MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

Paper Summary by Group 5:
Alok Thatikunta (111498776), Rahul Sihag (111462160), Sweta Kumari (111497926)
{athatikunta, rsihag, swkumari} @cs.stonybrook.edu

The paper approaches the issue of parallelizing the computation, distributing data and handling failures while processing humongous data on large clusters. It introduces a scalable programming model and an associated implementation for processing and generating big data sets which hides the details of parallelization, fault-tolerance, data distribution, and load balancing in a library. The model is inspired by *map* and *reduce* primitives already present in many functional programming languages.

As per the basic programming model suggested by the paper, the computation takes a set of input key/value pairs, and produces a set of output key/value pairs.The computations are two user defined functions - *Map* and *Reduce*. *Map* takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key *I* and passes them to the Reduce function. The Reduce function accepts an intermediate key *I* and a set of values for that key and merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per Reduce invocation.The paper also mentions few examples like calculating the word frequency, distributed Grep etc. It also discusses several refinements of the basic programming model, viz. Partitioning functions, ordering guarantee, combiner functions, input/output types, skipping unwanted records, status informations, counters for events, local execution and side- effects.

The paper also provides a detailed discussion on the implementation of the MapReduce interface tailored towards Google's cluster based computing environments and its performance measurements for variety of tasks. 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(configurable). It makes multiple copies, one of which is *Master* which assign M map and R reduce tasks to idle workers. A map task reads the input and produces intermediate key/value pairs which is first buffered and then is written in local disk, whose location is sent to master which then passes it to reduce workers where it is sorted and then inputted to the Reduce function which produces the final output. Master task periodically pings workers and if any worker has failed, it is assigned to another worker and all the reduce task are notified of the re-execution. In the current implementation, if the Master fails, MapReduce computation aborts. Considering the scarcity of network bandwidth, master attempts to assign the task to a machine that contains one of the replica of data(out of 3 made by GFS). Failing that, it attempts to assign

it to worker near to it. Partitioning of input into M and R pieces in the two phases has practical bounds, as master has to make *O(M+R)* scheduling decisions and keep *O(M*R)* states in the memory.

The paper also discuss the performance in detail by performing MapReduce on two computations running on a large cluster of machines. One computation(Grep) searches through approximately one terabyte of data looking for a particular pattern. The other computation(Sort) sorts approximately one terabyte of data. It is observed that the input rate is higher than the shuffle rate and the output rate because of our locality optimization - most data is read from a local disk and bypasses our relatively bandwidth constrained network. The shuffle rate is higher than the output rate because the output phase writes two copies of the sorted data (we make two replicas of the output for reliability and availability reasons). Also, in case of process failures, the underlying cluster scheduler immediately restarted new worker processes on those machines.

The paper also shows some statistics how massively MapReduce is being used and is successful at Google in various domains, viz. Google news, Froogle products, large scale indexing for Google web search service, ZeitGeist, etc. Comparison with related work like Bulk Synchronous programming, Charlotte system, NOW- sort, Condor, River, BAD-FS and TACC has also been mentioned in paper. Various aspects like abstraction, backup task mechanism, sorting facility have been taken into consideration while comparing it with the related models.

Although the paper presents a very influential library, it doesn't mention the comparative performance, i.e. performance statistics with and without using MapReduce. Also, it fails to elaborate on the problem of failure of Master. It just presumes that it's unlikely and doesn't address the problem completely.