



# CSE 332: Data Abstractions

## Lecture 17: Introduction to Multithreading & Fork-Join Parallelism

Ruth Anderson  
Autumn 2013

# *Announcements*

- **Project 2** – phase B due Wednesday at 11pm
- **Homework 5** – due Fri
- **Project 3** – posted soon, partners o.k.

# *Changing a major assumption*

So far most or all of your study of computer science has assumed

## *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”)
- 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

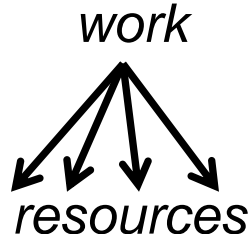
# *Parallelism vs. Concurrency*

Note: Terms not yet standard but the perspective is essential

- Many programmers confuse these concepts

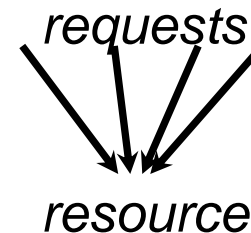
## Parallelism:

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*

CS1 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!)

- 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

# 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)

Ex: 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)  
    }  
}
```

# *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*

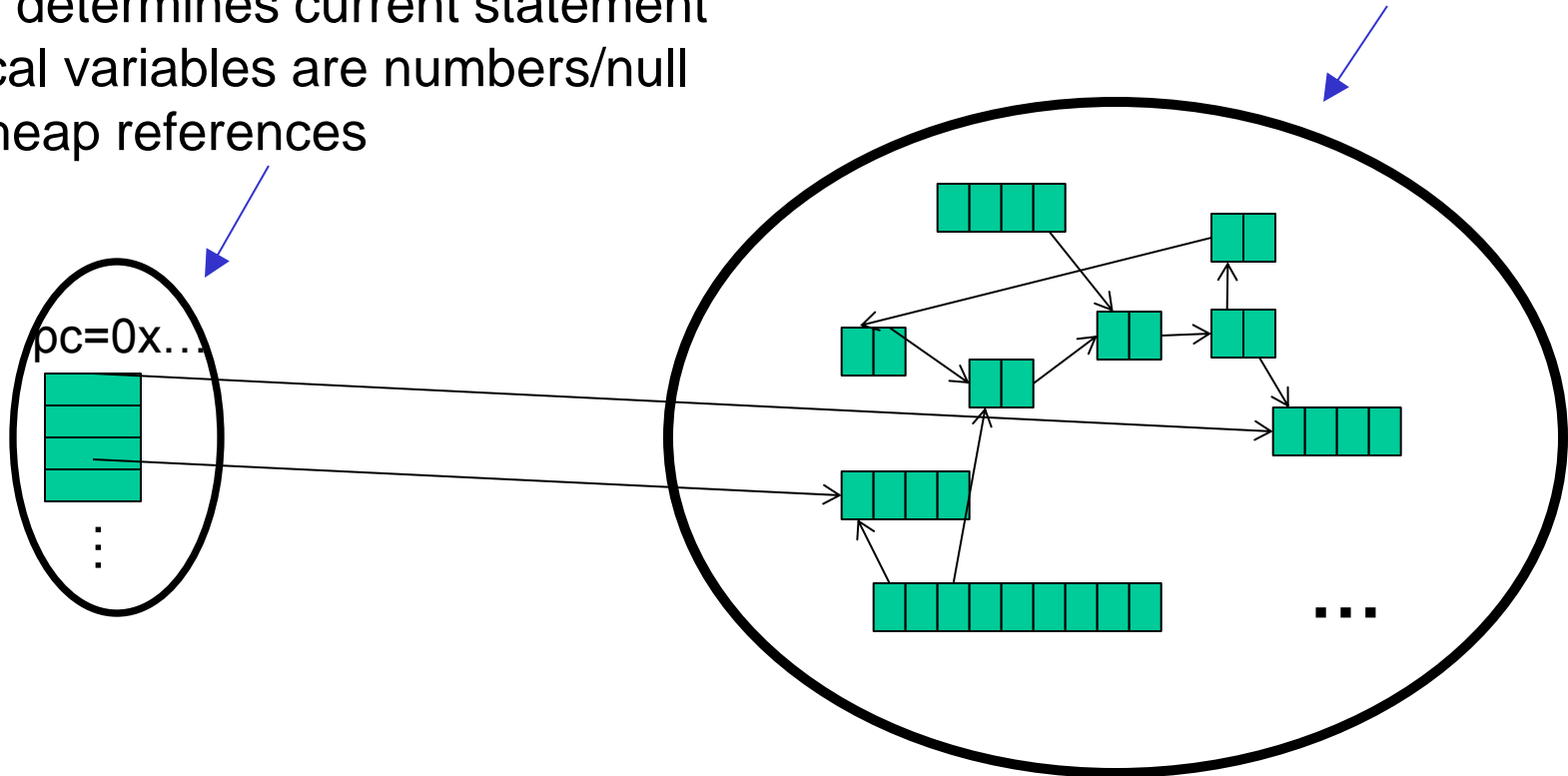
**New story:**

- 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

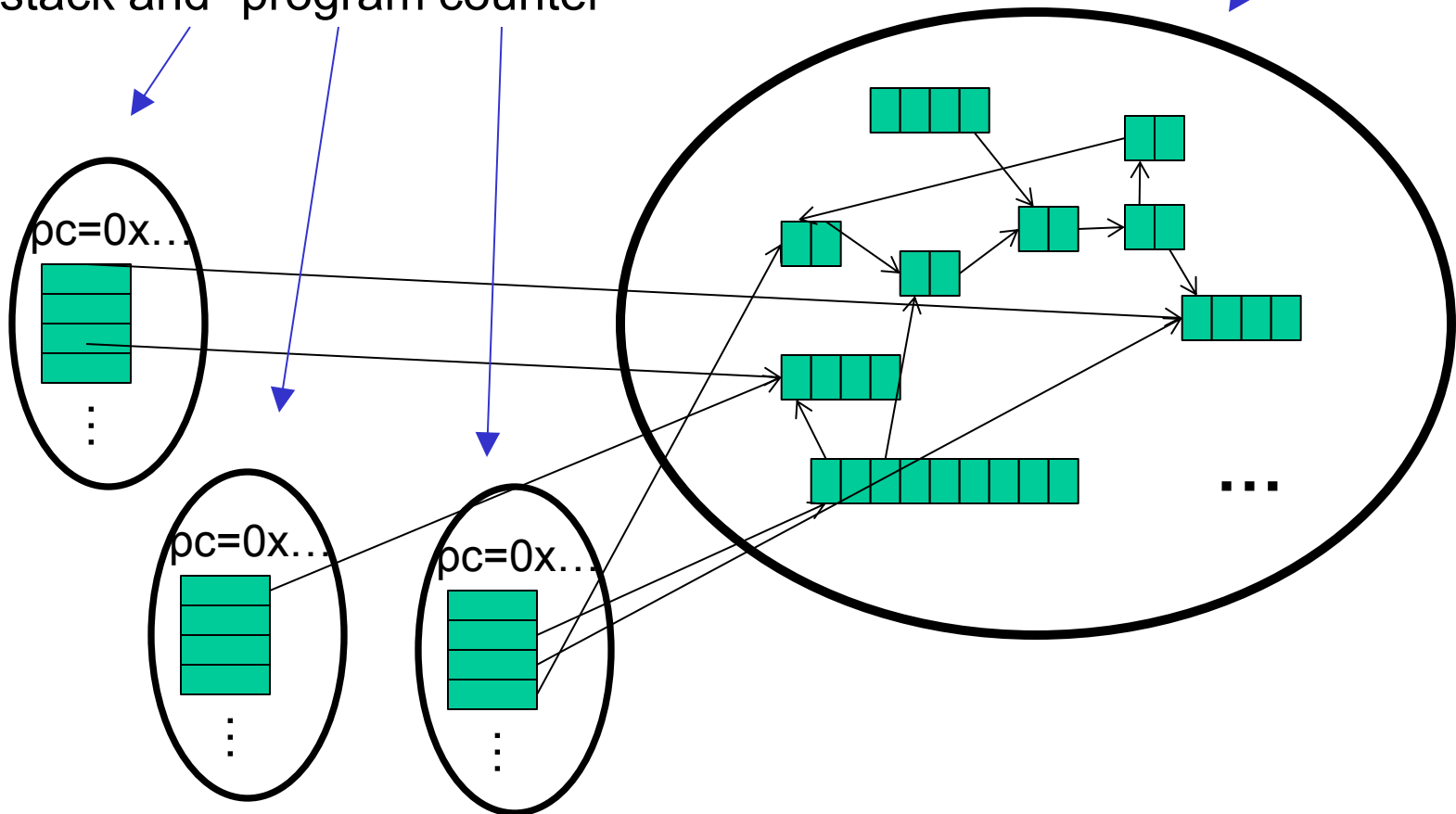
**Heap** for all objects and static fields



# 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

- **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

# *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:

1. Define a subclass `C` of `java.lang.Thread`, overriding `run`
2. Create an object of class `C`
3. Call that object's `start` method
  - `start` sets off a new thread, using `run` as its “main”

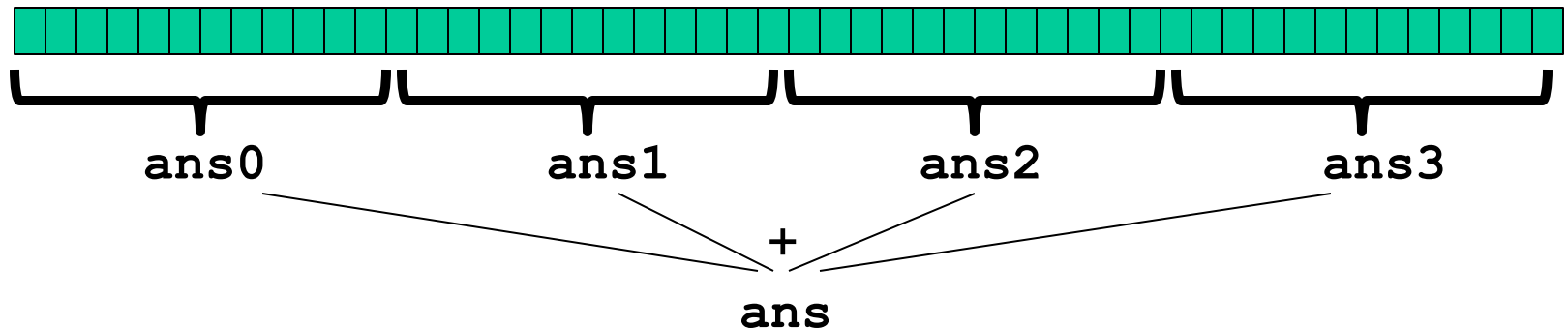
What if we instead called the `run` method of `C`?

- 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



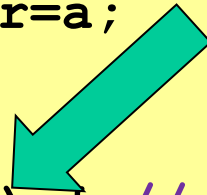
- Create 4 *thread objects*, each given a portion of the work
- Call `start()` on each thread object to actually *run* it in parallel
- *Wait* for threads to finish using `join()`
- Add together their 4 answers for the *final result*



# First attempt, part 1



```
class SumThread extends java.lang.Thread {  
  
    int lo; // fields, 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

## *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;
}
```

# *Join: Our “wait” method for Threads*

- The **Thread** class defines various methods you could not implement on your own
  - For example: **start**, which calls **run** in a new thread
- The **join** method is valuable for coordinating this kind of computation
  - Caller blocks until/unless the receiver is done executing (meaning the call to **run** finishes)
  - Else we would have a **race condition** on **ts[i].ans**
- This style of parallel programming is called “fork/join”
- Java detail: code has 1 compile error because **join** may throw **java.lang.InterruptedException**
  - In basic parallel code, should be fine to catch-and-exit

# *Shared memory?*

- Fork-join programs (thankfully) do not require much focus on sharing memory among threads
- But in languages like Java, there is memory being shared.  
In our example:
  - **lo**, **hi**, **arr** **fields** written by “main” thread, read by helper thread
  - **ans** **field** written by helper thread, read by “main” thread
- When using shared memory, you must avoid race conditions
  - While studying parallelism, we’ll stick with **join**
  - With concurrency, we will learn other ways to synchronize

# *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  $3 \times 2$  units of time

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



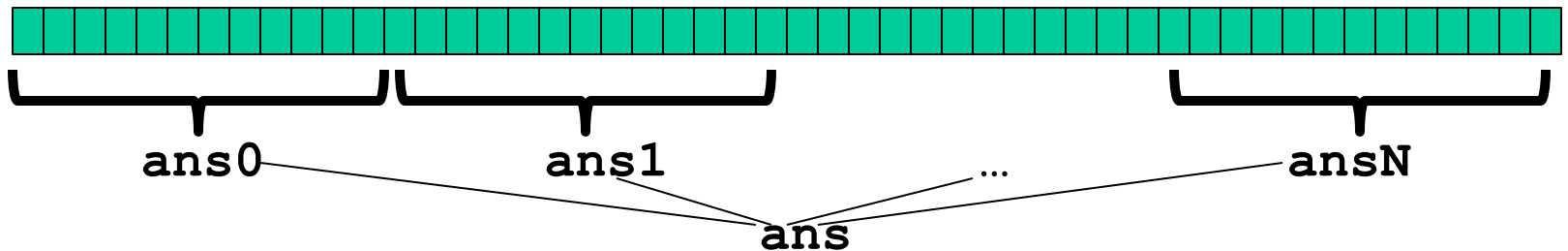
# *A Better Approach*

3. Though unlikely for `sum`, in general subproblems may take significantly different amounts of time
  - 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:** Lots of helpers each doing a small piece
2. **Processors available:** Hand out “work chunks” as you go
  - If 3 processors available and have 100 threads, then ignoring constant-factor overheads, extra time is < 3%
3. **Load imbalance:** No problem if slow thread scheduled early enough
  - Variation probably small anyway if pieces of work are small

# *Naïve algorithm is poor*

Suppose we create 1 thread to process every 1000 elements

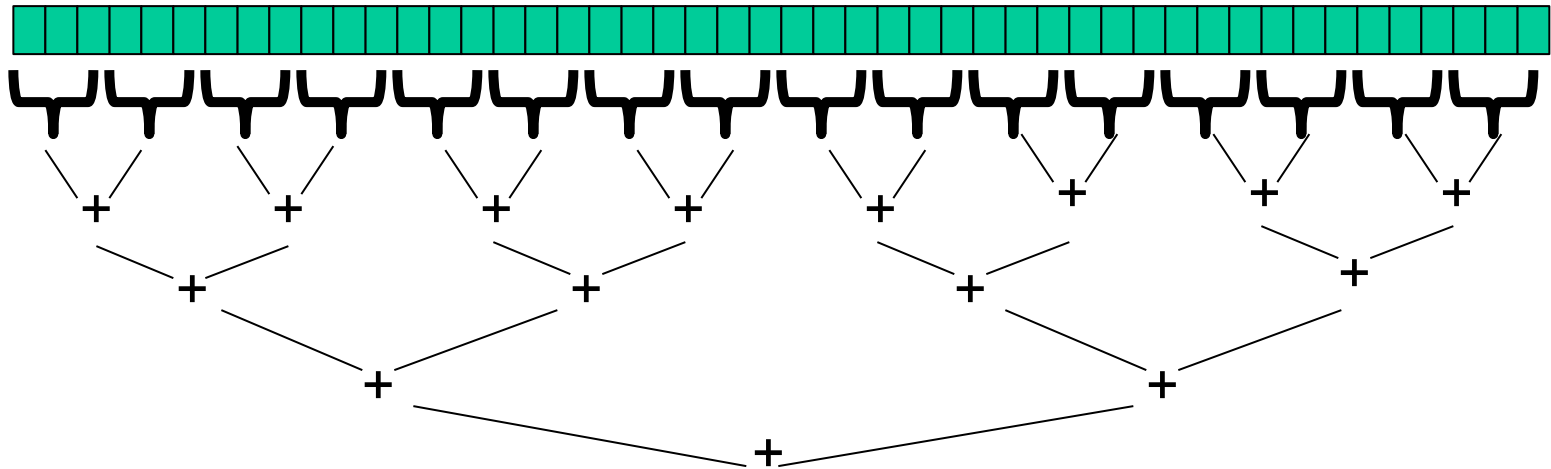
```
int sum(int[] arr) {  
    ...  
    int numThreads = arr.length / 1000;  
    SumThread[] ts = new SumThread[numThreads];  
    ...  
}
```

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)

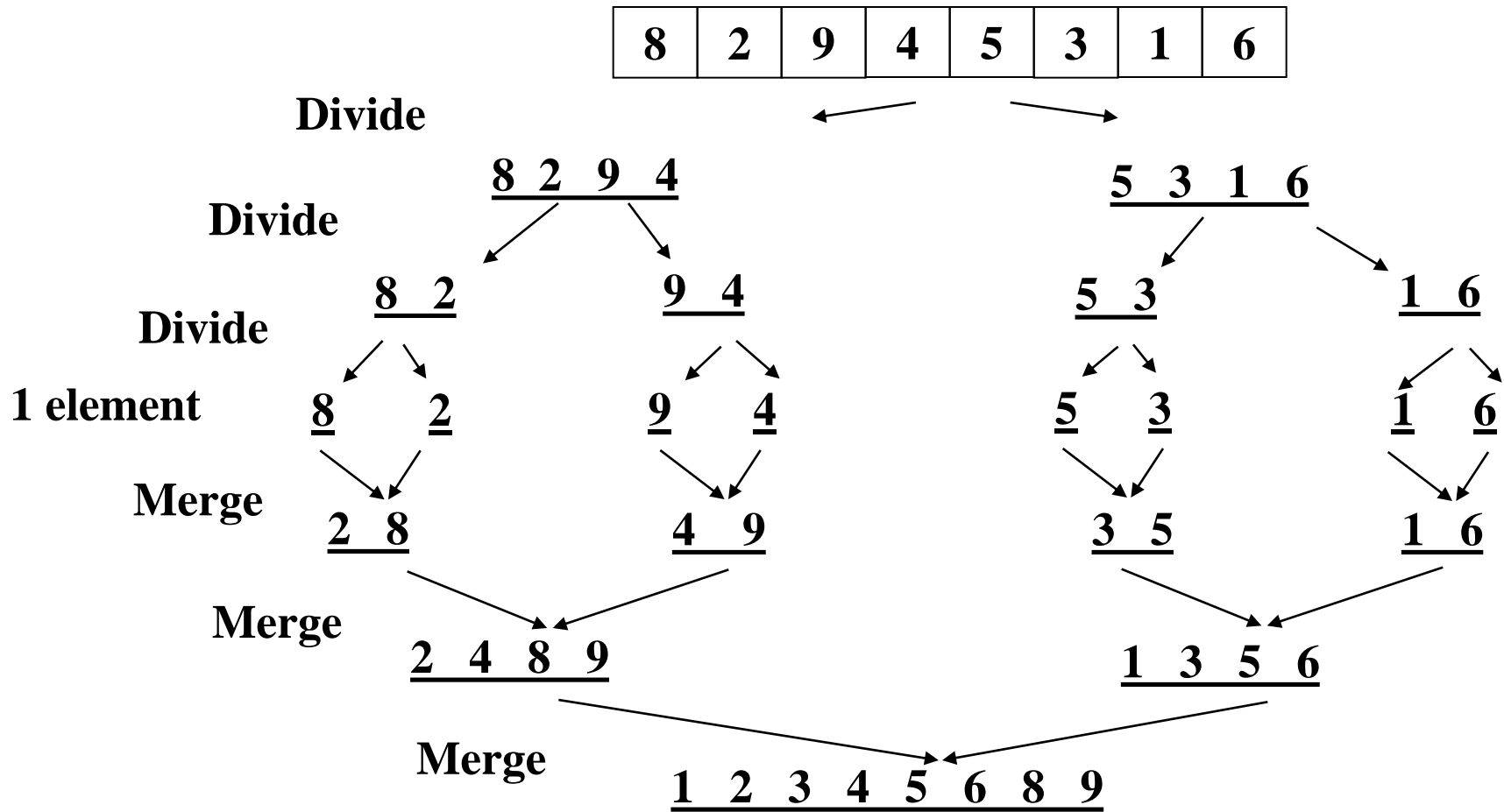


This will start small, and 'grow' threads to fit the problem

This is straightforward to implement using divide-and-conquer

- Parallelism for the recursive calls

# *Remember Mergesort?*



*Code looks something like this (still using Java Threads)*

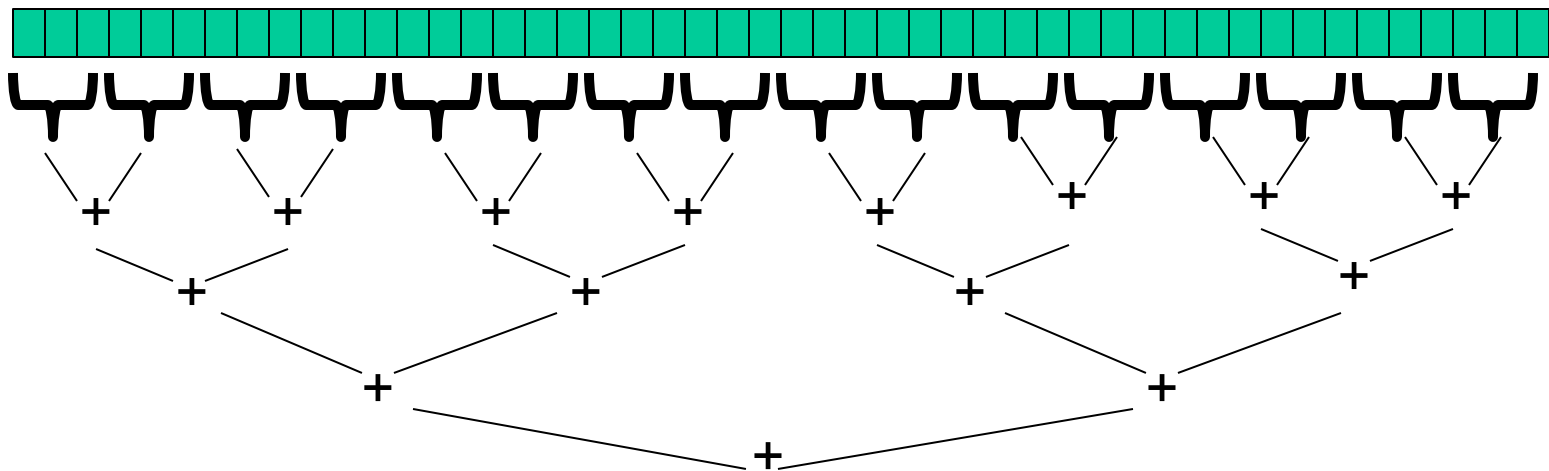
```
class SumThread extends java.lang.Thread {
    int lo; int hi; int[] arr; // fields to know what to do
    int ans = 0; // result
    SumThread(int[] a, int l, int h) { ... }
    public void run() { // override
        if(hi - lo < SEQUENTIAL CUTOFF)
            for(int i=lo; i < hi; i++)
                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;
}
```

11/05/2013

# Divide-and-conquer really works

- The key is divide-and-conquer parallelizes the result-combining
  - If you have enough processors, total time is **height of the tree:**  $O(\log n)$  (optimal, exponentially faster than sequential  $O(n)$ )
  - Next lecture: study reality of  $P \ll n$  processors
- Will write all our parallel algorithms in this style
  - But using a special library engineered for this style
    - Takes care of scheduling the computation well
  - Often relies on operations being associative (like +)



# 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

**Example: summing an array with 10 elements.**  
(too small to actually want to use parallelism)

**The algorithm produces the following tree of recursion, where the range [i,j) includes i and excludes j:**



# *Being realistic*

- In theory, you can divide down to single elements, do all your result-combining in parallel and get optimal speedup
  - Total time  $O(n/\text{numProcessors} + \log n)$
- In practice, creating all those threads and communicating swamps the savings, so do two things to help:
  1. Use a *sequential cutoff*, typically around 500-1000
    - Eliminates *almost all* the recursive thread creation (bottom levels of tree)
    - *Exactly* like quicksort switching to insertion sort for small subproblems, but more important here
  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 = ...

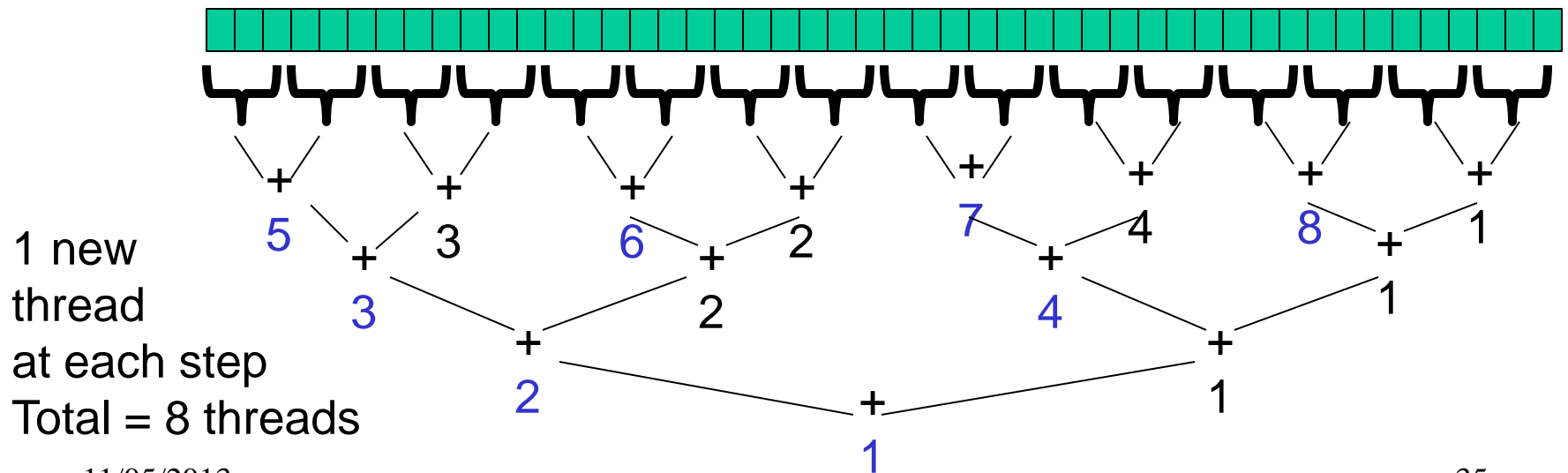
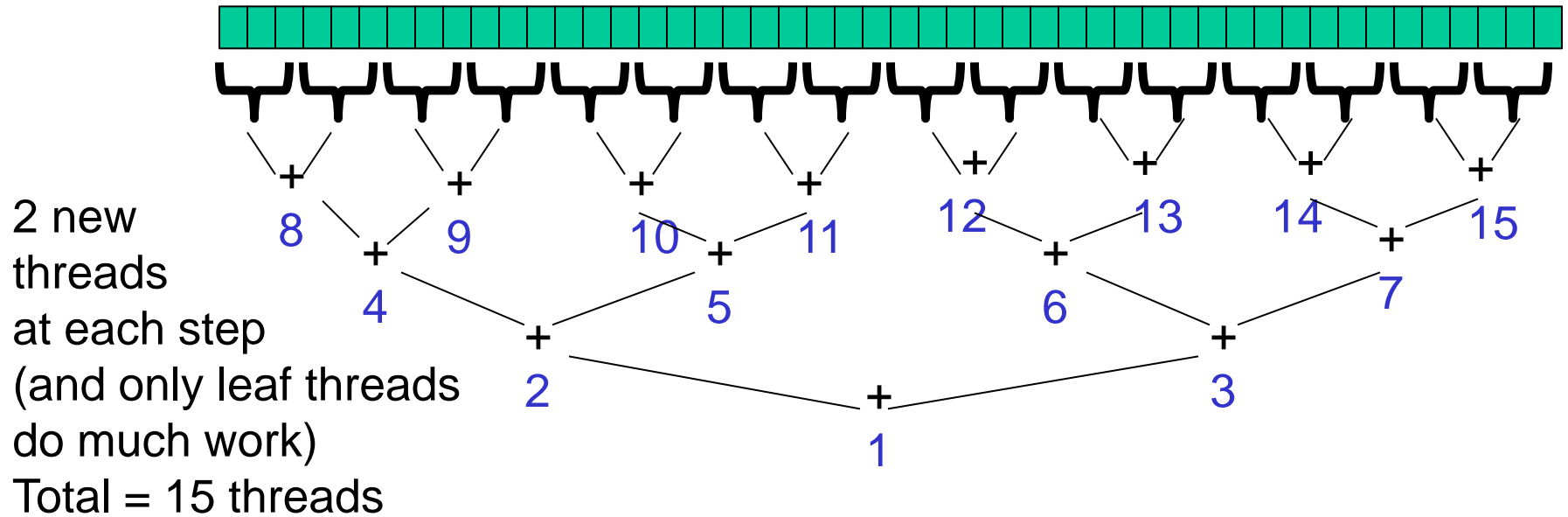
left.start();
right.run();

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*

- 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

# *Fewer threads pictorially*



# *That library, finally*

- 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
    - ...
  - 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**

See the web page for (linked in to project 3 description):

“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
}
```

# *Getting good results in practice*

- Sequential threshold
  - Library documentation recommends doing approximately 100-5000 basic operations in each “piece” of your algorithm
- Library needs to “warm up”
  - May see slow results before the Java virtual machine re-optimizes the library internals
  - Put your computations in a loop to see the “long-term benefit”
- Wait until your computer has more processors 😊
  - Seriously, overhead may dominate at 4 processors, but parallel programming is likely to become much more important
- Beware memory-hierarchy issues
  - Won’t focus on this, but often crucial for parallel performance