Parallel Algorithms

Last time …

- Introduction to Big Data
- Assignment #0

Questions?

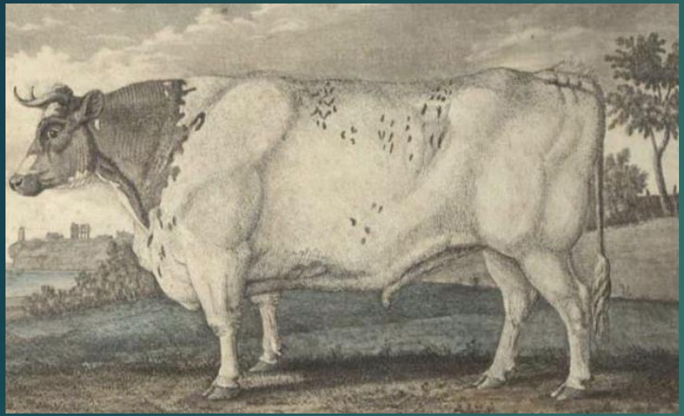
Today …

- Intro to Parallel Algorithms
- Map-Reduce
- Introduction to Spark

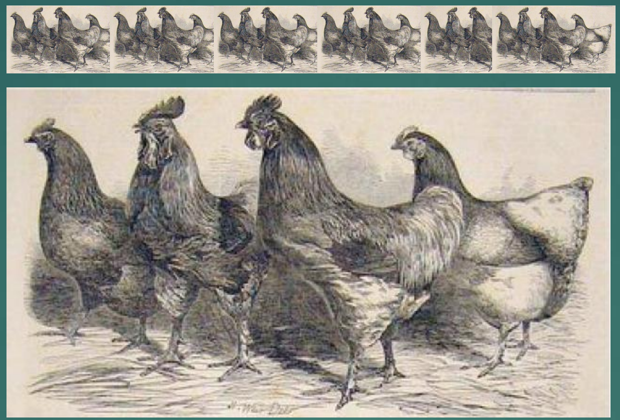
Parallel Thinking

THE MOST IMPORTANT GOAL OF TODAY'S LECTURE

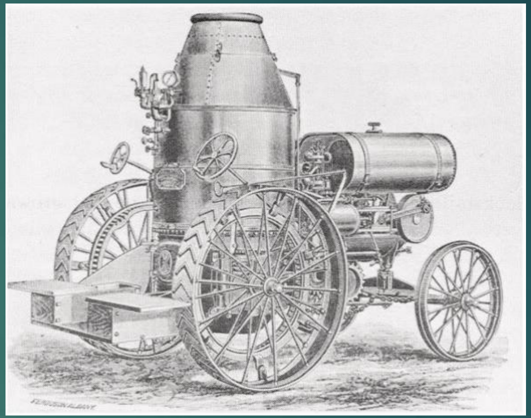
Parallelism & beyond …



1 ox: single core performance



1024 chickens: parallelism



tractor: better algorithms

If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?

Seymour Cray

Consider an array A with n elements,

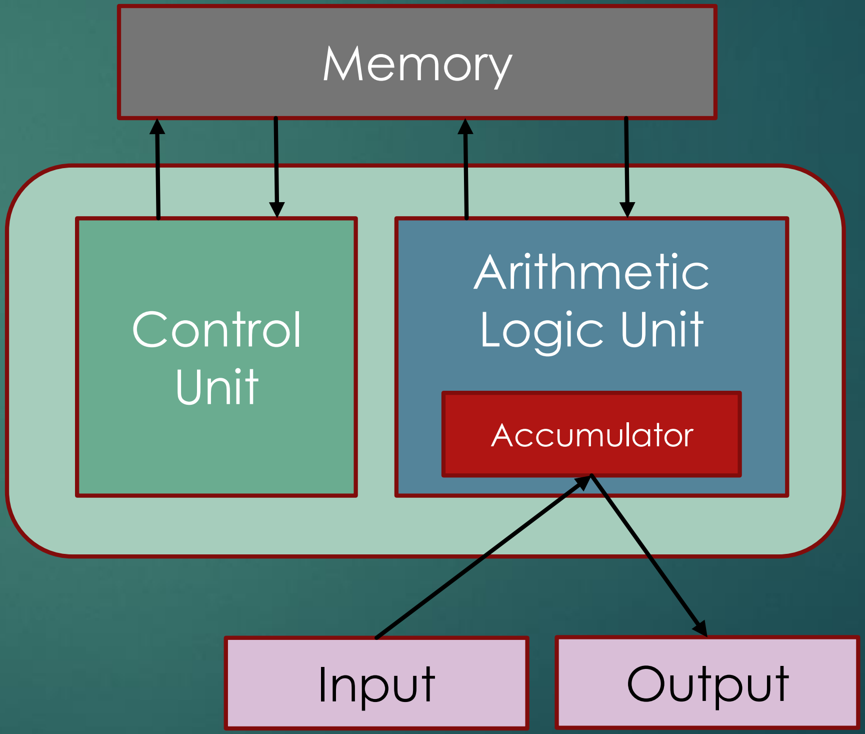
Goal: to compute,

$$x = \sum_{1}^{n} \sqrt{A_i}$$

Machine Model
Programming Model
Performance analysis

Von Neumann architecture

- Central Processing Unit (CPU, Core)
- Memory
- Input/Output (I/O)
- One instruction per unit/time
- **Sequential**



Characterizing algorithm performance

- O-notation
  - Given an input of size n, let $T(n)$ be the total time, and $S(n)$ the necessary storage
  - Given a problem, is there a way to compute lower bounds on storage and time → **Algorithmic Complexity**
  - $T(n) = O(f(n))$ means

    $T(n) \leq cf(n)$, where c is some unknown positive constant

    compare algorithms by comparing $f(n)$.

Scalability

- Scale Vertically → scale-up
  - Add resources to a single node
  - CPU, memory, disks,
- Scale Horizontally → scale-out
  - Add more nodes to the system

Parallel Performance

- Speedup

  best sequential time/time on p processors

- Efficiency

  $\text{speedup}/p, \qquad (< 1)$

Scalability

Amdahl's Law

- Sequential bottlenecks:

  Let s be the percentage of the overall work that is sequential

- Then, the speedup is given by

$$S = \frac{1}{s + \frac{1-s}{p}} \leq \frac{1}{s}$$

Gustafson

Sequential part should be independent of the problem size

Increase problem size, with increasing number of processors

Strong & Weak Scalability

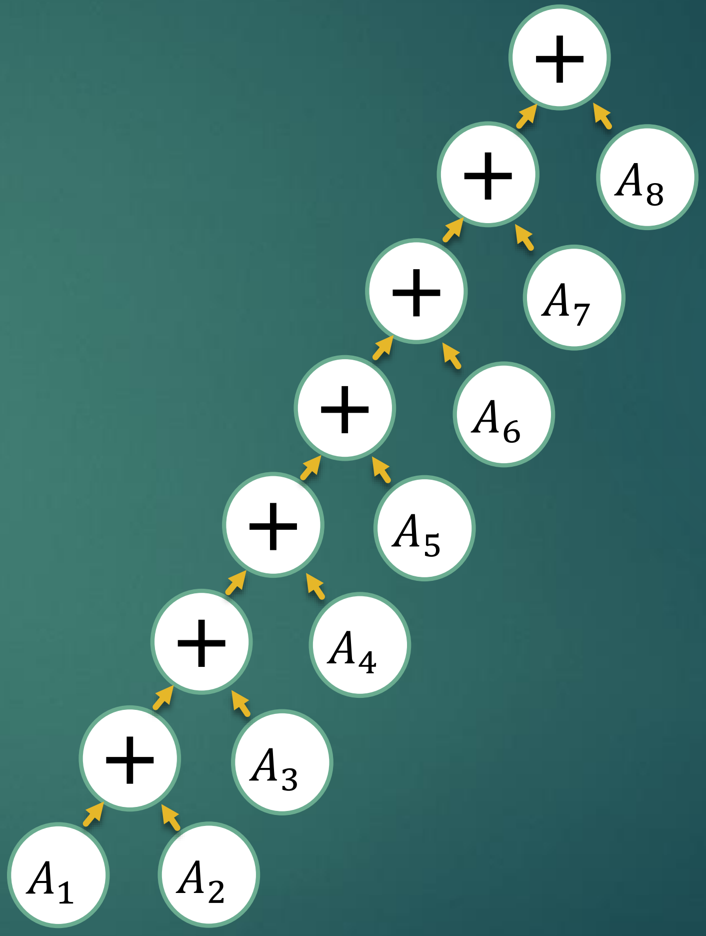
Increasing number of cores

Strong (fixed-sized) scalability — keep problem size fixed

Weak (fixed-sized) scalability — keep problem size/core fixed

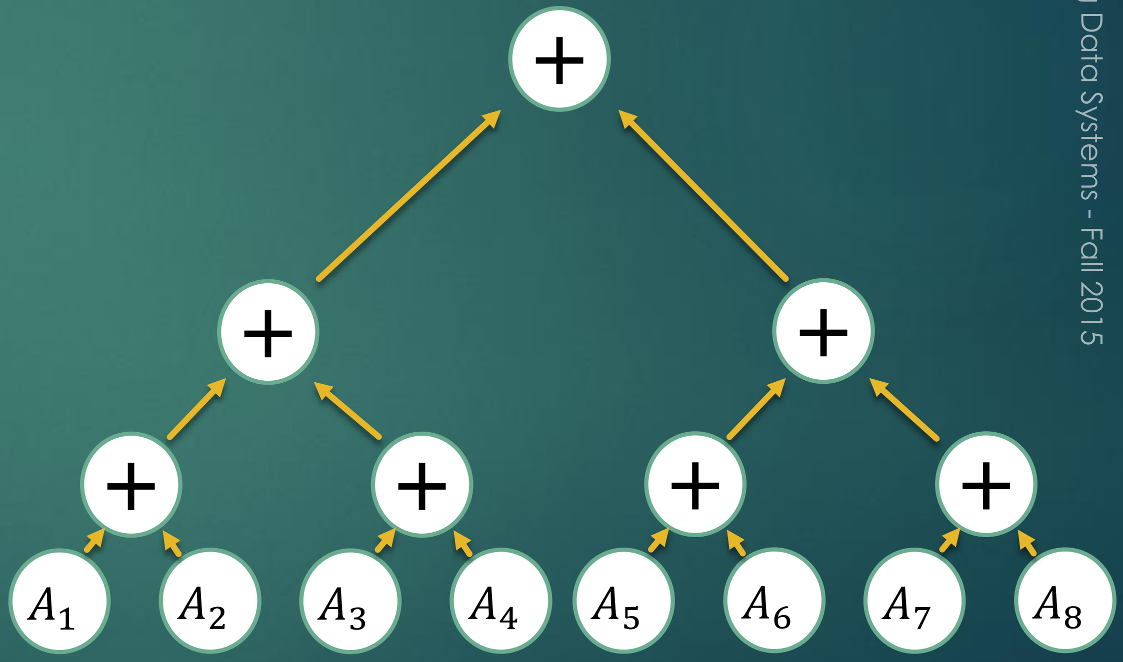
Work/Depth Models

- Abstract programming model
- Exposes the parallelism
  - Compute work W and depth D
  - D - longest chain of dependencies
  - $P = W/D$
- Directed Acyclic Graphs
- Concepts
  - parallel for (data decomposition)
  - recursion (divide and conquer)



Work/Depth Models

- Abstract programming model
- Exposes the parallelism
  - Compute work W and depth D
  - D - longest chain of dependencies
  - $P = W/D$
- Directed Acyclic Graphs
- Concepts
  - parallel for (data decomposition)
  - recursion (divide and conquer)



Parallel vs sequential for

- Dependent statements
  - $W = \sum W_i$
  - $D = \sum D_i$
- Independent statements
  - $W = \sum W_i$
  - $D = \max(D_i)$

Map Reduce

MapReduce programming interface

- Two-stage data processing
  - Data can be divided into many chunks
  - A map task processes input data & generates local results for one or a few chunks
  - A reduce task aggregates & merges local results from multiple map tasks
- Data is always represented as a set of key-value pairs
  - Key helps grouping for the reduce tasks
  - Though key is not always needed (for some applications, or for the input data), a consistent data represention eases the programming interface

Motivation & design principles

- Fault tolerance
  - Loss of a single node or an entire rack
  - Redundant file storage
- Files can be enormous
- Files are rarely updated
  - Read data, perform calculations
  - Append rather than modify
- Dominated by communication costs and I/O
  - Computation is cheap compared with data access
- Dominated by input size

Dependency in MapReduce

- Map tasks are independent from each other, can all run in parallel
- A map task must finish before the reduce task that processes its result
- In many cases, reduce tasks are commutative

- Acyclic graph model

Applications that don't fit

- MapReduce supports limited semantics
  - The key success of MapReduce depends on the assumption that the dominant part of data processing can be divided into a large number of independent tasks
- What applications don't fit this?
  - Those with complex dependencies – Gaussian elimination, k-means clustering, iterative methods, n-body problems, graph problems, …

MapReduce

- Map: chunks from DFS → (key, value)
  - User code to determine (k, v) from chunks (files/data)
- Sort: (k, v) from each map task are collected by a master controller and sorted by key and divided among the reduce tasks
- Reduce: work on one key at a time and combine all the values associated with that key
  - Manner of combination is determined by user code

MapReduce – word counting

- Input → set of documents
- Map:
  - reads a document and breaks it into a sequence of words
    $$w_1, w_2, \dots, w_n$$
  - Generates (k, v) pairs,
    $$(w_1, 1), (w_2, 1), \dots, (w_n, 1)$$
- System:
  - group all (k, v) by key
  - Given r reduce tasks, assign keys to reduce tasks using a hash function
- Reduce:
  - Combine the values associated with a given key
  - Add up all the values associated with the word → total count for that word

Node failures

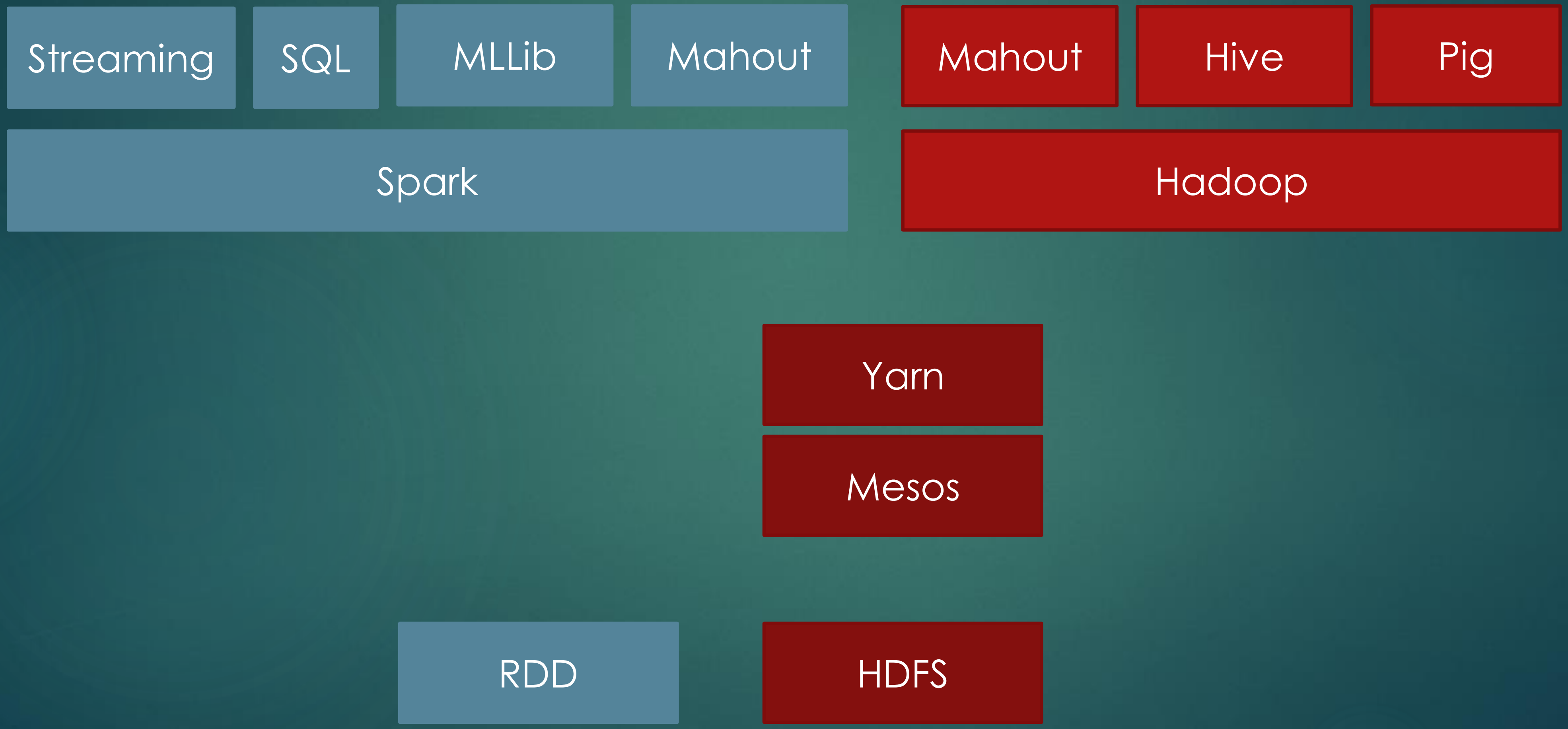
- Master node fails
  - Restart mapreduce job
- Node with Map worker fails
  - Redo all map tasks assigned to this worker
    - Set this worker as idle
    - Inform reduce tasks about change of input location
- Node with Reduce worker fails
  - Set the worker as idle

Spark

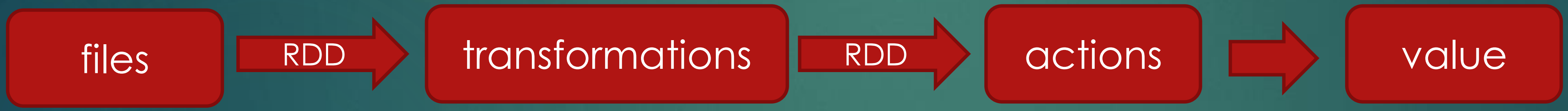
Spark

- Spark is a distributed in memory computational framework
- Attempts to provide a single platform for various data analytics scenarios → replace several specialized and fragmented solutions
- Specialized modules available in the form of libraries
  - SQL, Streaming, Graph Algorithms (GraphX), Machine Learning(MLLib)
- Introduces an abstract common data format that is used for efficient data sharing across processes – RDD

Spark


Streaming
SQL
MLLib
Mahout
Spark
Mahout
Hive
Pig
Hadoop
Yarn
Mesos
RDD
HDFS

General flow



Resilient Distributed Datasets

- Write programs in terms of transformation on distributed datasets
- RDD: collections of objects spread across a cluster, stored in RAM or on Disk
- Built through parallel transformations
- Automatically rebuilt on failure

RDD Operations

- Transformations
  - New RDDs from an existing one(s)
  - map, filter, groupBy
- Actions
  - Compute a result based on an RDD & return to driver or save to disk
  - count, collect, save
- Lazy evaluation → the first time used in an **action**
- Persistence → recomputed each time you run an **action**
  - use `data.persist()`

Spark Examples