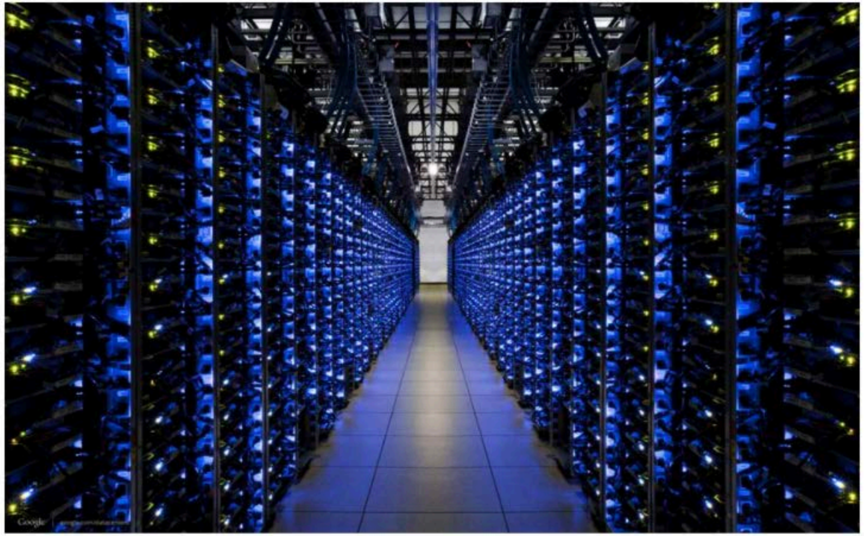
Warehouse Scale Computers, MapReduce

Instructor: Jenny Song



Review of Last Lecture

- OpenMP as simple parallel extension to C
  - Synchronization accomplished with critical/atomic/reduction
  - Pitfalls can reduce speedup or break program logic
- Cache coherence implements shared memory even with multiple copies in multiple caches
  - The protocol we learned was MOESI
  - False sharing renders a block useless! A ping-pong chain of invalidation
    - Coherence misses are the fourth cache miss type

Agenda

- Warehouse Scale Computers
- Cloud Computing
- Request Level Parallelism
- MapReduce

Great Idea #4: Parallelism

Software

- Parallel Requests
  Assigned to computer
  e.g. search “cs 61c”
- Parallel Threads
  Assigned to core
  e.g. lookup, ads
- Parallel Instructions
  > 1 instruction @ one time
  e.g. 5 pipelined instructions
- Parallel Data
  > 1 data item @ one time
  e.g. add a pair of 6 words
- Hardware descriptions
  All gates functioning in
  parallel at same time

Hardware

Warehouse
Scale
Computer

Smart
Phone

Leverage Parallelism & Achieve High Performance

Computer

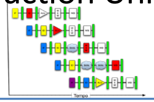
Core ... Core

Memory

Input/Output

Core

Instruction Unit(s)

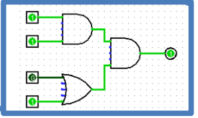


Functional Unit(s)

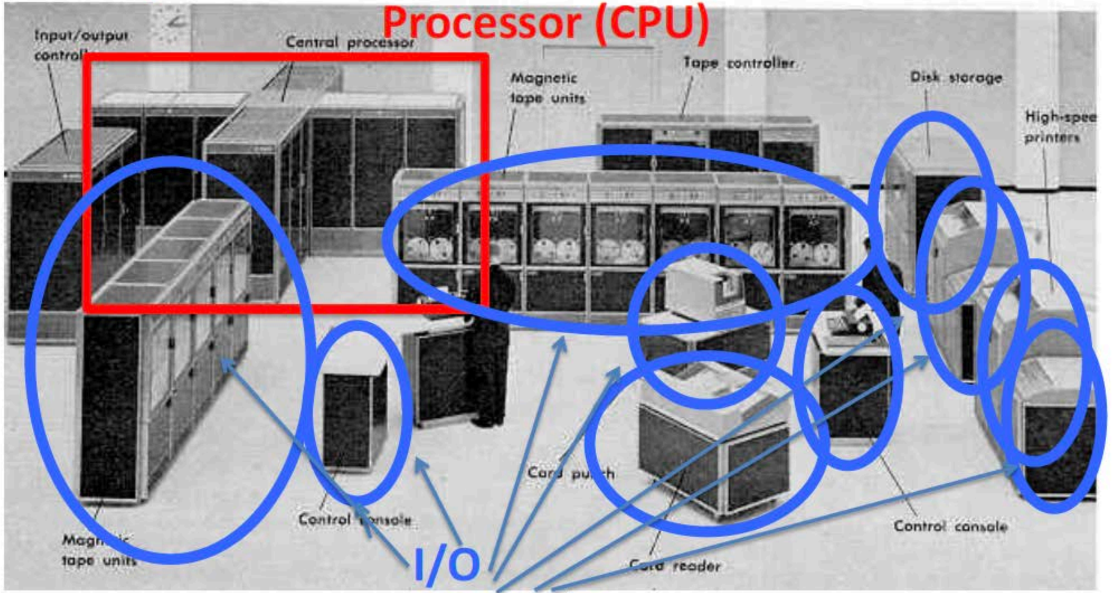
A_0+B_0 | A_1+B_1 | A_2+B_2 | A_3+B_3

Cache Memory

Logic Gates



Computer Eras: Mainframe 1950s-60s



"Big Iron": IBM, UNIVAC, … build $1M computers for businesses ➔ COBOL, Fortran, timesharing OS

Minicomputer Eras: 1970s

Using integrated circuits, Digital, HP… build $10k computers for labs, universities → C, UNIX OS

PC Era: Mid 1980s - Mid 2000s

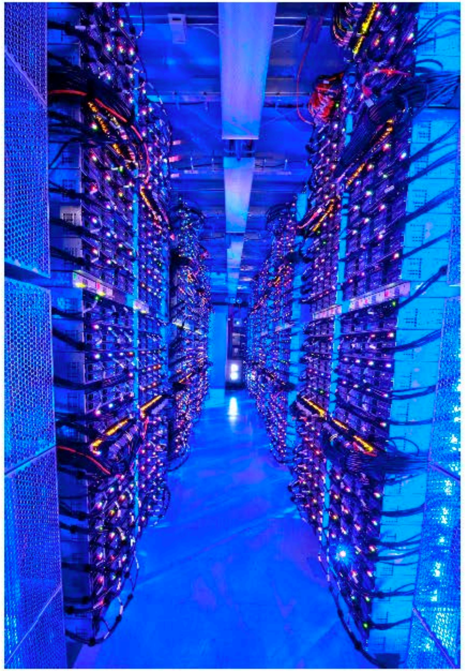
Using microprocessors, Apple, IBM, … build $1k computer for 1 person → Basic, Java, Windows OS

PostPC Era: Late 2000s - ??

Personal Mobile Devices (PMD): Relying on wireless networking, Apple, Nokia, … build $500 smartphone and tablet computers for individuals
➔ Objective C, Swift, Java, Android OS + iOS

Cloud Computing: Using Local Area Networks, Amazon, Google, … build $200M Warehouse Scale Computers with 100,000 servers for Internet Services for PMDs

➔ MapReduce, Ruby on Rails



7

Why Cloud Computing Now?

- "The Web Space Race": Build-out of extremely large datacenters (10,000's of ***commodity*** PCs)
  - Build-out driven by growth in demand (more users)
  - Infrastructure software and Operational expertise
- Discovered economy of scale: 5-7x cheaper than provisioning a medium-sized (1000 servers) facility
- More pervasive broadband Internet so can access remote computers efficiently
- Commoditization of HW & SW
- Better tooling for standardizing software

November 2019 AWS Instances & Prices
aws.amazon.com/ec2/pricing/on-demand

Instance	Per Hour	$ Ratio to Small	EC2 Compute Unit (integer)	Virtual Cores (vCPU)	Memory (GiB)	Disk (GiB)
Standard Small (t3.small)	$0.021	1	Variable	2	2	EBS
Standard Large (t3.large)	$0.083	4	Variable	2	8	EBS
Standard 2x Extra Large (t3.2xlarge)	$0.333	16	Variable	8	32	EBS
High-Mem Large (r5.large)	$0.140	6.7	9	2	16	EBS
High-Mem Double Xlarge (r5.2xlarge)	$0.504	24	38	8	64	EBS
High-Mem 24x Large (r5.24xlarge)	$6.048	288	347	96	768	EBS
High-CPU Large (c5.large)	$0.085	4	9	2	4	EBS
High-CPU 18x Large (c5.18xlarge)	$3.060	146	281	72	144	EBS

- Closest computer in WSC example is Standard 2X Extra Large
- At these low rates, Amazon EC2 can make money! (even utilized 50% time)
- EBS = Elastic Block Store (SSD=$0.12/GiB-month, HDD=$0.054/GiB-month)
- Each also comes with dedicated attached SSD if you choose & pay for that

Warehouse Scale Computers

- Massive scale datacenters: 10,000 to 100,000 servers + networks to connect them together
  - Emphasize cost-efficiency
  - Attention to power: distribution and cooling
  - (relatively) homogeneous hardware/software
- **Single gigantic** machine
- Offer very large applications (Internet services): search, voice search (Siri), social networks, video sharing
- Very highly available: < 1 hour down/year
  - Must cope with failures common at scale
- "…WSCs are no less worthy of the expertise of computer systems architects than any other class of machines" (Barroso and Hoelzle, 2009)

Design Goals of a WSC

- Unique to Warehouse-scale
  - *Ample parallelism*:
    - Batch apps: many independent data sets with independent processing (Data-Level and Request-Level Parallelism)
  - *Scale and its Opportunities/Problems*
    - Relatively small number of WSC make design cost expensive and difficult to amortize
    - But price breaks are possible from purchases of very large numbers of commodity servers
    - Must also prepare for high component failures
  - *Operational Costs Count*:
    - Cost of equipment purchases << cost of ownership

Google's Oregon WSC

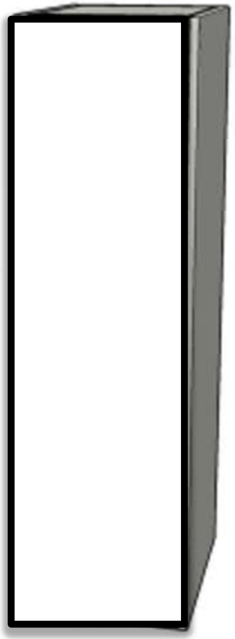
Containers in WSCs

Inside WSC

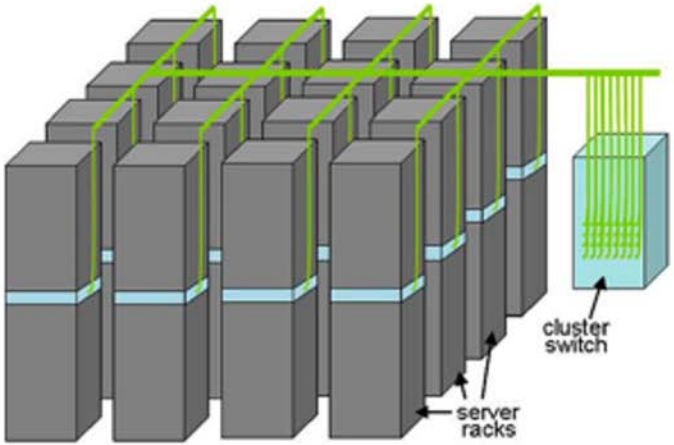
Inside Container

Equipment Inside a WSC

Server (in rack format):
1 ¾ inches high “1U”, x 19 inches x 16-20 inches: 8 cores, 16 GB DRAM, 4x1 TB disk



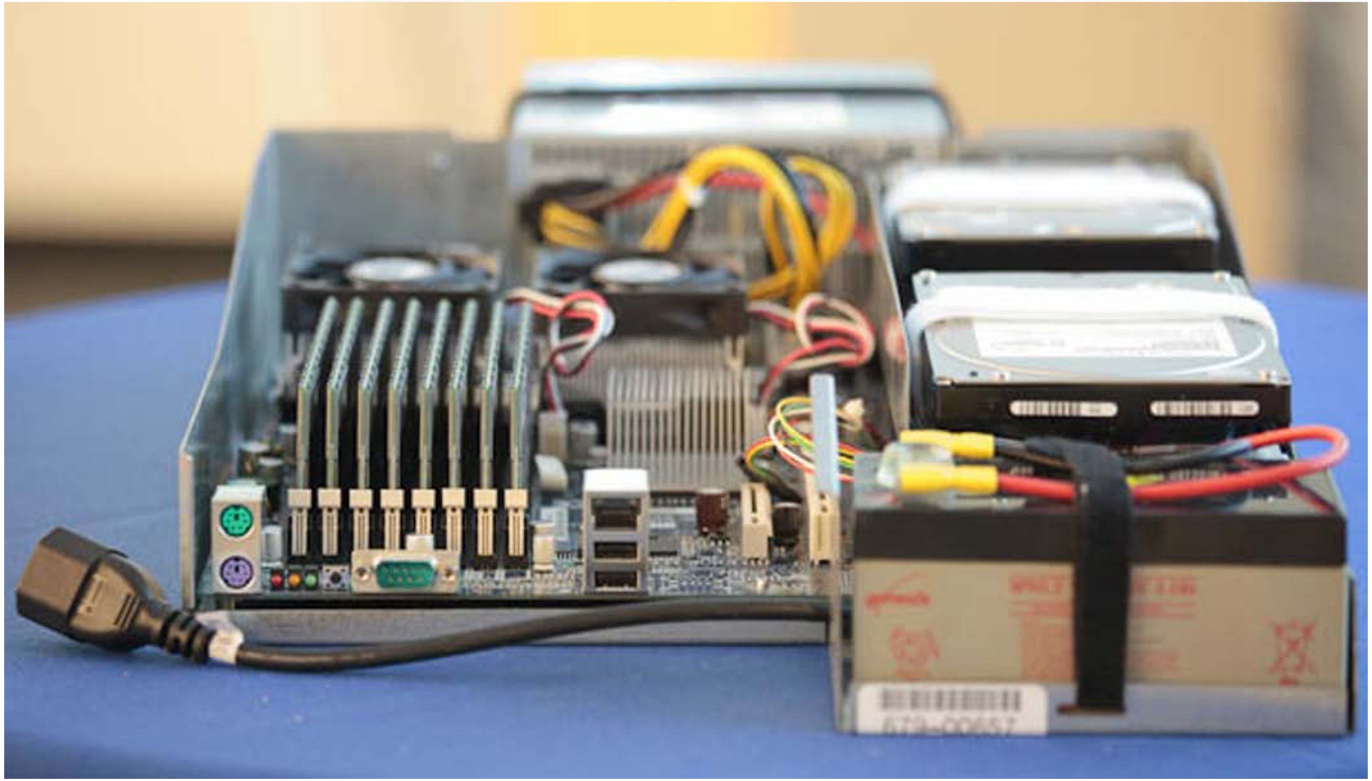
7 foot Rack: 40-80 servers + Ethernet local area network (1-10 Gbps) switch in middle (“rack switch”)



Array (aka cluster):
16-32 server racks + larger local area network switch (“array switch”)
10X faster => cost 100X: cost $f(N^2)$

Server, Rack, Array

Google Server Internals



Google Server Internals

Defining Performance

- What does it mean to say X is faster than Y?

- 2009 Ferrari 599 GTB
  - 2 passengers, 11.1 secs for quarter mile (call it 10sec)
- 2009 Type D school bus
  - 54 passengers, quarter mile time? (let's guess 1 min)

  https://youtu.be/ZSzOd__feIw?t=4s

- *Response Time* or *Latency*: time between start and completion of a task (time to move vehicle ¼ mile)
- *Throughput* or *Bandwidth*: total amount of work in a given time (passenger-miles in 1 hour)

Coping with Performance in Array

Lower latency to DRAM in another server than local disk

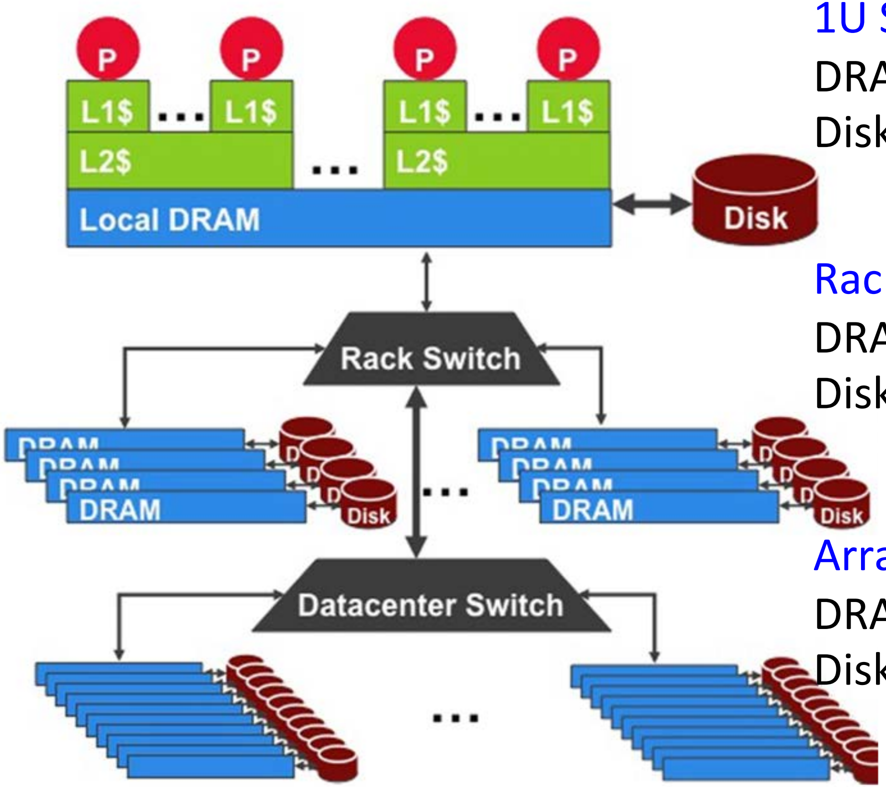
Higher bandwidth to local disk than to DRAM in another server

	Local	Rack	Array
Racks	--	1	30
Servers	1	80	2400
Cores (Processors)	8	640	19,200
DRAM Capacity (GB)	16	1,280	38,400
Disk Capacity (TB)	4	320	9,600
DRAM Latency (microseconds)	0.1	100	300
Disk Latency (microseconds)	10,000	11,000	12,000
DRAM Bandwidth (MB/sec)	20,000	100	10
Disk Bandwidth (MB/sec)	200	100	10

Coping with Performance in Array

Lower latency to DRAM in another server than local disk

Higher bandwidth to local disk than to DRAM in another server



1U Server:

DRAM: 16GB, 100ns, 20GB/s

Disk: 2TB, 10ms, 200MB/s

Rack(80 servers):

DRAM: 1TB, 300us, 100MB/s

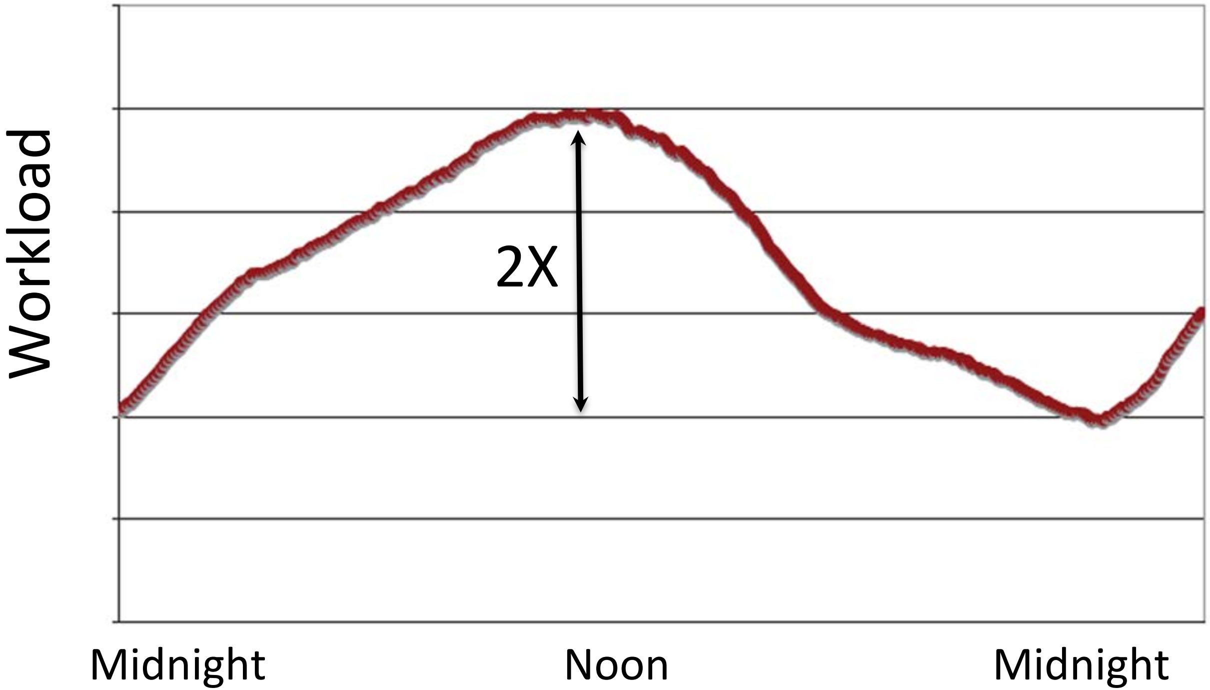
Disk: 160TB, 11ms, 100MB/s

Array(30 racks):

DRAM: 30TB, 500us, 10MB/s

Disk: 4.80PB, 12ms, 10MB/s

Coping with Workload Variation

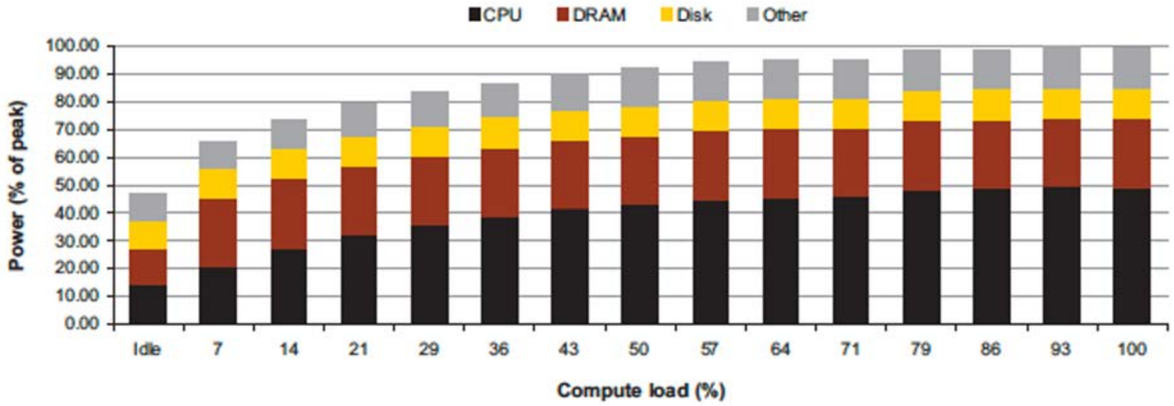


- Online service: Peak usage 2X off-peak

Impact of latency, bandwidth, failure, varying workload on WSC software?

- WSC Software must take care where it places data within an array to get good performance
  - **Latency & bandwidth** impact Performance
- WSC Software must cope with failures gracefully
  - **High failure rate** impact Reliability Availability
- WSC Software must scale up and down gracefully in response to varying demand
  - **Varying workloads** impact Availability
- More elaborate hierarchy of memories, failure tolerance, workload accommodation makes WSC software development more challenging than software for single computer

Power vs. Server Utilization



- Server power usage as load varies idle to 100%
- Uses ½ peak power when idle!
- Uses ⅔ peak power when 10% utilized! 90%@ 50%!
- Most servers in WSC utilized 10% to 50%
- Goal should be *Energy-Proportionality*:
  % peak load = % peak energy

Power Usage Effectiveness

- Overall WSC Energy Efficiency: amount of computational work performed divided by the total energy used in the process
- Power Usage Effectiveness (PUE):

$$\frac{\textbf{Total Building Power}}{\textbf{IT equipment Power}}$$

  - Power efficiency measure for WSC, *not* including efficiency of servers, networking gear
  - Power usage for non-IT equipment increases PUE
  - 1.0 is perfection, higher numbers are worse
  - Google WSC's PUE: 1.2

PUE in the Wild (2007)

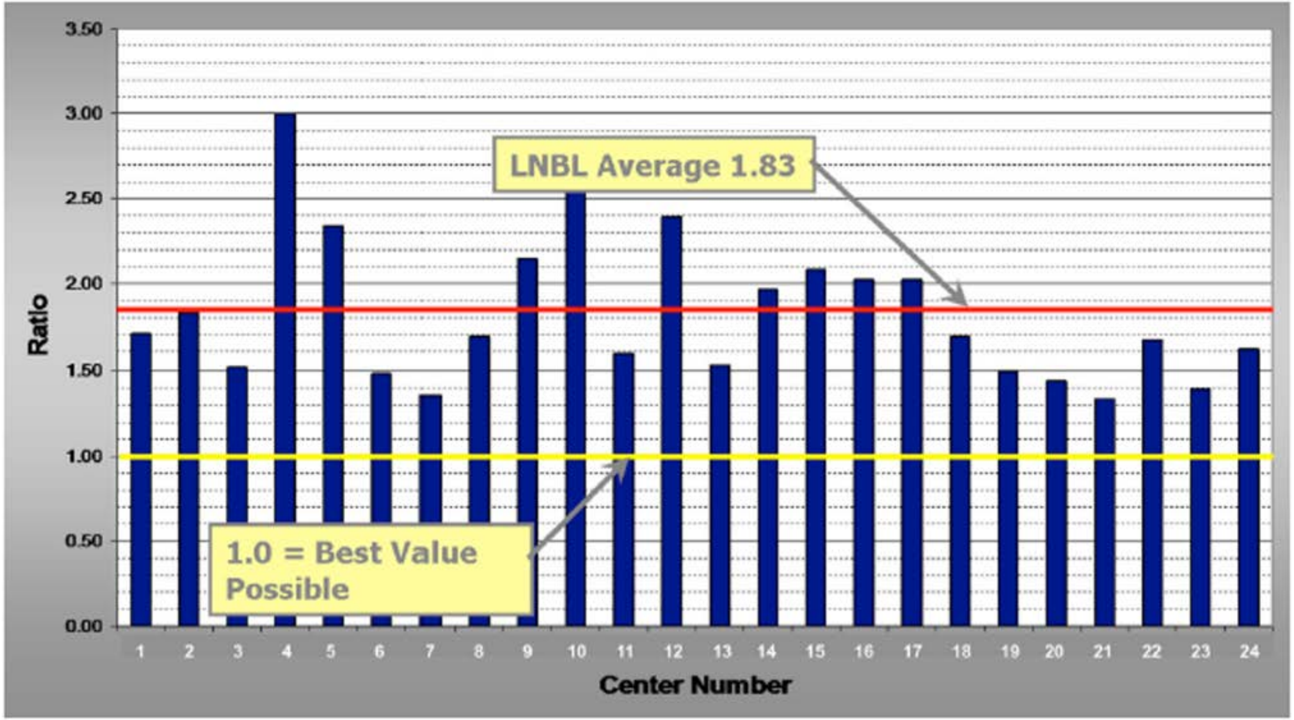
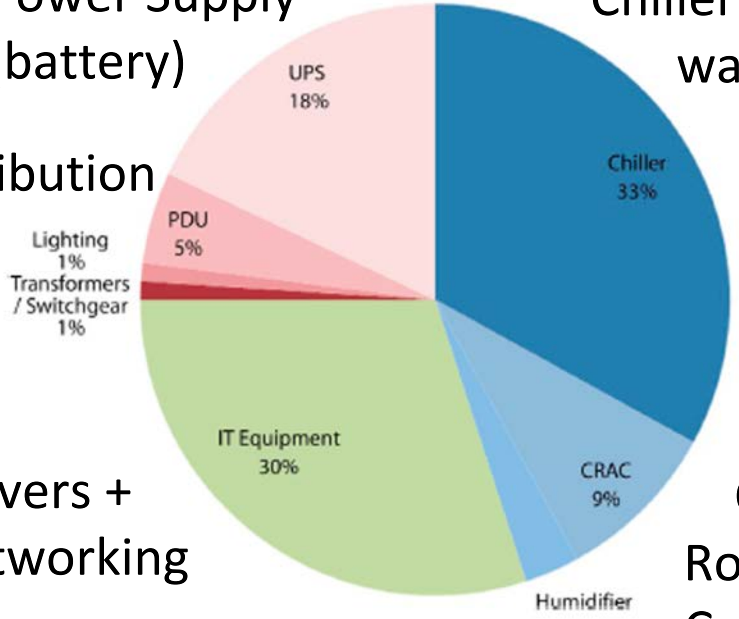


FIGURE 5.1: LBNL survey of the power usage efficiency of 24 datacenters, 2007 (Greenberg et al.)

High PUE: Where Does Power Go?

Uninterruptable Power Supply (battery)

Power Distribution Unit



Chiller cools warm water from Air Conditioner

Servers + Networking

Computer Room Air Conditioner

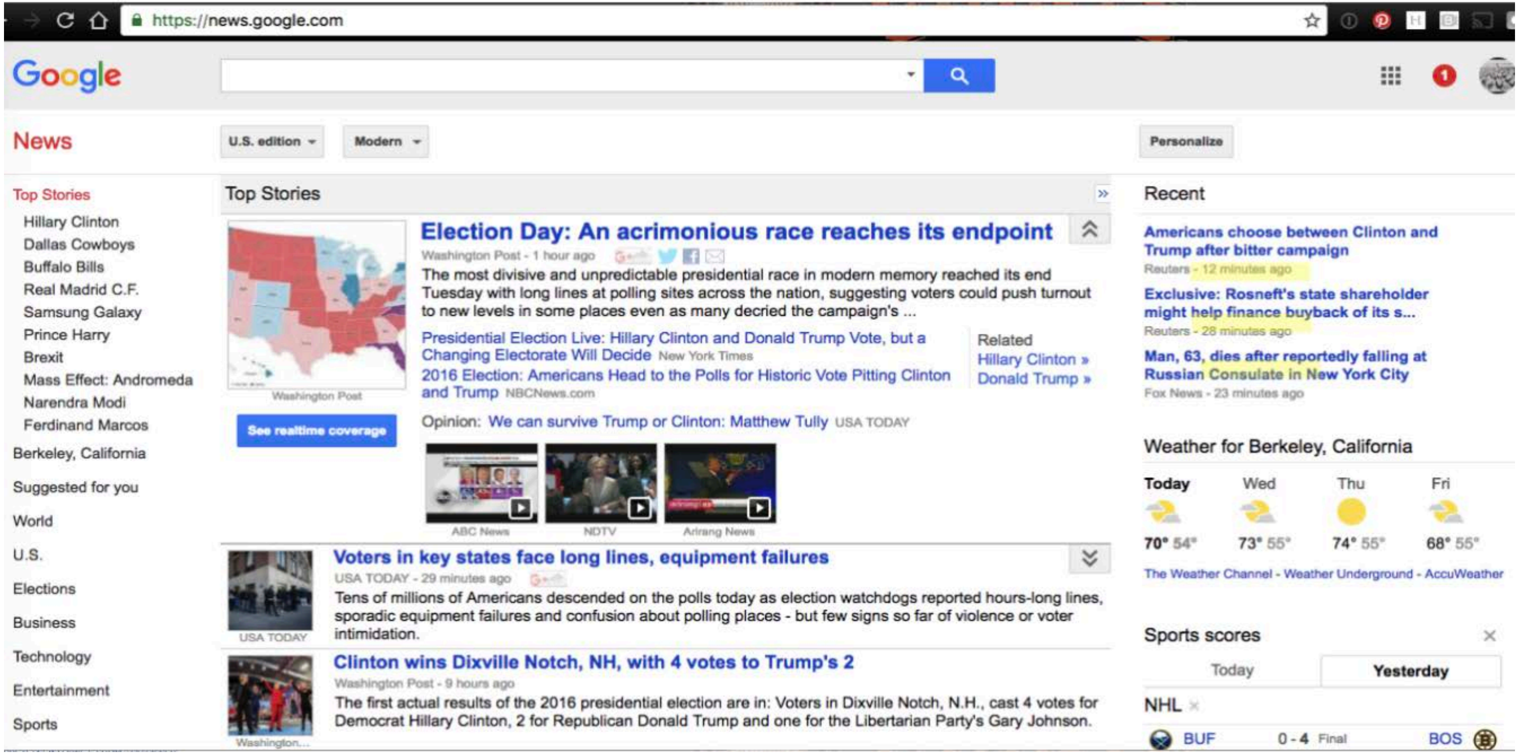
Google WSC A PUE: 1.24

- **Careful air flow handling**
  - Don’t mix server hot air exhaust with cold air (separate warm aisle from cold aisle)
  - Short path to cooling so little energy spent moving cold or hot air long distances
  - Keeping servers inside containers helps control air flow
- **Elevated cold aisle temperatures**
  - 81°F instead of traditional 65°- 68°F
  - Found reliability OK if run servers hotter
- **Use of free cooling**
  - Cool warm water outside by evaporation in cooling towers
  - Locate WSC in moderate climate so not too hot or too cold
- **Per-server 12-V DC UPS**
  - Rather than WSC wide UPS, place single battery per server board
  - Increases WSC efficiency from 90% to 99%
- **Measure vs. estimate PUE, publish PUE, and improve operation**

Agenda

- Warehouse Scale Computers
- Cloud Computing
- Request Level Parallelism
- MapReduce

Scaled Communities, Processing, and Data

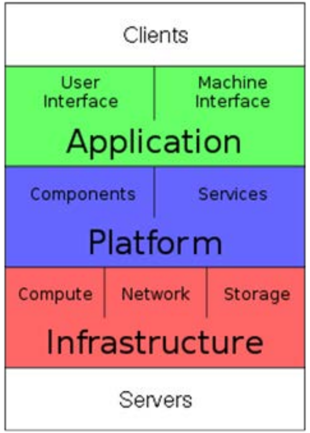


Cloud Distinguished by...

- Shared platform with illusion of isolation
  - Collocation with other tenants
  - Exploits technology of VMs and hypervisors
  - At best “fair” allocation of resources, but not true isolation
- Attraction of low-cost cycles
  - Economies of scale driving move to consolidation
  - Statistical multiplexing to achieve high utilization/efficiency of resources
- Elastic service
  - Pay for what you need, get more when you need it
  - But no performance guarantees: assumes uncorrelated demand for resources

Cloud Services

- SaaS: deliver apps over Internet, eliminating need to install/run on customer's computers, simplifying maintenance and support
  - E.g., Google Docs, Win Apps in the Cloud
- PaaS: deliver computing "stack" as a service, using cloud infrastructure to implement apps. Deploy apps without cost/complexity of buying and managing underlying layers
  - E.g., Hadoop on EC2, Apache Spark on GCP
- IaaS: Rather than purchasing servers, software, data center space or net equipment, clients buy resources as an outsourced service. Billed on utility basis. Amount of resources consumed/cost reflect level of activity
  - E.g., Amazon Elastic Compute Cloud, Google Compute Platform



Cloud Computing Stack

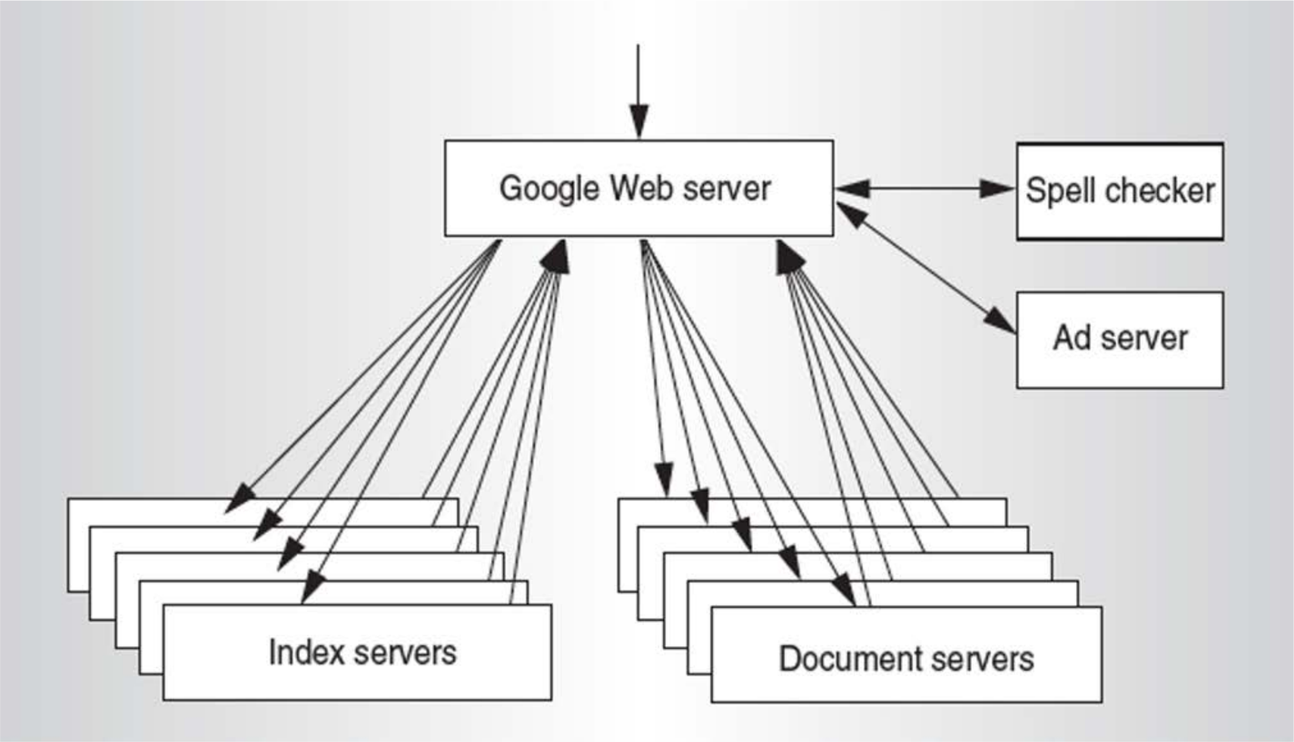
Agenda

- Warehouse Scale Computers
- Cloud Computing
- Request Level Parallelism
- MapReduce
- Spark

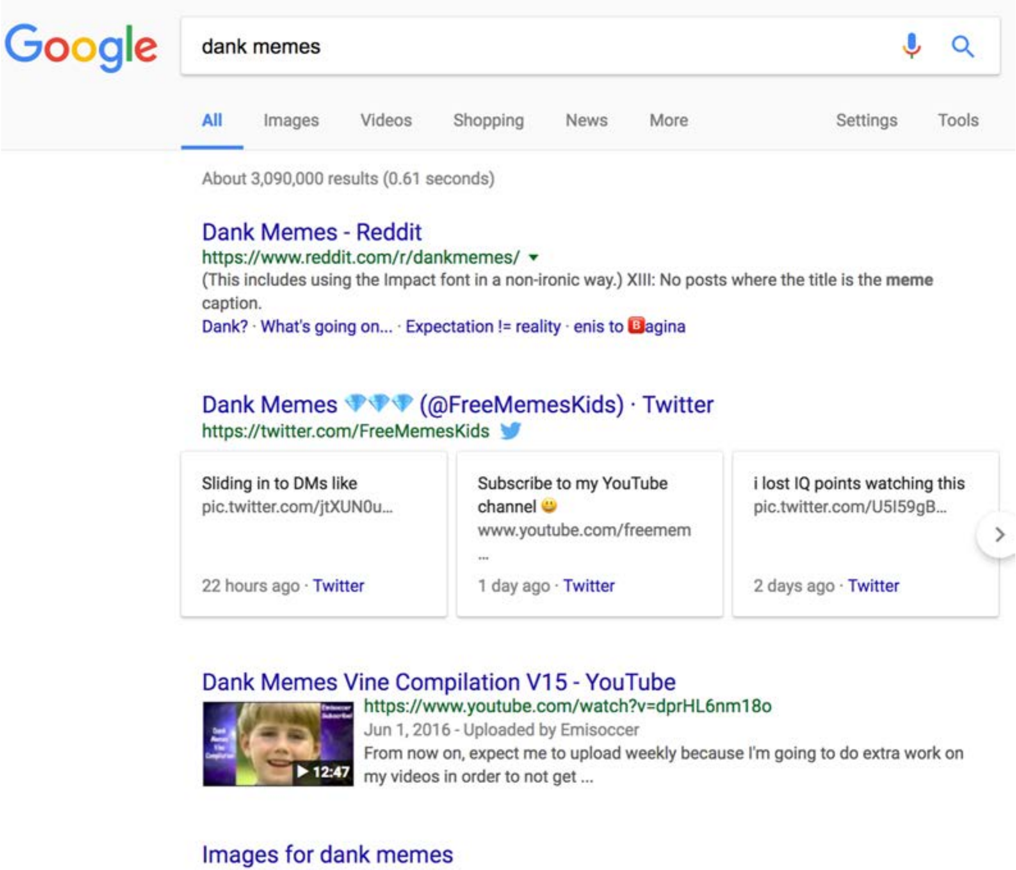
Request-Level Parallelism (RLP)

- Hundreds or thousands of requests per sec
  - Not your laptop or cell-phone, but popular Internet services like web search, social networking, …
  - Such requests are largely independent
    - Often involve read-mostly databases
    - Rarely involve strict read–write data sharing or synchronization across requests
- Computation easily partitioned within a request and across different requests

Google Query-Serving Architecture



Anatomy of a Web Search



Anatomy of a Web Search (1 of 3)

- Google “dank memes”
  - Direct request to “closest” Google Warehouse Scale Computer
  - Front-end load balancer directs request to one of many arrays (cluster of servers) within WSC
  - Within array, select one of many Google Web Servers (GWS) to handle the request and compose the response pages
  - GWS communicates with Index Servers to find documents that contain the search words, “dank”, “memes”, may use location of search as well
  - Return document list with associated relevance score

Anatomy of a Web Search (2 of 3)

- In parallel,
  - Ad system: run ad auction for bidders on search terms
  - Get images of dank memes and trash posts
- Use docids (document IDs) to access indexed documents
- Compose the page
  - Result document extracts (with keyword in context) ordered by relevance score
  - Sponsored links (along the top) and advertisements (along the sides)

Anatomy of a Web Search (3 of 3)

- Implementation strategy
  - Randomly distribute the entries
  - Make many copies of data (a.k.a. “replicas”)
  - Load balance requests across replicas
- Redundant copies of indices and documents
  - Breaks up hot spots, e.g. “UCBMFET”
  - Increases opportunities for request-level parallelism
  - Makes the system more tolerant of failures

Agenda

- Warehouse Scale Computers
- Cloud Computing
- Request Level Parallelism
- MapReduce

Great Idea #4: Parallelism

Software

- Parallel Requests
  Assigned to computer
  e.g. search “Steven Ho”
- Parallel Threads
  Assigned to core
  e.g. lookup, ads
- Parallel Instructions
  > 1 instruction @ one time
  e.g. 5 pipelined instructions
- Parallel Data
  > 1 data item @ one time
  e.g. add of 4 pairs of words
- Hardware descriptions
  All gates functioning in
  parallel at same time

Leverage Parallelism & Achieve High Performance

Hardware

Warehouse Scale Computer

Smart Phone

Computer

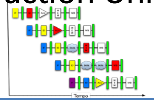
Core ... Core

Memory

Input/Output

Core

Instruction Unit(s)

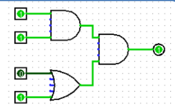


Functional Unit(s)

A_0+B_0 | A_1+B_1 | A_2+B_2 | A_3+B_3

Cache Memory

Logic Gates



Data Level Parallelism (DLP)

- SIMD
  - Supports data-level parallelism in a single machine
  - Additional instructions & hardware
  - e.g. Matrix multiplication in memory
- DLP on WSC
  - Supports data-level parallelism across multiple machines
  - MapReduce & scalable file systems
  - e.g. Training CNNs with images across multiple disks

MapReduce

- Simple data-parallel programming model and implementation for processing large dataset
- Users specify the computation in terms of
  - a ***map*** function, and
  - a ***reduce*** function
- Underlying runtime system
  - Automatically parallelize the computation across large scale clusters of machines.
  - Handles machine failure
  - Schedule inter-machine communication to make efficient use of the networks
- Invented at Google

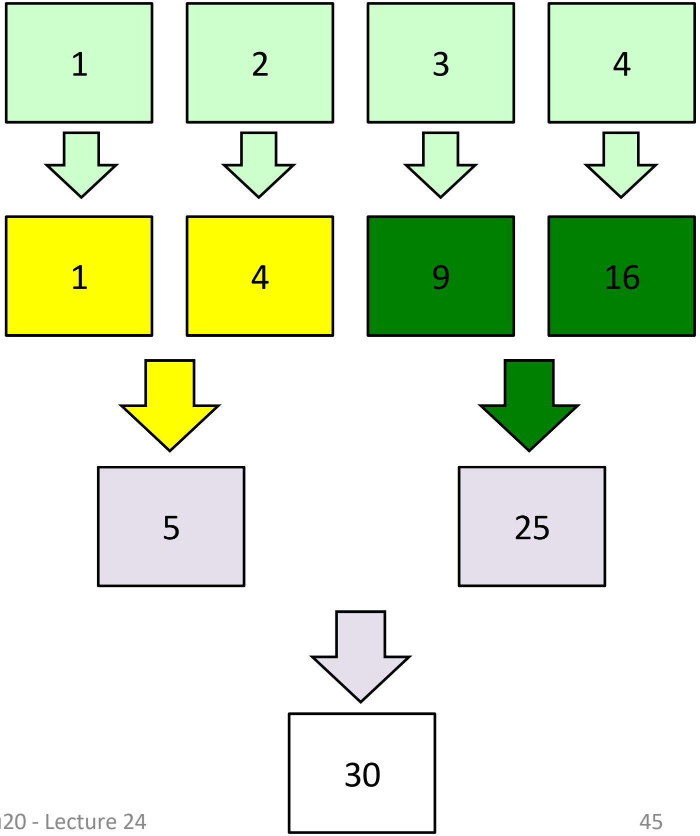
MapReduce Uses

- At Google:
  - Index construction for Google Search
  - Article clustering for Google News
  - Statistical machine translation
  - For computing multi-layers street maps
- At Yahoo!:
  - "Web map" powering Yahoo! Search
  - Spam detection for Yahoo! Mail
- At Facebook:
  - Data mining
  - Ad optimization
  - Spam detection

Map & Reduce Functions in Python

- Calculate : $\sum_{n=1}^{4} n^2$

```
list = [1, 2, 3, 4]
def square(x):
  return x * x
def sum(x, y):
  return x + y
reduce(sum,
  map(square, list))
```



MapReduce Programming Model

- ***Map***: `(in_key, in_value) → list(interm_key, interm_val)`

```
map(in_key, in_val):
    // DO WORK HERE
    emit(interm_key,interm_val)
```

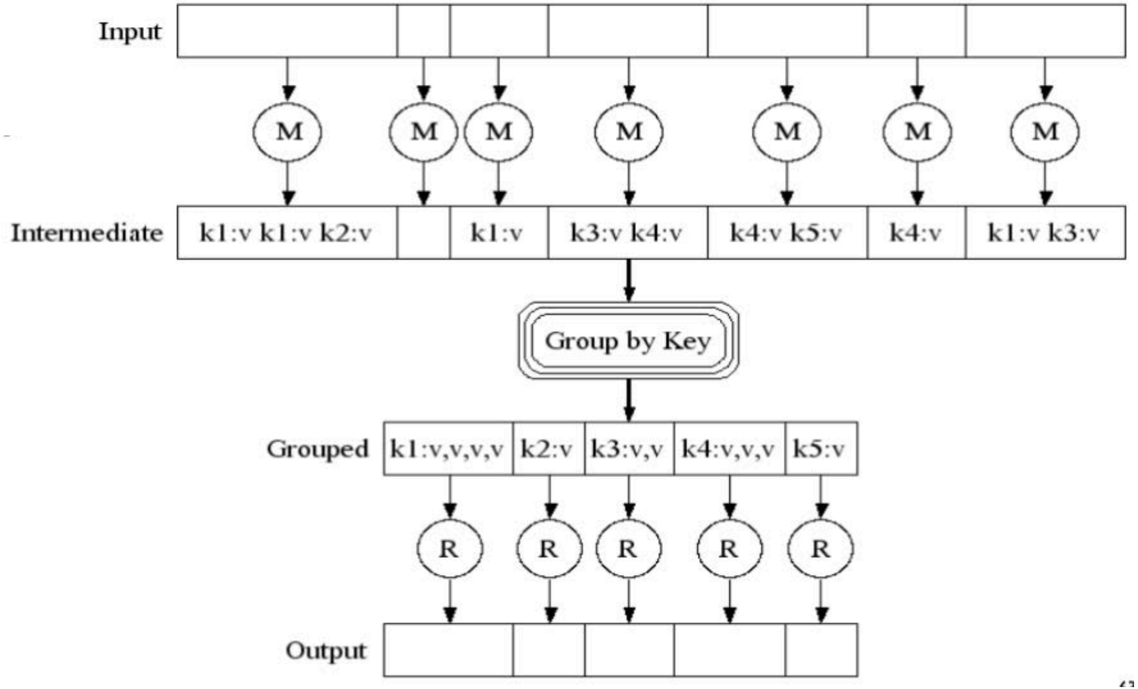
  - Slice data into “shards” or “splits” and distribute to workers
  - Compute set of intermediate key/value pairs

- ***Reduce***: `(interm_key, list(interm_value)) → list(out_value)`

```
reduce(interm_key, list(interm_val)):
    // DO WORK HERE
    emit(out_key, out_val)
```

  - Combines all intermediate values for a particular key
  - Produces a set of merged output values (usually just one)

MapReduce Execution



MapReduce Word Count Example

Distribute

that that is	is that that	is not is not	is that it it is
Map	**Map**	**Map**	**Map**
that 1, that 1, is 1	is 1, that 1, that 1	is 1, not 1, is 1, not 1,	is 1, that 1, it 1, it 1, is 1

Shuffle

Group by key

that 1 1 1 1 1	is 1 1 1 1 1 1	it 1 1	not 1 1
Reduce	**Reduce**	**Reduce**	**Reduce**
that 5	is 6	it 2	not 2

Collect

that 5; is 6; it 2; not 2

MapReduce Word Count Example

- ***Map*** phase: (doc name, doc contents) → list(word, count)

```
// "I do I learn" → [("I",1),("do",1),("I",1),("learn",1)]
```

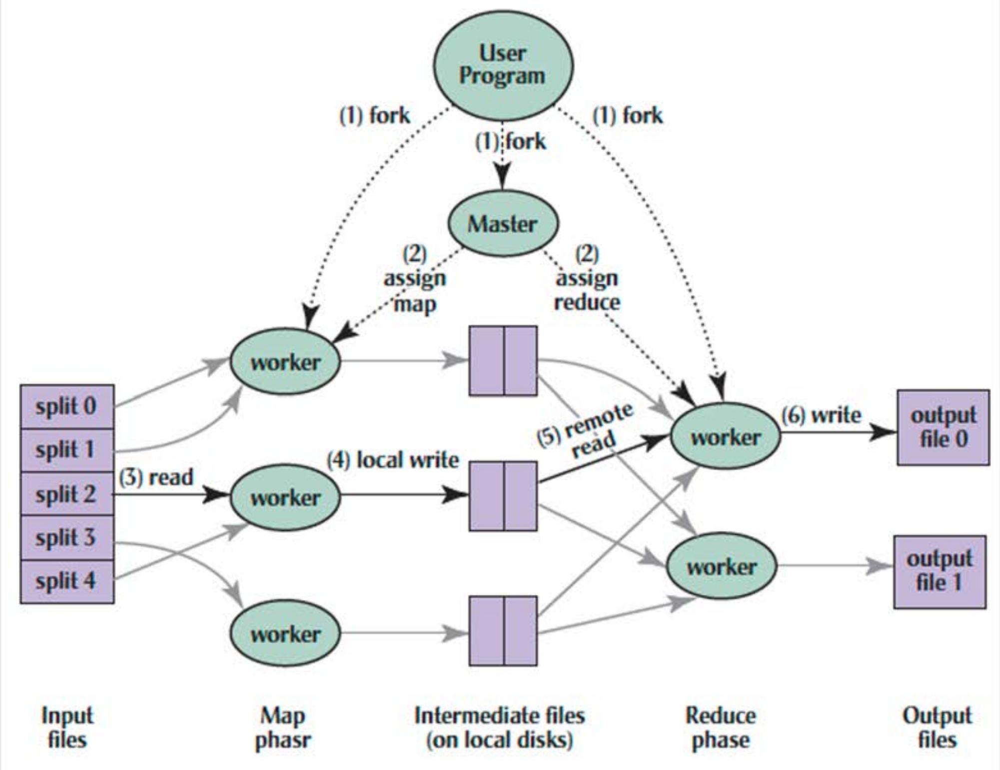
```
map(key, value):
  for each word w in value:
    emit(w, 1)
```

- ***Reduce*** phase: (word, list(count)) → (word, count_sum)

```
// ("I", [1,1]) → ("I",2)
```

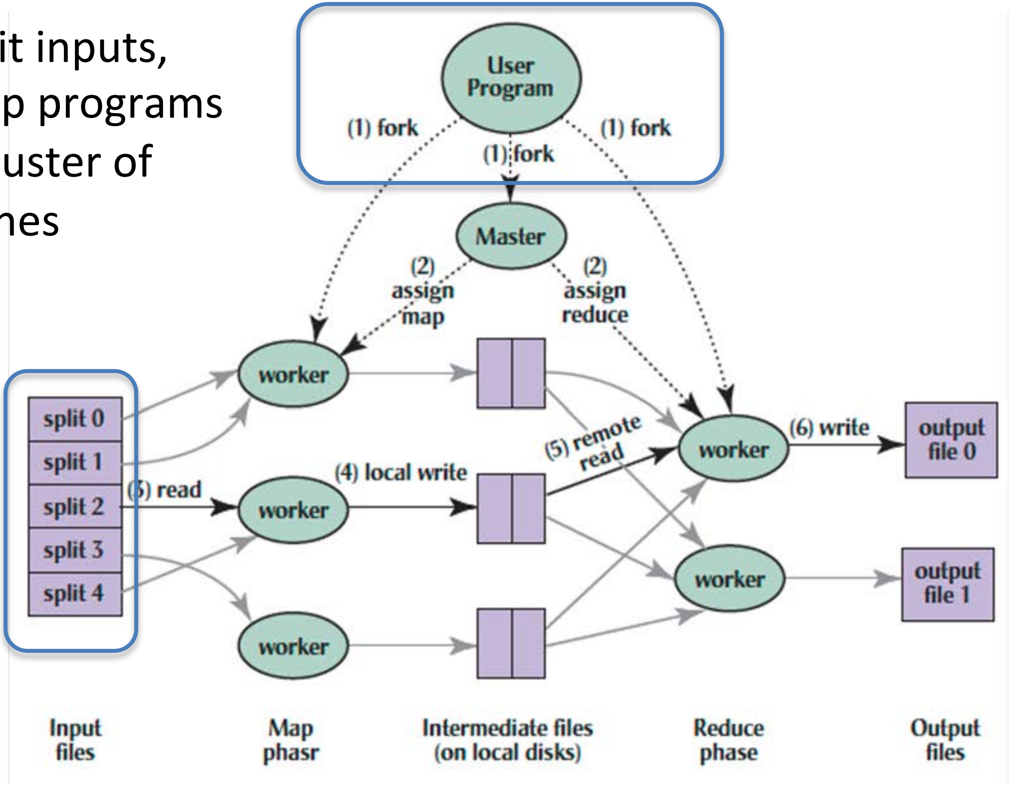
```
reduce(key, values):
  result = 0
  for each v in values:
    result += v
  emit(key, result)
```

MapReduce Implementation



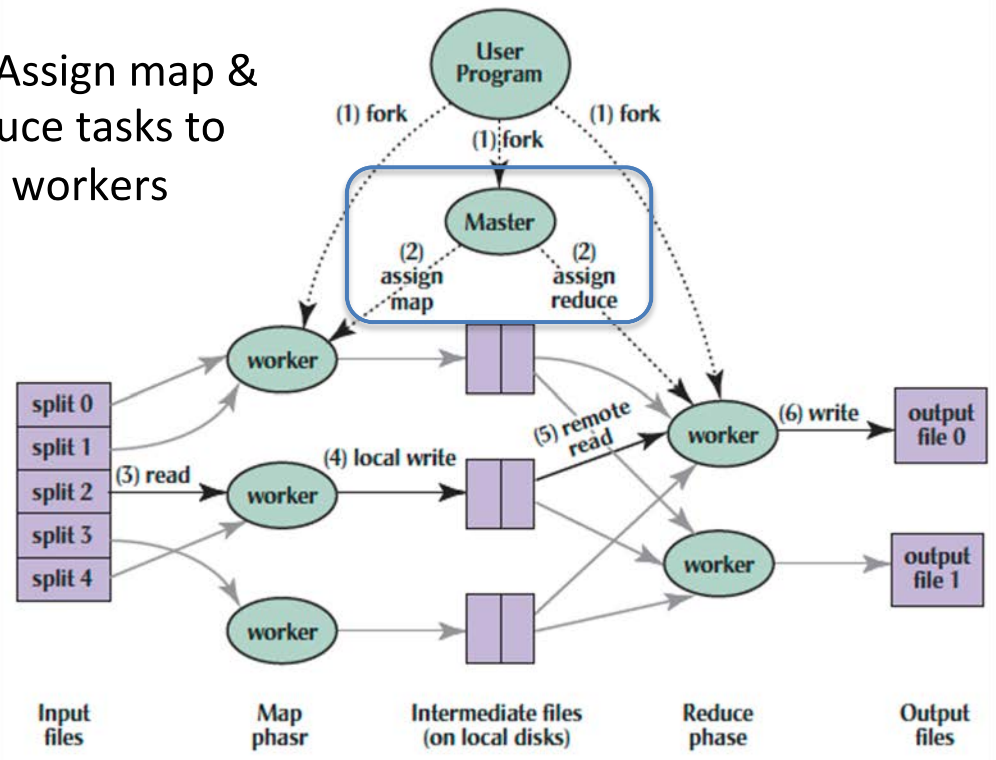
MapReduce Execution

(1) Split inputs, start up programs on a cluster of machines



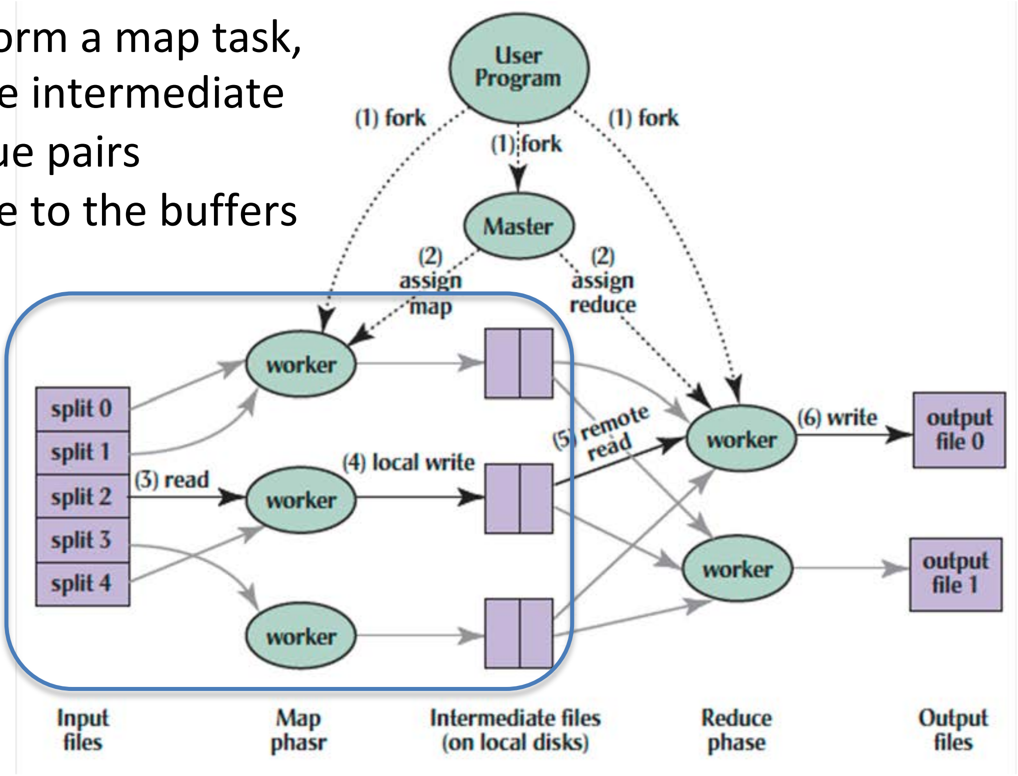
MapReduce Execution

(2) Assign map & reduce tasks to idle workers

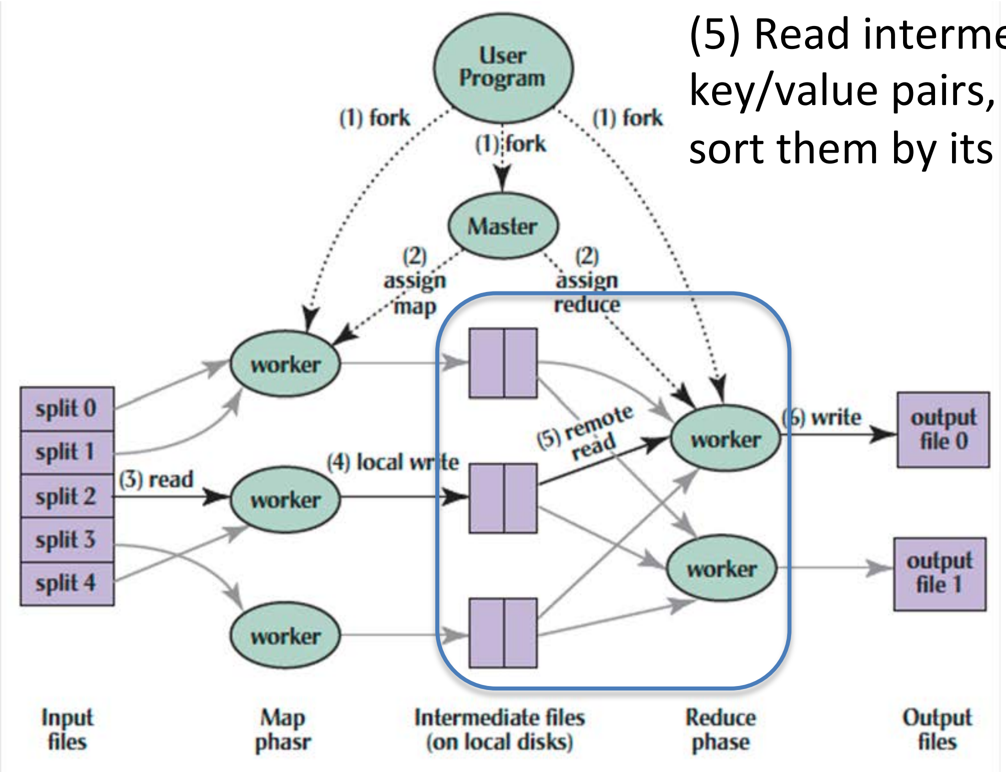


MapReduce Execution

(3) Perform a map task, generate intermediate key/value pairs
(4) Write to the buffers

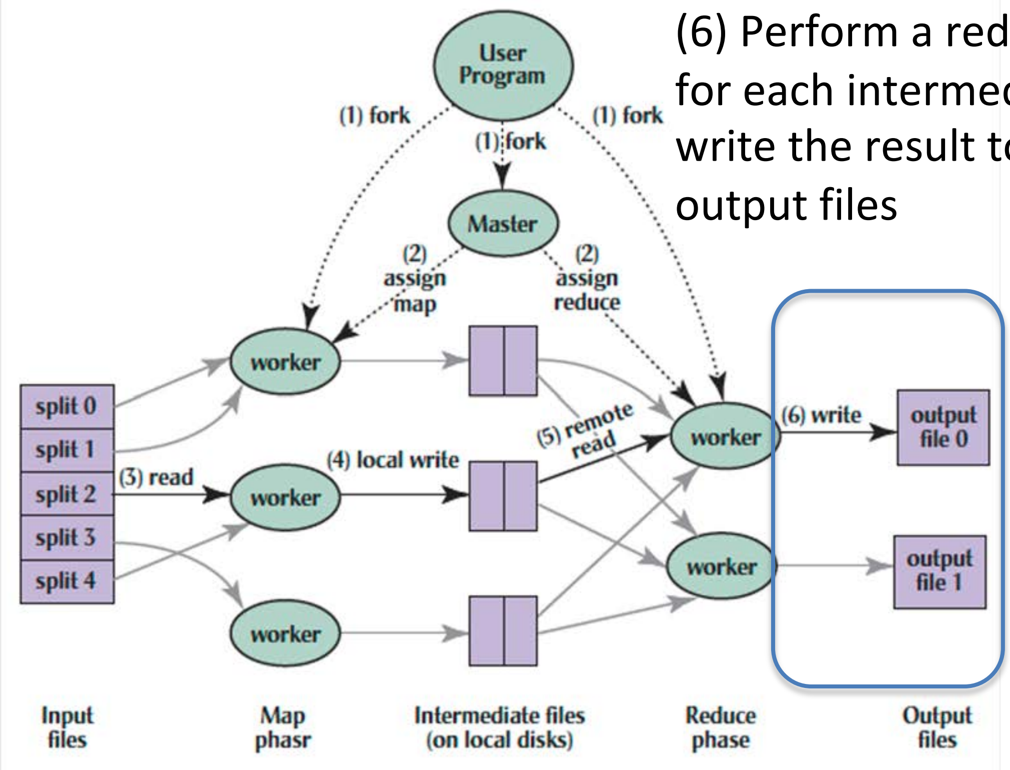


MapReduce Execution



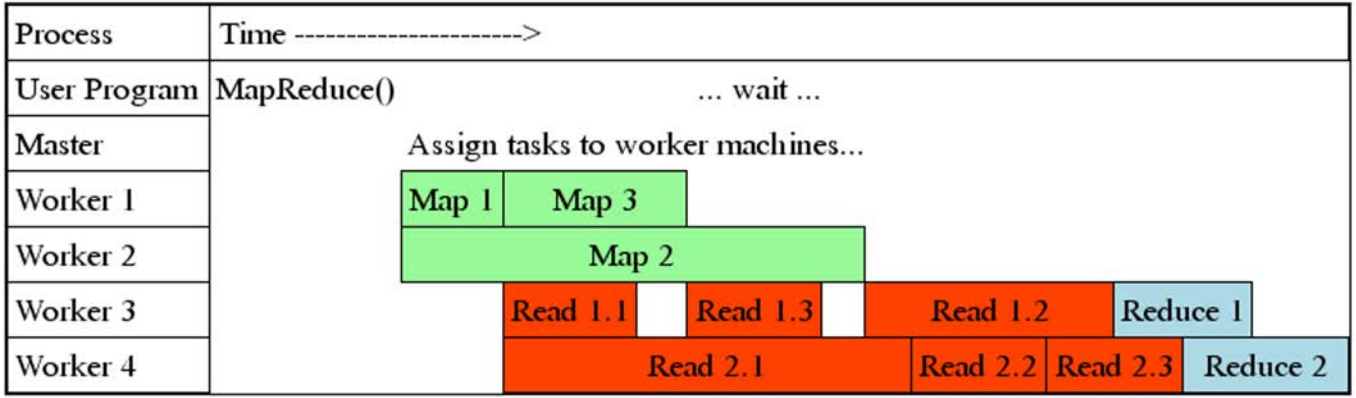
(5) Read intermediate key/value pairs,
sort them by its key.

MapReduce Execution



(6) Perform a reduce task for each intermediate key, write the result to the output files

MapReduce Processing Time Line



- Master assigns map + reduce tasks to “worker” servers
- As soon as a map task finishes, worker server can be assigned a new map or reduce task
- Data sort begins as soon as a given Map finishes
- Reduce task begins as soon as all data sort finish
- To tolerate faults, reassign task if a worker server “dies”

Big Data MapReduce Engine: Spark

- Fast and general engine for large-scale data processing.
- Originally developed in the AMPlab at UC Berkeley
- Running on Hadoop Distributed File System
- Provides Java, Scala, Python APIs for
  - Database
  - Machine learning
  - Graph algorithms
- MUCH faster and easier to use compared to predecessor Hadoop

Word Count in Spark's Python API

```
// RDD: primary abstraction of a distributed
collection of items
file = sc.textFile("hdfs://…")
// Two kinds of operations:
// Actions: RDD → Value
// Transformations: RDD → RDD
// e.g. flatMap, Map, reduceByKey
file.flatMap(lambda line: line.split())
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a + b)
```

MapReduce Word Count Example

- ***Map*** phase: (doc name, doc contents) → list(word, count)

```
// "I do I learn"  → ["I", "do", "I", "learn"]
map(key, value):
  for each word w in value:
    emit(w, 1)      → [("I",1), ("do",1), ("I",1), ("learn",1)]
```

- ***Reduce*** phase: (word, list(count)) → (word, count_sum)

```
// ("I", [1,1]) → ("I",2)
reduce(key, values):
  result = 0
  for each v in values:
    result += v
  emit(key, result)
```

Word Count in Spark's Python API

```
// RDD: primary abstraction of a distributed
collection of items
file = sc.textFile("hdfs://…")
// Two kinds of operations:
// Actions: RDD → Value
// Transformations: RDD → RDD
// e.g. flatMap, Map, reduceByKey
file.flatMap(lambda line: line.split())
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a + b)
```

MapReduce Word Count Example

- ***Map*** phase: (doc name, doc contents) → list(word, count)

```
// "I do I learn"
map(key, value):
  for each word w in value:
    emit(w, 1)
```

```
→ ["I", "do", "I", "learn"]

→ [("I",1 ), ("do",1),
   ("I",1), ("learn",1)]
```

- ***Reduce*** phase: (word, list(count)) → (word, count_sum)

```
// ("I", [1,1]) → ("I",2)
reduce(key, values):
  result = 0
  for each v in values:
    result += v
  emit(key, result)
```

Word Count in Spark's Python API

```
// RDD: primary abstraction of a distributed
collection of items
file = sc.textFile(“hdfs://…”)
// Two kinds of operations:
// Actions: RDD → Value
// Transformations: RDD → RDD
// e.g. flatMap, Map, reduceByKey
file.flatMap(lambda line: line.split())
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a + b)
```

MapReduce Word Count Example

- ***Map*** phase: (doc name, doc contents) → list(word, count)

```
// "I do I learn"" → [("I",1),("do",1),("I",1),("learn",1)]
map(key, value):
  for each word w in value:
    emit(w, 1)
```

- ***Reduce*** phase: (word, list(count)) → (word, count_sum)

```
// ("I", [1,1]) → ("I",2)
reduce(key, values):
  result = 0
  for each v in values:
    result += v
  emit(key, result)
```

a b

Summary

- Warehouse Scale Computers
  - Supports many of the applications we have come to depend on
  - Software must cope with failure, load variation, and latency/bandwidth limitations
  - Hardware sensitive to cost and energy efficiency
- Request Level Parallelism
  - High request volume, each largely independent
  - Replication for better throughput, availability
- MapReduce
  - Convenient data-level parallelism on large dataset across large number of machines
  - Spark is a framework for executing MapReduce algorithms