CSE373: Data Structures & Algorithms
Introduction to Multithreading & Fork-Join Parallelism

Hunter Zahn

Summer 2016

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?

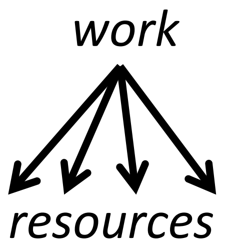
- Next computer you buy will likely have 4 processors
  - Wait a few years and it will be 8, 16, 32, …
  - The chip companies have decided to do this (not a "law")

- 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

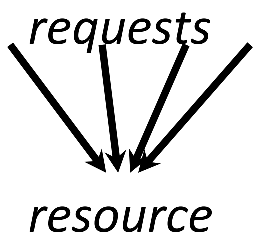
Parallelism:

Use extra resources to solve a problem faster



Concurrency:

Correctly and efficiently manage access to shared resources



Parallelism vs. Concurrency

Parallelism is when tasks literally run at the same time, eg. on a multicore processor.

Concurrency is when two or more tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant.

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

We will just do a little parallelism, avoiding concurrency issues

An analogy

CS1 idea: 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

Parallelism Example

Parallelism: Use extra resources to solve a problem faster

Pseudocode for array sum

- Bad style for reasons we'll see, but may get roughly 4x

```
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* (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)
   }
}
```

Shared memory

The model we will assume is shared memory with explicit threads

- *Not* the only approach, may not be best, but time for only one

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:

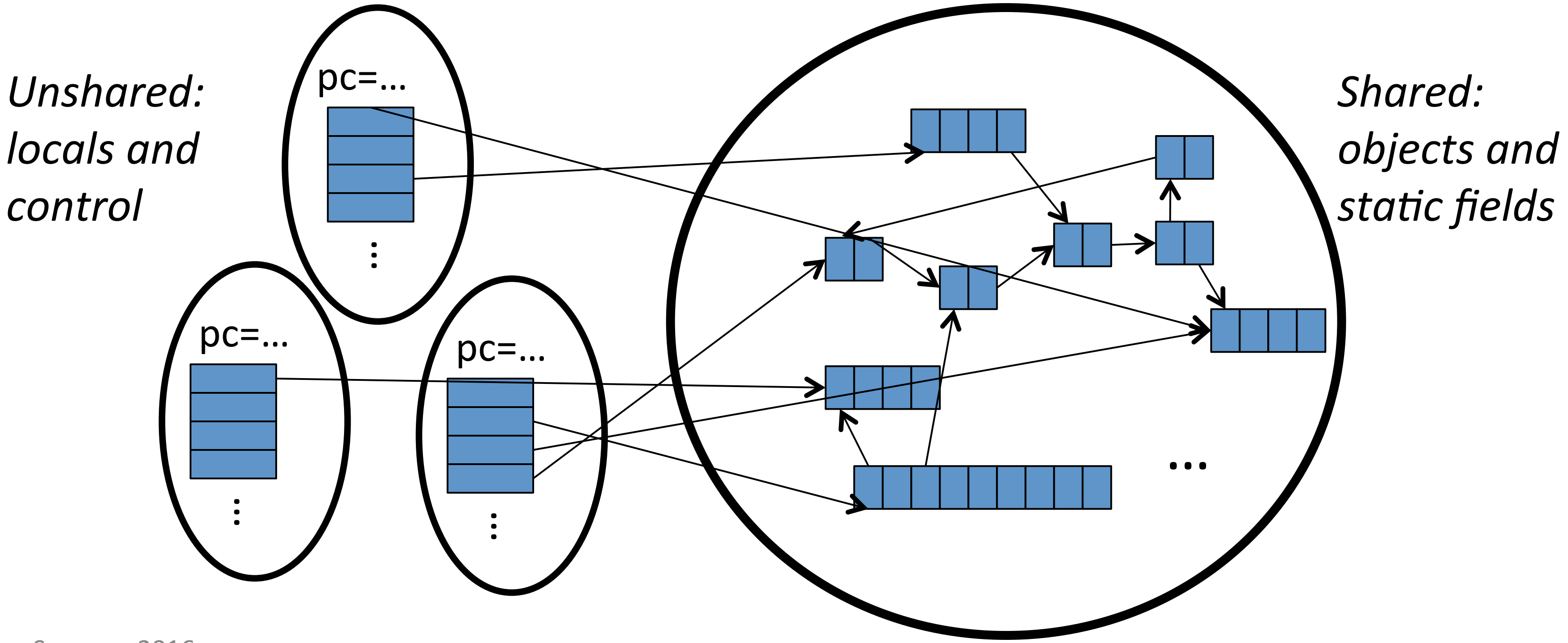
- A set of *threads*, each with its own program counter & call stack
  - No access to another thread's local variables
- Threads can (implicitly) share static fields / objects
  - To *communicate*, write somewhere another thread reads

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



Unshared: locals and control
pc=…
pc=…
pc=…
…
Shared: objects and static fields

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)*
  - A way for one thread to wait for another to finish
  - [Other features needed in practice for concurrency]

Java basics

Learn a couple basics built into Java via **`java.lang.Thread`**

- But for style of parallel programming we'll advocate, do *not* use these threads; use Java 7's ForkJoin Framework instead

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"

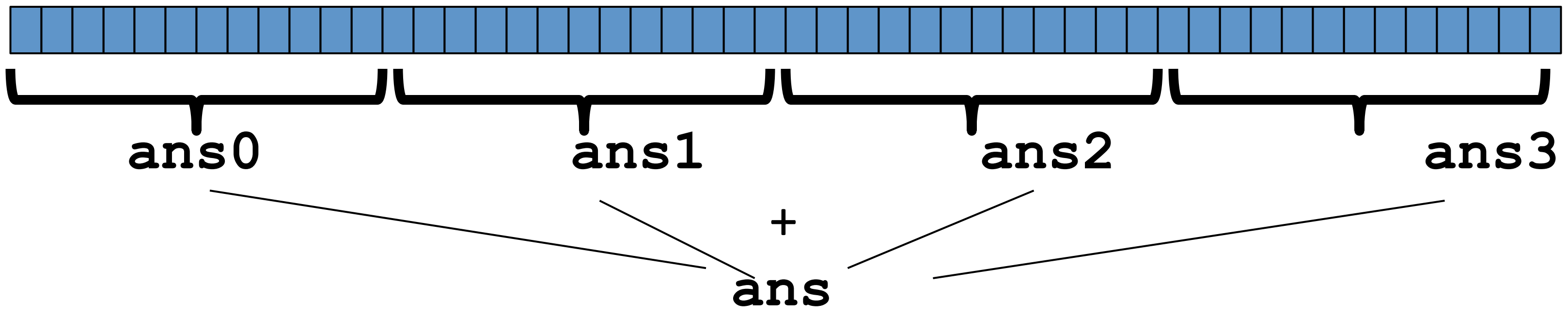
What if we instead called the **`run`** method of **`C`**?

- 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; // 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
  - We will stick with **`join`** to do so

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?

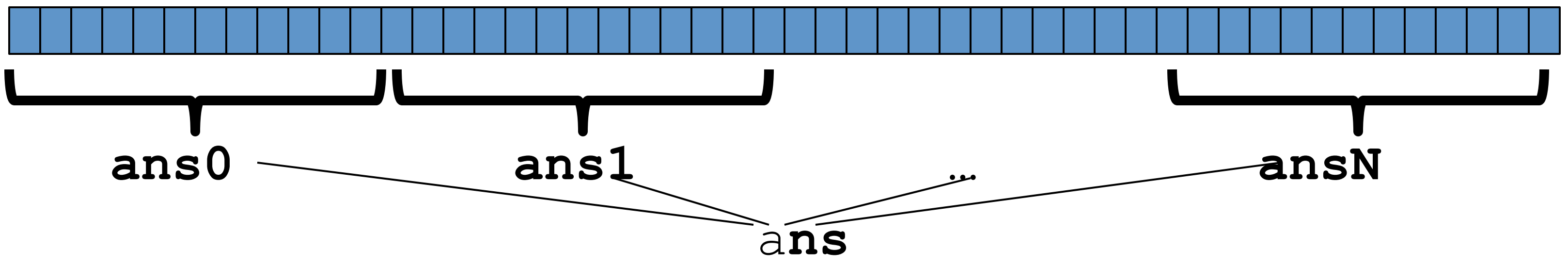
   – If we create 4 threads and all the slow data is processed by 1 of them, we won't get nearly a 4x speedup

     • 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 using a different Java library]



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];
  …
}
```

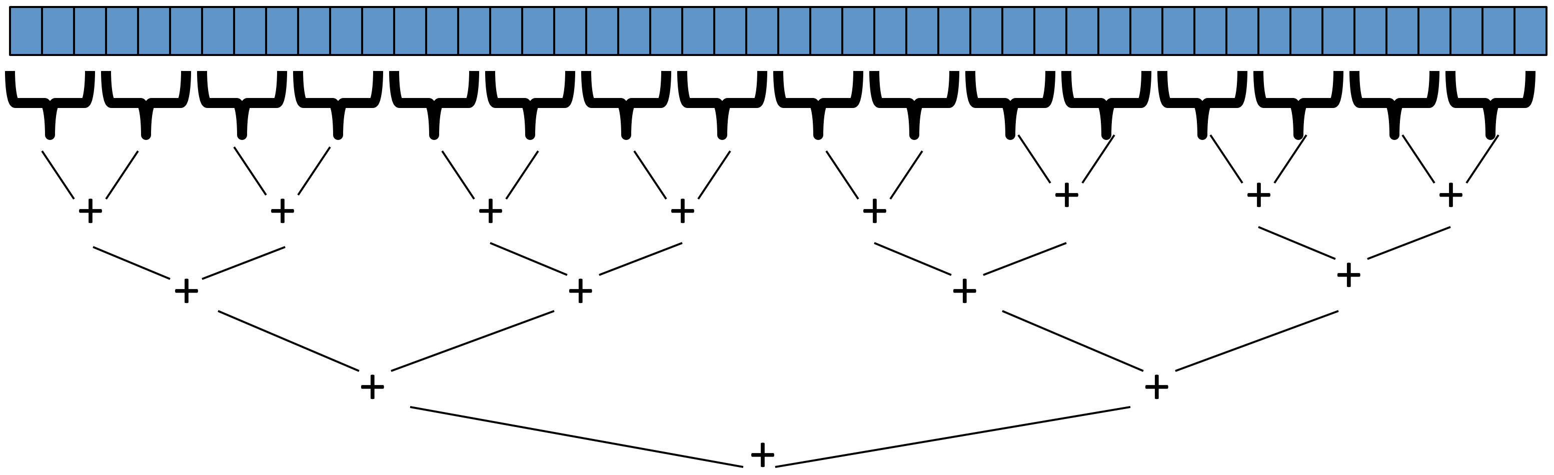
Then combining results will have `arr.length / 1000` additions

- 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

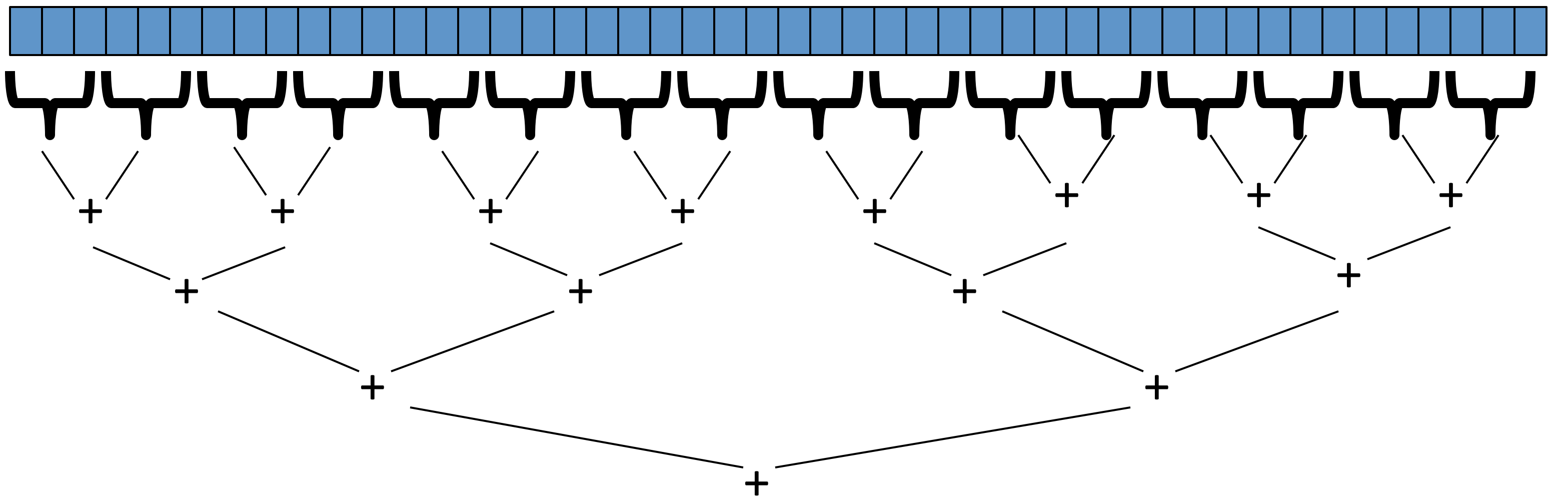
– Parallelism for the recursive calls

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: consider reality of **P** << *n* processors



Being realistic

- In theory, you can divide down to single elements, do all your result-combining in parallel and get optimal speedup
  - Total time $O(n/numProcessors + \log n)$
- 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

Being realistic, part 2

- 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
  - Library's implementation is a fascinating but advanced topic
    - Next lecture will discuss its guarantees, not how it does it

CSE373: Data Structures & Algorithms
Parallel Reductions, Maps, and Algorithm Analysis

Hunter Zahn

Summer 2016

Outline

Done:

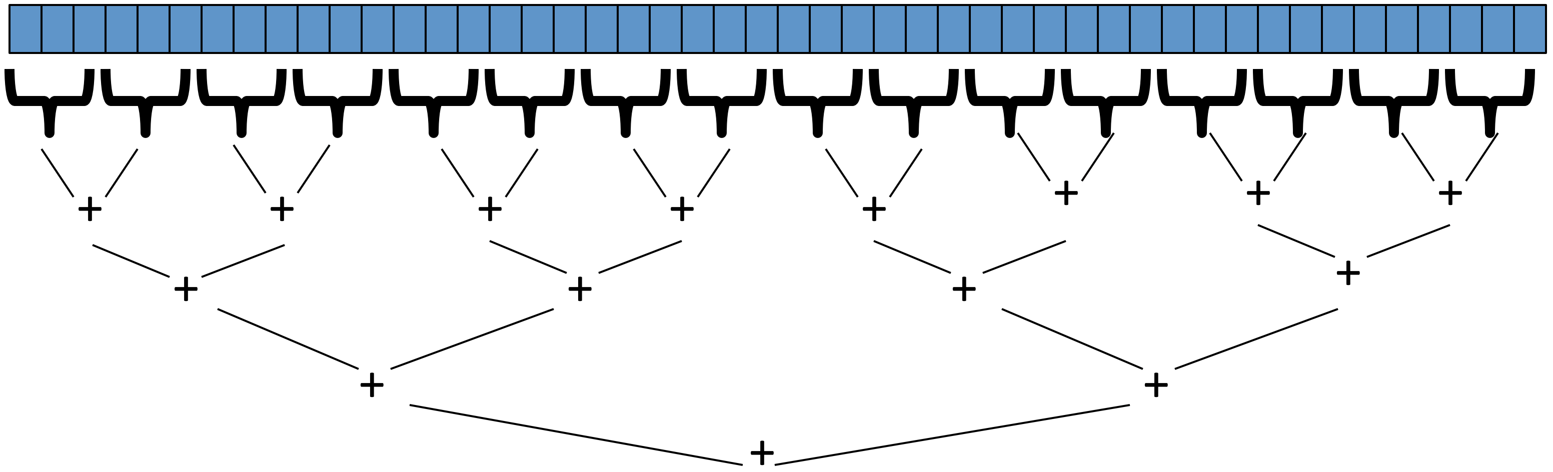
- How to write a parallel algorithm with fork and join
- Why using divide-and-conquer with lots of small tasks is best
  - Combines results in parallel
  - (Assuming library can handle "lots of small threads")

Now:

- More examples of simple parallel programs that fit the "map" or "reduce" patterns
- Teaser: Beyond maps and reductions
- Asymptotic analysis for fork-join parallelism
- Amdahl's Law

What else looks like this?

- Saw summing an array went from *O*(*n*) sequential to *O*(`log` *n*) parallel (*assuming **a lot** of processors and very large n*)!
  - Exponential speed-up in theory (*n* / `log` *n* grows exponentially)



- Anything that can use results from two halves and merge them in *O*(1) time has the same property…

Examples

- Maximum or minimum element
- Is there an element satisfying some property (e.g., is there a 17)?
- Left-most element satisfying some property (e.g., first 17)
  - What should the recursive tasks return?
  - How should we merge the results?
- Corners of a rectangle containing all points (a "bounding box")
- Counts, for example, number of strings that start with a vowel
  - This is just summing with a different base case
  - Many problems are!

Reductions

- Computations of this form are called reduction
- Produce single answer from collection via an associative operator
  - Associative: a + (b+c) = (a+b) + c
  - Examples: max, count, leftmost, rightmost, sum, product, …
  - Non-examples: median, subtraction, exponentiation
- But some things are inherently sequential
  - How we process **`arr[i]`** may depend entirely on the result of processing **`arr[i-1]`**

Even easier: Maps (Data Parallelism)

- A map operation 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;
}
```

In Java

```
class VecAdd extends java.lang.Thread {
  int lo; int hi; int[] res; int[] arr1; int[] arr2;
  VecAdd(int l,int h,int[] r,int[] a1,int[] a2){ … }
  protected void run(){
    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.start();
      right.run();
      left.join();
    }
  }
}
int[] add(int[] arr1, int[] arr2){
  assert (arr1.length == arr2.length);
  int[] ans = new int[arr1.length];
  (new VecAdd(0,arr.length,ans,arr1,arr2).run();
  return ans;
}
```

Maps and reductions

Maps and reductions: the "workhorses" of parallel programming

- By far the two most important and common patterns
- 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

Beyond maps and reductions

- Some problems are "inherently sequential"

  "Nine women can't make a baby in one month"

- But not all parallelizable problems are maps and reductions

- If had one more lecture, would show "parallel prefix", a clever algorithm to parallelize the *problem* that this sequential *code* solves

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

```
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;
}
```

Analyzing algorithms

- Like all algorithms, parallel algorithms should be:
  - Correct
  - Efficient
- For our algorithms so far, correctness is "obvious" so we'll focus on efficiency
  - Want asymptotic bounds
  - Want to analyze the algorithm without regard to a specific number of processors
  - Here: Identify the "best we can do" *if* the underlying *thread-scheduler* does its part

Work and Span

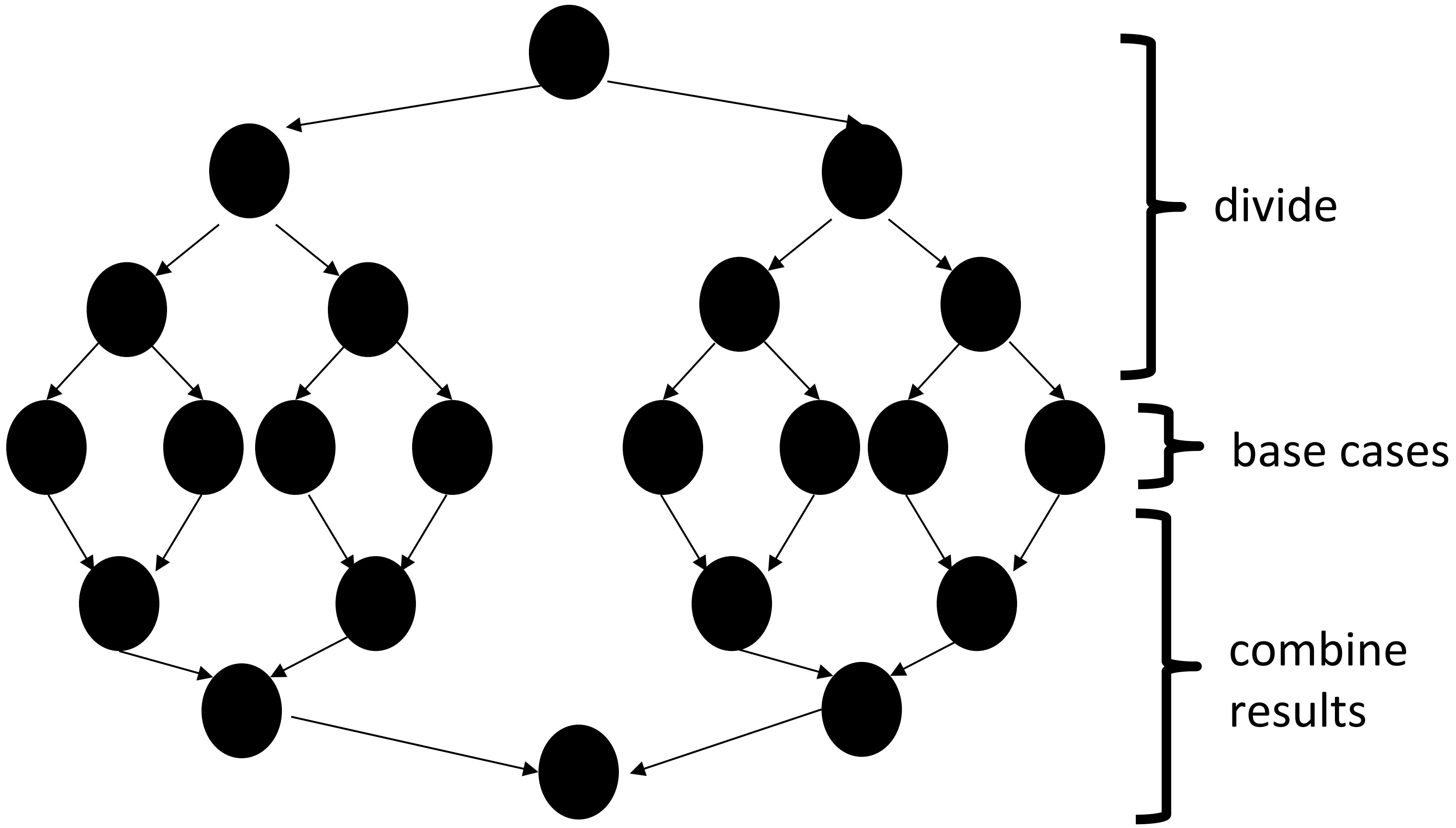
Let $\mathbf{T_P}$ be the running time if there are **P** processors available

Two key measures of run-time:

- Work: How long it would take 1 processor = $\mathbf{T_1}$
  - Just "sequential-ize" 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
    - Notice having > $n/2$ processors is no additional help

Our simple examples

- Picture showing all the "stuff that happens" during a reduction or a map: it's a (conceptual!) DAG



Connecting to performance

- Recall: $\mathbf{T_P}$ = running time if there are **P** processors available
- 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 maps and reductions
- Span = $\mathbf{T_\infty}$ = sum of run-time of all nodes on the most-expensive path in the DAG
  - Note: costs are on the nodes not the edges
  - Our infinite army can do everything that is ready to be done, but still has to wait for earlier results
  - $O(\texttt{log}\ n)$ for simple maps and reductions

Speed-up

Parallel algorithms is about decreasing span without increasing work too much

- Speed-up on **P** processors: $\mathbf{T_1 / T_P}$
- Parallelism is the maximum possible speed-up: $\mathbf{T_1 / T_\infty}$
  - At some point, adding processors won't help
  - What that point is depends on the span
- In practice we have **P** processors. How well can we do?
  - We cannot do better than $O(T_\infty)$ ("must obey the span")
  - We cannot do better than $O(T_1 / P)$ ("must do all the work")
  - Not shown: With a "good thread scheduler", can do this well (within a constant factor of optimal!)

Examples

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

- In the algorithms seen so far (e.g., sum an array):
  - $\mathbf{T_1} = O(n)$
  - $\mathbf{T_\infty} = O(\texttt{log}\ n)$
  - So expect (ignoring overheads): $\mathbf{T_P = \mathit{O}(max(\mathit{n}/P, \texttt{log}\ \mathit{n}))}$

- Suppose instead:
  - $\mathbf{T_1} = O(n^2)$
  - $\mathbf{T_\infty} = O(n)$
  - So expect (ignoring overheads): $\mathbf{T_P = \mathit{O}(max(\mathit{n}^2/P, \mathit{n}))}$

Amdahl’s Law (mostly 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/reductions over arrays

  …and parts that don’t parallelize at all

  - Such as reading a linked list, getting input, doing computations where each needs the previous step, etc.

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: $\mathbf{T_1 = S + (1\text{-}S) = 1}$

Suppose *parallel portion parallelizes perfectly (generous assumption)*

Then: $\mathbf{T_P = S + (1\text{-}S)/P}$

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

$$T_1 / T_P = 1 / (S + (1\text{-}S)/P)$$

And the parallelism (infinite processors) is:

$$T_1 / T_\infty = 1 / S$$

Why such bad news

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

- Suppose 33% of a program's execution 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)

All is not lost

Amdahl's Law is a bummer!

- Unparallelized parts become a bottleneck very quickly
- But it doesn't mean additional processors are worthless

- We can find new parallel algorithms
  - Some things that seem sequential are actually parallelizable

- We can change the problem 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