MapReduce 介紹

gooqwerty777@gmail.com

Introduction

- Motivation: lots of special-purpose programs should process large amounts of raw data
  - crawl(analyze) documents, web request logs, etc.
  - should use lots of machine to reduce processing time
  - implementation is time-consuming and complex
- Solution: Design a programming model–MapReduce
  - Hides the details of parallelization, fault-tolerance, locality optimization, and load balancing.

Map and Reduce

- Divide, Conquer, and Combine

-> Divide, Maps, and Reduces

- User only need to implement Map() and Reduce() functions!(and some arguments)

Programming Model

Map

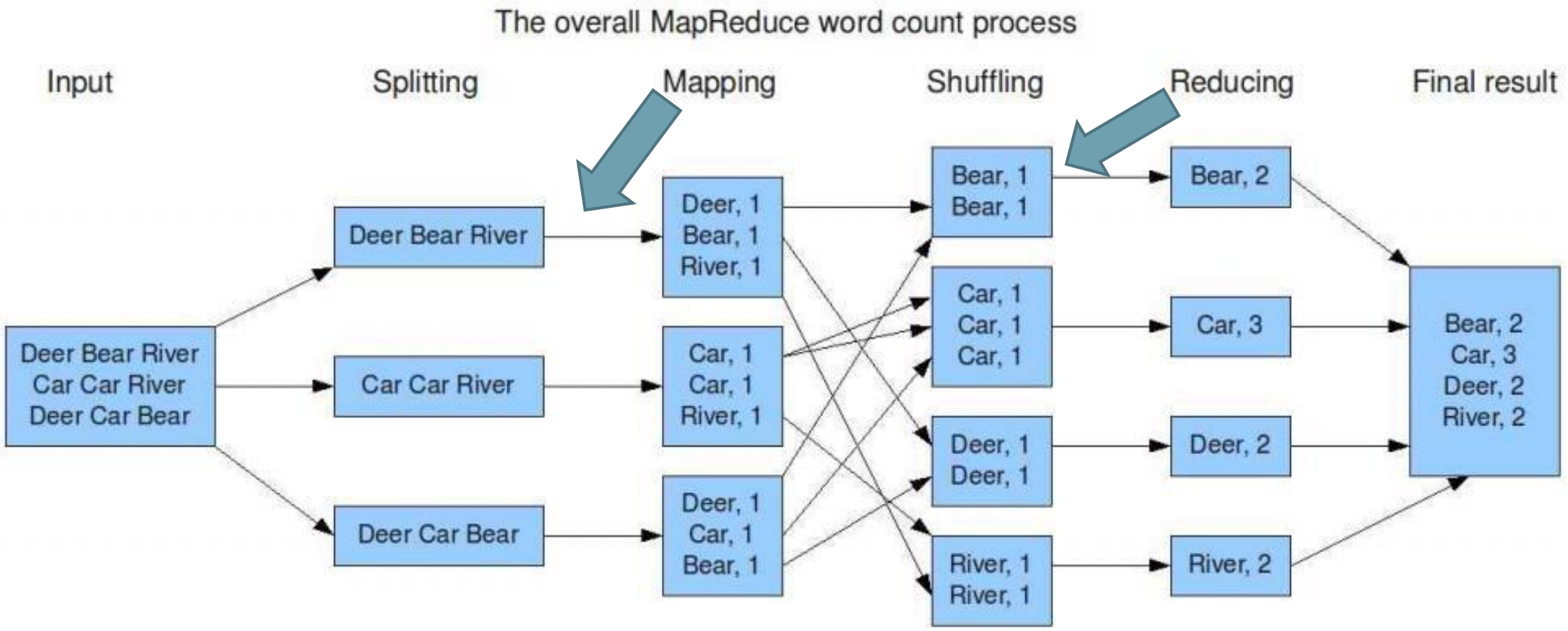
- Take an input pair
- produces a set of intermediate key/value pairs

Shuffling

- sorted intermediate pairs by key value
- groups together all intermediate values with the same intermediate key

Reduce

- Take intermediate key and value set of key
- merges together these value

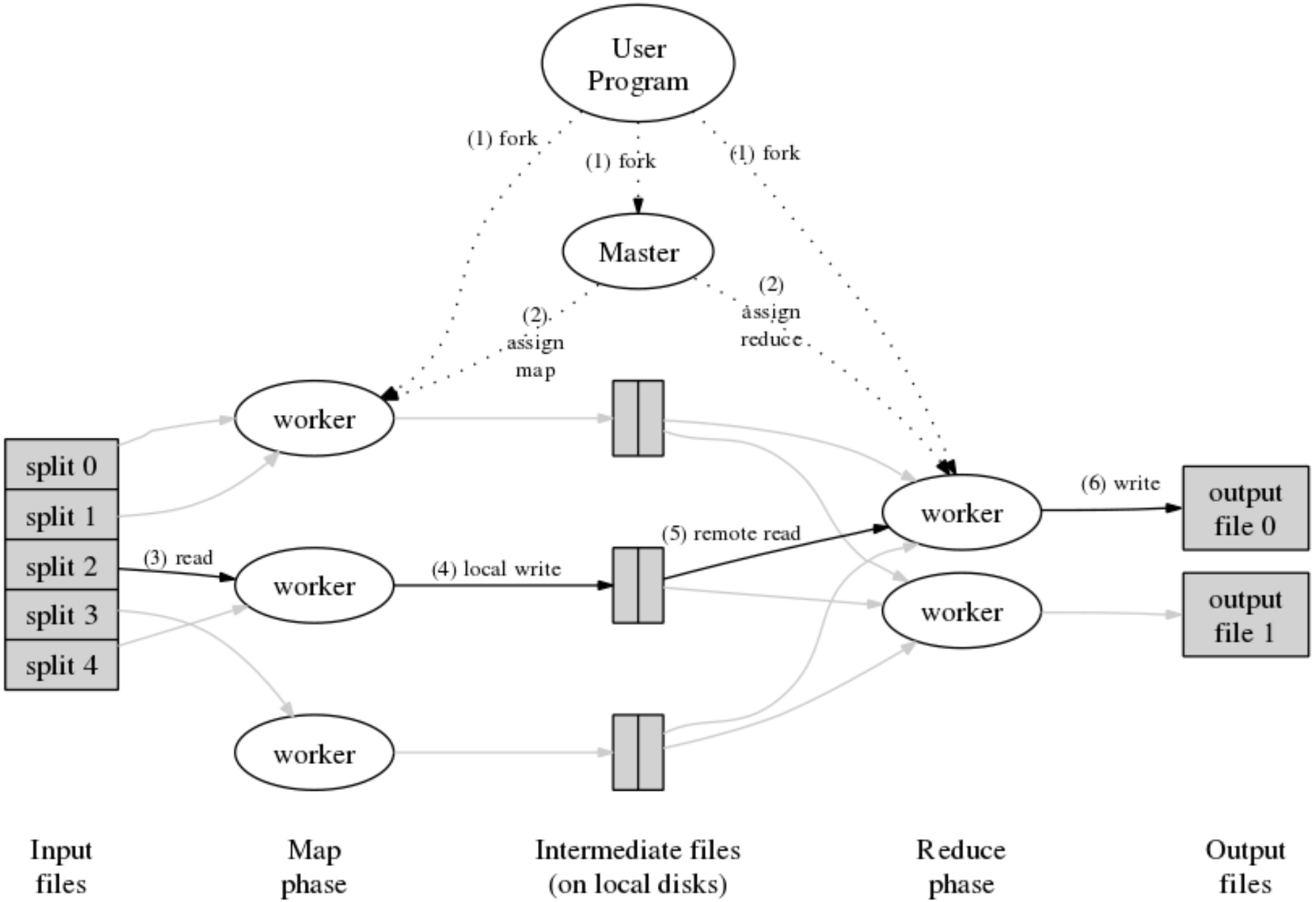

The overall MapReduce word count process
Input
Splitting
Mapping
Shuffling
Reducing
Final result
Deer Bear River
Car Car River
Deer Car Bear
Deer Bear River
Car Car River
Deer Car Bear
Deer, 1
Bear, 1
River, 1
Car, 1
Car, 1
River, 1
Deer, 1
Car, 1
Bear, 1
Bear, 1
Bear, 1
Car, 1
Car, 1
Car, 1
Deer, 1
Deer, 1
River, 1
River, 1
Bear, 2
Car, 3
Deer, 2
River, 2
Bear, 2
Car, 3
Deer, 2
River, 2

More Example

- Inverted Index
  - Find specified word in set of files
  - Input: <files(splited), docID>
  - Intermediate: <word, docID>
  - Final: <word, list<docID>>
- Distributed Grep
- Distributed Sort
- Count of URL Access Frequency
- Term-Vector per Host

More Example

- Distributed Grep
- Distributed Sort
- Inverted Index
  - Intermediate: <word, docID> → Final: <word, list<docID>>
- Count of URL Access Frequency
  - <URL, 1> → <URL, TotalCount>
- Term-Vector per Host
  - summarize the most important words in docs
  - <term, freq> → vector<term, freq>


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(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

Implementation

- Parallelization
  - Input of Map: partitioning the input data into M splits
  - Input of Reduce: partitioning intermediate data into R files
- Master program: assign M map tasks and R reduce tasks to worker programs
  - Map workers: Intermediate key/value pairs are written to local disk. Locations of these files would pass back to master.
  - Reduce workers: Get location from master and use remote procedure calls(RPC) to read data in local disk.

Fault Tolerance

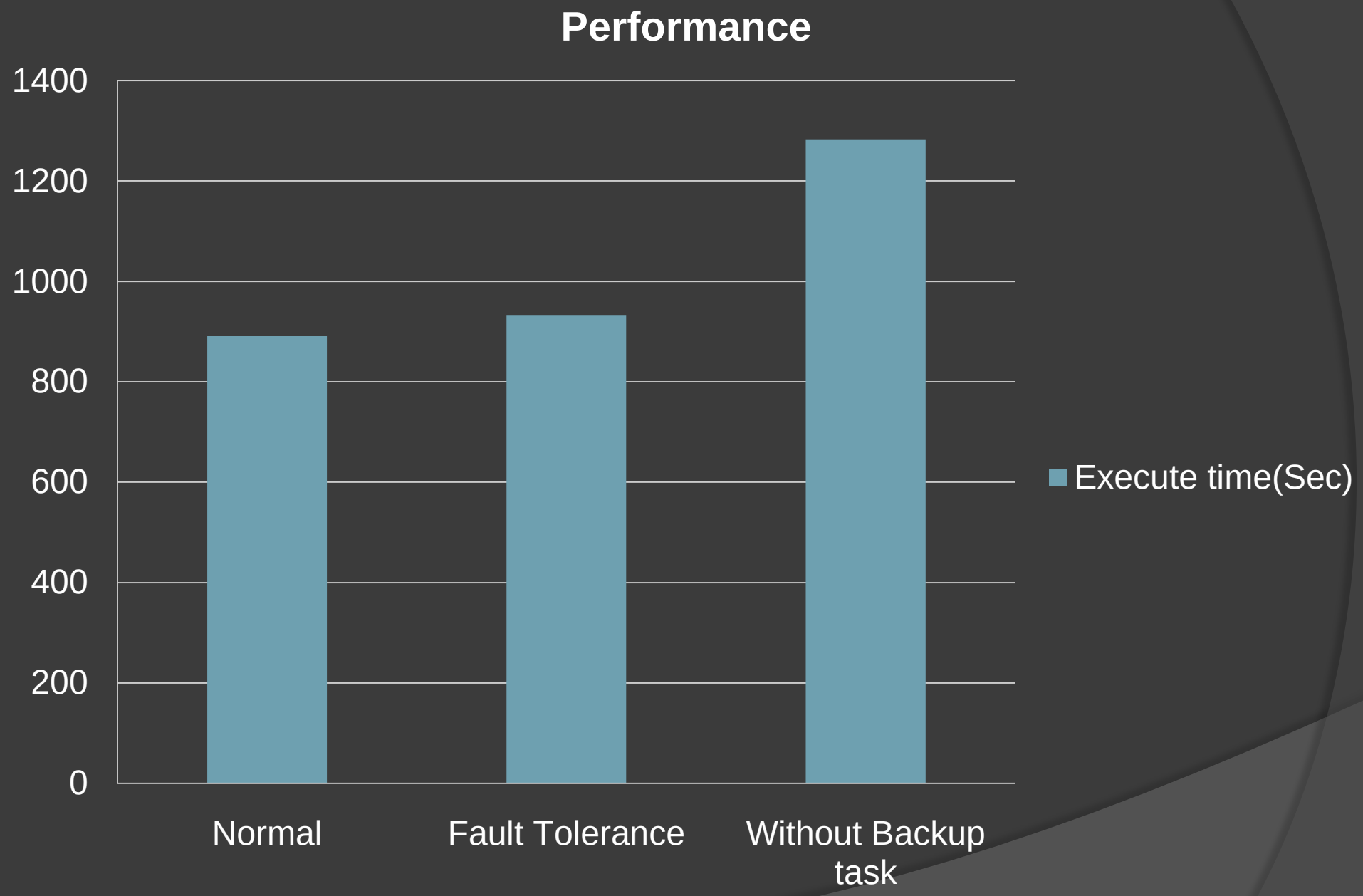
- Worker Failure
  - ping every worker periodically
  - tasks in failed machine :
    - rescheduling now assigning task
    - reset completed map tasks and rescheduling
    - but completed reduce tasks don't need to reset (files are stored in global file system)
- Master Failure
  - failure of master is unlikely
  - aborts the MapReduce, clients should check and retry it

Backup Tasks

- Straggler : machine that takes an unusually long time to complete tasks
  - bad disk
  - other tasks
- Solution: when MapReduce is close to completion, master schedules backup executions of the remaining in-progress tasks
  - Only wait one of they to complete
  - Takes 30% less time to complete, with computational resources increase by no more than a few percent

Performance

- Environment
  - 1800 machines
  - two 2GHz Intel Xeon processors with Hyper-Threading enabled, 4GB of memory, two 160GB IDE disks, and a gigabit Ethernet link.
- Sorts approximately 1TB of data
  - 891sec
- intentionally killed 200 out of 1746 workers several minutes
  - 933sec (just 5% increase)
- No backup tasks
  - 1283sec (44% increase)


Performance
1400
1200
1000
800
600
400
200
0
Normal
Fault Tolerance
Without Backup task
Execute time(Sec)

Advantage

- Large variety of problems are easily expressible as MapReduce
  - Every work which can be divided!
- Easy to use for programmers who have no experience with distributed or parallel systems
  - What you think is how to deal with splited data, and how to compose result
- Code is simpler, easier to understand and modify

Application

- large-scale machine learning
- extraction of data used to produce reports of popular queries
- extraction of properties of web pages
  - PageRank
- Open Source implementation
  - Hadoop

Refinements

- Locality optimization
  - Input file copies in local disks
- Skipping Bad Records
  - ignore a few bad records, when doing statistical analysis on a large data set.
  - signal handler (When error, send information to master)
- Counter object
  - Piggybacked on the ping response

Refinements

- Locality optimization
  - Input file copies in local disks
- Intermediate key/value pairs is in order
  - Utilized sort and random access
- Input and Output Types
  - the key is the offset in the file and the value is the contents of the line.
  - reader interface
- Skipping Bad Records
  - ignore a few bad records, when doing statistical analysis on a large data set.
  - signal handler (When error, send information to master)
- Counter object
  - Piggybacked on the ping response
- Combiner function
  - partial merging at local disk before sending record

Reference

- Jeffrey Dean and Sanjay Ghemawat, 2004, MapReduce: Simplified Data Processing on Large Clusters

The End