PADIMapNoReduce

Pedro Braz 73991, Ricardo Wagenmaker 73810, Rui Pereira 70600
Instituto Superior Técnico
Av. Prof. Doutor Aníbal Cavaco Silva
2744-016 Porto Salvo
{ruijosemangas, pbraz.93, rw1818}@gmail.com

Abstract

There has been an increase in the number of applications that require large scale data analytics and one of the most popular approaches is MapReduce, which is a programming model and an associated implementation for processing and generating large data sets. Users of this approach express the computation as two functions : Map and Reduce.

We implement the map part of MapReduce, showing that it can be improved by having the job tracker's functionalities implemented in a distributed manner, in opposition to the original implementation where this component is centralised.

1 Introduction

MapReduce is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster being first introduced by Google in 2004. By using this programming paradigm it is possible to reach massive scalability across hundreds or thousands of servers.

The term MapReduce actually refers to two separate and distinct tasks.

- **map :** takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs).
- **reduce :** takes the output from a map as input and combines those data tuples into a smaller set of tuples. As the sequence of the name MapReduce implies, the reduce job is always performed after the map job.

PADIMapNoReduce is a simplified implementation of the MapReduce middleware and programming model, concerning only about the mapping part. The computation in our implementation takes the input key/value pairs from input files, where the keys are the numbers of the line of the file being read and the values are the content

In the case of PADIMapNoReduce, the input key/value pairs are extracted from input files. The keys are the numbers of the line of the file being read and the values are the content of those lines.

2 Architecture

2.1 Application API and Clients

cenas API + Clients

2.2 Worker/Job Tracker

cenas dos workers + job tracker

2.3 Puppet Master

cenas do puppet master

3 Implementation

The computation in our implementation takes the input key/value pairs from input files, where the keys are the numbers of the line of the file being read and the values are the content

3.1 Execution Overview

Isto é preciso modificar!!!

The Map invocations are distributed across multiple machines by automatically partitioning the input data To appear in OSDI 2004 3 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 (e.g., hash(key) mod R). The number of partitions (R) and the partitioning function are specified by the user. Figure 1 shows the overall flow of a MapReduce operation in our implementation. When the user program calls the MapReduce function, the following sequence of actions occurs (the numbered labels in Figure 1 correspond to the numbers in the list below):

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 (controllable by the user via an optional parameter). 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. *Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.*

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.*

7. *After successful completion, the output of the mapreduce execution is available in the R output files (one per reduce task, with file names as specified by the user). Typically, users do not need to combine these R output files into one file – they often pass these files as input to another MapReduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.*

3.2 Master Data Structures

The master keeps several data structures. For each map task and reduce task, it stores the state (idle, in-progress, or completed), and the identity of the worker machine (for non-idle tasks). The master is the conduit through which the location of intermediate file regions is propagated from map tasks to reduce tasks. Therefore, for each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task. Updates to this location and size information are received as map tasks are completed. The information is pushed incrementally to workers that have in-progress reduce tasks.

4 Refinements

4.1 Fault Tolerance

Since the MapReduce library is designed to help process very large amounts of data using hundreds or thousands of machines, the library must tolerate machine failures gracefully.

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. Any map tasks completed by the worker are reset back to their initial idle state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to idle and becomes eligible for rescheduling. 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. Completed 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 (because A failed), all workers executing reduce tasks are notified of the reexecution. Any reduce task that has not already read the data from worker A will read the data from worker B. MapReduce is resilient to large-scale worker failures. For example, during one MapReduce operation, network maintenance on a running cluster was causing groups of 80 machines at a time to become unreachable for several minutes. The MapReduce mastersimply re-executed the work done by the unreachable worker machines, and continued to make forward progress, eventually completing the MapReduce operation.

Master Failure

It is easy to make the master write periodic checkpoints of the master data structures described above. If the master task dies, a new copy can be started from the last checkpointed state. However, given that 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.

5 Conclusions

References

[1] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation, *December 2004.*