MapReduce: Simplified Data Processing on Large Clusters

Garrett Kolpin
CS 739: Distributed Systems
University of Wisconsin, Madison

Spring 2006

1 Overview

MapReduce is a programming model designed to hide the complexities of scheduling, parallelization, failure handling, and computation distribution across a cluster of nodes. This programming model is especially well suited for very large sets of data. A program written in this way specifies both a map and a reduce function. The map function produces a set of intermediate key/value pairs which the reduce function uses as input. The reduce function merges all intermediate values with the same intermediate key.

2 Motivation and Assumptions

The motivation for MapReduce is to take advantage of the distributed file system GFS and to give a model to the user so that the user need not worry about the issues stated above such as parallelization, failure handling, etc. A user supplies both the map and reduce functions which are linked together with the MapReduce library. This library provides the hook into the MapReduce system.

The paper notes that many different implementations of MapReduce are possible, however the specific implementation at Google consists of a cluster of up to 1,800 nodes. Each node is built using commodity hardware and cheap inexpensive IDE disks which makes failures very common.

It is common to run a MapReduce operation on a data set on the order of terabytes in size. It is thus important for the MapReduce implementation to be able to handle and run on data that may not entirely fit into the cluster's memory.

3 Design

MapReduce is started by first spawning a master and some set of workers. Also, at job creation time there are parameters that can be used to tune runtime behavior such as scheduling. *In order to take loads across the entire cluster into consideration, there must be some other entity not described in the paper that is used for global load balancing. Therefore, there is probably not a single master for all programs, but lesser local masters for each MapReduce job. Also, there might need to be a high level work-flow scheduler so that output of one phase is sure to complete before it is needed as input to another phase of MapReduce.*

A user gives the MapReduce operation a set of input files on GFS whose locations are found by asking GFS about the chunkservers hosting each input file. The master will then start M mappers and R reducers. The master will also try to run the map function on the same chunkserver that is hosting a particular input file. If this is impossible, then it will try to locate a mapper on the same switch to reduce network transfer overhead.

3.1 Mappers and Reducers

The intermediate files produced by the mapping function reside on the local disk of the mapper. Since intermediate map files are not replicated, a failure of a completed mapper node will trigger an entire re-computation of the intermediate file. Worker failures are detected by the master through periodic heartbeat messages.

The reducer reads intermediate files produced from the mappers through RPC calls. It is unlikely that the interme-

diate files are local to the reducer. Therefore, the reducer must go to a remote worker for an intermediate input file. If a reducer fails before its computation is complete, the computation will be restarted.

Output files from the reducer workers are written to GFS. The files are atomically renamed so as to function as a sort of atomic commit. Any reducer failure that occurs once the final output has completed does not affect the output since GFS provides replication.

3.2 Master

If a master crashes, then the user must resubmit the job since a crashed master will abort the computation. The master also tracks the location of intermediate files and the workers currently working on each task. *The master could also checkpoint as it runs since there really is not too much state involved, however this is not done.* The designers favored re-computation than a little bit of added complexity and performance overhead for taking checkpoints. But since there is only a single worker, a worker failure will most likely happen much less often than worker failures.

4 When To Use GFS

GFS is not used for intermediate files since replication of the files is slow. *In general a system is simpler if one need not be concerned about replication and persistence. In this case, it is simpler to re-run the computation rather than providing the intermediate files with persistence.* However, GFS *is* used for output files. Again, making multiple replicas is a slow process, but the benefit of having persistent output data in this case outweighs the cost of recomputing the entire MapReduce operation.

The MapReduce designers have seen the importance of using the local bus for data whenever possible. That is, keeping data local is very important in order to have good performance. An example of this is when GFS nodes are used as mappers, and the master attempts to have input data be local to the mappers. Also, the intermediate file output is kept on a local disk as well.

Final output is stored on GFS, but a replica is not kept on the local disk. *The most probable reason is that the next stage of MapReduce can begin sooner if the replicas are non-local.* Otherwise the next stage would be competing with the execution of the current stage.

4.1 Load Balancing

In the River programming environment, data is very flexible in regards to where it runs. If either a producer or consumer is running slowly, then less data is requested from or sent to each respectively. This same idea is used somewhat in MapReduce. If there is a straggler reducer or mapper, then another one is co-scheduled on a separate node and the output of the fastest node wins. In this way, faster mappers and reducers grab more jobs than slower mappers, thereby increasing the performance of the entire computation.

5 MapReduce Sort vs. NOW Sort

The MapReduce sort has been written in roughly 50 lines of code which is impressive given the complexity of the parallelization of the computation. MapReduce already processes keys in a sorted order, so most of the work has already been done in the MapReduce infrastructure. As with NOW Sort, there are 100 byte records and 10 byte keys. The keys are then separated from the records. In order to partition inputs across the workers, the modulo function is used by looking at the top few bits and finding the remainder when divided by the number of mappers. In this case, a reducer is not really needed since the data is already sorted by the end of the map phase. The reducer then is simply the identity function.

Both the NOW sort and the MapReduce sort get the input data from the local disk initially. Also, both methods assume an initial even distribution of keys across all the nodes. As NOWSort reads the keys, it sends each key to the correct machine with the appropriate bucket (which could be remote). However, MapReduce sends to the local disk first before the shuffle phase begins. The final phase for both methods is simply a local sort of the data. NOWSort finishes by writing the output data to the local disk. MapReduce on the other hand uses GFS to store output data, so one replica is produced. This means that two writes are needed at the end which decreases performance but is more robust to failure.

6 Evaluation

6.1 Grep

The Grep application has the mappers doing all of the work. Only one reducer is required which writes to the output file the results from the mappers. It takes the grep application nearly one full minute to start up, which does not seem acceptable. However, this startup time consists of sending data to nearly 1,800 workers and conflicts in dealing with GFS. *It would be interesting to see how this would have performed given fewer total workers. Perhaps the faster startup time could more than make up for fewer peak workers.*

6.2 Sort

A sort of one terabyte of data required 891 seconds to complete which is better than the current Terasort record. Thus, it seems MapReduce is achieving very good performance.

An interesting note is the MapReduce had enough memory in the entire cluster used to keep all keys in memory. However, the designers designed the system for the common case where there is generally not enough memory for this, so data is written to disk in between the map and reduce operations.

7 Conclusion

To conclude, the MapReduce operation is impressive when one thinks about the amount of complexity hidden to the user. From the evaluation it appears that MapReduce also has very good performance and can scale to a few thousand nodes. *Given the emphasis on the simplicity of the system, it would have been nice to see some more concrete examples of applications written in the MapReduce framework so that the simplicity of the code could speak for itself.*