CSE 373: Data Structures and Algorithms

Lecture 23: Parallelism: Map, Reduce, Analysis

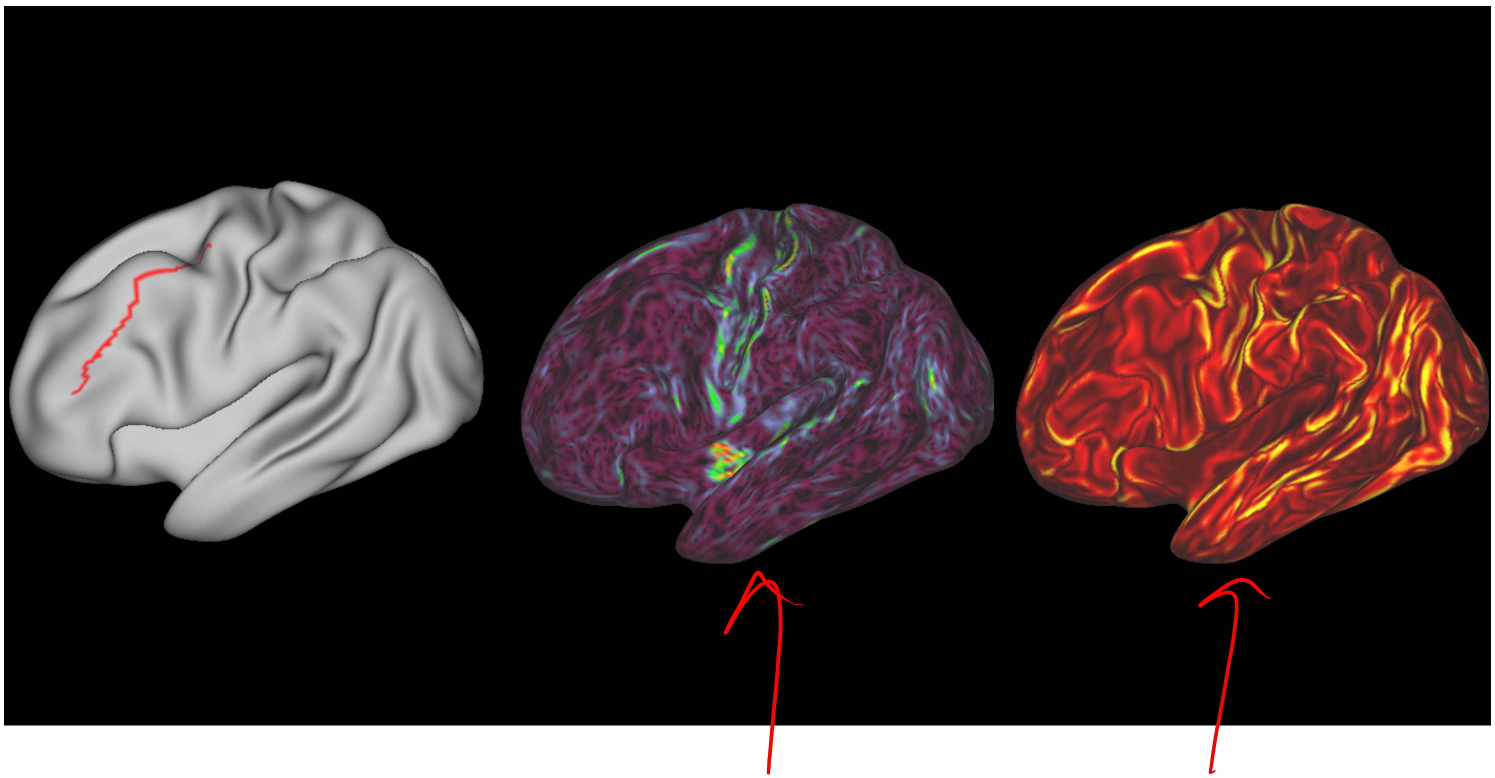
Instructor: Lilian de Greef
Quarter: Summer 2017

Today

- More on parallelism
  - Map & Reduce
  - Analysis of Efficiency

- Reminder:
  - Come visit my office hours to pick up midterm

Cool Homework 4 Extra Credit!



Outline

Done:

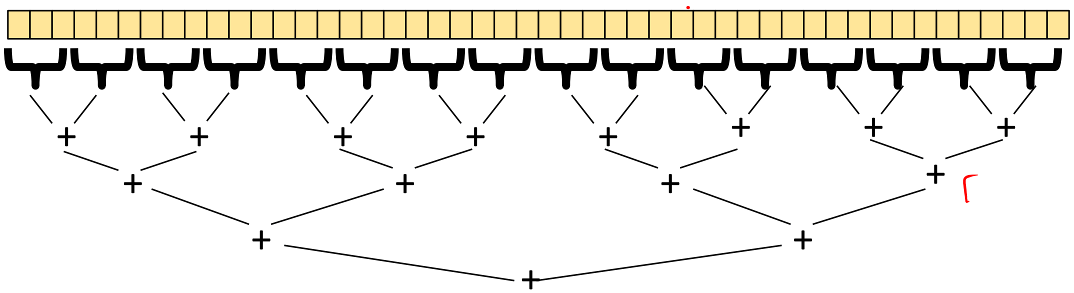
- How to write a parallel algorithm with fork and join
- Why using divide-and-conquer with lots of small tasks is best
  - Combines results in parallel
  - (Assuming library can handle “lots of small threads”)

Now:

- More examples of simple parallel programs that fit the “map” or “reduce” patterns
- Teaser: Beyond maps and reductions
- Asymptotic analysis for fork-join parallelism
- Amdahl’s Law

What else looks like this?

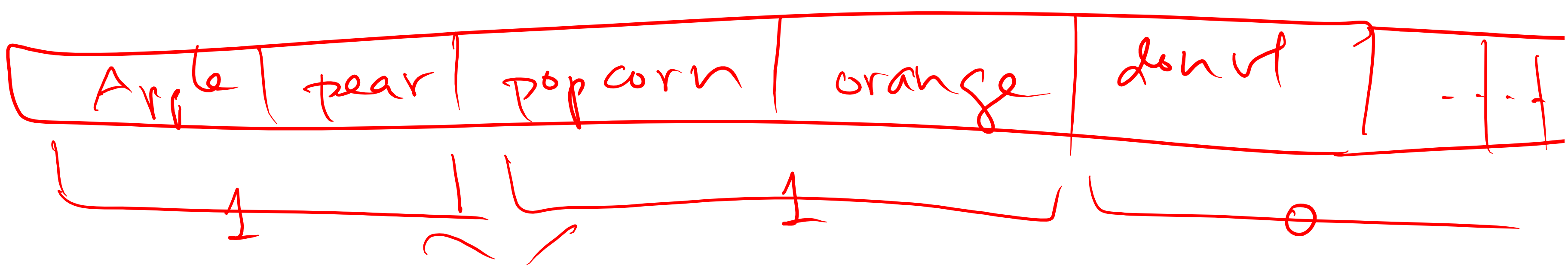
- Saw summing an array went from $O(n)$ sequential to $O(\texttt{log } n)$ parallel (*assuming **a lot** of processors and very large n*)
  - Exponential speed-up in theory (n / $\texttt{log }n$ grows exponentially)



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

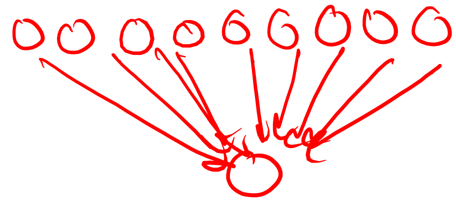
Examples

- Maximum or minimum element
- Check for an element satisfying some property e.g. is there a 17?
- Find left-most element satisfying some property e.g. where's the first 17?
- Counts e.g. # strings starting w/ vowel?



Reductions

- Computations of this form are called **reductions**

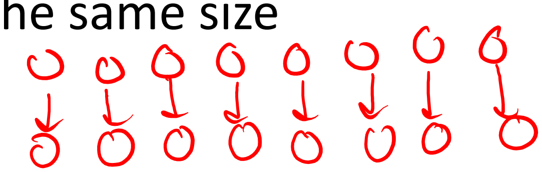


- Produce single answer from collection via an **associative operator**
  - Associative operator: $f(a, f(b,c)) = f(f(a,b), c)$

    order doesn't matter
  - Examples: max, sum, product, count …
    - max: $\max(a, \max(b,c)) = \max(\max(a,b), c)$
    - sum: $a + (b + c) = (a+b) + c$
    - product: $a * (b * c) = (a*b) * c$
  - Non-examples: subtraction, exponentiation, median, …
    - subtraction: $a - (b - c) \neq (a - b) - c$

Maps (Data Parallelism)

- A **map** operates on each element of a collection independently to create a new collection of the same size



- Example: Vector addition

v_1 input	6	4	16	10	16	14	2	8
v_2 input	2	10	6	6	2	6	8	7
$v_1 + v_2$ output	8	14	22	16	18	20	10	15

```
int[] vector_add(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;
}
```

Maps and reductions

Maps and reductions: the "workhorses" of parallel programming

- By far the two most important and common patterns
- 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

Practice: Map or Reduce?

map – apply to 1 elem
reduce – apply to >1 elem

For each of the following example scenarios, would you use *map* or *reduce*?

In the poll, vote for all that you would use ***reduce*** on.

output: collection?
one value?

A. Mark all the tasks in a to-do list as “done”
B. Get the total cost of a shopping list
C. Get the number of times someone said “like” or “um” in a transcription
D. Double a recipe by multiplying the amount for each ingredient by 2
E. Change driving directions to use “km” instead of “miles”
F. Find out whether a particular item you want to buy is in a store inventory

Beyond maps and reductions

- Some problems are "inherently sequential"

  "Six ovens can't bake a pie in 10 minutes instead of an hour"

- But not all parallelizable problems are maps and reductions

- Cool example that we don't have time for: "parallel prefix", a clever algorithm to parallelize the *problem* that this sequential *code* solves

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

```
int[] prefix_sum(int[] input){
  int[] output = new int[input.length];
  output[0] = input[0];
  for(int i=1; i < input.length; i++)
    output[i] = output[i-1]+input[i];
  return output;
}
```

MapReduce on computer clusters

- You may have heard of Google's "map/reduce"
  - Or the open-source version Hadoop
- Idea: Perform maps/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
  - Separating concerns is good software engineering

Speed-up from Parallelization

Moore's Law

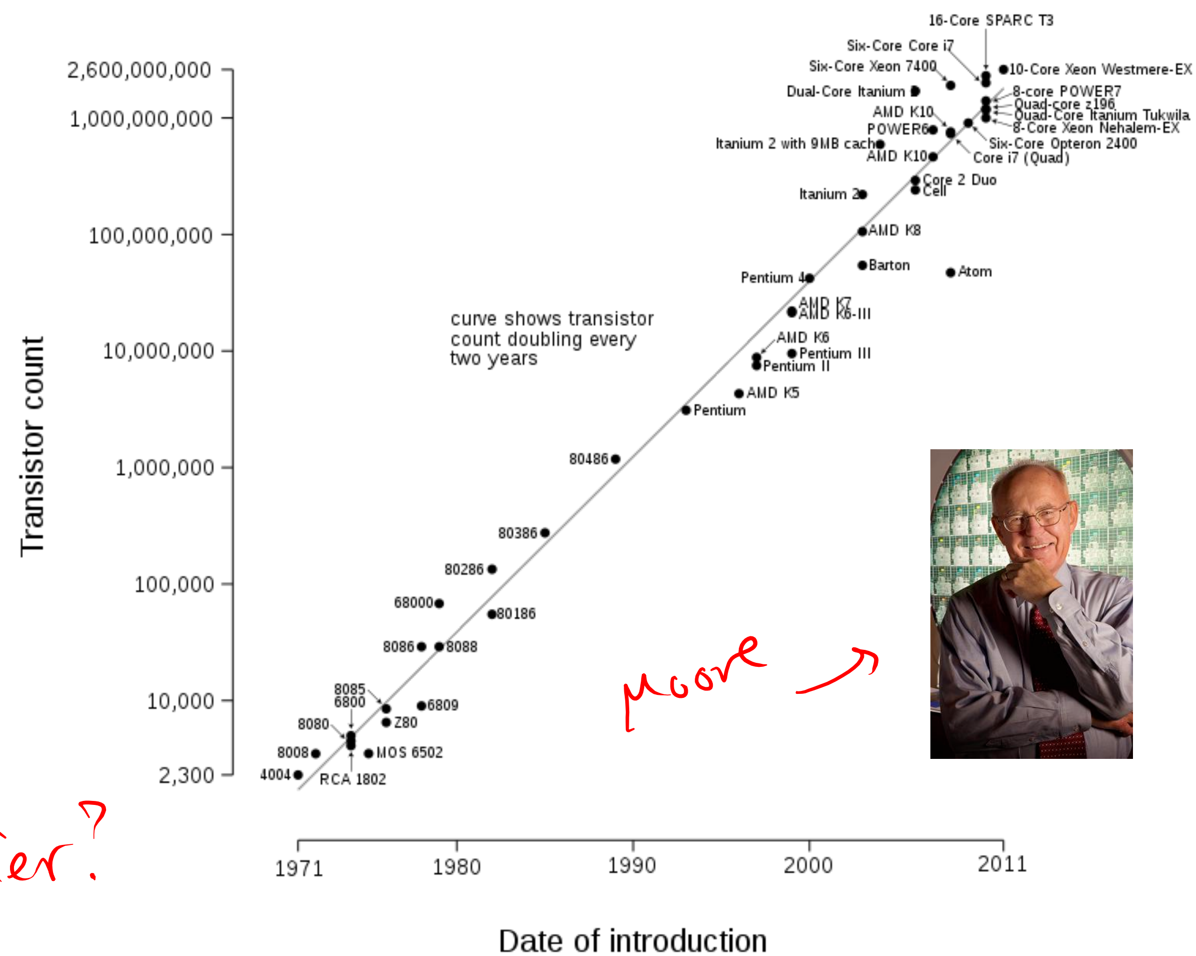
An observation on the semiconductor industry: the density of transistors doubles roughly every 2.5 years

↑ faster computers, exponentially!

Expected to reach limit around 2025 (can only fit so many atoms in one space!)

Parallelism the next frontier?

Microprocessor Transistor Counts 1971-2011 & Moore's Law

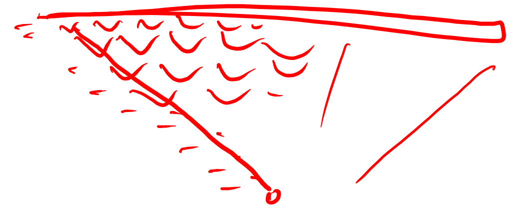


Moore →

Analyzing algorithms

- Like all algorithms, parallel algorithms should be:
  - correct
  - effient
- For our algorithms so far, we'll focus on efficiency
  (the correctness of summing numbers, etc. are not as interesting/insightful)
  - Want asymptotic bound
  - Want to analyze the algorithm without regard to a specific number of processors
  - Here: Identify the "best we can do" *if* the underlying *thread-scheduler* does its part

Work and Span



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

Two key measures of run-time:

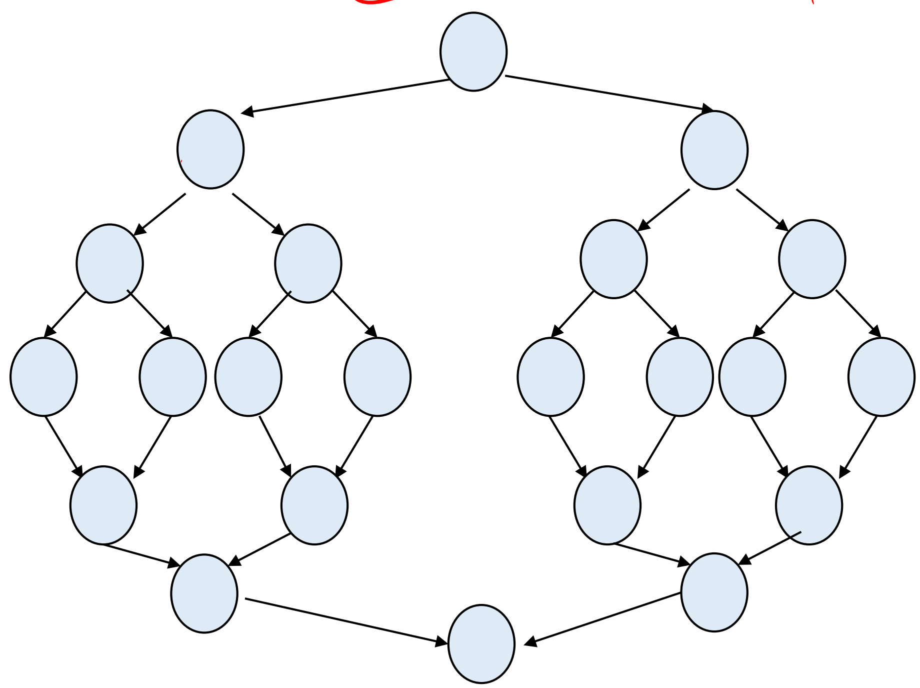
- **Work**: How long it would take 1 processor = T_1
  - Just "sequentialize" the recursive forking
- **Span**: How long it would take infinite processors = T_∞
  - The longest dependence-chain
  - Example: $O($`log` $n)$ for summing an array
    - Notice having > $n/2$ processors is no additional help

Our simple examples

Note: parallel programs can be more complex than our examples

Picture showing all the “stuff that happens” during a reduction or a map: it’s a (conceptual!) DAG

Directed Acyclic Graph



divide

base cases

combine results (conquer!)

Connecting to performance

- Recall: $\mathbf{T_P}$ = running time if we use P processors
- Work = $\mathbf{T_1}$ = 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 maps and reductions
- Span = $\mathbf{T_\infty}$ = sum of run-time of all nodes on 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

Speed-up

Parallelizing algorithms is about decreasing span without increasing work too much

- **Speed-up** on **P** processors: T_1 / T_P
- **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
- In practice we have **P** processors. How well can we do?
  - We cannot do better than $O(T_\infty)$ ("must obey the span") T_∞
  - We cannot do better than $O(T_1/P)$ ("must do all the work")

T_P

Best possible: $T_P = O(\max(T_\infty, T_1/P))$

Examples

Best possible $T_P = O(\max(T_\infty, T_1/P))$

- In the algorithms seen so far (e.g., sum an array):
  - $T_1 = O(n)$
  - $T_\infty = O(\log n)$
  - So expect at best (ignores overheads): $T_p = O(\max(\log n, n/p))$

do all work: n/p
obey the span: $\log n$

- Suppose instead:
  - $T_1 = O(n^2)$ ← work
  - $T_\infty = O(n)$ ← span
  - So expect at best (ignores overheads): $T_p = O(\max(\underbrace{n^2/p}_{T_1/p}, \underbrace{n}_{T_\infty}))$

Amdahl's Law (mostly bad news)

- So far: analyze parallel programs in terms of work and span
- In practice, typically have parts of programs that parallelize well…
  - Such as maps/reductions over arrays

  …and parts that don't parallelize at all

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

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

Suppose *parallel portion parallelizes perfectly (generous assumption)*

Then: $T_P = S + (1 - S)/P$

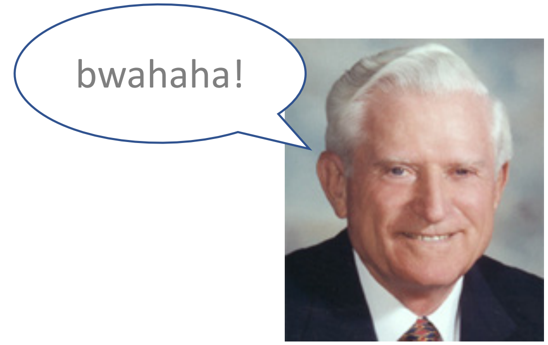
So the overall speedup with **P** processors is (Amdahl's Law):

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

And the parallelism (infinite processors) is:

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

Why such bad news



Amdahl's

speedup

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

- Suppose 33% of a program's execution is sequential
  - Then a billion processors won't give a speedup over 3
- From 1980-2005, 12 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

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

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

All is not lost

Amdahl's Law is a bummer!

- 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 sequential are actually parallelizable

- We can change the problem or do new things
  - Example: computer graphics use tons of parallel processors
    - Graphics Processing Units (GPUs) are massively parallel!

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

Practice problems

Question 1:

Time Efficiency & Space Efficiency

Better space efficiency

Given an array that contains the values 1 through 'n' two times each, find the one number that is contained only 1 time

Array of len n

v1: sort #'s, then iterate over sorted list

$\underbrace{\text{sort}}_{O(n^2) \text{ or } O(n\log n)}$ + $\underbrace{\text{iterate over sorted list}}_{O(n)}$ $\Rightarrow O(n\log n)$

Bucket sort: on original list $O(n)$

- sorting #'s
- # digits (max)

v2: HashSet

For each value x in the list:

if (find(x) in HashSet) ← O(1)

remove(x) ← O(1)

else put (x) into HashSet ← O(1)

n ⇒ O(n)

Output the one value in HashSet ← O(n)

over array of len ≥ 2n

open addressing

Fastest