A Study on Workload Imbalance Issues in Data Intensive Distributed Computing

Sven Groot[1], Kazuo Goda[1], and Masaru Kitsuregawa[1]

University of Tokyo, 4-6-1 Komaba, Meguro-ku, Tokyo 153-8505, Japan

Abstract. In recent years, several frameworks have been developed for processing very large quantities of data on large clusters of commodity PCs. These frameworks have focused on fault-tolerance and scalability. However, when using heterogeneous environments these systems do not offer optimal workload balancing. In this paper we present Jumbo, a distributed computation platform designed to explore possible solutions to this issue.

1 Introduction

Over the past decade, the volume of data processed by companies and research institutions has grown explosively; it is not uncommon for data processing jobs to process terabytes or petabytes at a time. There has also been a growing tendency to use large clusters of commodity PCs, rather than large dedicated servers. Traditional parallel database solutions do not offer the scalability and fault-tolerance required to run on such a large system.

As a result, several frameworks have been developed for the creation of customized distributed data processing solutions, the most widely known of which is Google's MapReduce [1], which provides a programming model based on the map and reduce operations used in functional programming, as well as an execution environment using Google File System [2] for storage. Hadoop [3] is a well-known open-source implementation of GFS and MapReduce.

Microsoft Dryad [4] is an alternative solution which offers a much more flexible programming model, representing jobs as a directed acyclic graph of vertex programs. This extra flexibility can however make it more difficult to effeciently parallelize complex job graphs.

Workload balancing is an important aspect of distributed computing. However, MapReduce does not provide adequate load balancing features in many scenarios, and the nature of the MapReduce model makes it unsuited to do so in some cases. Microsoft has to our knowledge not published any data on how Dryad behaves in a heterogeneous environment, thus it is unfortunately not possible for us to provide a comparison to it.

In the following sections, we will outline the issues with workload balancing in the MapReduce model and introduce Jumbo, our distributed computation platform which is designed to further investigate and ultimately solve these issues.

2 Workload Imbalance Issues in MapReduce

A MapReduce job consists of a map phase and a reduce phase. For the map phase, the input data is split and each piece is processed by a map task. Parallelism is achieved by running multiple map tasks at once in the cluster. Typically, there are far more map tasks in the job than there are nodes in the cluster, which means the map phase is well suited for load balancing. Faster nodes in the cluster take less time on the individual tasks, and therefore run more of them.

Data from the map phase is partitioned and distributed over several reduce tasks. In contrast to the map phase, the number of reduce tasks typically equals the capacity of the cluster. This means that if some nodes finish early, there are no additional reduce tasks for them to process. While it is possible to use more reduce tasks, this means that some tasks will not be started until the others complete. These tasks cannot do any of their processing in the background while the map tasks are still running, so doing this will typically reduce performance and is therefore not desirable.

Hadoop provides a mechanism for load balancing called speculative execution. Long-running map or reduce tasks will be started more than once on additional nodes, in the hope that those nodes can complete the task faster. This strategy works in some cases, but it is not optimal. Speculative execution will discard the work done by one of the two task instances, and the extra load caused by the additional instance - particularly in the case of a reduce task, which will need to retrieve all relevant intermediate data, causing additional disk and network overhead - can in some cases delay the job even further.

A further problem occurs with MapReduce's inflexible programming model. Many more complicated data processing jobs will need more than one MapReduce phase, done in sequence. For example, the frequent item set mining algorithm proposed in [5] consists of three consecutive MapReduce jobs. In this case, it is not possible to start consecutive jobs until the preceding job has finished completely. Even in cases where the next job could already have done some work with partial data, this is not possible. Any load balancing mechanisms available to MapReduce can only consider the tasks of one of the consecutive jobs at a time, rather than the whole algorithm.

3 Jumbo

In order to evaluate workload balancing and other issues in data intensive distributed computing, we have developed Jumbo, a data processing environment that allows us to investigate these issues. We have decided to develop our own solution, rather than building on Hadoop's existing open-source foundation, because some of the issues with the current MapReduce-based solutions are fundamental to the underlying model.

Jumbo consists of two primary components. The first of these is the Jumbo Distributed File System, which provides data storage. Jumbo DFS is very similar to GFS and Hadoop's HDFS in design.

Jumbo DFS uses a single name server to store the file system name space, which is kept in memory and persisted by using a log file and periodic checkpoints. Files are divided into large blocks, typically 64 or 128MB, which are stored on data servers. Each block is replicated to multiple servers, typically three, and the replicas are placed in a rack-aware manner for improved fault-tolerance.

The second component is Jumbo Jet, the data processing environment for Jumbo, providing a programming model as well as an execution environment.

Jumbo Jet represents jobs as a sequence of stages. The first stage reads data from the DFS, while each consecutive stage reads data from one or more preceding stages. Intermediate data from each of the stages is stored on disk to improve fault-tolerance. The final stage writes its output to the DFS. This sequence of stages forms a directed acyclic graph.

The stages are divided up into one or more tasks, each performing the same operation but on a different part of the data. The tasks in a stage can be executed in parallel.

In order to divide DFS input data across multiple tasks, the input data is simply split into pieces, typically using DFS blocks as a unit. Since each task reads data from a single block, the task scheduler can attempt to schedule that task to run on a node that has a local replica of that block, reducing network load.

When a stage reads input from another stage, the data from that input stage is partitioned by using a partitioning function. Every task in the input stage creates the same partitions, and each task in the stage reading that data will read all the pieces of just one partition from all the tasks in the input stage. Unlike in MapReduce, it is not required for the intermediate data to be sorted. Jumbo allows full flexibility in specifying how the data from each input task is processed. While you can perform a merge-sort like MapReduce does, you can also just process each piece in sequence, or process records in a round-robin fashion, or write a custom input processor that uses whatever method is required.

This design has many obvious similarities to MapReduce. Indeed, it is trivial to emulate MapReduce using this framework by creating a job with two stages, the first performing a map operation, the second a reduce operation, and sorting the intermediate data. However, Jumbo is not limited to this, and can more easily represent a larger variety of jobs.

Job scheduling is handled by a single job server, which performs a role similar to the JobTracker in Hadoop. Each server in the cluster will run a task server which receives tasks to execute from the job server. The job server also keeps track of failures, and reschedules failed tasks.

Currently, Jumbo does not yet contain any load balancing features beyond what Hadoop provides. However, Jumbo's more flexible design means that it will be much better suited for future experiments with load balancing than what we would be able to do with Hadoop. Jumbo's design allows us to implement complex algorithms such as the PFP algorithm from [5] as a single job so task scheduling decisions for load balancing can consider the entire job structure rather than just a part. It will also be possible to restructure jobs in different ways, besides the MapReduce structure, if this proves beneficial to distributing the workload.

Although we expect our solutions will also be applicable to other systems, using our own ensures we can fully control the design and implementation.

4 Example

In order to demonstrate the issue of workload balancing, we have run a simple experiment using both Hadoop and Jumbo. For this experiment we have used the GraySort (previously known as TeraSort) benchmark included with Hadoop, and created an equivalent job in Jumbo. Jumbo uses a sorting strategy that is very close to that of MapReduce. The job consists of two stages. The first stage partitions the input into N pieces (where N is the number of tasks in the second stage), and sorts each partition. The second stage performs a merge operation on all the input files for each partition from the first stage, and writes the result to the DFS.

Unfortunately we were not able to evaluate the behaviour of Microsoft Dryad in this experiment, as Dryad cannot run on our cluster.

The sort operation was executed on an increasing number of nodes, each time increasing the total amount of data so that the amount of data per node stays the same, 4GB per node. The number of reduce tasks (or in Jumbo, the number of second stage tasks) is also increased with the number of nodes. This means that ideally, the execution time should stay identical on a homogeneous cluster.

Two sets of nodes were used for this experiment: 40 older nodes, with 2 CPUs, 4GB RAM and one disk, and 16 newer nodes with 8 CPUs, 32GB RAM and two disks. At first we ran the job on only the older nodes. Once we were using all 40 older nodes, we added the 16 newer nodes.

Figure 1 shows the results of this. Neither Jumbo nor Hadoop quite achieve linear scalability for the first 40 nodes, as the execution time drops slightly. This is mainly because of the increasing number of map tasks or first stage tasks, which increases the number of network transfers and also affects the merge strategy to use. We are continuously reducing this overhead and improving the scalability of Jumbo.

It can also be seen that Jumbo is considerably faster than Hadoop. This difference is caused by some inefficient implementation choices in Hadoop, causing Hadoop to waste I/O operations which is very expensive, especially on the older nodes with just one disk.

However, the interesting part happens when going from 40 to 56 nodes. The final 16 nodes are much faster than the rest, which should lead to an overall improvement in performance. Although the execution time does drop, it doesn't drop as far as expected. Using Jumbo, the 40 old nodes alone take 616 seconds to sort 160GB, a total throughput of 273MB/s. We also executed the sort using only the 16 new nodes, which sorted 64GB in 223 seconds at 293MB/s. This means the total throughput for the 56 node cluster should be 566MB/s, which

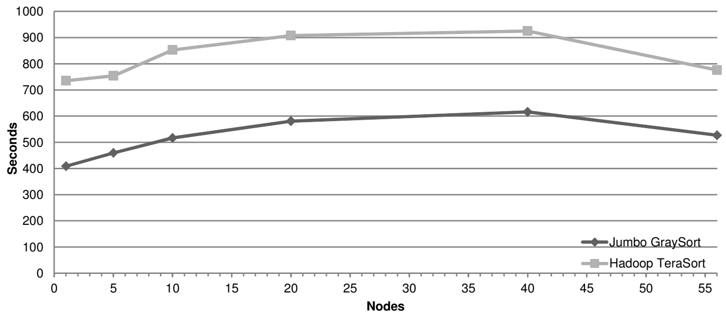

Fig. 1. Sorting performance of Hadoop and Jumbo. Up to 40 nodes, all nodes used are identical; only in the 56 nodes case was a heterogeneous environment used.

means that sorting 224GB on those 56 nodes should take 405 seconds, rather than the 527 seconds observed. In practice, 405 seconds is probably not realistic due to some additional overhead, but there is clearly room for improvement.

As indicated in Sect. 2, the issue in this particular scenario lies with the reduce phase, or in the case of Jumbo the second stage, of the job. For both Hadoop and Jumbo, the first stage finishes considerably faster with 56 nodes, because that stage consists of a very large number of tasks, 1792 in total, and the faster nodes are able to process more of them.

However, the reduce phase, or second stage in Jumbo, is where the bulk of the work is done. The I/O intensive merge operation takes up most of the job's total execution time, and because the number of tasks here equals the number of nodes in the cluster they cannot be balanced in any way.

This phenomenon can be clearly seen in Fig. 2, which shows the execution times for each node in the cluster. The 16 faster nodes finish their work considerably earlier than the 40 slower nodes. Because there are no additional tasks in the second stage, there is no way for Jumbo to assign additional work to those nodes after they finish.

One apparently obvious solution is to simply assign a larger amount of records to the partitions on the faster nodes, but this requires a-priori knowledge of how to divide the records. It also requires knowing which partition will be processed by which node, and since failures may cause any task to be reassigned to a different node this is also not a desirable scenario.

It should be noted that even amongst the identical nodes, various external circumstances cause these nodes to also have varied execution times. Ideally, a load balancing solution would be able to reduce this effect as well.

5 Conclusion

We have given an overview of the workload balancing issues in data intensive distributed computing, particularly when using the MapReduce model. We have also introduced Jumbo, our own data processing system.

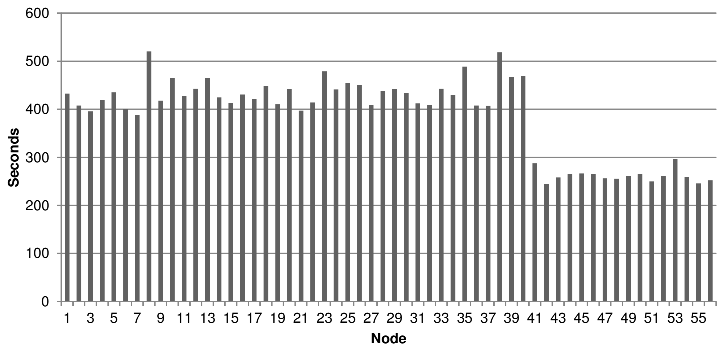

Fig. 2. Execution times of individual nodes in the cluster for Jumbo GraySort. Nodes 1-40 are the older, slower nodes, while 41-56 are the faster ones.

Improper workload balancing leads to a considerable waste of resources, with some nodes sitting idle while others are still working. Flexible methods to dynamically redistribute the workload will be required to solve this. However, naive methods of doing this such as Hadoop's speculative execution cause too much overhead, negating their potential benefits.

For our future work we intend to use Jumbo as a platform to develop and evaluate different methods for workload balancing so we can more fully utilize the resources available in a heterogeneous cluster.

References

1. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, Berkeley, CA, USA, USENIX Association (2004) 10
2. Ghemawat, S., Gobioff, H., Leung, S.T.: The google file system. In: SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, New York, NY, USA, ACM Press (2003) 29–43
3. Apache: Hadoop core. http://hadoop.apache.org/core
4. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. SIGOPS Oper. Syst. Rev. **41**(3) (June 2007) 59–72
5. Li, H., Wang, Y., Zhang, D., Zhang, M., Chang, E.Y.: Pfp: parallel fp-growth for query recommendation. In: RecSys '08: Proceedings of the 2008 ACM conference on Recommender systems, New York, NY, USA, ACM (2008) 107–114