MapReduce

Data Intensive Computing

- “Data-intensive computing is a class of parallel computing applications which use a data parallel approach to processing large volumes of data typically terabytes or petabytes in size and typically referred to as Big Data”

  -- Wikipedia

- Sources of Big Data
  - Walmart generates 267 million item/day, sold at 6,000 stores
  - Large Synoptic survey telescope captures 30 terabyte data/day
  - Millions of bytes from regular CAT or MRI scan

Adapted from Prof. Bryant’s slides @CMU

How can we use the data?

- Derive additional information from analysis of the big data set
    - business intelligence: targeted ad deployment, spotting shopping habit
    - Scientific computing: data visualization
    - Medical analysis: disease prevention, screening

Adapted from Prof. Bryant's slides @CMU

So Much Data

- Easy to get
  - Explosion of Internet, rich set of data acquisition methods
  - Automation: web crawlers
- Cheap to Keep
  - Less than $100 for a 2TB disk

Spread data across many disk drives

- Hard to use and move
  - Process data from a single disk --> 3-5 hours
  - Move data via network --> 3 hours - 19 days

Adapted from Prof. Bryant's slides @CMU

Challenges

- Communication and computation are much more difficult and expensive than storage
- Traditional parallel computers are designed for fine-grained parallelism with a lot of communication
- low-end, low-cost clusters of commodity servers
  - complex scheduling
  - high fault rate

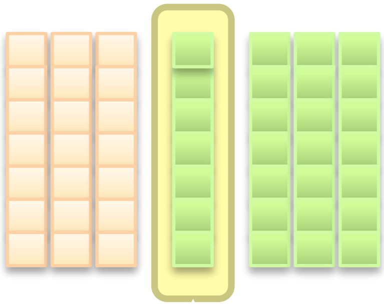
Data-Intensive Scalable Computing

- Scale out not up
  - data parallel model
  - divide and conquer
- Failures are common
- Move processing to the data
- Process data sequentially

However...

Fundamental issues

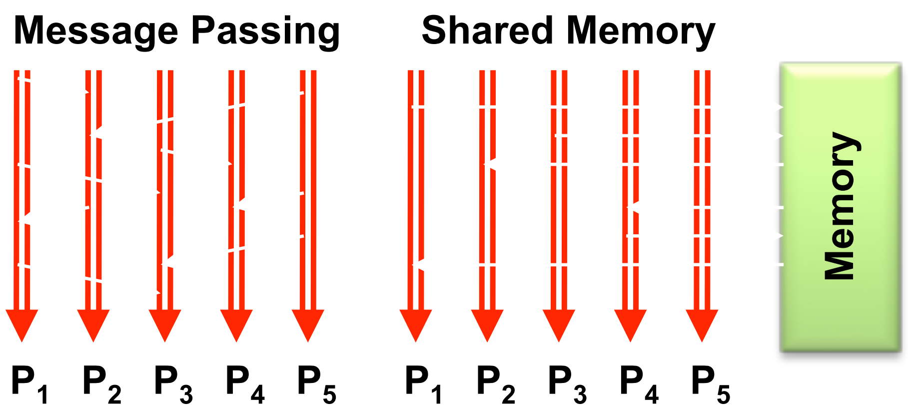
scheduling, data distribution, synchronization, inter-process communication, robustness, fault tolerance, …



Architectural issues

Flynn's taxonomy (SIMD, MIMD, etc.), network typology, bisection bandwidth UMA vs. NUMA, cache coherence

Different programming models



Different programming constructs

mutexes, conditional variables, barriers, …
masters/slaves, producers/consumers, work queues, …

Common problems

livelock, deadlock, data starvation, priority inversion…
dining philosophers, sleeping barbers, cigarette smokers, …

The reality: programmer shoulders the burden of managing concurrency…

slide from Jimmy Lin@U of Maryland

Typical Problem Structure

- Iterate over a large number of records
- Extract some of interest from each Parallelism Map function
- shuffle and sort intermediate results
- aggregate intermediate results Reduce function
- genera

Key idea: provide a functional abstraction for these two operations

slide from Jimmy Lin@U of Maryland

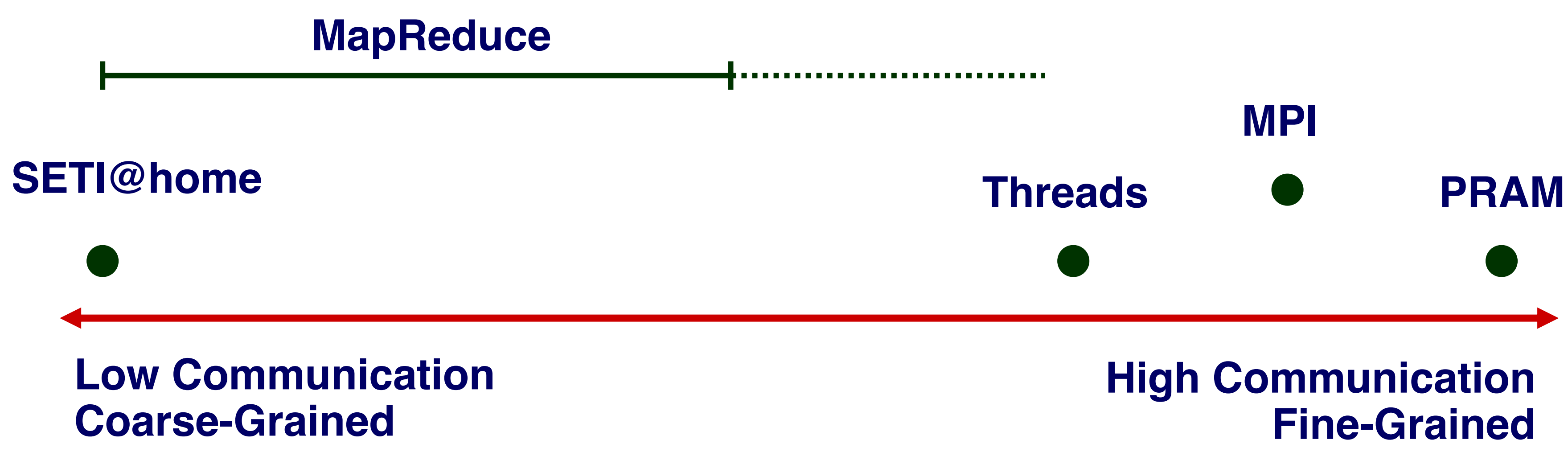
MapReduce

- A framework for processing parallelizable problems across huge data sets using a large number of machines
  - invented and used by Google [OSDI'04]
  - Many implementations
    - Hadoop, Dryad, Pig@Yahoo!
  - from interactive query to massive/batch computation
    - Spark, Giraff, Nutch, Hive, Cassandra

MapReduce Features

- Automatic parallelization and distribution
- Fault-tolerance
- I/O scheduling
- Status and monitoring

MapReduce v.s. Conventional Parallel Computers

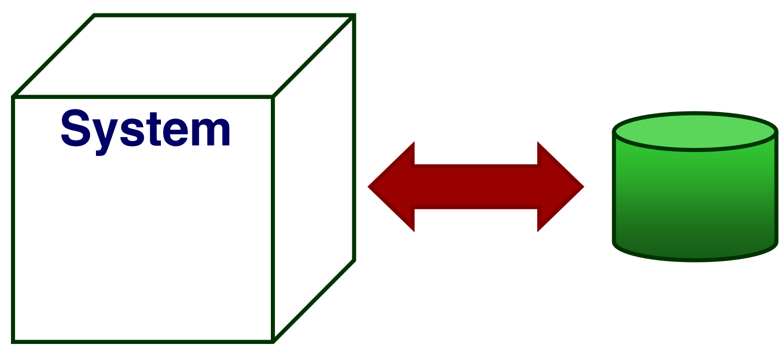


1. Coarse-grained parallelism
2. computation done by independent processors
3. file-based communication

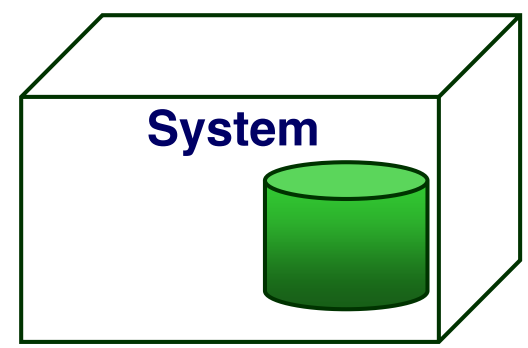
Adapted from Prof. Bryant's slides @CMU

Diff. in Data Storage

Conventional



MapReduce



C

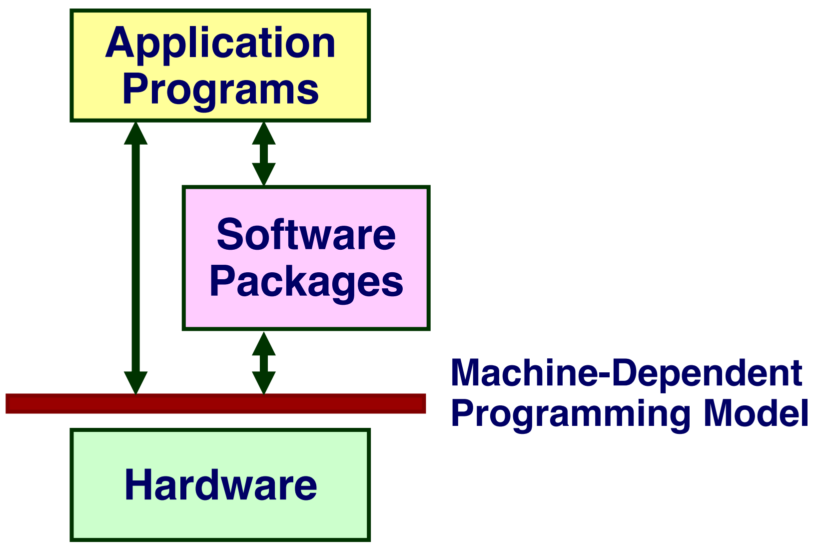
- Data stored in separate repository
- brought into system for computation

* Data stored locally to individual systems
* computation co-located with storage

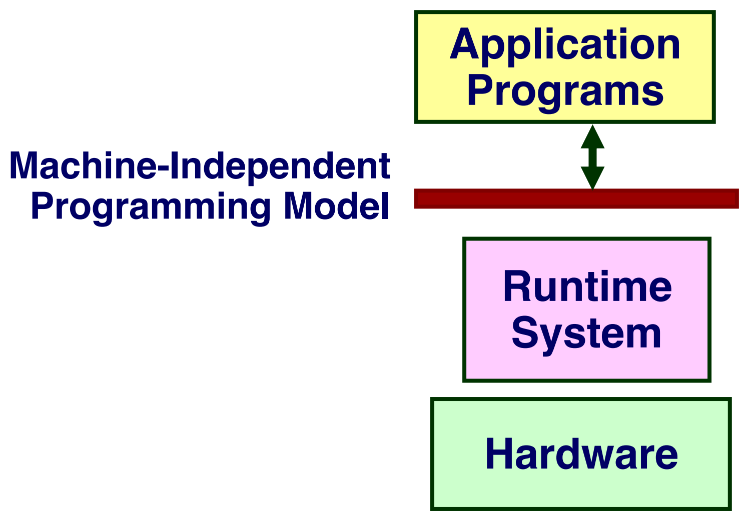
Adapted from Prof. Bryant’s slides @CMU

Diff. in Programming Models

Conventional



MapReduce



- Programs described at low level
- Rely on small number of software packages

* Application programs written in terms of high-level operations on data
* Run-time system controls scheduling, load balancing,...

Adapted from Prof. Bryant's slides @CMU

Diff. in Interaction

- Conventional
  - batch access
  - conserve machine rscs
  - admit job if specific rsc requirement is met
  - run jobs in batch mode

- MapReduce
  - interactive access
  - conserve human rscs
  - fair sharing between users
  - interactive queries and batch jobs

Adapted from Prof. Bryant's slides @CMU

Diff. in Reliability

- Conventional
  - restart from most recent checkpoint
  - bring down system for diagnosis, repair, or upgrades

- MapReduce
  - automatically detect and diagnosis errors
  - replication and speculative execution
  - repair or upgrade during system running

Programming Model

Input & Output: each a set of key/value pairs

Programmer specifies two functions:

map (in_key, in_value) -> list(out_key,intermediate_value)

Processes input key/value pair

Produces set of intermediate pairs

reduce (out_key, list(intermediate_value)) -> list(out_value)

Combines all intermediate values for a particular key

Produces a set of merged output values (usually just one)

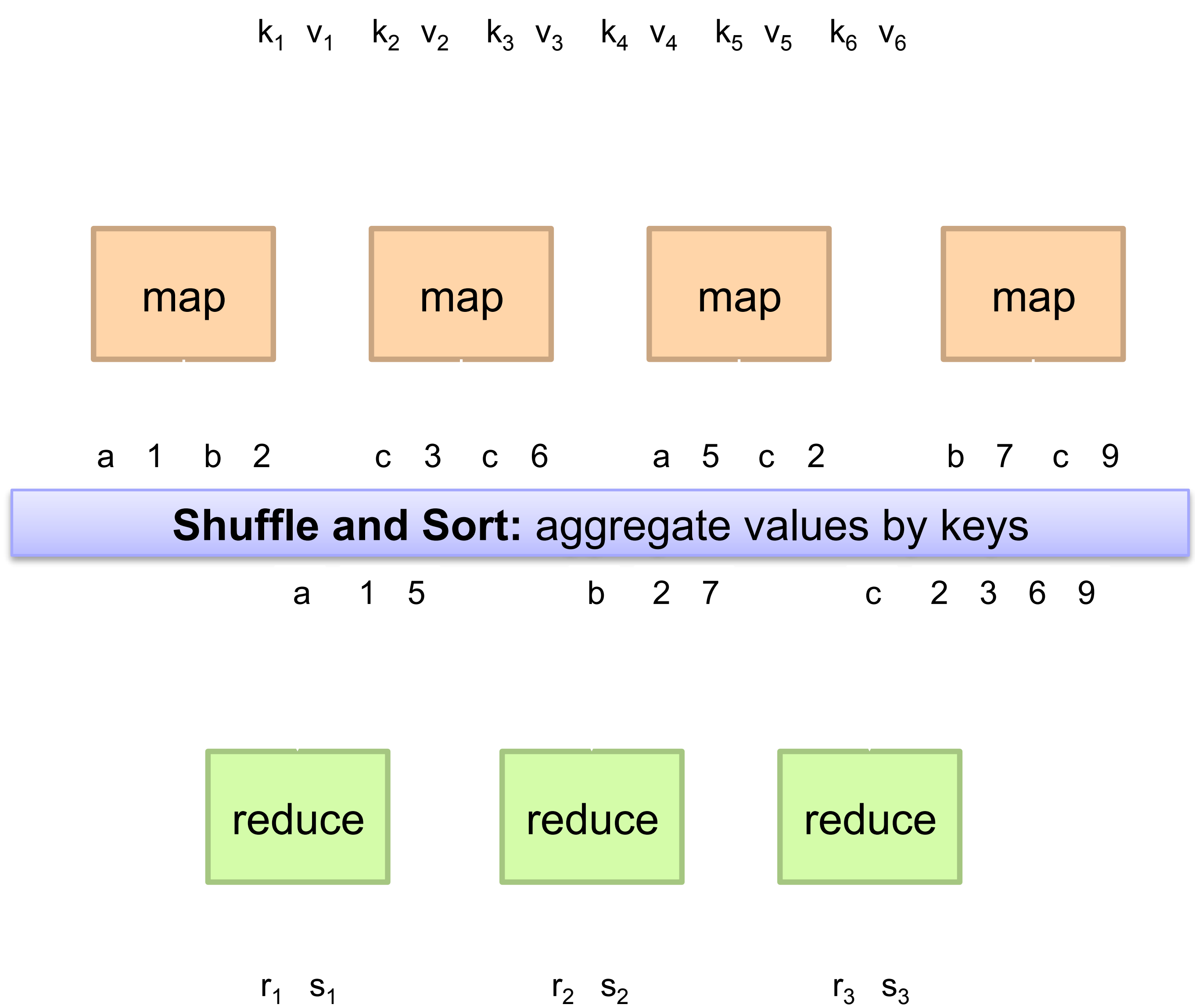
Inspired by similar primitives in LISP and other languages

slide from Dean et al. OSDI'04

Example: Count word occurrences

```
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");

reduce(String output_key, Iterator intermediate_values):
  // output_key: a word
  // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
  Emit(AsString(result));
```



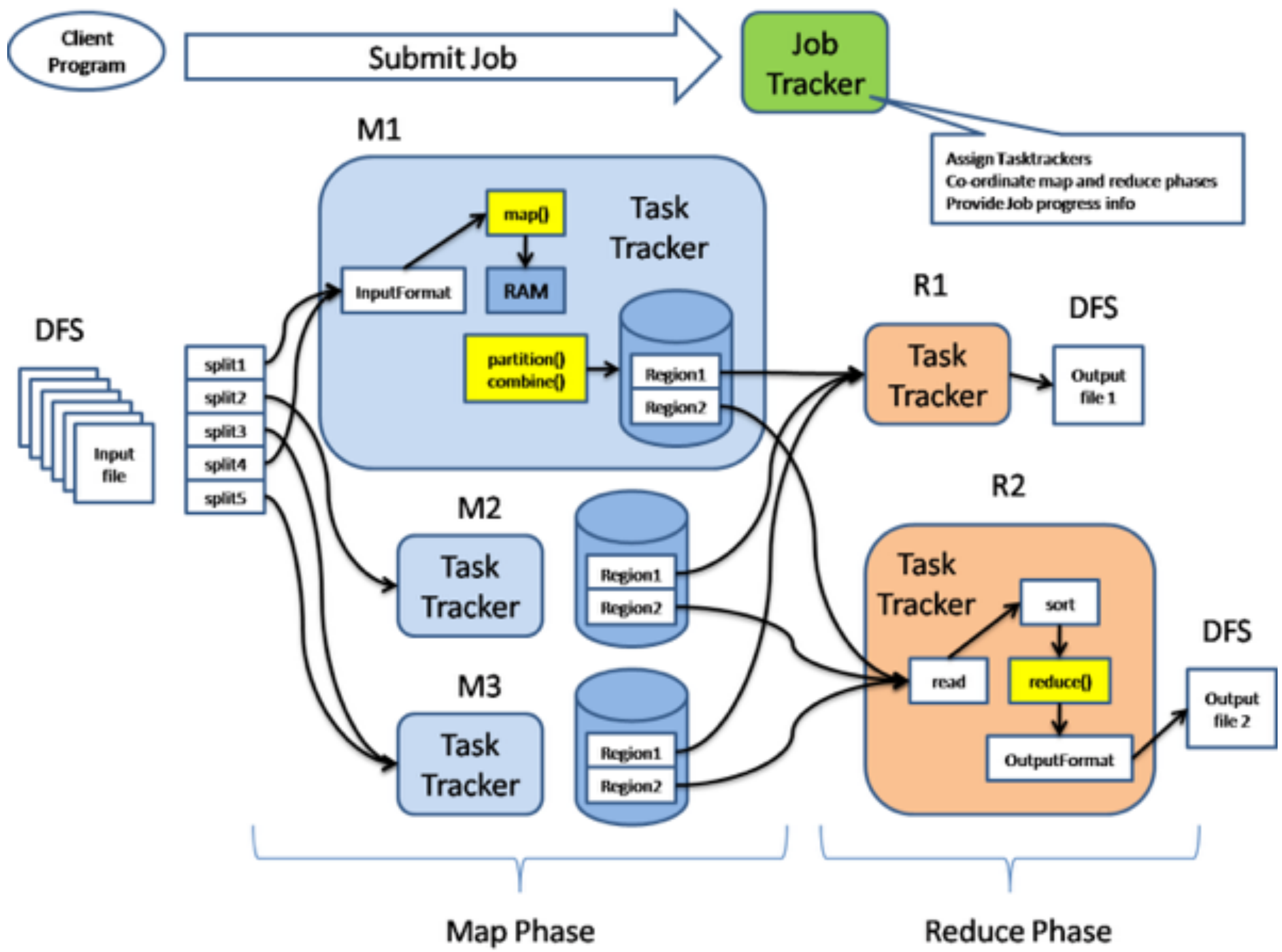
slide from Jimmy Lin@U of Maryland

MapReduce Runtime

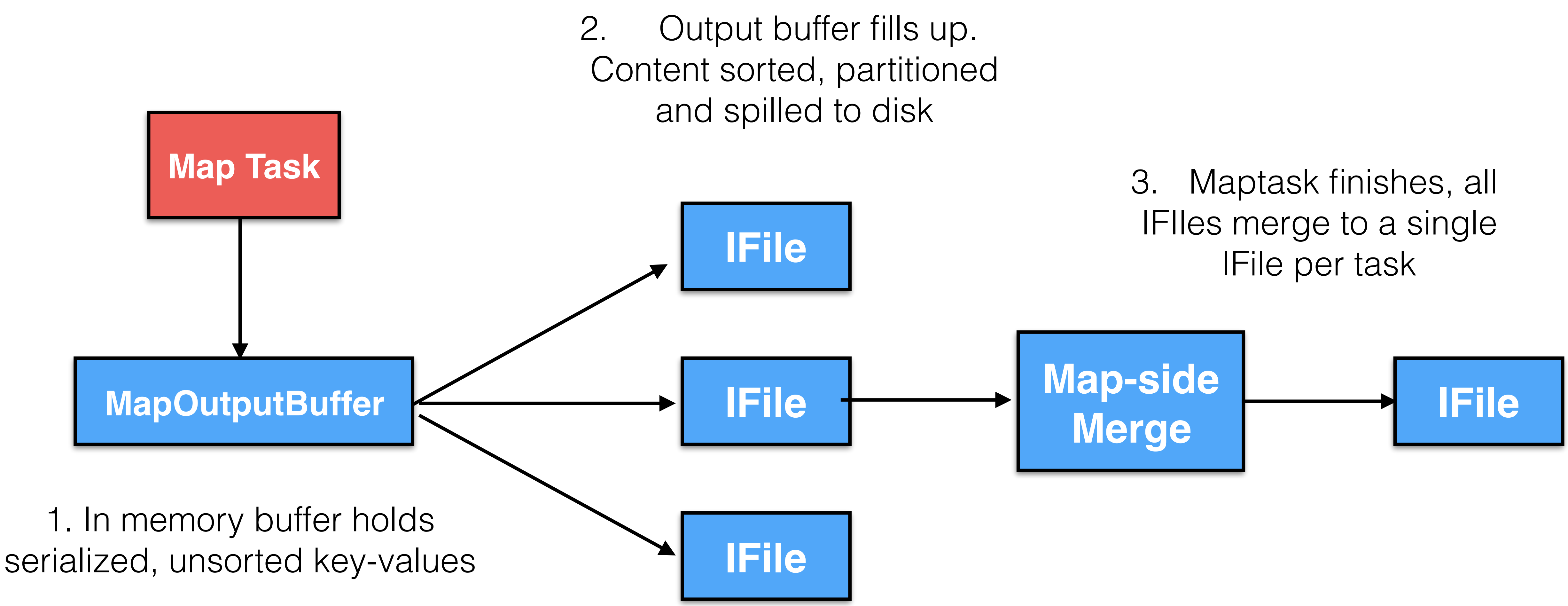
- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles “data distribution”
  - Moves the process to the data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed FS

slide from Jimmy Lin@U of Maryland

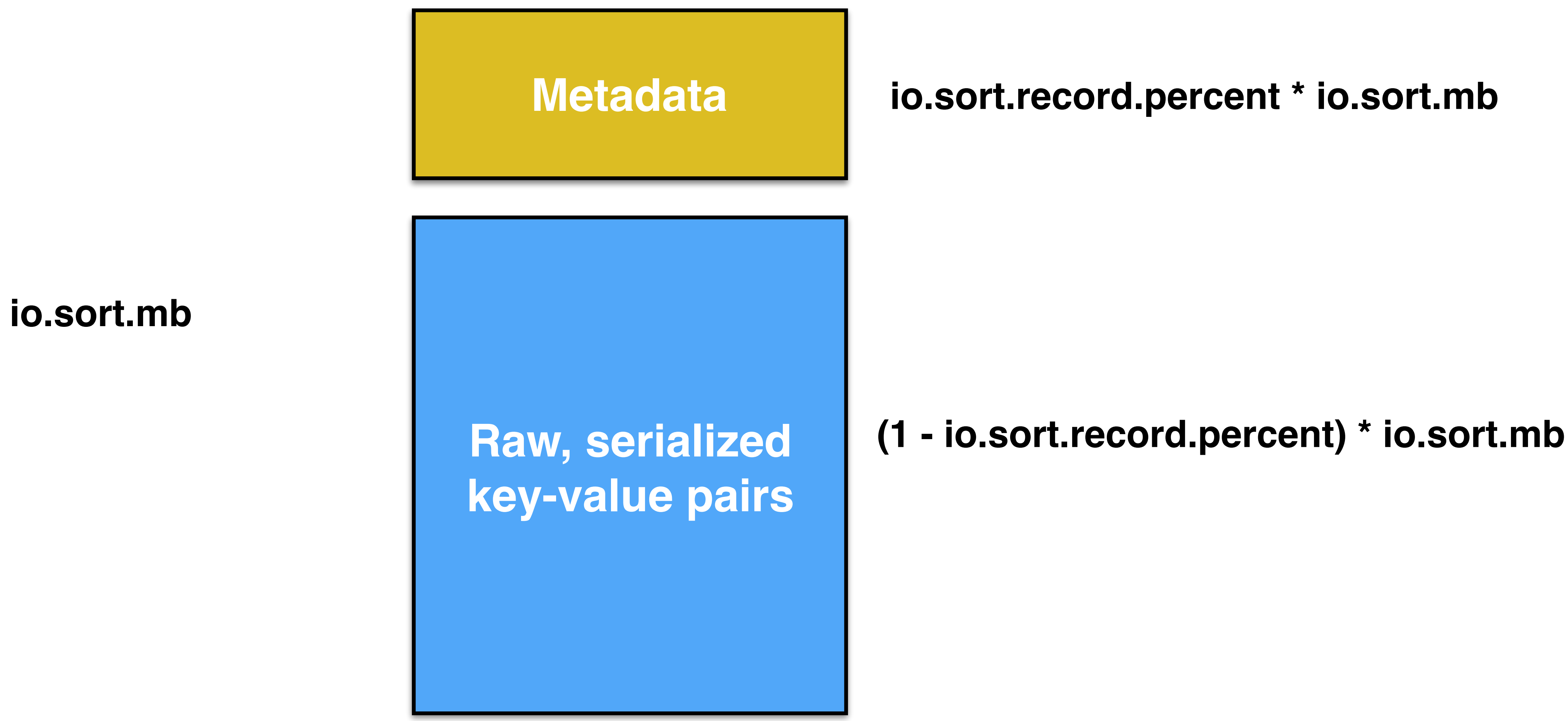
MapReduce Workflow



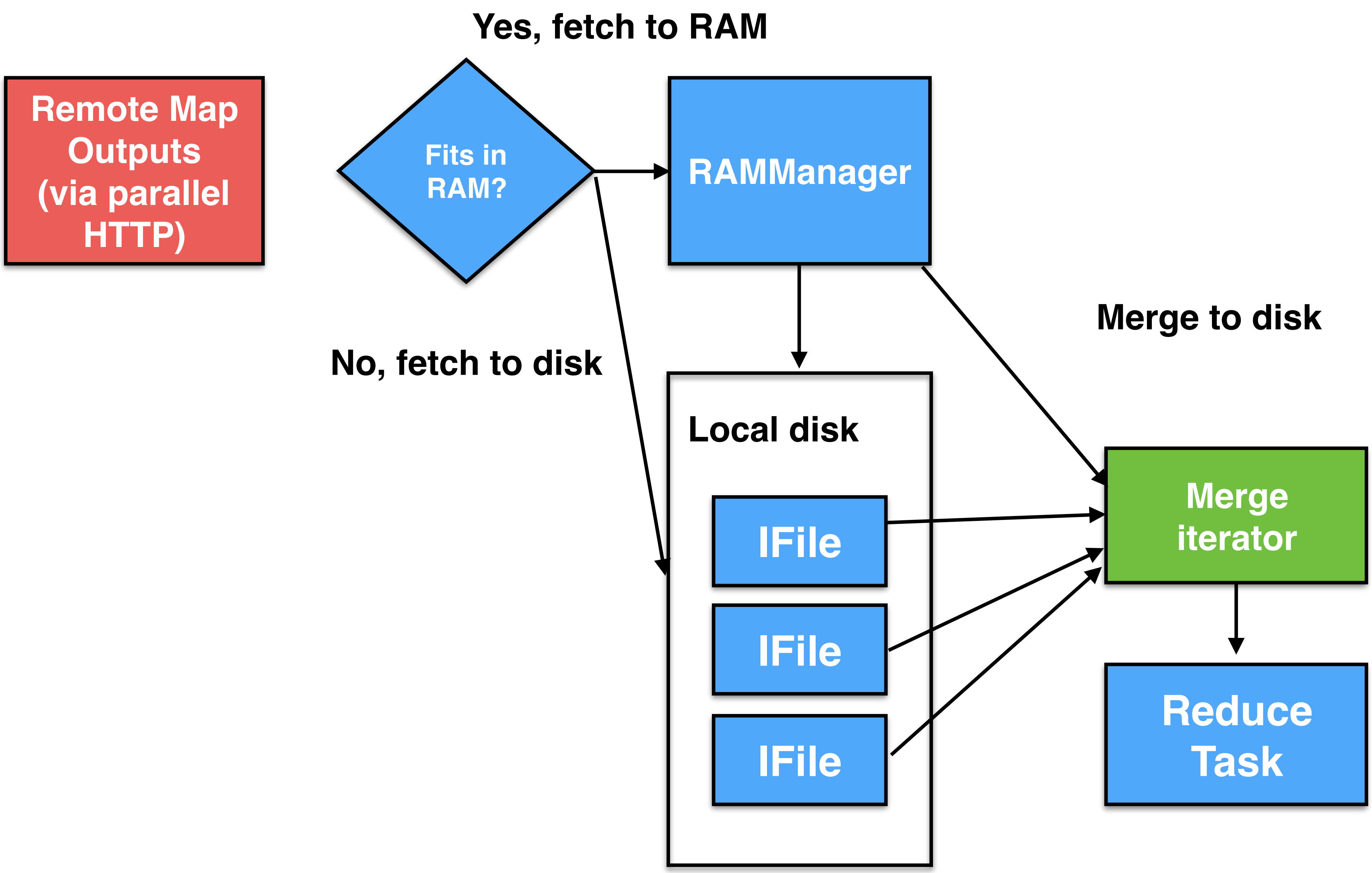
Map-side Sort/Spill



MapOutputBuffer



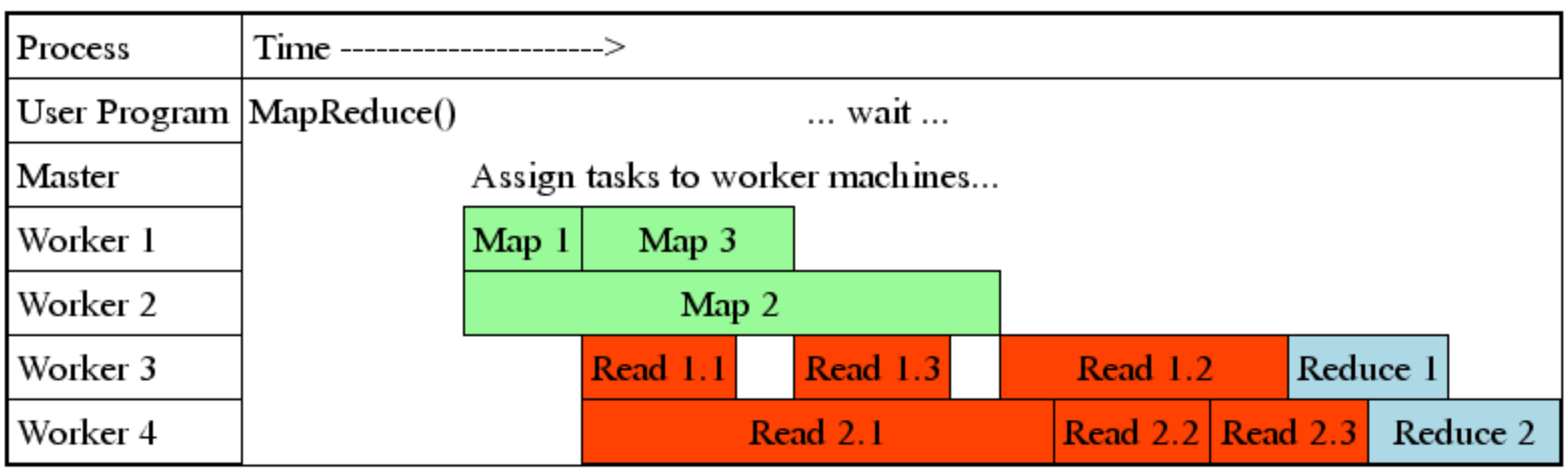
Reduce Merge



Task Granularity and Pipelining

Fine granularity tasks: many more maps than machines

- Minimizes time for fault recovery
- Can pipeline shuffling with map execution
-



slide from Dean et al. OSDI'04

MapReduce Optimizations

- # of map and reduce tasks on a node
  - A trade-off between parallelism and interferences
- Total # of map and reduce tasks
  - A trade-off between execution overhead and parallelism

Rule of thumb:

1. **adjust block size to make each map run 1-3 mins**
2. **match reduce number to the reduce slots**

MapReduce Optimizations (cont')

- Minimize # of IO operations
  - Increase MapOutputBuffer size to reduce spills
  - Increase ReduceInputBuffer size to reduce spills
  - Objective: avoid repetitive merges
- Minimize IO interferences
  - Properly set # of map and reduce per node
  - Properly set # of parallel reduce copy daemons

Fault Tolerance

- On worker failure
  - detect failure via periodic heartbeat
  - re-execute completed (data in local FS lost) and in-progress map tasks
  - re-execute in-progress reduce tasks
    - data of completed reduce is in global FS

Redundant Execution

- Some workers significantly lengthen completion time
  - resource contention form other jobs
  - bad disk with soft errors transfer data slowly
- Solution
  - spawn “backup” copies near the end of phase
  - the first one finishing commits results to the master, others are discarded

slide from Dean et al. OSDI’04

Distributed File System

- Move computation (workers) to the data
  - store data on local disks
  - launch workers (maps) on local disks
- A distributed file system is the answer
  - same path to the data
  - Google File System (GFS) and HDFS

GFS: Assumptions

- Commodity hardware over “exotic” hardware
- High component failure rates
    - Inexpensive commodity components fail all the time
- “Modest” number of HUGE files
- Files are write-once, mostly appended to
    - Perhaps concurrently
- Large streaming reads over random access
- High sustained throughput over low latency

slide from Jimmy Lin@U of Maryland

MapReduce Design

- GFS
  - File stored as chunks (64MB)
  - Reliability through replication (each chunk replicated 3 times)
- MapReduce
  - Inputs of map tasks match GFS chunks size
  - Query GFS for input location
  - Schedule map tasks to one of the replica as close as possible

Research in MapReduce

Issue: Fairness vs. Locality

- Place tasks on remote node due to fairness constraints

- A simple technique

  - Wait for 5 seconds before launch a remote task

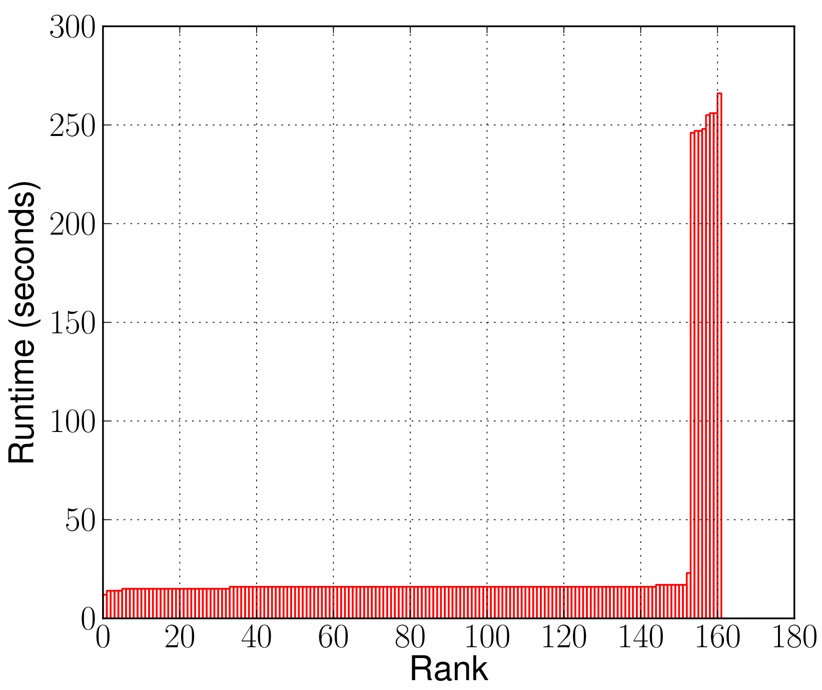
FromZaharia-EuroSys10

Issue: Heterogeneous Environment

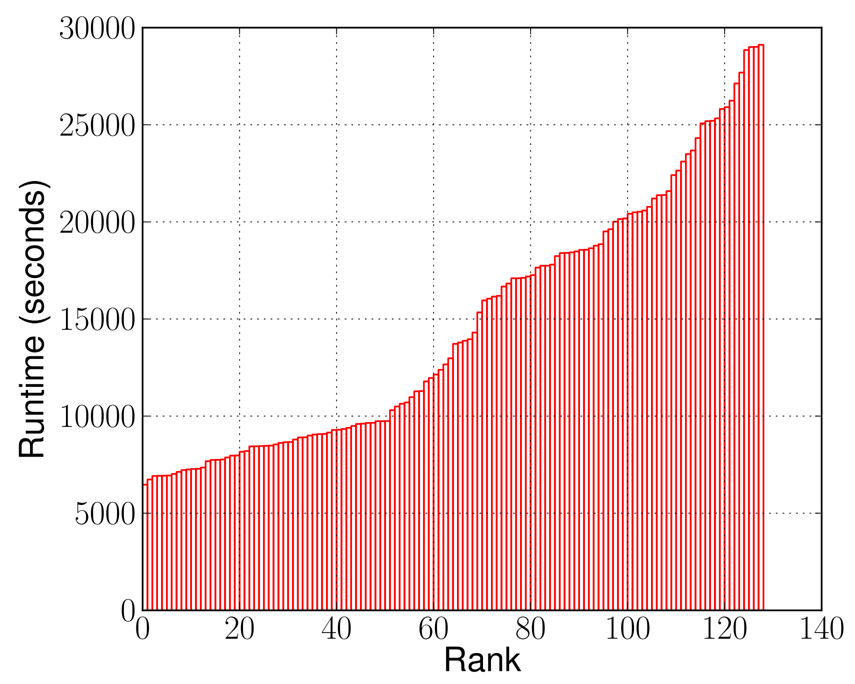
- MapReduce run speculative copy of tasks to address straggler issues
- Task execution progresses are inherently different on machines with different capabilities
- Speculative execution is not effective
- Solution: calibrate task progress with predictions on machine capabilities

From Zaharia-OSDI08

Data Skew



Map: heterogeneous data set



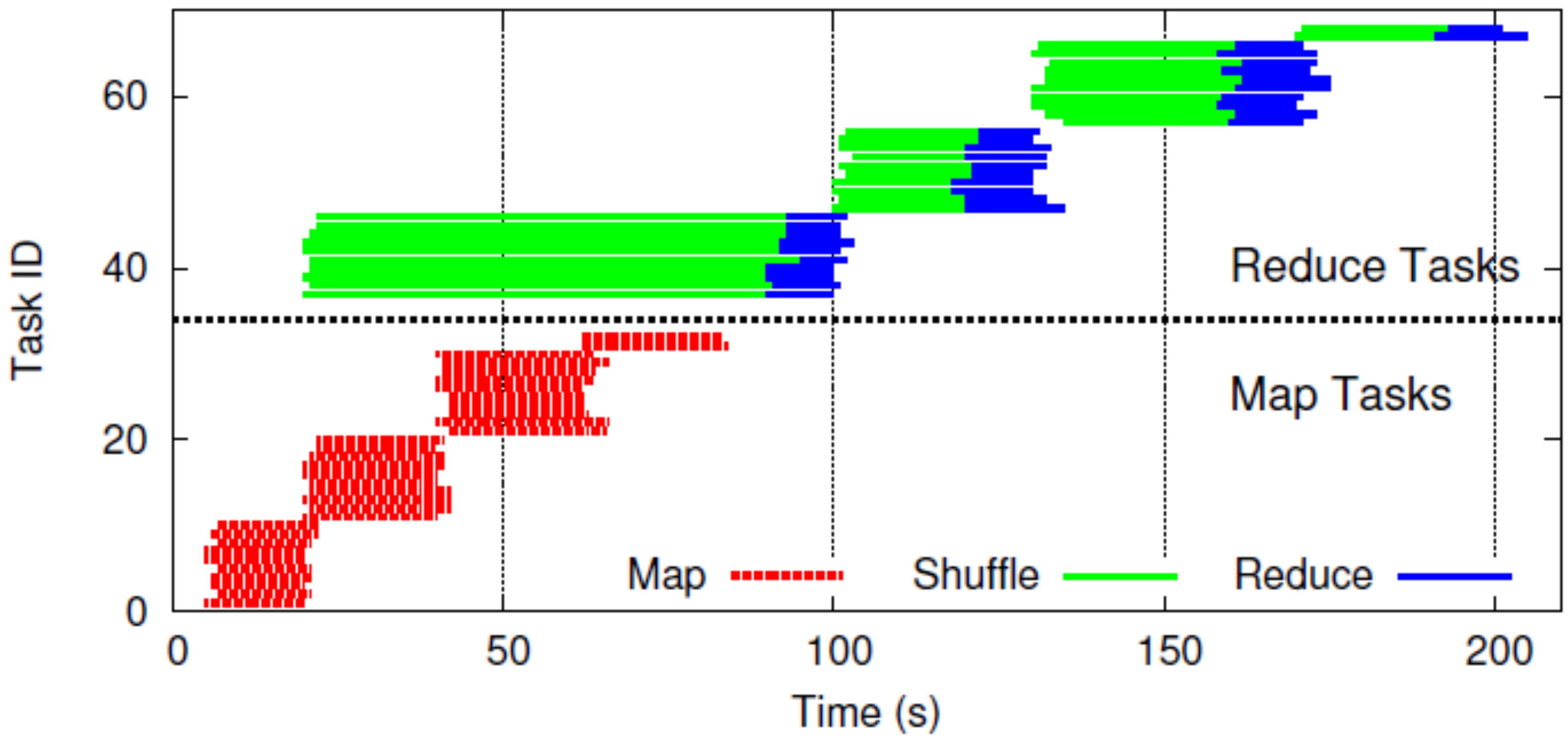
Reduce: expensive keys

From SkewTune-SIGMOD12

Issue: Hadoop Design

- Input data skew among reduce tasks
  - Non-uniform key distribution Different partition size
  - Lead to disparity in reduce completion time
- Inflexible scheduling of reduce task
  - Reduce tasks are created during job initialization
  - Tasks are scheduled in the ascending order of their IDs
  - Reduce tasks can not start even if their input partitions are available
- Tight coupling of shuffle and reduce
  - shuffle starts only the corresponding reduce is scheduled
  - Leave parallelism between and within jobs unexploited

A Close Look



Workload: tera-sort with 4GB dataset
Platform: 10-node Hadoop cluster
1 map and 1 reduce slots per node

Our Approach (ICAC’13)

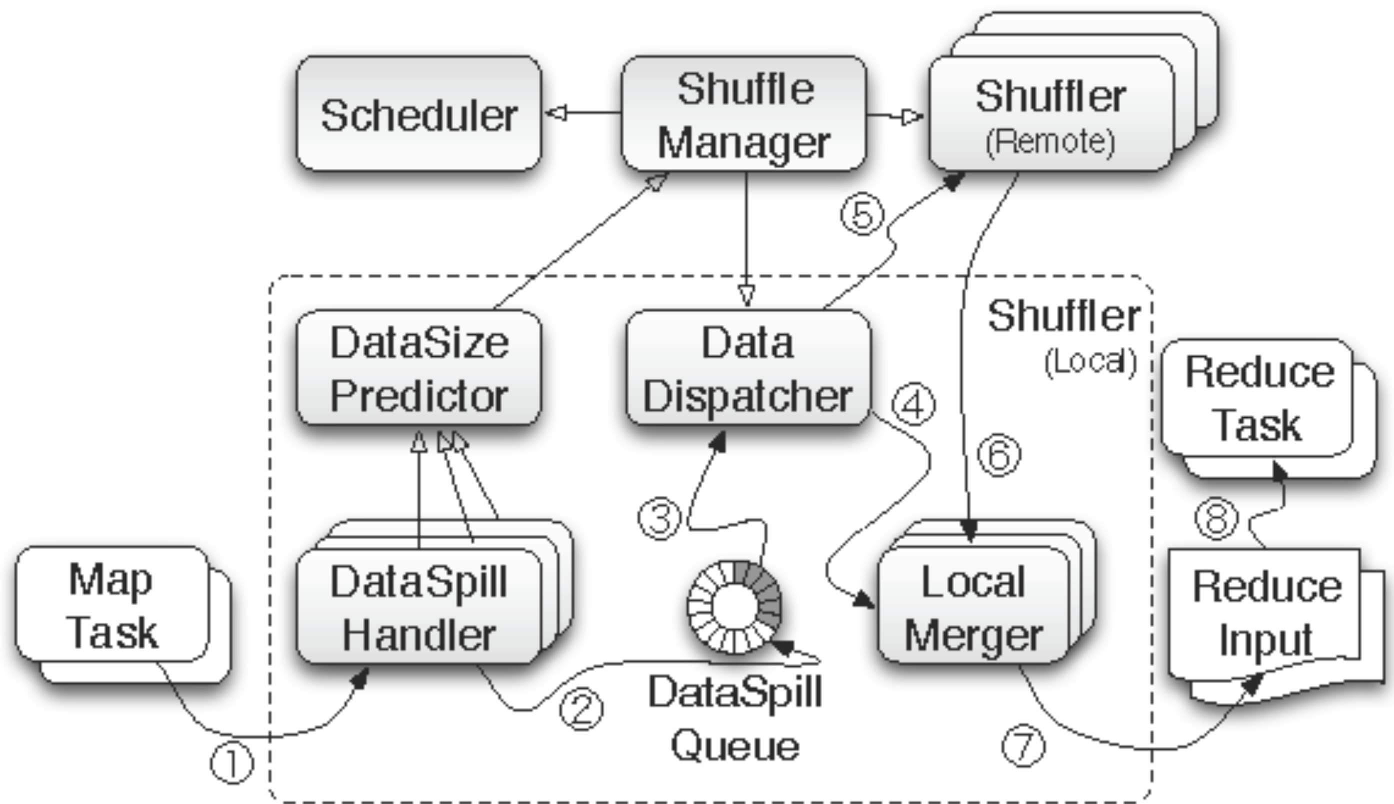
- Decouple shuffle phase from reduce tasks
  - Shuffle as a platform service provided by Hadoop
  - Pro-actively and deterministically push map output to different slave nodes
- Balancing the partition placement
  - Predict partition sizes during task execution
  - Determine which node should a partition been shuffled to
  - Mitigate data skew
- Flexible reduce task scheduling
  - Assign partitions to reduce tasks only when scheduled

Shuffle-on-Write

- Map output collection
  - MapOutputCollector
  - DataSpillHandler
- Data shuffling
  - Queuing and Dispatching
  - Data Size Predictor
  - Shuffle Manager

Map output merging

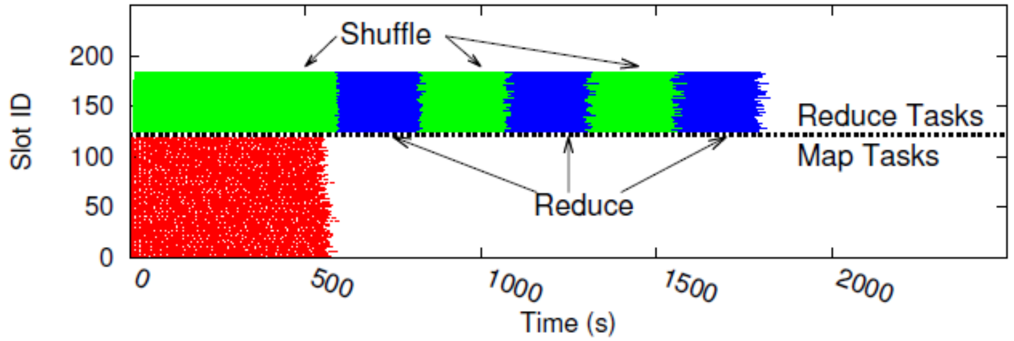
- Merger
- Priority-Queue merge sort



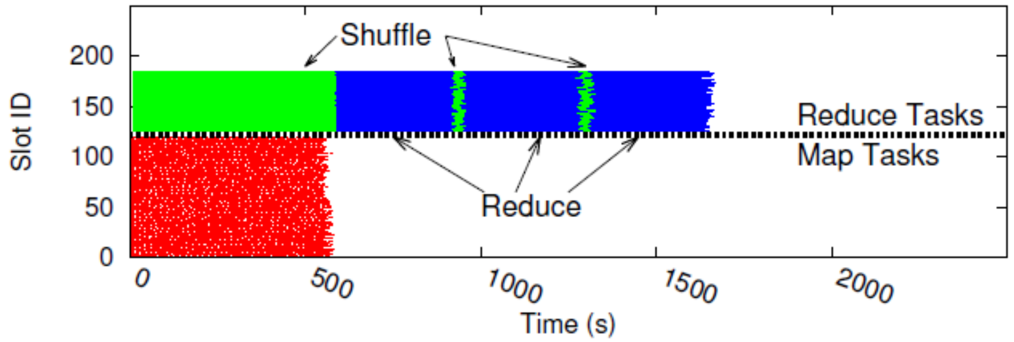
“shuffle” when Hadoop spills intermediate results

Results

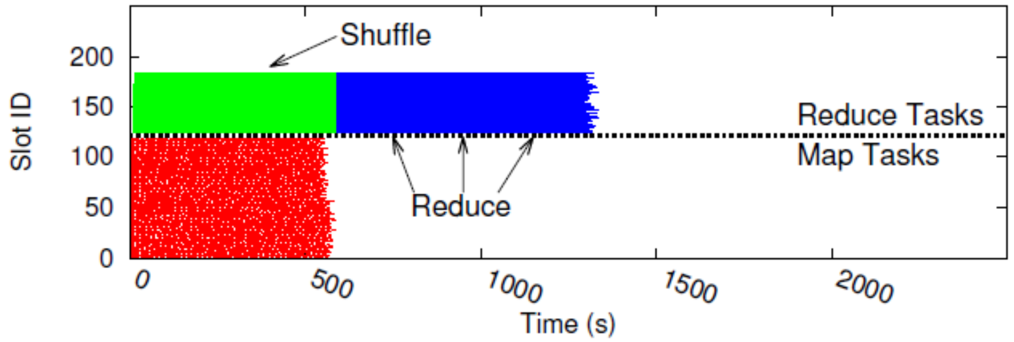
- Execution Trace
  - Slow start of Hadoop does not eliminate shuffle delay for multiple reduce wave
  - Overhead of remote disk access of Hadoop-A [SC’11]
  - iShuffle has almost no shuffle delay



(a) Hadoop.



(b) Hadoop-A.



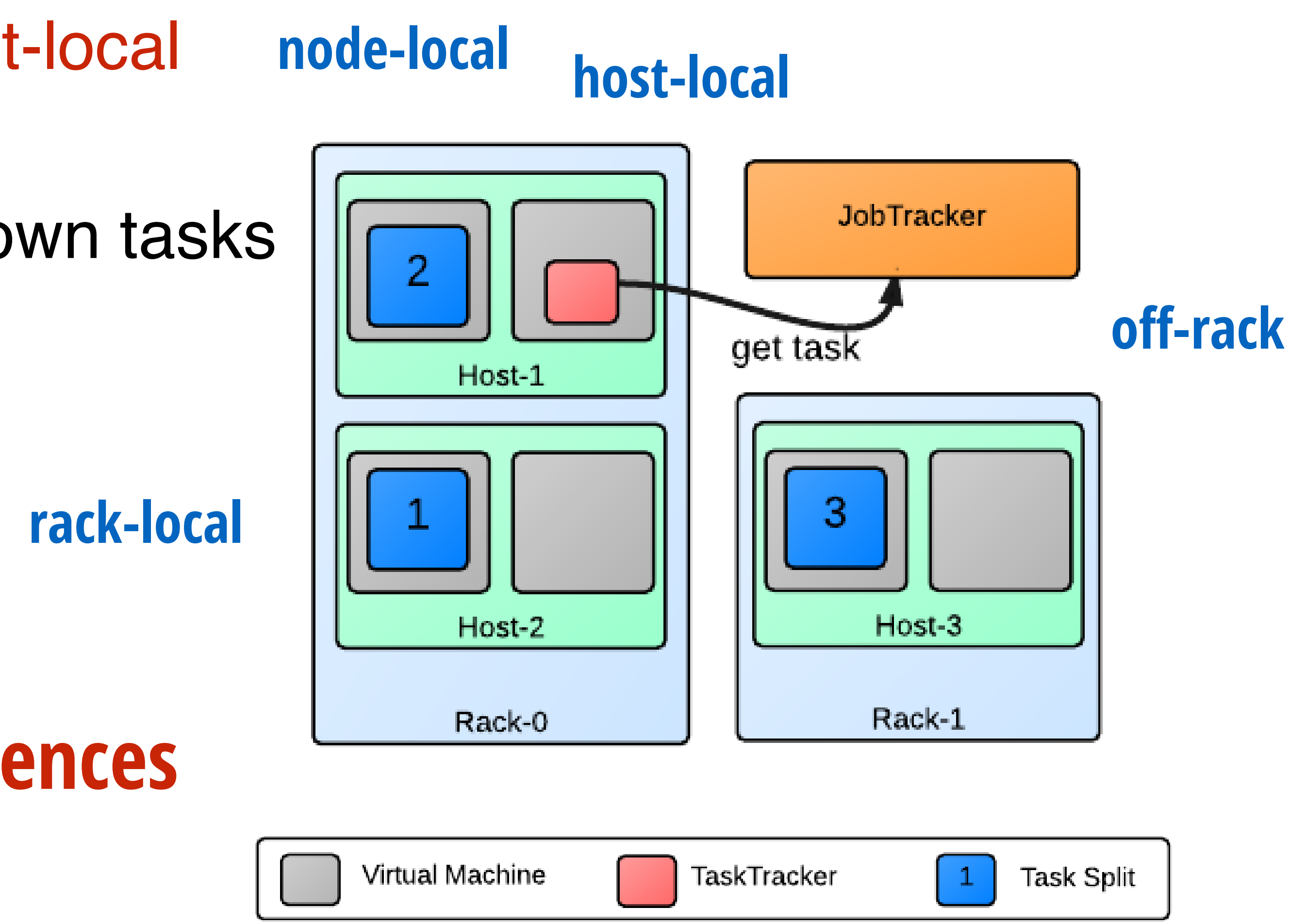
(c) iShuffle.

MapReduce in the Cloud?

- Amazon Elastic MapReduce
- Can possibly solve data skews
- Techniques for preserving locality ineffective
  - virtual topology ≠ physical topology
  - an extra layer of locality
    - off-rack, rack-local, node-local, host-local
- Unaware of interference in the cloud

MapReduce in the Cloud

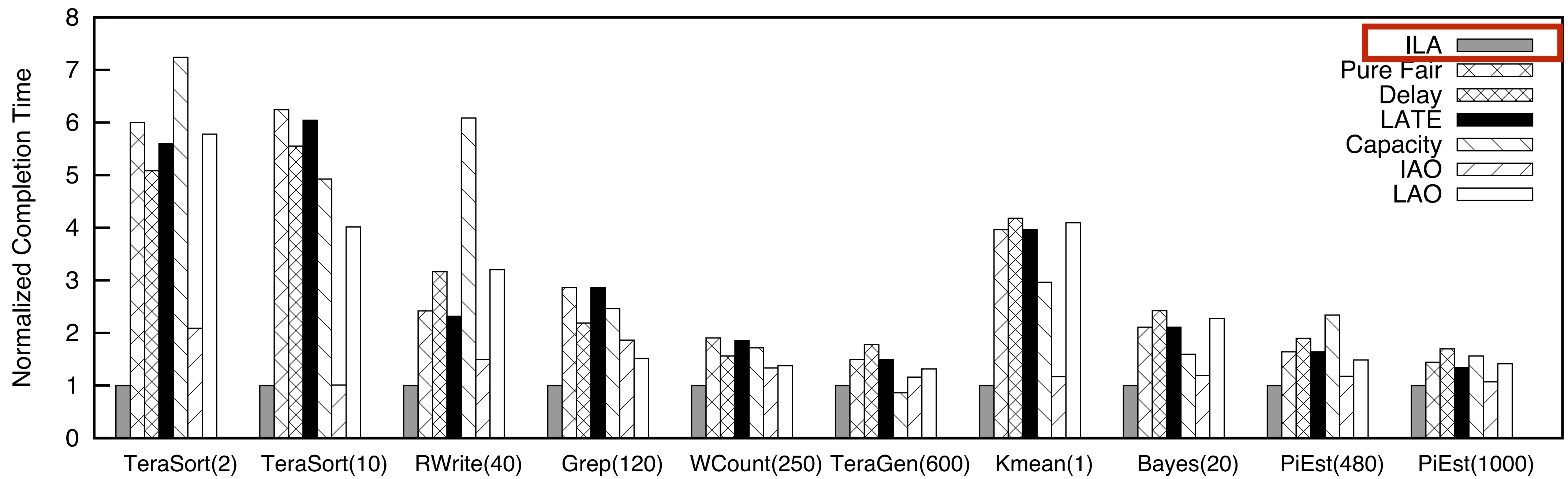
- An extra layer of locality
  - node-local, rack-local, and host-local
- Interferences significantly slow down tasks



Exploit locality and avoid interferences

Interference and Locality-Aware MapReduce Task Scheduling (HPDC'13)

- Export hardware topology information to Jobtracker
- Estimate interferences from finished tasks and host statistics



Significant improvement on job completion times

Performance Heterogeneity in Clouds

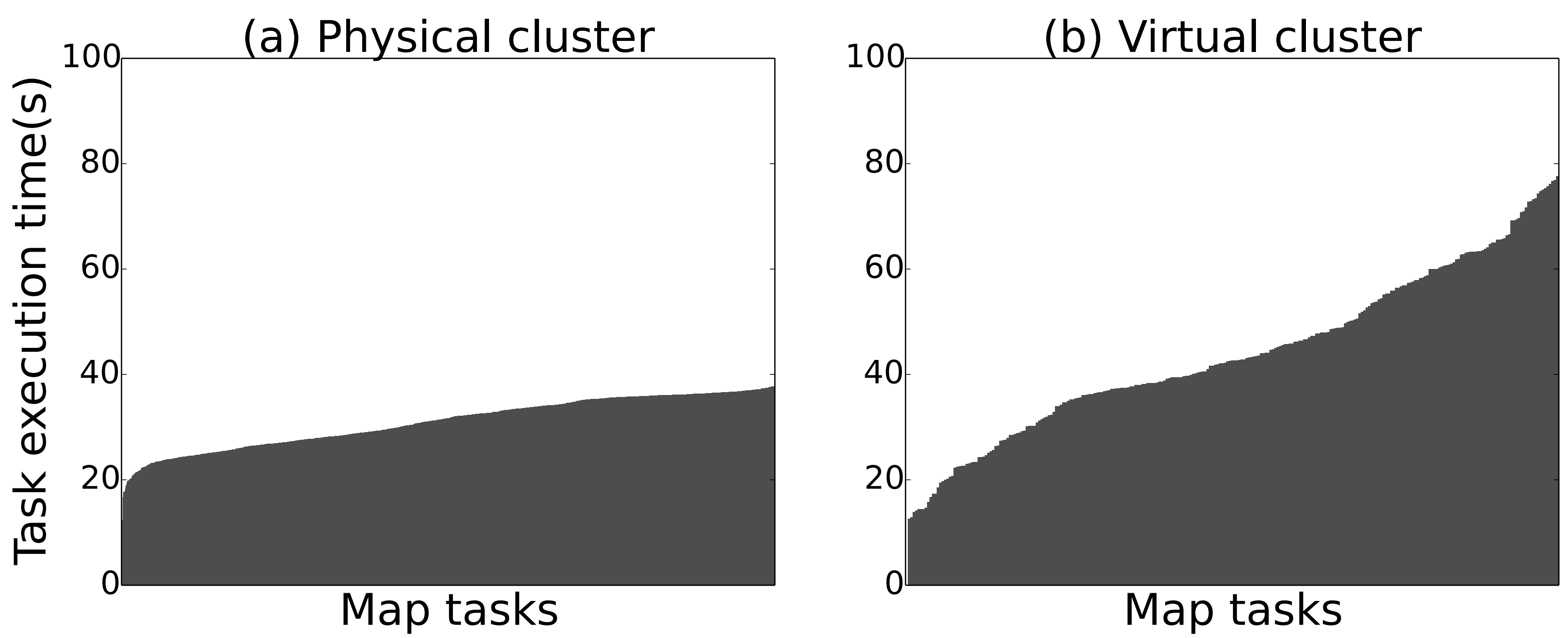
THE HARDWARE CONFIGURATION OF A HETEROGENEOUS CLUSTER

Machine model	CPU model	Memory	Disk	Number
PowerEdge T320	Intel Sandy Bridge 2.2GHz	24GB	1TB	2
PowerEdge T430	Intel Sandy Bridge 2.3GHz	128GB	1TB	1
PowerEdge T110	Intel Nehalem 3.2GHz	16GB	1TB	2
OPTIPLEX 990	Intel Core 2 3.4GHz	8GB	1TB	7

Hardware heterogeneity due to multiple generations of machines

Performance heterogeneity can also be due to multi-tenant interferences in the cloud

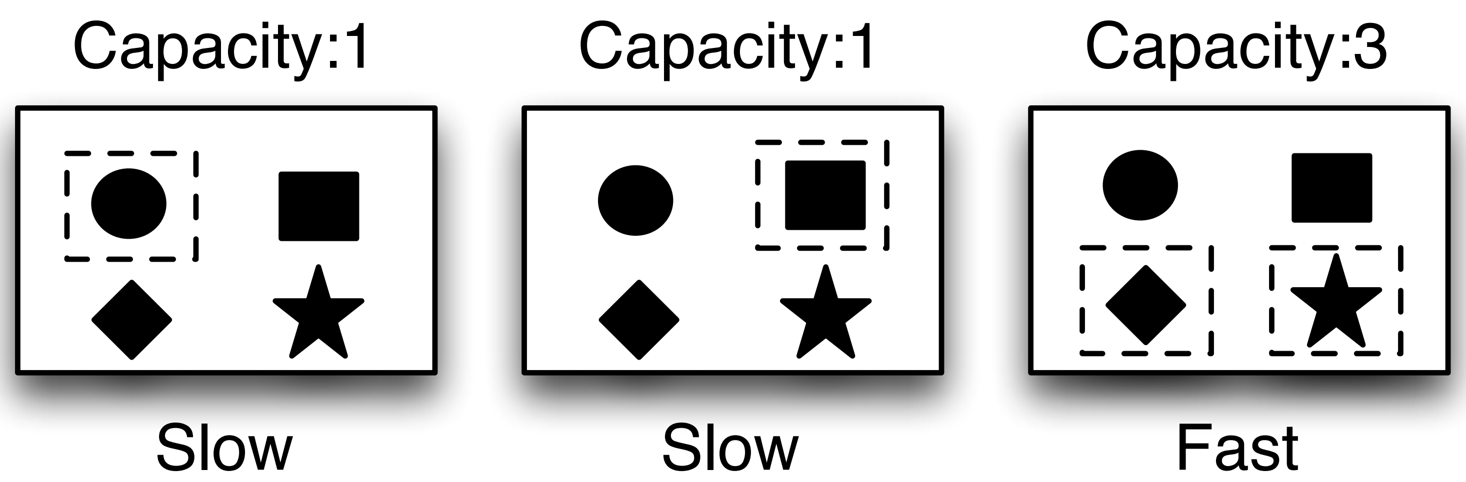
Imbalance Due to Performance Heterogeneity



2:1 **5:1**

fastest:slowest

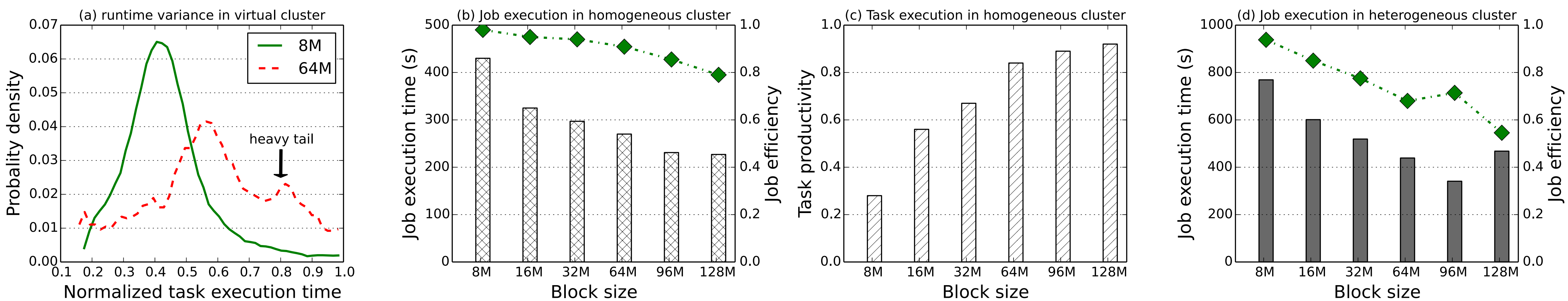
Load Balancing isn't Effective



Speculative execution or remote task execution is not effective for load balancing unless mappers are infinitely small

Mappers are not infinitely small and are statically bound to a HDFS block

Execution Overhead v.s. Load Balancing

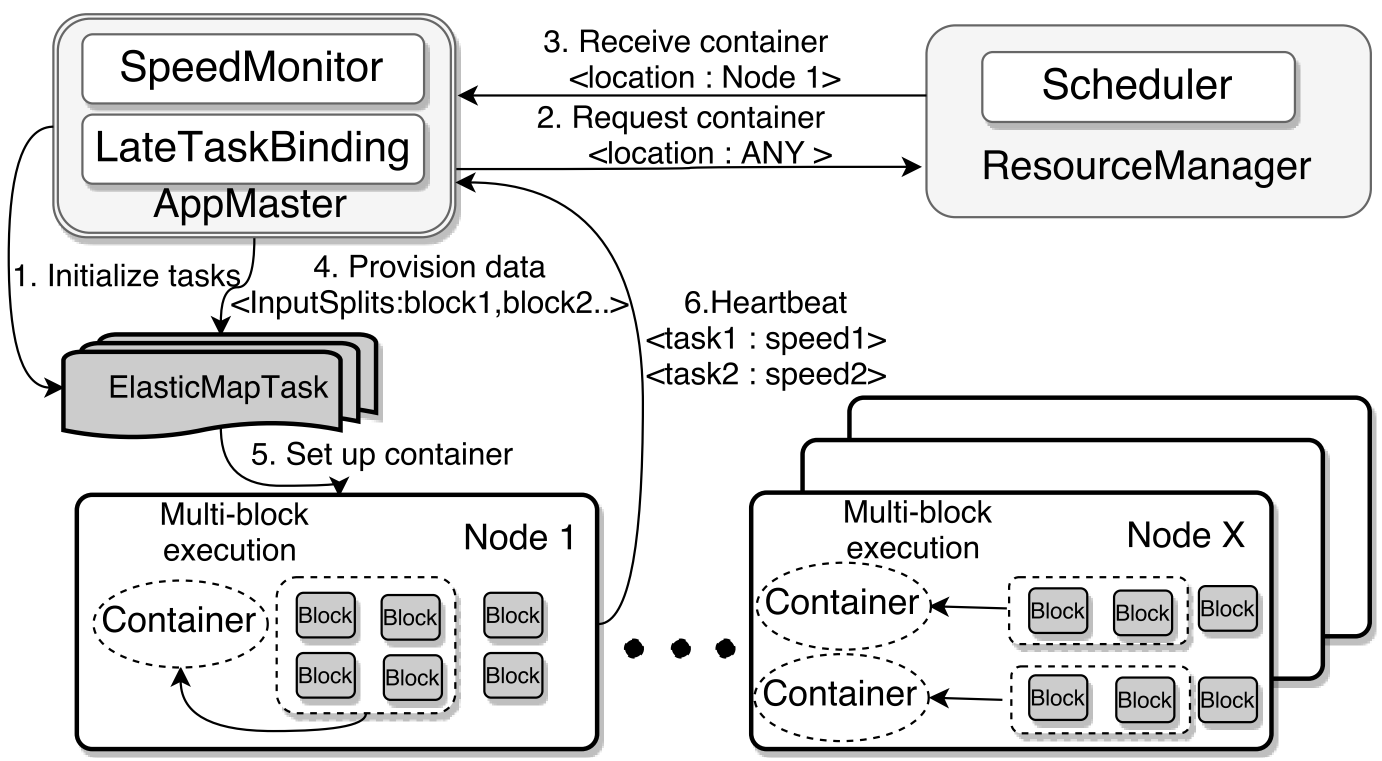


Productivity = Effective runtime/Total runtime

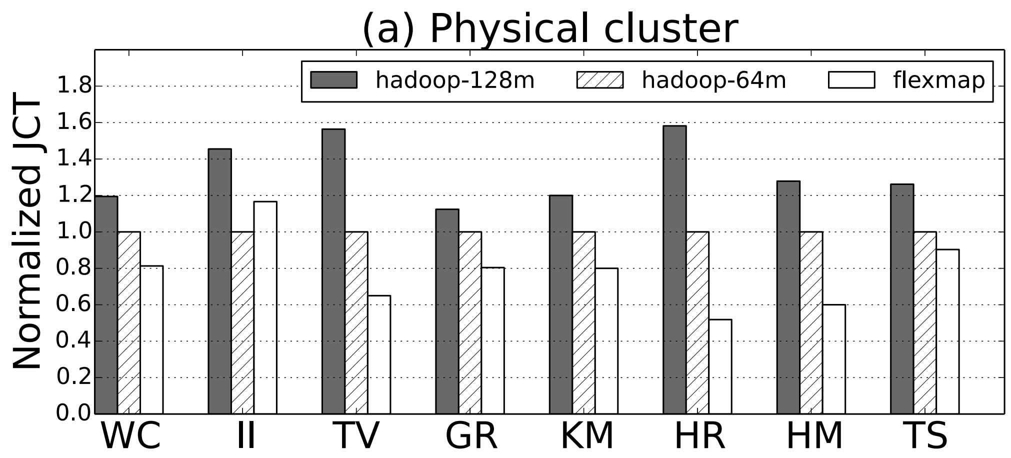
Efficiency = Serial time/Map phase time * # of slots

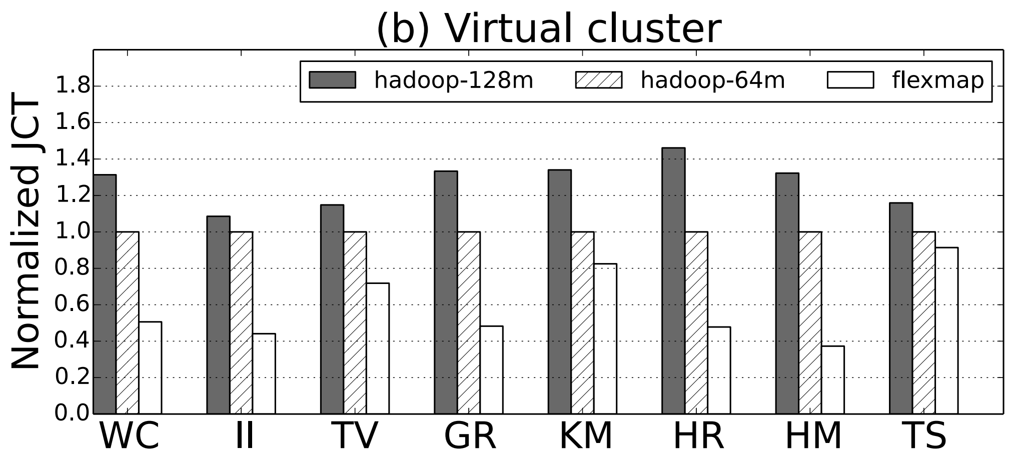
Elastic Mappers

- Idea: run large mappers on fast machines
- Approach: start with small mappers (8MB) and expand based on machine capacity

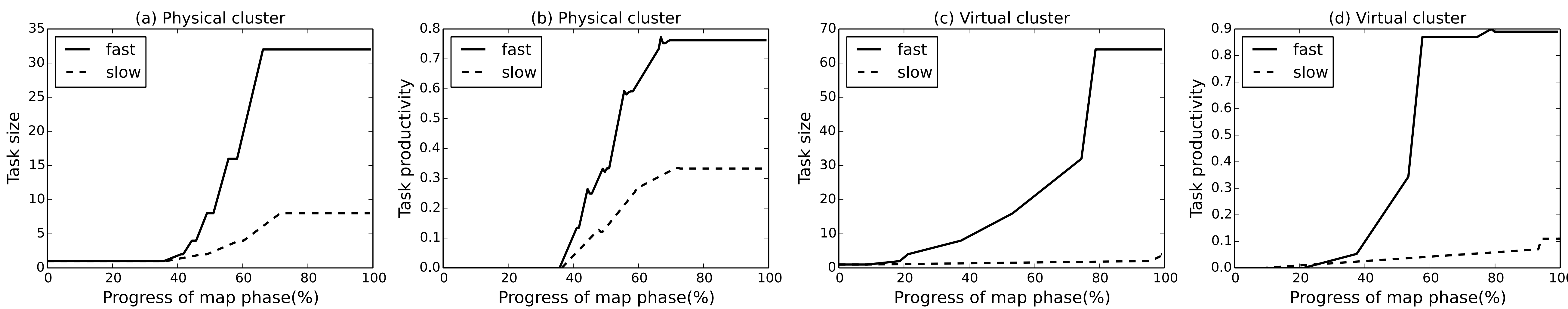


Improving Overall Performance





Expanding Mapper Size



Results on a 40-node Cluster