Lecture 6: MapReduce in Parallel Computing
MapReduce: Simplified Data Processing on Large Clusters

Motivation

Large-Scale Data Processing

- Want to use 1000s of CPUs
  - But don’t want hassle of managing things

MapReduce provides

- Automatic parallelization & distribution
- Fault tolerance
- I/O scheduling
- Monitoring & status updates

Jeffrey Dean, et al., MapReduce: Simplified Data Processing on Large Clusters in OSDI’04
Map/Reduce

- Map/Reduce
  - Programming model from Lisp
  - (and other functional languages)
- Many problems can be expressed this way
- Easy to distribute across nodes
- Nice retry/failure semantics
Map in Lisp (Scheme)

(map \( f \) \( list \) [\( list_2 \) \( list_3 \) …])

(map square `(1 2 3 4))
   • (1 4 9 16)

(reduce + `(1 4 9 16))
   • (+ 16 (+ 9 (+ 4 1) ) )
   • 30

(r \( 1_2 \) )))

Unary operator

Binary operator
Map/Reduce ala Google

map(key, val) is run on each item in set
  • emits new-key / new-val pairs

reduce(key, vals) is run for each unique key emitted by map()
  • emits final output
An Example: Count Words in docs

• Input consists of (url, contents) pairs

• map(key=url, val=contents):
  – For each word \( w \) in contents, emit \((w, \text{“1”})\)

• reduce(key=word, values=uniq_counts):
  – Sum all “1”s in values list
  – Emit result “(word, sum)”
Count, Illustrated

map(key=url, val=contents):
  For each word $w$ in contents, emit ($w$, “1”)

reduce(key=word, values=uniq_counts):
  Sum all “1”s in values list
  Emit result “(word, sum)”

see bob throw
see spot run

<table>
<thead>
<tr>
<th>word</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>see</td>
<td>1</td>
</tr>
<tr>
<td>bob</td>
<td>1</td>
</tr>
<tr>
<td>run</td>
<td>1</td>
</tr>
<tr>
<td>see</td>
<td>1</td>
</tr>
<tr>
<td>spot</td>
<td>1</td>
</tr>
<tr>
<td>throw</td>
<td>1</td>
</tr>
</tbody>
</table>

bob  | 1
run  | 1
see  | 2
spot | 1
throw| 1
Pseudo-code of WordCount

Map(String input_key, String input_value):
    //input_key: document name
    //input_value: document contents
    for each word w in input_values:
        EmitIntermediate(w, "1");

Reduce(String key, Iterator intermediate_values):
    //key: a word, same for input and output
    //intermediate_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
        result += ParseInt(v);
    Emit(AsString(result))
Java code of WordCount (for Mapper)

```java
public void map(Object key, Text value, Context context)
    throws IOException, InterruptedException {
    // Constructs a string tokenizer for the specified string. The tokenizer
    // uses the default delimiter set,
    StringTokenizer itr = new StringTokenizer(value.toString());

    while (itr.hasMoreTokens()) { //iterate all strings of the input text
        word.set(itr.nextToken());
        context.write(word, one); //emit the map output <word, 1>
    // in <text, IntWritable> format
    }
}
```
Java code of WordCount (for Reducer)

```java
public static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable result = new IntWritable();
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {  //iterate all values with the same key
            sum += val.get();               // calculate the sum
        }
        result.set(sum);
        context.write(key, result);    //emit the reduce output
        // in <text, IntWritable> format
    }
}
```
Grep

• Input consists of (url+offset, single line)
• map(key=url+offset, val=line):
  – If contents matches regexp, emit (line, “1”)

• reduce(key=line, values=uniq_counts):
  – Don’t do anything; just emit line
Reverse Web-Link Graph

Map
- For each URL linking to target (the link) in a source page …
- Output <target, source> pairs

Reduce
- Concatenate list of all source URLs
- Outputs: <target, list (source)> pairs
Implementation Overview

Typical cluster:

- 100s/1000s of x86 machines
- 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

How is this distributed?

1. Partition input key/value pairs into chunks, run map() tasks in parallel
2. After all map()s are complete, consolidate all emitted values for each unique emitted key
3. Now partition space of output map keys, and run reduce() in parallel

If map() or reduce() fails, reexecute!
Keys and Values

- The programmer in MapReduce has to specify two functions, the map function and the reduce function that implement the Mapper and the Reducer in a MapReduce program.

- In MapReduce data elements are always structured as key-value (i.e., (K, V)) pairs.

- The map and reduce functions receive and emit (K, V) pairs.
Partitions

- In MapReduce, intermediate output values are not usually reduced together.

- All values with the same key are presented to a single Reducer together.

- More specifically, a different subset of intermediate key space is assigned to each Reducer.

- These subsets are known as partitions using a partitioning function such as hash(key) mod R.

Different colors represent different keys (potentially) from different Mappers.

Partitions are the input to Reducers.
Execution
Parallel Execution
In MapReduce, chunks are processed in isolation by tasks called \textit{Mappers}

The outputs from the mappers are denoted as intermediate outputs (IOs) and are brought into a second set of tasks called \textit{Reducers}

The process of bringing together IOs into a set of Reducers is known as \textit{shuffling process}

The Reducers produce the final outputs (FOs)

Overall, MapReduce breaks the data flow into two phases, map phase and reduce phase
Task Scheduling in MapReduce

- MapReduce adopts a *master-slave architecture*

- The master node in MapReduce is referred to as *Job Tracker* (JT)

- Each slave node in MapReduce is referred to as *Task Tracker* (TT)

- MapReduce adopts a *pull scheduling* strategy rather than a *push one*
  - I.e., JT does not push map and reduce tasks to TTs but rather TTs pull them by making pertaining requests
Task Granularity & Pipelining

Fine granularity tasks: map tasks >> machines

- Minimizes time for fault recovery
- Can pipeline shuffling with map execution
- Better dynamic load balancing

Often use 200,000 map & 5000 reduce tasks

Running on 2000 machines
Fault Tolerance / Workers

Handled via re-execution

- Detect failure via periodic heartbeats
- Re-execute completed and in-progress \textit{map} tasks
  - The map outputs are on the local disks.
- Re-execute in progress \textit{reduce} tasks
- Task completion committed through master

Robust: lost 1600/1800 machines once $\rightarrow$ finished ok
What Makes MapReduce Unique?

MapReduce is characterized by:

1. Its simplified programming model which allows the user to quickly write and test distributed systems

2. Its efficient and automatic distribution of data and workload across machines

3. Its flat scalability curve. Specifically, after a Mapreduce program is written and functioning on 10 nodes, very little, if any, work is required for making that same program run on 1000 nodes
## Comparison With Traditional Models

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Shared Memory</th>
<th>Message Passing</th>
<th>MapReduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication</td>
<td>Implicit (via loads/stores)</td>
<td>Explicit Messages</td>
<td>Limited and Implicit</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Explicit</td>
<td>Implicit (via messages)</td>
<td>Immutable (K, V) Pairs</td>
</tr>
<tr>
<td>Hardware Support</td>
<td>Typically Required</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Development Effort</td>
<td>Lower</td>
<td>Higher</td>
<td>Lowest</td>
</tr>
<tr>
<td>Tuning Effort</td>
<td>Higher</td>
<td>Lower</td>
<td>Lowest</td>
</tr>
</tbody>
</table>