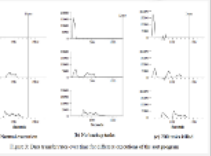
individual web pages

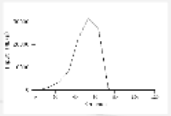
(source: Google Insights)

MR_Sort

backup tasks reduce job completion time significantly
system deals well with failures



MR_Grep



Locality optimization helps:
- 1800 machines read 1 TB of data at peak of ~31 GB/s
- Without this, rack switches would limit to 10 GB/s

Startup overhead is significant for short jobs

Performance

Tests run on cluster of 1800 machines:
- 4 GB of memory
- Dual processor 2 GHz Xeons with Hyperthreading
- Dual 160 GB IDE disks
- Gigabit Ethernet per machine
- Bisection bandwidth approximately 100 Gbps

Two benchmarks:

1. MR_Grep Scan: 10^10 100-byte records to extract records matching a rare pattern (92K matching records)

2. MR_Sort Sort: 10^10 100-byte records (modeled after TeraSort benchmark)

MapReduce: Simplified Data Processing on Large Clusters

By: Bikramdeep Singh

Motivation

130 trillion
individual web pages

(source: Google Insights)

Motivation

100 million GB
index data on google

- (source: Google Insights)

Some questions...

- What is the most frequent search query in a given day?

- How many number of pages are crawled per host?

- What is the most popular video in youtube Canada?

MapReduce provides:

- Automatic parallelization and distribution

- Fault-tolerance

- I/O scheduling

- Status and monitoring

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

See it in action...

https://goo.gl/DykPHB

Some other examples

- Distributed Grep
- Count of URL access frequency
- Reverse Web-Link Graph
- Term Vector per host
- Inverted Index
- Distributed Sort

Implementation Overview

Typical cluster:

- 100s/1000s of 2-CPU x86 machines, 2-4 GB of memory
- Limited bisection bandwidth
- Storage is on local IDE disks
- GFS: distributed file system manages data
- Job scheduling system: jobs made up of tasks, scheduler assigns tasks to machines

Implementation is a C++ library linked into user programs

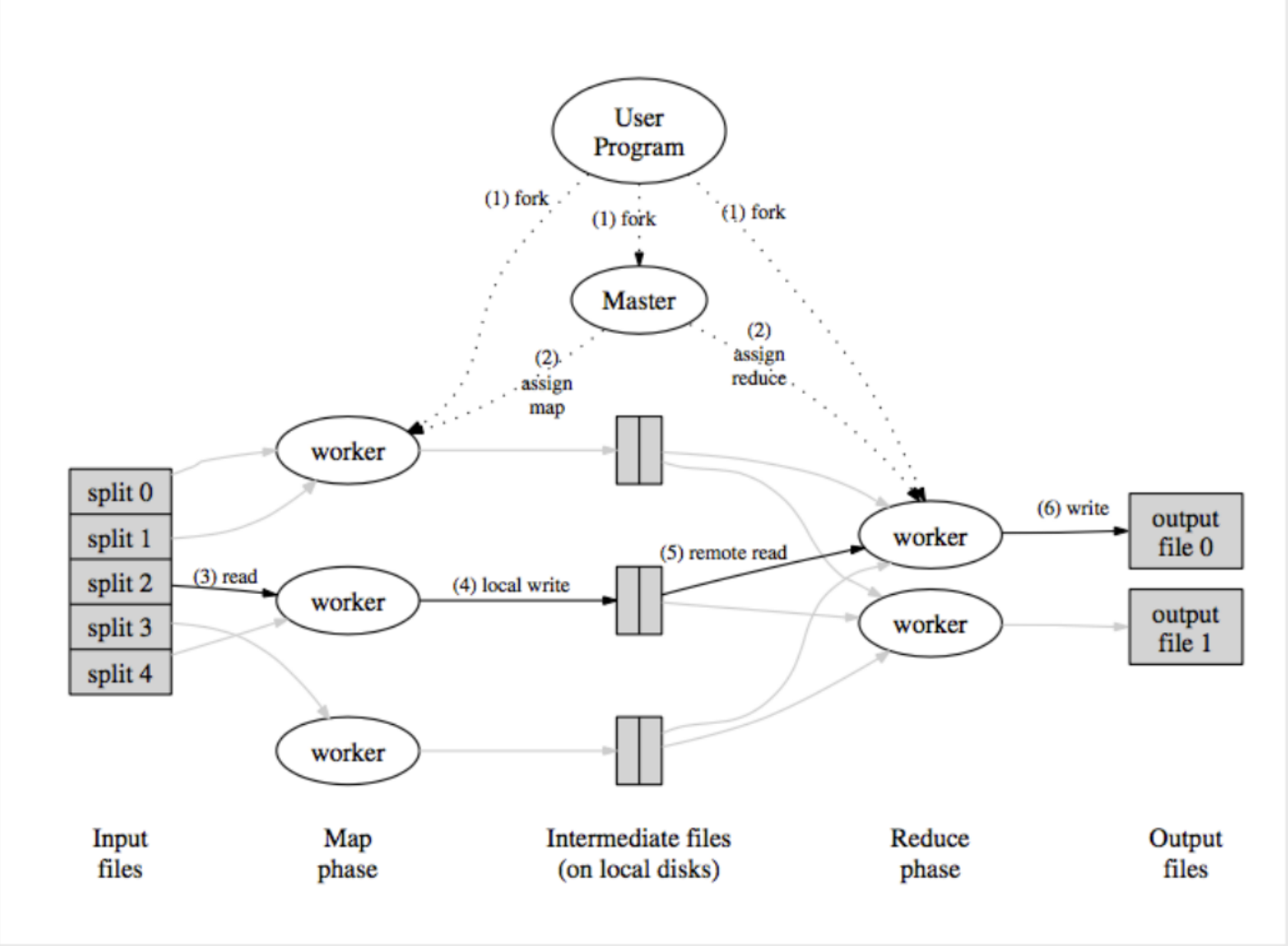
Implementation overview

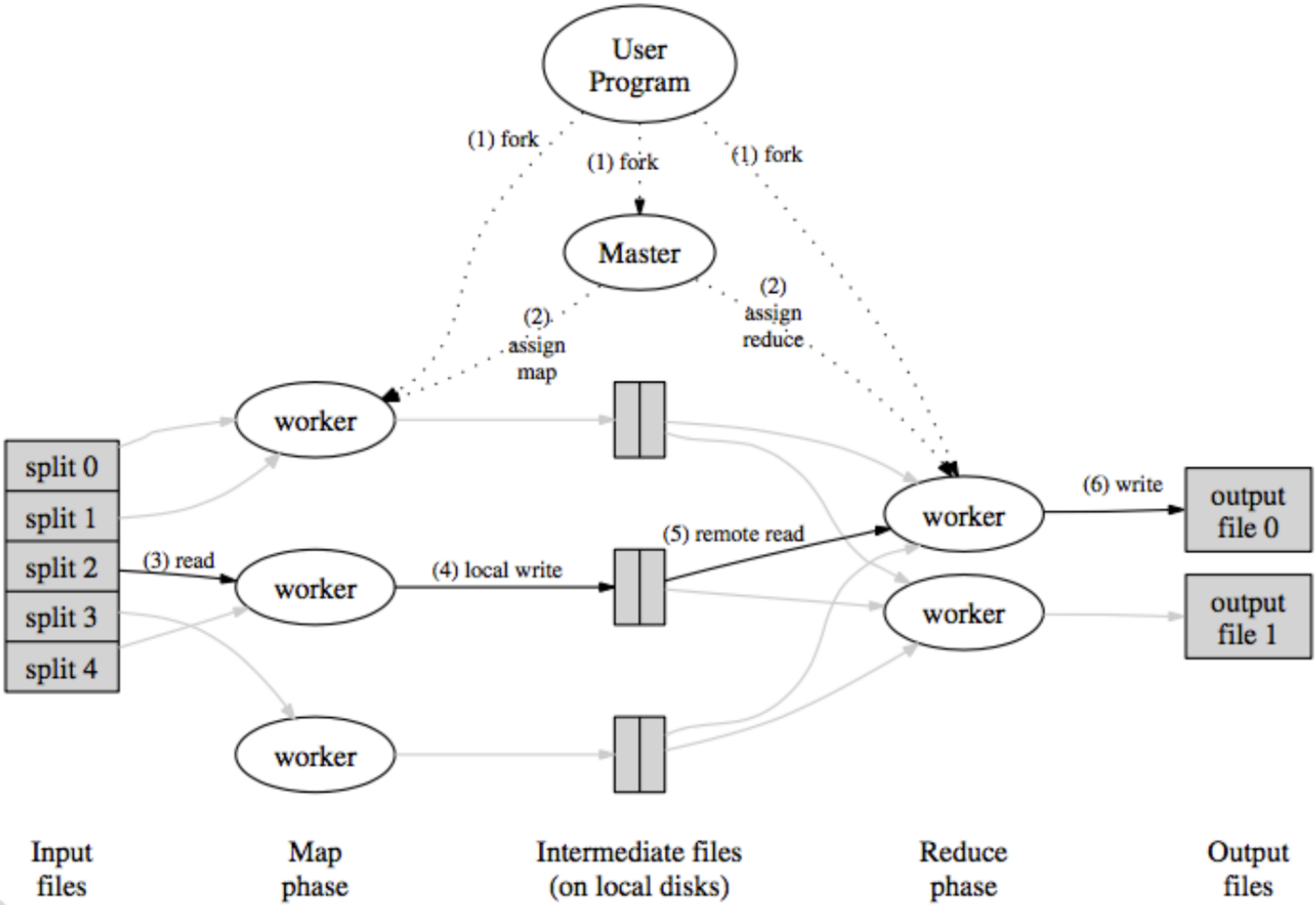
Typical cluster:

- 100s/1000s of 2-CPU x86 machines, 2-4 GB of memory
- Limited bisection bandwidth
- Storage is on local IDE disks
- GFS: distributed file system manages data
- Job scheduling system: jobs made up of tasks, scheduler assigns tasks to machines

Implementation is a C++ library linked into user programs

Execution


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


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

Fault Tolerance

On worker failure:

- Detect failure via periodic heartbeats
- Re-execute completed and in-progress map tasks
- Re-execute in progress reduce tasks
- Task completion committed through master

Master failure:
Could handle, but don't yet (master failure unlikely)

Robust: lost 1600 of 1800 machines once, but finished fine

Refinement: Redundant Execution

Slow workers (stragglers) significantly lengthen completion time

- Other jobs consuming resources on machine
- Bad disks with soft errors transfer data very slowly
- Processor caches disabled

Solution: Near end of phase, spawn backup copies of tasks, whichever one finishes first "wins"

Effect: Dramatically shortens job completion time

Refinement: Locality Optimization

Master scheduling policy:

- Asks GFS for locations of replicas of input file blocks
- Map tasks typically split into 64MB (== GFS block size)
- Map tasks scheduled so GFS input block replica are on same machine or same rack

Effect: Thousands of machines read input at local disk speed

Without this, rack switches limit read rate

Refinement: Skipping Bad Records

Map/Reduce functions sometimes fail for particular inputs

- Best solution is to debug & fix, but not always possible
- On seg fault:
  - Send UDP packet to master from signal handler
  - Include sequence number of record being processed
- If master sees two failures for same record:
- Next worker is told to skip the record

Effect: Can work around bugs in third-party libraries

Other Refinements

- Sorting guarantees within each reduce partition
- Compression of intermediate data
- Combiner: useful for saving network bandwidth
- Local execution for debugging/testing
- User-defined counters

Performance

Tests run on cluster of 1800 machines:

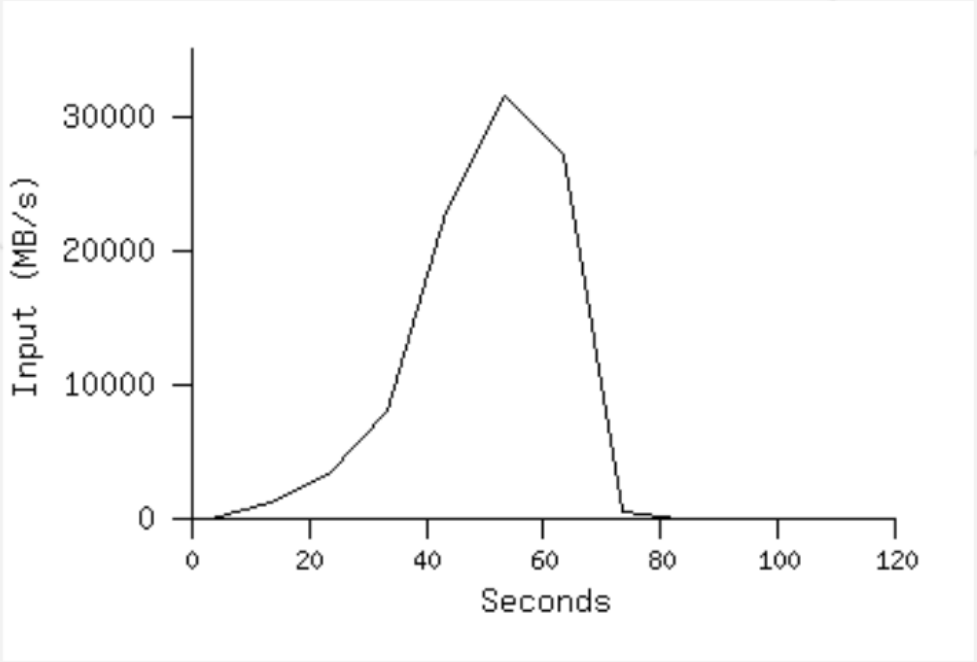
- 4 GB of memory
- Dual-processor 2 GHz Xeons with Hyperthreading
- Dual 160 GB IDE disks
- Gigabit Ethernet per machine
- Bisection bandwidth approximately 100 Gbps

Two benchmarks:

1. MR_Grep Scan: 10^10 100-byte records to extract records matching a rare pattern (92K matching records)

2. MR_Sort Sort: 10^10 100-byte records (modeled after TeraSort benchmark)

MR_Grep



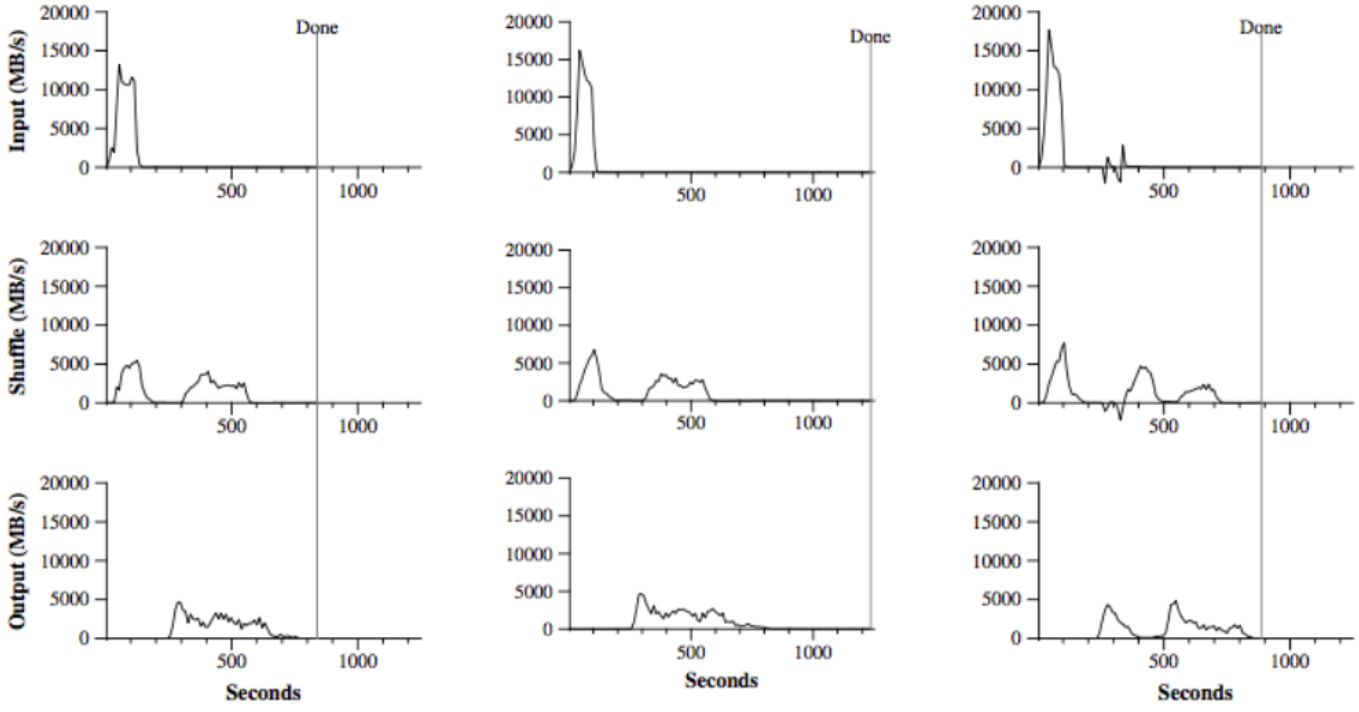
Locality optimization helps:

- 1800 machines read 1 TB of data at peak of ~31 GB/s
- Without this, rack switches would limit to 10 GB/s

Startup overhead is significant for short jobs

MR_Sort

- Backup tasks reduce job completion time significantly
- System deals well with failures



(a) Normal execution (b) No backup tasks (c) 200 tasks killed

Figure 3: Data transfer rates over time for different executions of the sort program

- System deals well with failures

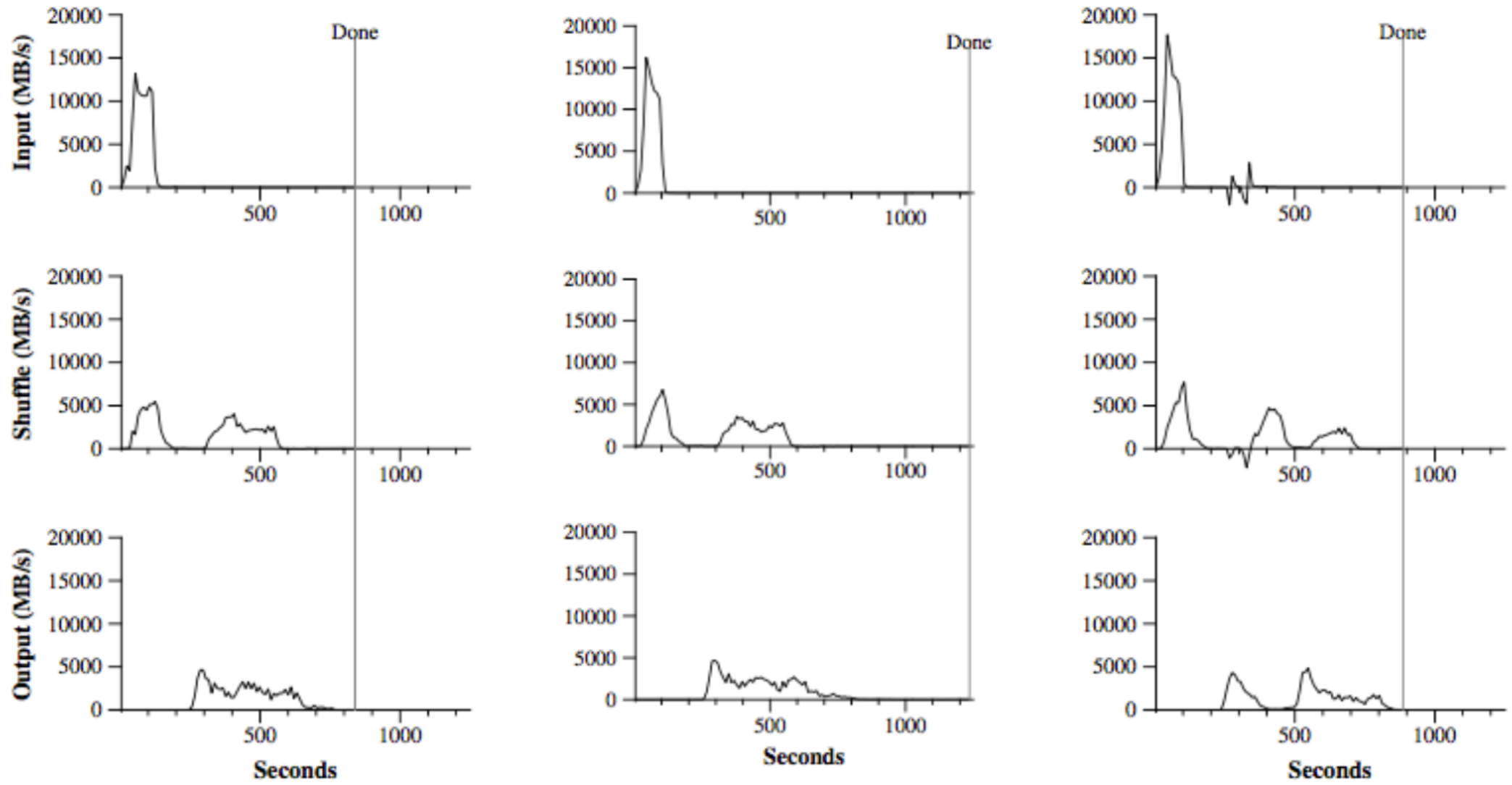


Figure 3: Data transfer rates over time for different executions of the sort program

Experience

Rewrote Google's production indexing system using MapReduce

- New code is simpler, easier to understand
- MapReduce takes care of failures, slow machines
- Easy to make indexing faster by adding more machines

Conclusion

- MapReduce has proven to be a useful abstraction
- Greatly simplifies large-scale computations at Google (and the world ;))
- Fun to use: focus on problem, let library deal w/ messy details

Thank You