Recap

• Last Thursday’s Lecture
  – Clouds vs. Clusters
    » At least 3 differences
  – A Cloudy History of Time
    » Clouds are the latest in a long generation of distributed systems

• Today’s Lecture
  – Cloud Programming: MapReduce (the heart of Hadoop)
  – Grids
Highly-Parallel Data-Processing

- Originally designed by Google (OSDI 2004 paper)
- Open-source version called Hadoop, by Yahoo!
  - Spun off into startup HortonWorks
- Hadoop written in Java. Your implementation could be in Java, or any executable
- Google (MapReduce)
  - Indexing: a chain of 24 MapReduce jobs
  - ~200K jobs processing 50PB/month (in 2006)
- Yahoo! (Hadoop + Pig)
  - WebMap: a chain of 100 MapReduce jobs
  - 280 TB of data, 2500 nodes, 73 hours
- Annual Hadoop Summit: 2008 had 300 attendees, now close to 1000 attendees
What is MapReduce?

- Terms are borrowed from Functional Languages (e.g., Lisp)
  
  Sum of squares:
  
  - \((\text{map square } '(1 2 3 4))\)
    - Output: \((1 4 9 16)\)
    [processes each record sequentially and independently]
  
  - \((\text{reduce } + ' (1 4 9 16))\)
    - \((+ 16 (+ 9 (+ 4 1)))\)
    - Output: 30
    [processes set of all records in groups]

- Let’s consider a sample application: Wordcount
  - You are given a huge dataset (e.g., collection of webpages) and asked to list the count for each word appearing in the dataset
• Process individual records to generate intermediate key/value pairs.

Input <filename, file text>
• **Parallelly** Process individual records to generate intermediate key/value pairs.

Input `<filename, file text>`

---

**Map Task 1**
- Welcome Everyone
- Hello Everyone
- Welcome
- Everyone
- Hello
- Everyone

**Map Task 2**
**Map**

- **Parallely** Process a large number of individual records to generate intermediate key/value pairs.

Input `<filename, file text>`

MAP TASKS
Reduce

• Processes and merges all intermediate values associated **per key** (that’s the group)

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Welcome</td>
<td>1</td>
</tr>
<tr>
<td>Everyone</td>
<td>1</td>
</tr>
<tr>
<td>Hello</td>
<td>1</td>
</tr>
<tr>
<td>Everyone</td>
<td>1</td>
</tr>
<tr>
<td>Everyone</td>
<td>2</td>
</tr>
<tr>
<td>Hello</td>
<td>1</td>
</tr>
<tr>
<td>Welcome</td>
<td>1</td>
</tr>
</tbody>
</table>
- **Parallelly** Processes and merges all intermediate values *by partitioning keys*.

<table>
<thead>
<tr>
<th>String</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Welcome</td>
<td>1</td>
</tr>
<tr>
<td>Everyone</td>
<td>1</td>
</tr>
<tr>
<td>Hello</td>
<td>1</td>
</tr>
<tr>
<td>Everyone</td>
<td>1</td>
</tr>
<tr>
<td>Everyone</td>
<td>2</td>
</tr>
<tr>
<td>Hello</td>
<td>1</td>
</tr>
<tr>
<td>Welcome</td>
<td>1</td>
</tr>
</tbody>
</table>

**Reduce Task 1**

**Reduce Task 2**
public static class MapClass 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 itr = new StringTokenizer(line);
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            output.collect(word, one);
        }
    }
}

public static class ReduceClass 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));
    }
}
// Tells Hadoop how to run your Map-Reduce job
public void run (String inputPath, String outputPath)
    throws Exception {
    // The job. WordCount contains MapClass and Reduce.
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("mywordcount");
    // The keys are words
    (strings) conf.setOutputKeyClass(Text.class);
    // The values are counts (ints)
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(MapClass.class);
    conf.setReducerClass(ReduceClass.class);
    FileInputFormat.addInputPath(
        conf, newPath(inputPath));
    FileOutputFormat.setOutputPath(
        conf, new Path(outputPath));
    JobClient.runJob(conf);
}
Some Other Applications of MapReduce

Distributed Grep:
- Input: large set of files
- Output: lines that match pattern

- Map – *Emits a line if it matches the supplied pattern*
- Reduce – *Simply copies the intermediate data (key-value pairs) to output*
- Partitioner – *Default (hash-based)*
Some Other Applications of MapReduce (2)

Reverse Web-Link Graph

- **Input**: Web graph: tuples (a, b) where (page a → page b)
- **Output**: For each page, list of pages that link to it

- **Map** – process web log and for each <source, target> it outputs <target, source>
- **Reduce** - emits <target, list(source)>
- **Partitioner** – Default (hash-based)
### Count of URL access frequency

- **Input:** Log of accessed URLs from proxy server
- **Output:** For each URL, % of total accesses for that URL (all use default partitioners)

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Map</strong></td>
<td>Process web log and outputs <code>&lt;URL, 1&gt;</code></td>
</tr>
<tr>
<td><strong>Multiple Reducers</strong></td>
<td>Emits <code>&lt;URL, URL_count&gt;</code> (So far, like Wordcount. But still need %)</td>
</tr>
<tr>
<td><strong>Chain</strong></td>
<td>another MapReduce job after above one</td>
</tr>
<tr>
<td><strong>Map</strong></td>
<td>Processes <code>&lt;URL, URL_count&gt;</code> and outputs <code>&lt;1, (&lt;URL, URL_count&gt;)&gt;</code></td>
</tr>
<tr>
<td><strong>1 Reducer task</strong></td>
<td>Sums up <code>URL_count</code>’s to calculate <code>overall_count</code>. Outputs <code>&lt;URL, URL_count/overall_count&gt;</code> pairs</td>
</tr>
</tbody>
</table>
Some Other Applications of MapReduce (4)

Sort

- Input: Series of (key, value) pairs
- Need Output: Sorted <key>s

- Map – <key, value> -> <key, value> (identity)
- Reducer – <key, value> -> <key, value> (identity)
- Partitioner – identity. For load-balance, assign equal number of keys to each reduce
  - Take data distribution into account to split key ranges across reduce tasks

• Why does this work?
  - Map task’s output is (always) auto-sorted (e.g., quicksort)
  - Reduce task’s input is (always) auto-sorted (e.g., mergesort)
Programming MapReduce

- **Externally: For user**
  1. Write a Map program (short), write a Reduce program (short)
  2. Specify number of tasks and submit job in configuration; wait for result
  3. Need to know practically nothing about parallel/distributed programming!

- **Internally: For the cloud (and for us distributed systems researchers)**
  1. Parallelize Map
  2. Transfer data from Map to Reduce
  3. Parallelize Reduce
  4. Implement Storage for Map input, Map output, Reduce input, and Reduce output
Inside MapReduce

For the cloud (and for us distributed systems researchers)

1. Parallelize Map: easy! Shard the data equally into requested map tasks.
2. Transfer data from Map to Reduce:
   » All Map output tuples with same key assigned to same Reduce task
   » use partitioning function: example is to hash the key of the tuple, modulo number of reduce jobs, or identity function for sort
4. Implement Storage for Map input, Map output, Reduce input, and Reduce output
   » Map input: from distributed file system
   » Intermediate data - Map output: to local disk (at Map node); uses local file system
   » Intermediate data - Reduce input: from (multiple) remote disks; uses local file systems
   » Reduce output: to distributed file system

local file system = Linux FS, etc.
distributed file system = GFS (Google File System), HDFS (Hadoop Distributed File System)
Internal Workings of MapReduce - Example

From the original MapReduce paper (OSDI 2004)
Barriers and Speculation

- Hadoop’s internal scheduler runs at master
- A Reduce task cannot start until all Map tasks done, and all its (Reduce’s) data has been fetched
  - Barrier between Map phase and Reduce phase
  - As a result, the slowest Map slows down the entire Map phase (and thus the entire job)
  - The slowest Reduce slows down the entire Reduce phase (and thus the entire job)

- Stragglers (slow nodes)
  - The slowest machine slows the entire job down
  - Due to Bad Disk, Network Bandwidth, CPU, or Memory
  - Solution: Speculative Execution.
    - Keep track of “progress” of each task (% done)
    - Replicate straggler task on other available machines – fastest machine wins (and other task replicas are then killed)
• **Locality**
  
  - Needed to avoid network traffic bottlenecks, since cloud has hierarchical topology (e.g., racks)
  - GFS stores 3 replicas of each of 64MB chunks
    » Maybe on different racks, e.g., 2 on a rack, 1 on a different rack
  - Scheduler attempts to schedule a map task on a machine that contains a replica of corresponding input data: why?
    » Failing this, it tries scheduling map on the same rack as data
    » Failing this, schedule map task anywhere

• **Failures**
  
  - Master tracks *progress* of each task
  - Similar to speculative execution, reschedules task with stopped progress or on failed machine
  - Highly simplified explanation here – failure-handling is more sophisticated (next lecture!)
Testbed: 1800 servers each with 4GB RAM, dual 2GHz Xeon, dual 169 GB IDE disk, 100 Gbps, Gigabit ethernet per machine

Grep

Locality optimization helps:
- 1800 machines read 1 TB at peak \(~31\) GB/s
- W/out this, rack switches would limit to 10 GB/s

Startup overhead is significant for short jobs

Workload: \(10^{10}\) 100-byte records to extract records matching a rare pattern (92K matching records)
The first datacenters!

Timesharing Companies & Data Processing Industry

1940
1950
1960
1970
1980
1990
2000
2010

Clusters
Grids
PCs (not distributed!)
Clouds and datacenters
Peer to peer systems

“A Cloudy History of Time” © IG 2010
Clouds are data-intensive while … Grids are/were computation-intensive

What is a Grid?
Example: Rapid Atmospheric Modeling System, ColoState U

• Hurricane Georges, 17 days in Sept 1998
  – “RAMS modeled the mesoscale convective complex that dropped so much rain, in good agreement with recorded data”
  – Used 5 km spacing instead of the usual 10 km
  – Ran on 256+ processors

• Computation-intensive application rather than data-intensive

• Can one run such a program without access to a supercomputer?
An Application Coded by a Physicist

Jobs 1 and 2 can be concurrent

Job 0
Output files of Job 0
Input to Job 2

Job 1

Job 2
Output of Job 2
Input to Job 3

Job 3
An Application Coded by a Physicist

Output files of Job 0
Input to Job 2

Job 2
Output of Job 2
Input to Job 3

Several GBs

May take several hours-days
4 stages of a job
Init
Stage in
Execute
Stage out
Publish

Computation Intensive,
so Massively Parallel
Condor Protocol

Globus Protocol

Wisconsin

MIT

NCSA

Job 0

Job 1

Job 2

Job 3

Lecture 3-30
Internal structure of different sites invisible to Globus

Globus Protocol

External Allocation & Scheduling Stage in & Stage out of Files
Condor Protocol

Internal Allocation & Scheduling
Monitoring
Distribution and Publishing of Files
The Grid Recently

Some are 40Gbps links. (The TeraGrid links)

“A parallel Internet”
Question to Ponder

- Cloud computing vs. Grid computing: what are the differences?
• MP1, HW1 out today
  – MP1 due 9/15 (Sun midnight). Demos soon after (likely Monday 9/16)
  – HW1 due 9/19 (hand-in at beginning of class)

• Effort
  – For HW: Individual effort only. You are allowed to discuss the problem and concepts (e.g., in study groups), but you cannot discuss the solution.
  – For MP: Groups of 2 students (pair up with someone taking class for same # credits)
    » If you don’t have an MP partner, hang around after class today (or use Piazza)
    » Please report groups to us by this Thursday 9/15. Subject line: “425 MP group” to cs425-ece428@mx.uillinois.edu
  – Please read instructions carefully!
  – Start NOW
MP1: Logging + Testing

- Distributed Systems hard to debug (you’ll know soon!)
- Creating log files at each machine to tabulate important messages/errors/status is critical to debugging
- MP1: Write a distributed program that lets you grep (+ regexp’s) all the log files across a set of machines (from any of those machines)
- Each line is a key-value pair
- Read the doc – it is very detailed
- How do you know your program works?
  - Write unit tests
  - E.g., Generate non-identical logs at each machine, then run grep from one of them and automatically verify that you receive the answer you expect
  - Writing tests can be hard work, but it is industry standard
  - We encourage (but don’t require) that you write tests for MP2 onwards
Readings

• For next lecture
  – Failure Detection
  – Readings: Section 15.1, parts of Section 2.4.2