Overview of MapReduce and Hadoop
Why Parallelism?

• Data size is increasing
  – Single node architecture is reaching its limit
    • Scan 100 TB on 1 node @ 100 MB/s = 12 days

• Standard/Commodity and affordable architecture emerging
  – Cluster of commodity Linux nodes
  – Gigabit ethernet interconnect
Design Goals for Parallelism

1. **Scalability** to large data volumes:
   - Scan 100 TB on 1 node @ 100 MB/s = 12 days
   - Scan on 1000-node cluster = 16 minutes!

2. **Cost-efficiency**:
   - Commodity nodes *(cheap, but unreliable)*
   - Commodity network
   - **Automatic** fault-tolerance (fewer admins)
   - Easy to use (fewer programmers)
What is MapReduce?

• Data-parallel programming model for clusters of commodity machines

• Pioneered by Google
  – Processes 20 PB of data per day

• Popularized by open-source Hadoop project
  – Used by Yahoo!, Facebook, Amazon, …
What is MapReduce used for?

• At Google:
  – Index building for Google Search
  – Article clustering for Google News
  – Statistical machine translation

• At Yahoo!:
  – Index building for Yahoo! Search
  – Spam detection for Yahoo! Mail

• At Facebook:
  – Data mining
  – Ad optimization
  – Spam detection

• In research:
  – Analyzing Wikipedia conflicts (PARC)
  – Natural language processing (CMU)
  – Bioinformatics (Maryland)
  – Astronomical image analysis (Washington)
  – Ocean climate simulation (Washington)
  – Graph OLAP (NUS)
  – ...
  – ...
  – <Your application>
Challenges

• **Cheap nodes fail, especially if you have many**
  – Mean time between failures for 1 node = 3 years
  – MTBF for 1000 nodes = 1 day
  – **Solution**: Build fault-tolerance into the storage infrastructure (replicate files multiple times)

• **Commodity network = low bandwidth**
  – **Solution**: Push computation to the data

• **Programming distributed systems is hard**
  – **Solution**: Data-parallel programming model: users write “map” and “reduce” functions, system handles work distribution and fault tolerance
Typical Hadoop Cluster

- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 GBps bandwidth in rack, 8 GBps out of rack
- Node specs (Yahoo terasort): 8 x 2.0 GHz cores, 8 GB RAM, 4 disks (= 4 TB?)

In 2011 it was guestimated that Google had 1M machines, http://bit.ly/Shh0RO
Hadoop Components

• Distributed file system (HDFS)
  – Single namespace for entire cluster
  – Replicates data 3x for fault-tolerance

• MapReduce implementation
  – Executes user jobs specified as “map” and “reduce” functions
  – Manages work distribution & fault-tolerance
Hadoop Distributed File System

- Files are **BIG** (100s of GB – TB)
- Typical usage patterns
  - Append-only
    - Data are rarely updated in place
  - Reads common
- Optimized for **large** files, **sequential** (why??) reads
- Files split into 64-128MB blocks (called *chunks*)
  - Blocks replicated (usually 3 times) across several *datanodes* (called chuck or slave nodes)
  - Chunk nodes are compute nodes too

![Diagram](image)
Hadoop Distributed File System

- Single *namenode* (master node) stores metadata (file names, block locations, etc)
  - May be replicated also
- Client library for file access
  - Talks to master to find chunk servers
  - Connects directly to chunk servers to access data
    - Master node is not a bottleneck
    - Computation is done at chuck node (close to data)
MapReduce Programming Model

- Data type: key-value records
- File – a bag of (key, value) records

- Map function:
  \[(K_{in}, V_{in}) \rightarrow \text{list}(K_{inter}, V_{inter})\]
  - Takes a key-value pair and outputs a set of key-value pairs, e.g., key is the line number, value is a single line in the file
  - There is one Map call for every \((k_{in}, v_{in})\) pair

- Reduce function:
  \[(K_{inter}, \text{list}(V_{inter})) \rightarrow \text{list}(K_{out}, V_{out})\]
  - All values \(V_{inter}\) with same key \(K_{inter}\) are reduced together and processed in \(V_{inter}\) order
  - There is one Reduce function call per unique key \(K_{inter}\)
Example: Word Count Execution

Input: the quick brown fox, the fox ate the mouse, how now brown cow

Map: the, 1 brown, 1 fox, 1 quick, 1

Sort & Shuffle:

Reduce: brown, 1

Output: brown, 2 fox, 2 how, 1 now, 1 the, 3

(K_inter, list(V_inter)) → list(K_out, V_out)
Example: Word Count

def mapper(line):
    foreach word in line.split():
        output(word, 1)

           ('brown', 1)
           ('brown', (1, 1))

def reducer(key, values):
    output(key, sum(values))

           ('brown', 2)
Can Word Count algorithm be improved?

• Some questions to consider
  – How many key-value pairs are emitted?
  – What if the same word appear in the same line/document?
  – What is the overhead?
  – How could you reduce that number?
An Optimization: The Combiner

• A combiner is a local aggregation function for repeated keys produced by the same map
  – For associative ops. like sum, count, max
  – Decreases size of intermediate data

• Example: local counting for Word Count:

  ```python
def combiner(key, values):
    output(key, sum(values))
  ```

NOTE: For Word Count, this turns out to be exactly what the Reducer does!
Word Count with Combiner

Input  Map  Shuffle & Sort  Reduce  Output

the quick brown fox

the fox ate the mouse

how now brown cow

Map  Map  Map  Reduce  Reduce

the, 1
brown, 1
fox, 1

the, 1
fox, 1

the, 1

the, 1
fox, 1

the, 1

how, 1
now, 1
brown, 1

ate, 1

mouse, 1

quick, 1

cow, 1

brown, 2
fox, 2
how, 1
now, 1
the, 3

ate, 1
cow, 1
mouse, 1
quick, 1
Care in using Combiner

• Usually same as reducer
  – If operation is associative and commutative, e.g., sum

• How about average?
  – Rework the reduce function into something that is associative and commutative, e.g., use (sum, count)-pair, and then compute average = sum/count

• How about median?
The Complete WordCount Program

```java
public class WordCount {

    public static class Map extends MapReduceBase implements
            Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(LongWritable key, Text value,
                OutputCollector<Text, IntWritable> output,
                Reporter reporter) throws IOException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                output.collect(word, one);
            }
        }

        public static class Reduce extends MapReduceBase implements
            Reducer<Text, IntWritable, Text, IntWritable> {

            public void reduce(Text key, Iterator<IntWritable> values,
                OutputCollector<Text, IntWritable> output,
                Reporter reporter) throws IOException {
                int sum = 0;
                while (values.hasNext()) {
                    sum += values.next().get();
                }
                output.collect(key, new IntWritable(sum));
            }

        }

        public static void main(String[] args) throws Exception {
            JobConf conf = new JobConf(WordCount.class);
            conf.setJobName("wordcount");
            conf.setOutputKeyClass(Text.class);
            conf.setOutputValueClass(IntWritable.class);
            conf.setMapperClass(Map.class);
            conf.setCombinerClass(Reduce.class);
            conf.setReducerClass(Reduce.class);
            conf.setInputFormat(TextInputFormat.class);
            conf.setOutputFormat(TextOutputFormat.class);
            FileInputFormat.setInputPaths(conf, new Path(args[0]));
            FileOutputFormat.setOutputPath(conf, new Path(args[1]));
            JobClient.runJob(conf);
        }
    }
}
```

Map function
Reduce function
Invoking Map, Combiner & Reduce
Run this program as a MapReduce job
Example 2: Search

- **Input**: (lineNumber, line) records
- **Output**: lines matching a given pattern

- **Map**:
  ```java
  if(line matches pattern):
    output(line)
  ```

- **Reduce**: identity function
  - Alternative: no reducer (map-only job)
  - Which is preferred?
Example 3: Inverted Index

Source documents

- **hamlet.txt**
  - to be or not to be

- **12th.txt**
  - be not afraid of greatness

Inverted index

- afraid, (12th.txt)
- be, (12th.txt, hamlet.txt)
- greatness, (12th.txt)
- not, (12th.txt, hamlet.txt)
- of, (12th.txt)
- or, (hamlet.txt)
- to, (hamlet.txt)
Example 3: Inverted Index

- **Input:** (filename, text) records
- **Output:** list of files containing each word

- **Map:**
  ```python
  foreach word in text.split():
      output(word, filename)
  ```

- **Combine:**

- **Reduce:**
  ```python
  def reduce(word, filenames):
      output(word, sort(filenames))
  ```
Example 4: Most Popular Words

• **Input:** (filename, text) records
• **Output:** the 100 words occurring in most files

• Two-stage solution:
  – **Job 1:**
    • Create inverted index, giving (word, list(file)) records
  – **Job 2:**
    • Map each (word, list(file)) to (count, word)
    • Sort these records by count as in sort job
Note

• #mappers, #reducers, #physical nodes may not be equal
• For different problems, the Map and Reduce functions are different, but the workflow is the same
  – Read a lot of data
  – Map
    • Extract something you care about
  – (Sort and Combine) and Shuffle
  – Reduce
    • Aggregate, summarize, filter or transform
  – Write the result
Refinement: Partition Function

• Want to control how keys are partitioned
  – Inputs to map tasks are created by contiguous splits of input file
  – Reduce needs to ensure that records with the same intermediate key end up at the same worker
• System uses a default partition function:
  – hash(key) mod R
• Sometimes useful to override the hash function:
  – E.g., hash(hostname(URL)) mod R ensures URLs from a host end up in the same output file;
  – E.g., How about sorting?
Example 5: Sort

- **Input:** (key, value) records
- **Output:** same records, sorted by key

- **Map:** identity function. Why?
- **Reduce:** identity function. Why?

- **Trick:** Pick partitioning function $h$ such that $k_1 < k_2 \Rightarrow h(k_1) < h(k_2)$
Job Configuration Parameters

- 190+ parameters in Hadoop
- Set manually or defaults are used
Not all tasks fit the MapReduce model …

• Consider a data set consisting of n observations and k variables
  – e.g., k different stock symbols or indices (say k=10,000) and n observations representing stock price signals (up / down) measured at n different times.

• Problem – Find very high correlations (ideally with time lags to be able to make a profit) - e.g. if Google is up today, Facebook is likely to be up tomorrow.

• Need to compute k * (k-1) /2 correlations to solve this problem

• Cannot simply split the 10,000 stock symbols into 1,000 clusters, each containing 10 stock symbols, then use MapReduce
  – The vast majority of the correlations to compute will involve a stock symbol in one cluster, and another one in another cluster
  – These cross-clusters computations makes MapReduce useless in this case

• The same issue arises if you replace the word "correlation" by any other function, say f, computed on two variables, rather than one

NOTE: Make sure you pick the right tool!
High Level Languages on top of Hadoop

• MapReduce is great, as many algorithms can be expressed by a series of MR jobs

• But it’s low-level: must think about keys, values, partitioning, etc

• Can we capture common “job patterns”?
Pig

- Started at Yahoo! Research
- Now runs about 30% of Yahoo!’s jobs
- Features:
  - Expresses sequences of MapReduce jobs
  - Data model: nested “bags” of items
  - Provides relational (SQL) operators (JOIN, GROUP BY, etc)
  - Easy to plug in Java functions
  - Pig Pen dev. env. for Eclipse

Suppose you have user data in one file, website data in another, and you need to find the top 5 most visited pages by users aged 18 - 25.

Load Users

Load Pages

Filter by age

Join on name

Group on url

Count clicks

Order by clicks

Take top 5

Example from http://wiki.apache.org/pig-data/attachments/PigTalksPapers/attachments/ApacheConEurope09.ppt
Notice how naturally the components of the job translate into Pig Latin.

Load Users

Filter by age

Join on name

Group on url

Count clicks

Order by clicks

Take top 5

Load Pages

Users = load ‘users’ as (name, age);
Filtered = filter Users by age >= 18 and age <= 25;
Pages = load ‘pages’ as (user, url);
Joined = join Filtered by name, Pages by user;
Grouped = group Joined by url;
Summed = foreach Grouped generate group, count(Joined) as clicks;
Sorted = order Summed by clicks desc;
Top5 = limit Sorted 5;

store Top5 into ‘top5sites’;
Hive

• Developed at Facebook
• Used for majority of Facebook jobs
• “Relational database” built on Hadoop
  – Maintains list of table schemas
  – SQL-like query language (HQL)
  – Can call Hadoop Streaming scripts from HQL
  – Supports table partitioning, clustering, complex data types, some optimizations
Creating a Hive Table

CREATE TABLE page_views(viewTime INT, userid BIGINT, 
    page_url STRING, referrer_url STRING, 
    ip STRING COMMENT 'User IP address') 
COMMENT 'This is the page view table' 
PARTITIONED BY(dt STRING, country STRING) // break the table into separate files 
STORED AS SEQUENCEFILE;

Sample Query

• Find all page views coming from xyz.com on March 31st:

    SELECT page_views.*
    FROM page_views
    WHERE page_views.date >= '2008-03-01'
    AND page_views.date <= '2008-03-31'
    AND page_views.referrer_url like '%xyz.com';

• Hive only reads partition 2008-03-01,* instead of scanning entire table
Announcement

- Project teams
  - Submit list of members of your team ASAP
  - Mail to tankl@comp.nus.edu.sg
MapReduce Environment

• MapReduce environment takes care of:
  – Partitioning the input data
  – Scheduling the program’s execution across a set of machines
  – Performing the “group-by key” step in the Sort and Shuffle phase
  – Handling machine failures
  – Managing required inter-machine communication
MapReduce Execution Details

- **Single master** controls job execution on multiple slaves as well as user scheduling
- Mappers preferentially placed on same node or same rack as their input block
  - Push computation to data, minimize network use
- Mappers save outputs to *local disk* rather than pushing directly to reducers
  - Allows recovery if a reducer crashes
- Output of reducers are stored in HDFS
  - May be replicated
Coordinator: Master Node

• Master node takes care of coordination:
  – Task status: (idle, in-progress, completed)
  – Idle tasks get scheduled as workers become available

• When a map task completes, it sends the master the location and sizes of its (R) intermediate files, one for each reducer

• Master pushes this info to reducers

• Master pings workers periodically to detect failures
MapReduce – Single reduce task
MapReduce – Multiple reduce tasks
MapReduce – No reduce tasks
MapReduce: Shuffle and sort
Two additional notes

• Barrier between map and reduce phases
  – Reduce cannot start processing until all maps have completed
  – But we can begin copying intermediate data as soon as a mapper completes

• Keys arrive at each reducer in sorted order
  – No enforced ordering across reducers
How many Map and Reduce tasks?

• M map tasks, R reduce tasks

• Rule of a thumb:
  – Make M much larger than the number of nodes in the cluster
    • One DFS chunk per map is common
    • Improves dynamic load balancing and speeds up recovery from worker failures
  – Usually R is smaller than M
    • Because output is spread across R files
Fault Tolerance in MapReduce

1. If a task crashes:
   - Retry on another node
     - Okay for map
     - Okay for reduce
   - If the same task repeatedly fails, fail the job or ignore that input block (user-controlled)

➤ Note: For this and the other fault tolerance features to work, your map and reduce tasks must be side-effect-free
Fault Tolerance in MapReduce

2. If a *node* crashes:
   - Relaunch its current tasks on other nodes
   - Relaunch any maps the node previously ran
     • Why?
Fault Tolerance in MapReduce

3. If a task is running slowly (straggler):
   – Launch second copy of task on another node
   – Take the output of whichever copy finishes first, and kill the other one

• Critical for performance in large clusters: stragglers occur frequently due to failing hardware, bugs, misconfiguration, etc
Takeaways

• By providing a data-parallel programming model, MapReduce can control job execution in useful ways:
  – Automatic division of job into tasks
  – Automatic placement of computation near data
  – Automatic load balancing
  – Recovery from failures & stragglers

• User focuses on application, not on complexities of distributed computing
Conclusions

• MapReduce’s data-parallel programming model hides complexity of distribution and fault tolerance

• Principal philosophies:
  – Make it scale, so you can throw hardware at problems
  – Make it cheap, saving hardware, programmer and administration costs (but requiring fault tolerance)

• Hive and Pig further simplify programming

• MapReduce is not suitable for all problems, but when it works, it may save you a lot of time
Resources

• Hadoop: http://hadoop.apache.org/core/
• Hadoop docs: http://hadoop.apache.org/core/docs/current/
• Pig: http://hadoop.apache.org/pig
• Hive: http://hadoop.apache.org/hive
• Hadoop video tutorials from Cloudera: http://www.cloudera.com/hadoop-training
Acknowledgements

- This lecture is adapted from slides from multiple sources, including those from
  - RAD Lab, Berkeley
  - Stanford
  - Duke
  - Maryland
  - …