Lecture 27 - MapReduce

ECE 459: Programming for Performance

Jon Eyolfson

University of Waterloo

March 19/23, 2012

Introduction

- Framework introduced by Google for large problems
- Consists of two functional operations: map and reduce

```
>>> map(lambda x: x*x, [1, 2, 3, 4, 5])
[1, 4, 9, 16, 25]
```

- **map** applies a function to an iterable data-set

```
>>> reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])
15
```

- **reduce** applies a function to an iterable data-set cumulatively
- ((((1+2)+3)+4)+5) in this example

Intuition

- In functional languages, the functions are “pure” (no side-effects)
- Since they are pure, and therefore independent, it’s safe to parallelize them
- **Note:** funtional languages, like Haskell, have their own parallel frameworks, which allows easy parallelization

- Many problems can be represented as a map operation followed by reduce (for example, Assignment 1)

Hadoop

- Apache Hadoop is a framework which implements MapReduce
- The most widely used open source one, used by Amazon's EC2 (elastic compute cloud)
- Allows work to be distrubed across many different nodes (or re-tried if a node goes down)
- Includes HDFS (Hadoop distributed file system), which distrributes the data across nodes and provides failure handling (you can also use Amazon's S3 storage service)

Map

- Input file is split up into multiple pieces
- The pieces are then processed as (key, value) pairs
- The **Mapper** function uses these (key, value) pairs and outputs another set of (key, value) pairs

Reduce

- Collects the input files from the previous map (which may be on different nodes, needing copying)
- The files are then merge sorted (so that the key-value pairs for a given key are contiguous)
- The file is read sequentially and the values split up into lists for the same key
- This data consisting of keys and lists of values are passed to the reduce method (done in parallel as well) then concatenated

Combine

- This step can be run right after the map, and before reduce
- It takes advantage of the fact that elements produced by the map operation are still available in memory
- Every so many elements, you can use your combine operation to take (key, value) outputs of the map and create new (key, value) inputs of the same types

WordCount Example

- Consider we just want to count the number of occurrences of words in some files
- Take for example the following files:

```
Hello World Bye World
```

```
Hello Hadoop Goodbye Hadoop
```

We want the following output:

```
(Bye, 1)
(Goodbye, 1)
(Hadoop, 2)
(Hello, 2)
(World, 2)
```

WordCount Example Operations

Mapper

- Split the input file into strings, representing words
- For each word, output the following (key, value) pair:
  (word, 1)

Reduce

- Sum all of the values of the word (key) and output:
  (word, sum)

Combine

- We could do the reduce step for the in-memory values while doing the map operation

Note: here, the output of the map and input/output of the reduce are the same, but they don't have to be

WordCount Example Operation (1)

```
Hello World Bye World
```

After map:

```
(Hello, 1)
(World, 1)
(Bye, 1)
(World, 1)
```

After combine:

```
(Hello, 1)
(Bye, 1)
(World, 2)
```

WordCount Example Operation (2)

```
Hello Hadoop Goodbye Hadoop
```

After map:

```
(Hello, 1)
(Hadoop, 1)
(Goodbye, 1)
(Hadoop, 1)
```

After combine:

```
(Hello, 1)
(Goodbye, 1)
(Hadoop, 2)
```

WordCount Example Operation (3)

After concatenation, sorting and creating lists of values

```
(Bye, [1])
(Goodbye, [1])
(Hadoop, [2])
(Hello, [1, 1])
(World, [2])
```

After the reduce, which is what we want:

```
(Bye, 1)
(Goodbye, 1)
(Hadoop, 2)
(Hello, 2)
(World, 2)
```

WordCount Example C++ Code (1)

- There's also APIs for Java/Python, etc.

```
#include "hadoop/Pipes.hh"
#include "hadoop/TemplateFactory.hh"
#include "hadoop/StringUtils.hh"

class WordCountMap: public HadoopPipes::Mapper {
public:
  WordCountMap(HadoopPipes::TaskContext& context){}
  void map(HadoopPipes::MapContext& context) {
    std::vector<std::string> words =
      HadoopUtils::splitString(context.getInputValue()," ");
    for(unsigned int i=0; i < words.size(); ++i) {
      context.emit(words[i], "1");
    }
  }
};
```

WordCount Example C++ Code (2)

```
class WordCountReduce: public HadoopPipes::Reducer {
public:
  WordCountReduce(HadoopPipes::TaskContext& context){}
  void reduce(HadoopPipes::ReduceContext& context) {
    int sum = 0;
    while (context.nextValue()) {
      sum += HadoopUtils::toInt(context.getInputValue());
    }
    context.emit(context.getInputKey(),
                 HadoopUtils::toString(sum));
  }
};

int main(int argc, char *argv[]) {
  return HadoopPipes::runTask(
    HadoopPipes::TemplateFactory<WordCountMap,
                                 WordCountReduce>());
}
```

Other Examples

- Distributed Grep
- Count of URL Access Frequency
- Reverse Web-Link Graph
- Term-Vector per Host
- Inverted Index
  - **Map:** parses each document, and emits a sequence of (word, document ID) pairs
  - **Reduce:** accepts all pairs for a given word, sorts the corresponding document IDs and emits a (word, list(document ID)) pair
  - **Output:** all of the output pairs from reducing forms a simple inverted index

Other Notes

- **Hive** builds on top of Hadoop, allowing you to use an SQL-like language to query outputs on HDFS, or provide custom mappers/reducers to get more information

- The cloud framework is a great way to start a new project, since you can add or remove nodes easily as your problem changes size (Hadoop or MPI are good examples)

References

`http://wiki.apache.org/hadoop/`

`http://code.google.com/edu/parallel/mapreduce-tutorial.html`

Summary

- The MapReduce is an excellent framework for dealing with massive data-sets
- Hadoop is the common implementation you can use (even use on most cloud computing services)
- You just need 2 functions (optionally 3): mapper, reducer and combiner
- Just remember the output of the mapper/combiner must be the input to the reducer