MapReduce: Simplified Data Processing on Large Clusters

Seminar "Algorithms for Big Data" WS 2014/15

Docent: Dr. Ullrich Köthe

Ruprecht-Karls-Universität Heidelberg
Heidelberg Collaboratory for Image Processing
List of Contents

- What is MapReduce?
- General Process Execution on Computers
- From Sequential to Parallel Processing on Computers
- Problems of Parallel Processing
- Parallel Processing using a Network of Computers
- Need for Methods to Simplify Parallel Computing
- Using MapReduce Programing Model and Infrastructure to Simplify Parallel Computing
- The MapReduce Programming Model
- MapReduce Example
List of Contents

- Sample Code for Word Count on MapReduce
- The MapReduce Reference Implementation
- Application Examples for MapReduce
- MapReduce Implementations
- Conclusions
- Additional
- References
What is MapReduce?

- **MapReduce** is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster.

- A MapReduce program is composed of a **Map()** procedure that performs filtering and sorting and a **Reduce()** procedure that performs a summary operation.
What is MapReduce?

Input data → Map() → Map() → Map() → Reduce() → Reduce() → Output data

Split \([k1, v1]\) by \(k1\) Merge \([k1, [v1, v2, v3...]]\)
General Process Execution on Computers

- On a common computer **general arithmetic and logical calculations** are performed by components referred to as **central processing units** (CPUs)
- Commodity personal computers (PCs) usually have one CPU component at the moment whereas special computers can handle multiple of these
From Sequential to Parallel Processing on Computers

- The first computers could run commands processed by the CPU one after another only (sequential execution)
- As a result maximal only one program per CPU could be run at the same time
From Sequential to Parallel Processing on Computers

- Several ways of *parallel handling programs* where introduced for computers gradually:
  - **Time-Sharing** (focus on simultaneous users)
  - **Multitasking** (focus on simultaneous processes)
    - **Multithreading** (focus on simultaneous sub processes)
  - Multicore processors
  - Multiprocessor systems
  - Networks of Computers
Problems of Parallel Processing

– To run parts of tasks that needs to interact with each other or share resources (like it is e.g. often the case in multithreaded programs) they must satisfy certain conditions to be always executable correctly

– Otherwise problems could occur like:
  – Deadlocks
  – Livelocks
  – Starvation
  – Necessarily atomic operations split errors
Parallel Processing using a Network of Computers

- Since the buy in of effort for faster single computers **accelerates**, it becomes more efficient for large scale data processing to use **multiple computers connected over a network**

- In computations **distributed** over such a network, **similar problems** can occur like in parallel processing on one computer unit

- As a result the creation of programs using such an infrastructure often is very specific and costly
Need for Methods to Simplify Parallel Computing

- Various problems of parallel processing with networks of computers (parallel computing) exist
- Searching new abstraction that allows to express the simple computations but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing
Using MapReduce Programming Model and Infrastructure to Simplify Parallel Computing

- Programming model to handle big unstructured or semi-structured datasets
- Software framework for distributed processing on commodity hardware
Using MapReduce Programming Model and Infrastructure to Simplify Parallel Computing

- Programs written in this functional style are automatically parallelized and can be executed on a large cluster of commodity machines.

- The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication.
Using MapReduce Programing Model and Infrastructure to Simplify Parallel Computing

- This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.
- Many real world tasks are expressible in this model.
- Usual implementations of MapReduce run on a large cluster of commodity machines and are highly scalable.

A typical MapReduce computation processes many terabytes of data on thousands of machines.
The MapReduce Programming Model

- Users specify a **map function** that processes a key-value pair to generate a set of **intermediate key-value pairs**, and a **reduce function** that merges all intermediate values associated with the same intermediate key.
  - Map function calculates **for each key** a related result for an arbitrary part of the input data
  - Reduce function **combines the results** of the map function instances to generate the final output data
The MapReduce Programming Model

Word Count
Sample Code for Word Count on MapReduce

Map function:

mapper (filename, file-contents):
   for each word in file-contents:
      emit (word, 1)
Sample Code for Word Count on MapReduce

**Reduce function:**

```python
reducer (word, values):
    sum = 0
    for each value in values:
        sum = sum + value
    emit (word, sum)
```
Sample Code for Word Count on MapReduce

Java Implementation:

```java
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 Reduce 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));
    }
}
The MapReduce Reference Implementation

- Many **different implementations** of the *MapReduce* interface are possible. The right choice depends on the **environment**
- The publishers of the *MapReduce* architecture delivered a **reference implementation** along with the publication (as used by *Google*)
The MapReduce Reference Implementation

Intermediate files are partitioned by hash function for reduce workers

- Remote read data -> wait for all partitions to finish -> sort data by key -> execute reduce function per key
The MapReduce Reference Implementation

- **Map** invocations are distributed across multiple machines by automatically partitioning the input data into a set of **splits**. The input splits can be processed in parallel by different machines.

- **Reduce** invocations are distributed by partitioning the intermediate key space into pieces using a partitioning function (e.g. $\text{hash(key)} \mod R$). The number of partitions ($R$) and the partitioning function are specified.
The MapReduce Reference Implementation: Fault Tolerance

- **Worker Failure**
  - If no response is received from a worker in a certain amount of time, the master marks the worker as failed.
  - When a map or reduce tasks is **completed** by the worker, the worker becomes eligible for rescheduling.
    - MapReduce is **resilient** to large-scale worker failures.

- **Master Failure**
  - Restart of calculation needed.
    - Suggestion to write periodic checkpoints of the master data structures instead.
    - Many current implementations use redundant master nodes.
The MapReduce Reference Implementation: Locality

- **Network bandwidth** is a relatively scarce resource in our computing environment.
- Conservation of network bandwidth by taking advantage of the fact that the input data (managed by *GFS*) is **stored on the local disks** of the machines that make up our cluster.
The MapReduce Reference Implementation: Locality

- The *MapReduce* master takes the location information of the input files into account and attempts to schedule a map task on a machine that contains a replica of the corresponding input data.
- Failing that, it attempts to schedule a map task near a replica of that task's input data.
- When running large operations on a significant fraction of the workers, most input data is read locally and consumes no network bandwidth.
The MapReduce Reference Implementation: Backup Tasks

- One of the common causes that *lengthens* the total time taken for a *MapReduce* operation is a “*straggler*” (a machine that takes an *unusually long time* to complete one of the last few map or reduce tasks in the computation)

- Stragglers can arise for a whole host of reasons
  - Other tasks on the machine are scheduled
  - Disk problems
  - Bug in machine initialization
The MapReduce Reference Implementation: Backup Tasks

Alleviating the problem:

- When a MapReduce operation is close to completion, the master schedules backup executions of the remaining in-progress tasks.
- The task is marked as completed whenever either the primary or the backup execution completes.

- Programs usually take much longer to complete when the backup task mechanism is disabled.
Application Examples for MapReduce

- Distributed Sort
- Distributed Grep
- Count of URL Access Frequency
- Reverse Web-Link Graph
- Term-Vector per Host
- Inverted Index
Application Examples for MapReduce: Distributed Sort

- Takes advantage of reducer properties:
  - (key, value) pairs are processed in order by key
  - reducers are themselves ordered

- Mapper:
  - Identity function for value: (k, v) \rightarrow (v, \_)

- Reducer:
  - Identity function: (k', \_ \_\_) \rightarrow (k', "")

- Must pick the hash function for your reducers such that \( k_1 < k_2 \Rightarrow \text{hash}(k_1) < \text{hash}(k_2) \)
Application Examples for MapReduce: Distributed Grep

- Mapper key is file name, line number
- Mapper value is the contents of the line
- Search pattern sent as special parameter

- Mapper:
  - Given (filename and line number, some text) and “pattern”, if “text” matches “pattern” output (filename and line number, _)

- Reducer:
  - Identity function
MapReduce Implementations

- Google MapReduce
- Hadoop
- Disco
- MapReduce-MPI
- MARIANE
- MARISSA
- Phoenix
Conclusions

- The *MapReduce* concept simplifies the procession of **large amounts of data**
- It delivers a **performant and stable** environment for massive calculations

- It is a great step towards establishing **Big Data** technologies
Additional

- Introduction Concept Video
References


C. Lengauer, S. Apel, and J. Dörre, “MapReduceFoundation: Typing of MapReduce.”
http://www.infosun.fim.uni-passau.de/cl/MapReduceFoundation/

http://users.sdsc.edu/~glockwood/comp/mapreduce.php

Yahoo! Inc., “Yahoo! Hadoop Tutorial.”
Thanks for your attention!