CONCURRENT OBJECT ORIENTED PROGRAMMING

Changing a major assumption

So far: *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

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?

- Today computers likely have several processors
  - 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

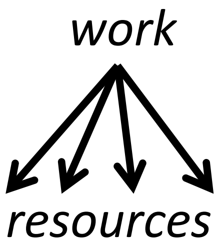
Parallelism vs. Concurrency

Note: Terms not yet standard but the perspective is essential

- Many programmers confuse these concepts

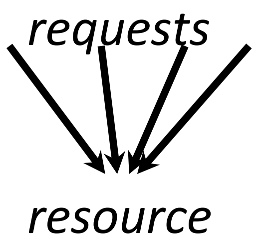
Parallelism:

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

A program is like a recipe for a cook

- One cook who does one thing at a time! (*Sequential*)

Parallelism:

- 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:

- 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

Our learning goal

- Understand concurreny and parallism as sources of complexity in software development
- Understand the common abstraction for parallelism and concurrency, and the trade-off among them
  - Explicit concurrency: Write thread-safe concurrent programs in Java
  - Know common thread-safe data structures, including high-level details of their implementation
  - Understand trade-offs between mutable and immutable data structures
  - Know common uses of concurrency in software design

Parallelism Example

Parallelism: Use extra computational resources to solve a problem faster (increasing throughput via simultaneous execution)

Pseudocode for array sum

- Bad style, but 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)

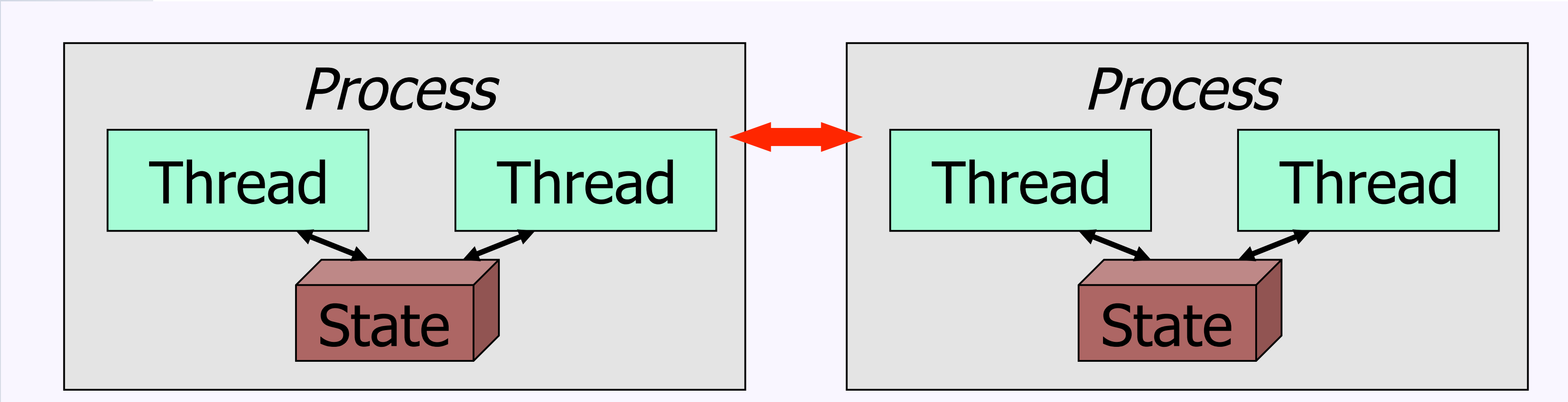
Pseudocode for a shared chaining hashtable

- Prevent *bad interleavings* (correctness)
- But allow some concurrent access (performance)

```
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)
   }
}
```

Programming abstractions

- Processes
  - Execution environment is isolated
    - ✓ Processor, in-memory state, files,
  - Inter-process communication via message passing
    Sockets, pipes, ...
- Threads
  - Execution environment is shared
  - Inter-thread communication typically fast, via shared state


Process
Thread
Thread
State
Process
Thread
Thread
State

Shared memory

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:

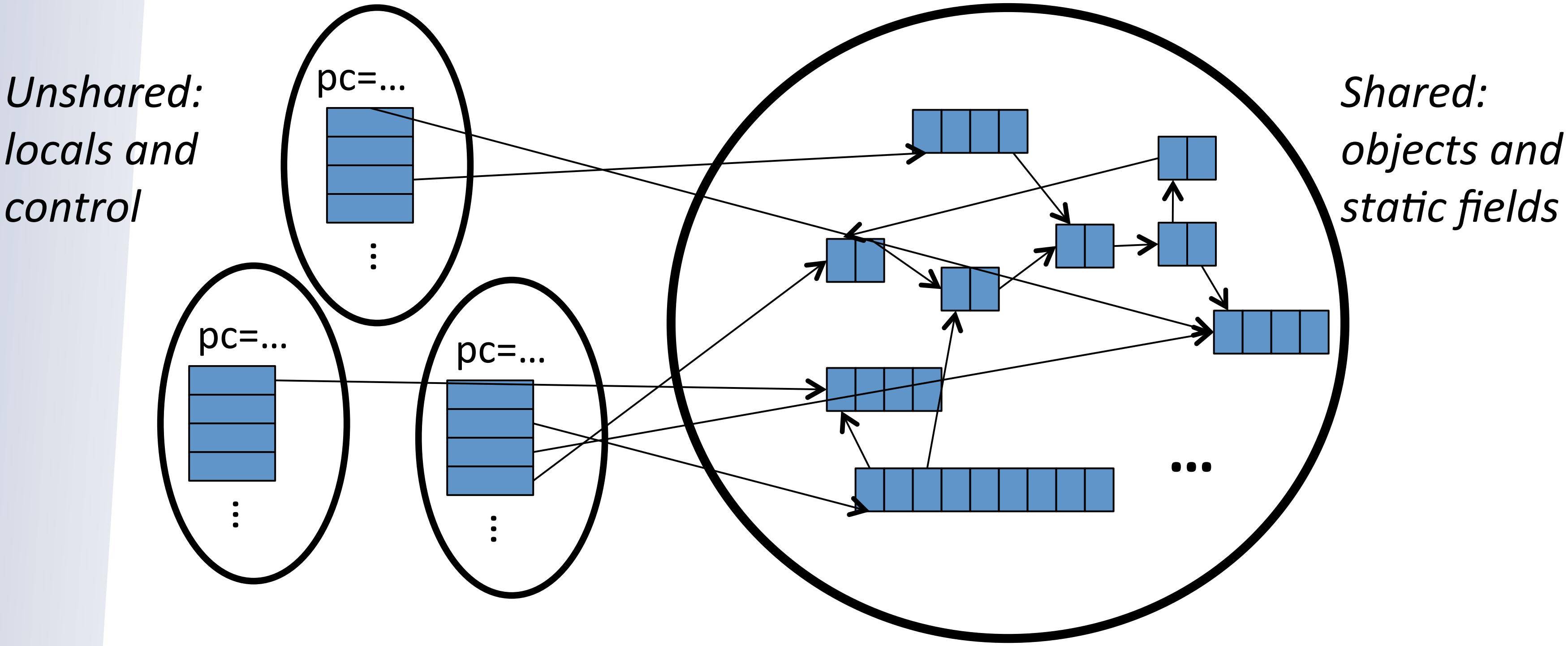
- A set of *threads*, each with its own program counter & call stack
  - ✓ No access to another thread's local variables
- Threads can (implicitly) share static fields / objects
  - ✓ To *communicate*, write somewhere another thread reads

Shared memory

Threads each have own unshared call stack and current statement

- (pc for “program counter”)
- local variables are numbers, **`null`**, or heap references

Any objects can be shared, but most are not



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`**

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…

Thread

```
public class HelloThread extends Thread {
    public void run() {
    System.out.println("Hello from a thread!");

    public static void main(String args[]) {
        (new HelloThread()).start(); }
}
```

Sleep

- **Thread.sleep** causes the current thread to suspend execution for a specified period.

Interrupts

- An ***interrupt*** is an indication to a thread that it should stop what it is doing and do something else.
- It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate.

Interrupts

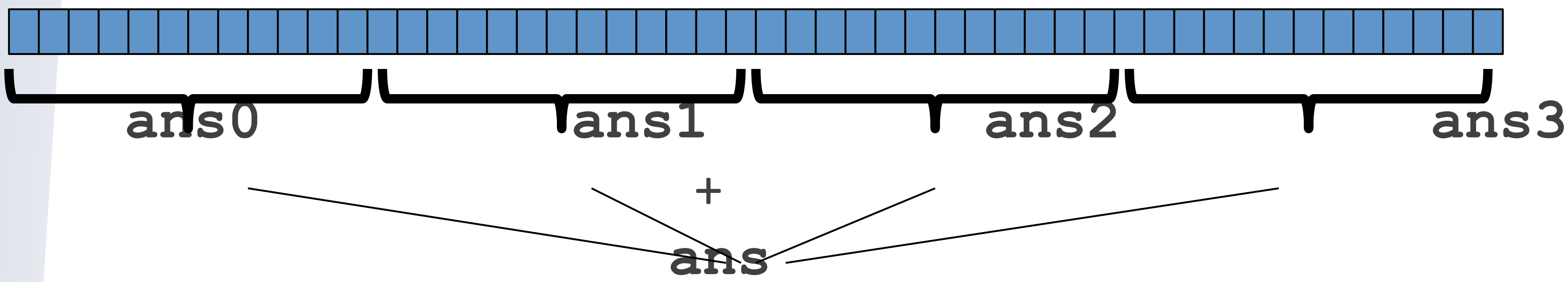
```
for (int i = 0; i < importantInfo.length; i++) {
    // Pause for 4 seconds
    try {
        Thread.sleep(4000);
    } catch (InterruptedException e) {
        // We've been interrupted: no more messages.
        return;
    }
    // Print a message
    System.out.println(importantInfo[i]);
}
```

join

- The join method allows one thread to wait for the completion of another.
- If t is a Thread object whose thread is currently executing,
  - **t.join()**;
- causes the current thread to pause execution until t's thread terminates.

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; // arguments
  int hi;
  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; // arguments
  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; // arguments
  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; // arguments
  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 (not the most descriptive word)

- 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`** returns)
  - 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 will 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*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

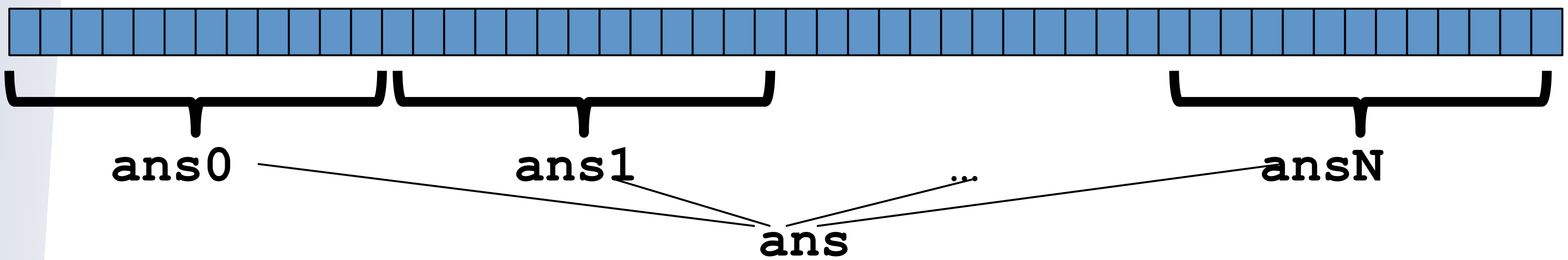
   - Example: Apply method `f` to every array element, but maybe `f` is much slower for some data items
     - ✓ Example: Is a large integer prime?

   - 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 use lots of threads, 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];
  …
}
```

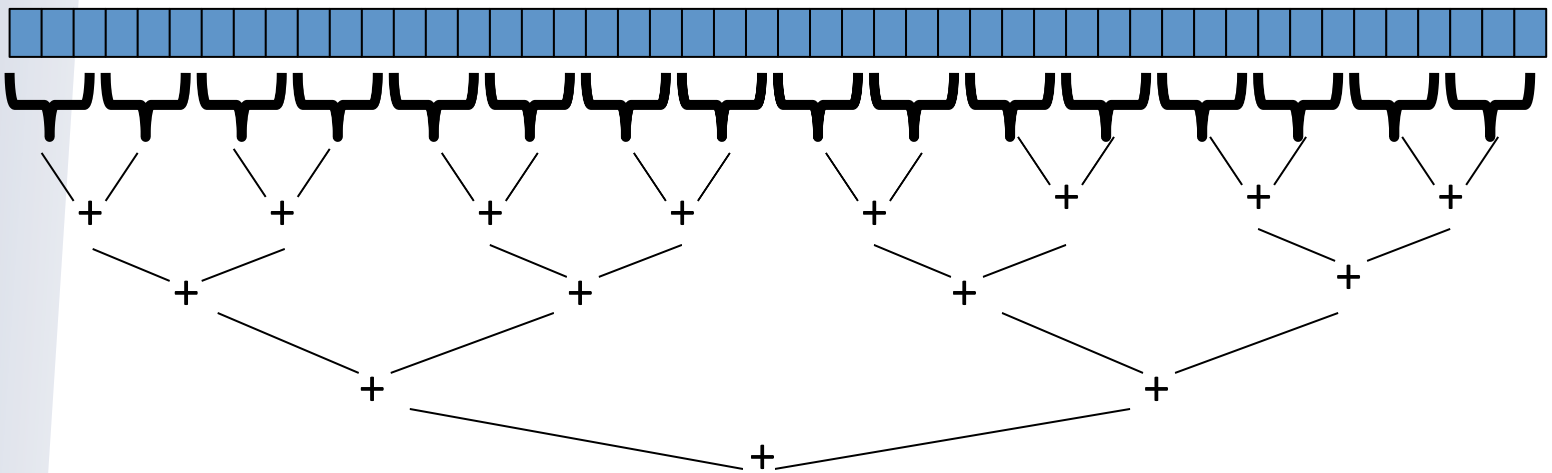
Then combining results will have **`arr.length / 1000`** additions

- Linear in size of array (with constant factor 1/1000)
- Previously we had only 4 pieces (constant in size of array)

In the extreme, if we create 1 thread for every 1 element, the loop to combine results has length-of-array iterations

- Just like the original sequential algorithm

A better idea



This is straightforward to implement using divide-and-conquer

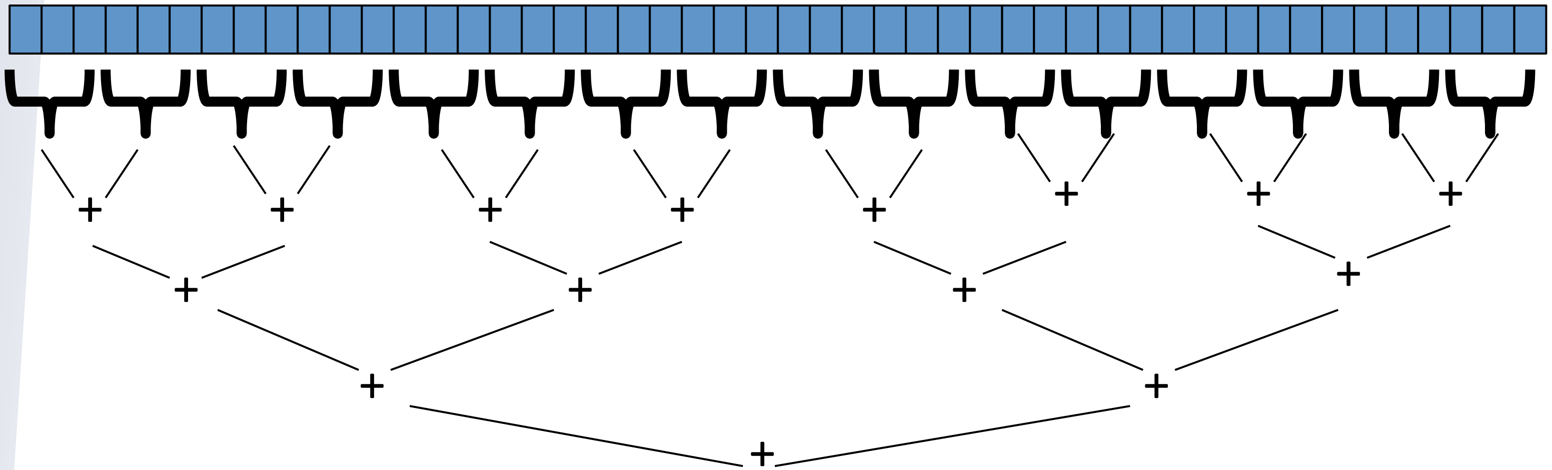
- Parallelism for the recursive calls

Divide-and-conquer to the rescue!

```
class SumThread extends java.lang.Thread {
  int lo; int hi; int[] arr; // arguments
  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){
   SumThread t = new SumThread(arr,0,arr.length);
   t.run();
   return t.ans;
}
```

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** << *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 +)



Being realistic

- In theory, you can divide down to single elements, do all your result-combining in parallel and get optimal speedup
  - Total time $O(n/numProcessors + \texttt{log}\ n)$
- In practice, creating all those threads and communicating swamps the savings, so:
  - 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
  - Do not create two recursive threads; create one and do the other "yourself"
    - ✓ Cuts the number of threads created by another 2x

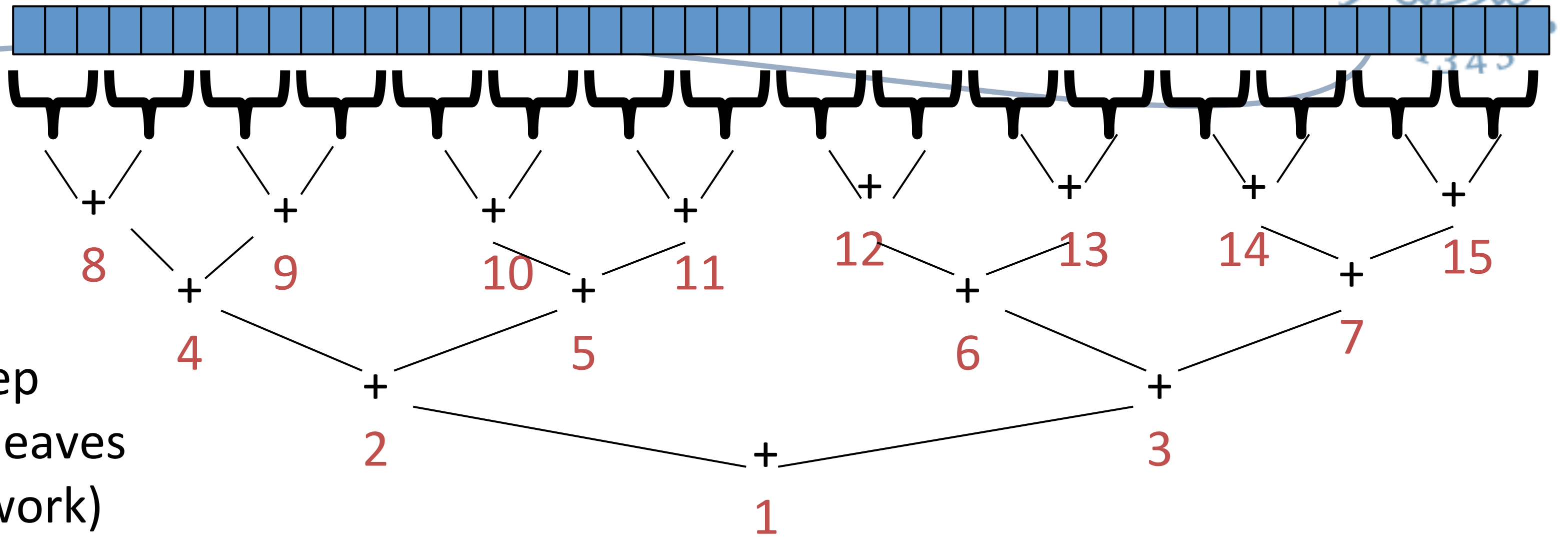
Half the threads

```
// wasteful: don't
SumThread left  = …
SumThread right = …
left.start();
right.start();
left.join();
right.join();
ans=left.ans+right.ans;
```

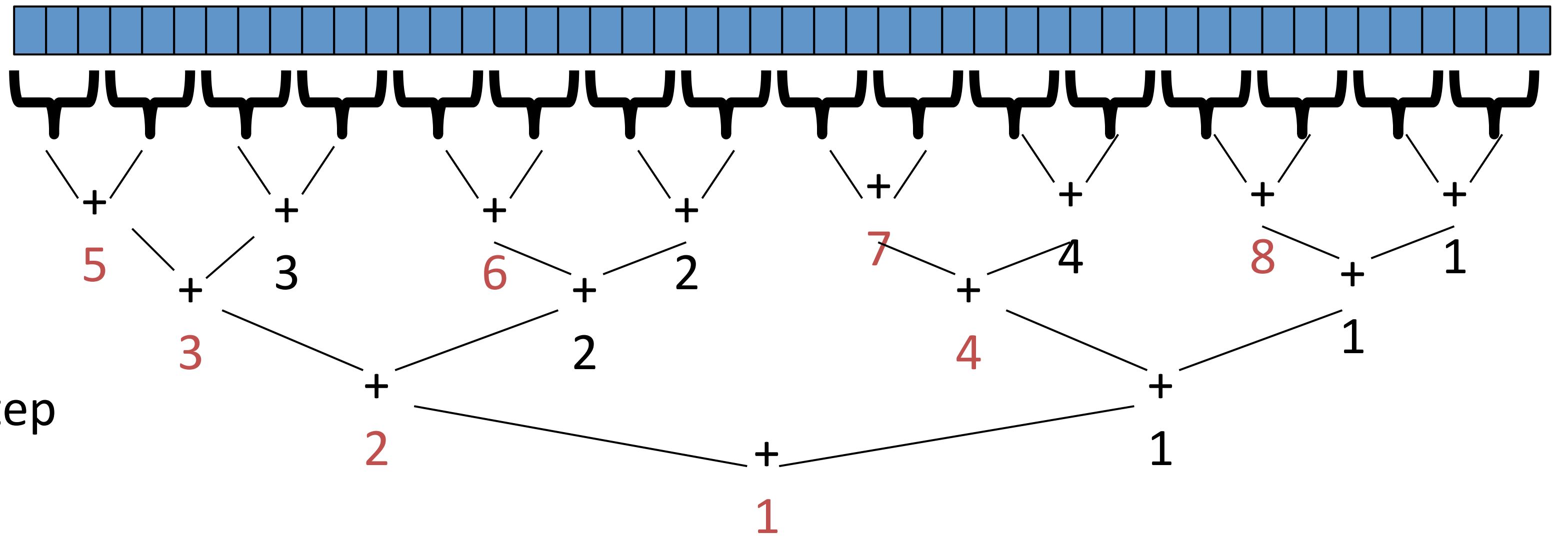
```
// better: do
SumThread left  = …
SumThread right = …
// order of next 4 lines
// essential – why?
left.start();
right.run();
left.join();
ans=left.ans+right.ans;
```

- If a *language* had built-in support for fork-join parallelism, we 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



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



1 new thread at each step

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`**)

Don't subclass **`Thread`** Do subclass **`RecursiveTask<V>`**

Don't override **`run`** Do override **`compute`**

Do not use an **`ans`** field Do return a **`V`** from **`compute`**

Don't call **`start`** Do call **`fork`**

Don't just call **`join`** Do call **`join`** which returns answer

Don't call **`run`** to hand-optimize Do call **`compute`** to hand-optimize

Don't have a topmost call to **`run`** Do create a pool and call **`invoke`**

Example: final version (missing imports)

```
class SumArray extends RecursiveTask<Integer> {
  int lo; int hi; int[] arr; // arguments
  SumArray(int[] a, int l, int h) { … }
  protected Integer compute(){// return answer
    if(hi – lo < SEQUENTIAL_CUTOFF) {
      int ans = 0;
      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();
      int rightAns = right.compute();
      int leftAns  = left.join();
      return leftAns + rightAns;
    }
  }
}
static final ForkJoinPool fjPool = new ForkJoinPool();
int sum(int[] arr){
  return fjPool.invoke(new SumArray(arr,0,arr.length));
}
```

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