Parallel Computing: Proposal MapReduceRing

Eom Young Moon
Vicente Adolfo Bolea Sánchez

Introduction

The MapReduce is well-known parallel programming model which is getting popular over recent years. A MapReduce program only needs definitions of Map procedure and Reduce procedure, and the MapReduce framework automatically determine how much Map tasks and Reduce tasks should be deployed considering the input or output data size, and number of capable computing nodes. So MapReduce gives programmer flexible and efficient parallelism without any need for consideration of capacities or resources of the cluster in use. Because of its simplicity, flexibility and powerful programmability, MapReduce programming model is getting popular. Large set of researches have been conducted to improve the performance of MapReduce framework. One approach to improve the MapReduce framework can be improving the underlying distributed file system.

MapReduce frameworks rely on their underlying distributed file systems when they read inputs and write outputs, i.e, Hadoop uses Hadoop Distributed File System(HDFS). Because many MapReduce workloads takes and produces huge amount of data, the distributed file system takes very important role on the performance of overall framework. In this project, we introduces a MapReduce framework which uses specially designed distributed file system. The file system dynamically adjusts and manages their input, output and intermediate result data. And we believe this key factor will improve the overall performance of the MapReduce framework.

System model

MapReduce

In our implementation of MapReduce itself does not have a critical difference from other MapReduce implementation except that it uses specially designed underlying distributed file system which is equipped with distributed hash table (DHT) and data migration etc. With help of those features, the MapReduce can maintain load balance dynamically. And because all key value pairs are hashed in the DHT, the framework can skip the shuffle phase in the MapReduce execution model.

Overall, It has a single master node, and multiple slave nodes and the slave nodes are connected to the master node via network. When a job is submitted to the master node, the master determines how many map tasks and reduce tasks should be deployed. And master determines which slave node each task should be launched according to the information where the input data of each task exist. As the information of intermediate results can be referenced by the DHT, reduce tasks are launched on the slave node where the intermediate results exist without the shuffle phase.

Distributed Hash Table (DHT)

A distributed hash table (DHT) is a class of a decentralized distributed system that provides a lookup service similar to a hash table. Such kind of data structure is needed to perform the reduce operation. In our project, the master node will synchronize the DHT with the rest of the nodes. That synchronization will take place during the scheduling of the task. Similarly, whenever a migration of data occurs the DHT will register it.

Data Migration

Previous works shows how ignoring load balancing can affect the performance of the system. For that reason, we introduce a dynamic load balance policy consisting in migration of data among the neighbor nodes of each cache nodes. Such migration will take place whenever the cache of a given node is full and there is a remaining slot in one of its neighbor nodes. Later, that node will notify to the DHT server those changes. In addition, the

decision of which data should be migrate will be take place in each node, which is be able to determine independently which of its entry is the least likely to be use.

Experiments

Setup

For experiment, we are planning to use the raven cluster consisting in 40 Linux nodes. Each node has dual Quad-Core Xeon E5506 2.13 GHz CPUs, 12 GB of memory and 7000 rpm 250 GB HDD. And they are connected by gigabit switched ethernet. For comparison, we are planning to use 3 kinds of underlying distributed file systems. First one is the system we implemented, and second is Network File System(NFS), and last one is the Hadoop Distributed File System (HDFS). And we will also measure the performance gap between our framework and Hadoop, which is one of the famous MapReduce framework written in Java.

Measure

Number of migrations

Data migration implies that the system has a bad load balance which needs to be corrected. As result, we consider the number of migrations a valuable estimator which can be used to compare MRR with other systems which does not migrate data.

Single-large job latency

The key difference between existing MapReduce implementations and our implementation is the underlying distributed file system. Therefore, as mentioned above, we are planning to compare performances when using three different kinds of underlying file systems, NFS, HDFS and our system. Using each file system, we will measure the job latency for a single-large job which consumes a huge amount of input data and generates huge amount of output. Another way to evaluate our framework is to compare the performance with existing MapReduce frameworks. We choose Hadoop for the comparison and we will conduct same measurement with a single-large job in Hadoop environment as well.

Many job submission throughput

The multiple job throughput has been considered as the most important performance metric in the MapReduce frameworks. So we will also compare the overall throughput while a multiple set of jobs are submitted. The job will have various input sizes and output sizes. The throughput will be measured with three different underlying distributed file system as above. And also, same measurement will be conducted on Hadoop environment to evaluate our framework.

Expectations

Better Latency

Using the newly designed distributed file system, we expect to get better latency because we can skip the shuffle phase and directly launch reduce tasks after map tasks are finished. The HDFS generates replicas of each data to have more balanced access to target data throughout entire cluster and to prepare for the recovery from some failures. Although it have advantages that it can guarantee some load balance, HDFS will have some overhead to have redundant unnecessary replicas. As we do not consider the node failure in this measurement, HDFS is expected to show performance degradation from the overhead.

Better throughput

We expect similar results above on the overall throughput. With multiple concurrent job, shuffle phase can cause a network congestion throughout the cluster. So Hadoop may have a limited scalability compared to our framework, expecting that our implementation will perform better with larger number of concurrent jobs.

References

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[2] Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. M3r: increased performance for in-memory hadoop jobs. *Proceedings of the VLDB Endowment*, 5(12):1736–1747, 2012.

[3] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.