Multithreading (Pretty) Early for Everyone: Parallelism & Concurrency in 2nd-Year Data-Structures

Dan Grossman
University of Washington
SIGCSE 2011, Workshop 19

http://www.cs.washington.edu/homes/djg/teachingMaterials/

Executive Summary

Ready-to-use parallelism/concurrency in data-structures

- 2.5-week unit with reading notes, slides, homeworks, a Java project, sample exam questions
- 1st taught at Washington Spring 2010
  - Taught every term; 4 different instructors so far
- If you can teach balanced trees and graph algorithms, then you can teach this

Valuable approach and place-in-curriculum for an introduction

- Programmer's view (not the OS or HW implementation)
- Focus on shared memory
- Basic parallel algorithms and analysis
- Basic synchronization and mutual exclusion

Different audiences

Persona #1: I hear multicore is important, but I'm skeptical I can do something meaningful and low-maintenance in a low-level course. And would my colleagues go along?

I had you in mind from Day 1

- The concepts have to be timeless and straightforward
- 3 weeks maximum, as part of an existing course
- No fancy hardware / software assumed
- Free, modifiable course materials
- Not advocating a revolution

Naturally, adapt material to personal style, local circumstances

Different audiences

Persona #2: Multicore is everything. We need to revamp the entire curriculum. All courses need to assume parallel throughout.

To avoid unhappiness, remember:

- The choir never understands why the pews aren't more full
- This is an introduction, easy to append more
  - "This is important" not "Other stuff isn't"
- Essential foundations *before* an upper-level course on parallelism, OS, networking, graphics, etc.
- Material required in 1st 2 years is a zero-sum game
  - I wouldn't cut more other stuff from our curriculum

Tonight: A whirlwind tour!

- Context: What I mean by “in data structures”
- Introductions: Name, rank, and serial number ☺, plus
  - 1-3 terms, concepts, ideas related to parallelism/concurrency

- Distinguishing parallelism and concurrency

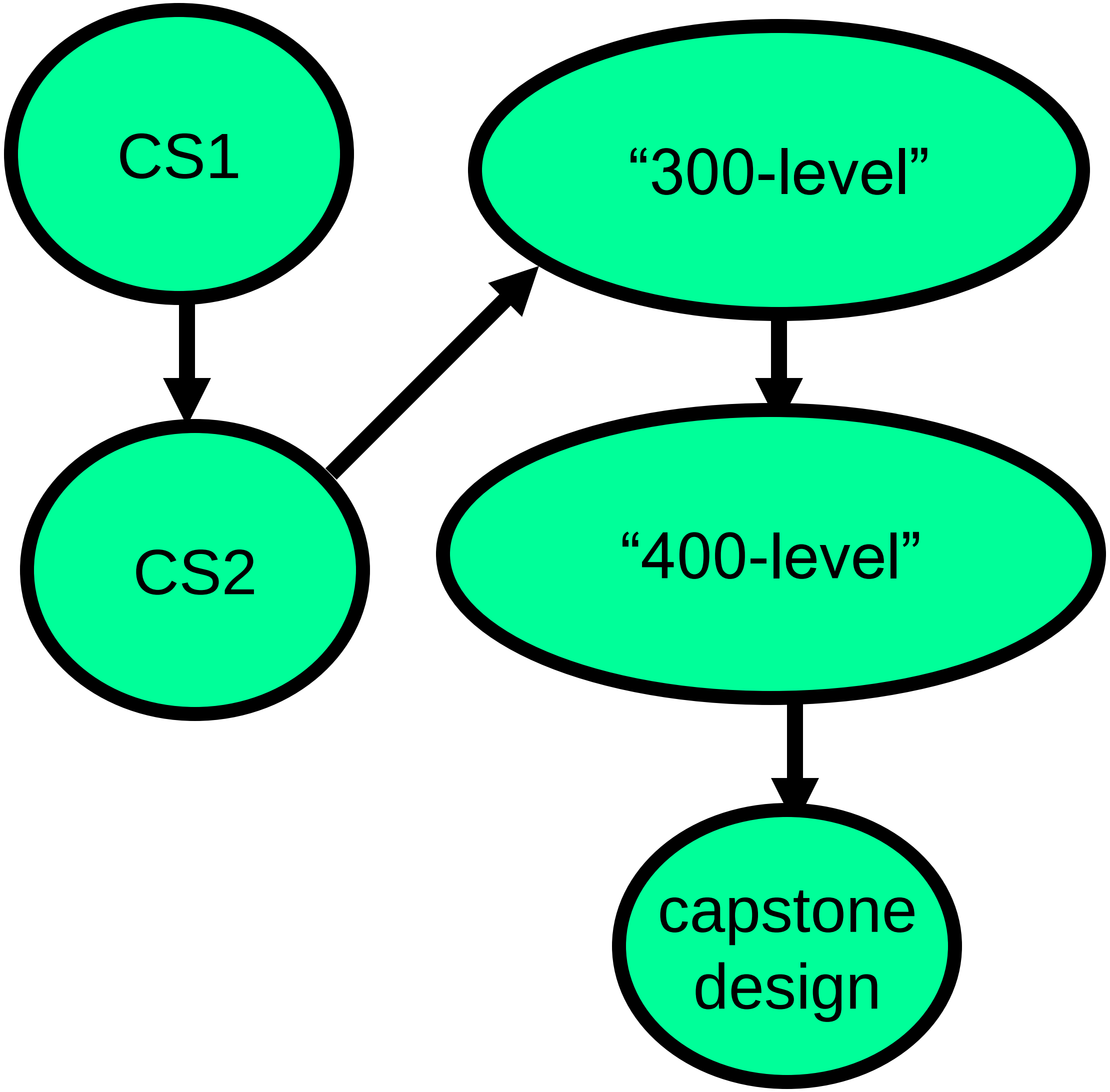
- Parallelism with Java’s ForkJoin Framework – and try it out

- Asymptotic analysis of parallel algorithms
- Fancier parallel algorithms

- Synchronization and mutual exclusion
  - Locks, programming guidelines, memory-consistency models, condition variables, …

- Review: The *N* main concepts & why they fit in data structures

Why the 300-level?



CS1+2:

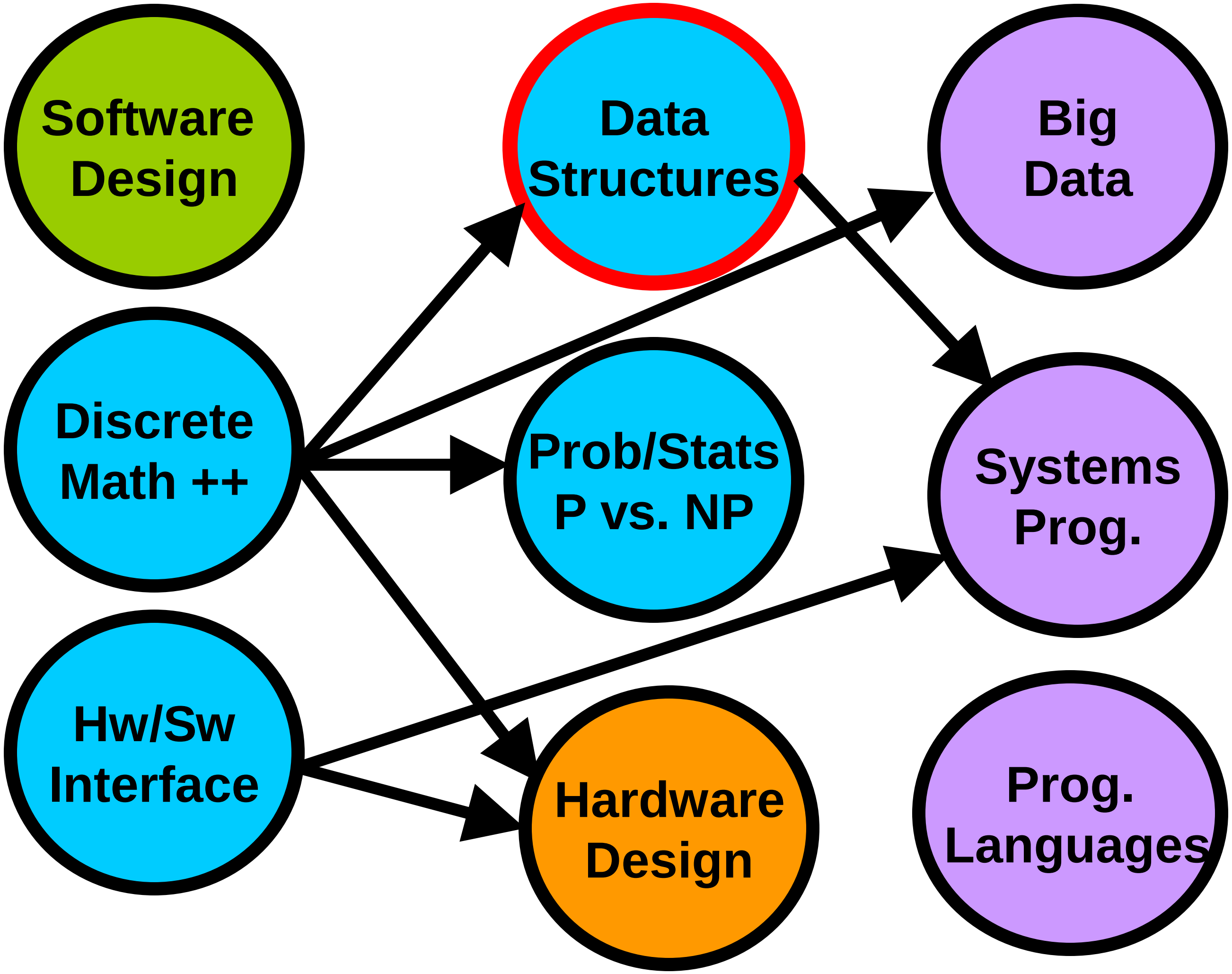
- Loops, recursion, objects, trees
- < 25% CS majors
- Late CS2 maybe

Senior year:

- Too late
- Too specialized
- Too redundant
  - Rely on concepts throughout

UW's 300-level (10-week quarters)

🟦 **required**
🟩 **CS required**
🟧 **CompE required**
🟪 **recommended**



(Note: reality slightly more complex)

Data Structures: Old vs. New

Old and new: 20 lectures

Big-Oh, Algorithm Analysis

Binary Heaps (Priority Qs)

AVL Trees

B Trees

Hashing

Sorting

Graph Traversals

Topological Sort

Shortest Paths

Minimum Spanning Trees

Amortization

Data Structures: Old vs. New

Old and new: 20 lectures

Big-Oh, Algorithm Analysis
Binary Heaps (Priority Qs)
AVL Trees
B Trees
Hashing
Sorting
Graph Traversals
Topological Sort
Shortest Paths
Minimum Spanning Trees
Amortization

Removed: 7-8 lectures

D-heaps
Leftist heaps
Skew heaps
Binomial queues
Network flow
Splay trees ☹
Disjoint sets ☹
Hack job on NP (moves elsewhere)

Introductions

- Introductions: Name, rank, and serial number ☺, plus
  - 1-2 terms, concepts, ideas related to parallelism/concurrency

I'll go first:

“locks”

“speedup”

Tonight

- Context: What I mean by “in data structures”
- Introductions: Name, rank, and serial number ☺, plus
  - 1-2 terms, concepts, ideas related to parallelism/concurrency

- Distinguishing parallelism and concurrency

- Parallelism with Java’s ForkJoin Framework – and try it out

- Asymptotic analysis of parallel algorithms
- Fancier parallel algorithms

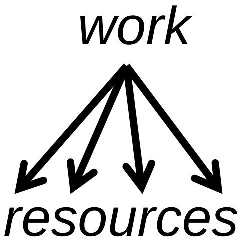
- Synchronization and mutual exclusion
  - Locks, programming guidelines, memory-consistency models, condition variables, …

- Review: The *N* main concepts & why they fit in data structures

A key distinction

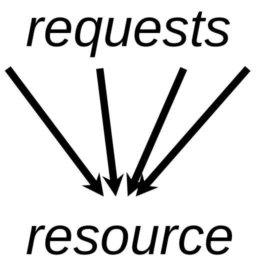
Parallelism:

Use extra computational resources to solve a problem faster



Concurrency:

Correctly and efficiently manage access to shared resources



Note: Terms not standard, but becoming more so

- Distinction is paramount

An analogy

CS1: 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 the burners, but not cause spills or incorrect burner settings

Parallelism Example

Parallelism:

Use extra computational resources to solve a problem faster

Pseudocode for array sum

```
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

Pseudocode for a shared chaining hashtable

- Prevent *bad interleavings* but allow some concurrent access

```
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 arr[bucket];
   }
   V lookup(K key) {
      (like insert, but can allow concurrent
       lookups to same bucket)
   }
}
```

Activity

For each introduction term, pick one:

A. (Almost all) about parallelism
B. (Almost all) about concurrency
C. Equally related to both
D. Unsure

Why parallelism first

- Structured, shared-nothing parallelism is easier to reason about
  - Synchronization is easy
  - Race conditions just don’t show up much
  - Focus on algorithms
- *After comfortable with threads*, deal with mutual exclusion, interleavings, etc.
  - Focus on thread-safe APIs rather than algorithms
- Yes, in reality, parallelism and concurrency co-mingle
  - In a 2nd-year course, emphasize the difference
  - Many separate curriculum topics co-mingle in practice

A Programming Model

To write parallel programs, need a way for threads (broadly construed) to *communicate* and *coordinate*

Approaches I barely mention – a full course would cover them

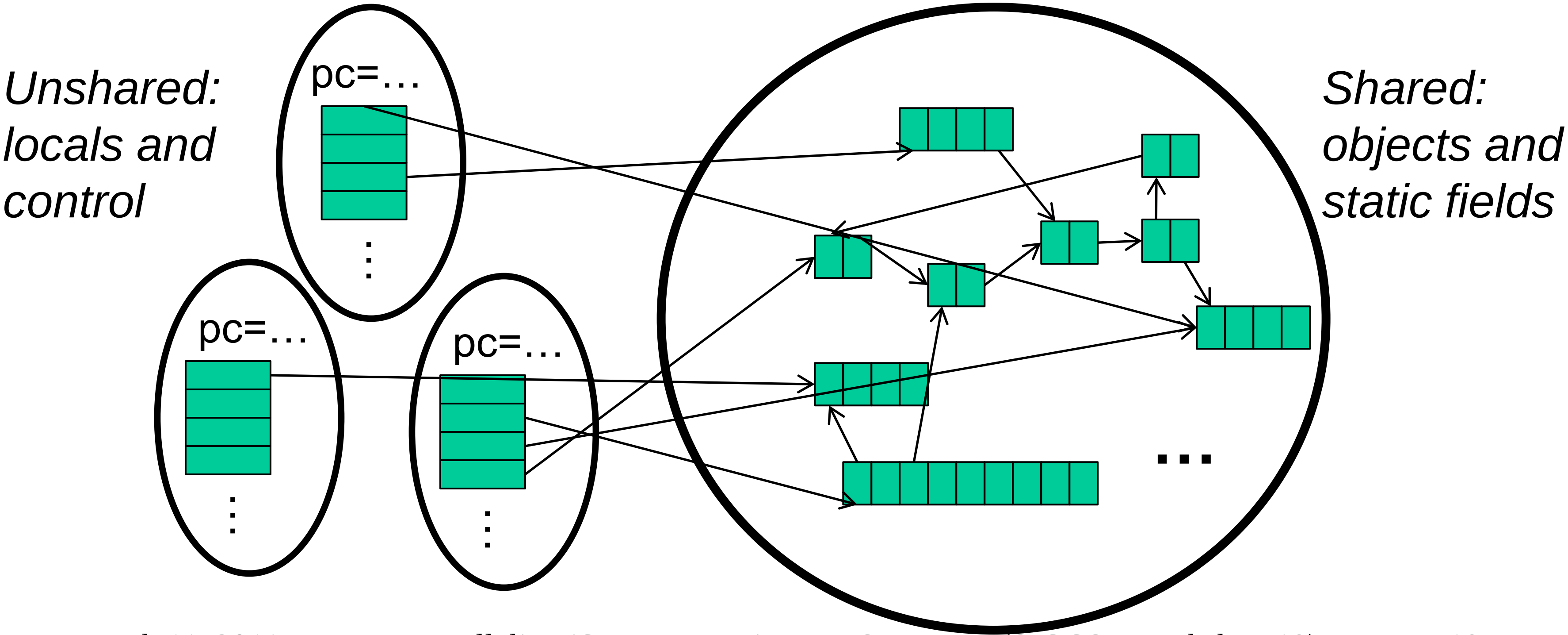
- Message-passing: Each thread has its own collection of objects. Communication via explicitly sending/receiving messages
  - Cooks working in separate kitchens, mail around ingredients
- Dataflow: Programmers write programs in terms of a DAG. A node executes after all of its predecessors in the graph
  - Cooks wait to be handed results of previous steps
- Data parallelism: Primitives for things like “apply function to every element of an array in parallel”

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



Why just shared memory

- 1 model enough for 3-week introduction
  - Could add more given more time

- Previous slide is all students need to “get it”

- Fits best with rest of course
  - Asymptotics, trees, hashtables, etc.

- Fits best with Java

Note: Not claiming it’s the best model

Our needs

A way to:

- Create threads
- Share objects among threads
- Coordinate: threads wait for each other to finish something

In class: I show Java threads (**`java.lang.Thread`**) and then why they are less than ideal for parallel programming

- If create 10,000 at once, JVM won't handle it well

Tonight: To save time, skip to ForkJoin *tasks*

- A Java 7 library available for Java 6
- Similar libraries available for C++, C#, …
- Use "real" Java threads for concurrency (later)

Tonight

- Context: What I mean by “in data structures”
- Introductions

- Distinguishing parallelism and concurrency

- Parallelism with Java’s ForkJoin Framework – and try it out

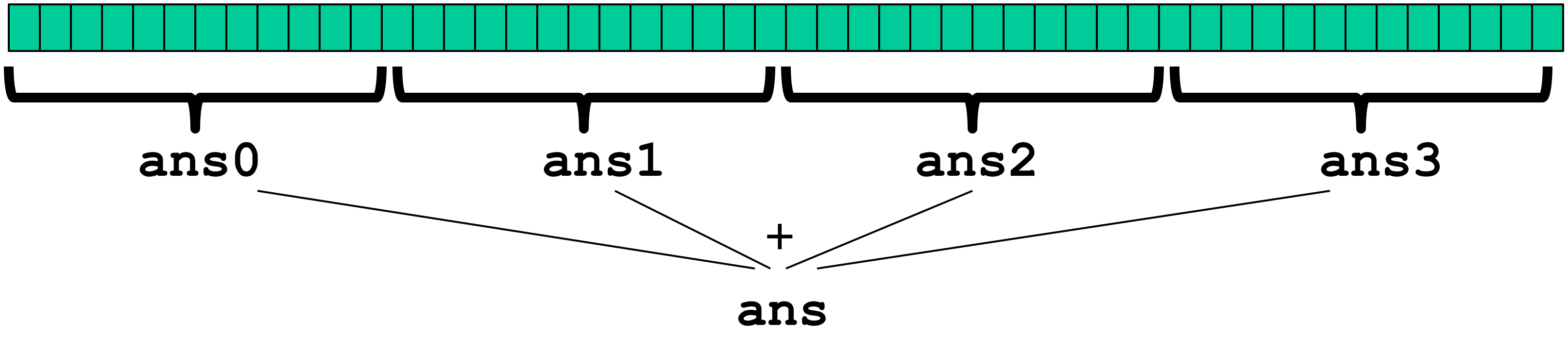
- Asymptotic analysis of parallel algorithms
- Fancier parallel algorithms

- Synchronization and mutual exclusion
  - Locks, programming guidelines, memory-consistency models, condition variables, …

- Review: The *N* main concepts & why they fit in data structures

Canonical example: array sum

- Sum elements of a large array
- Idea: Have 4 simultaneous tasks each sum 1/4 the array
  - Warning: Inferior first approach



  - Create 4 *special objects*, assigned a portion of the work
  - Call **`fork()`** on each object to actually *run* it in parallel
  - *Wait* for each object to finish using **`join()`**
  - Sum 4 answers for the *final result*

First attempt, part 1

```
class SumThread extends RecursiveAction {

  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 compute(){//override must have this type
    for(int i=lo; i < hi; i++)
      ans += arr[i];
  }
}
```

First attempt, continued (wrong!)

```
class SumThread extends RecursiveAction {
  int lo, int hi, int[] arr; // arguments
  int ans = 0; // result
  SumThread(int[] a, int l, int h) { … }
  public void compute(){ … }
}
```

```
int sum(int[] arr){
  SumThread[] ts = new SumThread[4];

  int len = arr.length; // do parallel computations
  for(int i=0; i < 4; i++){
    ts[i] = new SumThread(arr, i*len/4, (i+1)*len/4);
    ts[i].fork(); // fork not compute
  }

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

2nd attempt: almost right (but still inferior)

```
class SumThread extends RecursiveAction {
  int lo, int hi, int[] arr; // arguments
  int ans = 0; // result
  SumThread(int[] a, int l, int h) { … }
  public void compute(){ … }
}
```

```
int sum(int[] arr){
  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].fork(); // fork not compute
  }
  for(int i=0; i < 4; i++) { // combine results
    ts[i].join(); // wait for helper to finish!
    ans += ts[i].ans;
  }
  return ans;
}
```

The primitives

Needed "magic" library for things we can't implement ourselves:

- **`fork`** method of **`RecursiveAction`** calls **`compute()`** in a new thread/task
  - Calling **`compute`** directly is a plain-old method call
- **`join`** method of **`RecursiveAction`** blocks its caller until/unless the receiver is done executing (its **`compute`** returns)
  - *Must* wait to read the **`ans`** field
- Example so far is "right in spirit"
  - But doesn't enter the library correctly (won't work yet)
    - Fix after learning better approach

Shared memory?

- Fork-join programs (thankfully) don't require much focus on sharing memory among threads

- Memory *is* shared
  - **`lo`**, **`hi`**, **`arr`** fields written by "main" thread, read by helpers
  - **`ans`** field written by helpers, read by "main" thread

- Must avoid *data races*
  - For this kind of parallelism, **`join`** suffices
  - For concurrency, learn about locks

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 numThreads){
  SumThread[] ts = new SumThread[numThreads];
  int subLen = arr.length / numThreads;
  …
}
```

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 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`**

A better approach

3. Though unlikely for **`sum`**, in general different 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

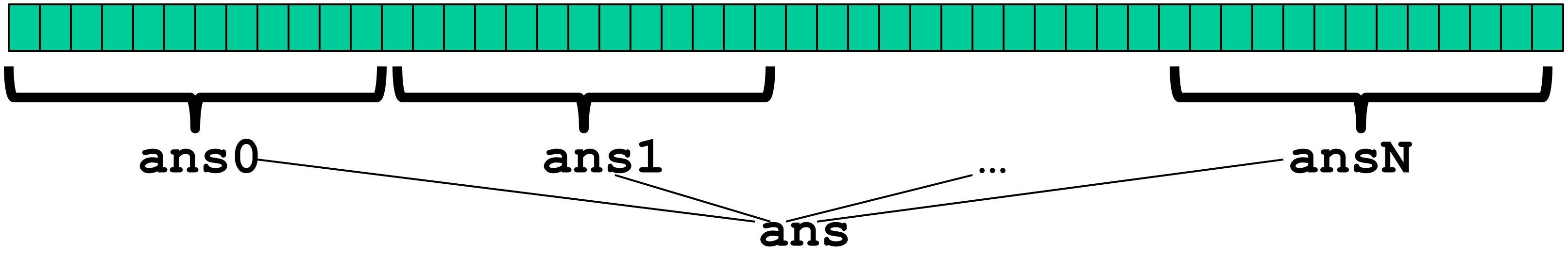
     - Example: Is a large integer prime?

   – Leads to load imbalance

A Better Approach

The counterintuitive(?) solution to all these problems is to use lots of tasks, far more than the number of processors

- But will require changing our algorithm



1. Forward-portable: Lots of helpers each doing a small piece
2. Processors available: Hand out “work chunks” as you go
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

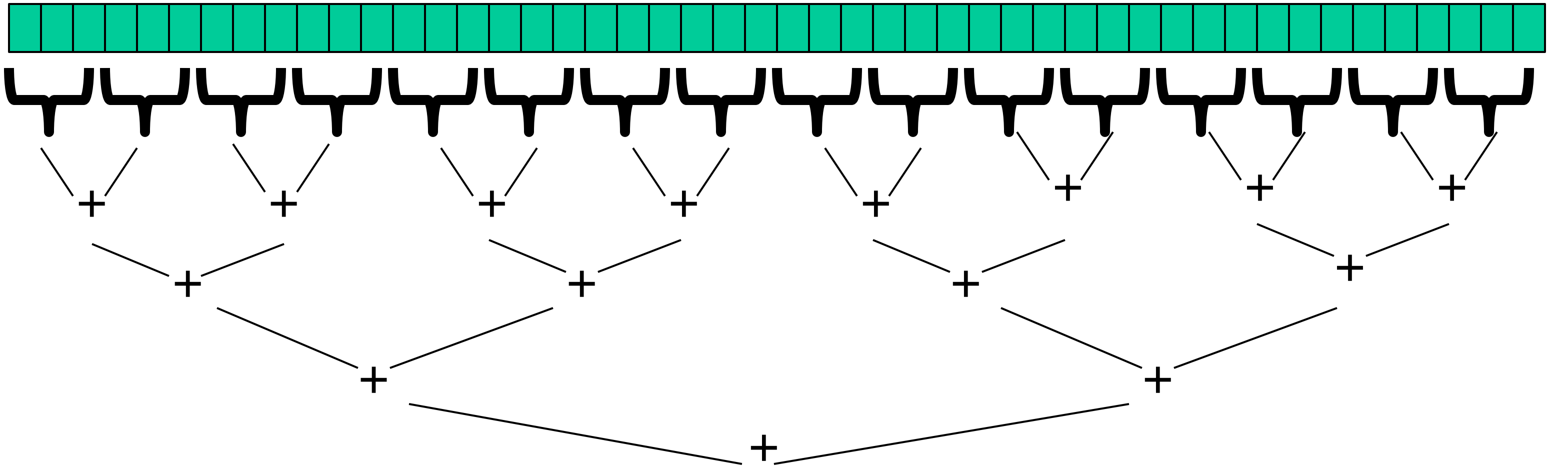
Suppose we create 1 thread to process every 1000 elements

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

Then combining results will have **`arr.length / 1000`** additions to do – still linear in size of array

In fact, if we create 1 thread for every 1 element, we recreate a sequential algorithm

A better idea



Straightforward to implement using divide-and-conquer

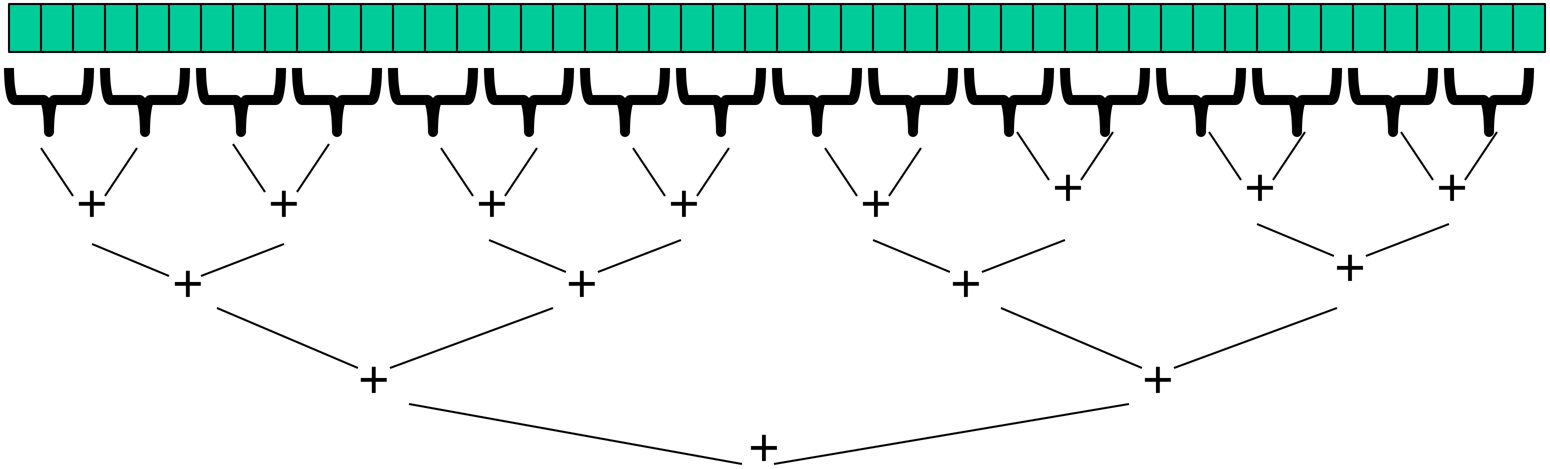
- Parallelism for the recursive calls
- Will write all our parallel algorithms in this style
- Asymptotic exponential speedup “with enough processors”

Divide-and-conquer to the rescue!

```
class SumThread extends RecursiveAction {
  int lo; int hi; int[] arr; // arguments
  int ans = 0; // result
  SumThread(int[] a, int l, int h) { … }

  public void compute(){
    if(hi - lo < SEQUENTIAL_CUTOFF) // around 1000
      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.fork();
      right.fork();
      left.join(); // don’t move this up a line - why?
      right.join();
      ans = left.ans + right.ans;
    }
  }
}
```

Sequential cut-offs

- Cutting off last 10 levels of recursion saves > 99% of task-creation overhead
- *Exactly like* having quicksort switch to insertion sort for small subproblems!



Finishing the story

Need to start the recursion for the entire array

- Slightly awkward boilerplate to “enter the library”
- Can’t just call `compute` directly ☹

```
static final ForkJoinPool fjPool = new ForkJoinPool();

static int sum(int[] arr){
 return fjPool.invoke(new SumThread(arr,0,arr.length));
}
```

- Create 1 pool for whole program
- Start recursion by passing `invoke` an object
  - `invoke` calls the object’s `compute` and returns the result

(I use recitation section to go over this stuff)

Improving our example

Two final changes to our example:

- For *style*, instead of an **`ans`** field:
  - Subclass **`RecursiveTask<Ans>`** (e.g., **`Integer`**)
  - **`compute`** method now *returns* an **`Ans`** (e.g., **`Integer`**)
  - **`join`** returns what task's **`compute`** returns

- For *performance*, don't have each task do nothing but create two other tasks and add results
  - Create one other task and do the other half *yourself*
  - Makes a surprisingly large difference

Final version

```
class SumThread extends RecursiveTask<Integer> {
  int lo; int hi; int[] arr; // arguments
  SumThread(int[] a, int l, int h) { … }
  public Integer compute(){
    if(hi - lo < SEQUENTIAL_CUTOFF)
      int ans = 0;
      for(int i=lo; i < hi; i++)
        ans += arr[i];
      return ans;
    } else {
      SumThread left = new SumThread(arr,lo,(hi+lo)/2);
      SumThread right= new SumThread(arr,(hi+lo)/2,hi);
      left.fork();
      int rightAns = right.compute();
      int leftAns  = left.join(); // don't move up!
      return leftAns + rightAns;
    }
  }
 static int sum(int[] arr){
  return fjPool.invoke(new SumThread(arr,0,arr.length));
 }
}
```

Reductions and Maps

- Array-sum is a reduction
  - Single answer from collection via associative operator
  - (max, count, leftmost, rightmost, average, …)
- Even simpler is a map
  - Compute new collection independently from elements
    - Or update in place (standard trade-offs)
  - Example: Increment all array elements
- These two patterns are *the workhorses* of parallel programming
  - Pedagogically, have students write them out *N* times rather than use map and reduce *primitives*
  - To save time tonight, I’m trying informal code *templates*
    - *In provided Java files (and next two slides)*

Reduction template for arrays

```
class MyClass extends RecursiveTask<AnsType> {
  int lo; int hi; ArrayType[] arr;
  SumThread(ArrayType[]a,int l,int h){lo=l;hi=h;arr=a;}
  public AnsType compute(){
    if(hi - lo < SEQUENTIAL_CUTOFF)
      // sequential algorithm
      return ans;
    } else {
      MyClass left = new MyClass(arr,lo,(hi+lo)/2);
      MyClass right= new MyClass(arr,(hi+lo)/2,hi);
      left.fork();
      AnsType rightAns = right.compute();
      AnsType leftAns = left.join();
      return // combine leftAns and RightAns
    }
  }
 static int SEQUENTIAL_CUTOFF = 1000;
 static AnsType myAlgorithm(ArrayType[] arr){
   ForkJoinPool pool = Main.fjPool;
   return pool.invoke(new MyClass(arr,0,arr.length));
 }
}
```

Map template for arrays (update-in-place)

```
class MyClass extends RecursiveAction {
  int lo; int hi; ArrayType[] arr;
  SumThread(int[] a, int l, int h){lo=l; hi=h; arr=a;}
  public void compute(){
    if(hi - lo < SEQUENTIAL_CUTOFF)
      // sequential algorithm
    } else {
      MyClass left = new MyClass(arr,lo,(hi+lo)/2);
      MyClass right= new MyClass(arr,(hi+lo)/2,hi);
      left.fork();
      right.compute();
      left.join();
    }
  }
 static int SEQUENTIAL_CUTOFF = 1000;
 static void myAlgorithm(ArrayType[] arr){
   ForkJoinPool pool = Main.fjPool;
   pool.invoke(new MyClass(arr,0,arr.length));
 }
}
```

Exercises

See handout and Java files for more details

Reductions over a `String[]`

- Easier: Leftmost `String` starting with `’S’` (`null` for none)
- Easier: Index of leftmost `String` starting with `’S’` (`-1` for none)
- More Challenging: Second-to-left `String` starting with `’S’`
- Even More Challenging: *k*th-from-left `String` starting with `’S’`

Maps over a `String[]`

- Easier: Replace every `String` starting with `’S’` with `"[redacted]"`
- More Challenging: Take as parameter an object with a method taking and returning a `String`; apply method to each element

Break

Where are we

- Students really can write maps and reductions over arrays
  - Trees, 2D arrays easy too
  - Easier for homework than during a workshop

- Remaining parallelism topics (necessarily brief tonight)
  - Asymptotic analysis (great fit in course)
  - Amdahl’s Law (incredibly important and sobering)
  - 2-3 non-trivial algorithms (just like with graphs!)

- Then concurrency
  - Locks and how to use them
  - Other topics as time permits

Work and Span

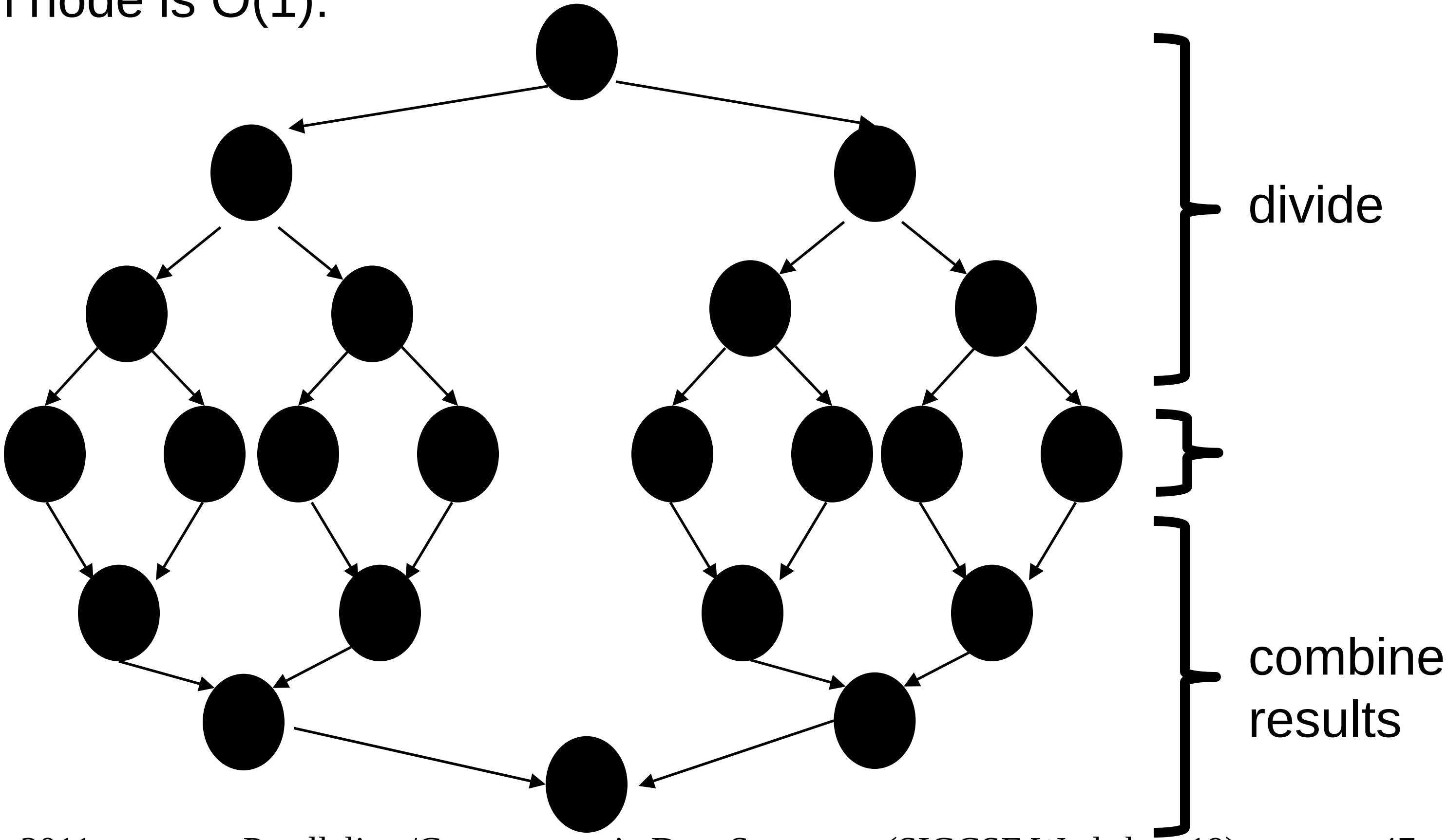
Let $\mathbf{T_P}$ be the running time if there are **P** processors available

Two key measures of running time

- Work: How long it would take 1 processor = $\mathbf{T_1}$
  - Just "sequentialize" the recursive forking
- Span: How long it would take infinity processors = $\mathbf{T_\infty}$
  - The longest dependence-chain
  - Example: $O(\texttt{log}\ n)$ for summing an array since > $n/2$ processors is no additional help
  - Also called "critical path length" or "computational depth"

The DAG

- Can treat execution as a (conceptual) DAG where nodes cannot start until predecessors finish
- A general model, but our fork-join reductions look like this, where each node is O(1):



Connecting to performance

- Work = $\mathbf{T_1}$ = sum of run-time of all nodes in the DAG
  - That lonely processor does everything
  - Any topological sort is a legal execution
  - $O(n)$ for simple maps and reductions

- Span = $\mathbf{T_\infty}$ = sum of run-time of all nodes on the most-expensive path in the DAG
  - An infinite army can do everything that is ready to be done, but still has to wait for earlier results
  - $O(\log n)$ for simple maps and reductions

Parallel algorithms is about decreasing span without increasing work too much

Finish the story: thanks ForkJoin library!

- So we know $\mathbf{T_1}$ and $\mathbf{T_\infty}$ but we want $\mathbf{T_P}$ (e.g., **P**=4)
- (Ignoring caching issues), $\mathbf{T_P}$ can't beat
  - $\mathbf{T_1 / P}$ why not?
  - $\mathbf{T_\infty}$ why not?
- So an *asymptotically* optimal execution would be:

$$\mathbf{T_P = \mathit{O}((T_1 / P) + T_\infty)}$$

  - First term dominates for small **P**, second for large **P**
- The ForkJoin Framework gives an *expected-time guarantee* of asymptotically optimal! (It flips coins when *scheduling*)
  - How? For an advanced course (few need to know)
  - Assumes your base cases are small-ish and balanced

Now the bad news

- So far: analyze parallel programs in terms of work and span
- In practice, typically have parts of programs that parallelize well…
  - Such as maps/reduces over arrays and trees

  …and parts that don't parallelize at all

  - Reading a linked list, getting input, doing computations where each needs the previous step, etc.
  - "Nine women can't make a baby in one month"

Amdahl's Law (mostly bad news)

Let the ***work*** (time to run on 1 processor) be 1 unit time

Let **S** be the portion of the execution that can't be parallelized

Then:

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

Suppose we get *perfect linear speedup on the parallel portion*

Then:

$$T_P = S + (1-S)/P$$

So the overall speedup with **P** processors is (Amdahl's Law):

$$T_1 / T_P = 1 / (S + (1-S)/P)$$

Why such bad news

$$T_1 / T_P = 1 / (S + (1-S)/P)$$

- Suppose 33% of a program is sequential
  - Then a billion processors won’t give a speedup over 3
- Suppose you miss the good old days (1980-2005) where 12ish years was long enough to get 100x speedup
  - Now suppose in 12 years, clock speed is the same but you get 256 processors instead of 1
  - For 256 processors to get at least 100x speedup, we need

$$100 \leq 1 / (\mathbf{S} + (1-\mathbf{S})/256)$$

Which means $\mathbf{S} \leq .0061$ (i.e., 99.4% perfectly parallelizable)

Homework problem: Depressing plots with a spreadsheet!!

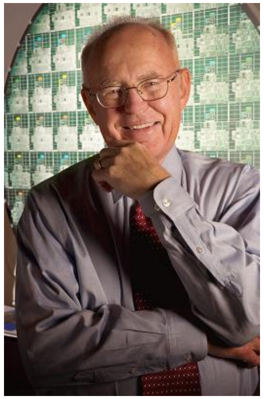
All is not lost

Amdahl’s Law is a bummer!

  – But it doesn’t mean additional processors are worthless

- Can find new parallel algorithms
  - Some things that seem sequential are actually parallelizable
- Can change the problem we’re solving or do new things
  - Example: Video games use tons of parallel processors
    - They are not rendering 10-year-old graphics faster
    - They are rendering more beautiful(?) monsters

Moore and Amdahl



- Moore’s “Law” is an observation about the progress of the semiconductor industry
  - Transistor density doubles roughly every 18 months
- Amdahl’s Law is a mathematical theorem
  - Diminishing returns of adding more processors
  - Fits beautifully in data structures!
- Both are incredibly important in designing computer systems

Tonight: A whirlwind tour!

- Context: What I mean by “in data structures”
- Introductions: Name, rank, and serial number ☺, plus
  - 1-3 terms, concepts, ideas related to parallelism/concurrency

- Distinguishing parallelism and concurrency

- Parallelism with Java’s ForkJoin Framework – and try it out

- Asymptotic analysis of parallel algorithms
- Fancier parallel algorithms

- Synchronization and mutual exclusion
  - Locks, programming guidelines, memory-consistency models, condition variables, …

- Review: The *N* main concepts & why they fit in data structures

The prefix-sum problem

Given **`int[] input`**, produce **`int[] output`** where **`output[i]`** is the sum of **`input[0]+input[1]+…+input[i]`**

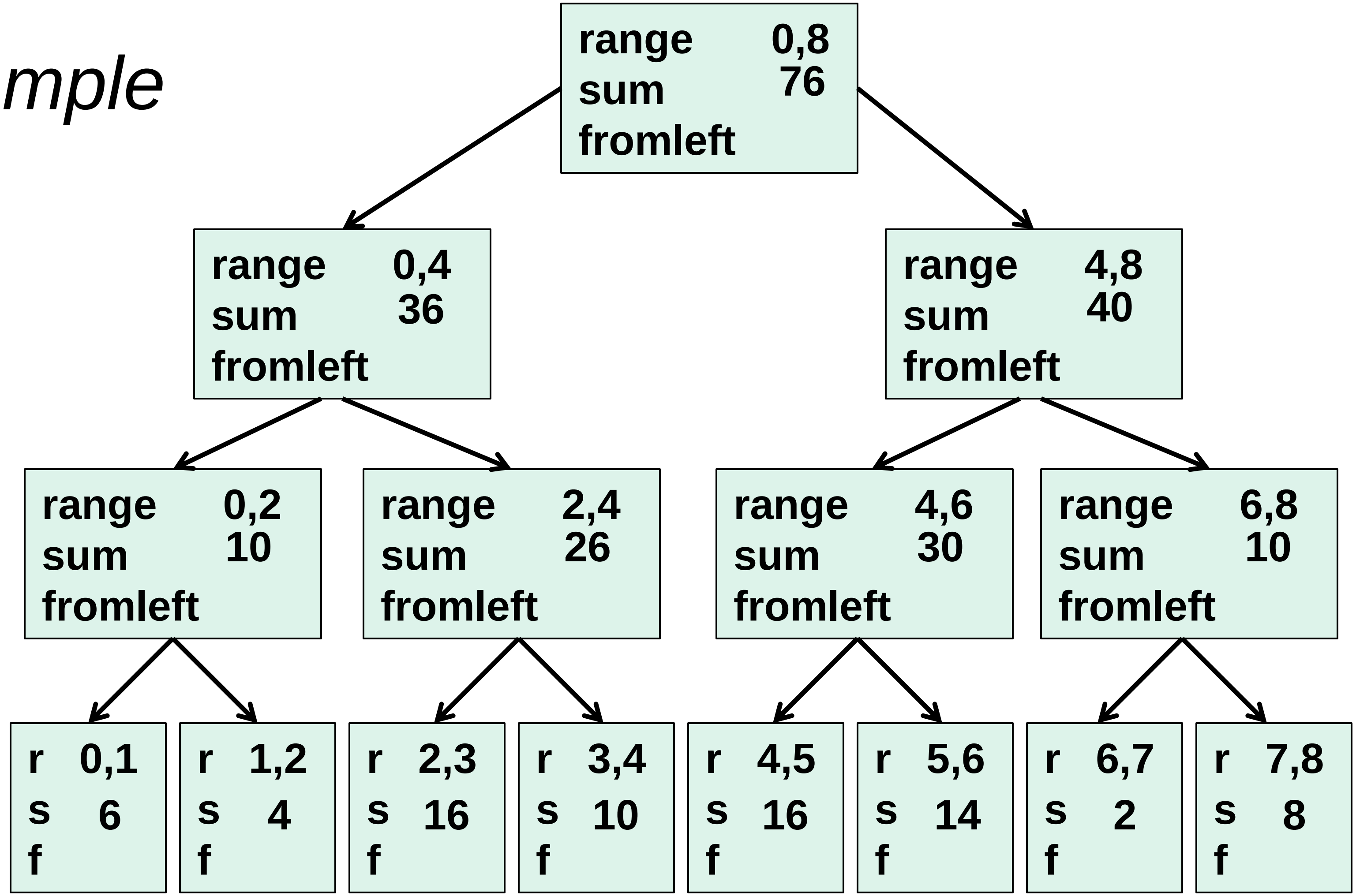
Sequential can be a CS1 exam problem:

```
int[] prefix_sum(int[] input){
  int[] output = new int[input.length];
  output[0] = input[0];
  for(int i=1; i < input.length; i++)
    output[i] = output[i-1]+input[i];
  return output;
}
```

Does not appear parallelizable

- Work: $O(n)$, Span: $O(n)$
- This *algorithm* is sequential, but a *different algorithm* has Work: $O(n)$, Span: $O(\log n)$

Example



range 0,8
sum 76
fromleft

range 0,4
sum 36
fromleft

range 4,8
sum 40
fromleft

range 0,2
sum 10
fromleft

range 2,4
sum 26
fromleft

range 4,6
sum 30
fromleft

range 6,8
sum 10
fromleft

r 0,1
s 6
f

r 1,2
s 4
f

r 2,3
s 16
f

r 3,4
s 10
f

r 4,5
s 16
f

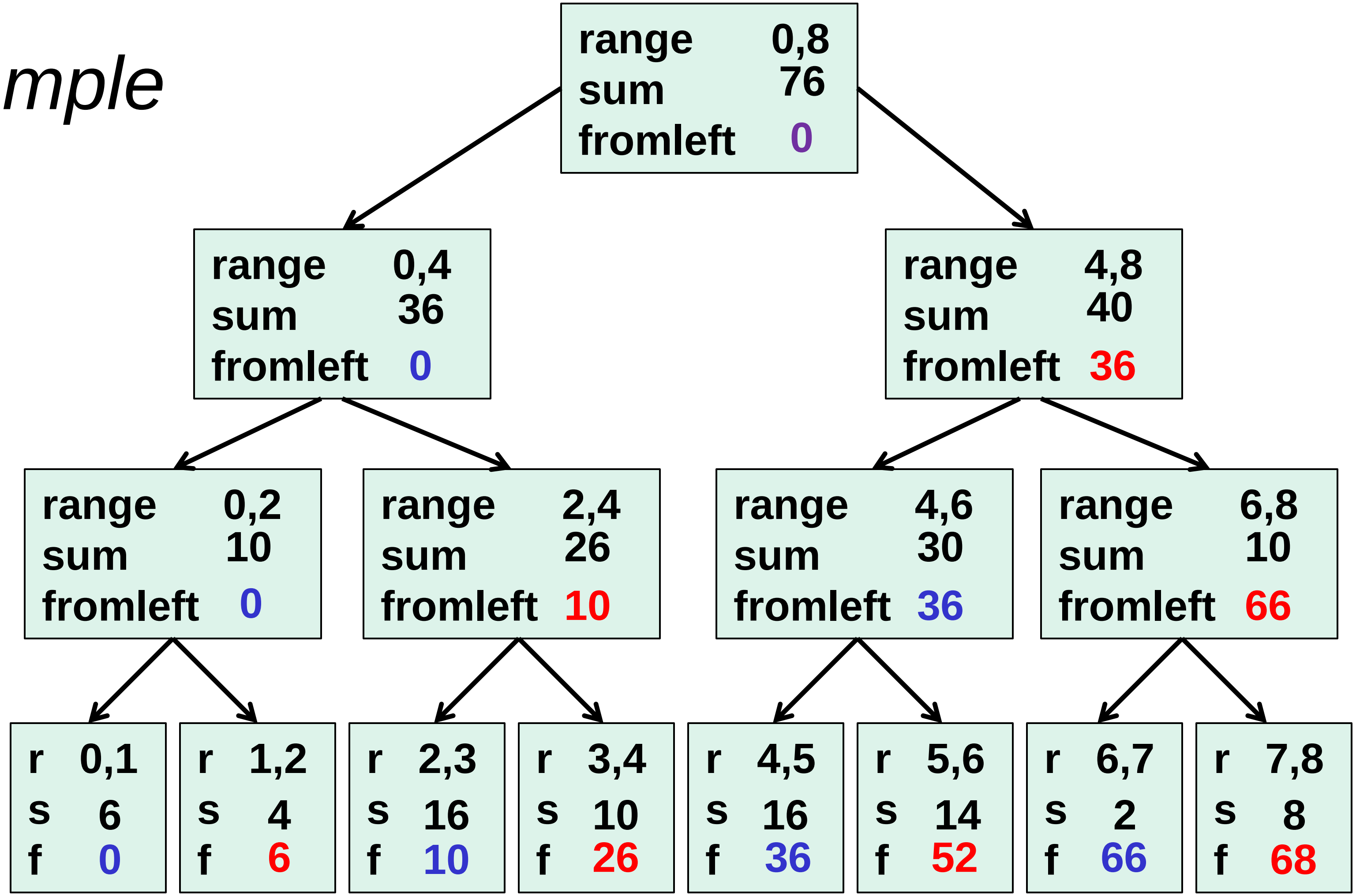
r 5,6
s 14
f

r 6,7
s 2
f

r 7,8
s 8
f

`input`	6	4	16	10	16	14	2	8
`output`								

Example



range 0,8
sum 76
fromleft 0

range 0,4
sum 36
fromleft 0

range 4,8
sum 40
fromleft 36

range 0,2
sum 10
fromleft 0

range 2,4
sum 26
fromleft 10

range 4,6
sum 30
fromleft 36

range 6,8
sum 10
fromleft 66

r 0,1
s 6
f 0

r 1,2
s 4
f 6

r 2,3
s 16
f 10

r 3,4
s 10
f 26

r 4,5
s 16
f 36

r 5,6
s 14
f 52

r 6,7
s 2
f 66

r 7,8
s 8
f 68

`input`	6	4	16	10	16	14	2	8
`output`	6	10	26	36	52	66	68	76

Pack

[Non-standard terminology]

Given an array **`input`**, produce an array **`output`** containing only elements such that **`f(elt)`** is **`true`**

Example:
```
input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]
f: is elt > 10
output [17, 11, 13, 19, 24]
```

Parallelizable?

- Finding elements for the output is easy
- But getting them in the right place seems hard

Parallel prefix to the rescue

1. Parallel map to compute a bit-vector for true elements

```
input   [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]
bits    [1,  0, 0, 0,  1, 0,  1,  1, 0,  1]
```

2. Parallel-prefix sum on the bit-vector

```
bitsum [1,  1, 1, 1,  2, 2,  3,  4, 4,  5]
```

3. Parallel map to produce the output

```
output [17, 11, 13, 19, 24]
```

```
output = new array of size bitsum[n-1]
FORALL(i=1; i < input.length; i++){
  if(bits[i]==1)
    output[bitsum[i]-1] = input[i];
}
```

Keep Layering

- In turn, pack is the key piece for a parallel variant of quicksort with a very good span
  - Parallelize the partition, not just the recursive calls

- In any case, the point is to show very useful, very non-obvious parallel algorithms
  - Just as Dijkstra’s shortest-paths is a very useful, very non-obvious sequential algorithm

Mini-Break Before Concurrency?

Tonight: A whirlwind tour!

- Context: What I mean by “in data structures”
- Introductions: Name, rank, and serial number ☺, plus
  - 1-3 terms, concepts, ideas related to parallelism/concurrency
- Distinguishing parallelism and concurrency
- Parallelism with Java’s ForkJoin Framework – and try it out
- Asymptotic analysis of parallel algorithms
- Fancier parallel algorithms
- Synchronization and mutual exclusion
  - Locks, programming guidelines, memory-consistency models, condition variables, …
- Review: The *N* main concepts & why they fit in data structures

A warning

Workshop time-allotment misleading:

Teaching interleaving, race conditions, locks, etc. takes a lot of time

- Switch mindset: Loosely coordinated threads, occasionally accessing shared data
- More difficult for students than parallelism
- Slightly more than half the lecture time

The good news:

Basic *data structures* (stacks, queues, hashtables) provide canonical examples

  - Leave to O/S course scheduling, fairness, context-switching, …

Canonical example

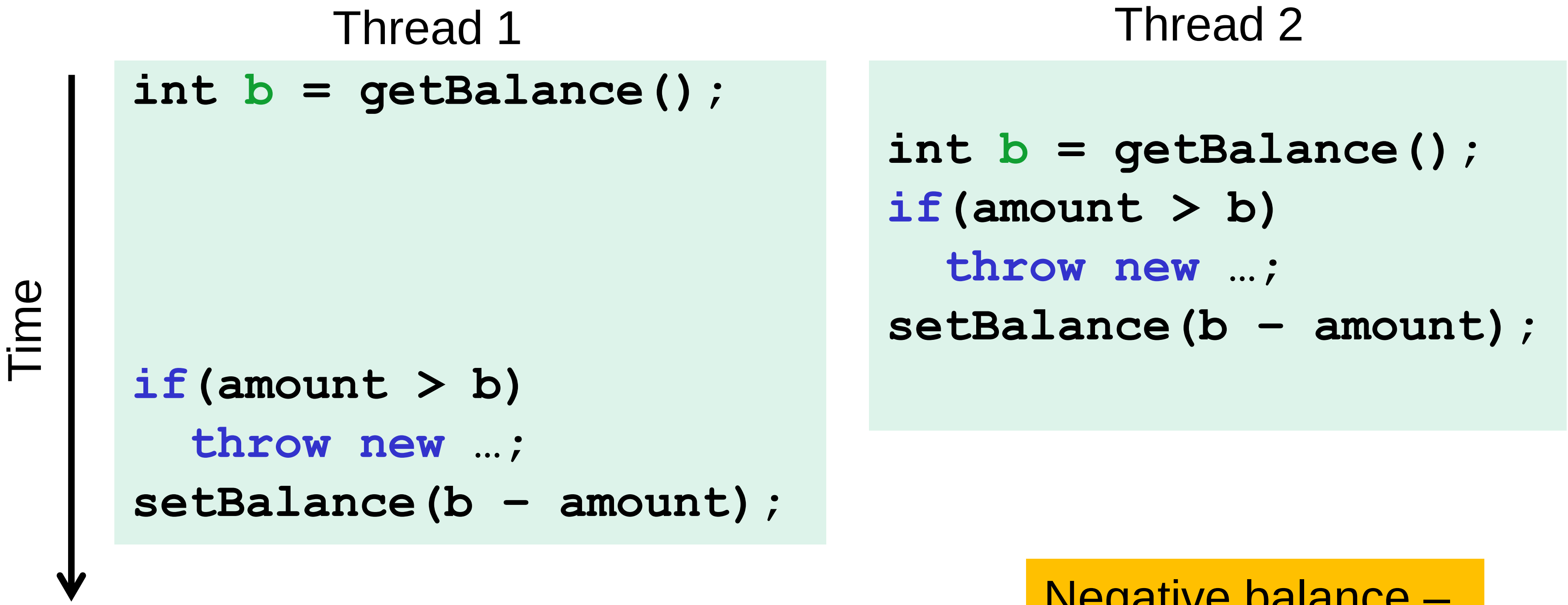
Correct code in a single-threaded world

```
class BankAccount {
  private int balance = 0;
  void setBalance(int x) { balance = x; }
  int  getBalance()      { return balance; }
  void withdraw(int amount) {
    int b = getBalance();
    if(amount > b)
      throw new WithdrawTooLargeException();
    setBalance(b – amount);
  }
  … // other operations like deposit, etc.
}
```

A bad interleaving

Interleaved **`withdraw(100)`** calls on the *same account*

- Assume initial **`balance`** 150



Thread 1

```
int b = getBalance();




if(amount > b)
  throw new …;
setBalance(b – amount);
```

Thread 2

```

int b = getBalance();
if(amount > b)
  throw new …;
setBalance(b – amount);
```

Negative balance – unhappy bank

What next

1. Try to fix without locks: it won't work!
2. Explain locks as an ADT in pseudocode:
   - **`new`**: make a new lock
   - **`acquire(lk)`**: *blocks* if this lock is already currently *"held"*
     - Once *"not held"*, makes lock *"held"*
   - **`release(lk)`**: makes this lock *"not held"*
     - if >= 1 threads are blocked on it, exactly 1 will acquire it
3. Explain re-entrant locks as an extended ADT
   - **`acquire`** and **`release`** manage a counter for "same thread"
4. Java's convenient **`synchronized`** statement
   - Every object is a lock
   - **`synchronized`** methods as a shorthand

Java version #1 (correct but non-idiomatic)

```
class BankAccount {
  private int balance = 0;
  private Object lk = new Object();
  void setBalance(int x)
    { synchronized (lk) { balance = x; } }
  int getBalance()
    { synchronized (lk) { return balance; } }
  void withdraw(int amount) {
    synchronized (lk) {
      int b = getBalance();
      if(amount > b)
        throw …
      setBalance(b – amount);
    }
  }
  // deposit also uses synchronized(lk)
}
```

Java version #2

```
class BankAccount {
  private int balance = 0;
  int getBalance()
    { synchronized (this){ return balance; } }
  void setBalance(int x)
    { synchronized (this){ balance = x; } }
  void withdraw(int amount) {
    synchronized (this) {
      int b = getBalance();
      if(amount > b)
        throw …
      setBalance(b – amount);
    }
  }
  // deposit also uses synchronized(this)
}
```

Java version #3 (final version)

```
class BankAccount {
  private int balance = 0;
  synchronized int getBalance()
    { return balance; }
  synchronized void setBalance(int x)
    { balance = x; }
  synchronized void withdraw(int amount) {
     int b = getBalance();
     if(amount > b)
       throw …
     setBalance(b – amount);
  }
  // deposit also uses synchronized
}
```

Key points from example

- All methods must use the same lock

- But different instances can/should use different locks
  - More concurrency
  - Okay because methods only access instance’s fields

- Second version exposes lock to clients
  - Surprisingly, good style so client can make larger synchronized operations

Another example: Stacks

```
class Stack<E> {
  … // state used by isEmpty, push, pop
  synchronized boolean isEmpty() { … }
  synchronized void push(E val) { … }
  synchronized E pop() {
    if(isEmpty())
      throw new StackEmptyException();
    …
  }
  E peek() { // this is wrong
     E ans = pop();
     push(ans);
     return ans;
  }
}
```

Data race vs. Bad Interleaving

This point is not well-understood by most teachers & programmers

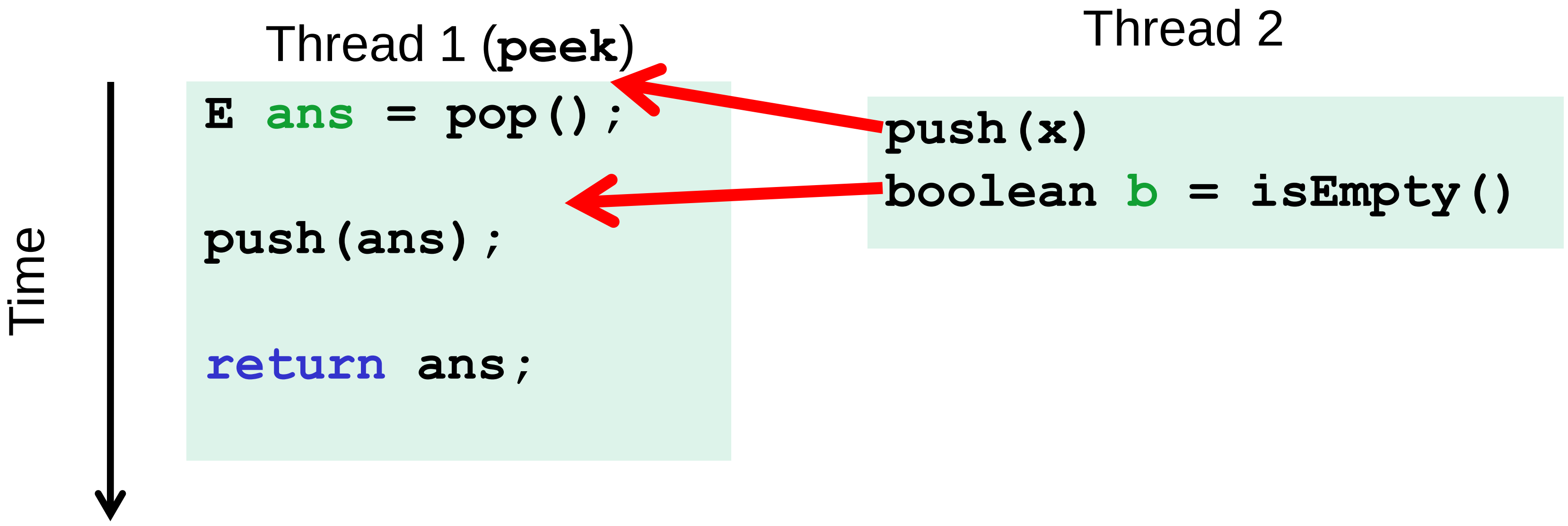
- Please read the notes about this

The (poor) term “race condition” can refer to two *different* things resulting from lack of synchronization:

1. Data races: Simultaneous read/write or write/write of the same memory location
   - This is (for mortals) **always** an error, due to compiler & HW
   - Stack example has no data races
2. Bad interleavings: Despite lack of data races, exposing bad intermediate state
   - “Bad” depends on your specification
   - Stack example has lots of these…

peek and isEmpty

- Property we want: If there has been a **`push`** and no **`pop`**, then **`isEmpty`** returns **`false`**
- With **`peek`** as written, property can be violated – how?



Thread 1 (**`peek`**)

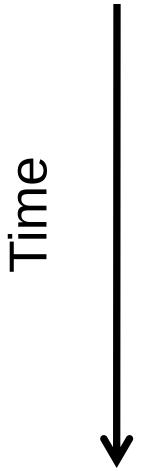
```
E ans = pop();

push(ans);

return ans;
```

Thread 2

```
push(x)
boolean b = isEmpty()
```

Activity?

- Property we want: Values are returned from **`pop`** in LIFO order
- With **`peek`** as written, property can be violated – how?



Thread 1 (**`peek`**)

```
E ans = pop();

push(ans);

return ans;
```

Thread 2

```
push(x)
push(y)
E e = pop()
```

Given enough practice, students get good at finding bad interleavings – an essential reasoning skill for concurrency

Time for another?

- Property we want: **`peek`** doesn't throw an exception if number of pushes exceeds number of pops

- With **`peek`** as written, property can be violated – how?

Time

Thread 1 (**`peek`**)

```
E ans = pop();

push(ans);

return ans;
```

Thread 2

```
E ans = pop();

push(ans);

return ans;
```

Then what?

- Finding errors is easier than avoiding them!
  - So far: Gave them a chainsaw without a safety manual ☺

- So I spend most of a lecture on programming guidelines
  - Avoid mutating shared memory
  - Simple and consistent locking protocols
  - Start with coarse-grained locking
  - Use libraries for shared data structures
  - …

This is all new to them and I don't think they get it

- But hopefully they go back to the slides and reading notes during their internships!

Lastly

Three more things are part of a proper introduction:

- Deadlock: Too much synchronization instead of too little

- Reader/writer locks: Dictionaries are a great example
  - Key concept: read/read sharing is okay

- Passive waiting:
  - A queue for transferring work
    - An empty or full queue is not an error; it means wait
  - Avoid busy waiting with condition variables
  - Alas, condition variables, especially in Java, are very hard to use correctly, but I show them anyway
    - Taking a blocking-queue as a primitive and building on top of it might work better

Tonight: A whirlwind tour!

- Context: What I mean by “in data structures”
- Introductions: Name, rank, and serial number ☺, plus
  - 1-3 terms, concepts, ideas related to parallelism/concurrency

- Distinguishing parallelism and concurrency

- Parallelism with Java’s ForkJoin Framework – and try it out

- Asymptotic analysis of parallel algorithms
- Fancier parallel algorithms

- Synchronization and mutual exclusion
  - Locks, programming guidelines, memory-consistency models, condition variables, …

- Review: The *N* main concepts & why they fit in data structures

Conclusions: Main Concepts

- Parallelism vs. concurrency

- Parallelism
  - Reductions vs. maps vs. fancy algorithms
  - Divide-and-conquer using fork-join
  - Work vs. span
  - Amdahl's Law

- Concurrency
  - The need for synchronization
  - Data races (*always* wrong) vs. bad interleavings
  - Guidelines for programming with locks
  - Deadlock
  - Passive waiting

Conclusions: Meta

Why in a data structures course:

Parallelism:

- Same kind of obvious and non-obvious algorithms
- Basic asymptotic analysis, including Amdahl's Law
- Balanced trees have logarithmic height (divide-and-conquer)
- More useful than skew heaps and network flow

Concurrency

- Making an ADT thread-safe requires thinking about what intermediate states are exposed
- Stacks, queues, and dictionaries are key shared resources

You can do this! (2 of the 3 instructors after me had no experience with parallelism/concurrency, just as I had to re-learn AVL trees)

What I have

http://www.cs.washington.edu/homes/djg/teachingMaterials/

- 8 hours of Powerpoint
- 65 pages of reading notes
- A cool (?) programming project (hang around after for a demo?)
- Sample homeworks and exam

Also: Eagerness to answer your questions

Also: No problem with you modifying, adapting, etc.

Also: I'd be delighted to foster an informal community

Feedback?

Your turn:

- What of this would you use?
- What are the barriers you face or concerns you have?
- What do you think is missing?

Separate question: Feedback on the workshop and its focus