Mesos Background

https://web.stanford.edu/class/cs114/

Philip Levis
Stanford University
pal@cs.stanford.edu

Background for Mesos

Computer Systems

Computer systems focus on *abstraction*

- Software (operating systems): I have some hardware resources, what software API do I provide?
  - File systems: disk blocks become files
  - Spark: network of computers runs thousands of small tasks
- Hardware (architecture): I have digital logic, what mechanisms do I provide to software?
  - An instruction set defines how arithmetic and memory work
  - A bus defines how hardware devices can access each other

A good abstraction is easy to use and efficient while being simple to implement

Distributed Systems

Collections of computers connected over a network

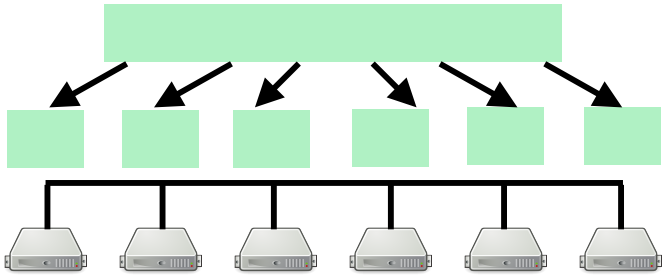
- No computer has a perfect view of system state
- Different computers see things differently
- Networks fail: even if rarely, it'll happen, and your system needs to be able to handle it

Three major considerations

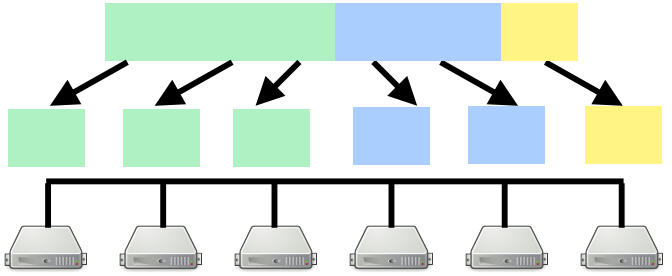
- Performance
- Scalability
- Correctness

Parallel vs. Distributed

Parallel: multiple processors/cores execute equal, homogenous parts of a computation



Distributed: multiple cores on a network execute different parts of a computation



Value of Parallelism

More cores: if you can keep them busy with useful work, it's faster

More network: if machines can load data in parallel, it's faster

More memory: you have more space for data

A large number of smaller computers is less expensive than a few huge computers

More is Better

A large number of smaller _____ is less expensive than a few huge _____

- Computers
- Disks
- Servers
- Displays
- Network switches

Price performance tradeoff

Size	Cost	Cost/TB
500GB	$87	$174
1TB	$109	$109
2TB	$209	$105
4TB	$729	$183

Samsung 860 EVO SSDS

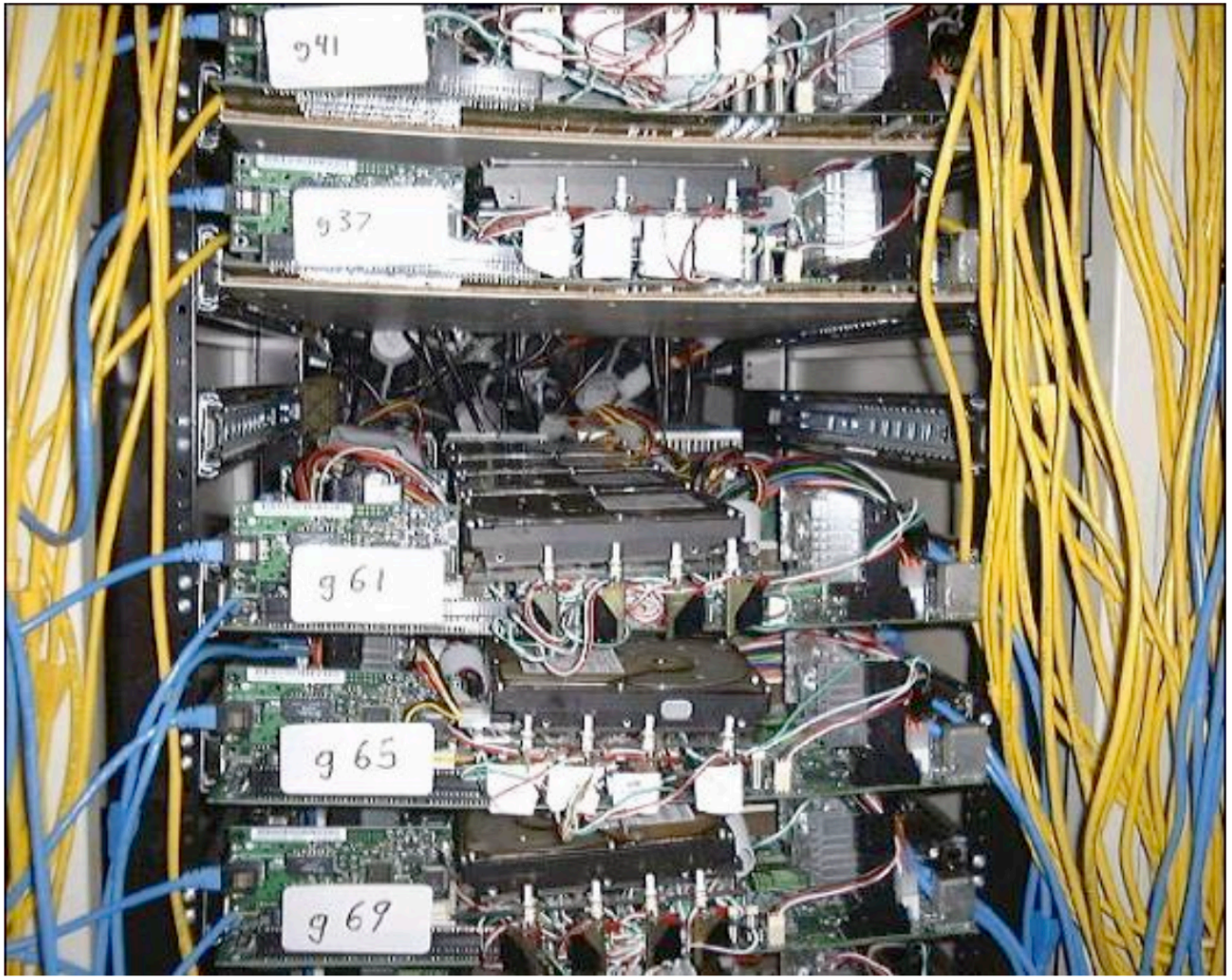
Rise of the Datacenter

Pictures stolen from Jeff Dean's 2010 talk at Stanford

“Google” Circa 1997 (google.stanford.edu)

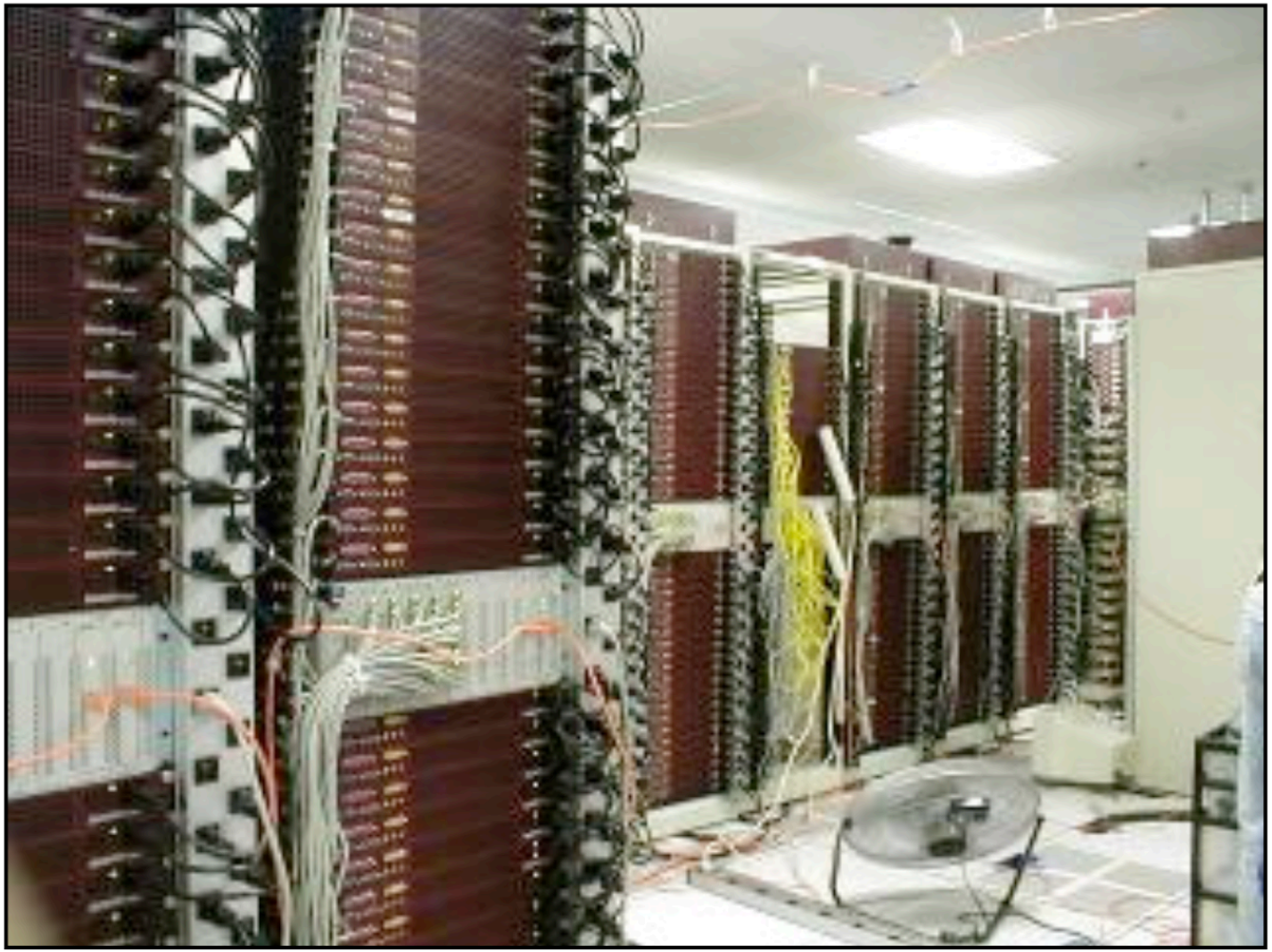
Google

“Corkboards” (1999)



Google

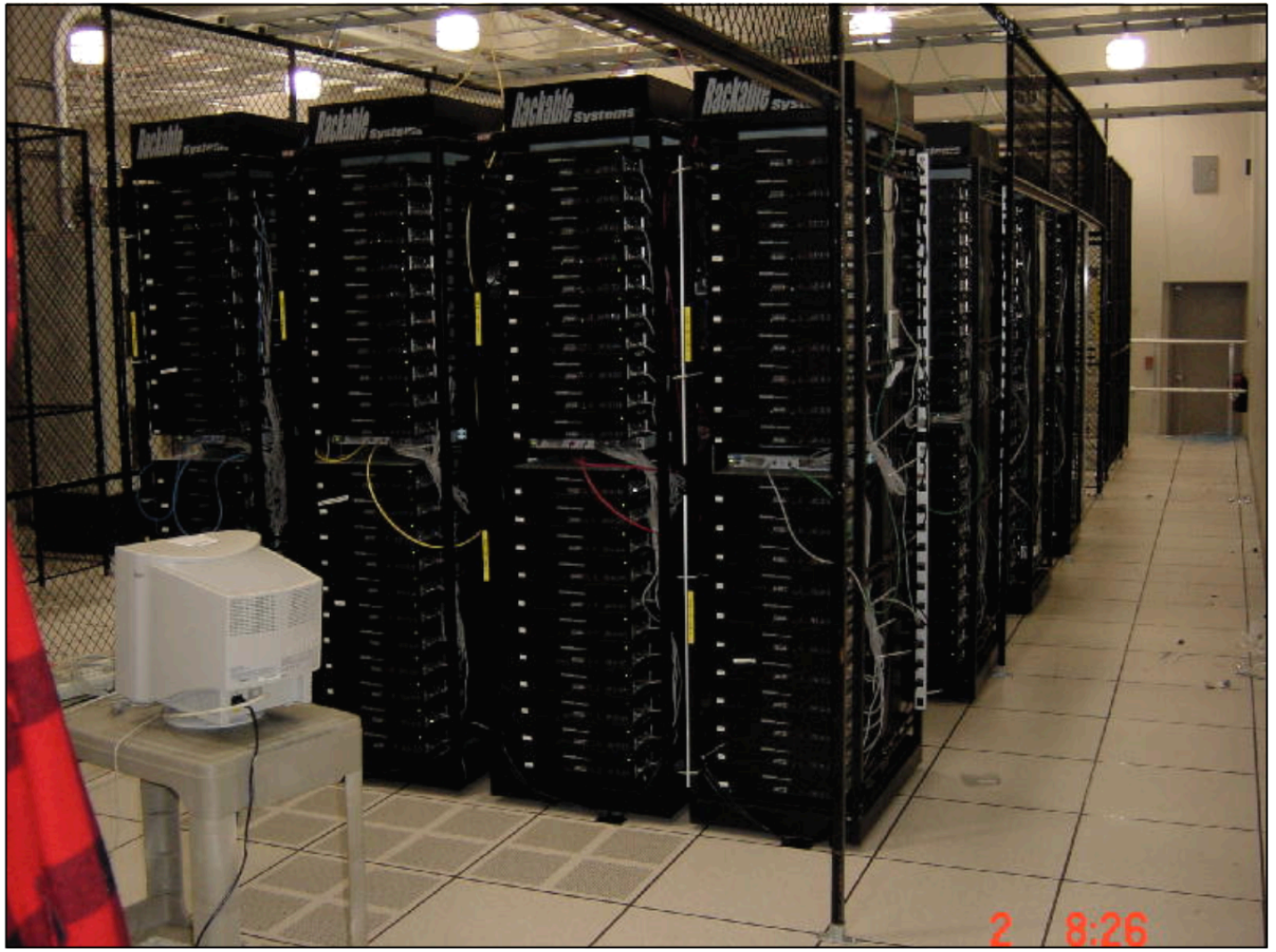
Google Data Center (2000)



Google

2003

Google Data Center (3 days later)



Google

MapReduce (2004)

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to eas-

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to

MapReduce Model

MapReduce is a programming model and associated runtime to manage the program

- Programming model: the abstraction
- Associated runtime: the system

Intended for processing very large (can't fit in memory on lots of nodes) data sets

- E.g., computing an index of the web

Basic idea: take data and split it into pieces (e.g., 16-64GB), process pieces in parallel

Programming Model

```
map(k1, v1) -> list(k2, v2)
```

```
reduce(k2, list(v2)) -> list(v2)
```

User writes these two functions (plus an optional partitioner): MapReduce does the rest

Compose sequences of map and reduce into complex pipelines

Example: Word Count

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;

  for each v in values
    result += ParseInt(v);
  Emit(AsString(result));
```

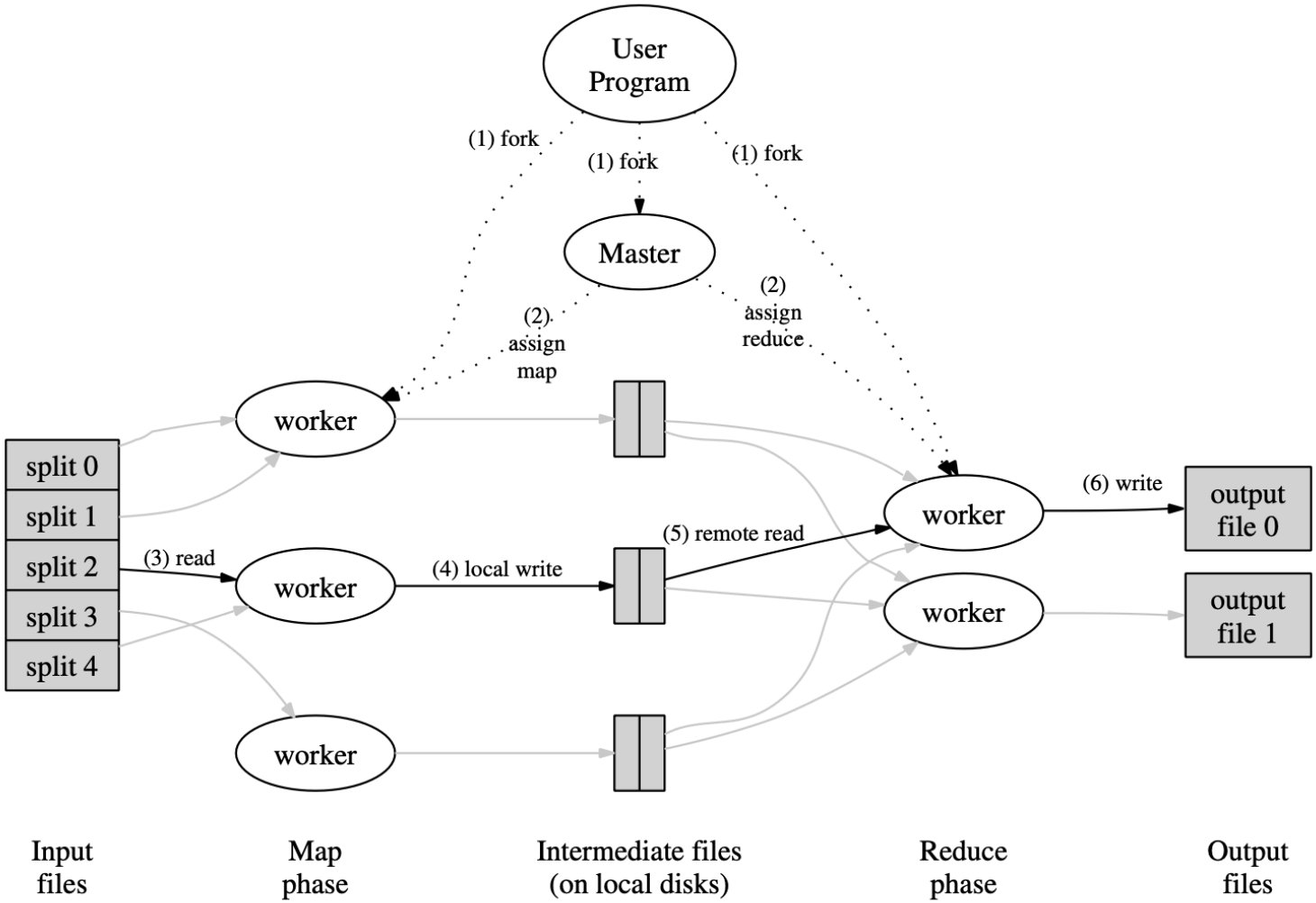
Example: Inverted Index

```
map(String doc, String value):
  // doc: document ID
 // value: document contents
  for each word w in value:
    EmitIntermediate(w, key);

reduce(String word, Iterator list):
  // word: a word
  // list: a list of document IDs
  list.sort().unique()

  Emit(AsString(word, list));
```

MapReduce Runtime



Operations are Idempotent

If a worker fails, just send the work to another node

- E.g., if a map task on a key fails, rerun it on another node

Stragglers: what if some tasks just take much longer (slow nodes)?

- If a task takes too long just spawn another copy
- Variance, fraction, lots of approaches

Early datacenters were unreliable

- "For example, during one MapReduce operation, network maintenance on a running cluster was causing groups of 80 machines at a time to become unreachable for several minutes."

Datacenter Computing

MapReduce opened the door to easily computing on huge volumes of data (can't even fit in the memory of 1000s of machines)

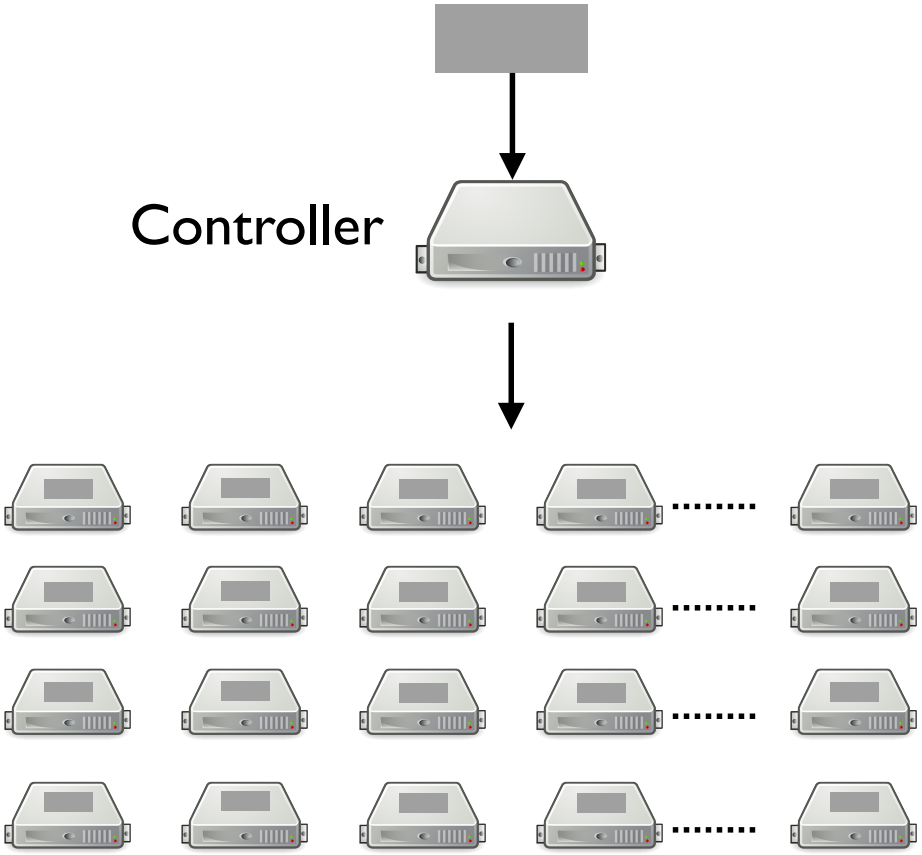
Frameworks started to proliferate: want to run many different kinds of jobs on a cluster

How do you schedule lots of heterogenous data-parallel jobs on a cluster?

Different Kinds of Frameworks

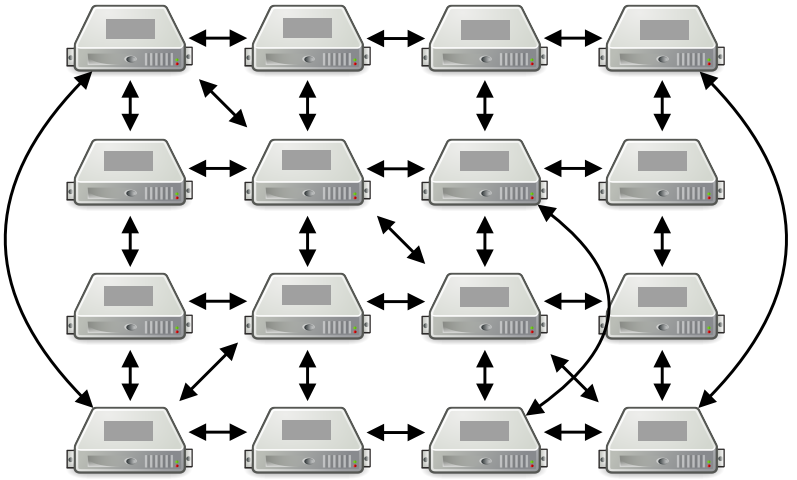
Centralized

MapReduce, Spark, Hadoop

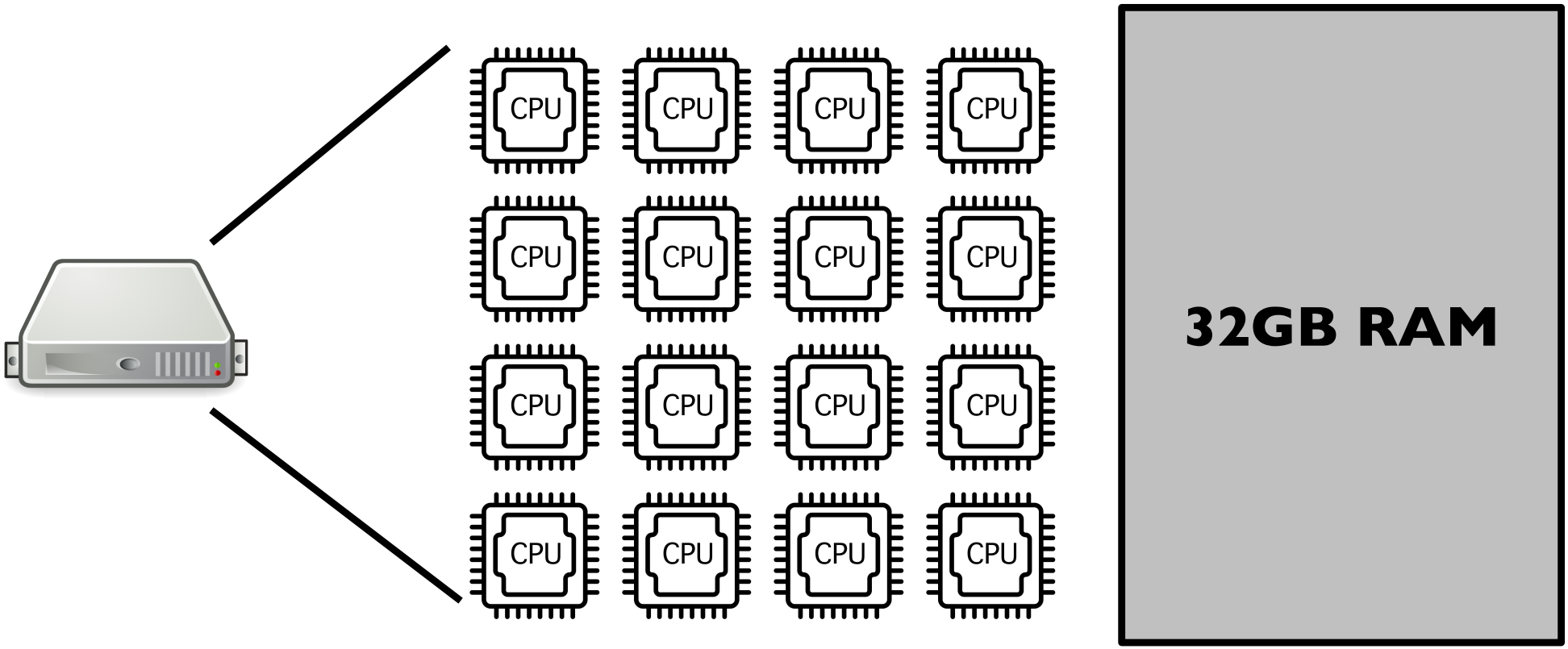


Decentralized

Dryad, MPI, Torque



Finer-grained Sharing



Fault Tolerance

At scale, things fail (computers, programs, disks)

Early in datacenter computing (e.g., MapReduce), failures were common

- Much less common today, driven by AWS reliability
- Tension with early Google Compute Platform offerings: they designed all their systems to be robust to failures, but consumer applications aren't (e.g., an Oracle license server)

Soft State vs. Hard State

Hard state: information stored on a computer that's required for forward progress and correctness

- Regenerating it might require a lot of recompilation
- Example: your user account on a website

Soft state: information that improves performance, but isn't required

- Losing it might prevent certain optimizations but doesn't stop forward progress
- Example: cookies/passwords stored on your computer

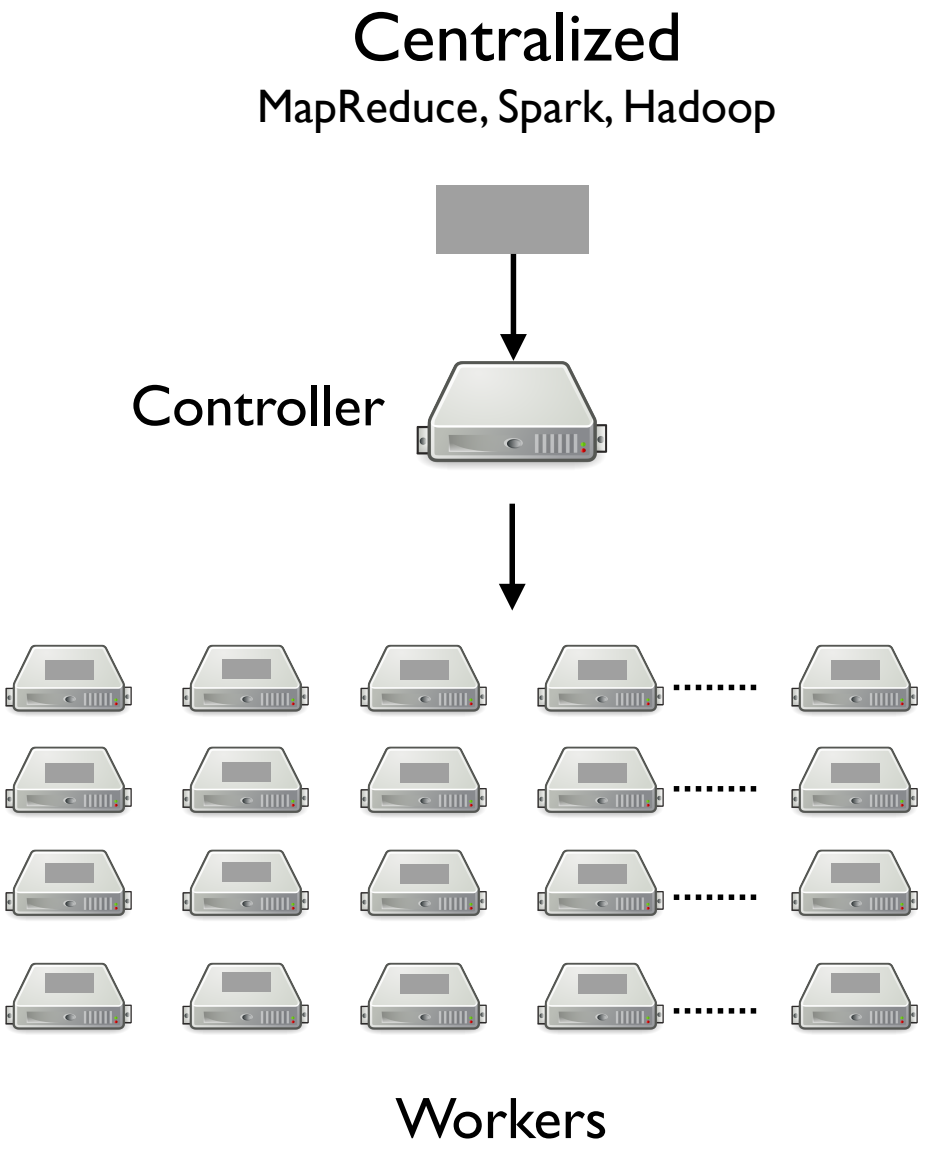
Dataflow Systems

Systems like MapReduce (and Spark) use a centralized controller: it sends commands to nodes, telling them what operations to execute

Dataflow systems do not have this explicit control channel

- Instead, computations run when they receive the data they need
- Plasticine is a dataflow architecture

Different Kinds of Frameworks



Controller sends commands:

Node 15, execute task B on data π

Node 18, execute task S on data ∂

Centralization makes debugging, fault tolerance, and management much easier

Easy to adapt to failures, stragglers, and dynamically schedule load

Sending commands is an overhead if tasks are short (not true in MapReduce)

Different Kinds of Frameworks

Program installed on each node

Tasks on each node execute when they receive the data they need

Execute task B on data of type π

Execute task S on data of type ∂+μ

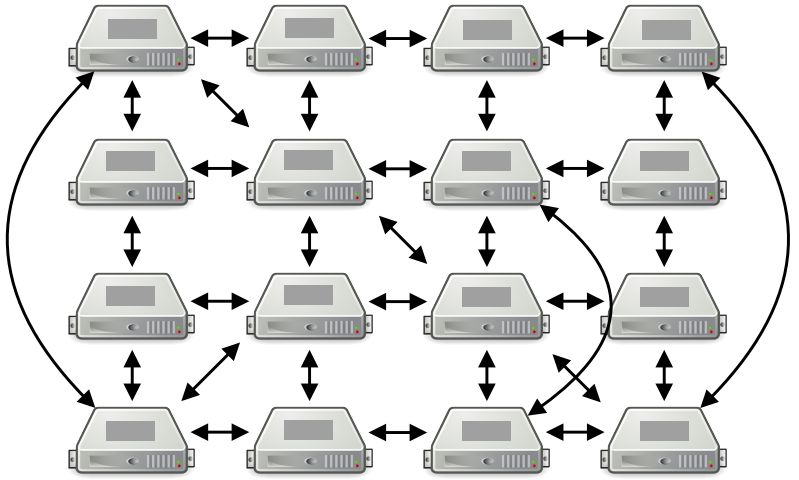
Executing tasks produces new data items, sent to other nodes

The job sets up these data flow edges between nodes

Low overhead, but rescheduling is expensive

Decentralized

Dryad, MPI, Torque



Workers

Spark

Designed by our very own Matei Zaharia!

- With a lot of help from others

Observation: MapReduce has low CPU utilization because it reads/writes all data to disk

Spark: cache data in memory when possible, track how data is generated so it can be regenerated if lost

- Plus, add a simple shell-like interface to allow interactive use and programming
- Key abstraction: partitioning data by keys for parallelism

Framework Performance

Spark can be an order of magnitude faster than MapReduce

- Disk is very very slow compared to memory

Early versions were still 50x slower than C++

- Lots of unnecessary overheads: copies, Java, Scala

Modern versions are ~3x slower than C++

- Acceptable performance cost for the productivity increase of writing Java/shell code

Performance Bottlenecks

As framework performance increases, the central controller becomes a bottleneck: it can't dispatch tasks fast enough: modern Spark jobs run on 10s of cores, not 1,000

Execution Templates: Caching Control Plane Decisions for Strong Scaling of Data Analytics

Omid Mashayekhi Hang Qu Chinmayee Shah Philip Levis
Stanford University

Abstract

Control planes of cloud frameworks trade off between scheduling granularity and performance. Centralized systems schedule at task granularity, but only schedule a few thousand tasks per second. Distributed systems schedule hundreds of thousands of tasks per second but changing the schedule is costly.

We present *execution templates*, a control plane abstraction that can schedule hundreds of thousands of tasks per second while supporting fine-grained, per-task scheduling decisions. Execution templates leverage a program's repetitive control flow to cache blocks of frequently-executed tasks. Executing a task in a template

Figure 1: The control plane is a bottleneck in modern analytics workloads. Increasingly parallelizing logistic regression on 100GB of data with Spark 2.0's MLlib reduces computation time (black bars) but control overhead outstrip these gains, *increasing* completion time.