CSE332: Data Abstractions

Lecture 16: Into to Parallelism and Concurrency

James Fogarty

Winter 2012

Including slides developed in part by
Ruth Anderson, James Fogarty, Dan Grossman

From Our Previous Lecture

```
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;
}
```

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){
  … // note: shows idea, but has integer-division bug
  int subLen = arr.length / numThreads;
  SumThread[] ts = new SumThread[numThreads];
  for(int i=0; i < numThreads; i++){
   ts[i] = new SumThread(arr,i*subLen,(i+1)*subLen);
   ts[i].start();
  }
  for(int i=0; i < numThreads; i++) {
    …
  }
  …
```

A Better Approach

2. Want to use only the 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 3 processors available and 3 threads would take time **`X`**, creating 4 threads can have worst-case time of **`1.5X`**

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

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

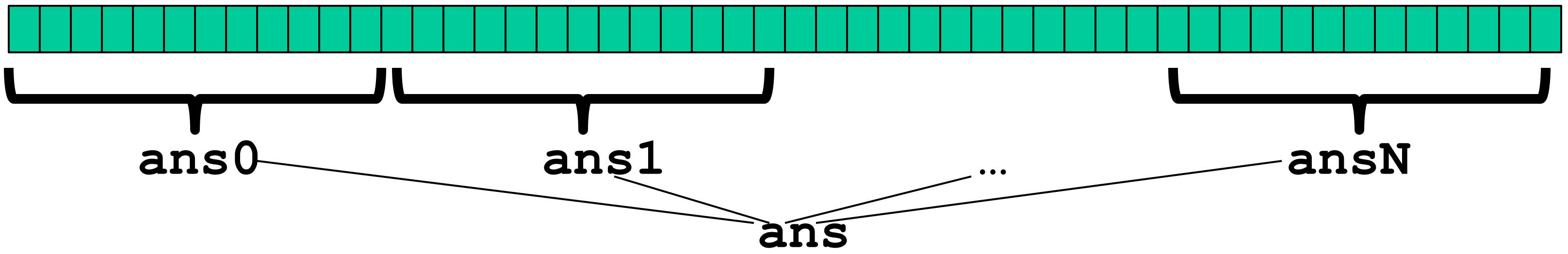
     • Example of a load imbalance

A Better Approach

The perhaps counterintuitive solution to all these problems is:

to use lots of threads, far more than the number of processors

- When a processor finishes a piece, it can start another
- Require a different algorithm, and will abandon Java 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, worst-case extra time is < 3% (if we ignore constant factors and load imbalance)
3. **Load Imbalance**: No problem if slow thread scheduled early enough
   - Variation probably small if pieces of work are small

Naïve Algorithm is Poor

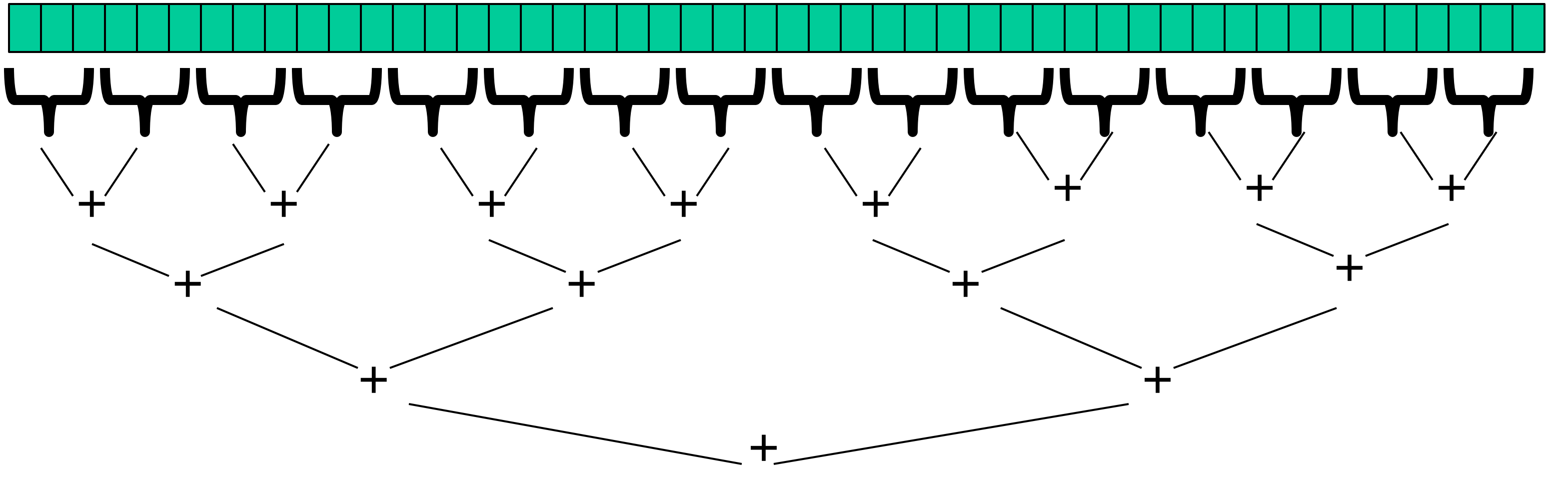
- Suppose we create 1 thread to process every 100 elements

```
int sum(int[] arr){
  …
  // How many pieces of size 100 do we have?
  int numThreads = arr.length / 100;
  SumThread[] ts = new SumThread[numThreads];
  …
}
```

- Combining results will require **`arr.length / 100`** additions
  - Linear in size of array
  - Previously we only had 4 pieces, Θ(1) to combine

- In the extreme, suppose we create one thread per element
  - Using a loop to combine the results requires N iterations

A Better Idea



This is straightforward to implement using divide-and-conquer

- Parallelism for the recursive calls

Halve and make new thread until size is at some cutoff

Combine answers in pairs as we return

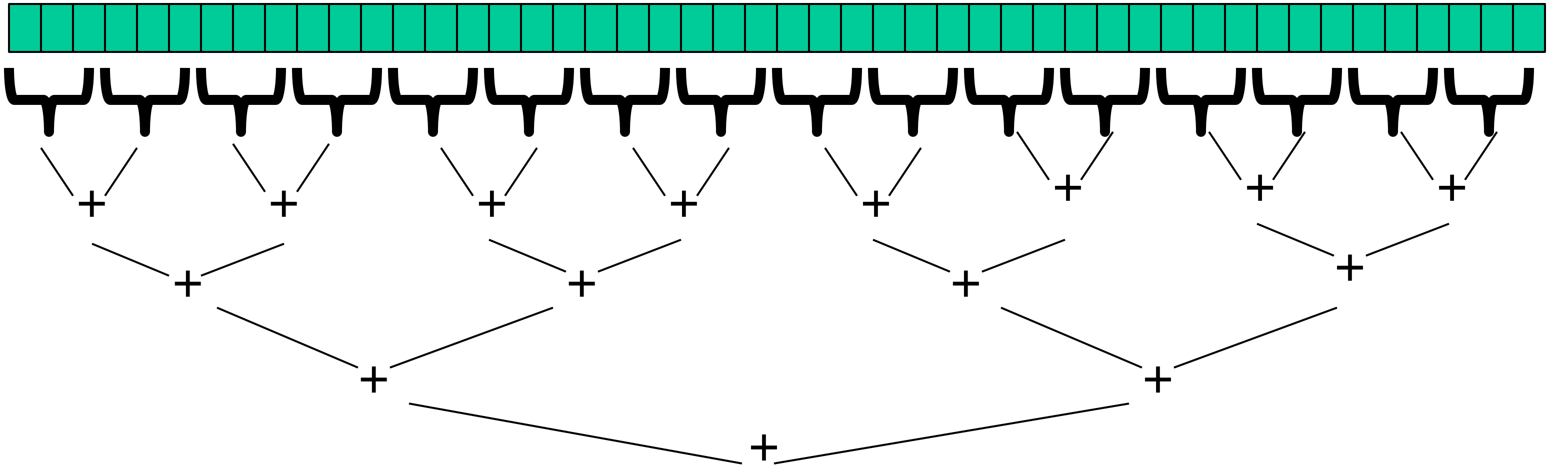
This will start small, and ‘grow’ threads to fit the problem

Divide-and-Conquer

```
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: $O(\texttt{log}\ n)$ (optimal, exponentially faster than sequential $O(n)$)
- Will write our parallel algorithms in this style
  - But using a special library designed and engineered for this style
    - Takes care of scheduling the computation well
  - Often relies on operations being associative (as with +)



Being Realistic

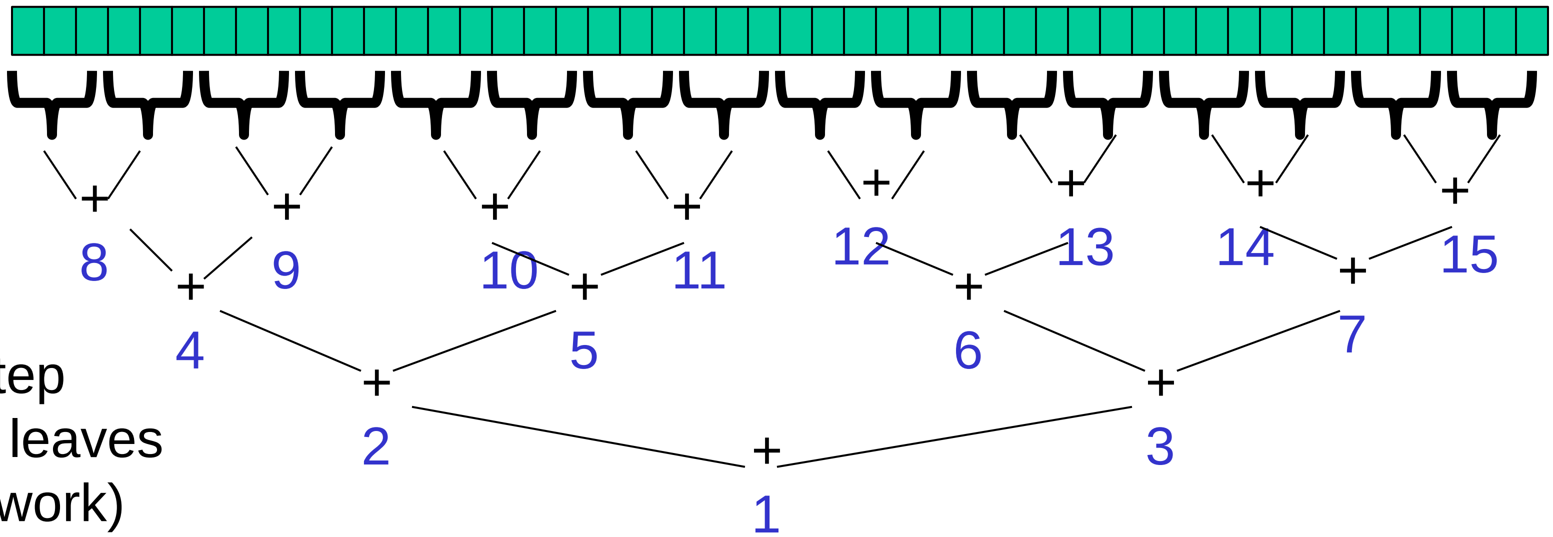
- In theory, you can divide down to single elements,
  do all your result-combining in parallel and get optimal speedup

- In practice, creating all those threads and communicating amongst them swamps the savings, so:

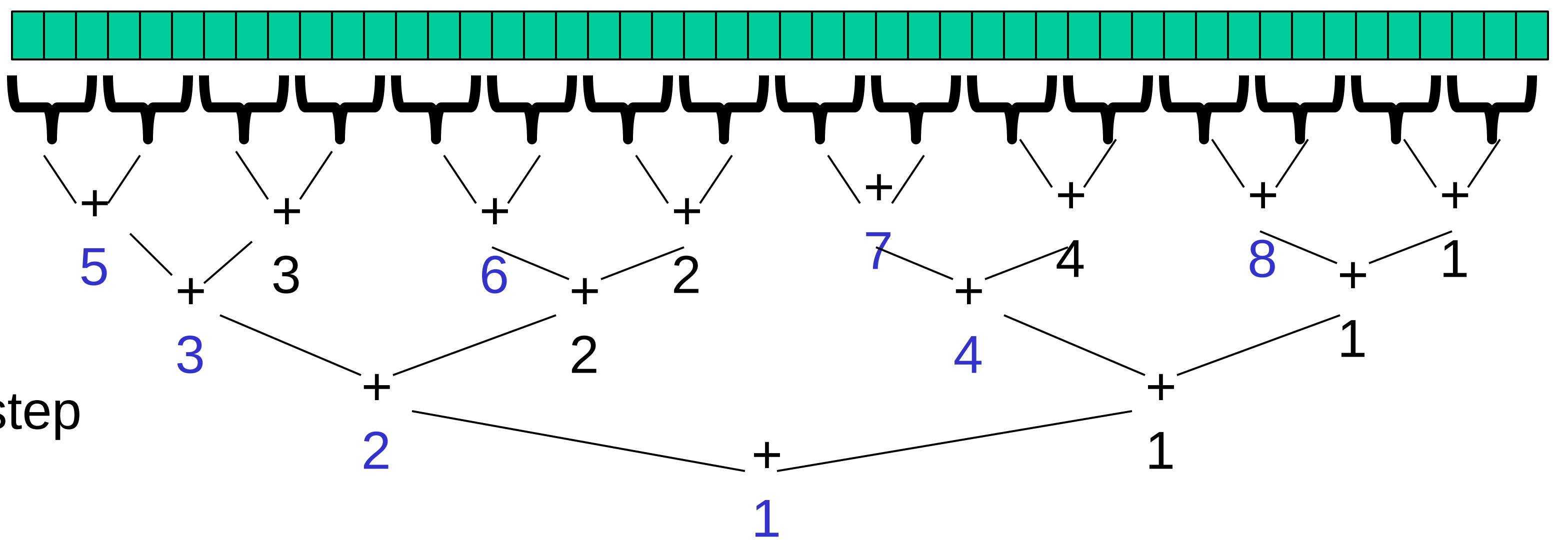
  - Use a *sequential cutoff*, typically around 500-1000
    - Eliminates *almost all* the recursive thread creation (because it eliminates the bottom levels of tree)
    - *Exactly* like quicksort switching to insertion sort for small subproblems, but more important here

Illustration of Fewer Threads

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



1 new thread at each step



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;
```

Half the Threads

Do not create two threads; create one and do the other "yourself"

- Cuts the number of threads created by 2x
- And the difference is surprisingly substantial

If a *language* had built-in support for fork-join parallelism,
we would expect this hand-optimization to be unnecessary

The *library* we are using allows you to do it yourself

- ForkJoinTask.invokeAll(…) probably does something similar
- You will do this yourselves for the same reason you implement your own data structures

But no difference in theory or asymptotic analysis

The Library

- 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 and engineered to meet the needs of divide-and-conquer fork-join parallelism
  - Included in the Java 7 standard libraries
    - Also available as a downloaded **`.jar`** file for Java 6
  - Section will discuss some pragmatics/logistics
  - Similar libraries available for other languages
    - C/C++: Cilk, Intel's Thread Building Blocks
    - C#: Task Parallel Library
  - Library implementation is an advanced topic

Different Terms but Same Basic Idea

To use the ForkJoin Framework:

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

Java Threads	ForkJoin Framework
Don't subclass **`Thread`**	Do subclass **`RecursiveTask<V>`**
Don't override **`run`**	Do override **`compute`**
Don't 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 topmost call to **`run`**	Do create a pool and call **`invoke`**
	See **`ForkJoinTask.invokeAll(…)`**

See the Dan's web page for

[“A Beginner's Introduction to the ForkJoin Framework”]

Example: Final Version in ForkJoin Framework

```
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));
}
```

For Comparison: Java Threads Version

```
class SumThread extends java.lang.Thread {
  int lo; int hi; int[] arr;//fields to know what to do
  int ans = 0; // for communicating result
  SumThread(int[] a, int l, int h) { … }
  public void run(){
    if(hi - lo < SEQUENTIAL_CUTOFF)
      for(int i=lo; i < hi; i++)
        ans += arr[i];
    else { // create 2 threads, each will do ½ the work
      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;
    }
  }
}
class C {
 static int sum(int[] arr){
   SumThread t = new SumThread(arr,0,arr.length);
   t.run(); // only creates one thread
   return t.ans;
 }
}
```

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
  - When evaluating speed, put your computations in a loop to see the “long-term benefit” after these optimizations have occurred
- 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
  - Will not focus on this, but can be crucial for parallel performance

Work and Span

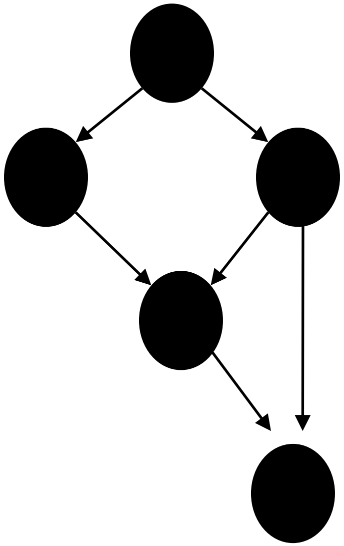
Let $\mathbf{T_P}$ be the running time if there are **P** processors available

Two key measures of run-time:

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

The DAG

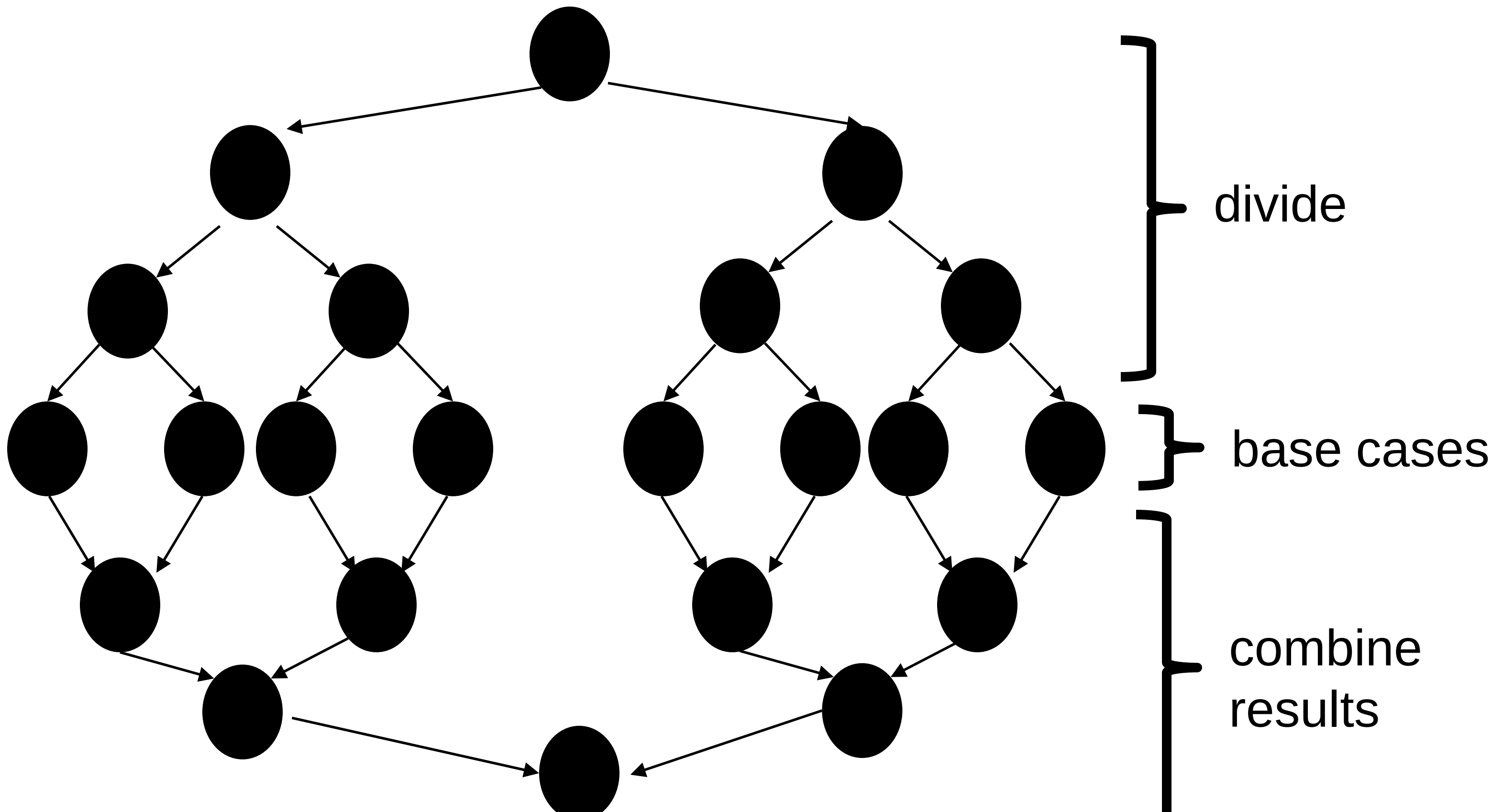
- A program execution using **`fork`** and **`join`** can be seen as a DAG
  - Nodes: Pieces of work
  - Edges: Source must finish before destination starts



- A **`fork`** "ends a node" and makes two outgoing edges
  - New thread
  - Continuation of current thread
- A **`join`** "ends a node" and makes a node with two incoming edges
  - Node just ended
  - Last node of thread joined on

Our Simple Examples

- **`fork`** and **`join`** are very flexible, but divide-and-conquer maps and reductions use them in a very basic way:
  - A tree on top of an upside-down tree

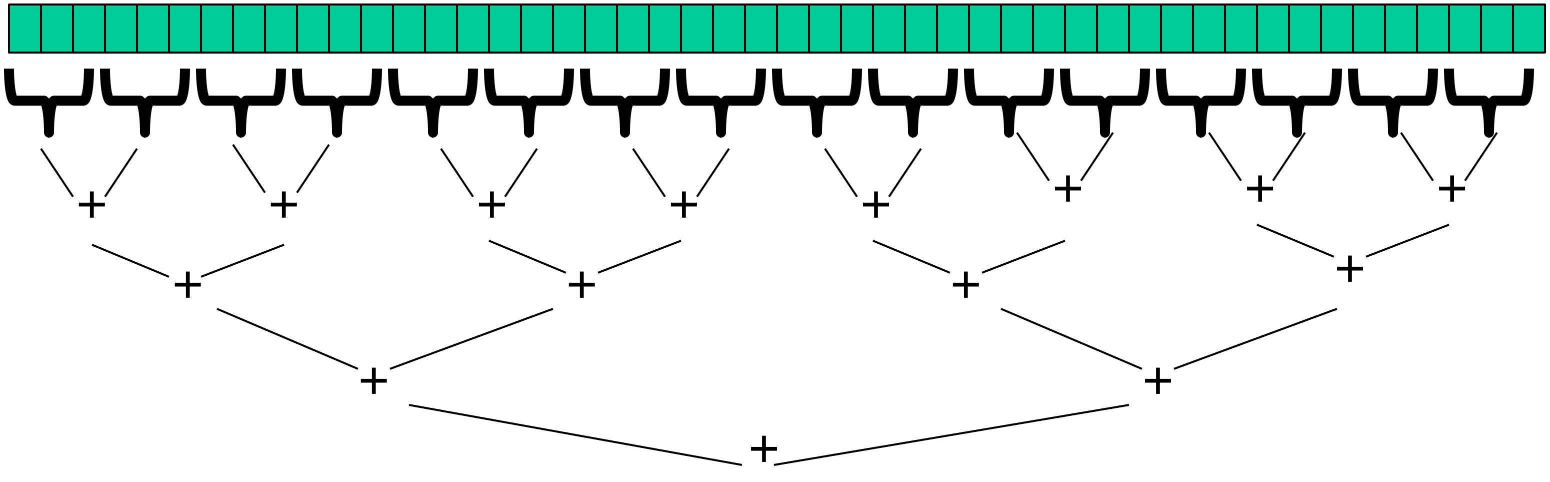


More Interesting DAGs?

- The DAGs are not always this simple
- Example:
  - Suppose combining two results might be expensive enough that we want to parallelize each one
  - Then each node in the inverted tree on the previous slide would itself expand into another set of nodes for that parallel computation

What Else Looks Like This?

- Summing an array went from $O(n)$ sequential to $O(\texttt{log } n)$ parallel (*assuming **a lot** of processors and very large n*)
  - An exponential speed-up in theory



- 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

Reductions

- Computations of this form are called reductions (or reduces?)
- Produce single answer from collection via an associative operator
  - Examples: max, count, leftmost, rightmost, sum, …
  - Non-example: median
- Recursive results don't have to be single numbers or strings. They can be arrays or objects with multiple fields.
  - Example: Histogram of test results is a variant of sum
- But some things are inherently sequential
  - How we process **`arr[i]`** may depend entirely on the result of processing **`arr[i-1]`**

Maps and Data Parallelism

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

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

Maps in ForkJoin Framework

```
class VecAdd extends RecursiveAction {
  int lo; int hi; int[] res; int[] arr1; int[] arr2;
  VecAdd(int l,int h,int[] r,int[] a1,int[] a2){ … }
  protected void compute(){
    if(hi – lo < SEQUENTIAL_CUTOFF) {
      for(int i=lo; i < hi; i++)
        res[i] = arr1[i] + arr2[i];
    } else {
      int mid = (hi+lo)/2;
      VecAdd left = new VecAdd(lo,mid,res,arr1,arr2);
      VecAdd right= new VecAdd(mid,hi,res,arr1,arr2);
      left.fork();
      right.compute();
      left.join();
    }
  }
}
static final ForkJoinPool fjPool = new ForkJoinPool();
int[] add(int[] arr1, int[] arr2){
  assert (arr1.length == arr2.length);
  int[] ans = new int[arr1.length];
  fjPool.invoke(new VecAdd(0,arr.length,ans,arr1,arr2);
  return ans;
}
```

Maps and Reductions

Maps and reductions: the “workhorses” of parallel programming

- By far the two most important and common patterns
  - We will discuss two more advanced patterns later

- Learn to recognize when an algorithm can be written in terms of maps and reductions

- Often Use maps and reductions to describe parallel algorithms

- Programming them becomes “trivial” with a little practice
  - Exactly like sequential for-loops seem second-nature

Digression: MapReduce on Clusters

- You may have heard of Google's "map/reduce"
  - Or the open-source version Hadoop
- Idea: Perform maps/reduces on data using many machines
  - The system takes care of distributing the data and managing fault tolerance
  - You just write code to map one element and reduce elements to a combined result
- Separates how to do recursive divide-and-conquer from what computation to perform
  - Old idea in higher-order functional programming transferred to large-scale distributed computing
  - Complementary approach to declarative queries for databases

Trees

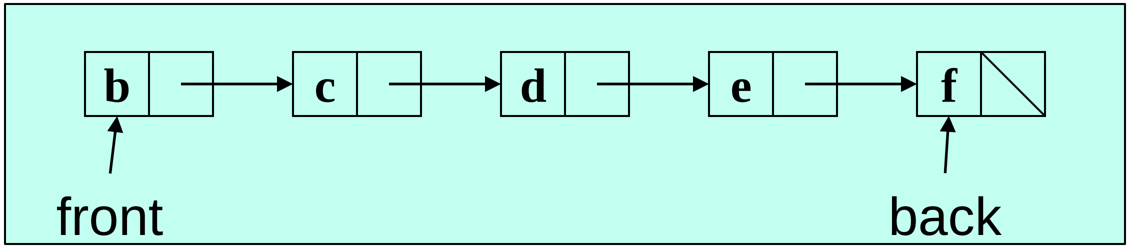
- Maps and reductions work just fine on balanced trees
  - Divide-and-conquer each child rather than array subranges
  - Correct for unbalanced trees, but won't get much speed-up

- Example: minimum element in an unsorted but balanced binary tree in $O(\texttt{log}\ n)$ time given enough processors

- How to do the sequential cut-off?
  - Store number-of-descendants at each node (easy to maintain)
  - Or could approximate it with, e.g., AVL-tree height

Linked Lists

- Can you parallelize maps or reduces over linked lists?
  - Example: Increment all elements of a linked list
  - Example: Sum all elements of a linked list



- Once again, data structures matter!

- For parallelism, balanced trees generally better than lists so that we can get to all the data exponentially faster $O(\mathtt{log}\ n)$ vs. $O(n)$
  - Trees have the same flexibility as lists compared to arrays