CSE 332 Spring 2026
Lecture 19: ForkJoin

Nathan Brunelle

http://www.cs.uw.edu/332

Parallelism Vs. Concurrency (with Potatoes)

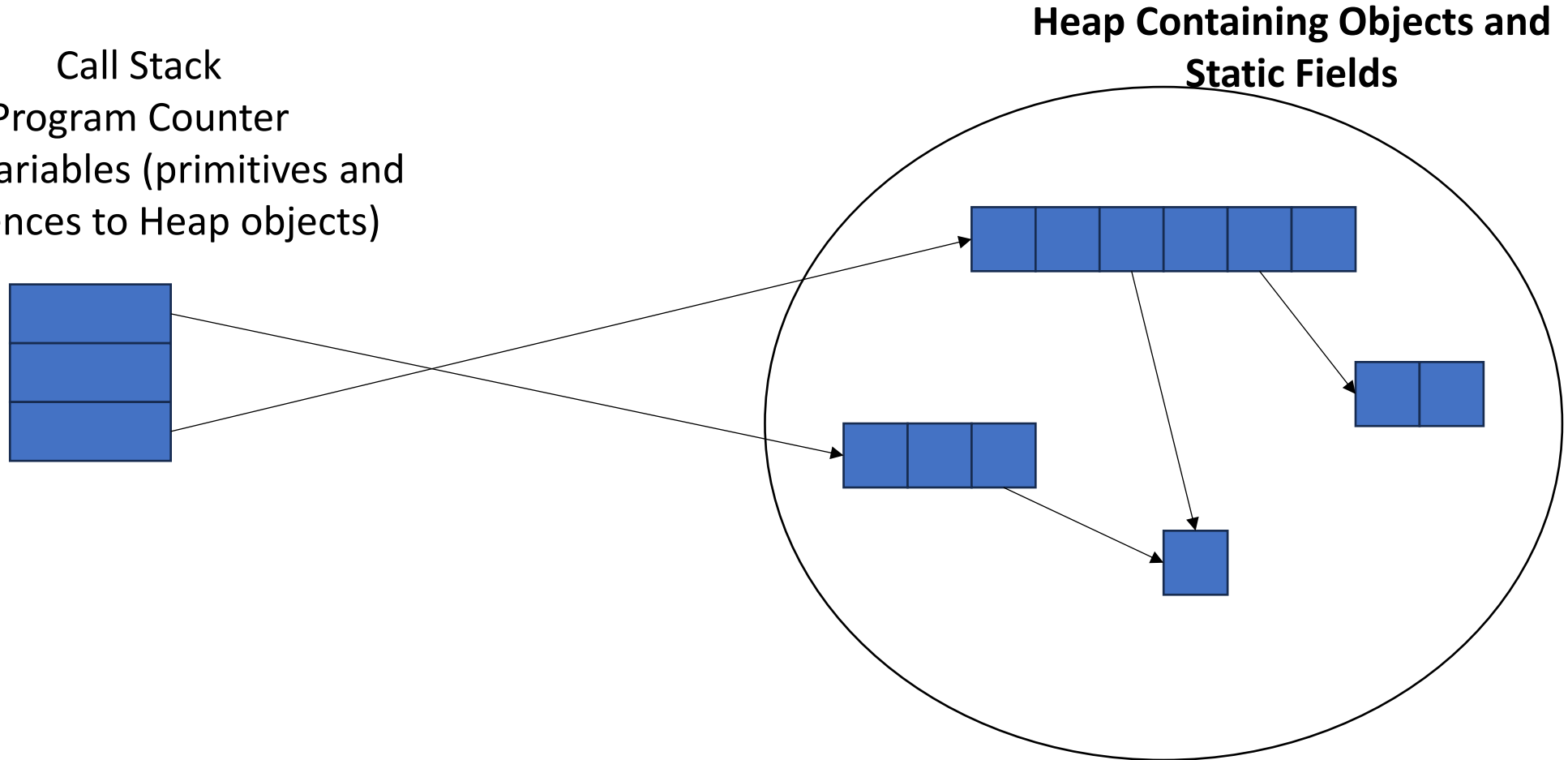
- Sequential:
  - The task is completed by just **one processor** doing one thing at a time
  - There is one cook who peels all the potatoes
- Parallelism:
  - One task being completed by **many threads/processors**
  - Recruit several cooks to peel a lot of potatoes faster
- Concurrency:
  - Parallel tasks using a **shared resource**
  - Several cooks are making their own recipes, but there is only 1 oven

New Story of Code Execution

- Old Story:
  - One **program counter** (current line of code)
  - One **call stack** (with each stack frame holding local variables)
  - **Objects in the heap** created by memory allocation (i.e., **new** ___)
    - (nothing to do with data structure called a heap)
  - Operating System **time slicing** where programs may be paused by the OS at any time, with their execution being resumed later
- New Story:
  - Collection of threads each with its own:
    - Program Counter
    - Call Stack
    - Local Variables
    - References to objects in the heap
  - **One shared heap**
  - **Time slicing**

Old Stroy

Call Stack
Program Counter
Local Variables (primitives and references to Heap objects)



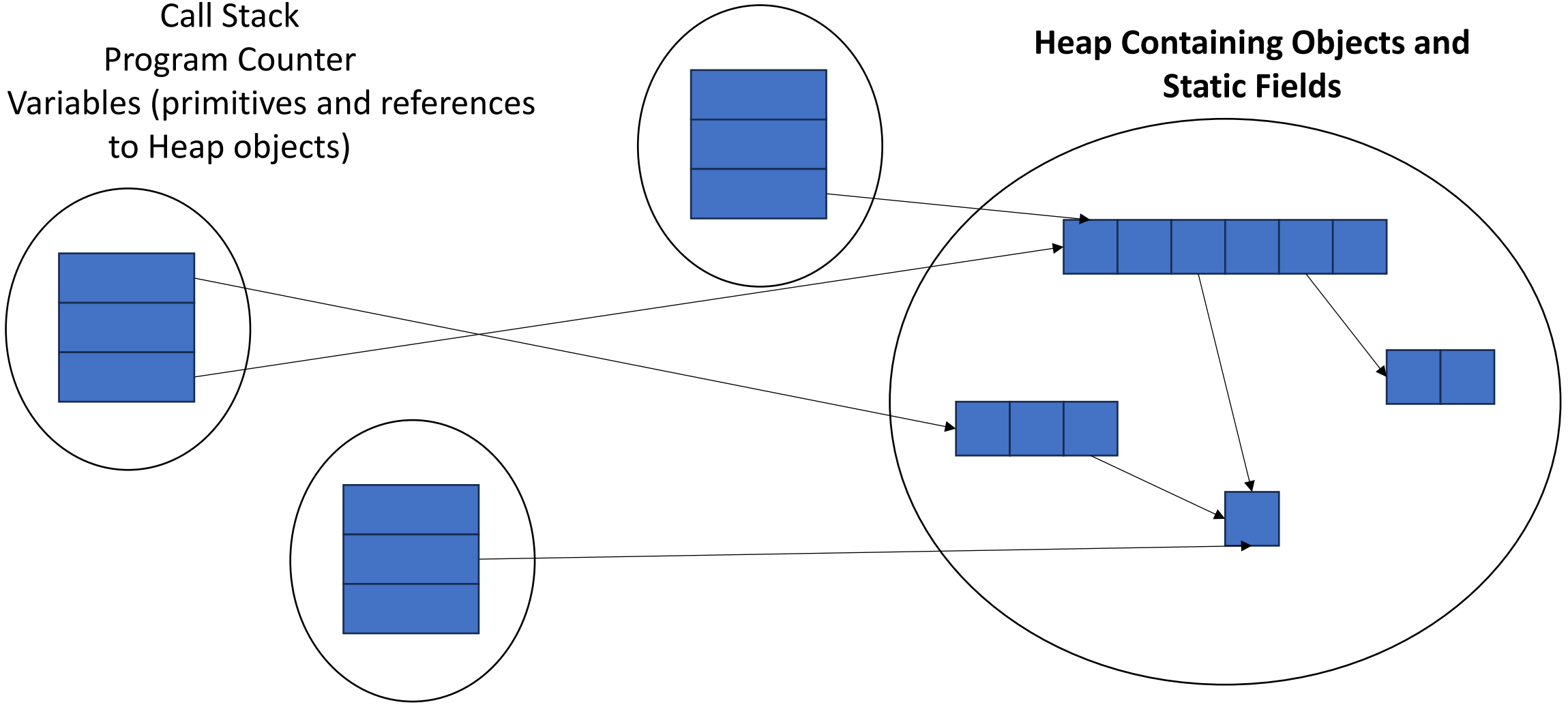
New Story

Threads, each with its own unshared:

Call Stack

Program Counter

Local Variables (primitives and references to Heap objects)



Running Example: Summing an Array

- **Goal**: Find the sum of an array
- **Idea**: Split array into 4 pieces, sum those pieces in parallel, and then sum the results.

Input: array (arr) of integers

1. **In parallel**: split array into 4 equal pieces, sum up those four pieces.
   1. res[i] = sum arr[i*len/4, (i+1)*len/4]
2. **Return** res[0] + res[1] + res[2] + res[3]

- **Needs from Programming language**
  - Way to **create threads**, and run them in parallel
  - Way to access shared memory (refs to common objects)
  - Way to **synchronize** threads (**wait** until finished)
    - Cannot sum res[0] + … + res[3] until each thread has finished!

Accomplishing this in Java (**Java.lang.Thread**)

1. Create class C extending **java.lang.Thread**
    1. C must have a **run** method (acts as **main**)
2. Call C's **start** method to
    1. create a new thread (i.e. parallel task, not Thread object in java)
    2. execute **run** in that separate, parallel thread

- Calling "**run**" directly causes the program to execute "**run**" sequentially

First Attempt (part 1, Defining Thread Object)

```
class SumThread extends java.lang.Thread {
        int lo;    int hi;    int[] arr;  int ans = 0;
        SumThread(int[] a, int l, int h) {
                lo=l; hi=h; arr=a;
        }
        public void run() { //override, must have this signature
                                // no arguments and no return allowed
                                // must use the object's fields for input/output
                for(int i=lo; i < hi; i++)
                        ans += arr[i];
        }
```

First Attempt (part 2, Creating Thread Objects)

```
static int parallelSum(int[] arr){ // this method could be anywhere
        int len = arr.length;
        int ans = 0;
        SumThread[] threads = new SumThread[4];
        for(int i=0; i < 4; i++) // create threads
                threads[i] = new SumThread(arr, i*len/4, (i+1)*len/4);
        // more stuff to follow
}
```

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

First Attempt (part 3, Running Thread Objects)

```
static int parallelSum(int[] arr){ // this method could be anywhere
        int len = arr.length;
        int ans = 0;
        SumThread[] threads = new SumThread[4];
        for(int i=0; i < 4; i++){ // create threads, do parallel computations
                threads[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
                threads[i].start(); // start not run
        }
        for(int i=0; i < 4; i++) // combine results
                ans += threads[i].ans;
        return ans;
}
```

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

First Attempt (part 4, Synchronizing)

```
static int parallelSum(int[] arr){ // this method could be anywhere
        int len = arr.length;
        int ans = 0;
        SumThread[] threads = new SumThread[4];
        for(int i=0; i < 4; i++){ // do parallel computations
                threads[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
                threads[i].start(); // start not run
        }
        for(int i=0; i < 4; i++){ // combine results
                threads[i].join(); // wait for thread to finish!
                ans += threads[i].ans;
        }
        return ans;
}
```

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

Join

- Causes program to pause until the other thread completes its **run** method
- Avoids a **race condition**
  - Without join the other thread's **ans** field may not have its final answer yet

Recap so far

- Way to **create threads**, and run them in parallel
  - C extends java.lang.Thread
  - C.start()
- Way to **wait for threads to finish**
  - C.join()

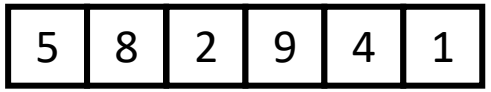
More Threads?

- **Issue:** our parallel algorithm is not that parallel! Each thread takes $O\left(\frac{n}{4}\right) = O(n)$ time.
- **Idea 1:** make # of threads (# of array chunks) a parameter. Each thread runs in $O\left(\frac{n}{k}\right)$ time, combining takes $O(k)$ time.
  - **Pros**: Different machines have different # processors. More efficient/reusable across machines
  - **Cons:** OS ultimately in charge of how processors get used, and perhaps not all subproblems take same amount of time
    - **Runtime barrier**: cannot do better than $O(\sqrt{n})$

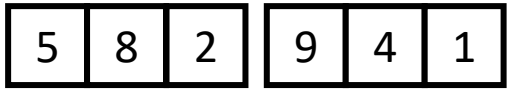
A Better Solution: Divide and Conquer!

- **Idea**: Each thread checks its input size. If smaller than ℓ, sum sequentially. Otherwise, split the problem in half across two separate threads.
  - ℓ is the "sequential cutoff" (typical range: 500 to 5000)
  - Creating threads takes some time, at some point it's more efficient to do things sequentially!

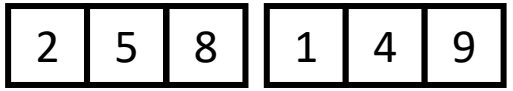
Merge Sort



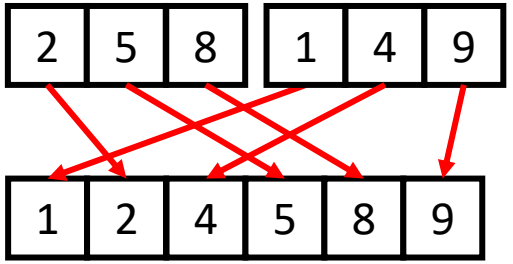
- **Base Case:**
  - If the list is of length 1 or 0, it's already sorted, so just return it



- **Divide:**
  - Split the list into two "sublists" of (roughly) equal length

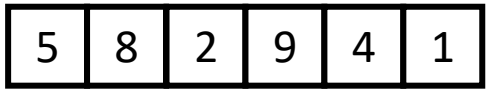


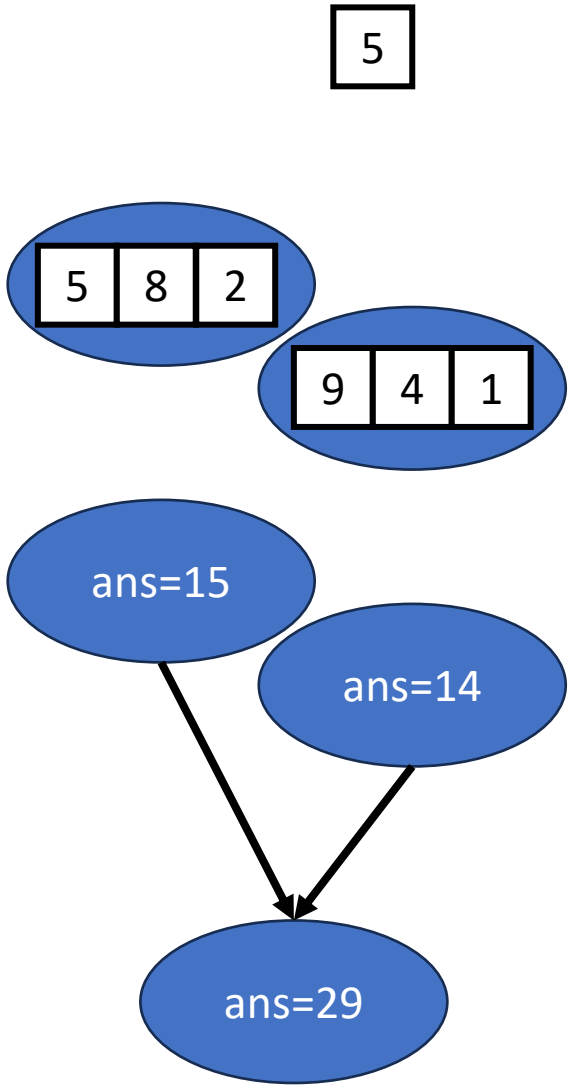
- **Conquer:**
  - Sort both lists recursively



- **Combine:**
  - **Merge** sorted sublists into one sorted list

Parallel Sum





- **Base Case**:
  - If the list's length is smaller than the Sequential Cutoff, find the sum sequentially
- **Divide**:
  - Split the list into two "sublists" of (roughly) equal length, create a Thread to sum each sublist.
- **Conquer**:
  - Call **start()** for each thread
- **Combine**:
  - Sum together the answers from each thread

Parallel Divide and Conquer Pseudocode

RecursiveSum(arr)

1. **If** len(arr) < ℓ: return sum of elements in arr
2. **Else:**
    1. Divide arr in half into arr1 and arr2
    2. Conquer in parallel: call **RecursiveSum**(arr1) and **RecursiveSum**(arr2) in new threads
3. Wait for the recursive calls/threads to finish
4. Combine the sum of the two recursive calls (and return)

Divide and Conquer with Java Threads

```
class SumThread extends java.lang.Thread {
        public void run(){ // override
                if(hi – lo < SEQUENTIAL_CUTOFF) // “base case”
                        for(int i=lo; i < hi; i++) ans += arr[i];
                else {
                        SumThread left = new SumThread(arr,lo,(hi+lo)/2); // divide
                        SumThread right= new SumThread(arr,(hi+lo)/2,hi); // divide
                        left.start(); right.start(); // conquer
                        left.join(); right.join(); // wait
                        ans = left.ans + right.ans; // combine
                }
        }
}
int sum(int[] arr){ // just make one thread!
        SumThread t = new SumThread(arr,0,arr.length);
        t.run();
        return t.ans; }
```

Small optimization

- Instead of calling two separate threads for the two subproblems, create one parallel thread (using **start**) and one sequential thread (using **run**)

Divide and Conquer with Threads (optimized)

```
class SumThread extends java.lang.Thread {
        public void run(){ // override
                if(hi – lo < SEQUENTIAL_CUTOFF) // “base case”
                        for(int i=lo; i < hi; i++) ans += arr[i];
                else {
                        SumThread left = new SumThread(arr,lo,(hi+lo)/2); // divide
                        SumThread right= new SumThread(arr,(hi+lo)/2,hi); // divide
                        left.start(); // conquer in parallel in new thread
                        right.run(); // conquer in this thread
                        left.join(); // don’t move this up a line – why?
                        //right.join();
                        ans = left.ans + right.ans; // combine
                }
        }
}
int sum(int[] arr){ // just make one thread!
        SumThread t = new SumThread(arr,0,arr.length);
        t.run();
        return t.ans; }
```

ForkJoin Framework

- This strategy is common enough that Java (and C++, and C#, and…) provides a library to do it for you!

What you would do in Threads	What to instead in ForkJoin
Subclass **Thread**	Subclass **RecursiveTask<V>** (to return a value of type V) or **RecursiveAction** (for void return)
Override **run**	Override **compute**
Store the answer in a field	Return a V from compute
Call **start**	Call **fork**
join synchronizes only	**join** synchronizes and returns the answer
Call **run** to execute sequentially	Call **compute** to execute sequentially
Have a topmost thread and call **run**	Create a pool and call **invoke**

Divide and Conquer with ForkJoin

```
class SumTask extends RecursiveTask<Integer> {
        int lo; int hi; int[] arr; // fields to know what to do
        SumTask(int[] a, int l, int h) { … } // constructor
        protected Integer compute(){ // return answer
                if(hi – lo < SEQUENTIAL_CUTOFF) {  // base case
                        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); // divide
                        SumTask right= new SumTask(arr,(hi+lo)/2,hi); // divide
                        left.fork(); // conquer in parallel
                        int rightAns = right.compute(); // conquer in this thread
                        int leftAns = left.join(); // wait
                        return leftAns + rightAns; // combine
                }
        }
}
```

Divide and Conquer with ForkJoin (continued)

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

Find Max with ForkJoin

```
class MaxTask 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) {  // base case
                        int ans = Integer.MIN_VALUE; // local var, not a field
                        for(int i=lo; i < hi; i++) {
                                ans = Math.max(ans, arr[i]);}
                        return ans;
                else {
                        MaxTask left = new MaxTask(arr,lo,(hi+lo)/2); // divide
                        MaxTask right= new MaxTask(arr,(hi+lo)/2,hi); // divide
                        left.fork(); // fork a thread and calls compute (conquer)
                        int rightAns = right.compute(); //call compute directly (conquer)
                        int leftAns = left.join(); // get result from left
                        return Math.max(rightAns, leftAns); // combine
                }
        }
}
```

Other Problems that can be solved similarly

- Element Search
  - Is the value 17 in the array?
- Counting items with a certain property
  - How many elements of the array are divisible by 5?
- Checking if the array is sorted
- Find the smallest rectangle that covers all points in the array
- Find the first thing that satisfies a property
  - What is the leftmost item that is divisible by 20?

Reductions/Folds

- All examples of a category of computation called a reduction
  - We “reduce” all elements in an array to a single item
  - Requires operation done among elements is **associative**
    - $(x + y) + z = x + (y + z)$
    - min(min(x,y), z) = min(x, min(y, z))
  - The “single item” can itself be complex
    - E.g. create a histogram of results from an array of trials

Map

- **New task**: Apply a function (map) to each element of an array
- Examples:
  - Vector addition:
    - sum[i] = arr1[i] + arr2[i]
  - Function application:
    - out[i] = f(arr[i]);
- **Observation**: maps also parallelizable in the same way!
  - No need for combination step

Map with ForkJoin

```
class AddVecs extends RecursiveAction {
        int lo; int hi; int[] arr; // fields to know what to do
        AddVecs(int[] a, int[] b, int[] sum, int l, int h) { … }
        protected void compute(){// return answer
                if(hi – lo < SEQUENTIAL_CUTOFF) {  // base case
                        for(int i=lo; i < hi; i++) {
                                sum[i] = a[i] + b[i];}
                else {
                        AddTask left = new AddVecs(a,b,sum,lo,(hi+lo)/2); // divide
                        AddTask right= new AddVecs(a,b,sum,(hi+lo)/2,hi); // divide
                        left.fork(); // fork a thread and calls compute (conquer)
                        right.compute(); //call compute directly (conquer)
                        left.join(); // wait for thread to finish
                        return; // combine
                }
        }
}
```

Map with ForkJoin (continued)

```
static final ForkJoinPool POOL = new ForkJoinPool();
int[] add(int[] a, int[] b){
        int[] ans = new int[a.length];
        AddVecs task = new AddVecs(a, b, ans, 0, a.length)
        POOL.invoke(task);
        return ans;
}
```

Maps and Reductions

- “Workhorse” constructs in parallel programming
- Many problems can be written in terms of maps and reductions
- With practice, writing them will become second nature
  - Like how over time for loops and if statements have gotten easier

Section

- Working with examples of ForkJoin
- Make sure to bring your laptops!
  - And charge it!