

Fork/Join Parallelism and Its Analysis

CSE 332 Summer 2020

Instructor: Richard Jiang

Teaching Assistants:

Hamsa Shankar Kristin Li Winston Jodjana

Maggie Jiang Hans Zhang Michael Duan

Jeffery Tian Annie Mao

Lecture Q&A: pollev.com/332summer

Lecture clarifications: tinyurl.com/332-07-27A

Announcements

- ❖ Ex 10-13 released on Gitlab
- ❖ Project 3 released tomorrow
 - You will need to fill out partner surveys one more time!

pollev.com/332summer :: tinyurl.com/332-07-27A

Lecture Outline

- ❖ Concurrency Frameworks in Java
 - Improving `java.lang.Thread`
 - **Asymptotically**
 - Constants
 - ForkJoin Library

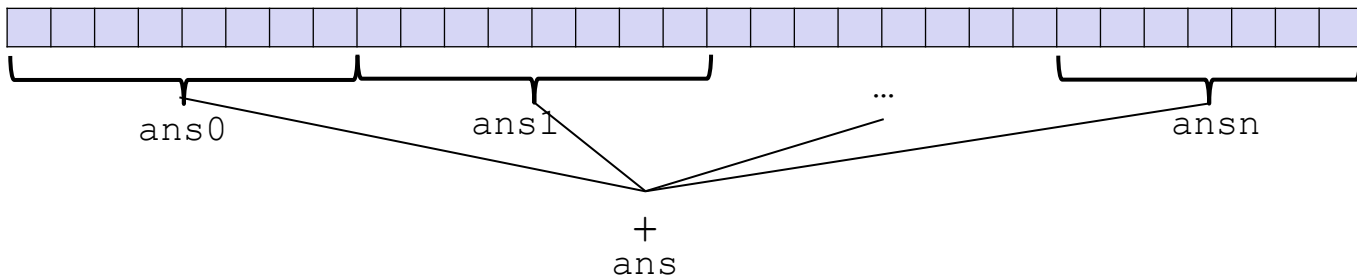
- ❖ More examples of parallel programs
 - Common patterns: reduce and map
 - Non-array inputs

- ❖ Asymptotic Analysis for Fork/Join-style Parallelism

pollev.com/332summer :: tinyurl.com/332-07-27A

Review: Many Small Chunks

- ❖ The solution: *cut up our problem into many small chunks*
 - We want far more chunks than the number of processors!
 - ... but this will require changing our algorithm



1. *Portable?* Yes! (Substantially) more chunks than processors
2. *Adapts to Available Processors?* Yes! Hand out chunks as you go
3. *Load Balanced?* Yes(ish)! Variation is smaller if chunks are small

Naïve Thread Creation/Joining Algorithm

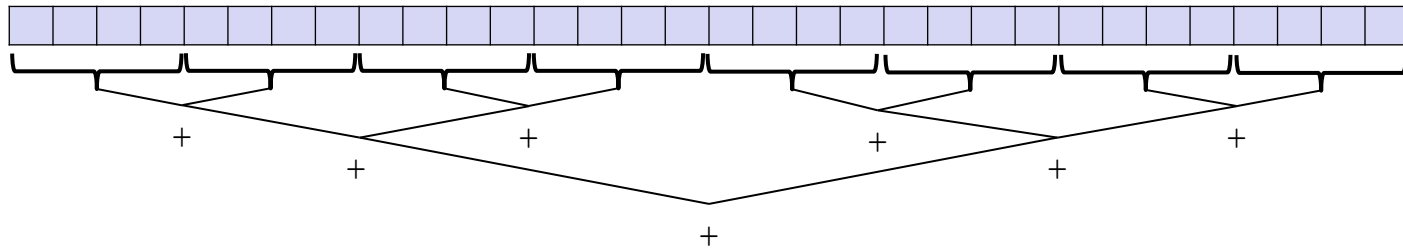
- ❖ Suppose we create 1 thread to process every 1000 elements

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

- ❖ “Combine results” part has `arr.length/1000` additions
 - $\Theta(N)$ to combine!
 - Previously, we had only 4 pieces ($\Theta(1)$ to combine)
- Will a $\Theta(N)$ algorithm to create threads/combine results be a bottleneck?

Smarter Thread Creation/Joining: Divide and Conquer!

- ❖ Divide and Conquer:
 - “Grows” the number of threads to fit the problem
 - Uses parallelism for the recursive calls
 - This style of parallel programming is called “fork/join”



- ❖ 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 (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(); // don't move this up a line (why?)
            right.join();
            ans = left.ans + right.ans;
        }
    }
}
```

Fork/Join-style Parallelism (2 of 2)

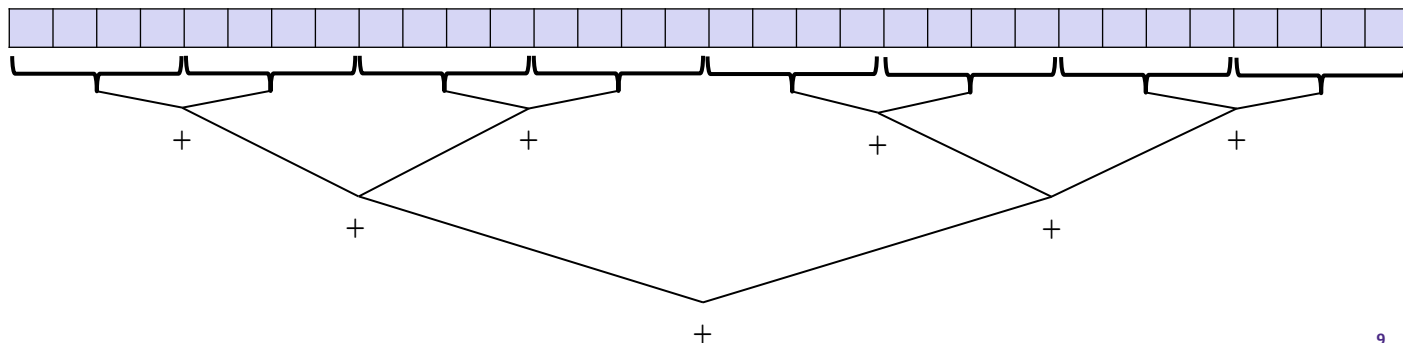
```
class SumThread extends java.lang.Thread {
    int lo, int hi, int[] arr; // input: arguments
    int ans = 0;                // output: result
    SumThread(int[] a, int l, int h) { ... }
    public void run(){ ... }    // override: implement "main"
}
```

```
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
```

- ❖ The computation and the result-combining are both in parallel
 - Using recursive divide-and-conquer makes this natural
 - Easier to write *and* more efficient asymptotically!

Fork/Join-style Parallelism Really Works!

- ❖ The key is in parallelizing both the executor-creation and the result-combining phases
 - If enough processors, runtime is **height of the tree**: $O(\log n)$
 - Optimal and exponentially faster than sequential $O(n)$
 - Relies on operations being associative (like +)
- ❖ We'll write all our parallel algorithms in this style
 - But using a special library engineered for this style



Lecture Outline

- ❖ Concurrency Frameworks in Java
 - Improving `java.lang.Thread`
 - Asymptotically
 - **Constants**
 - ForkJoin Library

- ❖ More examples of parallel programs
 - Common patterns: reduce and map
 - Non-array inputs

- ❖ Asymptotic Analysis for Fork/Join-style Parallelism

pollev.com/332summer :: tinyurl.com/332-07-27A



Poll Everywhere

pollev.com/332summer

- ❖ 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
 3. Somehow reuse threads when they're done
- A. Cutoff only
- B. Cutoff + Fewer Threads
- C. Cutoff + Thread Reuse
- D. Cutoff + Fewer Threads + Thread Reuse
- E. I'm not sure ...

Being Pragmatic #2: Constants Matter

- ❖ *In theory*, can divide down to single elements, do all the result-combining in parallel, and get optimal speedup
 - Total time: $O(n / \text{numProcessors} + \log n)$

- ❖ *In practice*, thread creation/joins eat into the savings, so:
 1. Use a cutoff, after which computation proceeds sequentially
 - Cutoff value depends on type of computation; 500-1000 is a good start
 - Eliminates *almost all* the recursive thread creation (bottom levels of tree)
 - *Exactly* like QuickSort switching to InsertionSort, but more important here
 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 normal 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;
```

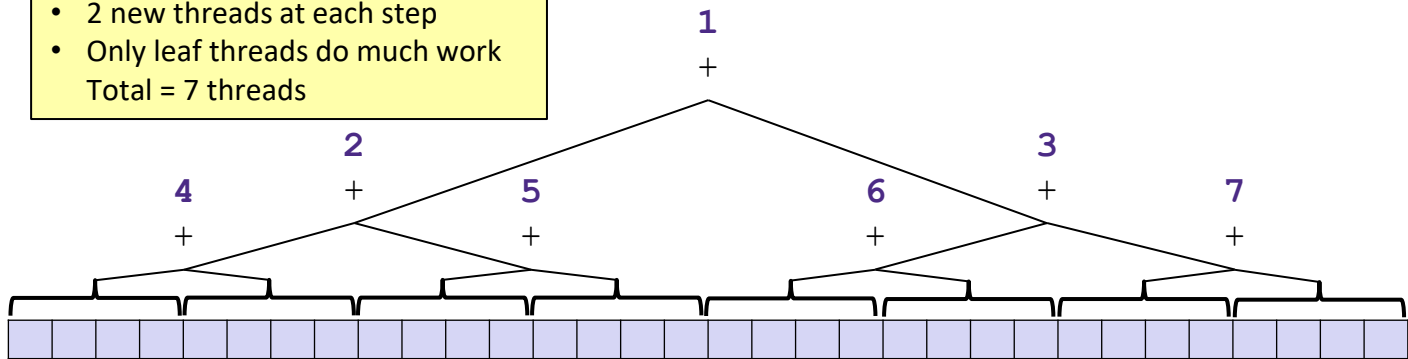
```
// Do this instead:
SumThread left = ...
SumThread right = ...

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

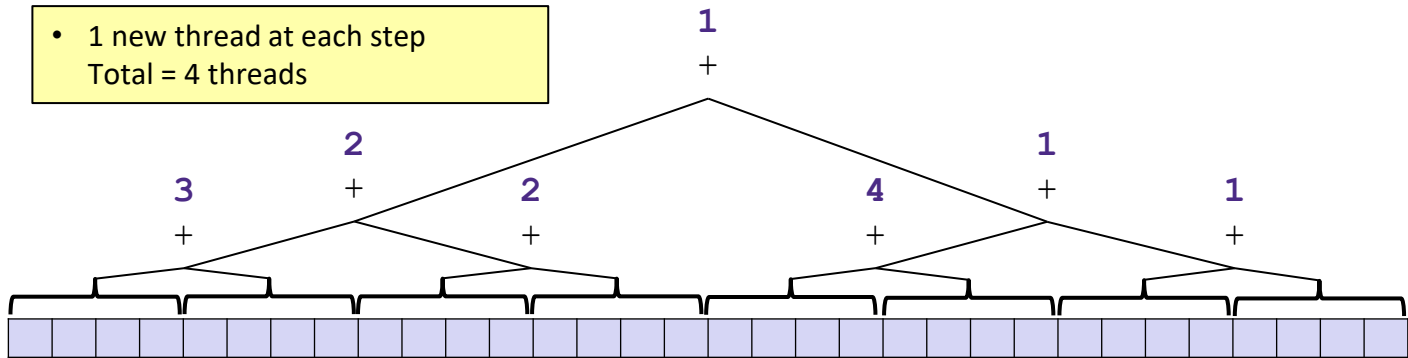
left.join();
// no right.join() needed
ans = left.ans + right.ans;
```

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`
 - Asymptotically
 - Constants
 - **ForkJoin Library**

- ❖ More examples of parallel programs
 - Common patterns: reduce and map
 - Non-array inputs

- ❖ Asymptotic Analysis for Fork/Join-style Parallelism

pollev.com/332summer :: tinyurl.com/332-07-27A

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

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;
        }
    }
}
```

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);
}
```

Being Pragmatic #2: Performance Tuning the Library

- ❖ Sequential threshold
 - Library documentation recommends doing approximately 100-5000 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”

Summary: Parallelism

- ❖ **Parallelism**: increasing efficiency/decreasing total runtime
- ❖ **Concurrency**: correctly accessing shared resources
 - They intersect when *parallel computations access shared resources*
- ❖ Model: shared memory with explicit threads:
 - **Threads** are the minimum fields necessary to represent “computation”: a program counter and a stack
 - Everything else is **shared** (eg, static variables, heap)
- ❖ Threading:
 - `run()`/`compute()` are “regular” function calls, but `start()`/`fork()` create a new thread and then call `run()`/`compute()`
 - Parallelizing many small chunks of work is portable, adaptable, and load-balancable

Summary: Fork/Join Parallelism

- ❖ **Fork/Join Parallelism** is a *model* that grows the parallelism to fit the problem size using recursion
- ❖ The **ForkJoin library** that cleanly enables this model
 - You still need to manually specify the sequential cutoff
 - Halving the created threads also requires manual intervention

Lecture Outline

- ❖ Concurrency Frameworks in Java
 - Improving `java.lang.Thread`
 - Asymptotically
 - Constants
 - ForkJoin Library

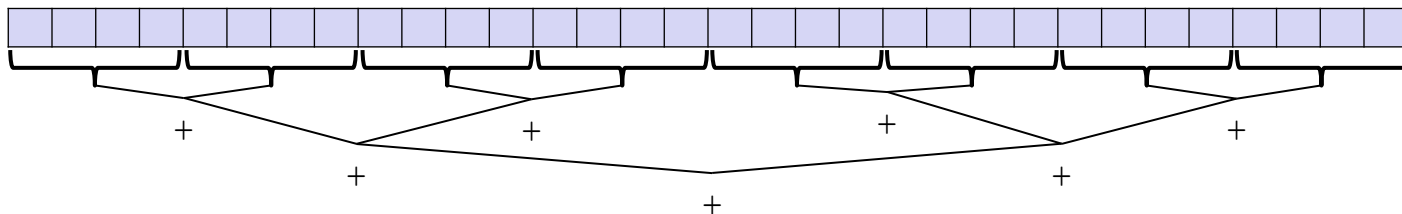
- ❖ More examples of parallel programs
 - **Common patterns: reduce and map**
 - Non-array inputs

- ❖ Asymptotic Analysis for Fork/Join-style Parallelism

pollev.com/332summer :: tinyurl.com/332-07-27A

A Common Pattern

- ❖ Summing 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)

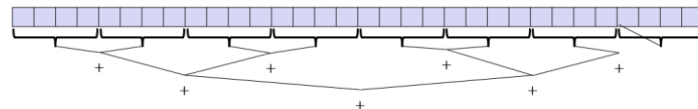


- ❖ Any solution which can merge two subsolutions in $O(1)$ time has this property! Usually just need to “plug in” 2 parts:
 - How to compute the result at the cut-off
(*Parallel-Sum: Iterate through sequentially and add up*)
 - How to merge results
(*Parallel-Sum: Just add ‘left’ and ‘right’ results*)

Examples

❖ Parallelization Pattern #1:

- How to compute result at the 'cut-off'
- How to merge results



❖ 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’ 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
- ❖ But some things are inherently sequential..
 - *Example*: How we process `arr[i]` depends entirely on the result of processing `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);

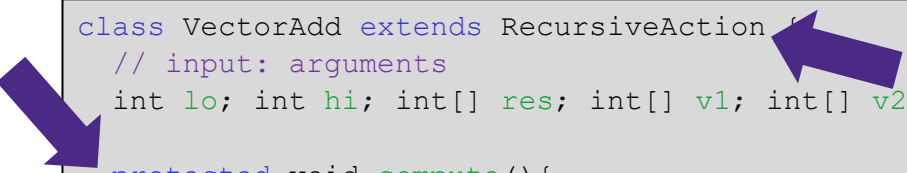
    result = new int[arr1.length];
    FORALL (i=0; i < arr1.length; i++) {
        result[i] = arr1[i] + arr2[i];
    }
    return result;
}
```

- ❖ Using a map? Only 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 $O(\log n)$; Everything else is $O(1)$

```
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
 - These are sometimes call embarrassingly parallelable programs
 - Two more-advanced patterns in next lecture

- ❖ An important learning objectives!
 - Recognize when an algorithm can use maps and reductions
 - Use maps and reductions to describe (parallel) algorithms



- ❖ Goal: 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`
 - Asymptotically
 - Constants
 - ForkJoin Library

- ❖ More examples of parallel programs
 - Common patterns: reduce and map
 - **Non-array inputs**  

- ❖ Asymptotic Analysis for Fork/Join-style Parallelism

pollev.com/332summer :: tinyurl.com/332-07-27A

Parallelized Computation on Trees

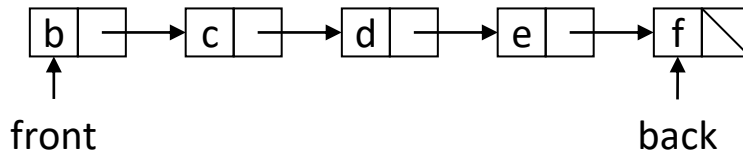
- ❖ Maps and reductions work on trees
 - Divide-and-conquer each child rather than array sub-ranges
 - Correct for unbalanced trees, but won't get much speed-up unless tree is balanced
- ❖ *Example*: minimum in an unsorted-but-balanced binary tree
 - $O(\log n)$ time given enough processors
- ❖ How to 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`
 - Asymptotically
 - Constants
 - ForkJoin Library

- ❖ More examples of parallel programs
 - Common patterns: reduce and map
 - Non-array inputs

- ❖ **Asymptotic Analysis for Fork/Join-style Parallelism**

pollev.com/332summer :: tinyurl.com/332-07-27A

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

A directed acyclic graph (DAG) is:

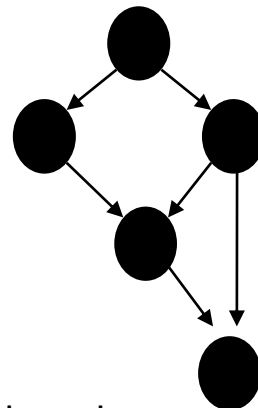
- A graph that is directed (edges have direction/arrows)
- And whose edges do not create a cycle (ability to trace a path that starts and ends at the same node)

❖ A `fork` makes two outgoing edges:

- New thread
- Continuation of current thread

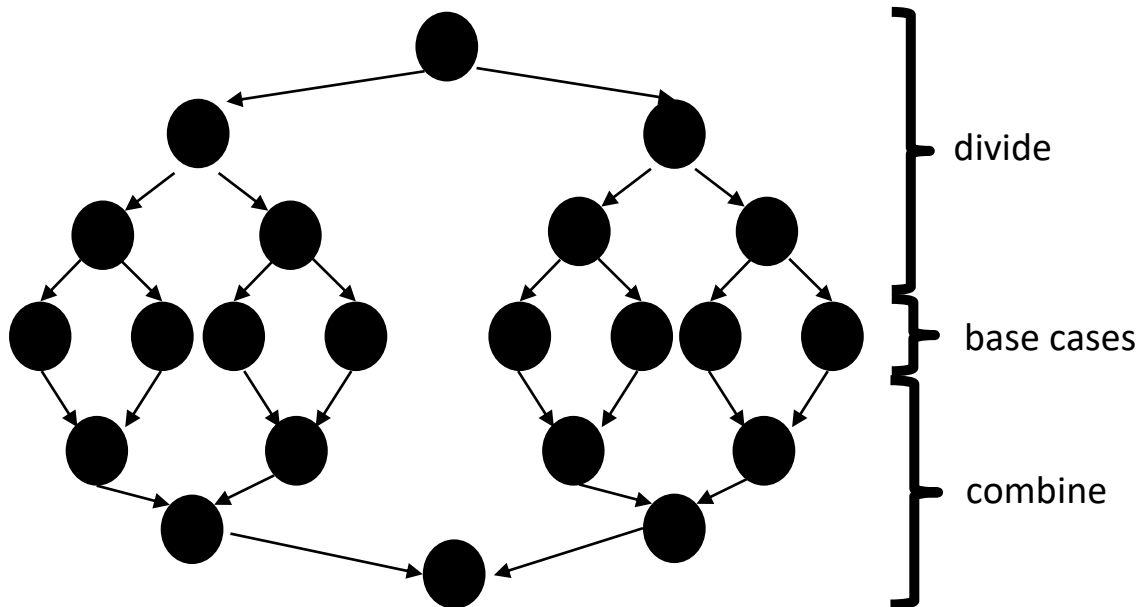
❖ A `join` takes two incoming edges

- The final node of the joined thread
- The computation that just finished in the current thread



Our Simple Examples

- ❖ **fork** and **join** are very flexible, but maps and reductions use them in a very basic way
 - A (perfect) tree, on top of an upside-down (perfect) tree



Aside: More Interesting DAGs?

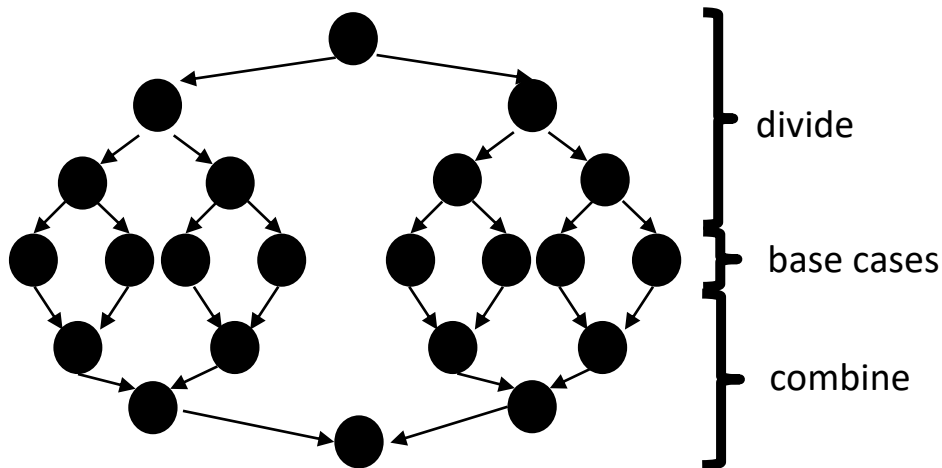
- ❖ The execution DAGs are not always this simple
 - *Example*: combining results might be so expensive that we parallelize it. Then each node in the *inverted* tree would expand into another set of nodes for that parallel computation

Definitions: Work and Span

- ❖ Let 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, 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, T_∞)
 - 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

Our Simple Examples + Our Definitions

- ❖ In this context, the span (T_∞) is:
 - The longest dependence-chain; i.e., longest ‘branch’ in parallel ‘tree’
 - *Example*: $O(\log n)$ for summing an array
 - We halve the data down to our sequential cut-off, then add back together
 - $2 * \log n$ steps, $O(1)$ time for each: $O(\log n)$



Work and Span in Fork/Join-style DAGs

- ❖ **Span** (T_∞) = sum of runtime of all nodes in the DAG's *most-expensive path*
 - Note: costs are on the nodes not the edges
 - $O(\log n)$ for simple maps and reductions
- ❖ **Work** (T_1) = sum of runtime of all nodes in the DAG
 - Any topological sort is a legal execution
 - Run nodes when they no longer have dependencies
 - $O(n)$ for simple maps and reductions

More Definitions: Speed-up and Parallelism

- ❖ **Speed-up**, using P processors: T_1 / T_P
- ❖ If speed-up is P as we vary P , we call it **perfect linear speed-up**
 - Perfect linear speed-up means doubling P halves running time
 - Usually our goal, but hard to get in practice
- ❖ **Parallelism**: T_1 / T_∞
 - Parallelism is the maximum possible speed-up; the point at which adding processors doesn't help
 - That point depends on the span

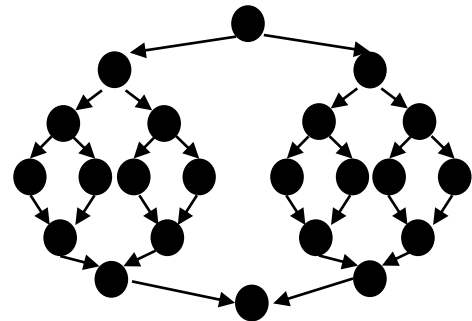
Parallel algorithms attempt to decrease span without increasing work too much

Obtaining Optimality for T_p

- ❖ What is the asymptotically optimal T_p , for any value of P ?
 - (as usual, we ignore memory-hierarchy issues; i.e. caching)

- ❖ We know T_p is greater than or equal to:

- T_1 / P (why?)
- T_∞ (why?)



- ❖ So an *asymptotically optimal* execution must be:

$$O\left(\frac{T_1}{P} + T_\infty\right)$$

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

Optimal T_p Examples

$$T_p = O((T_1/P) + T_\infty)$$

- ❖ In the examples we've seen so far (e.g., sum an array):
 - $T_1 = O(n)$
 - $T_\infty = O(\log n)$
 - So expect : $T_p = O(n/P + \log n)$

- ❖ Suppose instead:
 - $T_1 = O(n^2)$
 - $T_\infty = O(n)$
 - So expect: $T_p = O(n^2/P + n)$

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
 - Give the **expected-time optimal guarantee** (assuming the library user did their job): $T_p = O((T_1 / P) + T_\infty)$

And Now for the Good / Bad News ...

- ❖ In practice, it's common that a program has:
 - a) Parts that **parallelize** well:
 - E.g. maps/reduces over arrays and trees
 - b) ... and parts that **don't parallelize** at all:
 - E.g. reading a linked list
 - E.g. waiting on input
 - E.g. computations where each step needs the results of previous step

- ❖ These unparallelizable parts turn out to be a big bottleneck, which brings us to Amdahl's Law ...

Amdahl's Law

- ❖ Let the work (T_1) be 1 unit of time and S be the unparallelizable portion of execution time:

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

- ❖ Suppose *perfect linear speed-up* on the parallelizable portion. Then:

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

- ❖ Amdahl's Law states the speed-up with P processors is:

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

- ❖ and the parallelism (maximum possible speed-up) is:

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

Amdahl's Law Example

- ❖ Recall: $T_1 = 1 = S + (1-S)$ and $T_p = S + (1-S)/P$
- ❖ Suppose: $T_1 = 1/3 + 2/3 = 1$ (eg, $T_1 = 100s = 33s + 67s$)
- ❖ Then: $T_p = 33 \text{ sec} + (67 \text{ sec})/P$
 $T_3 = 33 \text{ sec} + (67 \text{ sec})/3 =$
 $T_6 = 33 \text{ sec} + (67 \text{ sec})/6 =$
 $T_{67} = 33 \text{ sec} + (67 \text{ sec})/67 =$
- ❖ If 33% of a program is sequential, a billion processors won't give a speedup over 3!!!
- ❖ No matter how many processors you use, your speedup is bounded by the sequential portion of the program

Implications of Amdahl's Law

Speedup:	$T_1 / T_P = 1 / (S + (1-S)/P)$
Max Parallelism:	$T_1 / T_\infty = 1 / S$

- ❖ In “the good old days” (1980-2005), ~12 years = 100x speedup
- ❖ Now suppose in 12 years, clock speed is the same but you get 256 processors instead of 1. What portion of the program must be parallelizable to get 100x speedup?
 - *For 256 processors to get at least 100x speedup, we need*
$$100 \leq 1 / (S + (1-S)/256)$$
 - *Which means $S \leq .0061$ (i.e., 99.4% must be parallelizable)*

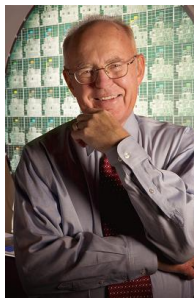
The Challenge Posed by Amdahl's Law

- ❖ Amdahl's Law tells us unparallelized parts become a bottleneck very quickly
 - But it *doesn't* tell us additional processors are worthless
- ❖ ... because we can find new parallel algorithms
 - Some things that seem sequential turn out to be parallelizable
 - Eg: How can we parallelize a 'running sum' array?

input	6	4	16	10	16	15	2	8
output	6	10	26	36	52	67	69	77

- ❖ We can also change the problem we're solving
 - Eg: Video games use tons of parallel processors; they are not rendering 10-year-old graphics faster

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
- ❖ Both are incredibly important in designing computer systems