Big Data Parallel Programming

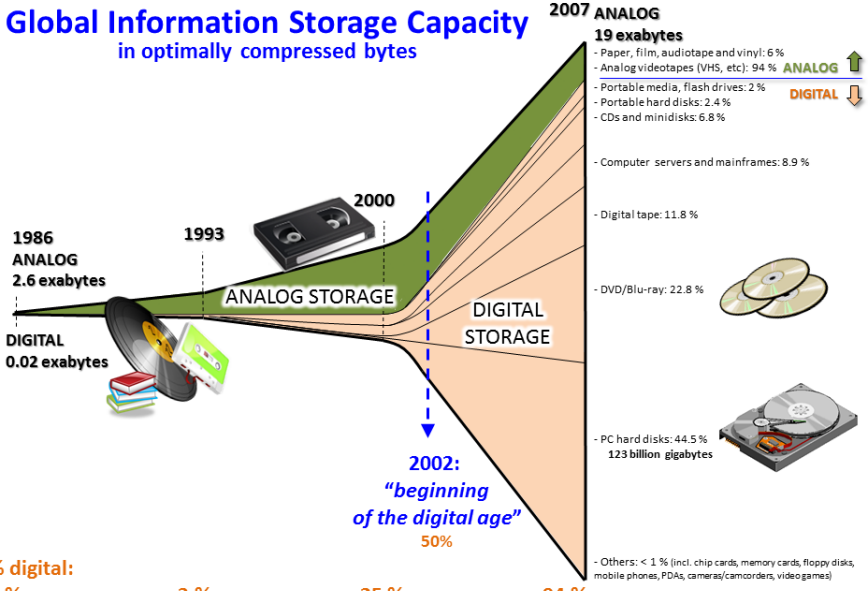
Part 1: Course Introduction

Slawomir Nowaczyk

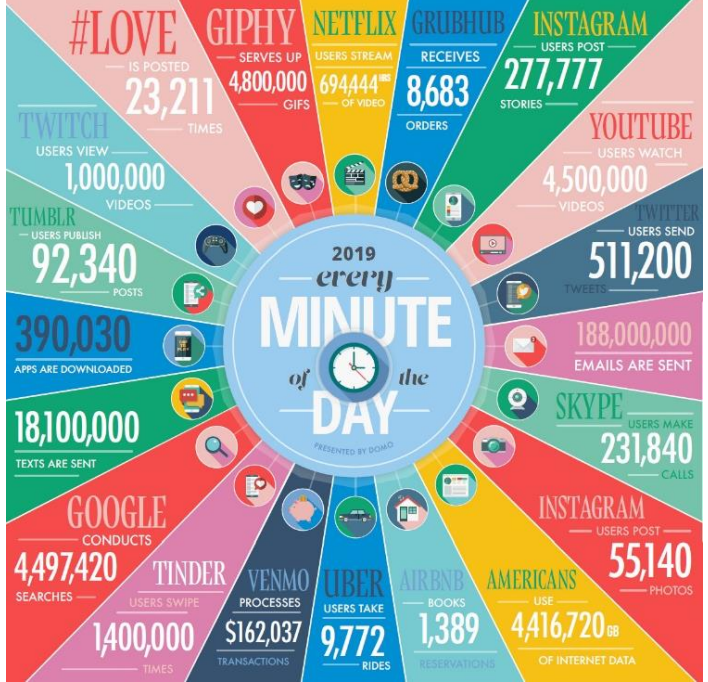

BIG DATA

Global Information Storage Capacity

in optimally compressed bytes



Source: Hilbert, M., & López, P. (2011). The World's Technological Capacity to Store, Communicate, and Compute Information. *Science*, 332(6025), 60 –65. http://www.martinhilbert.net/WorldInfoCapacity.html


#LOVE
IS POSTED
23,211
TIMES
GIPHY
SERVES UP
4,800,000
GIFS
NETFLIX
USERS STREAM
694,444 HRS
OF VIDEO
GRUBHUB
RECEIVES
8,683
ORDERS
INSTAGRAM
USERS POST
277,777
STORIES
TWITCH
USERS VIEW
1,000,000
VIDEOS
YOUTUBE
USERS WATCH
4,500,000
VIDEOS
TUMBLR
USERS PUBLISH
92,340
POSTS
TWITTER
USERS SEND
511,200
TWEETS
2019
every
MINUTE
of the
DAY
PRESENTED BY DOMO
390,030
APPS ARE DOWNLOADED
188,000,000
EMAILS ARE SENT
18,100,000
TEXTS ARE SENT
SKYPE
USERS MAKE
231,840
CALLS
GOOGLE
CONDUCTS
4,497,420
SEARCHES
INSTAGRAM
USERS POST
55,140
PHOTOS
TINDER
USERS SWIPE
1,400,000
TIMES
VENMO
PROCESSES
$162,037
TRANSACTIONS
UBER
USERS TAKE
9,772
RIDES
AIRBNB
BOOKS
1,389
RESERVATIONS
AMERICANS
USE
4,416,720 GB
OF INTERNET DATA

The FOUR V's of Big Data

From traffic patterns and music downloads to web history and medical records, data is recorded, stored, and analyzed to enable the technology and services that the world relies on every day. But what exactly is big data, and how can these massive amounts of data be used?

As a leader in the sector, IBM data scientists break big data into four dimensions: **Volume, Velocity, Variety and Veracity**

Depending on the industry and organization, big data encompasses information from multiple internal and external sources such as transactions, social media, enterprise content, sensors and mobile devices. Companies can leverage data to adapt their products and services to better meet customer needs, optimize operations and infrastructure, and find new sources of revenue.

By 2015
4.4 MILLION IT JOBS
will be created globally to support big data, with 1.9 million in the United States

Volume
SCALE OF DATA

40 ZETTABYTES
[43 TRILLION GIGABYTES]
of data will be created by 2020, an increase of 300 times from 2005

2005
2020

6 BILLION PEOPLE
have cell phones

WORLD POPULATION: 7 BILLION

It's estimated that
2.5 QUINTILLION BYTES
[2.3 TRILLION GIGABYTES]
of data are created each day

Most companies in the U.S. have at least
100 TERABYTES
[100,000 GIGABYTES]
of data stored

Variety
DIFFERENT FORMS OF DATA

As of 2011, the global size of data in healthcare was estimated to be
150 EXABYTES
[161 BILLION GIGABYTES]

30 BILLION PIECES OF CONTENT
are shared on Facebook every month

By 2014, it's anticipated there will be
420 MILLION WEARABLE, WIRELESS HEALTH MONITORS

4 BILLION+ HOURS OF VIDEO
are watched on YouTube each month

400 MILLION TWEETS
are sent per day by about 200 million monthly active users

Velocity
ANALYSIS OF STREAMING DATA

The New York Stock Exchange captures
1 TB OF TRADE INFORMATION
during each trading session

Modern cars have close to
100 SENSORS
that monitor items such as fuel level and tire pressure

By 2016, it is projected there will be
18.9 BILLION NETWORK CONNECTIONS
– almost 2.5 connections per person on earth

Veracity
UNCERTAINTY OF DATA

1 IN 3 BUSINESS LEADERS
don't trust the information they use to make decisions

27% OF RESPONDENTS
in one survey were unsure of how much of their data was inaccurate

Poor data quality costs the US economy around
$3.1 TRILLION A YEAR

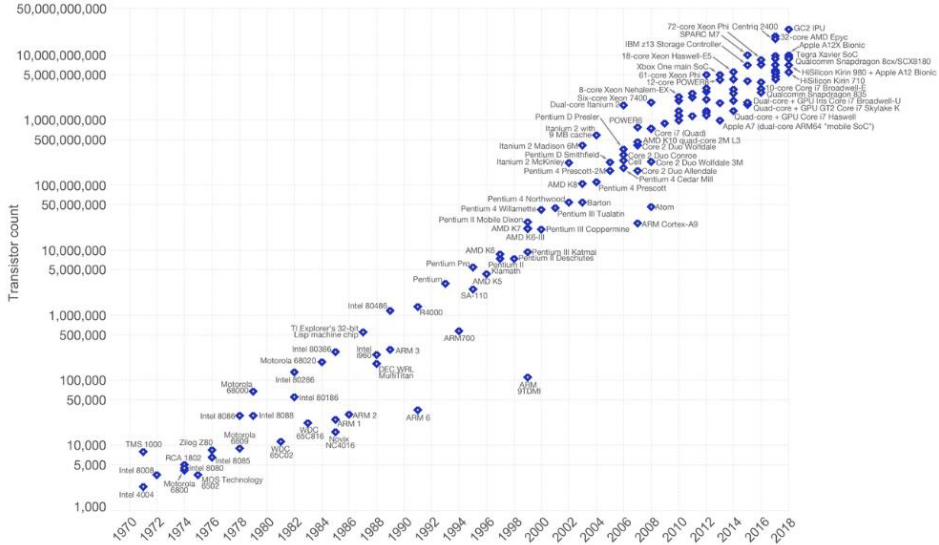
Sources: McKinsey Global Institute, Twitter, Cisco, Gartner, EMC, SAS, IBM, MEPTEC, QAS

IBM

Moore's Law – The number of transistors on integrated circuit chips (1971-2018)

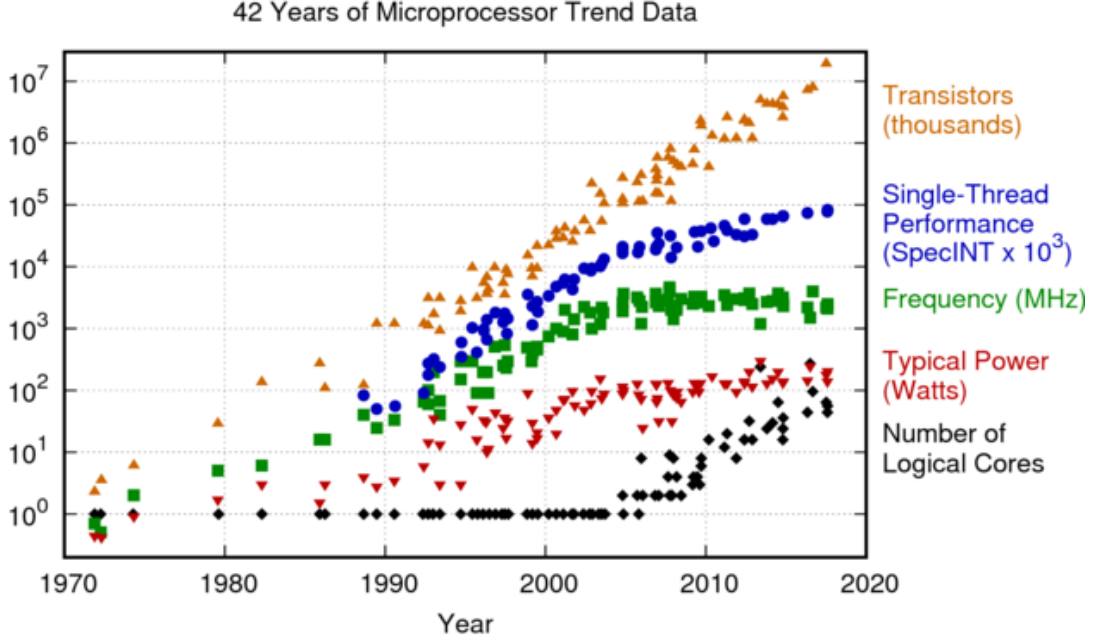
Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.

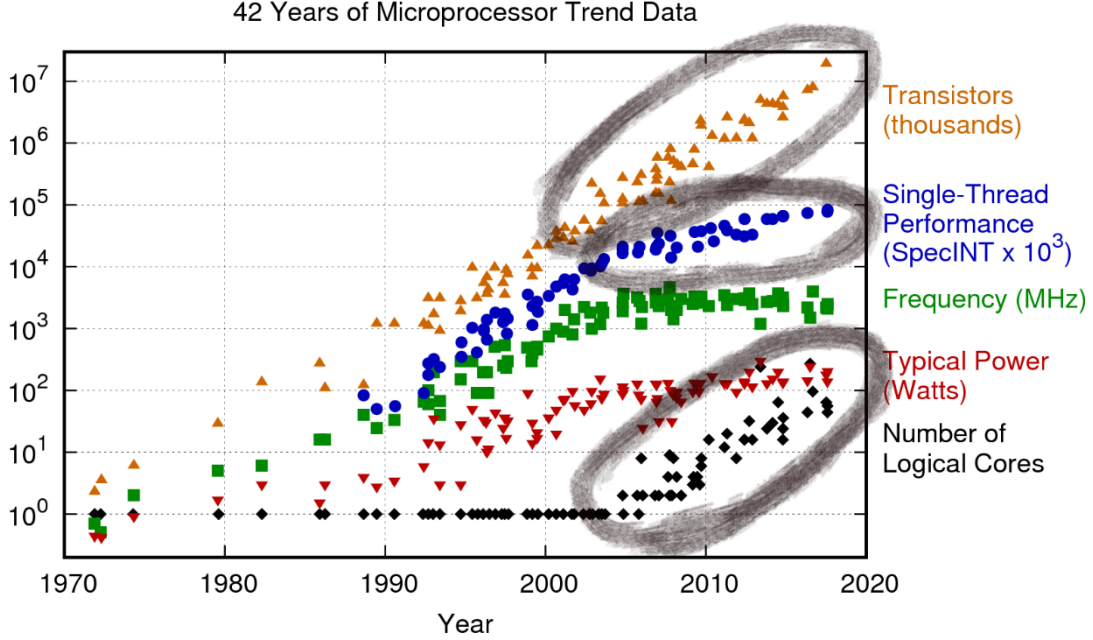


Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)

The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

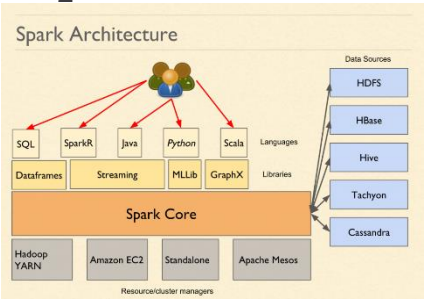
Big Data Parallel Programming Course

Goals

Processing vast amounts of data is at the core of data mining, deep learning and real-time autonomous decision making. All these are in turn at the core of modern artificial intelligence applications. Data can reside more or less permanently in the cloud and accessed via distributed file systems and/or be streamed in real-time from multiple sensors at very high rates.

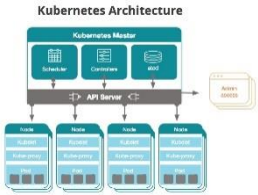
Access to data is done through well-engineered frameworks where both storage and processing are distributed, so that operations can be done in parallel. This course introduces you to this infrastructure, including parallel programming for the implementation of these frameworks. This should enable you to judge how to choose a framework for your applications, identify pros and cons, suggest improvements and even implement specific adaptations that you may need.

Three Topics in the BDPP Course



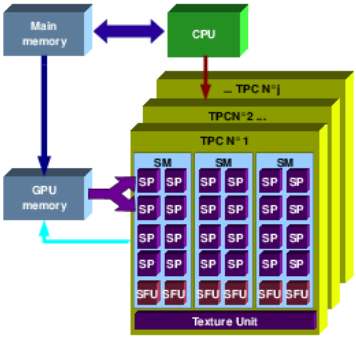
4 lectures, 2 labs & a project

kubernetes



1 lecture & 1 lab

NVIDIA CUDA



2 lectures & 2 labs

Big Data Parallel Programming - teachers

Slawomir Nowaczyk

Peyman Mashhadi

Mahmoud Rahat

Urban Bilstrup

Sepideh Pashami

Abdallah Alabdallah

Learning Outcomes

- Explain distributed file systems, distributed systems processing concepts, and parallel programming concepts.
- Describe tools and programming languages for storing and processing data in machine learning and data mining applications.
- Use specialized frameworks and methods for programming machine learning and data mining applications, such as Hadoop, Spark, etc.
- Use specialized programming languages for GPUs, such as CUDA and/or OpenCL for achieving high-performance.
- Discuss suitable methods and tools for storing and processing data in machine learning and data mining applications.

Teaching Format

Lectures

We will cover distributed file systems & processing concepts, parallel programming paradigms, specialized methods, frameworks and tools. The lectures will introduce the material in the literature.

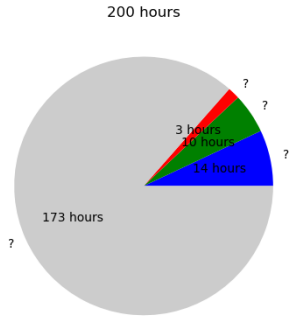
Labs

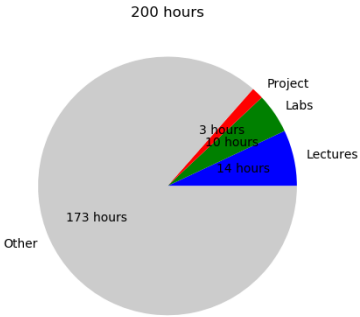
Labs include hands-on exercises on data-intensive computing and parallel programming. They are used to allow you to develop skills and practical abilities in the area. Labs are mandatory & graded.

Project

Final project on data-intensive computing (Spark part) gives you a chance to demonstrate the knowledge acquired during the course in the context of a realistic application. This is by far the largest part of the course, and it determines the final grade.

Formalities





Examination

To pass the course you need to pass:

- the labs (2.5 credits, Pass/Fail) – work in pairs, reports in Jupyter.
- the project (5 credits, 5-4-3/Fail) – individual work; design and implement a program, write a report & pass oral exam [which includes a presentation of your project and answering questions both about your solution as well as general course material]

Questions?

Big Data Parallel Programming

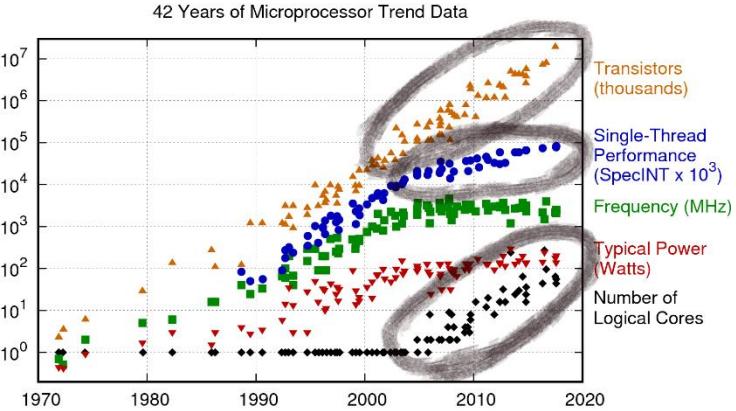
Part 2: Parallel Programming

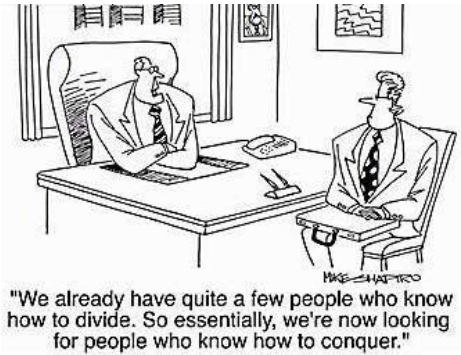
Slawomir Nowaczyk

Parallel Programming

Why?
The only way to scale up!

How?
Divide and conquer!





Two Kinds of Parallelism

Data parallelism

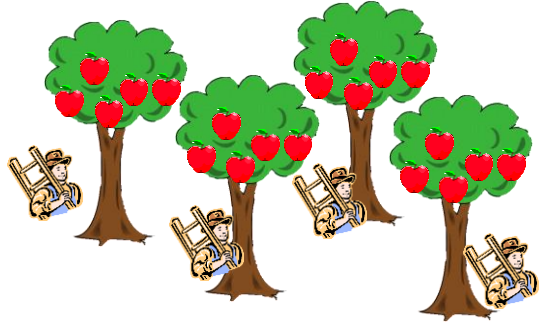
The same task run on different data in parallel

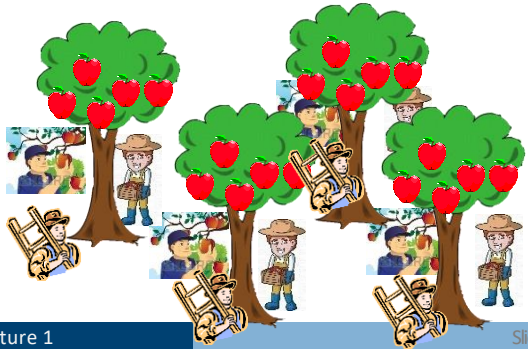
Task parallelism

Different tasks running on the same data in parallel

Hybrid & Pipeline

In practice, most actual solutions lie somewhere along this spectrum… but it's not balanced!





Two Very Different Approaches

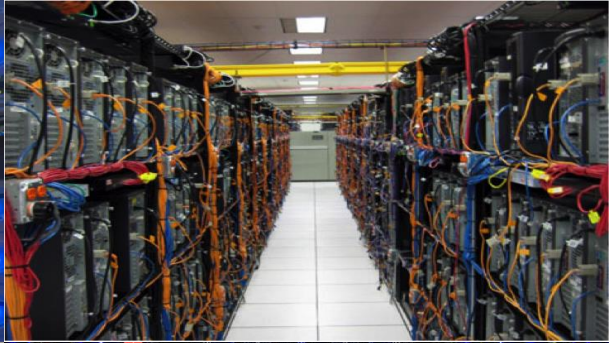
Two Very Different Frameworks

General-purpose frameworks

The course includes modern techniques, methods and tools for ***distributed storage*** (e.g., distributed and fault-tolerant key-value tables including replication and coordination mechanisms) as well as for ***distributed processing*** (e.g., frameworks such as MapReduce and Spark). It covers both static and streamed massive data. As a principle, all kinds of algorithms and data types are supported.

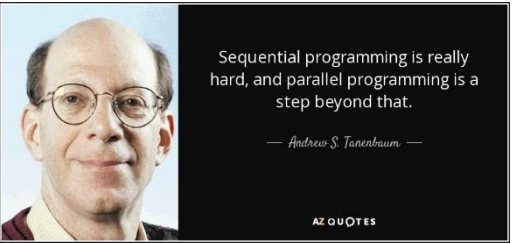
Specialised frameworks

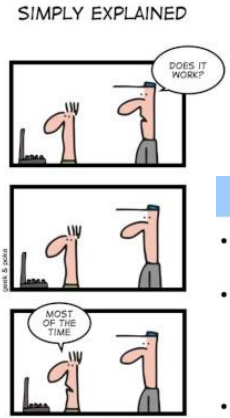
In addition, the course includes concepts, methods and tools for parallel programming for homogenous clusters, in particular CUDA library for GPUs. As a principle, this kind of processing is highly specialised and supports only a limited set of operations.


RTX 2080Ti
NVIDIA

Parallel Programming is hard





- The human mind tends to be sequential
- The notion of time is often misleading
- Thinking about all possible sequences of events in a computer system is at least error prone and frequen tly impossible

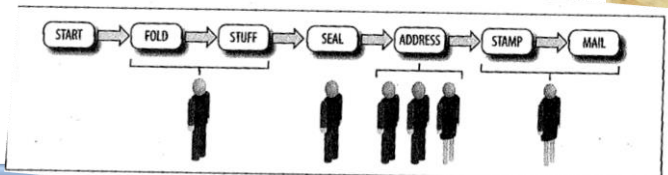
Concurrent programming is **HARD**

- Concurrent execution is difficult to reason about and get right (even for experts!)
- Rare process interleaving results in bugs that are
  - hard to anticipate
  - difficult to find, reproduce, and debug ("Heisenbugs")
  - hard to be sure whether they are really fixed
- Big productivity problem: it can waste significant developers' time and resources

- **Classical problem classes of concurrent programs:**
  - ***Races:*** outcome depends on arbitrary scheduling decisions elsewhere in the system
    - Example: who gets the last seat on the airplane?
  - ***Deadlock:*** improper resource allocation prevents forward progress
    - Example: traffic gridlock
  - ***Livelock / Starvation / Fairness***: external events and/or system scheduling decisions can prevent sub-task progress
    - Example: people always jump in front of you in line

Pipeline parallelism

igure 2-7. Unequal tasks are best combined or split to match



equal, assign more people to the addressing task

Assumption

Output of one task is the input to the next

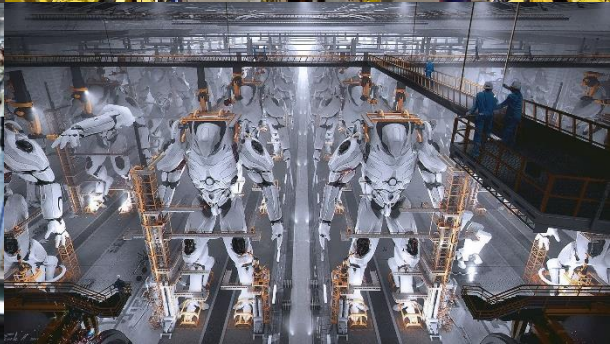
Benefits

All the tasks can run in parallel

Limitations

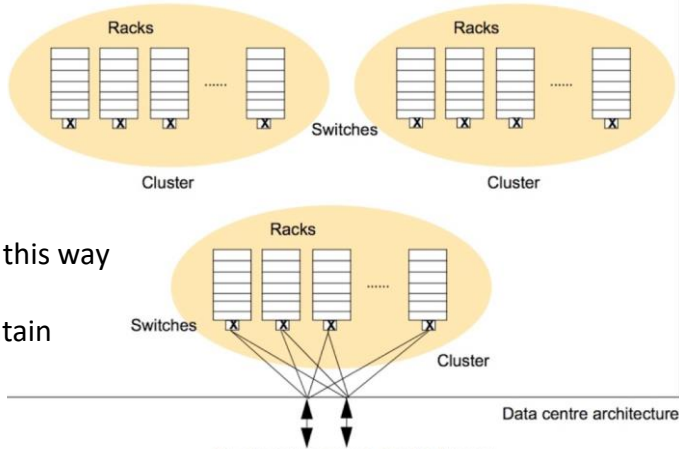
Limited by the highest-latency element


组装



Programming the Pipeline

- MapReduce pattern
  - inspired by Lisp, ML, etc.
  - functional programming
- Expressive paradigm
  - many problems can be phrased this way
  - results in clean code
  - easy to implement/debug/maintain
- Simple programming model
  - built-in retry/failure semantics
  - efficient and portable
  - easy to distribute across nodes

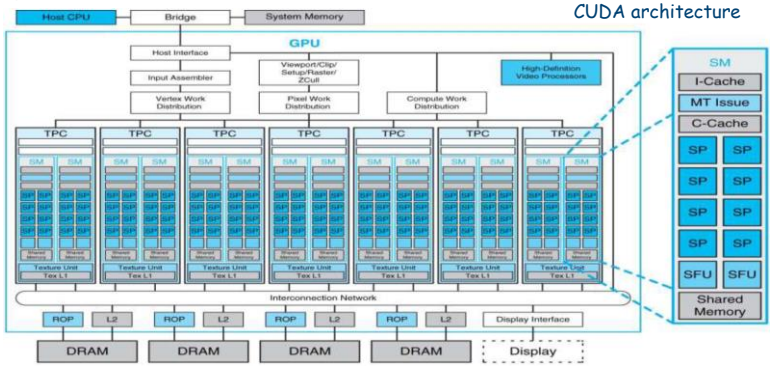


Programming the Pipeline

- Specialised compute-intensive highly parallel repetitive processing
  - originally designed for graphics and physics calculations (in computer games)
  - now mostly used for Machine Learning with Deep Artificial Neural Networks

- Heterogeneous computing model
  - two interacting units
  - host (the CPU)
  - device (the GPU)

- Matrices / tensors
  - a single class of problems
  - common & important
  - worth creating dedicated specialised hardware



Spark

Why? – Process Big Data in a flexible way, with arbitrary code
How? – Build upon ideas from functional programming
The trick? – Forbid global state & automate worker management

CUDA

Why? – Make very repetitive computations on data run very fast
How? – Only allow simple math operations on tensors/matrices
The trick? – Use dedicated hardware, carefully designed to the task

Big Data Parallel Programming

Part 3: Parallel Programming Example

Slawomir Nowaczyk

Simple Example

Goals

- Introduce a little bit of Python programming
- Discuss the difference between having “small” or “big” data
- Design an algorithm for counting the number of word occurrences
  - i.e., how many times each word is used in a given text

Counting Word Occurrences in a (small) File

An Algorithm

- Read all the text from the file, and split into words
- Count the number of occurrences of each unique word
- Write the output to a file or a screen
  - each unique word & the number of times it occurs in the file

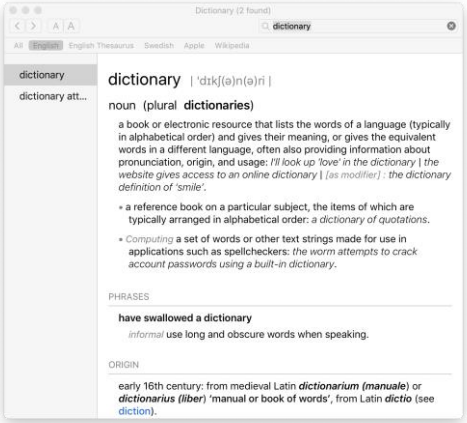
Simple Code

Reading text from a file

- Control structure `with` takes care of resource management
  - it will open the file and make sure it gets closed correctly
- Method `read()` loads the content of the file into memory
- Method `split()` applied to a string returns a list of sub-strings separated by whitespaces and newlines

```
with open("shakespeare.txt", 'r') as ff:
    text = ff.read().split()
```

A Little on Data Structures



Dictionary in Python

A data structure that allows you to:

- create an empty dictionary
- insert a {key, value} pair
- efficiently(!) lookup the value associated to a key
- remove a key, value pair
- iterate through all the keys

It is usually implemented using hash tables or search trees

Counting Words

Python *dict* reference

```
d = dict()
d[key] = value
d.get(key,if_not)
key in d
del d[key]
for key in d:
```

Algorithm

- Create an empty dictionary
  - keys will be words
  - values will be counts
- Insert new words with count=1
- Update existing words by incrementing the value already in the dictionary

Method `word.lower()` turns all characters to lower case
(we consider the word “Word” to be the same word as “word”)

```
counts = dict()
for word in text:
    lowercase = word.lower()
    counts[lowercase] = 1 + counts.get(lowercase,0)
```

The Complete Python Program

```
18126: of
27729: the
 5930: this
 9168: is
 8000: for
26099: and
 7981: with
10696: you
 5879: it
14436: a
10730: in
```

emacs@DEIRDRE

File Edit Options Buffers Tools Python Virtual Envs Elpy Flymake YASnippet Help

```
with open("shakespeare.txt", 'r') as ff:
    text = ff.read().split()
counts = dict()
for word in text:
    lowercase = word.lower()
    counts[lowercase] = 1 + counts.get(lowercase,0)
total = 0
for word,nr in counts.items():
    if nr > 5000:
        total += 1
        print(f"{nr:5}: {word}")
print("Total:",total)
```

Consider "Big Data"

Count the occurrences of words on the web

Based on estimate of J Leskovec, A Rajaraman, J Ullman (Stanford University, mmds.org) "the web" is 20+ billion web pages at 20kb…

That's a total of 400+ TB of data…

Given that a single computer reads ≈30-35 MB/sec from HDD/SSD…

We would need approximately 4 months to read all the web…

And ≈200 hard drives to store this information while processing!

Counting Word Occurrences in a (BIG) File

Goals

- Present an algorithm that can be parallelised
- A little bit more Python programming
- Discuss the abstract structure of the new algorithm
- Consider other problems that exhibit similar structure
  - i.e., that can be solved with the same class of algorithms
- Motivate the need for general-purpose distributed infrastructure

Counting Word Occurrences in a (BIG) File

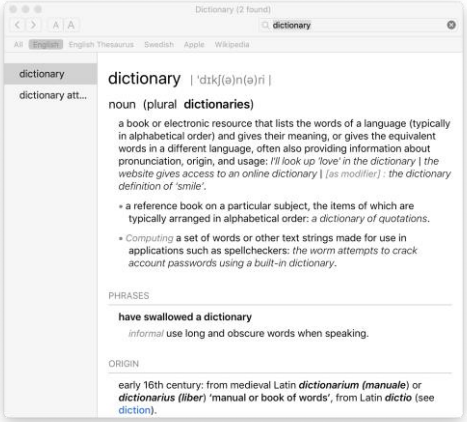
Small File (a reminder)

- Read all the text from the file, and split into words
- For each word, increment the count in a dictionary

Divide-and-Conquer

- Divide file into parts, read text from each part, and split into words
- Count the number of occurrences of unique words in each part
- Merge all the partial results together (somehow)
- Write the output to a file or a screen

Data Structures Again



Lists rather than dictionaries

A dictionary data structure is quite complicated internally, and not easily parallelisable.

However, we can use lists instead

- they are much simpler to maintain
- support the same operations, e.g., add/remove elements & iteration
- efficient membership test if sorted
- lists can be merged in linear time

In particular, lists of *key-value* pairs.

Counting Words – Parallel Version

Algorithm

Pair each word read from the file with a number "1"

Sort the resulting list of pairs in the word order

Group all "1"s for each, i.e., now pair each word with a list of "1"s

Sum up all the numbers in the list for each word

[these, are, some, words, and, these, are, more, words]

[('these', 1), ('are', 1), ('some', 1), ('words', 1), ('and', 1), ('these', 1), ('are', 1), ('more', 1), ('words', 1)]

[('and', 1), ('are', 1), ('are', 1), ('more', 1), ('some', 1), ('these', 1), ('these', 1),('words', 1), ('words', 1)]

[('and', [1]), ('are', [1, 1]), ('more', [1]), ('some', [1]), ('these', [1, 1]), ('words', [1, 1])]

[('and', 1), ('are', 2), ('more', 1), ('some', 1), ('these', 2), ('words', 2)]

Counting Words – Parallel Version

Algori

Pair each wo
the file with

Sort the result
pairs in the word or

Group all “1”s for each, i.e., now pair each word with a list of “1”s

Sum up all the numbers in the list for each word

[these, are, some, words, and, these, are, more, words]

[(‘these’, 1), (‘are’, 1), (‘some’, 1), (‘words’, 1), (‘and’, 1), (‘these’, 1), (‘are’, 1), (‘more’, 1), (‘words’, 1)]

(‘are’, 1), (‘more’, 1),
(‘words’, 1), (‘words’, 1)]

[(‘and’, 1), (‘a

EACH OF THESE STEPS CAN BE PARALLELISED!

The Abstract Algorithm

Map – Reduce

The Map-Reduce framework consists of two steps… the first is called "Map" and the second is called "Reduce" (after LISP functions).

Map

Apply the ***same*** function to all the elements in the input list.
In the example, it was "pair with 1" – that's a function. This can be done in parallel, over chunks of the input, equal or different sizes…
Then, instead of one list, we will get several lists!

Reduce

The last part is called reducing: we reduce a list of values to a value by applying an accumulating function: plus This can also be done in parallel, we can take parts of the list and reduce those words.
Instead of one output file we might get several output files!

Big Data Parallel Programming

Part 4: MapReduce Paradigm

Slawomir Nowaczyk

Counting Word - MapReduce in Python

```
with open("shakespeare.txt", 'r') as ff:
    text = filter(None,re.split('[^a-zA-Z\']+', ff.read()))

mapped = []
for word in text:
    mapped.append((word.lower(),1))
mapped.sort()
grouped = group(mapped)
reduced = []
for word, vals in grouped:
    reduced.append((word, sum(vals)))

total = 0
for word,nr in reduced:
    if nr > 5000:
        total += 1
        print(f"{nr:5}: {word}")
print("Total:",total)
```

The Secret/Automatic Step

itertools.**groupby**(*iterable*[, *key*])

Make an iterator that returns consecutive keys and groups from the *iterable*. The *key* is returns the element unchanged. Generally, the iterable needs to already be sorted on the

```
def group(iter):
    accumulator = []
    for pair in iter:
        if not accumulator:
            accumulator.append(pair)
        elif accumulator[0][0] == pair[0]:
            accumulator.append(pair)
        else:
            key = accumulator[0][0]
            yield (key,[vv for kk,vv in accumulator])
            accumulator = [pair]
```

The Goal of "Parallel Platforms/Frameworks"

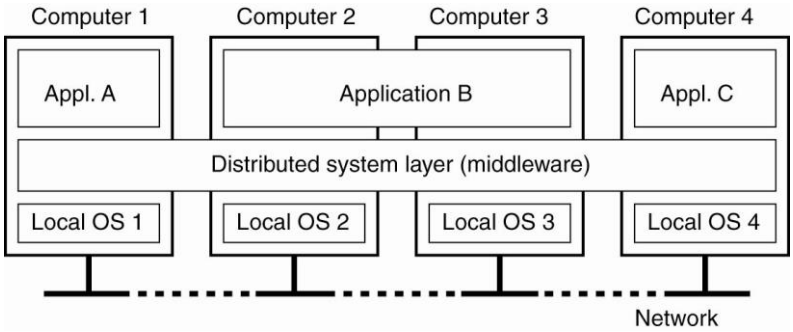
- Provide services with higher-level abstractions
  - e.g., file system, database, programming model, RESTful web service, …
- Hide complexity
  - scalable (scale-out)
  - reliable (fault-tolerant)
  - well-defined semantics (consistent)
  - security
- Do "heavy lifting" so the application developer doesn't have to
  - make common scenarios easy, and uncommon ones possible

What is a Distributed System?

- "A (large) collection of independent computers that appears to its users as a single coherent system"
- General features:
  - no shared memory
  - message-based communication
  - each runs its own local OS
  - possible heterogeneity
- Ideally, to present a single-system image
  - the distributed system can execute arbitrary code that a single computer could execute, with the same result…
  - just faster!

Distributed System as a Middleware

- A distributed system is generally organised as a middleware layer
- Runs on all machines & offers a uniform interface towards programmer



MapReduce as Middleware

- "A simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs."
- More simply, MapReduce is a parallel programming model
- Today we have a number of implementations following this paradigm
  - originally developed internally at Google (initial white paper, 2004)
  - first open source implementation was Apache Hadoop in 2005
  - we will be working with Apache Spark, initially created in 2014
  - but there is a lot more…

MapReduce Overview

- Remember our original big data problem of counting the number of occurrences of each word in a large collection of documents
- How did we make it run in parallel?
- Solution
  - divide documents among workers
  - each worker parses document to find all words
  - each worker outputs (word, count) pairs
  - partition (word, count) pairs across workers based on “word”
  - for each word at a worker, locally add up counts
  - aggregate the results in the end

MapReduce Programming Model

- Inspired by “map” and “reduce” operations commonly used in functional programming languages like Lisp
- Main data structure
  - a list of key-value pairs
  - or, conceptually, a set of such pairs!
- User supplies two functions:
  - map(k, v) → list(k_1,v_1)
  - reduce(k_1, list(v_1)) → v_2
- (k_1,v_1) is an intermediate key/value pair
- Output is the set of (k_1,v_2) pairs

Data Management Approaches

- **High-performance single machines**
  - "Scale-up" with limits (hardware, software, costs)
  - Workloads today are beyond the capacity of any single machine
  - I/O Bottleneck
- **Parallel Databases**
  - Fast and reliable
  - "Scale-out" restricted to some hundreds machines
  - Maintaining & administrations of parallel databases is hard
- **Specialized cluster of powerful machines**
  - "specialized" = powerful hardware satisfying individual software needs
  - fast and reliable but also very expensive
  - For data-intensive applications: scaling "out" is superior to scaling "up" → performance gab insufficient to justify the price

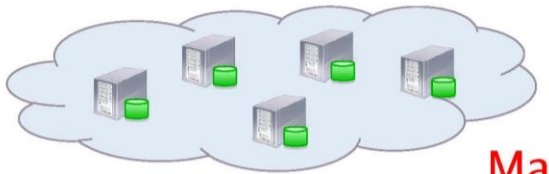
MapReduce

- **MapReduce**
  - Popularized by Google & widely used
  - Algorithms that can be expressed as (or mapped to) a sequence of Map() and Reduce() functions are automatically parallelized by the framework
- **Distributed File System**
  - Data is split into equally sized blocks and stored distributed
  - Clusters of commodity hardware
    → Fault tolerance by replication
  - Very large files / write-once, read-many pattern
- **Advantages**
  - Partitioning + distribution of data
  - Parallelization and assigning of task
  - Scalability, fault-tolerance, scheduling,…

That all is done automatically!

Distributed Processing of Data

- **Problem:** How to compute the PageRank for a crawled set of websites on a cluster of machines?



MapReduce!

- **Main Challenges:**
  - How to break up a large problem into smaller tasks, that can be executed in parallel?
  - How to assign tasks to machines?
  - How to partition and distribute data?
  - How to share intermediate results?
  - How to coordinate synchronization, scheduling, fault-tolerance?

Programming Models -- von Neumann model

- Execute a stream of instructions (machine code)
- These instructions can specify
  - arithmetic operations
  - data addresses
  - next instruction to execute
- Complexity
  - track billions of data locations and millions of instructions
- Manage with
  - modular design
  - high-level programming languages

Programming Models – Parallel Programming Models

- Message passing
  - independent tasks encapsulating local data
  - tasks interact by exchanging messages
- Shared memory
  - tasks share a common address space
  - tasks interact by reading and writing this space asynchronously
- Data parallelisation
  - tasks execute a sequence of independent operations
  - data usually evenly partitioned across tasks
  - also referred to as “embarrassingly parallel”
- Challenge: unacceptable ease of use vs efficiency trade-offs

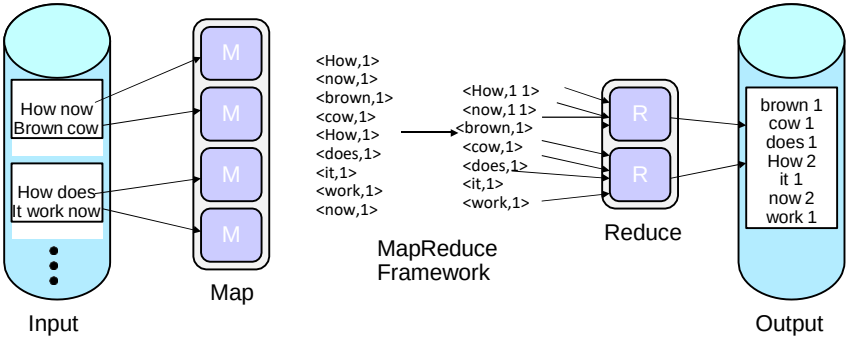
Programming Model – MapReduce

- Process data using a limited set of functions, map() and reduce()
- The map() function is called on every item in the input and emits a series of intermediate key/value pairs
- All values associated with a given key are grouped together
- The reduce() function is called on every unique key, and its value list, and emits a value that is added to the output
- Benefits:
  - limited language expressivity allows for very efficient execution
  - specialised runtime engines offer high scalability and performance

Success = Programming Model + Engine

- Simple control flow makes dependencies evident
- Engine can automate scheduling of tasks and optimization
  - map & reduce for different keys is embarrassingly parallel
  - pipeline between mappers & reducers is evident
- map() and reduce() are pure functions
  - engine can re-run them to get the same answer
  - in the case of failure, or to use idle resources toward faster completion
- No need to worry about typical concurrency problems
  - data races, deadlocks, and so on since there is no shared state
- The execution engine itself is very complex
  - but it only needs to be created once!

MapReduce: Programming Model



MapReduce Workflow

(1) **Map Phase**
- Raw data read and converted to key/value pairs
- Map() function applied to any pair

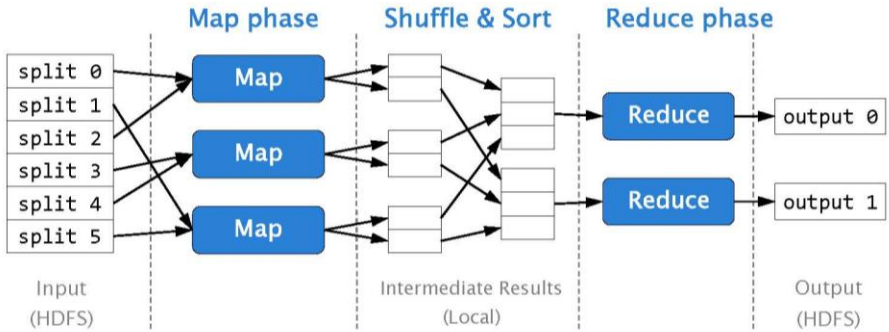
(2) **Shuffle Phase**

THIS IS AUTOMATIC!
- All key/value pairs are sorted and grouped by their keys

(3) **Reduce Phase**
- All values with a the same key are processed by within the same reduce() function

MapReduce Workflow (2)

- Steps of a MapReduce execution



- Signatures
  - `Map():` (in_key, in_value) → *list* (out_key, intermediate_value)
  - `Reduce():` (out_key, *list* (intermediate_value)) → *list* (out_value)

More Examples

Design algorithms to calculate…

- … the largest integer
- … the average of all the integers
- … the same set but with each integer appearing only once

(in all cases assume input is a large set of integers)

MapReduce Benefits

- Greatly simplifies challenges of parallel programming
- Reduces synchronization complexity
- Automatically partitions data
- Provides failure transparency
- Handles load balancing
- Limited but practical
  - approximately 1000 Google MapReduce jobs were running everyday within the first year (2004)

The MapReduce Engine

No Free Lunch???

The engine will automatically do the following:

- partition the input data
- schedule the program execution across a set of machines (the mappers, the reducers, the sorters and the groupers)
- handle machine failures & restart computations
- manage inter-machine communication

Programmer's Job

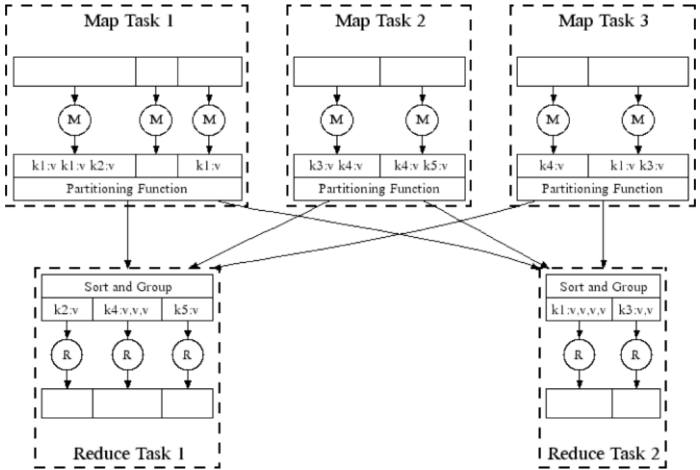
As a programmer, you only need to say what to do on each piece of data – for the mapping part, map() – and what to do on each piece of grouped value – for the reducing part, reduce()!

Scale vs Failure Tolerance

- (Source: Indranil Gupta, Lectures 2-3 distributed systems CS425)
  - Facebook (GigaOm): 30K machines in 2009 → 60K in 2010 → 180K 2012
  - Microsoft (NYTimes, 2008): 150K, growth rate of 10K per month
  - Yahoo! 2009: 100K machines, split into clusters of 4000
  - eBay (2012): 50K machines
  - HP (2012): 380K machines
  - Google (2011, Data Center Knowledge): 900K machines (currently over 1M)

- At this size, failure is the norm
  - rate of failure one machine 10 years
  - for 120 servers the mean time to failure of next machine is 1 month
  - for 12000 servers MTTF is 7.2 hours

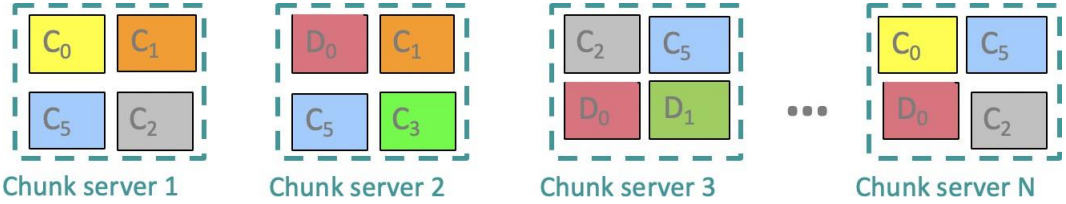
Task Execution



J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, http://www.mmds.org

Chunk Servers

J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, http://www.mmds.org



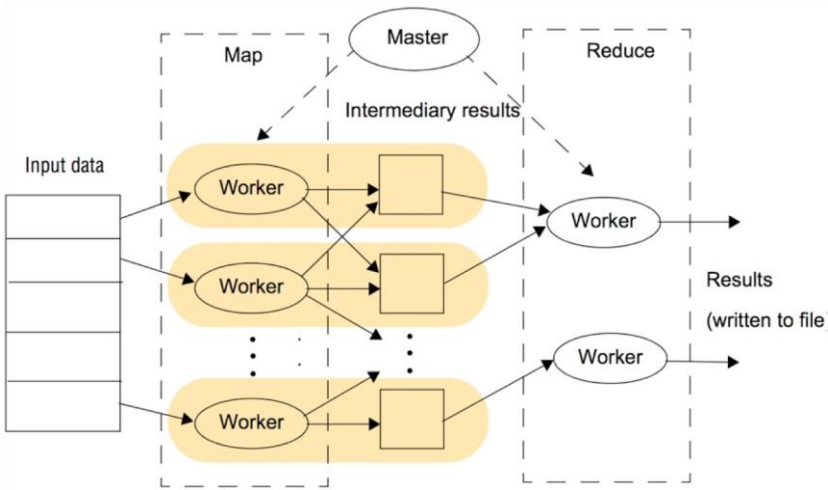
Data Split

Data is stored in “chunk servers” (distributed and redundant)

And the chunk servers can also be doing computations…

But then we need to organize computations in ways that they can work on pieces of data, for example Map-Reduce algorithms!

Overall Execution of Map-Reduce



More MapReduce Examples

1. Map-only processing
2. Filtering and accumulation
3. Database join
4. Reversing graph edges
5. Producing inverted index for web search
6. PageRank graph processing

MapReduce Use Case 1: Map Only

- Data distributive tasks – Map Only
  - e.g. classify individual documents
- Map does everything
  - input: (docno, doc_content), …
  - output: (docno, [class, class, …]), …
- No reduce tasks

MapReduce Use Case 2: Filtering and Accumulation

- Filtering & Accumulation – Map and Reduce
  - e.g. counting total enrollments of two given student classes (e.g. 117 & 114)
- Map selects records and outputs initial counts
  - in: (Jamie, 117), (Tom, 114), (Jim, 115), … → out: (117, 1), (114, 1), …
- Shuffle/partition by class-id
- Sort
  - in: (117, 1), (114, 1), (117, 1), … → out: (114, 1), …, (117, 1), (117, 1), …
- Reduce accumulates counts
  - in: (114, [1, 1, …]), (117, [1, 1, …])
  - sum and output: (114, 16), (117, 35)

MapReduce Use Case 3: Database Join

A JOIN is a means for combining fields from two tables by using values common to each. For example, for each employee, find the department they work in…

Employee Table	
LastName	**DepartmentID**
Rafferty	31
Jones	33
Steinberg	33
Robinson	34
Smith	34

JOIN

Predicate: EMPLOYEE.DepID = DEPARTMENT.DepID

JOIN RESULT	
LastName	**DepartmentName**
Rafferty	Sales
Jones	Engineering
Steinberg	Engineering
…	…

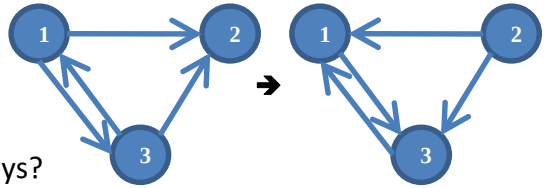
Department Table	
DepartmentID	**DepartmentName**
31	Sales
33	Engineering
34	Clerical
35	Marketing

MapReduce Use Case 3 – Database Join

- Problem is massive lookups
  - given (URL, ID) and (URL, doc_content) pairs produce (URL, ID, doc_content)
- Input stream is both (URL, ID) and (URL, doc_content) lists
  - in: (http://del.icio.us/, 0), (http://digg.com/, 1), …
  - in: (http://del.icio.us/, <html0>), (http://digg.com/, <html1>), …
- Map simply passes input along…
- Shuffle & sort on URL (group ID & doc_content for URL together)
  - (http://del.icio.us/, 0), (http://del.icio.us/, <html0>), (http://digg.com/, <html1>), (http://digg.com/, 1), …
- Reduce outputs result stream of (ID, doc_content) pairs
  - In: (http://del.icio.us/, [0, html0]), (http://digg.com/, [html1, 1]), …
  - Out: (http://del.icio.us/, 0, <html0>), (http://digg.com/, 1, <html1>), …

MapReduce Use Case 4: Reverse graph edge directions (& output in node order)

- Input example: adjacency list of graph (3 nodes and 4 edges)
  - (3, [1, 2]) (1, [3])
  - (1, [2, 3]) ➔ (2, [1, 3])
  - (3, [1])



- Sort node-ids in the output
  - even though MapReduce only sorts on keys?
- MapReduce format
  - Input: (3, [1, 2]), (1, [2, 3])
  - Intermediate: (1, [3]), (2, [3]), (2, [1]), (3, [1]) {reverse edge direction}
  - Out: (1,[3]) (2, [1, 3]) (3, [1])

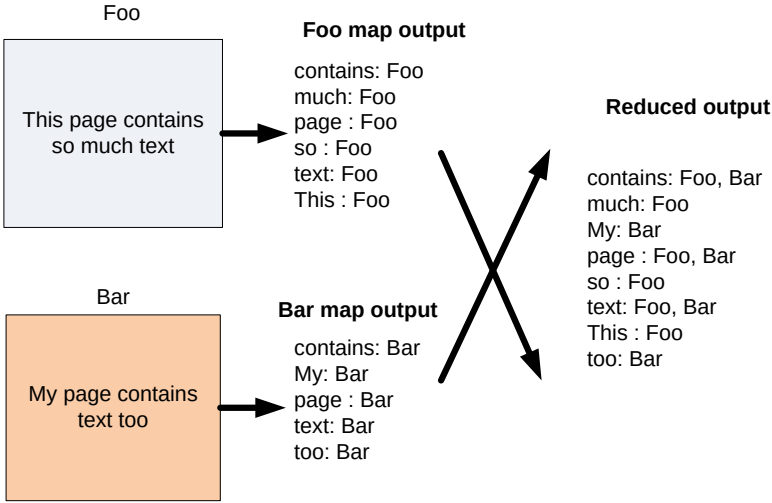
MapReduce Use Case 5: Inverted Indexing Preliminaries

- Construction of inverted lists for document search
- Input: documents: (docid, [term, term..]), (docid, [term, ..]), ..
- Output: (term, [docid, docid, …])
  - e.g., (apple, [1, 23, 49, 127, …])
- A document id is an internal document id, e.g., a unique integer
- Not an external document id such as a URL

Using MapReduce to Construct Indexes: A Simple Approach

- A simple approach to creating inverted lists
- Each Map task is a document parser
  - input: A stream of documents
  - output: A stream of (term, docid) tuples
    - (long, 1) (ago, 1) (and, 1) … (once, 2) (upon, 2) …
    - we may create internal IDs for words
- Shuffle sorts tuples by key and routes tuples to Reducers
- Reducers convert streams of keys into streams of inverted lists
  - input: (long, 1) (long, 127) (long, 49) (long, 23) …
  - the reducer sorts the values for a key and builds an inverted list
  - output: (long, [df:492, docids:1, 23, 49, 127, …])

Inverted Index: Data flow



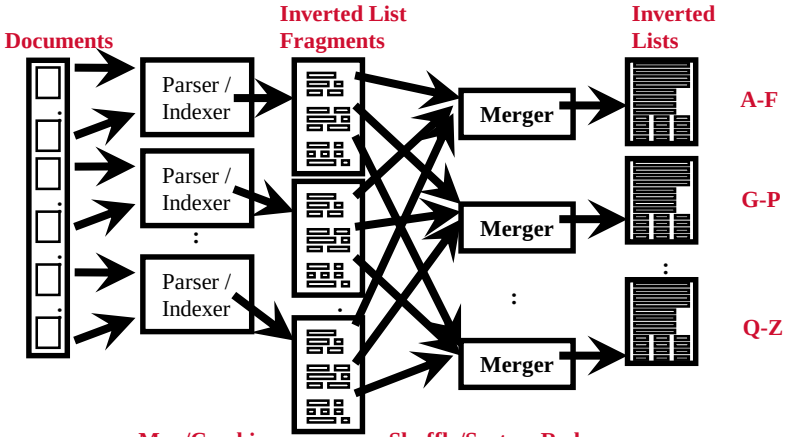
Processing Flow Optimization

- A more detailed analysis of processing flow
  - map: (docid1, content1) → (t1, docid1) (t2, docid1) …
- Shuffle by t, prepared for map-reducer communication
- Sort by t, conducted in a reducer machine
  - (t5, docid1) (t4, docid3) … → (t4, docid3) (t4, docid1) (t5, docid1) …
- Reduce: (t4, [docid3 docid1 …]) → (t, ilist)
  - docid: a unique integer
  - t: a term, e.g., “apple”
  - ilist: a complete inverted list
- but a) inefficient, b) docids are sorted in reducers, and c) assumes ilist of a word fits in memory

Using Combine () to Reduce Communication

- Map: (docid1, content1) → (t1, ilist1,1) (t2, ilist2,1) (t3, ilist3,1) …
  - each output inverted list covers just one document
- Combine locally
  - sort by t & combine: (t1 [ilist1,2 ilist1,3 ilist1,1 …]) → (t1, ilist1,27)
  - each output inverted list covers a sequence of documents
- Shuffle by t & sort by t
  - (t4, ilist4,1) (t5, ilist5,3) … → (t4, ilist4,2) (t4, ilist4,4) (t4, ilist4,1) …
- Reduce: (t7, [ilist7,2, ilist3,1, ilist7,4, …]) → (t7, ilistfinal)
  - ilisti,j: the j'th inverted list fragment for term i

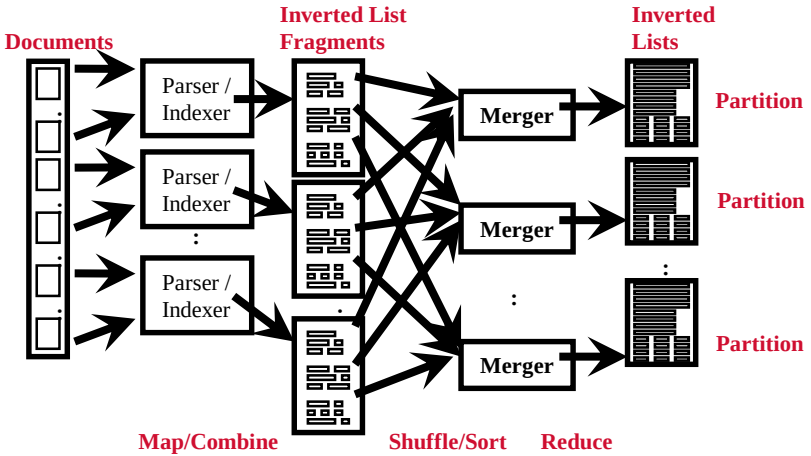
Using MapReduce to Construct Indexes



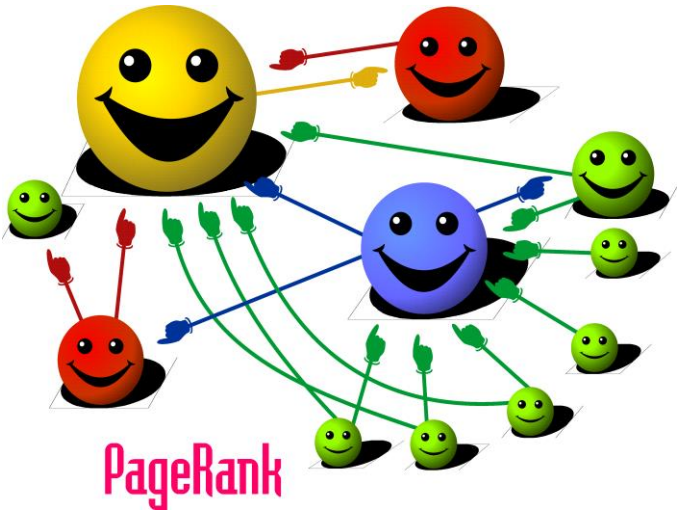
Construct Partitioned Indexes

- Useful when the document list of a term does not fit memory
  - Map: (docid1, content1) → ([p, t1], ilist1,1)
- Combine to sort and group values
  - ([p, t1] [ilist1,2 ilist1,3 ilist1,1 …]) → ([p, t1], ilist1,27)
- Shuffle by p & sort values by [p, t]
- Reduce: ([p, t7], [ilist7,2, ilist7,1, ilist7,4, …]) → ([p, t7], ilistfinal)
  - p: partition (shard) id

Generate Partitioned Index



MapReduce Use Case 6: PageRank

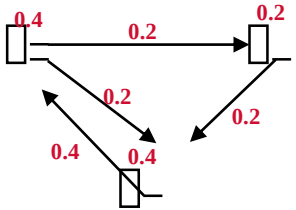


PageRank

- Model page reputation on the web

$$PR(x) = (1-d) + d\sum_{i=1}^{n} \frac{PR(t_i)}{C(t_i)}$$

- i=1,n lists all parents of page x.
- PR(x) is the page rank of each page.
- C(t) is the out-degree of t.
- d is a damping factor .

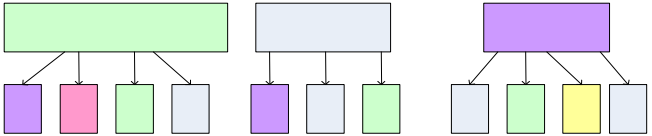


Computing PageRank Iteratively

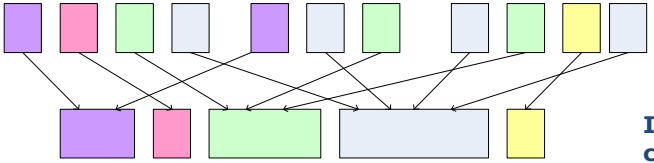
- Effects at each iteration is local. i+1th iteration depends only on ith iteration
- At iteration i, PageRank for individual nodes can be computed independently

PageRank using MapReduce

Map: distribute PageRank "credit" to link targets



Reduce: gather up PageRank "credit" from multiple sources to compute new PageRank value



Iterate until convergence

Source of Image: Lin 2008

PageRank Calculation: Preliminaries

- One PageRank iteration:
- Input:
  - (id1, [score1(t), out11, out12, ..]), (id2, [score2(t), out21, out22, ..]) ..
- Output:
  - (id1, [score1(t+1), out11, out12, ..]), (id2, [score2(t+1), out21, out22, ..]) ..
- MapReduce elements
- Score distribution and accumulation
- Database join

PageRank: Score Distribution and Accumulation

- Map
  - In: (id1, [score1(t), out11, out12, ..]), (id2, [score2(t), out21, out22, ..]) ..
  - Out: (out11, score1(t)/n1), (out12, score1(t)/n1) .., (out21, score2(t)/n2), ..
- Shuffle & Sort by node_id
  - In: (id2, score1), (id1, score2), (id1, score1), ..
  - Out: (id1, score1), (id1, score2), .., (id2, score1), ..
- Reduce
  - In: (id1, [score1, score2, ..]), (id2, [score1, ..]), ..
  - Out: (id1, score1(t+1)), (id2, score2(t+1)), ..

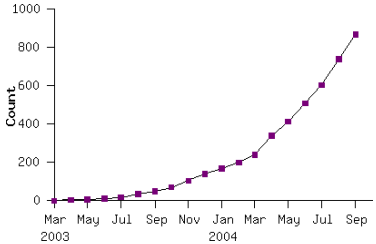
PageRank: Database Join to associate out-links with score

- Map
  - In & Out: (id1, score1(t+1)), (id2, score2(t+1)), .., (id1, [out11, out12, ..]), (id2, [out21, out22, ..]) …
- Shuffle & Sort by node_id
  - Out: (id1, score1(t+1)), (id1, [out11, out12, ..]), (id2, [out21, out22, ..]), (id2, score2(t+1)), ..
- Reduce
  - In: (id1, [score1(t+1), out11, out12, ..]), (id2, [out21, out22, .., score2(t+1)]), ..
  - Out: (id1, [score1(t+1), out11, out12, ..]), (id2, [score2(t+1), out21, out22, ..]) ..

Summary

- Programming model applicable to many different tasks
- Simple to conceptualise and understand
- Powerful engines taking care of technical details
- Widely used in practice
- Powers a lot of Big Data hype

MapReduce Programs In Google Source Tree at end of 2004



Example uses:

distributed grep	distributed sort	web link-graph reversal
term-vector / host	web access log stats	inverted index construction
document clustering	machine learning	statistical machine translation
...	...	...

Questions?