CSE332: Data Abstractions

Lecture 19: Analysis of Fork-Join Parallel Programs

Dan Grossman
Spring 2010

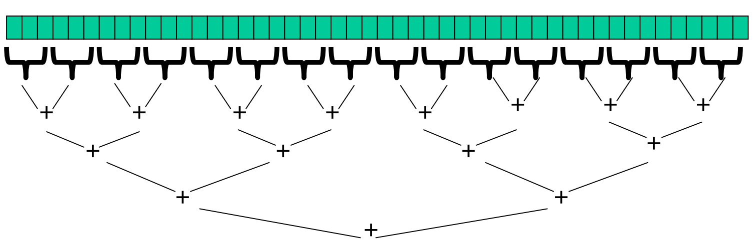
Where are we

Done:
- How to use **`fork`**, and **`join`** to write a parallel algorithm
- Why using divide-and-conquer with lots of small tasks is best
  – Combines results in parallel
- Some Java and ForkJoin Framework specifics
  – More pragmatics in section and posted notes

Now:
- More examples of simple parallel programs
- Arrays & balanced trees support parallelism, linked lists don't
- Asymptotic analysis for fork-join parallelism
- Amdahl's Law

What else looks like this?

- Saw summing an array went from $O(n)$ sequential to $O(\mathtt{log}\ n)$ 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?
- In project 3: corners of a rectangle containing all points
- Counts, for example, number of strings that start with a vowel
  – This is just summing with a different base case
  – Many problems are!

Reductions

- Computations of this form are called reductions (or reduces?)
- They take a set of data items and produce a single result
- 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
  - Example on project 3: Kind of like a 2-D histogram
- 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: 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)
  - 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];
  len = arr.length;
  FORALL(i=0; i < arr.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();
    }
  }
}
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;
}
```


Digression on maps and reduces

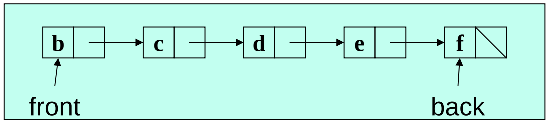
- You may have heard of Google's "map/reduce"
  - Or the open-source version Hadoop
- Idea: Perform maps and reduces on data using many machines
  - The system takes care of distributing the data and managing fault tolerance
  - You just write code to map one element and reduce elements to a combined 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
  - Complementary approach to declarative queries (see 344)

Trees

- Our basic patterns so far – maps and reduces – 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 $O(\mathtt{log}\ n)$ time given enough processors
- How to do the sequential cut-off?
  - Store number-of-descendants at each node (easy to maintain)
  - Or I guess 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



- 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

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

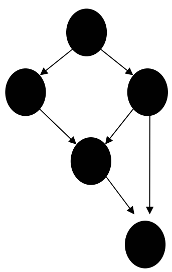
Let $\mathbf{T_P}$ be the running time if there are **P** processors available

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 longest dependence-chain
  - Example: $O(\mathtt{log}\ n)$ for summing an array since > $n/2$ processors is no additional help
  - Also called "critical path length" or "computational depth"

The DAG

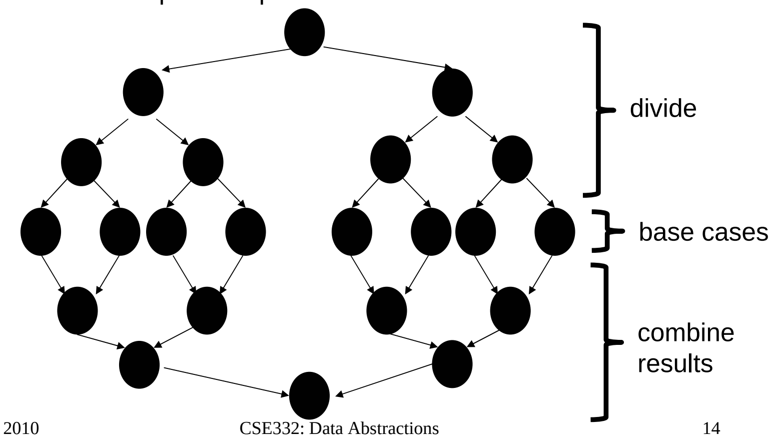
- A program execution using **`fork`** and **`join`** can be seen as a DAG
  - I *told* you graphs were useful! ☺
- Nodes: Pieces of work
- Edges: Source must finish before destination starts



- 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 our divide-and-conquer maps and reduces so far 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

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
  - That lonely 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

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

What that means (mostly good news)

The fork-join framework guarantee

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

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

Examples

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

- 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
  - "Nine women can't make a baby in one month"

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

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):

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

And the parallelism (infinite processors) is:

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

Why such bad news

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

- 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)

Plots you gotta see

1. Assume 256 processors
   - x-axis: sequential portion **S**, ranging from .01 to .25
   - y-axis: speedup $\mathbf{T_1 / T_P}$ (will go down as **S** increases)
2. Assume **S** = .01 or .1 or .25 (three separate lines)
   - x-axis: number of processors **P**, ranging from 2 to 32
   - y-axis: speedup $\mathbf{T_1 / T_P}$ (will go up as **P** increases)

Too important for me just to show you: *Homework problem!*

- Chance to use a spreadsheet or other graphing program
- Compare against your intuition
- A picture is worth 1000 words, especially if you made it

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

- 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 more beautiful 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