

CS 5450

MapReduce

Vitaly Shmatikov

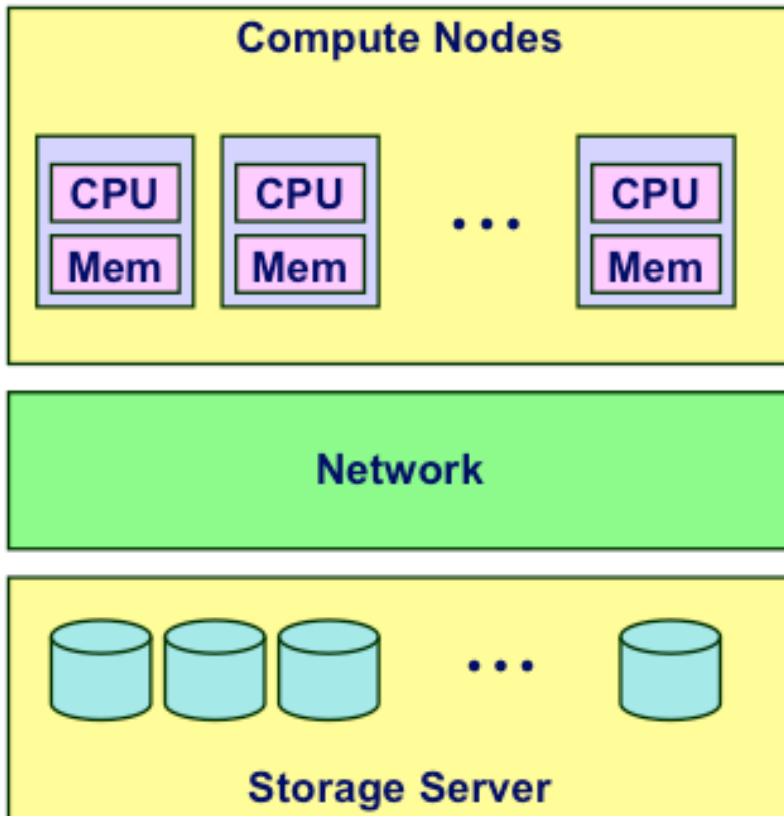
BackRub (Google), 1997



NEC Earth Simulator, 2002

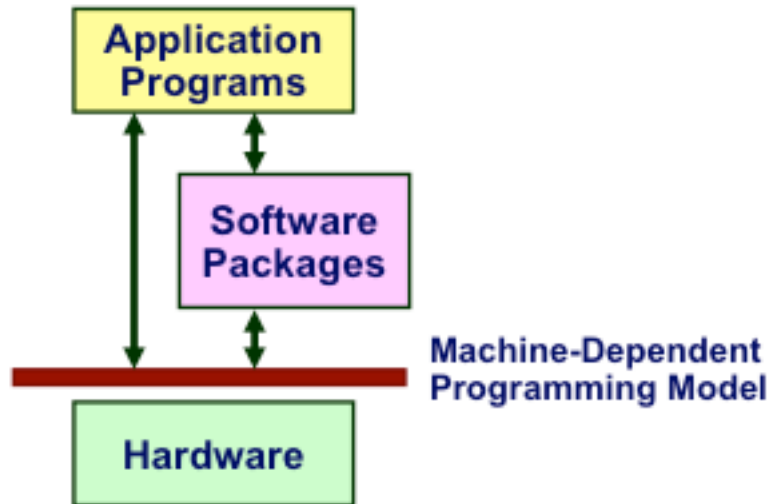


Conventional HPC Machine



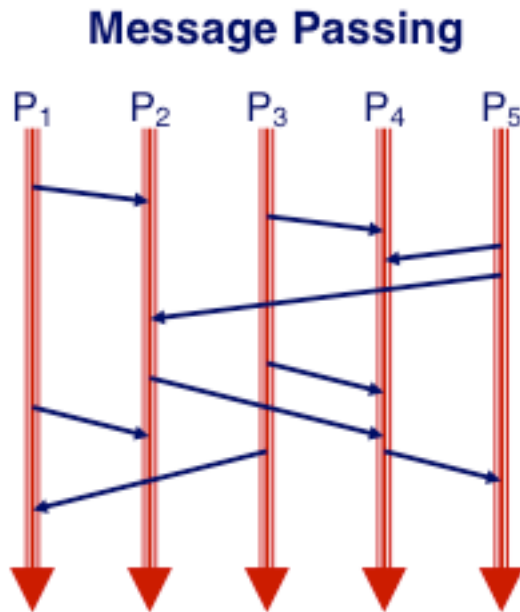
- ◆ Compute nodes
 - High-end processors
 - Lots of RAM
- ◆ Network
 - Specialized
 - Very high performance
- ◆ Storage server
 - RAID disk array

HPC Programming Model



- ◆ Programs described at a very low level
 - Detailed control of processing and communication
- ◆ Rely on small number of software packages
 - Written by specialists
 - Limits classes of problems and solution methods

Typical HPC Operation



◆ Characteristics

- Long-lived processes
- Make use of spatial locality
- Hold all data in memory (no disk access)
- High-bandwidth communication

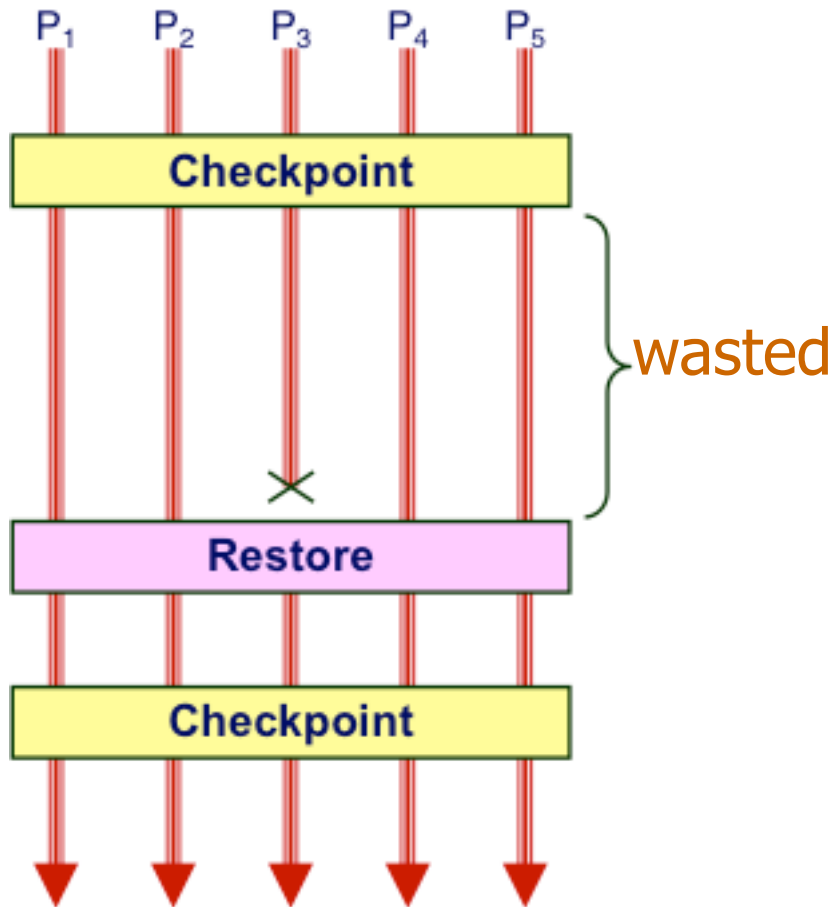
◆ Strengths

- High utilization of resources
- Effective for many scientific applications

◆ Weaknesses

- Requires careful tuning of applications to resources
- Intolerant of any variability

HPC Fault Tolerance



◆ Checkpoint

- Periodically store state of all processes
- Significant I/O traffic

◆ Restore after failure

- Reset state to last checkpoint
- All intervening computation wasted

◆ Performance scaling

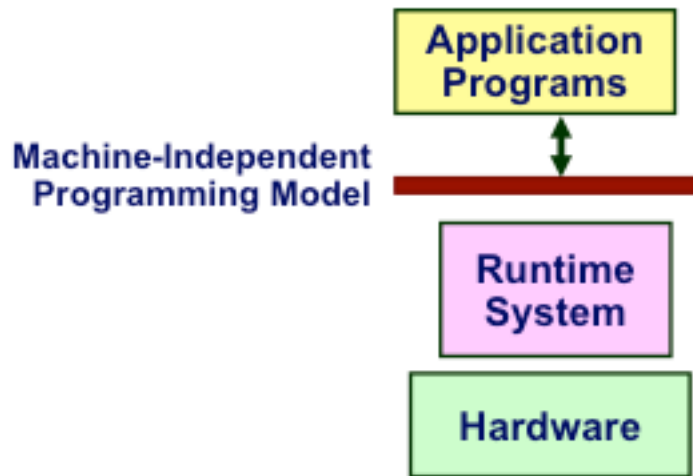
- Very sensitive to the number of failing components

Google Data Center, 2016

Dozens of these!



Ideal Cluster Programming Model



- ◆ Applications written in terms of high-level operations on the data
- ◆ Runtime system controls scheduling, load balancing

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 easily utilize the resources of a large distributed system.

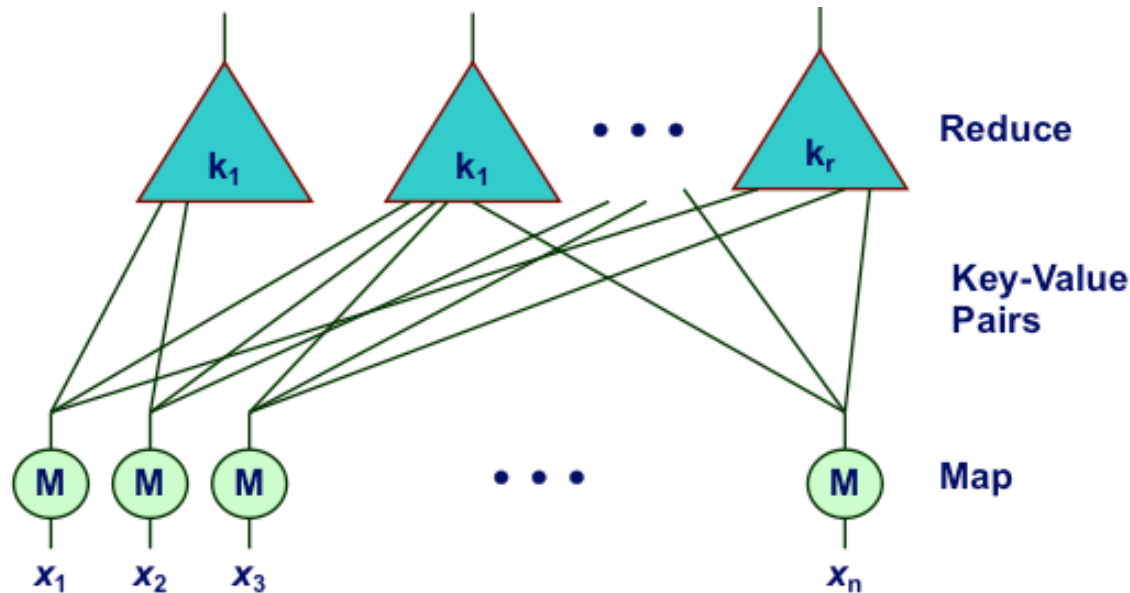
Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers

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 compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to paral-



MapReduce Programming Model



- ◆ Map computation across many data objects
 - For example, 10^{10} webpages
- ◆ Aggregate results in many different ways
- ◆ System deals with resource allocation and availability

Programming in Lisp

Computing the sum of squares

- `(map square '(1 2 3 4))`

Processes each record sequentially and independently

Output: `(1 4 9 16)`

- `(reduce + '(1 4 9 16))`

Computes `(+ 16 (+ 9 (+ 4 1)))`

Output: 30

Application: Word Count

```
SELECT count(word) FROM data  
GROUP BY word
```

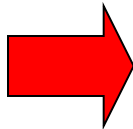
Partial Aggregation

1. In parallel, each worker computes word counts from individual files
2. Collect results, wait until all finished
3. Merge intermediate output
4. Compute word count on merged intermediates

Map

Process individual records to generate
(key, value) pairs

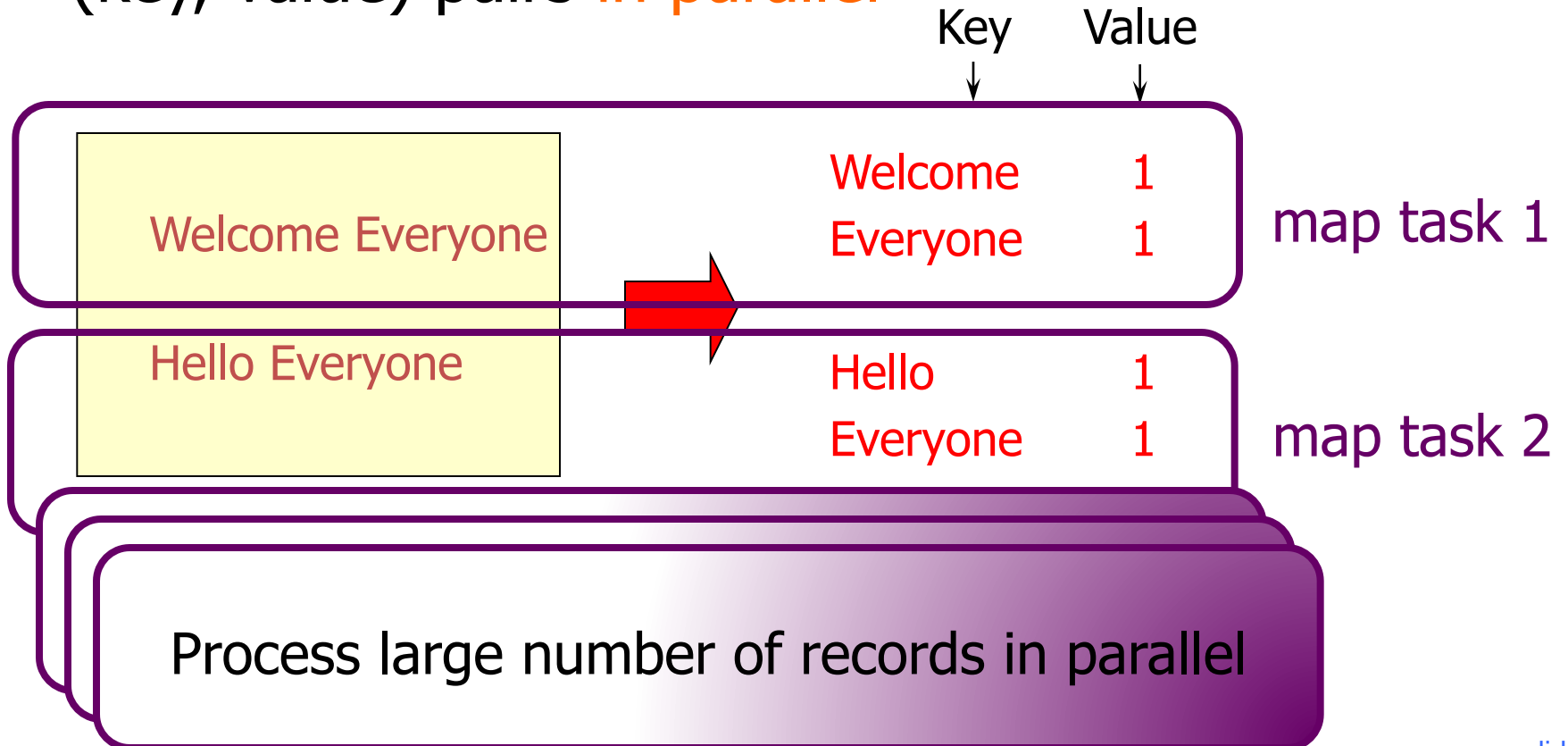
Welcome Everyone
Hello Everyone



Key ↓	Value ↓
Welcome	1
Everyone	1
Hello	1
Everyone	1

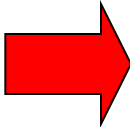
Parallel Map

Process individual records to generate (key, value) pairs **in parallel**



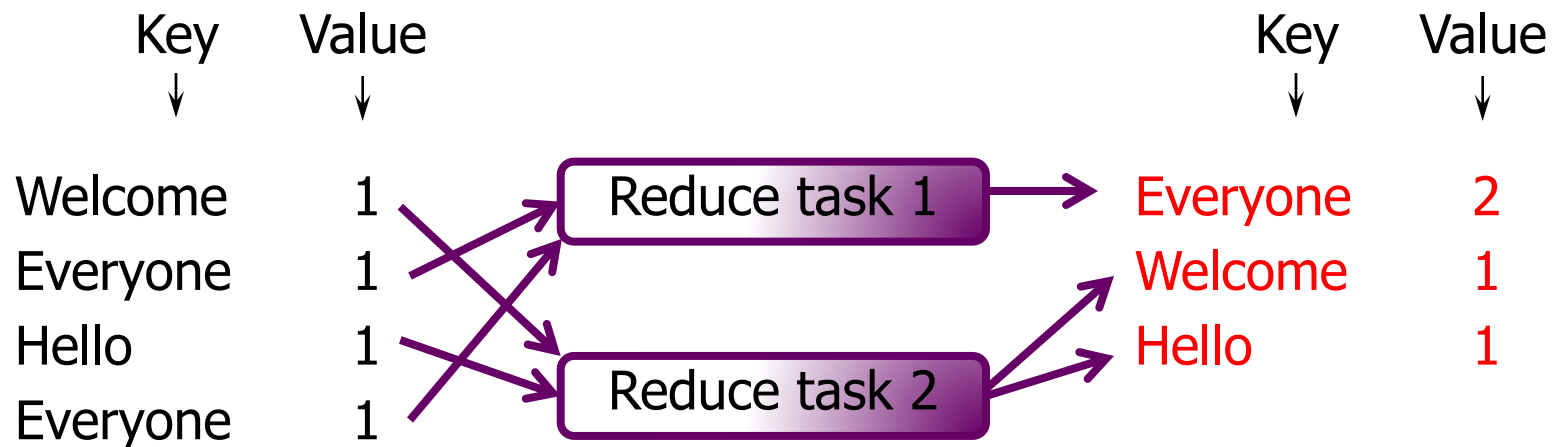
Reduce

Merge all intermediate values per key

Key ↓	Value ↓		Key ↓	Value ↓
Welcome	1		Everyone	2
Everyone	1		Welcome	1
Hello	1		Hello	1
Everyone	1			

Partitioning

Merge all intermediate values **in parallel**:
partition keys, assign each key to one Reduce



MapReduce API

`map(key, value) -> list(<k', v'>)`

- Apply function to (key, value) pair and produces set of intermediate pairs

`reduce(key, list<value>) -> <k', v'>`

- Applies aggregation function to values
- Outputs result

MapReduce WordCount

```
map(key, value):  
    for each word w in value:  
        EmitIntermediate(w, "1");
```

```
reduce(key, list(values):  
    int result = 0;  
    for each v in values:  
        result += ParseInt(v);  
    Emit(AsString(result));
```

Optimizations

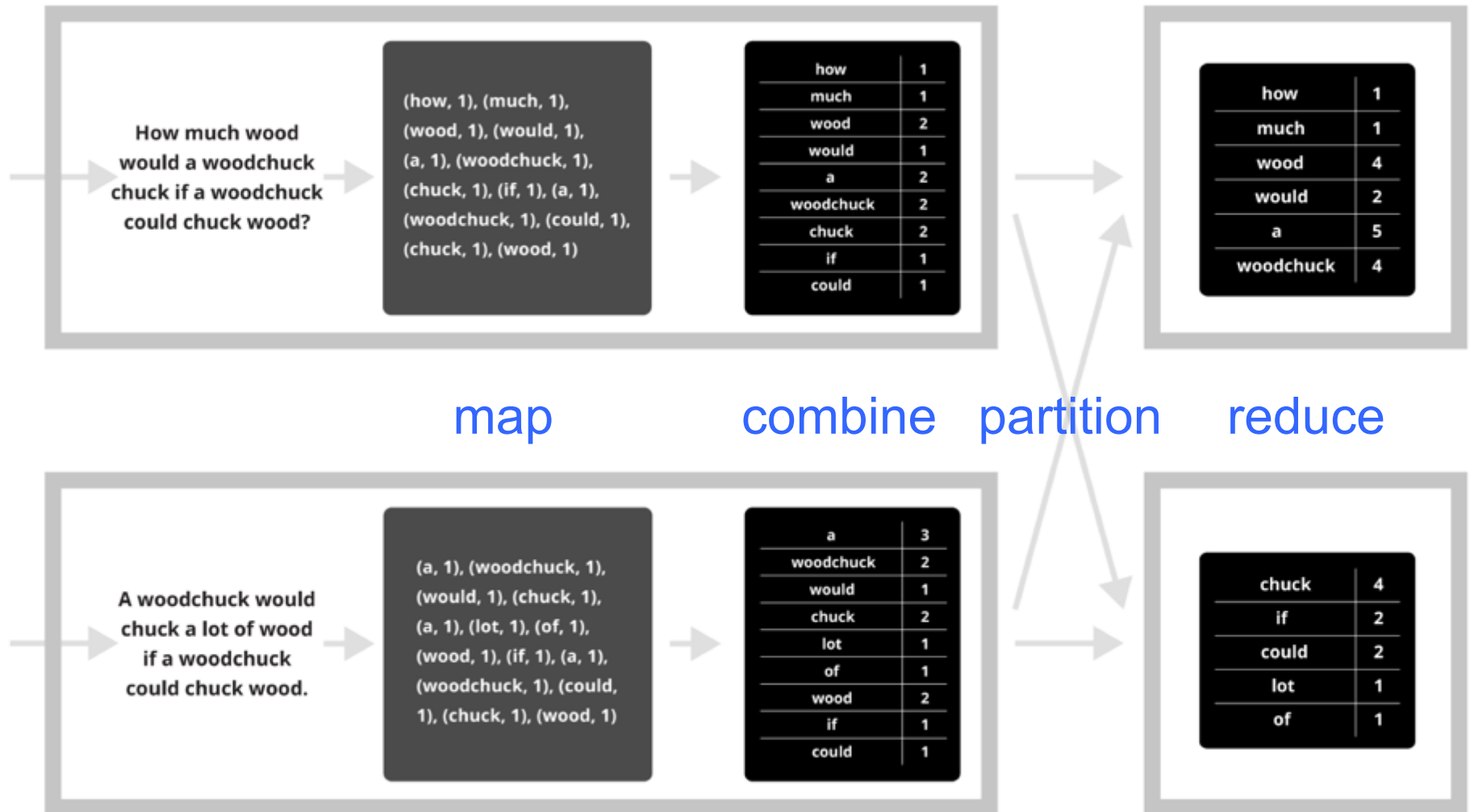
`combine(list<key, value>) -> list<k, v>`

- Perform partial aggregation on each mapper node
`<the, 1>, <the, 1>, <the, 1> → <the, 3>`
- `reduce()` should be commutative and associative

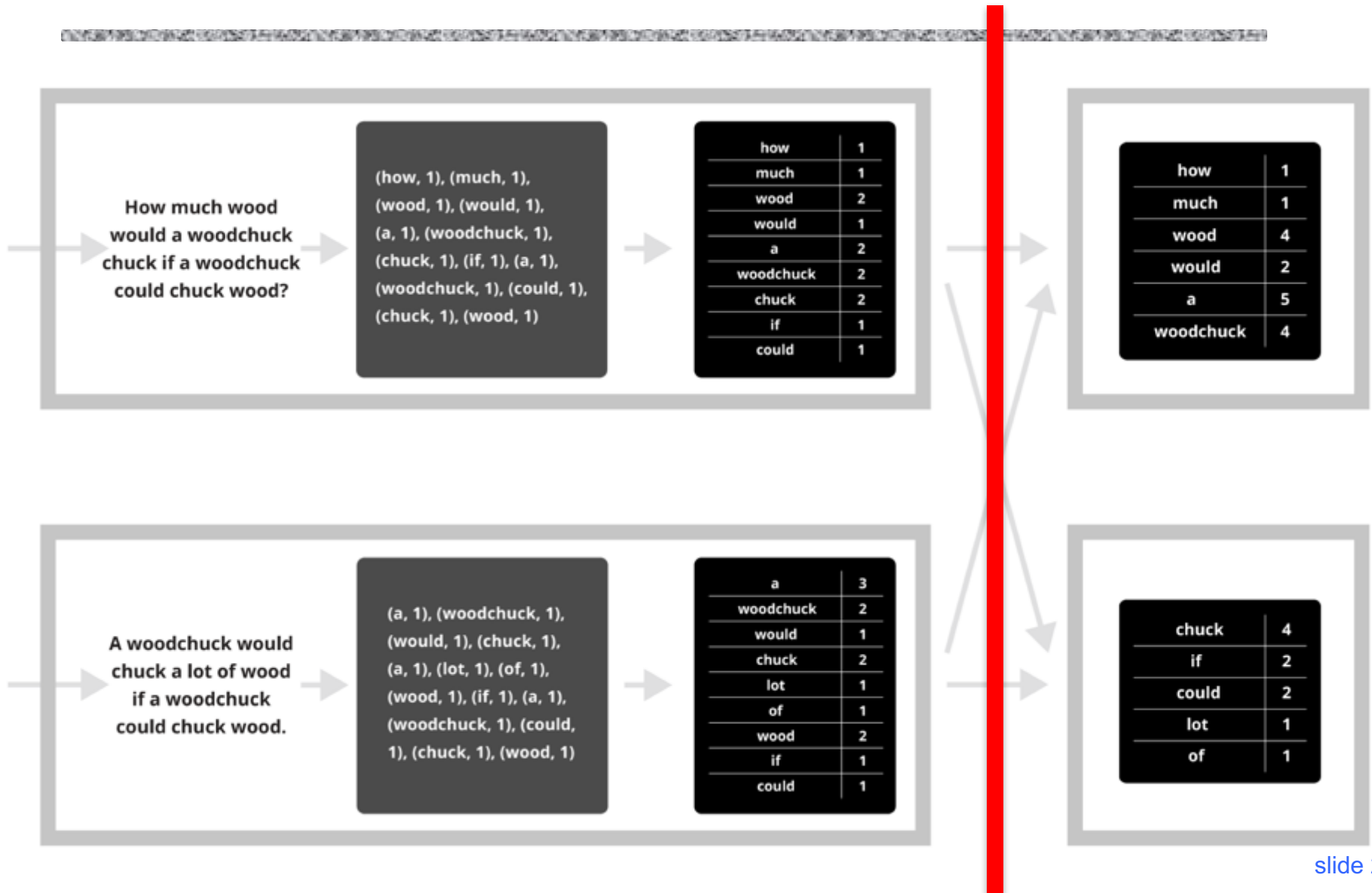
`partition(key, int) -> int`

- Need to aggregate intermediate values with same key
- Given n partitions, map key to partition $0 \leq i < n$
 - Typically via `hash(key) mod n` - but not always!

Putting It Together



Synchronization Barrier



App: grep

- ◆ Input: large set of files
- ◆ Output: lines that match pattern

- ◆ Map:

Output a line if it matches the supplied pattern

- ◆ Reduce:

Copy the intermediate data to output

App: Reverse Web-Link Graph

- ◆ Input: web graph, i.e., tuples (A, B) where page A links to page B
- ◆ Output: for each page, list of pages that link to it

- ◆ **Map:**

For each (source, target), output (target, source)

- ◆ **Reduce:**

Output (target, list(source))

App: URL Access Frequency

- ◆ Input: log of accessed URLs (e.g., from a proxy)
- ◆ Output: for each URL, % of total accesses for that URL

- ◆ Map:

For each URL, output (URL, 1)

- ◆ Multiple reducers:

Output (URL, urlCount)

Chain another MapReduce...

- ◆ Map:

For each (URL, urlCount), output (1, (URL, urlCount))

- ◆ Reduce:

Compute TotalCount, output multiple (URL, urlCount/TotalCount)

App: Sorting

◆ Input: series of values

◆ Output: sorted values

◆ Map:

For each value, output (value, _)

◆ Partitioning:

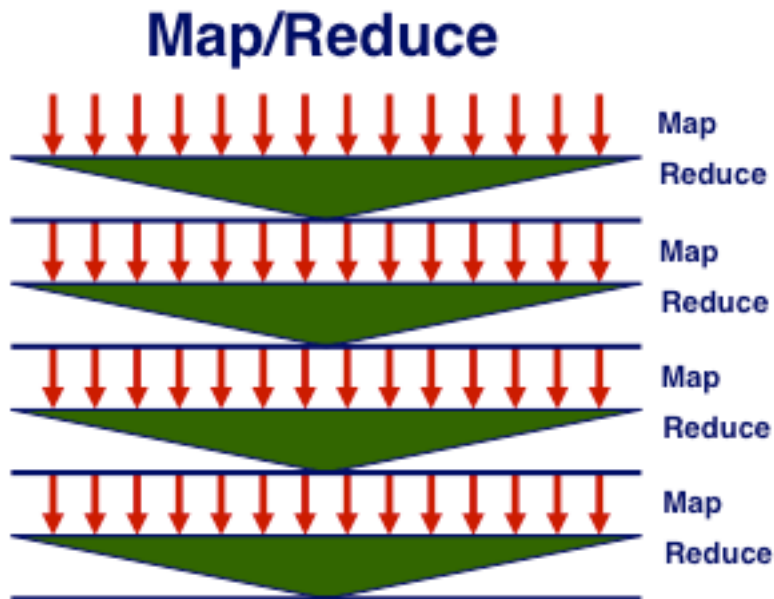
Partition keys across reducers based on ranges

- Take data distribution into accounts to balance reducer tasks

◆ Reduce:

Copy the intermediate data to output

MapReduce Features



◆ Characteristics

- Computation broken up into many short-lived tasks (mapping, reducing)
- Disk storage to hold intermediate results

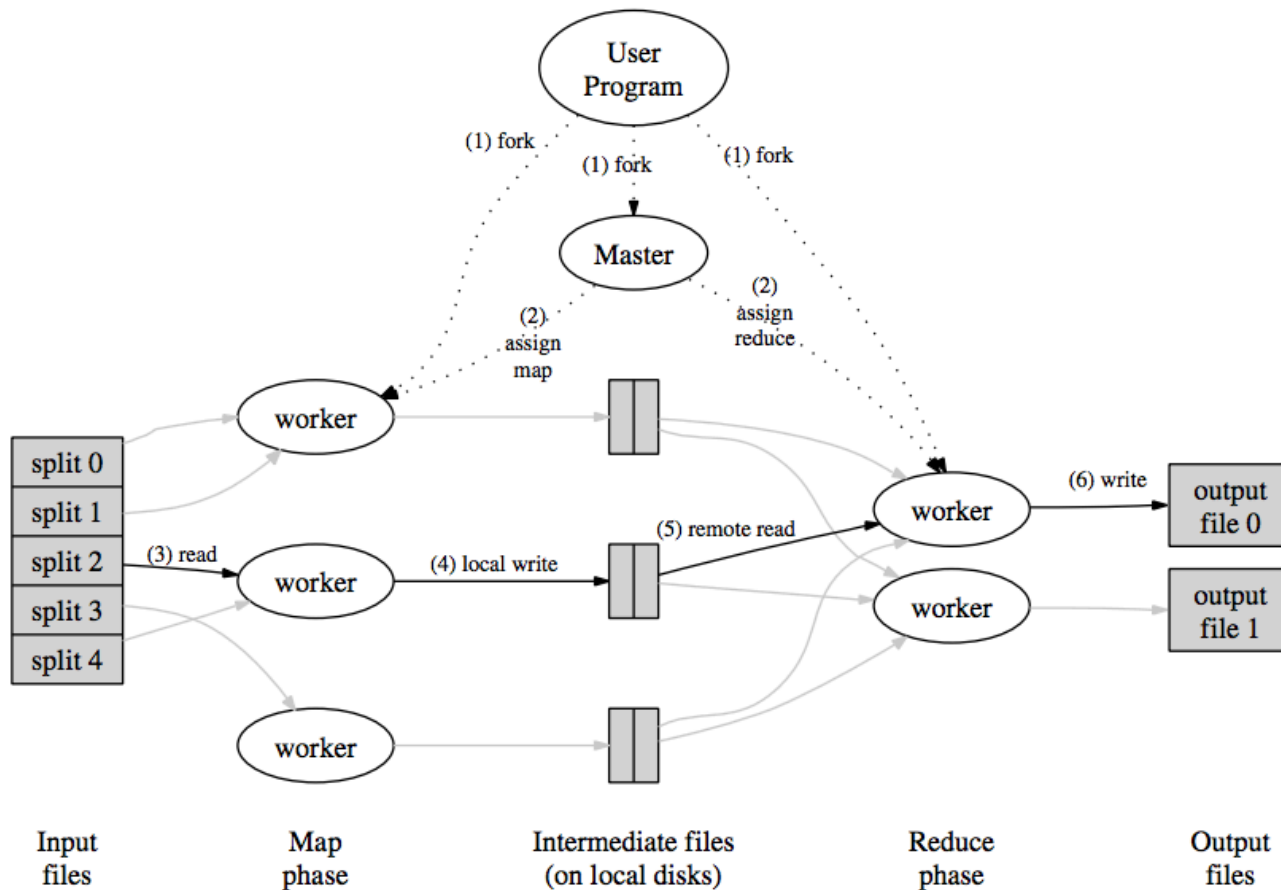
◆ Strengths

- Very flexible placement, scheduling, load balancing
- Can access large datasets

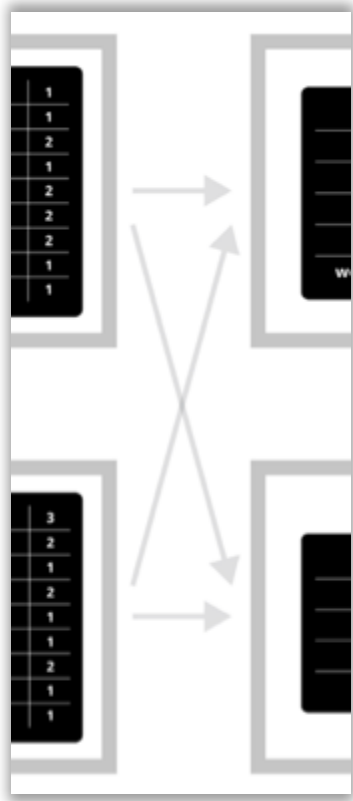
◆ Weaknesses

- Higher overhead
- Lower raw performance

MapReduce Execution



Fault Tolerance in MapReduce



- ◆ Map worker writes intermediate output to local disk, separated by partitioning; once completed, tells master node
- ◆ Reduce worker told of location of map task outputs, pulls their partition's data from each mapper, executes function across data
- ◆ Note:
 - “All-to-all” shuffle between mappers and reducers
 - Written to disk (“materialized”) before each stage

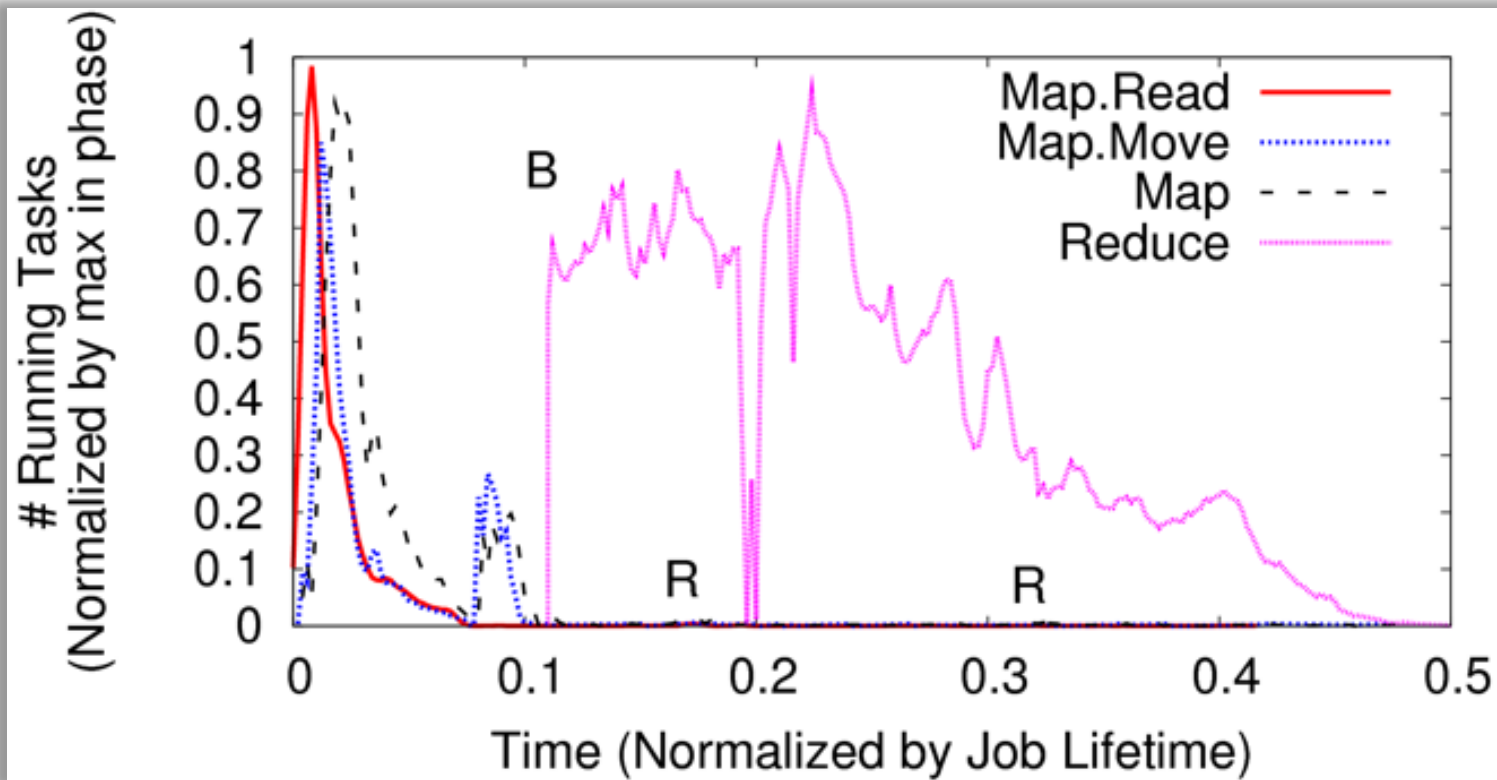
Fault Tolerance in MapReduce

- ◆ Master node monitors state of system
 - If master fails, job aborts
- ◆ Map worker failure
 - In-progress and completed tasks marked as idle
 - Reduce workers notified when map task is re-executed on another map worker
- ◆ Reducer worker failure
 - In-progress tasks are reset and re-executed
 - Completed tasks had been written to global file system

Stragglers

- ◆ Straggler = task that takes long time to execute
 - Bugs, flaky hardware, poor partitioning
- ◆ For slow map tasks, execute in parallel on second “map” worker as backup, race to complete task
- ◆ When done with most tasks, reschedule any remaining executing tasks
 - Keep track of redundant executions
 - Significantly reduces overall run time

Straggler Mitigation



Goodbye MapReduce

NEWS

6/27/2014
10:09 AM



Charles Babcock
News

Google I/O: Hello Dataflow, Goodbye MapReduce

Google introduces Dataflow to handle streams and batches of big data, replacing MapReduce and challenging other public cloud services.

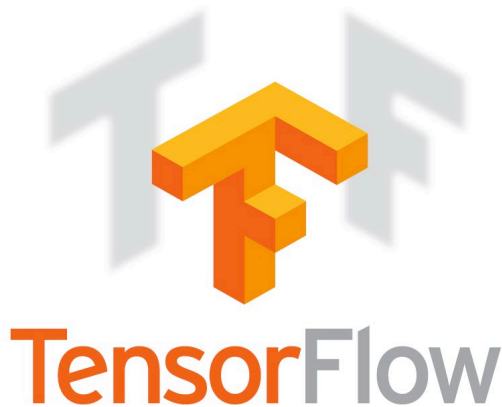
Google I/O this year was overwhelmingly dominated by consumer technology, the end user interface, and extension of the Android



Modern Data Processing



Big driver: machine learning



Apple acquires machine learning and AI startup Turi for \$200M, report says