CSE 332
Data Structures & Parallelism

Analysis of Fork-Join Parallel Programs

Melissa Winstanley

Spring 2024

Class Updates

- PLEASE BRING YOUR COMPUTERS TO SECTION TOMORROW!
- Also please clone the section repo beforehand - it is listed on the course website under tomorrow’s section.

- P2 is due TOMORROW at 11:59pm
  - You can use up to 2 late days
- Ex 8 (Dijkstra’s) is due next Tuesday
- P3 will be released tomorrow

ForkJoin Framework Version of sum

```
class SumTask extends RecursiveTask<Integer> {
  int lo; int hi; int[] arr;
  SumTask(int[] a, int l, int h) { ... }
  protected Integer compute() { // override
    if(hi - lo < SEQUENTIAL_CUTOFF)
      int ans = 0; // not a field -INF
      for(int i=lo; i < hi; i++)
        ans += arr[i]; max(ans, arr[i])
      return ans;
    else {
      SumTask left =
          new SumTask(arr,lo,(hi+lo)/2);
      SumTask right =
          new SumTask(arr,(hi+lo)/2,hi);
      left.fork(); // forks a thread and calls compute
      int rightAns = right.compute(); // call directly
      int leftAns = left.join(); // get result from left
      return leftAns + rightAns;
                 max
  }
}
```

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

int sum(int[] arr){
  SumTask task =
      new SumTask(arr,0,arr.length)
  return POOL.invoke(task);
  // invoke returns the value
}
```

What needs to change to find the max value in an array, instead of sum?

ForkJoin Framework Version of max

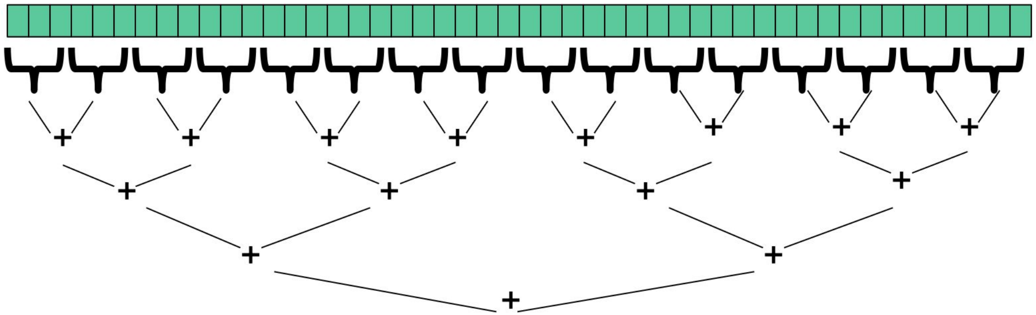
```
class MaxTask extends RecursiveTask<Integer> {
  int lo; int hi; int[] arr;
  MaxTask(int[] a, int l, int h) { ... }
  protected Integer compute() { // override
    if(hi - lo < SEQUENTIAL_CUTOFF)
      int ans = -INF;
      for(int i=lo; i < hi; i++)
        ans = Math.max(ans, arr[i]);
      return ans;
    else {
      MaxTask left =
            new MaxTask(arr,lo,(hi+lo)/2);
      MaxTask right =
            new MaxTask(arr,(hi+lo)/2,hi);
      left.fork(); // forks a thread and calls compute
      int rightAns = right.compute(); // call directly
      int leftAns = left.join(); // get result from left
      return Math.max(leftAns, rightAns);
  }
}
```

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

int max(int[] arr){
  MaxTask task =
      new MaxTask(arr,0,arr.length)
  return POOL.invoke(task);
  // invoke returns the value
}
```

What needs to change to find the max value in an array, instead of sum?

Examples



Parallelization (for some algorithms)

- Describe how to compute result at the ‘cut-off’
- Describe how to merge results

How would we do the following (assuming data is given as an array)?

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?

Reductions

This class of computations are called reductions

- We ‘reduce’ a large array of data to a single item
- Produce single answer from collection via an associative operator
- Examples: max, count, leftmost, rightmost, sum, product, …

Note: Recursive results don’t have to be single numbers or strings. They can be arrays or objects with multiple fields.

- Example: create a Histogram of test results from a much larger array of actual test results

While many can be parallelized due to nice properties like associativity of addition, some things are inherently sequential

- How we process **`arr[i]`** may depend entirely on the result of processing **`arr[i-1]`**

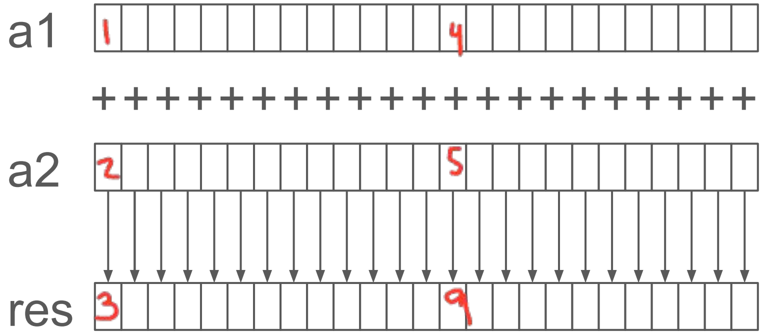
Even easier: Maps (data parallelism)

A map 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){
  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() { // override
    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 POOL =
    new ForkJoinPool();

int[] add(int[] arr1, int[] arr2){
  int[] res = new int[arr1.length];
  POOL.invoke(new VecAdd(
      0,arr.length,res, arr1,arr2));
  return ans res
}
```

Maps and reductions

Maps and reductions: the “workhorses” of parallel programming

- By far the two most important and common patterns
    - Two more-advanced patterns in next lecture
- 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

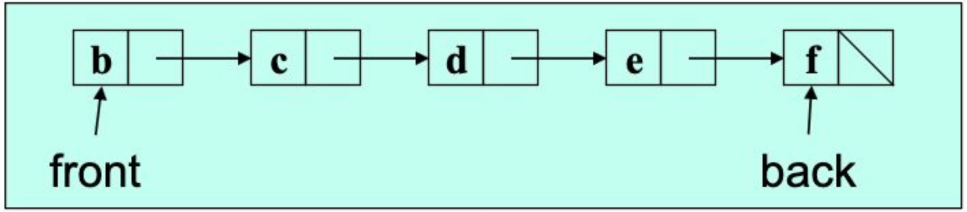
Trees

- Maps and reductions work just fine on balanced trees
  - Divide-and-conquer each child rather than array sub-ranges
  - Correct for unbalanced trees, but won't get much speed-up
- Example: minimum element in an *unsorted* but balanced binary tree in 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

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 beneficial for expensive per-element operations



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(log n) vs. O(n)
  - Trees have the same flexibility as lists compared to arrays (in terms of say inserting an item in the middle of the list)

Analyzing algorithms

How to measure efficiency?

- Want asymptotic bounds
- Want to analyze the algorithm without regard to a specific number of processors
- The key “magic” of the ForkJoin Framework is getting expected run-time performance asymptotically optimal for the available number of processors
  - So we can analyze algorithms assuming this guarantee

Work and Span

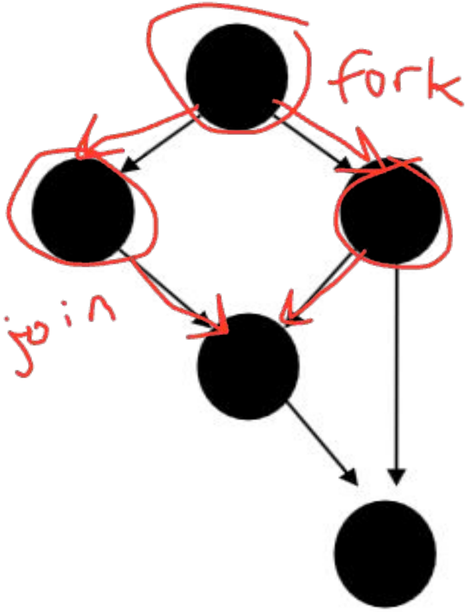
Let T_P be the running time if there are **P** processors available

Two key measures of run-time:

- **Work**: How long it would take 1 processor = T_1
  - Just "sequentialize" the recursive forking
  - Cumulative work that all processors must complete
- **Span**: How long it would take infinity processors = T_∞
  - The hypothetical ideal for parallelization
  - This is the longest "dependence chain" in the computation
  - Example: O(log n) for summing an array
  - Also called "critical path length" or "computational depth"

The DAG (Directed Acyclic Graph)

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

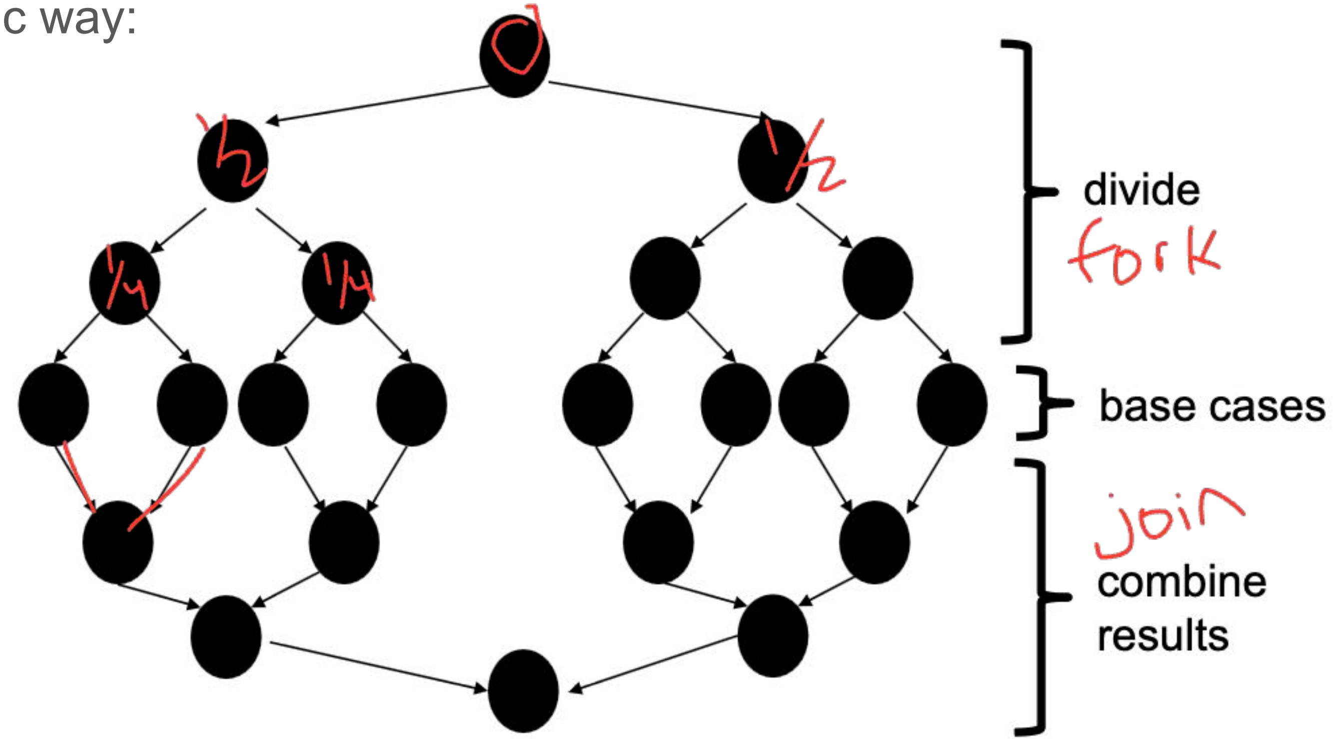


- A fork “ends a node” and makes two outgoing edges
  - New thread
  - Continuation of current thread
- A join “ends a node” and makes a node with two incoming edges
  - Node just ended
  - Last node of thread joined on

Our simple examples

`fork` and `join` are very flexible, but divide-and-conquer maps and reductions use them in a very basic way:

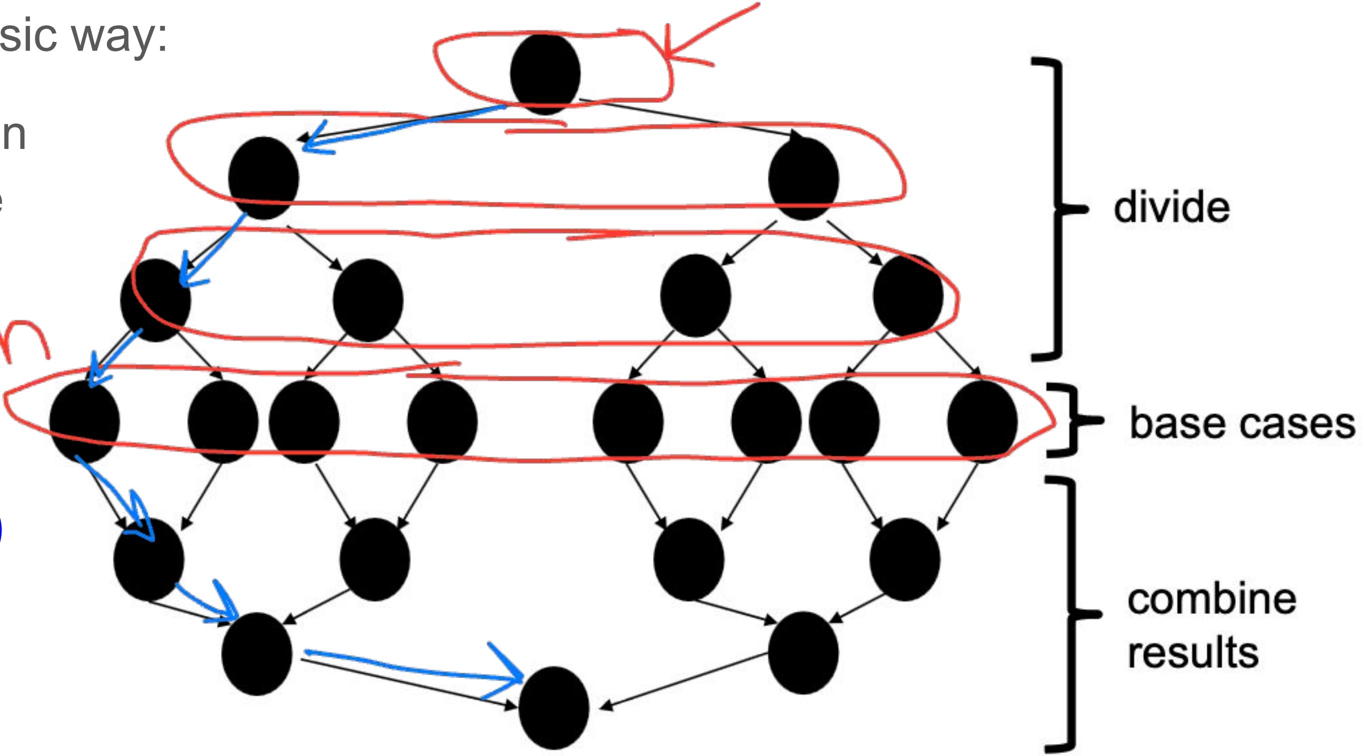
- A tree on top of an upside-down tree



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



$T_1 = O(n)$

$T_\infty = O(\log n)$

Connecting to performance

- Recall: $\mathbf{T_P}$ = running time if there are **P** processors available
- Work = T_1 = sum of run-time of all nodes in the DAG
  - That lonely processor does everything
  - Any topological sort is a legal execution
  - O(n) for simple maps and reductions
- Span = T_∞ = 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(log n) for simple maps and reductions

Definitions

$100/50 = 2$

A couple more terms:

- Speed-up on **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; hard to get in practice
- Parallelism is the maximum possible speed-up: T_1 / T_∞
  - At some point, adding processors won't help
  - What that point is depends on the span

Parallel algorithms are about decreasing span without increasing work too much

Optimal T_P: Thanks ForkJoin library!

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

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

  - First term dominates for small **P**, second for large **P**
- The ForkJoin Framework gives an *expected-time guarantee* of asymptotically optimal!
  - Expected time because it flips coins when *scheduling*
  - How? For an advanced course (few need to know)
  - Guarantee requires a few assumptions about your code…

Division of responsibility

- Our job as ForkJoin Framework users:
  - Pick a good algorithm, write a program
  - When run, program creates a DAG of things to do
  - *Make all the nodes a small-ish and approximately equal amount of work*

- The framework-writer's job:
  - Assign work to available processors to avoid idling
    - Let framework-user ignore all scheduling issues
  - Keep constant factors low
  - Give the expected-time optimal guarantee assuming framework-user did his/her job

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

And now for the bad news...

- So far: talked about a parallel program in terms of work and span
- In practice, it's common that your program has:

  a) parts that **parallelize well:**
     - Such as maps/reduces over arrays and trees

  b) …and parts that **don't parallelize at all**:
     - Such as reading a linked list, getting input, or just doing computations where each step needs the results of previous step

- These **unparallelized** parts can turn out to be a big bottleneck, which brings us to Amdahl's Law …

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 seq

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

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

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

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

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

0

And the parallelism (infinite processors) is:

30%

$\frac{1}{.3} \sim 3x$

$$\mathbf{T_1 / T_\infty = 1 / S}$$

Amdahl's Law Example

Suppose: $T_1 = S + (1\text{-}S) = 1$ (aka total program execution time)

$T_1 = 1/3 + 2/3 = 1$

$T_1 = 33 \text{ sec} + 67 \text{ sec} = 100 \text{ sec}$

Time on P processors: $T_P = S + (1\text{-}S)/P$

So: $T_P = 33 \text{ sec} + (67 \text{ sec})/P$

$T_3 = 33 \text{ sec} + (67 \text{ sec})/3 =$ 33+22 = 55

$T_6 = 33 \text{ sec} + (67 \text{ sec})/6 =$ 11 = 44

$T_{67} = 33 \text{ sec} + (67 \text{ sec})/67 =$ 33+1 = 34

100/34 = 3

Why such bad news?

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

- Suppose 33% of a program is sequential
  - Then 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

Amdahl's Law is a bummer - but all is not lost!

- 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 entirely sequential turn out to be parallelizable
- Eg. How can we parallelize the following?
  - Take an array of numbers, return the 'running sum' array

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

- At a glance, not sure; we'll explore this shortly

- We can also change the problem we're solving or do new things
  - Example: Video games use tons of parallel processors
    - They are not rendering 10-year-old graphics faster
    - They are rendering richer environments and more beautiful (terrible?) monsters