MapReduce: Simplified Data Processing on Large Clusters

MapReduce: is a programming model and an associated implementation for processing and generating large data sets.

This model is divided into two functions:

map function that processes a key/value pair to generate a set of intermediate key/value pairs

reduce function that merges all intermediate values associated with the same intermediate key.

The goal of this model is to parallel and utilize the resource of the working machine on a big cluster

In order for the computations to be completed in a reasonable length of time, the input data is typically huge, and the computations must be distributed across hundreds or thousands of machines.

The map and reduce primitives found in Lisp and many other functional languages serve as the basis for our abstraction.

A straightforward and effective interface that permits automatic parallelization and distribution of large-scale calculations is one of this work's main achievements.

Programming Model:

Map, written by the user, takes an input pair and produces a set of intermediate key/value pairs.

The Reduce function is called once the MapReduce library has collected all intermediate values connected to the same intermediate key I.

The user-written Reduce function accepts a set of values for the intermediate key I as well as an intermediate key I. It merges together these values to form a possibly smaller set of values.

The intermediate values are supplied to the user’s reduce function via an iterator.

Programs that can utilize the MapReduce model:

Distributed Grep

Count of URL Access Frequency

Reverse Web-Link Graph

Term-Vector per Host

Inverted Index

Distributed Sort

Implementation:

There are many implementations for the MapReduce based on the program, one implementation may be suitable for a small shared-memory machine, another for a large NUMA multi-processor

1. Machines are dual-processor x86 processors running Linux, with 2-4 GB of memory per machine.
2. Commodity networking hardware is either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in overall bisection bandwidth.
3. A cluster consists of hundreds or thousands of machines.
4. Storage is provided by inexpensive IDE disks attached directly to individual machines, The file system uses replication to provide availability and reliability on top of unreliable hardware.
5. Users submit jobs to a scheduling system

Execution Overview

partitioning the input data into a set of M splits, The input splits can be processed in parallel by different machines. Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function.

Step sequence of implementation:

1. The MapReduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece. It then starts up many copies of the program on a cluster of machines.
2. One of the copies of the program is special – the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.
3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory

4. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory
5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.
6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.
7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.

Master Data Structures

For each map task and reduce task, it stores the state and the identity of the worker machine.
The master is the conduit through which the location of intermediate file regions is propagated from map tasks to reduce tasks.
for each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task.

Fault Tolerance

- **Worker Failure:**

The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed.

Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible.
reduce tasks do not need to be re-executed since their output is stored in a global file system.
When a map task is executed first by worker A and then later executed by worker B, all workers executing reduce tasks are notified of the reexecution.

- **Master Failure**

If the master task dies, a new copy can be started from the last checkpointed state. there is only a single master, its failure is unlikely; therefore our current implementation aborts the MapReduce computation if the master fails. Clients can check for this condition and retry the MapReduce operation if they desire.

- **Semantics in the Presence of Failures**

our distributed implementation produces the same output as would have been produced by a non-faulting sequential execution of the entire program.
We rely on atomic commits of map and reduce task outputs to achieve this property. Each in-progress task writes its output to private temporary files. A reduce task produces one such file, and a map task produces R such files.
When a map task completes, the worker sends a message to the master and includes the names of the R temporary files in the message.

Locality

We conserve network bandwidth by taking advantage of the fact that the input data is stored on the local disks of the machines that make up our cluster. GFS divides each file into 64 MB blocks, and stores several copies of each block (typically 3 copies) on different machines.
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.

Task Granularity

We subdivide the map phase into M pieces and the reduce phase into R pieces.
M and R should be much larger than the number of worker machines.

Having each worker perform many different tasks improves dynamic load balancing, and also speeds up recovery when a worker fails: the many map tasks it has completed can be spread out across all the other worker machines.

Backup Tasks

Stragglers can arise for a whole host of reasons.
The cluster scheduling system may have scheduled other tasks on the machine, causing it to execute the MapReduce code more slowly due to competition for CPU, memory, local disk, or network bandwidth. A recent problem we experienced was a bug in machine initialization code that caused processor caches to be disabled: computations on affected machines slowed down by over a factor of one hundred.
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.
We have tuned this mechanism so that it typically increases the computational resources used by the operation by no more than a few percent.

Performance

One computation searches through approximately one terabyte of data looking for a particular pattern. The other computation sorts approximately one terabyte of data. These two programs are representative of a large subset of the real programs written by users of MapReduce – one class of programs shuffles data from one representation to another, and another class extracts a small amount of interesting data from a large data set.

Cluster Configuration

All of the programs were executed on a cluster that consisted of approximately 1800 machines.

Each machine had two 2GHz Intel Xeon processors with HyperThreading enabled, 4GB of memory, two 160GB IDE disks.

Grep

The grep program scans through 1010 100-byte records, searching for a relatively rare three-character pattern (the pattern occurs in 92,337 records). The input is split

into approximately 64MB pieces (M = 15000), and the entire output is placed in one file (R = 1).

Sort

The sort program sorts 1010 100-byte records (approximately 1 terabyte of data).
This program is modeled after the TeraSort benchmark.
The sorting program consists of less than 50 lines of user code.

Effect of Backup Tasks

we show an execution of the sort program with backup tasks disabled.
After 960 seconds, all except 5 of the reduce tasks are completed.

Machine Failures

The underlying cluster scheduler immediately restarted new worker processes on these machines.
The worker deaths show up as a negative input rate since some previously completed map work disappears and needs to be redone.

Experience

It has been used across a wide range of domains within Google, including:

- large-scale machine learning problems,
- clustering problems for the Google News and Froogle products
- extraction of data used to produce reports of popular queries, extraction of properties of web pages for new experiments and products.
- large-scale graph computations.

Related Work

- Bulk Synchronous Programming and some MPI primitives provide higher-level abstractions that make it easier for programmers to write parallel programs. A key difference between these systems and MapReduce is that MapReduce exploits a restricted programming model to parallelize the user program automatically and to provide transparent fault-tolerance.
- River provides a programming model where processes communicate with each other by sending data over distributed queues.The restricted programming model also allows us to schedule redundant executions of tasks near the end of the job which greatly reduces completion time in the presence of non-uniformities (such as slow or stuck workers).
- BAD-FS has a very different programming model from MapReduce, and unlike MapReduce, is targeted to the execution of jobs across a wide-area

network.there are two fundamental similarities. (1) Both systems use redundant execution to recover from data loss caused by failures. (2) Both use locality-aware scheduling to reduce the amount of data sent across congested network links.