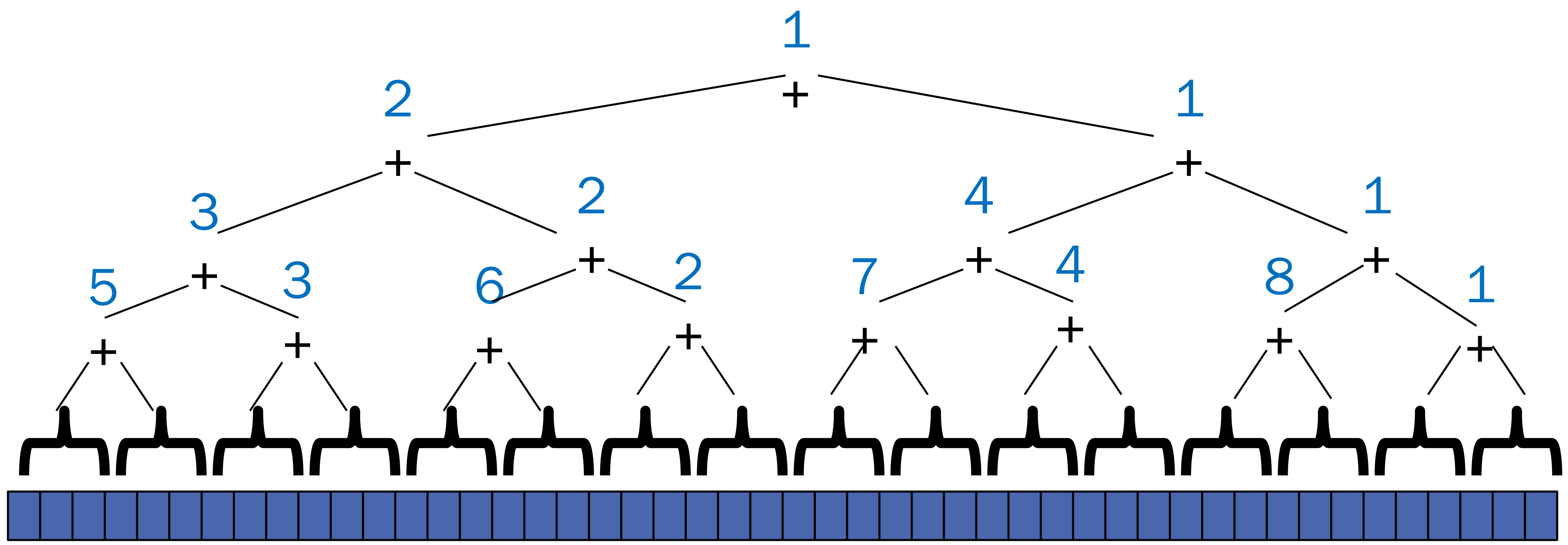
Current thread actually executes the right hand side.
Ordering of these commands is very important!

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



Analysis

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

But they will make a difference in practice.

That library, finally

- 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 to meet the needs of divide-and-conquer fork-join parallelism
  - In the Java 8 standard libraries
  - Section will focus on pragmatics/logistics
  - Similar libraries available for other languages
    - C/C++: Cilk (inventors), Intel's Thread Building Blocks
    - C#: Task Parallel Library
    - …
  - Library's implementation is a fascinating but advanced topic

ForkJoin Library

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.RecursiveAction;
```

Two possible classes to extend

`RecursiveTask<E>`

- Returns an E object

`RecursiveAction`

- Doesn’t return anything.

First thread created by:

```
POOL.invoke( ThreadObject );
```

ForkJoin Library summary

- Start (and run) a new thread: `fork`()
- Wait for a thread to finish: `join`()
    - `join`() will return an object, if you extended `RecursiveTask`
- Your Thread objects need to write a `compute`() method
    - This is the code that gets executed
    - Calling `compute()` does NOT start a new thread in the JVM.

- Use the ForkJoinPool to start off the initial task (instead of a top-level call to run)

Fork Join Framework Version: (missing imports)

```
class SumTask extends RecursiveTask<Integer> {
  int lo; int hi; int[] arr; // fields to know what to do
  SumTask(int[] a, int l, int h) { … }
  protected Integer compute(){// return answer
    if(hi - lo < SEQUENTIAL_CUTOFF) {
      int ans = 0; // local var, not a field
      for(int i=lo; i < hi; i++)
        ans += arr[i];
      return ans;
    } else {
      SumTask left = new SumTask(arr,lo,(hi+lo)/2);
      SumTask right= new SumTask(arr,(hi+lo)/2,hi);
      left.fork(); // fork a thread and calls compute
      int rightAns = right.compute();//call compute directly
      int leftAns  = left.join(); // get result from left
      return leftAns + rightAns;
    }
  }
}
```

```
static final ForkJoinPool POOL = new ForkJoinPool();
int sum(int[] arr){
   SumTask task = new SumTask(arr,0,arr.length)
   return POOL.invoke(task);
   // invoke returns the value compute returns
}
```