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.)

Challenges with Parallelism
- **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)

Concurrency? Parallelism? Synchronization?

**Definition (Concurrency*)**
Programing 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.
Concurrent Programming: Parallelism and Synchronization

**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**

- Work
  - Worker 1
  - Worker 2
  - Worker 3
  - Worker 4
  - Worker 5

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

**Synchronization**

- Resource
  - Worker 1
  - Worker 2
  - Worker 3
  - Worker 4
  - Worker 5

Our First Parallel Program (not real code)

**Parallel Psuedocode**

```java
def sum(int[] arr, int lo, int hi) {
    result = 0;
    for (i = lo; i < hi; i++) {
        result += arr[i];
    }
    return result;
}
```

```java
int i = 0;
return arr[i];
```

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

A Model: Shared Memory with Threads

**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).
From Sequential to Parallel

**Sum Array**

Sum the numbers in an array.

**Sequential Solution**

```java
public static long sum(long[] arr, int lo, int hi) {
    long result = 0;
    for (int i = lo; i < hi; i++) {
        result += arr[i];
    }
    return result;
}
```

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

**Using ForkJoinPool**

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

```java
public static final ForkJoinPool POOL = new ForkJoinPool();
```

**Using The Thread Pool**

```java
static class SunTask extends RecursiveTask<Long> {
    long[] arr; int lo, hi;
    public SunTask(long[] arr, int lo, int hi) {
        this.arr = arr; this.lo = lo; this.hi = hi;
    }
    protected Long compute() {
        long result = 0;
        for (int i = lo; i < hi; i++) {
            result += arr[i];
        }
        return result;
    }
    public static long sum(long[] arr) {
        SunTask task = new SunTask(arr, 0, arr.length);
        return POOL.invoke(task);
    }
}
```

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

**What Do We Need To Write Parallel Programs?**

**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

**Using RecursiveTask**

In the ForkJoin library, our threads should subclass RecursiveTask:

```java
static class SunTask extends RecursiveTask<Long> {
    long[] arr; int lo, hi;
    public SunTask(long[] arr, int lo, int hi) {
        this.arr = arr; this.lo = lo; this.hi = hi;
    }
    protected Long compute() {
        long result = 0;
        for (int i = lo; i < hi; i++) {
            result += arr[i];
        }
        return result;
    }
    public static long sum(long[] arr) {
        SunTask task = new SunTask(arr, 0, arr.length);
        return task.compute();
    }
}
```

But we’re just calling compute sequentially.
So, we don’t even spawn ONE new thread.

**Using fork**

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

```java
protected Long compute() {
    if (hi < arr.length) {
        int nextLo = lo + arr.length/4;
        int nextHi = hi + arr.length/4;
        SunTask task = new SunTask(arr, nextLo, nextHi);
        Task fork = task.fork();
    }
    long result = 0;
    for (int i = lo; i < hi; i++) {
        result += arr[i];
    }
    return result; /* BROKEN: We didn’t use task’s result! */
}
```

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

Joining:
```java
protected Long compute() {
    long result = 0;
    SumTask task = null;
    if (hi < arr.length) {
        int nextLo = lo + arr.length/4;
        int nextHi = hi + arr.length/4;
        task = new SumTask(arr, nextLo, nextHi);
        task.fork();
    }
    for (int i = lo; i < hi; i++) {
        result += arr[i];
    }
    result += task.join();
    return result;
}
```

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

Generalizing 4 to n

Our algorithm so far is bad. Here’s why:

- 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

- 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...

- In general, different sub-problems might take 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!

Parallel Prime Sieve

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

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

\[
\text{Fork left, Fork right, Wait for left, Wait for right}
\]

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

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