CSE332: Data Abstractions

Lecture 20: Analysis of Fork-Join Parallel Programs

Tyler Robison

Summer 2010

Where are we

So far we've talked about:

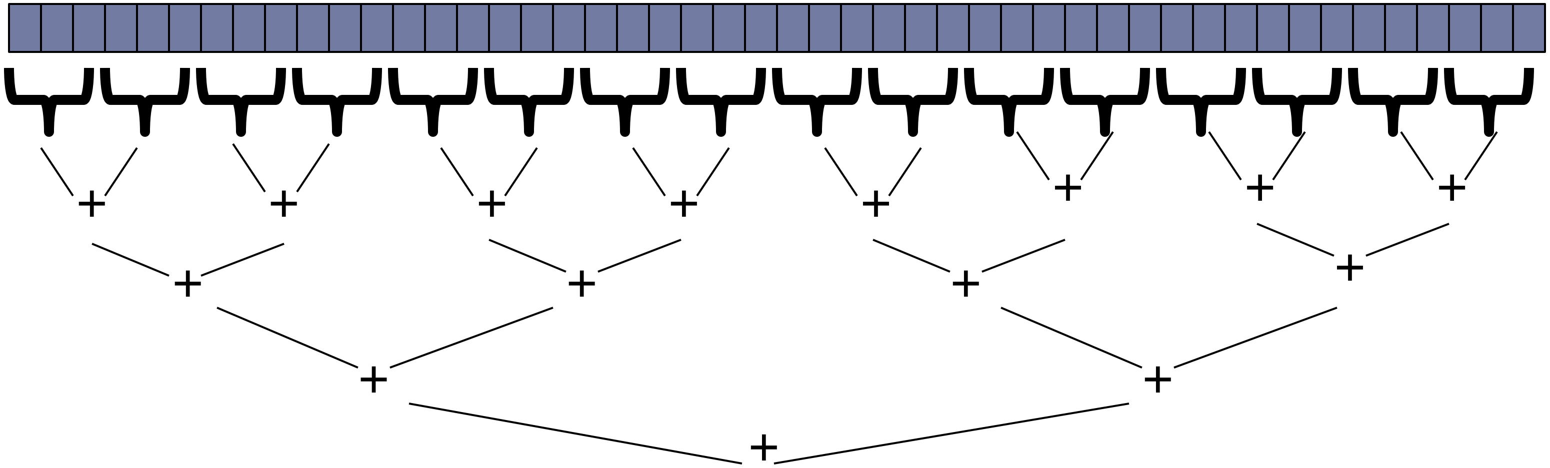
- How to use **`fork`**, and **`join`** to write a parallel algorithm
  - You'll see more in section
- Why using divide-and-conquer with lots of small tasks works well
  - Combines results in parallel
- Some Java and ForkJoin Framework specifics
  - More pragmatics in section and posted notes

Now:

- More examples of simple parallel programs
- How well different data structures work w/ parallelism
- Asymptotic analysis for fork-join parallelism
- Amdahl's Law

We looked at summing an array

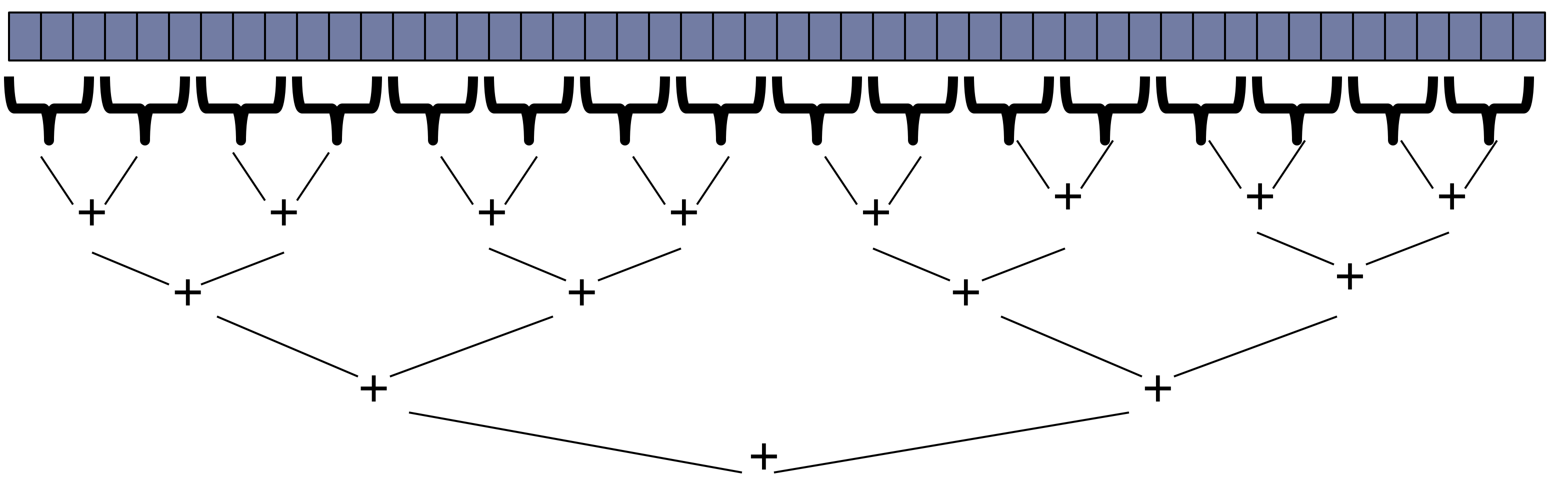
- Summing an array went from $O(n)$ sequential to $O(\texttt{log}\ n)$ parallel (*assuming **a lot** of processors and very large n*)
  - An exponential speed-up in theory
  - Not bad; that's 4 billion versus 32 (without constants, and in theory)



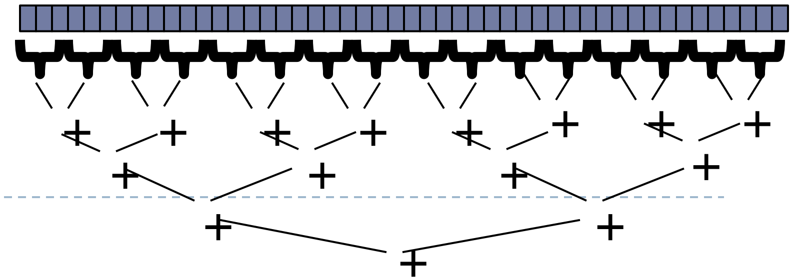
- Anything that can use results from two halves and merge them in $O(1)$ time has the same property…

Extending Parallel Sum

- We can tweak the 'parallel sum' algorithm to do all kinds of things; just specify 2 parts (usually)
  - Describe how to compute the result at the 'cut-off' (Sum: Iterate through sequentially and add them up)
  - Describe how to merge results (Sum: Just add 'left' and 'right' results)



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 (proj3)
  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
- Note: 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
- While many can be parallelized due to nice properties like associativity of addition, some things are inherently sequential
  - Ex: if we process **`arr[i]`** may depend entirely on the result of processing **`arr[i-1]`**

Even easier: Data Parallel (Maps)

- While reductions are a simple pattern of parallel programming, maps are even simpler
  - Operate on set of elements to produce a new set of elements (no combining results); generally of the same length
- Ex: Map each string in an array of strings to another array containing its length
  - {“abc”,”bc”,”a”} maps to {3,2,1}
- Ex: Add two Vectors

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

Example of 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();
    }
  }
}
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;
}
```

Map vs reduce

- In our examples:
- Reduce:
  - Parallel-sum extended RecursiveTask
  - Result was returned from compute()
- Map:
  - Class extended was RecursiveAction
  - Nothing returned from compute()
  - In the above code, the ‘answer’ array was passed in as a parameter
- Doesn’t *have* to be this way
  - Map can use RecursiveTask to, say, return an array
  - Reduce could use RecursiveAction; depending on what you’re passing back via RecursiveTask, could store it as a class variable and access it via ‘left’ or ‘right’ when done

Digression on maps and reduces

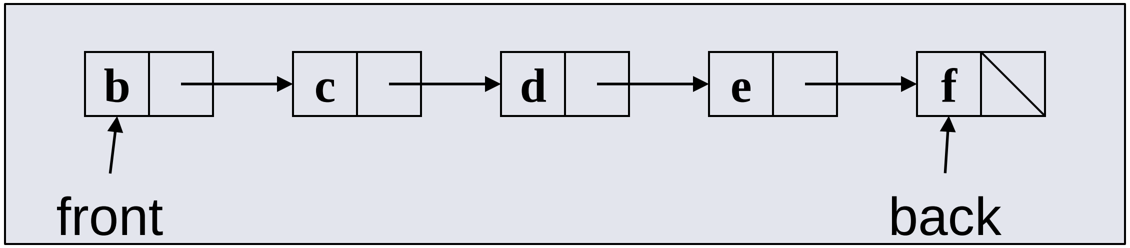
- You may have heard of Google's "map/reduce"
  - Or the open-source version Hadoop
- Idea: Want to run algorithm on enormous amount of data; say, sort a petabyte (10^6 gigabytes) of data
  - Perform maps and reduces on data using many machines
    - The system takes care of distributing the data and 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 programming (see 341) transferred to large-scale distributed computing

Works on Trees as well as Arrays

- Our basic patterns so far – maps and reduces – 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 $O(\texttt{log}\ n)$ time given enough processors
- How to do the sequential cut-off?
  - Store number-of-descendants at each node (easy to maintain)
  - Or you could approximate it with, e.g., AVL 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



- Not really…
  - 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 inserting in the middle)

Analyzing algorithms

- Parallel algorithms still need to be:
  - Correct
  - Efficient
- For our algorithms so far, correctness is “obvious” so we’ll focus on efficiency
  - Still 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
    - Lets us just analyze our algorithms given this “guarantee”

Work and Span

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

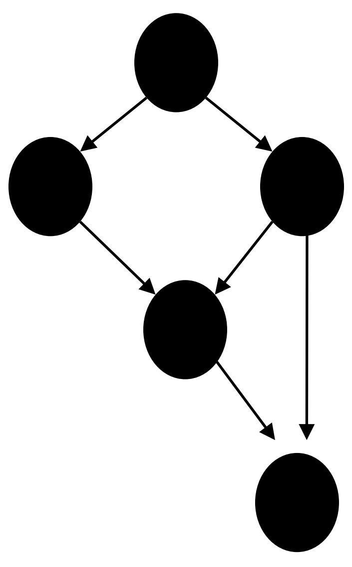
Type/power of processors doesn't matter; $\mathbf{T_P}$ used asymptotically, and to compare improvement by adding a few processors

Two key measures of run-time for a fork-join computation

- Work: How long it would take 1 processor = $\mathbf{T_1}$
  - Just "sequentialize" all the recursive forking
- Span: How long it would take infinity processors = $\mathbf{T_\infty}$
  - The hypothetical ideal for parallelization

The DAG

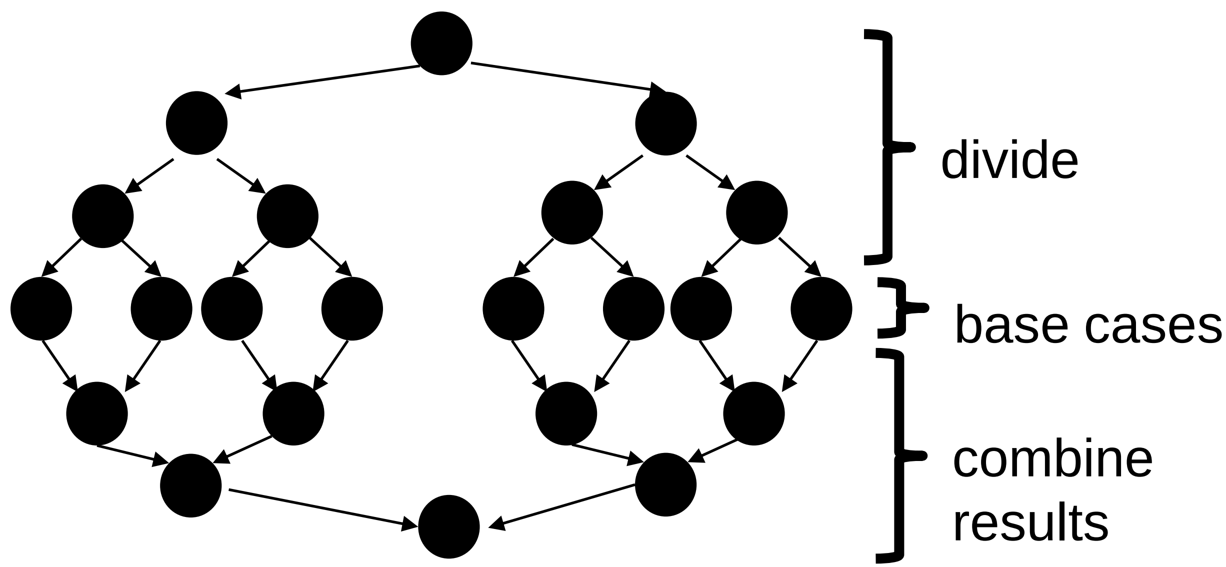
- A program execution using **`fork`** and **`join`** can be seen as a DAG
- Nodes: Pieces of work
- Edges: Source must finish before destination starts



- 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

Our **`fork`** and **`join`** frequently look like this:



In this context, the span ($\mathbf{T}_\infty$) is:

- The longest dependence-chain; longest 'branch' in parallel 'tree'
- Example: $O(\log n)$ for summing an array; we halve the data down to our cut-off, then add back together; $O(\log n)$ steps, O(1) time for each
- Also called "critical path length" or "computational depth"

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
    - You get to do this on project 3

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
  - One processor has to do all the work
  - Any topological sort is a legal execution

- 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

Definitions

A couple more terms:

- Speed-up on **P** processors: $\mathbf{T_1 / T_P}$
- 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: $\mathbf{T_1 / T_\infty}$
  - At some point, adding processors won't help
  - What that point is depends on the span

Division of responsibility

- Our job as ForkJoin Framework users:
  - Pick a good algorithm
  - Write a program. When run it 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 (won't study how to do it):
  - Assign work to available processors to avoid idling
  - Keep constant factors low
  - Give an expected-time guarantee (like quicksort) assuming framework-user did his/her job

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

What that means (mostly good news)

The fork-join framework guarantee

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

- No implementation of your algorithm can beat $O(T_\infty)$ by more than a constant factor
- No implementation of your algorithm on **P** processors can beat (T_1 / P) (ignoring memory-hierarchy issues)
- So the framework on average gets within a constant factor of the best you can do, assuming the user (you) did his/her job

So: You can focus on your algorithm, data structures, and cut-offs rather than number of processors and scheduling

- Analyze running time given T_1, T_∞, and **P**

Examples

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

- In the algorithms seen so far (e.g., sum an array):
  - $\mathbf{T_1} = O(n)$
  - $\mathbf{T_\infty} = O(\mathtt{log}\ n)$
  - So expect (ignoring overheads): $\mathbf{T_P} \leq O(n/\mathbf{P} + \mathtt{log}\ n)$
- Suppose instead:
  - $\mathbf{T_1} = O(n^2)$
  - $\mathbf{T_\infty} = O(n)$
  - So expect (ignoring overheads): $\mathbf{T_P} \leq O(n^2/\mathbf{P} + n)$

Amdahl's Law (mostly bad news)

- So far: talked about a parallel program in terms of work and span

- In practice, it's common that there are parts of your program that parallelize well…
  - Such as maps/reduces over arrays and trees

  …and parts that don't parallelize at all

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

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 **cannot** be parallelized

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

Makes sense, right?

Non-parallelizable + parallelizable = total = 1

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

That is, we double the # of processors, and that portion takes halve the time

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

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

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

And the parallelism (infinite processors) is:

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

Why such bad news

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

- Suppose 33% of a program 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

$$100 \le 1 / (\mathbf{S} + (1\text{-}\mathbf{S})/256)$$

Which means $\mathbf{S} \le .0061$ (i.e., 99.4% perfectly parallelizable)

All is not lost

Amdahl's Law is a bummer!

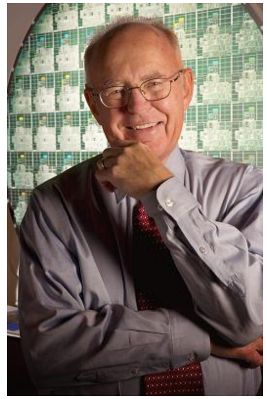
- But it doesn't mean additional processors are worthless

- We can find new parallel algorithms
  - Some things that seem clearly sequential turn out to be parallelizable
  - How parallelizable is 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

- We can 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

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
  - Implies diminishing returns of adding more processors
- Both are incredibly important in designing computer systems