Big Data Infrastructure

Session 1: Introduction

Jimmy Lin
University of Maryland
Monday, January 26, 2015

What is this course about?

- What is big data?
- Why big data?
- *Infrastructure* for big data

From the Ivory Tower...

Source: Wikipedia (All Souls College, Oxford)


… to building sh*t that works
KEEP CLEAR
Source: Wikipedia (Factory)

Source: Wikipedia (All Souls College, Oxford)

Big Data

Source: Wikipedia (Hard disk drive)

Processes 20 PB a day (2008)
Crawls 20B web pages a day (2012)
Search index is 100+ PB (5/2014)
Bigtable serves 2+ EB, 600M QPS (5/2014)

400B pages, 10+ PB (2/2014)

Hadoop: 365 PB, 330K nodes (6/2014)

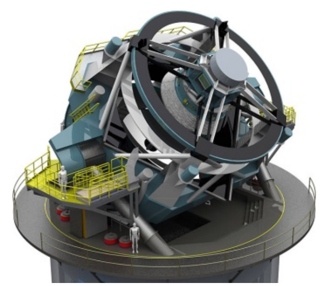
150 PB on 50k+ servers running 15k apps (6/2011)

Hadoop: 10K nodes, 150K cores, 150 PB (4/2014)

300 PB data in Hive + 600 TB/day (4/2014)

S3: 2T objects, 1.1M request/second (4/2013)

LHC: ~15 PB a year



LSST: 6-10 PB a year (~2020)

SKA: 0.3 – 1.5 EB per year (~2020)

How much data?

Why big data?

Science

Engineering

Commerce

Source: Wikipedia (Everest)

Science

Emergence of the 4th Paradigm

Data-intensive e-Science

Maximilien Brice, © CERN

Engineering

The unreasonable effectiveness of data

Count and normalize!

Source: Wikipedia (Three Gorges Dam)

Know thy customers

Data → Insights → Competitive advantages

Commerce

Source: Wikiedia (Shinjuku, Tokyo)

Source: Wikipedia (Noctilucent cloud)

Course Administrivia

Source: http://www.flickr.com/photos/artmind_etcetera/6336693594/

My Expectations

- You're *already* a good Java programmer
  - This course does *not* teach programming
  - You're expected to pick up Hadoop with minimal help
- You're good at debugging
  - Your own code
  - Compiling, patching, and installing open source software
- You have basic knowledge of:
  - Probability and statistics, discrete math
  - Computer architecture

How will I actually learn Hadoop?

- Hadoop: The Definitive Guide
- RTFM
- RTFC(!)

This course is not for you…

- If you're not genuinely interested in the topic
- If you can't put in the time
- If you're uncomfortable with the uncertainty, unpredictability, etc. that comes with immature software

Otherwise, this will be a rewarding and fun course!

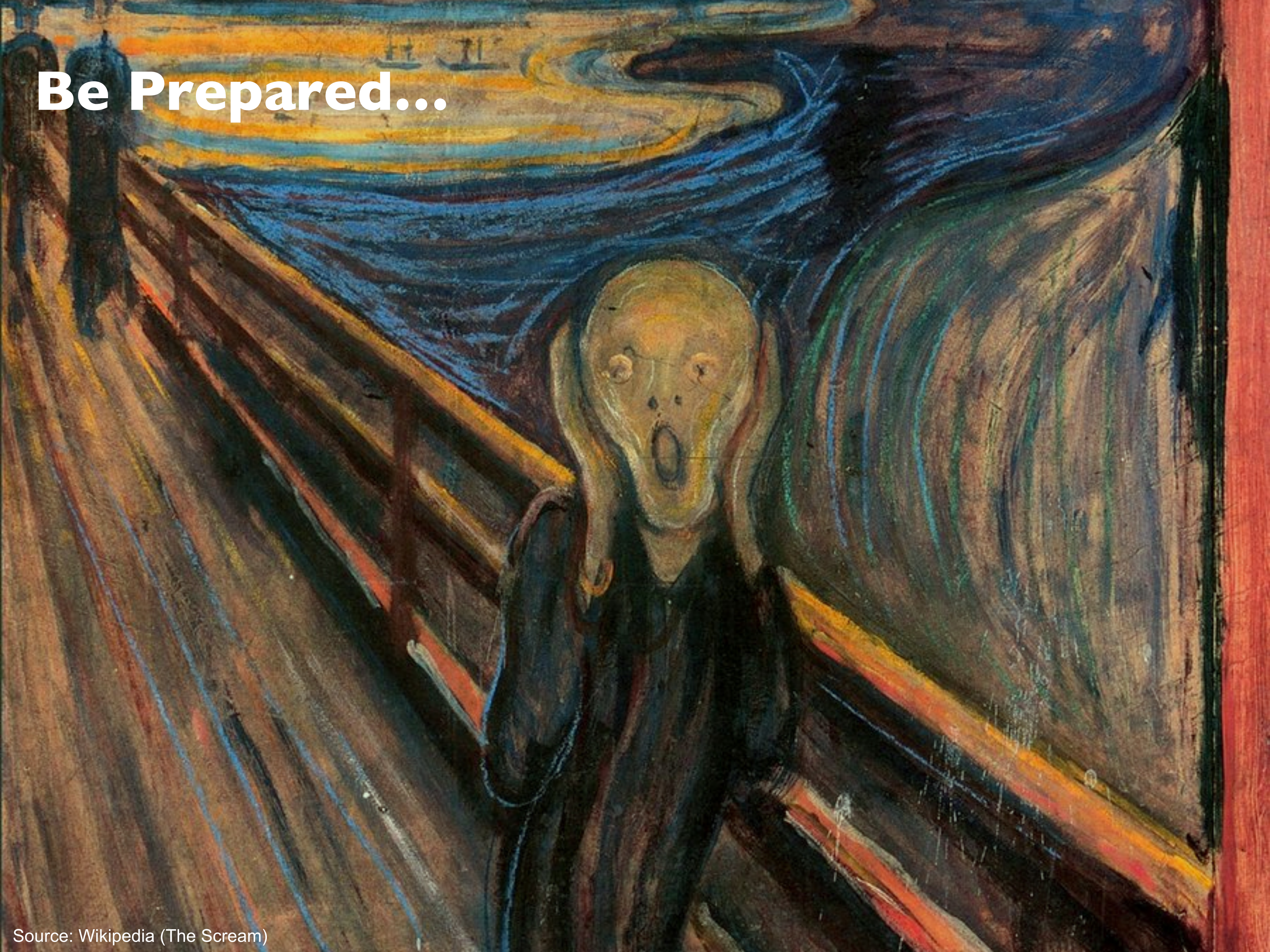
Details, Details…

- Make sure you're on the mailing list!
- Textbooks
- Components of the final grade:
  - Assignments
  - Final exam
  - Final project
- I am unlikely to accept the following excuses:
  - "Too busy"
  - "It took longer than I thought it would take"
  - "It was harder than I initially thought"
  - "My dog ate my homework" and modern variants thereof

Hadoop Resources

- Hadoop on your local machine
- Hadoop in a virtual machine on your local machine
- Hadoop on a UMIACS cluster

Be Prepared...



Source: Wikipedia (The Scream)

"Hadoop Zen"

- Parts of the ecosystem are still immature
  - We've come a long way since 2007, but still far to go…
  - Bugs, undocumented "features", inexplicable behavior, etc.
- Don't get frustrated (take a deep breath)…
  - Those W$*#T@F! moments
- Be patient…
  - We will inevitably encounter "situations" along the way
- Be flexible…
  - We will have to be creative in workarounds
- Be constructive…
  - Tell me how I can make everyone's experience better

“Hadoop Zen”

Source: Wikipedia (Japanese rock garden)

Interlude: Cloud Computing

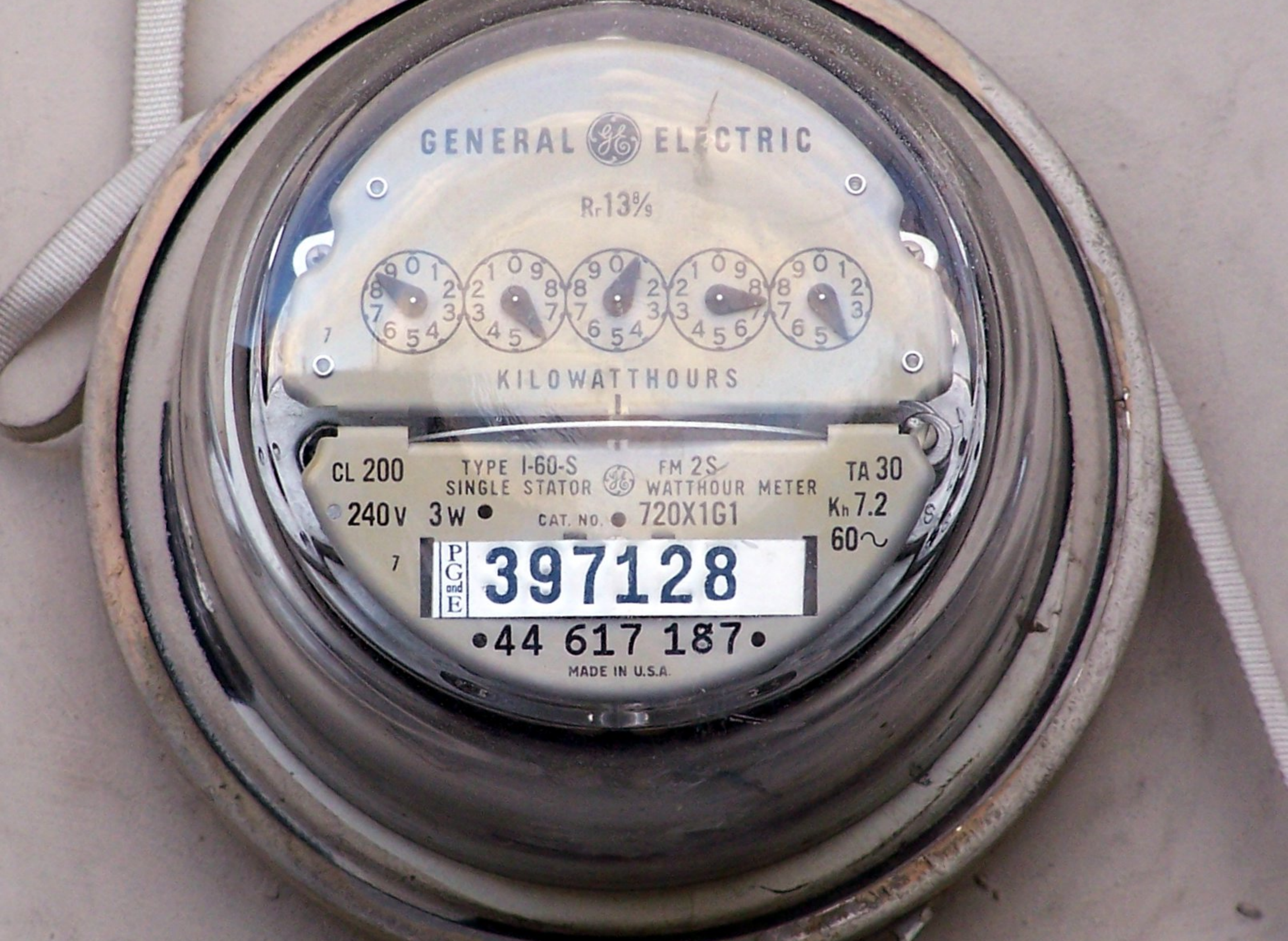
Source: Wikipedia (Clouds)

The best thing since sliced bread?

- Before clouds…
  - Grids
  - Connection machine
  - Vector supercomputers
  - …
- Cloud computing means many different things:
  - Big data
  - Rebranding of web 2.0
  - Utility computing
  - Everything as a service

Rebranding of web 2.0

- Rich, interactive web applications
  - Clouds refer to the servers that run them
  - AJAX as the de facto standard (for better or worse)
  - Examples: Facebook, YouTube, Gmail, …
- "The network is the computer": take two
  - User data is stored "in the clouds"
  - Rise of the netbook, smartphones, etc.
  - Browser *is* the OS



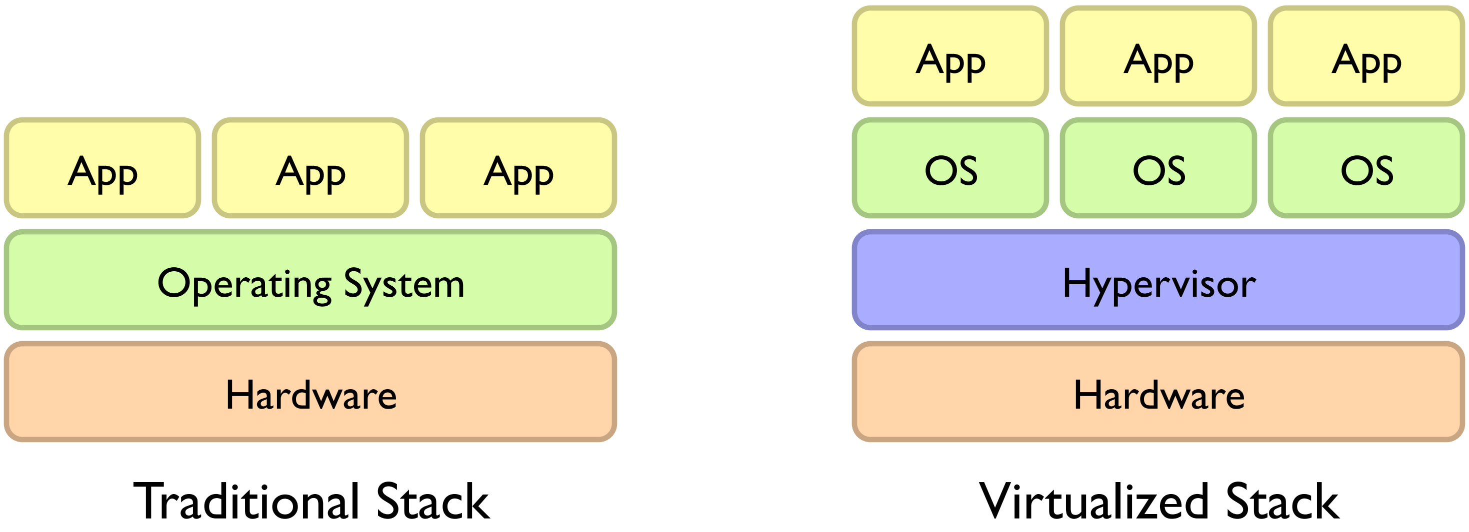
Source: Wikipedia (Electricity meter)

Utility Computing

- What?
  - Computing resources as a metered service (“pay as you go”)
  - Ability to dynamically provision virtual machines
- Why?
  - Cost: capital vs. operating expenses
  - Scalability: “infinite” capacity
  - Elasticity: scale up or down on demand
- Does it make sense?
  - Benefits to cloud users
  - Business case for cloud providers

I think there is a world market for about five computers.

Enabling Technology: Virtualization



Everything as a Service

- Utility computing = Infrastructure as a Service (IaaS)
  - Why buy machines when you can rent cycles?
  - Examples: Amazon's EC2, Rackspace
- Platform as a Service (PaaS)
  - Give me nice API and take care of the maintenance, upgrades, …
  - Example: Google App Engine
- Software as a Service (SaaS)
  - Just run it for me!
  - Example: Gmail, Salesforce

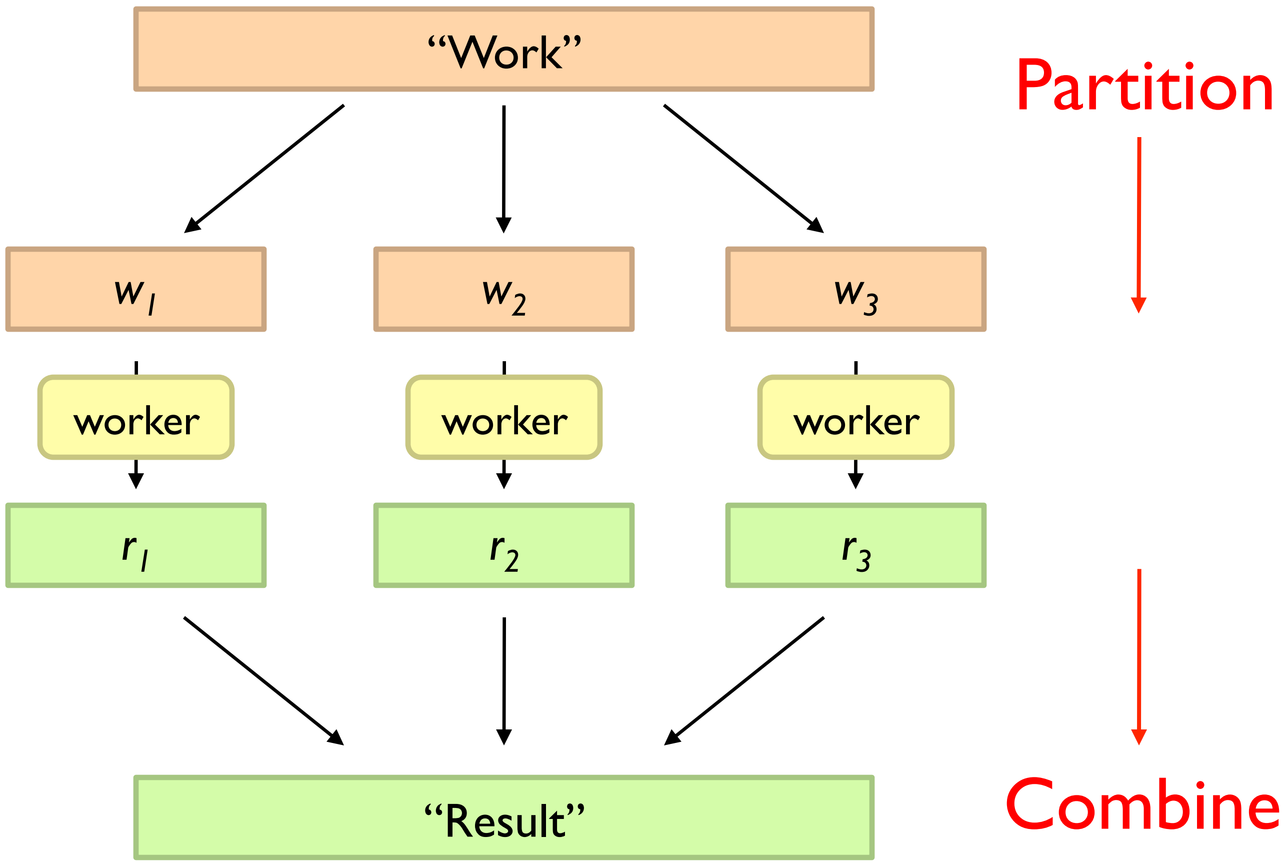
Who cares?

- A source of problems…
  - Cloud-based services *generate* big data
  - Clouds make it easier to start companies that *generate* big data
- As well as a solution…
  - Ability to provision analytics clusters on-demand in the cloud
  - Commoditization and democratization of big data capabilities

Tackling Big Data

Source: Google

Divide and Conquer



Parallelization Challenges

- How do we assign work units to workers?
- What if we have more work units than workers?
- What if workers need to share partial results?
- How do we aggregate partial results?
- How do we know all the workers have finished?
- What if workers die?

What's the common theme of all of these problems?

Common Theme?

- Parallelization problems arise from:
  - Communication between workers (e.g., to exchange state)
  - Access to shared resources (e.g., data)
- Thus, we need a synchronization mechanism

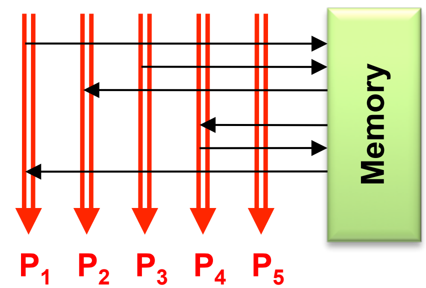
Source: Ricardo Guimarães Herrmann

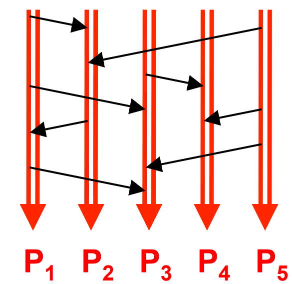
Managing Multiple Workers

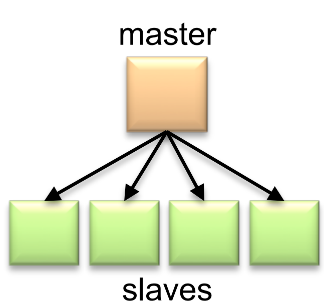
- Difficult because
  - We don't know the order in which workers run
  - We don't know when workers interrupt each other
  - We don't know when workers need to communicate partial results
  - We don't know the order in which workers access shared data
- Thus, we need:
  - Semaphores (lock, unlock)
  - Conditional variables (wait, notify, broadcast)
  - Barriers
- Still, lots of problems:
  - Deadlock, livelock, race conditions...
  - Dining philosophers, sleeping barbers, cigarette smokers...
- Moral of the story: be careful!

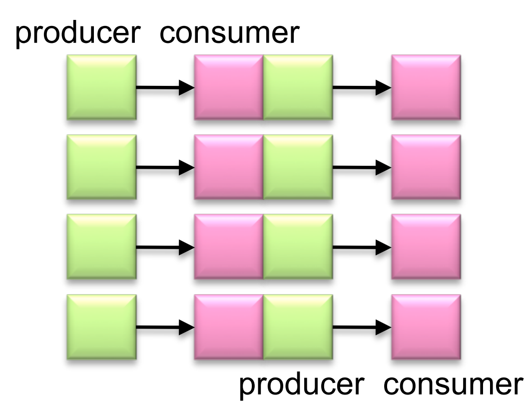
Current Tools

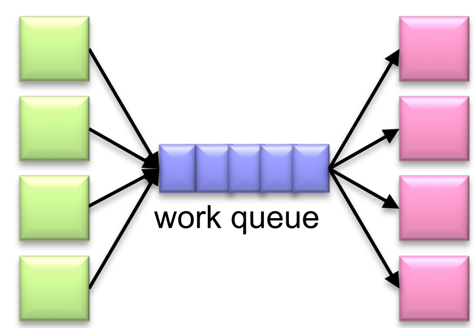
- Programming models
  - Shared memory (pthreads)
  - Message passing (MPI)
- Design Patterns
  - Master-slaves
  - Producer-consumer flows
  - Shared work queues







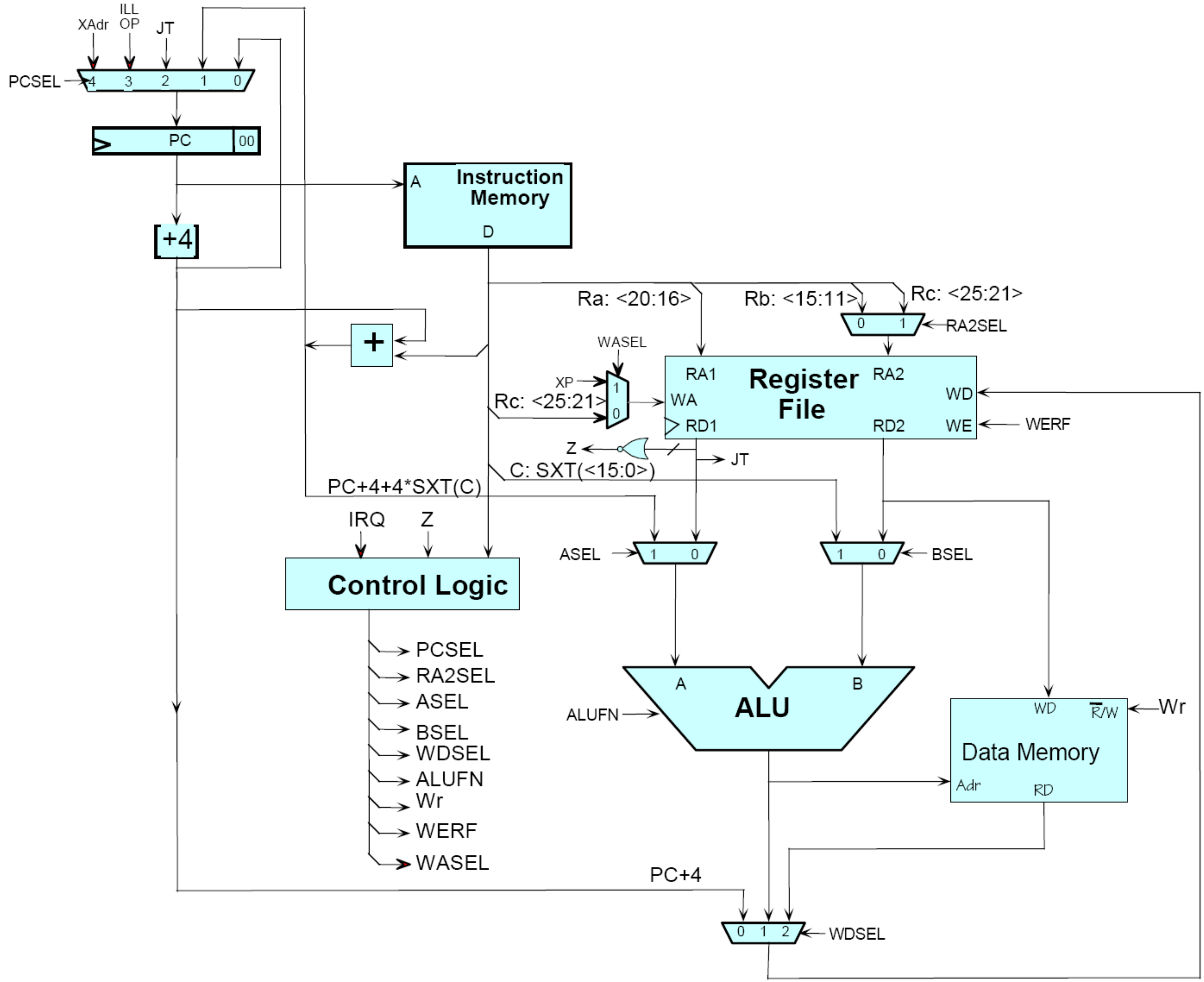




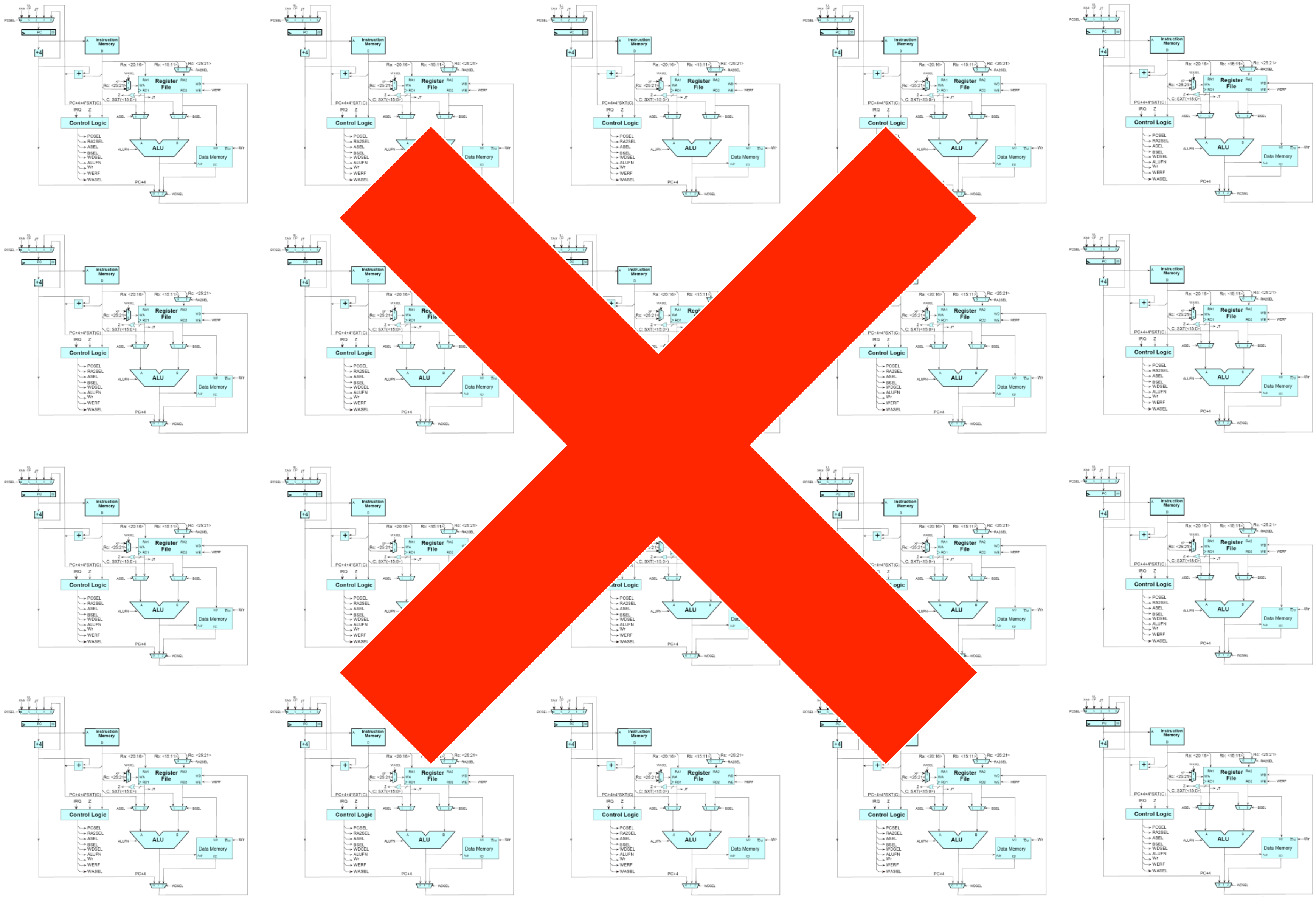
Where the rubber meets the road

- Concurrency is difficult to reason about
- Concurrency is even more difficult to reason about
  - At the scale of datacenters and across datacenters
  - In the presence of failures
  - In terms of multiple interacting services
- Not to mention debugging…
- The reality:
  - Lots of one-off solutions, custom code
  - Write you own dedicated library, then program with it
  - Burden on the programmer to explicitly manage everything

Source: Wikipedia (Flat Tire)



Source: MIT Open Courseware



Source: MIT Open Courseware


The datacenter *is* the computer!
Source: Google

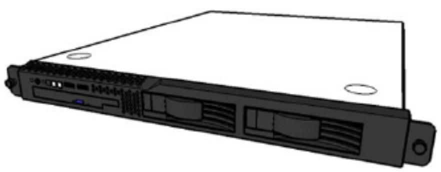
Source: Wikipedia (The Dalles, Oregon)

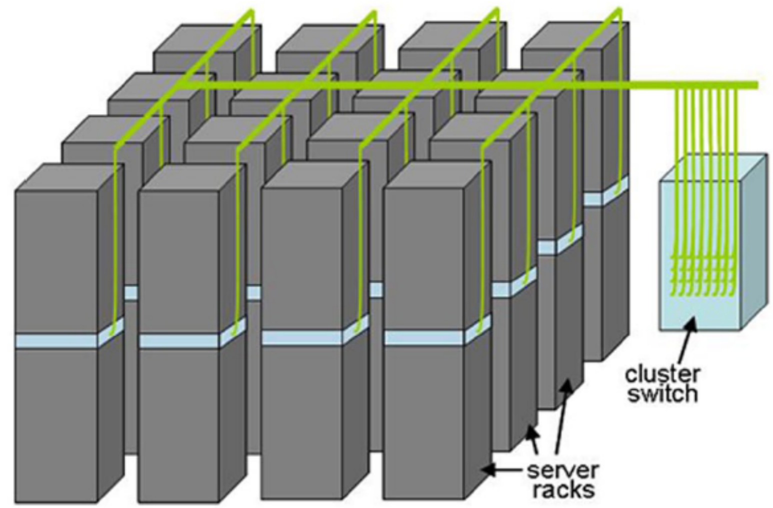
Source: Google

Source: Google

Source: Bonneville Power Administration

Building Blocks





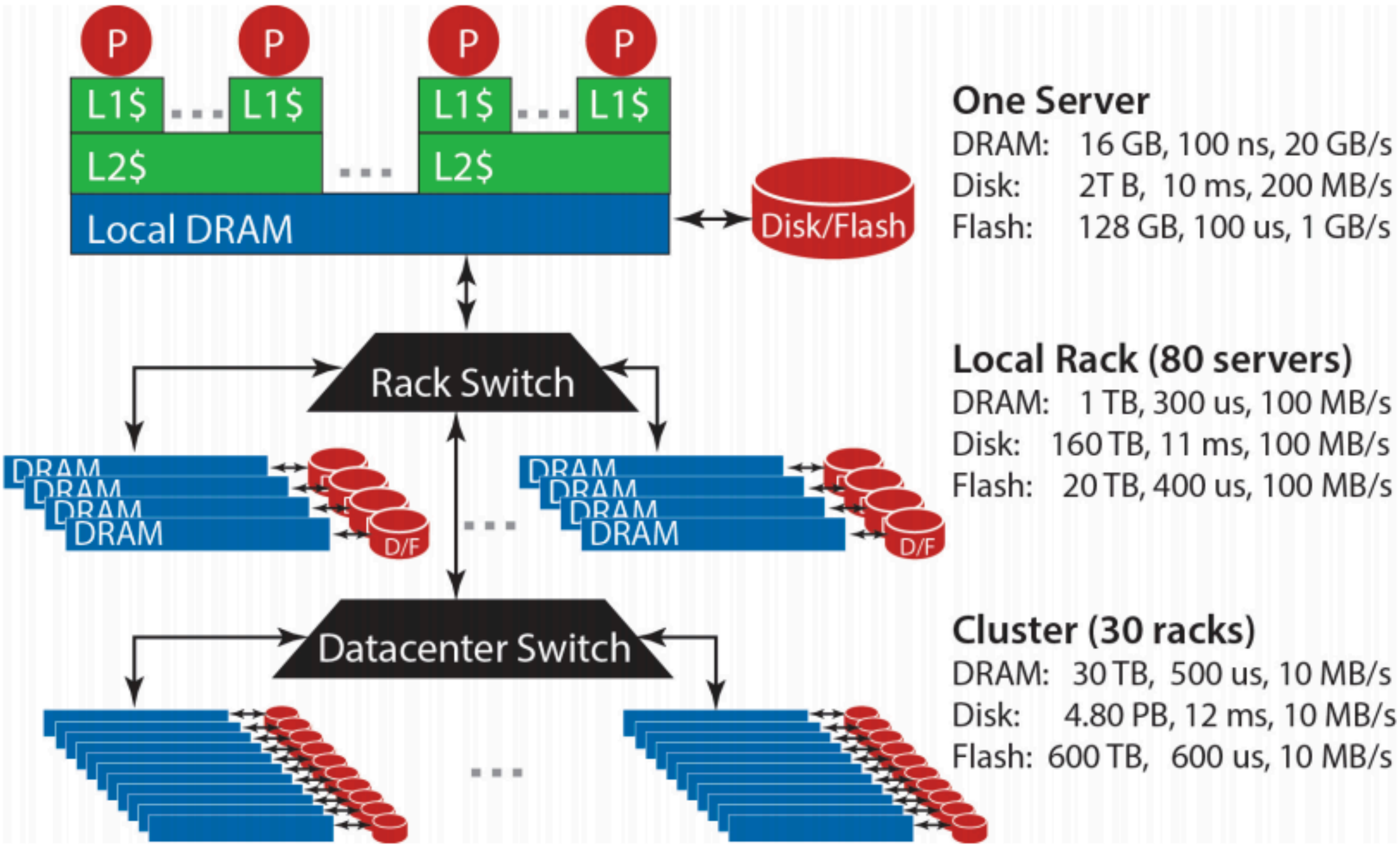
Source: Barroso and Urs Hölzle (2009)

Source: Google

Source: Google

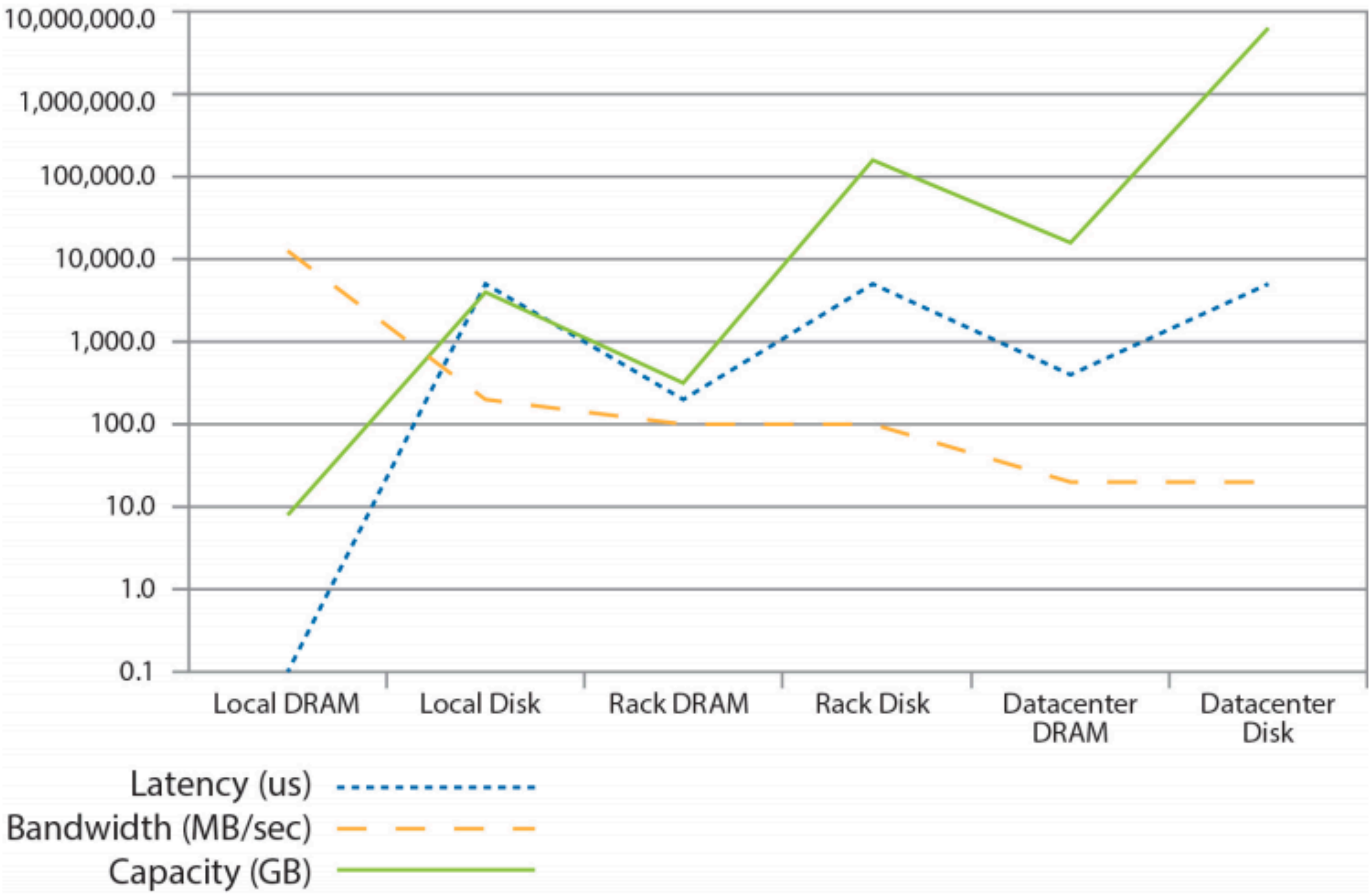
Source: Facebook

Storage Hierarchy



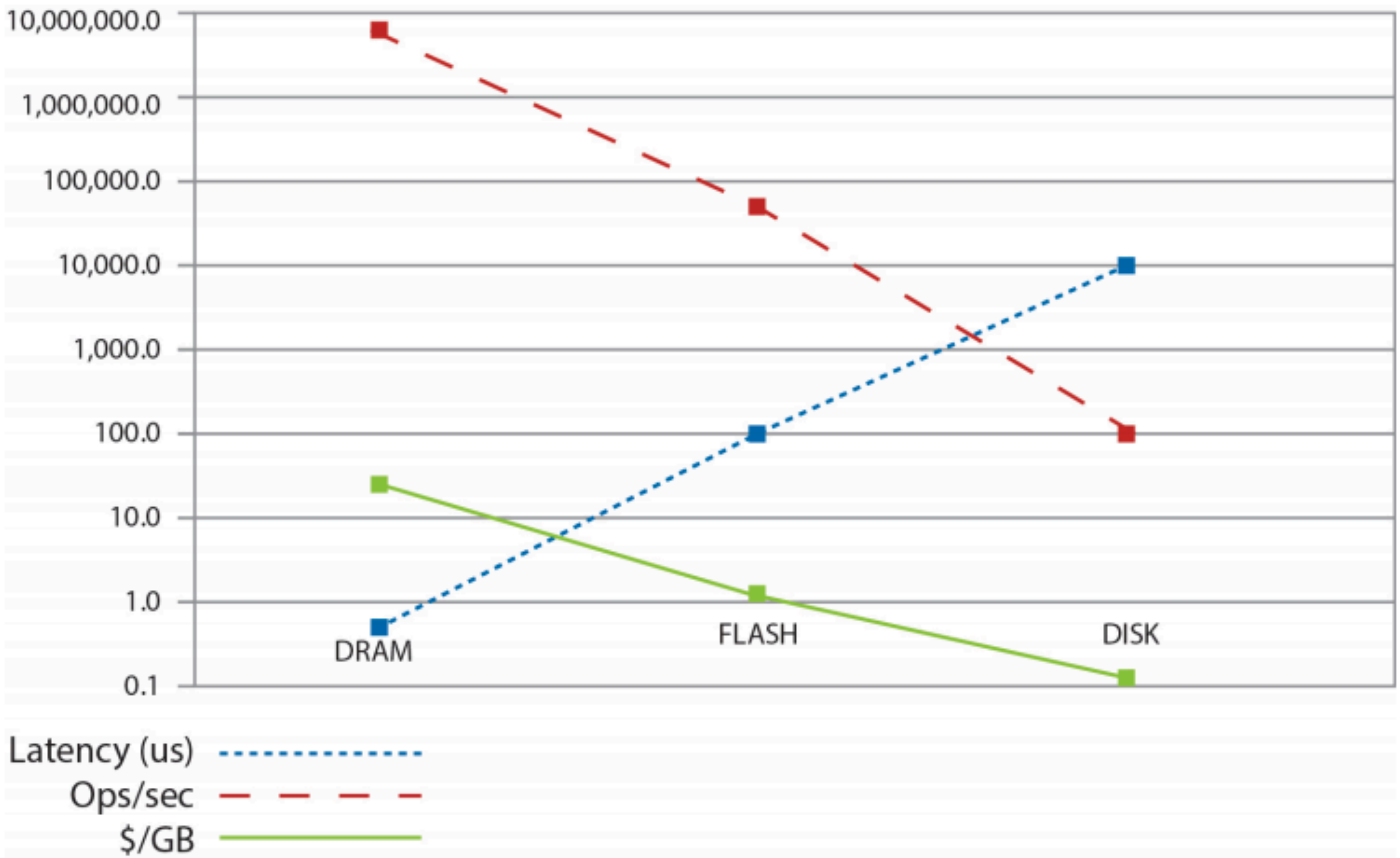
Source: Barroso and Urs Hölzle (2013)

Storage Hierarchy



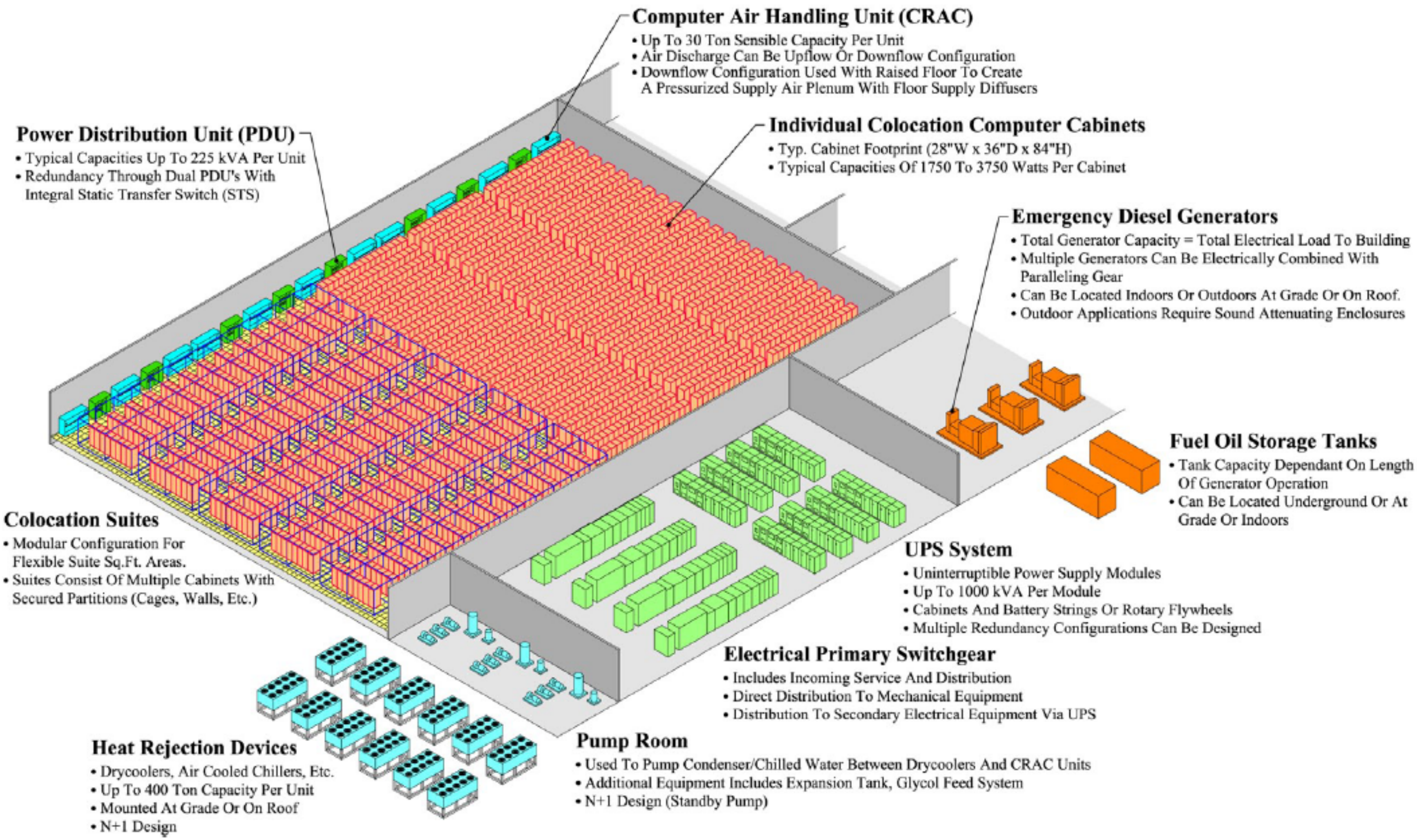
Source: Barroso and Urs Hölzle (2013)

Storage Hierarchy



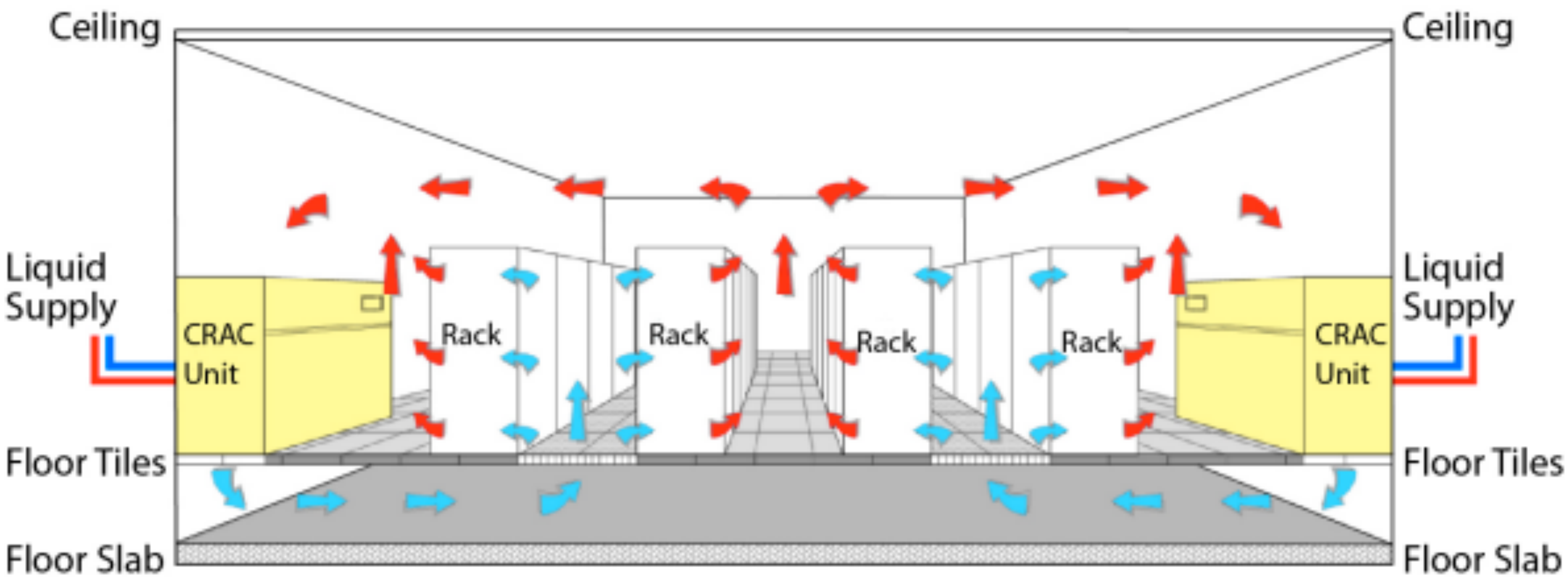
Source: Barroso and Urs Hölzle (2013)

Anatomy of a Datacenter



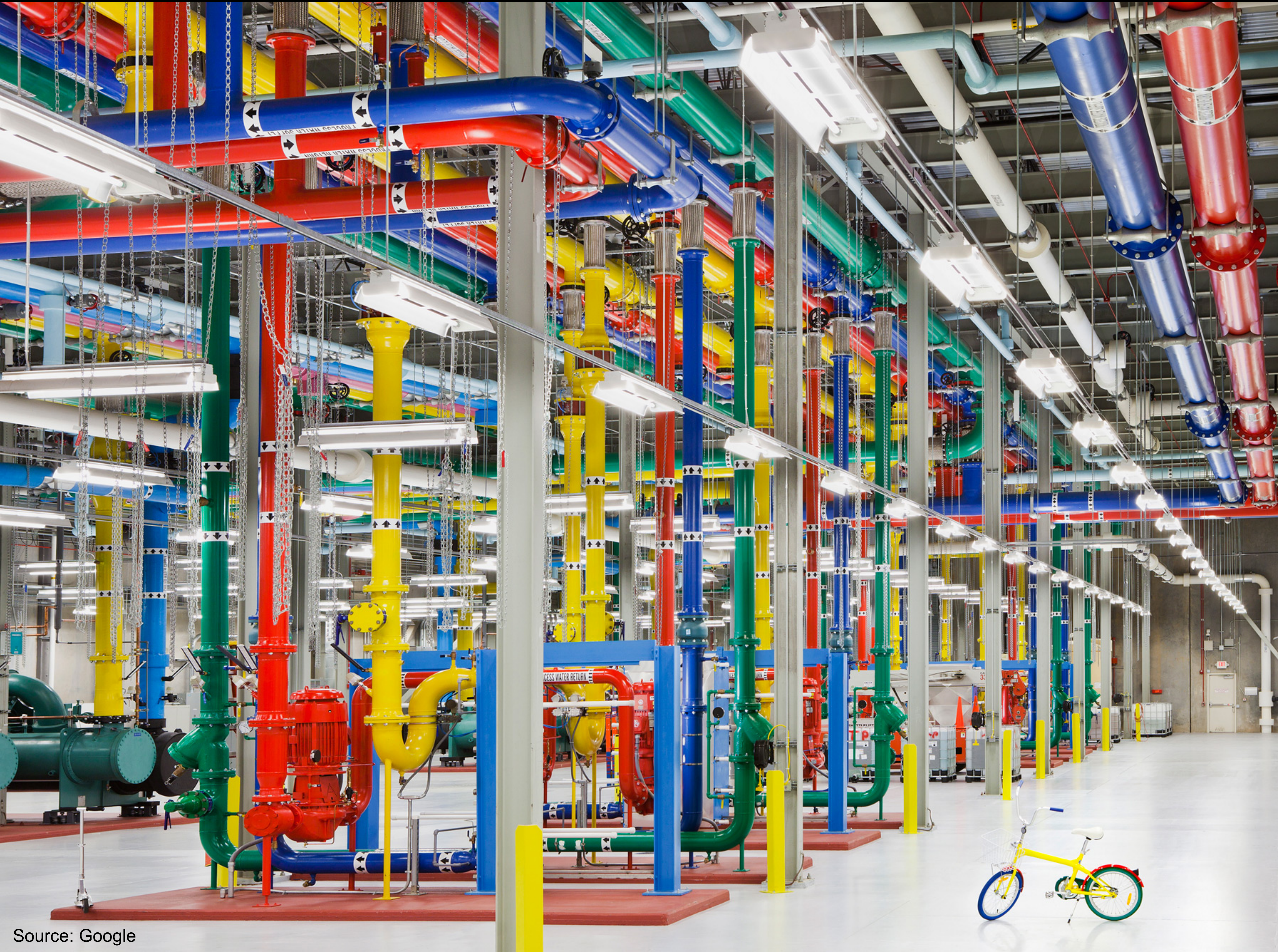
Source: Barroso and Urs Hölzle (2013)

Anatomy of a Datacenter

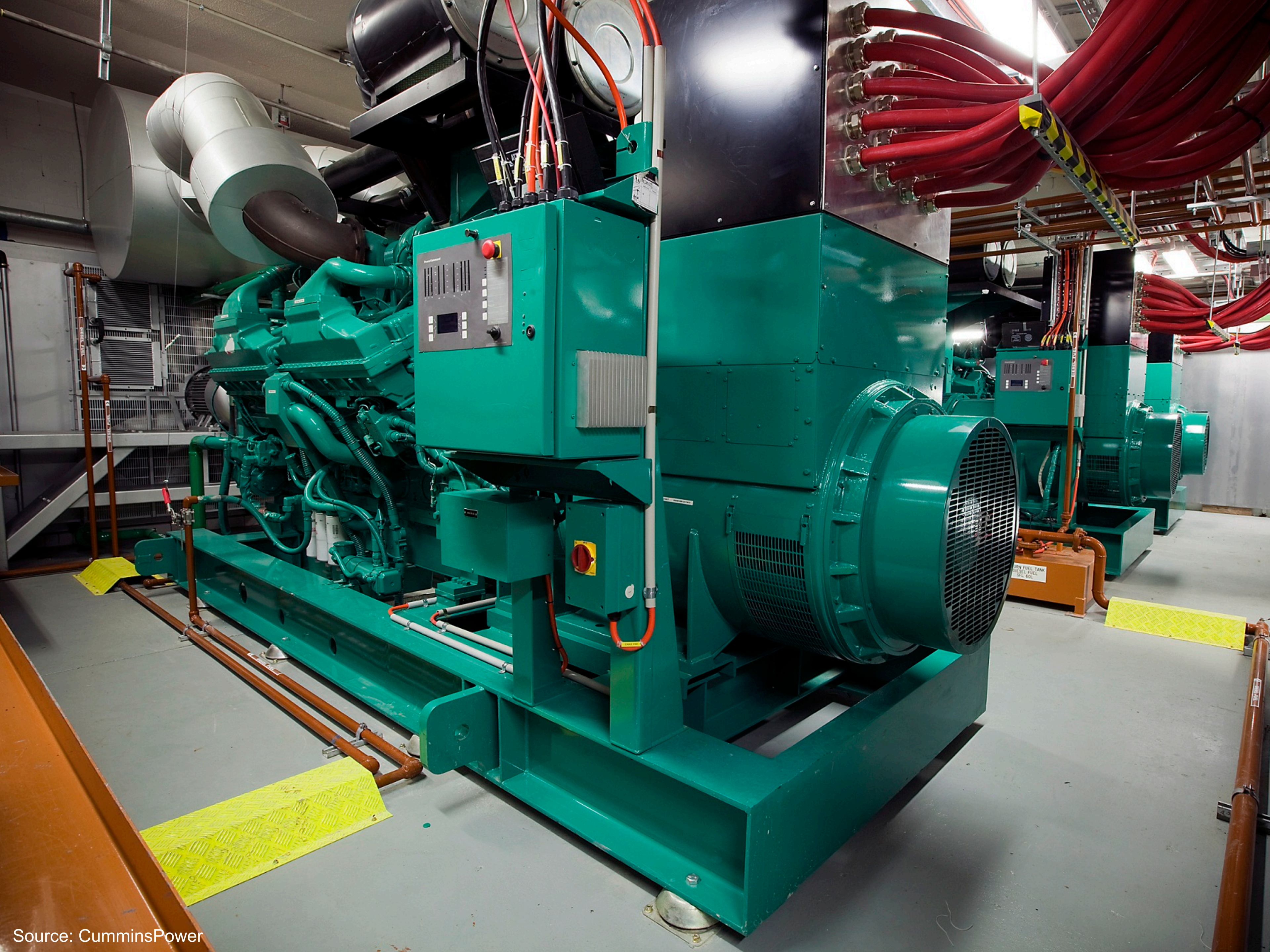


Source: Barroso and Urs Hölzle (2013)

Source: Google



Source: Google



Source: CumminsPower

Source: Google


Aside: How much is 30 MW?
Source: Google

The datacenter *is* the computer

- It's all about the right level of abstraction
  - Moving beyond the von Neumann architecture
  - What's the "instruction set" of the datacenter computer?
- Hide system-level details from the developers
  - No more race conditions, lock contention, etc.
  - No need to explicitly worry about reliability, fault tolerance, etc.
- Separating the *what* from the *how*
  - Developer specifies the computation that needs to be performed
  - Execution framework ("runtime") handles actual execution

"Big Ideas"

- Scale "out", not "up"
  - Limits of SMP and large shared-memory machines
- Move processing to the data
  - Cluster have limited bandwidth
- Process data sequentially, avoid random access
  - Seeks are expensive, disk throughput is reasonable
- Seamless scalability
  - From the mythical man-month to the tradable machine-hour

Scaling "up" vs. "out"

- No single machine is large enough
  - Smaller cluster of large SMP machines vs. larger cluster of commodity machines (e.g., 16 128-core machines vs. 128 16-core machines)
- Nodes need to talk to each other!
  - Intra-node latencies: ~100 ns
  - Inter-node latencies: ~100 μs
- Let's model communication overhead…

Source: analysis on this an subsequent slides from Barroso and Urs Hölzle (2009)

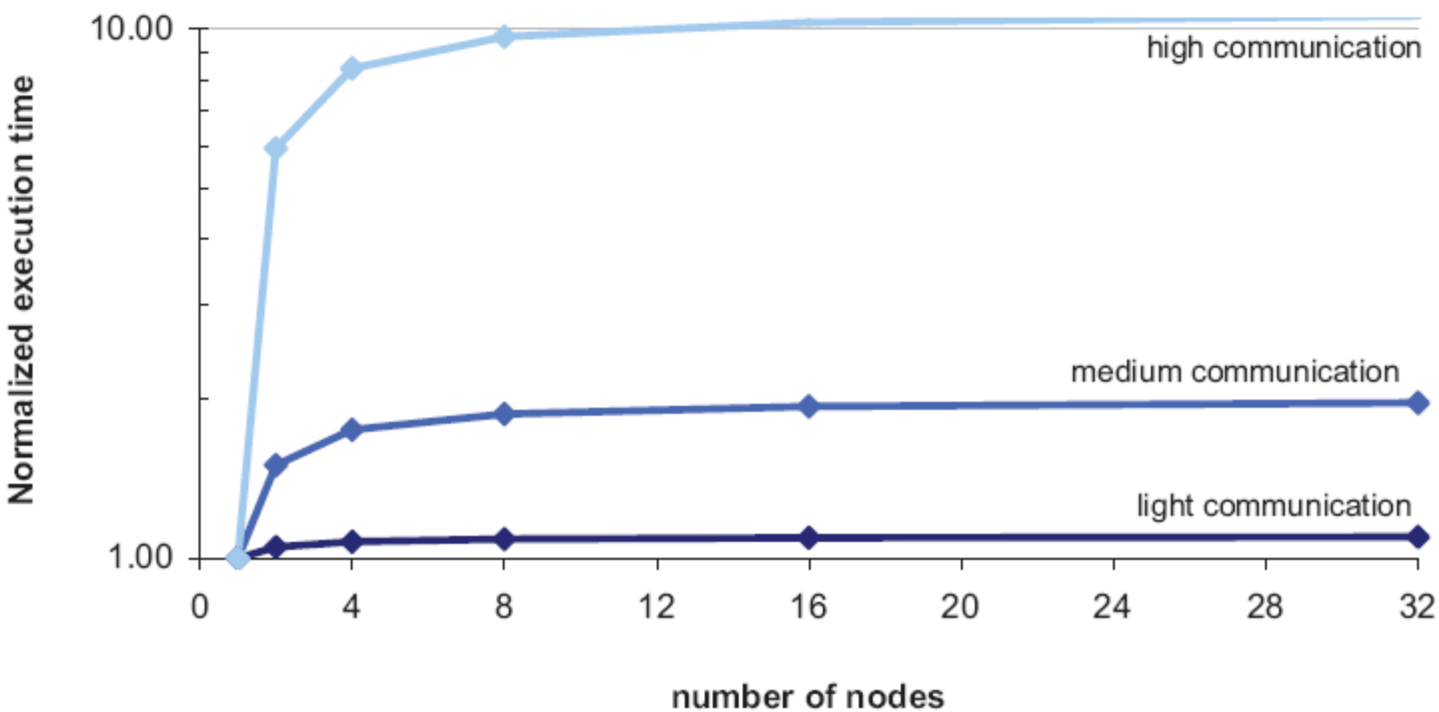
Modeling Communication Costs

- Simple execution cost model:
  - Total cost = cost of computation + cost to access global data
  - Fraction of local access inversely proportional to size of cluster
  - *n* nodes (ignore cores for now)

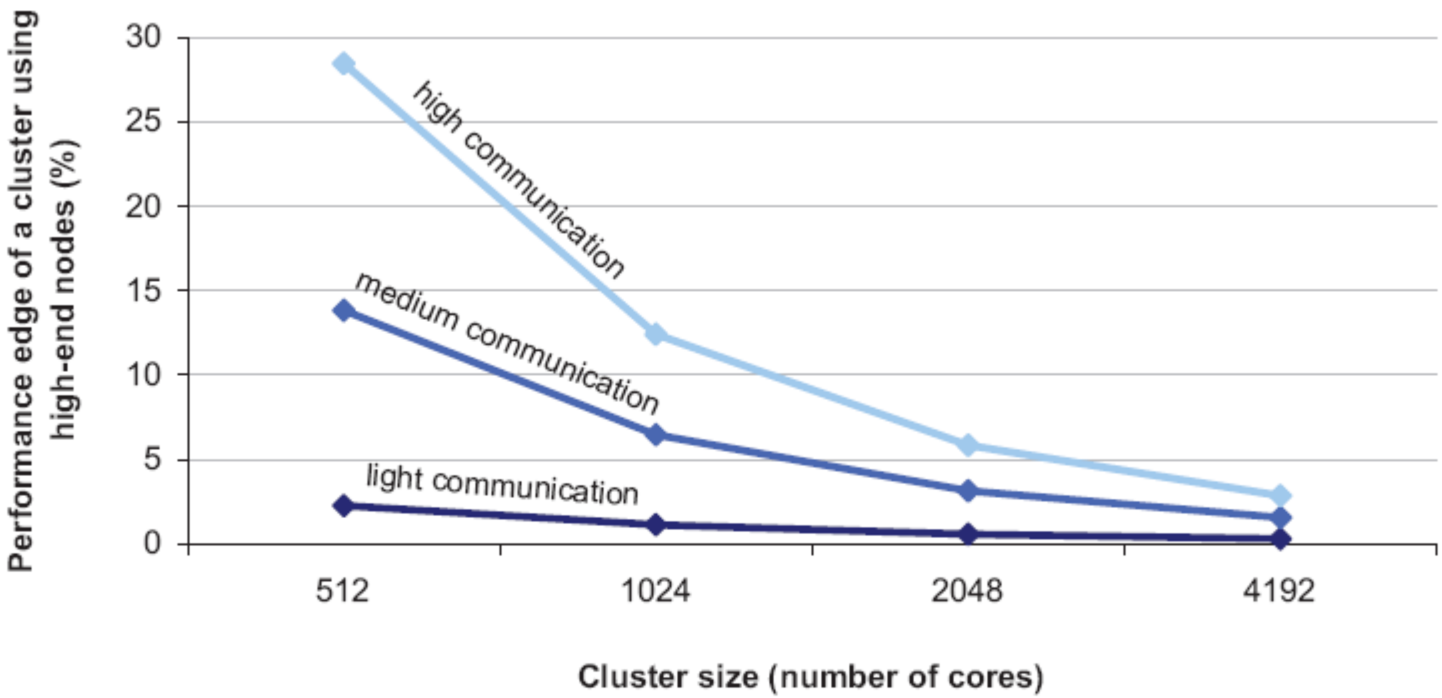
$$1 \text{ ms} + f \times [100 \text{ ns} \times (1/n) + 100\ \mu\text{s} \times (1 - 1/n)]$$

    - Light communication: $f = 1$
    - Medium communication: $f = 10$
    - Heavy communication: $f = 100$
- What are the costs in parallelization?

Cost of Parallelization



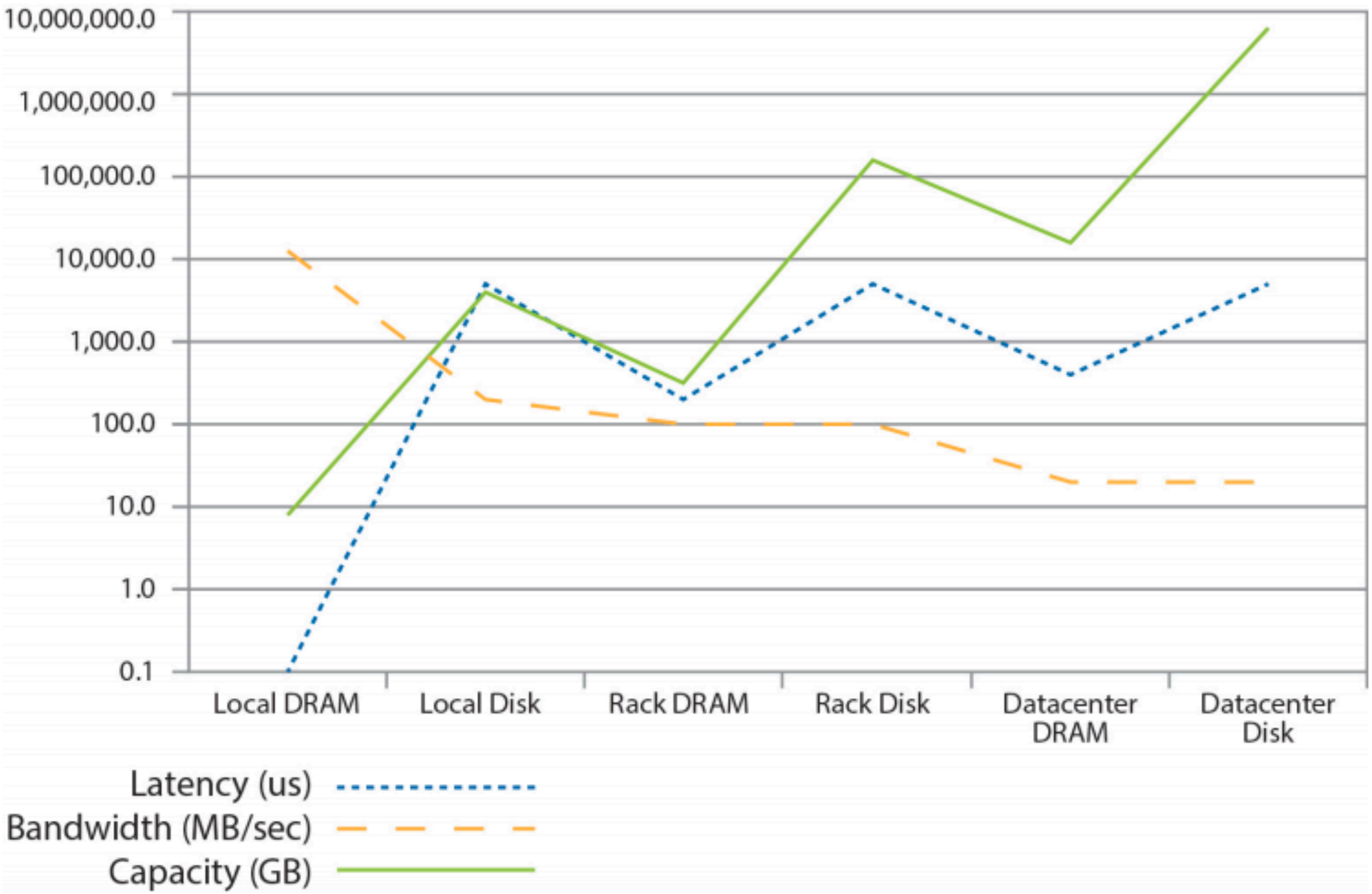
Advantages of scaling "up"



So why not?

Why does commodity beat exotic?

Moving Data Around



Source: Barroso and Urs Hölzle (2013)

Seeks vs. Scans

- Consider a 1 TB database with 100 byte records
  - We want to update 1 percent of the records
- Scenario 1: random access
  - Each update takes ~30 ms (seek, read, write)
  - 10^8 updates = ~35 days
- Scenario 2: rewrite all records
  - Assume 100 MB/s throughput
  - Time = 5.6 hours(!)
- Lesson: avoid random seeks!

Source: Ted Dunning, on Hadoop mailing list

Justifying the "Big Ideas"

- Scale "out", not "up"
  - Limits of SMP and large shared-memory machines
- Move processing to the data
  - Cluster have limited bandwidth
- Process data sequentially, avoid random access
  - Seeks are expensive, disk throughput is reasonable
- Seamless scalability
  - From the mythical man-month to the tradable machine-hour


MapReduce
Source: Google

Typical Big Data Problem

- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

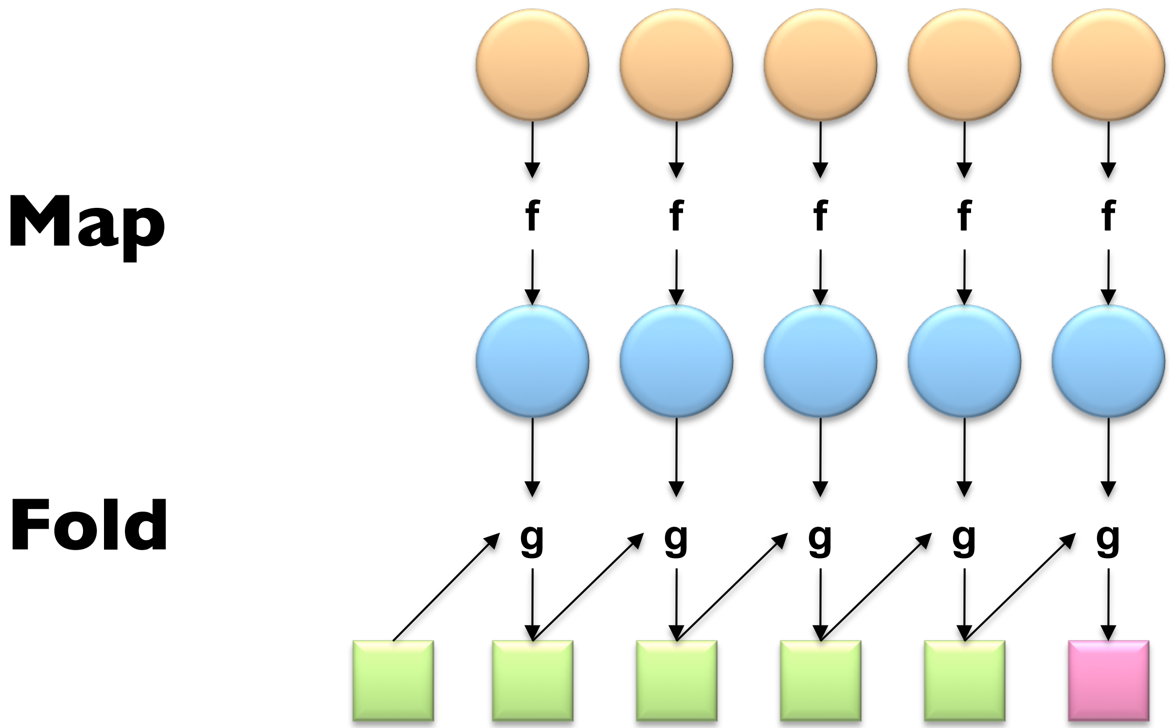
Map

Reduce

Key idea: provide a functional abstraction for these two operations

(Dean and Ghemawat, OSDI 2004)

Roots in Functional Programming



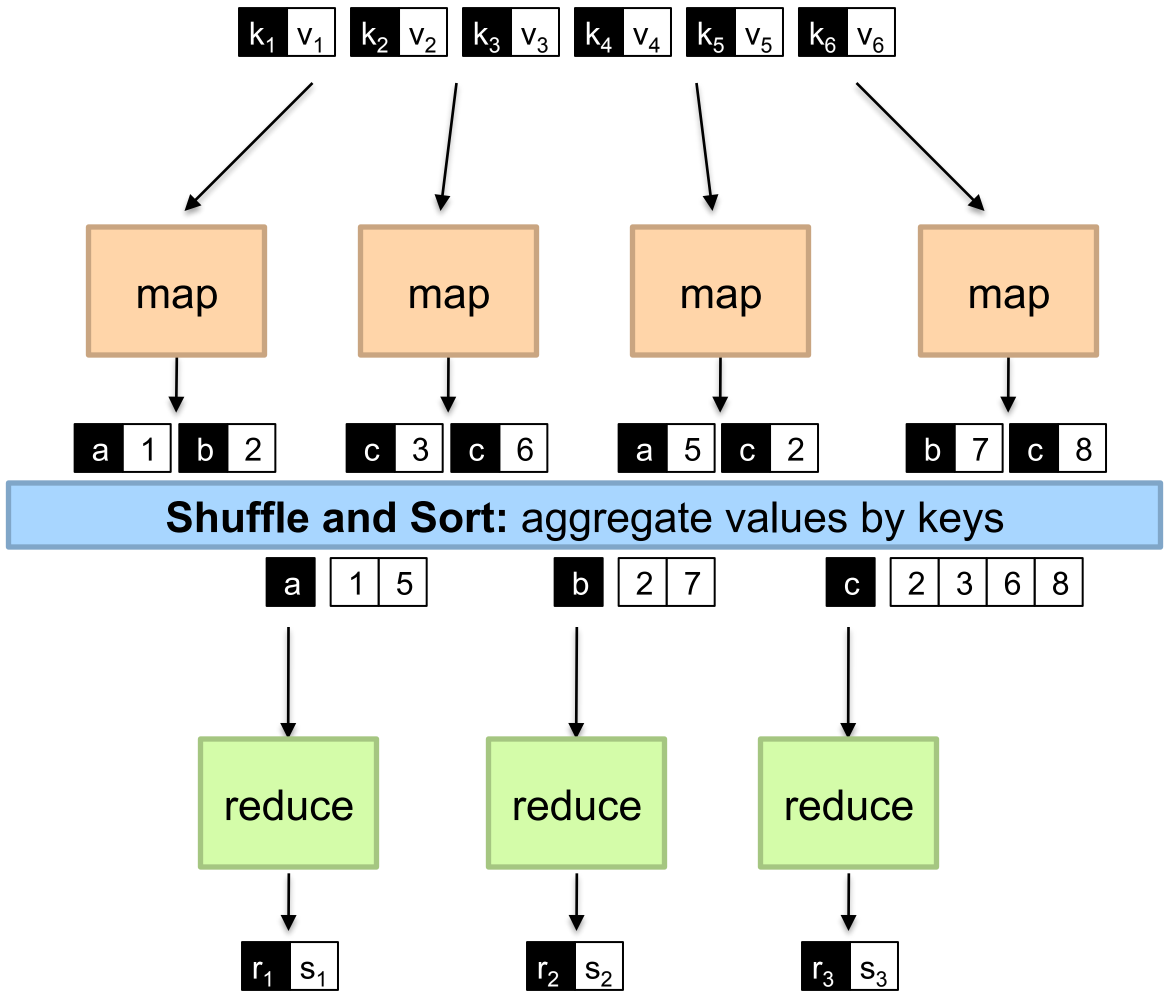
MapReduce

- Programmers specify two functions:

  map $(k_1, v_1) \rightarrow [<k_2, v_2>]$

  reduce $(k_2, [v_2]) \rightarrow [<k_3, v_3>]$

  - All values with the same key are sent to the same reducer
- The execution framework handles everything else…



k_1 v_1
k_2 v_2
k_3 v_3
k_4 v_4
k_5 v_5
k_6 v_6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
r_1 s_1
r_2 s_2
r_3 s_3

MapReduce

- Programmers specify two functions:

  map (k, v) → <k’, v’>*

  reduce (k’, v’) → <k’, v’>*

  - All values with the same key are sent to the same reducer
- The execution framework handles everything else…

What’s “everything else”?

MapReduce "Runtime"

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles "data distribution"
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed FS (later)

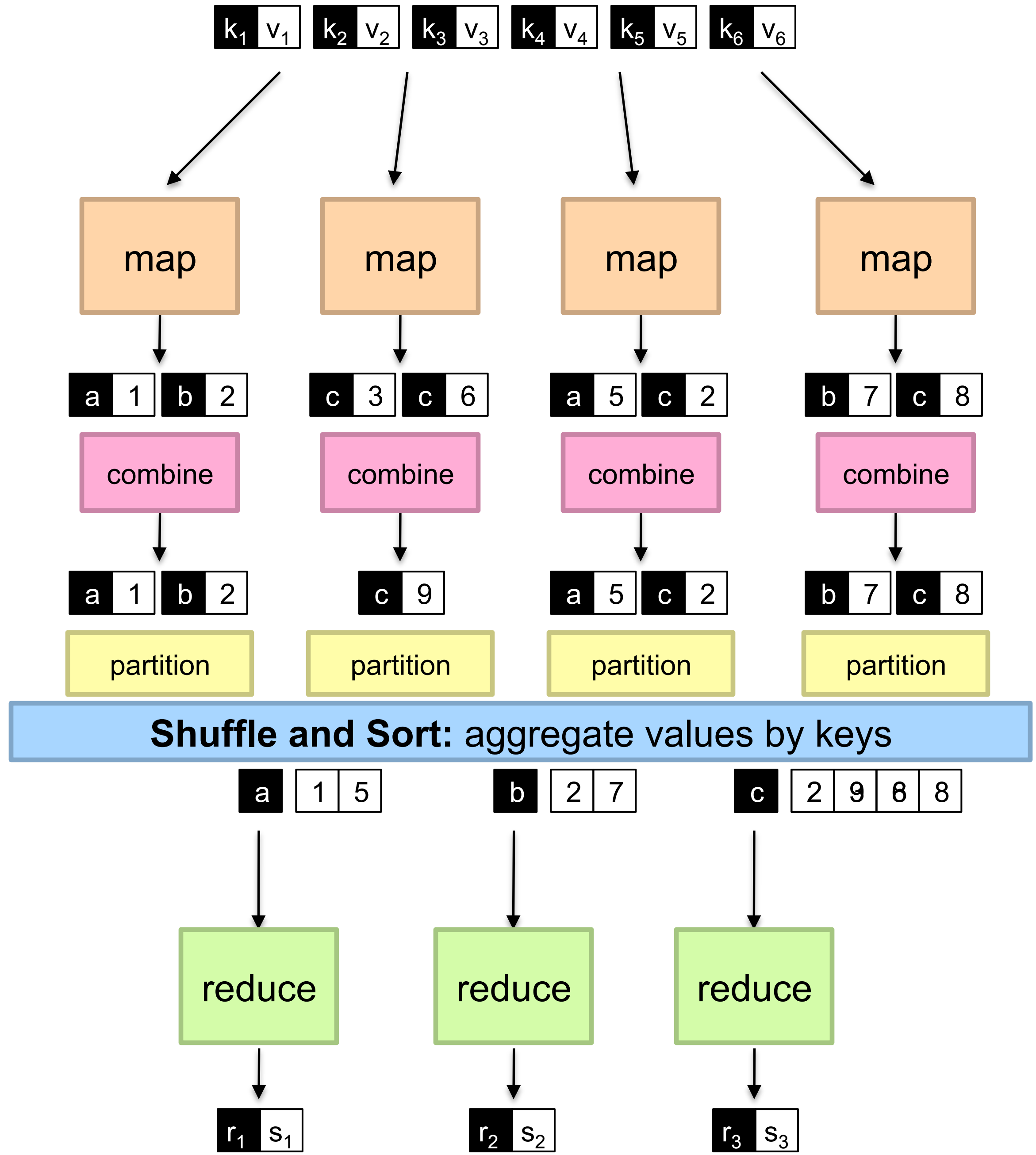
MapReduce

- Programmers specify two functions:

  map (k, v) → <k’, v’>*
  reduce (k’, v’) → <k’, v’>*
  - All values with the same key are reduced together
- The execution framework handles everything else…
- Not quite…usually, programmers also specify:

  partition (k’, number of partitions) → partition for k’
  - Often a simple hash of the key, e.g., hash(k’) mod n
  - Divides up key space for parallel reduce operations

  combine (k’, v’) → <k’, v’>*
  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce network traffic


k1 v1 k2 v2 k3 v3 k4 v4 k5 v5 k6 v6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
combine
combine
combine
combine
a 1 b 2
c 9
a 5 c 2
b 7 c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 8 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

Two more details…

- Barrier between map and reduce phases
  - But we can begin copying intermediate data earlier
- Keys arrive at each reducer in sorted order
  - No enforced ordering *across* reducers

“Hello World”: Word Count

```
Map(String docid, String text):
    for each word w in text:
        Emit(w, 1);

Reduce(String term, Iterator<Int> values):
    int sum = 0;
    for each v in values:
        sum += v;
    Emit(term, value);
```

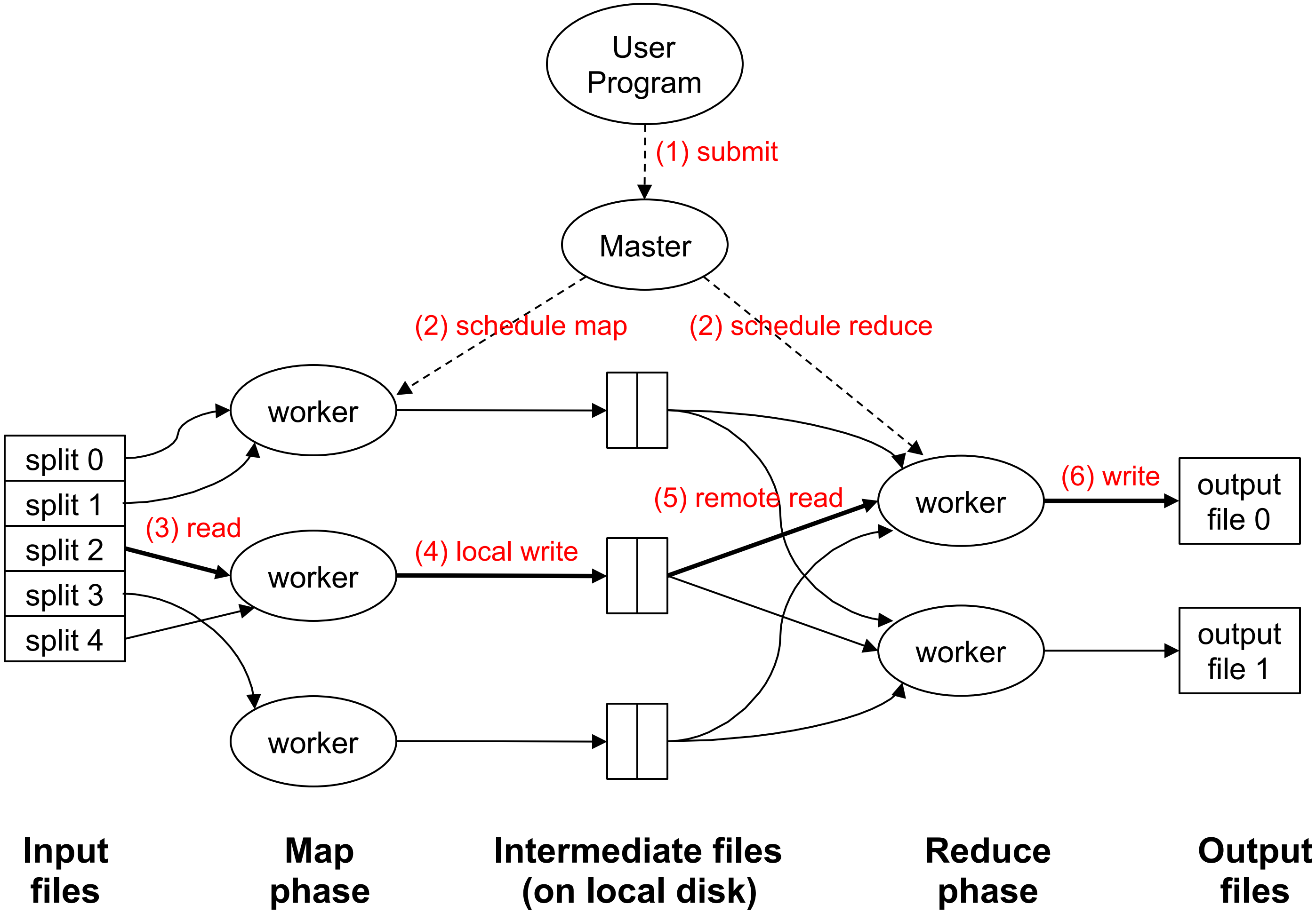
MapReduce can refer to…

- The programming model
- The execution framework (aka "runtime")
- The specific implementation

Usage is usually clear from context!

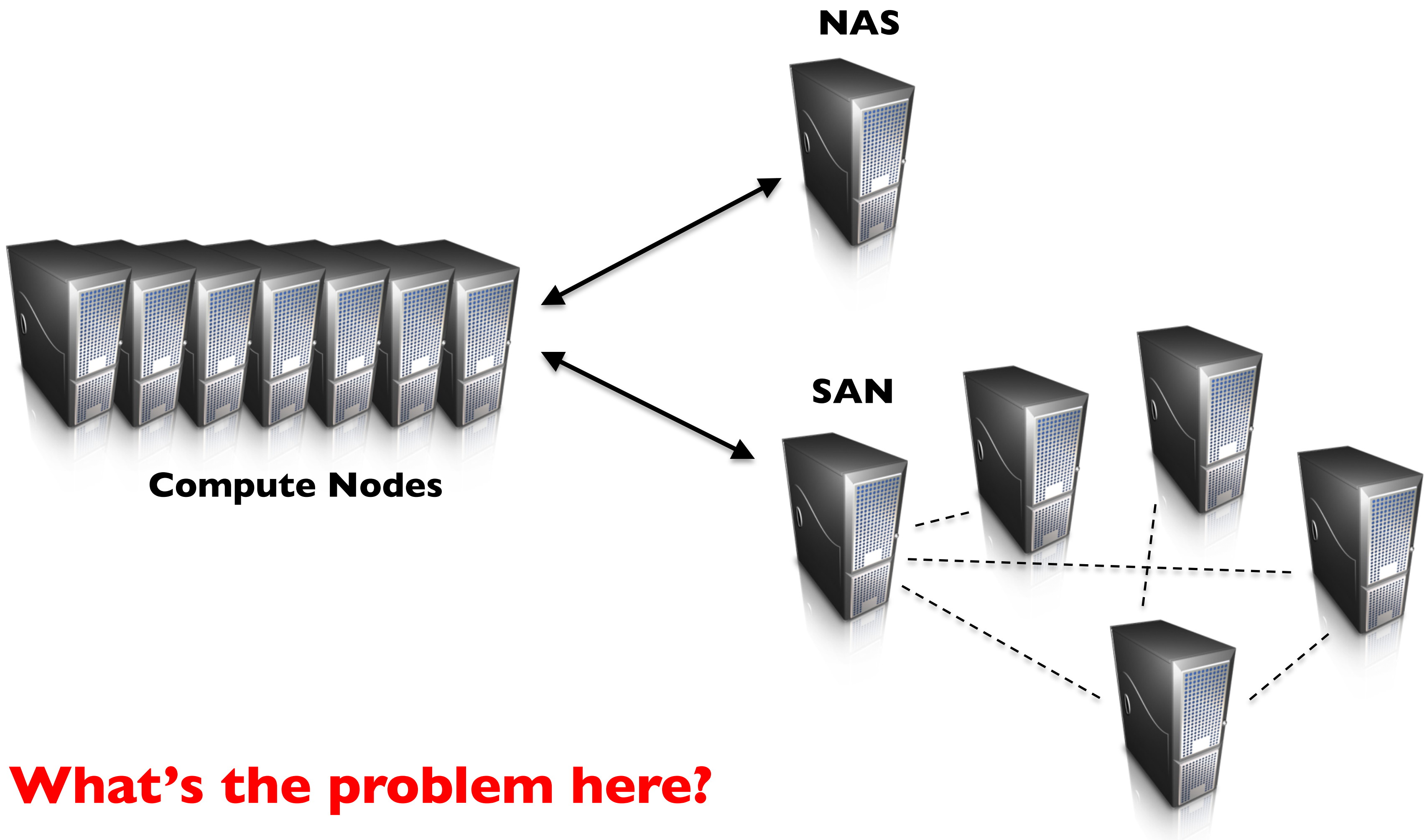
MapReduce Implementations

- Google has a proprietary implementation in C++
  - Bindings in Java, Python
- Hadoop is an open-source implementation in Java
  - Development led by Yahoo, now an Apache project
  - Used in production at Yahoo, Facebook, Twitter, LinkedIn, Netflix, …
  - The *de facto* big data processing platform
  - Large and expanding software ecosystem
- Lots of custom research implementations
  - For GPUs, cell processors, etc.



Adapted from (Dean and Ghemawat, OSDI 2004)

How do we get data to the workers?



What's the problem here?

Distributed File System

- Don't move data to workers… move workers to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- Why?
  - (Perhaps) not enough RAM to hold all the data in memory
  - Disk access is slow, but disk throughput is reasonable
- A distributed file system is the answer
  - GFS (Google File System) for Google's MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop

GFS: Assumptions

- Commodity hardware over “exotic” hardware
  - Scale “out”, not “up”
- High component failure rates
  - Inexpensive commodity components fail all the time
- “Modest” number of huge files
  - Multi-gigabyte files are common, if not encouraged
- Files are write-once, mostly appended to
  - Perhaps concurrently
- Large streaming reads over random access
  - High sustained throughput over low latency

GFS: Design Decisions

- Files stored as chunks
  - Fixed size (64MB)
- Reliability through replication
  - Each chunk replicated across 3+ chunkservers
- Single master to coordinate access, keep metadata
  - Simple centralized management
- No data caching
  - Little benefit due to large datasets, streaming reads
- Simplify the API
  - Push some of the issues onto the client (e.g., data layout)

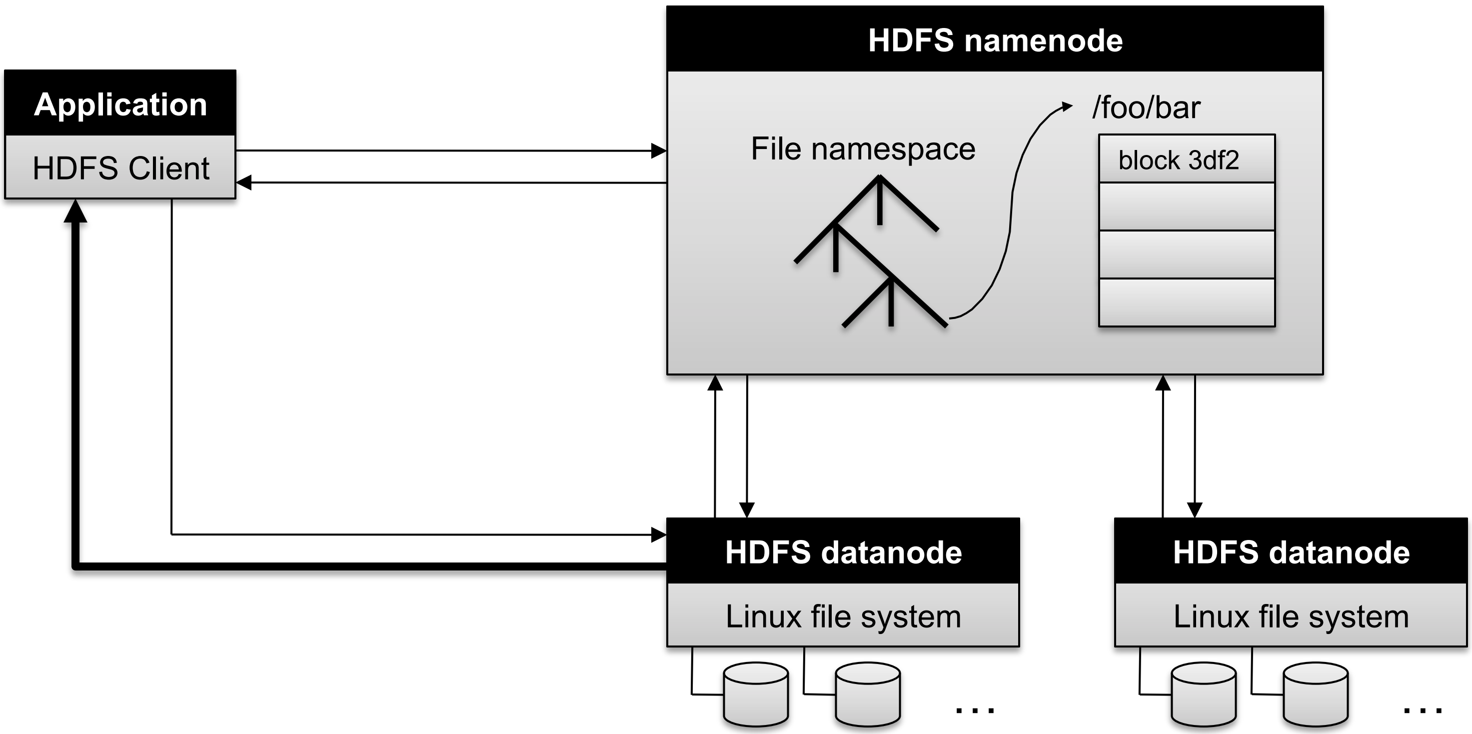
HDFS = GFS clone (same basic ideas)

From GFS to HDFS

- Terminology differences:
  - GFS master = Hadoop namenode
  - GFS chunkservers = Hadoop datanodes
- Differences:
  - Different consistency model for file appends
  - Implementation
  - Performance

For the most part, we'll use Hadoop terminology…

HDFS Architecture

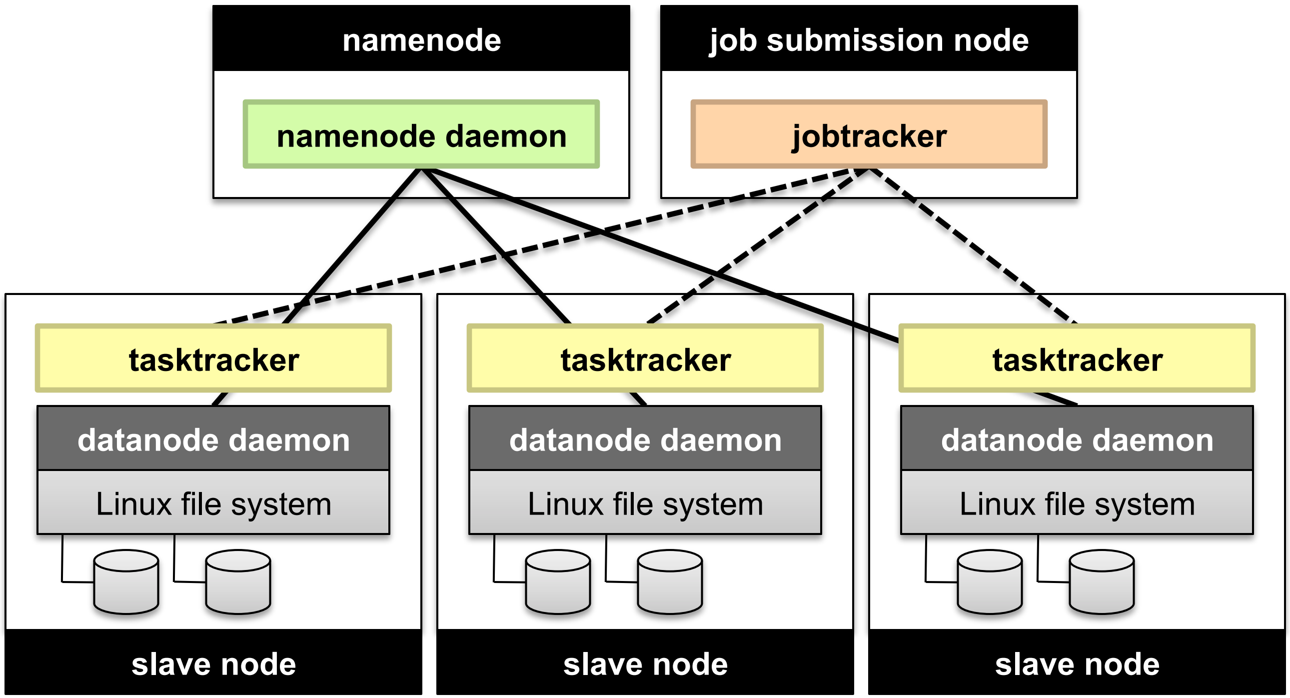


Adapted from (Ghemawat et al., SOSP 2003)

Namenode Responsibilities

- Managing the file system namespace:
  - Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc.
- Coordinating file operations:
  - Directs clients to datanodes for reads and writes
  - No data is moved through the namenode
- Maintaining overall health:
  - Periodic communication with the datanodes
  - Block re-replication and rebalancing
  - Garbage collection

Putting everything together…



(Not Quite… We'll come back to YARN later)

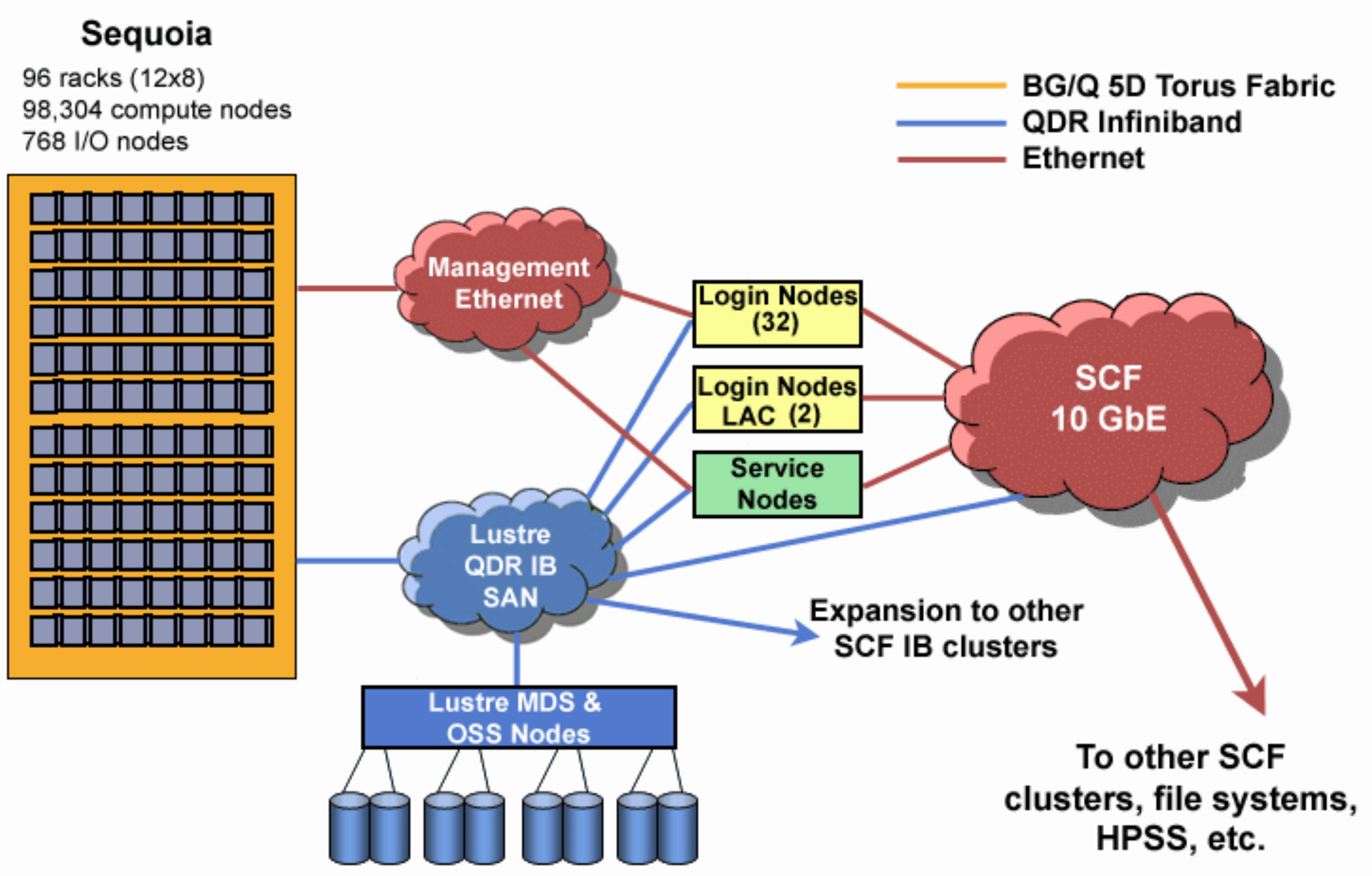
Sequoia

16.32 PFLOPS

98,304 nodes with 1,572,864 million cores

1.6 petabytes of memory

7.9 MWatts total power


Sequoia
96 racks (12x8)
98,304 compute nodes
768 I/O nodes
BG/Q 5D Torus Fabric
QDR Infiniband
Ethernet
Management Ethernet
Login Nodes (32)
Login Nodes LAC (2)
Service Nodes
SCF 10 GbE
Lustre QDR IB SAN
Expansion to other SCF IB clusters
Lustre MDS & OSS Nodes
To other SCF clusters, file systems, HPSS, etc.

Source: LLNL

Source: Wikipedia (Japanese rock garden)