ForkJoin

CSE 332 Spring 2021

Instructor: Hannah C. Tang

Teaching Assistants:

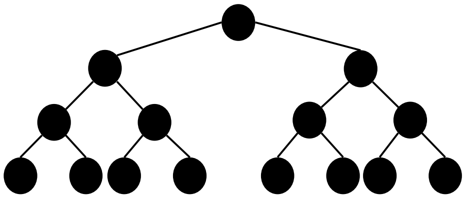
Aayushi Modi	Khushi Chaudhari	Patrick Murphy
Aashna Sheth	Kris Wong	Richard Jiang
Frederick Huyan	Logan Milandin	Winston Jodjana
Hamsa Shankar	Nachiket Karmarkar	

gradescope

gradescope.com/courses/256241

- Assume a perfect tree with *n* leaves
  - What is the height of the tree (as a function of n)? log n
  - How many internal nodes does this tree have (as a function of n)? n − 1

N leaves →



Announcements

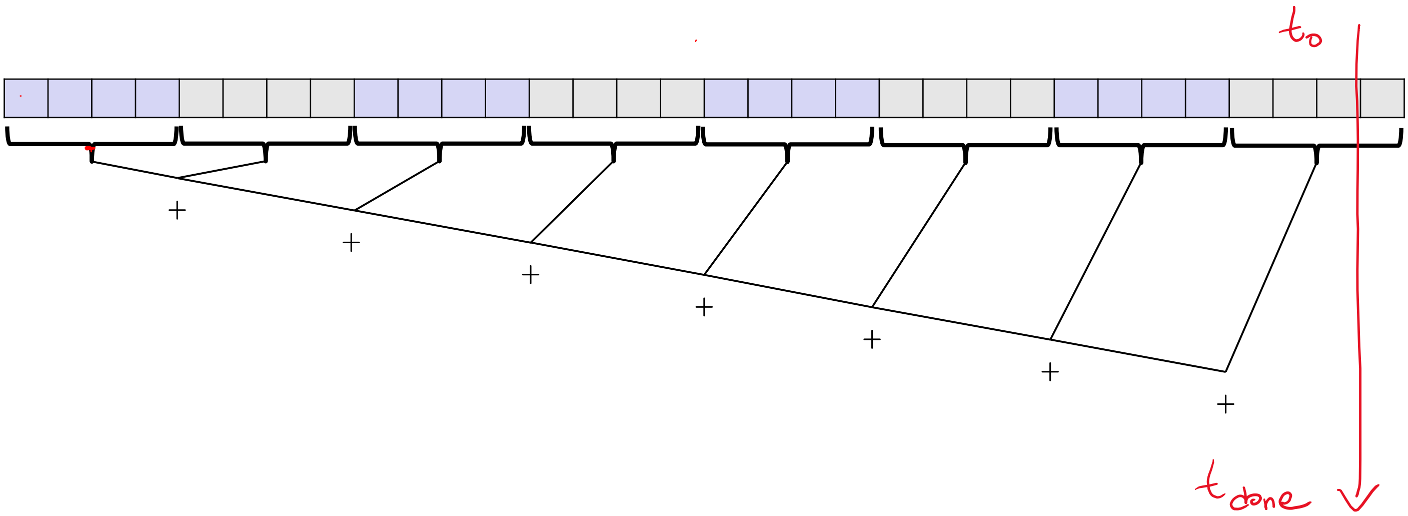
- "Para" mini-projects released!

Lecture Outline

- Concurrency Frameworks in Java
  - **Improving java.lang.Thread's constants**
  - ForkJoin Library
- More examples of parallel programs
  - Common patterns: reduce and map
  - Non-array inputs
- Asymptotic Analysis for Fork/Join-style Parallelism
- Amdahl's Law: Is the  half-empty or half-full?

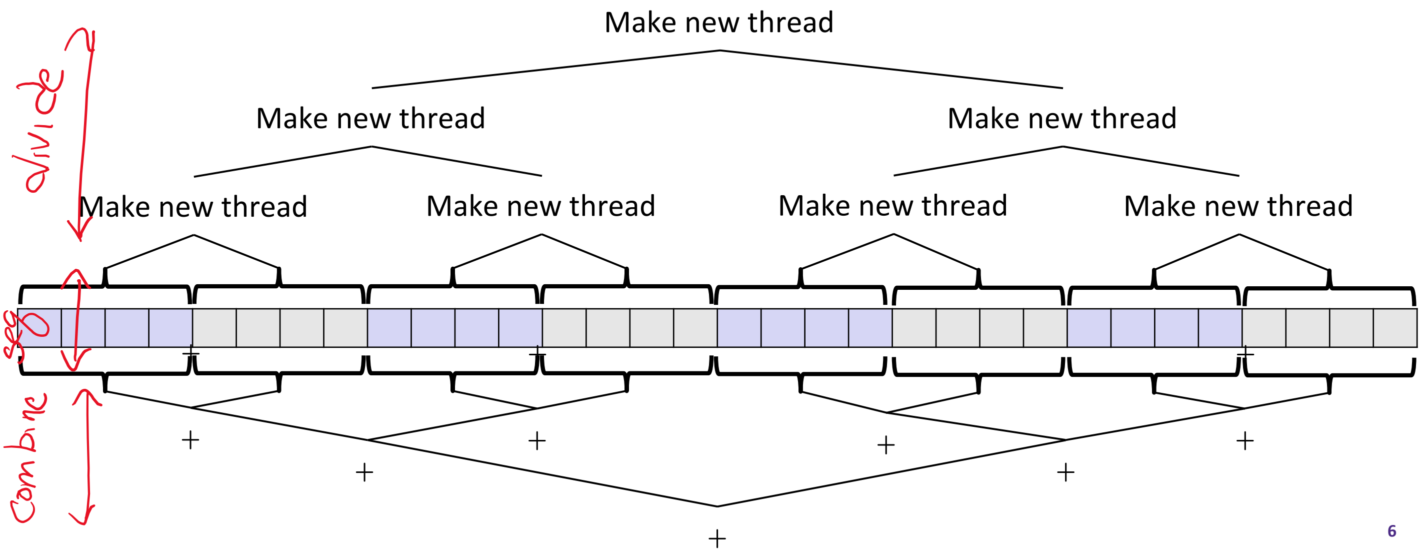
Why Fork/Join-style parallelism model? (1 of 2)

- Solve the result-combining bottleneck
  - The calls to `run()` can execute in parallel, but combining intermediate results is still sequential!



Why Fork/Join-style parallelism? (2 of 2)

- Fork/Join Phases:
  1. Divide the problem
     - Start with full problem at root
     - Make two new threads, halving the problem, until size is at cutoff
  2. Combine answers as we return from recursion



Fork/Join-style Parallelism: Code (1 of 2)

```
class SumThread extends java.lang.Thread {
  // … member fields and constructors elided …
  public void run() {  // override: implement “main”
    if(hi - lo < SEQUENTIAL_CUTOFF) {
      // Just do the calculation in this thread
      for (int i=lo; i < hi; i++)
        ans += arr[i];
    }
    else {
      // Create two new threads to calculate the left and right sums
      SumThread left = new SumThread(arr, lo, (hi+lo)/2);
      SumThread right= new SumThread(arr, (hi+lo)/2, hi);
      left.start();
      right.start();

      // Combine their results
      left.join();
      right.join();
      ans = left.ans + right.ans;
    }
  }
}
```

sequential

divide

combine

Fork/Join-style Parallelism: Code (2 of 2)

```
class SumThread extends java.lang.Thread {
  int lo, int hi, int[] arr; // input: arguments
  int ans = 0;               // output: result
  SumThread(int[] a, int l, int h) { … }
  public void run(){ … }     // override: implement “main”
}
```

```
int sum(int[] arr) {
  SumThread t = new SumThread(arr, 0, arr.length);// just 1 obj since
  t.run();                                        // we don’t need
  return t.ans;                                   // parallelism to
}                                                 // start recursion
```

- What’s up with the sequential cutoff?
  - QuickSort and MergeSort switch to InsertionSort because “the constants are better”
  - Similarly, Fork/Join-style parallelism switches to sequential execution because “the constants are better”

Performance Tuning Our Constants

- Expensive "constant time" operations include:
  - Accessing "lower tiers" of the memory hierarchy
    - Won't focus on this, but crucial for parallel performance
  - Thread-creation and thread-joining

- *In theory*, can divide down to single elements, do all the result-combining in parallel, and get optimal speedup
  - Total time: O(n / numExecutors + log n)
- *In practice*, thread creation/joins eat into the savings ☹

- Remember: computers are getting more parallel, not faster
  - attu6 has 4 CPUs with 14 cores each = 56 "processors"

gradescope

gradescope.com/courses/256241

- Assume that thread creation and joining are expensive. Which of the following optimizations might improve our constants?
  1. Use a cutoff, after which computation proceeds sequentially
  2. Somehow create fewer threads during the recursion
  3. Somehow reuse threads when they're done
  4. Use "hardware-backed threads" instead of "software threads"

Being Pragmatic #1: Sequential Cutoff

- If thread-creation and thread-joining are expensive, what can we do?
  1. Use a cutoff, after which computation proceeds sequentially
     - Cutoff value depends on type of computation; 1000-5000 machine instructions is a good start
     - Eliminates *almost all* the recursive thread creation (bottom levels of tree)
     - *Exactly* like MergeSort switching to InsertionSort, but more important here

Being Pragmatic #2: Fewer "Intermediate" Threads

- If thread-creation and thread-joining are expensive, what can we do?
  2. Do not create *two* recursive threads; create one thread and do the other piece of work "yourself"
     - Halves the number of threads created (?!?!)

Halving the Created Threads: Code

- If the *language* had built-in support for fork/join-style parallelism, this hand-optimization would be unnecessary
- But the *library* we're using expects you to do it yourself
  - … and the difference is surprisingly substantial
- Again: no difference in theory, "only" the constants

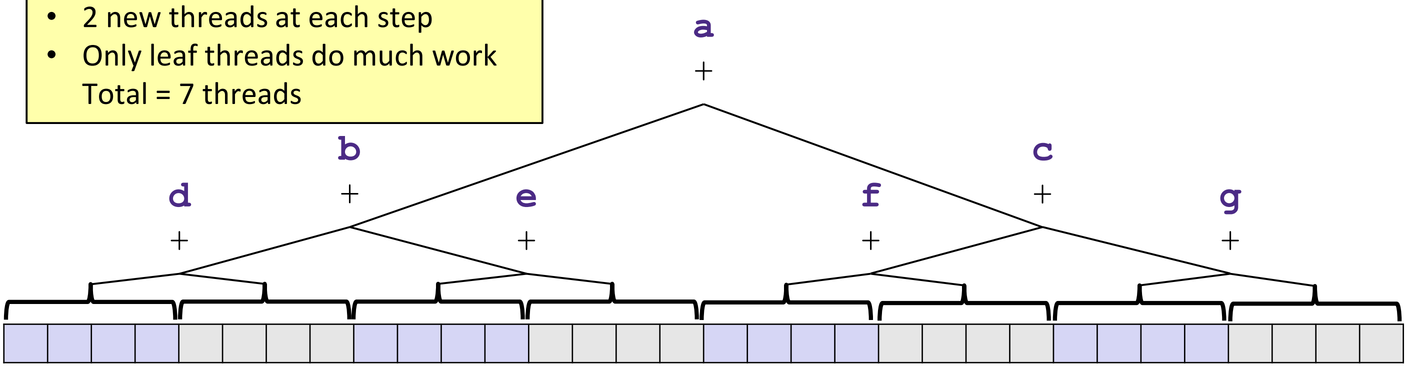
run() is a nomal function call! Execution won't proceed until it completes

```
// Don't do this:
SumThread left  = …
SumThread right = …

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

left.join();
right.join();
ans = left.ans + right.ans;
```

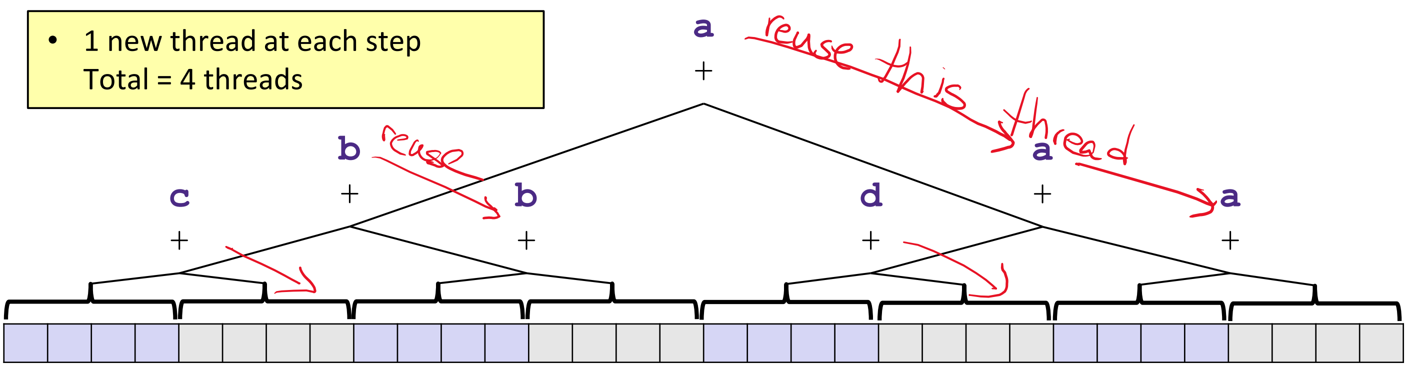
```
// Do this instead:
SumThread left  = …
SumThread right = …

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

left.join();
// no right.join() needed
ans = left.ans + right.ans;
```

Halving the Created Threads: Pictorially

- 2 new threads at each step
- Only leaf threads do much work
  Total = 7 threads



- 1 new thread at each step
  Total = 4 threads



Lecture Outline

- Concurrency Frameworks in Java
  - Improving java.lang.Thread's constants
  - **ForkJoin Library**
- More examples of parallel programs
  - Common patterns: reduce and map
  - Non-array inputs
- Asymptotic Analysis for Fork/Join-style Parallelism
- Amdahl's Law: Is the  half-empty or half-full?

Finally! The ForkJoin Library

- Even using fork/join-style code, java.lang.Thread is still too "heavyweight"
  - Constant factors, especially space overhead
  - Creating 20,000 Java threads just a bad idea ☹
- So use the **ForkJoin Library** instead
  - Introduced in Java 8 (2014)
  - Similar libraries available for other languages
    - C/C++: Cilk (inventors), Intel's Thread Building Blocks
    - C#: Task Parallel Library
    - …
  - Its implementation is a fascinating but advanced topic

Thread -> ForkJoin: Terminology

Java Built-in Threads	ForkJoin Library
Subclass `Thread`	Subclass `RecursiveTask<V>`
Override `run()`	Override `compute()`
Call `start()` to begin parallel computation	Call `fork()` to begin parallel computation
Return results via member fields (eg, `ans`)	Return results via return value (ie, an instance of `V`)
Call `join()`, then check its "returned" member field	Call `join()`, then check its return value
Halve created threads by calling `run()` directly	Halve created threads by calling `compute()` directly
Begin recursion with top-level call to `run()` (instead of `start()`)	Begin recursion by creating a `ForkJoinPool` and calling its `invoke()`

Fork/Join-style Parallelism with ForkJoin (1 of 2)

```
class SumTask extends RecursiveTask<Integer> {
  int lo; int hi; int[] arr;      // just the "input" arguments!

  protected Integer compute() {  // override: implement "main"
    if(hi - lo < SEQUENTIAL_CUTOFF) {
      // Just do the calculation in this thread
      int ans = 0;    // local variable instead of a member field
      for (int i=lo; i < hi; i++)
        ans += arr[i];
      return ans;     // direct return of answer
    } else {
      // Create ONE new thread to calculate the left sum
      SumTask left  = new SumTask(arr, lo, (hi+lo)/2);
      SumTask right = new SumTask(arr, (hi+lo)/2, hi);
      left.fork();    // create a thread and call its compute()
      int rightAns = right.compute(); // call compute() directly

      // Combine results
      int leftAns  = left.join();
      return leftAns + rightAns;
    }
  }
}
```

Only need to change this; all else is "boilerplate"

Fork/Join-style Parallelism with ForkJoin (2 of 2)

```
class SumTask extends RecursiveTask<Integer> {
  int lo; int hi; int[] arr;          // input: arguments
  SumTask(int[] a, int l, int h) { … }
  protected Integer compute() { … } // override: implement "main"
}
```

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

int sum(int[] arr) {
  SumTask task = new SumTask(arr, 0, arr.length);

  // invoke() returns the value which is returned by the
  // top-level compute()
  return POOL.invoke(task);
}
```

ForkJoin Library: Tips

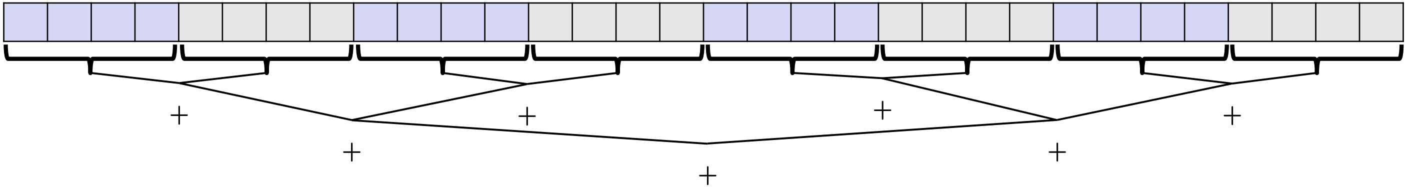
- Sequential threshold
  - Library documentation recommends doing approximately 1000-5000 basic operations in each "piece" of your algorithm
- ForkJoin library needs to "warm up"
  - May see slow results before JVM re-optimizes the library internals
  - Put computations in a loop to see the "long-term benefit"

Lecture Outline

- Concurrency Frameworks in Java
  - Improving java.lang.Thread's constants
  - ForkJoin Library
- More examples of parallel programs
  - **Common patterns: reduce and map**
  - Non-array inputs
- Asymptotic Analysis for Fork/Join-style Parallelism
- Amdahl's Law: Is the half-empty or half-full?

A Common Pattern

- Summing went from $O(n)$ sequential to $O(\texttt{log}\ n)$ parallel
  - Assuming **a lot** of processors and very large n
  - Exponential speed-up in theory: n / $\texttt{log}\ n$ grows exponentially

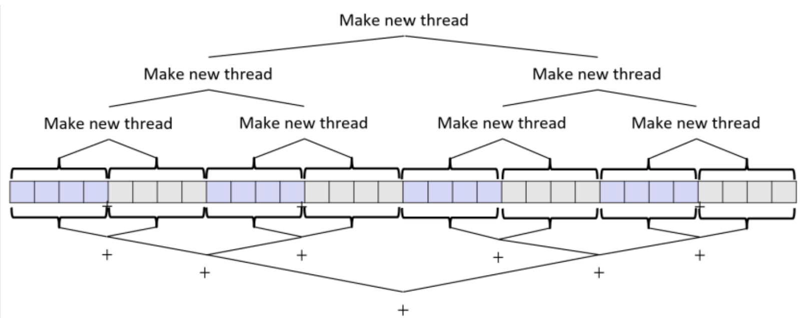


- Any solution which can merge two subsolutions in $O(1)$ time has this property!

- Just need to "plug in" 2 parts:
  - How to compute the result at the cut-off
    (Parallel-Sum: Iterate through sequentially and add up)
  - How to merge results
    (Parallel-Sum: Just add 'left' and 'right' results)

Examples of our Common Pattern

- Assume the input is an array; how would we do the following?
  1. Maximum or minimum element
  2. Is there an element satisfying some property (e.g., is there a 17)?
  3. Left-most element satisfying some property (e.g., first 17)
  4. Smallest rectangle encompassing a number of points
  5. Counts; for example, number of strings that start with a vowel
  6. Are these elements in sorted order?



A Common Pattern: Reductions

- This class of computations are called **reductions**
  - We 'reduce' or summarize a large array of data to a single final result
  - Intermediate results must be combined with an **associative operator**
  - *Examples*: max, count, leftmost, rightmost, sum, product, …
- Intermediate and final results can be "aggregates": arrays or multi-field objects
  - *Example*: histogram from a much larger array of test results
- Some things are inherently sequential
  - *Example*: `arr[i]`'s is the sum of `arr[1]`…`arr[i-1]`

Another Common Pattern: Maps

- A **map** transforms each element of a collection independently, creating a new-but-same-sized collection of modified elements
  - No combining results
- *Example*: Vector addition

```
int[] vectorAdd(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;
}
```

- Just need to "plug in" one part:
  - How to map element E to transformed E'
  - *(Vector-add: generate result[i] from arr1[i])*

Maps in the ForkJoin Library (1 of 2)

- Many small tasks still helps with load balancing
  - Maybe not for vector-add, but definitely for compute-intensive maps
  - The forking is O(log n); theoretically other approaches are O(1)

```
class VectorAdd extends RecursiveAction {
  // input: arguments
  int lo; int hi; int[] res; int[] v1; int[] v2;

  protected void compute(){
    if(hi - lo < SEQUENTIAL_CUTOFF) {
      for(int i=lo; i < hi; i++)
        res[i] = v1[i] + v2[i];
    } else {
      int mid = (hi+lo)/2;
      VectorAdd left = new VectorAdd(lo, mid, res, v1, v2);
      VectorAdd right= new VectorAdd(mid, hi, res, v1, v2);
      left.fork();
      right.compute();
      left.join();
    }
  }
}
```

Maps in the ForkJoin Library (2 of 2)

```
class VectorAdd extends RecursiveAction {
  // input: arguments
  int lo; int hi; int[] res; int[] v1; int[] v2;

  protected void compute() { … } // override: implement "main"
}
```

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

int[] add(int[] arr1, int[] arr2){
  assert (arr1.length == arr2.length);

  // Use ans as an "output argument" instead of looking at the
  // top-level compute()'s return value (which is void).
  int[] ans = new int[arr1.length];

  POOL.invoke(new VectorAdd(0, arr.length, ans, arr1, arr2);
  return ans;
}
```

Map and Reduce in the ForkJoin Library

- Map (vector-add)
  - `VectorAdd` extended `RecursiveAction`
  - Result was an output parameter; nothing returned from `compute()`

- Reduce (parallel-sum):
  - `SumTask` extended `RecursiveTask`
  - Result directly returned from `compute()`

- … but it doesn't *have* to be this way
  - Map could've used `RecursiveTask` to return an array
  - Reduce could've used `RecursiveAction` and returned result as an output parameter

Maps and Reductions, Generally

- Maps and reductions are the “workhorses” of parallel programming
  - By far, the two most important and common patterns
    - Two more-advanced patterns in next lecture
- Goal:
  - Recognize when an algorithm can use maps and reductions
  - Use maps and reductions to describe (parallel) algorithms
- Result: programming them becomes “trivial”
  - Exactly like sequential for-loops seem second-nature nowadays

Digression: MapReduce on clusters

- You may have heard of Google's "map/reduce"
  - Or the open-source version, Hadoop
- Performs maps and reduces using many machines
  - System takes distributes input data and manages fault tolerance
  - You just write code to map one element and reduce elements to a combined result
- Separates how the recursive divide-and-conquer "frame" from the computation to perform
  - An old idea in higher-order functional programming, transferred to large-scale distributed computing
  - Complementary approach to declarative queries for databases

Lecture Outline

- Concurrency Frameworks in Java
  - Improving java.lang.Thread's constants
  - ForkJoin Library
- More examples of parallel programs
  - Common patterns: reduce and map
  - **Non-array inputs**
- Asymptotic Analysis for Fork/Join-style Parallelism
- Amdahl's Law: Is the half-empty or half-full?

Parallelized Computation on Trees

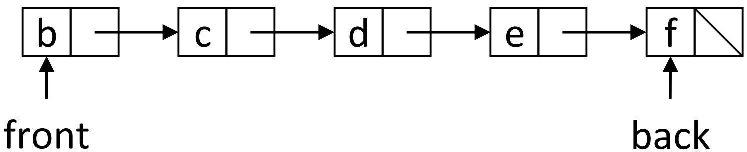
- Maps and reductions work on trees
  - Divide-and-conquer each child rather than array sub-ranges
  - Correct for unbalanced trees, but won't get much speed-up unless tree is balanced

- *Example*: minimum in an *unsorted-*but-balanced binary tree
  - $O(\texttt{log}\ n)$ 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

Parallelized Computation on 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



- Parallelism still helps with expensive per-element operations

- Once again, data structures matter!
  - Balanced trees allow faster access to all the data: O(log n) vs. O(n)
  - Trees and lists have the same flexibility compared to arrays (eg, inserting an item in the middle of the list)

Lecture Outline

- Concurrency Frameworks in Java
  - Improving java.lang.Thread's constants
  - ForkJoin Library
- More examples of parallel programs
  - Common patterns: reduce and map
  - Non-array inputs 🏃💨
- **Asymptotic Analysis for Fork/Join-style Parallelism**
- Amdahl's Law: Is the 🥛 half-empty or half-full?

Analyzing Parallel Algorithms

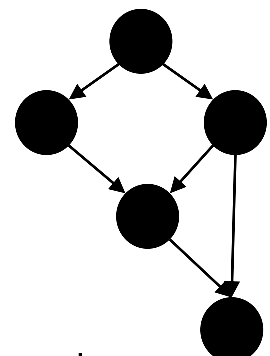
- How to measure efficiency?
  - Want asymptotic bounds
  - Want an analysis that's independent of a specific number of processors
- Fork/Join parallelism gets *asymptotically optimal* runtime for the available number of processors
  - So we can analyze algorithms assuming this guarantee

Modelling Fork/Join Parallelism with DAGs

- A program execution using can be modeled as a DAG
  - Nodes: Pieces of work
  - Edges: Source must finish before destination can start
  - Costs are in the nodes, not the edges!

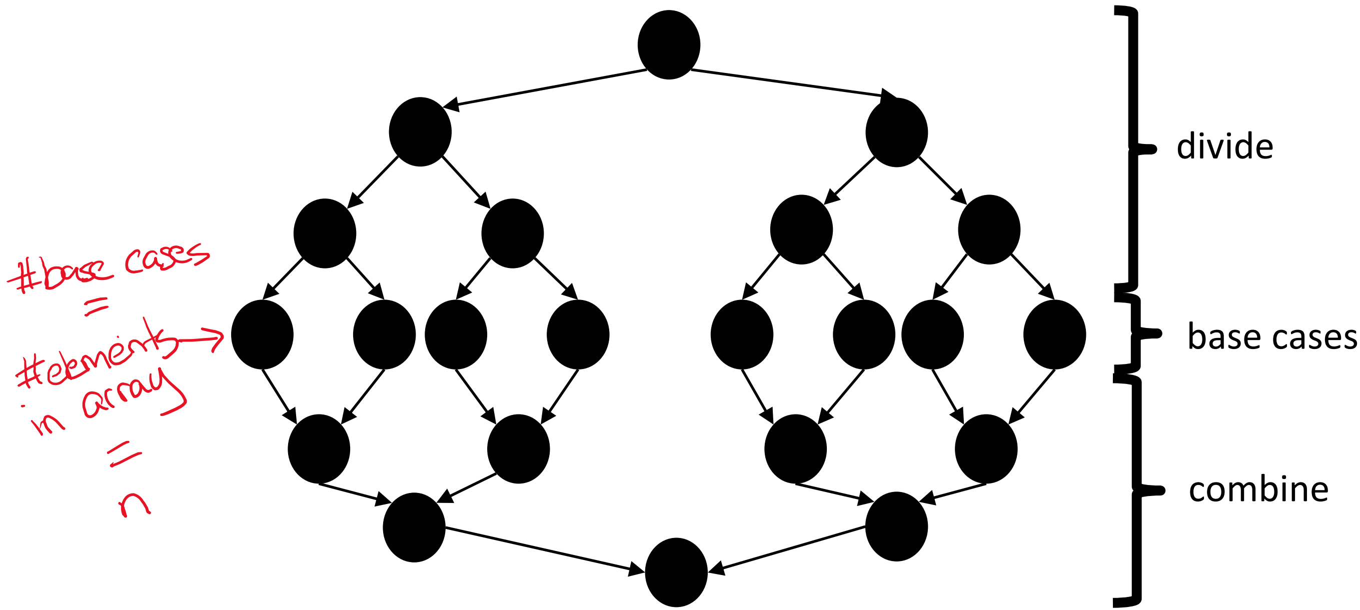
A directed acyclic graph (DAG) is:
- A graph that is directed (edges have direction/arrows)
- And whose edges do not create a cycle (ability to trace a path that starts and ends at the same node)

- A **`fork`** makes two outgoing edges:
  - New thread
  - Continuation of current thread
- A **`join`** takes two incoming edges
  - The final node of the joined thread
  - The computation that just finished in the current thread



Our Simple Examples

- Maps and reductions use **`fork`** and **`join`** in a very basic way: as a (perfect) tree on top of an upside-down (perfect) tree
  - Constant amount of processing at each node: O(1)



Aside: More Interesting DAGs?

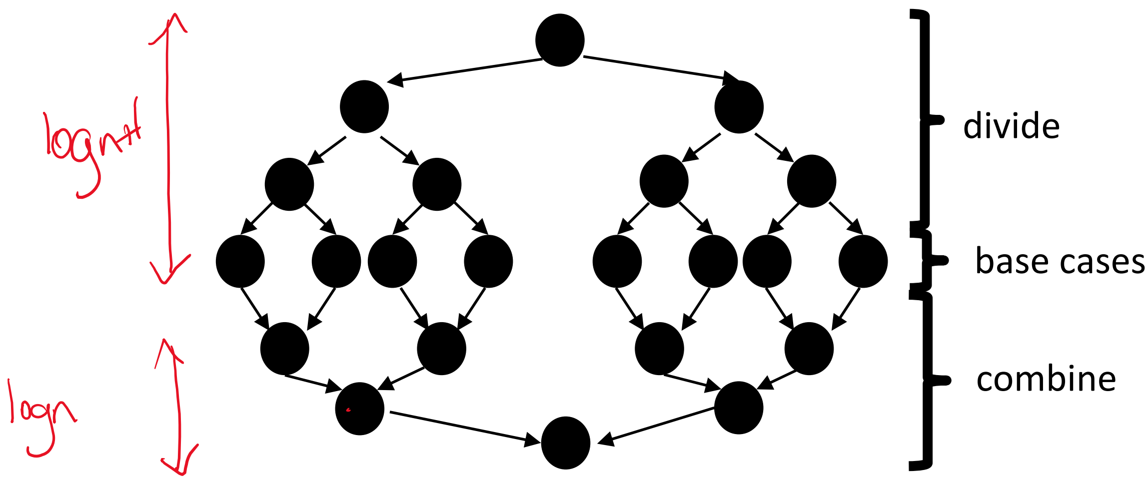
- The execution DAGs are not always this simple
  - *Example*: combining results might so expensive that we parallelize it. Then each node in the *inverted* tree would expand into another set of nodes for that parallel computation

Definitions: Work and Span

- Let $\mathbf{T_P}$ be the *running time* if there are **P** *processors* available
- Two important definitions:
  - **Work**: How long it would take with 1 processor (ie, $\mathbf{T_1}$)
    - Just "sequentialize" the recursive forking
    - Cumulative work that all processors must complete
  - **Span**: How long it would take with infinitely many processors (ie, $\mathbf{T_\infty}$)
    - The hypothetical ideal; aka "critical path length" or "computational depth"
    - This is the longest "dependence chain" in the computation
    - *Example*: $O($`log` $n)$ for summing an array
      - Notice how having $>n/2$ processors doesn't reduce the span

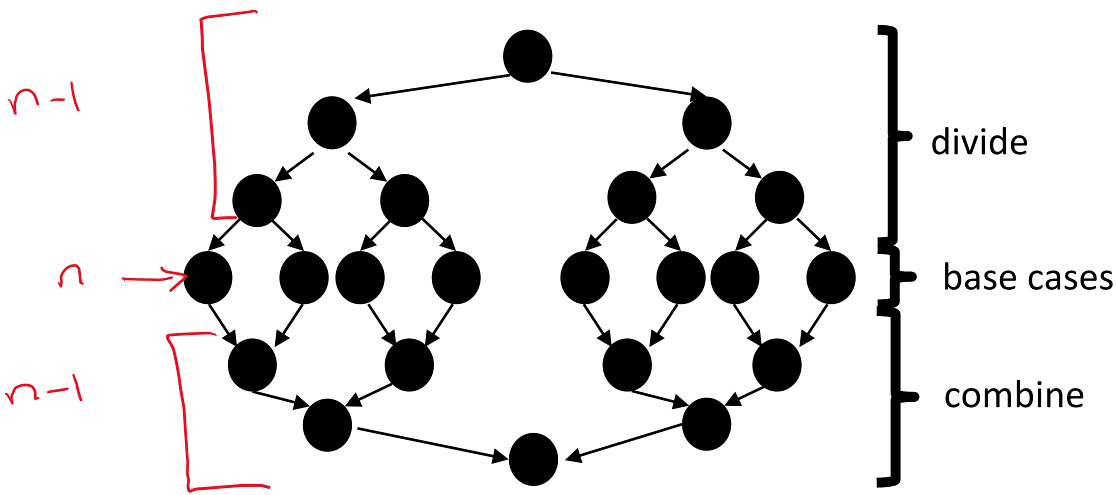
Definitions Applied to Maps/Reductions (1 of 2)

- In this context, the **span** (T_∞) is:
  - The longest dependence-chain; i.e., longest 'branch' in parallel 'tree'
  - $T_\infty \in O($ log n $)$ for simple maps and reductions



Definitions Applied to Maps/Reductions (2 of 2)

- And the **work** (T_1) is:
  - The sum of runtime of all nodes in the DAG
  - $T_1 \in O($ n $)$ for simple maps and reductions



Span = $\mathbf{T_\infty}$ = sum of runtime of all nodes in the DAG's *most-expensive path*

Work = $\mathbf{T_1}$ = sum of runtime of all nodes in the DAG

More Definitions: Speed-up

- **Speed-up** using **P** processors: $\mathbf{T_1 / T_P}$
  - Example: $T_1 = 100$ and $T_4 = 50$
- If speed-up = **P** as we vary **P**, we call it **perfect linear speed-up**
  - Example: $T_1 = 100$ and $T_4 = 25$
- Perfect linear speed-up means doubling **P** halves running time
  - Usually our goal, but hard to get in practice

Span = T_∞ = sum of runtime of all nodes in the DAG's *most-expensive path*

Work = T_1 = sum of runtime of all nodes in the DAG

Speed-up = T_1 / T_P

Last Definition: Parallelism

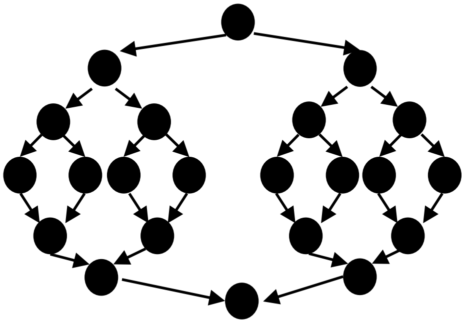
- **Parallelism**: T_1 / T_∞ is the maximum possible speed-up; the point at which adding executors doesn't help
  - Depends on the span!

P	T_P	Speedup
1	T_1=100	-
2	T_2=50	2
10	T_{10}=25	4
50	T_{50}=22	4.54
100	T_{100}=20.5	4.87
∞	T_∞=20	5

Parallel algorithms attempt to decrease span without increasing work too much

Obtaining Optimality for T_P

- What is the asymptotically optimal $\mathbf{T_P}$, for any value of **P**?
  - (as usual, we ignore memory-hierarchy issues; i.e. caching)



- We know $\mathbf{T_P}$ is greater than or equal to:
  - $\mathbf{T_1 / P}$ *(why?)* perfect linear speedup!
  - $\mathbf{T_\infty}$ *(why?)* span!

- So an *asymptotically optimal* execution must be:

$$O\left((T_1/P) + T_\infty \right)$$

  - First term dominates for small **P**, second for large **P**

Optimal T_P: Thanks, ForkJoin library!

- The ForkJoin library gives an *expected-time guarantee* of asymptotically optimal!
  - "Expected time" because it flips coins when scheduling

- To obtain this guarantee, our job as ForkJoin library users is to make all the nodes in our execution DAG *small-ish* and *approximately equal*

- In exchange, the library-writers:
  - Assign work to avoid **idling**; we can ignore **scheduling** issues
  - Keep constant factors low
  - Honor the **expected-time optimal guarantee** of $\mathbf{T_P = \mathit{O}((T_1/P) + T_\infty)}$ for your hardware