Optimum Checkpoint Interval for MapReduce Fault-Tolerance

Naychi Nway Nway, Julia Myint
University of Information Technology, Yangon, Myanmar
naychinwaynway@uit.edu.mm,juliamyint@uit.edu.mm

Abstract

MapReduce is the efficient framework for parallel processing of distributed big data in cluster environment. In such a cluster, task failures can impact on performance of applications. Although MapReduce automatically reschedules the failed tasks, it takes long completion time because it starts from scratch. The checkpointing mechanism is the valuable technique to avoid re-execution of failed tasks in MapReduce. However, defining incorrect checkpoint interval can still decrease the performance of MapReduce applications and job completion time. In this paper, the optimum checkpoint interval is proposed to reduce MapReduce job completion time when failures occur. The proposed system defines checkpoint interval that is based on five parameters: expected job completion time without checkpointing, checkpoint overhead time, rework time, down time and restart time. Therefore, because of proposed checkpoint interval, MapReduce does not need to re-execute the failed tasks, so it reduces job completion time when failures occur. The proposed system reduces job completion time even though the number of failures increases and the performance of this system can be improved 4 times better than the original MapReduce.

Keywords- MapReduce, big data, task failures, completion time, checkpoint interval

1. Introduction

Data-intensive applications process vast amounts of data with special-purpose programs. Even though the computations behind these applications are conceptually simple, the size of input datasets requires them to be run over thousands of computing nodes. For this, Google developed the MapReduce framework, which allows non-expert users to run complex tasks easily over very large datasets on large clusters. The large datasets are often messy, containing data inconsistencies and missing value (bad records). This may, in turn, cause a task or even an application to crash. Google reports 5 average worker deaths per MapReduce job in March 2006 [8], and at least one disk failure in every run of a 6 hour MapReduce job with 4,000 machines [16].

The impact of task failures can be considerable in terms of performance [7]. In MapReduce process, after map stages the intermediate data is produced and it is the input for reduce stages. So, intermediate data is important to be successful MapReduce process. Although MapReduce can restart the process and produce intermediate data again when task failures occur, it can prolong job completion time.

Fault-tolerance is, in fact, an important aspect in large clusters because the probabilities of task failures increase with the growing of computing nodes. It allows a computation in progress in spite of having individual failures in system. Checkpoint saves the system state in stable storage so it can reduce the amount of lost computation. The performance of defining correct checkpoint interval can reduce job completion time when failures occur.

Therefore, in this paper, checkpoint-based fault-tolerance with optimum checkpoint interval is proposed to reduce the job completion time when task failures occur in Hadoop MapReduce. The proposed system addresses the surveys of related work in Section 2. Section 3 describes the basic flow and built-in fault-tolerance of MapReduce. The checkpoint interval and implementation of proposed system are described in Section 4 and 5. Section 6 proposes the experimental results and finally, the conclusion of this paper is presented in Section 7.

2. Related Work

MapReduce [1] is a parallel programming model which is originally proposed by Google in 2004 to deal with the rapidly increasing demand of processing mass data concurrently. Through well-defined interfaces and runtime support library, MapReduce can automatically perform the large-scale computing tasks in parallel, hide the underlying implementation details, and reduce the difficulty of parallel programming, which makes MapReduce become one of the most widely used parallel programming models in the concurrent processing vast amount of data. MapReduce considers task and worker failures as characteristic rather than exception. As a result, it comes with fault tolerance strategies. However, applications can experience significant performance downgrade in case of failures. According to a recent study [11], a single failure on a Hadoop job could cause a 50% increase in completion time.

RAFTing MapReduce presented in [9] tries to create several kinds of checkpoint to handle different failures. RAFT-LC is a local checkpointing algorithm that allows a

ISBN 978-99971-0-381-9 © 2017 ICAIT

map task to store progress metadata on local disk and later restore based on this in case of failures. RAFTing mappers push data to reducers instead of the opposite way and make the intermediate data replicated without bringing much overhead.

To prevent task failures in MapReduce, CROFT [13] proposed a checkpoint and replication oriented fault tolerant scheduling algorithm, which uses a checkpoint based active replication method. It also creates a local checkpoint file which is responsible for recording the progress of the current task and a global index file which is responsible for recording the characteristics of the current execution.

In paper [14], the author introduced two checkpoint algorithms to eliminate the costs of re-reading, re-copying, and re-computing the partial processed data. It makes an input checkpoint to record the location of unprocessed input data, while the output checkpoint consists of spilled files and their index information. Young proposed a first-order model that defines the optimal checkpoint interval in terms of checkpoint overhead and mean time to interrupt (MTTI). Young's model does not consider failures during checkpointing and recovery [12].

Given the checkpointing parameters such as checkpoint latency and MTTI, Daly's model [3] provides a method for computing the optimal checkpoint which is associated with the optimal execution time. The choice of a checkpoint interval influences the number of checkpoint operations performed during an application's execution. Checkpoints are created when the progress reaches 0.5 (or) 0.25 by calculation progress rate and estimated task execution time [2]. When the checkpoints are created in 50% of execution time, the failed tasks before 50% cannot be recovered. The checkpointing mechanism for 25% of progress score can cause the network traffic.

To ensure that checkpoints can be used effectively, the proposed system introduces optimum checkpoint interval that aims to recover from task failures and to improve performance as the main goal. Unlike original MapReduce, the proposed system reschedules the failed tasks without starting again. The experiments show that the proposed system outperforms original MapReduce with a 20% increasing of performance.

3. The MapReduce Framework

3.1. Execution Flow of MapReduce

MapReduce [5] adopted a two-stage and shared-nothing design. In the first stage, the map stage, MapReduce takes a list of key value pairs as input, and applies a map function on each of the pair to generate arbitrary number of intermediate key value pairs. In the second stage, all the intermediate values associated with the same keys are grouped together as a list, and a reduce function takes each of the groups as input to generate another arbitrary number of final output key value pairs. The paradigm behind MapReduce is a quite simple behavior because a map or reduce function call on a key value pair shall depend neither on other pairs nor on the processing order. This makes it easy to split the whole job into smaller independent subtasks that can run in parallel.

The input data files of MapReduce are usually stored on a DFS (distributed file system) such as HDFS, an open-source implementation of GFS. The data files are split into small pieces logically, every one of which will be fed to a map task. Map tasks, also known as mappers, parse raw input data splits into k1 v1 pairs, and invoke the map function on every single pair, the generated k2 and v2 pairs are written to a memory buffer. When the buffer verges to overflow, the mapper flushes it to a local disk file, which is called a spill. A mapper may create several spill files, however, it will merge the spill files into a single output file on local disk after all input records are processed. There are usually several reduce tasks, or reducers, key value pairs with the same key hash value go to the same reducer. As a result, the single map output file shall be logically spilt into R parts, each part will be fed to a reducer. A reduce task can be summarized to 3 main phases: shuffle, sort and reduce. During the shuffle phase, reducers copy outputs from each mapper, and merge the outputs into less amount of files in the sort phase. The shuffle phase and sort phase often overlap in practice, but the reduce phase shall not start until shuffle phase finishes, which is limited by the MapReduce semantics. The execution flow of MapReduce is shown in Figure. 1.

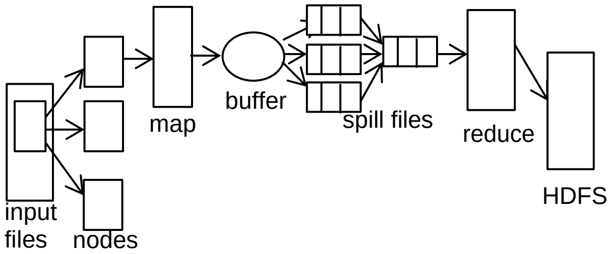

Figure 1. Execution Flow of MapReduce

3.2. Fault-Tolerance in MapReduce

Hadoop has been built with some level of faults tolerance [10]. MapReduce adopted a centralized design, an instance of Hadoop MapReduce deployment basically consists of a master and several slaves [4]. The master keeps several data structures, like the state and the identity of the worker machines [15]. Slaves execute the task on master's request, and each execution of a task is called a task attempt. A task attempt periodically informs the master about its latest status information [5]. Once the master receives status report from a task attempt indicating failure, or a task attempt fails to contact the master for a certain amount of time, the task attempt is considered to have failed and the master will schedule another attempt for the same task. The new attempt will recompute the whole input split of the task regardless of the progress of last attempt. Task attempt failures may result from bad records, such as invalid or inconsistent field values, which is common in big data analysis. In the worst case, the last record of an input split is corrupted and it will result in a second task attempt processing the exact same input and doubles the task execution time at least. In Hadoop, the bad record will be skipped in a third attempt, and apparently the delay caused by the single bad record is too high and not tolerable.

While checkpointing is one of the most widely used techniques in fault tolerance [12], a naïve implementation of checkpointing in Hadoop may downgrade the performance. Due to the fact that a MapReduce job often processes vast amount of input data, the intermediate data generated is usually also very large. Checkpointing requires the intermediate data to be replicated among several nodes, which involves huge amount of disk IO and network IO, the two most critical resources in MapReduce. Checkpointing strategy in MapReduce needs to be carefully designed.

4. Checkpoint Interval

A checkpoint interval [3] is defined as the duration between the establishments of two consecutive checkpoints. That is, an interval begins when one checkpoint is established, the interval ends when the next checkpoint is established. Figure 2 shows how to define checkpoint interval and T is the amount of useful computation in each interval, C is checkpoint overhead and L means the duration of time needed to save the checkpoint [6].

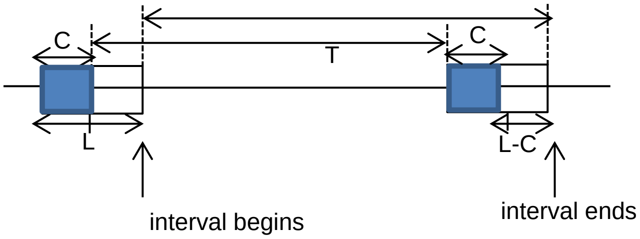

Figure 2. Checkpoint Interval

5. Proposed System Design

The proposed system aims to minimize job completion time due to failures in MapReduce by determining checkpoint interval that is based on task failures. Before calculating checkpoint interval, the system calculates the expected job completion time [5] without checkpoint using equation (1)

$$Tc = \left(\frac{Tn}{w}\right) * \left(Jt + \frac{Dsize}{Jp}\right) \tag{1}$$

where Tc means job completion time, Tn means the number of tasks, w means the number of workers, Jt means time to take JVM, $Dsize$ means input data size and Jp means processing size of JVM per second.

After that, based on job completion time, the system calculates interval between checkpoint files that minimizes the time lost when failures occur using equation (2)

T = Completion Time + Overhead Time + Rework Time
+ Down Time + Restart Time (2)

Completion Time is defined as actual completion time without checkpoints. Overhead Time is overhead for writing checkpoint files, Rework Time is time lost due to failures, Down Time is time lost when an application cannot reach current running state and Restart Time is time required before an application resumes to current work. Completion Time will be Tc and Overhead Time will be $\beta(C(\tau) - 1$ where $C(\tau)$ is number of checkpoint taken and one is subtracted because there is no need to write checkpoint files in last segment. For Rework Time, it will be described by $\frac{1}{2}(\tau + \beta)N(\tau)$ where $N(\tau)$ is expected number of interrupts. Down Time is used as $DN(\tau)$ and finally, Restart Time is $RN(\tau)$, the amount of time required to restart times total number of failures. So, the system constructs the formula as equation (3)

$$T = Tc + (C(\tau) - 1)\beta + \tfrac{1}{2}(\tau + \beta)N(\tau) + DN(\tau) + RN(\tau) \tag{3}$$

Next, system determines the number of interrupts $N(\tau)$ and numbers of checkpoints are calculated by dividing completion time by checkpoint interval. The expected number of interrupts can be calculated by the product of numbers of checkpoints required to complete calculation and the probability of each segment failing as in equation (4)

$$N(\tau) = \frac{Tc}{\tau}\left(e^{\frac{\tau+\beta}{M}} - 1\right) \cong \frac{Tc}{\tau}\left(\frac{\tau + \beta}{M}\right) \tag{4}$$

Then, $N(\tau)$ is substituted in equation 3:

$$T = Tc + \left(\frac{Tc}{\tau} - 1\right)\beta + \left[\frac{1}{2}(\tau + \beta) + D + R\right]\frac{Tc}{\tau}\left(\frac{\tau + \beta}{M}\right) \tag{5}$$

Using equation 5, the system finds the minima with respect to τ that sets the derivation to zero.

$$e^{\frac{\tau+\beta}{M}}[\tau^2 + (\beta + 2R + 2D)\tau - (\beta + 2R +)M] + 2RM - \beta M = 0 \tag{6}$$

Instead of expanding the exponential term, recast equation 6 as follows:

$$\frac{\tau + \beta}{M} = ln\left[\frac{(\beta - 2R)M}{\tau^2 + (\beta + 2R + 2D)\tau - (\beta + 2R + 2D)M}\right] = \ln[g(\tau)] \tag{7}$$

The system which calculates a Taylor series expansion for natural logarithm of $g(\tau)$ is as follows:

$$\frac{\tau+\beta}{M} = \frac{g(\tau)-1}{g(\tau)} + \frac{1}{2}\left(\frac{g(\tau)-1}{g(\tau)}\right)^2 + \frac{1}{3}\left(\frac{g(\tau)-1}{g(\tau)}\right)^3 + \cdots = \left(1-\frac{1}{g(\tau)}\right) + \frac{1}{2}\left(1-\frac{1}{g(\tau)}\right)^2 + \frac{1}{3}\left(1-\frac{1}{g(\tau)}\right)^3 + \cdots \quad (8)$$

Reduce the equation 8 to quadratic form as in (9)

$$\tau^2 + 2D\tau + (\beta^2 - 2\beta(R+M) - 2DM) = 0 \quad (9)$$

Finally, the value of τ which minimize equation 5 as follows:

$$\tau = -\beta + \sqrt{2\beta(R+M) + 2DM} \quad (10)$$

The proposed system defines checkpoint interval (τ) after processing 50 seconds. After calculating checkpoint interval, the system creates a checkpoint file in local disk with three checkpoint information: taskID, a unique task identifier and offset that specify the last byte of input data processed by map tasks.

6. Experiment

We analyze the performance of the proposed system in this section. Experiments are designed to measure the job completion time in the case of task failures. The implementation of the proposed system is based on Hadoop 2.7.1, Java 1.8 and Hadoop Distributed File System (HDFS) with data size of 400MB, 500MB and 600MB. The jobs for experiments are word count over user-submitted comments on StackOverflow.

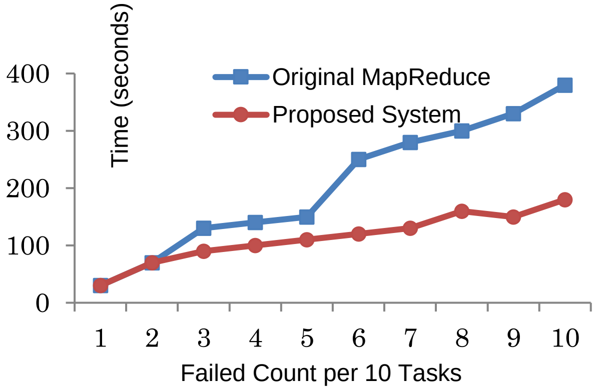

Figure 3. Comparison of Completion Time of 10 Tasks with Task Failure

Figure 3 shows the comparison of MapReduce job completion time between original MapReduce and proposed system with 400MB. The x-axis is the number of task errors per 10 tasks and y-axis is the total completion time. According to the experiment, if a number of errors increase, the completion time of the job will take 4 times less than the original Hadoop. When failures occur, the proposed system reads checkpoint files more frequently so it saves job completion time. The experiment of Figure 4 with 500MB and Figure 5 with 600MB also show that the performance of proposed system is better than original MapReduce when the number of failures increases.

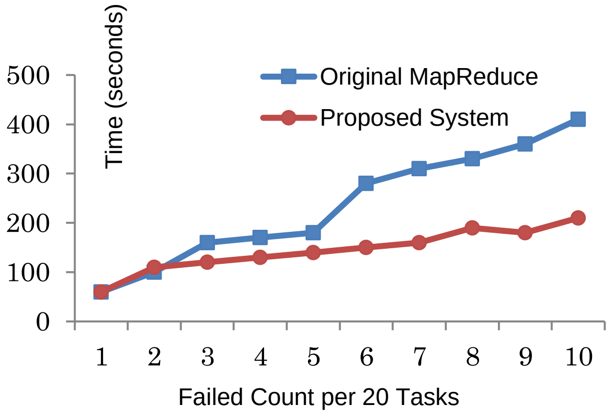

Figure 4. Comparison of Completion Time of 20 Tasks with Task Failure

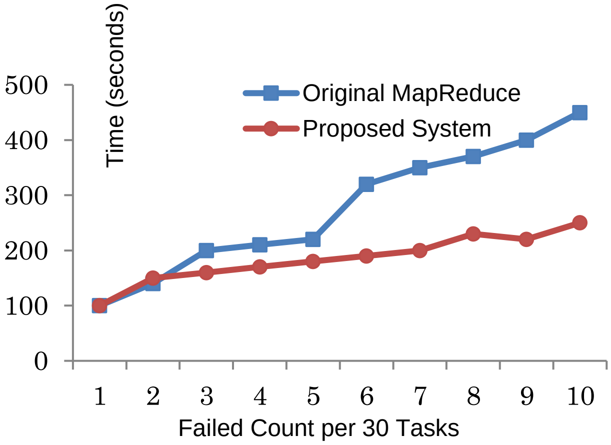

Figure 5. Comparison of Completion Time of 30 Tasks with Task Failure

7. Conclusion

MapReduce is a popular programming model that allows the user with simple APIs and is able to run big data applications. MapReduce is also able to retry the failure tasks but it performs poorly because of start from scratch. Although the original MapReduce facilitates fault-tolerance with re-executing of failed tasks, it can prolong job completion time when failures occur. The proposed system presents checkpointing mechanism not to re-execute failed tasks from start. In order not to delay long job completion because of checkpointing, the proposed system defines optimum checkpoint interval that has the advantageous of reducing job completion time when failures occur.

As future direction, we intend to propose a task migration technique for slow tasks in MapReduce. The main causes of slow tasks in MapReduce are (i) a slow node and (ii) input data skew. Slow tasks in MapReduce also threaten the job completion so we will combine checkpointing and task migration techniques to solve the problem of slow tasks in MapReduce.

8. References

[1] B.Cho, and I.Gupta, "Making cloud intermediate data fault-tolerant", ACM symposium on Cloud Computing,2010.

[2] C.Lin, T.Chen, and Y. Cheng. "On Improving Fault Tolerance for Heterogeneous Hadoop MapReduce Clusters", IEEE International Conference on Cloud Computing and Big Data, 2014.

[3] D.John."Future Generation Computer Systems", Volume 22, Issue 3, February 2006, pp. 303-312.

[4] H.Wang, H.Chen, and F.Hu. "ReCT: Improving MapReduce Performance under Failures with Resilient Checkpoint Tactics", IEEE International Conference on Big Data,2014.

[5] H.Wang, H.Chen, and F.Hu, "BeTL: MapReduce Checkpoint Tactics Beneath the Task Level", IEEE Transactions on Services Computing,2016.

[6] H.Nitin, "On Checkpoint Latency", Technical Report,1995.

[7] J.Dean and S Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", In 6th symposium on Operating System Design and Implementation (OSDI), San Francisco, December 2004.

[8] J.Dean, "Experiences with MapReduce: an Abstraction for Large-Scale Computation", In Keynote I: PACT 2006.

[9] J.Quiane Ruiz, C. Pinkel, J. Schad, and J. Dittrich, "RAFTing MapReduce :Fast Recovery on the RAFT", IEEE International Conference on Data Engineering, 2011.

[10] P. Costa, M. Pasin, "Byzantine Fault-Tolerant MapReduce: Faults are Not Just Crashes", IEEE International Conference on Cloud Computing Technology and Science, 2011.

[11] Q.Zheng. "Improving MapReduce Fault Tolerance in the Cloud", IEEE International Symposium on Parallel & Distributed Processsing and Phd Forum(IPDPSW), 2010.

[12] W. Yong, "A first order approximation to the optimum checkpoint interval", Communication of the ACM, 1974.

[13] W. Wei, Y. Liu, and Y. Zhang, " Checkpoint and Replication Oriented Fault Tolerant Mechanism for MapReduce", IEEE International Conference on Data Engineering .,2011.

[14] Y.Wang, W.Lu, R.Lou, B. Wei, "Journal of Grid Computing",Volume 13,Issue 4, December 2015, pp. 587-604.

[15] M.Bunjamin, I.Shadi, P. Maria, A. Gabriel, "Resource Management for Big Data Platforms", Springer, 2016.

[16] Sorting 1PB with MapReduce: http:// googleblog. blogspot.com/2008/11/sorting-1pb-with-mapreduce.html.