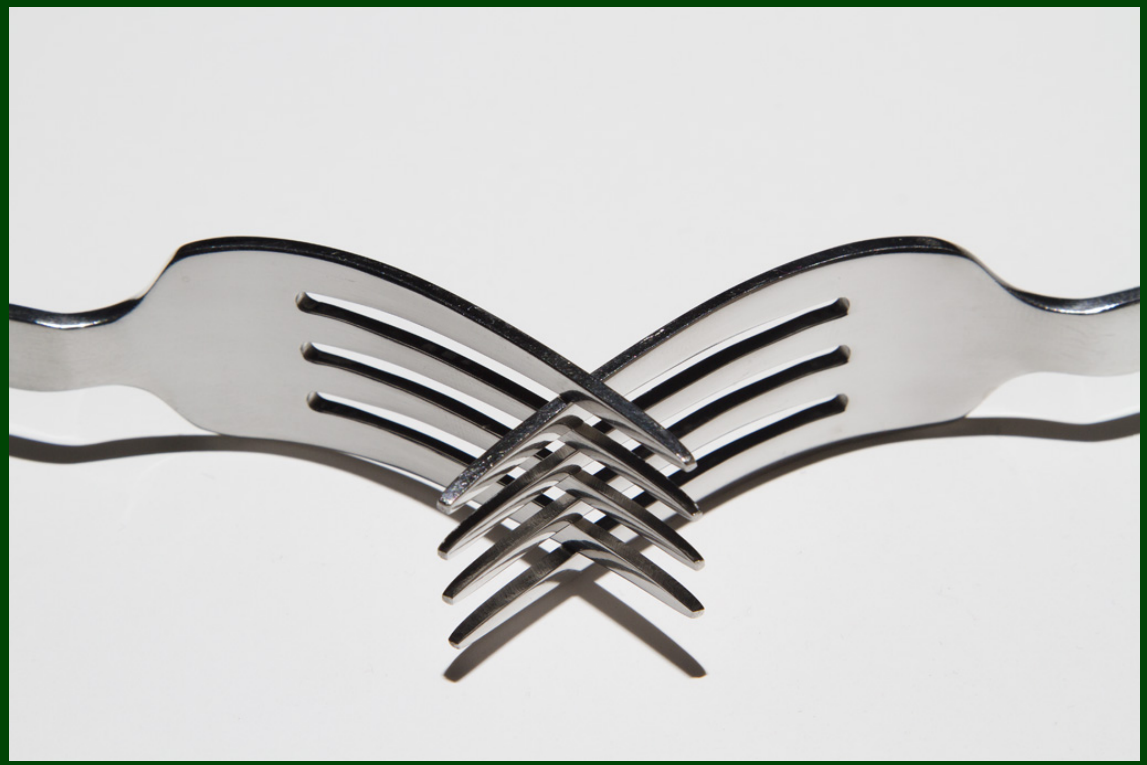
CSE 332

Data Structures and Parallelism

Introduction to `ForkJoin` Parallelism



Where We Are

This course was designed to be more "modern" than your standard "data structures" course. The next few weeks are where this really shines!

For the duration of the course, we will

Drop the assumption that only one thing is happening at a time!

In doing so, we go from Sequential Programming to

Parallel Programming

This brings with it a bunch of **benefits** and **challenges**.

Benefits of Parallelism

It Makes Code Faster!

Computers are cheap, but time isn't.

(Also, it takes into account the reality that single CPUs can't keep up with the speed we want things to run at.)

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

Definition (Concurrency*)

Programming as the composition of independently executing computations.

> Concurrency is about dealing with lots of things at once.
>
> (Rob Pike)

Concurrency Examples

- That time you **simultaneously** opened 70 tabs in your browser and they loaded "at the same time" (Time-slicing)
- Your computer can handle keyboard and mouse input at the same time
- If you run one program, you can still run others

Notice that these examples all involve concurrency **over a single CPU**. It doesn't have to be that way.

*Note that we **will not** cover concurrency over a single CPU in this course. See CSE 333.

Definition (Parallelism)

Programming as the simultaneous execution of (possibly related) computations.

Parallelism is about doing lots of things at once.

(Rob Pike, again)

Parallelism Example (please indulge me)

1. You have an index card. If your birthday is in March, write a 1 on the card. Otherwise, write 0.
2. Hand your card off to someone who only has one card. If you now have two cards, add the numbers on the two cards, write the new number on one of the cards, and discard the other one.
3. Repeat step two until there is only one person with a card.

In this example, each of you were **a thread**, you executed **simultaneously**, and, perhaps most importantly, did it really quickly.

Definition (Synchronization)

Dealing with the access/editing of a shared resource among concurrently executing programs.

Synchronization Example

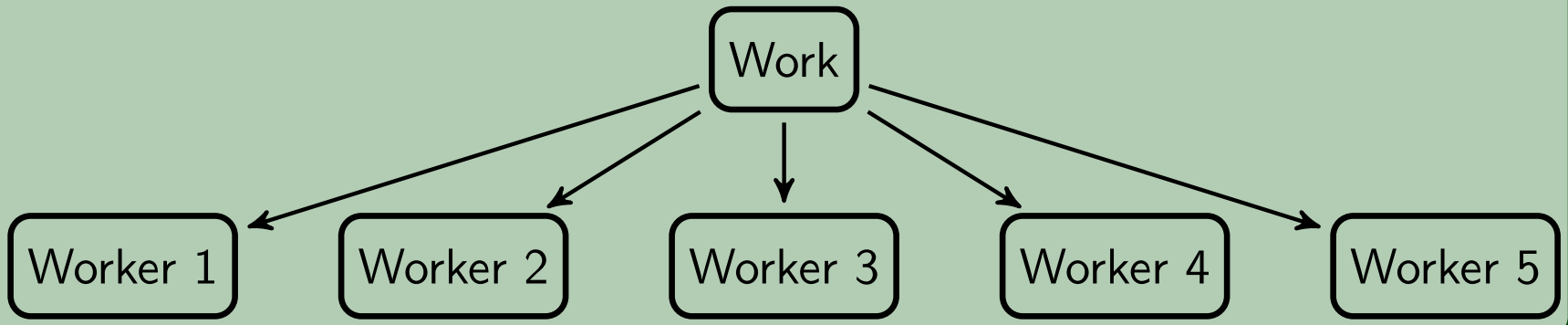
(We won't actually do this one.)

1. In the front of the room, there are 12 index cards and twelve pencils (one corresponding to each index card). One pair for each month of the year.
2. First, get the pencil corresponding to your birth month, then add 1 to the number already on the card.

In this example, each of you were **a thread**, the **index cards** were a hash table, and the **pencils** provided synchronization.

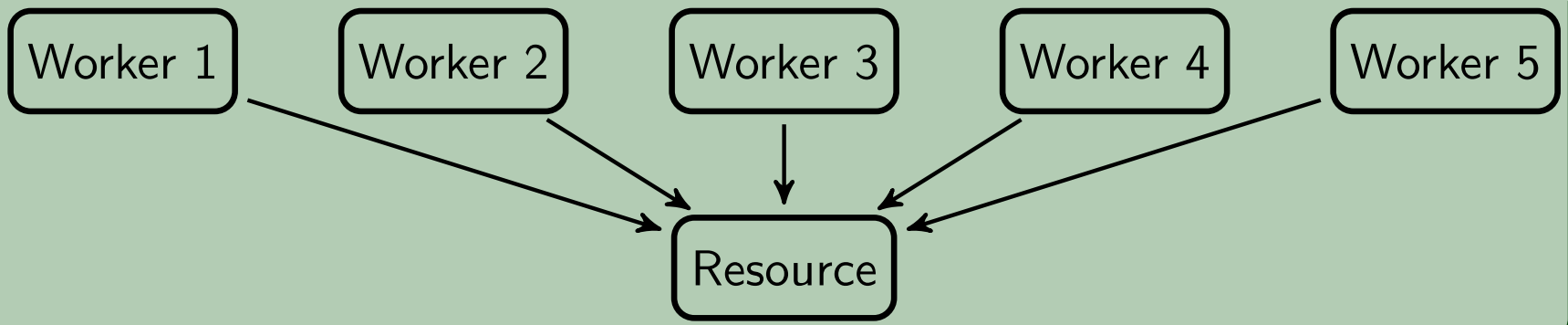
In this course, we will focus on parallelism and synchronization:

Parallelism



Again, concurrency **is not the same as** parallelism! They're easy to confuse, but the distinction is important.

Synchronization



Our First Parallel Program (not real code)

Parallel Psuedocode

```
1  int sum(int[] arr, int lo, int hi) {
2     result = 0;
3     for (i = lo; i < hi; i++) {
4        result += arr[i];
5     }
6     return result;
7  }
8  int sum(int[] arr) {
9     pieces = new int[4];
10    parallel (i = 0; i < 4; i++) {
11       lo = i * arr.length / 4
12       hi = (i + 1) * arr.length / 4
13       pieces[i] = sum(arr, lo, hi);
14    }
15    return sum(pieces, 0, 4);
16 }
```

If we're super lucky, this idea would get us a 4x speed-up.

Sequential Programming Model

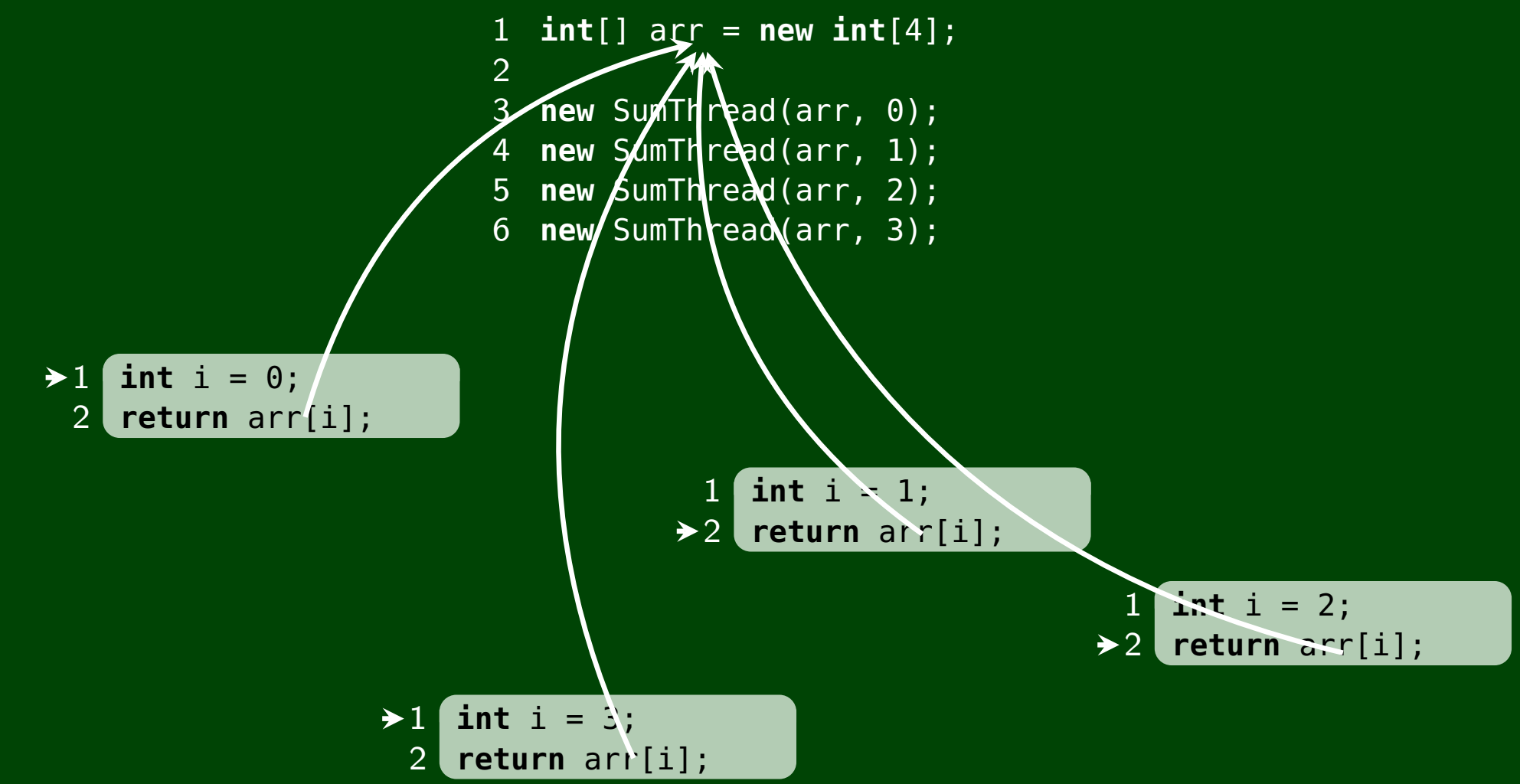
- One **program counter** (instruction executing)
- One **call stack** (all variables **not made** with `new`)
- `Objects` in the heap (all variables **made** with `new`)

Shared Memory with Threads

A set of threads, each of which has:

- One **program counter** (instruction executing)
- One **call stack** (all variables **not made** with `new`)

`Objects` in the heap may be shared by passing a common pointer to multiple threads (and then editing them).


1 int[] arr = new int[4];
2
3 new SumThread(arr, 0);
4 new SumThread(arr, 1);
5 new SumThread(arr, 2);
6 new SumThread(arr, 3);
1 int i = 0;
2 return arr[i];
1 int i = 1;
2 return arr[i];
1 int i = 2;
2 return arr[i];
1 int i = 3;
2 return arr[i];

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 (e.g. Cooks working in separate kitchens, mail around ingredients)
- **Dataflow**: Programmers write programs in terms of a graph. 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"

Parallel Primitives

- We need to be able to create multiple "things" running at once:

  We call these threads. To get a new thread, we **fork** an old one.

- We need threads to share memory:

  Pass two threads the same array, object, etc. (Remember, Java passes Object **references**.)

- We need threads to coordinate:

  For now, we can combine them together using **join**. Later, we'll learn about a synchronization primitive.

Unsurprisingly, the framework we'll be using is called `ForkJoin`:

`ForkJoin` API

`worker.fork()`	Sets the **worker** in action (in parallel)
`worker.join()`	Pauses the execution of **this** thread until **worker** is done executing. Returns the result from **worker**

Sum Array

Sum the numbers in an array.

Sequential Solution

```
1  public static long sum(long[] arr, int lo, int hi) {
2     long result = 0;
3     for (int i = lo; i < hi; i++) {
4        result += arr[i];
5     }
6     return result;
7  }
8
9  public static long sum(long[] arr) {
10    return sum(arr, 0, arr.length);
11 }
```

We need to replace the sequential for loop with a parallel for loop in some way.

Using RecursiveTask

In the ForkJoin library, our threads should subclass RecursiveTask:

Using RecursiveTask

```
1  static class SumTask extends RecursiveTask<Long> {
2     long[] arr; int lo, hi;
3
4     public SumTask(long[] arr, int lo, int hi) {
5        this.arr = arr; this.lo = lo; this.hi = hi;
6     }
7     protected Long compute() {
8        long result = 0;
9        for (int i = lo; i < hi; i++) {
10          result += arr[i];
11       }
12       return result;
13    }
14 }
15
16 public static long sum(long[] arr) {
17    SumTask task = new SumTask(arr, 0, arr.length);
18    return task.compute();
19 }
```

But we're just calling compute sequentially.

So, we **don't even spawn ONE new thread**.

To set the first thread running, we **invoke** it using a thread pool:

Using The Thread Pool

```
1  public static final ForkJoinPool POOL = new ForkJoinPool();
2
3  static class SumTask extends RecursiveTask<Long> {
4     long[] arr; int lo, hi;
5
6     public SumTask(long[] arr, int lo, int hi) {
7        this.arr = arr; this.lo = lo; this.hi = hi;
8     }
9     protected Long compute() {
10       long result = 0;
11       for (int i = lo; i < hi; i++) {
12          result += arr[i];
13       }
14       return result;
15    }
16 }
17
18 public static long sum(long[] arr) {
19    SumTask task = new SumTask(arr, 0, arr.length);
20    return POOL.invoke(task);
21 }
```

All of the threads `fork` creates will come from the `ForkJoinPool`.

From now on, we'll focus on the `compute` method, because that's where the majority of the work goes in.

Forking!

```
1  protected Long compute() {
2     if (hi < arr.length) {
3        int nextLo = lo + arr.length/4;
4        int nextHi = hi + arr.length/4;
5        SumTask task = new SumTask(arr, nextLo, nextHi);
6        task.fork();
7     }
8
9     long result = 0;
10    for (int i = lo; i < hi; i++) {
11       result += arr[i];
12    }
13
14    return result; /* BROKEN: We didn't use task's result! */
15 }
```

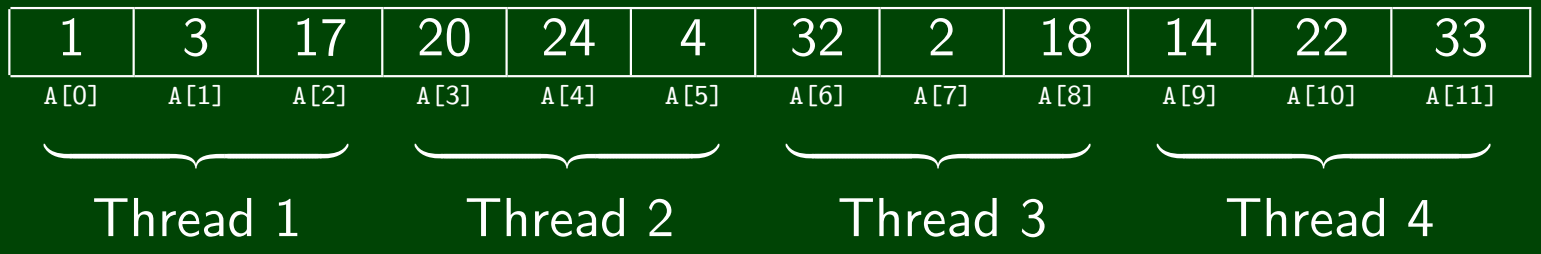
Here, we break the work into four pieces (each thread creates the next):

Joining!

```
1  protected Long compute() {
2     long result = 0;
3     SumTask task = null;
4     if (hi < arr.length) {
5        int nextLo = lo + arr.length/4;
6        int nextHi = hi + arr.length/4;
7        task = new SumTask(arr, nextLo, nextHi);
8        task.fork();
9     }
10
11    for (int i = lo; i < hi; i++) {
12       result += arr[i];
13    }
14
15    // If we call join earlier, then it won't be parallel!
16    if (task != null) {
17       result += task.join();
18    }
19
20    return result;
21 }
```

The idea is to start the next thread, do our own work, and sync up with the one after us to combine results.

Shared Memory?



What Memory is Shared?

In our `sum` example, the same array is passed to all the threads! The `lo` and `hi` tell the threads what work to do, and they don't step over each other. Fundamentally, if we couldn't share the array, we wouldn't be able to do the parallelism.

So, are we done? Is there anything wrong with this algorithm?

Generalizing 4 to n

Our algorithm so far is **bad**. Here's why:

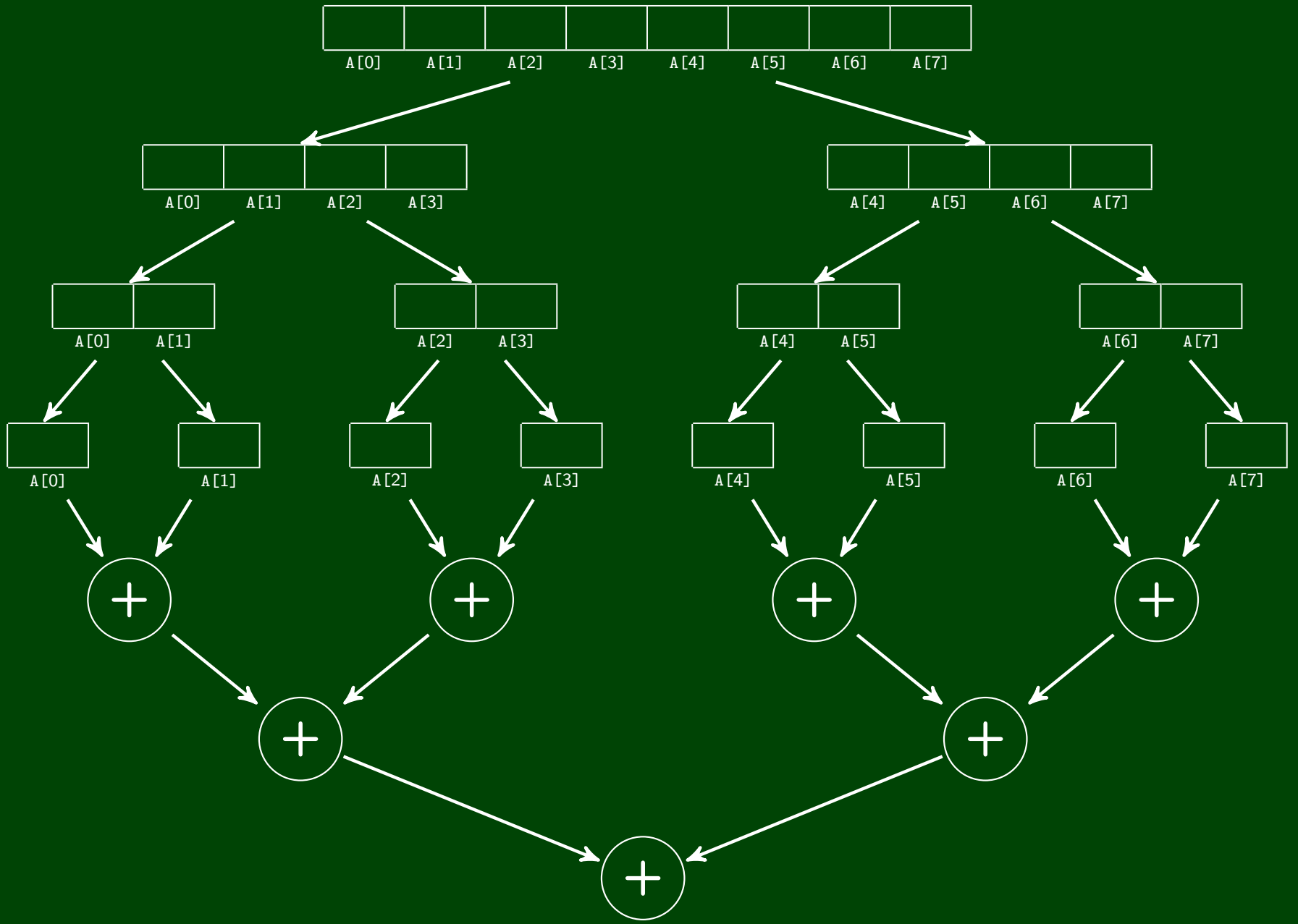
1. We want our code to be reusable and efficient across platforms:
   - "Forward-portable" as core count grows
   - We should (at the very least...) parametrize by the number of threads

2. We only want to use **available** processors:
   - There are probably other programs?
   - If we attempt to use more processors than there are, we actually end up with a **slower** algorithm!
   - If we have 3 processors available and 3 threads would take x time, then four would take $1.5x$ time...

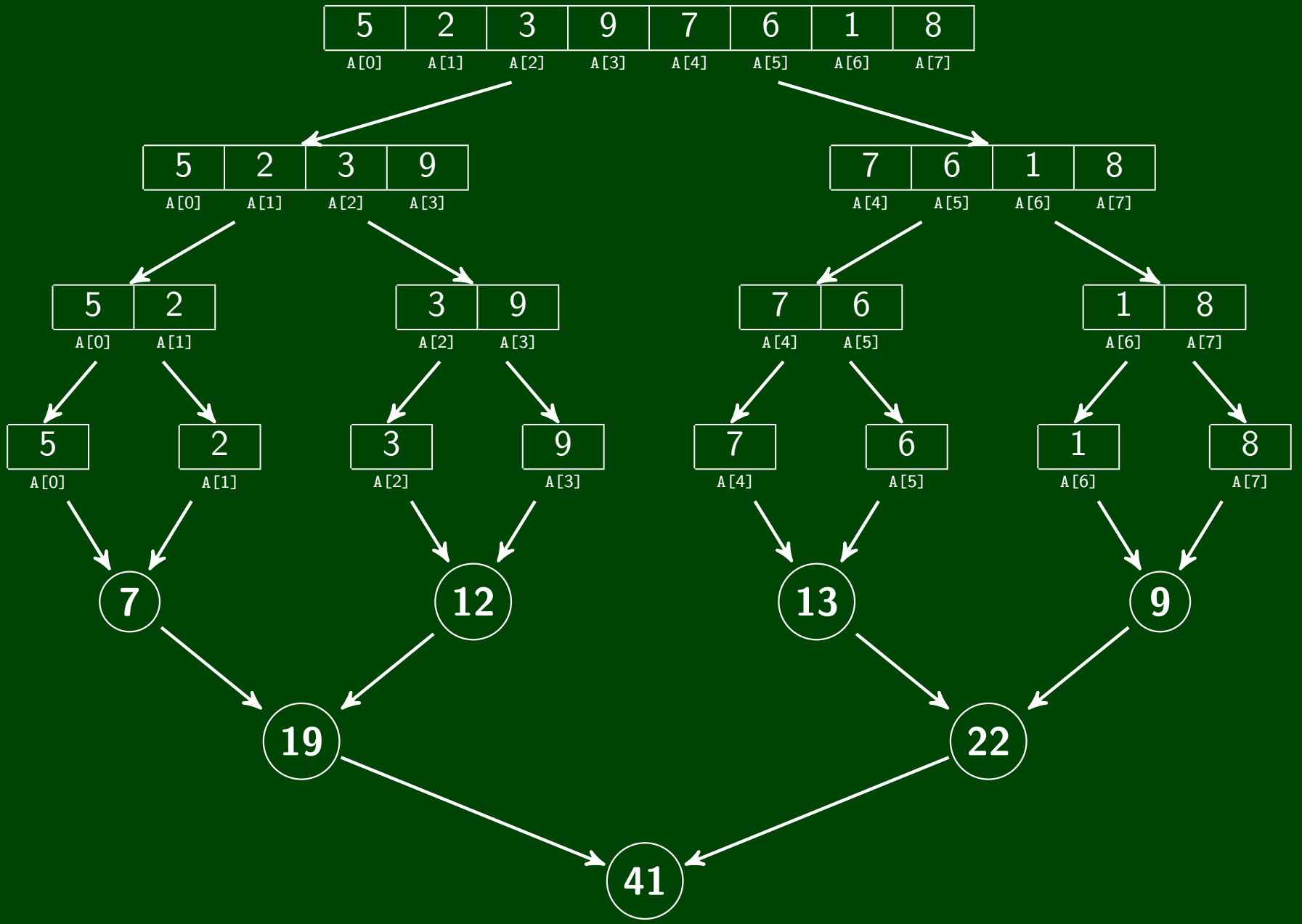
3. In general, different sub-problems might takes different amounts of time. Consider our next example: "How many primes in this range?"

So, how do we fix this?

Break Up The Work More!



Break Up The Work More!



Prime Sieve

```
1  protected Integer compute() {
2     PrimeFinderTask task = null;
3     if (lo < hi - 1) {
4        task = new PrimeFinderTask(lo + 1, hi);
5        task.fork();
6     }
7
8     int result = 0;
9     result += isPrime(lo) ? 1 : 0;
10
11    if (task != null) {
12       result += task.join();
13    }
14    return result;
15 }
```

This is is going to be very slow (and run out of stack space very quickly). Divide and conquer can do better.

Prime Sieve

```
1  protected Integer compute() {
2     if (lo < hi - 1) {
3        int mid = lo + (hi - lo) / 2;
4        PrimeFinderTask left = new PrimeFinderTask(lo, mid);
5        PrimeFinderTask right = new PrimeFinderTask(mid, hi);
6
7        left.fork();
8        right.fork();
9
10       return left.join() + right.join();
11    }
12
13    return isPrime(lo) ? 1 : 0;
14 }
```

Unfortunately, this is going to be really slow in practice. The issue is that the **overhead** of creating a thread to do a single prime task will overtake the benefit of the parallelism at some point. The solution is to use a cutoff after which we switch to the sequential solution.

Instead of cutting the work into fourths, cut it into n pieces (where n is the size of the array). Let's switch to a (slightly) more realistic example:

Prime Sieve

```
1  protected Integer compute() {
2     if (hi – lo <= CUTOFF) {
3        return sequentialNumberOfPrimes(lo, hi);
4     }
5
6     int mid = lo + (hi – lo) / 2;
7     PrimeFinderTask left = new PrimeFinderTask(lo, mid);
8     PrimeFinderTask right = new PrimeFinderTask(mid, hi);
9
10    left.fork();
11    right.fork();
12
13    return left.join() + right.join();
14 }
```

We're almost there. One last problem: consider what work each forking thread does:

Fork left, Fork right, Wait for left, Wait for right

Insight: It doesn't do any work itself. This is a waste.

Prime Sieve

```
1  protected Integer compute() {
2     if (hi – lo <= CUTOFF) {
3        return sequentialNumberOfPrimes(lo, hi);
4     }
5
6     int mid = lo + (hi – lo) / 2;
7     PrimeFinderTask left = new PrimeFinderTask(lo, mid);
8     PrimeFinderTask right = new PrimeFinderTask(mid, hi);
9
10    left.fork();
11    int result = right.compute();
12
13    return left.join() + result;
14 }
```

Finally, this is reasonable parallel code. And we can even see a great speed-up if we run both sequential and parallel.

Sequential Threshold

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

Library Needs To "Warm Up"

You may see slow results before the Java virtual machine reoptimizes the library internals.

Use A Machine With More Processors

For p3, we'll be using Google Compute Engine! So, you can see parallelism at its best.

Beware The Memory-Hierarchy

We won't focus on this (in this course), but it's often crucial for parallel performance.