CS 110
Computer Architecture
Warehouse-Scale Computing, MapReduce, and Spark

Instructors:
Sören Schwertfeger and Chundong Wang

https://robotics.shanghaitech.edu.cn/courses/ca/21s

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkeley's CS61C

Agenda

- Warehouse Scale Computing
- Request-level Parallelism

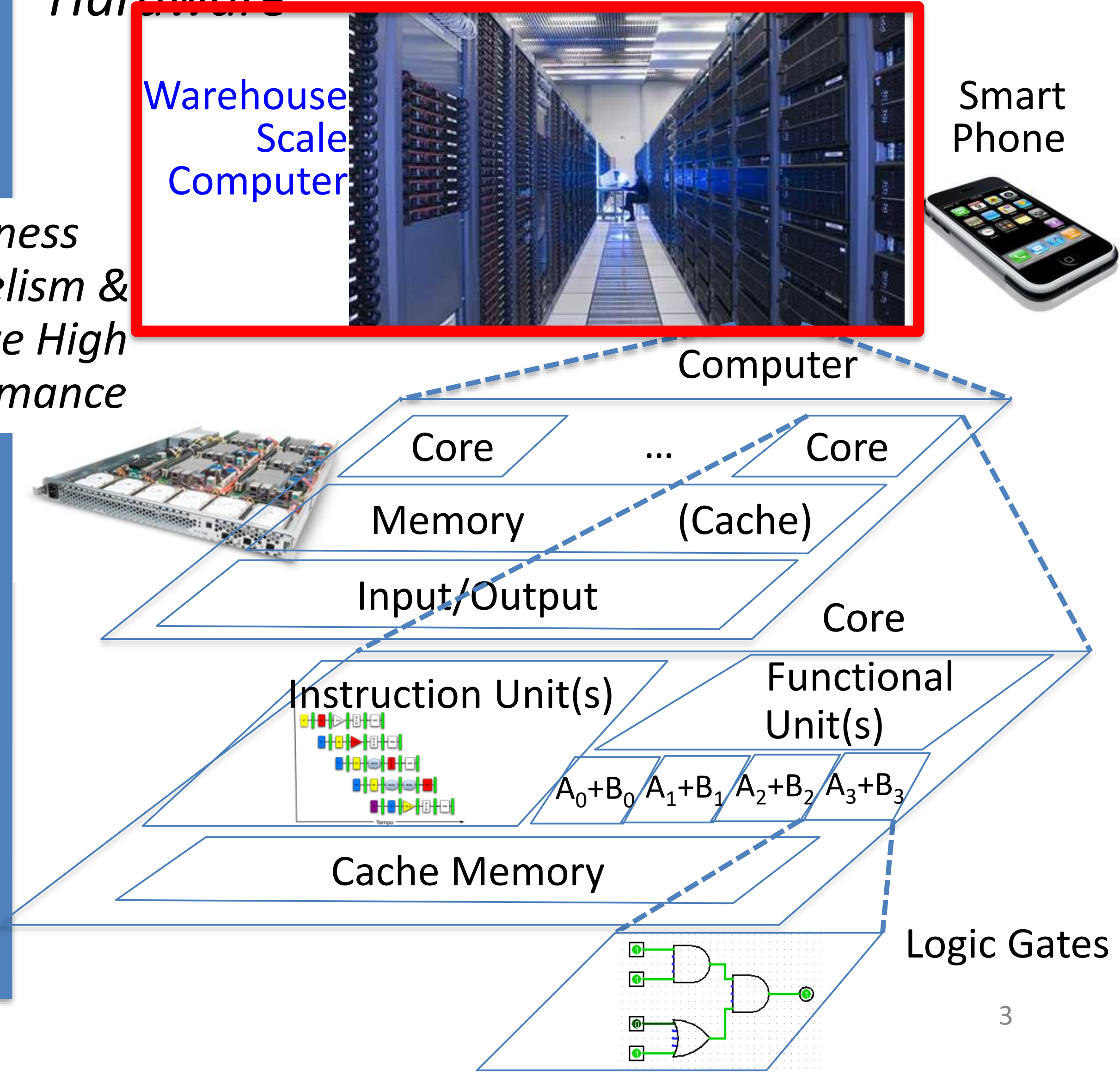
  e.g. Web search
- Data-level Parallelism
  - MapReduce
  - Hadoop, Spark

New-School Machine Structures

Software | *Hardware*

- Parallel Requests
  Assigned to computer
  e.g., Search "F9"
- 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., Deep Learning for image classification
- Hardware descriptions
  All gates @ one time
- Programming Languages

Harness Parallelism & Achieve High Performance



Google's WSCs

Containers in WSCs

Inside WSC

Inside Container

Server, Rack, Array

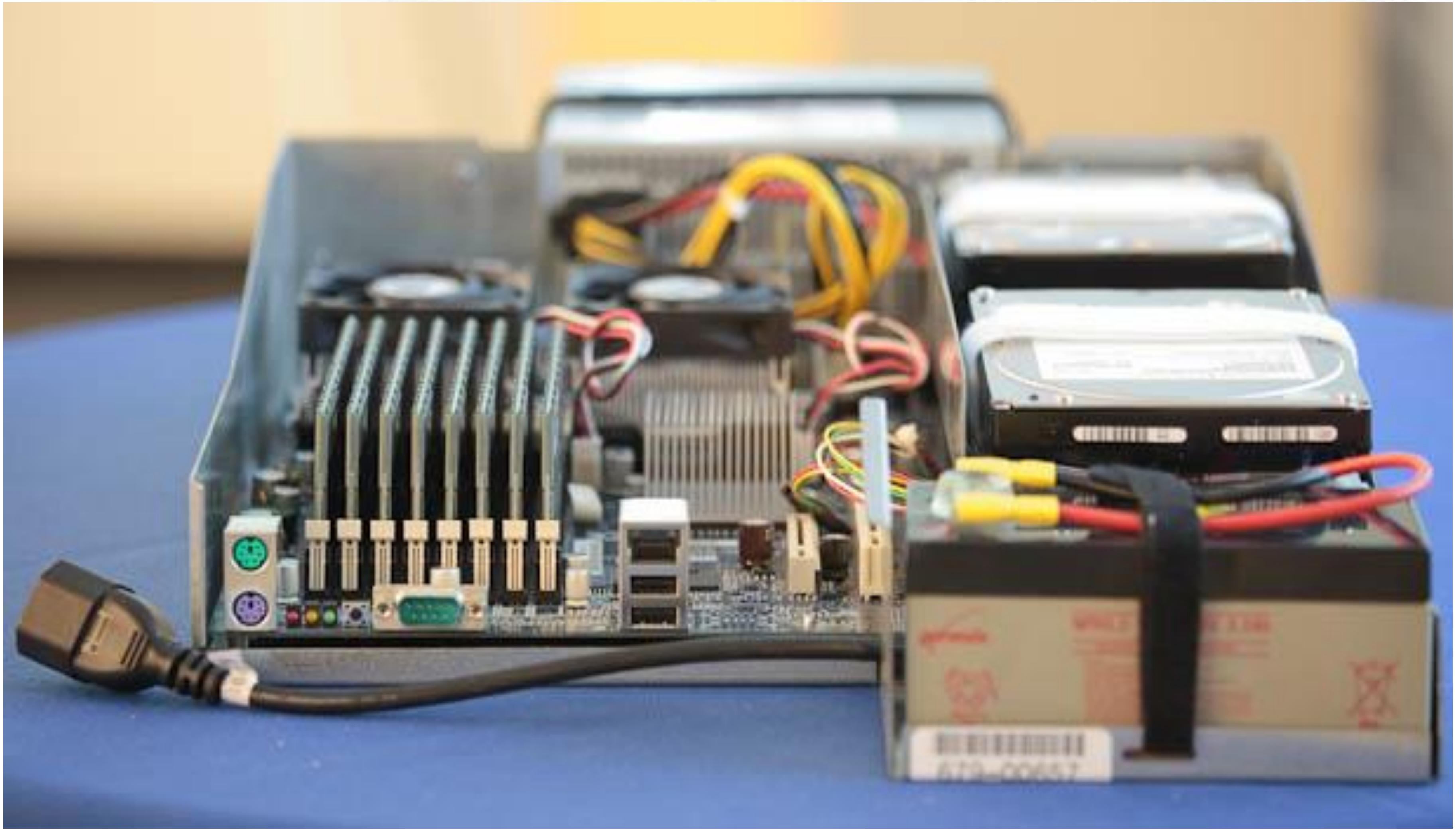
A Giant Computer

- **Sunway TaihuLight**

系统峰值性能	125.436PFlops
实测持续运算性能	93.015PFlops
处理器型号	"申威26010" 众核处理器
整机处理器个数	40960个
实整机处理器核数	10649600个
系统总内存	1310720 GB
操作系统	Raise Linux
编程语言	C、C++、Fortran
并行语言及环境	MPI、OpenMP、OpenACC等
SSD存储	230TB
在线存储	10PB，带宽288GB/s
近线存储	10PB，带宽32GB/s

http://www.nsccwx.cn/swsource/5d2fe23624364f0351459262

Google Server Internals



Open Compute Project

- Share designs of data center products
  - Facebook, Intel, Nokia, Google, Apple, Microsoft, Seagate Technology, Dell, Cisco, Goldman Sachs, Lenovo, …
- Design and enable the delivery of the most efficient server, storage and data center hardware designs for scalable computing.
- Openly sharing ideas, specifications and other intellectual property is the key to maximizing innovation and reducing operational complexity
- All Facebook Data Centers are 100% OCP

Warehouse-Scale Computers

- Datacenter
  - Collection of 10,000 to 100,000 servers
  - Networks connecting them together
- ***Single gigantic*** machine
- Very large applications (Internet service): search, email, video sharing, social networking
- Very high availability
- "…WSCs are no less worthy of the expertise of computer systems architects than any other class of machines" Barroso and Hoelzle, 2009

Unique to WSCs

- Ample Parallelism
  - **Request-level Parallelism:** e.g., web search
  - **Data-level Parallelism:** e.g., image classifier training
- Scale and its Opportunities/Problems
  - **Scale of economy**: low per-unit cost
  - Cloud computing: rent computing power with low costs (e.g., AWS)
  - **High # of failures**

    e.g.: 4 disks/server, annual failure rate: 4%
    → WSC of 50,000 servers: 1 disk fail/hour

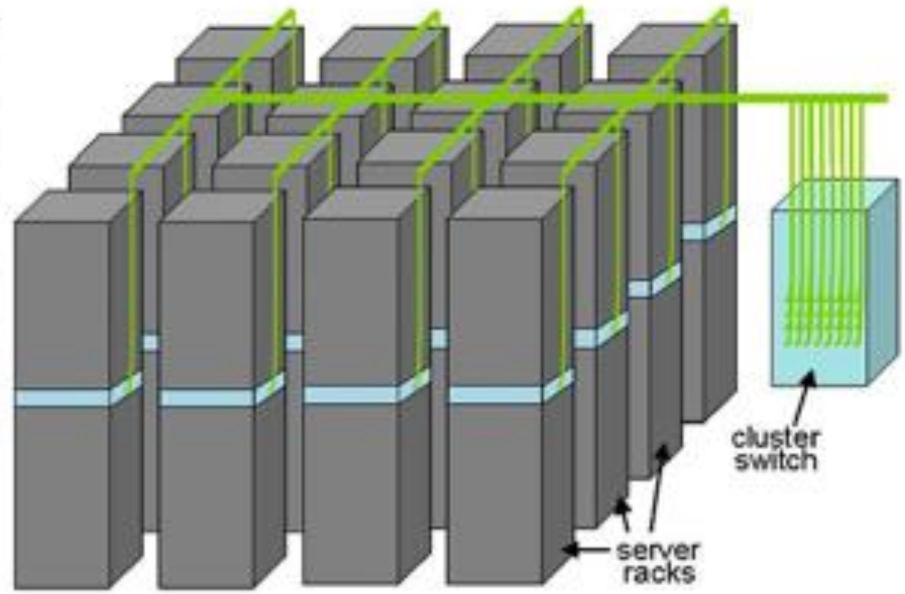
$$\frac{50000 \times 4 \times 4\%}{365 \times 24} \approx 0.913$$

- Operation Cost Count
  - Longer life time (>10 years)
  - **Cost of equipment purchases << cost of ownership**

WSC Architecture

1U Server:
8 cores,
16 GB DRAM,
4x1 TB disk

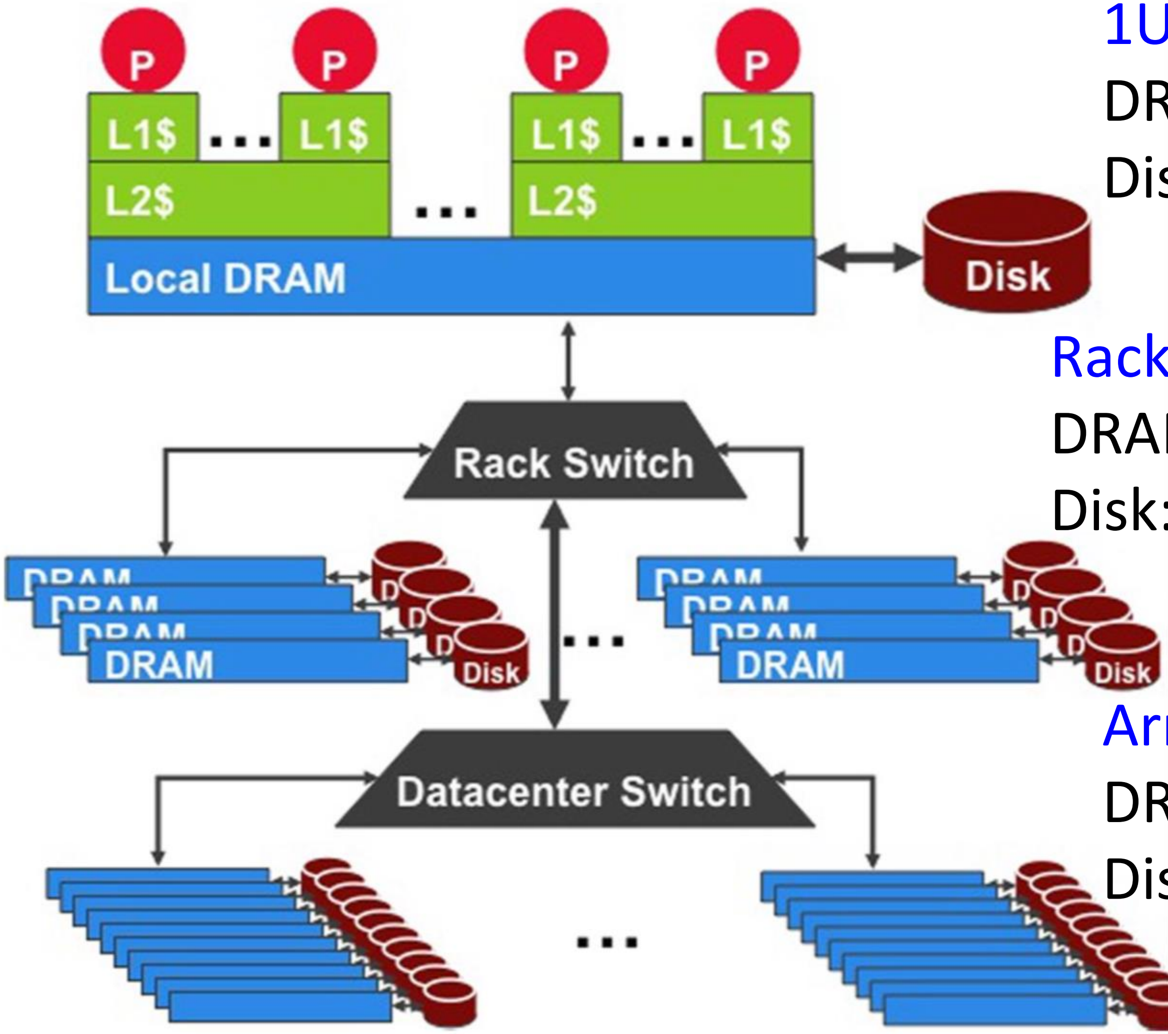
Rack:
40-80 severs,
Local Ethernet (1-10Gbps) switch



Array (aka cluster):
16-32 racks
Expensive switch
(10X bandwidth → 100x cost)

WSC Storage Hierarchy

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, 0.1us, 20GB/s

Disk: 2TB, 10^4us, 200MB/s

Rack(80 severs):

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

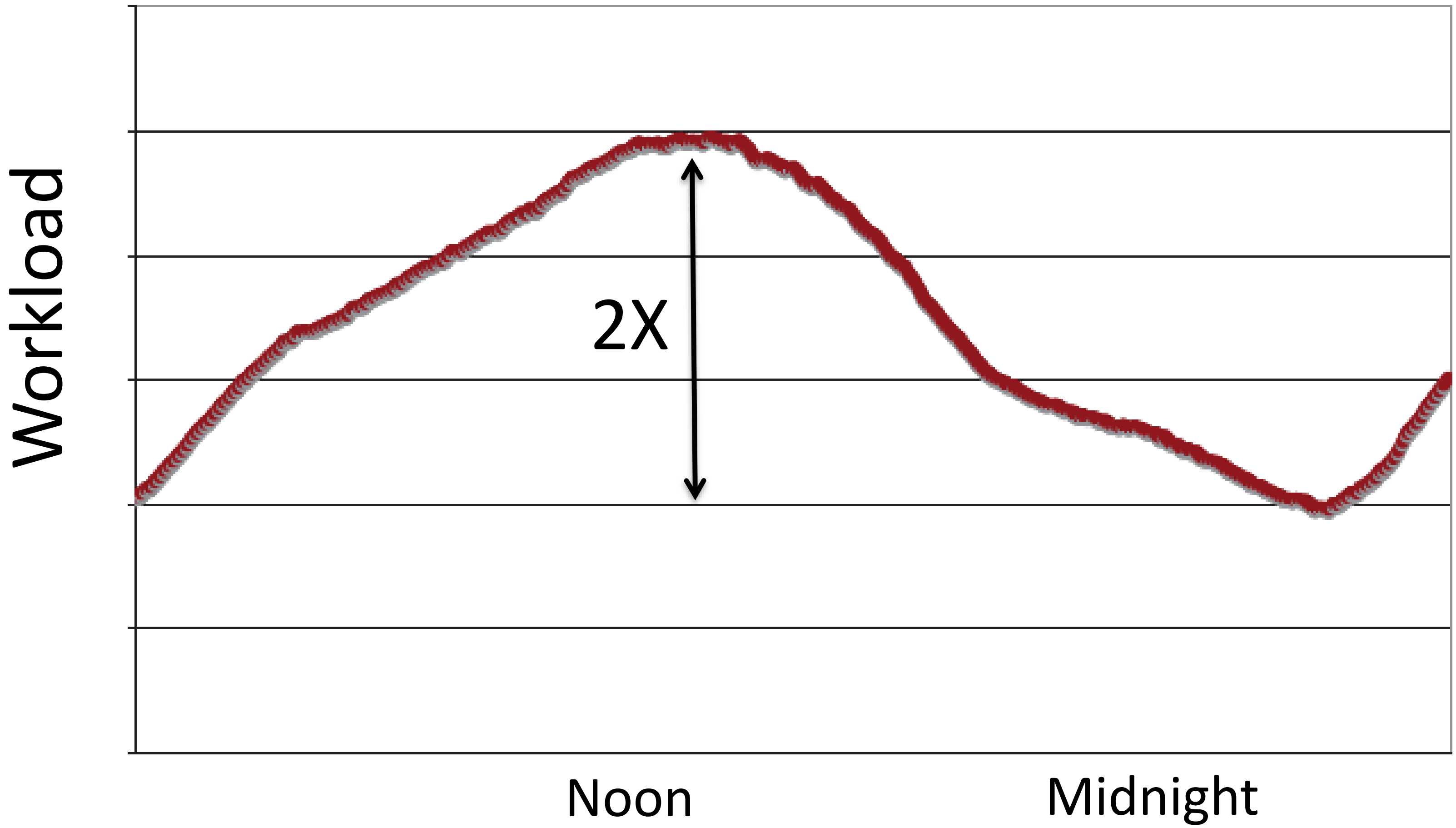
Disk: 160TB, 11ms, 100MB/s

Array(30 racks):

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

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

Workload Variation



- Online service: Peak usage 2X off-peak

Impact on WSC software

- ***Latency, bandwidth*** → Performance
  - Independent data set within an array
  - Locality of access within server or rack
- ***High failure rate*** → Reliability, Availability
  - Preventing failures is expensive
  - Cope with failures gracefully
- ***Varying workloads*** → Scalability, Availability
  - Scale up and down gracefully
- More challenging than software for single computers!

Power Usage Effectiveness

- Energy efficiency
  - Primary concern in the design of WSC
  - Important component of the total cost of ownership
- Power Usage Effectiveness (PUE):

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

  - A power efficiency measure for WSC
  - Not considering efficiency of servers, networking
  - Perfection = 1.0
  - Google WSC's PUE = 1.2

PUE in the Wild (2007)

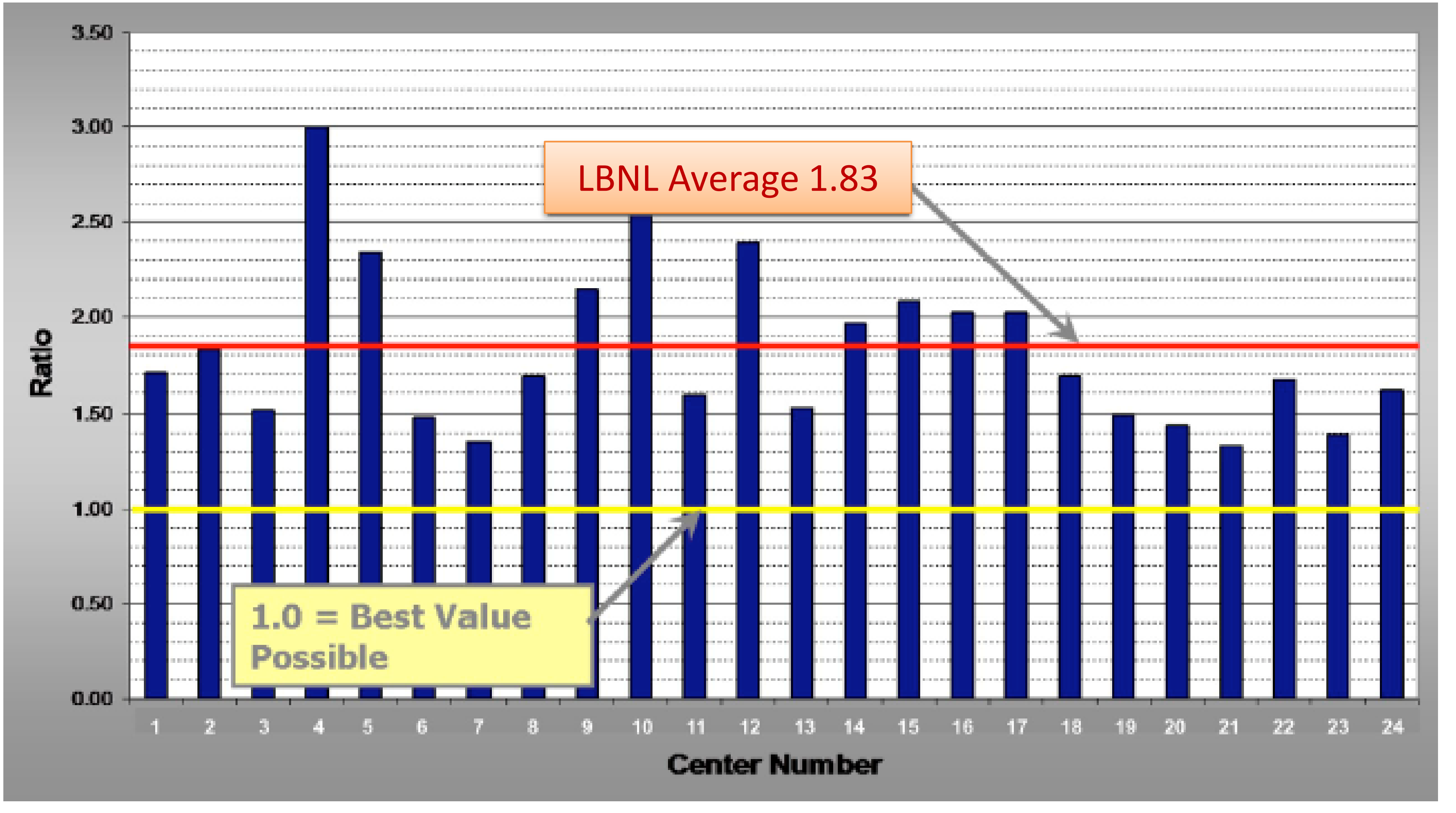
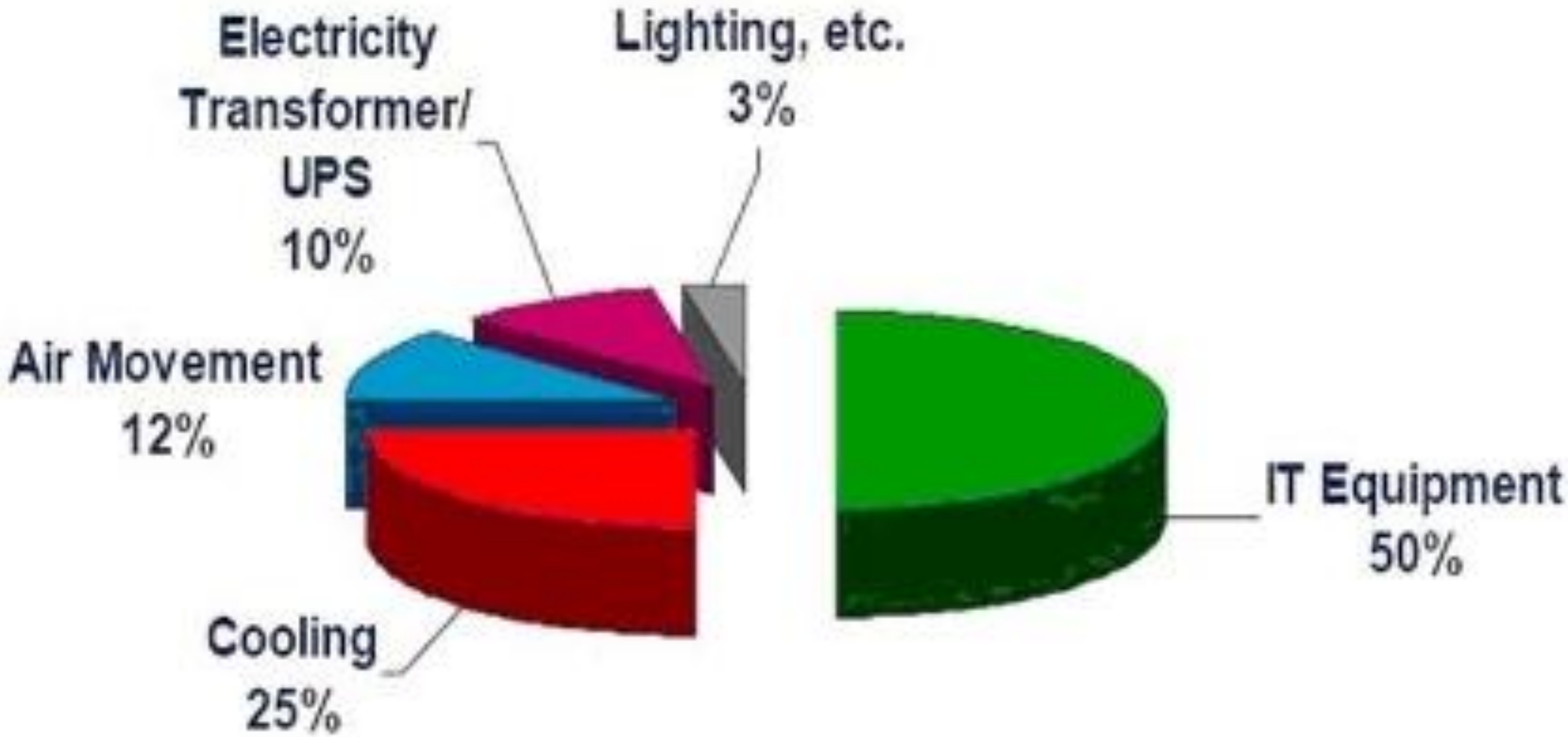
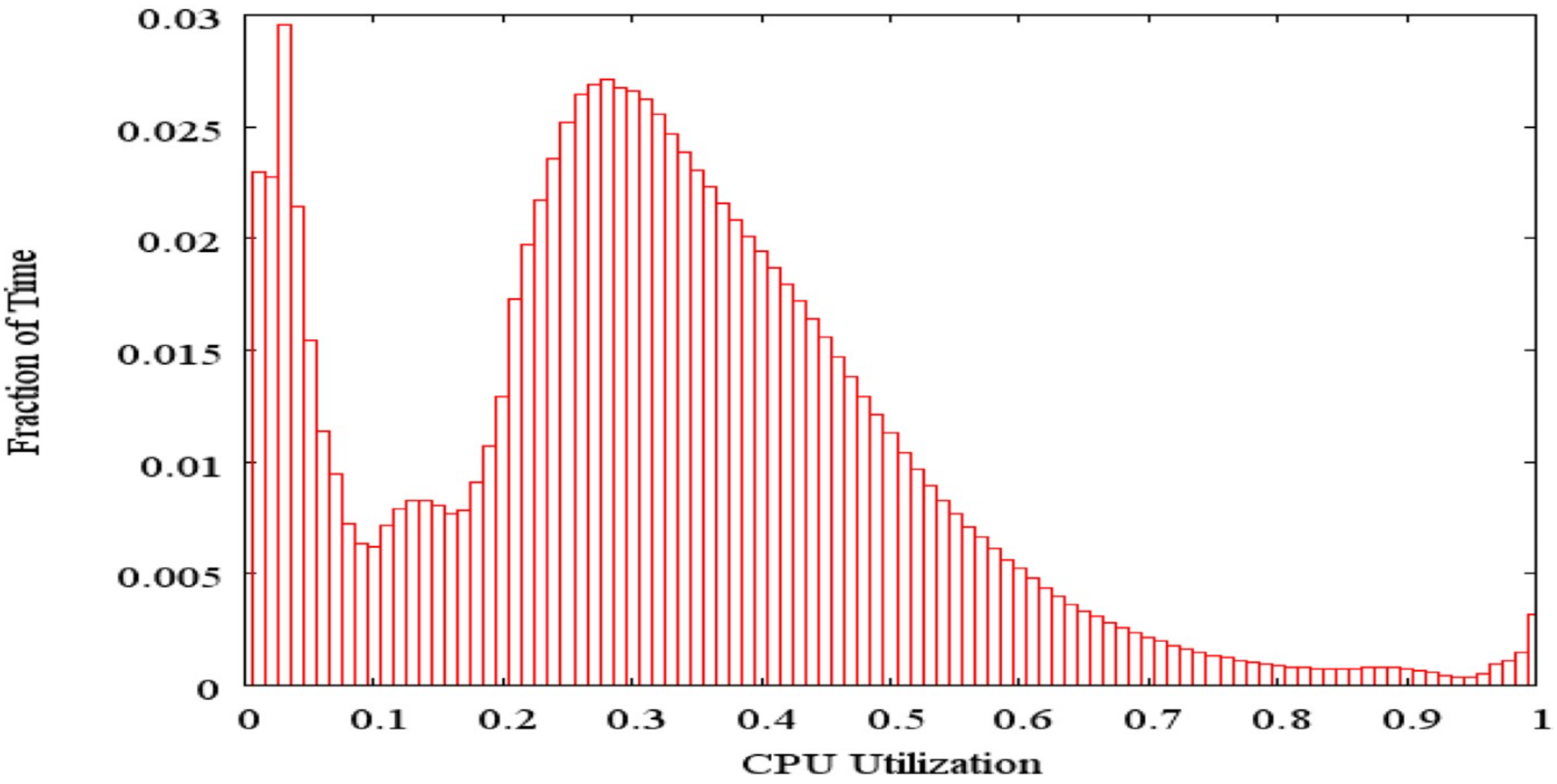


FIGURE 5.1: LBNL survey of the power usage efficiency of 24 datacenters, 2007 (Greenberg et al.)

Where Data Center Power Goes



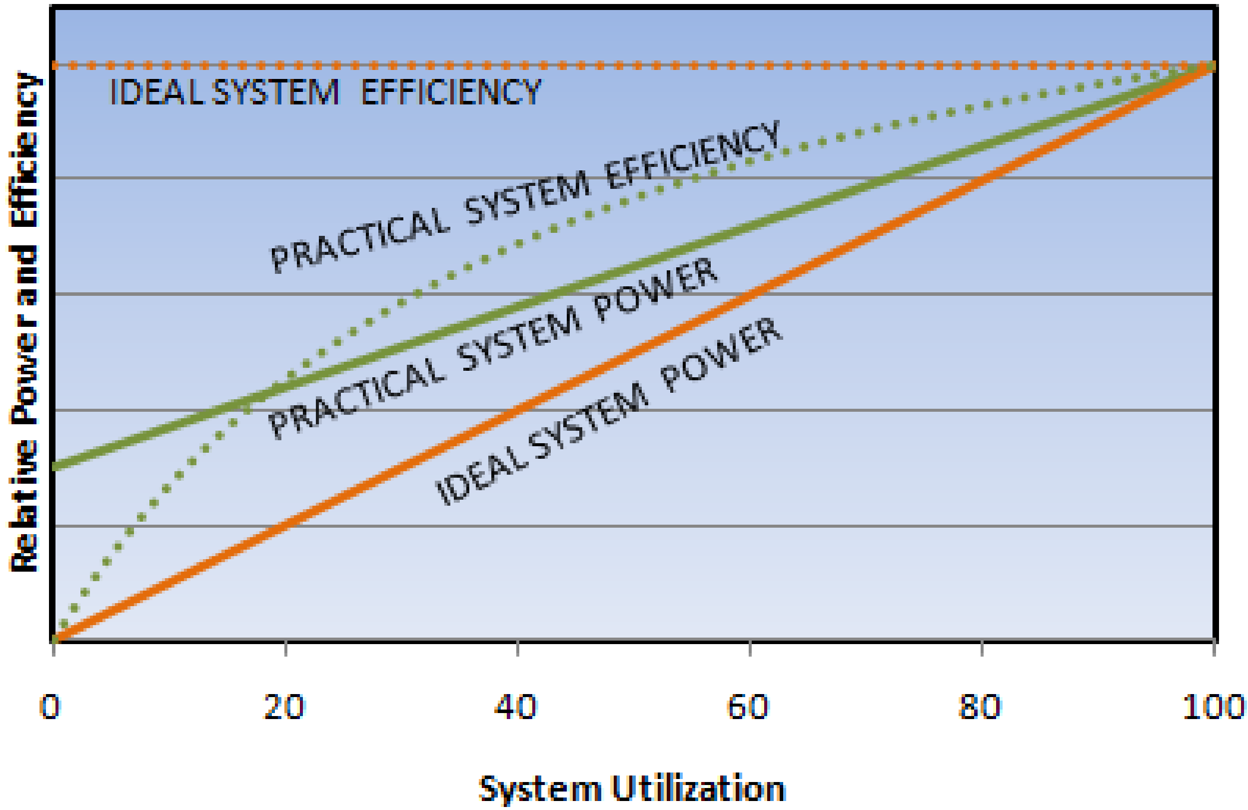
Load Profile of WSCs



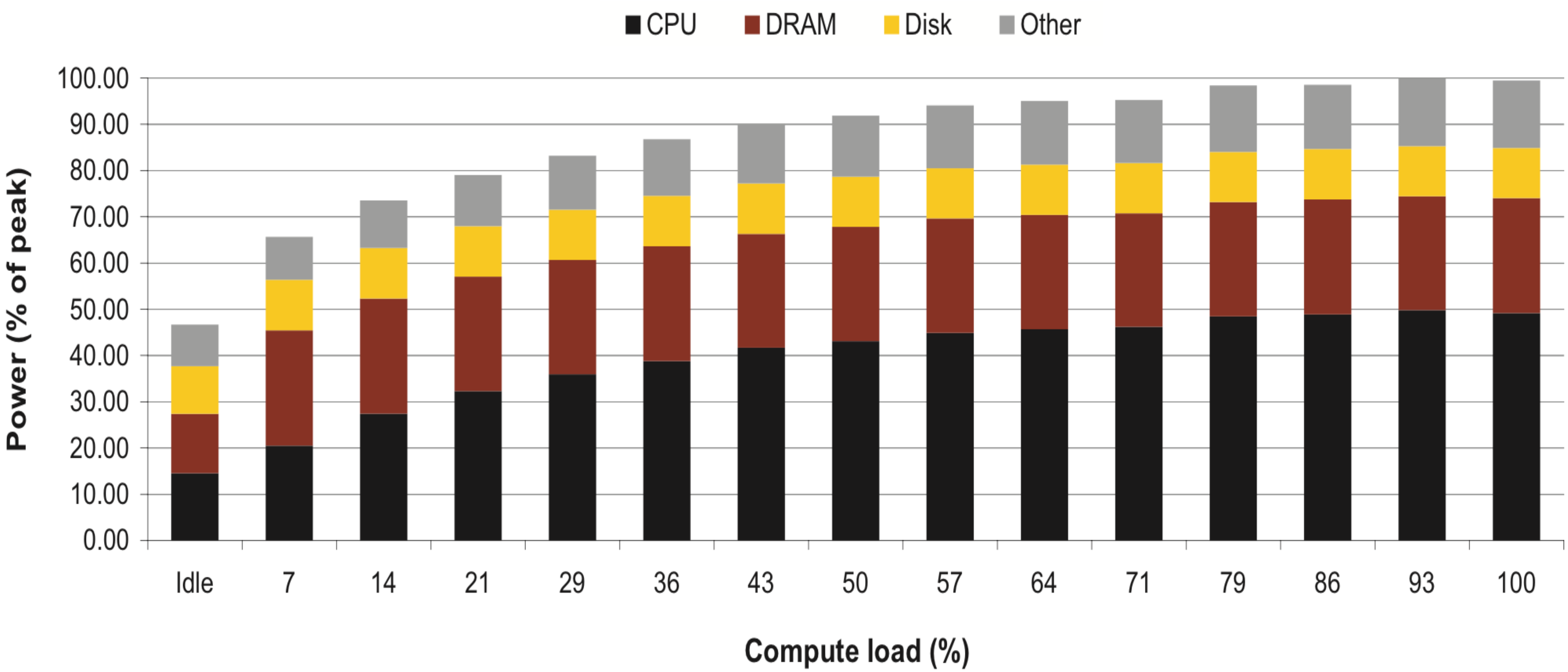
- Average CPU utilization of 5,000 Google servers, 6 month period
- Servers rarely idle or fully utilized, operating most of the time at ***10% to 50%*** of their maximum utilization

Energy-Proportional Computing: Design Goal of WSC

- Energy = Power x Time, Efficiency = Computation / Energy
- Desire:
  - Consume almost no power when idle ("Doing nothing well")
  - Gradually consume more power as the activity level increases



Cause of Poor Energy Proportionality



- CPU: 50% at peek, 30% at idle
- DRAM, disks, networking: 70% at idle!
- Need to improve the energy efficiency of peripherals

Cloud Computing: Scale of Economy

Name	Memory	vCPUs	Storage	Arch	Network Performance	Linux On Demand
M1 General Purpose Small	1.7 GB	1	160 GB	32/64-bit	Low	$0.044 hourly
M1 General Purpose Medium	3.75 GB	1	410 GB	32/64-bit	Moderate	$0.087 hourly
M1 General Purpose Extra Large	15.0 GB	4	1680 GB	64-bit	High	$0.35 hourly
C1 High-CPU Medium	1.7 GB	2	350 GB	32/64-bit	Moderate	$0.13 hourly
C1 High-CPU Extra Large	7.0 GB	8	1680 GB	64-bit	High	$0.52 hourly
I2 Extra Large	30.5 GB	4	800 GB	64-bit	Moderate	$0.853 hourly
I2 Double Extra Large	61.0 GB	8	1600 GB	64-bit	Moderate	$1.705 hourly
M4 Large	8.0 GB	2	EBS only	64-bit	Moderate	$0.108 hourly
M4 Extra Large	16.0 GB	4	EBS only	64-bit	High	$0.215 hourly
M4 16xlarge	256.0 GB	64	EBS only	64-bit	20 Gigabit	$3.447 hourly
General Purpose GPU Extra Large	61.0 GB	4	EBS only	64-bit	High	$0.9 hourly
General Purpose GPU 16xlarge	732.0 GB	64	EBS only	64-bit	20 Gigabit	$14.4 hourly
X1 Extra High-Memory 16xlarge	976.0 GB	64	1920 GB	64-bit	10 Gigabit	$6.669 hourly

- May 2017 AWS Instances & Prices
- Closest computer in WSC example is Standard Extra
- At these low rates, Amazon EC2 can make money!
  - even if used only 50% of time
- Virtual Machine (VM) plays an important role

Agenda

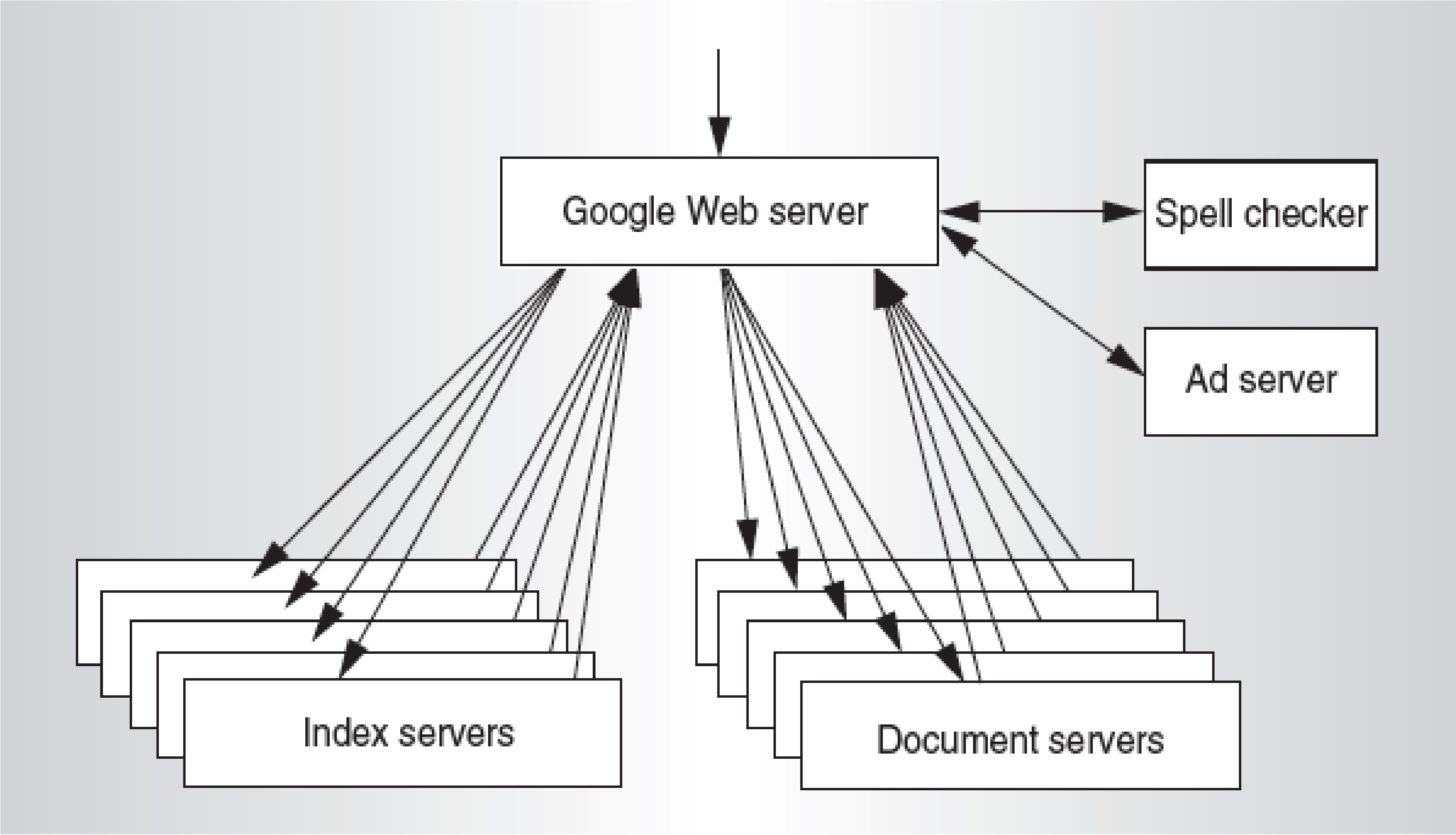
- Warehouse Scale Computing
- Request-level Parallelism

  e.g. Web search
- Data-level Parallelism
  - Hadoop, Spark
  - MapReduce

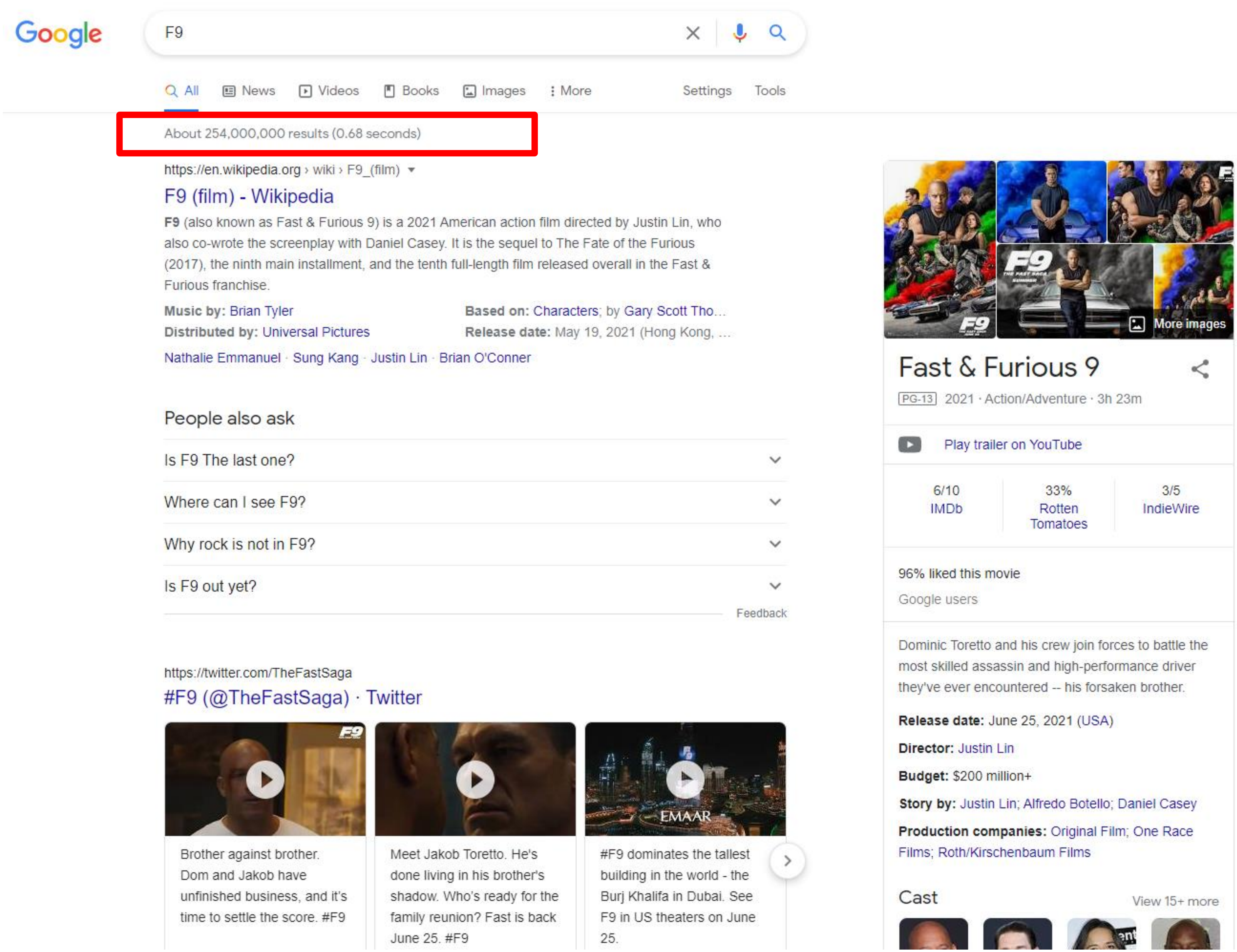
Request-Level Parallelism (RLP)

- Hundreds of thousands of requests per sec.
  - Popular Internet services like web search, social networking, …
  - Such requests are largely independent
    - Often involve read-mostly databases
    - Rarely involve read-write sharing or synchronization across requests
- Computation easily partitioned across different requests and even within a request

Google Query-Serving Architecture



Anatomy of a Web Search



Anatomy of a Web Search (1/3)

- Google “F9”
  - Direct request to “closest” Google WSC
  - 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 contains the search word, “F9”
  - Return document list with associated relevance score

Anatomy of a Web Search (2/3)

- In parallel,
  - Ad system: run ad auction for bidders on search terms
- 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 (3/3)

- 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 search hot spots, e.g., “F9”
  - Increases opportunities for ***request-level parallelism***
  - Makes the system more ***tolerant of failures***

Agenda

- Warehouse Scale Computing
- Request-level Parallelism

  e.g. Web search

- Data-level Parallelism
  - MapReduce
  - Hadoop, Spark

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

Problem Statement

- How to process large amounts of raw data (crawled documents, request logs, …) every day to compute derived data (inverted indices, page popularity, …), when computation is conceptually simple but input data is large and distributed across 100s to 1000s of servers, so as to finish in reasonable time?
- Challenge: Parallelize computation, distribute data, tolerate faults without obscuring simple computation with complex code to deal with issues

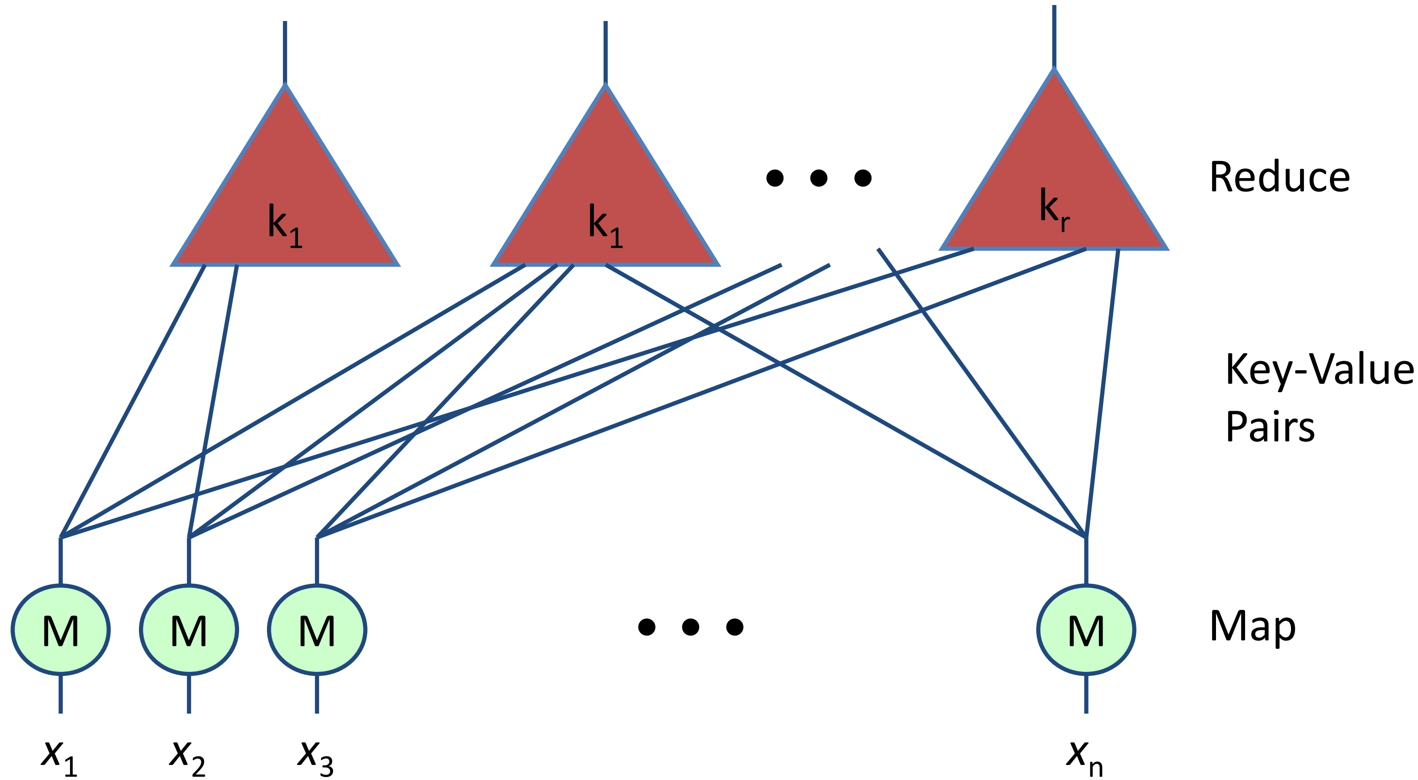
Solution: MapReduce

- Simple data-parallel ***programming model*** and ***implementation*** for processing large datasets
- 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

What is MapReduce used for?

- 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 Programming Model



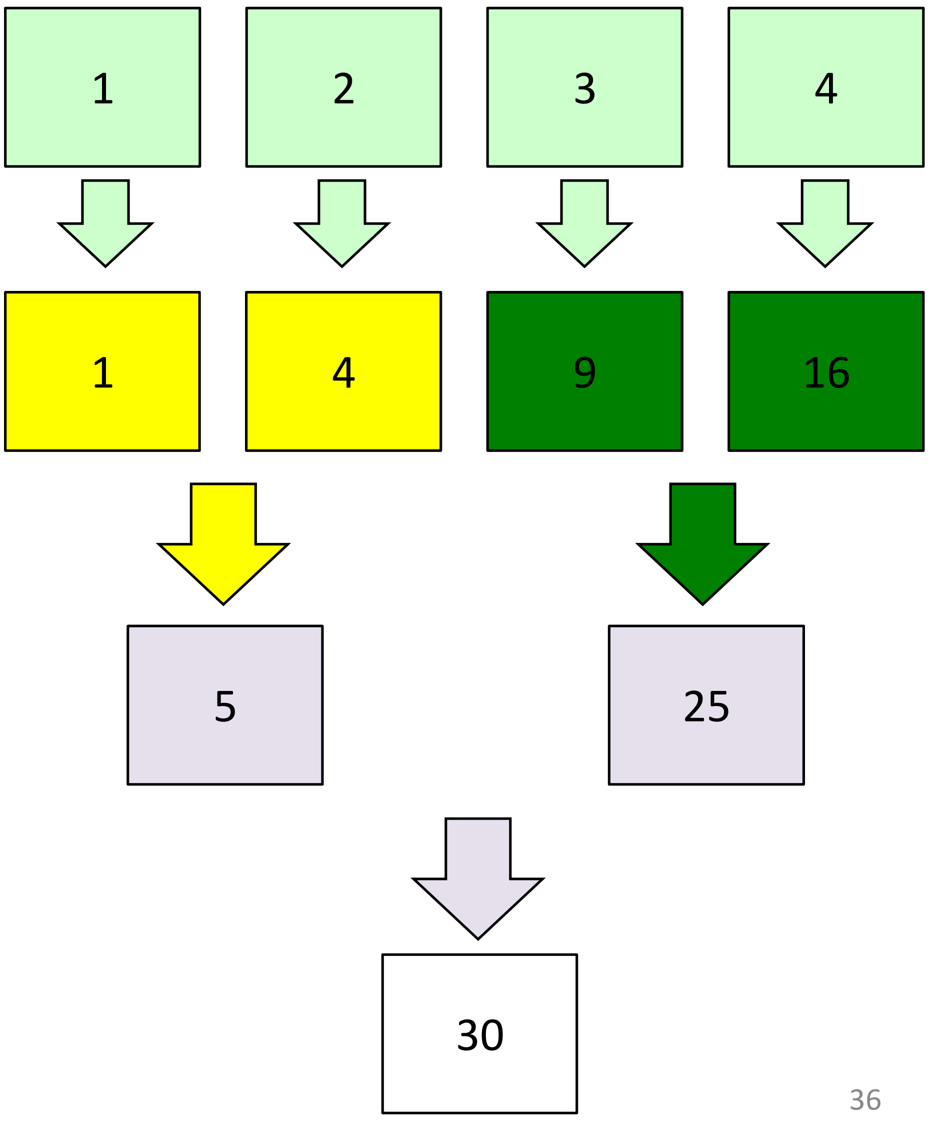
– Map computation across many objects
  - E.g., 10^{10} Internet web pages
– Aggregate results in many different ways
– System deals with issues of resource allocation & reliability

Dean & Ghemawat: "MapReduce: Simplified Data Processing on Large Clusters", OSDI 2004

Inspiration: Map & Reduce Functions, ex: Python

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

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

reduce(sum,
  map(square, A))
```



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)
```

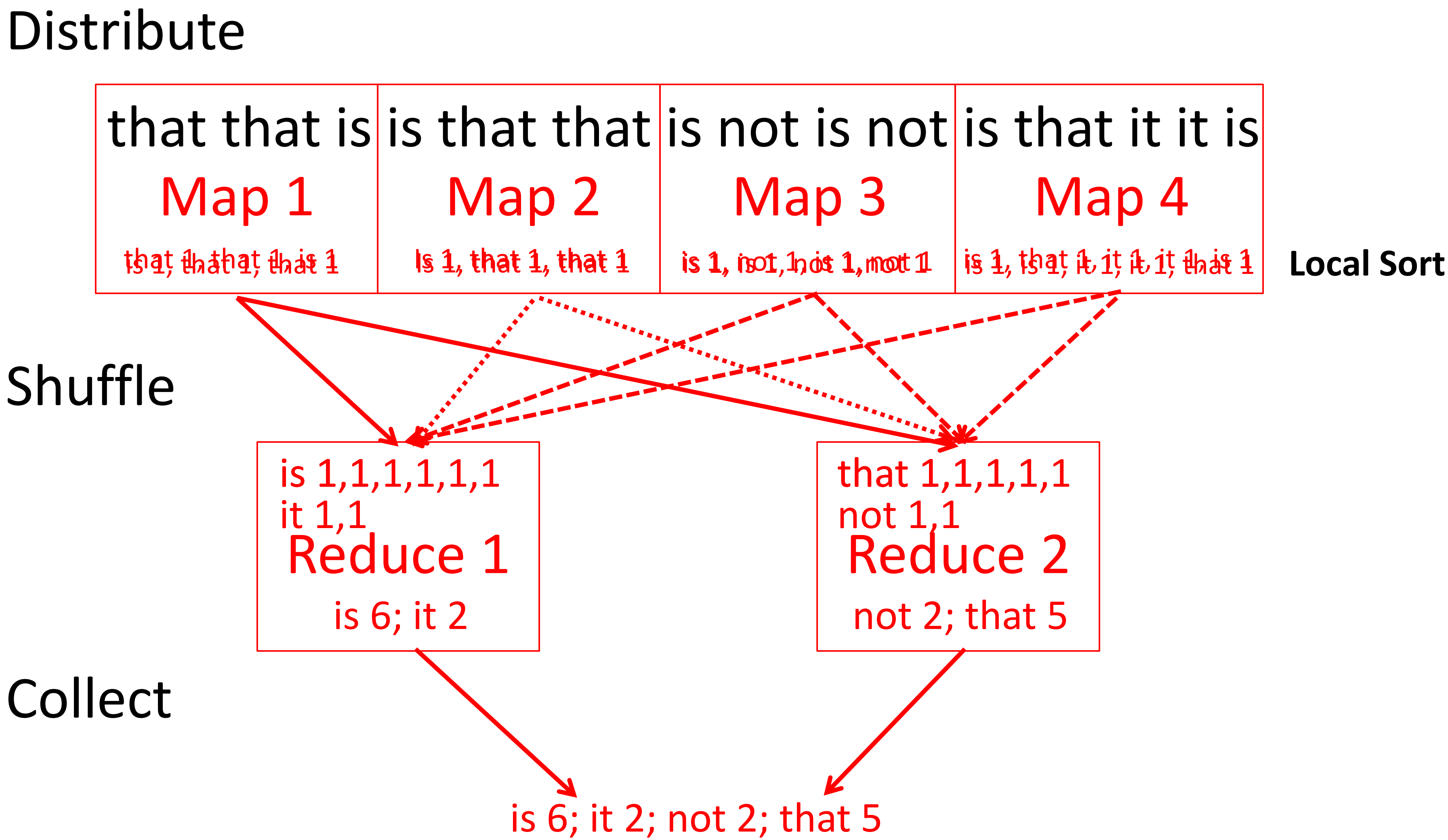
  - Slice data into "shards" or "splits" and distribute to workers
  - Compute set of intermediate key/value pairs

- ***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 Word Count Example



MapReduce Word Count Example

User-written Map function reads the document data and parses out the words. For each word, it writes the (key, value) pair of (word, 1). That is, the word is treated as the intermediate key and the associated value of 1 means that we saw the word once.

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)
```

MapReduce Word Count Example

The intermediate data is then sorted by MapReduce by keys and the user's Reduce function is called for each unique key. In this case, Reduce is called with a list of a "1" for each occurrence of the word that was parsed from the document. The function adds them up to generate a total word count for that word.

Reduce phase: (word, list(counts)) → (word, count_sum)

```
// ("I", [1,1]) → ("I",2)
```

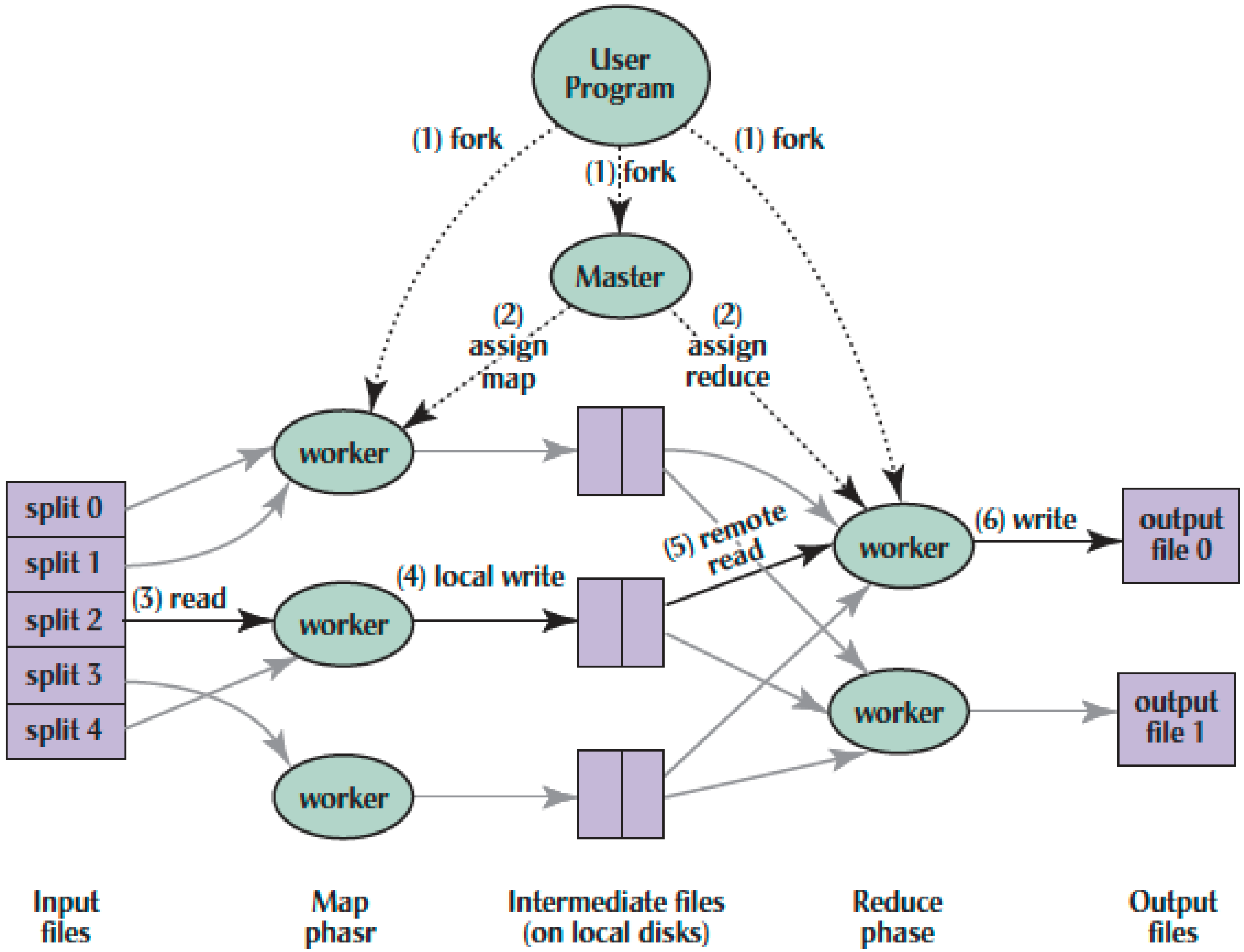
```
reduce(key, values):
  result = 0
  for each v in values:
    result += v
  emit(key, result)
```

MapReduce Processing Example: Count Word Occurrences

- Pseudo Code: for each word in input, generate <key=word, value=1>
- Reduce sums all counts emitted for a particular word across all mappers

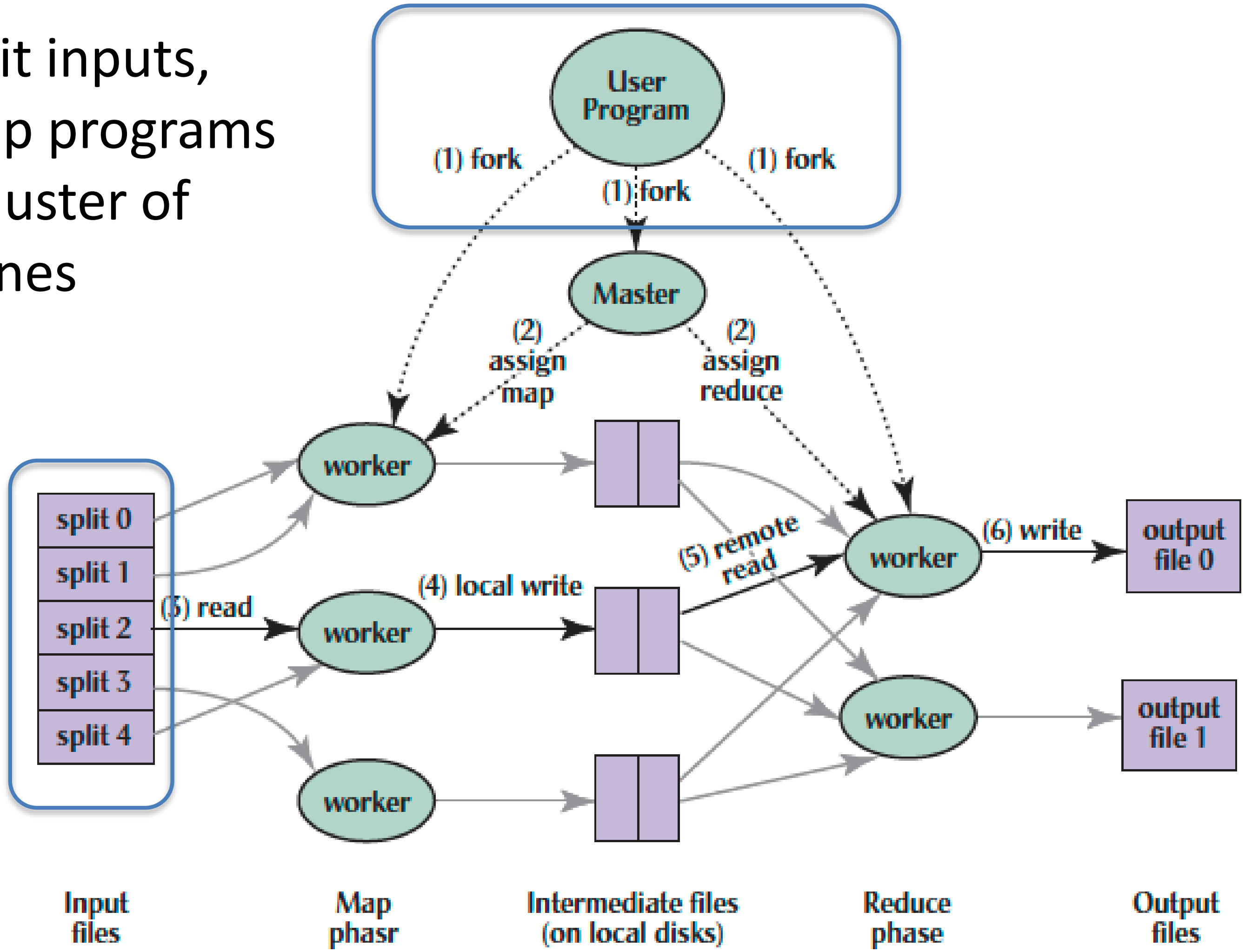
```
map(String input_key, String input_value):
    // input_key: document name
    // input_value: document contents
    for each word w in input_value:
      EmitIntermediate(w, "1"); // Produce count of words

reduce(String output_key, Iterator intermediate_values):
    // output_key: a word
    // intermediate_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
      result += ParseInt(v); // get integer from key-value
    Emit(output_key, result);
```

MapReduce Implementation



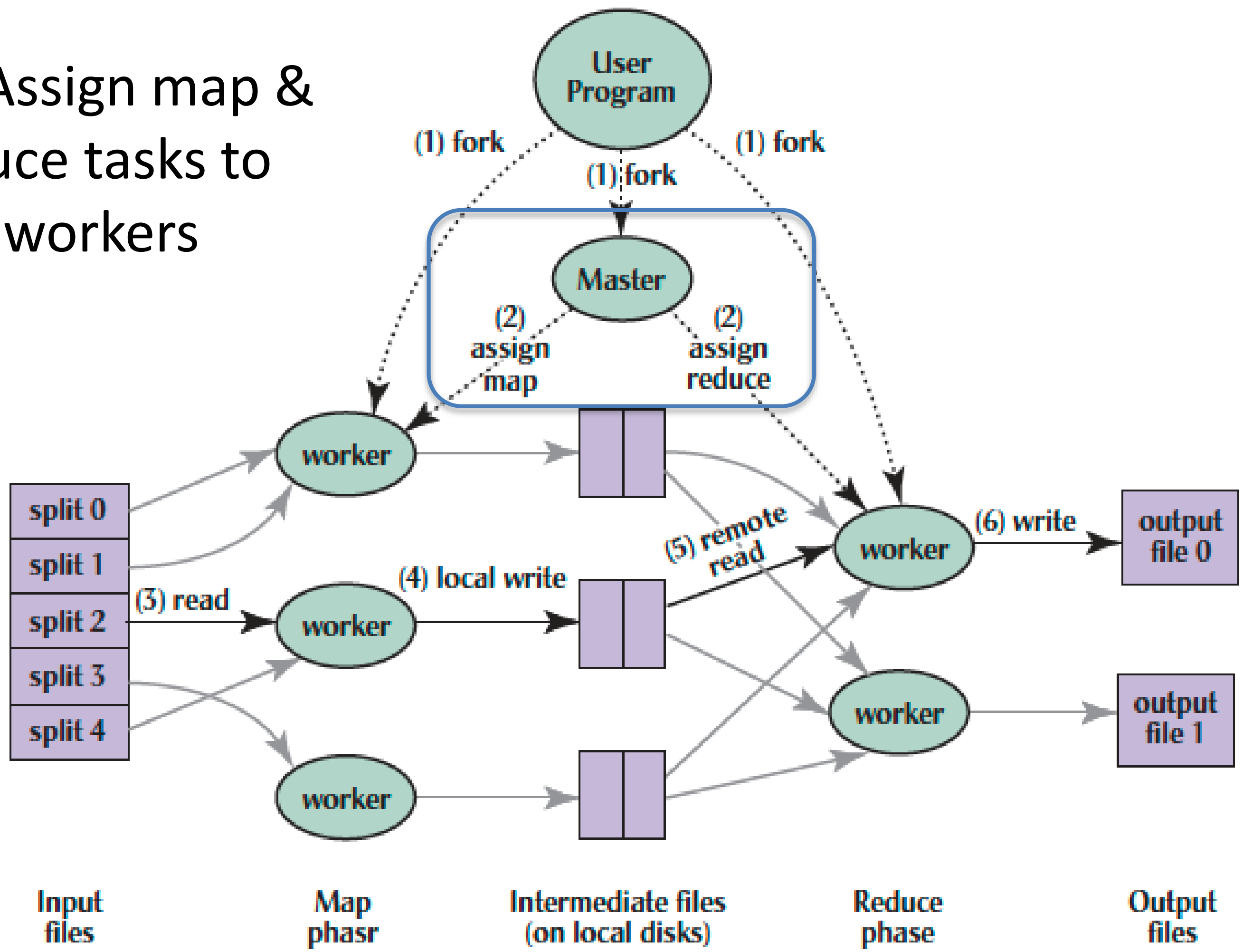
MapReduce Execution

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



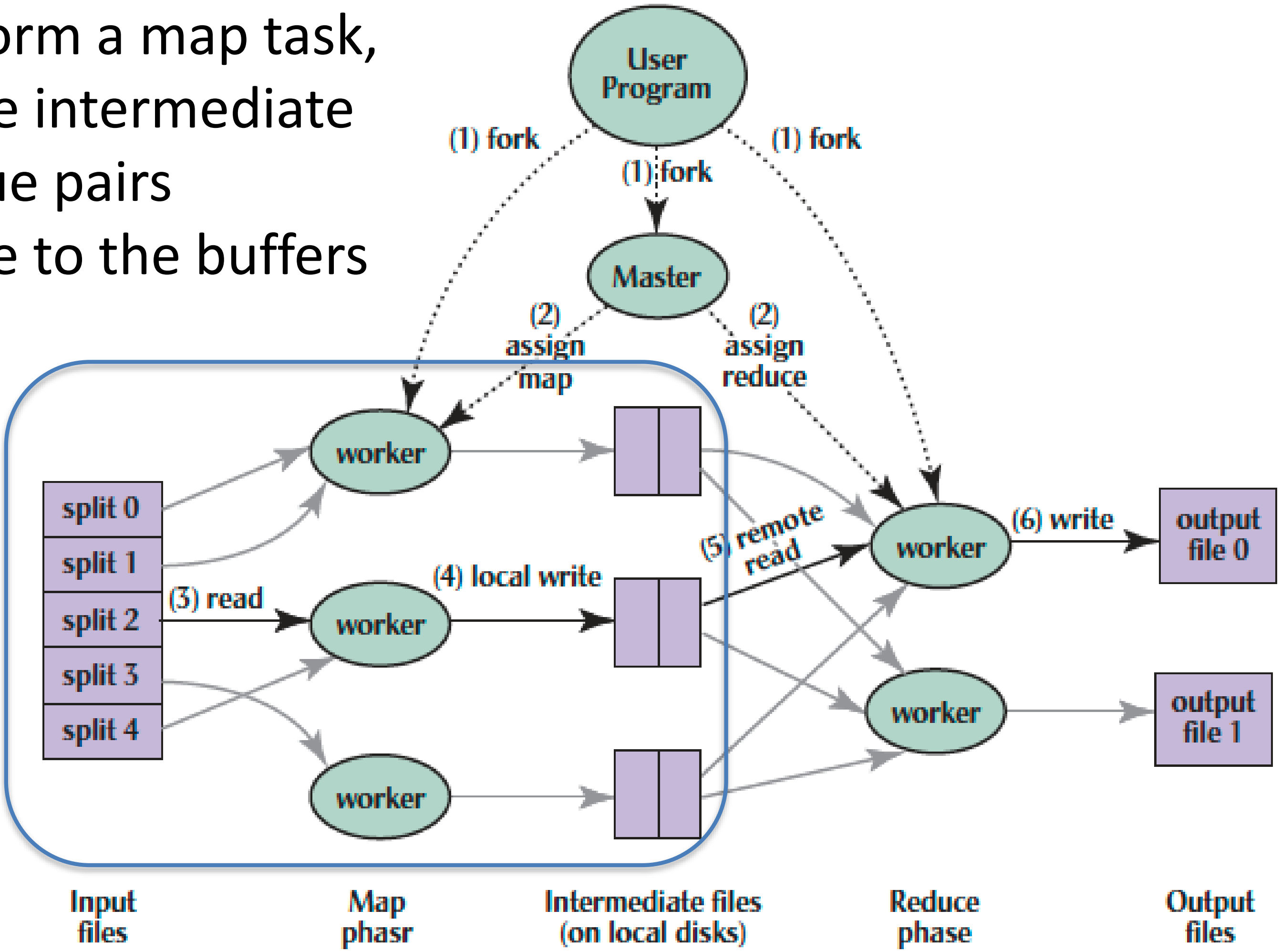
MapReduce Execution

(2) Assign map & reduce tasks to idle workers

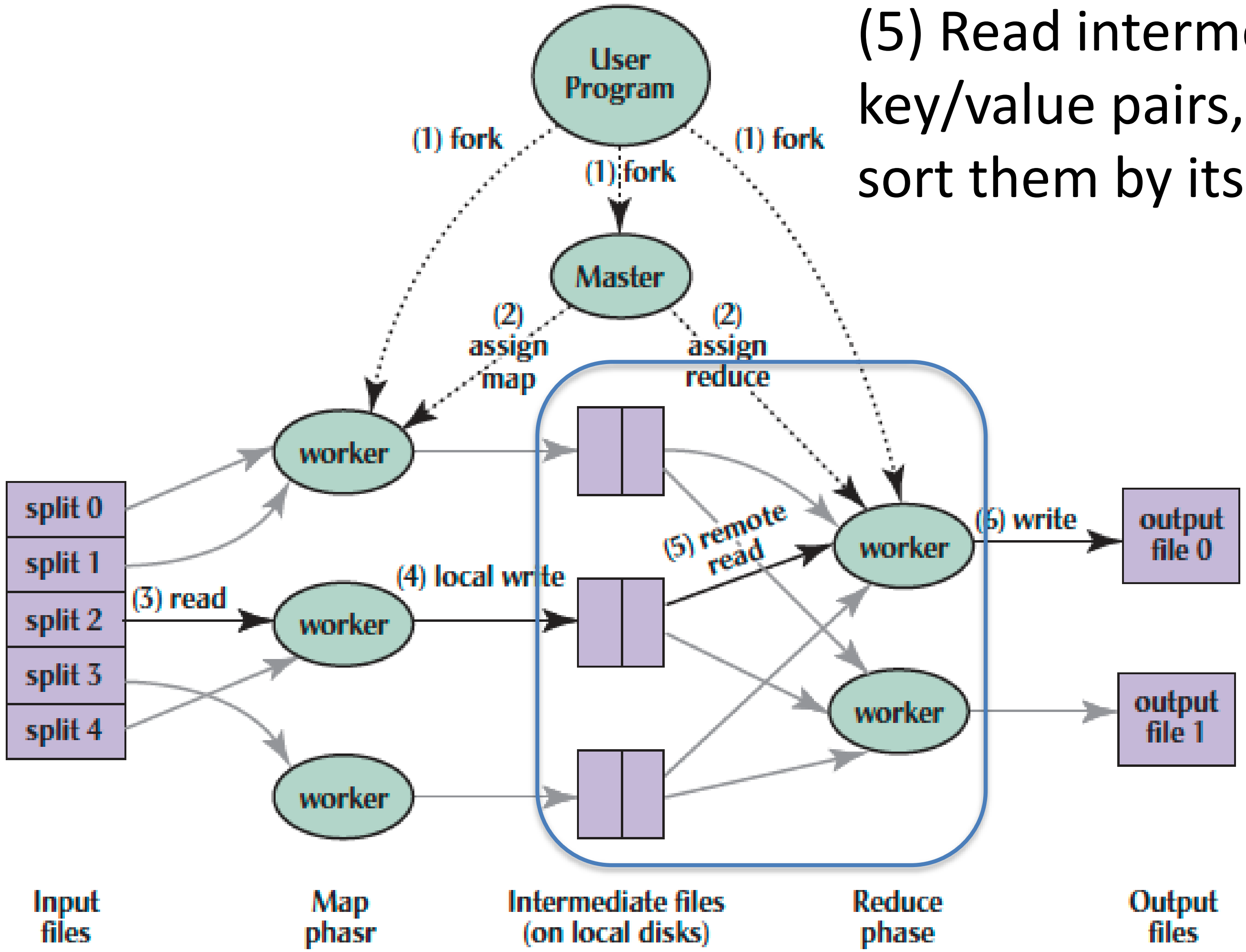


MapReduce Execution

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

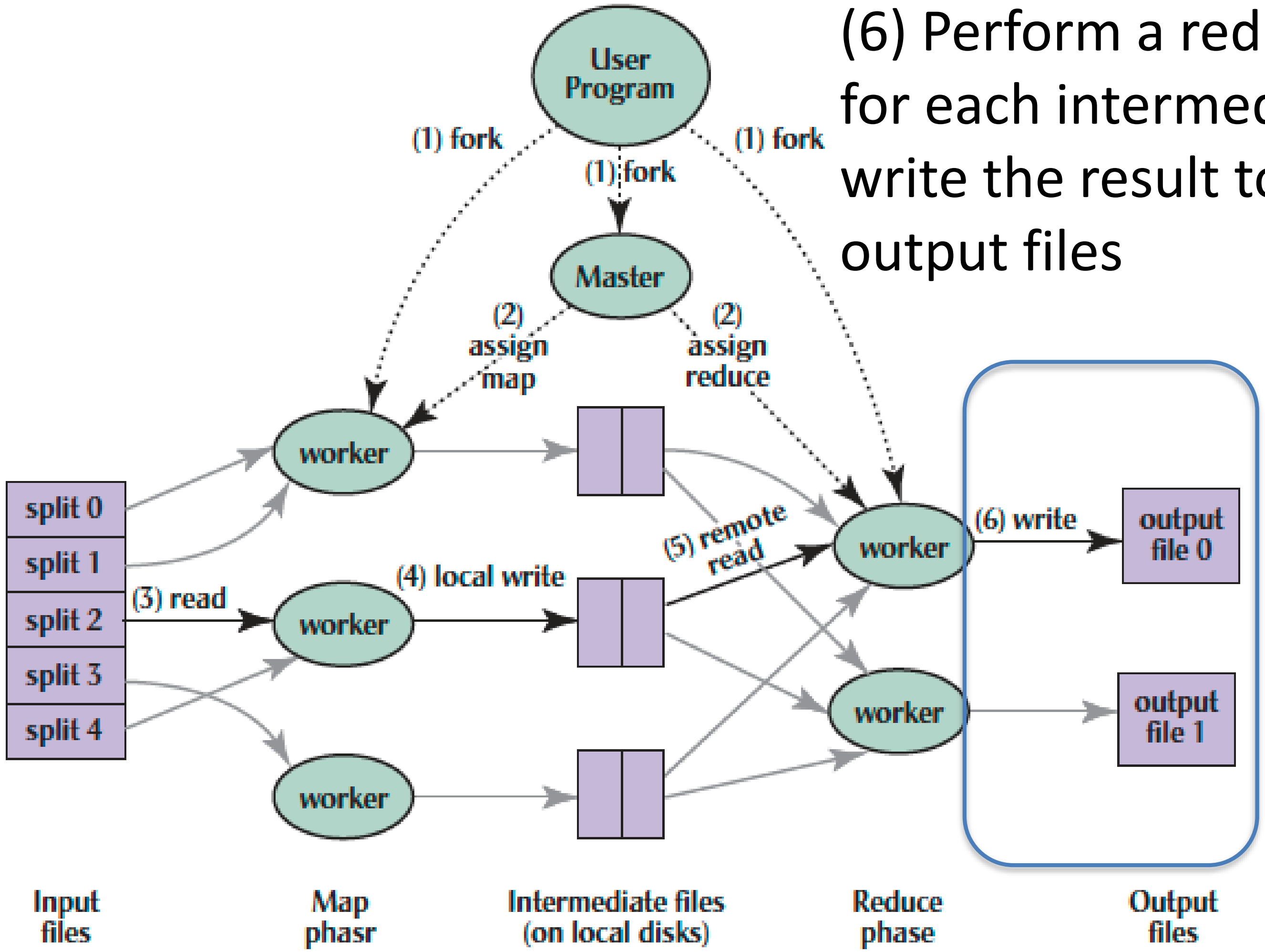


MapReduce Execution



(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

Big Data Framework: Hadoop & Spark

- Apache Hadoop
  - Open-source MapReduce Framework
  - Hadoop Distributed File System (HDFS)
  - Hadoop YARN Resource Management
  - MapReduce Java APIs
  - more than half of the Fortune 50 used Hadoop (2013)
- Apache Spark
  - Fast and general engine for large-scale data processing.
  - Running on HDFS
  - Provides Java, Scala, Python APIs for
    - Database
    - Machine learning
    - Graph algorithm

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)
```

And, in Conclusion ...

- Warehouse-Scale Computers (WSCs)
  - New class of computers
  - Scalability, energy efficiency, high failure rate
- Cloud Computing
  - Benefits of WSC computing for third parties
  - “Elastic” pay as you go resource allocation
- Request-Level Parallelism
  - High request volume, each largely independent of other
  - Use replication for better request throughput, availability
- MapReduce Data Parallelism
  - Map: Divide large data set into pieces for independent parallel processing
  - Reduce: Combine and process intermediate results to obtain final result
  - Hadoop, Spark