

Lecture 18: Introduction to Multithreading & Fork-Join Parallelism

CSE 332: Data Structures & Parallelism

Yafqa Khan

Summer 2025

Changing a major assumption

Assumption: **One thing happened at a time**

Called sequential programming – everything part of one sequence

Removing this assumption creates major challenges & opportunities

- 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)

A simplified view of history

- Writing correct and efficient multithreaded code is often much more difficult than for single-threaded (i.e., sequential) code
 - Especially in common languages like Java and C
 - So typically stay sequential if possible
- From roughly 1980-2005, desktop computers got exponentially faster at running sequential programs
 - About twice as fast every couple years
- 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”)

What to do with multiple processors?

- Next computer you buy will likely have 4 processors
 - Wait a few years and it will be 8, 16, 32, ...
 - The chip companies have decided to do this (not a “law”)

2010s

• What can you do with them?

- Run multiple totally different programs at the same time
 - Already do that? Yes, but with time-slicing
- Do multiple things at once in one program
 - Our focus – more difficult
 - Requires rethinking everything from asymptotic complexity to how to implement data-structure operations

chrome word

① Parallelism vs ② Concurrency

- No agreed definition :(

Our definition:

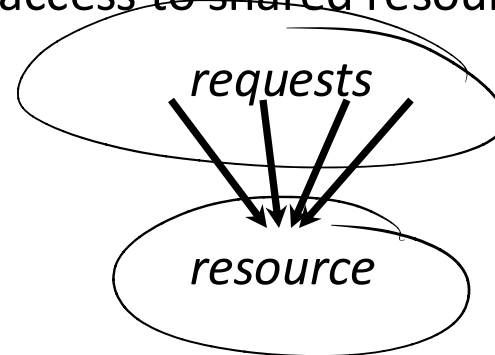
Parallelism:

Use extra resources to solve a problem faster



Concurrency:

Correctly and efficiently manage access to shared resources



There is some connection:

- Common to use threads for both
- If parallel computations need access to shared resources, then the concurrency needs to be managed

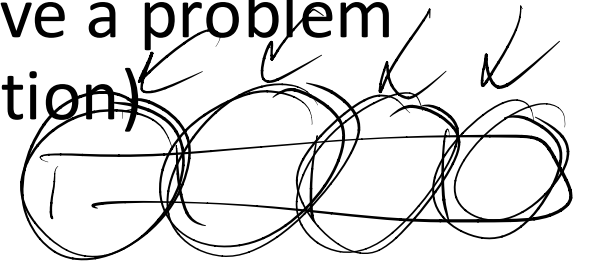
An analogy

12x

- Intro CS idea: A program is like a recipe for a cook
 - One cook who does one thing at a time! (Sequential)
- Parallelism: (Let's get the job done faster!) *multiple*
 - 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 1 fridge
 - Want to allow access to this fridge without fighting

Parallelism Example

- Parallelism: Use extra computational resources to solve a problem faster (increasing throughput via simultaneous execution)
- Pseudocode (not Java yet) for array sum:
 - No such 'FORALL' construct, but we'll see something similar
 - Bad style, but with 4 processors may get roughly 4x speedup



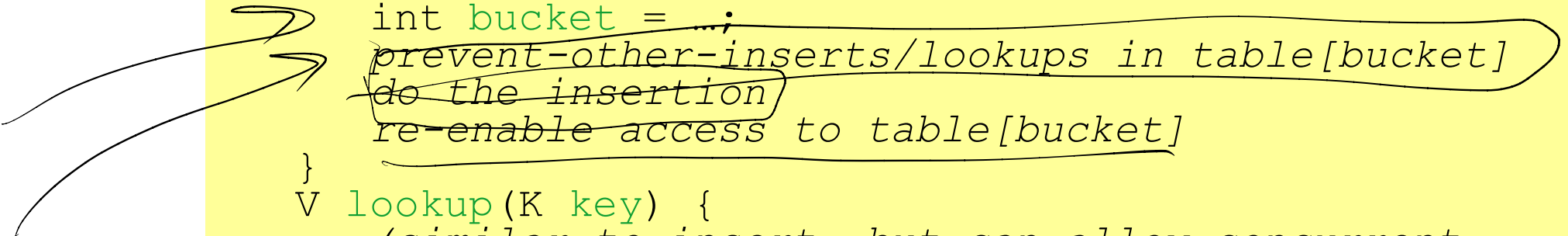
```
int sum(int[] arr){
    res = new int[4];
    len = arr.length;
    FORALL(i=0; i < 4; i++){ //parallel iterations
        res[i] = sumRange(arr, i*len/4, (i+1)*len/4);
    }
    return res[0]+res[1]+res[2]+res[3];
}

int sumRange(int[] arr, int lo, int hi) {
    result = 0;
    for(j=lo; j < hi; j++)
        result += arr[j];
    return result;
}
```

Concurrency Example

- Concurrency: Correctly and efficiently manage access to shared resources (from multiple possibly-simultaneous clients)
 - e.g., Multiple threads accessing a hash-table, but not getting in each others' ways
- Pseudocode (not Java) for a shared chaining hashtable
 - Essential correctness issue is preventing bad interleavings
 - Essential performance issue not preventing good concurrency
 - One 'solution' to preventing bad inter-leavings is to do it all sequentially

```
class Hashtable<K,V> {  
    ...  
    void insert(K key, V value) {  
        int bucket = ...;  
        prevent-other-inserts/lookups in table[bucket]  
        do the insertion  
        re-enable access to table[bucket]  
    }  
    V lookup(K key) {  
        (similar to insert, but can allow concurrent  
        lookups to same bucket)  
    }  
}
```



Aside

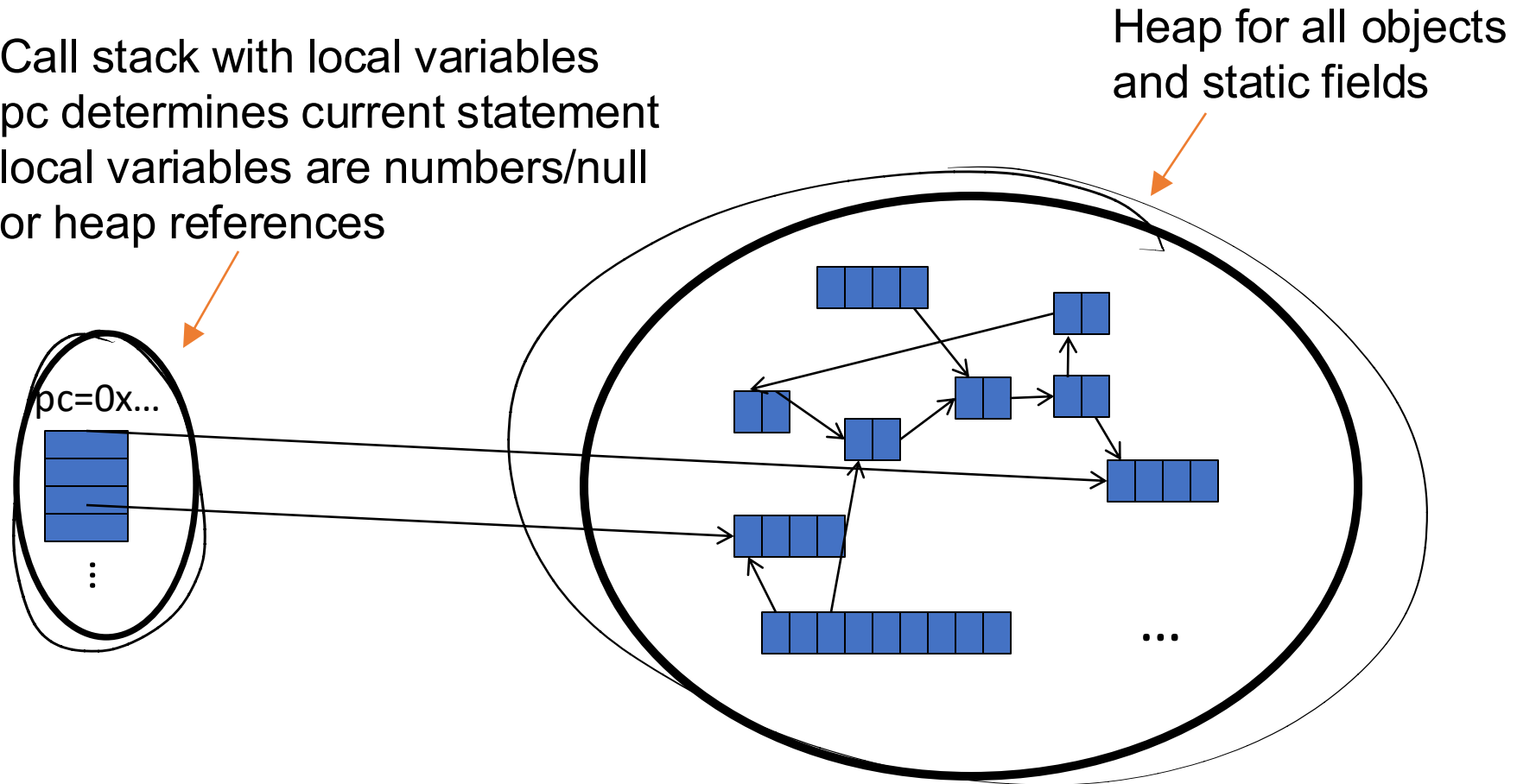
Shared memory with Threads

The model we will assume is shared memory with explicit 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 (implicitly) share static fields / objects
 - To communicate, write values to some shared location that another thread reads from

Old Story: One call stack, one pc

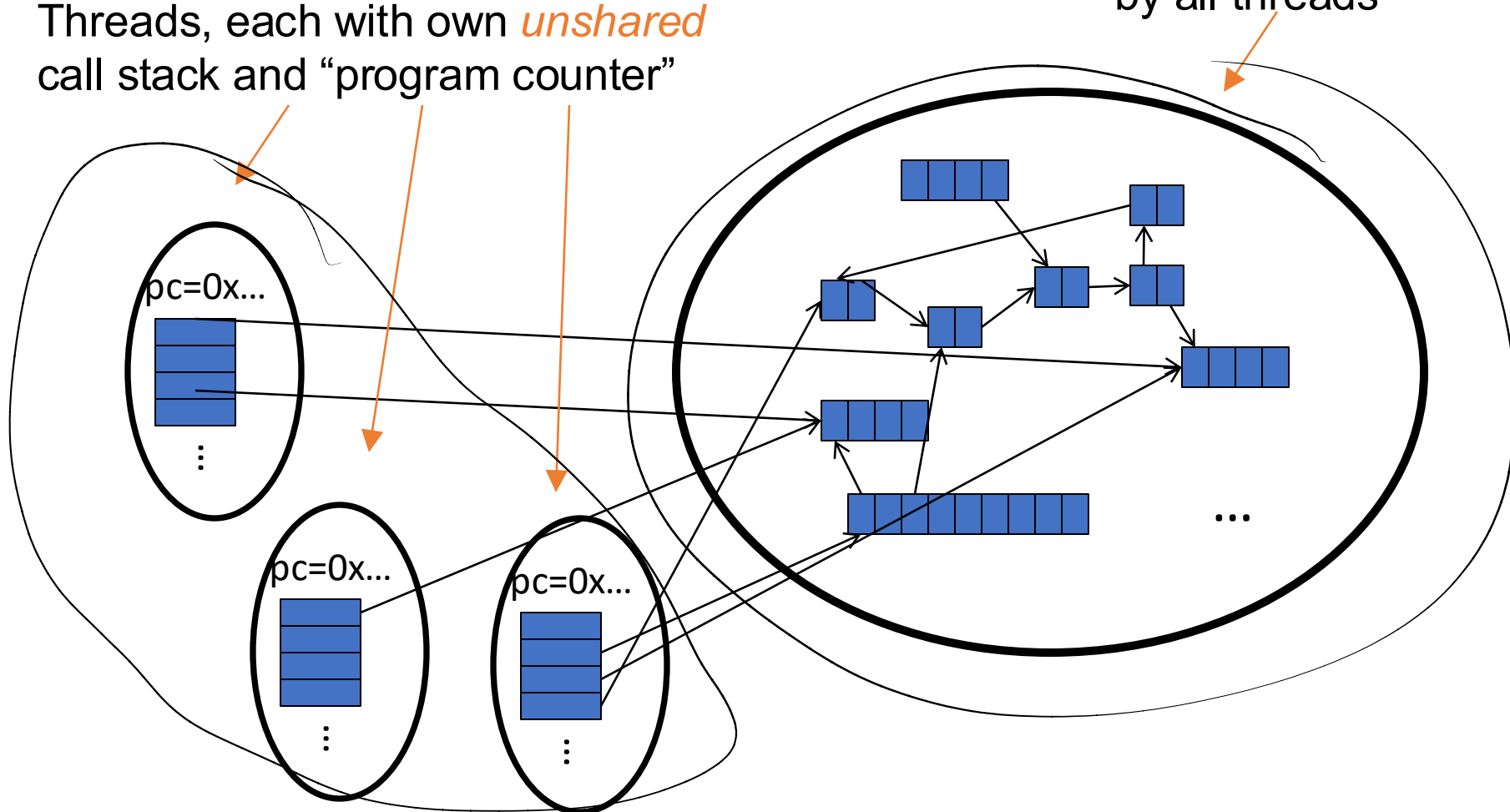
- Call stack with local variables
- pc determines current statement
- local variables are numbers/null or heap references



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

- 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”

map / reduce

Our Needs

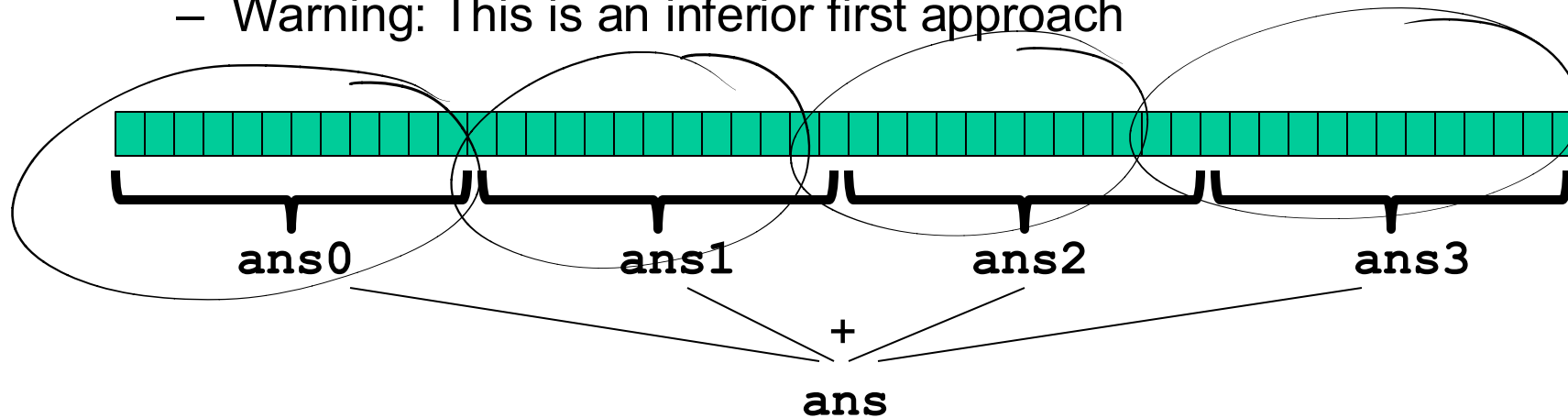
- To write a shared-memory parallel program, need new primitives from a programming language or library
- Ways to create and run multiple things at once
 - Let's call these things threads
- Ways for threads to share memory
 - Often just have threads with references to the same objects
- Ways for threads 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

Java basics

- First learn some basics built into Java via java.lang.Thread
 - Then a better library for parallel programming
- To get a new thread running:
 - Define a subclass C of java.lang.Thread, overriding run
 - Create an object of class C
 - Call that object's start method
 - start sets off a new thread, using run as its "main"
- What if we instead called the run method of C?
 - This would just be a normal method call, in the current thread
- Let's see how to share memory and coordinate via an example...

Parallelism Idea

- Example: Sum elements of a large array
- Idea: Have 4 threads simultaneously sum 1/4 of the array
 - Warning: This is an inferior first approach



- Create 4 thread objects, each given a portion of the work
- Call `start()` on each thread object to actually run it in parallel
- Wait for threads to finish using `join()`
- Add together their 4 answers for the final result

First attempt, part 1

```
class SumThread extends java.lang.Thread {  
    int lo; // fields, assigned in the 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];  
    }  
}
```

First attempt, continued (wrong)

```
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  
}
```

```
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);  
    for(int i=0; i < 4; i++) // combine results  
        ans += ts[i].ans;  
    return ans;  
}
```

Second attempt (still wrong)

```
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
}
```

```
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(); // start not run
    }
    for(int i=0; i < 4; i++) // combine results
        ans += ts[i].ans;
    return ans;
}
```

Third attempt (correct in spirit)

```
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
}
```

```
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;
}
```

A better approach

Several reasons why this is a poor parallel algorithm

1. Want code to be reusable and efficient across platforms

- “Forward-portable” as core count grows
- So at the *very* least, parameterize by the 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;  
}
```

A better approach

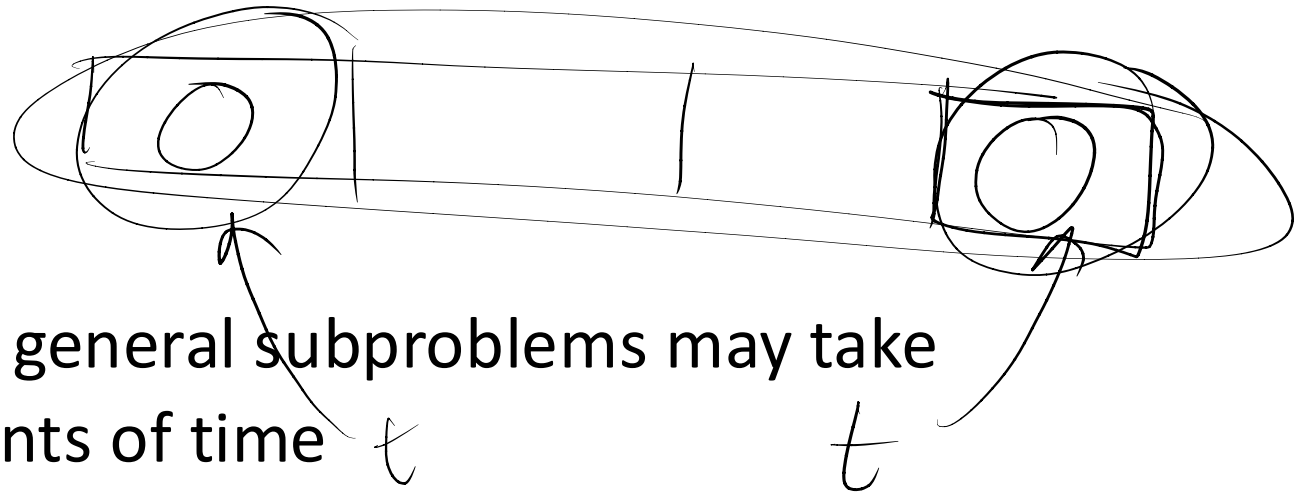
2. Want to use (only) processors “available to you *now*”

- Not used by other programs or threads in your program
 - Maybe caller is also using parallelism
 - Available cores can change even while your threads run
- If you have 3 processors available and using 3 threads would take time **X**, then creating 4 threads would take time **1.5X**
 - Example: 12 units of work, 3 processors
 - Work divided into 3 parts will take 4 units of time
 - Work divided into 4 parts will take 3*2 units of time

```
// numThreads == numProcessors is bad
// if some are needed for other things
int sum(int[] arr, int numTs) {
    ...
}
```

A better approach

3. Though unlikely for **sum**, in general subproblems may take significantly different amounts of time t



Example: Apply method **f** to every array element, but maybe **f** is much slower for some data items

Example: Is a large integer prime?

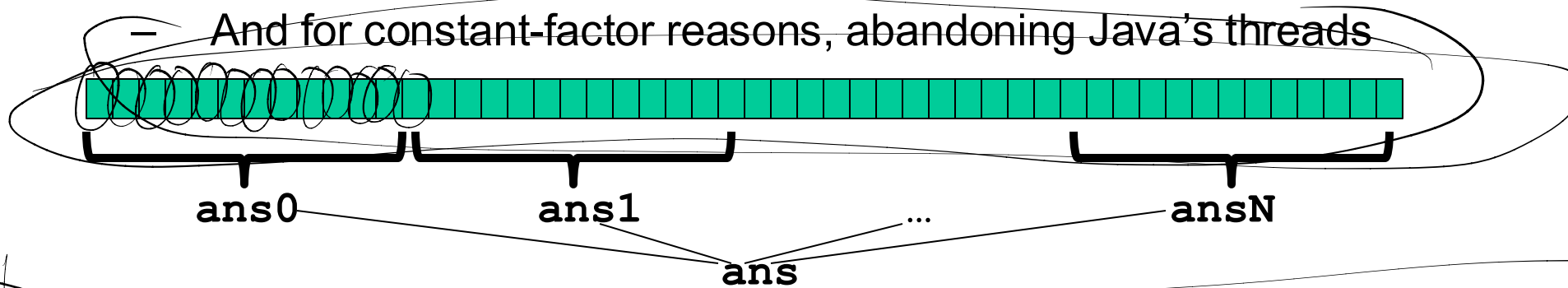
If we create 4 threads and all the slow data is processed by 1 of them, we won't get nearly a 4x speedup

Example of a load imbalance

A better approach

The counterintuitive (?) solution to all these problems is to **cut up our problem into many pieces**, far more than the number of processors

- But this will require changing our algorithm
- And for constant-factor reasons, abandoning Java's threads

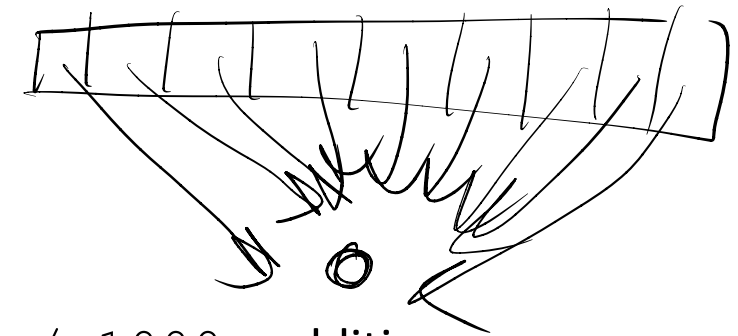


1. **Forward-portable:** However many processors exist, they will be kept busy w/ small chunks
2. **Processors available:** Hand out “work chunks” as you go
3. **Load imbalance:** Variation probably small if pieces of work are small

Naive algorithm is poor

Suppose we create 1 thread to process every 1000 elements

```
int sum(int[] arr) {  
    ...  
    int numThreads = arr.length / 1000;  
    SumThread[] ts = new SumThread[numThreads];  
    ...  
}
```



Then the “combining of results” part of the code will have `arr.length / 1000` additions

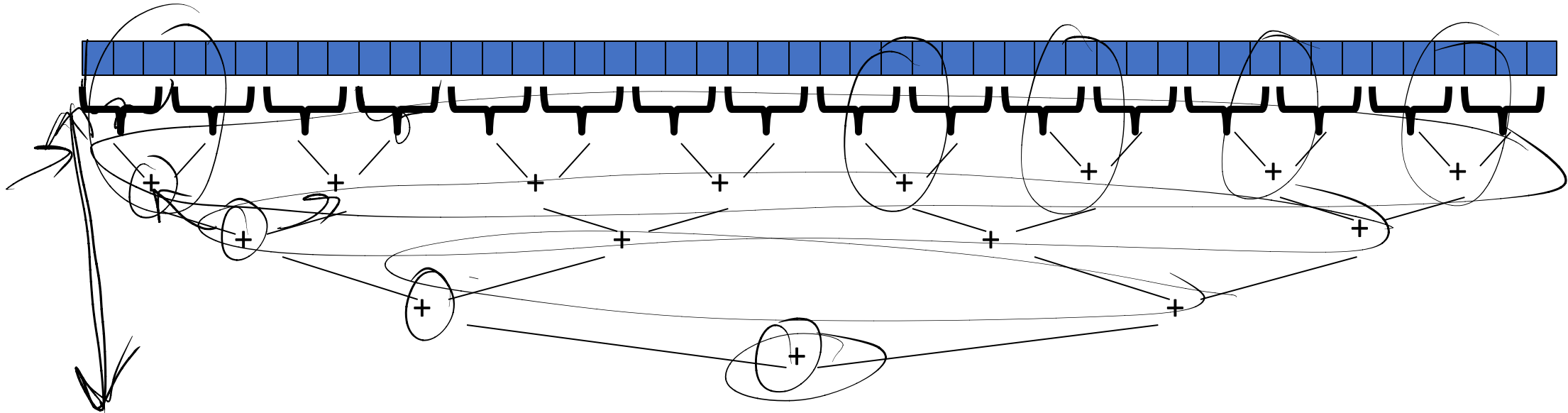
- Linear in size of array (with constant factor 1/1000)
- Previous we had only 4 pieces ($\Theta(1)$) to combine)
- In the extreme, suppose we create one thread per element – If we use a for loop to combine the results, we have N iterations
- In either case we get a $\Theta(N)$ algorithm with the combining of results as the bottleneck....

A better idea: Divide and Conquer!

1) Divide problem into pieces recursively:

- Start with full problem at root
- Halve and make new thread until size is at some cutoff

2) Combine answers in pairs as we return from recursion (see diagram)

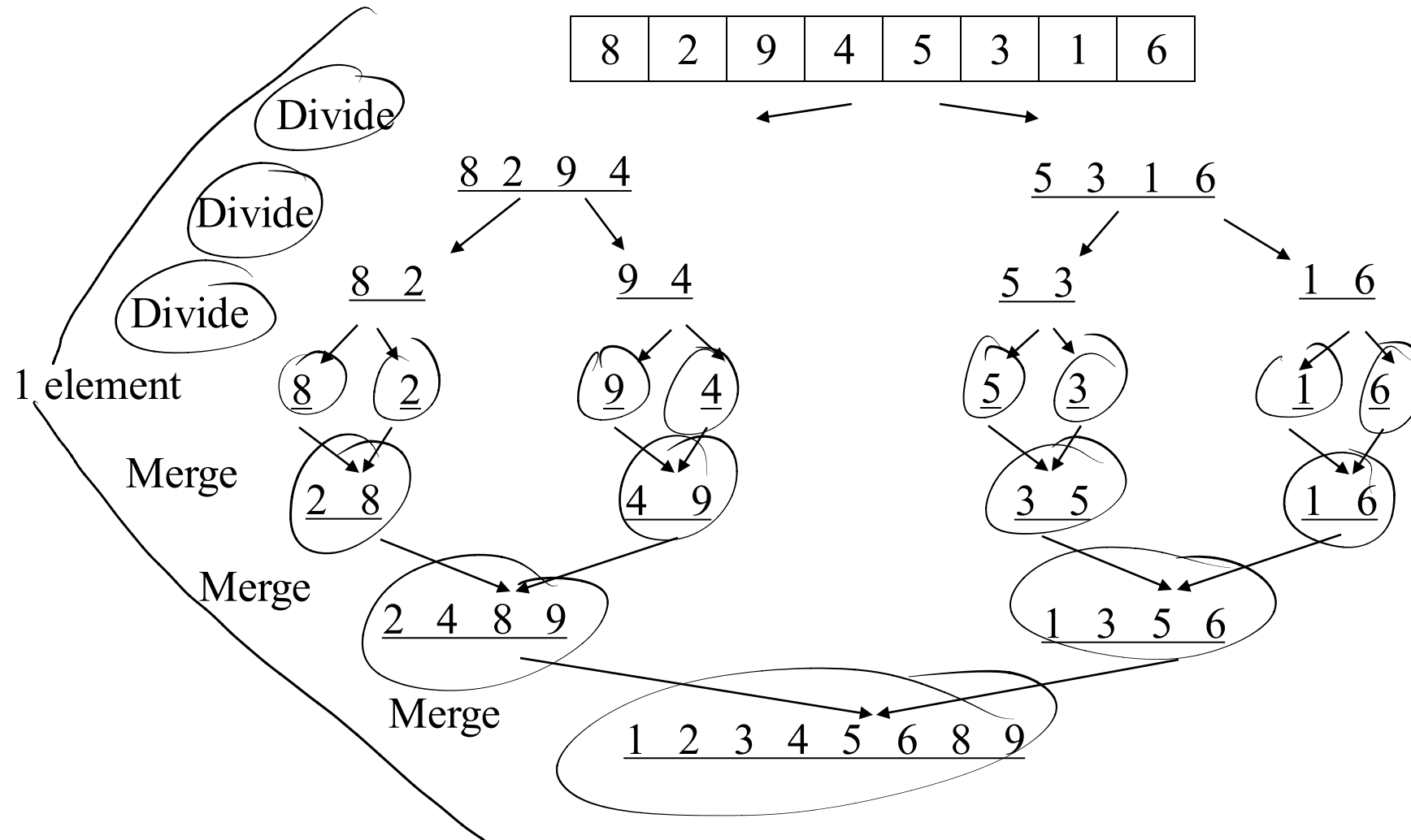


This will start small, and 'grow' threads to fit the problem

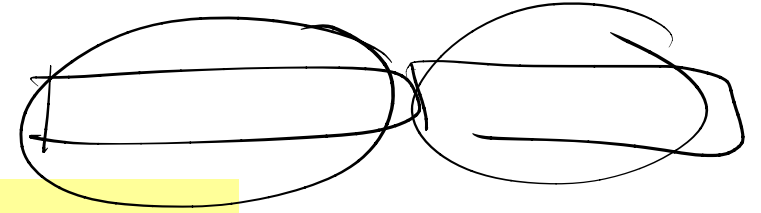
This is straightforward to implement using divide-and-conquer

- Parallelism for the recursive calls

Remember Mergesort?



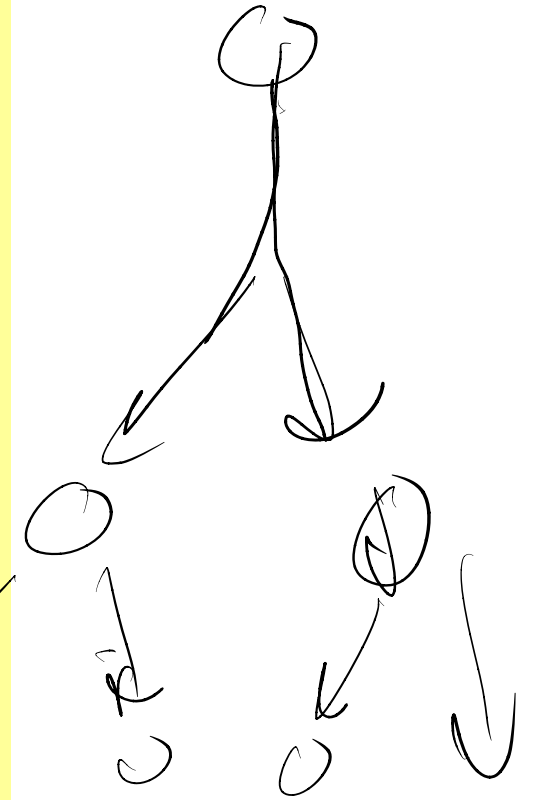
Code looks something like this



```
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;
}
```

100



Optimization: ~Half the threads!

order of last 4 lines
is critical – why?

```
// wasteful: don't  
SumThread left = ...  
SumThread right = ...
```

```
left.start();  
right.start();
```

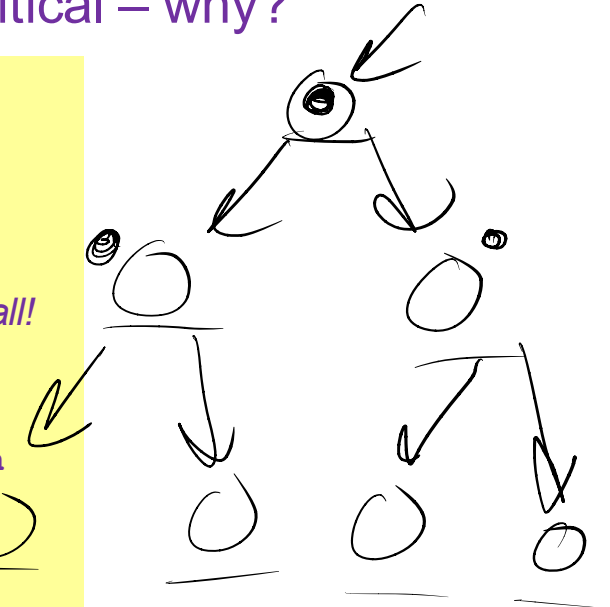
```
left.join();  
right.join();  
ans=left.ans+right.ans;
```

```
// 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`

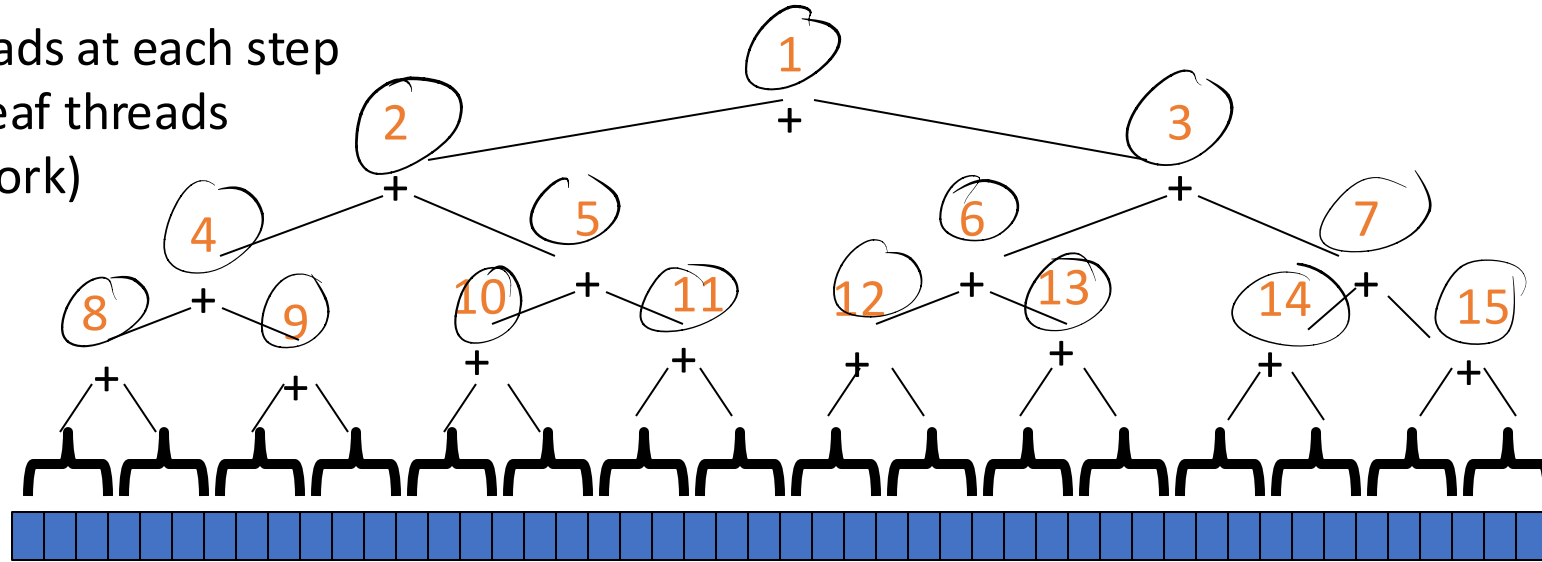


- If a *language* had built-in support for fork-join parallelism, I would expect this hand-optimization to be unnecessary
- But the *library* we are using expects you to do it yourself
 - And the difference is surprisingly substantial
- Again, no difference in theory

Creating Fewer Threads

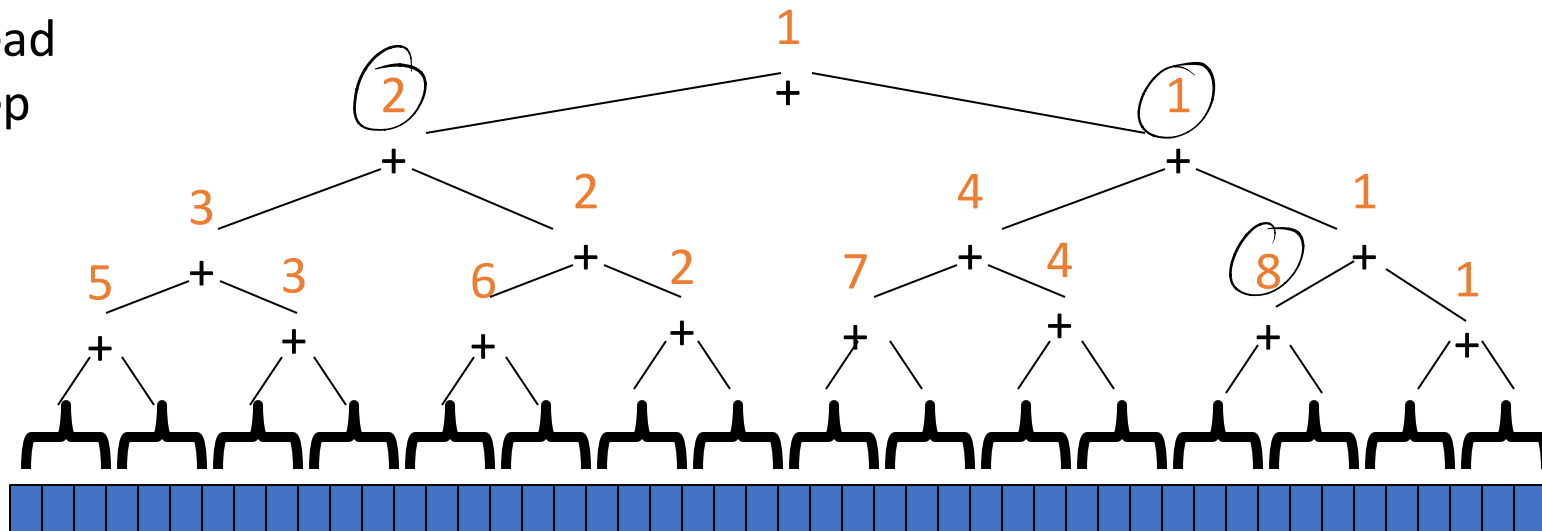
2 new threads at each step
(and only leaf threads
do much work)

Total =
15 threads



1 new thread
at each step

Total =
8 threads



That library, finally

Use the FJ framework
for parallelism!

- 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
 - ...
 - Library's implementation is a fascinating but advanced topic

Different terms, same basic idea

To use the ForkJoin Framework:

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

Java Threads:

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**

ForkJoin Framework:

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**

Fork-Join Framework Version: ✓

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

Any Questions?

Reduce



It shouldn't be too hard to imagine how to modify our code to:

1. **Find the maximum element in an array.**
2. Determine if there is an element meeting some property.
3. Find the left-most element satisfying some property.
4. Count the number of elements meeting some property.
5. Check if elements are in sorted order.
6. [And so on...]

In $O(\log N)$!!!

Fork-Join Reduce:

```
class MaxTask extends RecursiveTask<Integer> {
    int lo; int hi; int[] arr; // fields to know what to do
    MaxTask(int[] a, int l, int h) { ... }
    protected Integer compute() { // return answer
        if (hi - lo <= SEQUENTIAL CUTOFF) {
            int ans = a[lo]; // local var, not a field
            for (int i = lo; i < hi; i++)
            ans = Math.max(ans, a[i]);
            return ans;
        } else {
            MaxTask left = new MaxTask(arr, lo, (hi+lo)/2);
            MaxTask right = new MaxTask(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 Math.max(leftAns, rightAns);
        }
    }
}

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

Reduce

You'll do similar problems in section.

The key is to describe:

1. How to compute the answer at the cut-off.
2. How to merge the results of two subarrays.

We say parallel code like this “**reduces**” the array

We're reducing the arrays to a single item

Then combining with an **associative** operation.

e.g. sum, max, leftmost, product, count, or, and, ...

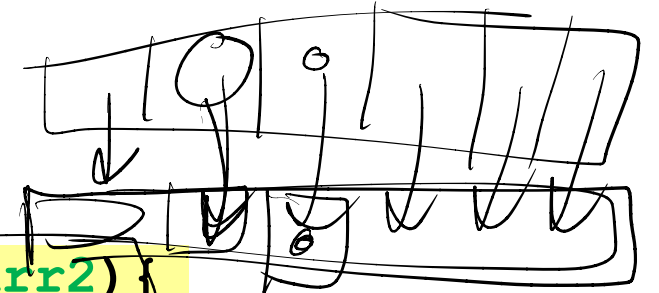
Doesn't have to be a single number, could be an object.

A handwritten diagram illustrating the reduction of an array to a single item. It shows two expressions: $(a + b) + c$ and $a + (b + c)$, both enclosed in large hand-drawn oval shapes. An equals sign is placed between the two expressions, demonstrating the associative property of addition.

Even easier: Maps (Data Parallelism)

- A **map** operates on each element of a collection independently to create a new collection of the same size
 - No combining results
 - For arrays, this is so trivial some hardware has direct support
- Canonical example: Vector addition

```
int[] vector_add(int[] arr1, int[] arr2) {  
    assert (arr1.length == arr2.length);  
    result = new int[arr1.length];  
    FORALL(i=0; i < arr1.length; i++) {  
        result[i] = arr1[i] + arr2[i];  
    }  
    return result;  
}
```



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 POOL = new ForkJoinPool();
int[] add(int[] arr1, int[] arr2) {
    assert (arr1.length == arr2.length);
    int[] ans = new int[arr1.length];
    POOL.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 two most important and common patterns
 - Two more-advanced patterns in next lecture
- Learn to recognize when an algorithm can be written in terms of maps and reductions
- Use maps and reductions to describe (parallel) algorithms
- Programming them becomes “trivial” with a little practice
 - Exactly like sequential for-loops seem second-nature

Map vs reduce in ForkJoin framework

In our examples:

- Reduce:
 - Parallel-sum extended RecursiveTask
 - Result was returned from compute()
- Map:
 - Class extended was RecursiveAction
 - Nothing returned from compute()
 - In the above code, the 'answer' array was passed in as a parameter

Map vs reduce in ForkJoin framework

In our examples:

- Reduce:
 - Parallel-sum extended RecursiveTask
 - Result was returned from compute()
- Map:
 - Class extended was RecursiveAction
 - Nothing returned from compute()
 - In the above code, the 'answer' array was passed in as a parameter