Improving MapReduce Speculative Executions with Global Snapshots

Ebenezer Komla Gavua[1], Gabor Kecskemeti[2]
Institute of Information Technology, Miskolc-Egyetemvaros 3515, Miskolc, Hungary
Computer Science Department, Koforidua Technical University, Koforidua, Ghana[1,2]
Department of Computer Science
Liverpool John Moores University, Liverpool, UK[2]

***Abstract*—Hadoop's MapReduce implementation has been employed for distributed storage and computation. Although efficient for parallelizing large-scale data processing, the challenge of handling poor-performing jobs persists. Hadoop does not fix straggler tasks but instead launches equivalent tasks (also called a *backup task*). This process is called Speculative Execution in Hadoop. Current speculative execution approaches face challenges like incorrect estimation of tasks run times, high consumption of system resources and inappropriate selection of backup tasks. In this paper, we propose a new speculative execution approach, which determines task run times with consistent global snapshots and K-Means clustering. Task run times are captured during data processing. Two categories of tasks (i.e. fast and stragglers) are detected with K-Means clustering. A silhouette score is applied as decision tool to determine when to process backup tasks, and to prevent extra iterations of K-Means. This helped to reduce the overhead incurred in applying our approached. We evaluated our approach on different data centre configurations with two objectives: i) the overheads caused by implementing our approach and ii) job performance improvements. Our results showed that i) the overheads caused by applying our approach is becoming more negligible as data centre sizes increase. The overheads reduced by 1.9%, 1.5% and 1.3% (comparatively) as the size of the data centre and the task run times increased, ii) longer mapper tasks runs have better chances for improvements, regardless of the amount of straggler tasks. The graphs of the longer mappers were below 10% relative to the disruptions introduced. This showed that the effects of the disruptions were reduced and became more negligible, while there was more improvement in job performance.**

***Keywords*—MapReduce; Hadoop; speculative executions; stragglers; consistent global snapshots; K-means algorithm**

I. Introduction

The Hadoop software environment provides a widespread implementation for distributed data storage and MapReduce computing [1], [2], [3]. However, the challenge of handling poor-performing jobs persists. Hadoop launches equivalent tasks (also called a *backup task*) in place of straggler tasks to finish the computation faster. This process is Hadoop's speculative execution [4].

Previous research into speculative execution has shown efforts to improve job performance in MapReduce. Past strategies such as LATE [5] and MCP [6] recognize straggler tasks based on self-estimation of the tasks' remaining time. SAMR [7] and ESAMR [8] use historical information to classify nodes into slow map and reduce nodes. SECDT [9] predicts the remaining time of running tasks based on real-time information on tasks. However, these previous approaches have various problems. Most have challenges with the accurate estimation of the remaining time of slow tasks. Some have significant overheads during the estimation of straggler tasks' remaining times.

Our work proposes a new speculative execution approach, which estimates task runtimes with consistent global snapshots and K-Means clustering. Task progress is captured consistently during data processing. Two categories of tasks (fast and straggler) are identified with K-Means. A silhouette score is applied as decision tool to determine when to process backup tasks. This helped to reduce the overhead incurred in applying our approached since, the stragglers were quickly detected and rescheduled.

We evaluated our approach on different data centre configurations. The data centre configurations were selected after considering Hadoop cluster requirements from the industry. We focused our experiments on two objectives: (i) the overheads caused by implementing our approach and (ii) job performance improvements. Two categories of backup tasks were considered in our experiments. (i) backup tasks that can transfer their states when shifting from one node to another and (ii) backup tasks that need to restart after their transfer (i.e. tasks supported by Hadoops original speculative execution).

We experimented with mappers as there are typically more of them in a Hadoop application than reducers. Since reducer handling is done the same way as mapper handling in Hadoop, our results are applicable to both. We also focused on different durations of mappers, which provided the details of how long the mappers took to process data. From these, we concluded on the following (i) the overheads caused by applying our approach is becoming more negligible as data centre sizes increase. The overheads reduced by 1.9%, 1.5% and 1.3% (comparatively) as the size of the data centre and the task run times increased. (ii) longer mapper tasks runs have better chances for improvements, regardless of the amount of straggler tasks. The graphs of the longer mappers were below 10% relative to the disruptions introduced. This showed that the effects of the disruptions were reduced and became more negligible, while there was more improvement in job performance.

The remainder of this paper is structured as follows. In Section II, we reviewed concepts and related works about MapReduce and Speculative Executions. In Section III, we present our methodology for detecting straggler tasks and proposed improvements. In Section IV, we discussed the

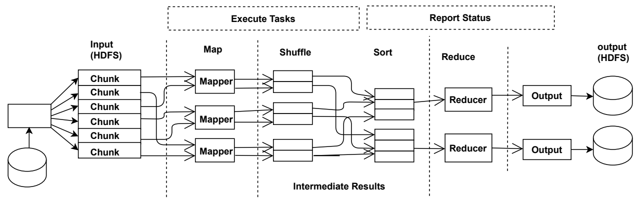


Fig. 1. MapReduce structure.

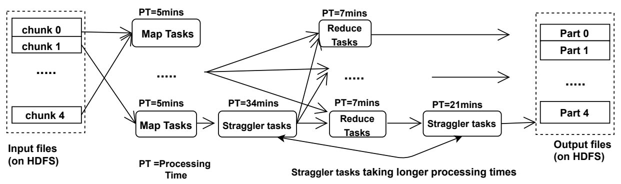


Fig. 2. Speculative execulation.

procedure for evaluating our approach via experiments and we analysed the results. Section V concludes the paper with recommendations for future work.

II. Concepts and Related Works

A. Concepts

In this subsection, we briefly discuss few concepts related to the MapReduce programming model.

1) MapReduce: MapReduce (MR) is a programming model for massive data computing used in Apache Hadoop. MapReduce is used for writing applications that process and analyse large data sets. These applications ran in a parallel fashion on large clusters in a scalable and fault-tolerant manner. A MapReduce job breaks and divides the input data into chunks which are first processed by the "Map phase" in parallel and then by the "Reduce phase" [10], [11] as seen in Fig. 1.

2) Hadoop: Apache Hadoop is an open-source software implementation of MapReduce. The core of Hadoop includes a distributed file system, and a MapReduce processor [1]. The Hadoop distributed file system (HDFS) works closely with MapReduce by distributing storage and computation across large clusters [12], [13]. During job processing on Hadoop, if a task of a job requires an abnormally long execution time, the total completion time of the job is affected. Such a task is called a straggler task. MR reruns straggler tasks on a different machine to finish the computation faster. The process of diagnosing straggler tasks and assigning them to other nodes is called *speculative execution* [14]. These faults are mainly due to IO contentions, background services, hardware behaviours, unbalanced load or uneven distribution of resources and other reasons [15]. Some straggler tasks run significantly slower than other tasks as shown in Fig. 2, where the straggler tasks take more processing times than the normal MR tasks [4].

3) Prior speculative execution strategies: A couple of research activities have been conducted to solve poor-performing tasks.

The Hadoop Naïve Method was implemented with the Hadoop architecture. However, most of the tasks processed during runtime were detected as slow tasks and processed as backup tasks. This affected job completion because there was no improvement in job completion time after processing the backup tasks. Also, this strategy is not suitable in heterogeneous environments. Therefore, an approach that distinguishes straggler tasks from the normal tasks during job processing will ensure job performance improvements.

Zaharia et al. [5] developed the Longest Approximate Time To End (LATE) algorithm. LATE is a simple, robust scheduling

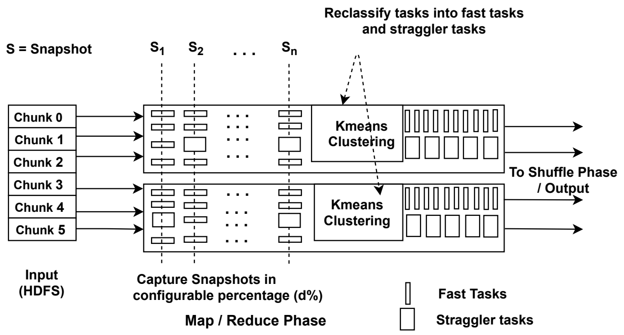


Fig. 3. Structure of task performance monitoring algorithm.

algorithm that uses estimated finish times to detect straggler tasks. LATE is not suitable in heterogeneous environments. Therefore, a dynamic approach that works in all types of environments will help estimate the task runtime to ensure the improvement of job performance.

Chen et al. [7] proposed a Self-Adaptive MR Scheduling Algorithm (SAMR). SAMR uses historical information to classify nodes into the slow map- and reduce-nodes. This makes SAMR dependent on previous tasks information. Therefore, an approach that applies the information of current tasks without depending on previous nodes will be welcomed in the research community.

Sun et al. [8] designed an Enhanced Self-Adaptive MR Scheduling Algorithm (ESAMR) as an improvement on SAMR by utilising the K-means clustering algorithm to classify historical information. Therefore, the reliance of ESAMR on previous task information makes it only applicable when there is historical information. Moreover, the K-means clustering algorithm utilised was not validated to determine the straggler tasks. Therefore, an approach that is not affected by changes in dataset and validates the kmeans clustering will allow users to better assess tasks behaviours.

Chen et al. [16] proposed the Maximum Cost Performance approach, which considers the cost performance of cluster computing resources to estimate the slow tasks. However, in the map phase, task satisfying data localisation executes faster than those not satisfying data localisation. This provide an unfair comparison between the tasks at the same level. Therefore, an approach that considers all tasks at the same level will ensure an appropriate estimation of task run times.

Huang et al. [9] proposed a new Speculative Execution Algorithm based on C4.5 Decision Tree (SECDT) to improve predicted execution times among previous research resulting in poor job performance. However, navigating the decision tree implemented by this strategy is prone to significant overheads. Therefore, an approach that determines task run times via snapshot captures will enable the improvement of job performance.

In summary, the existing speculative execution strategies still encounter challenges in managing straggler tasks in Hadoop. We now discuss the design of our proposed approach.

III. Methodology

This section focuses on the design of our approach which is designed to improve job performance on MapReduce Hadoop (MRH). The approach consists of two algorithms that are interconnected to ensure correct determination of task run times, appropriate selection of backup tasks and reduction in the consumption of system resources. The goals of this section are:

- To design an algorithm that captures task run times during data processing on mappers and reducers. This is achieved by repetitive capturing of the task run times at specific intervals.
- To design an algorithm that monitors task performance on their nodes to foster the rescheduling of straggler tasks to available nodes for reprocessing.
- To implement K-means clustering algorithm to determine straggler tasks. The K-means clustering algorithm is applied with the Silhouette Coefficient to validate the outputs of the clustered data sets.
- To assess the algorithms on scalable configurations of MRH to prove their applicability. A survey of industry and real-life MRH configurations is conducted to ensure that the solution is applicable in industry.

A. Consistent Global Snapshots on MapReduce

This approach comprises of snapshot capturing and task performance monitoring algorithms as seen in Algorithms 1 to 2, and Fig. 3 to 5. Fig. 4 shows state transitions during the capturing of task run times. The two algorithms work together to ensure that straggler tasks are detected correctly and processed as backup tasks.

This approach is designed to dynamically collects real-time data from all types of environments. The collected real-time data fosters the early detection of straggler tasks to reduce high consumption of system resources. Moreover, this approach is applicable in most environments compared to a few existing approaches which struggle in heterogeneous environments. Additionally, some of the existing approaches have limitations with accurate estimation of remaining time of straggler tasks. Also, some have significant overheads during the estimation of straggler tasks' remaining times. The details of the algorithms are discussed below.

1) Snapshots capturing algorithm: Algorithm 1 is applied to explain the snapshots capturing process. The algorithm is designed with specific parameters to foster its comprehension. T_s and C_s are utilised to model tasks state transitions (*Task_state*) and the snapshot capturing state transitions (*Snap_state*) during job processing as seen in Fig. 5 and 4. These two parameters

Algorithm 1 *Snapshot Capturing Algorithm*

```
Require: Variables: $T_s = Task\_state, C_s = Snap\_state,$
    $N = Node$
Require: Variables: $Q_I = Tasks\_Instances,$
    $i = counter\ for\ Q_I$
Require: Variables: $G_s = captured\_snapshots$
Require: SnapStateFunction:
    $SnapState : C_s \rightarrow \{snap\_ready,$
    $snapping, snap\_paused, snap\_completed\}$
 1: for $i < Q_I$ do
 2:   if $T_s = ready$ then
 3:     $C_s := snap\_ready$   // Snap_state updates to ready
        when tasks processing begins.
 4:   else
 5:     $C_s := pause\_snapping$
 6:     $check\ the\ system\ and\ restart\ the\ task$
 7:   end if
 8:   while $T_s = running$ do
 9:     $C_s := snapping$ // Snap_state updates to snapping
        during tasks processing.
10:     $Save\ the\ captured\ snapshots$
11:     $G_s = G_s + 1snapping$
12:   end while
13:   $i = i + 1$   // Tasks are monitored until they are
      processed.
14: end for
15: if $T_s = terminated$ then
16:   $C_s := pause\_snapping$   // Snap_state pauses when a
      task is terminated with task_state updated to terminated.
17: end if
18: while $i <= Q_I$ do
19:   $C_s := snapping$   // Task run times capturing contines
      until all tasks are processed.
20:   $Save\ the\ captured\ snapshots$
21:   if $T_s = completed$ then
22:     $C_s := snapping\_completed$   // Snap_state updates
        to snap_completed when tasks processing ends.
23:     $Save\ the\ captured\ snapshots$
24:     $G_s = G_s + 1snapping$
25:   end if
26:   $i = i + 1$
27: end while
```

together with the *SnapStateFunction* and the *TaskStateFunction* help to describe the status of task processing and snapshot capturing at any specific period.

Algorithm 1 initialises with task processing to foster the capturing of task run times as seen in Fig. 3. Nodes (*N*) are monitored before task processing begins. This is done to capture the commencement of task processing (*start times*), as seen in the snapshot capturing state diagram in Fig. 4. When job processing commences, data is uploaded into the system for task processing to commence.

SnapStateFunction is activated, which causes *Snap_state* (C_s) to be updated to *ready_snap* as seen in lines 1 to 3 of Algorithm 1. However, if a task is not ready (due to a fault), C_s is updated to *pause_snapping* and the system is checked (for the task to be restarted) as seen in lines lines 4 to 7.

While the updated data is being processed, C_s is updated

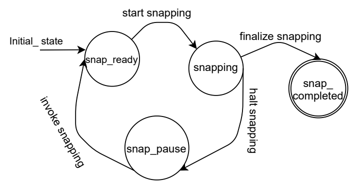


Fig. 4. Snapshot capturing state transition diagram.

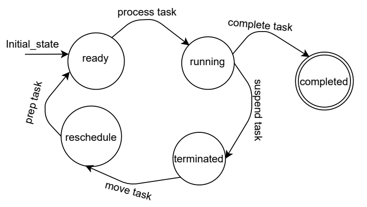


Fig. 5. Tasks state transition diagram.

from *snap_ready* to *start_snapping* as as seen in lines 8 to 13. Task run times are captured and saved as seen lines 9 to 10. Task run times are captured on all nodes and saved on snapshots text files. Snapshots are captured repeatedly for all the tasks running on nodes. The accumulated local snapshots captured constitute the global snapshots (G_s). The details captured include (i) task start-time, (ii) task completion-time, (iii) the node (N) on which the task is running, (iv) task identification, and (v) task status. The task start-time is the specific time the processing of a particular task *t* commences. While the completion-time is the specific time the processing of *t* ends. The task identification is the unique key given to every task *t* when their processing commences. The task status is reflective of the current state of *t* as seen in the state transition Fig. 5.

Additionally, the *TaskStateFunction* is activated, which causes *task_state* (T_s) to be updated from *ready* to *running* as seen in the task state diagram in Fig. 5. T_s remains unchanged until all the tasks are completely processed. Then, it transitions from *running* to *completed*. However, when a task's run time is unnecessarily longer that expected, the task is suspended, which causes T_s to be updated to *terminated* and C_s to *pause_snapping* as seen in lines 15 to 17.

When a configurable percentage of the tasks have been processed with captured run times as seen in Fig. 3; K-means clustering algorithm is employed to classify the captured data on the snapshots text files, to determine the straggler tasks. The straggler tasks identified are then processed as backup task on available nodes. This causes T_s to be updated from *terminated* to *rescheduled* as seen in Fig. 5.

When the tasks rescheduling is completed, the re-processing of the backup tasks commences. This causes T_s to transition from *rescheduled* to *ready*. The backup tasks are processed together with the snapshot capturing until all the tasks are completely processed as seen in lines 18 to 27 of Algorithm 1.

2) Task performance monitoring algorithm: The task performance monitoring and the snapshot capturing algorithms work concurrently to ensure job performance improvement, as seen in Fig. 3.

Algorithm 2 $Task\ Performance\ Monitoring\ Algorithm$

```
Require: Variables: $Q_I = Tasks\_Instances,$
    $T_R = running\ tasks,\ i = counter\ for\ Q_I$
Require: Variables: $N_i = Node,\ A_n = AvailNodes,$
    $T_C = completed\ tasks$
Require: Variables: $Q_I = \{t_1, t_2, t_3, ...t_n\},\ T_s = Task\_State$
Require: Variables: $T_{ET} = task\ execution\ time,$
    $T_{MET} = task\ maximum$
     $execution\ time$
Require: TaskStateFunction:
    $TaskState : T_s \rightarrow \{ready, running,$
    $terminate, reschedule, completed\}$
 1: $Begin\ Tasks\ Processing\ in\ the\ Map\ or$
     $Reduce\ Phase$
 2: for $i <= Q_I$ do
 3:   $status \leftarrow checkTasksStatus$
 4:   switch $(status)$
 5:   $T_s := ready$   // task is ready for processing.
 6:   case $still\_Running$:
 7:     $monitor\ the\ progress\ of\ the\ task$
 8:     $T_s := running$
 9:     if $T_{ET} > T_{MET}$ then
10:       $terminate\ the\ task$
11:       $T_s := terminated$   // straggler tasks are stopped.
12:       $reschedule\ straggler\ tasks\ on\ available\ nodes$
13:       $T_s := rescheduled$
14:       $T_{\rm R} \leftarrow (T_{\rm R} + t)$ // Running tasks list increased.
15:     else
16:       $process\ all\ the\ tasks$
17:     end if
18:   case $finished\_Running$:
19:     $Tasks\ completely\ processed$
20:     $T_s := completed$
21:     $Output\ results$
22:     $A_{\rm n} \leftarrow (A_{\rm n} + N_i)$   // Available nodes list increased
        for backup task.
23:     $T_C \leftarrow (T_{\rm C} + t)$   // Monitor the tasks completed.
24:     $Q_i \leftarrow (Q_{\rm i} - t)$   // Task instance list is reduced.
25:   end switch
26:   if $|Q_{\rm i}| = 0$ then
27:     $Stop\ tasks\ monitoring$
28:   else
29:     $Continue\ with\ tasks\ monitoring$
30:   end if
31:   $i = i + 1$   // Counter is increased to process all tasks.
32: end for
```

Algorithm 2 is applied during task processing to monitor and evaluate task performance. When data processing begins, all tasks i.e., $t = t_1, t_2, t_3, \ldots t_n$ are expected to process data at the same rate. These tasks are allocated processes on compute nodes as seen in lines 1 to 2.

The *TaskStateFunction* is activated which causes T_s to be updated to *ready* as seen in line 5. Tasks-instances (Q_I) are monitored to determine whether they are still running (T_R) or are completely processed (T_C) as seen in lines 6 and 18.

During task processing, tasks which have relatively longer run times than the maximum execution times (T_{MET}) of the

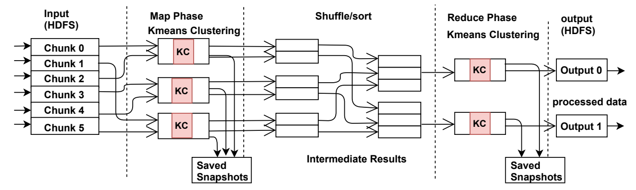


Fig. 6. Structure of algorithm implementation with k-means clustering.

tasks being processed are terminated as seen in lines 6 to 10. This causes T_s to be updated to *terminated* as seen in line 11 and Fig. 5. The K-means algorithm is applied to cluster all the captured task run times as seen in Fig. 6 on page 16. The tasks identified as straggler tasks are rescheduled as seen in lines 12 to 14. T_s transitions from *terminated* through *rescheduled* to *ready* as seen in Fig. 5.

The states of the tasks which do not exhibit relative longer run times, transition from *ready* through *running* to *completed*. This enables their compute nodes to be availed for processing backup tasks, and reduces the number of task instances. These processes are seen in lines 18 to 25.

A vital aspect of this algorithm is the monitoring of task instances (Q_I). The number of tasks are monitored throughout their processing stages. When the number of active tasks are exhausted, job processing ends. Otherwise, the task processing continues until the jobs generated are completely processed seen in lines 26 to 32.

B. Identifying Straggler Tasks with K-Means Clustering Algorithm

The identification of straggler tasks during job processing was a challenge that required addressing in our approach. This was achieved via the adoption of a clustering technique.

Clustering was considered because it is the type of unsupervised machine learning where its goal is to partition sets of objects into groups called *clusters*. These groups can be mutually exclusive or they may overlap, depending on the approach used. It is in contrast to the supervised learning techniques where the goal is to make predictions about output value *y* given an input object or instance *x* [17]. This made the choice of clustering suitable for our approach since there was no need for training any data set to achieve our groupings.

Additionally, we considered K-means clustering as the clustering technique for our approach because it is a hard clustering algorithm which delivers mutually exclusive groupings. K-means partitions a set of n objects into *k* clusters, so that the resulting intra-cluster similarity is high but the inter-cluster similarity is low [18]. It was the most suitable clustering algorithm for our approach since two distinct groups are required; thus fast tasks and straggler tasks.

Our approach applied the K-means clustering algorithm to categorise task run times (dataset) received from the snapshot capturing algorithm. The dataset saved on snapshots text files during the map or the reduce phases are clustered into fast and straggler tasks as seen in Fig. 6. K-means optimizes the distance between the task run times to their centre points, as seen in eq. (1) [19].

$$J(V) = \sum_{i=1}^{C_d} \sum_{j=1}^{C_i} (|x_i - y_j|)^2 \tag{1}$$

Also, the K-means algorithm is implemented with a validation technique as a decision-making tool in our work. It directs whether to process backup tasks or not. There are cases where the dataset presented for clustering is uniform. However, K-means still tries to cluster it. Thus, clustering results require validation to determine the goodness of fit of the clusters created as seen in Fig. 7.

In order to ensure the effective creation of clusters, four clustering validation techniques were considered. These are Dunn [20], Davie-Bouldin, Calinski-Harabasz indices, and the Silhouette score [21]. However, the silhouette score was selected for our approach. The first three were not implemented because of the following: first, although the Calinski-Harabasz index defines how dense and separated a cluster is, the absence of upper- and lower-bounds ranges made it inapplicable. Second, the Davies-Bouldin index utilizes zero (0) as the upper bound; and values closer to zero indicate a better partition. Moreover, the Davies-Bouldin index did not have a lower bound. In the case of the Dunn index, higher indices indicate better clustering. However, the absence of a lower bound makes it inapplicable in our context. Since, without a closed range of clustering validation values, a deterministic algorithm based on them would be unreliable. Also, the presence of the upper-lower bounds fosters faster determination of the goodness of fit of clusters created. Its absence introduces extra overheads into our strategy and makes the choice inappropriate for our research. Nevertheless, the Silhouette score S_i utilizes an easy-to-evaluate metric to determine the goodness of the clustering. Silhouette score values have a closed range of -1 to 1 [22]. Thus, the silhouette score was chosen for this work.

Algorithm 3 is utilised to identify the suitability of a clustering output for fast and straggler tasks. This algorithm validates the silhouette scores after the clustering exercise. It utilizes the values to decide whether to process backup tasks or not. For instance, Fig. 7 shows two very close data clusters which is difficult to ascertain the fast tasks or poor-performing ones. However, Fig. 8a to 8d on page 17 show well defined data clusters which will require rescheduling of straggler tasks.

Algorithm 3 $Kmeans\ Clustering\ Validation\ Algorithm$

```
Require: $Set\ the\ S_i\ threshold\ lower-bound\ as$
    $Z_x = 0.685$
Require: $Set\ the\ S_i\ threshold\ upper-bound\ as$
    $Z_y = 0.99$
1: $Initialize\ the\ clustering\ output\ as\ an\ array\ A[k]$
2: for $k = 1$ to $A.length$ do
3:     if $S_i > Z_x\ \&\ S_i \leq Z_y$ then
4:         $Reschedule\ tasks\ on\ available\ nodes$
5:     else
6:         $Run\ the\ tasks\ on\ current\ nodes$
7:     end if
8: end for
```

The results from the silhouette score are utilized to determine the goodness of the K-means clustering. If the silhouette

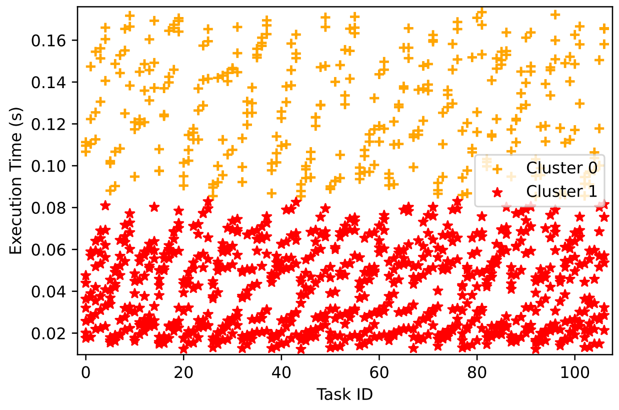


Fig. 7. Data processing execution times on 20 nodes with 8 individual cores.

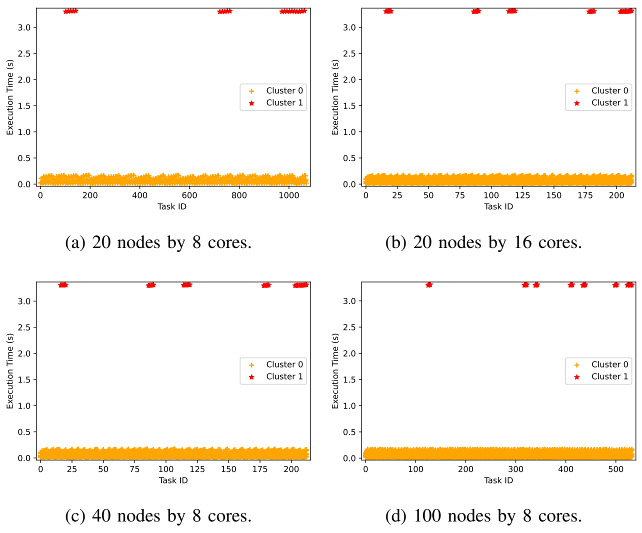


Fig. 8. K-means clustering of run times from experimental scenarios.

score is higher than a threshold lower-bound value Z_x but less than a threshold upper bound value Z_y; backup tasks are required for the cluster with the straggler tasks as seen in line 3. Otherwise, no intervention is applied to the task executions. The remainder of the tasks are then processed on their original nodes as seen in line 6.

Our silhouette score threshold (i.e., lower and upper bound) values were determined from several clustering experiments carried out on our dataset to ensure that the range given satisfies all possible scenarios.

IV. Evaluation

The goal of our evaluation was to assess our approach via two major experiments to prove its applicability. They are (i) strategy implementation overheads experiments (ii) job performance experiments. The experiments enabled us to draw the necessary conclusions on the benefits of using our approach. The experiments aimed to detect and process straggler tasks as backup tasks to improve job performance. The experimental setup is described below.

A. *Experimental Setup*

The following objectives were considered in order to achieve the goal of the experiment:

- To determine the start-time of task processing.
- To determine the completion-time of task processing.
- To capture snapshots of task execution times.
- To capture task run times at specific intervals.
- To terminate straggler tasks.
- To restart straggler tasks on available nodes.

The first two bullet points foster the determination of the overheads introduced by this approach. The last four bullet point ensures the measurements of the jobs improvement performance. The termination and restart of the straggler tasks reduces the high consumption of system resources. The experiment was conducted on our extension of HDMSG MapReduce (a MapReduce simulator with Simgrid as the main backbone) available on GitHub[1].

In order to utilise HDMSG for the development of this approach, a couple of features had to be added to the simulator to make it applicable. Several methods and classes were created for specific functions. A task monitoring method was created to monitor the tasks running on nodes on the MapReduce Hadoop cluster. This method was responsible for terminating long running tasks. A task rescheduling method was created to move the terminated task to available nodes to be processed as backup tasks. A snapshot capturing method was created to capture the start times and completion times of tasks on nodes. These captured task run times were saved on text files for k-means clustering. A disruption injection method was created to send extra tasks unto arbitrary nodes to serve as background activities. These extra tasks caused the map or reduce task on those nodes to experience longer run times. The methods for the creation of map and reduce tasks were extended to foster the scalability of the framework. A node scheduling class was created to foster the chronological processing of data nodes to enable the capturing of snapshots. Also the class fosters the selection of available nodes as exhibited in the Hadoop infrastructure.

Tasks were divided into ten-equal-length subtasks to simulate the snapshot capturing behaviour with simgrid. This was done to ensure that the snapshot could capture the start-times and completion-times of subtasks. The experiments required the capturing of task processing timelines (i.e. when a particular subtask ends and when the other begins). Therefore, dividing a task into ten-equal-length allowed the runtime behaviours of each subtask to be monitored and captured as snapshots.

In setting up the experiments, the infrastructure of HDMSG with Simgrid were defined. The infrastructure was defined in terms of the following: the number of nodes, CPU cores, bandwidth, latency metrics, and the nodes' speed. Additionally, the number of mappers and reducers, file input size (in megabytes), and block size (HDFS chunk size in megabytes) were configured to foster MR computations.

[1]https://github.com/EbenezerKomlaGavua/MapReduce_Snapshots

To determine real life MR cluster infrastructure and application configurations, two surveys about Hadoop cluster requirements were carried out. The first survey focused on identifying typical hadoop configurations and the second one focussed on organisations actively utilising Hadoop clusters for their data processing in industry.

Several keywords such as hadoop clusters (requirements), industry cluster infrastructure (setup, configurations) were employed on several search engines to locate current MRH cluster configurations. The first survey identified Hadoop cluster configurations such as basic or standard deployments, advances deployments, hadoop cluster hardware recommendations for batch processing, in-memory processing, medium data size and large data size. The first survey found that the most used CPU speed was 2-2.5Ghz, data block sizes were between 128-256MB, network bandwidth was 1-10Gbps, cluster nodes was 4-40, number of mappers and reducers were 5-12 per node, disk capacity range was 32 GB to 1.2TB and total system memory was 16-512 GB. All hadoop cluster configurations modes were fully distributed.

The second survey found over one hundred and twenty top companies actively utilising hadoop clusters from several websites. Notable companies amongst the list include Alibaba, AOL, Yahoo, Spotify, Last.fm, Ebay, University of Glasgow-Terrier Team and Criteo. From this list, the modal CPU cores per node identified was eight and the modal cluster nodes was forty.

The findings of the survey fostered the selection of four infrastructure scenarios (displayed in Table I) for our experiments. The experimental scenarios comprise data nodes that ranges from 20 to 100 nodes. The range of CPU cores was 8 to 16. Aside the values displayed in the infrastructure scenarios table, network bandwidth of 10Gbps was simulated for all infrastructure scenarios. The smallest data block size employed was 128MBs. All the experiments were run on fully distributed hadoop cluster mode to foster conformance with industry standard.

The details identified from the survey ensure the modelling of real life applications on the above infrastructures as described below. The number of mappers per node was obtained via eq. (2), as stipulated in[2]:

$$Y = \frac{3Cores}{2}, \tag{2}$$

where Y is a positive rational number that represents the number of mappers per node. $Cores$ is a positive integer which represents the CPU cores per node.

The number of reducers per node was obtained via eq. (3), as stipulated in[3]:

$$R = 0.95 \times N \times T, \tag{3}$$

Where R is a positive rational number representing the number of reducers per node, T is a positive rational number representing the mapred tasktracker reduce tasks maximum value. T is the maximum number of reduce tasks that will be run simultaneously by a task tracker (2 was used, since it is the default maximum value). N is a positive integer representing the number of nodes running on the cluster.

[2]https://data-flair.training/forums/topic/how-one-can-decide-for-a-job-how-many-mapper-reducers-are-required/

[3]https://hadoop.apache.org/docs/r1.2.1/api/org/apache/hadoop/mapred/JobConf.html

TABLE I. EXPERIMENTAL INFRASTRUCTURE SCENARIOS

Features	20N×8c	20N×16c	40N×8c	100N×8c
No. of Nodes	20	20	40	100
No. of Cores	8	16	8	8
Mappers per node	5	5	11	5
Total Reducers	38	38	76	190
Total Mappers	107	213	213	533
Input Size	13696	27264	27264	68224

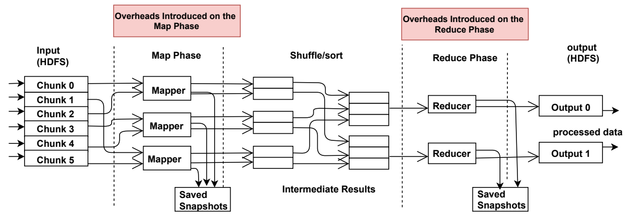


Fig. 9. Structure of algorithm implementation with expected overheads.

The proposed approach applies to both mappers and reducers. However, the evaluation was centred on mappers; since the number of mappers are bigger than reducers. Hence, the effects of our approach are expected to be more on mappers than reducers.

B. Determining the Overheads of our Strategy

This experiment determined the overheads introduced into the infrastructure by the implementation of this approach. The overheads were caused by the effects of the snapshots capturing process on the infrastructure as seen in Fig. 9. The comprehension of the effects of the overheads fosters the appreciation of the challenges and benefits in applying this approach on MRH.

The experiment was conducted on the four data centre scenarios discussed in sub-section IV-A. Mapper tasks with execution times from 0.5 to 2000 seconds were utilized. The range for the experiment was derived via the multiplication of the single values of one, two and five with the power series of ten. The value of negative one produced 0.5 seconds and we scaled the task run times until the graph converged at 2000 seconds. This process was done in order to obtain a scalable range of task run times.

Since this approach involves capturing snapshots during the processing of subtasks, two measurements were taken. These are (i) the commencement of tasks processing and (ii) The completion of task processing. To determine the overhead on a single mapper, the differences between the *completion-times* of the processed portion of the task and the *start-times* of the next portion of that same task are determined (i.e. the period for snapshot capturing). The summation of the differences of these values (i.e. differences between *completion-times* and *start-times*) is subtracted from the task's run times (which is the ten-equal-length subtasks) as shown in Fig. 10. The value realized

TABLE II. STRATEGY IMPLEMENTATION OVERHEADS

	Scenario overheads (%)			
Map time (ms)	20N×8c	20N×16c	40N×8c	100N×8c
0.5	96	66	66	51
1	53	53	53	39
2	45	36	36	23
5	28	29	29	16
10	24	20	20	12
20	18	15	15	8
50	8	7	7	3
100	7	5	5	1
200	5	3	3	∼ 0
500	3	1	1	∼ 0
1000	1	∼ 0	∼ 0	∼ 0
2000	∼ 0	∼ 0	∼ 0	∼ 0

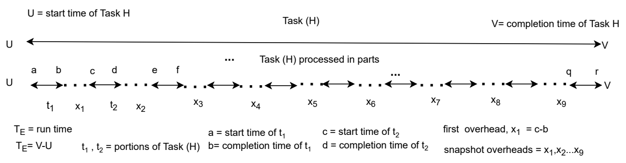


Fig. 10. Strategy overheads.

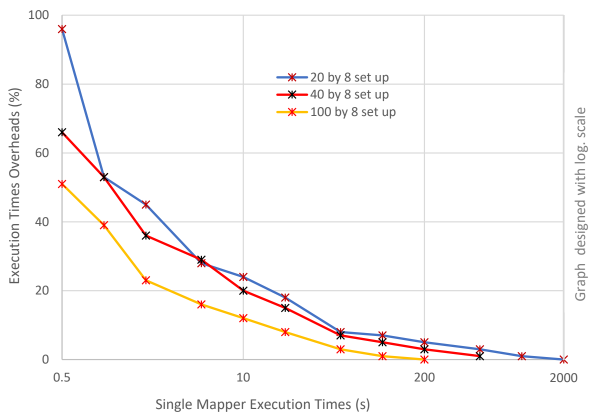


Fig. 11. Cluster overheads.

is the overhead on a single mapper as shown in eq. (4) and Fig. 10.

$$Q^H = 100 - \frac{100(T_E^H - (\sum_{i=1}^{n-1} t_c^{H,i} - t_s^{H,i}))}{T_E^H} \tag{4}$$

Where Q^H is the overhead of applying our approach on a mapper H in percentage. T_E^H is the task runtime of the given mapper. $t_s^{H,i}$ is the time the i^{th} snapshot of the mapper H was started to be captured. Similarly, $t_c^{H,i}$is the time when we finished capturing the snapshot of the same task. Equation 4 is exemplified in Fig. 10. T_E^H is obtained from subtracting U from V. The letters a and c are the start times of the first two subtasks, whilst b and d are the completion times of the first two subtasks. Hence subtracting b from c produces the first gap (x_1) introduced because of our approach. These gaps x_1 to x_9 are summed up and divided by the task run times to generate the overhead of a task.

C. Discussion of the Overheads of our Strategy Experiment Results

The overheads of a single mapper were measured on the four data centre scenarios as seen in Tables II and illustrated in Fig. 11.

- **Scenario 20N×8c:** The impact of applying our approach was gradual. The overheads were high at the initial stages of the experiment. However, the impact of the approach caused the high overheads to reduce gradually with longer runtimes, as seen in Fig. 11. Therefore, in such small infrastructures, our approach is only advisable to use with long run times.
- **Scenarios 20N×16c and 40N×8c:** The two scenarios exhibited similar overhead behaviours during task runs. Therefore, only the 40 nodes by 8 cores set up was shown in Fig. 11. The initial overheads observed were 1.5% lower (relatively) than the 20N×8c data centre scenario. The figure shows that the larger infrastructure scenarios converged faster than the previous. Also, this graph shows that the overheads of larger data centres improve better than smaller ones when our approach is applied. Moreover, the overheads of applying our approach with long run times have a higher chance of improving than smaller ones.
- **Scenario 100N×8c:** This demonstrates how our approach deals with larger data-centres. The overhead further reduced over the above scenarios. The initial overheads were 1.9% lower (relatively) than scenario 20N×8c and 1.3% lower (relatively) than the other two scenarios (i.e. 20N×16c and 40N×8c). Also, as the task run times increased, the overheads reduced drastically. Therefore, applying our approach to this scenario shows that initial overheads are mostly lower in large data centres. Additionally, the graph shows that with the large configurations, the overheads reduce faster with long mapper run times than in the other scenarios.

Therefore, from the experiments and industry surveys, it is recommended that infrastructures with 14 to 20 cluster nodes (with eight cores) should use scenario 20N×8c data centre configuration. Infrastructures with 25 to 35 cluster nodes (with eight or sixteen cores) should use our 20N×16c data centre configuration. Infrastructures with 40 to 60 cluster nodes (with eight cores) should use 40N×8c data centre configuration. Finally, infrastructures with 100 to 150 cluster nodes (with eight cores) should use 100N×8c data centre configuration. Aside the above recommendations, our approach is customizable to suit user data configurations preferences.

D. Application Performance Experiments

This experiment determined the impact of our approach on job performance. Four measurements were taken to evaluate our approach. These are:

- Total execution times when there was *no disruption* on the MapReduce set-up (i.e. a dedicated Hadoop cluster scenario).

- Total execution times when *disruptions* were introduced on arbitrarily nodes on the infrastructure. These disruptions were created to interfere with task processing so that the task will have long run times than expected (this experiment was meant to represent a Hadoop cluster hosted in a multi-tenant environment). These *disruptions* were introduced via the running of extra tasks on arbitrarily nodes which were not linked to the original map or reduce tasks. The extra tasks were designed to consume extra system resources during the map and reduce phase. Also, the *disruptions* represent background services, IO contentions or uneven distribution of resources on data nodes for industry research.
- Total execution times when tasks were terminated and processed as backup tasks (*reschedule*) on a different node. This represents situations where mappers can restore their mid-execution states. This is applied by applications with the capabilities of storing their states during data processing. When such applications get terminated abruptly (due to factors contributing to speculative execution); the applications resume task processing on available nodes from the point they were halted.
- Total execution times when tasks were terminated and processed as backup tasks (*restart*) on a different nodes (providing insight into applications which cannot take advantage of state restoration).

These measurements were utilized to draw the graphs shown in Fig. 12. The graphs of *no disruption*, *reschedule* and *restart* were drawn relative to the *disruption* graphs which are shown in Fig. 13. The graphs were drawn relative to the *disruption* graphs, because we wanted to observe the levels of job improvements in light of the disruptions introduced into the system. The details of the various scenarios are discussed in the next sub-section.

E. Discussion Performance Experiments Results

First, Fig. 12a displays the behaviour of scenario 20N×8c data centre when our approach was applied. The task improvement on this data centre was gradual as seen in the figure. In relation to the *disruption* graph, the slope began from above 80% and reduced gradually below 10%. Furthermore, *reschedule* backup tasks improve better with our approach than *restart* backup tasks after disruption.

Second, scenario 20N×16c data centre demonstrated considerably more job performance improvement than the previous data centre as most of the graphs were below the 80% mark as seen in Fig. 12b. Also, tasks that transfer their states perform better with our approach than those that cannot. Tasks with long run times exhibited big improvements as their values were below 10% relative to disruption. This means that as the tasks are processed for long run times, the effects of the disruptions were reduced as the graphs approached the 0% mark. For industry practitioners, it is advisable to apply our approach for long run times.

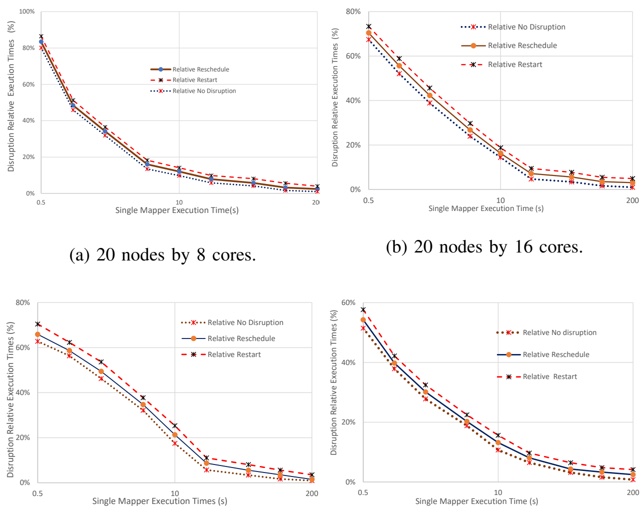
Third, scenario 40N×8c data centre improved more compared to the previous two scenarios as seen in Fig. 12c. Finally, scenario 100N×8c improved more than all the previous scenarios as the graph showed a gradual improvement from below the 60% as seen in Fig. 12d. The figure showed that tasks with long run times had higher chances of improvement in this data centre. As most of the graphs were below 10% relative to disruption. Also, *reschedule* backup tasks improved much better than the (*restart*) backup tasks. Since the *reschedule* backup tasks have the capability to save their states, it was easily for them to continue data processing when they were moved to other nodes. In contrast, the *restart* backup tasks do not store their states, hence they could not reschedule their states, which delayed their task processing durations when moved to other nodes.

In conclusion, larger data centres have a higher chance of improvement when applying this approach. This approach works better with larger data centres because the sizes fosters scalability with long run times, which also ensures reduction in system overheads.

F. Disruption Identification with K-Means Clustering

The task run times captured during the experiments were utilized for the K-means clustering. Two categories of results were observed after the clustering. Disruption-induced and disruption-free categories. The straggler tasks formed the disruption-biased data clusters are seen in Fig. 8a to 8d on page 17. The large magnitudes of the straggler tasks, enabled k-means to properly create the two categories.

Silhouette score was applied to validate the data clusters



(a) 20 nodes by 8 cores. (b) 20 nodes by 16 cores.

(c) 40 nodes by 8 cores. (d) 100 nodes by 8 cores.

Fig. 12. Tasks improvement experimental scenarios.

TABLE III. KMeans Clustering Silhouette Scores

Fig.	Silhouette Scores
7	0.685
8a	0.985
8b	0.985
8c	0.985
8d	0.985

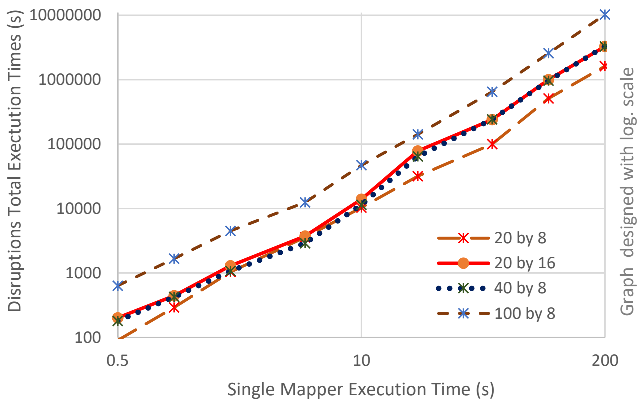


Fig. 13. Scenario disruptions.

and to foster overhead reduction in the application of the kmeans clustering algorithm. Once the dissimilarities between the dataset was detected, the clustering process was stopped. Table III shows the disruption-free and disruption-induced map task runtime data clusters and their respective silhouette scores. The result of Fig. 7 was closer to our silhouette score threshold lower-bound value. Since the data clusters created were not separated enough to ensure task transfer, the task were not shifted to other nodes. However, the rest of the experiments show significantly higher silhouette scores resulting in identifiable straggler tasks. The straggler tasks were then moved to available nodes as either a *reschedule* or *restart* backup tasks.

V. Conclusion

This paper proposed a new speculative execution approach to estimate task run times with consistent global snapshots and K-Means clustering. Our approach applied two algorithms to monitor and capture task run times as snapshots. A K-means clustering technique was applied to classify the captured run times into two categories (fast and straggler tasks). We applied a silhouette score as a decision-making tool to determine when to process backup tasks on available nodes. The silhouette scores also helped to reduce the number of iterations by the K-means. We evaluated our approach on different data centre configurations. These were selected based on a survey of industry requirements for Hadoop clusters and applications. Our experiments were focused on two objectives; (i) the overheads caused by implementing our approach and (ii) job performance improvements. Our experiments enabled us to show that (i) the overheads caused by applying our approach were reduced faster with large data centres than the smaller data centres. The overheads reduced by 1.9%, 1.5% and 1.3% (comparatively) as the size of the data centre and the task run times increased. (ii) Mapper tasks with typical longer task durations had better chances for improvements. The graphs of the longer mappers were below 10% relative to the disruptions introduced. This showed that the effects of the disruptions were reduced and became more negligible. This approach measured the job performance improvement achieved via the *restart* back up tasks. A further work was done for applications capable of transferring their states as *reschedule* back up tasks. Most of the previous approaches did not consider measurements for *reschedule* back up tasks.

For future work, we consider implementing auto-scaling algorithms on MapReduce Hadoop clouds. Our Snapshot capturing algorithm will be applied to foster a comparison with the job performance approach. Also, a couple of classification and clustering techniques will be considered to provide further extensions.

Acknowledgment

This research was supported by the Hungarian Scientific Research Fund under the grant number OTKA FK 131793.

References

[1] T. White, *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[2] D. E. O'Leary, "Artificial intelligence and big data," *IEEE Intelligent Systems*, vol. 28, no. 2, pp. 96–99, 2013.

[3] S. De and M. Panjwani, "A comparative study on distributed file systems," in *Modern Approaches in Machine Learning and Cognitive Science: A Walkthrough*. Springer, 2021, pp. 43–51.

[4] H. Xu and W. C. Lau, "Speculative execution for a single job in a mapreduce-like system," in *2014 IEEE 7th International Conference on Cloud Computing*, 2014, pp. 586–593.

[5] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments." in *Osdi*, vol. 8, no. 4, 2008, p. 7.

[6] 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. [Online]. Available: https://doi.org/10.1016/j.compeleceng.2015.06.013

[7] Q. Chen, D. Zhang, M. Guo, Q. Deng, and S. Guo, "Samr: A self-adaptive mapreduce scheduling algorithm in heterogeneous environment," in *2010 10th IEEE International Conference on Computer and Information Technology*. IEEE, 2010, pp. 2736–2743. Available: https://doi.org/10.1109/cit.2010.458

[8] X. Sun, C. He, and Y. Lu, "Esamr: An enhanced self-adaptive mapreduce scheduling algorithm," in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*. IEEE, 2012, pp. 148–155. [Online]. Available: https://doi.org/10.1109/icpads.2012.30

[9] Y. Li, Q. Yang, S. Lai, and B. Li, "A new speculative execution algorithm based on c4. 5 decision tree for hadoop," in *International Conference of Young Computer Scientists, Engineers and Educators*. Springer, 2015, pp. 284–291.

[10] H. B. Abdalla, A. M. Ahmed, and M. A. Al Sibahee, "Optimization driven mapreduce framework for indexing and retrieval of big data," *KSII Transactions on Internet and Information Systems (TIIS)*, vol. 14, no. 5, pp. 1886–1908, 2020. [Online]. Available: https://doi.org/10.3837/tiis.2020.05.002

[11] I. A. T. Hashem, N. B. Anuar, M. Marjani, E. Ahmed, H. Chiroma, A. Firdaus, M. T. Abdullah, F. Alotaibi, W. K. M. Ali, I. Yaqoob *et al.*, "Mapreduce scheduling algorithms: a review," *The Journal of Supercomputing*, vol. 76, no. 7, pp. 4915–4945, 2020.

[12] K. Kalia and N. Gupta, "Analysis of hadoop mapreduce scheduling in heterogeneous environment," *Ain Shams Engineering Journal*, vol. 12, no. 1, pp. 1101–1110, 2021. Available: https://doi.org/10.1016/j.asej.2020.06.009

[13] M. Sun, H. Zhuang, C. Li, K. Lu, and X. Zhou, "Scheduling algorithm based on prefetching in mapreduce clusters," *Applied Soft Computing*, vol. 38, pp. 1109–1118, 2016. [Online]. Available: https://doi.org/10.1016/j.asoc.2015.04.039

[14] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 185–198.

[15] S. Sakr, A. Liu, and A. G. Fayoumi, "The family of mapreduce and large-scale data processing systems," *ACM Computing Surveys (CSUR)*, vol. 46, no. 1, pp. 1–44, 2013. [Online]. Available: https://doi.org/10.1201/b17112-6

[16] Q. Chen, C. Liu, and Z. Xiao, "Improving mapreduce performance using smart speculative execution strategy," *IEEE Transactions on Computers*, vol. 63, no. 4, pp. 954–967, 2013. [Online]. Available: https://doi.org/10.1109/tc.2013.15

[17] A. Cornuéjols, C. Wemmert, P. Gançarski, and Y. Bennani, "Collaborative clustering: Why, when, what and how," *Information Fusion*, vol. 39, pp. 81–95, 2018. [Online]. Available: https://doi.org/10.1016/j.inffus.2017.04.008

[18] D. J. Bora, D. Gupta, and A. Kumar, "A comparative study between fuzzy clustering algorithm and hard clustering algorithm," *arXiv preprint arXiv:1404.6059*, 2014. [Online]. Available: https://doi.org/10.3390/j2020016

[19] C. Yuan and H. Yang, "Research on k-value selection method of k-means clustering algorithm," *J*, vol. 2, no. 2, pp. 226–235, 2019. [Online]. Available: https://doi.org/10.14445/22312803/ijctt-v10p119

[20] T. Gupta and S. P. Panda, "Clustering validation of clara and k-means using silhouette & dunn measures on iris dataset," in *2019 International conference on machine learning, big data, cloud and parallel computing (COMITCon)*. IEEE, 2019, pp. 10–13. [Online]. Available: https://doi.org/10.1109/comitcon.2019.8862199

[21] R. Ünlü and P. Xanthopoulos, "Estimating the number of clusters in a dataset via consensus clustering," *Expert Systems with Applications*, vol. 125, pp. 33–39, 2019. [Online]. Available: https://doi.org/10.1016/j.eswa.2019.01.074

[22] H. B. Zhou and J. T. Gao, "Automatic method for determining cluster number based on silhouette coefficient," in *Advanced materials research*, vol. 951. Trans Tech Publ, 2014, pp. 227–230. [Online]. Available: https://doi.org/10.4028/www.scientific.net/amr.951.227