International Journal of Electrical and Computer Engineering (IJECE)
Vol. 9, No. 5, October 2019, pp. 4396~4407
ISSN: 2088-8708, DOI: 10.11591/ijece.v9i5.pp4396-4407

Optimized memory model for hadoop map reduce framework

Archana Bhaskar[1], Rajeev Ranjan[2]
[1]Department Computer Applications, Acharya Institute of Technology, India
[2]REVA University, India

Article Info

Article history:

Received Jan 25, 2019
Revised Apr 2, 2019
Accepted Ape 19, 2019

Keywords:

Big data
Bioinformatics
Cloud computing
Hadoop
Map Reduce
Parallel computing

ABSTRACT

Map Reduce is the preferred computing framework used in large data analysis and processing applications. Hadoop is a widely used Map Reduce framework across different community due to its open source nature. Cloud service provider such as Microsoft azure HDInsight offers resources to its customer and only pays for their use. However, the critical challenges of cloud service provider is to meet user task Service level agreement (SLA) requirement (task deadline). Currently, the onus is on client to compute the amount of resource required to run a job on cloud. This work present a novel memory optimization model for Hadoop Map Reduce framework namely MOHMR (Optimized Hadoop Map Reduce) to process data in real-time and utilize system resource efficiently. The MOHMR present accurate model to compute job memory optimization and also present a model to provision the amount of cloud resource required to meet task deadline. The MOHMR first build a profile for each job and computes memory optimization time of job using greedy approach. Experiment are conducted on Microsoft Azure HDInsight cloud platform considering different application such as text computing and bioinformatics application to evaluate performance of MOHMR of over existing model shows significant performance improvement in terms of computation time. Experiment are conducted on Microsoft Azure HDInsight cloud. Overall, good correlation is reported between practical memory optimization values and theoretical memory optimization values.

Corresponding Author:

Archana Bhaskar,
Computer Applications,
Acharya Institute of Technology,
Bangalore, India.
Email: bhaskararchana3@gmail.com

1. INTRODUCTION

Many organizations such as industrial, government and education institution collects massive amount of data from various sources such as sensor network, social network, bioinformatics and World Wide Web etc. for various application uses. Performing scalable and analysis on these unstructured data is most desired across many organizations. The state-of-art model finds difficulties in performing real-time analysis on continuous/stream data. For performing real-time analysis for data intensive applications, Google have come up with parallel programming model called Map Reduce framework [1]. It is highly scalable, fault tolerant and parallelize execution in distributed nature across cluster of computing nodes. Hadoop Map Reduce framework [2] has been widely adopted across various organization when compared with counter parts Phoenix [3], Mars [4] and Dryad [5] due to open source nature [6].

The Hadoop Map Reduce model predominantly consist of following phases, Setup, Map, Shuffle, Sort and Reduce which is shown in Figure 1. The Hadoop frameworks consists of a master node and a cluster of computing nodes. Jobs submitted to Hadoop are further distributed into Map and Reduce tasks. In setup phase, input data of a job to be processed (residing generally on the Hadoop Distributed File Systems

(HDFS)) is logically partitioned into homogenous volumes called chunks for the Map worker nodes. Hadoop divides each Map Reduce job in to set of tasks were Map worker processes each chunk. Map phase takes input as key/value pair as (k_1, v_1) and generate list of (k_2, v_2) intermediate key/value pair as output. Shuffle phase begins with completion of Map phase that collects the intermediate key/value pair from the entire Map task. A sort operation is performed on the intermediate key/value pair of map phase. For simplicity, sort and shuffle phases are cumulatively considered in the shuffle phase. Reduce phase processes sorted intermediate data based on user defined function. Output of reduce phase is stored/written to HDFS.

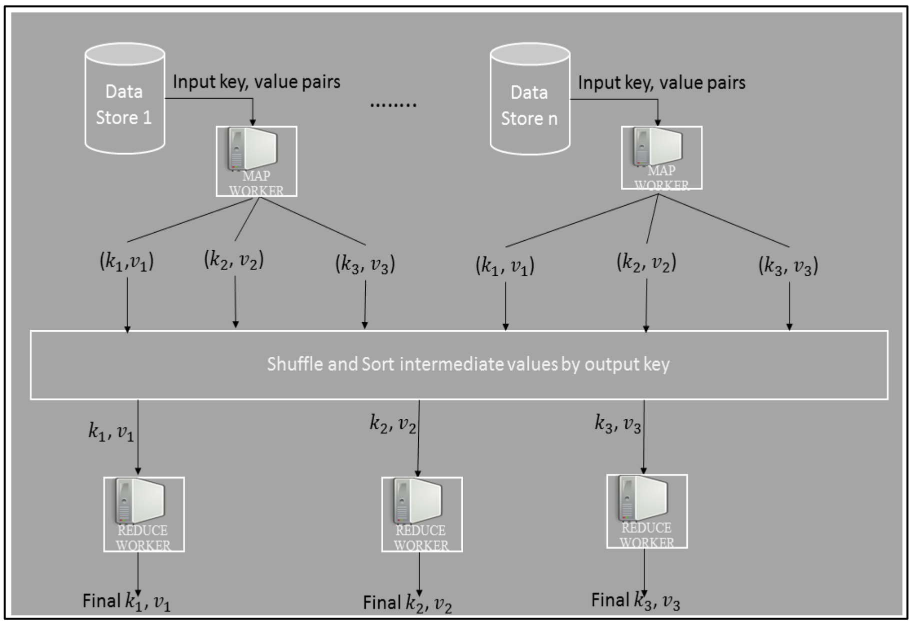

Figure 1. Hadoop map reduce computation model

The Azure HDInsight Cloud aid in achieving scalable performance i.e. user can set up and run Hadoop application on a large-scale cluster. Azure HDInsight Cloud allow user to configure the amount of resource (virtual computing node) required to perform certain task. However, at present Hadoop job with deadline requirement is not supported in HDInsight cloud. The onus is on the cloud user/client to compute the amount of resource requirement to meet task deadline, which is a challenging task. Therefore, Hadoop memory optimization modelling has become an important criterion in computing amount of resources required to meet task deadline. It should be noted that memory optimization modeling is a challenging task since Hadoop jobs involves multiple processing stage which composed of three core stage (i.e. Map, Shuffle and Reduce stage). Moreover, the first wave of shuffle stage is generally processed in parallel fashion with Map stage (i.e. overlapping phase) and rest of the waves of the Shuffle stage are processed post completion of Map stage (i.e. non-overlapping phase). To utilize the cloud resources efficiently, numerous memory optimization models for Hadoop is presented [7, 8]. However, these approaches are not accurate and incurs high computing overhead/time. Since these approaches did not consider overlapping and non-overlapping phases of the Shuffle stage.

In recent years, to enhance the performance of Hadoop application, various efficient Hadoop models are presented by numerous researchers [9-14]. A comprehensive information for job prediction and optimization can be presented using Starfish [9] model, which gathers an active Hadoop task profile at a satisfactory granularity. Elasticiser [10] is presented for resource allocation based on VMs, which is considered above Starfish. However, gathering an active Hadoop task profile with comprehensive information can lead to large overhead and hence high over-predicted task run-time. In [11, 13] utilizes both overlapping and non-overlapping phases of shuffle stage and for task prediction a conventional linear regression method is used. This application also helps to predict the required number of resources for various tasks with deadline constraints. CRESP [14] predicts task execution efficiently and helps to allocate resources based on Map Reduce slots. However, in CRESP application models, the effect of number of reduced jobs are discarded. In [13, 14], the number of reduced jobs are constant. In CRESP model, a single wave of a reduced stage is used. Moreover, in this model, reduced job number must be same as reduced slot number. It is very impractical to keep similar number of reduced jobs or for every job, a single wave of reduced stage. Practically, the reduced job number, rely upon the dataset size, type of Hadoop model and subscriber's need. Moreover, the use of multiple waves in a reduced stage can helps to analysis huge data with much faster rate and with less number of resources in contrast to single wave. Moreover, the use of multiple waves can enhance the resource utilization of an I/O disk whereas single wave can decrease the job setup delay.

To address the research challenges this work present an accurate and efficient memory optimization model for Hadoop Map Reduce framework namely MOHMR (Optimized Hadoop Map Reduce) to process data in real-time and utilize system resource efficiently. The MOHMR present accurate model to compute job memory optimization time and also present a model to provision the amount of cloud resource required to meet task deadline. The MOHMR first build a profile for each job and computes memory optimization time of job using greedy approach. Furthermore, to provision amount of resource required to meet task deadline Lagrange Multipliers technique is applied.

The Contribution of research work is as follows:

1. This work present an accurate memory optimization model for HMR aiding performance improvement.
2. Experiments considering diverse cloud configurations and varied application configuration.
3. Correlation between theoretical memory optimization model and experimental values.

The rest of the paper is organized as follows. Extensive research survey is carried out in section II. In section III the proposed memory optimization modelling for Hadoop Map Reduce framework is presented. In penultimate section experimental study is carried out. The conclusion and future work is described in last section.

2. RELATED WORK

Explaining In this section, a detailed literature is presented about the conventional state-of-art data analytic techniques. In [9], a locality based Hadoop cluster model is adopted which rely upon the distance between input information and processing nodes. This technique try to overcome from various issues of state-of-art techniques such as high overhead, required large storage capacity and expensive in real time. However, it also induces large delay and causes performance degradation.

In [10], a cloud based optimization framework is adopted to meet deadlines and accomplish data locality. They presented heuristic technique to provision task SLA requirement of cloud user. This technique presented an optimization technique to meet task dead line and minimize the number of nodes required for task processing. They solved single node failure and presented a tradeoff between minimizing deadline and locality constraint. Outcome shows reduction of storage and computation overhead. However they did not considered task deadline aware scheduling and performance evaluation considering compute intensive application.

In [11], a performance enhancement technique is introduced for Hadoop model based on metadata of interrelated tasks. This technique permits Name Nodes to find block which are preset in the cluster to store specific data. Their model attained superior performance than Hadoop framework. For performance evaluation they considered Bioinformatics application. Experiment outcome shows good performance in terms of I.O cost minimization and memory optimization time reduction. However, they did not considered performance evaluation considering different application and they considered performance evaluation for small genomic data size.

In [12], a Hadoop model is presented based on Map Reduce performance modules to reduce delay and contention in the network and enhance performance of the system. And it also helps to decrease synchronization delay and schedule different tasks at a time. They also presented a theoretical evaluation of their memory optimization model. Attained good accuracy and performance evaluation is carried out for word count applications. However, they did not considered performance evaluation considering diverse application and evaluation on cloud platform.

In [13], an Afford-Hadoop application is adopted to reduce cost in finishing various tasks and to allocate data and schedule tasks and hence efficiency of system get enhanced. However, a NP-hard problem occurs while scheduling different tasks in state-of-art technique. To address NP-hardness, they adopted integer programming techniques and heuristic reduction and optimization to enable an optimal solution. Experiment are conducted considering Word count and Sort application attained good results in terms of cost minimization. However, theoretical accuracy performance evaluation is not presented.

In [14], a Hadoop model is proposed to predict tasks run-time and allocate some specified resources to accomplish tasks in an assigned time period. Hence, the deadline constraints are met. It uses multiple waves of a shuffle stage. Experiment are conducted considering word count and sort application.

Theoretical accuracy performance evaluation of memory optimization model is presented shows good accuracy. However, it induces high overhead to finish tasks and data intensive and diverse application such as bioinformatics application is not considered for performance evaluation.

In [15], A Hadoop model is adopted to optimize Hadoop parameters with the help of programming based PSO. The PSO technique helps to find optimal parameters in Hadoop networks for a specified task. However, performance evaluation under cloud computing environment is not considered. In [16], a Big-Data computational model is adopted to reduce cost with the help of geo-distributed datacenters. This technique helps to decide the parameters to select the final data center. Here, a framework for efficient information movement and to provide resource allocation and to select a required data center to decrease cost of the system is described. However, task deadline requirement of task is not considered.

Extensive research survey carried out shows numerous approach is presented to minimize cost, time and amount of resource required to compute a task on Hadoop Map Reduce framework. The survey shows need to develop a new memory optimization model that minimize amount of resource required to task deadline with good accuracy considering diverse application. In next section the proposed memory optimization model for Hadoop Map Reduce framework is presented.

3. DYNAMIC MEMORY MANAGEMENT FOR HADOOP MAP REDUCE FRAMWORK

This work present a novel memory management model for Hadoop Map Reduce (HMR) framework for execution of both stream and non-stream application.

3.1. System architecture

This section presents the system architecture of Mammoth and how to reform the Map Reduce execution model. To improve the memory usage, a thread-based execution engine is implemented in Mammoth. Figure 2 shows the overall architecture of the Execution Engine in Mammoth. The execution engine runs inside a single JVM (Java Virtual Machine). In Mammoth, all Map/Reduce tasks in a physical node run inside the execution engine, and therefore in a single JVM, which is one of the key architectural differences between Mammoth and Hadoop. Mammoth retains the upper cluster management model in Hadoop, i.e. the master Job-Tracker manages the slave Task-Trackers through heartbeats. When the Task-Tracker obtains a new Map/Reduce task, the task is assigned to the execution engine through RPC (Remote Procedure Call) and the execution engine informs the Task Tracker of the task's real-time progress through RPC too.

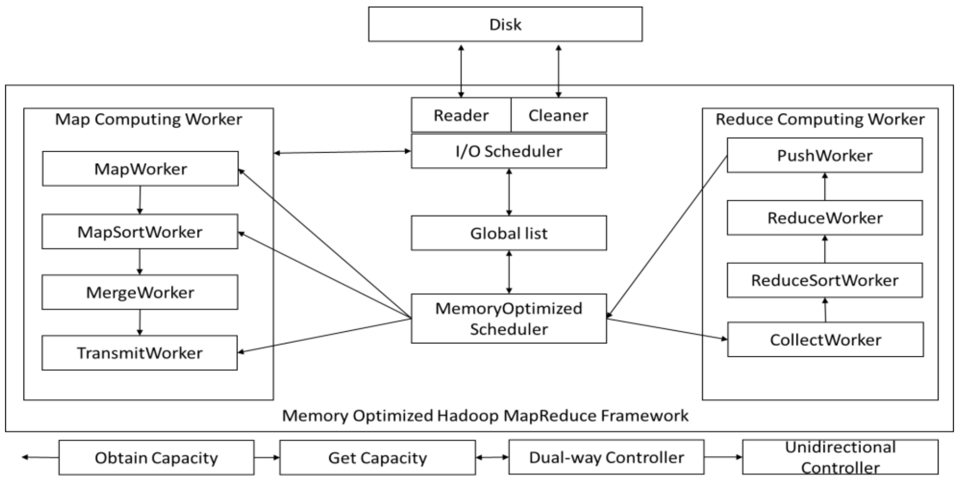

Figure 2. Architecture of proposed memory optimized HMR

3.2. Memory optimization for hadoop cache scheduler

This section presents a memory optimization for Hadoop cache scheduler for improving system efficiency. To attain the following issues needs to be resolved. Firstly, there exist different data capacity/buffers. As a result memory allocation to these buffers needs to properly design. Secondly, the memory allocation design should be dynamic in nature. Since, different memory is required for different stage of Map Reduce task. The capacity of various buffers can be computed by Cache List (CL) using following equations. The total size $\mathcal{T}_S$ of cache list is obtained as follows

$$\mathcal{T}_{\mathcal{S}} = \mathcal{S}^{\uparrow} - \mathcal{D}_{list_{\mathcal{S}}} - \mathcal{M}_{\mathcal{C}_{\mathcal{S}}} \tag{1}$$

where $\mathcal{S}^{\uparrow}$ is the maximum memory size that can be utilized for intermediary data, $\mathcal{D}_{list_{\mathcal{S}}}$ is the total size of Data pair list, and $\mathcal{M}_{\mathcal{C}_{\mathcal{S}}}$ is the total size of memory utilize by I/O Scheduler for the I/O buffer.

Similarly, the MMC size $MMC_{\mathcal{S}}$ (i.e., the memory size that MMC can utilize to assign among Map tasks) is obtained as follows

$$MMC_{\mathcal{S}} = \min\left(\mathcal{O}_{\mathcal{C}_{\mathcal{S}}} + \mathcal{P}trns_{\mathcal{C}_{\mathcal{S}}}, \mathcal{T}_{\mathcal{S}}\right) \tag{2}$$

where $\mathcal{O}_{\mathcal{C}_{\mathcal{S}}}$ is the sort capacity (buffer) size, and $\mathcal{P}trns_{\mathcal{C}_{\mathcal{S}}}$ is the present size of transmission capacity. The sort capacity size is obtained as follows

$$\mathcal{O}_{\mathcal{C}_{\mathcal{S}}} = \begin{cases} MO^{\uparrow} * M_n & MO^{\uparrow} \neq 0 \\ \mathcal{P}O_{\mathcal{C}_{\mathcal{S}}} & MO^{\uparrow} = 0 \end{cases} \tag{3}$$

where $MO^{\uparrow}$ is the maximum size of sort capacity in the prior history of Map Reduce tasks i.e., the highest size of each map task's sort capacity, $\mathcal{P}O_{\mathcal{C}_{\mathcal{S}}}$ is the present size of sort capacity., and M_n is the number of Map jobs that is running presently. Finally, the RMC memory size $RMC_{\mathcal{S}}$ which can be utilized among Reduce task can be obtained as follow

$$RMC_{\mathcal{S}} = \mathcal{T}_{\mathcal{S}} - MMC_{\mathcal{S}} \tag{4}$$

The proposed memory optimized Hadoop scheduler (MOHS) model avoid frequent memory recycling from transmission and collection buffer when the memory is controlled and possess or keep enough memory for map jobs. When one Map job is initialized or finished, it will record or unregister to MMC. In this manner, the MMC can obtain the parameter of M_n. MergeSort request (.i.e., reserve) the cache data for Sort Capacity from MMC, and Transmission Worker returns (i.e., unreserve) the cache units that been shuffled from Transmission Capacity and give it back to MMC. In this way MMC computes $\mathcal{O}_{\mathcal{C}_{\mathcal{S}}}$ and $\mathcal{P}trns_{\mathcal{C}_{\mathcal{S}}}$. Each Map jobs computes it's $MO^{\uparrow}$ dynamically (i.e., runtime), and just before it is finished, and it informs to MMC its $MO^{\uparrow}$. Then MMC reports CMC when the parameter of these variables changes and CMC finalizes the heap size for MMC and RMC, as shown in Algorithm 1.

Algorithm 1: Central memory Controller (CMC) schedules memory among Map Memory Controller (MMC) and Reduce Memory Controller (RMC)

```
Input: MMC updates CMC.
Output: MMC has higher selectivity than RMC.
Step 1: Start
Step 2: $\mathcal{T}_{\mathcal{S}}$ is initialized at establishment, MMC updates CMC about the recently updated value.
Step 3: if $MO^{\uparrow} = 0$ then
Step 4:    $\mathcal{O}_{\mathcal{C}_{\mathcal{S}}} \leftarrow \mathcal{P}O_{\mathcal{C}_{\mathcal{S}}}$
Step 5: else
Step 6:    $\mathcal{O}_{\mathcal{C}_{\mathcal{S}}} \leftarrow MO^{\uparrow} * M_n$
Step 7: end if
Step 8: $MMC_{\mathcal{S}} \leftarrow \min\left(\mathcal{O}_{\mathcal{C}_{\mathcal{S}}} + \mathcal{P}trns_{\mathcal{C}_{\mathcal{S}}}, \mathcal{T}_{\mathcal{S}}\right)$
Step 9: setMCHeap($MMC, MMC_{\mathcal{S}}$)
Step 10: setMCHeap($RMC_{\mathcal{S}}, \mathcal{T}_{\mathcal{S}} - MMC_{\mathcal{S}}$)
Step 11: End.
```

Similar to Map, when a Reduce job is initialized or completed, it indexes or un-indexes to RMC, and RMC then partition the $RMC_{\mathcal{S}}$ uniformly among the running Reduce jobs. CollectionWorker obtains the memory for the Collection Buffer (CB) from RMC, while PushWorker un-reserve the resource from the CB and gives it back to RMC. RMC reports CMC of its memory utilization status, which is then utilized to establish when memory resource is a constraint for the Map jobs (this process will be defined in Algorithm 2). During whole execution process of jobs, the memory utilization for CMs will vary and their resource utilization will be optimized dynamically. When the optimized heap size for MC is greater than previous updation, then there is no memory bottleneck for the jobs to utilize. However, when the optimized heap size is lesser than previous updation, it depicts that the resource is not adequate and certain area of the capacity/buffer have to be cleaned to the disk quickly.

```
Algorithm 2: Memory resource optimization of MMC
Input: job identifier, request type, request size
Output: a cache data
Step 1: Start
Step 2: CMC initialize $MMC_{\mathcal{S}}$ dynamically (i.e., in runtime), $\mathcal{U}$ is a statistical parameter.
Step 3: $\mathcal{T} \leftarrow \mathcal{S} + \mathcal{U}$
Step 4: if $(\mathcal{T} > MMC_{\mathcal{S}})$ then
Step 5:     leftover resource cannot fit $\mathcal{S}$
Step 6:     if $(\mathcal{T} > CMC.\mathcal{T}_{\mathcal{S}})$ then
Step 7:         CMC obtains resource from RMC
Step 8:         CMC.set$RMC_{\mathcal{S}}(RMC_{\mathcal{S}} - \mathcal{S})$
Step 9:     else if $(\mathcal{P}trns_{C_{\mathcal{S}}} > \mathcal{S})$ then
Step 10:        MMC obtains resource from transmission capacity or buffer
Step 11:        $clean(TransmissionCapacity, \mathcal{S})$
Step 12:    else
Step 13:        MMC obtains resource from Sort capacity
Step 14:        $clean(SortCapacity, \mathcal{S})$
Step 15:    end if
Step 16: end if
Step 17: obtain a cache data from cache list.
Step 18: $\mathcal{U} \leftarrow \mathcal{U} + \mathcal{S}$
Step 19: $report(CMC)$
Step 20: get cache data.
Step 21: end.
```

Algorithm 2 describes the working structure of a Map job keeping the memory from MMC. Further, there is a threshold defined for memory utilization for MCs and similarly for MMC, the threshold is defined as $MMC_{\mathcal{S}}$, which is obtained using (2). When a Map jobs request exceeds$MMC_{\mathcal{S}}$, some capacity data required to be released. Considering the case, the type of capacity with low allocation selectivity has high overturning selectivity, as shown in algorithm 2 (line 6 and 15). Since the ReceiveWorker capacity has low allocation selectivity, MMC first checks CMC of RMC's memory state (Line 5). If RMC has sufficient memory, CMC will push certain quantum of memory from RMC to MMC in order to meet the demand (Line 6 to 8).

3.3. I/O optimization for HMR framework

This section presents an I/O optimization for HMR framework for minimizing disk seek and attain better parallel I/O performance. Further, optimization is carried out for performing MergeSort operation in HMR. In HMR, performing parallel I/O operation can incur huge disk seek overhead. Since jobs are executed in individual JVM and they do not communicate with each other. As a result, decrease I/O performance. To attain better I/O performance, this work presented sequential I/O and overlay the CPU execution and dis I/O. The proposed I/O scheduler is composed of two component namely, ReadWorker (RW), and CleanWorker (CW) which is accountable for read and write operations respectively. Both RW and CW possess request capacity pools, with each pools associates with one I/O request, these capacity are called multi-capacity or multi-buffer. Each capacity pool has a selectivity, utilized by RW and CW to rearrange the read/write operations.

In proposed model, the I/O operation is composed of two types such as active and passive I/O. An active I/O, reads the input from Hadoop distribute file system, and write the final output to Hadoop distribute file system, and also write Map jobs intermediate output for attaining fault tolerance. Active I/O has higher selectivity than passive I/O. Since active I/O is more essential for job execution and should be carried out in prompt manners as possible. On the other side an I/O operation is known as passive I/O, if I/O is initialized because the intermediate data cannot be kept into the buffer and they are cleaned to the disk temporally. Passive I/O, operates on buffer with high allocation selectivity possess the low clean selectivity and high read selectivity.

As described in Algorithm 3, when write request is given to the I/O scheduler, the scheduler will allocated this request a selectivity based on its class $\mathbb{C}$ ($Line$ 3 to 11), then set a pool and add the pool to the pool list in CleanWorker. Line 14 to 22 describes how CleanWorker cleans the data in different pools. Requests with high selectivity will be satisfied first. However, for request with same selectivity, CleanWorker will poll their pools and write one block per instance in round robin manner, which in this work we represent it as interleaved-I/O. ReadWorker perform a read request in similar manner as CleanWorker.

Algorithm 3: Clean in I/O optimization HMR scheduler

```
Step 1: Start
Step 2: Establish the selectivity for this task demand.
Step 3: if ($\mathbb{C} = \mathcal{A}$ ) then
Step 4:     $selectivity \leftarrow A$
Step 5: else if ($\mathbb{C} = \mathcal{C}$)
Step 6:     $selectivity \leftarrow B$
Step 7: else if ($\mathbb{C} = trns$)
Step 8:     $selectivity \leftarrow C$
Step 9: else if ($\mathbb{C} = \mathcal{O}$)
Step 10:    $selectivity \leftarrow D$
Step 11: end if
Step 12: $pool \leftarrow inzPool(selectivity)$
Step 13: add $pool$ to Cleaner's $poolList$
Step 14: Cleaner cleans the data in buffer pools
Step 15: while $poolList \neq \emptyset$ do
Step 16:     $s \leftarrow$ highest selectivity of pools
Step 17:     $\forall pool \in poolList$ do
Step 18:         if ($s = pool.selectivity$) then
Step 19:             clean($pool[0]$)
Step 20:         end if
Step 21:     end for
Step 22: end while
Step 23: end.
```

In our work, the merge-sort operation are mainly carried out on the data kept in the memory, which in this work we call it as in-memory merge-sort. However, in HMR we call it as external sort. Since sort algorithm is carried out over the data kept in the disks. As result, incurs disk I/O and CPU-bound sort operation are interleaved in executions. Subsequently, once buffer is filled with full of sorted data, the CPU must block and wait. However, with multi-capacity can use non-blocking I/O. As for the CleanWorker, the units of a cache data are added to the CleanWorkers capacity pools. Post that cache units is written back to the disk and will be given it back to CacheList immediately. In next section the performance evaluation of proposed memory optimized HMR scheduler over state of art technique is shown.

4. RESULT AND ANNALYSIS

This section present performance evaluation of proposed MOHMR over state-of-art Hadoop Map Reduce Framework [11]. Hadoop is the most widely used/adopted Map Reduce platform for computing on cloud environments [17], hence it is considered for comparisons. Hadoop 2.0 i.e. version 2.7 is used and is deployed on azure cloud using HDInsight. The Hadoop cluster is composed of one master worker node and four worker/slave nodes. Each worker node is deployed on A3 virtual machine instances which composed of 4 virtual computing cores, 7 GB RAM and 120 GB of storage space. Uniform configuration is considered for both MOHMR and HMR. For experiment analysis different application are considered such as Gene sequencing (Bioinformatics), Word frequency statistics computation and Hot-word detection.

4.1. Bioinformatics application performance evaluation

Gene sequence alignment is a fundamental operation adopted to identify similarities that exist between a query protein sequence, DNA or RNA and a database of sequences maintained. Sequence alignment is computationally heavy and its computation complexity is relative to product of two sequences being currently analyzed. Massive volumes of sequences maintained in the database to be searched induces additional computation burden. BLAST is a widely adopted bioinformatics tool for sequence alignment which perform faster alignments, at expense of accuracy (possibly missing some potential hits) [18, 19].

Experiment are conducted to evaluate MOHMR and HMR performance for performing gene sequence alignment. The dataset for experiment analysis is obtained from NCBI [20]. For performing alignment Drosophila database as a reference database and Query sequence of varied sizes of from Homo sapiens chromosomal sequences and genomic scaffolds is considered similar to [19] which are tabulated in Table 1. All four experiment are conducted using BLAST algorithm on HMR and MOHMR frameworks. The total makespan time of both HMR and MOHMR for all six experiment is noted and graph is plotted as shown in Figure 2. It must be noted that the initialization time of the VM cluster is not considered is computing makespan as it is uniform in both MOHMR and HMR owing to similar cluster configurations.

The total makespan of MOHMR and HMR is dependent on task execution time of virtual computing/worker nodes during Map and Reduce phase. The total makespan observed in BLAST sequence alignment experiments executed on HMR and MOHMR frameworks is shown in Figure 3. The outcomes shows significant performance in terms of reduce makespan times of MOHMR over HMR. A makespan reduction of 46.37%, 46.72%, 58.52%, and 65.51% is obtained for four experiment by MOHMR over HMR. An average makespan reduction of 54.28% is achieved by MOHMR over HMR across all experiments. Similarly, the total memory usage observed in BLAST sequence alignment experiments executed on HMR and MOHMR frameworks is shown in Figure 4. The outcomes shows significant performance in terms of memory usage reduction of MOHMR over HMR. A memory usage reduction of 39.99%, 38.88%, 43.47%, and 48.48% is obtained for four experiment by MOHMR over HMR. An average memory usage reduction of 43.81% is achieved by MOHMR over HMR across all experiments.

Table 1 Gene sequence considered for experiment analysis

Experiment Id	Query genome	Query genome size	Experiment Id	Query genome
1	NT_007914	14866257	Drosophila database	122,653,977
2	AC_000156	19317006	Drosophila database	122,653,977
3	NT_033899	47073726	Drosophila database	122,653,977
4	NT_022517	90712458	Drosophila database	122,653,977

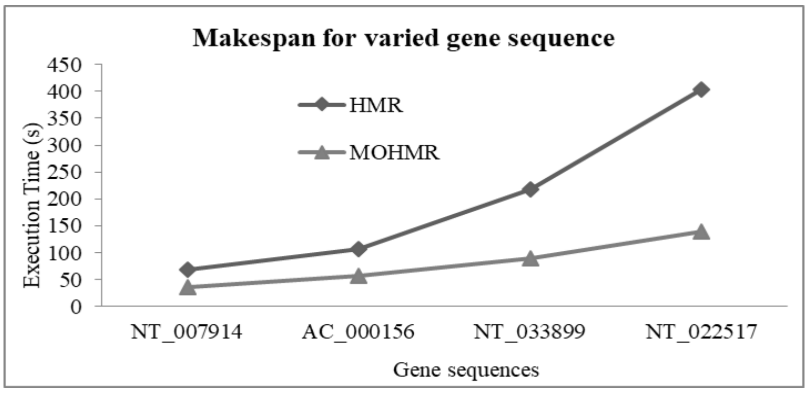

Figure 3. BLAST sequence alignment total makespan time observed for experiments conducted on OHMR and HMR frameworks

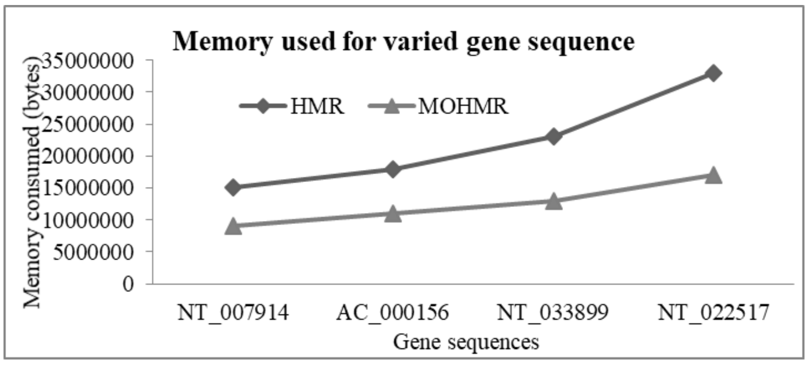

Figure 4. BLAST sequence alignment memory used for experiments conducted on OHMR and HMR frameworks

4.2. Non-stream analysis/Word frequency statistics computations

For non-stream data analysis, this work considers word frequency statistic algorithm and is developed using Java programing language. The Wikipedia non-stream dataset [21] is considered for experiment analysis. The Wikipedia dataset is huge in size (i.e. >100 GB) and is split into 4048 MB each and stored in Azure cloud storage container. For experimental analysis this work consider 16GB of data. The word frequency statistics algorithm were executed on both MOHMR and HMR framework and the results obtained are noted. The outcomes shows significant performance in terms of reduce makespan times and memory usage of OHMR over HMR. A makespan reduction of 44.92%, 45.07%, 46.55%, and 55.17% is obtained for data size of 4096 MB, 8192 MB, 16384 MB and 32768 respectively by MOHMR over HMR as shown in Figure 5. An average makespan reduction of 47.92% is achieved by MOHMR over HMR across all experiments. Similarly, memory consumption reduction of 55.91%, 60.28%, 64.93%, and 64.64% is obtained for data size of 4096 MB, 8192 MB, 16384 MB and 32768 respectively by MOHMR over HMR as shown in Figure 6. An average memory usage reduction of 62.8% is achieved by MOHMR over HMR across all experiments.

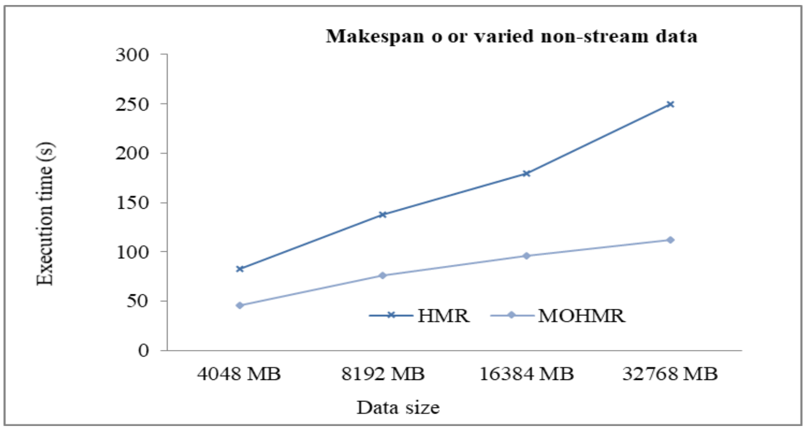

Figure 5. Non-stream data makespan time for experiment conducted on MOHMR and HMR frameworks

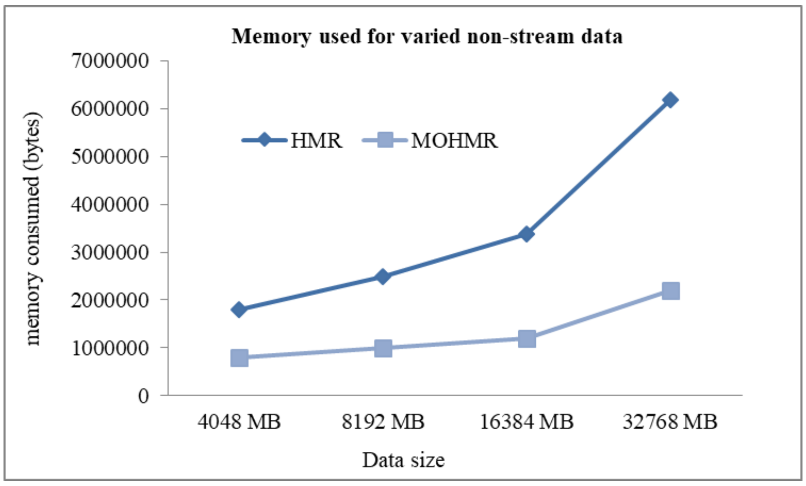

Figure 6. Non-stream data memory used for experiment conducted on MOHMR and HMR frameworks

4.3. Stream analysis/hot-word detection computations

For stream data analysis, hot-word detection algorithm [22] is developed using Java programing language. The “Movietweetings” dataset [23] is considered for experiment analysis and stored in Azure cloud container. Tweets consisting of 10000, 20000, 40000, and 80000 movies is considered and is represented as 10K, 20K 40K, and 80K. The hot-word detection algorithm were executed on the MOHMR and HMR framework and the results obtained are noted. The total makespan time of MOHMR and existing model is noted and is shown in Figure 7 and memory consumption is shown in Figure 8. Experiment analyses shows as number of tweets increases the computation time of both MOHMR and HMR increases. The outcomes shows significant performance in terms of reduce makespan times of MOHMR over HMR. A makespan reduction of 56.63%, 48.55%, 62.65%, and 61.07% is obtained for tweet size of 10K, 20K, 40K, and 80K respectively by MOHMR over HMR as shown in Figure 7. An average makespan reduction of 58.64% is achieved by MOHMR over HMR across all experiments. Similarly, memory usage reduction of 62.95%, 65.55%, 72.14%, and 71.81% is obtained for tweet size of 10K, 20K, 40K, and 80K respectively by MOHMR over HMR as shown in Figure 7. An average memory usage reduction of 68.11% is achieved by MOHMR over HMR across all experiments.

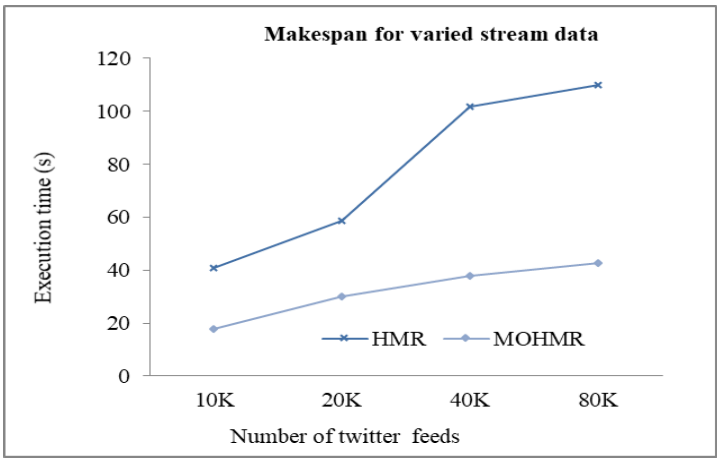

Figure 7. Stream data makespan time for experiment conducted on MOHMR and HMR frameworks

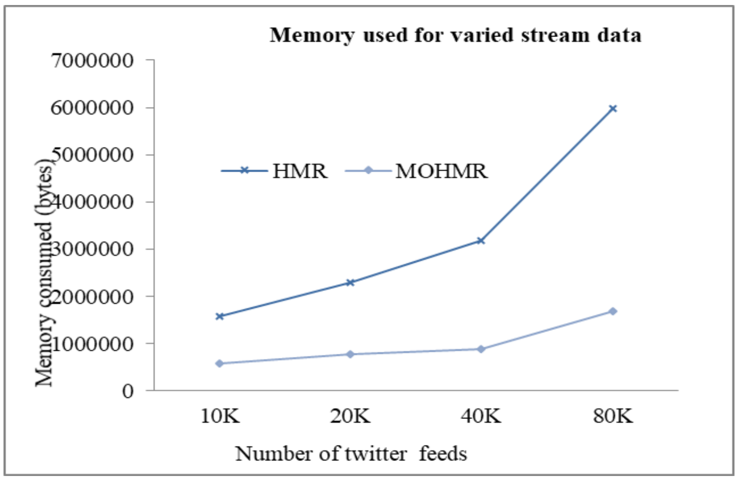

Figure 8. Stream data memory used for experiment conducted on MOHMR and HMR frameworks

4.4. Result and discussion

In this section the execution of the imprecise and bioinformatics applications namely word frequency statistics, hot word detection, and gene sequencing (BLAST) is presented. The results presented here prove that the OHMR model reduces the makespan observed due to the optimized makespan model incorporated in to HMR. An average reduction of 58.64% for stream data analysis and 47.92% for the non-stream analysis is reported and 58.64% for the gene sequencing (BLAST) considering the MOHMR model when compared to the existing HMR model [11]. Further MOHMR reduces an average memory consumption by 43.81% considering bioinformatics and text mining application. A memory performance improvement of 42.81% is achieved by MOHMR over existing HMR model [11] considering bioinformatics application. The cumulative analysis over state-of-art technique in Table 2 shows the efficiency of MOHMR over state-of-art technique in terms of robustness and scalability. Since, MOHMR support execution of different application such as Bioinformatics and text mining over cloud platforms. Our MOHMR model aided in better cloud resource utilization. Adoption cloud platform aid in proving scalability of processing of large amount of data of various types on large computing clusters. All these feature attributed to the performance improvement of MOHMR over state-of-art models.

Table 2. Comparson with state of art technique

	[11]	[12]	[13]	[14]	[15]	MOHMR
MapReduce platform considered	Hadoop	Hadoop	Hadoop	Hadoop	Hadoop	Hadoop
Cloud adopted	Yes	NO	Yes	Yes	No	Yes
Application considered	Bioinformatics	Word count	Word count and Tera sort	Word count and Sort	Word count and Sort	Bioinformatics and text mining
Average makespan percentage improvement over HMR framework	40.01%	13.01%	34.08%	27.1%	43.67%	51.16%
Average memory consumed percentage improvement over HMR framework	15%	-	-	-	-	57.81%

5. CONCLUSION

This work discussed the drawback of Hadoop MapReduce framework are discussed. Further, the significance of memory and I/O requirement for designing efficient HMR framework are discussed here. This work aimed to minimize makespan by presenting novel thread based execution of MapReduce task. Further to realize global memory management, optimization of memory and I/O for HMR is presented. MOHMR solve issues of Garbage Collection (GC) in the VM and reduce Disk I/O seek. To evaluate the performance of MOHMR framework bioinformatics, stream and non-stream application is used. Performance of MOHMR framework is compared with HMR framework in terms of makespan time. Average overall makespan times reduction of 54.28%, 58.64%, and 47.92% is achieved using OHMR framework when compared to HMR framework for bioinformatics, stream and non-stream applications respectively. Similarly, average overall memory consumption reduction of 43.81%, 68.11%, and 62.92% is achieved using OHMR framework when compared to HMR framework for bioinformatics, stream and non-stream applications respectively. Experiments outcome presented prove robustness of MOHMR framework, its capability to handle diverse applications on public cloud platforms. Results presented through experiments conducted prove superior performance of MOHMR against Hadoop framework. The future work would consider performance evaluation considering different application and also would further consider optimization of MapReduce scheduler for further reduction of computation time.

REFERENCES

[1] D. Radhika, D. Aruna Kumari, "Misusability Measure Based Sanitization of Big Data for Privacy Preserving Map Reduce Programming," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 8, no. 6, pp. 4524~4532, Dec 2018.

[2] Md. Armanur Rahman, J. Hossen, Venkataseshaiah C, CK Ho4, Tan Kim Geok, Aziza Sultana, Jesmeen M. Z. H., Ferdous Hossain, "A Survey of Machine Learning Techniques for Self-tuning Hadoop Performance," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 8, no. 3, pp. 1854-1862, Jun 2018.

[3] K. Taura, T. Endo, K. Kaneda, and A. Yonezawa, "Phoenix: a parallel programming model for accommodating dynamically joining/leaving resources," *in SIGPLAN Not.*, vol. 38, no. 10, pp. 216–229, 2003.

[4] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a Map Reduce framework on graphics processors," *in Proceedings of the 17th international conference on Parallel architectures and compilation techniques - PACT* '08, pp. 260, 2008.

[5] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 59–72, Mar. 2007.

[6] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: Mining Peta-scale Graphs," *Knowl. Inf. Syst.*, vol. 27, no. 2, pp. 303–325, May 2011.

[7] X. Lin, Z. Meng, C. Xu, and M. Wang, "A Practical Performance Model for Hadoop Map Reduce," *in Cluster Computing Workshops (CLUSTER WORKSHOPS), 2012 IEEE International Conference on*, pp. 231–239, 2012.

[8] X. Cui, X. Lin, C. Hu, R. Zhang, and C. Wang, "Modeling the Performance of Map Reduce under Resource Contentions and Task Failures," in Cloud Computing Technology and Science (CloudCom), *2013 IEEE 5th International Conference on*, vol. 1, pp. 158–163, 2013.

[9] M. Khan, Y. Liu and M. Li, "Data locality in Hadoop cluster systems," *2014 11th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, Xiamen, pp. 720-724, 2014.

[10] M. Xu, S. Alamro, T. Lan and S. Subramaniam, "CRED: Cloud Right-Sizing with Execution Deadlines and Data Locality," *in IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 12, pp. 3389-3400, 2017.

[11] H. Alshammari, J. Lee and H. Bajwa, "H2Hadoop: Improving Hadoop Performance using the Metadata of Related Jobs," *in IEEE Transactions on Cloud Computing*, vol. PP, no. 99, pp. 1-1, 2016.

[12] Daria Glushkova, Petar Jovanovic, Alberto Abelló, "Map Reduce Performance Models for Hadoop 2.x," *in Workshop Proceedings of the EDBT/ICDT 2017 Joint Conference*, 2017.

[13] M. Ehsan, K. Chandrasekaran, Y. Chen and R. Sion, "Cost-Efficient Tasks and Data Co-Scheduling with AffordHadoop," *in IEEE Transactions on Cloud Computing*, vol. PP, no. 99, pp. 1-1, 2017.

[14] M. Khan, Y. Jin, M. Li, Y. Xiang and C. Jiang, "Hadoop Performance Modeling for Job Estimation and Resource Provisioning," *in IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 441-454, 2016.

[15] Khan M., Huan, Z., Li M., Taylor GA., "Optimizing Hadoop parameter settings with gene expression programming guided PSO," *Concurrency Computation: Practice and Experience*, 2016.

[16] W. Xiao, W. Bao, X. Zhu and L. Liu, "Cost-Aware Big Data Processing Across Geo-Distributed Datacenters," *in IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 11, pp. 3114-3127, 2017.

[17] T. White, "Hadoop: The Definitive Guide. O'Reilly Media," 2009.

[18] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[19] K. Mahadik, S. Chaterji, B. Zhou, M. Kulkarni and S. Bagchi, "Orion: Scaling Genomic Sequence Matching with Fine-Grained Parallelization," *SC14: International Conference for High Performance Computing, Networking,* Storage and Analysis, New Orleans, LA, 2014, pp. 449-460.

[20] "National Center for Biotechnology Information," [Online]. Available: http://www.ncbi.nlm.nih.gov/, 2015.

[21] Kajdanowicz, T.; Indyk, W.; Kazienko, P.; Kukul, J., "Comparison of the Efficiency of Map Reduce and Bulk Synchronous Parallel Approaches to Large Network Processing," *Data Mining Workshops (ICDMW), 2012 IEEE 12th International Conference on*, Dec. 2012, pp. 218-225.

[22] S. Dooms, T. De Pessemier, and L. Martens, "Movietweetings: a movie rating dataset collected from twitter," *in Workshop on Crowdsourcing and Human Computation for Recommender Systems*, CrowdRec at RecSys, vol. 13, 2013.

[23] G. Zhai, L. Tian, Y. Zhou, Q. Sun and J. Shi, "A computing resource adjustment mechanism for communication protocol processing in centralized radio access networks," in *China Communications*, vol. 13, no. 12, pp. 79-89, Dec 2016.