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
Programming Cloud Applications - New Parallel Programming Paradigms: **MapReduce**

- Highly-Parallel Data-Processing
- Originally designed by Google (OSDI 2004 paper)
- Open-source version called **Hadoop**, by Yahoo!
  - 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 Language (e.g., Lisp)

Sum of squares:

- \((\text{map} \ \text{square} \ ‘(1 \ 2 \ 3 \ 4))\)
  - 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>

Welcome Everyone
Hello Everyone

Key
Welcome
Everyone
Hello
Everyone

Value
1
1
1
1
Parallelly Process individual records to generate intermediate key/value pairs.

- **MAP TASK 1**
  - Welcome 1
  - Everyone 1
  - Hello 1
  - Everyone 1

- **MAP TASK 2**
  - Input <filename, file text>
Map

- **Parallelly** 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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<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

- **Parallelly** Processes and merges all intermediate values by partitioning keys

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Welcome</td>
<td>1</td>
<td>Everyone</td>
<td>2</td>
<td>Hello</td>
<td>1</td>
</tr>
<tr>
<td>Everyone</td>
<td>1</td>
<td>Everyone</td>
<td>2</td>
<td>Welcome</td>
<td>1</td>
</tr>
<tr>
<td>Hello</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Everyone</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Lecture 3-9
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);
        }
    }
}  // Source: http://developer.yahoo.com/hadoop/tutorial/module4.html#wordcount
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));
    }
}
Hadoop Code - Driver

// 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 - Copies the intermediate data to output
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 outputs <target, source>
- Reduce - emits <target, list(source)>
Some Other Applications of MapReduce (3)

Count of URL access frequency

- Input: Log of accessed URLs from proxy server
- Output: For each URL, % of total accesses for that URL

- Map – Process web log and outputs <URL, 1>
- Multiple Reducers - Emits <URL, URL_count>
  (So far, like Wordcount. But still need %)
- Chain another MapReduce job after above one
- Map – Processes <URL, URL_count> and outputs <1, (<URL, URL_count>)>
- 1 Reducer – Sums up URL_count’s to calculate overall_count.
  Emits <URL, URL_count/overall_count>
Some Other Applications of MapReduce (4)

Map task’s output is sorted (e.g., quicksort)
Reduce task’s input is sorted (e.g., mergesort)

Sort

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

- Map – <key, value> -> <value, _> (identity)
- Reducer – <key, value> -> <key, value> (identity)
- Partitioning function – partition keys across reducers based on ranges
  » Take data distribution into account to balance reducer tasks
**Programming MapReduce**

- **Externally: For user**
  1. Write a Map program (short), write a Reduce program (short)
  2. Decide number of tasks and submit job; wait for result
  3. Need to know 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
     » Map output: to local disk (at Map node); uses local file system
     » 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

Figure 1: Execution overview

From the original MapReduce paper (OSDI 2004)
Etcetera

• Failures
  – Master tracks **progress** of each task
  – reschedules task with stopped progress or on failed machine
  – Highly simplified explanation here – failure-handling is more sophisticated (next lecture!)

• Slow tasks
  – The slowest machine slows the entire job down
  – Hadoop **Speculative Execution**: Spawn multiple copies of tasks that have a slow progress. When one finishes, stop other copies.

• What about bottlenecks within the datacenter?
  – CPUs? Disks? Switches?
Testbed: 1800 servers each with 4GB RAM, dual 2GHz Xeon, dual 169 GB IDE disk, 100 Gbps, Gigabit ethernet per machine

Grep

Workload: $10^{10}$ 100-byte records to extract records matching a rare pattern (92K matching records)

From the original MapReduce paper (OSDI 2004)
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
Clouds are data-intensive
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

Jobs 1 and 2 can be concurrent

Job 2

Output of Job 2
Input to Job 3

Job 3

Output of Job 2
Input to Job 3

Lecture 3-26
An Application Coded by a Physicist

Several GBs

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

Computation Intensive, so Massively Parallel

Output files of Job 0
Input to Job 2

Job 2
Output of Job 2
Input to Job 3

Lecture 3-27
Lecture 3

Job 0

Job 1

Job 2

Job 3

Wisconsin

MIT

Condor Protocol

Globus Protocol

NCSA

Lecture 3-29
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 9/16 (Sun midnight)
  – HW1 due 9/20 (in class)
  – For HW: Individual. 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 a partner, hang around after class today
    » Please report groups to us by this Thursday 9/16. Subject line: “425 MP group”
  – 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)
- 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
• For next lecture
  – Failure Detection
  – Readings: Section 15.1, parts of Section 2.4.2