ECE459: Programming for Performance Winter 2011

Lecture 21 — March 24, 2011

Patrick Lam *version 1*

MapReduce/Hadoop: Another Distributed Computing Paradigm

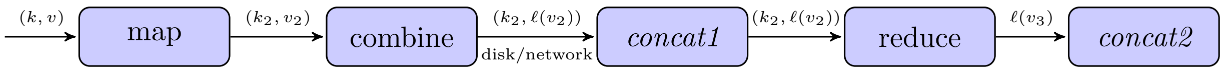
The key idea behind MapReduce is to enable parallelization on huge datasets by distributing the data over a huge collection of commodity PCs. The input is a (large) set of (key, value) pairs, and the output is also a set of (key, value) pairs, with both keys and values possibly of different types.

Some of the following information is from the Hadoop MapReduce tutorial:

`http://hadoop.apache.org/mapreduce/docs/current/mapred_tutorial.html`

The two key boxes that you need to implement are the `map` box and the `reduce` box. There is also an optional `combine` box between `map` and `reduce`, which transforms the `map` output in-place.

You can use various languages to implement the mappers and reducers; there are easy-to-find Java and Python examples. We'll present partial Java code for the word count example; the tutorial includes full code.



Map. The massively parallel step is taking the input (k, v) pairs and producing any number of new (k_2, v_2) pairs from each input. Each pair is mapped independently, so it's possible to run 10 billion input pairs on a million computers quite quickly.

The tutorial describes the canonical MapReduce example, a word counting application; the input to the map is a list of lines. This mapper ignores the key and uses the value as the data. Its output maps words (as keys) to their counts (as values) in each line.

For instance, taking the above paragraph as an example, let's say that the input contains a sentence per line. We will also consider just the base form of words (i.e. `words` and `words` are the same for our purposes). Our code emits each word with the count 1; the reduce phase will combine pairs.

- `the`, 1; `tutorial`, 1; `describe`, 1; `the`, 1; `canonical`, 1; `MapReduce`, 1; `example`, 1; `a`, 1; `word`, 1; `count`, 1; `application`, 1; `the`, 1; `input`, 1; `to`, 1; `the`, 1; `map`, 1; `is`, 1; `a`, 1; `list`, 1; `of`, 1; `line`, 1
- `this`, 1; `mapper`, 1; `ignore`, 1; `the`, 1; `key`, 1; `and`, 1; `use`, 1; `the`, 1; `value`, 1; `as`, 1; `the`, 1; `data`, 1
- `its`, 1; `output`, 1; `word`, 1; `as`, 1; `key`, 1; `to`, 1; `their`, 1; `count`, 1; `as`, 1; `value`, 1; `in`, 1; `each`, 1; `line`, 1.

Here is the corresponding code from the tutorial; this code does not take roots of words, but that would be easy to add.

```
public void map(LongWritable key, Text value, Context context)
                      throws IOException, InterruptedException {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);
    while (tokenizer.hasMoreTokens()) {
      word.set(tokenizer.nextToken());
      context.write(word, one);
    }
}
```

Combine. The combine phase is optional, but can increase performance. It is like the reduce phase (and implementations of reduce can often be used to combine), but it gets run locally on the output of map, while the output is still in-memory.

Our combine phase combines multiple instances of the same word on the same line. We could get the following output:

- `the`, 4; `tutorial`, 1; `describe`, 1; `canonical`, 1; `MapReduce`, 1; `example`, 1; `a`, 1; `word`, 1; `count`, 1; `application`, 1; `input`, 1; `to`, 1; `map`, 1; `is`, 1; `list`, 1; `of`, 1; `line`, 1
- `this`, 1; `mapper`, 1; `ignore`, 1; `the`, 3; `key`, 1; `and`, 1; `use`, 1; `value`, 1; `as`, 1; `data`, 1
- `its`, 1; `output`, 1; `word`, 1; `as`, 2; `key`, 1; `to`, 1; `their`, 1; `count`, 1; `value`, 1; `in`, 1; `each`, 1; `line`, 1.

Actually, the partitioning would depend on how the input gets partitioned among the nodes.

Reduce. The reduce phase is less parallelizable than the map (and combine) phases. First, MapReduce groups together (*concat1*) all identical keys k_2 from all outputs of `map` (or `combine`), creating a list $\ell_{k_2}(v_2)$ of the values associated with each k_2, and distributes these keys and values to appropriate nodes in the cluster. As we've mentioned before, reducing communication is important, and good MapReduce implementations will attempt to keep data on the same node. It then applies `reduce` to each of these $(k_2, \ell_{k_2}(v))$ pairs, obtaining values of a third type v_3 from the `reduce` calls.

The MapReduce framework then concatenates (*concat2*) the v_3 values it obtains from all of the `reduce` calls.

In the word count example, $\ell_{k_2}(v_2)$ is a list of counts. The `reduce` phase totals the different word counts for each word from the different lines. Each v_3 is a pair $\langle \text{word}, \text{count} \rangle$:

- `the`, 7; `tutorial`, 1; `describe`, 1; `canonical`, 1; `MapReduce`, 1; `example`, 1; `a`, 1; `word`, 2; `count`, 2; `application`, 1; `input`, 1; `to`, 2; `map`, 1; `is`, 1; `list`, 1; `of`, 1; `line`, 2; `this`, 1; `mapper`, 1; `ignore`, 1; `key`, 2; `and`, 1; `use`, 1; `value`, 2; `as`, 3; `data`, 1; `its`, 1; `output`, 1; `their`, 1; `in`, 1; `each`, 1.

Here is some code that carries out reduction for the word count example:

```
public void reduce(Text key, Iterable<IntWritable> values,
                   Context context) throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    context.write(key, new IntWritable(sum));
}
```

A different example. MapReduce has many applications beyond counting words. I looked on the Internet and found an array-summing example[1]. The `map` function takes an array segment and emits the sum, in the form of a ⟨`1, sum`⟩ pair. The `reduce` function then combines pairs (which all have key `1`) to get the total sum of the array.

More applications. The canonical MapReduce paper (OSDI 2004, Dean and Ghemawat) provides the following MapReduce examples: distributed grep; URL access frequency (log analysis is a common use); reverse web-link graph; term-vector per host; inverted index; distributed sort.

Hive. Hive builds on top of Hadoop and makes it easy for users to execute ad-hoc Hadoop queries using an SQL-like query language. Many of the Hadoop examples for Amazon's Elastic MapReduce use Hive. You get answers back in minutes. Important features include filtering (`WHERE` clauses) and (equi-)joins between tables. You can also explicitly write and use custom mappers and reducers in your Hive queries.

MapReduce Implementation Details. There are multiple implementations of MapReduce out there. Hadoop is perhaps the dominant one; it is maintained by the Apache Foundation. Amazon deploys Hadoop in their Elastic Compute Cloud. You can also run MapReduce on GPUs.

Hadoop on a Cluster. The Hadoop implementation requires a number of moving parts; they can run on one machine or on a cluster. In general, there is a master node, which contains a job tracker (where you submit jobs), and a number of task trackers, which distribute parts of the job to compute nodes. Task trackers also run on the compute nodes.

HDFS. The Hadoop Distributed File System[2] is a key part of actually using Hadoop; it (or some other filesystem) stores intermediate results. The goals of HDFS include dealing with hardware failure; permitting streaming access (like GPUs); handling large data sets; and being portable between different hardware and software platforms. HDFS also provides a write-once-read-many access model (e.g. web crawlers), and assumes that computation is going to migrate across nodes, rather than data.

HDFS runs as a distributed filesystem implemented in Java. It supports a hierarchical namespace for files. The design includes a `NameNode`, which contains metadata and manages the `DataNodes`. The `DataNodes` just store data. They can fail, and the `NameNode` will deal with it. The `NameNode`, however, cannot fail.

[1] `http://pages.cs.wisc.edu/~gibson/mapReduceTutorial.html`

[2] `http://hadoop.apache.org/hdfs/docs/current/hdfs_design.html`