

CSE 332: Data Structures and Parallelism

Spring 2022

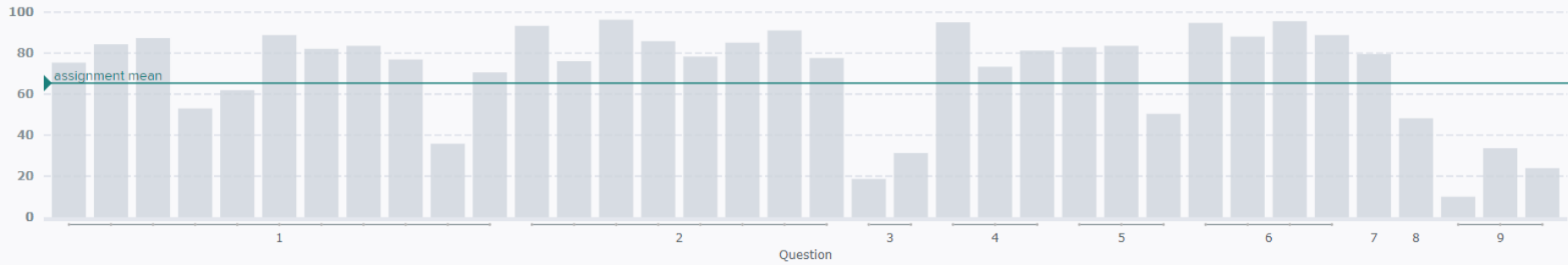
Richard Anderson

Lecture 20: Analysis of Fork-Join Programs

Announcements

- Read parallel computing notes by Dan Grossman 2.1-4.3
- Bring laptop to section this week with IntelliJ set up
 - Work on fork-join parallelism for most of section

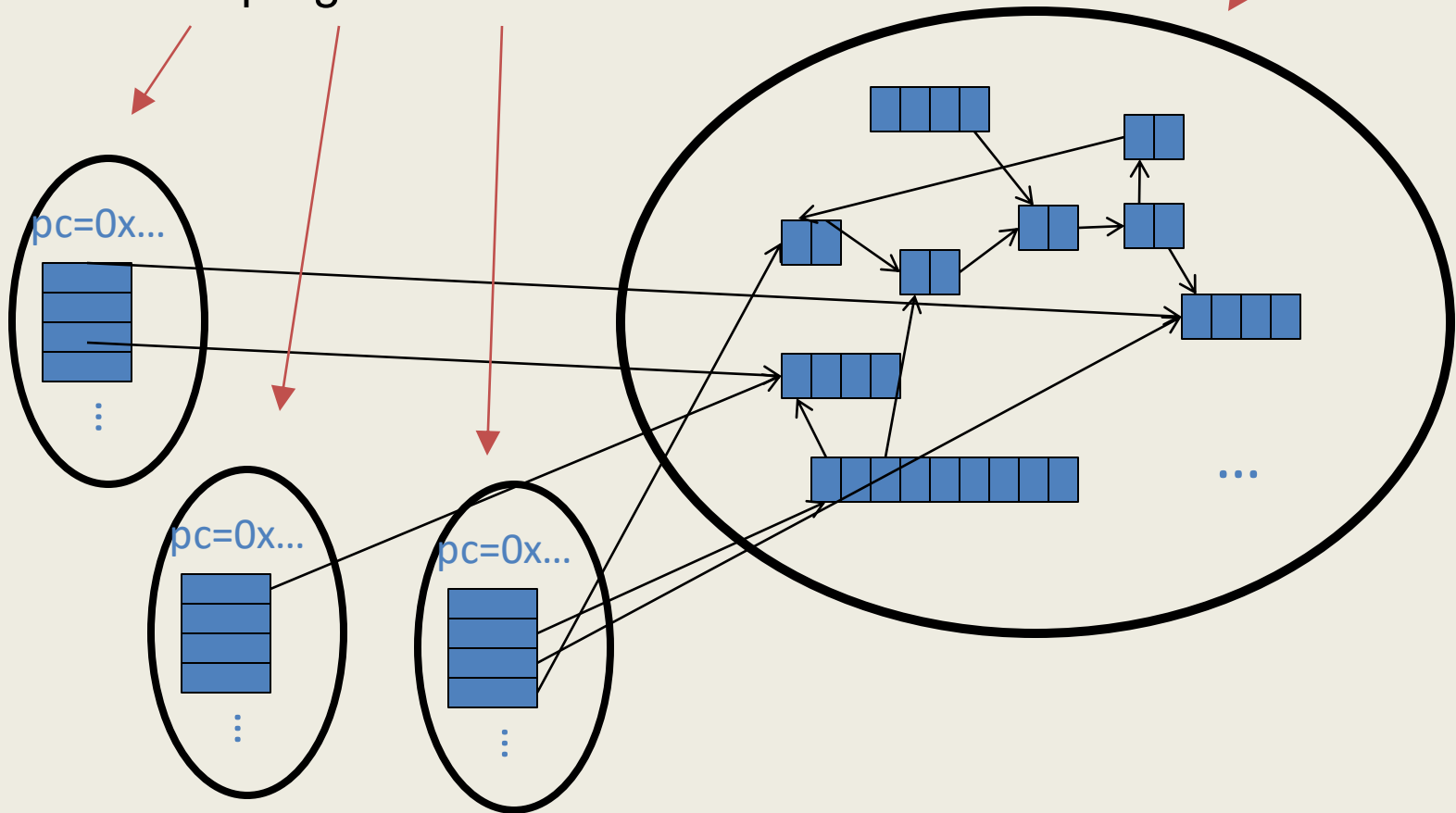
Midterm Problems



Shared Memory with Threads

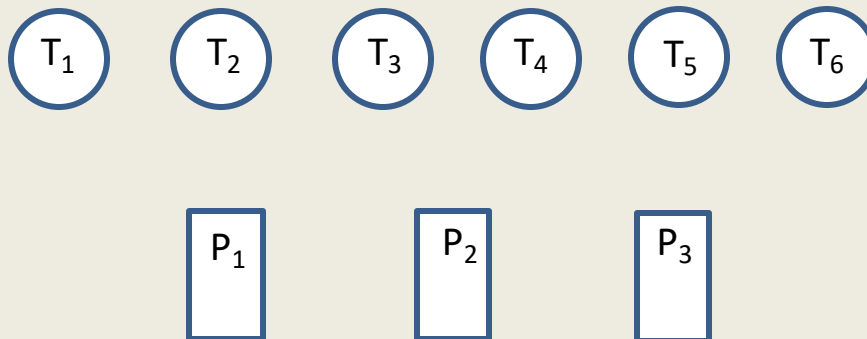
Threads, each with own *unshared* call stack and “program counter”

Heap for all objects and static fields, *shared* by all threads



Threads and Processors

- Simple model
 - Threads are either running or idle
 - Processors select idle threads and execute them for “a while”
- Scheduling of threads is outside of the scope of this course
 - Many different approaches
 - Programmer has limited control on scheduling



Fork-Join Parallelism

1. Define thread

- Java: define subclass of `java.lang.Thread`,
- Override `run` implement operation of the thread

2. Fork: instantiate a thread and start executing

- Java: create thread object, call `start()`

3. Join: wait for thread to terminate

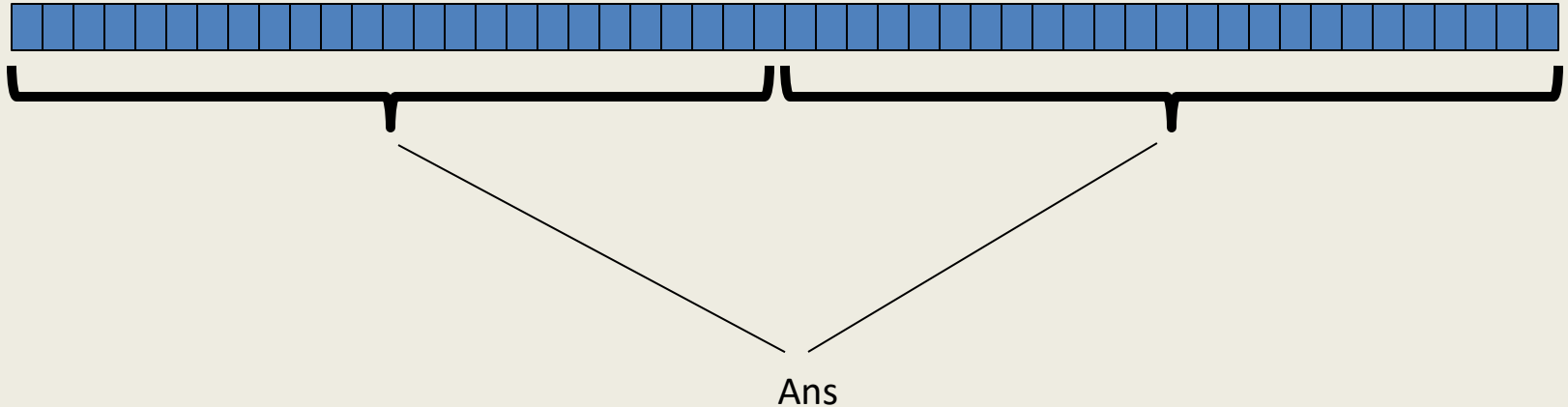
- Java: call `join()` method, which returns when thread finishes

Above uses basic thread library build into Java

Later we'll introduce a better `ForkJoin Java library` designed for parallel programming

Sum with Threads

For starters: have two threads simultaneously sum half of the array



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

Part 1: define thread class

```
class SumThread extends java.lang.Thread {  
  
    int lo; // fields, passed to 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() {  
        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

Part 2: sum routine

```
int sum(int[] arr) {
    int len = arr.length;

    SumThread ts1 = new SumThread(arr, 0, len/2);
    SumThread ts2 = new SumThread(arr, len/2, len);

    ts1.start();
    ts2.start();

    ts1.join();
    ts2.join();

    return ts1.ans + ts2.ans;
}
```

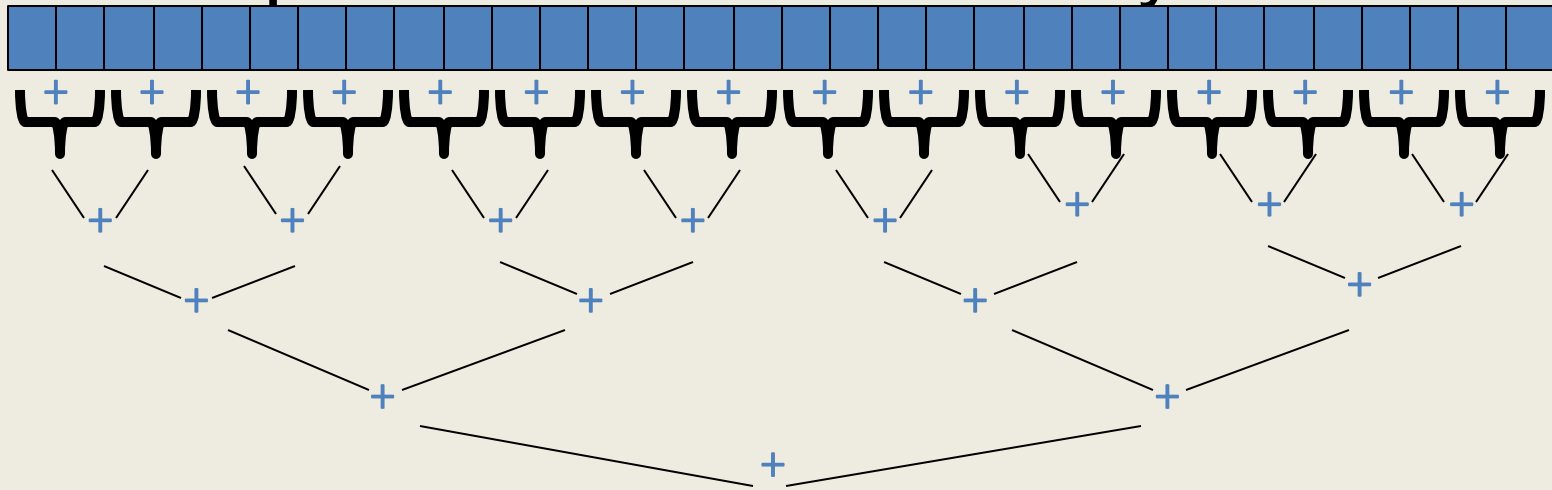
Parameterizing by number of threads

```
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;
}
```

Recall: Parallel Sum

- Sum up N numbers in an array



- Let's implement this with threads...

Code looks something like this (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;
}
```

Thread: sum range [0,10)

Thread: sum range [0,5)

Thread: sum range [0,2)

Thread: sum range [0,1) (return arr[0])

Thread: sum range [1,2) (return arr[1])

add results from two helper threads

Thread: sum range [2,5)

Thread: sum range [2,3) (return arr[2])

Thread: sum range [3,5)

Thread: sum range [3,4) (return arr[3])

Thread: sum range [4,5) (return arr[4])

add results from two helper threads

add results from two helper threads

add results from two helper threads

Thread: sum range [5,10)

Thread: sum range [5,7)

Thread: sum range [5,6) (return arr[5])

Thread: sum range [6,7) (return arr[6])

add results from two helper threads

Thread: sum range [7,10)

Thread: sum range [7,8) (return arr[7])

Thread: sum range [8,10)

Thread: sum range [8,9) (return arr[8])

Thread: sum range [9,10) (return arr[9])

add results from two helper threads

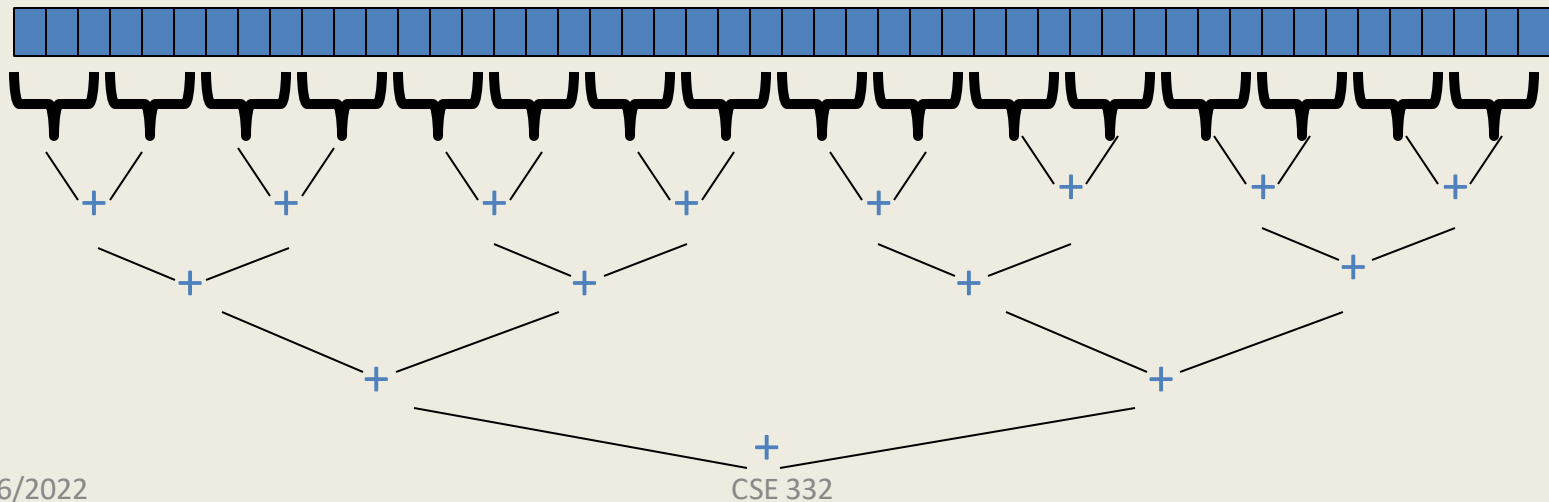
add results from two helper threads

add results from two helper threads

Divide-and-conquer

Same approach useful for many problems beyond sum

- If you have enough processors, total time $O(\log n)$
- Next lecture: study reality of $P \ll n$ processors
- Will write all our parallel algorithms in this style
 - But using a special fork-join library engineered for this style
 - Takes care of scheduling the computation well
 - Often relies on operations being associative (like +)



Thread Overhead

Creating and managing threads incurs cost

Two optimizations:

1. Use a *sequential cutoff*, typically around 500-1000
 - Eliminates lots of tiny threads
2. Do not create two recursive threads; create one thread and do the other piece of work “yourself”
 - Cuts the number of threads created by another 2x

Half the threads!

order of last 4 lines

Is critical – why?

```
// 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;
```

Note: run is a normal function call! execution won't continue until we are done with run

Better Java Thread Library

- 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
 - ...

Different terms, same basic idea

To use the ForkJoin Framework:

- A little standard set-up code (e.g., create a **ForkJoinPool**)

Don't subclass **Thread**

Don't override **run**

Do not use an **ans** field

Don't call **start**

Don't *just* call **join**

Don't call **run** to hand-optimize

Don't have a topmost call to **run**

Do subclass **RecursiveTask<V>**

Do override **compute**

Do return a **V** from **compute**

Do call **fork**

Do call **join** (which returns answer)

Do call **compute** to hand-optimize

Do create a pool and call **invoke**

See the web page for (linked from Handouts page on course website):

“A Beginner’s Introduction to the ForkJoin Framework”

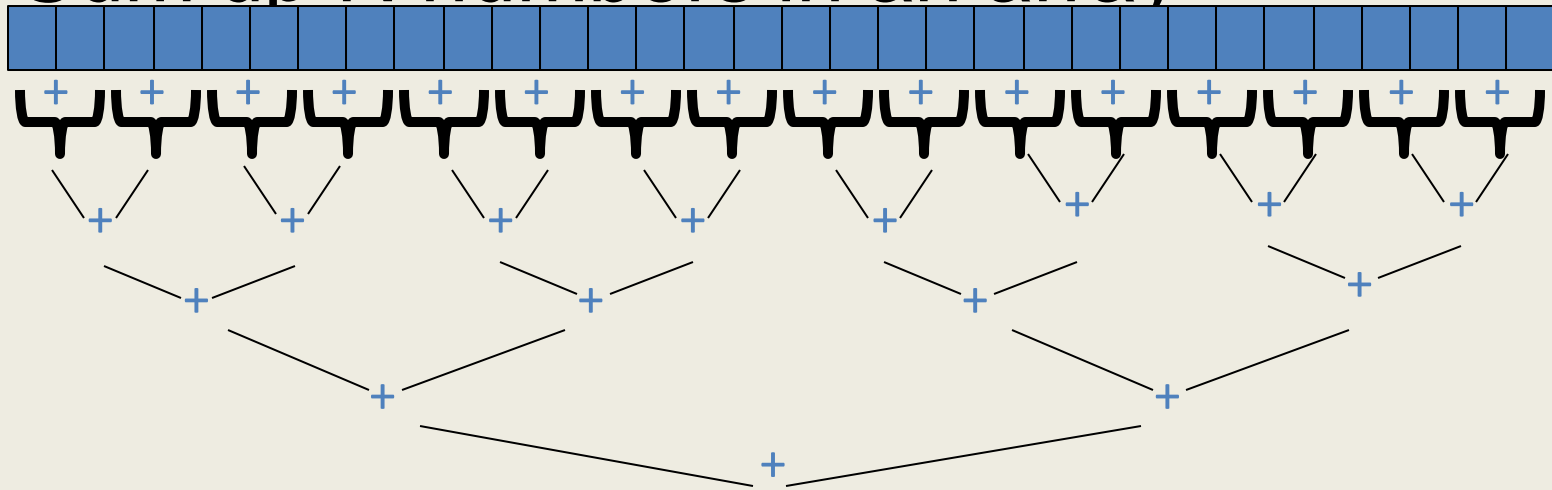
Fork Join Framework Version: (missing imports)

```
class SumArray extends RecursiveTask<Integer> {
    int lo; int hi; int[] arr; // fields to know what to do
    SumArray(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 {
            SumArray left = new SumArray(arr, lo, (hi+lo)/2);
            SumArray right = new SumArray(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;
        }
    }
}

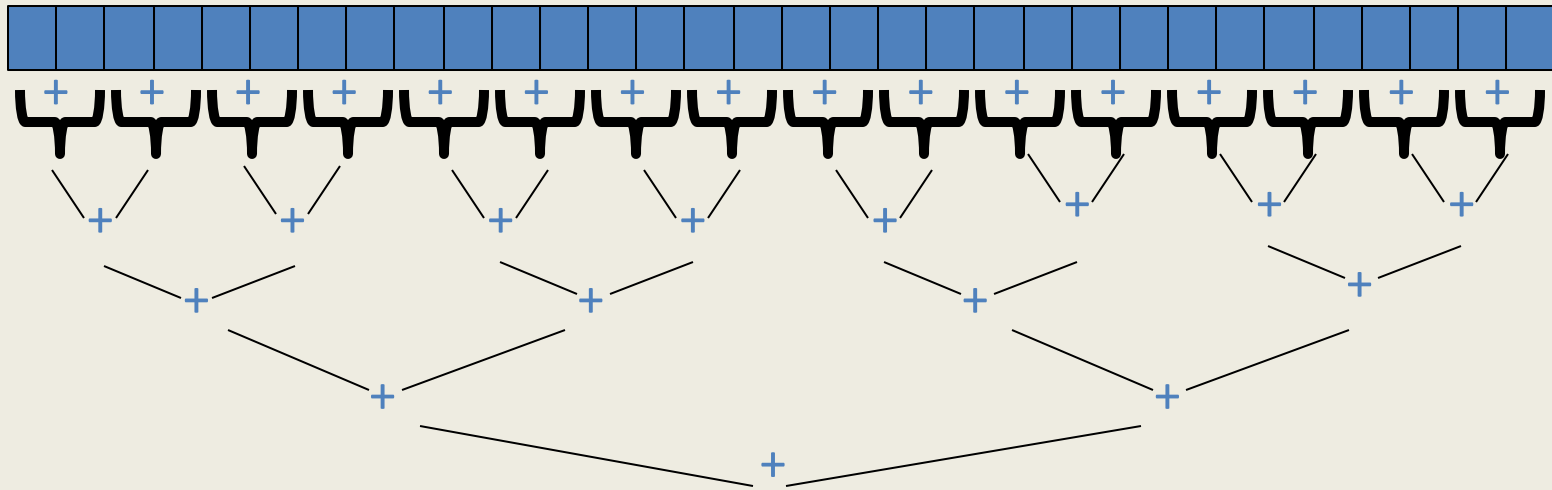
static final ForkJoinPool fjPool = new ForkJoinPool();
int sum(int[] arr) {
    return fjPool.invoke(new SumArray(arr, 0, arr.length));
    // invoke returns the value compute returns
}
```

Parallel Sum

- Sum up N numbers in an array



Parallel Max?



Reductions

- Same trick works for many tasks, e.g.,
 - is there an element satisfying some property (e.g., prime)
 - left-most element satisfying some property (e.g., first prime)
 - counts: number of strings that start with a vowel
 - are these elements in sorted order?
- Called a **reduction**, or **reduce** operation
 - reduce a collection of data items to a single item
 - result can be more than a single value, e.g., produce histogram from a set of test scores
- Very common parallel programming pattern

Parallel Vector Scaling

- Multiply every element in the array by 2



Maps

- A **map** operates on each element of a collection of data to produce a new collection of the same size
 - each element is processed independently of the others, e.g.
 - vector scaling
 - vector addition
 - test property of each element (is it prime)
 - uppercase to lowercase
 - ...
- Another common parallel programming pattern

Maps in ForkJoin Framework

```
class VecAdd extends RecursiveAction {
    int lo; int hi; int[] res; int[] arr1; int[] arr2;
    VecAdd(int l, int h, int[] r, int[] a1, int[] a2) { ... }
    protected void compute() {
        if (hi - lo < SEQUENTIAL_CUTOFF) {
            for (int i=lo; i < hi; i++)
                res[i] = arr1[i] + arr2[i];
        } else {
            int mid = (hi+lo)/2;
            VecAdd left = new VecAdd(lo, mid, res, arr1, arr2);
            VecAdd right = new VecAdd(mid, hi, res, arr1, arr2);
            left.fork();
            right.compute();
            left.join();
        }
    }
}

static final ForkJoinPool fjPool = new ForkJoinPool();
int[] add(int[] arr1, int[] arr2) {
    assert (arr1.length == arr2.length);
    int[] ans = new int[arr1.length];
    fjPool.invoke(new VecAdd(0, arr1.length, ans, arr1, arr2));
    return ans;
}
```

Maps and Reductions

Maps and reductions: the “workhorses” of parallel programming

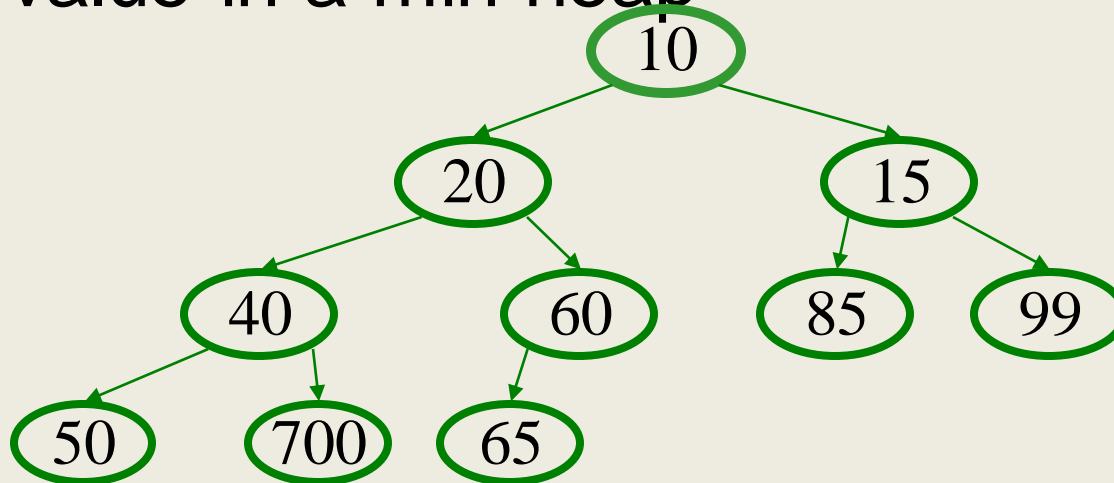
- By far the most important and common patterns
- Learn to recognize when an algorithm can be written in terms of maps and reductions
- makes parallel programming easy (plug and play)

Distributed Map Reduce

- You may have heard of Google's map/reduce
 - or open-source version called Hadoop
 - powers much of Google's infrastructure
- Idea: maps/reductions using many machines
 - same principles, applied to distributed computing
 - system takes care of distributing data, fault-tolerance
 - you just write code to handle one element, reduce a collection
- Co-developed by Jeff Dean (UW alum!)

Maps and Reductions on Trees

- Max value in a min-heap



- How to parallelize?
- Is this a map or a reduce?
- Complexity?

Analyzing Parallel Programs

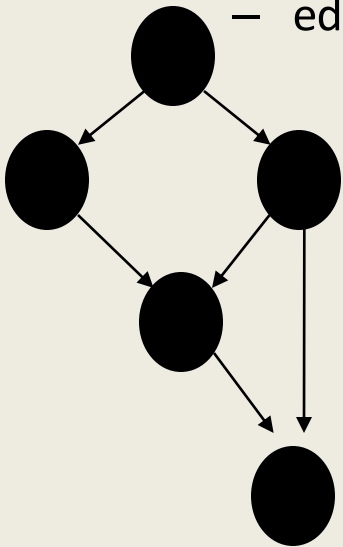
Let T_P be the running time on P processors

Two key measures of run-time:

- **Work**: How long it would take 1 processor = T_1
- **Span**: How long it would take infinity processors = T_∞
 - The hypothetical ideal for parallelization
 - This is the longest “dependence chain” in the computation
 - Example: $O(\log n)$ for summing an array
 - Also called “critical path length” or “computational depth”

The DAG

- Fork-join programs can be modeled with a DAG
 - nodes: pieces of work
 - edges: order dependencies



What's T_1 (work):

What's T_∞ (span):

A **fork** creates two children

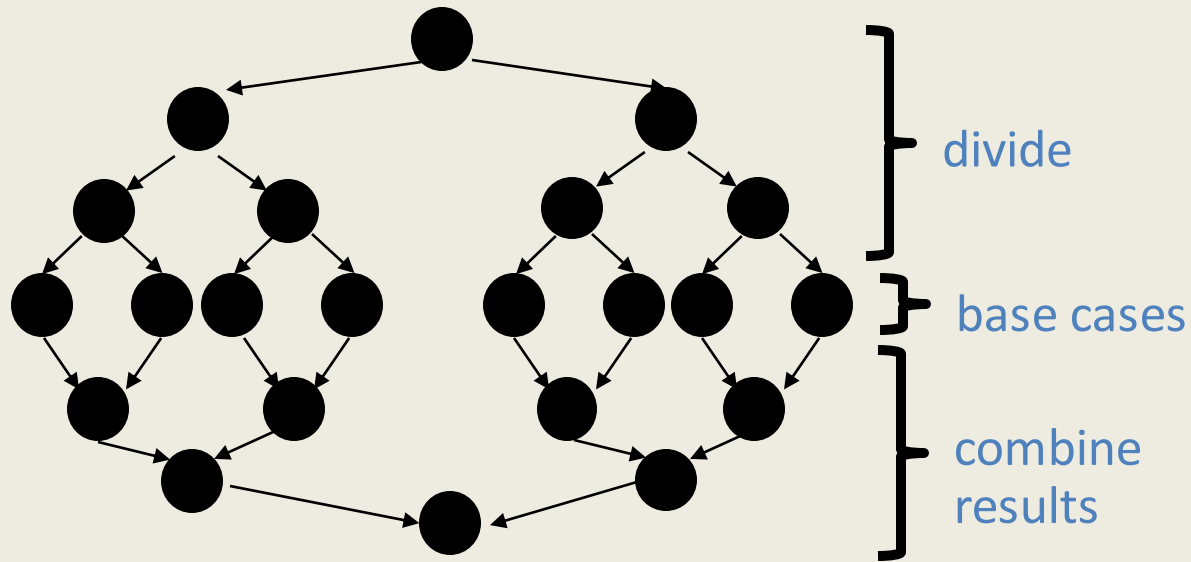
- new thread
- continuation of current thread

A **join** makes a node with two incoming edges

- terminated thread
- continuation of current thread

Divide and Conquer Algorithms

Our **fork** and **join** frequently look like this:



In this context, the span (T_∞) is:

- The longest dependence-chain; longest ‘branch’ in parallel ‘tree’
- Example: $O(\log n)$ for summing an array; we halve the data down to our cut-off, then add back together; $O(\log n)$ steps, $O(1)$ time for each
- Also called “critical path length” or “computational depth”

Parallel Speed-up

- **Speed-up** on **P** processors: T_1 / T_P
- If speed-up is **P**, we call it **perfect linear speed-up**
 - e.g., doubling **P** halves running time
 - hard to achieve in practice
- **Parallelism** is the maximum possible speed-up: T_1 / T_∞
 - if you had infinite processors

Estimating T_p

- How to estimate T_p (e.g., $P = 4$)?
- Lower bounds on T_p (ignoring memory, caching...)
 1. T_∞
 2. T_1 / P
 - which one is the tighter (higher) lower bound?
- The ForkJoin Java Framework achieves the following expected time asymptotic bound:

$$T_p \in \mathbf{O}(T_\infty + T_1 / P)$$

- this bound is optimal

Amdahl's Law

- Most programs have
 1. parts that parallelize well
 2. parts that don't parallelize at all

- The latter become bottlenecks

Amdahl's Law

- Let $T_1 = 1$ unit of time
- Let $S =$ proportion that can't be parallelized

$$1 = T_1 = S + (1 - S)$$

- Suppose we get perfect linear speedup on the parallel portion:

$$T_p =$$

- So the overall speed-up on P processors is (Amdahl's Law):

$$T_1 / T_p =$$

$$T_1 / T_\infty =$$

- If 1/3 of your program is parallelizable, max speedup is:

Pretty Bad News

- Suppose 25% of your program is sequential.
 - Then a billion processors won't give you more than a 4x speedup!
- What portion of your program must be parallelizable to get 10x speedup on a 1000 core GPU?
 - $10 \leq 1 / (S + (1-S)/1000)$
- Motivates minimizing sequential portions of your programs

Take Aways

- Parallel algorithms can be a big win
- Many fit standard patterns that are easy to implement
- Can't just rely on more processors to make things faster (Amdahl's Law)