Pertanika J. Sci. & Technol. 32 (5): 2281 - 2298 (2024)

PERTANIKA JOURNALS

SCIENCE & TECHNOLOGY

Journal homepage: http://www.pertanika.upm.edu.my/

# A Reliable Multimetric Straggling Task Detection

**Lukuman Saheed Ajibade[1,2]*, Kamalrulnizam Abu Bakar[1], Muhammed Nura Yusuf[1,3] and Babangida Isyaku[1,4]**

[1]*Faculty of Computing, Universiti Teknologi Malaysia, 81310, Skudai, Johor, Malaysia*
[2]*Department of Computer Science, Federal Polytechnic Offa, Nigeria*
[3]*Department of Mathematical Sciences, Abubakar Tafawa Balewa University, Bauchi, Nigeria*
[4]*Department of Mathematics and Computer Science, Sule Lamido University, K/Hausa, Nigeria*

## ABSTRACT

One of the most difficult issues in using MapReduce for parallelising and distributing large-scale data processing is detecting straggling tasks. It is defined as recognising processes that are operating on weak nodes. When two steps in the Map phase (copy, combine) and three stages in the Reduce phase (shuffle, sort, and reduce) are included, the overall execution time is the sum of the execution times of these five stages. The main objective of this study is to calculate the remaining time to complete a task, the time taken, and the straggler(s) detected in parallel execution. The suggested method is based on the use of Progress Score (PS), Progress Rate (PR), and Remaining Time (RT) metrics to detect straggling tasks. The results obtained have been compared with popular algorithms in this domain, such as Longest Approximate Time to End (LATE) and Combinatory Late-Machine (CLM), and it has been demonstrated to be capable of detecting straggling tasks, accurately estimating execution time, and supporting task acceleration. RMSTD outperforms LATE by 23.30% and CLM by 19.51%.

*Keywords:* Big data, MapReduce, progress score, straggling tasks, stragglers

ARTICLE INFO

*Article history:*
Received: 23 October 2023
Accepted: 01 February 2024
Published: 26 August 2024

DOI: https://doi.org/10.47836/pjst.32.5.19

*E-mail addresses*:
saheed2066@graduate.utm.my (Lukuman Saheed Ajibade)
knizam@utm.my (Kamalrulnizam Abu Bakar)
ymnura@atbu.edu.ng (Muhammed Nura Yusuf)
babangida.isyaku@slu.edu.ng (Babangida Isyaku)
* Corresponding author

## INTRODUCTION

This paper presents a Reliable Multimetric Straggling Task Detection algorithm (RMSTD) strategy for detecting straggling tasks among tasks executing in parallel. The main aim of the RMSTD is to present an approach that uses multiple metrics to make straggler detection much more reliable and

ISSN: 0128-7680
e-ISSN: 2231-8526

accurate. Straggler detection in the Hadoop MapReduce framework refers to identifying tasks that take longer than expected and are known as "stragglers" (Ouyang et al., 2016). In straggler detection, the overall execution time is the sum of the execution times of these five steps, consisting of two phases in the Map phase (copy, combine) and three stages in the Reduce phase (copy shuffle/sort, and reduce) (Katrawi et al., 2021).

Hadoop MapReduce and Apache Spark represent two widely adopted technologies for processing large datasets in the industry. Although both frameworks excel at managing substantial volumes of data, they diverge in terms of their architectural designs (Ketu et al., 2020). Hadoop MapReduce employs a cost-effective approach, utilising the Hadoop Distributed File System (HDFS) to execute batch processing. It is recognised for its stability and maturity, having been in use for an extended period and earned the trust of numerous organisations for handling extensive data volumes. The framework boasts a straightforward programming model, enhancing its usability. In contrast, Apache Spark offers a different approach and architecture for large-scale data processing.

Common methods of straggler detection include resource usage monitoring, where usage of system resources such as CPU, memory and disk utilisation is monitored as tasks execute, such that tasks that use fewer resources during their execution relative to other tasks may be declared as stragglers (Javadpour et al., 2020). Also, the Ensemble method integrates various straggler detection techniques to improve detection accuracy (Kumar et al., 2021). Profile-based analysis entails profiling tasks and finding outliers based on runtime, resource utilisation, or other factors. Another approach is the use of machine learning techniques where factors such as task running time, resource utilisation, input data records and cluster conditions are used to detect straggling tasks; this approach identifies trends in historical data in forecasting likely straggling tasks (Ouyang et al., 2018).

However, in resource usage monitoring, a task's resource utilisation may be unreliable as an indicator of a straggler because it could legally use fewer resources if it has minimal data requirements or is executing on a node with limited resources. On the other hand, the ensemble approach, profile-based strategy, and machine learning techniques have the disadvantages of complexity, high profiling, and computational overheads. In most of the previous works on straggler detection, the use of a single metric to detect straggling tasks is quite common because the MapReduce framework is designed for a homogeneous environment where the computational power of the various machines is the same; hence, there is little consideration for the CPU capability since all the tasks are expected to run at the same rate. This assumption is not very true in all circumstances because, in a typical data centre, the resources are not dedicated exclusively to a particular job; hence, the resources are shared. Therefore, the load on each node varies, which may, in turn, affect the rate of execution of the tasks spread across the nodes. Therefore, this paper proposes a Reliable

Multimetric Straggler Detection Algorithm (RMSTD) to address the inadequacy of the previous studies and improve the effectiveness of straggler detection. The contribution of this paper includes:

- Improvement of the reliability of the straggling tasks detection by using Average Remaining Time (ART) to complete execution of the tasks to ensure that scenarios like the late start of execution of the task(s) due to the load on the node(s) and data skew problem rather than the usage of a single metric.
- The task's historical behaviour mitigates the impact of short-term oscillations or outliers in the Remaining Time, resulting in a more reliable estimate of job completion.
- RMSTD offers a more trustworthy approach by considering the past average, giving a more informed perspective on the anticipated completion timeframes of tasks.

## RELATED WORKS

Phan et al. (2019) proposed a Framework for Assessing the Stragglers Detection (FASD) mechanism over MapReduce for straggler detection because other studies tend to focus more on the impact of stragglers. FASD presented a comprehensive straggler detection and reduction approach. However, the evaluation only applies to one application, and the study does not consider how it will be used in practice or include empirical data. The study also did not consider using optimal metrics in straggler detection, even though it offers a method for assessing straggler detection algorithms. Ghare and Leutenegger (2005) suggest task replication to enhance job response time. MapReduce. Dean and Ghemawat (2008) employ speculative execution to finish straggler jobs when parallel processing is nearing completion. Mantri reduces stragglers from MapReduce cluster processing nodes (Ananthanarayanan et al., 2019). Mantri's core strategies are straggler task restarting, network-aware task placement, and task output protection. Chen et al. (2014) introduce the speculative execution method of Maximum Cost Performance (MCP). Zaharia et al. (2019)'s Longest Approximate Time to End (LATE) improves Hadoop task scheduling.

In the LATE technique, the remaining running task time for each phase has been assumed to be the same; however, in the Reduce phase, the shuffle stage takes longer to complete than other stages as they are based on the prior task. According to Javadpour et al. (2020), Self-Adaptive MapReduce Scheduling Algorithm (SAMR), Enhanced Self-Adaptive MapReduce Scheduling Algorithm (ESAMR), and Speculative Execution Algorithm Based on Decision Tree (SECDT) algorithms are unable to accurately anticipate the running duration. It is insufficient because the present task differs in several ways from the prior ones. While it is crucial to consider this because the node processing durations vary depending on their characteristics, ESAMR only uses executable information and ignores

node specifications (CPU and memory). A better job assignment scheme for Hadoop, the Earliest Completion Time (ECT) scheme, was presented by Dai and Bassiouni (2013). Two improved replica placement policies for Hadoop, the Partition Replica Placement Policy and the Slot Replica Placement Policy, were proposed by Qiang et al. (2014) and Dai et al. (2016). In contrast to the widely used Standard Deviation (SD) method, Tukey's method (Dai et al., 2017) adopts a statistical technique for identifying outliers that seems more suitable for identifying stragglers and starting speculative execution early. The sensitivity to extreme observations and the time it takes to find stragglers limit this strategy, too. As a result, most existing works declare non-straggling tasks as straggling tasks while ignoring true straggling tasks since they do not apply ideal parameters to recognise straggling tasks among parallel running tasks.

In summary, most existing studies on straggling task detection are not based on optimum metrics and are usually focused on a specific situation; hence, their usage in other situations usually leads to failure. For example, if a system is designed to work in a homogeneous environment, it cannot be used in a heterogeneous environment because issues like skews are not considered.

## PROPOSED SOLUTION

This paper presents a Reliable Multimetric Straggling Task Detection algorithm (RMSTD) strategy for detecting straggling tasks among tasks; the RMSTD algorithm is designed to improve the straggler detection strategy irrespective of the environments (homogeneous or heterogeneous). This approach uses optimal parameters to detect straggling tasks applicable in all environments, including skew situations. The primary objective of this problem is to estimate the correct execution time in each stage of the MapReduce framework, which results in the correct total execution time. The constraints associated with this problem are the two stages in the Map phase (copy, combine) and three stages of Reduce (shuffle, sort, and reduce). The total execution time is the total sum of the execution time of these five stages.

The proposed method for solving this problem is calculating the Average Remaining Time (ART) to complete the execution of the tasks running in parallel. All tasks whose remaining time to complete execution is greater than the calculated ART is/are declared as stragglers. The computational complexity of the RMSTD algorithm is $O(n)$, where $n$ is the number of tasks in the *Task_List*. The algorithm iterates through each task in the list once and performs constant work for each task. Therefore, the algorithm's time complexity is linear with respect to the input size. The design and operational process of RMSTD is structured into two phases: (1) the initial task allocation phase and (2) progress monitoring and the straggler detection phase. Figure 1 presents the design flowchart, and Algorithm 1 shows the steps to achieve the desired result.

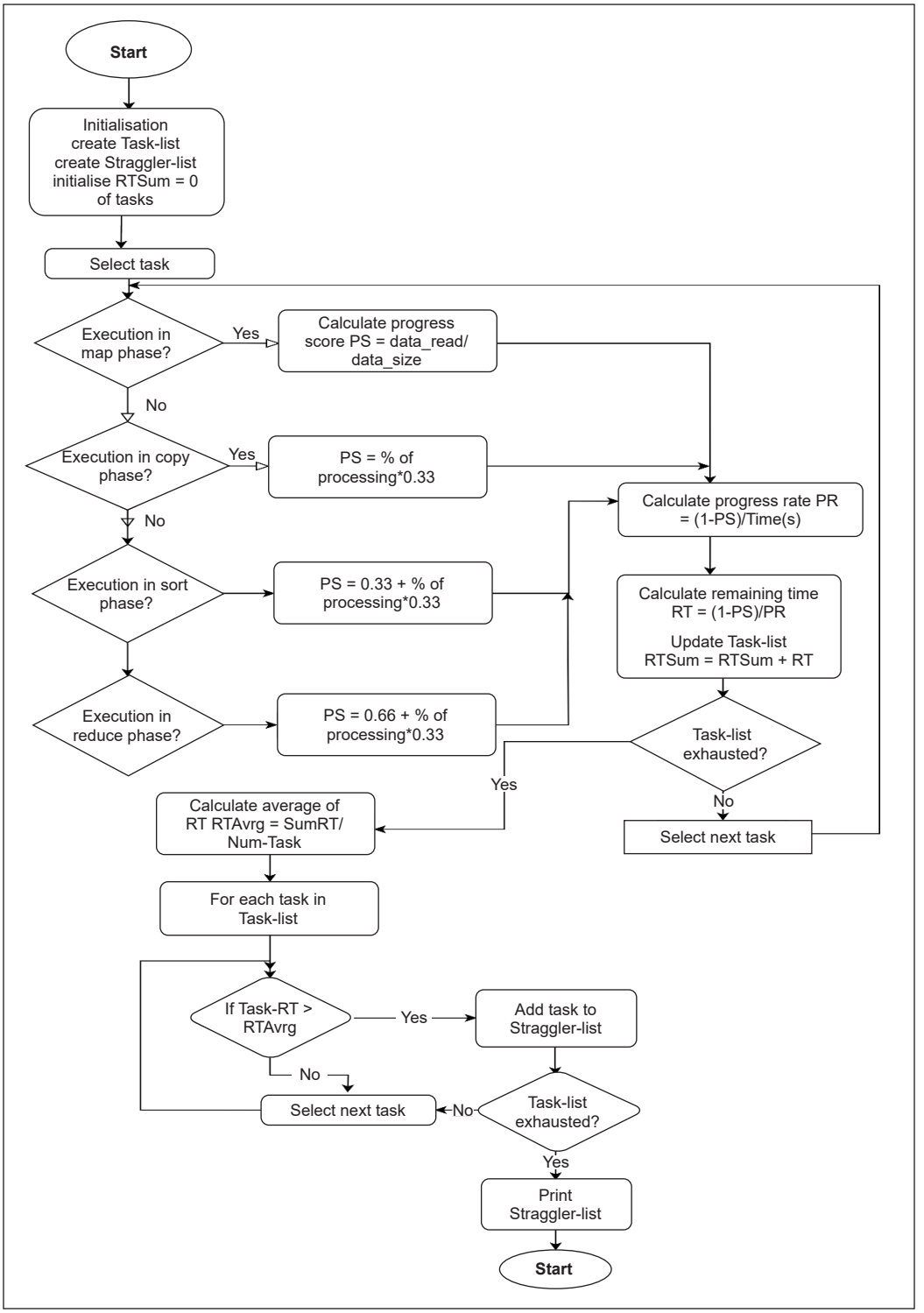


*Figure 1.* Flowchart of RMSTD design

Algorithm 1: RMSTD Algorithm

```
# Create a list of all tasks, "Task_list" with task_id, PS, PR, and RT attributes
# Create a list of all Straggling_tasks, "Straggler_list"

Initialise RTSum = 0, Num-Task = total no of tasks
For each task in Task_list
    If execution = map_phase
        PS =data read/data size
    Else
    If execution = copy_phase
        PS = %processing*0.33
      elseif execution = sort_phase
        PS = 0.33+ %processing*0.33
      elseif execution = reduce_phase
        PS = 0.66+ %processing*0.33
      endif
    Endif

#  Calculate the progress rate (PR)
    PR = PS/Time(s)

calculate the remaining time to finish the task (RT)
    RT = (1-PS)/PR
    Update Task_list with PS, PR, and RT
    RTSum = RTSum+RT
Endfor

RTavrg = SumRT/Num-Task

For each task in task_list:
    If TaskRT > RTavrg
        Add task to Straggler_list
    Endif
Endfor
```

## Data Generation for RMSTD

For testing the design using ART, a Java code (Algorithm 2) was used to generate test data for simplicity since all kinds of data can be handled by the design. Different data types to be processed will have a program to process the data exclusively.

Algorithm 2: Text data generator

```
function NewTextFile(filePath, fileSizeInBytes):
    open file at filePath for writing
    create a random number generator
    create a TextString object
    bytesWritten = 0

    while byteswritten < fileSizeInBytes
        clear the TextString
        lineLength = generate a random number between 1 and 10
        randomLine = LineOfText(lineLength)
        write randomLine to the file
        write a new line character to the file
        bytesWritten += length of randomLine + length of new line
        character
        if bytesWritten is a multiple of 10 MB
            flush the writer to free up memory
        endif
    endwhile
    close the file
 return  NewTextFile

 function LineOfText(lineLength):
    create a TextString object
    for i = 0 to lineLength - 1
        XterLetter = generate a random letter
        append XterLetter to the TextString
     endfor
  return the TextString as a string

main:
fileSizeInBytes = desired file size in bytes
filePath = path to the output file

NewTextFile(filePath, fileSizeInBytes)
```

**Initial Task Allocation**

The task allocation procedure in the Map phase is an important part of the MapReduce framework. It helps ensure that the MapReduce job is executed efficiently, and the results are timely. In this design phase, the input data is read and uploaded into HDFS together with

the job configuration. The allocation of input data to nodes is responsible for processing the input data and generating intermediate key-value pairs in this architecture phase. The Namenode begins the task allocation method by breaking the input data into chunks and assigning each chunk to the Map task. The Map jobs are run in parallel on the available nodes in the Hadoop cluster, and it also generates intermediate key-value pairs that are saved in HDFS, as depicted in Figure 2. The following are the steps involved in allocating tasks to selected nodes. The Namenode splits the input data into 64Mb/128MB/256Mb record-size chunks. Each split is assigned to a Map job. The Job Tracker considers the load on each node, the locality of the chunks' data, and the available resources on each one while allocating tasks to nodes. A task is often assigned to the nodes with the available resources and close to the data to improve the MapReduce job performance (Algorithm 3).

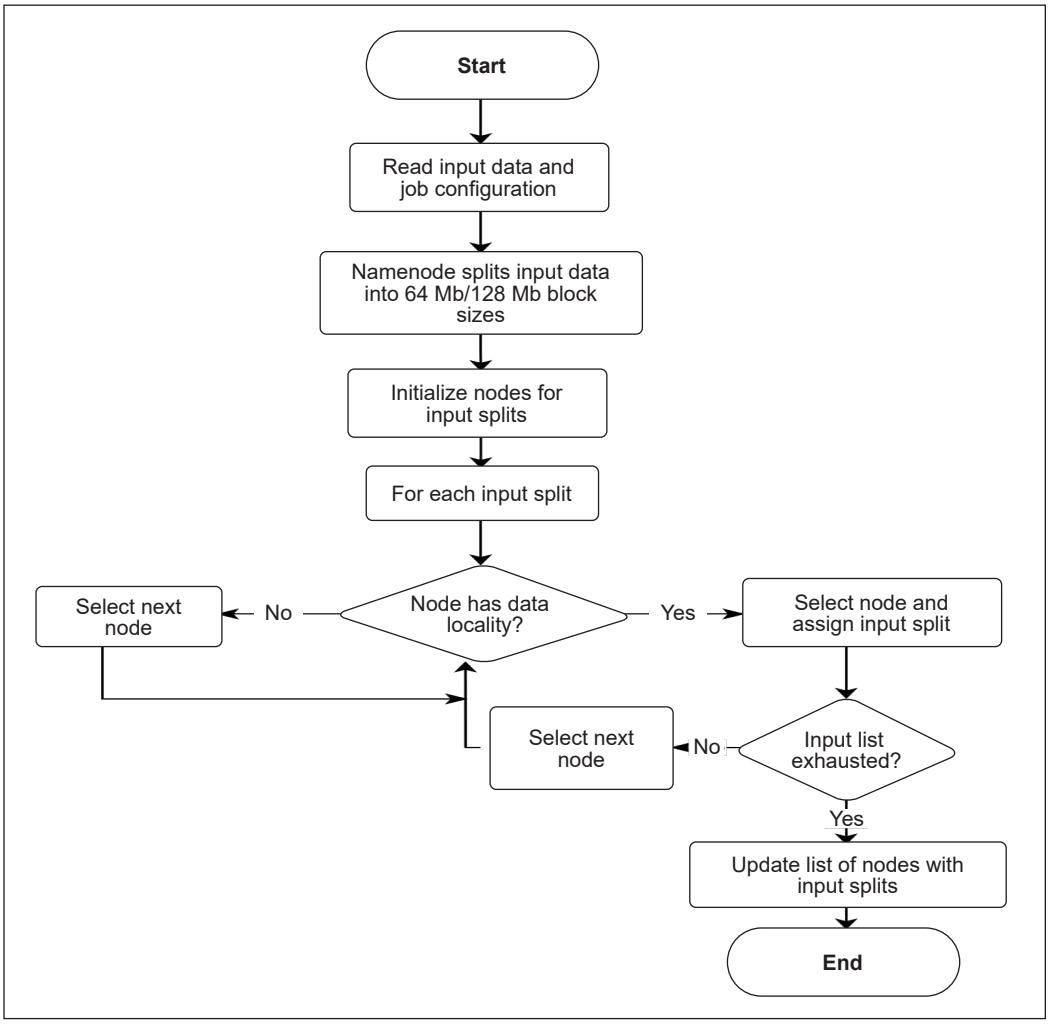


*Figure 2.* Flowchart of initial task allocation

Algorithm 3: Initial Task Allocation Algorithm

1. Input: MapReduce job configuration, input data
2. Output: Map task assignments
3. Split input data into input splits based on block size
4. Initialise an empty map of nodes to assigned tasks
5. For each input split:
6. Select the nodes with data locality for the input split
7. Assign the input split to a selected node with the least number of assigned tasks
8. EndFor
9. Update the map of the node to assigned tasks
10. Return the map of the node to the assigned tasks

**Progress Monitoring and Straggler Detection**

The role of this phase in the operation of RMSTD is to use Progress Score, Progress Rate, and Remaining Time to complete and detect straggling tasks among tasks executing in parallel on the Hadoop cluster. In a Hadoop cluster, where tasks are executed in parallel, it is crucial to monitor the progress of individual tasks to ensure the timely completion of jobs. Straggling task(s) whose progress is/are slower than others can significantly impact the job's overall performance and completion time. To address this challenge, progress monitoring techniques employing metrics such as Progress Score, Progress Rate, and Remaining Time have emerged as valuable tools for detecting straggling tasks in Hadoop clusters.

**Progress Score Calculation**

The Progress Score metric provides an overview of the progress made by a task relative to the total amount of work it needs to complete. It is calculated by dividing the amount of work completed by the total amount of work. It is a score between 0 and 1 (from the literature), where 0 indicates that the task has not started, and 1 indicates that the task is complete. A high progress score indicates that a task is nearing completion, while a low score suggests that a task is lagging. Tracking each task's Progress Score makes it possible to identify tasks that have fallen behind in their progress. A lower Progress Score compared to others suggests a potential straggler.

In a homogeneous environment, that is, where all the nodes are the same in terms of processing capacity, the processing is expected to proceed at the same rate; hence, any task/node with a problem can be easily detected by using PS calculated as Equation 1:

$$PS[i] = \begin{cases} \frac{M}{N} & for\ map\ tasks \\ \frac{1}{3}\left(k + \frac{M}{N}\right) & for\ reduce\ tasks \end{cases} \quad [1]$$

where, *PS[i]* is the *ith* task's P, *N* is the number of key/value pairs that must be processed in a task, *M* is the number of key/value pairs that have already been processed in a certain task, and *K* is the completed phase of a reduction task.

The PS for a Map is the fraction of input data read, but the execution of a Reduce is broken into three phases (copy, sort, and reduce), each accounting for one-third of the total PS. This weighting can be changed by modifying the scheduling parameters. For example, a task halfway through the copy phase will have a PS of 0.5*0.33 = 0.165. while a task halfway through the reduce phase will have a PS of 0.33+0.33+(0.5*0.33) = 0.66+0.165 = 0.83

The value of the Progress score (PS) is taken for each task, and the task(s) whose PS < threshold (determined by individual work) is/are then declared as straggler(s). A threshold of 0.2 is commonly used for comparison as in (LATE) such that any task whose PS < 0.2 is identified as a straggler.

**Progress Rate Calculation**

The Progress Rate measures the speed at which a task is progressing. It is calculated by dividing the amount of work completed by the time taken to complete that work. A high progress rate indicates that a task is progressing quickly, while a low rate suggests that a task is progressing slowly. Monitoring the PR allows the identification of tasks that are progressing at a slower rate than expected, indicating a potential straggler calculated as Equation 2:

$$PR_i = \frac{PS[i]}{T} \quad [2]$$

where, $PR_i$ is the Progress Rate of $Task_i$, and $T$ is the time the task has been executed. A threshold is then determined for PR, at which point a task whose PR is less than the threshold is declared a straggler, which means that its progress is very slow.

There are certain disadvantages to using Progress Score or Progress Rate alone to identify straggling tasks in Hadoop.

- Inaccuracy of Detection: Progress Score and PR are not always accurate because PS is based on how much data has been processed by a task and the amount of time the task has been running, while PR is the rate of progress made by a task. Both can be misleading when a task is simply waiting for input data from a slow network connection or a slow input source. Such a task may have a low PS or PR that can make it be declared as a straggler when it is not.

- The PS or PR status of a job can change if additional parameters like the data collection size, the data locality, the number of concurrent tasks executing on the same node, and network congestion or latency are not considered.
- Lack of context: Progress Score and Progress Rate do not provide context for a task, such as the complexity of the data being processed or the degree of processing difficulty. Some jobs could seem to have a low PS or PR due to the volume of data they are handling, their inherent complexity, or their resource-intensive nature, leading to their being declared as stragglers when they are not.
- Unpredictable progress: Some tasks in Hadoop, particularly those that involve iterative algorithms or complex data dependencies, may exhibit non-linear progress, which causes an inaccurate assessment of their progress and may result in false positives or negatives when identifying straggling tasks using PS or PR alone.
- Data and Computational Skews: In Hadoop, data is frequently distributed unevenly between tasks because of the data's nature or the utilised partitioning method (Data Skew). The processing capacities will differ since the nodes' capacities in a heterogeneous environment are different (Computational skew). These metrics do not consider both skews, leading to some tasks taking longer than others to complete. Hence, the progress of the tasks might not be shown correctly.
- Wrong Assumption: It is usually assumed that tasks progress at a constant rate, which is not usually the case because the nodes are not dedicated to the job alone. The nodes are processing other tasks. Hence, not all the nodes' resources are always available to the tasks.

**Remaining Time (RT) to Complete Calculation**

The RT metric estimates the time required for a task to be completed based on its current PR. It is calculated by dividing the remaining work by the progress rate. A high remaining time to complete suggests that such a task will take a long time to finish, indicating a potential straggler. By comparing the estimated remaining time of each task, it is possible to identify tasks with significantly longer estimated completion times compared to others. Such tasks may be potential stragglers. RT of a task is calculated according to Equation 3.

$$RT_i = \frac{1 - PS[i]}{PR[i]} \quad [3]$$

where $RT_i$, $PS[i]$ Remaining Time (RT), Progress Score (PS) and Progress Rate (PR) of $Task_i$

**Average Remaining Time (ART) to Complete Calculation**

The performance and efficiency of a system rely largely on the timely completion of tasks in distributed computing systems such as Hadoop, where large-scale data processing tasks

are broken into smaller sub-tasks and completed across a cluster of nodes. However, certain tasks, known as stragglers, may take substantially longer time to run to completion than others. These straggling tasks can slow overall throughput and lengthen job completion times, reducing system efficiency and user experience. To address this challenge in this study, the use of Average Remaining Time (ART), a metric that estimates the time remaining for each task to complete based on their Progress Score (PS), Progress Rate (PR) and Remaining Time to Complete (RT), is employed. By continuously monitoring these metrics, the progress of tasks, and comparing their ART values, it becomes possible to identify potential stragglers among tasks being executed in parallel. The ART is calculated as Equation 4:

$$RT_{avg} = \frac{\sum \frac{1 - PS[i]}{PR[i]}}{n} \quad [4]$$

Where, $RT_{avg}$ is the average of the remaining time of execution of all the tasks, $PS$ is the progress score, $PR$ is the progress rate, $n$ is the number of tasks executed in parallel. Any Task$i$ whose value is less than the calculated $RT_{avg}$ is then declared/identified as a straggler.

## PERFORMANCE EVALUATION

### Experimental Setup of the proposed RMSTD

The experiment was set up and conducted on a Google Cloud platform. Eight nodes were used for this experiment. Tables 1 and 2 show the Cluster and Software configurations. Figure 3 shows the screenshot of the nodes running on the Google Cloud platform. 10 GB, 20 GB and 30 GB data (text) were generated using a Java program to test the design. The result of 10 GB data size, when executed using a single node (laptop) with other hardware specifications, is shown in Figure 4. For the 10 GB data on the Google Cloud platform, the readings were taken after 10 s of execution, and the average remaining time was calculated. In comparison, 20 GB and 30 GB data were taken after 30 s and 50 s, respectively, as shown in Figures 4, 5 and 6.

Table 1
*Cluster configurations*

| Cluster Configurations | | | |
|---|---|---|---|
| **Node** | **Main Memory** | **CPU Cores** | **Storage** |
| myclustertask-m (master) | 16G | 4 | 50G |
| myclustertask-0 (slave-1) | 12G | 2 | 50G |
| myclustertask-1 (slave-2) | 12G | 2 | 50G |
| myclustertask-2 (slave-3) | 12G | 2 | 50G |
| myclustertask-3 (slave-4) | 12G | 2 | 50G |
| myclustertask-4 (slave-5) | 12G | 2 | 50G |
| myclustertask-5 (slave-6) | 12G | 2 | 50G |
| myclustertask-6 (slave-7) | 12G | 2 | 50G |
| myclustertask-1 (slave-2) | 12G | 2 | 50G |
| myclustertask-2 (slave-3) | 12G | 2 | 50G |

Table 2
*Software configurations*

| Software Configurations | |
|---|---|
| Operating System | Ubuntu 20.04 |
| Hadoop | 2.8.5 |
| JDK | 1.8 |

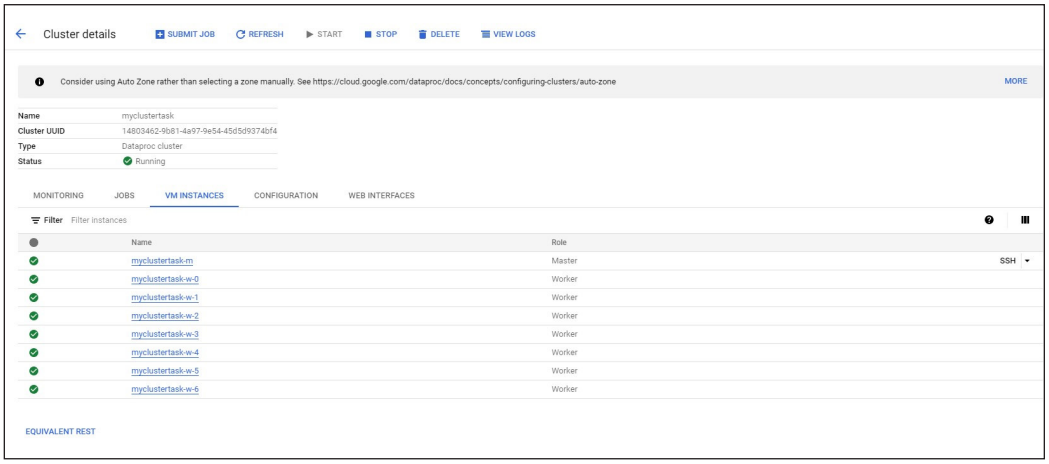


*Figure 3.* Screenshot of the nodes running on Google Cloud Platform

Job Counters
- *Killed reduce tasks=2*
- *Launched map tasks=1*
- *Launched reduce tasks=10*
- *Data-local map tasks=1*
- *Total time spent by all maps in occupied slots (ms)=216810*
- *Total time spent by all reduces in occupied slots (ms)=10717740*
- *Total time spent by all map tasks (ms)=4818*
- *Total time spent by all reduced tasks (ms)=119086*
- *Total vcore-milliseconds taken by all map tasks=4818*
- *Total vcore-milliseconds taken by all reduce tasks=119086*
- *Total megabyte-milliseconds taken by all map tasks=6937920*
- *Total megabyte-milliseconds taken by all reduce tasks=342967680*

*Map-Reduce Framework*
- *Map input records=8*
- *Map output records=25*
- *Map output bytes=256*
- *Map output materialised bytes=423*
- *Input split bytes=98*
- *Combine input records=0*
- *Combine output records=0*
- *Reduce input groups=22*
- *Reduce shuffle bytes=423*
- *Reduce input records=25*
- *Reduce output records=25*
- *Spilled Records=50*
- *Shuffled Maps =10*
- *Failed Shuffles=0*
- *Merged Map outputs=10*
- *GC time elapsed (ms)=3730*
- *CPU time spent (ms)=16880*
- *Physical memory (bytes) snapshot=4417970176*
- *Virtual memory (bytes) snapshot=48205594624*
- *Total committed heap usage (bytes)=4498391040*

*Figure 4.* Result of the experiment (10 GB) on a single node (laptop)

## RESULTS AND DISCUSSION

The experiment was set up and conducted on the Google Cloud platform and tested with 10 GB, 20 GB and 30 GB data; the ART was taken after 10 s, 30 s and 50 s, respectively, because of the data sizes. The result showed consistency for the number of tasks whose remaining time to complete was greater than the ART. At the 10 s threshold, two of the tasks (myclustertask-1 and myclustertask-3) RT were above the threshold of ART; at 30 s threshold, (myclustertask-1 and myclustertask-5) RT were above the threshold of ART and at 50 s threshold, (myclustertask-1 and myclustertask-4) RT were above the threshold of ART with consistency of number of straggling tasks detected. Table 3 shows the experiment results; a graphical representation of the result comparison for different data sizes is shown in Figures 5, 6, and 7. From the results when compared with Katrawi et al. (2020) (CLM) and Dean and Ghemawat (2008) (LATE) on straggling task detection, RMSTD shows an improvement of 19.51% and 23.30%, respectively.

## PAPER CONTRIBUTIONS

The following are the contributions of this paper to the research work:

1. Increased accuracy: RMSTD using ART offers a more precise prediction of when the activity is anticipated to be finished by considering the task's typical behaviour over time. Temporary sluggishness, sporadic resource conflict, or network congestion that can lead to fluctuations in the RT have less of an impact

Table 3
*Experimental data sizes and results*

| | RMSTD | CLM | LATE |
|---|---|---|---|
| | **Data size = 10 GB, Time threshold = 10 s** | | |
| No of Stragglers | 2 | 3 | 4 |
| Time Taken | 12.86 s | 17.20 s | 20.30 s |
| ART | 7.52 s | 12.71 s | 14.09 s |
| | **Data size = 20GB, Time threshold = 30 s** | | |
| No of Stragglers | 2 | 4 | 3 |
| Time Taken | 31.20 s | 37.30 s | 37.10 s |
| ART | 29.30 s | 31.70 s | 33.71 s |
| | **Data size = 30 GB, Time threshold = 50 s** | | |
| No of Stragglers | 2 | 4 | 3 |
| Time Taken | 55.00 s | 69.70 s | 71.20 s |
| ART | 49.55 s | 62.90 s | 64.81 s |
| | | **Averages** | |
| No of Stragglers | 2 | 3.67 | 3.33 |
| Time Taken | 33.02 | 41.4 | 42.87 |
| ART | 28.79 | 35.77 | 37.54 |

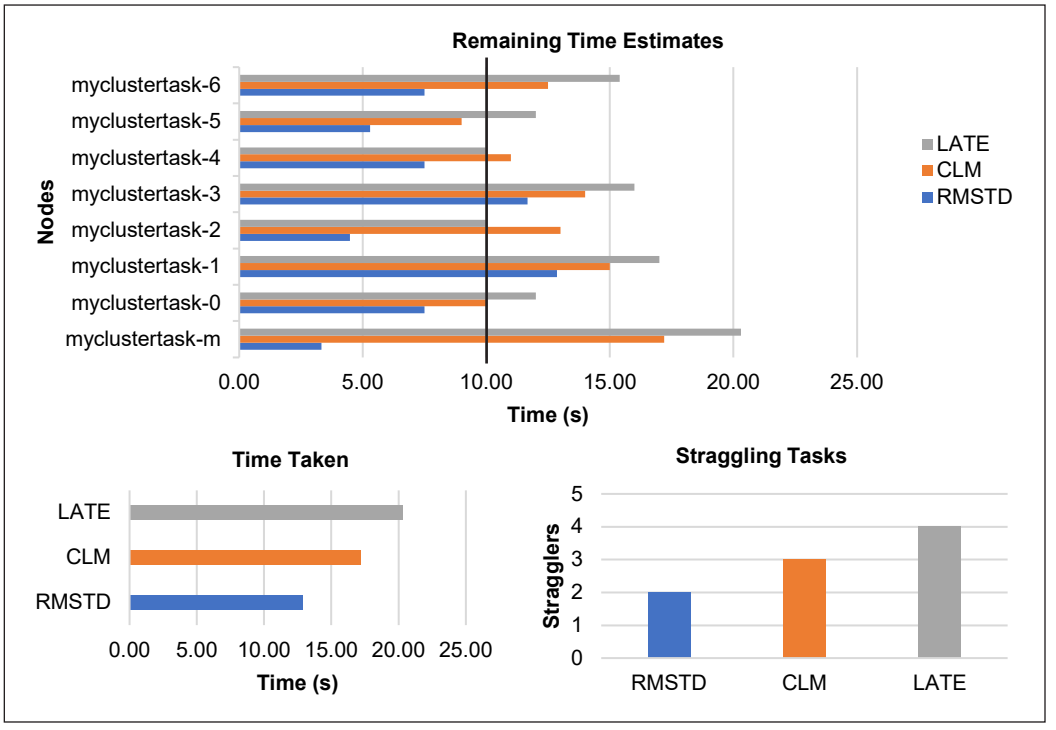


*Figure 5*. Experimental result with data size of 10 GB text file

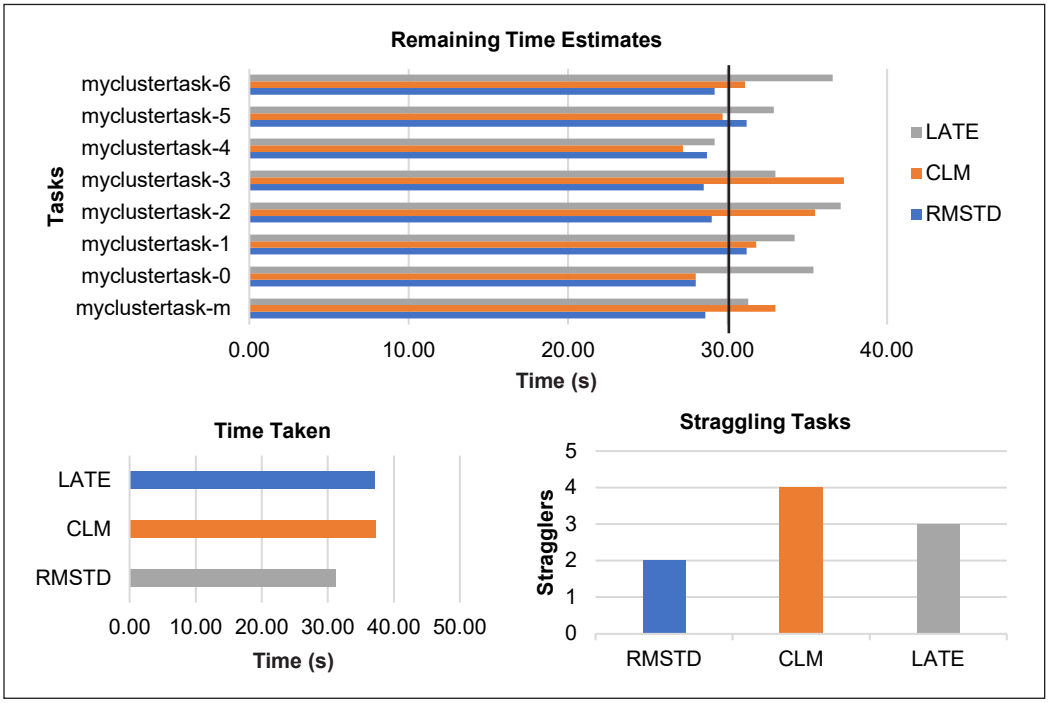


*Figure 6.* Experimental result with data size of 20 GB text file

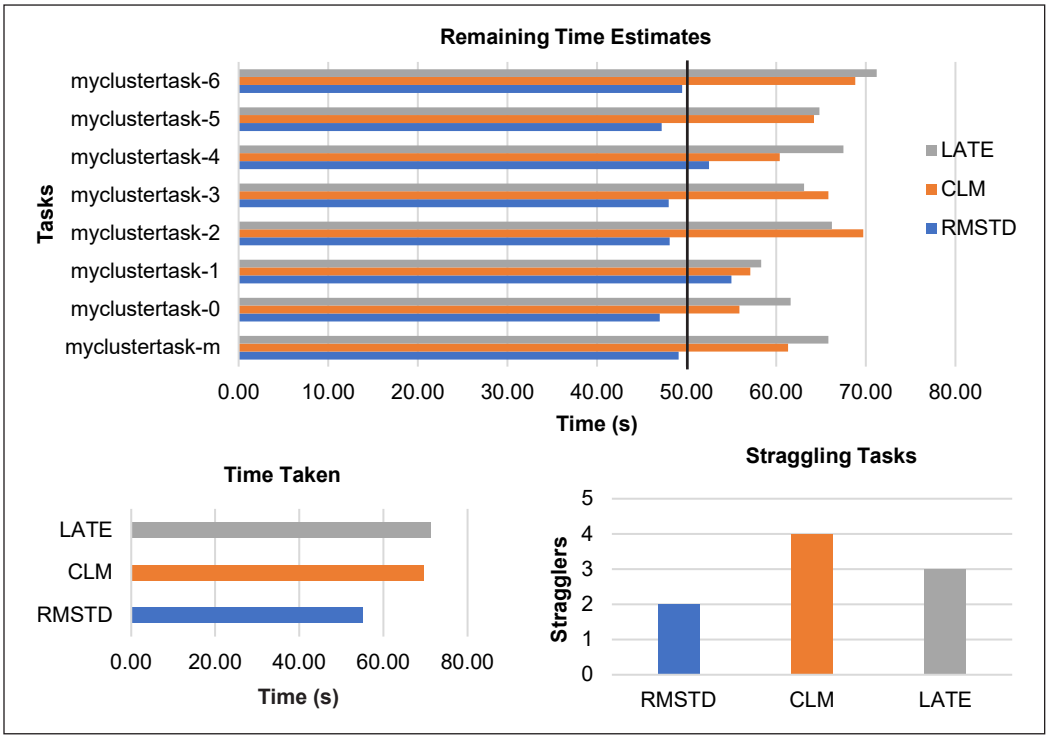


*Figure 7.* Experimental result with data size of 30 GB text file

on this calculation. RMSTD through ART provides a more reliable and accurate indicator of work advancement.

2. Better straggler detection: Straggling tasks take significantly longer than the average or projected completion time. RMSTD makes detecting tasks that deviate from the norm and demonstrate straggling behaviour easier. This comparison aids in differentiating between jobs that may be suffering true performance concerns and those that are advancing at a normal rate.
3. Smoothing out fluctuations: RMSTD provides a more reliable and smoothed estimate of the task's remaining time compared to other RT approaches. It considers the task's historical behaviour, considering the average time it has taken to accomplish identical pieces of work. It mitigates the impact of short-term oscillations or outliers in the Remaining Time, resulting in a more reliable estimate of job completion. Outliers and skews do not overly influence the average, and this can make it easier to identify tasks that are consistently taking longer than expected.
4. Improved decision-making: When deciding on tasks, resources, or task scheduling, RMSTD offers a more trustworthy base in giving a more informed perspective on the anticipated completion timeframes of tasks by taking the past

average into account. Through this, it is easier to allocate resources more wisely and develop better task management plans, ultimately improving the efficiency of the work.

## CONCLUSION

RMSTD provides a more stable, accurate, and reliable measure of task progress. It helps identify straggling tasks more effectively and facilitates better decision-making in Hadoop environments. It also offers a better advantage in detecting straggling tasks in Hadoop, and it has proven to be a useful approach for raising the effectiveness and performance of massively parallel data processing systems. Identifying stragglers and taking proactive steps to reduce their impact on job completion timeframes is feasible by continuously monitoring their progress and calculating the remaining execution time for tasks. In distributed computing systems, this strategy helps to optimise resource usage, shorten job execution times, and improve user experience.

## ACKNOWLEDGEMENT

The authors acknowledged the contributions of all the authors to the successful completion of this article. The authors would also like to express their gratitude to Universiti Teknologi Malaysia for providing an enabling environment to carry out the study.

## REFERENCES

Ananthanarayanan, G., Kandula, S., Greenberg, A., Stoica, I., Lu, Y., Saha, B., & Harris, E. (2019). Reining in the outliers in MapReduce clusters using Mantri. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)* (pp. 265-278). USENIX Association.

Chen, Q., Liu, C., & Xiao, Z. (2014). Improving MapReduce performance using smart speculative execution strategy. *IEEE Transactions on Computers*, *63*(4), 954-967. https://doi.org/10.1109/TC.2013.15

Dai, W., & Bassiouni, M. (2013). An improved task assignment scheme for Hadoop running in the clouds. *Journal of Cloud Computing*, *2*, Article 23. https://doi.org/10.1186/2192-113X-2-23

Dai, W., Ibrahim, I., & Bassiouni, M. (2016). Improving load balance for data-intensive computing on cloud platforms. In *2016 IEEE International Conference on Smart Cloud (SmartCloud)* (pp. 140-145). IEEE Publishing. https://doi.org/10.1109/SmartCloud.2016.44

Dai, W., Ibrahim, I., & Bassiouni, M. (2017). An improved straggler identification scheme for data-intensive computing on cloud platforms. n *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)* (pp. 211-216). IEEE Publishing. https://doi.org/10.1109/CSCloud.2017.64

Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, *51*(1), 107-113. https://doi.org/10.1145/1327452.1327492

Ghare, G. D., & Leutenegger, S. T. (2005). Improving speedup and response times by replicating parallel programs on a SNOW. In D. G. Feitelson, L. Rudolph, U Schwiegelshohn (Eds.), *Job scheduling strategies for parallel processing*. *JSSPP 2004*. *Lecture Notes in Computer Science* (pp. 264-287). Springer. https://doi.org/10.1007/11407522_15

Javadpour, A., Wang, G., Rezaei, S., & Li, K. C. (2020). RETRACTED ARTICLE: Detecting straggler MapReduce tasks in big data processing infrastructure by neural network. *Journal of Supercomputing*, *76*, 6969-6993. https://doi.org/10.1007/s11227-019-03136-6

Katrawi, A. H., Abdullah, R., Anbar, M., & Abasi, A. K. (2020). Earlier stage for straggler detection and handling using combined CPU test and LATE methodology. *International Journal of Electrical and Computer Engineering*, *10*(5), Article 4910. https://doi.org/10.11591/ijece.v10i5.pp4910-4917

Katrawi, A. H., Abdullah, R., Anbar, M., AlShourbaji, I., & Abasi, A. K. (2021). Straggler handling approaches in MapReduce framework: A comparative study. *International Journal of Electrical and Computer Engineering*, *11*(1), 375-382. https://doi.org/10.11591/ijece.v11i1.pp375-382

Ketu, S., Mishra, P. K., & Agarwal, S. (2020). Performance analysis of distributed computing frameworks for big data analytics: Hadoop vs Spark. *Computacion y Sistemas*, *24*(2), 669-686. https://doi.org/10.13053/CyS-24-2-3401

Kumar, G., Mohan, S., & Nagesh, A. (2021). An ensemble of feature subset selection with deep belief network based secure intrusion detection in big data environment. *Indian Journal of Computer Science and Engineering*, *12*(2), 409-420. https://doi.org/10.21817/indjcse/2021/v12i2/211202101

Ouyang, X., Garraghan, P., McKee, D., Townend, P., & Xu, J. (2016). Straggler detection in parallel computing systems through dynamic threshold calculation. In *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)* (pp. 414-421). IEEE Publishing. https://doi.org/10.1109/AINA.2016.84

Ouyang, X., Wang, C., Yang, R., Yang, G., Townend, P., & Xu, J. (2018). ML-NA: A machine learning based node performance analyzer utilizing straggler statistics. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)* (pp. 73-80). IEEE Publishing. https://doi.org/10.1109/ICPADS.2017.00021

Phan, T. D., Pallez, G., Ibrahim, S., & Raghavan, P. (2019). A new framework for evaluating straggler detection mechanisms in MapReduce. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, *4*(3), Article 14. https://doi.org/10.1145/3328740

Qiang, Y., Li, Y., Wei, W., Pei, B., Zhao, J., & Zhang, H. (2014). A job scheduling policy based on the job-classification and dynamic replica mechanism. *Information Technology Journal*, *13*(3), Article 501. https://doi.org/10.3923/itj.2014.501.507

Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R., & Stoica, I. (2019). Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008* (pp. 29-42). USENIX Association.