CS 8803 Big Data Systems and Analytics

Big Data Processing Platforms (II)

Ling Liu

Professor
School of Computer Science
College of Computing, Georgia Tech

Outline

Review of Last week's Lectures

- Hadoop Distributed File System (HDFS)
- Hadoop MapReduce Programming Model
- Sample applications
- Higher-level languages: Pig and Hive

Today's Lecture

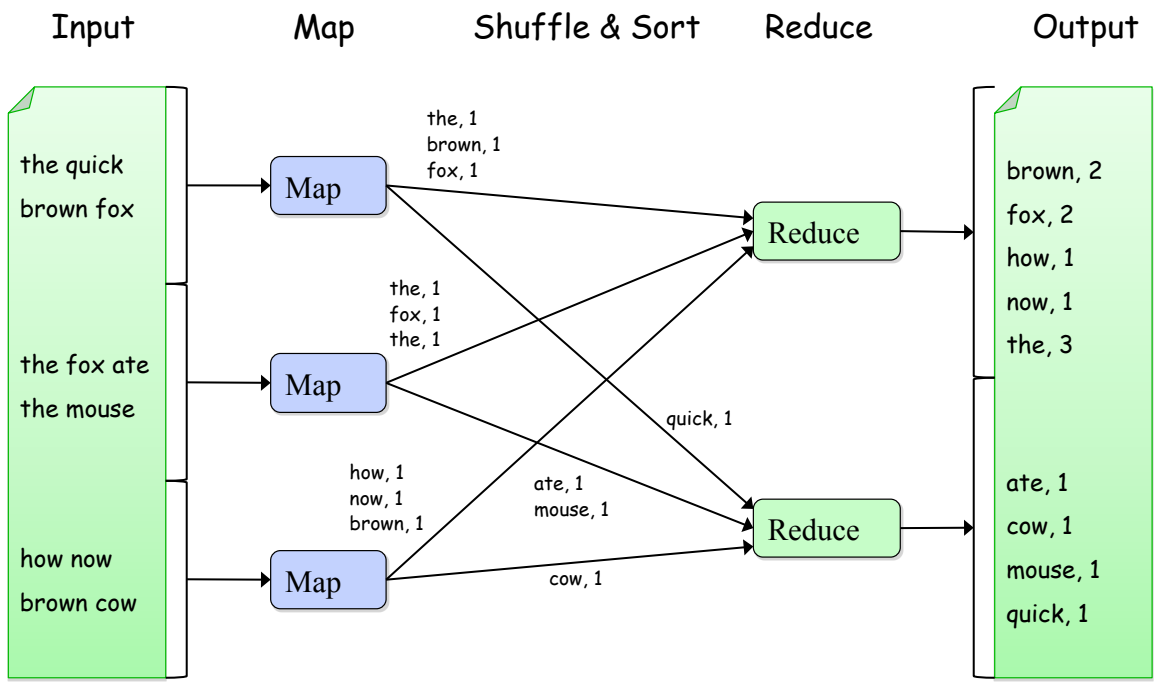
- MapReduce Computation Model and Runtime Engine
  - Three Phases: Map + Shuffle + Reduce
- Fault tolerance in MapReduce
- Optimization Opportunities

Word Count using MapReduce

```
map(key, value):
// key: document name; value: text of document
   for each word w in value:
        emit(w, 1)

reduce(key, values):
// key: a word; value: an iterator over counts
        result = 0
        for each count v in values:
              result += v
        emit(result)
```


Word Count Execution



Map-Reduce Computation

- How is the Map/Reduce Computation is distributed (and parallelized)?
  - A MapReduce job consists of one or more rounds of Map and Reduce processing
  - Each round of Map-Reduce processing consists of multiple map tasks (N) and multiple reduce tasks (M)
  - The input data to a MapReduce job is partitioned by HDFS into small chunks of equal size (64KB).
    - #chunks = N (#map tasks) and M << N
  - Three Execution Phases
    - Map
    - Shuffle
    - Reduce

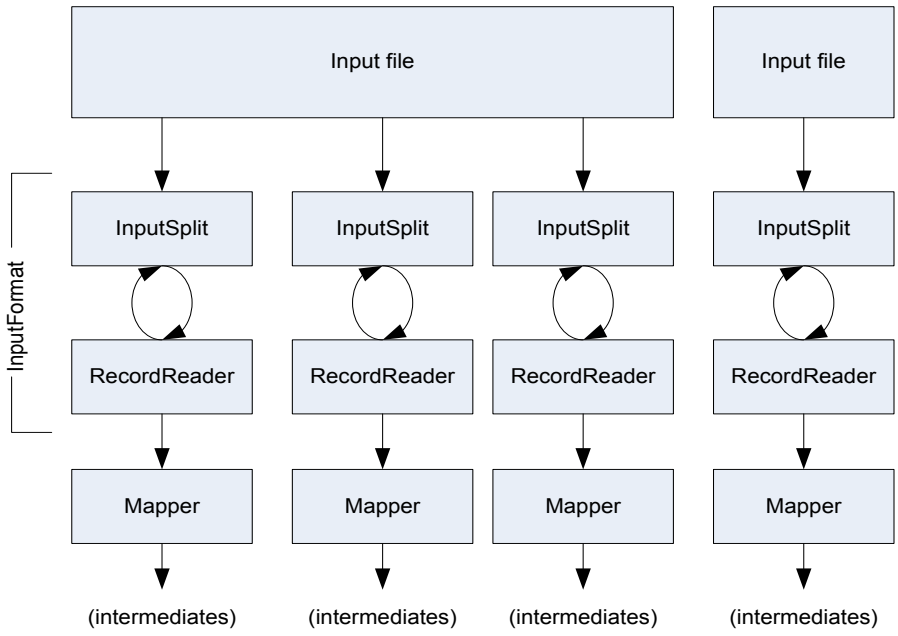
Map-Reduce Execution Summary

- Map/Reduce Computation is distributed (and parallelized) in three phases
  1. **Map phase**:
     - Multiple map tasks are executed in parallel.
     - Each execute the map code with map input on a map worker.
     - Each map task map() partitions input data into key/value pairs as map output
  2. **Shuffle phase**:
     - After all map() tasks are complete, for each map task, consolidate all emitted values for each unique emitted key and partition the map output into K partitions, where K is the # of reduce tasks
  3. **Reduce phase**:
     - run reduce() in parallel and aggregate the values for each key collected from M map tasks.

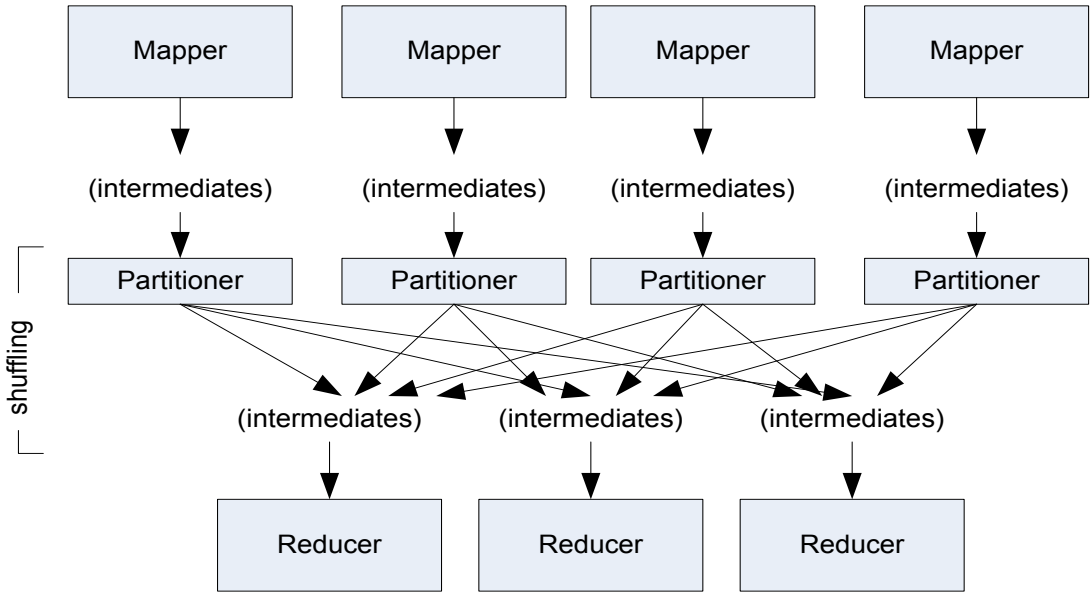
Data flow

- Input, final output are stored on a distributed file system
  - Scheduler tries to schedule map tasks "close" to physical storage location of input data
- Intermediate results are stored on local FS of map workers and/or reduce workers
- Output is stored in designated storage locations and triple replicated
  - often input to another map reduce task

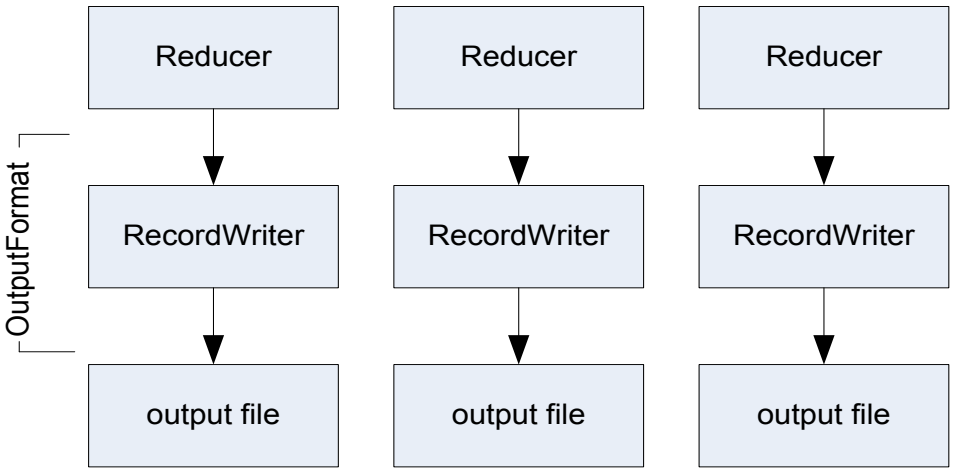
Getting Data To The Mapper



Partition And Shuffle



Finally: Writing The Output



Triple Replication

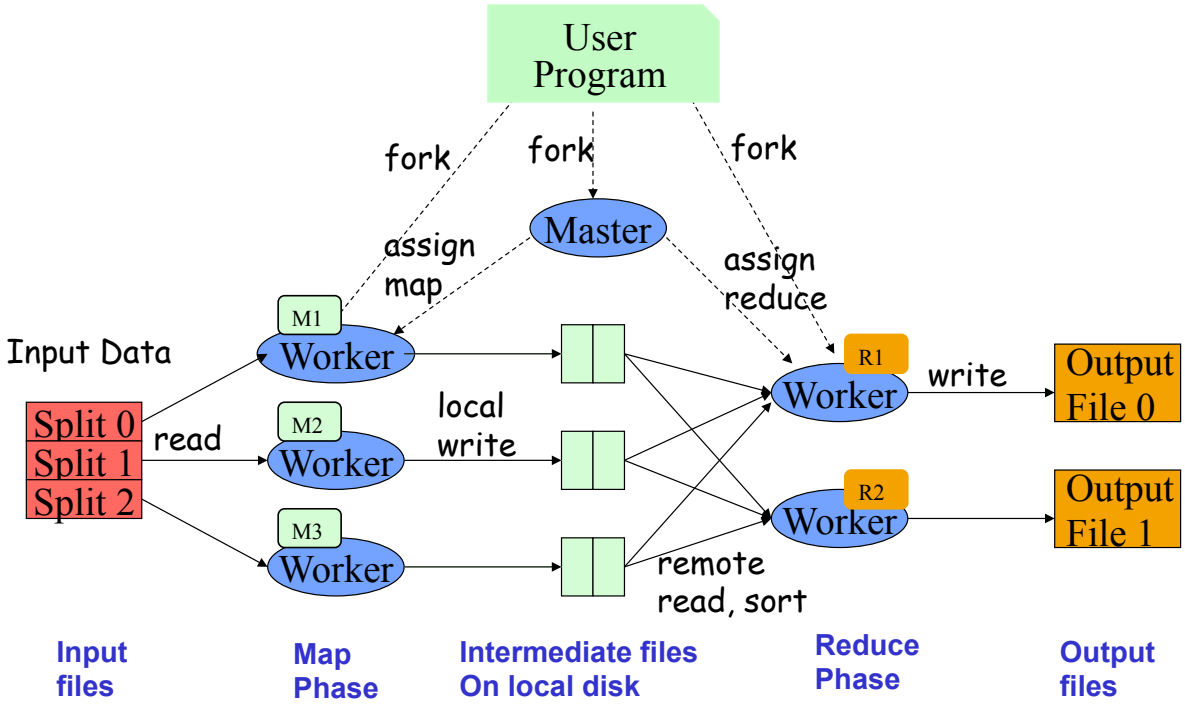
MapReduce Execution Engine

- Mapper-Reducer Coordination: Data Flow
- Master-Slave Coordination: Execution Flow
- Execution Support for Scaling to Big Data

MapReduce Execution Details

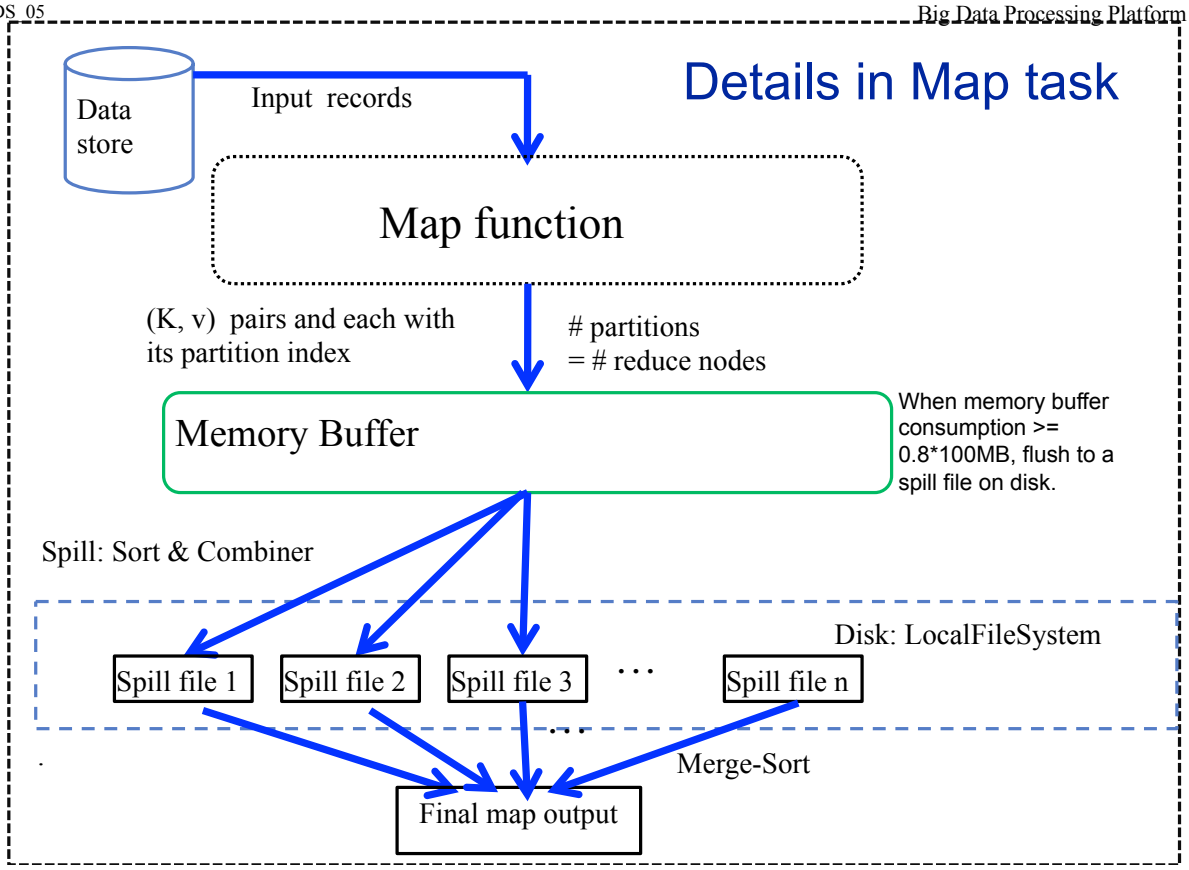
- Single *master* controls job execution on multiple *slaves*
- Mappers preferentially placed on same node or same rack as their input block
  - Minimizes network usage
- Mappers save outputs to local disk before serving them to reducers
  - Allows recovery if a reducer crashes
  - Allows having more reducers than nodes
- If map() or reduce() fails, re-execute!

Distributed Execution: Overview

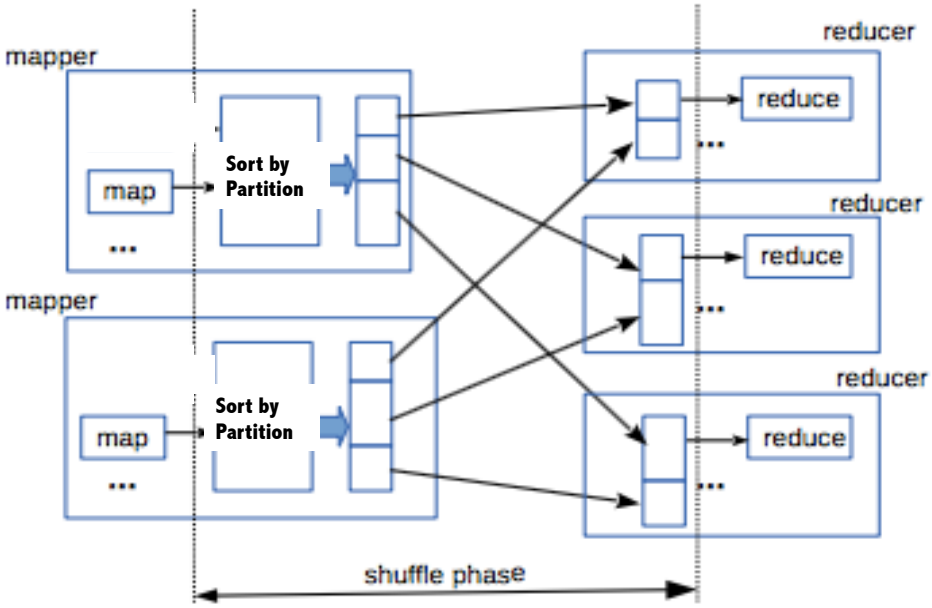


How does MapReduce scale to big data

- Input file splits into chunks of equal size and stored in HDFS with triple replication
- Each map takes one chunk (input-split) as an input dataset.
- When input dataset is too big to fit into the allocated memory for each map task (slot), spill files will be generated.
- At the end of a map() task, spill files are merged through sorting by key.



MapReduce Computation Model



Execution overview

1. The MapReduce library in the user program first splits input files into M pieces of typically 16 MB to 64 MB/piece. It then starts up many copies of the program on a cluster of machines.
2. One of the copies of the program is the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.
3. A worker who is assigned a map task reads the contents of the assigned input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.
4. The locations of these buffered pairs on the local disk are passed back to the master, who forwards these locations to the reduce workers.

Execution overview (cont.)

5. When a reduce worker is notified by the master about these locations, it uses RPC remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together.
6. The reduce worker iterates over the sorted intermediate data and for each **unique intermediate key** encountered, it passes the key and the corresponding set of intermediate values to the user's *Reduce function*. The output of the *Reduce function is appended* to a final output file for this reduce partition.
7. When all map tasks and reduce tasks have been completed, the master wakes up the user program---the MapReduce call in the user program returns back to the user code. The output of the mapreduce execution is available in the R output files (one per reduce task).

Coordination

- Master data structures
  - Task status: (idle, in-progress, completed)
  - Idle tasks get scheduled as Map workers become available
  - When a map task completes, it produces its map output file and sends the master the location and sizes of its R intermediate files, one for each reducer
  - Master pushes this info to reducers
- Failure handling
  - Master pings workers periodically to detect failures
- Speculation
  - When a map slot is done and no more map task is left, the map slot is available for speculation task (helping stragglers)

Fault Tolerance in MapReduce

- Task crashes (map or reduce)
- Node crashes
- Speculative Execution
  - If a task is going slowly (straggler)
- Master crashes

Fault Tolerance in MapReduce

1. If a task crashes:
   - Retry on another node
     - Okay for a map task because it had no dependencies
     - Okay for a reduce task because map outputs are on disk
   - If the same task repeatedly fails, fail the job or ignore that input block (user-controlled)

➢ Note: For this and the other fault tolerance features to work, *your map and reduce tasks must be side-effect-free*

Fault Tolerance in MapReduce

2. If a node crashes:
   - Re-launch its current tasks on other nodes
   - Re-launch any maps the node previously ran
     - Why is this necessary?
     - Because their output files were lost along with the crashed node

Fault Tolerance in MapReduce

3. If a task is going slowly (straggler):
   - Launch the second copy of task on another node (speculative execution);
   - Take the output of whichever copy finishes first, and kill the other one.
   - Speculation is turned on only when there is no more pending map input chunks

- Critical for performance in large clusters: stragglers occur frequently due to failing hardware, bugs, misconfiguration, etc.

Failures

Q&A: Why is a completed map task discarded?

- Map worker failure
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when a map task is rescheduled on another worker (early shuffle)
- Reduce worker failure
  - Only in-progress tasks are reset to idle
- Master failure
  - MapReduce job is aborted and client is notified

Summary

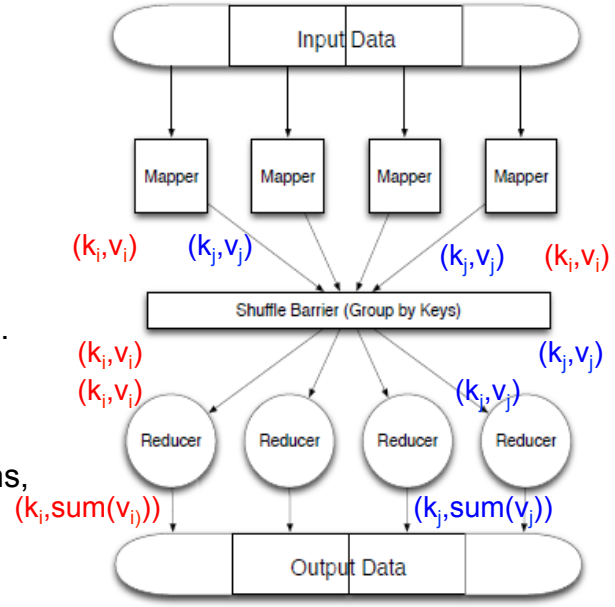
- By providing a data-parallel programming model, MapReduce can control job execution in useful ways:
  - Automatic division of job into tasks
  - Automatic placement of computation near data
  - Automatic load balancing
  - Recovery from failures & stragglers
- Users can focus on application, not on complexities of distributed computing

MapReduce: Optimization Opportunities

- Performance tuning challenges for big data processing
  - MapReduce stage barrier
    - Efficient configuration of early shuffle or adaptive setting of Barrier
    - Task Failure handling in a long running job
    - Complex data processing with high correlation
  - MapReduce Task Scheduling → Late Scheduler
  - MapReduce Job Scheduling → YARN
- MapReduce Workload benchmark
- MapReduce Configuration Management

MapReduce Stage Barrier

- The MapReduce model uses a barrier in the Shuffle stage between the Map and Reduce stages.
  - **Advantage:**
    - It provides simplicity in both programming and implementation.
    - It ensures consistency
  - **Disadvantage:**
    - overly restrictive in many situations, may hurt performance

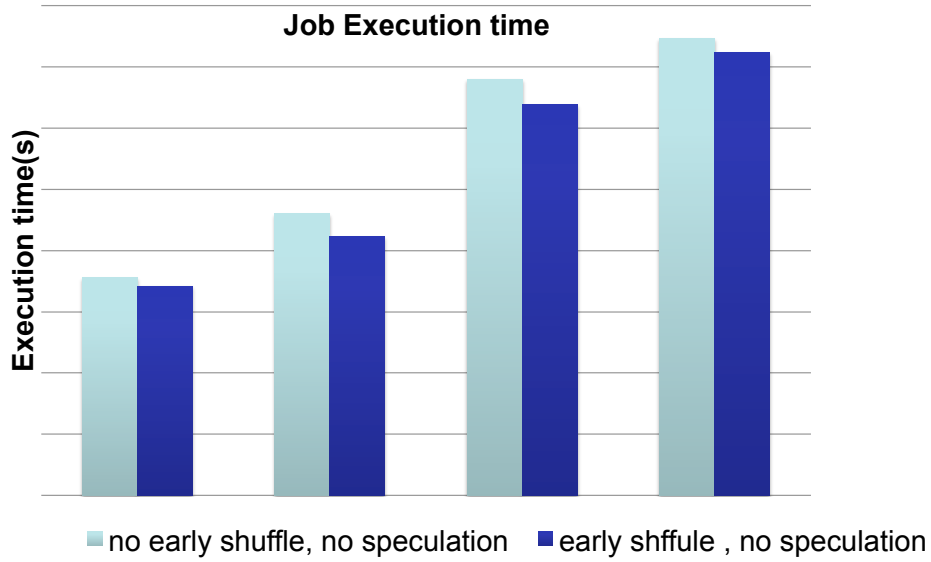


Connecting Map programs with Reduce programs

- **Standard Shuffle**
  - Start Shuffle phase only when all maps have been completed
  - Who keep the accounting: MapReduce Master Node
  - How:
    - Each map task reports to the master when it completes its map task execution.
    - Status report includes the location and size of the map input file, the time to complete the map task, etc.
- **Early Shuffle**
  - Upon the completion of certain percentage (e.g., 5%) of the total Map tasks, shuffle phase can start without further waiting.
  - Improve latency and throughput of MapReduce Jobs.

Early Shuffle v.s. No Early Shuffle

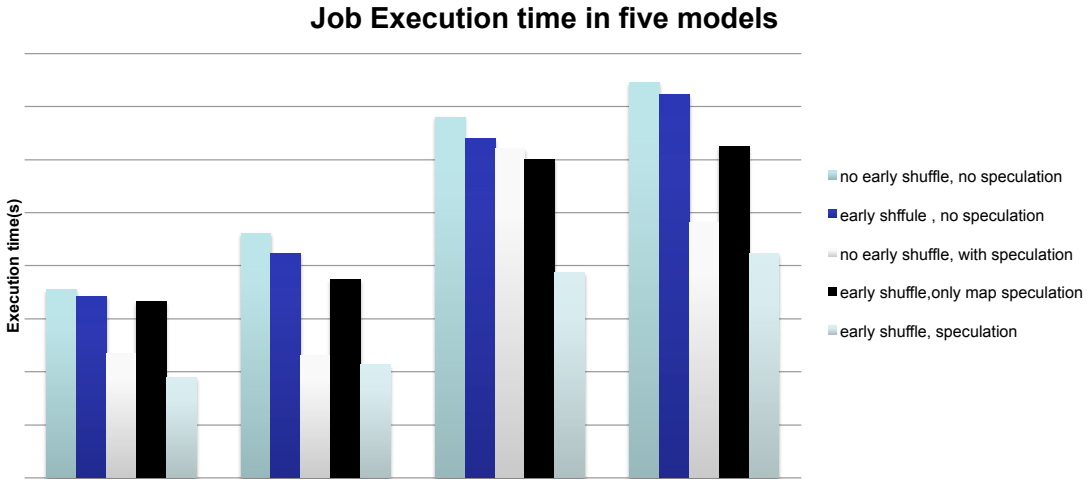
[Zhao Xu, Ling Liu, 2014]



Speculation

- Master maintains each map or reduce task execution statistics
- Straggler
  - Upon a map or reduce node completes its task and request for next, if no more map or reduce task pending at Master, then compute which one of the current map tasks is a straggler (slow to finish)
- Speculation
  - For each straggler, choose one speculative execution node and assign the same task running on the straggler to run on the chosen speculation node (as redundant execution
  - When one of the two duplicated executions is complete, the slow one will be killed.

Different Job Execution Models A Comparison



Q&A

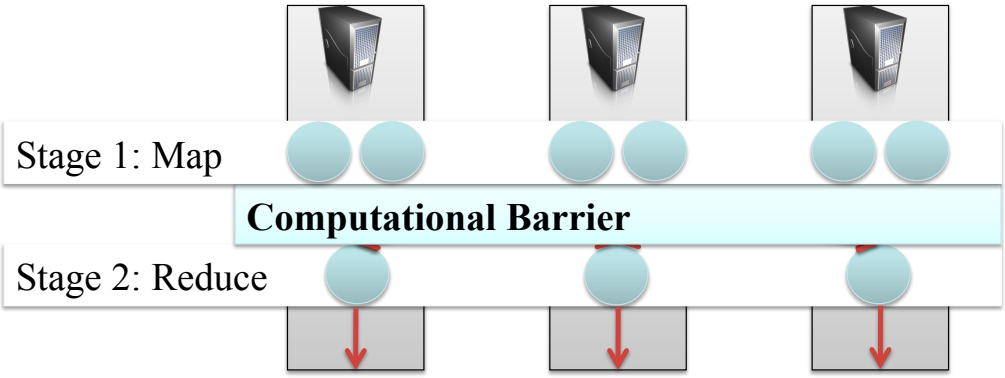
- For each server node (slave) in a MapReduce cluster, what types of MapReduce workloads it is running at any given time?
- For a node running only Map task:
  - **Map task**
  - **Shuffle (Reduce serving) task**
- For a node running only reduce task:
  - **Shuffle (Reduce serving) task**
  - Reduce task
- For a node running both Map and Reduce tasks:
  - **Map task**
  - Reduce task
  - **Shuffle (Reduce serving) task**

MapReduce: Optimization Opportunities

- Performance tuning challenges for big data processing
  - MapReduce stage barrier
    - Efficient configuration of early shuffle or adaptive setting of barries
    - Task Failure handling in a long running job
    - Complex data processing with high correlation
  - MapReduce Task Scheduling → Late Scheduler
  - MapReduce Job Scheduling → YARN
- MapReduce Workload benchmark
- MapReduce Configuration Management

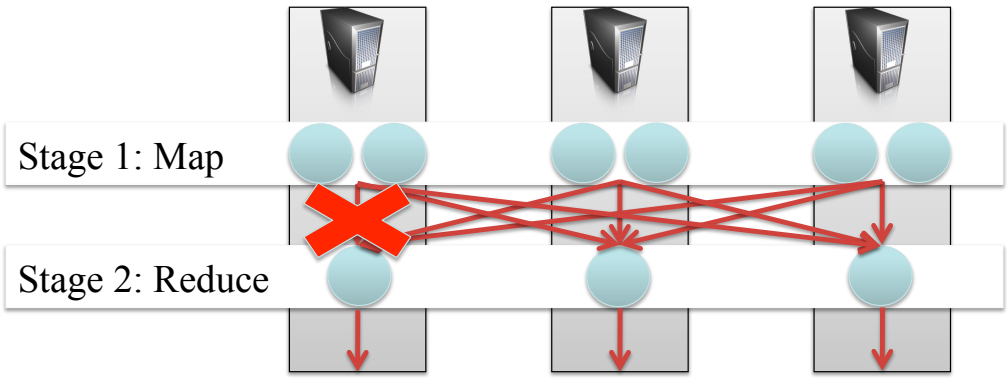
Hadoop/MapReduce Behavior

- Intermediate data
  - Data generated in between stages
  - Similarities to traditional intermediate data in traditional file systems, e.g., .o files
  - Critical to produce the final output
  - Short-lived, written-once and read-once, & used-immediately



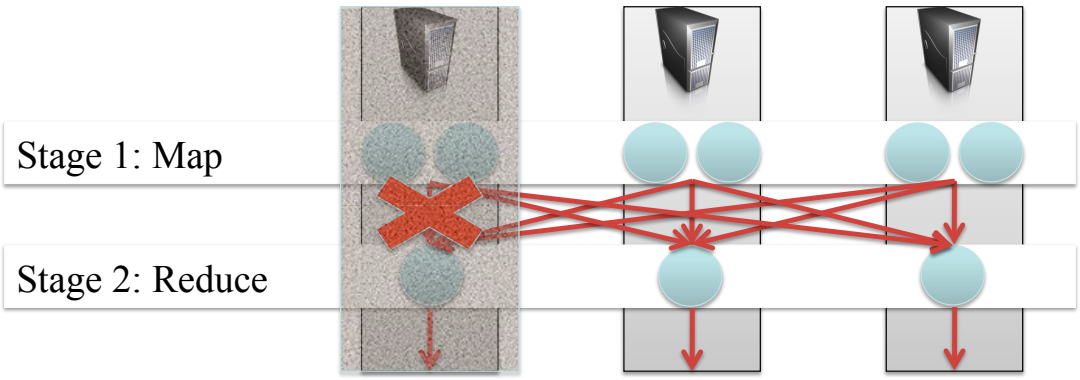
Computational Barrier + Failures

- Availability becomes critical
  - Loss of intermediate data before or during the execution of a task
    => the task can't proceed



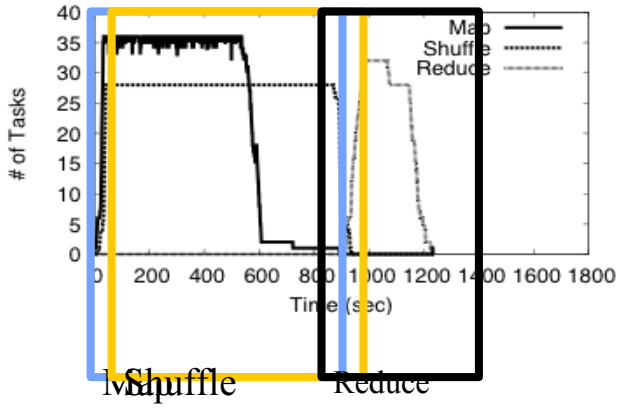
A Common Solution in Systems

- Store locally & re-generate when lost
  - Re-run affected map & reduce tasks
  - No support from a storage system
- Assumption: re-generation is cheap and easy



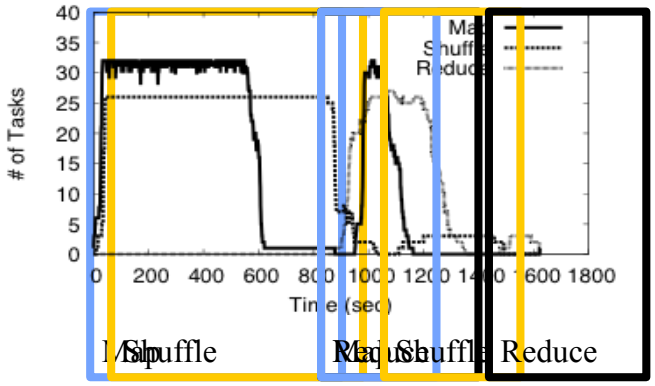
Hadoop: Effect of a Failure

- Emulab: 20 machines sorting 36GB, 4 LANs and a core switch (all 100 Mbps)



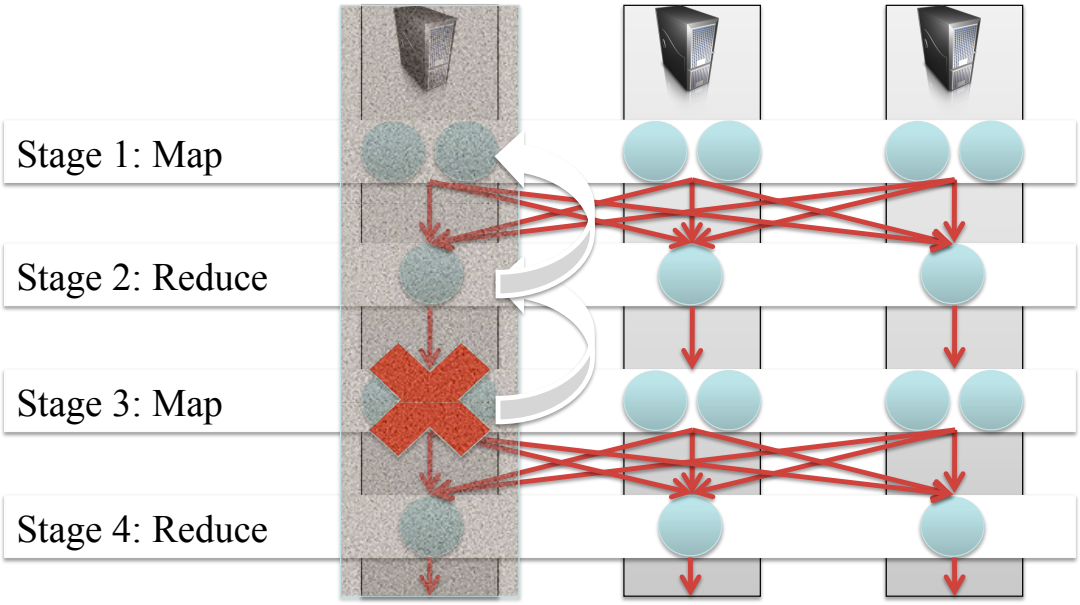
Hadoop: Effect of a Failure

- 1 failure after Map
  - Re-execution of Map-Shuffle-Reduce
- ~33% increase in completion time



Effect of Failures is worse with Multi-Stage Computations

- Cascaded re-execution: expensive

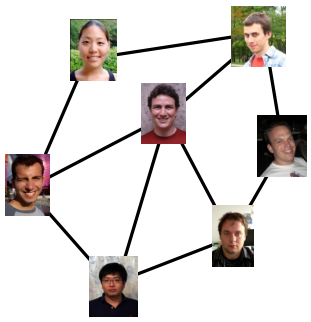


MapReduce: Optimization Opportunities

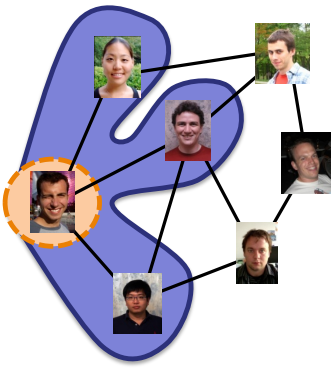
- Performance tuning challenges for big data processing
  - MapReduce stage barrier
    - Efficient configuration of early shuffle or adaptive setting of barries
    - Task Failure handling in a long running job
    - Complex data processing with high correlation
  - MapReduce Task Scheduling → Late Scheduler
  - MapReduce Job Scheduling → YARN
- MapReduce Workload benchmark
- MapReduce Configuration Management

Properties of Graph Parallel Algorithms

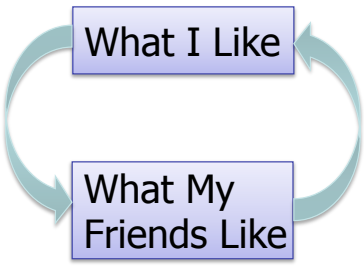
Dependency Graph



Factored Computation



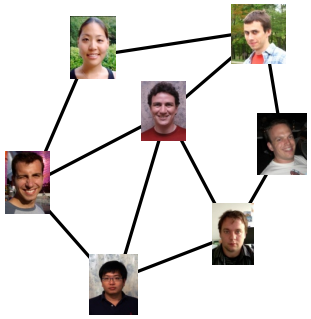
Iterative Computation

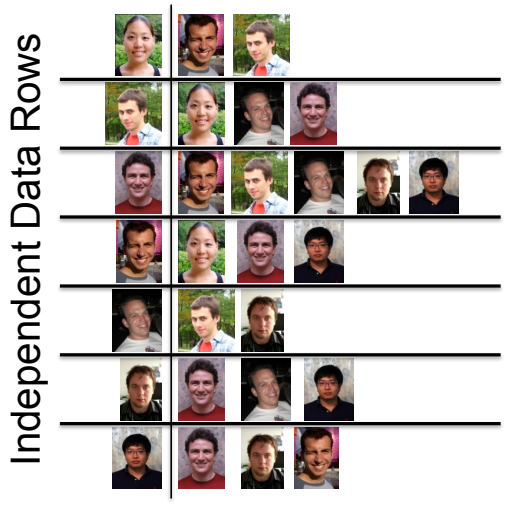


Courtesy GraphLab, CMU, 2013

Data Dependencies

- Map-Reduce does not efficiently express dependent data
  - User must code substantial data transformations
  - Costly data replication





Courtesy GraphLab, CMU, 2013

Map-Reduce for Data-Parallel Computation

- Excellent for large data-parallel tasks!

Map Reduce

Feature Extraction

Cross Validation

Computing Sufficient Statistics

Map Reduce?

Label Propagation

Lasso

SSSP

Kernel Methods

Belief Propagation

Tensor Factorization

PageRank

Deep Belief Networks

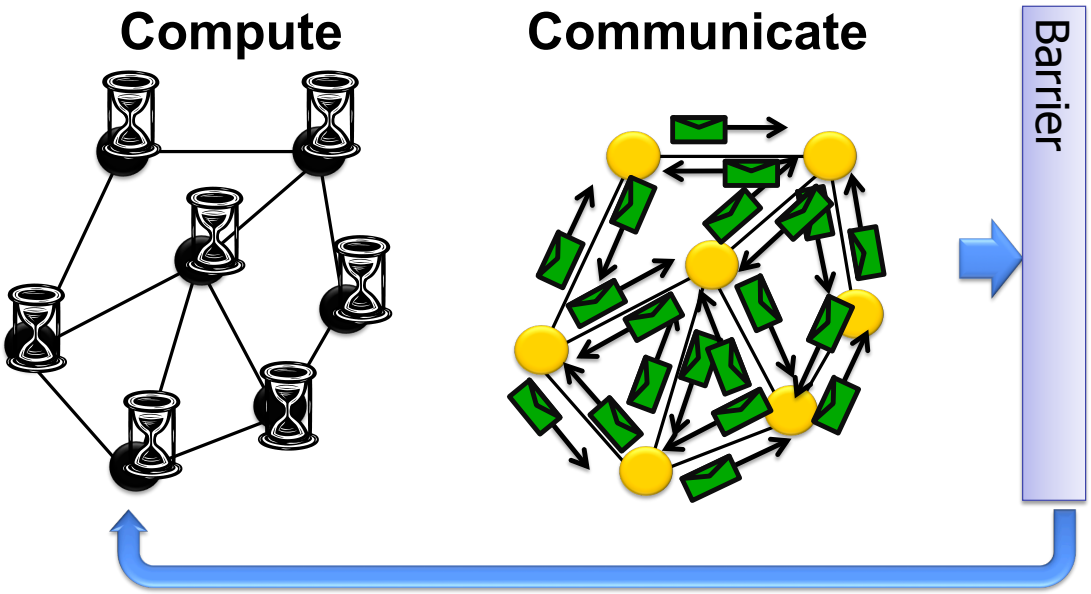
Neural Networks

Courtesy GraphLab, CMU, 2013

Why is Map-Reduce inefficient for Graph Parallel Algorithms?

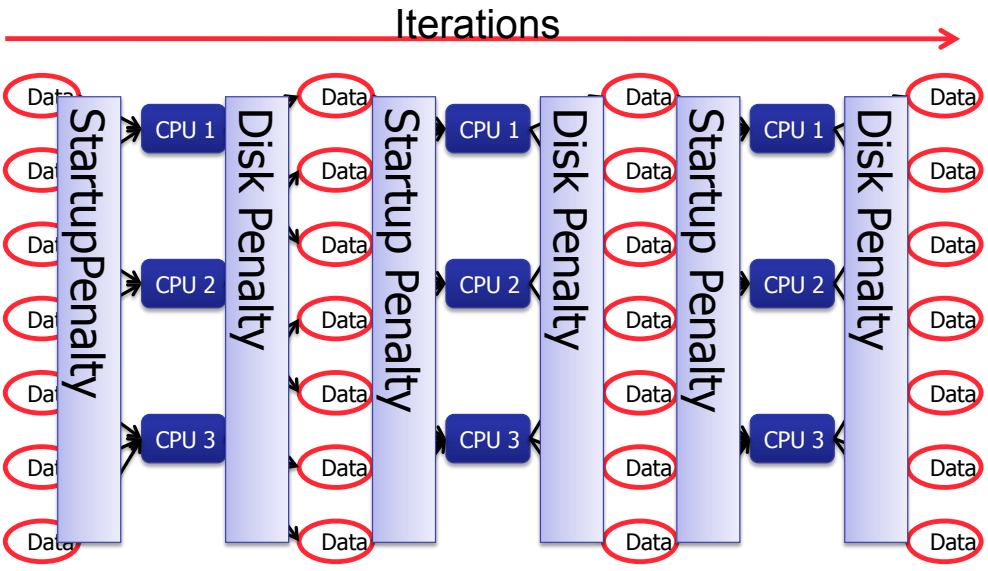
Pregel (Giraph)

- Bulk Synchronous Parallel Model:

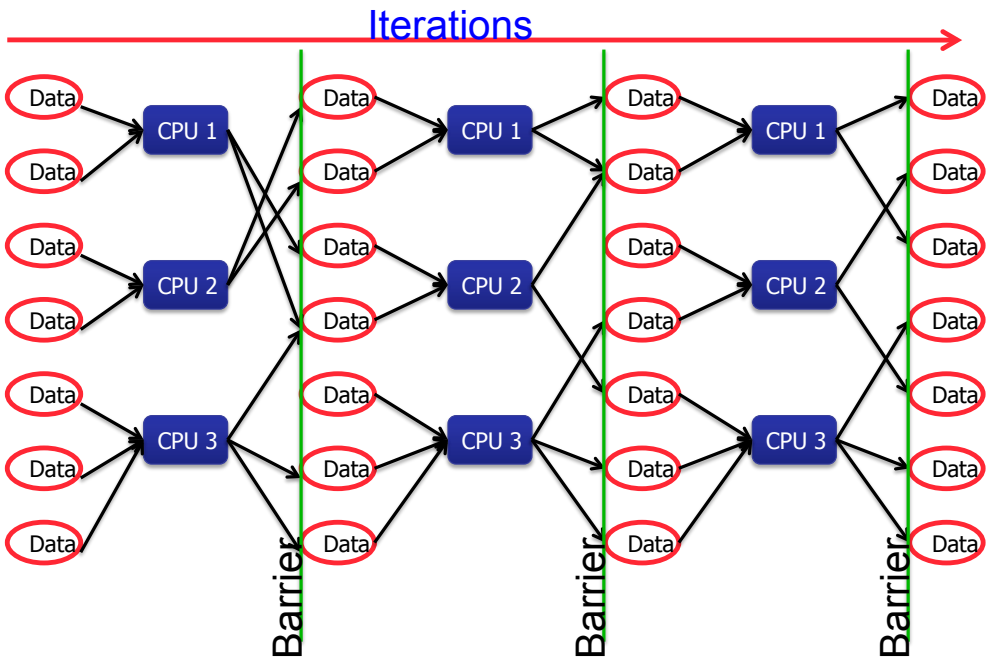


Iterative MapReduce

with Conventional Hadoop MapReduce



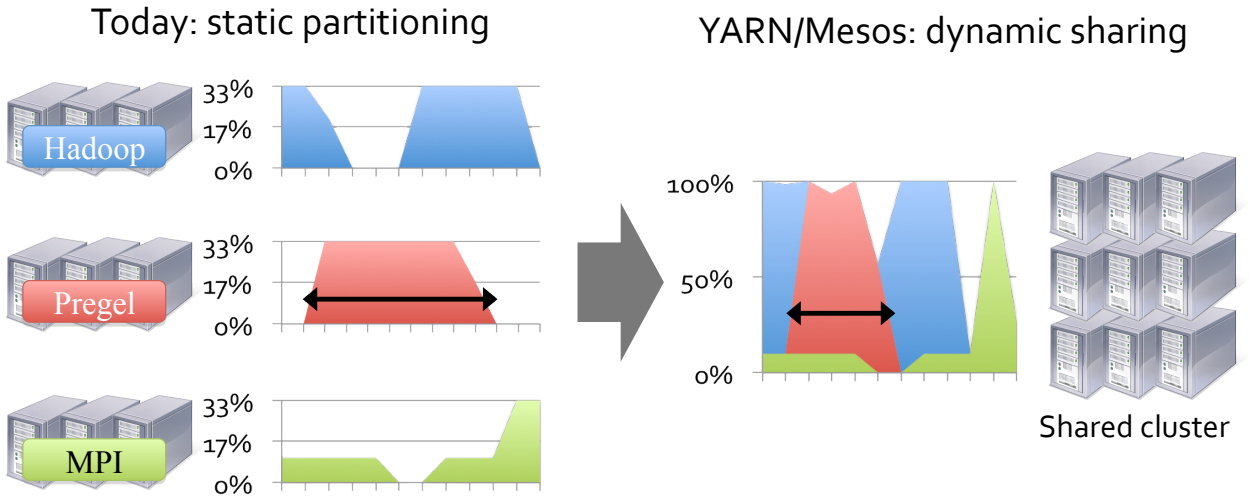
Streaming MapReduce with Bulk Synchronous Parallel Model + Check-points



MapReduce: Optimization Opportunities

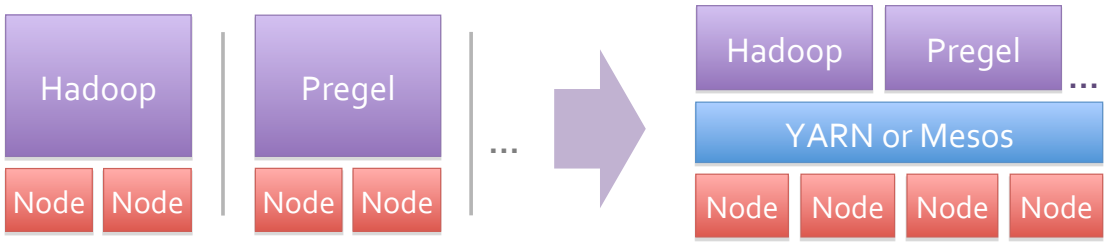
- Performance tuning challenges for big data processing
  - MapReduce stage barrier
    - Efficient configuration of early shuffle or adaptive setting of barries
    - Task Failure handling in a long running job
    - Complex data processing with high correlation
  - MapReduce Task Scheduling → Late Scheduler
  - MapReduce Job Scheduling → YARN
- MapReduce Workload benchmark
- MapReduce Configuration Management

Why YARN / Mesos



YARN or Mesos: Design Idea

- A common resource sharing layer over which diverse frameworks can run



Fully utilize Cluster

Design Goals

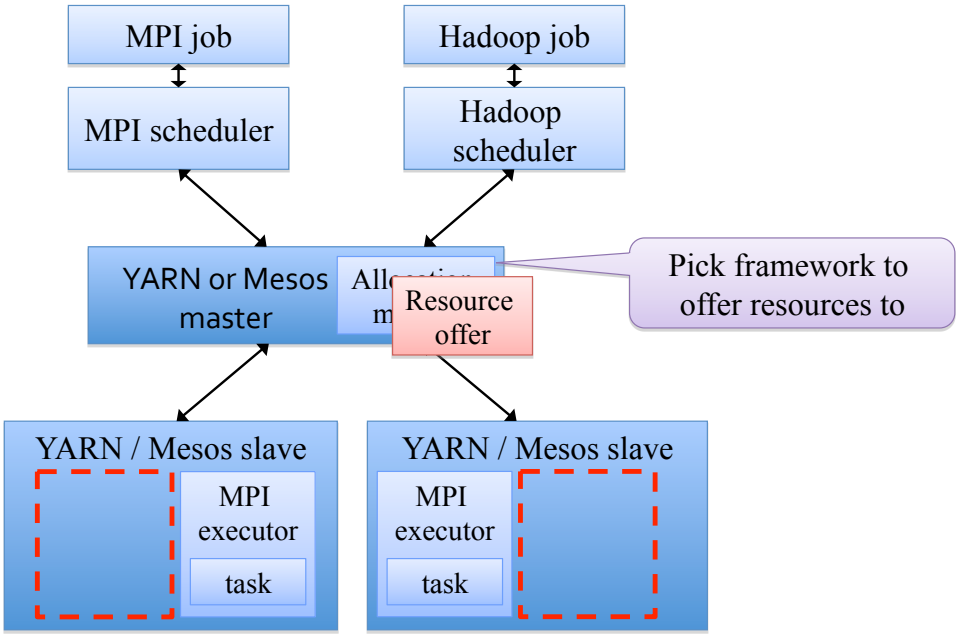
- **High utilization** of resources
- **Support diverse frameworks**
- **Scalability** to 10,000's of nodes
- **Reliability** in face of failures

Resulting design: Small microkernel-like core that pushes scheduling logic to frameworks

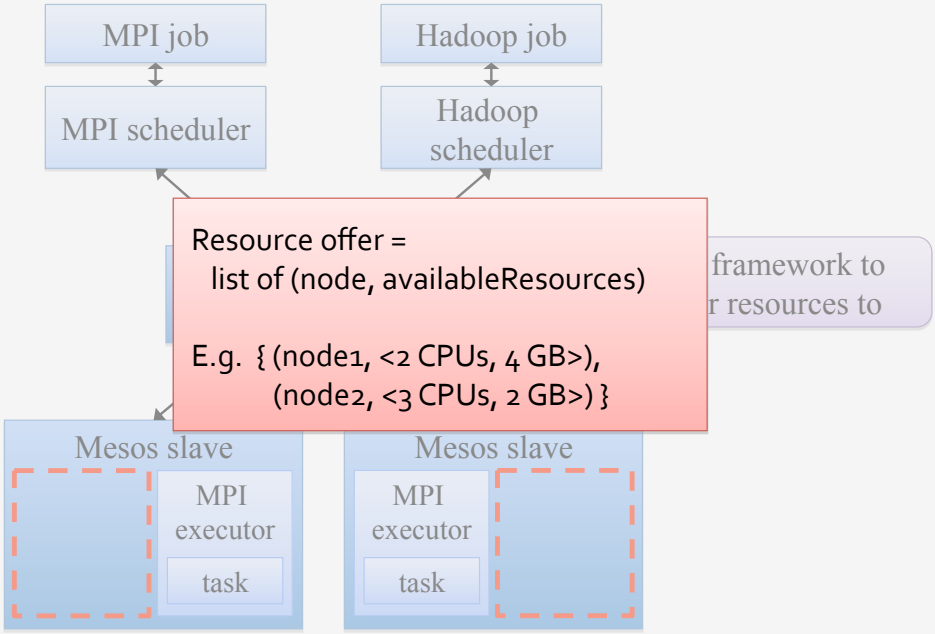
Design Elements

- Fine-grained sharing:
  - Allocation at the level of *tasks* within a job
  - Improves utilization, latency, and data locality
- Resource offers:
  - Simple, scalable application-controlled scheduling mechanism

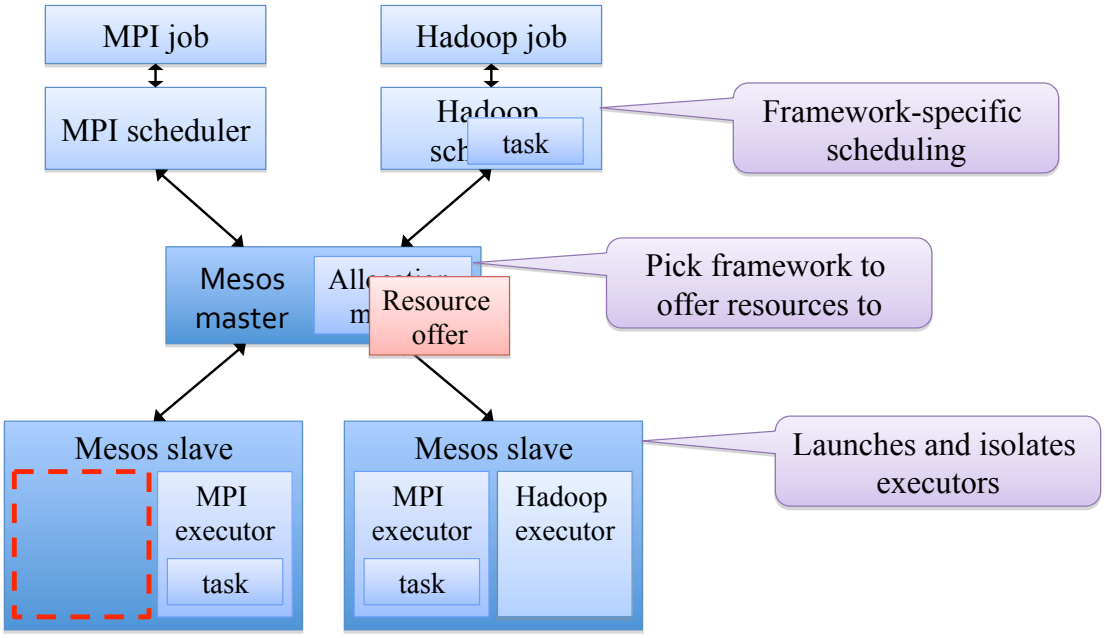
YARN / Mesos Architecture



Mesos Architecture



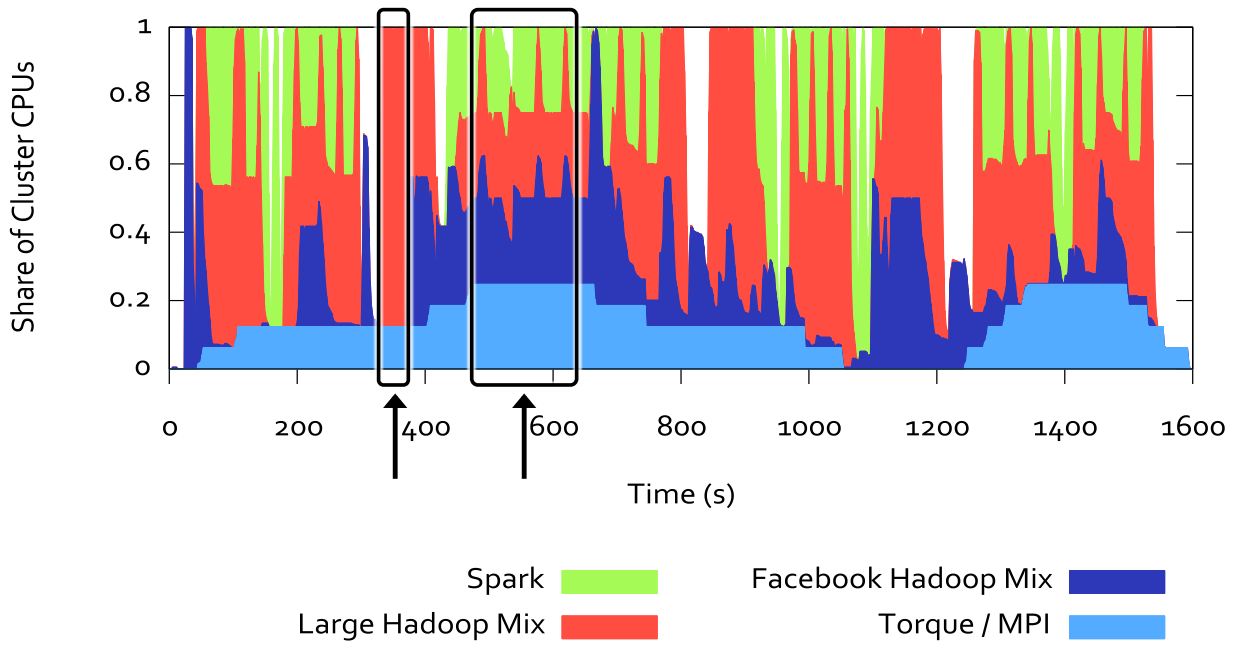
Mesos Architecture



Framework Isolation

- YARN / Mesos uses OS isolation mechanisms, such as Linux containers and Solaris projects
- Containers currently support CPU, memory, IO and network bandwidth isolation
- Not perfect, but much better than no isolation

Dynamic Resource Sharing



YARN

- Goal: allowing you to share a large cluster of machines between different frameworks.
- Splits up the two major functions of JobTracker
  - **Global Resource Manager** - Cluster resource management
  - **Application Master** - Job scheduling and monitoring (one per application). The Application Master negotiates resource containers from the Scheduler, tracking their status and monitoring for progress. Application Master itself runs as a normal *container*.
- Tasktracker
  - **NodeManager (NM)** - A new per-node slave is responsible for launching the applications' containers, monitoring their resource usage (cpu, memory, disk, network) and reporting to the Resource Manager.
- YARN maintains compatibility with existing MapReduce applications and users.

Homework & Project

- Homework Example:
  - Programming Assignment:
    - Install Hadoop MapReduce on your laptop, run its wordcount MR program and report runtime for two datasets of different sizes
  - Reading Assignment:
    - read two papers in the topic area (Given 6 papers for selection)

- **Interesting Projects:**
  - How to solve real world problems using Hadoop MapReduce Cloud/architecture (Application Level)
  - How to improve the performance of MapReduce program execution (Distributed OS/Middleware Level)

Interesting Projects

- Optimization of Streaming Graph Systems
  - Design benchmarks for comparing performance of Gigraph, Pregel, Hama, GraphX/Spark, etc.
  - Improving existing solutions with optimizations
  - Develop real world applications that can benefit from Hadoop Graph Systems.

- Graph Parallel Optimized Graph Partitioning Algorithms
  - Compare and beat random partition and simple hash partitioning.

Interesting Projects

- Hadoop MapReduce Configuration Manager
  - Study and compare performance differences for some alternative configuration parameters
  - Understand the config tuning points
  - Develop rule based configuration manager
- Spark Configuration Manager
  - Similar design process
- MapReduce benchmark
  - A representative set of MapReduce workloads
    - Word Count, Sort, Grep
    - K-means, SVM
  - A representative collections of datasets
    - Different Sizes, features, etc.

Terabyte Sort Benchmark

- Started by Jim Gray at Microsoft in 1998
- Sorting 10 billion 100 byte records
  - The sort benchmark specifies the input data (10 billion 100 byte records), which must be completely sorted and written to disk.
- Previous records was 297 seconds
- Hadoop won the general category in 209 seconds
  - 910 nodes
  - 2 quad-core Xeons @ 2.0Ghz / node
  - 4 SATA disks / node
  - 8 GB ram / node
  - 1 gb ethernet / node
  - 40 nodes / rack
  - 8 gb ethernet uplink / rack
- The sort used 1800 maps and 1800 reduces and allocated enough memory to buffers to hold the intermediate data in memory.
- Only hard parts were:
  - Getting a total order
  - Converting the data generator to map/reduce

Lecture Summary

- MapReduce's data-parallel programming model hides complexity of distribution and fault tolerance
- Principal philosophies:
  - *Make it scale*, so you can throw hardware at problems
  - *Make it cheap*, saving hardware, programmer and administration costs (but requiring fault tolerance)
- Hive and Pig further simplify programming
- MapReduce is not suitable for all problems, but when it works, it may save you a lot of time

Questions