MapReduce Programming

Pangfeng Liu
National Taiwan University

June 15, 2015

Topics

- Google File System
- MapReduce Programming Model

GFS in a Nutshell

- Google File System (GFS) is a scalable distributed file system for large distributed data-intensive applications.
- GFS provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

Objectives

- GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment.

Observations

- Component failures are the norm rather than the exception.
- Files are huge by traditional standards.
- Most files are mutated by appending new data rather than overwriting existing data. Random writes within a file are practically non-existent.
- Co-designing the applications and the file system API benefits the overall system by increasing our flexibility.

Assumptions

- The system is built from many inexpensive commodity components that often fail.
- The system stores a modest number of large files.
- The workloads primarily consist of two kinds of reads: large streaming reads and small random reads.
- The workloads also have many large, sequential writes that append data to files.
- The system must efficiently implement well-defined semantics for multiple clients that concurrently append to the same file.
- High sustained bandwidth is more important than low latency.

Characteristics

- One master stores metadata including data locations and multiple chunk servers stores actual data.
- Metadata are stored in memory for fast access.
- Actual data has three replicas for fault tolerance, and replicas are stored within and not within locality.
- Complicated protocol for storing data.

Discussion

- Describe the advantage and disadvantages of large block size in a file system.

MapReduce

- A programming model and an associated implementation for processing and generating large data sets.
- Programs written in MapReduce style are automatically parallelized and executed on a large cluster of commodity machines.
- Google implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable.

Simple Computation

- Google has implemented hundreds of special-purpose computations that process large amounts of raw data.
- Most such computations are conceptually straightforward but the input data is usually large and the computations have to be distributed across hundreds or thousands of machines.
- The issues of how to parallelize the computation, distribute the data, and handle failures obscure the original simple computation with large amounts of complex code to deal with these issues.

Abstraction

- Google designed a new abstraction that allows programmers to express the simple computations but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing.
- The programmers focus on the computation, not the parallelization.

Abstraction

- The abstraction applies a *map* operation to each logical record in our input in order to compute a set of intermediate *key/value pairs*.
- Then the system applies a *reduce* operation to all the values that shared the *same* key, in order to combine the derived data.

Discussion

- Give example of most high level and most low level of parallel programming you are aware of.
- Discuss the advantage and disadvantage of abstraction on parallel programming.

Map and Reduce

- The programming model consists of a *map* function, a *reduce* function, and a *shuffle stage*.
- Map and reduce are borrowed from functional programming.

Stages

- In the map stage the map function reads inputs and produces intermediate key/value pairs.
- In the shuffle stage the system processes intermediate key/value pairs so that later the reduce function can read them.
- In the reduce stage the reduce function reads the processed intermediate key/value pairs and produces the output.

Responsibility

- The programmer provides the map and reduce function.
- The MapReduce runtime system does the shuffling.

Discussion

- What does a programmer have to do in MapReduce style of programming?

Map

- Map is a function.

  input An input I.

  output A multi-set $\{(k_1, v_1), \ldots, (k_n, v_n)\}$.

A Map Example

- Given a string, produces a multi-set of pairs that are in the form of $(w, 1)$, where w is a word in the string.
- The input is "this is a book not a pencil".
- The output will be (this, 1), (is, 1), (a, 1), (book, 1), (not, 1), (a, 1), (pencil, 1).

An Example

- The same map function may have different inputs.
- Assume that the input is "this is a pencil not a chair".
- the output will be (this, 1), (is, 1), (a, 1), (pencil, 1), (not, 1), (a, 1), (chair, 1).

Key/Value Pairs

- Now the system collects all the output from all inputs from map function. This is called *intermediate key/value pairs*.
- From the previous example we have the intermediate key/value pairs from two inputs as (this, 1), (is, 1), (a, 1), (book, 1), (not, 1), (a, 1), (pencil, 1), (this, 1), (is, 1), (a, 1), (pencil, 1), (not, 1), (a, 1), (chair, 1).

Another Example

- Given a document d, produces a set of pairs that ares in the form of (w, d), where w is a word in u.
- The input is a document "readme", which has words "this", "is", "a", "pencil".
- The output will be a set (this, readme), (is, readme), (a, readme), (pencil, readme).

An Example

- The same map function may have different inputs.
- The input is a document "note", which has words "that", "is", "a", "chair".
- The output will be (that, note), (is, note), (a, note), (chair, note).

Key/Value Pairs

- We have the intermediate key/value pairs as (this, readme), (is, readme), (a, readme), (pencil, readme), (that, note), (is, note), (a, note), (chair, note).

Discussion

- Can you guess the purpose of these two examples? What kind of problem they want to solve?

Shuffle

- The *shuffle* in MapReduce means transferring data from map to reduce via network transfer.
- Before we pass the intermediate key/value pairs to the reduce function, we must *sort* them.
- The reason is that a reduce function will work on the values from the *same* key only, so we need to shuffle the pairs so that the pairs of the same key go to the same reduce function.

Key/Value Pairs

- From the previous example we have the intermediate key/value pairs as (this, 1), (is, 1), (a, 1), (book, 1), (not, 1), (a, 1), (pencil, 1), (this, 1), (is, 1), (a, 1), (pencil, 1), (not, 1), (a, 1), (chair, 1)

Shuffle

- The intermediate key value pairs will be sorted according to keys for later use.
- We will have (a, 1), (a, 1), (a, 1), (a, 1), (book, 1), (chair, 1) (is, 1), (is, 1), (not, 1), (not, 1), (pencil, 1), (pencil, 1), (this, 1), (this, 1)

Group Values Together

- The values of the same key will be grouped into a list
- We will have the following lists.
  - (a, (1, 1, 1, 1))
  - (book, (1))
  - (chair, (1))
  - (is, (1, 1))
  - (not, (1, 1))
  - (pencil, (1, 1))
  - (this, (1, 1))

Discussion

- Can you guess the purpose of this example now? What kind of problem they want to solve?
- Now if we want to solve this problem, what else do we (or the reducer) need to do?

Key/Value Pairs

- We have the intermediate key/value pairs as (this, readme), (is, readme), (a, readme), (pencil, readme), (that, note), (is, note), (a, note), (chair, note).

Shuffle

- The intermediate key value pairs will be sorted according to keys for later use.
- We will have (a, note), (a, readme), (chair, note), (is, readme), (is, note), (pencil, readme), (that, note), (this, readme).

Group Values Together

- The values of the same key will be grouped into a list
- We will have the following lists.
  - (a, (note, readme))
  - (chair, (note))
  - (is, (note, readme))
  - (pencil, (readme))
  - (that, (note))
  - (this, (readme))

Discussion

- Can you guess the purpose of this example now? What kind of problem they want to solve?
- Now if we want to solve this problem, what else do we (or the reducer) need to do?

Reduce

- Reduce is a function.

  input A key and a list of values.

  output A result from the list of values.

A Reduce Example

- Given a key and a list of numbers, produces a pair of the key and the sum of these numbers.

Word Count

- Given (a, (1, 1, 1, 1)) produces (a, 4).
- Given (book, (1)) produces (book, 1).
- Given (chair, (1)) produces (chair, 1).
- Given (is, (1, 1)) produces (is, 2).
- Given (not, (1, 1)) produces (not, 2).
- Given (pencil, (1, 1)) produces (pencil, 2).
- Given (this, (1, 1)) produces (this, 2).

We have a word counting program!!!

Discussion

- Can you think of any optimization that a map can do to reduce the amount of data from map to reduce in this example?

A Reduce Example

- Given a key and a list of values, produces the *same thing*.
- Like an identify function.

Word Count

- Given (a, (note, readme)) produces (a, (note, readme)).
- Given (chair, (note)) produces (chair, (note)).
- Given (is, (note, readme)) produces (is, (note, readme)).
- Given (pencil, (readme)) produces (pencil, (readme)).
- Given (that, (note)) produces (that, (note)).
- Given (this, (readme)) produces (this, (readme)).

We have a word index program!!!

Discussion

- If the map function produces a multi-set instead of a set, then what will be the output?

Mapper and Reducer

- Both map and reduce functions are run by multiple machines in parallel.
- A machine running a map function is called a *mapper*.
- A machine running a reduce function is called a *reducer*.
- We will assume that we have M mappers and R reducers.

Partition Function

- A *partition function* determines the reducer an intermediate key/value pair will go to.
- Usually we use a simple hash function to hash the key into a integer from 0 to $R-1$.
- Each reducer writes its data according to its index (from 0 to $R-1$.
- The final answer is a concatenation of all files according to reducer ids.

Discussion

- Can any reducer run before any mapper finishes? Explain your answer.
- How do we do sorting with MapReduce if we know the key distribution? In particular, how do we design our partition function, so that the final result is sorted?

Map

- The user defined map function extends (inherits) the base class `MapReduceBase`
- The first two template parameters of interface `Mapper` describe the input, and second two template parameters describe the output.
- The output type is a template using the parameter of `Mapper`, and is the type of intermediate key/value pairs.

Map Parameters

key	The line number of file.
value	The line at `key`.
output	The output object.
reporter	The error/information handle.

StringTokenizer

- We convert value to string, then convert the string with the constructor of `StringTokenizer`.
- The *hasMoreToken* and *nextToken* are methods of `StringTokenizer` that works as an iterator.

OutputColelctor

- `OutputColelctor` is the type of intermediate key value.
- We write the string (of type `Text`) first, then a constant 1.

Map

Example 1: (WordCount.java)

```
14  public static class Map extends MapReduceBase
15    implements Mapper<LongWritable, Text, Text, IntWritable> {
16    public void map(LongWritable key, Text value,
17                    OutputCollector<Text, IntWritable> output,
18                    Reporter reporter)
19      throws IOException {
20      StringTokenizer stk =
21        new StringTokenizer(value.toString());
22      while (stk.hasMoreTokens()) {
23        output.collect(new Text(stk.nextToken()),
24                       new IntWritable(1));
25      }
26    }
27  }
```

Discussion

- Did we use the parameter `key`?
- Guess why the parameter `key` has type `LongWritable` not simply a `IntWritable`.

Reduce

- The user defined `reduce` function extends (inherits) the base class `MapReduceBase`
- The first two template parameters of interface `Reducer` describe the input, and second two template parameters describe the output.
- The input type is a template using the parameter of `Reducer`, and is the type of intermediate key/value pairs.

Reduce Parameters

key	The intermediate key.
values	intermediate values.
output	The output object.
reporter	The error/information handle.

Note that we have only one key but multiple values, so we need an iterator over values.

IntWritable

- The *hasNext* and *next* are methods of `Iterator` that works as an iterator over `IntWritable`.
- `get` is a method to get the actual value of an `IntWritable`.

OutputColelctor

- `OutputColelctor` is the type of intermediate key value.
- We write the string (of type `Text`) first, then the sum.

Reduce

Example 2: (WordCount.java)

```
29  public static class Reduce extends MapReduceBase
30    implements Reducer<Text, IntWritable, Text, IntWritable> {
31    public void reduce(Text key, Iterator<IntWritable> values,
32                       OutputCollector<Text, IntWritable> output,
33                       Reporter reporter)
34      throws IOException {
35      int sum = 0;
36      while(values.hasNext())
37        sum += values.next().get();
38      output.collect(key, new IntWritable(sum));
39    }
40  }
```

Discussion

- Check the consistency of the second two parameter of `Map` and first two parameter of `Reduce`. Why do they have to be the same?

Reduce

Example 3: (WordCount.java)

```
29  public static class Reduce extends MapReduceBase
30    implements Reducer<Text, IntWritable, Text, IntWritable> {
31    public void reduce(Text key, Iterator<IntWritable> values,
32                       OutputCollector<Text, IntWritable> output,
33                       Reporter reporter)
34      throws IOException {
35      int sum = 0;
36      while(values.hasNext())
37        sum += values.next().get();
38      output.collect(key, new IntWritable(sum));
39    }
40  }
```

Main Program

- The `JobConf` is the job configuration object.
- First we set the name of the job, which has to be the same as our object name (and the file name).
- Then we set the mapper and reducer.
- We also set the combiner, which is also the reducer.

Main Program

Example 4: (WordCount.java)

```
42   public static void main(String[] args) throws Exception {
43     JobConf conf = new JobConf(WordCount.class);
44     conf.setJobName("WordCount");
45
46     conf.setMapperClass(Map.class);
47     conf.setCombinerClass(Reduce.class);
48     conf.setReducerClass(Reduce.class);
```

Set Key/Value Type

- We set the output type for mapper and reducer.
- The API for mapper is `setMapOutputKeyClass` and `setMapOutputValueClass`
- The API for reducer is `setOutputKeyClass` and `setOutputValueClass`

Set Key/Value Type

Example 5: (WordCount.java)

```
50      conf.setMapOutputKeyClass(Text.class);
51      conf.setMapOutputValueClass(IntWritable.class);
52
53      conf.setOutputKeyClass(Text.class);
54      conf.setOutputValueClass(IntWritable.class);
```

Set I/O Type

- We set the input and output for the main program.
- `setInputFormat` is for mappers to read.
- `setOutputFormat` is for reducers to write.
- `setInputPaths` is the path name of the input
- `setOutputPaths` is the path name of the output.

Reduce

Example 6: (WordCount.java)

```
56      conf.setInputFormat(TextInputFormat.class);
57      conf.setOutputFormat(TextOutputFormat.class);
58
59      FileInputFormat.setInputPaths(conf, new Path(args[0]));
60      FileOutputFormat.setOutputPath(conf, new Path(args[1]));
61
62      JobClient.runJob(conf);
63    }
```

Discussion

- Why the combiner is also the reducer?

Implementation

- Commodity machines and networking hardware are used.
- A cluster consists of hundreds or thousands of machines, and therefore machine failures are common.
- Storage is provided by inexpensive disks attached directly to individual machines.
- GFS is used to manage the data stored on these disks. The file system uses replication to provide availability and reliability on top of unreliable hardware.

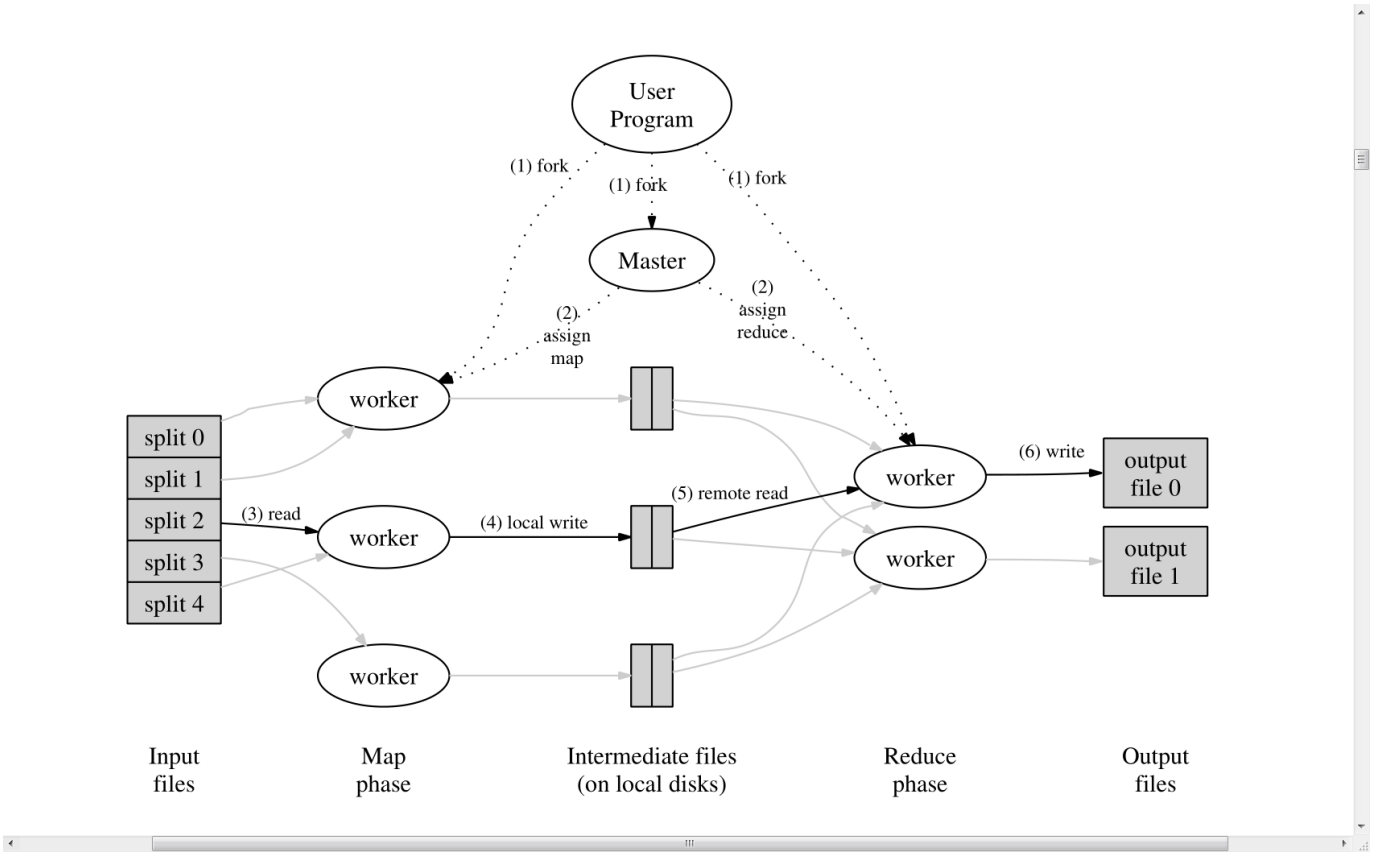
Stages

- The execution of a MapReduce program goes through seven stages.
  - Split input
  - Master initialization
  - Start mappers
  - Network transfer
  - Start reducers
  - Reduces output
  - Master cleanup

Split Input

- The MapReduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece.
- The MapReduce library then starts up many copies of the program on a cluster of machines.

MapReduce


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

Discussion

- Why input files are split into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece?

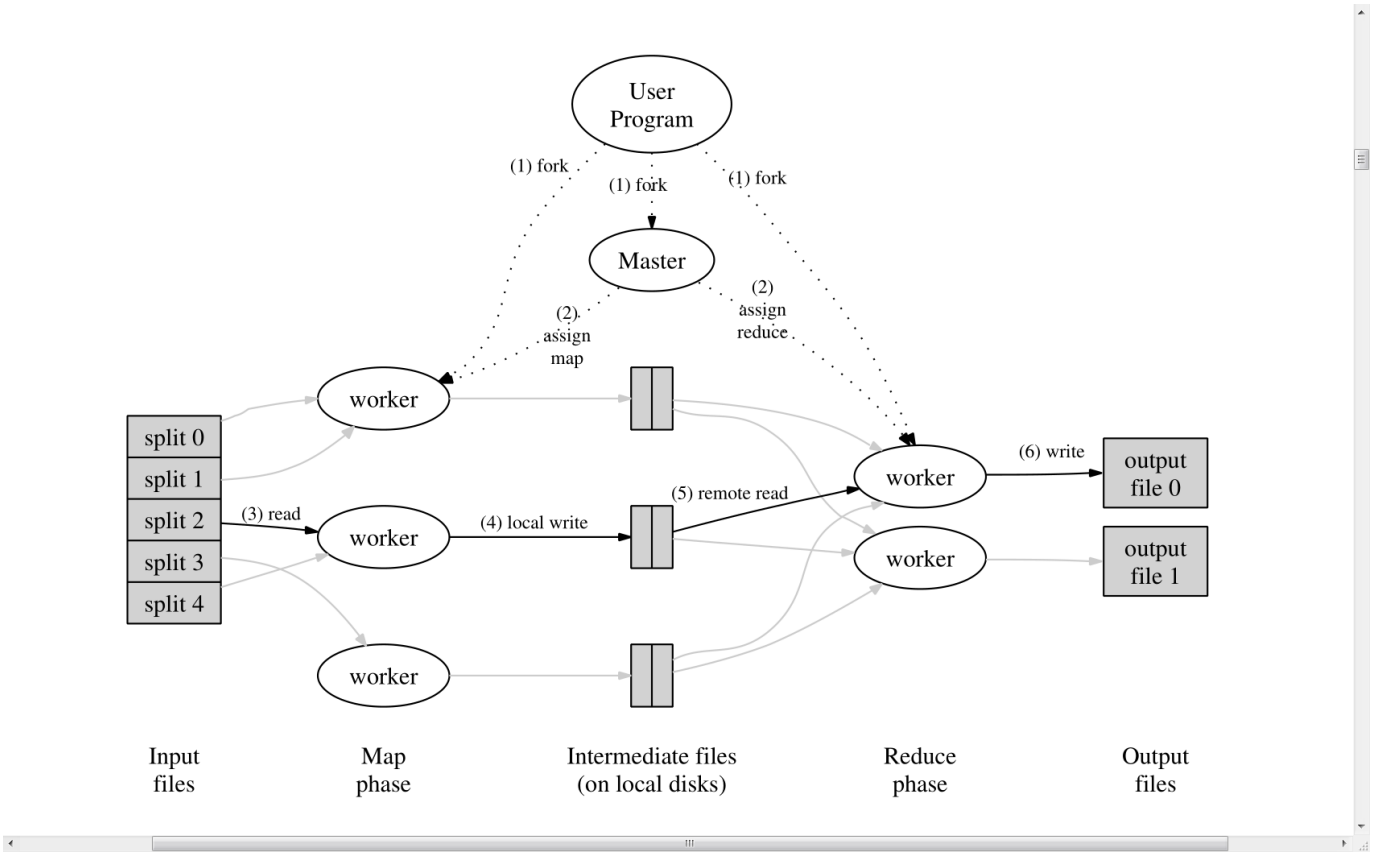
Master Initialization

- One of the copies of the program is the master and 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.

Start Mappers

- A worker who is assigned a map task reads the contents of the corresponding 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*.

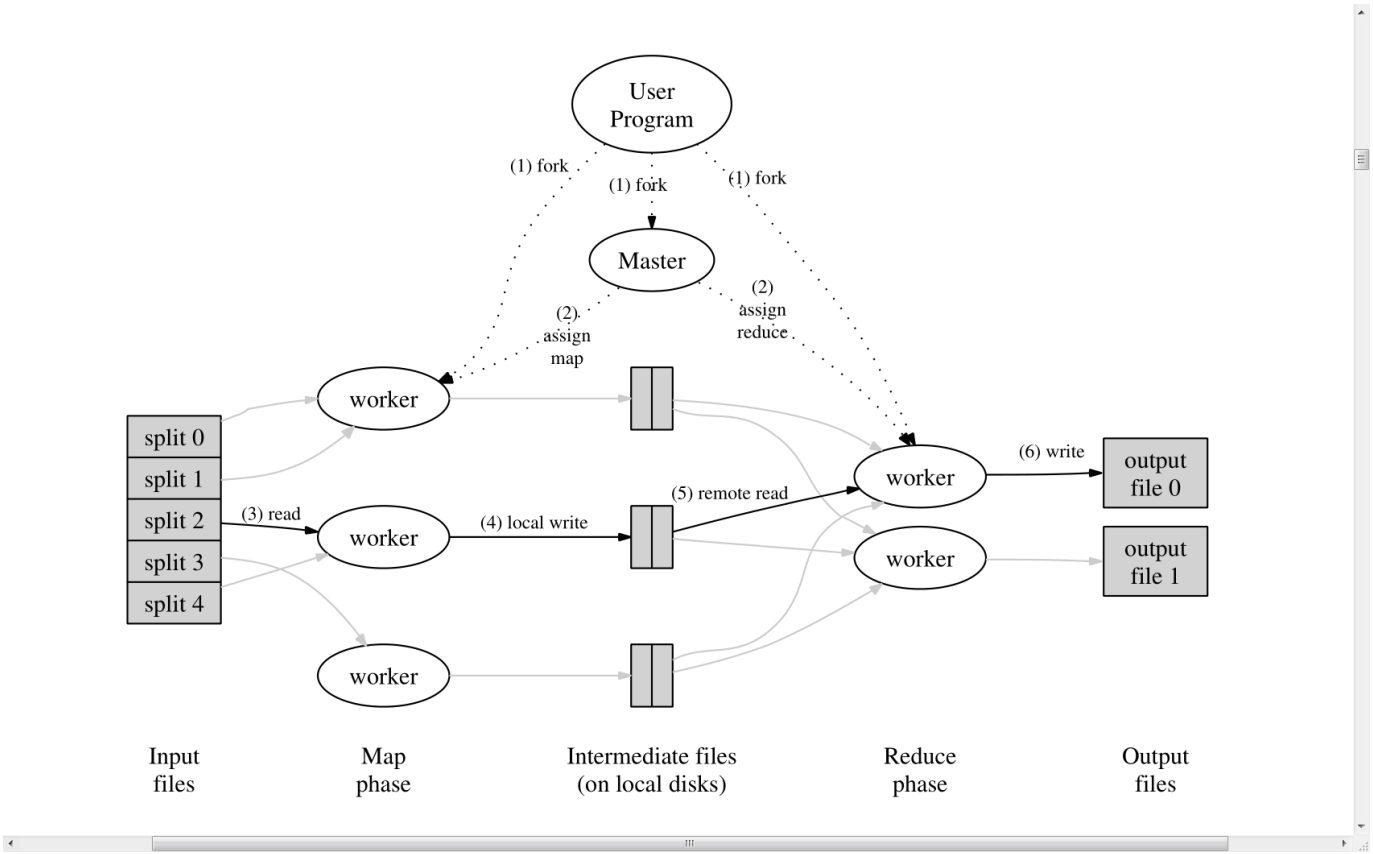
MapReduce


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

Network Transfer

- Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function.
- The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

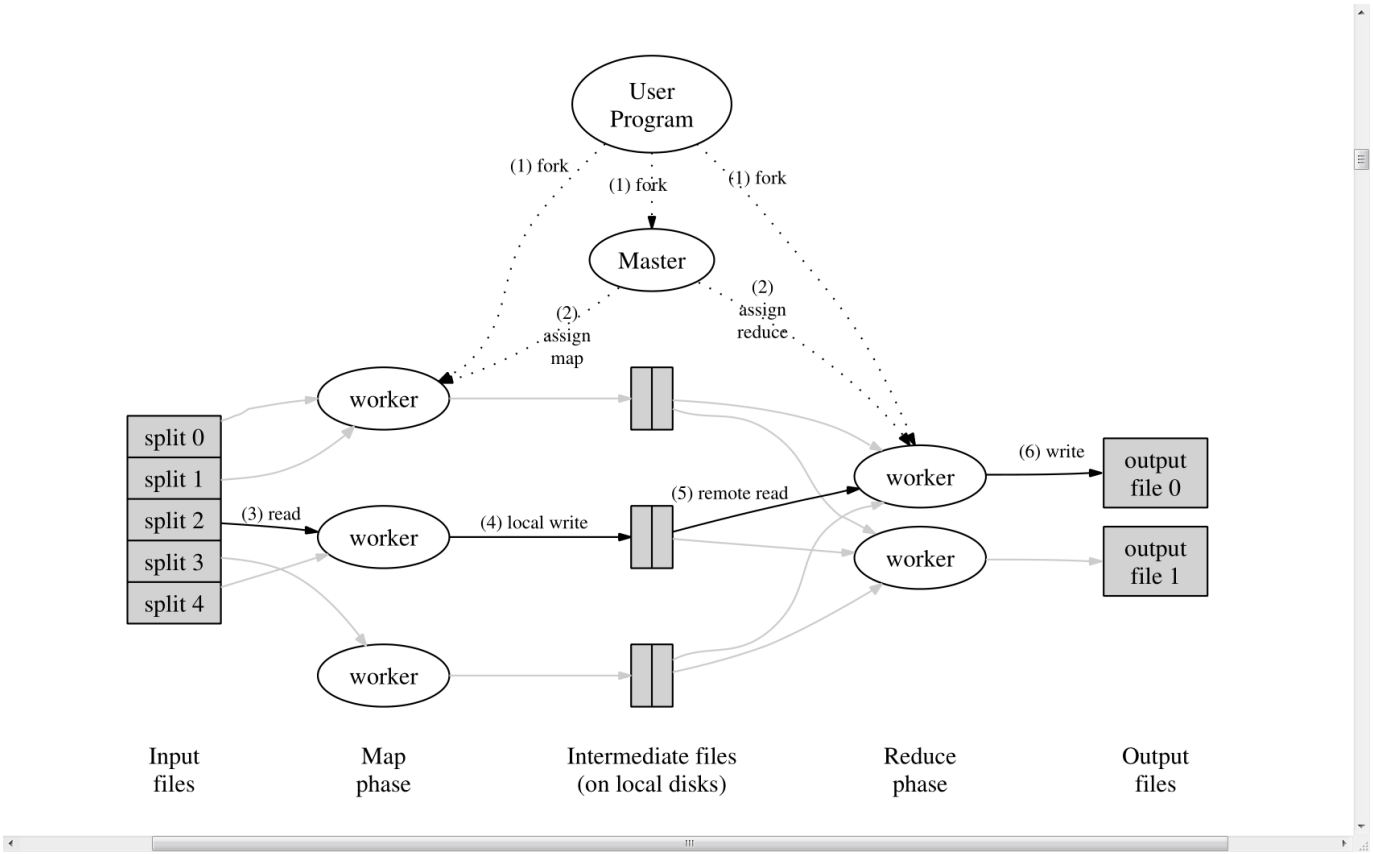
MapReduce


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

Start Reducers

- When a reduce worker is notified by the master about these locations, it uses 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.
- The sorting is needed because typically many different keys map to the same reduce task.

MapReduce


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

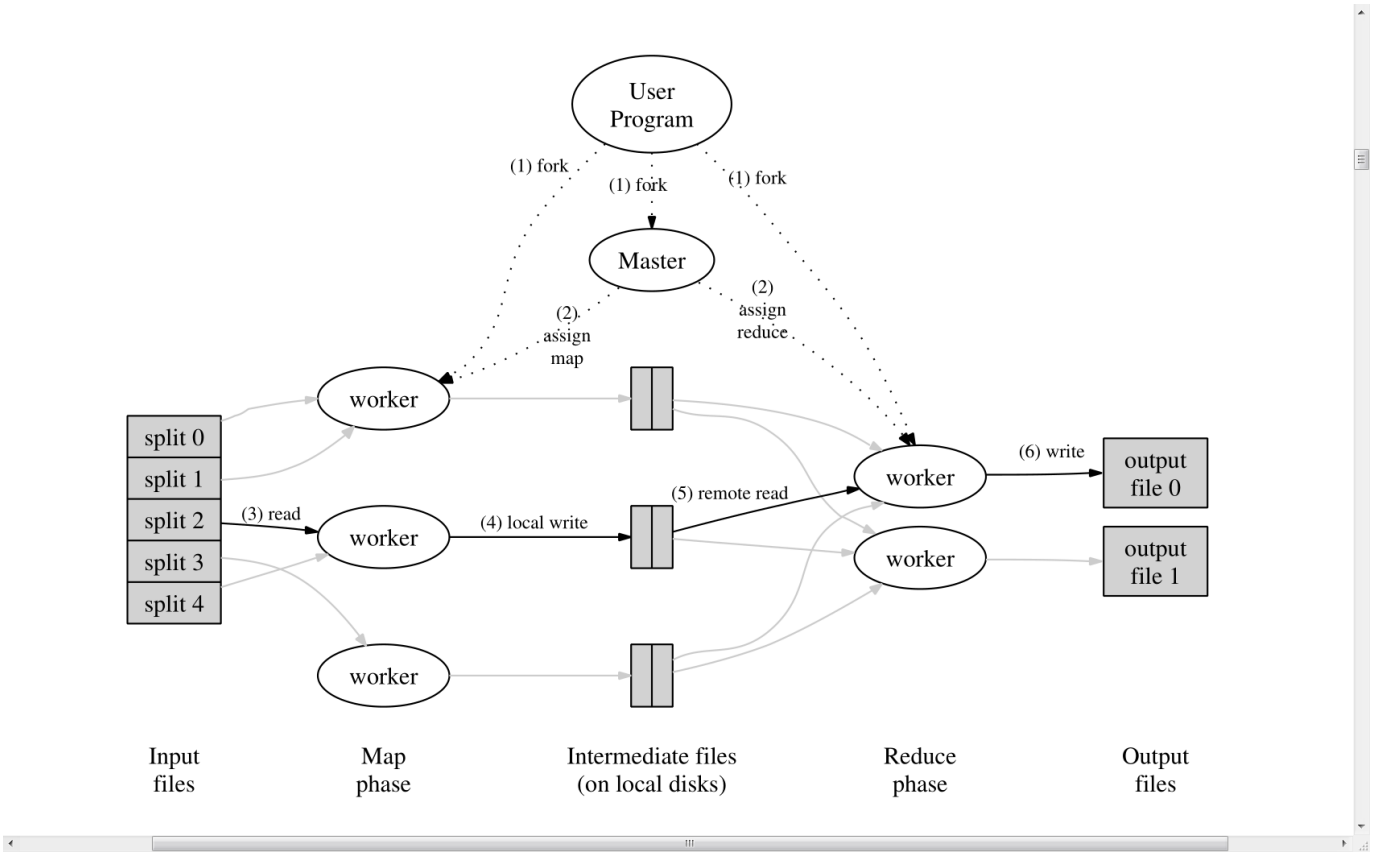
Discussion

- What is remote procedure call? What requirement is needed for doing RPC?

Reduces Output

- 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 users Reduce function.
- The output of the Reduce function is *appended* to a final output file for this reduce partition.

MapReduce


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

Master Cleanup

- When all map tasks and reduce tasks have been completed, the master wakes up the user program.
- At this point, the MapReduce call in the user program returns back to the user code.

Fault Tolerance

- Since the MapReduce library is designed to help process very large amounts of data using hundreds or thousands of machines, the library must tolerate machine failures gracefully.
- We will discuss worker failure and master failure

Worker Failure

- The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed.
- Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible.
- Completed reduce tasks do not need to be re-executed since their output is stored in a global file system.

Master Failure

- The current implementation aborts the MapReduce computation if the master fails.

Discussion

- Why do we re-execute a failed mapper, but not a reducer?
- Why master failure is different from worker failure? Explain you answer.

Locality

- We conserve network bandwidth by taking advantage of the fact that the input data, managed by GFS, is stored on the local disks of the machines that make up our cluster.
- GFS divides each file into 64 MB blocks, and stores several copies of each block (typically 3 copies) on different machines.

Locality

- The MapReduce master takes the location information of the input files into account and attempts to schedule a map task on a machine that contains a replica of the corresponding input data.
- Failing that, it attempts to schedule a map task near a replica of that tasks input data (e.g., on a worker machine that is on the same network switch as the machine containing the data).

Discussion

- How dose the system know where to find the replica of a block? Who has that location information?

Granularity

- We tend to choose M so that each individual task is roughly 16 MB to 64 MB of input data (so that the locality optimization described above is most effective),
- We make R a small multiple of the number of worker machines we expect to use.

Straggler

- A "straggler" is a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation.
- For example, a machine with a bad disk may experience frequent correctable errors that slow its read performance from 30 MB/s to 1 MB/s.

Backup Tasks

- When a MapReduce operation is close to completion, the master schedules *backup executions* of the remaining in-progress tasks.
- The task is marked as completed whenever either the primary or the backup execution completes.
- The sort program takes 44% longer to complete when the backup task mechanism is disabled.

Discussion

- If the probability for a machine to be a straggler is p, what is the pro-ability that a system of 1800 machines does *not* have a straggler? Let p be 0.0001 and calculate the answer.

Partition Function

- The default partitioning function uses hashing (e.g. hash(key) mod R) and tends to result in fairly well-balanced partitions.
- Sometimes the output keys are URLs, and we want all entries for a single host to end up in the same output file.
- We can specify a partitioning function using hash(Hostname(urlkey)) mod R, and all URLs from the same host will end up in the same output file.

Combiner Function

- There is significant repetition in the intermediate keys produced by each map task, and the user-specified Reduce function is commutative and associative.
- For example in work counting each map task will produce hundreds or thousands of records of the form (the, 1).
- We allow the user to specify an optional *Combiner function* that does partial merging of this data before it is sent over the network.

Example

- For example, the map function of word counting example after seeing "this is a book not a pencil", will report (a, (2)), instead of (a (1, 1)).
- Similar, the map function of word counting example after seeing "this is a pencil not a chair" will also report (a, (2)), instead of (a (1, 1)).
- The reducer will see two (a, (2)), instead of two (a, (1, 1)).
- This is like "preprocessing" in order to reduce the network traffic.

Discussion

- Give another example of combiner.

Local Execution

- Debugging problems in Map or Reduce functions can be tricky, since the actual computation happens in a distributed system of several thousand machines.
- To help facilitate debugging, profiling, and small-scale testing, we have developed an alternative implementation of the MapReduce library that sequentially executes all of the work for a MapReduce operation on the local machine.

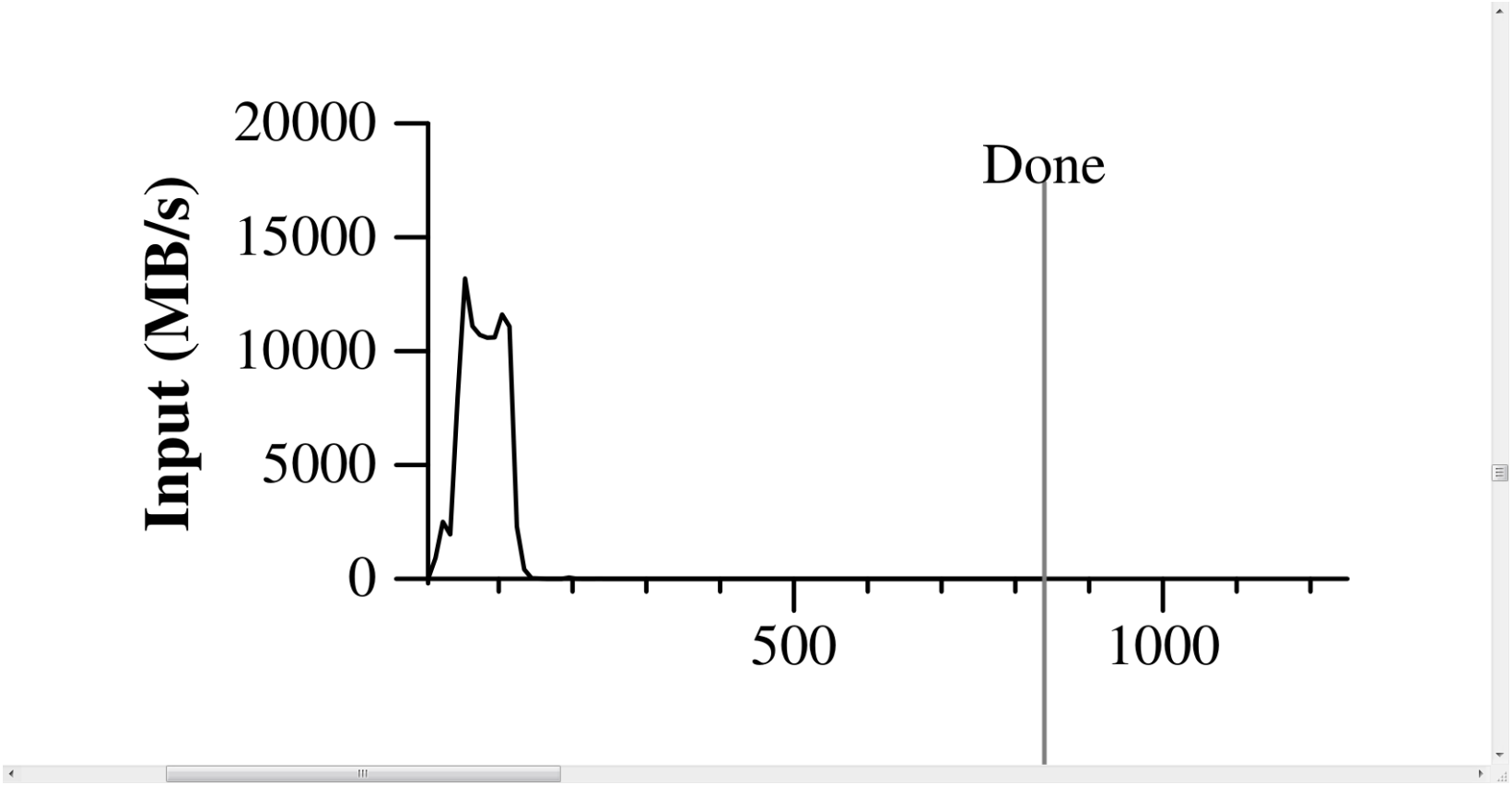
Counters

- The MapReduce library provides a counter facility to count occurrences of various events.
- Some counter values are automatically maintained by the MapReduce library, such as the number of input key/value pairs processed and the number of output key/value pairs produced.
- Users have found the counter facility useful for sanity checking the behavior of MapReduce operations.

Performance

- The experiments run on a cluster of 1800 machines connected by a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root.
- Each machine had two 2GHz Intel Xeon processors with Hyper-Threading enabled, 4GB of memory, two 160GB IDE disks.
- The benchmarks include Grep and sort.

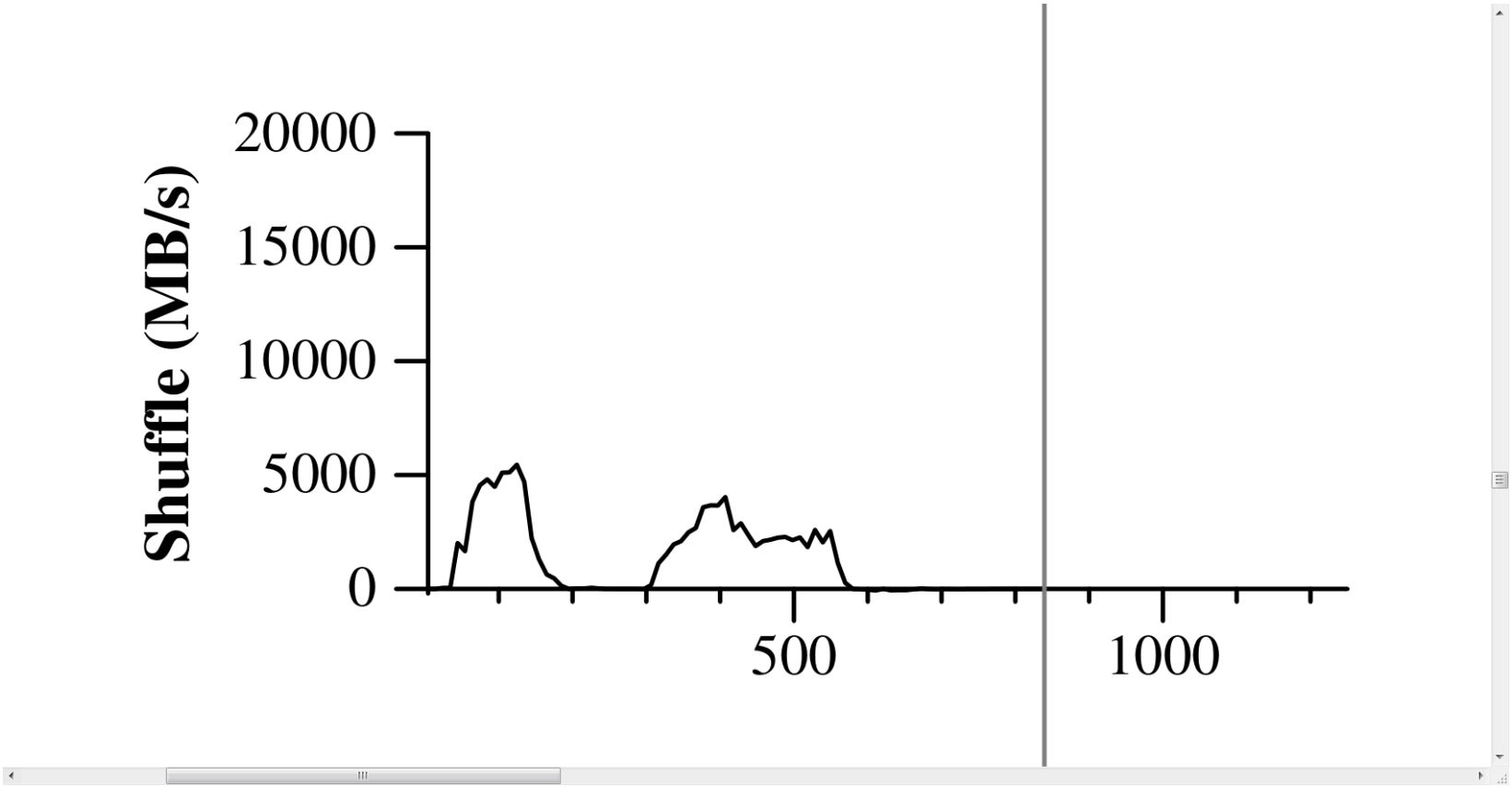
Mapper Reading



Mapper Reading Rate

- This graph shows the rate at which input is read.
- The rate peaks at about 13 GB/s and dies off fairly quickly since all map tasks finish before 200 seconds have elapsed.

Shuffle



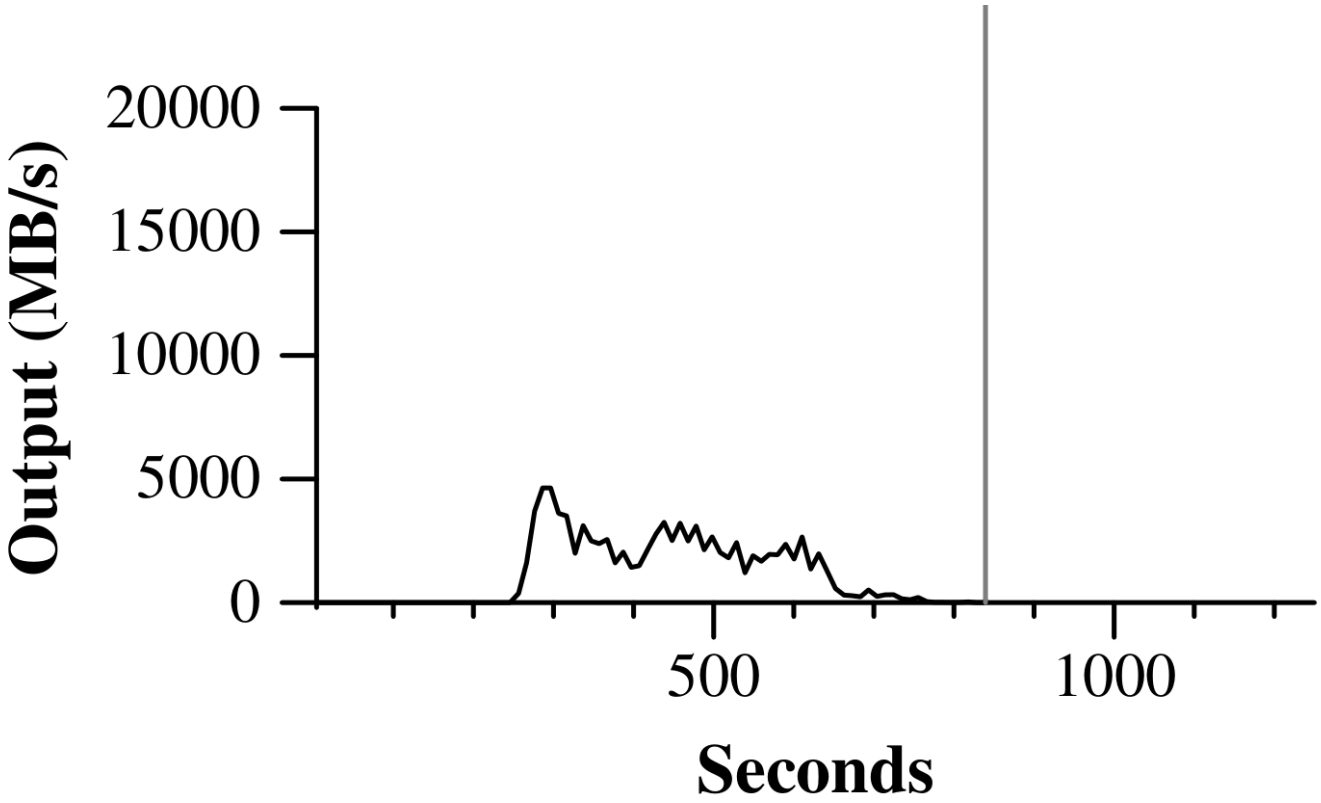
Network Transfer Rate

- This graph shows the rate at which data is sent over the network from the map tasks to the reduce tasks.
- This shuffling starts as soon as the first map task completes.

Network Transfer Rate

- The first hump in the graph is for the first batch of approximately 1700 reduce tasks (the entire MapReduce was assigned about 1700 machines and each machine executes at most one reduce task at a time).
- Roughly 300 seconds into the computation, some of these first batch of reduce tasks finish and we start shuffling data for the remaining reduce tasks.

Reducer Writing



Reducer Writing Rate

- There is a delay between the end of the first shuffling period and the start of the writing period because the machines are busy sorting the intermediate data.
- The writes continue at a rate of about 2-4 GB/s for a while and all writes finish about 850 seconds into the computation.

Observation

- The input rate is higher than the shuffle rate and the output rate because of our locality optimization.
- The shuffle rate is higher than the output rate because the output phase writes two copies of the sorted data (we make two replicas of the output for reliability and availability reasons).