International Journal on Advanced Science Engineering Information Technology

Vol.10 (2020) No. 5
ISSN: 2088-5334

Asynchronous Non-Blocking Algorithm to Handle Straggler Reduce Tasks in Hadoop System

Arwan A. Khoiruddin[a,1], Nordin Zakaria[a,2], Hitham Alhussian[a,3]

[a] *High Performance Cloud Computing Center, Universiti Teknologi PETRONAS, Seri Iskandar, Perak, 32610, Malaysia*
E-mail: [1]arwa003@lipi.go.id; [2]nordinzakaria@utp.edu.my; [3]seddig.alhussian@utp.edu.my

***Abstract*— Hadoop is widely adopted as a big data processing application as it can run on commercial hardware at a reasonable time. Hadoop uses asynchronous blocking concurrency using Thread and Future class. Therefore, in some cases such as network link or hardware failure, a running task may block other tasks from running (the task becomes straggler). Hadoop releases are equipped with algorithms to handle straggler tasks problem. However, the algorithms manage Map and Reduce task similarly, while the straggler root cause might be different for both tasks. In this paper, the Asynchronous Non-Blocking (ANB) method is proposed to improve the performance and avoid the blocking of Reduce task in Hadoop. Instead of using the single queue, our approach uses two queues, i.e. task queue and callback queue. When a task is not ready or detected as a straggler, it is removed from the main task queue and temporarily sent to the callback queue. When the task is ready to run, it will be sent back to the main task queue for running. The performance of the algorithm is compared with rTuner, the latest paper found on handling straggler task in Reduce task. From the comparison, it is shown that ANB consistently gives faster time to complete because any unready tasks will be directly put into the callback queue without blocking other tasks. Furthermore, the overhead time in rTuner is high as it needs to check the straggler status and to find the reason for a task to become straggler.**

***Keywords*—Hadoop; scheduler; reduce task; asynchronous; non-blocking.**

I. INTRODUCTION

Recently, human digital activities generate an overwhelming amount of data. Based on data reported by Seagate, in 2018, there are about 18 zettabytes of data in the world [1]. The data is collected in either semi-structured, structured, or unstructured format. It is characterized by 4 (four) Vs, namely Volume, Variety, Velocity and Value, hence named as Big Data [2]. Volume means the scale of the size of the data is increasingly massive. Variety indicates that the data is diverse, which may include structured, semi-structured, or even unstructured data. The types are also varied, such as audio, video, webpage, and text. Velocity means that the data need to be acquired swiftly, but also processed at a quick rate.

The vast amount of data becomes invaluable if it is only stored. However, when ones do mining on the data, they will get information, knowledge and wisdom from the data. In other words, the data will become valuable. Thus, the 4th characteristic of the big data, i.e. value, can be defined as a business benefit that gives an organization a compelling advantage due to the ability to decide based on the information acquired from the data. As the data volume in big data is collected at a fast velocity, the data must be processed in a reasonable time. When the processing time is too lengthy, the information acquired might be no longer relevant for the organization.

For this purpose, a specific type of application is needed. Hadoop is one of the applications widely adopted to process big data. Hadoop is renowned because it can run on commercial hardware, thus can be adopted by nearly all organizations. In addition to that, the MapReduce programming framework adopted by Hadoop is simple yet powerful. Moreover, scaling up/down Hadoop is simple and straightforward, hence able to adapt to the performance needed by the company.

Aside from scaling up/down the cluster, to achieve a certain performance level, Hadoop can also be optimized at the application level by managing the file placement [3], or by managing the jobs [4]. In this paper, the authors improve Hadoop performance by optimizing the job concurrency. To deal with the concurrency of the tasks, Hadoop uses synchronized Thread and java.util.concurrent.Future class. It should be noted that synchronized Thread and Future class have several limitations. One of the limitations is that they cannot be attached to a callback function which will call a task automatically when the results are available. In other words, though both can run a task asynchronously, both will block the channel until the task has finished running. Note that a task running in the asynchronous channel can block the channel when it becomes a straggler task. A task

becomes straggler when it submits the result slow due to numerous reasons stated below [5]:

- Network link failures, network traffic and network bandwidth
- Hardware failures and crashes
- Fault at a data block level
- Low computational power as compared to other computers
- Misconfiguration of the job
- Heavy background noise, e.g. the CPU is busy with other tasks

Straggler task may constitute a severe impact on task allocation and scheduling. Especially when the data is skewed, and the processing efficiency of the node is low, there will be a particular type of task that runs at a significantly slower speed than other tasks. In some cases, some tasks run at five times slower than the average duration of other tasks. Thus, it is mentioned in. Thus, it is mentioned in [5] that straggler task may delay average job completion time by 47% in Hadoop cluster. Moreover, because the straggler task may block the channel, there might be a chance where the resources are allocated but not utilized.

In this paper, a method to handle straggler tasks using asynchronous non-blocking method is proposed. Instead of using a single task queue, our method incorporates two queues, i.e. task queue and callback queue. The nature of Map and Reduce tasks are both different. Thus, our method focusses on managing Reduce Tasks. The main contribution of this paper is on dealing with straggler tasks, especially during Reduce phase in the Hadoop system in a non-blocking manner. Though in this paper, the method is simulated and not implemented in real Hadoop yet, it can be a proof of concept that changing the concurrency method in Hadoop will tremendously increase the performance. Note that the Hadoop and MapReduce version referred in this paper is Hadoop 1.1.2.

The paper is organized as follows. Section II presents the related works, while Section III discusses the proposed method named Asynchronous Non-blocking Scheduler (ANB Scheduler). Section IV shows the setup and the results of the implementation of ANB Scheduler compared to other schedulers. Section V concludes the paper.

II. Materials and Method

In this section, the basic concept of Hadoop will be discussed, including the architecture and the default schedulers. The straggler problem and schedulers that handle the straggler tasks problem are also reviewed. The next subsection discusses the existing schedulers that handle straggler reduce tasks. The last subsection addresses the method proposed.

A. Hadoop Architecture

In general, Hadoop consists of two layers, i.e. MapReduce and Hadoop Distributed File System (HDFS) layer (Fig 1). MapReduce layer is used for distributed data processing, while HDFS is used for distributed data storage. Both MapReduce and HDFS occur in master and slaves. HDFS is a distributed file system that manages big data running on commercial hardware. HDFS is designed to promote fast recovery from failures, streaming access to data, accommodation of any size of big data and compatibility with various operating systems. In HDFS, the data is split into blocks then stored across multiple machines. Before spreading, the blocks are replicated to avoid data loss due to either hardware or network failures. The original and the replicated blocks are spread into different nodes in the cluster.

By default, the size of the block is 64 MB. The size is chosen because having a much smaller size will cause high seek overhead. On the other side, the significantly larger block size will reduce the parallelism [6]. The blocks are replicated and spread into nodes. NameNode keeps tracks on the block ID and the ID of the node having the block.

When an application is submitted, the job is initialized on the job queue, and JobTracker creates the corresponding Map and Reduce tasks. Each task needs a particular data block. JobTracker talks to the NameNode to determine the location of the data. Sending the job to the node that has the data needed is cheaper than transferring the data. Thus, JobTracker locates TaskTracker nodes with available slots at or near the data then submits the jobs to the chosen node. The TaskTracker nodes are then monitored using a mechanism called heartbeat.

The heartbeat signal carries information like total storage capacity, the fraction of storage in use and the number of data transfer. If the node does not send a heartbeat signal during a specified time interval, the node is considered as failed. JobTracker follows up the information by determining whether the task will be submitted to a different node, identify the specific record as something to avoid, or blacklist the TaskTracker as unreliable.

As mentioned earlier, to achieve the best performance, instead of moving the block into the node having the job, the job is copied into the node having the data. In addition to that, to avoid the data collision in the network switch, Hadoop schedulers should minimize the data transfer. Nevertheless, it is challenging because Hadoop is data-intensive. The blocks of data are spread in different nodes. On the other hand, a job can only run when it is in the same node with the data block needed by the job. To handle the problem, Hadoop allocates the jobs at the nodes at the node having the block needed or near to it. The closeness between the data and the task requiring the data is called data locality [7]–[10].

There are three levels of data locality in Hadoop, i.e.:

- First level locality, occurs when data and task are in the same node
- Second level locality, when task and data are in the different node but in the same rack
- Third level locality, when data are in different node and different rack.

To improve the data locality and to reduce network traffic, the scheduler should be equipped with data-locality awareness policy. Intuitively, the best locality is the first level data locality, where the jobs are in the node holding the required input data. When a job cannot attain the first level data locality, the scheduler moves the jobs into the node having the data required by the job. The job will be moved to a different node in a similar rack (second-level data locality). If all nodes in similar racks are busy, it will be sent to a node in a different rack (third-level data locality).

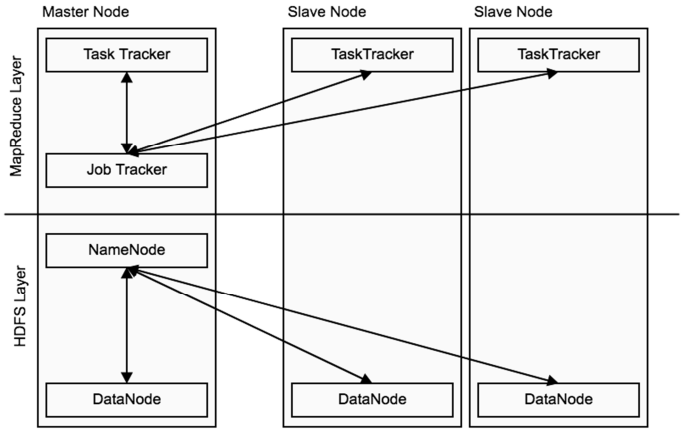

Fig. 1 Hadoop Architecture

An example of data locality problem in Hadoop system is presented in Fig 2. The system contains two racks, while each rack contains three nodes. In order to simplify, each node holds a block of data and a container with a particular job. Job *X* will only run if it is in the same node with data block *X*, needed by the job. If both are not in similar node, the job will be transferred to the node having the block required before it can be executed. Accordingly, as job *A* and *F* have data block needed, both achieve first-level data locality. Job *C* and job *E* attain second-level data locality since the data required are in another node but still in a similar rack. Job *D* and job *B* gain the third-level data locality as the data needed are in a different node in a separate rack.

B. Hadoop Scheduler

Hadoop is designed to process an immense amount of data. In order to acquire the value, the data needs to be processed in the specific time frame. Thus, one should ensure that Hadoop achieves a certain level of performance. The performance is mainly defined by the scheduler used. Consequently, the scheduler needs to be properly chosen and configured. Hadoop is shipped along with three (3) default scheduling algorithms that support data locality, i.e. FIFO, Fair and Capacity Scheduler [11].

The schedulers allocate the incoming tasks into the available resources. As mentioned in subsection A, resource availability is monitored through a heartbeat mechanism sent from slave nodes to master nodes. The Hadoop job scheduling problem is a multi-objective, as well as an NP-hard problem.

FIFO is the original and the default Hadoop scheduler. The main goal is to schedule jobs based on the job arrival in the queue (First-in First-out) [11], [12]. Resources will be allocated to the job in the frontmost of the queue. It does not consider job size and priority. Consequently, it may have some limitations, such as poor response times for small jobs and low performance on running multiple types of jobs. Furthermore, the scheduler is not suitable for a shared cluster as large jobs will utilise all the available resources.

Fair Scheduler is a method for assigning resources to jobs such that all jobs get a nearly equal share of resources [11], [12]. Fair scheduler organises jobs employing resource pools and fairly shares resources between these pools. The Fair Scheduler manages the allocation of resources to pools or queue and also handles the distribution of jobs to pools. Jobs can also be explicitly submitted to pools. When there is a conflict for resources, high priority jobs or jobs waiting too long can run. By default, the fairness decision is based only on memory. However, the scheduler can be configured to decide based on both memory and CPU in the form of (x MB, y vCores).

By default, all users share a single queue, named “default.” If an app specifically lists a queue in a container resource request, the request is submitted to that queue. The queue can also be allocated based on the user name included through configuration. Within each queue, a scheduling policy is employed to share resources among the apps. Queues can be organized in a hierarchy to distribute resources and configured with weights.

When only a single job is submitted, it will utilize the entire cluster. When there are other jobs submitted, they will be allocated to the free task slots. This policy will guarantee that each job will get a similar amount of resources. It allows small jobs to complete within a reasonable time while not starving long jobs. The scheduler can work well in both small and large jobs, which is the limitation in FIFO. The Fair Scheduler is more flexible and allows jobs to consume unused resources in the cluster, thus maximizing resource utilization. The scheduler is a good default for small to medium-sized clusters.

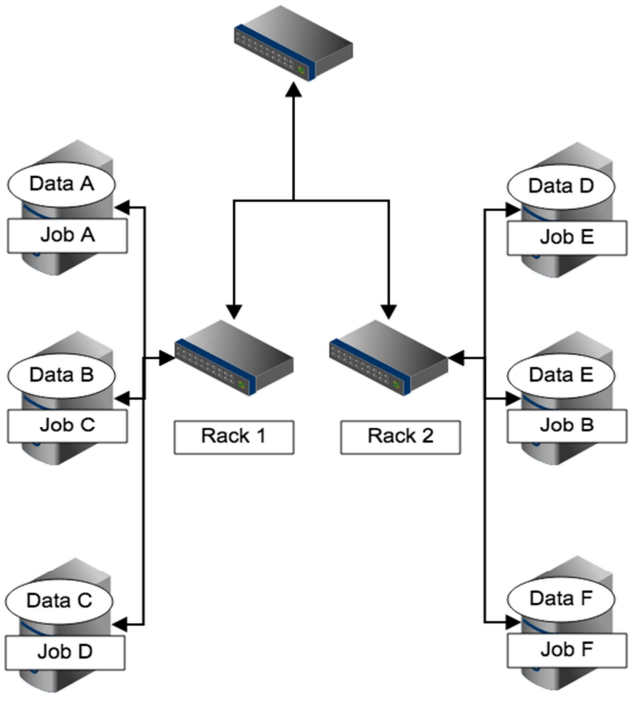

Fig. 2 Example of Data Locality in Hadoop

The CapacityScheduler is a pluggable scheduler designed to run Hadoop applications as a shared, multi-tenant cluster in an operator-friendly manner while maximising the throughput and the utilisation of the cluster. The scheduler uses the concept of queues. Each queue is allocated to an organisation or user, and resources are divided among these queues [13], [14]. Jobs are distributed in multiple queues according to their conditions and allocate specific capacity for each queue. The scheduler allows the priority-based

scheduling of jobs [15]. Moreover, by supporting parallel execution of multiple jobs and cluster sharing among multiple users or organisation, it can improve the utilisation of the cluster resource. However, the scheduler does not consider the user and job heterogeneity. In addition to that, because capacity scheduler uses the concept of queue capacity, queues may have less capacity to take more time to process the job [16].

The CapacityScheduler is designed to allow sharing a large cluster while giving each organization capacity guarantee. The fundamental concept is that the resources in the Hadoop cluster are shared among multiple organizations based on their computing needs. The CapacityScheduler provides a robust set of limits to ensure that a single application or user or queue cannot consume an excessive amount of resources in the cluster. Additionally, the CapacityScheduler provides boundaries on initialized and pending applications from a single user and queue to ensure fairness and stability of the cluster. Moreover, the Capacity Scheduler supports hierarchical queues to guarantee that resources are shared among the sub-queues of an organization before other queues are allowed to use free resources.

C. Straggle-aware Scheduler

As mentioned in Section I, several factors can make a task to become a straggler task. One of them is a network link problem, where the network throughput degrades due to collision in a network switch. The problem happens because the round trip time in the data center is two or three magnitudes higher than the default Retransmission Timeouts (RTO) Timer used in Transmission Control Protocol (TCP) [17]. The problem is called TCP Incast problem and is illustrated in Fig 3.

Fig 3 shows a case when some nodes send data at the same time, and the switch cannot handle the flow. Consequently, the throughput degrades to a small percentage of the actual link capacity. When this problem occurs, the task resides in the other node will be late in sending the result to the master node, thus resulting in straggler tasks.

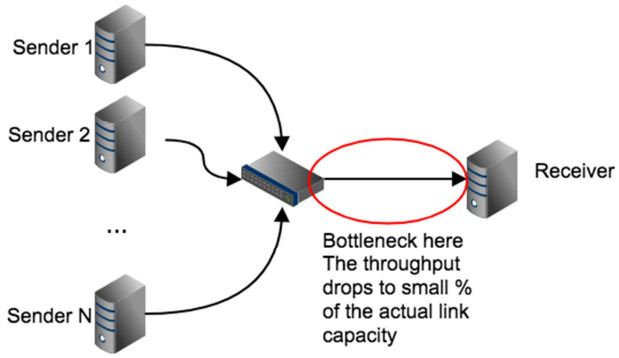

Fig. 3 TCP Incast Problem

Hardware failures may also cause straggler tasks. In Yahoo cluster, there are 2-3 nodes failures per 1000 nodes per day [18], [19]. Google also reports that in each cluster's first year, around 1,000 machines will fail [20] and there is about a 50% chance that the cluster will overheat that will take down most of the servers and take about 1 to 2 days to recover. Both examples show that even in a big company, the failures are inevitable.

The hardware failure will bring the next source of straggler task problem, i.e. the low computational power compared to other nodes. This problem occurs because during the replacement, sometimes it is difficult to find similar hardware to replace the failed one(s). In this case, the latest hardware will have better computing capacity compared to the existing ones. The constant replacement will cause further heterogeneity in the cluster. On the other hand, Hadoop assumes that the cluster is homogeneous [12].

When a cluster is not dedicated to run the specific application (in this case, Hadoop), it may drain the CPU and RAM resources to work on other tasks (heavy background noise). In this case, the tasks allocated by Hadoop may have less priority to run by the busy CPU. If this occurs, the task assigned to run in the node may become straggler.

Straggler task may also happen when the job is misconfigured. This may happen especially on the capacity scheduler, where detail configurations are needed for the tasks and the Hadoop users. Finally, the fault on the data block may also lead to the straggler task problem where the disk where the block is copied is corrupted. If this case is not handled, the task will run forever as it waits for the input from the corrupted block.

Several methods have been proposed to handle straggler tasks in Hadoop. The methods can be categorized into three categories, i.e. reduce skewness, dynamic resource allocation and speculative execution strategy.

Methods that falls into the category of Reduce skewness include [21], [22]. The fundamental conception of the method is that in some cases, the distribution of nodes running tasks is not balanced. For example, due to speculative tasks, a node having lower specification will always be killed because most of the time, it processes the data slower than the other counterparts. The advantage of the methods is the balance data distribution across nodes using a user-defined cost function. Moreover, it takes advantage of idle nodes freed by short tasks.

The methods that belong to dynamic resource allocation are [23]–[25] and [26]. The methods focus on the cluster running above groups of virtual machines (VMs). The idea of the method is to dynamically perform the tuning of the cluster when Hadoop is running. The advantage of the method is that it will dynamically allocate overall capacity among VMs based on their demand. The method will automatically tune the Hadoop cluster that will reduce straggler task problem. However, the disadvantages are that the methods focus on VM Management and less effective for grid nor cloud setup. Moreover, as it may repartition the VM, moving repartitioned data requires extra I/O operation.

The methods that fall into the next category, i.e. speculative execution strategy are [27]–[31]. The idea is to speculate tasks if they are detected as a straggler. The advantage of the methods is that it will mitigate hardware failures by running the straggler tasks in different nodes (hoping that the tasks can run faster thus will finish on the desired time). Despite that, sometimes the speculative execution is performed at the finishing stage of the job, so it cannot address the straggler task problem promptly.

D. Straggler-aware Reduce Task Schedulers

Zaharia et al. proposed the Longest Approximated Time to End (LATE). The basic idea of the algorithm is that all tasks predicted to finish farthest into the future will always be speculated [32]. The idea is developed to overcome performance problem due to the following assumptions in default Hadoop schedulers:

- Nodes are homogeneous and can perform tasks at about the same speed
- Throughout time, all tasks run at a constant rate
- No cost on performing a speculative task on any node in the cluster having an idle slot
- A task's progress score can be represented as a fraction of its total work done.
- Tasks in Hadoop tends to finish in waves
- A task with a low progress score is likely to become a straggler.
- The same amount of works is required for tasks that fall in the same category (map or reduce)

LATE calculates the progress using Equation (1)

$$progressRate = \frac{progressScore}{\tau} \quad (1)$$

Where T is the time spent by the task. Using the equation, LATE can calculate the time for a task to complete as shown in Equation 2

$$\text{timeToComplete} = \frac{\text{1-progressScore}}{\text{progressRate}} \quad (2)$$

However, because LATE always speculate straggler tasks without knowing the reason, it may cause misjudgment. Furthermore, in some cases, the time to end approximation is inaccurate. Due to these problems, the scheduler may waste the resources. The calculation of the approximation is also too long. Thus, some tasks already run for several times longer than it should before it is detected as straggler task.

It is important to note that Map and Reduce tasks have different characteristics presented in Table I. Despite that, the default Hadoop schedulers (FIFO, Fair and Capacity), as well as LATE algorithm, assume that both tasks are similar [33].

TABLE I
COMPARISON BETWEEN MAP AND REDUCE TASKS

Map Task	**Reduce Task**
Independent task	Dependent on Map Task
Contains only a single phase	Contains three phases, i.e. sort, shuffle and reduce

To deal with the inaccuracy of the approximation time in Reduce tasks, Huang et al. [29] proposed Estimate Remaining time Using Linear relationship (ERUL) and extensional Maximum Cost Performance (exMCP) algorithm. The algorithms consider loads of each slot in Hadoop (slot-aware strategy). However, misjudgment can still occur on the estimation of the progress of Reduce tasks.

Other work to improve LATE on Reduce Tasks scheduling was done by Patgiri et al. [22]. In the proposed method, named rTuner, the calculation of the straggler tasks is based on LATE calculation. However, the algorithm checks the reason for straggler and the speculation is based on the reason. Despite that, until the tasks are detected as a straggler, the container will be occupied by straggler tasks and cannot be allocated to other tasks in the queue. The pseudo-code for rTuner algorithm is presented in Algorithm 1.

Algorithm 1 The rTuner Algorithm [22]

```
1: procedure rTuner(TaskTracker T)
2:   for RT in T do
3:   if RT in reduce then
4:     flag = CheckForStraggler(RT)
5:     reasonForStraggler(RT)
6:     decision = CheckForSpeculation(RT)
7:     if decision=true and crossSpeculativeLimit=false then
8:         speculate (RT)
```

E. Proposed Method

As discussed in the previous sections, in the current Hadoop schedulers, when straggler tasks exist, the container handling the task will be blocked so it can be used for other tasks. In other fields such as CPU scheduling discussed in the previous sections, in the current Hadoop schedulers, when straggler tasks exist, the container handling the task will be blocked so it can be used for other tasks. In other fields such as CPU scheduling [34], the blocking problem is handled by removing the blocking tasks from the main queue and sending them into a temporary place (a callback). However, the method has not been implemented in Hadoop, especially in Reduce task scheduling.

In this paper, we adopt the non-blocking method and name our algorithm as asynchronous non-blocking scheduler (ANB Scheduler). In ANB, instead of having a single queue for tasks, it incorporates two queues, i.e. tasks and callback queue. When a task is not ready or detected as a straggler, it is removed from the primary task queue and temporarily sent to the callback queue. We do not want to detect the cause of the straggler because we only want to move the long-run task to callback queue. Thus, the calculation for straggler detection uses the equation used by LATE. When the task is ready to run, it will be sent back to the main task queue for running. The pseudo-code for ANB Scheduler is presented in Algorithm 2. An example of ANB Scheduler is illustrated in Fig 4.

System model: In our system, a job is divided into k tasks. The tasks are allocated into m number of node n through x number of link l. Because the node may have different hardware specification, the nodes can be defined as $n = \{n_1, n_2, \ldots n_m\}$. The link may also have different capacity. Thus, the link should also be defined as $l = \{l_1, l_2, .., l_x\}$.

A straggler task k_n may be likely to happen when the hardware specification of n_m is lower with n_{m-1} or when the link capacity or throughput l_y is lower than l_{y-1}. When k_n is straggler, the task is removed from the task queue and put into callback queue. In this case, Hadoop can run k_{n+1} without interrupted. When k_n is ready, a callback function

will put the task back in task queue and remove from the callback queue.

Algorithm 2 Asynchronous Non-blocking Scheduler

```
1: procedure ANBScheduler(TaskTracker T)
2:  FOR RT in T
3:  IF RT in reduce THEN
4:    IF RT is waiting or straggler THEN
5:      remove RT from task queue
6:      put RT to callback queue
7:    IF RT is ready THEN
8:      put RT to task queue
```

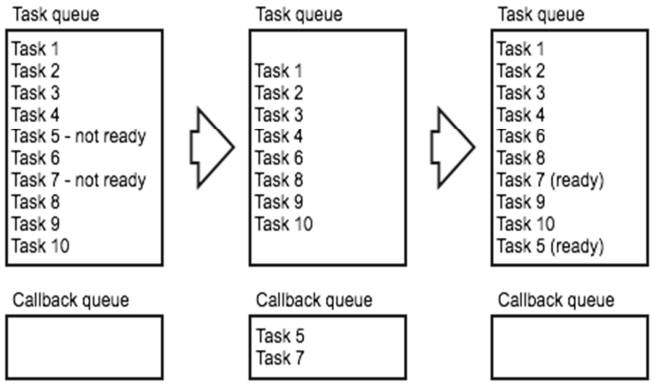

Fig. 4 An Example of ANB Scheduler

III. Results and Discussion

We implement our algorithm in Vultr cloud service using containernet. Containernet is a network function virtualisation (NFV) tool based on mininet [36] and Docker container. Containernet is chosen because it can be used to simulate network in any size and condition. In addition to that, using containernet, it is also possible to vary the CPU power between nodes thus can simulate the condition that can trigger a straggler task problem.

TABLE II
Simulation Setup

Item	Specification
CPU Core	4
RAM	8 MB
Number of nodes	3
CPU Variation	2 same, 1 lower spec (10% of the power of the other two CPUs)
Network link	10 Mbps 5ms delay 2% loss 1000 packet queue
Number of experiments run	10 runs

The details of the setup of the experiment conducted are shown in Table II. A total of 10 experiments are conducted to compare rTuner and ANB Scheduler. The algorithms are run on three (3) nodes to measure the time-to-complete on running Reduce tasks using ANB and rTuner scheduler. The results are shown in Fig 5.

From the result, it is shown that from 10 experiments, the time-to-complete to run all tasks using ANB scheduler (in blue) is faster than when rTuner is used. In average, the time needed to complete all tasks using rTuner is 1,386.8 ms while the average time using ANB is 717.5 ms. It means that the average time in our algorithm is 51.89% less than the time needed to complete tasks run using rTuner. The standard deviation for the time-to-complete in ANB is 54.99% while the standard deviation for the time-to-complete in rTuner is 84.29%. From the standard deviation, it is shown that our algorithm gives a more consistent result than rTuner.

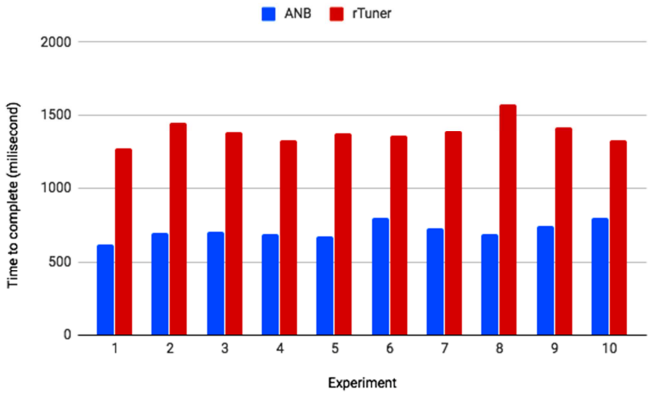

Fig. 5 Comparison of Time To Complete (in milliseconds) using rTuner (read) vs ANB Scheduler (blue)

In general, our algorithm performs better than rTuner, mainly because of at least two reasons. First, using ANB, when a task is not ready to run, it is directly put into the callback queue without blocking other tasks. Once it is ready, a callback mechanism will put the task back into the task queue for execution. Secondly, the overhead time of rTuner is high as it needs to check the straggler status of all tasks. In addition to that, the algorithm needs to find the reason for straggler before speculating the tasks. While finding the reason is good because it will decrease the possibility for speculating near-to-complete task and the finished task that detected as straggler due to network problem, the procedure will consequently increase the overhead time.

IV. Conclusion

In this paper, the algorithm to handle straggler Reduce Tasks in Hadoop has been presented, namely Asynchronous NonBlocking Scheduler (ANB Scheduler). The scheduler uses the callback queue as a temporary queue for blocking tasks. Thus, the other ready tasks can run on the cluster. The algorithm has been compared to the latest algorithm found in the same area, i.e. rTuner. Our simulation shows that our algorithm performs better than rTuner.

In this paper, we implement our method in a simulated Hadoop system based on the concept of Hadoop 1.1.2. In the future, we will implement this concept in Apache Hadoop and perform some experiments in a heterogeneous environment using Containernet. Containernet is used as it can simulate the network with various hardware specifications as well as the link capacity between nodes. We will also experiment with our real cluster with some old and problematic hardware installed. By implementing our algorithm in Hadoop, and performing the experiments in

both containernet and real cluster to run various tasks, we will be able to ensure that our algorithm can increase the performance of Hadoop system, especially under heterogeneous setup.

REFERENCES

[1] D. Reinsel, J. Gantz, and J. Rydning, "The digitisation of the world: from edge to core," *IDC White Paper*, 2018.

[2] Y. Sun, Y. Shi, and Z. Zhang, "Finance Big Data: Management, Analysis, and Applications," *Int. J. Electron. Commer.*, vol. 23, pp. 9–11, 2019.

[3] M. Nakagami, J. A. B. Fortes, and S. Yamaguchi, "Job-Aware Optimization of File Placement in Hadoop," *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2, pp. 664–669, 2019.

[4] X. Luo and X. Fu, "Configuration optimisation method of Hadoop system performance based on genetic simulated annealing algorithm," *Cluster Computing*, pp. 1–9, 2018.

[5] P. Garraghan, X. Ouyang, R. Yang, D. McKee, and J. Xu, "Straggler Root-Cause and Impact Analysis for Massive-scale Virtualized Cloud Datacenters," *IEEE Transactions on Services Computing*, vol. 12, pp. 91–104, 2019.

[6] H.-G. Kim, "Effects of Design Factors of HDFS on I/O Performance," *J. Comput. Sci.*, vol. 14, pp. 304–309, 2018.

[7] D. Choi, M. Jeon, N. Kim, and B.-D. Lee, "An Enhanced Data-Locality-Aware Task Scheduling Algorithm for Hadoop Applications," *IEEE Systems Journal*, vol. 12, pp. 3346–3357, 2018.

[8] X. Du, Y. Liu, and C. Zhao, "A Hadoop Yarn Scheduling Based on Node Computing Capability and Data Locality in Heterogeneous Environments," 2018.

[9] K. Midoun, W.-K. Hidouci, M. Loudini, and D. Belayadi, "RTSBL: Reduce Task Scheduling Based on the Load Balancing and the Data Locality in Hadoop," 2018.

[10] P. Zhang, C. Li, and Y. Zhao, "An Improved Task Scheduling Algorithm Based on Cache Locality and Data Locality in Hadoop," *2016 17th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pp. 244–249, 2016.

[11] K. Kalia and N. Gupta, "A Review on Job Scheduling for Hadoop Mapreduce," *2017 International Conference on Next Generation Computing and Information Systems (ICNGCIS)*, pp. 75–79, 2017.

[12] A. Sharma and G. Singh, "A Review of Scheduling Algorithms in Hadoop," 2020.

[13] A. M. S. Lakshmi, N. S. Chandra, and M. BalRaju, "Optimised Capacity Scheduler for MapReduce Applications in Cloud Environments," 2019.

[14] H. Chen and D. Cui, "SLA-based Hadoop Capacity Scheduler Algorithm," 2015.

[15] J. A. Murali and T. Brindha, "Analysis of Scheduling Algorithms in Hadoop," 2018.

[16] J. V. Gautam, H. B. Prajapati, V. K. Dabhi, and S. Chaudhary, "Empirical Study of Job Scheduling Algorithms in Hadoop MapReduce," *Cybernetics and Information Technologies*, vol. 17, pp. 146–163, 2017.

[17] Y. Xu *et al.*, "RAPID: Avoiding TCP Incast Throughput Collapse in Public Clouds With Intelligent Packet Discarding," *IEEE Journal on Selected Areas in Communications*, vol. 37, pp. 1911–1923, 2019.

[18] P. Pandey, S. Singh, and S. Singh, "Cloud computing," in *ICWET*, 2010.

[19] B. T. Rao, N. V. Sridevi, V. K. Reddy, and L. S. S. Reddy, "Performance Issues of Heterogeneous Hadoop Clusters in Cloud Computing," *ArXiv*, vol. abs/1207.0894, 2012.

[20] S. Shankland, "Google spotlights data center inner workings," *CNET*. https://www.cnet.com/news/google-spotlights-data-center-inner-workings/ (accessed Jun. 07, 2020).

[21] M. Liroz-Gistau, R. Akbarinia, D. Agrawal, and P. Valduriez, "FP-Hadoop: Efficient processing of skewed MapReduce jobs," *Information Systems*, vol. 60, pp. 69–84, 2016.

[22] R. Patgiri and R. Das, "rTuner: A Performance Enhancement of MapReduce Job," in *ICCMS 2018*, 2018.

[23] S. Ghemawat *et al.*, "Performance Tuning and Scheduling of Large Data Set Analysis in Map Reduce Paradigm by Optimal Configuration using Hadoop," 2019.

[24] X. Hua, M. C. Huang, and P. Liu, "Hadoop Configuration Tuning with Ensemble Modeling and Metaheuristic Optimization," *IEEE Access*, vol. 6, pp. 44161–44174, 2018.

[25] M. A. Rahman, A. Hossen, J. Hossen, C. Venkataseshaiah, T. Bhuvaneswari, and A. Sultana, "Towards machine learning-based self-tuning of Hadoop-Spark system," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 15, p. 1076, 2019.

[26] W. Wang, Y. Shi, X. Liu, Y. Feng, and N. Tao, "Hadoop Performance Tuning based on Parameter Optimization," 2018.

[27] Y. Guo, J. Rao, C. Jiang, and X. Zhou, "Moving Hadoop into the cloud with flexible slot management and speculative execution," *IEEE Transactions on Parallel and Distributed systems*, vol. 28, no. 3, pp. 798–812, 2016.

[28] Y. Guo, J. Rao, C. Jiang, and X. Zhou, "Moving Hadoop into the Cloud with Flexible Slot Management and Speculative Execution," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, pp. 798–812, 2017.

[29] X. Huang, L. Zhang, R. Li, L. Wan, and K. Li, "Novel heuristic speculative execution strategies in heterogeneous distributed environments," *Computers & Electrical Engineering*, vol. 50, pp. 166–179, 2016.

[30] D. C. Vinutha and G. T. Raju, "Evolutionary Approach based Scheduler for Speculative Task Execution," *2019 1st International Conference on Advances in Information Technology (ICAIT)*, pp. 485–490, 2019.

[31] Q. Liu, W. Cai, J. Shen, Z. Fu, X. Liu, and N. Linge, "A speculative execution strategy based on node classification and hierarchy index mechanism for heterogeneous Hadoop systems," *2017 19th International Conference on Advanced Communication Technology (ICACT)*, pp. 889–894, 2017.

[32] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments," in *OSDI*, 2008.

[33] S. R. Pakize, "A Comprehensive View of Hadoop MapReduce Scheduling Algorithms," 2014.

[34] M. Beckert and R. Ernst, "Response time analysis for sporadic server-based budget scheduling in real time virtualisation environments," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, pp. 1–19, 2017.

[35] F. Kaltenberger, C. Roux, M. Buczkowski, and M. Wewior, "The OpenAirInterface application programming interface for schedulers using Carrier Aggregation," in *2016 International Symposium on Wireless Communication Systems (ISWCS)*, 2016, pp. 497–500.

[36] M. Peuster, J. Kampmeyer, and H. Karl, "Containernet 2.0: A Rapid Prototyping Platform for Hybrid Service Function Chains," *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, pp. 335–337, 2018.