Efficient Checkpoint Interval for Speculative Execution in MapReduce

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

Ei Chaw Htoon
University of Information Technology
Yangon, Myanmar
eichawhtoon@uit.edu.mm

Abstract

The MapReduce has become popular in big data environment due to its efficient parallel processing. However, MapReduce still has the problem from job delay caused by straggling tasks, which prolong job completion time. In MapReduce framework, although the existing speculative execution mechanism mitigate stragglers, its tasks are slower than their original tasks so this makes job completion time get long when straggling tasks occur. So, in this paper, a checkpoint mechanism is proposed in order to increase the efficiency of speculative execution of MapReduce, and not to prolong job completion time in case of straggling tasks. However, MapReduce produces too much intermediate data; as a result, checkpoint of every intermediate data can still decrease the performance of MapReduce. So, to avoid this problem, the proposed system evaluates checkpoint interval in order to reduce job completion time in case of stragglers. Then, the proposed system defines stragglers using LATE scheduler. 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. Experimental results show that the proposed system leads to less completion time, rework time and checkpoint overhead.

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

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 [6]. For this, Google developed the MapReduce framework [5], which allows non-expert users to run complex tasks easily over very large datasets on large clusters. The large datasets are often messy that causes I/O overload and contain skewed data. This may, in turn, cause a task or even an application to be long completion time. It points out that MapReduce has a performance problem while slow tasks also called stragglers occur.

The impact of stragglers can be considerable in terms of performance. In MapReduce process, after map stages, the intermediate data is produced and it is the input for reduce stages [1]. So, intermediate data is important to be a successful MapReduce process. Although MapReduce can restart the process and produce intermediate data again when slow tasks occur, it can prolong job completion time.

A few of straggler mitigation techniques have been developed and can be divided into two classes: black-listing and speculative execution [9]. Blacklisting uses a user-provided health-check script to detect the status of the slaves. If a slave is not performing properly, it can be blacklisted so that no job will be scheduled to run on it. However, a strict or incorrect health-check program will result in reduced numbers of resources. Besides, stragglers can arise on the non-blacklisted machines at times, often due to some complex reasons like I/O contentions, background services, and hardware behaviors. In speculative execution, the master schedules speculative tasks for those straggling tasks and puts them in the queue. They will be launched when there are available slots. For each original task, the scheduler also ensures that at most one speculative task is running at a time. The original task is killed if the speculative task finishes first and vice versa.

Although the original speculative execution has fault-tolerance feature, it has drawback because of re-executing tasks from start as their original tasks. It re-reads the input data, re-copies the intermediate data and re-computes the processed data so straggling tasks cause the job completion time to take longer [9].

Therefore, in this paper, checkpoint interval-based speculative execution is proposed to reduce the job completion time when straggling tasks 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 in local disk of a node and takes checkpoint according to proposed checkpoint interval. The proposed system evaluates the performance of job completion time based on mean time between slow tasks, which is the expected time between two slow tasks 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 less overhead, completion time and rework time because of proposed checkpointing strategy.

The paper is structured as follows:

the related work of proposed system is discussed in Section 2. Section 3 explains the basic flow and built-in speculative execution of MapReduce. The checkpoint interval and implementation of proposed system are described in Section 4. Section 5 describes the experimental results and finally, the conclusion of this paper is presented in Section 6.

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.

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 restores 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.

In [7], authors also proposed new scheduling algorithm in order to improve the speculative re-execution of straggling tasks in MapReduce. ESMAR differentiates historical stage weight information on each node and divides them into k clusters in order to identify straggling tasks accurately.

In paper [9], the author introduced two checkpoint algorithms to eliminate the costs of re-reading, re-copying, and re-computing the partially 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 [8].

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 of MapReduce [1], all of the straggling tasks are re-executed again in case of slow tasks. 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 25% of execution time, the speculative execution before 25% is not recovered. To overcome the problem of previous work in [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 evaluates checkpoint interval and finds stragglers using LATE scheduler that aims to recover from straggling tasks and to improve performance as the main goal. Unlike original MapReduce, the proposed system reschedules the straggling tasks without starting again. The experiments show the performance comparison among original MapReduce, the proposed system and one of the related work [2].

3. The MapReduce Framework

3.1. Execution Flow of MapReduce

MapReduce [4] adopts a two-stage and shared-nothing design. The first stage, the map stage, 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 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 that 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 that goes to the same reducer. As a result, the single map output file shall be logically spilt into 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.
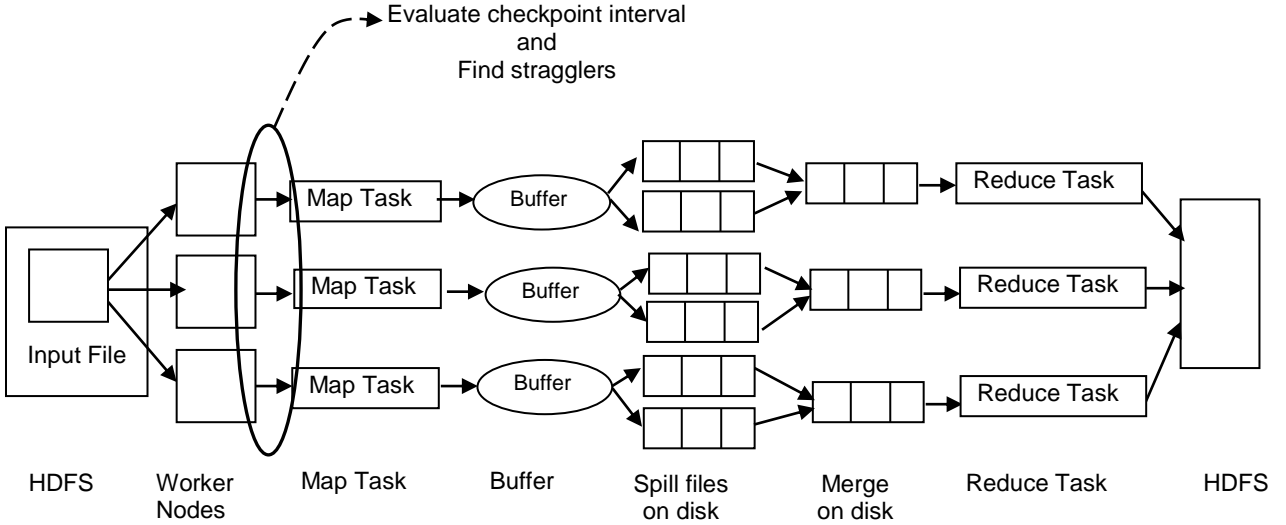
Figure 1. **Proposed system architecture**

3.2. Speculative Execution in MapReduce

Firstly, all the tasks for the jobs are launched in Hadoop MapReduce. The JobTracker monitors the progress of each task using a progress score between 0 and 1. The average progress score of each category of tasks (maps or reduces) is used as the threshold for speculative execution: if a task's progress score is less than the average minus 0.2, it is considered as a straggler. The speculative tasks are launched for those tasks that have been running for some time (at least one minute) and have not made any much progress, on average, as compared with other tasks from the job. The speculative task is killed if the original task completes before the speculative task, on the other hand, the original task is killed if the speculative task finishes before it [11]. However, speculative execution re-executes from start as their original tasks so speculative execution in MapReduce cause the job completion time to get long although it has fault-tolerance features. So, this paper uses LATE[7] which defines a task is straggling or not. After that, based on expected job completion time, the formulated checkpoint interval is proposed in order to keep going after straggler tasks. So, the proposed system can save a lot of time when straggling tasks are involved.

4. Proposed System

In this paper, a checkpointing strategy for MapReduce is proposed, which defines checkpoint interval to improve the efficiency of checkpoint in speculative execution and the job completion time. To preserve this, Figure 1 shows the architecture of the proposed system.

4.1. Expected Job Completion Time without Failure

Although original MapReduce processes with its own speculative execution for stragglers, it reworks a task from start. So, stragglers 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 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, creates 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 straggling tasks occur. To calculate the proposed checkpoint interval, firstly, the system calculates the expected job completion time [4] without checkpoint using (1)

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

where T_c means job completion time, T_n means the 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.

4.2. 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 1. Checkpoint interval parameters

Parameters	Description
M	Mean Time Between Slow Tasks
B	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 slow tasks
D	Down Time-time that cannot arrive current running state in case of slow tasks

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

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

Completion Time is defined as actual completion time without checkpoints. Completion Time will be Tc and Overhead Time will be $\beta(C(\tau)-1)$ where $C(\tau)$ is the number of checkpoint taken and one is subtracted because there is no need to write checkpoint files in the last segment. For Rework Time, it will be described by $\frac{1}{2}(\tau+\beta)N(\tau)$ where $N(\tau)$ is the expected numbers of interrupt. Down Time is used as $DN(\tau)$ and finally, Restart Time is $RN(\tau)$, the amount of time required to restart into total number of slow tasks. So, the system constructs the formula as (3)

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

Next, the system determines the numbers of interrupt $N(\tau)$ and numbers of checkpoints are calculated by dividing completion time by checkpoint interval. The expected numbers of interrupt 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, $N(\tau)$ 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 $g(\tau)$ 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 slow tasks and down time of a MapReduce job.

4.3. Speculative Execution in Proposed System

After evaluating checkpoint interval, the system checks stragglers using LATE scheduler. To select tasks for speculative re-execution, Hadoop default scheduler monitors the progress of tasks using Progress Score (PS) between 0 and 1. Suppose: a job has K number of tasks being executed; a task has a total of N number of key/value pairs to be processed and M of them have been processed successfully. Hadoop default scheduler gets PS according to (11).

$$PS = \begin{cases} \frac{M}{N} & For\ Map\ Task \\ \frac{1}{3} * \left(K + \frac{M}{N}\right) & For\ Reduce\ Task \end{cases} \quad (11)$$

18

$$PSavg = \sum_{i=1}^{K} PS[i]/K \qquad (12)$$

For task Ti: $PS[i] < PSavg\ \text{-}20\%$ (13)

Here, it is assumed that a map task spends negligible time in the order stage and a reduce task has finished K stages and each stage takes the same amount of time. If (13) is satisfied, task Ti needs a backup task. The backup task is started from the last checkpoint interval; as a result, it saves not only completion time but also rework time.

5. Experimental Results

To evaluate the effectiveness of this proposed system in the presence of straggling tasks, the mean times between slow tasks are thought of the thing. That is, defining values of mean time between slow tasks in order to consider the job completion time that is measured from performance aspect of the proposed system. Compare the checkpoint overhead aspect and rework time in the case of straggling tasks. 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.

In scenario with only slow tasks, Figure 2 shows the relationship between job completion time and numbers of checkpoint. It introduces mean time between slow tasks 20 which means slow tasks occur too frequently. This shows that when slow tasks occur frequently, the system needs to take more checkpoints in order to save completion time. To avoid making completion time long, the numbers of checkpoint should be taken carefully.

As shown in Figure 3, the comparison among the proposed system, original MapReduce and one of related works whose checkpoint intervals is 25% of execution time. In accordance with Equation 10, checkpoint intervals are calculated based on different mean time between slow tasks. According to the experiment, the proposed system takes less completion time not only in mean time between slow tasks 100 but also in mean time between slow tasks 20. As a result, the proposed checkpoint interval works efficiently in the case of slow tasks that occur in MapReduce. Although, the completion time of proposed system is slightly the same with related work, the completion time is decreased when slow tasks appear frequently.

As another comparison aspect, as in Figure 4, the experiment will show the checkpoint overhead aspect of proposed system. The values of x-axis are checkpoint intervals that are obtained by calculating Equation 10. In The checkpoint interval values are calculated based on mean time between slow tasks from 10 to 100 in seconds.

Figure 4 shows the job completion time under different values of checkpoint overhead. We compare three different checkpoint overhead times, C=5, C=3 and C=1 in seconds. For these experiments, start time and down time take 2 seconds. The experiment shows that slightly difference checkpoint overhead that is negligible for our proposed system. So, our proposed system is suitable not only checkpoint overhead in 1 second but also checkpoint overhead in 5 seconds.

Figure 5 shows the performance of proposed system based on rework time. It is shown that along with straggler tasks, the proposed system significantly decreases job completion time compared with other systems because of proposed checkpoint interval. The proposed system can also save rework time because the system continues the work from last checkpoint in case of straggling tasks.
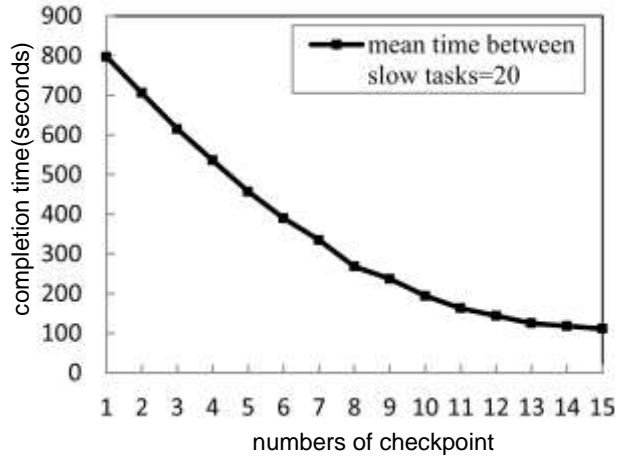

Figure 2. Job completion time versus numbers of checkpoint
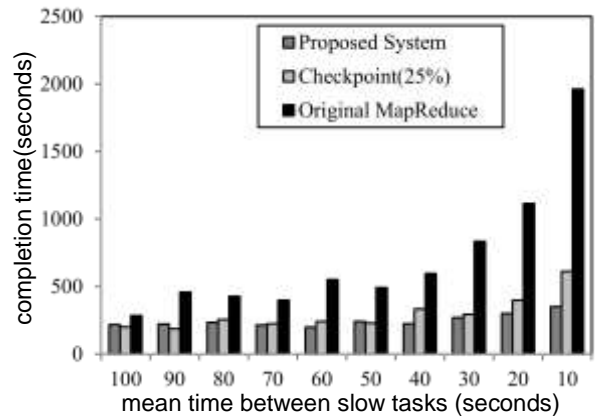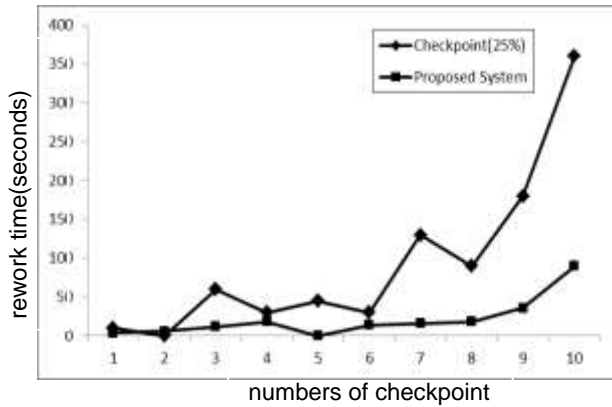

Figure 3. Comparison of completion time with checkpoint overhead=5s, restart time=2s and downtime=2s

Figure 5. Comparison of completion time based on rework time

6. 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 speculative execution strategy. Although MapReduce is able to retry the straggling tasks, it performs poorly because it re-executes all finished ranges again in case of stragglers. As a result, MapReduce job can prolong job completion time when straggling tasks occur.

To overcome the limitation of existing speculative execution in MapReduce, the proposed system uses checkpointing strategy in order to avoid re-execution of finished tasks in case of straggling tasks. Proposed checkpointing mechanism which defines the most suitable interval to take checkpoints, as a result, saves job completion time, rework time and checkpoint overhead.
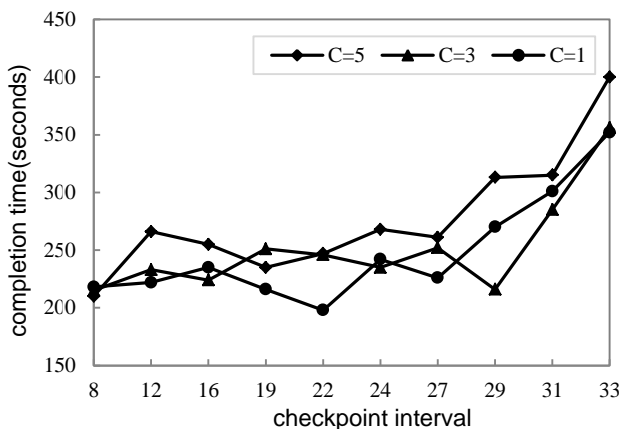
Figure 4. Comparison of checkpoint overhead

The proposed system implemented on the base of Hadoop that is the most popular open source implementation of MapReduce. The proposed system outperforms original MapReduce while decreasing mean time between slow tasks.

7. 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] L.Ying, H.Chen and S.Xiaoyu, "ESAMR: An Enhanced Self-Adaptive MapReduce Scheduling Algorithm," IEEE 18th International Conference on Parallel and Distributed Systems, 2012.

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

[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.

[11] https://data-flair.training/blogs/speculative-execution-in-hadoop-mapreduce/