CS 764: Topics in Database Management Systems

Lecture 14: MapReduce

Xiangyao Yu

10/21/2020

Announcement

Mid-term course evaluation **DDL: 10/23**

Please submit project proposal to the review website **DDL: Oct 26**

Please submit a review for the guest lecture within 3 days after the lecture **DDL: Oct 28 11:59pm**

Today's Paper: MapReduce

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 find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose 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 parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards

OSDI 2004

Outline

Background

MapReduce

- Programming model
- Implementation
- Optimizations

MapReduce vs. Databases

Challenges in Distributed Programming

[Within a server] Multi-threading

[Across servers] Inter-server communication (MPI, RPC, etc.)

Fault tolerance

Load balancing

Scalability

Distributed Challenges in Databases?

[Within a server] Multi-threading

[Across servers] Inter-server communication (MPI, RPC, etc.)

- The interface is SQL, parallelism is invisible to users

Fault tolerance

- Logging and high availability, invisible to users

Load balancing

Scalability

- Shared-nothing databases are very scalable

Limitations of Distributed Databases

Programming model: SQL

Data format: Relational (i.e., structured)

Lack of support for failures during an OLAP query

MapReduce

MapReduce Programming Model

A user of the MapReduce library writes two functions:

Map function

- Input: <key, value>
- Output: list(<key, value>)

Reduce function

- Input: <key, list(value)>
- Output: list(value)

MapReduce Programming Model

A user of the MapReduce library writes two functions:

Map function

- Input: <key, value>
- Output: list(<key, value>)

Reducer function

- Input: <key, list(value)>
- Output: list(value)

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));
```

Other Application Examples

Grep:

- Map: emits a line if it matches the pattern
- Reduce: identity function—copy input to output

Other Application Examples

Grep:

- Map: emits a line if it matches the pattern
- Reduce: identity function—copy input to output

Count of URL access frequency:

- Map: emit **⟨URL, 1⟩**
- Reduce: adds values for the same URL and emits **⟨URL, total count⟩**

Other Application Examples

Grep:

- Map: emits a line if it matches the pattern
- Reduce: identity function—copy input to output

Count of URL access frequency:

- Map: emit **⟨URL, 1⟩**
- Reduce: adds values for the same URL and emits **⟨URL, total count⟩**

Reverse web-link graph:

- Map: outputs **⟨target, source⟩** for each target URL found in page source
- Reduce: concatenates sources associated with a given target **⟨target, list(source)⟩**

Other Application Examples

Grep:

- Map: emits a line if it matches the pattern
- Reduce: identity function—copy input to output

Count of URL access frequency:

- Map: emit **⟨URL, 1⟩**
- Reduce: adds values for the same URL and emits **⟨URL, total count⟩**

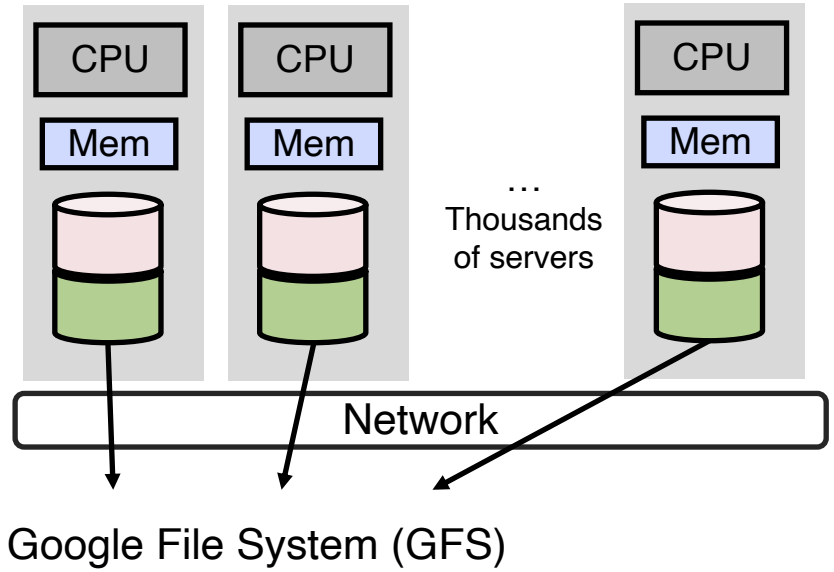
Reverse web-link graph:

- Map: outputs **⟨target, source⟩** for each target URL found in page source
- Reduce: concatenates sources associated with a given target **⟨target, list(source)⟩**

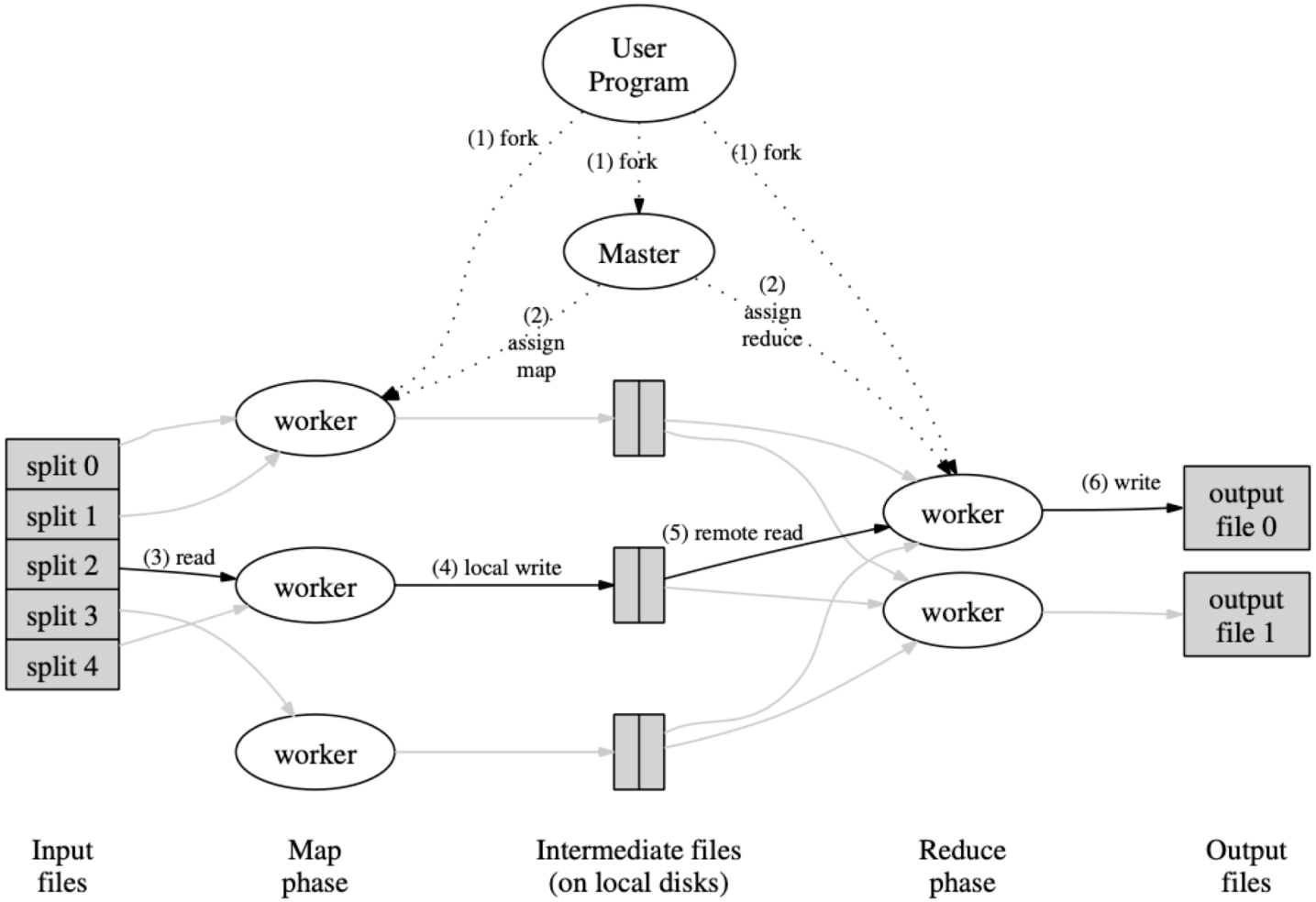
Inverted index:

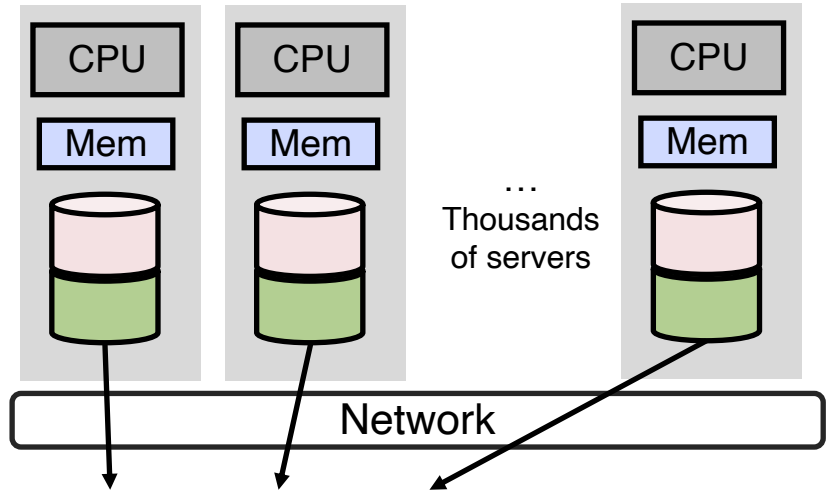
- Map: Emit **⟨word, doc ID⟩** for words in a document
- Reduce: for a word, sorts document IDs and emits **⟨word,list(doc ID)⟩**

Implementation



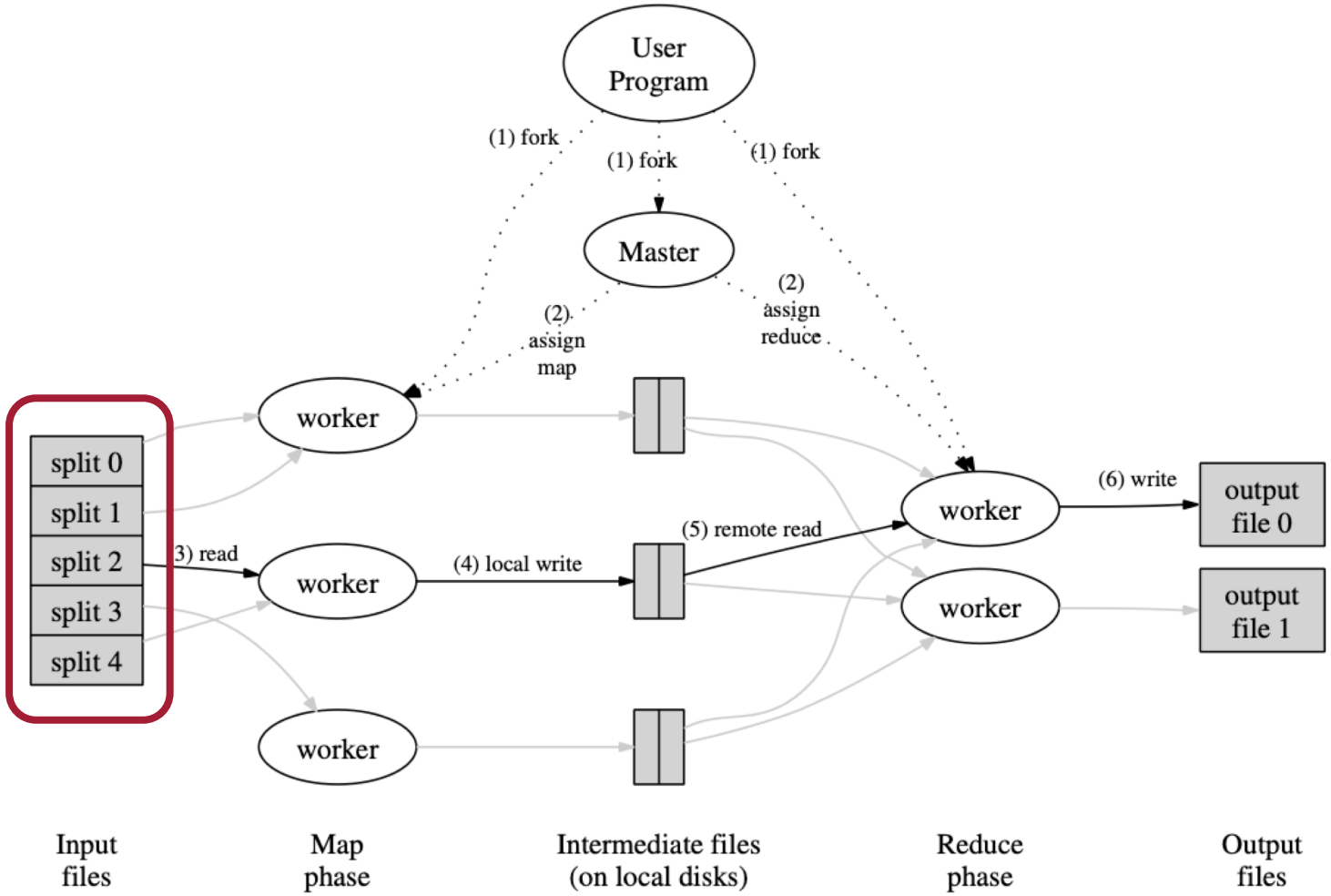
Implementation

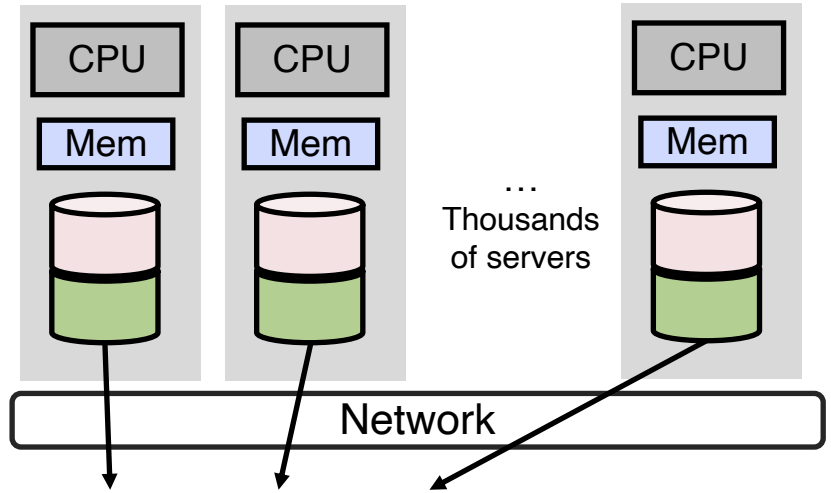




Google File System (GFS)

Implementation – Step 1

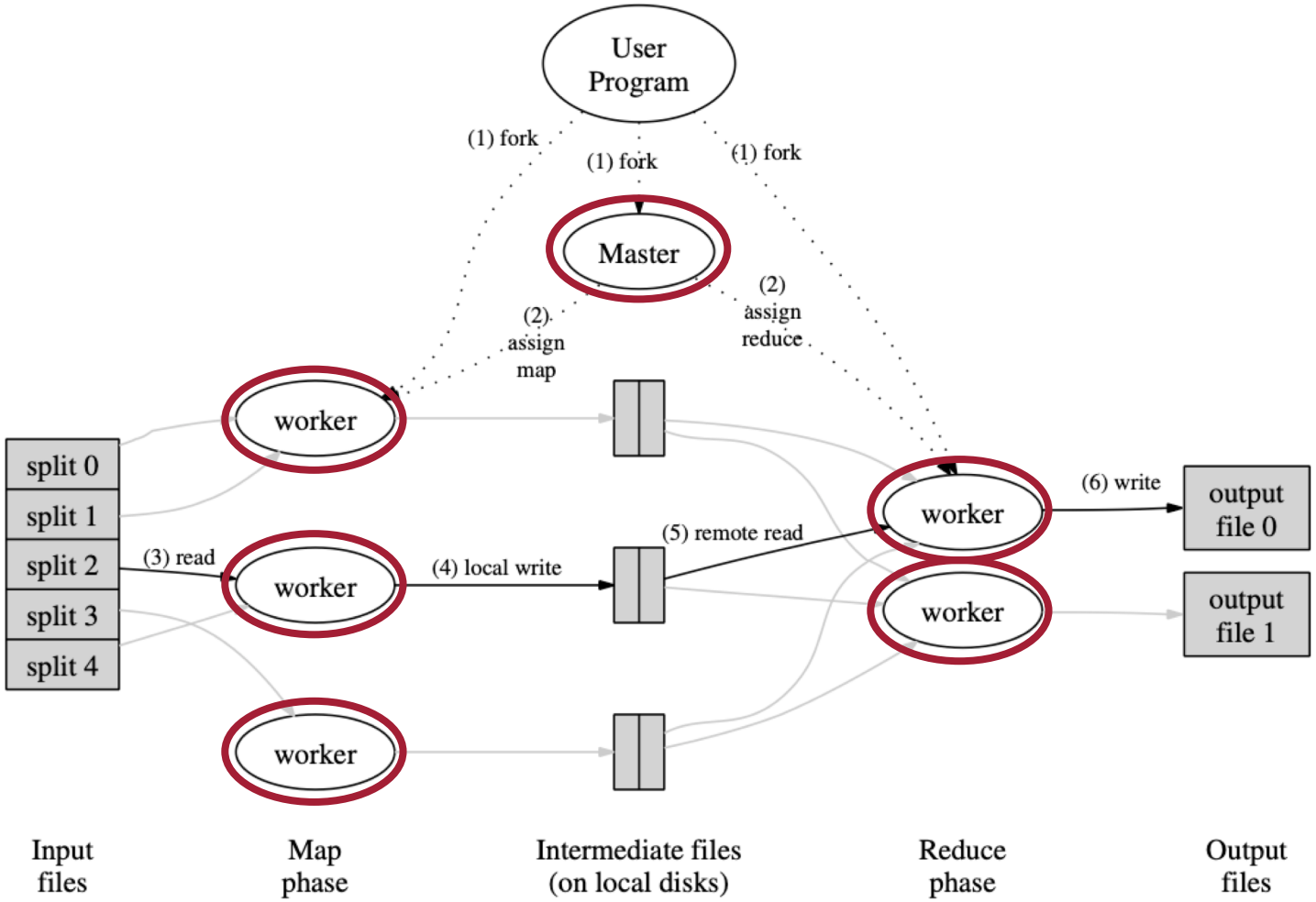


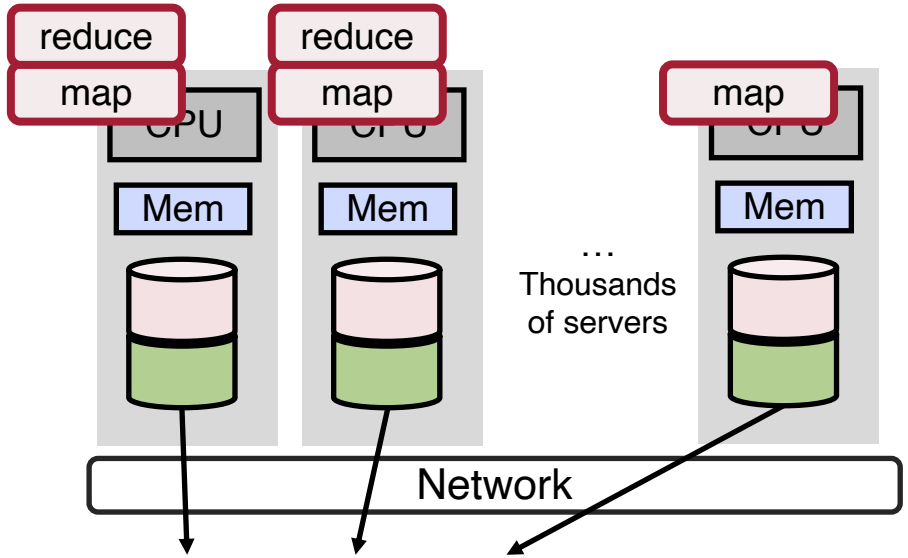


Google File System (GFS)

Splits input files into *M* pieces (16 to 64 MB per piece)

Implementation – Step 2

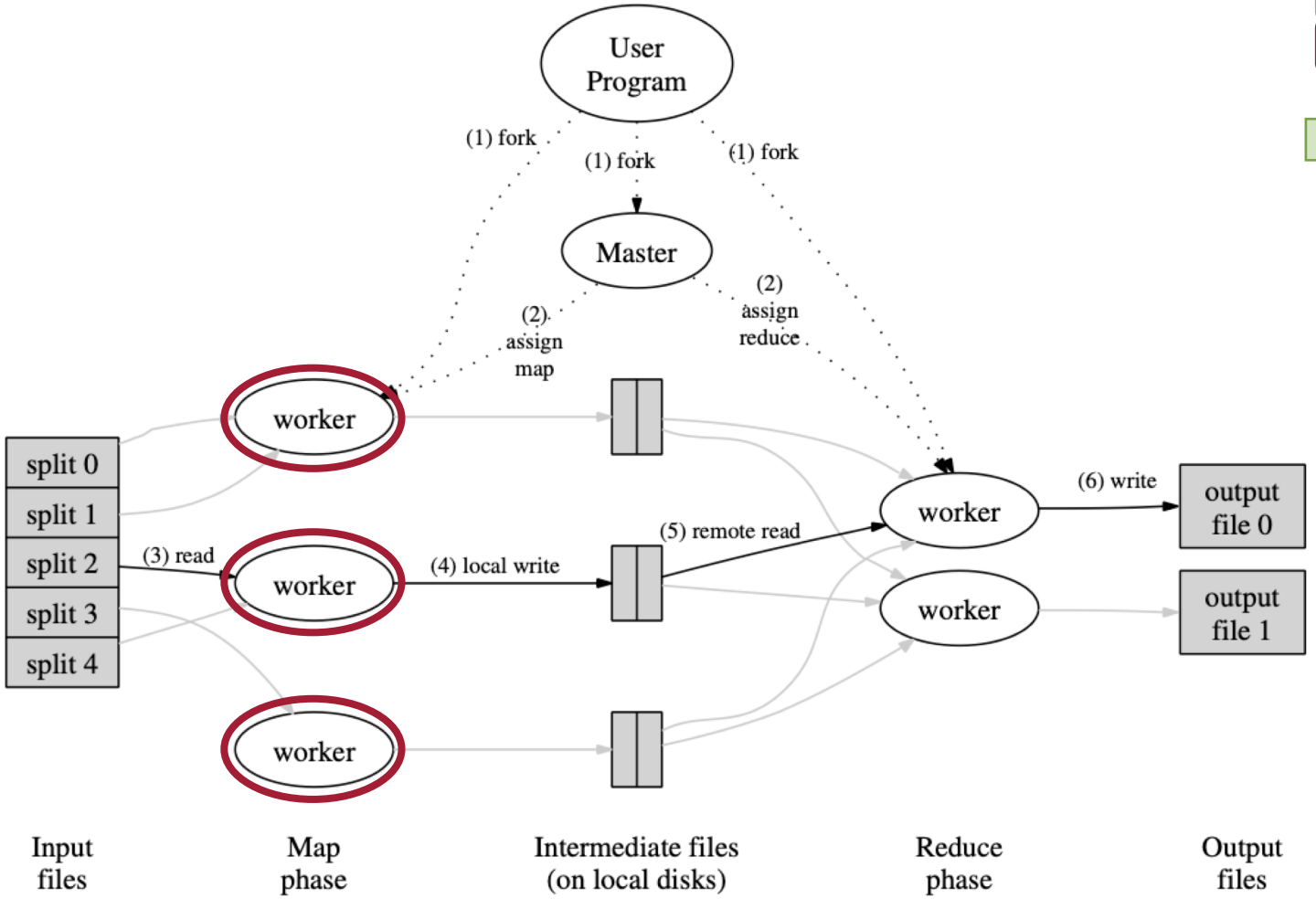


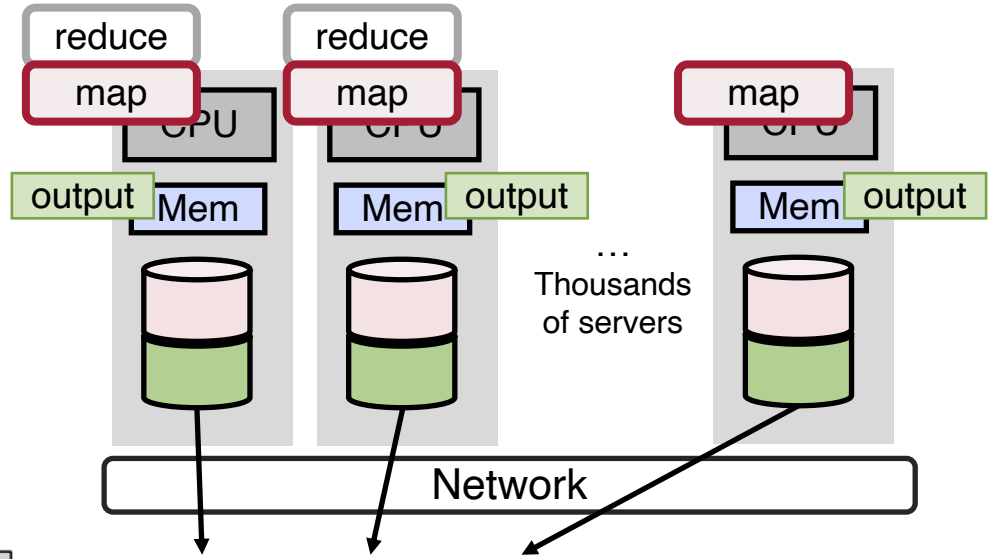


Google File System (GFS)

Assign *M* map and *R* reduce tasks to servers

Implementation – Step 3

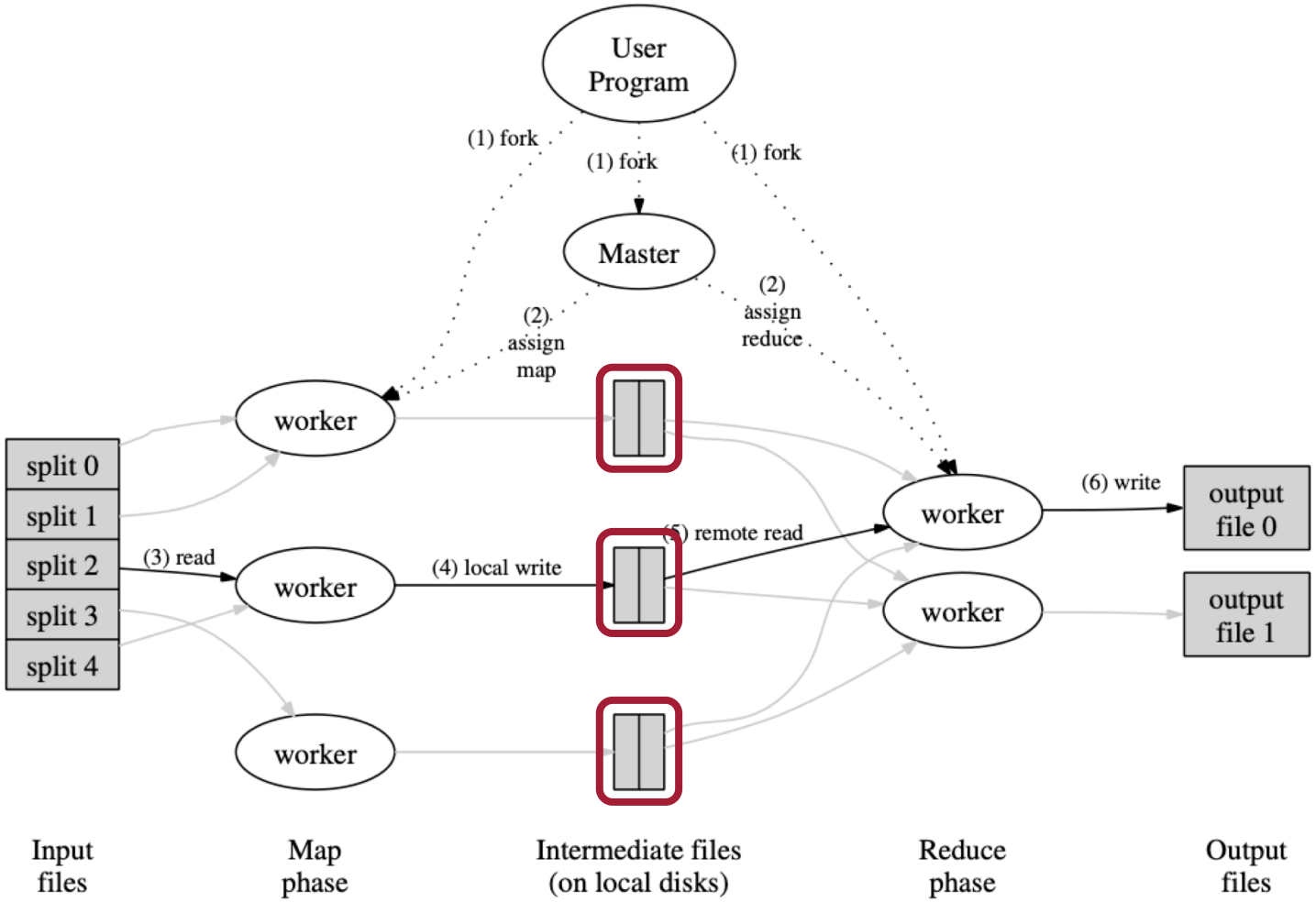


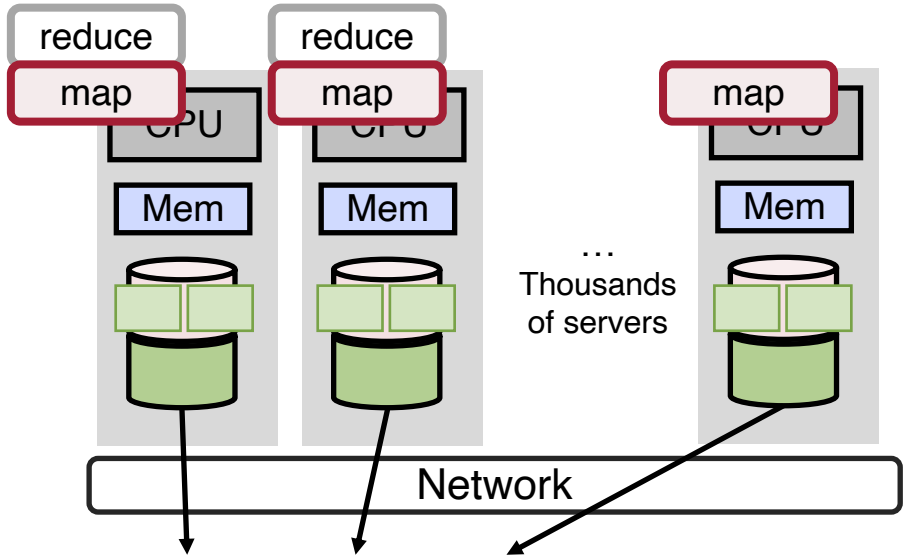


Google File System (GFS)

Execute map tasks and write output to local memory

Implementation – Step 4

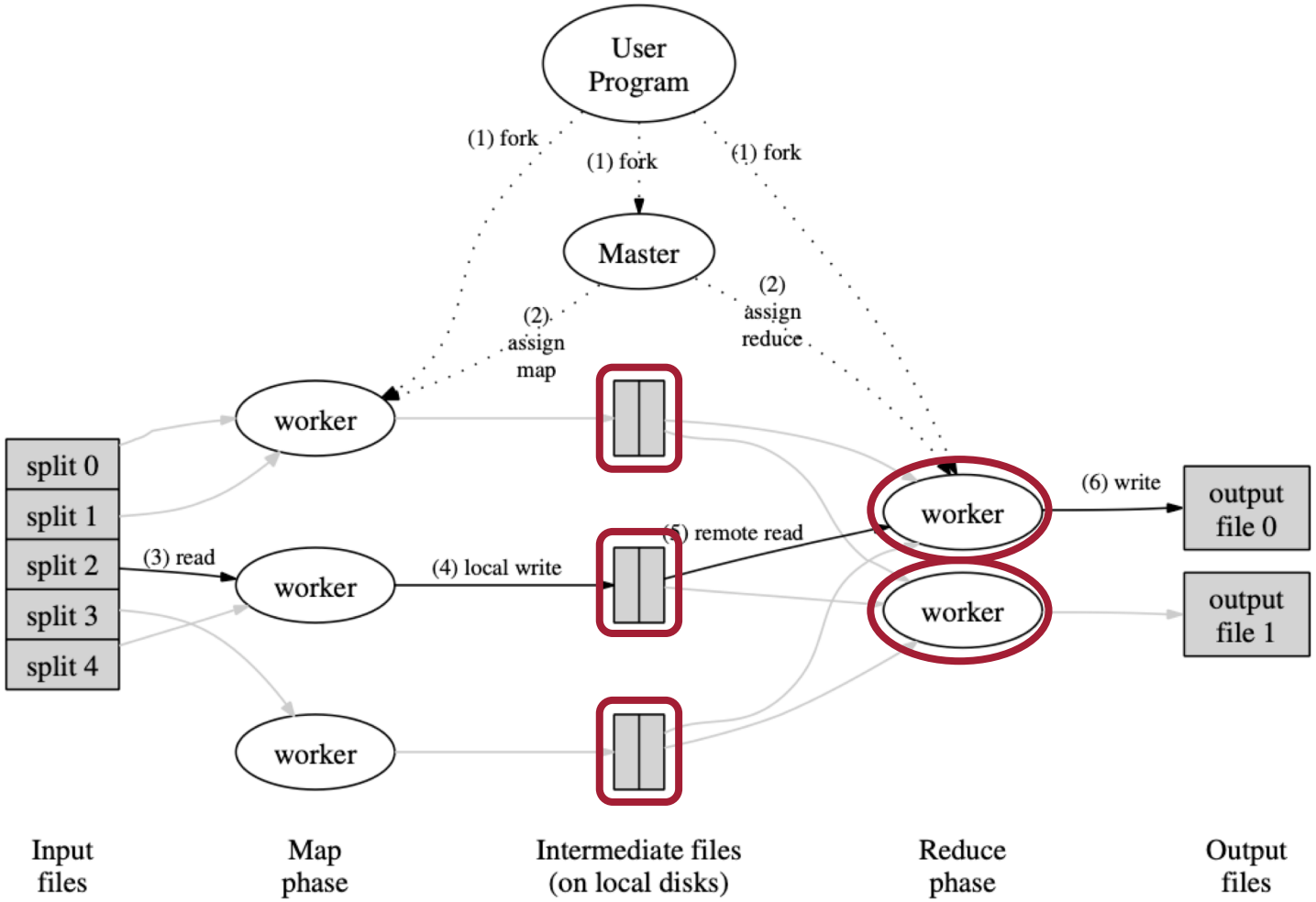


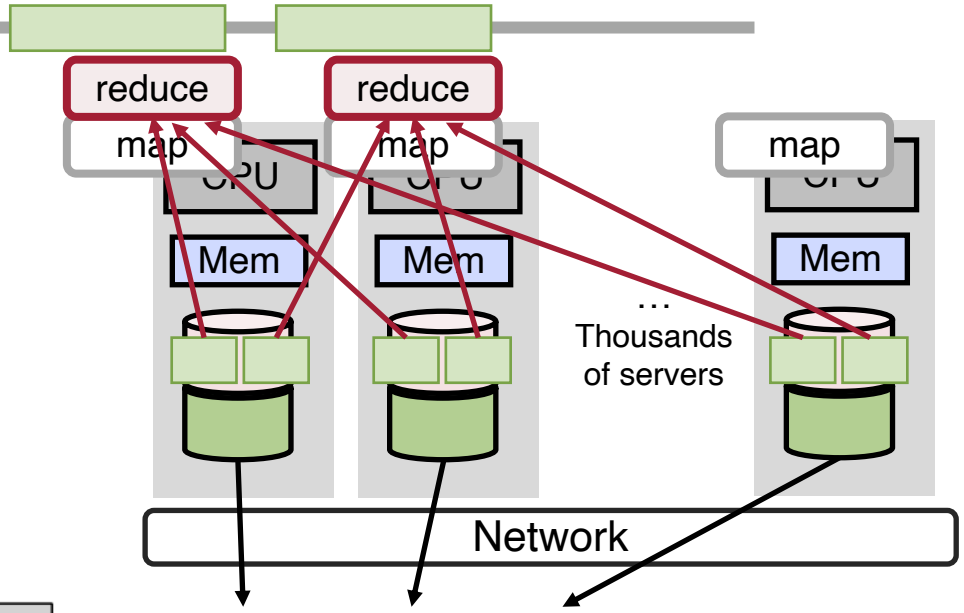


Google File System (GFS)

Partition the output into *R* regions and write them to disk

Implementation – Step 5

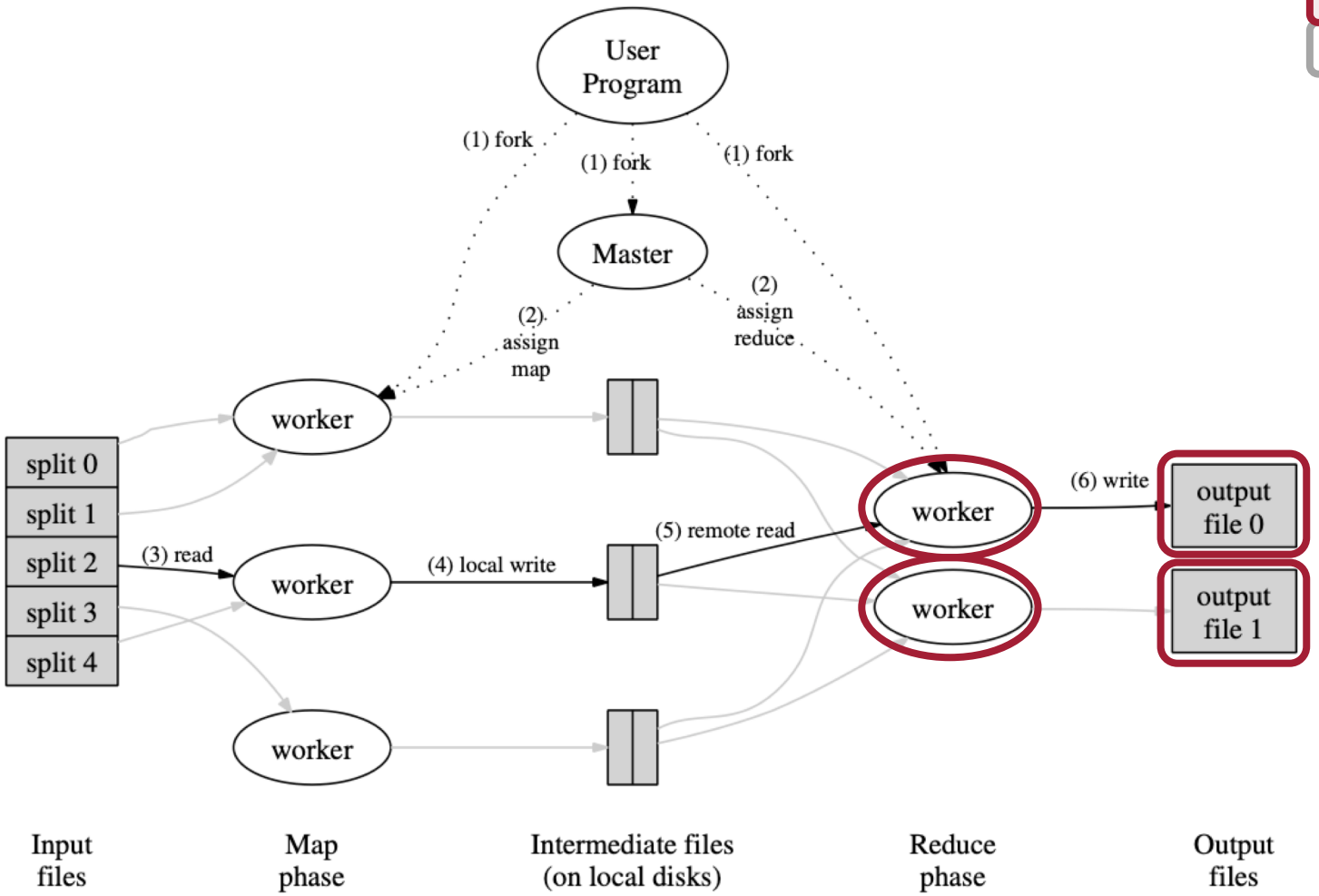


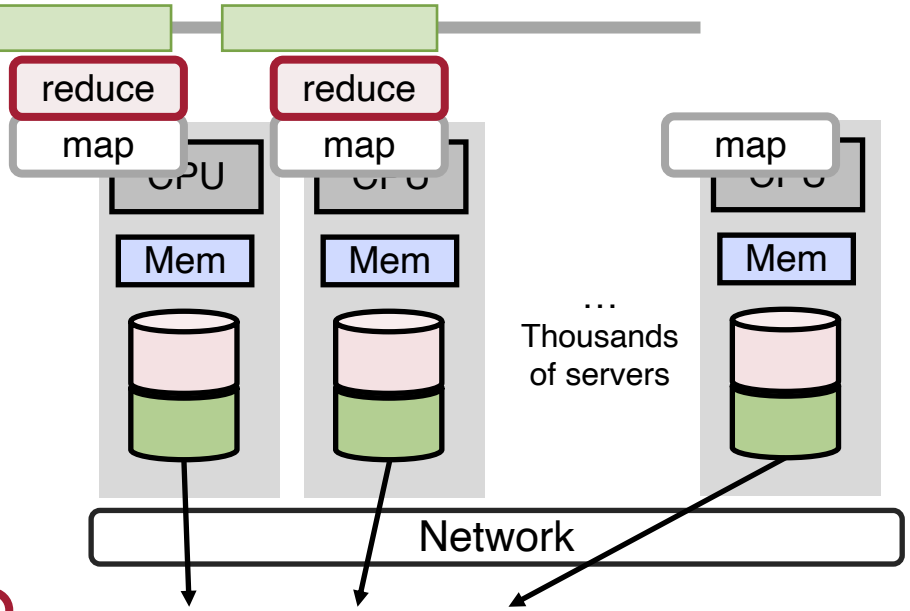


Google File System (GFS)

Reduce task reads corresponding intermediate data (i.e., output of map tasks) and sort them

Implementation – Step 6

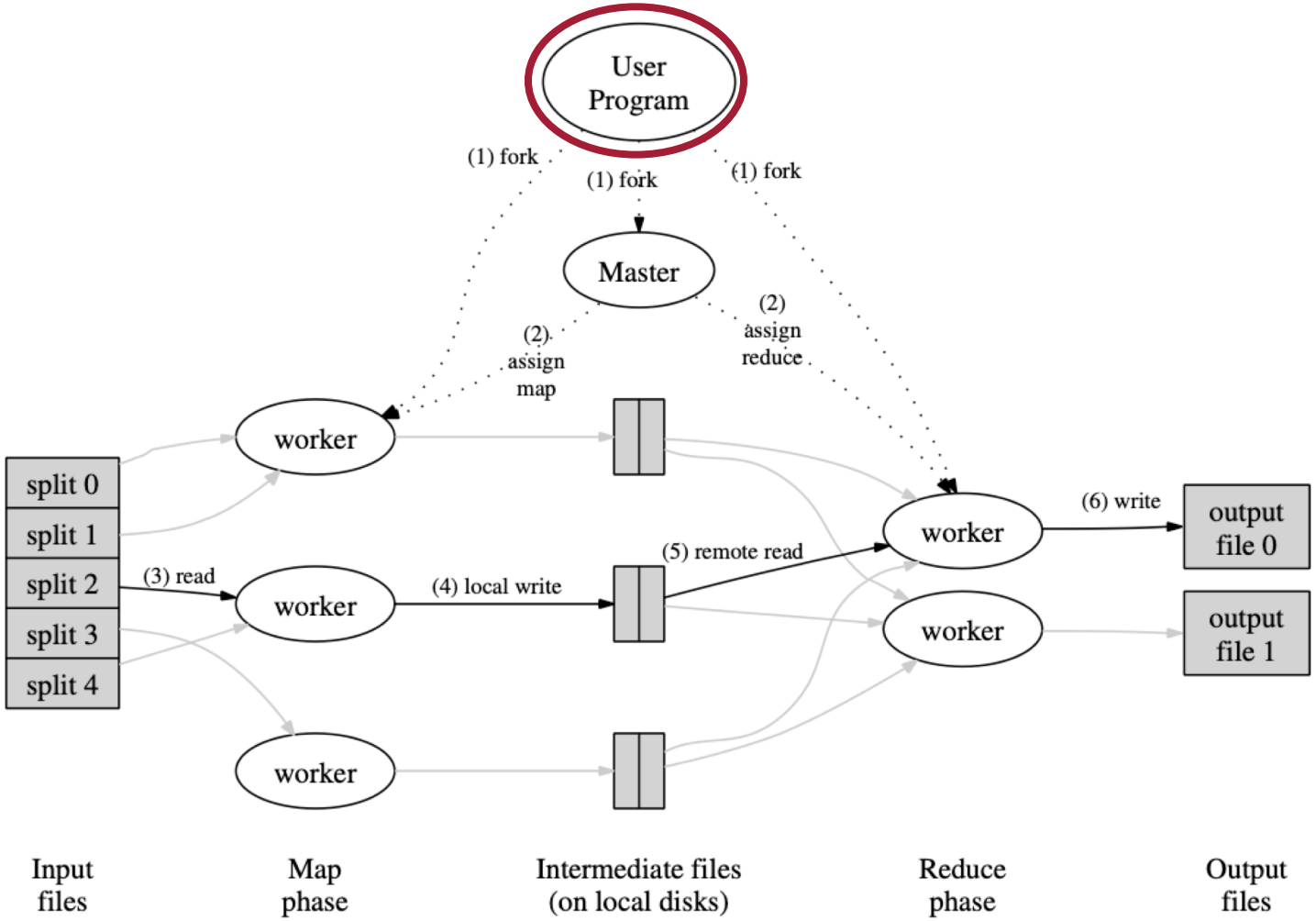


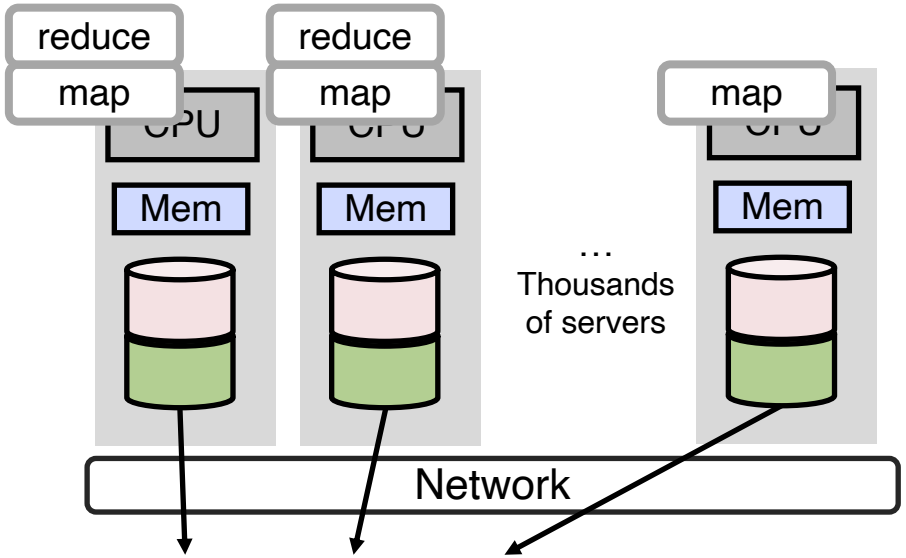


Google File System (GFS)

Execute reduce tasks and write output to GFS

Implementation – Step 7





Google File System (GFS)

Wake up the user program after all map and reduce tasks finish

Master Node

Orchestrates the MapReduce job

For each map task and reduce task, maintains states (idle, in-progress, or complete) and identity of worker machine

Collect locations of map tasks' outputs on disk and forward them to the reduce tasks

Fault Tolerance

The master pings every worker periodically

At a timeout, reschedule tasks mapped to this worker to other workers

- Map task: **all map tasks** are rescheduled
- Reduce task: **incomplete reduce tasks** are rescheduled

Master failure

- Unlikely since the master is a single machine
- Abort the MapReduce computation if the master fails
- **Single point of failure**

Backup Tasks

A straggler task can take unusually long time to complete

- Bad disk
- Contention with other tasks on the server
- Misconfiguration

Solution: Schedule backup execution for in-progress tasks when the MapReduce computation is close to finish

- Overhead is small (a few percent)
- Improvement is significant (44% for the sort program)

Other Optimizations

Locality

- Try to schedule a map task on a machine that contains (or is close to) a replica of the corresponding input data

Combiner function

- Local reduce function on each map task to reduce the intermediate data size
- Similar to pushing down group-by in query optimization

Performance Evaluation — Grep

Grep

- 1 TB of 100-byte records
- Search for a rare three character pattern
- Map: emits a line if it matches the pattern
- Reduce: identity function—copy input to output

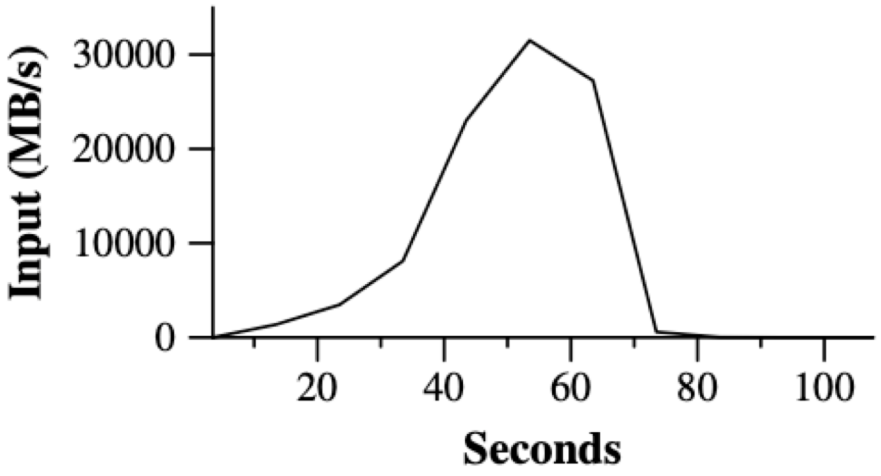


Figure 2: Data transfer rate over time

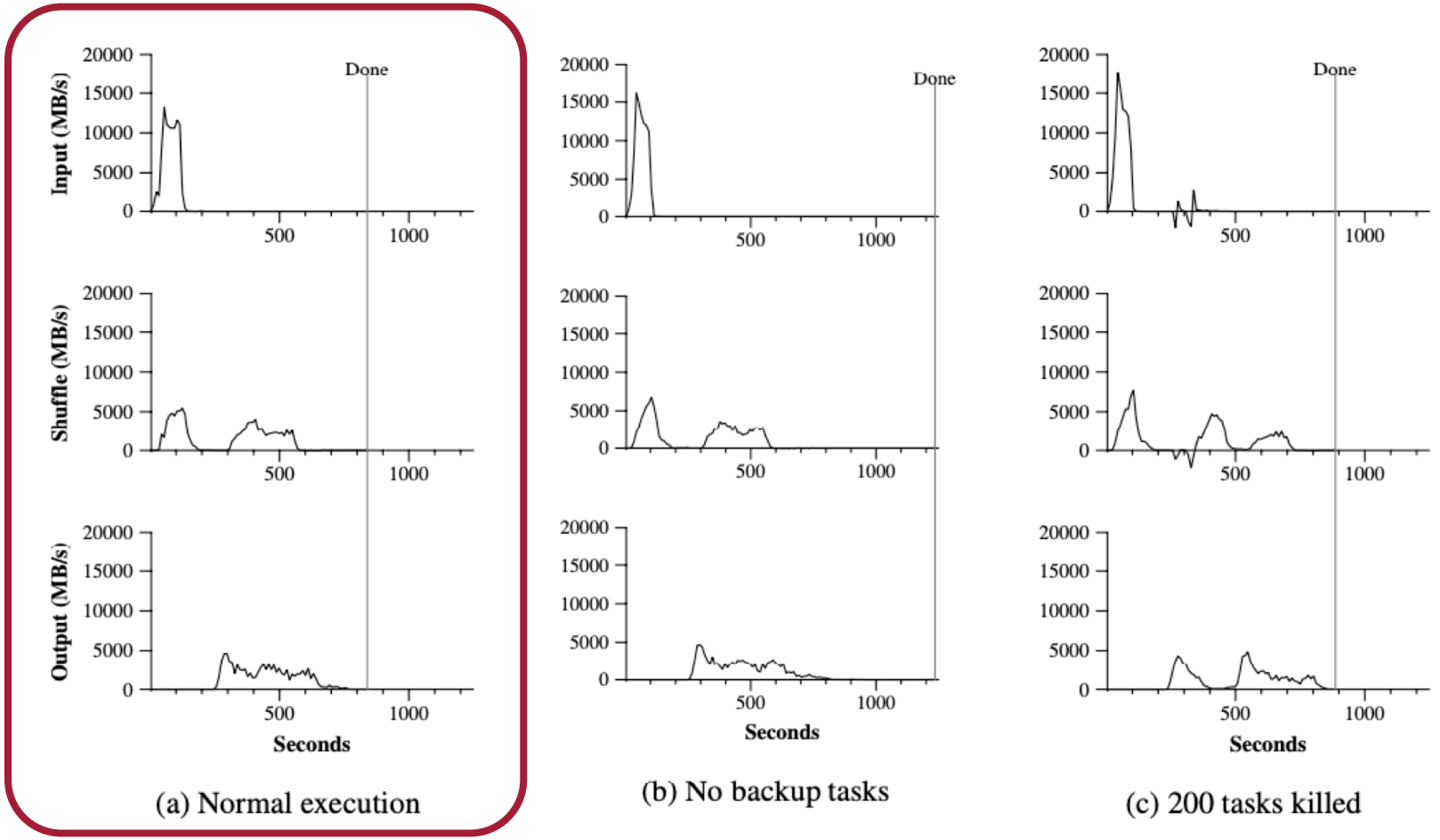
- Input data scan rate increases as more machines assigned to the MapReduce computation and peaks at over 30 GB/s when 1764 workers have been assigned
- The rate declines after map tasks finish reading the input data

Performance Evaluation — Sort

Sort

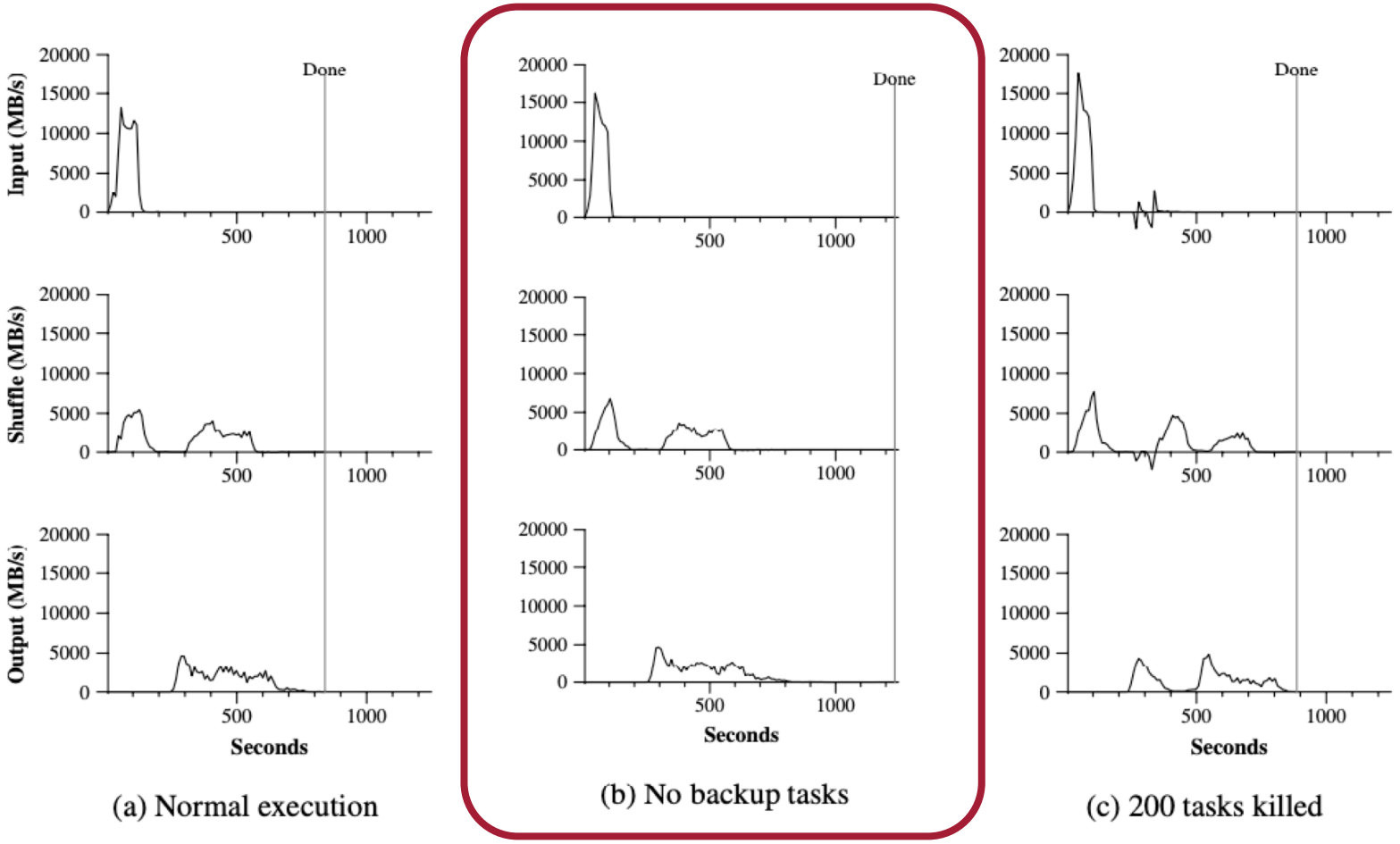
- 1 TB of 100-byte records
- Map: extract a 10-byte key and emit <key, original record in text>
- Reduce: identity function
- Partitioning function: range partition
- Note that a **reducer task by default sorts its input data**

Performance Evaluation — Sort



Two batches of reduce tasks

Performance Evaluation — Sort



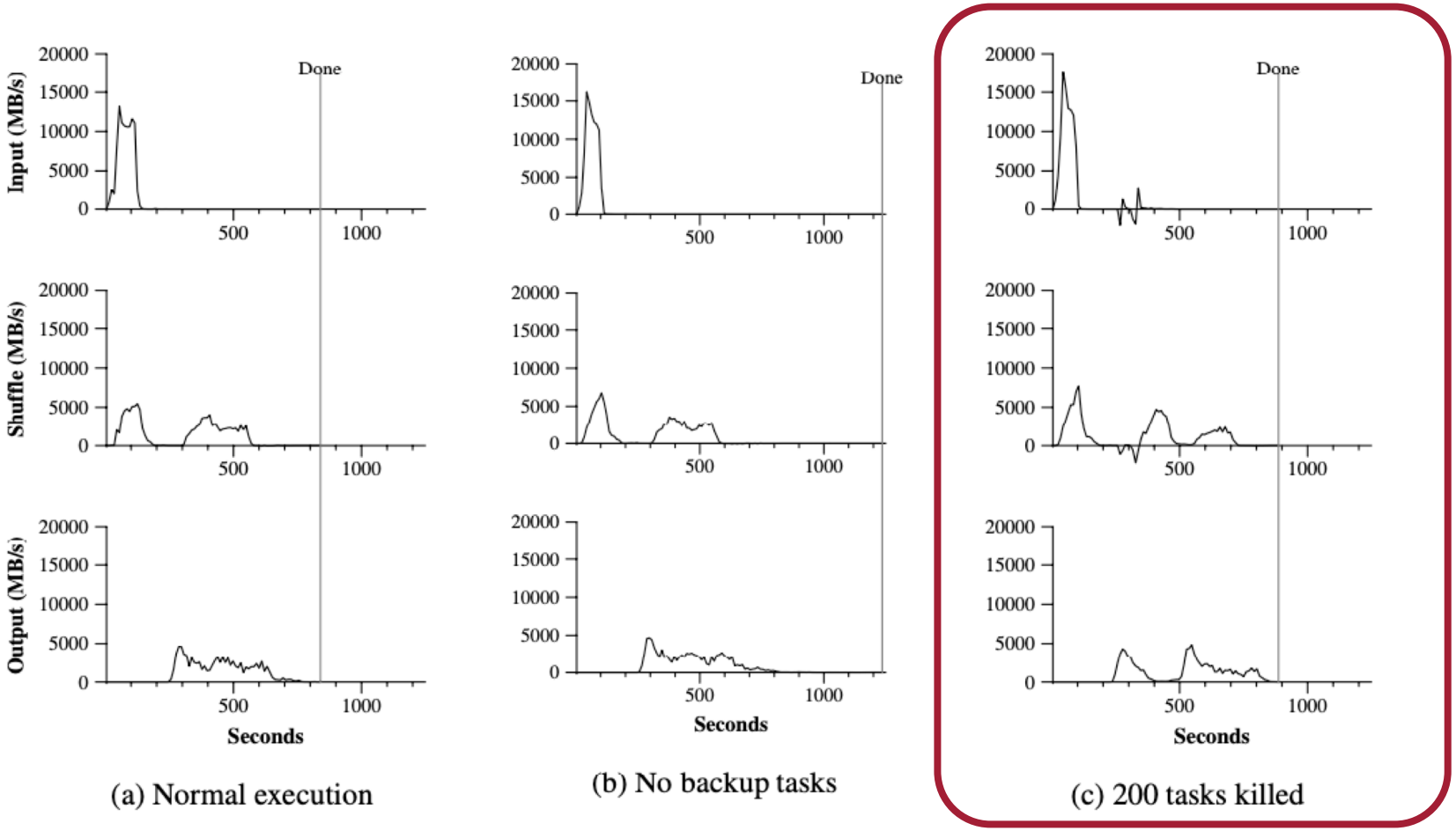
(a) Normal execution

(b) No backup tasks

(c) 200 tasks killed

Straggler tasks increase the total runtime by 44%

Performance Evaluation — Sort



Failure of processes has small performance impact

MapReduce vs. Databases[1]

With user defined functions, Map and Reduce functions can be written in SQL; the shuffle between Map and Reduce is equivalent to a Group-By

Performance

Benchmark performance on a 100-node cluster.

	Hadoop	DBMS-X	Vertica	Hadoop/DBMS-X	Hadoop/Vertica
Grep	284s	194s	108x	1.5x	2.6x
Web Log	1,146s	740s	268s	1.6x	4.3x
Join	1,158s	32s	55s	36.3x	21.0x

[1] Stonebraker, Michael, et al. "MapReduce and parallel DBMSs: friends or foes?." *Communications of the ACM* 2010

MapReduce vs. Databases[1]

Technical differences

- Repetitive parsing
- Compression
- Pipelining
- Scheduling
- Column-oriented storage
- Query optimization

[1] Stonebraker, Michael, et al. "MapReduce and parallel DBMSs: friends or foes?." *Communications of the ACM* 2010

MapReduce vs. Databases[1]

Technical differences

- Repetitive parsing
- Compression
- Pipelining
- Scheduling
- Column-oriented storage
- Query optimization

Conclusions:

- Parallel DBMSs excel at efficient querying of large data sets; MR-style systems excel at complex analytics and ETL tasks.
- High-level languages are invariably a good idea for data-processing systems
- What can DBMS learn from MapReduce?
  - Out-of-the-box experience (one-button install, auto tuning)
  - Semi-structured or un-structured data

[1] Stonebraker, Michael, et al. "MapReduce and parallel DBMSs: friends or foes?." *Communications of the ACM* 2010

Q/A – MapReduce

Computational models that do not work well with MapReduce?

Is the master a single-point of failure and performance bottleneck?

Why old papers have no performance evaluation?

MapReduce used in DBMS? (e.g., Hadapt, Hive, SparkSQL)

Why is the atomic rename necessary in the reducer?

Other systems like MapReduce (e.g., Apache Hadoop, Spark)

Why do we need sorting and shuffling?

Discussion

How to implement the following joining query in MapReduce?

```
SELECT *
FROM S, R
WHERE S.id = R.id
```

Next Lecture

Mid-term course evaluation **DDL: 10/23**

Please submit your proposal to the review website: **(DDL Oct 26)**

- https://wisc-cs764-f20.hotcrp.com

Please submit a review for the guest lecture within 3 days after the lecture (by **Oct 28 11:59pm**)