2018 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery

Evaluating Checkpoint Interval for Fault-Tolerance in MapReduce

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

Julia Myint, Ei Chaw Htoon
University of Information Technology
Yangon, Myanmar
juliamyint@uit.edu.mm,eichawhtoon@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 finished tasks in MapReduce. However, defining incorrect checkpoint interval can still decrease the performance of MapReduce applications and job completion time. So, in this paper, checkpoint interval is proposed to avoid re-execution of whole tasks in case of task failures and save job completion time. The proposed checkpoint interval is based on five parameters: expected job completion time without checkpointing, checkpoint overhead time, rework time, down time and restart time. The experiments show that the proposed checkpoint interval takes the advantage of less checkpoints overhead and reduce completion time at failure time.**

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

I. 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, and at least one disk failure in every run of a 6-hour MapReduce job with 4,000 machines [10]. It points out that MapReduce has a performance problem while failures occur.

The impact of task failures can be considerable in terms of performance [5]. 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 a 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 computation in progress in spite of having individual failures in system. Generally, there are two types of fault tolerance techniques that are replication and checkpoint. As in replication, this needs secondary node to keep standing by until the primary node fails over so the resource will be doubled to meet the requirement of fault tolerance. Thus, replication is not a good solution for the data-intensive applications. Another solution, checkpoint, saves the system state in stable storage so it can reduce the amount of lost computation. However, while checkpointing is one of the most widely used techniques in fault tolerance, 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 generated intermediate data 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. So, checkpointing strategy in MapReduce needs to be carefully designed. Thus, defining interval time to take checkpoint is the best way to be effective use of checkpointing strategy in MapReduce.

Therefore, in this paper, checkpoint interval-based fault-tolerance is proposed to reduce the job completion time when task failures occur in Hadoop MapReduce. This proposed checkpoint interval is calculated before starting the process of map tasks. After defining checkpoint interval, checkpoint file is created and takes checkpoint according to proposed checkpoint interval. The proposed system evaluates the performance of job completion time based on mean time between failures, which is the expected time between two failures for a repairable system. The evaluations measure the performance of job completion time of the proposed system, original MapReduce and one of the related works. And then, the experiments show that this proposed system takes little overhead because of checkpointing strategy.

The paper is structured as follows: the related work of proposed system is discussed in Section II. Section III describes the basic flow and built-in fault-tolerance of MapReduce. The implementation of proposed system and checkpoint interval are described in Section IV. Section V describes the experimental results and finally, the conclusion of this paper is presented in Section VI.

II. 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

978-0-7695-6584-2/18/$31.00 ©2018 IEEE
DOI 10.1109/CyberC.2018.00046

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.

RAFTing MapReduce presented in [6] tries to create several kinds of checkpoint to handle different failures. RAFT-LC is a local checkpointing algorithm that allows a 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 [8] proposed a checkpoint and replication oriented fault tolerance 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 [9], 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. Yong proposed a first-order model that defines the optimal checkpoint interval in terms of checkpoint overhead and mean time to interrupt (MTTI). Yong's model does not consider failures during checkpointing and recovery [7].

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. Checkpoints are created when the progress reaches 0.5 (or) 0.25 by calculation progress rate and estimated task execution time [2].

In original version MapReduce [1], all of the failed tasks are re-executed again in case of task failures. As a result, the job completion time can be long because of starting the tasks from scratch. In work [2], when the checkpoints are created in 50% of execution time, the failed tasks before 50% are not recovered. To overcome the problem of previous works [1] and [2], the proposed system defines a checkpoint interval that influences the number of checkpoint operations performed during an application's execution. To ensure that checkpoints can be used effectively, the proposed system introduces 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 the performance comparison between original MapReduce, the proposed system and one of the related works [2].

III. THE MAPREDUCE FRAMEWORK

A. Execution Flow of MapReduce

MapReduce [4] 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 pairs 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 calls on a key value pair that 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 execution flow of MapReduce is shown in Fig. 1.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.

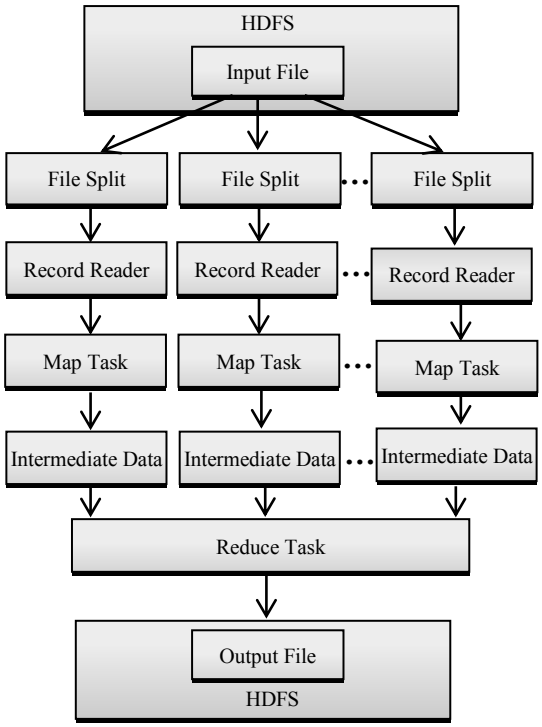

Figure 1. Execution Flow of MapReduce

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 that 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 fewer amounts of files in the sort phase. The shuffle phase and sort phase often overlap in practice, but the reduce phase shall not start until the shuffle phase finishes, which is limited by the MapReduce semantics

B. Fault-Tolerance in MapReduce

Hadoop has been built with some level of fault-tolerance. Hadoop MapReduce deployment basically consists of a master and several slaves. The master keeps several data structures, like the state and the identity of the worker machines. 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 [4]. 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. So, a checkpoint technique with formulated checkpoint interval is proposed in order to keep going after failing tasks, it can save a lot of time when failures are involved.

IV. PROPOSED SYSTEM

In this paper, a checkpointing strategy for MapReduce is proposed, which defines checkpoint interval to provide neglectable overhead for taking checkpoints and rescheduling of finished tasks as little as possible. To preserve this, Fig. 2 shows the execution flow of the proposed system.

A. The Proposed System Basics

The original MapReduce process, although a map task has its own rescheduling of failed task, reworks a task from start. So, the failed tasks in MapReduce make a job completion time long because they require finished process ranges to be executed again. The main design goal of this proposed system is to provide a checkpointing strategy by

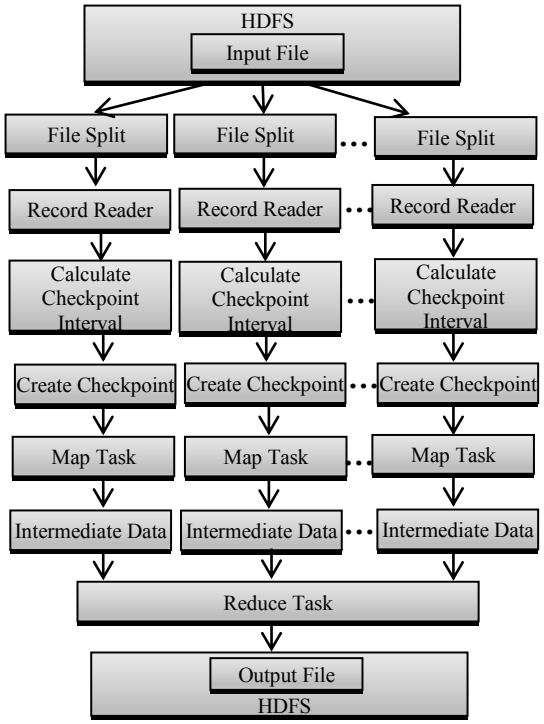

Figure 2. Proposed Execution Flow in MapReduce

permitting the tasks to checkpoint at formulated checkpoint interval.

Initially, the input file is taken from HDFS and InputFormat class is used to split the input into multiple file splits. After dividing the file, this proposed system will calculate checkpoint interval, and then, based on this interval, create the checkpoint to keep track of progress of MapReduce job. All of task progresses are saved in checkpoint file before the execution of one Mapper task. The checkpoint file is saved in local disk of the node that runs the current MapReduce process so the node can restart tasks from recent status with the help of checkpoint file when task failures occur.To calculate the proposed checkpoint interval, firstly, the system calculates the expected job completion time [4] without checkpoint using (1).

$$T_{c=} \left(\frac{T_n}{w}\right) * \left(J_t + \frac{Dsize}{J_p}\right) \quad (1)$$

where T_c means job completion time, T_n means numbers of tasks, w means number of workers, J_t means time to take JVM, $Dsize$ means input data size and J_p means processing size of JVM per second.

B. Checkpoint Interval Model

The proposed checkpoint interval is based on Daly's model [3] except downtime parameter. The proposed system adds downtime parameter because there are many map tasks in MapReduce, which are important for successful completion of a MapReduce job. So, the downtime is needed to consider as a parameter for calculating checkpoint interval. The checkpoint interval model is defined by five parameters given in Table 1.

TABLE I. CHECKPOINT INTERVAL PARAMETERS

Parameters	Description
M	Mean Time Between Failure
β	Checkpoint Overhead-time to take a checkpoint file
R	Restart Time- time required before an application resumes to current work
Rework Time	Time needed to rework job due to failures
D	Down Time-time that cannot arrive current running state in case of failures

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

$$\text{T} = \text{Completion Time} + \text{Overhead Time} + \text{Rework Time} + \text{Down Time} + \text{Restart Time} \quad (2)$$

Completion Time is defined as actual completion time without checkpoints. Completion Time will be Tc and Overhead Time will be $\beta(\text{C}(\tau) - 1)$ where $\text{C}(\tau)$ 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 $\frac{1}{2}(\tau + \beta)\text{N}(\tau)$ where $\text{N}(\tau)$ is the expected number of interrupts. Down Time is used as $\text{DN}(\tau)$ and finally, Restart Time is $\text{RN}(\tau)$, the amount of time required to restart into total number of failures. So, the system constructs the formula as (3)

$$\text{T} = T_c + (\text{C}(\tau) - 1)\beta + \frac{1}{2}(\tau + \beta)\text{N}(\tau) + \text{DN}(\tau) + \text{RN}(\tau) \quad (3)$$

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 (4)

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

Then, $\text{N}(\tau)$ is substituted in (3):

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

Using (5), the system finds the minima with respect to τ that set the derivation to zero.

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

Instead of expanding the exponential term, recast (6) as follows

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

The system which calculates a Taylor series expansion for natural logarithm of $\text{g}(\tau)$ 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 (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 (5) as follows:

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

According to the above derivation, checkpoint interval for MapReduce process can be calculated using (10). The input for checkpoint interval is checkpoint overhead, restart time, mean time between failures and down time of a MapReduce job.

V. EXPERIMENTAL SETUP AND RESULTS

To evaluate the effectiveness of this proposed system in the presence of task failures, the mean time between failures are thought of the thing. That is, define values of mean time between failures in order to consider the job completion time that is measured from performance aspect of the proposed system. Compare the checkpoint overhead aspect in the case of task failures. The implementation of the proposed system is based on Hadoop 2.7.4, Java 1.8 and Hadoop Distributed File System (HDFS) with data size of 1GB. The jobs for experiments are word count over user-submitted comments on StackOverflow. The proposed jobs contain 8 map tasks and 1 reduce task, each map task processes about 128 megabytes of data.

The job completion time performance of the proposed system is implemented by these two factors: (1) the number of checkpoint interval that depends on the number of failures (2) checkpoint overhead value, which affects the job completion time. In a MapReduce system, checkpoint intervals are taken too frequently, the checkpoint overhead causes the job completion time to last longer. If the checkpoints are taken infrequently, computation can lose when computation restarts from a checkpoint. In this proposed system, when checkpointing approach starts, checkpoint interval for a job is computed and this interval is tradeoff between failures and performance. In Fig. 3, job completion time of using downtime is less than without downtime. In Fig. 4, the experiment is based on checkpoint overhead that takes five seconds, restart time and down time, two seconds respectively. It shows that the more the number of failures occurs, the more checkpoint intervals are needed.

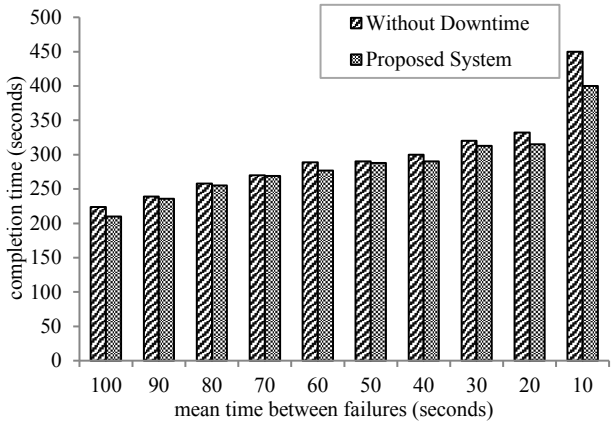

Figure 3. Comparison of completion time with downtime and without downtime

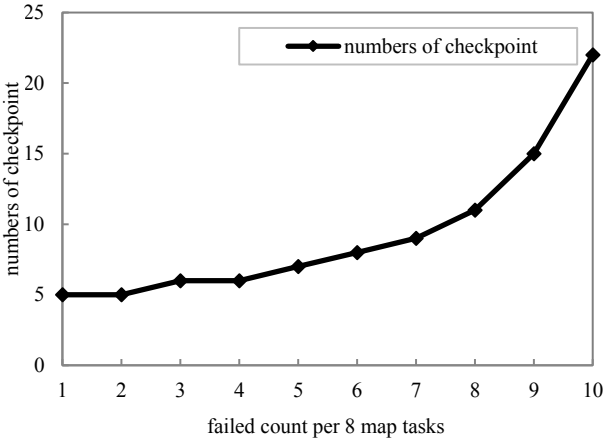

Figure 4. Relationship between numbers of failure and numbers of checkpoint

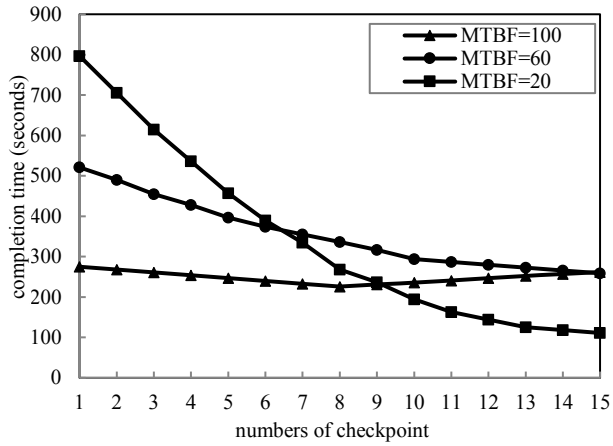

Figure 5. Job completion time versus numbers of checkpoint for MTBF=20,MTBF=60 and MTBF=100

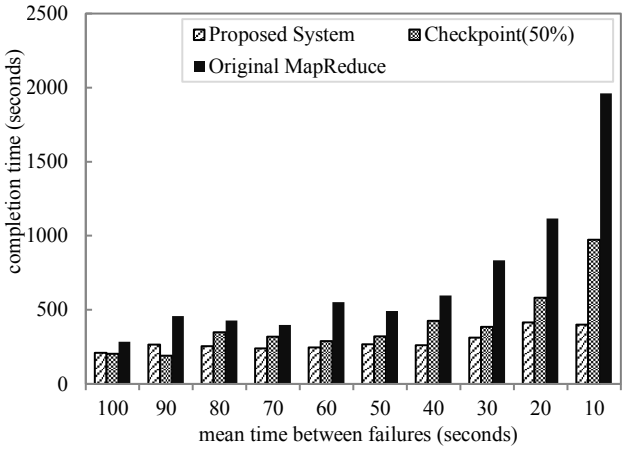

Figure 6. Comparison of completion time with checkpoint overhead=5s, restart time=2s and downtime=2s

Fig. 5 shows the relationship between completion time and numbers of checkpoint. It also introduces with three mean time between failure values: 100, 60 and 20 respectively. The value of mean time between failures 20, which means failures occur too frequently. This shows that the more the checkpoints, the less the completion time. When the failures occur less frequently, as in the value of mean time between failures 100, the completion time is slightly high because of taking too much checkpoints. So, using checkpoints are needed to take care of because these can cause job completion time to get longer. As a result, the shorter the mean time between failures, the longer the completion time and more checkpoints should be used in order to reduce recovering time as little as possible.

As shown in Fig. 6, the comparison among the proposed system, original MapReduce and one of related works whose checkpoint intervals are 50% of execution time. All of these comparisons are based on ten values of mean time between failures. According to the experiment, the job completion time of the proposed system is significantly decreased compared with other systems when the numbers of errors occur frequently. Especially, when the errors appear frequently, the local checkpoint file will be read more frequently, and as a result, the proposed system saves more job completion time.

As another comparison aspect, the checkpoint overhead aspect of proposed system will be shown. Estimated checkpoint overhead times are shown in Fig. 7. In case with MTBF=10, the overheads also increases as failures occur frequently. In case of failures, although checkpoint overhead percentage is increased, completion time of the proposed system is less than original MapReduce. So, checkpoint overhead of the proposed system is negligible in case of failure compared with original MapReduce.

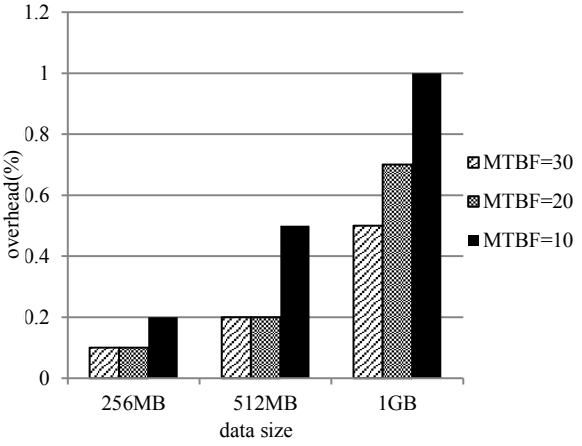

Figure 7. Checkpoint overhead versus input data size

VI. CONCLUSION

MapReduce is a popular programming model that allows the user with simple APIs and is able to run big data applications. The popularity of MapReduce is that it makes the parallelization easy and has fault-tolerance strategy. Although MapReduce is able to retry the failed tasks, it performs poorly because it re-executes all finished ranges again in case of failures. As a result, MapReduce job can prolong job completion time when failures occur.

This proposed system removes the drawback of fault-tolerance strategy in original MapReduce. The proposed system uses checkpointing strategy that saves all of the progress of finished tasks so the failed tasks are not needed to re-execute from the beginning of the process. As a result, proposed MapReduce system saves job completion time in case of failures. Checkpoint interval method is also proposed that defines which interval is the most suitable time to take checkpoints and as a result, checkpoint overhead time is little when checkpoint file is created. We implemented the proposed system on the base of Hadoop that is the most popular open source implementation of MapReduce. This proposed system outperforms Hadoop while decreasing mean time between failure values.

REFERENCES

[1] B. Cho, 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," vol. 22,Issue 3, February 2006, pp. 303–312.

[4] H.Wang, H.Chen and F.Hu, "BeTL: MapReduce checkpoint tactics beneath the task level," IEEE Transactions on Services Computing,2016.

[5] J.Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," 6th symposium on operating system design and implementation (OSDI), San Francisco, December 2004.

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

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

[8] W.Wei, Y. Liu and Y.Zhang, "Checkpoint and replication oriented fault tolerant mechanism for MapReduce," IAES Indonesian Journal of Electrical Engineering and Computer Science,2013.

[9] Y.Wang, W. Lu, R. Lou and B.Wei, "Journal of grid computing," vol.13, Issue 4, December 2015, pp. 587–604 .

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