The title of the paper I read is : MapReduce: Simplified Data Processing on Large Clusters.

This paper introduced a programming model to deal with partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures and managing the required inter-machine communication, help programmers make full use of the resources of a large distributed system. In the past, many programmers have implemented a plenty of computations in order to process large amounts of raw data, but because of the large input data, the original simple computation needs large amounts of complex code. In another word, there are massive data processing requirements, but the actual calculation code is not complicated. The core demand is that the amount of data is too large to be executed on a single machine. Therefore, how to parallelize the computation, distribute the data and handle failures are crucial to this problem. The solution is distribute computations to thousands of machine, which can help shorten run time.

Many systems have provided programming models and used the restrictions to parallelize the computation automatically. For instance, associative function. An associative function can be computed over all prefixes of an N element array on N processors using parallel prefix, but it leaves the details of handling machine failures to the programmer, while MapReduce provide a fault-tolerant implementation that scales to thousands of processors. Bulk Synchronous Programming and some MPI primitives can also be used in write parallel programs, however, compared to MapReduce, the latter make use of restricted programming model to parallelize the user program automatically and to provide transparent fault-tolerance. NOW-Sort and MapReduce are similar in terms of sorting facility, but the library of NOW-Sort cannot widely applicable for it does not have the user-definable Map and Reduce functions. River achieves balanced completion times by careful scheduling of disk and network transfers, MapReduce achieves this by restricting the programming model to partition the problem into a large number of fine-grained tasks, then these tasks are dynamically scheduled on available workers so that faster workers process more tasks.

There are some technical challenges in fault tolerance, bandwidth resource, task granularity, how to choose appropriate value of M and R, and the length of total time. MapReduce handling fault tolerance from three aspects, worker failure, master failure and semantics in the presence of failures, each kind of failure has different mechanism. As bandwidth is a precious resource, MapReduce storing the input data on the local disks of the clustered machines to conserve network bandwidth. Ideally, the value of M and R should be much larger than the number of worker machines, but M and R are limited by the fact. When we use MapReduce model in practice, it would perform better if we choose an M which can make each individual task is roughly 16 MB to 64 MB of input data, and make R a small multiple of the number of worker machines we expect to use. A machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation is called "straggler", it can causes total execution time increased. There is a general mechanism to deal with stragglers: when a MapReduce operation is close to completion, the master schedules backup executions of the remaining in-progress tasks. This solution can reduces the time to complete large

MapReduce operations.

This paper measure the performance of MapReduce on two computations running on a large cluster of machines. One computation searches through approximately one terabyte of data looking for a particular pattern, the other computation sorts approximately one terabyte of data. For search computation, the entire computation takes about 150 seconds from start to finish. For sort computation, under normal execution, the entire task takes 891 seconds including startup overhead. The sort execution with backup tasks disabled takes 1283 seconds, an increase of 44% in elapsed time compared to the normal execution time. If we intentionally killed 200 out of 1746 worker processes, the entire computation finishes in 933 seconds, just an increase of 5% over the normal execution time. These performances have showed that MapReduce is an efficient programming model.

In summary, MapReduce is a cluster-based computing platform, a computing framework that simplifies distributed programming, and a programming model that abstracts distributed computing into Map and Reduce section. As far as I concerned, MapReduce provide a simple but powerful programming pattern and runtime environment, it is easy for programmers to implement, it also has high scalability and a strong fault-tolerant.

MapReduce is trying to save time by spending more space, which means it may use many machines just to solve a single problem, that is a tradeoff. But the information transfer and delay between machines will affect the efficiency and effectiveness, Therefore, great scheduling and fault-tolerant facilities are very important, the former will affect the speed and quality of computation while the latter can address problems from error of transmission.

MapReduce also has disadvantages. It does not work well in real-time computation and return results in milliseconds or seconds. Its design features determine that the data source have to be static, so it does not good at stream-oriented computation, because stream-oriented computation requires dynamic input data. A simple query needs to write Map and reduce functions, which is complex and time consuming, and it may spends a couple of hours even a whole day to address large amounts of data.

Some questions in my mind:

1. Can Mapper and Reducer be processed in parallel?
2. Is each Worker only execute Map or Reduce tasks?
3. Why the intermediate results produced by the Map function are buffered in memory?
4. Why use GFS to manage the input data?
5. Is there any better model but also similar to MapReduce?
6. How to handle data skew?

Large amounts of small size document will produce large amounts of Map tasks, so combine small size documents into a big size documents can improve the running speed.