An Innovative Approach to Parallel Processing Data
The Context: Big-data

- Man on the moon with 32KB (1969); my laptop had 2GB RAM (2009)
- Google collects 270PB data in a month (2007), 20PB a day (2008) ...
- 2010 census data is a huge gold mine of information
- Data mining huge amounts of data collected in a wide range of domains from astronomy to healthcare has become essential for planning and performance.
- We are in a knowledge economy.
  - Data is an important asset to any organization
  - Discovery of knowledge; Enabling discovery; annotation of data
- We are looking at newer
  - programming models, and
  - Supporting algorithms and data structures
- National Science Foundation refers to it as “data-intensive computing” and industry calls it “big-data” and “cloud computing”
Rear Admiral Grace Hopper: “In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers.”

---From the Wit and Wisdom of Grace Hopper (1906-1992),
http://www.cs.yale.edu/homes/tap/Files/hopper-wit.html
Text processing: web-scale corpora (singular corpus)
Simple word count, cross reference, n-grams, ...
A simpler technique on more data beat a more sophisticated technique on less data.
Google researchers call this: “unreasonable effectiveness of data”
--Alon Halevy, Peter Norvig, and Fernando Pereira.
The unreasonable effectiveness of data.
MapReduce
What is MapReduce?

- MapReduce is a programming model Google has used successfully in processing its “big-data” sets (~ 20 peta bytes per day in 2008)
  - Users specify the computation in terms of a map and a reduce function,
  - Underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, and
  - Underlying system also handles machine failures, efficient communications, and performance issues.

Big idea behind MR

- **Scale-out** and not scale-up: Large number of commodity servers as opposed to a large number of high-end specialized servers
  - Economies of scale, ware-house scale computing
  - MR is designed to work with clusters of commodity servers
  - Research issues: Read Barroso and Holzle’s work

- **Failures are norm** or common:
  - With typical reliability, MTBF of 1000 days (about 3 years), if you have a cluster of 1000, the probability of at least 1 server failure at any time is nearly 100%
• **Moving “processing” to the data**: not literally, data and processing are co-located versus sending data around as in HPC

• **Process data sequentially vs random access**: analytics on large sequential bulk data as opposed to search for one item in a large indexed table

• **Hide system details from the user application**: user application does not have to get involved in which machine does what. Infrastructure can do it.

• **Seamless scalability**: Can add machines / server power without changing the algorithms: this is in-order to process larger data set
Issues to be addressed

- How to break large problem into smaller problems? Decomposition for parallel processing
- How to assign tasks to workers distributed around the cluster?
- How do the workers get the data?
- How to synchronize among the workers?
- How to share partial results among workers?
- How to do all these in the presence of errors and hardware failures?
- MR is supported by a distributed file system that addresses many of these aspects.
Fundamental concept:

Key-value pairs form the basic structure of MapReduce <key, value>

Key can be anything from a simple data types (int, float, etc) to file names to custom types.

Examples:

- <docid, docitself>
- <yourName, yourLifeHistory>
- <graphNode, nodeCharacteristicsComplexData>
- <yourId, yourFollowers>
- <word, itsNumofOccurences>
- <planetName, planetInfo>
- <geneNum, <{pathway, geneExp, proteins}>
- <Student, stuDetails>
Consider a large data collection:
{web, weed, green, sun, moon, land, part, web, green,...}

Problem: Count the occurrences of the different words in the collection.

Let's design a solution for this problem:
- We will start from scratch
- We will add and relax constraints
- We will do incremental design, improving the solution for performance and scalability
Word Counter and Result Table

{web, weed, green, sun, moon, land, part, web, green,...}

Data collection

Main

DataCollection

WordCounter

ResultTable

parse()
count()

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>web</td>
<td>2</td>
</tr>
<tr>
<td>weed</td>
<td>1</td>
</tr>
<tr>
<td>green</td>
<td>2</td>
</tr>
<tr>
<td>sun</td>
<td>1</td>
</tr>
<tr>
<td>moon</td>
<td>1</td>
</tr>
<tr>
<td>land</td>
<td>1</td>
</tr>
<tr>
<td>part</td>
<td>1</td>
</tr>
</tbody>
</table>
Multiple Instances of Word Counter

Observe:
Multi-thread
Lock on shared data
Improve Word Counter for Performance

No need for lock

Separate counters

<table>
<thead>
<tr>
<th>KEY</th>
<th>web</th>
<th>weed</th>
<th>green</th>
<th>sun</th>
<th>moon</th>
<th>land</th>
<th>part</th>
<th>web</th>
<th>green</th>
<th>.......</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.......</td>
</tr>
</tbody>
</table>

CSE4/587 B. Ramamurthy
Peta-scale Data

Data collection

Main

Thread

Parser

Counter

DataCollection

WordList

ResultTable

KEY

VALUE

web 2
weed 1
green 2
sun 1
moon 1
land 1
part 1
Addressing the Scale Issue

- Single machine cannot serve all the data: you need a distributed special (file) system
- Large number of commodity hardware disks: say, 1000 disks 1TB each
  - Issue: With Mean time between failures (MTBF) or failure rate of 1/1000, then at least 1 of the above 1000 disks would be down at a given time.
  - Thus failure is norm and not an exception.
  - File system has to be fault-tolerant: replication, checksum
  - Data transfer bandwidth is critical (location of data)

- Critical aspects: fault tolerance + replication + load balancing, monitoring
- Exploit parallelism afforded by splitting parsing and counting
- Provision and locate computing at data locations
Peta-scale Data

Data collection

Diagram:
- Main
- Thread
- Parser
- Counter
- DataCollection
- WordList
- ResultTable

Table:
<table>
<thead>
<tr>
<th>KEY</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>web</td>
<td>2</td>
</tr>
<tr>
<td>weed</td>
<td>1</td>
</tr>
<tr>
<td>green</td>
<td>2</td>
</tr>
<tr>
<td>sun</td>
<td>1</td>
</tr>
<tr>
<td>moon</td>
<td>1</td>
</tr>
<tr>
<td>land</td>
<td>1</td>
</tr>
<tr>
<td>part</td>
<td>1</td>
</tr>
</tbody>
</table>

CSE4/587 B. Ramamurthy
Peta Scale Data is Commonly Distributed

Issue: managing the large scale data

<table>
<thead>
<tr>
<th>KEY</th>
<th>web</th>
<th>weed</th>
<th>green</th>
<th>sun</th>
<th>moon</th>
<th>land</th>
<th>part</th>
<th>web</th>
<th>green</th>
<th>........</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Write Once Read Many (WORM) data

Data collection

Data collection

Data collection

Data collection

KEY

VALUE

web 2
weed 1
green 2
sun 1
moon 1
land 1
part 1
WORM Data is Amenable to Parallelism

1. Data with WORM characteristics: yields to parallel processing;
2. Data without dependencies: yields to out of order processing
For our example,

#1: Schedule parallel parse tasks
#2: Schedule parallel count tasks

This is a particular solution; 
Lets generalize it:

Our parse is a mapping operation:
MAP: input → <key, value> pairs

Our count is a reduce operation:
REDUCE: <key, value> pairs reduced

Map/Reduce originated from Lisp
But have different meaning here

Runtime adds distribution + fault tolerance + replication + monitoring + load balancing to your base application!
Mapper and Reducer

Remember: MapReduce is simplified processing for larger data sets
Map Operation

MAP: Input data ➞ <key, value> pair

Data Collection: split 1

Data Collection: split 2

Data Collection: split n

Split the data to
Supply multiple processors

2/19/2018
MapReduce Example #2

Cat

Bat

Dog

Other Words (size: TByte)

split  map  combine  reduce

split  map  combine  reduce

split  map  combine  reduce

split  map  combine  reduce

barrier

part0

part1

part2

CSE4/587 B. Ramamurthy

2/19/2018
You focus on Map function, Reduce function and other related functions like combiner etc.

Mapper and Reducer are designed as classes and the function defined as a method.

Configure the MR “Job” for location of these functions, location of input and output (paths within the local server), scale or size of the cluster in terms of #maps, # reduce etc., run the job.

Thus a complete MapReduce job consists of code for the mapper, reducer, combiner, and partitioner, along with job configuration parameters. The execution framework handles everything else.

The way we configure has been evolving with versions of hadoop.
The code

1: class Mapper
2: method Map(docid a; doc d)
3: for all term t in doc d do
4: Emit(term t; count 1)

1: class Reducer
2: method Reduce(term t; counts [c1; c2; : : :])
3: sum = 0
4: for all count c in counts [c1; c2; : : :] do
5: sum = sum + c
6: Emit(term t; count sum)
This is a cat
Cat sits on a roof

The roof is a tin roof
There is a tin can on the roof

Cat kicks the can
It rolls on the roof and falls on the next roof

The cat rolls too
It sits on the can

Problem: Count the word frequency. Include all the words. We will worry about stop words and stemming later.
This is a cat
Cat sits on a roof
<this 1> <is 1> <a 1> <cat 1> <cat 1> <sits 1> <on 1><a 1> <roof 1>

The roof is a tin roof
There is a tin can on the roof
<the 1> <roof 1> <is 1> <a 1> <tin 1 ><roof 1> <there 1> <is 1> <a 1> <tin 1><can 1> <on 1><the 1> <roof 1>

Cat kicks the can
It rolls on the roof and falls on the next roof
<cat 1> <kicks 1> <the 1><can 1> <it 1> <rolls 1> <on 1> <the 1> <roof 1> <and 1> <falls 1><on 1> <the 1> <next 1> <roof 1>

The cat rolls too
It sits on the can
<the 1> <cat 1> <rolls 1> <too 1> <it 1> <sits 1> <on 1> <the 1> <can 1>
MapReduce Example: Shuffle to the Reducer

Output of Mappers:
<this 1> <is 1> <a 1> <cat 1> <cat 1> <sits 1> <on 1> <a 1> <roof 1>
<the 1> <roof 1> <is 1> <a 1> <tin 1> <roof 1> <there 1> <is 1> <a 1> <tin 1> <can 1> <on 1> <the 1> 
<roof 1>
<cat 1> <kicks 1> <the 1> <can 1> <it 1> <rolls 1> <on 1> <the 1> <roof 1> <and 1> <falls 1> <on 1> 
<the 1> <next 1> <roof 1>
<the 1> <cat 1> <rolls 1> <too 1> <it 1> <sits 1> <on 1> <the 1> <can 1>

Input to the reducer: delivered sorted... By key
...
<can <1, 1>>
<cat <1,1,1,1>>
...
<roof <1,1,1,1,1,1>>
.....
Reduce (sum in this case) the counts: comes out sorted!!!
..
<can 2>
<cat 4>
..
<roof 6>
All Mappers work in parallel.
Barriers enforce all mappers completion before Reducers start.
Mappers and Reducers typically execute on the same machine
You can configure job to have other combinations besides Mapper/Reducer: ex: identify mappers/reducers for realizing “sort” (that happens to be a Benchmark)
Mappers and reducers can have side effects; this allows for sharing information between iterations.
MapReduce Characteristics

- Very large scale data: peta, exa bytes
- Write once and read many data: allows for parallelism without mutexes
- Map and Reduce are the main operations: simple code
- There are other supporting operations such as combine and partition: we will look at those later.
- Operations are provisioned near the data.
- Commodity hardware and storage.
- Runtime takes care of splitting and moving data for operations.
- Special distributed file system: Hadoop Distributed File System and Hadoop Runtime.
Classes of problems “mapreducable”

- Benchmark for comparing: Jim Gray’s challenge on data-intensive computing. Ex: “Sort”
- Google uses it (we think) for wordcount, adwords, pagerank, indexing data.
- Simple algorithms such as grep, text-indexing, reverse indexing
- Bayesian classification: data mining domain
- Facebook uses it for various operations: demographics
- Financial services use it for analytics
- Astronomy: Gaussian analysis for locating extra-terrestrial objects.
- Expected to play a critical role in semantic web and web3.0
Scope of MapReduce

Data size: small
- Pipelined Instruction level
  - Single-core
  - Multi-core
  - Service Object level
  - Cluster
  - Indexed File level
    - Embarrassingly parallel processing
  - Mega Block level
    - MapReduce, distributed file system
  - Virtual System Level
    - Cloud computing
  - Data size: large
    - Multi-core, single processor
    - Multi-core, multi-processor
    - Cluster of processors (single or multi-core) with shared memory
    - Cluster of processors with distributed memory

Data size: large
- Map function maps one <key,value> space to another. One to many: “expand” or “divide”
- Reduce does that too. But many to one: “merge”
- There can be multiple “maps” in a single machine...
- Each mapper(map) runs parallel with and independent of the other (think of a bee hive)
- All the outputs from mappers are collected and the “key space” is partitioned among the reducers. (what do you need to partition?)
- Now the reducers take over. One reduce/per key (by default)
- Reduce operation can be anything.. Does not have to be just counting...(operation [list of items]) – You can do magic with this concept.
Hadoop
What is Hadoop?

- At Google MapReduce operation are run on a special file system called Google File System (GFS) that is highly optimized for this purpose.
- GFS is not open source.
- Doug Cutting and Yahoo! reverse engineered the GFS and called it Hadoop Distributed File System (HDFS).
- The software framework that supports HDFS, MapReduce and other related entities is called the project Hadoop or simply Hadoop.
- This is open source and distributed by Apache.
Hadoop

2013/14

Hadoop 1.0  Hadoop 2.0
What has changed? Hmm...
Basic Features: HDFS

- Highly fault-tolerant
- High throughput
- Suitable for applications with large data sets
- Streaming access to file system data
- Can be built out of commodity hardware
- HDFS core principles are the same in both major releases of Hadoop.
Hadoop Distributed File System

HDFS Client

Application

Local file system
Block size: 2K

HDFS Server

Masters: Job tracker, Name node, Secondary name node

Slaves: Task tracker, Data Nodes
Block size: 128M Replicated

Block size: 2K

Replicated
Hadoop Distributed File System

**Masters:** Job tracker, Name node, Secondary name node

**Slaves:** Task tracker, Data Nodes

**Block size:** 2K

**Replicated**

**Block size:** 128M

**HDFS Client**

**Local file system**

**Application**

**HDFS Server**
From Brad Hedlund: a very nice picture

Hadoop Cluster

World

switch

switch

Name Node
DN + TT
DN + TT
DN + TT
DN + TT

Job Tracker
DN + TT
DN + TT
DN + TT
DN + TT
DN + TT

Secondary NN
DN + TT
DN + TT
DN + TT
DN + TT
DN + TT
DN + TT

Client
DN + TT
DN + TT
DN + TT
DN + TT
DN + TT
DN + TT
DN + TT
DN + TT

switch

Rack 1
Rack 2
Rack 3
Rack 4
Rack N
• What are: Job tracker, Name node, Secondary name node, data node, task tracker...?
• What are their roles?
• Before we discuss those: lets look a demo of mapreduce on Hadoop MapReduce