MapReduce Summary

Dean and Ghemawat describe MapReduce as a programming model and an associated implementation for “processing and generating large data sets” [1]. It was developed in response to the increasing need for a system which handled the parallelization of large computations, distribution of data and failure handling without overtaking the initial simplicity of the code produced to do these computations in the first place. Thus, MapReduce was proposed - based on the map and reduce primitives in the Lisp functional language, it as an abstraction that allows the above concerns to be hidden from the user.

The model consists of two distinct paradigms: map and reduce. Map applies a map to a record of input to compute intermediate key/value pairs, whilst Reduce applies a reduce operation to all values with the same key, to reduce the large data resulting from map into smaller derived data. This results in a system that encompasses easy parallelization and use of re-execution for fault tolerance, seeing as data is duplicated to avoid data loss by storage devices. These are the main advantages of MapReduce, as large volumes of data are processed very quickly by distributing processing tasks across clusters of commodity servers [2]. In addition, there is effective load balancing since incoming network traffic is distributed across a group of backend servers, also known as a server farm or server pool.

Map	**Reduce**
Written by user	Written by user
Takes input pair and produces intermediate key/ value pair	Accepts intermediate key I, set of values for said key (which are supplied via an iterator - helpful for large lists of values)
A library groups intermediate values with same key I together, and passes them to Reduce	Merges values together to form smaller sets of values for said key
	Output in the form of 0 or 1

Table 1. A breakdown of Map and Reduce functions

An example application is presented in the paper, which involves the pseudocode of counting occurrences of a word in a large doc collection. After map and reduce are written, a map reduce object is invoked with the names of input/output files and tuning parameters. This object is then passed to the MapReduce function, linked with the library and executed.

Types aren’t strings, but rather have input keys and values drawn from diff domain as output keys and values - same format as intermediate. Conversion from strings to such types must be accounted for by users. Other examples presented include distributed grep, count of url access frequency, reverse web-link graph, etc.

The Google Set Up

According to the paper, there are many implementations possible depending on the chosen environment. The Google implementation is presented below.

1. Commodity PC clustered via ethernet
2. Dual processor x86 processors running Linux, 2-4GB of RAM per machine
3. 100 megabits/second or 1gigabit/second networking speeds.
4. Hundreds/thousands of computers
5. Distributed file system manages data on inexpensive disks, which is convenient since the data is accessed and processed as if it was stored on the local client machine, and can be shared in a controlled and authorized way.
6. Scheduler for managing job submission, where a job is a set of tasks which is mapped to a set of available worker machines within a cluster by a master machine.

Master Data Structures

For each Map and Reduce job, various data structures are kept by the master. It stores the state (whether idle, in progress, or completed), as well as the identity of the worker machine for non-idle tasks. There is the need for all these metrics to be tracked in order to supplement the fault tolerant nature of the model in cases any failure, including worker or master failure. These cases are considered in the following sections.

Worker Failure

- Master pings workers periodically; if no response, worker failure - set idle state.
- The task is then scheduled to another cluster; even if the map task is complete, it must be reelected since data is stored on the worker's local disk which is now inaccessible. Completed reduce tasks have output stored on global file system so there's no need for their re-execution.
- Upon re-execution of map task, all workers executing reduce tasks are notified, and they will read the new data if the old data wasn't read. As a result, this methodology makes the system resilient to large scale worker failure.

Master Failure

- Checkpoints of current data structures are periodically saved, so its easy for a new master to take over from last checkpoint. Although failure is theoretically unlikely, computation is aborted if master fails. Clients can check for this condition and retry the MapReduce operation if they desire.

Atomicity

Due to the above contingencies, if there is a system failure, the output of the map and reduce jobs are just as though no error was present. This is due to the fact that there is atomicity in map/reduce outputs, which ensures that the master is able to ignore completion messages for an already completed map task (to prevent duplicates). It also records the names of the R temporary files written by in-progress map tasks.

For reduce tasks, there is one temporary file, and the reduce worker atomically renames it to the final output file. If the same reduce task is executed on multiple machines, multiple rename calls will be executed for the same final output file. The atomic rename is used to guarantee that the final file system state contains just the data by one reduce task.

Task Granularity

Granularity can be defined as a relative measure of the ratio of the amount of computation to the amount of communication within a parallel algorithm implementation [3]. In order to determine the granularity of the system, the map phase is divided into M pieces, and the reduce phase into R pieces. According to the paper, M and R should ideally be larger than the number of machines, such that each worker is able to perform many different tasks. The bounds of the system are constrained by the fact that the master must make O(M+R) scheduling decisions and keep O(M*R) states in memory.

Additionally, the paper states that R is often constrained by users, since "the output of each reduce task ends up in a separate output file." This is why M is chosen such that each individual task is roughly 16 MB to 64 MB of input data and R is a small multiple of the number of worker machines being used. Google often performs MapReduce computations with M = 200, 000 and R = 5, 000, using 2,000 worker machines.

The concept of stragglers is also introduced, which describes the phenomenon which occurs when one machine takes an "unusual" amount of time to complete, resulting in a delay[1]. This delay can be alleviated by ensuring that the master schedules backup executions of the remaining in-progress tasks on a different worker machine, and marks said task as completed whenever either the primary or the backup execution completes. The use of this model is validated, as the authors mention that without the backup mechanism, one sort program can take 44% longer to complete.

Refinements

In this section, various helpful extensions which give the model better functionality are presented, including:

- Partition function: allows users to specify the number of reduce task outputs they desire.
- Ordering Guarantees: allows for generation of sorted output per given partition .
- Combiner function: executed on each worker processing a map task, this function writes to an intermediate file and partially merges the data before it is sent over the network.

Others extensions include skipping bad records, local execution, status information, counters, atomic auxiliary output files and input and output type support

Performance

This section considers the performance of MapReduce on two computation applications - grep and sort. The grep computation searches through approximately 1 TB of data looking for a particular pattern, and takes approximately 150 seconds from start to finish, including about a minute of startup overhead due to the propagation of the program to all worker machines, delays interacting with the file system, and obtaining the information needed for optimization. The sorting computation considers approximately 1 TB of data, and all the writes finish about 850 seconds into the computation. Including startup overhead, the entire computation takes 891 seconds, similar to the current best reported result of 1057 seconds for the TeraSort benchmark. Consequently, other applications of MapReduce as used in Google are highlighted, including large scale Machine Learning, clustering problems and the extraction of data.

Conclusion & Lessons Learned

Dean and Ghemawat present MapReduce as an easy to use model for all programmers as it implements abstraction and hides the complexities involved, allowing for the easy expression of all problems. Another plus is that it scales up well, uses resources efficiently and handles failures and data loss as a result of its redundant execution.

Conversely, the model requires high processing power, which is detrimental since network bandwidth and large server farms are scarce resources that many lone programmers may not have access to. Personally, this is reminiscent of the high cost involved with incorporating concurrent processes as I've learned in Operating Systems class. In addition, ideas such as persistence are presented in the reliability of disks during the computations, as well as virtualization of resources through the sharing of network and processing resources.

The importance of atomic execution of certain commands as presented in the paper also struck me, as this is another topic area which was heavily focused on in class, and I realized the potential ramifications of a lack of mutual exclusion in large scale processes as in the case of Google could be incredible.

Another point to note concerns the need for extensions in order to tweak the existing model to further improve it. I thought this was interesting because it goes to show there's always more one can do to improve a model and increase it's usability. This ties in well with the concept of abstraction, as its implementation in MapReduce clarified my confusion - rather than presenting elaborate functions and classes to implement the concept, the creators focused on "hiding the messy details" [1].

Finally, I questioned the idea of replication of data to provide "availability and reliability on top of unreliable hardware", as it could potentially result in severe memory overheads that a standard user might not have the architecture to handle. Initially, I believed that the waste may not necessarily be justified when considering the low likelihood of failure, however, after leaning about the RAID system, I concede that in a large-scale environment like Google, all possible precautions must be taken in order to ensure constant uptime.

In summarizing this paper, I have come to gain some insight into the concept of the MapReduce paradigm, which I had previously heard of but had no background information. In addition, I have been able to draw parallels with various OS concepts as implemented in the model, and have gleaned the importance of theoretical background in real life implementations, whilst gaining insight into the nuanced nature of said implementations in a world where the usual assumptions made in class do not hold.

References

[1]Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", OSDI, 2004.

[2]"The MapReduce Programming Paradigm", *Dummies*, 2018. [Online]. Available: https://www.dummies.com/programming/big-data/data-science/the-mapreduce-programming-paradigm/. [Accessed: 22- Nov- 2018].

[3]"Granularity", *WLU,* 1992. [Online]. Available: http://home.wlu.edu/~whaleyt/classes/parallel/topics/granularity.html. [Accessed: 22- Nov- 2018].