

CSE 332: Intro to Parallelism: Multithreading and Fork-Join

Richard Anderson
Spring 2016

Announcements

- Read parallel computing notes by Dan Grossman 2.1-3.4
- Homework 5 – available Wednesday
- Exams – not graded yet

Sequential

- Sum up N numbers in an array
 - Complexity?

Parallel Sum

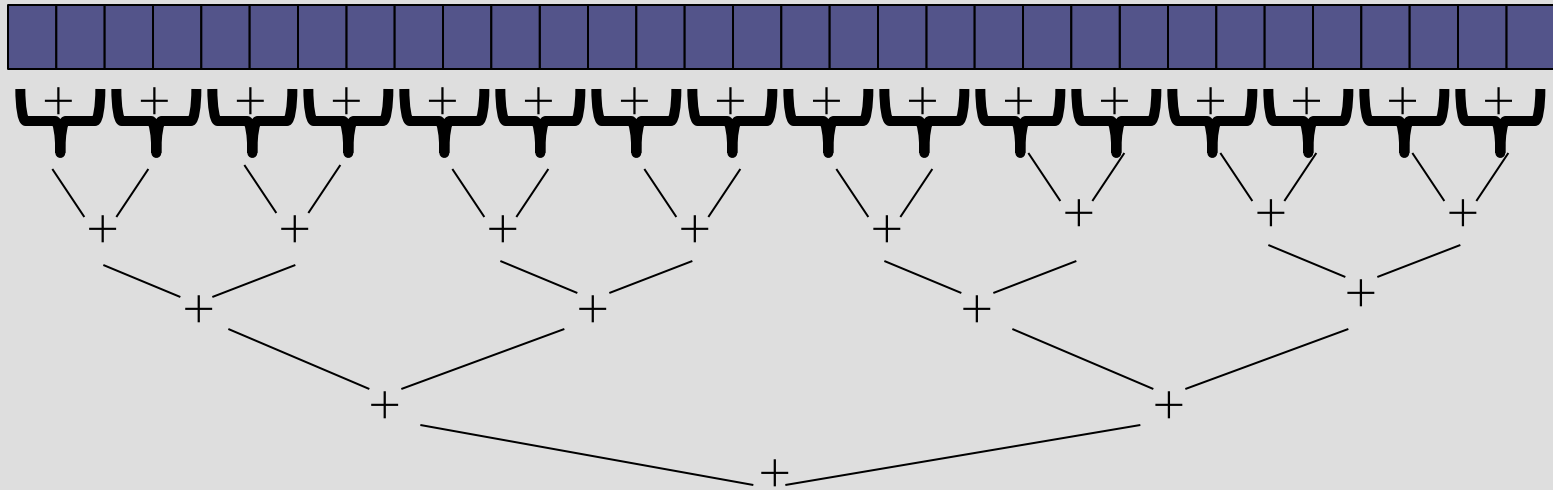
- Sum up N numbers in an array
 - with two processors

Parallel Sum

- Sum up N numbers in an array
 - with N processors?

Parallel Sum

- Sum up N numbers in an array



- Complexity?
- How many processors?
- Faster with infinite processors?

Changing a Major Assumption

- So far, we have assumed:

One thing happens at a time

- Called sequential programming
- Dominated until roughly 2005
 - what changed?

A Simplified History

From roughly 1980-2005, desktop computers got exponentially faster at running sequential programs

- About twice as fast every couple years

Writing parallel (multi-threaded) code is harder than sequential

- Especially in common languages like Java and C

But nobody knows how to continue this

- Increasing clock rate generates too much heat
- Relative cost of memory access is too high
- But we can keep making “wires exponentially smaller” (Moore’s “Law”), so put multiple processors on the same chip (“multicore”)

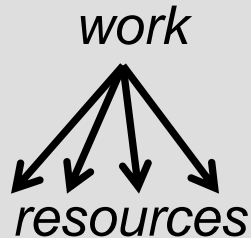
Who Implements Parallelism

- User
- Application
- Operating System
- Programming Language, Compiler
- Algorithm
- Processor Hardware

Parallelism vs. Concurrency

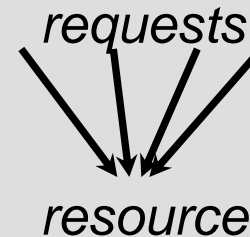
Parallelism:

Use extra resources to solve a problem faster



Concurrency:

Manage access to shared resources



An analogy

A program is like a recipe for a cook

- Sequential: one cook who does one thing at a time

Parallelism: (Let's get the job done faster!)

- Have lots of potatoes to slice?
- Hire helpers, hand out potatoes and knives
- But too many chefs and you spend all your time coordinating

Concurrency: (We need to manage a shared resource)

- Lots of cooks making different things, but only 4 stove burners
- Want to allow access to all 4 burners, but not cause spills or incorrect burner settings

Shared Memory with Threads

Old story: A running program has

- One *program counter* (current statement executing)
- 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)
- *Static fields*

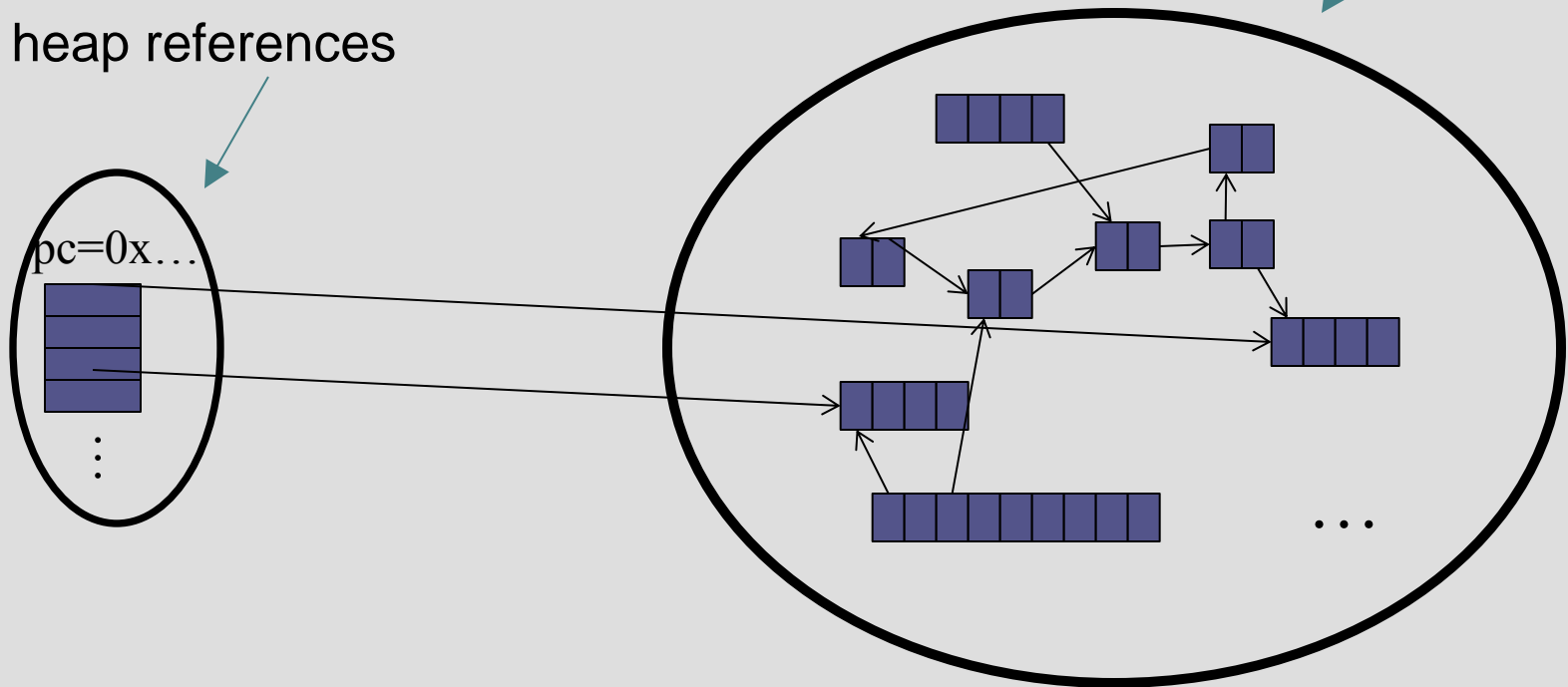
New story:

- A set of *threads*, each with its own program counter & call stack
 - No access to another thread's local variables
- Threads can share static fields / objects
 - To *communicate*, write values to some shared location that another thread reads from

Old Story: one call stack, one pc

- Call stack with local variables
- pc determines current statement
- local variables are numbers/null or heap references

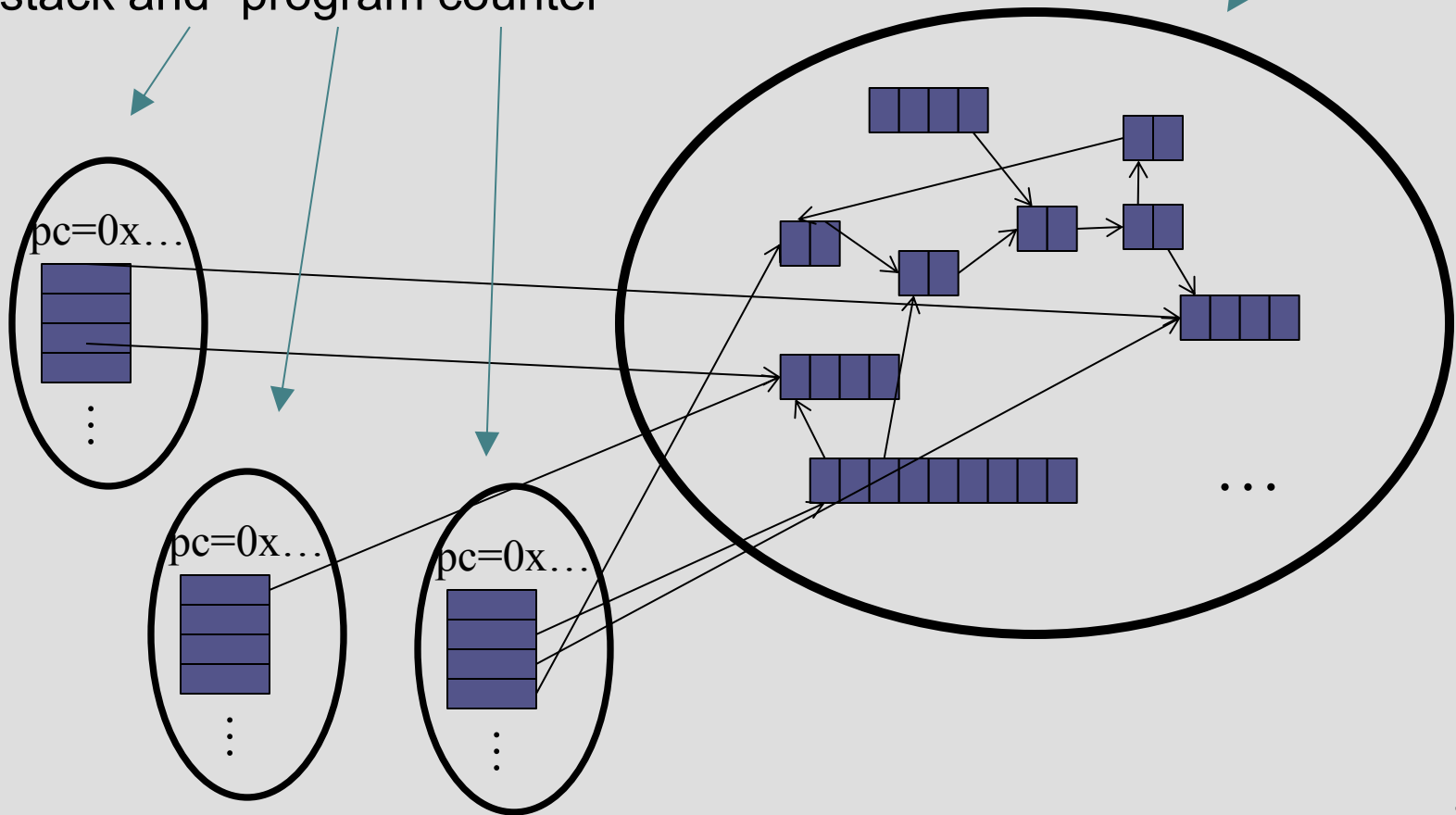
Heap for all objects and static fields



New Story: 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



Other models

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

- **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 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”

Our Needs

To write a shared-memory parallel program, need new primitives from a programming language or library

- Ways to create and *run multiple things at once*
 - Let's call these things **threads**
- Ways for threads to *share memory*
 - Often just have threads with references to the same objects
- Ways for threads to *coordinate (a.k.a. synchronize)*
 - For now, a way for one thread to wait for another to finish
 - Other primitives when we study concurrency

Threads vs. Processors

What happens if you start 5 threads on a machine with only 4 processors?

Threads vs. Processors

For sum operation:

- with 3 processors available,
using 4 threads would take 50% more time than 3 threads

Fork-Join Parallelism

1. Define thread

- Java: define subclass of `java.lang.Thread`, override `run`

2. Fork: instantiate a thread and start executing

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

3. Join: wait for thread to terminate

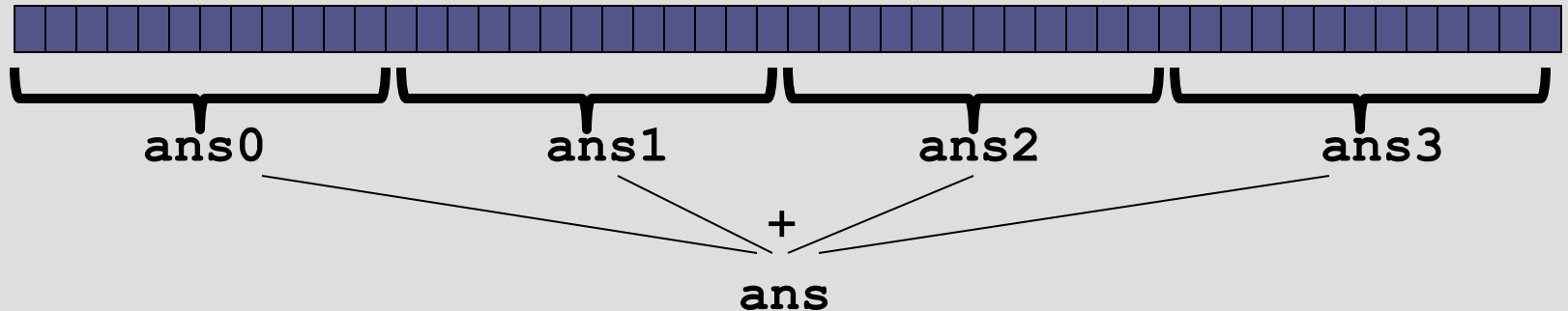
- Java: call `join()` method, which returns when thread finishes

Above uses basic thread library build into Java

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

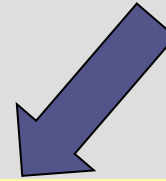
Sum with Threads

For starters: have 4 threads simultaneously sum $\frac{1}{4}$ of the array

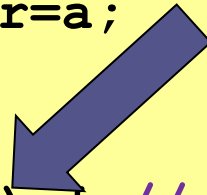


- Create 4 *thread objects*, each given $\frac{1}{4}$ of the array
- Call `start()` on each thread object to run it in parallel
- Wait for threads to finish using `join()`
- Add together their 4 answers for the final result

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() { //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

Part 2: sum routine

```
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;
}
```

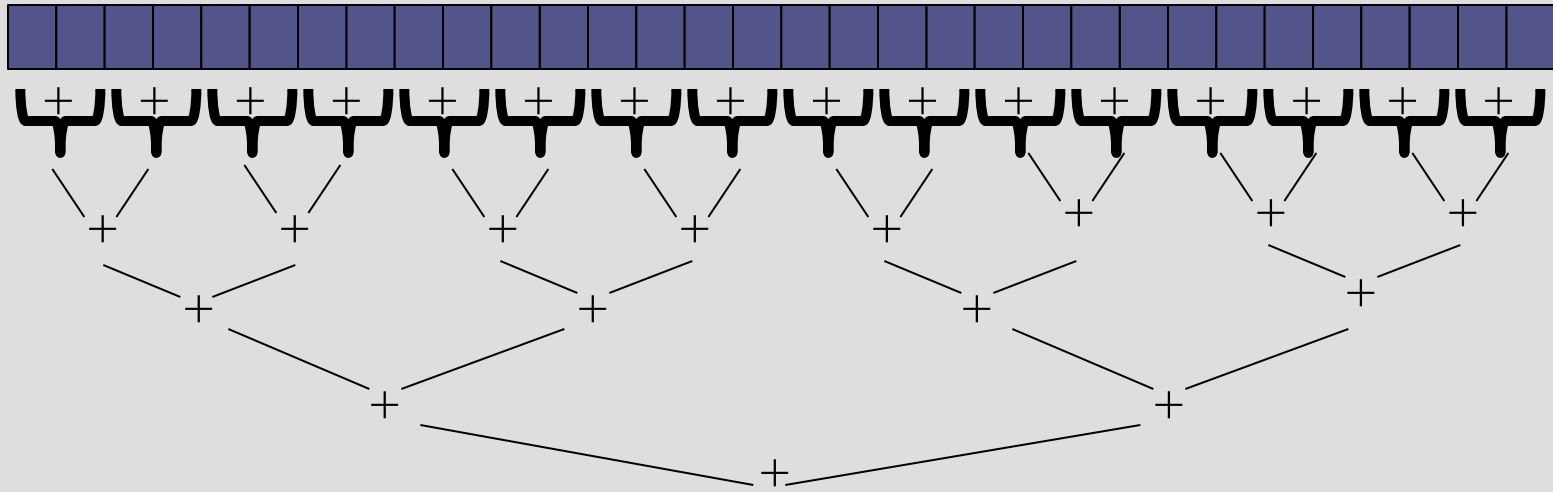
Parameterizing by number of threads

```
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;
}
```

Recursive problem decomposition

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

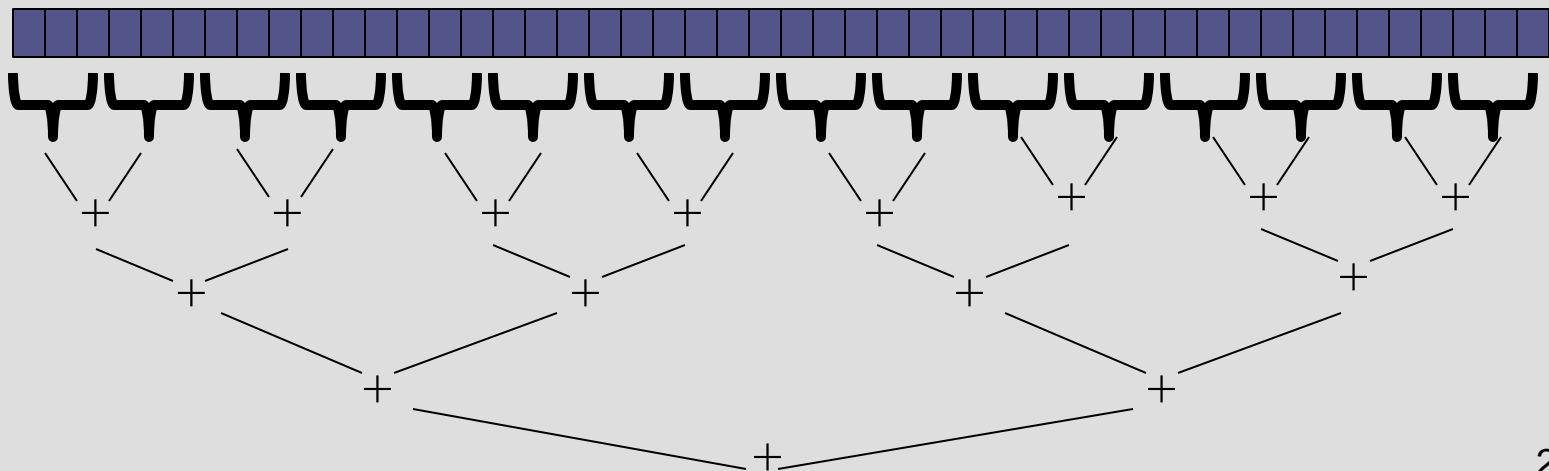
add results from two helper threads

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 7 standard libraries
 - (Also available for Java 6 as a downloaded `.jar` file)
 - 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 in to project 3 description):

“A Beginner's Introduction to the ForkJoin Framework”

Fork Join Framework Version:

```
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
}
```