SS 2009
Proseminar: Beautiful Code

Distributed Programming with MapReduce
by Jeffrey Dean and Sanjay Ghemawat

Presentation: Elke Weber
04 June 2009
Structure

1. What is MapReduce?
2. Motivation
3. Example program: word count
4. Naïve, non-parallel word count program
5. Parallelized word count programs
6. The MapReduce Programming Model
7. Implementations
8. Demonstration: Disco
9. Conclusion
What is MapReduce?

- a programming system for large-scale data processing problems
- a parallelization pattern/programming model
- separates the details of the original problem from the details of parallelization
Motivation

• simplifying the large-scale computations at Google
• originally developed for rewriting the indexing system for the Google web search product
• MapReduce programs are automatically parallelized and executed on a large-scale cluster
• programmers without any experience with parallel and distributed systems can easily use large distributed resources
Example program: word count

- you have 20 billion documents
- generate a count of how often each word occurs in the documents
- average document size 20 kilobytes → 400 terabytes of data
Naïve, non-parallel word count program

Assumption: Machine has sufficient memory!

```cpp
    map<string, int> word_count;
    for each document d {
        for each word w in d {
            word_count[w]++;
        }
    }
    ... save word_count to persistent storage ...

→ will take roughly 4 months
Parallelized word count program

```c++
Mutex lock; // protects word_count
map<string, int> word_count;
for each document d in parallel {
    for each word w in d {
        lock.Lock();
        word_count[w]++;
        lock.Unlock();
    }
}
... save word_count to persistent storage ...

→ problem: uses a single global data structure for generated counts
```
Parallelized word count program with partitioned storage

```cpp
struct CountTable {
    Mutex lock;
    map<string, int> word_count;
};

const int kNumBuckets = 256;
CountTable tables[kNumBuckets];
```
for each document d in parallel {
  for each word w in d {
    int bucket = hash(w) % kNumBuckets;
    tables[bucket].lock.Lock();
    tables[bucket].word_count[w]++;
    tables[bucket].lock.Unlock();
  }
}
for (int b = 0; b < kNumBuckets; b++){
  ... save tables[b].word_count to persistent storage ...
}
Parallelized word count program with partitioned storage

- no more than the number of processors in a single machine
- most machines 8 or fewer processors → requires still multiple weeks of processing

- solution: distribute the data and the computation across multiple machines
Parallelized word count program with partitioned processors

Assumption: Machines do not fail!

```cpp
cost int M = 1000; // number of input processes
cost int R = 256; // number of output processes
```
Parallelized word count program with partitioned processors 2

```c
main() {
    // Compute the number of documents to assign to each process
    const int D = number of documents / M;
    for (int i = 0; i < M; i++) {
        fork InputProcess(i * D, (i + 1) * D);
    }
    for (int i = 0; i < R; i++) {
        fork OutputProcess(i);
    }
    ... wait for all processes to finish ... 
}
```
void InputProcess(int start_doc, int end_doc) {
    // Separate table per output process
    map<string, int> word_count[R];
    for each doc d in range [start_doc .. end_doc-1] {
        for each word w in d {
            int b = hash(w) % R;
            word_count[b][w]++;
        }
    }
    for (int b = 0; b < R; b++) {
        string s = EncodeTable(word_count[b]);
        ... send s to output process b ...
    }
}
Parallelized word count program with partitioned processors

```cpp
void OutputProcess(int bucket) {
    map<string, int> word_count;
    for each input process p {
        string s = ... read message from p ...
        map<string, int> partial = DecodeTable(s);
        for each <word, count> in partial {
            word_count[word] += count;
        }
    }
    ... save word_count to persistent storage ...
}
```
Parallelized word count program with partitioned processors

- scales nicely on a network of workstations
- but more complicated and hard to understand
- deals not with machine failures
- adding failure handling would further complicate things
The MapReduce Programming Model

• separate the details from the original problem from the details of parallelization

• parallelization pattern:
  • For each input record, extract a set of key/value pairs that we care about from each record.
  • For each extracted key/value pair, combine it with other values that share the same key.
Division of word counting problem into Map and Reduce

```java
void Map(string document) {
    for each word w in document {
        EmitIntermediate(w, "1");
    }
}

void Reduce(string word, list<string> values) {
    int count = 0;
    for each v in values {
        count += StringToInt(v);
    }
    Emit(word, IntToString(count));
}
```
map<string, list<string> > intermediate_data;

void EmitIntermediate(string key, string value){
    intermediate_data[key].append(value);
}

void Emit(string key, string value) {
    ... write key/value to final data file ... 
}
void Driver(MapFunction mapper,
            ReduceFunction reducer) {
    for each input item do {
        mapper(item)
    }
    for each key k in intermediate_data {
        reducer(k, intermediate_data[k]);
    }
}

main() {
    Driver(Map, Reduce);
}
The MapReduce Programming Model

- example implementation runs on a single machine
- because of separation → we can now change the implementation of the driver program
- driver dealing with:
  - distribution
  - automatic parallelization
  - fault tolerance
- independent of the Map and Reduce functions
Other Map and Reduce Examples 1

- distributed grep
- reverse web-link graph
- term vector per host
- inverted index
- distributed sort
- many more ...
Other Map and Reduce Examples 2

- complex computations:
  - a sequence of MapReduce steps
  - iterative application of a MapReduce computation
- March 2003: small handful of MapReduce programs at Google
- December 2006: 6,000 distinct MapReduce programs
A distributed MapReduce Implementation

- for running large-scale MapReduce jobs
- large clusters of PCs connected together with switched Ethernet (in wide use at Google)
- machines with dual-processors x86, Linux, 2-4 GB memory
- GFS (Google File System): file chunks of 64 MB, 3 copies of each chunk on different machines
- user submits jobs to a scheduling system
Relationships between processes in MapReduce
Implementation details

• load balancing
• fault tolerance
• locality
• backup tasks
Extensions to the Model

- partitioning function
- ordering guarantees
- skipping bad records
- some other extensions (see paper about MapReduce)
Implementations

- The Google MapReduce framework
- Apache Hadoop (Yahoo!, Facebook, IBM, Last.fm ...)
- Cell Broadband Engine
- NVIDIA GPUs (Graphics Processors)
- Apache CouchDB
- Skynet
- Disco (Nokia)
- Aster Data Systems (MySpace)
- Bashreduce
Demonstration: Disco
massive data – minimal code

• open-source implementation of MapReduce
• Nokia Research Center
• supports parallel computations over large data sets on unreliable clusters of computers
• Disco core: Erlang
• jobs: Python
Conclusion

- Google:
  - early 2007 more than 6,000 distinct MapReduce programs
  - more than 35,000 MapReduce jobs per day
  - about 8 petabytes of input data per day
  - about 100 gigabytes per second
Conclusion

• useful across a very broad range of problems:
  • machine learning
  • statistical machine translation
  • log analysis
  • information retrieval experimentation
  • general large-scale data processing and computation tasks
References

● "Distributed Programming with MapReduce" by Jeffrey Dean & Sanjay Ghemawat:


● list of different implementations:
  ● http://en.wikipedia.org/wiki/MapReduce

● Disco:
  ● http://discoproject.org/
Further Reading

- "MapReduce: Simplified Data Processing on Large Clusters."
  http://labs.google.com/papers/mapreduce.html

- "The Google File System."
  http://labs.google.com/papers/gfs.html

- "Web Search for a Planet: The Google Cluster Architecture."
  http://labs.google.com/papers/googlecluster.html

- "Interpreting the Data: Parallel Analysis with Sawzall."
  http://labs.google.com/papers/sawzall.html