Human-centric Computing and Information Sciences

RESEARCH Open Access

TMaR: a two-stage MapReduce scheduler for heterogeneous environments

Neda Maleki[1*], Hamid Reza Faragardi[2], Amir Masoud Rahmani[3,6,7], Mauro Conti[4] and Jay Lofstead[5]

*Correspondence: neda.maleki@srbiau.ac.ir
[1] Department of Computer Engineering, Science and Research Branch, Islamic Azad University, Tehran, Iran
Full list of author information is available at the end of the article

Abstract
In the context of MapReduce task scheduling, many algorithms mainly focus on the scheduling of Reduce tasks with the assumption that scheduling of Map tasks is already done. However, in the cloud deployments of MapReduce, the input data is located on remote storage which indicates the importance of the scheduling of Map tasks as well. In this paper, we propose a two-stage Map and Reduce task scheduler for heterogeneous environments, called TMaR. TMaR schedules Map and Reduce tasks on the servers that minimize the task finish time in each stage, respectively. We employ a dynamic partition binder for Reduce tasks in the Reduce stage to lighten the shuffling traffic. Indeed, TMaR minimizes the makespan of a batch of tasks in heterogeneous environments while considering the network traffic. The simulation results demonstrate that TMaR outperforms Hadoop-stock and Hadoop-A in terms of makespan and network traffic and achieves by an average of 29%, 36%, and 14% performance using Wordcount, Sort, and Grep benchmarks. Besides, the power reduction of TMaR is up to 12%.

Keywords: MapReduce, Hadoop, Heterogeneous systems, Scheduling, Performance, Shuffling, Power, Cloud computing

Introduction

Today, we are surrounded by a massive amount of data which are produced by social media, web surfing, embedded sensors, IoT nodes, and so on. According to the International Data Corporation (IDC) report in 2017, the size of the world's information is increasing and would be 140 ZB by 2050 [1]. Such a huge volume of data necessitates a substantial scaling of the resources horizontally [2] in which the massive produced data can be processed in parallel on distributed machines. One of the most popular parallel and distributed frameworks is MapReduce introduced by Google in 2004 [3]. Hadoop [4] is an open-source implementation of the MapReduce for cloud computing. Each MapReduce job consists of two dependent phases, Map and Reduce. The user-defined Map and Reduce tasks are distributed independently onto multiple resources in a tree-style network topology for parallel execution. The Shuffle phase performs an all-to-all remotely fetching of intermediate data from the Map phase to the Reduce phase. It involves intensive data communications (flows) between resources and can significantly delay job completion. Therefore, effective use of resources such as computation and the network is a critical factor in MapReduce performance which can be significantly enhanced by task scheduling and flow scheduling respectively [5, 6]. A preferred task scheduling results in a better performance which is measured by the makespan. Moreover, in the shuffle phase, the data transmission time from a source to a destination across the network directly influences makespan [7].

A growing emphasis on a variety of MapReduce jobs and the inclusion of different configurations of nodes in the existing cluster has led to an increased acceptance of the heterogeneous environment. Heterogeneity in a system is introduced due to the presence of resources that have different characteristics, including speed, memory space, special processing functionalities, etc. [8]. The considered heterogeneity includes two factors, (1) the processors in the network are not identical and have different computation power which can result in different execution times for running the same task. (2) different types of jobs in terms of being CPU or IO-intensive where the tasks have different input data sizes which can significantly affect the performance of Hadoop scheduler and limit the overall throughput of the system. Therefore, in a heterogeneous system with multiple tasks belonging to various jobs, designing an efficient scheduling algorithm is a vital challenge [9–12].

HadoopMR and Hadoop YARN are two versions of MapReduce implementation that offer three levels of scheduling: (i) User-level, (ii) Job-level, and (iii) Task-level, as shown in Fig. 1.

According to the taxonomy, there are two built-in schedulers including HFS (Hadoop Fair Scheduler) and HCS (Hadoop Capacity Scheduler) [4, 13, 14] as user-level scheduling. The objective of HCS is to maximize the resource utilization and throughput in a multi-tenant cluster environment by applying separated queues/pools to each user while guarantees the minimum required capacity. However, HCS does not guarantee the resource efficiency that could lead to unnecessarily idle resources and inefficient scheduling. Therefore, HFS was proposed to provide a fair share of cluster capacity over time among the users. HFS is a preemptive algorithm useful in environments with different types of jobs. The separated security mechanisms in terms of control access are applied in each queue which avoids any interference of users' jobs.

FIFO (First In First Out) and Priority schedulers are the two built-in scheduling algorithms designed for scheduling of jobs. FIFO as the Hadoop scheduler schedules users' jobs based on their order of submission. In FIFO, since there is only one queue for all users' jobs, the preemption (priority) is not supported. Hence a long-running job makes delay the completion time of the other jobs. The Priority scheduler assigns the free resources to the job that has the fewest running tasks to ensure that the cluster is shared fairly between jobs. Priority scheduler allows the small jobs to finish in an optimal time while does not make the big jobs being starved.

The scheduling tasks of a job by considering different criteria such as performance (completion time), locality, network traffic, cost, etc., is the third level and fine-grained of scheduling. There are three levels of tasks scheduling, including, Map, Reduce, and Speculative tasks scheduling. At the Map task scheduling level, the default Hadoop scheduler is based on the data locality criteria, i.e. it selects the local Map tasks for a given resource by inquiry the meta-data service to find the hosted data chunks. However, Hadoop randomly selects the Reduce tasks of the selected job

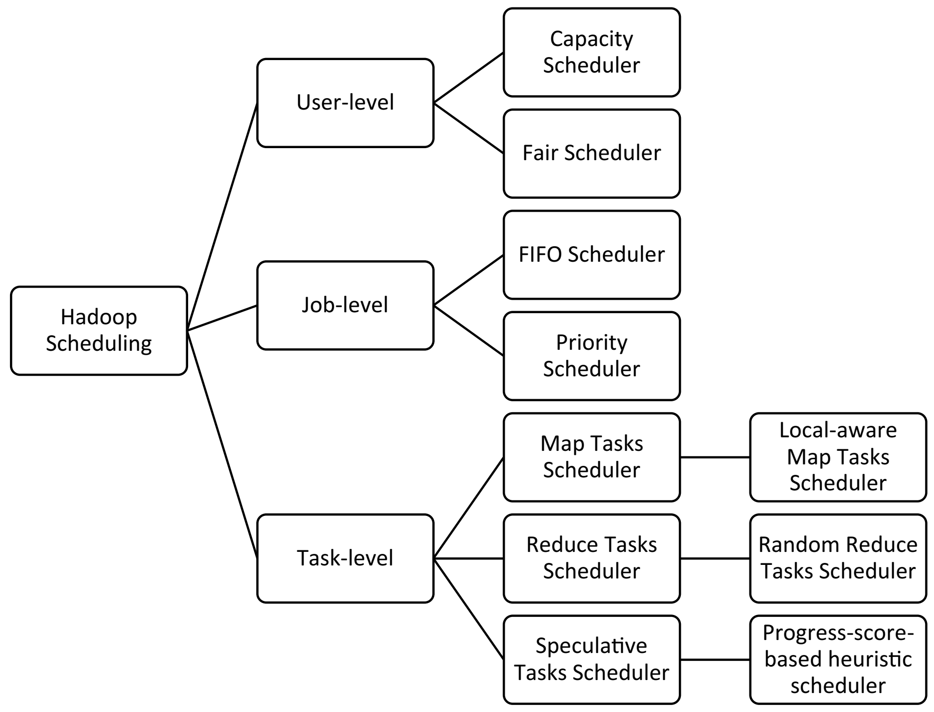


Fig. 1 Levels of Hadoop MapReduce Scheduler

for scheduling on the available resource. To reach better turn-around time through higher parallelism, once a Map task execution is completed, the Reduce task scheduler starts shuffling the intermediate data.

The studies that aim at improving the parallel performance of MapReduce job either try to schedule only the Reduce tasks to diminish data transmission cost in the shuffle phase [7, 15–17] or try to minimize the job completion time by only considering scheduling Map or Reduce tasks [18–23] (see "Related work" section). They usually focus on only the assignment of Reduce tasks with the assumption that Map scheduling is determined by the initial data distribution of the file system hosted on the MapReduce compute nodes. However, this assumption is not valid in the cloud or high-performance environments since the input data often resides in a remote shared file system such as Lustre [24] or Amazon S3 [25]. In such a setup, since all the data is loaded from remote locations, the scheduling of Map tasks also becomes important [26].

To the best of our knowledge, there are only a small number of scheduling algorithms considering both Map and Reduce tasks scheduling simultaneously in the literature. Furthermore, there is still much room for improving the performance of MapReduce in terms of minimizing the makespan while considering network traffic in heterogeneous environments. In this paper, we propose a scheduler which aims to decrease the entire tasks completion time (makespan) by reducing the execution time of Map and Reduce stage individually while considering network traffic in heterogeneous environments. The main contributions of this paper are as follows:

1. An improved scheduler for heterogeneous environments while almost most of Hadoop schedulers are designed for homogeneous environments.
2. A dynamic partition binder for Reduce tasks to reduce the network traffic in the Shuffle phase.
3. A two-stage Map and Reduce scheduler for improving makespan which works in polynomial computation time.
4. A power-aware selection of resources to minimize the total power consumption of the cluster.

The paper is organized as follows: In "Related work" section, a brief review of related work is presented. "Problem statement and system model" section introduces the problem statement and formulating the problem. In "Proposed solution" section, our proposed scheduler is introduced. "Performance evaluation" section shows the results of the simulations experiments. Finally, the main results are discussed, and directions for future work are presented in "Conclusions" section.

Related work

A large number of studies [18–23, 27–30] have been conducted to minimize the makespan of jobs and improve Hadoop performance. We classified the works into two categories: (i) Studies ignoring resource and workload heterogeneity, (ii) Studies considering the heterogeneity in terms of resource and workload.

First category. Studies [18, 21] have presented a Johnson-based method which aims to minimize the makespan of MapReduce jobs. The proposed static scheduler is inspired by the two-flowshop problem where the Map and Reduce execution stage is known in prior. In [18], a heuristic Johnson-based method is introduced where separated pools (called Balanced Pools) are employed to minimize the makespan of jobs of each pool. However, their proposed method is not optimal, and it cannot minimize the overall makespan. The study [21] has proposed a modified Johnson algorithm which minimizes the overall makespan of users' jobs. The deficiency of the scheduler is that to achieve the minimum makespan, it places all types of users' jobs in one work queue, and it shares all capacity of the cluster between jobs while ignoring the type of jobs and their data size.

In [19], authors have proposed an approximation algorithm for scheduling tasks to minimize makespan and total completion time. Authors assumed that Reduce tasks are non-parallelizable, whereas Map tasks are parallelizable in a homogeneous Hadoop cluster. The preemption of jobs has been taken into account to achieve fairness of jobs. In [20], Jiang et al. have presented an online scheduler with the objective of minimization of makespan of MapReduce jobs. Authors have considered both preemptive and non-preemptive Reduce tasks in a homogeneous Hadoop cluster. The proposed scheduler is optimal for cluster up to two nodes while the scalability and heterogeneity have not been considered. The proposed methods consider neither the heterogeneity of resources nor jobs.

Second category. In [22], the authors proposed a static task scheduler inspired by the bin packing problem to minimize the makespan while considering the heterogeneity of the cluster. In this method, first, the Reduce tasks with higher execution time

(Large Reduce First) and their related Map tasks are assigned to the nodes with the more top speed. Afterwards, the same process is repeated for other remained Map and Reduce tasks subsequently to minimize the makespan. To achieve a shorter makespan, authors have assumed that Map tasks are parallelizable and can execute on multiple machines.

In [23] the authors proposed a multi-objective scheduling algorithm in MapReduce-based cloud environments. In the proposed model, the job completion time and cost of cloud services have been considered to minimize the makespan of tasks of a job. Compared to FIFO and Fair schedulers, the scheduler achieves higher tasks throughput and is cost-effective in terms of resource usage by cloud users. The proposed scheduler model is designed for only one job while there are many and different jobs in the MapReduce cluster.

With the premise of improving Hadoop performance in terms of makespan, Yao et al. [27] have presented a new scheduler for a batch of MapReduce jobs. The proposed schedulers use the information of requested resources, resource capacities and dependency between tasks which constitutes the tasks' fitness for scheduling. Authors have conducted experiments under different workloads but have not considered the resource heterogeneity.

In [28] authors have considered the Map tasks scheduling problem of MapReduce jobs to obtain network traffic and tasks throughput optimally in the heterogeneous environment. The scheduler is based on the Shortest Queue and the MaxWeight policy. It can achieve the full capacity region and minimization of the expected number of backlogged tasks in the considered heavy-traffic regime. However, authors have not considered the Reduce tasks scheduling problem that is the leading cause of network cost in the shuffling phase.

Authors in [29] proposed a Map tasks locality-aware scheduler, TSMJS, a Time-Sharing MapReduce Job Scheduler to minimize the makespan by mitigating the amount of intermediate data in the shuffle phase. Since for combining the records produced in the shuffle phase, there is a per-combiner memory reservation need, the idea is minimizing the non-local Map tasks on a node to ensure the least number of combiners. Authors have considered the Map tasks scheduling problem in the cloud environment with heterogeneous workload; however, they have not considered the Reduce tasks scheduling.

In [31] the authors have proposed two consolidation-based techniques to reduce power consumption. The two methods are based on the Best Fit Decreasing (BFD) approach. In the first technique, called Minimum Power BFD (MPBFD), servers with the lowest power consumption are selected for consolidation. In the other method, called the Maximum CPU Capacity BFD (MCBFD) technique, servers with the highest capacity of computing are chosen. The authors have determined both the upper and lower threshold value to avoid the violation of the SLA through migration and to reduce power consumption by turning the underutilized servers off, respectively.

Liaqat et al. [32] extended the Nova scheduler to propose a multi-resource based VM placement approach to improving application performance in terms of the central processing unit (CPU) utilization and execution time in the heterogeneous environments. The authors have designed three modules, including, Compute Load (CL), Load Analyzer (LA), and Load Filter (LF) for implementing their VM placement architecture.

Nita et al. [33] proposed a multi-objective scheduling algorithm of many independent MapReduce tasks, called, MOMTH, for big data processing in the heterogeneous system. The objectives of MOMTh is avoiding resource contention and having an optimal workload of the cluster while meeting the deadline and budget constraints.

In [15], the authors presented Hadoop-A, an acceleration framework to optimize Hadoop by removing the sequence between the Shuffle phase and the Reduce phase. Hadoop-A uses high-speed hardware called RDMA, which is based on Infini-band communications, to get faster access to the output of the Map tasks by the Reduce tasks. The framework uses an external queuing algorithm based on the priority queue to remove the number of disk accesses and duplicate mergers in the Reduce phase.

A comprehensive study for makespan minimization has been conducted in [30]. Authors have done a systematic literature review on the Hadoop platform and investigated the solutions to enhance Hadoop performance in terms of makespan and network traffic by introducing new and robust existing methods in the task and job scheduling. A survey of metaheuristic-based schedulers for MapReduce jobs and the comparative analysis have been provided in [34]. The schedulers such as Ant Colony Optimization (ACO), Genetic Algorithm (GA), Particle Swarm Optimization (PSO), League Championship Algorithm (LCA), and BAT algorithm can find near-optimal solutions in many areas such as Grid, Cloud, and distributed environments for minimizing makespan of jobs. However, since the metaheuristic solutions take a long time to find an optimal solution due to the large solution space or non-optimal fitness function, we employed the greedy heuristic solution which is suitable for scheduling problem in a short time.

Problem statement and system model

Problem statement

A scheduling model consists of (i) multiple applications, (ii) a target computing environment, and (iii) one or more performance criteria for scheduling. Let's suppose that there is a set of Map and Reduce tasks of different jobs in a MapReduce-based Hadoop cluster. The Map tasks could be executed in parallel on a set of heterogeneous resources. For the logical correctness of the MapReduce programming model, the Reduce tasks can start only once the entire Map tasks execution is completed. Considering a MapReduce job where the makespan is equal to the Job Completion Time (JCT), there are four options to minimize makespan shown in Fig. 2. Let us look at these options:

1. Defining the optimal number of Map tasks.
2. Defining the optimal number of Reduce tasks.
3. Reducing the execution time of the last Map task.
4. Reducing the execution time of the last Reduce task.

In Hadoop-stock [4], the input dataset is divided into equal parts, called Splits. Since each Map task is responsible for processing one split, the number of Map tasks of a job is predefined by the system which is equal to the number of splits (see Eq. 1).

$$\#\text{ of Map tasks} = \frac{\text{Input data size}}{\text{Split size}} \tag{1}$$

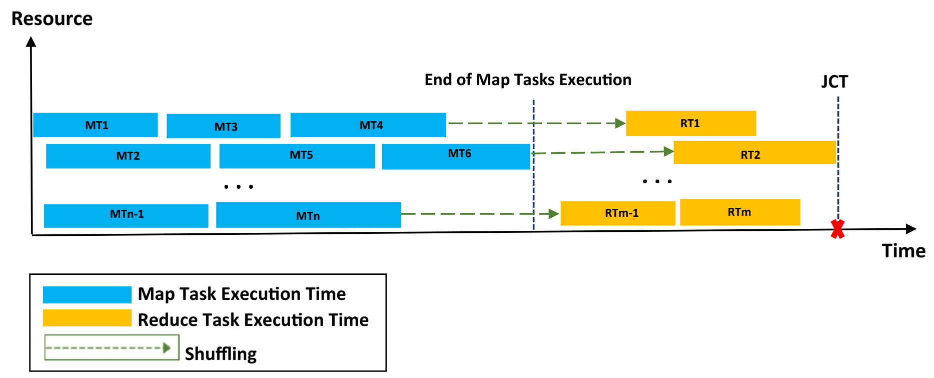


Fig. 2 Minimization problem

Therefore, the first option (Defining the optimal number of Map tasks) is out of our control and depends on the input data size and the configured split size. For example, if we have 1TB of the input file and the block size of the HDFS is 128MB, then number of input splits are (1024/128 =)8 input splits. Thus the number of Map tasks of the job is set to 8. However, reducing the block size from 128MB to 64Mb results in (1024/64 =)16, corresponding to 16 Map tasks. The second alternative (the number of Reduce tasks) is defined dynamically after all the output partitions of all Map tasks are produced, i.e., the number of Reduce tasks depends on the size of the partitions. Therefore, the optimal size of partition to be assigned to a Reduce task is system dependent, at least to some degree, [35]. However, there is a trade-off between performance in terms of improving storage performance (due to larger sequential I/O) and fault-tolerance in terms of the amount of computation that must be re-done when a Reduce task fails.

However, there are many solutions to define the optimal number of Map and Reduce tasks by defining the optimal split data size through applying the meta-heuristics and machine learning solutions which are discussed in the MapReduce parameter tuning research field which is not included in the scope of our paper. Therefore, only option 3 (Reducing the execution time of the last Map task) and 4 (Reducing the execution time of the last Reduce task) are targeted by TMaR, which is explained in "Proposed solution" section.

System model

We make several common assumptions in this study, given the relatively high complexity of MapReduce job scheduling. The problem can be formally described as follows. Given a set of n different jobs $J = \{J_1, J_2, \ldots, J_n\}$, which must be processed on m different computing nodes $N = \{N_1, N_2, \ldots, N_m\}$. Each node consists of some containers, and each job is assigned a logical container which is physically distributed among the processing cores of the nodes. A job is fractional which means that it can be arbitrarily split between the nodes (on its associated container), in other words, the parts of the same job can be processed on different nodes simultaneously. These parts are known as Map and Reduce tasks which are independent and executed in parallel. The heterogeneity has been modelled by assuming different runtimes of tasks on different processors. Reduce tasks can only be launched when all the Map tasks have been completed. Let M_j and R_j

be the sets of Map tasks and Reduce tasks of J_j $(1 \leqslant j \leqslant n)$ where the Map tasks can produce a set of partition sizes from the set of partitions $P = \{P_1, P_2, \ldots, P_i\}$ on their local disk after execution. $|M_j|$ and $|R_j|$ denote the number of tasks in M_j and R_j respectively which their summation shows the job size. The number of Map tasks M_j is defined by the size of input dataset Ij while the number of Reduce tasks R_j is specified in run time after the intermediate data are produced. Let C_j be the completion time of job J_j; our goal is to minimize the makespan, i.e., the maximum finish time of all jobs, $max_{1 \leqslant j \leqslant n} Cj$. As we consider only one job, then makespan is equal to the completion time of the job.

In our model, the task execution of a given application is assumed to be non-preemptive, i.e. the Map or Reduce task is not interrupted (paused or killed) during its processing [22]. Moreover, the data transfer rate (network bandwidth) between nodes of the cluster is stored in matrix C_B of size $m * m$ and the propagation delay of nodes is given in an m-dimensional vector L.

Proposed solution

The proposed framework is shown in Fig. 3. As seen, we simulate YARN [36] architecture since it optimally manages resource allocation, i.e., there is no fixed number of slots separately allocated for Map and Reduce tasks. Therefore, unlike Hadoop-stock, TMaR does not statically schedule the Reduce tasks (the number of Reduce tasks are defined in run time after the partitions are produced) which results in better utilization of available capacity by Map tasks. YARN uses a double-layer resource scheduling model: (i) Resource to Jobs scheduling, (ii) Resource to the tasks (of a job) scheduling. In the first layer, the resource scheduler in ResourceManager allocates resources per-application ApplicationMasters; then in the second layer, ApplicationMasters will allocate containers to each task of their jobs. TMaR focuses on resource allocation in the second layer. How to set appropriate resource requirements for each job in the first layer is out of the scope of our study, and we assume it is completely determined by ResourceManager (our future work). Therefore, according to Fig. 3, when the jobs are submitted by the Client to the system and placed in the queue, one AppMaster is assigned to each job, and a container is allocated to the AppMaster. The container is a logical concept, and it is indeed physically distributed as the cores of the nodes in the system. For example, the logical yellow container which is assigned to AppMaster2 in the first layer is shown distributed on the nodes in the second layer.

First, Client copies the input files into the Hadoop file system (HDFS) where the files are divided as splits, and they are scattered on the nodes of the cluster. Each node consists of a NodeManager that reports the status of the node to the ResourceManager. Then, Client queries the information of the execution time of the tasks of the submitted job for scheduling from the Preprocessing stage and send the required information to the two-stage scheduler. In the first stage, TMaR schedules the Map tasks list of the job using the Map stage scheduler (Algorithm 1) and write the output on the local hard disk of each node as Map Output File (MOF). According to Fig. 4, MOF consists of a key-range sub-partitions (partition 1, partition 2, ...).

MOFs should be assigned to the Reduce tasks of the job for processing. Since in TMaR the Reduce tasks are not statically scheduled, after all the sub-partitions of MOFs are ready, it calculates the size of a partition and determines the number of required Reduce tasks for each partition. Therefore, by using the Reduce tasks execution time

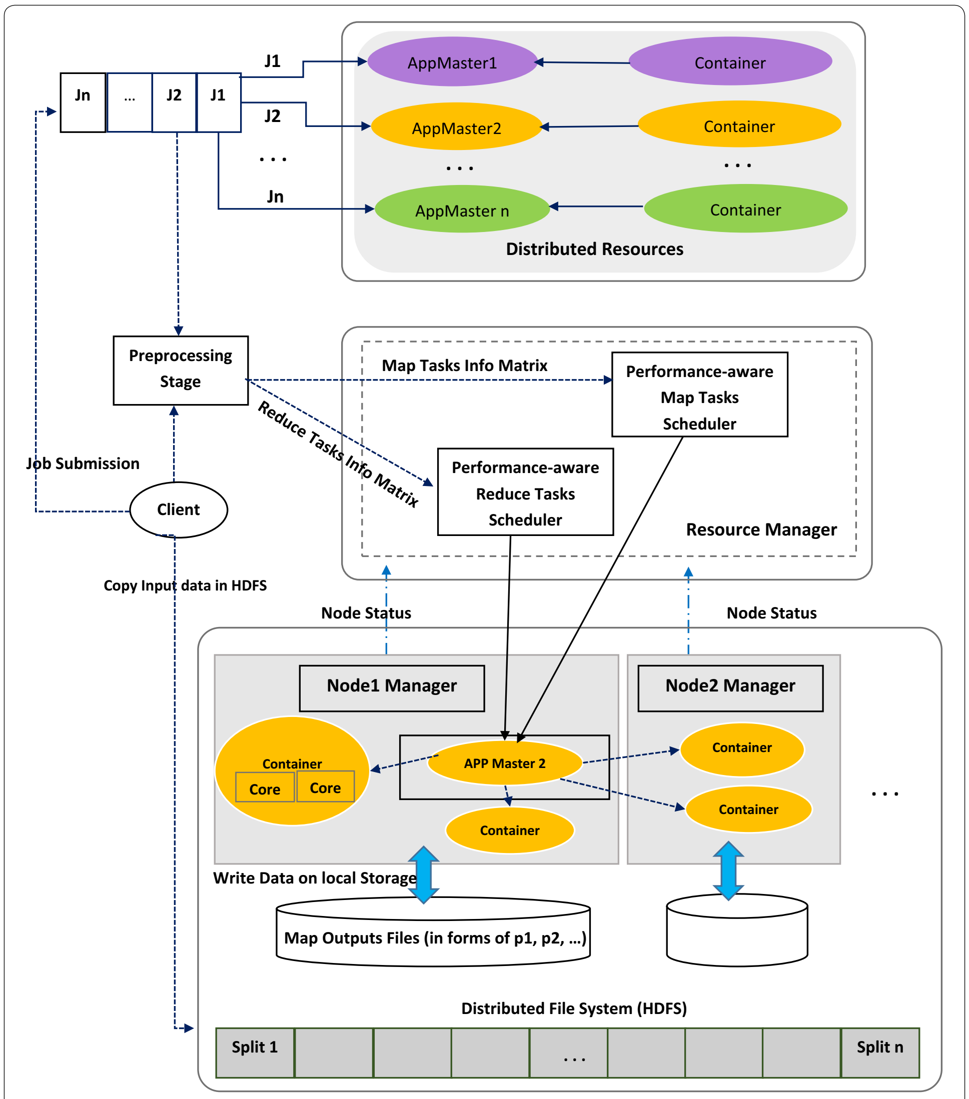


Fig. 3 Representation of the proposed framework, TMaR

from Preprocessing stage, the Reduce tasks are scheduled using Reduce stage scheduler (Algorithm 3 and Algorithm 4).

Preprocessing

In the context of static scheduling, we need to know prior to the execution time of tasks of a job, i.e. the Map task and Reduce task for making the decision. According to [38], authors conducted a comprehensive MapReduce job profiling by executing a smaller input dataset and observed the execution time of all phases of the job, i.e. initialization, Map, shuffle and Reduce. We name this initial calculation stage as Preprocessing stage and store the obtained information into the *Map_Matrix* and *Reduce_Matrix*, respectively. We will use the Matrixes as the input of the Map and Reduce stage scheduling algorithm, respectively. We note that, since under the mixture of workload, the size of produced intermediate data, i.e. the Map output data partition size of a job is not a good

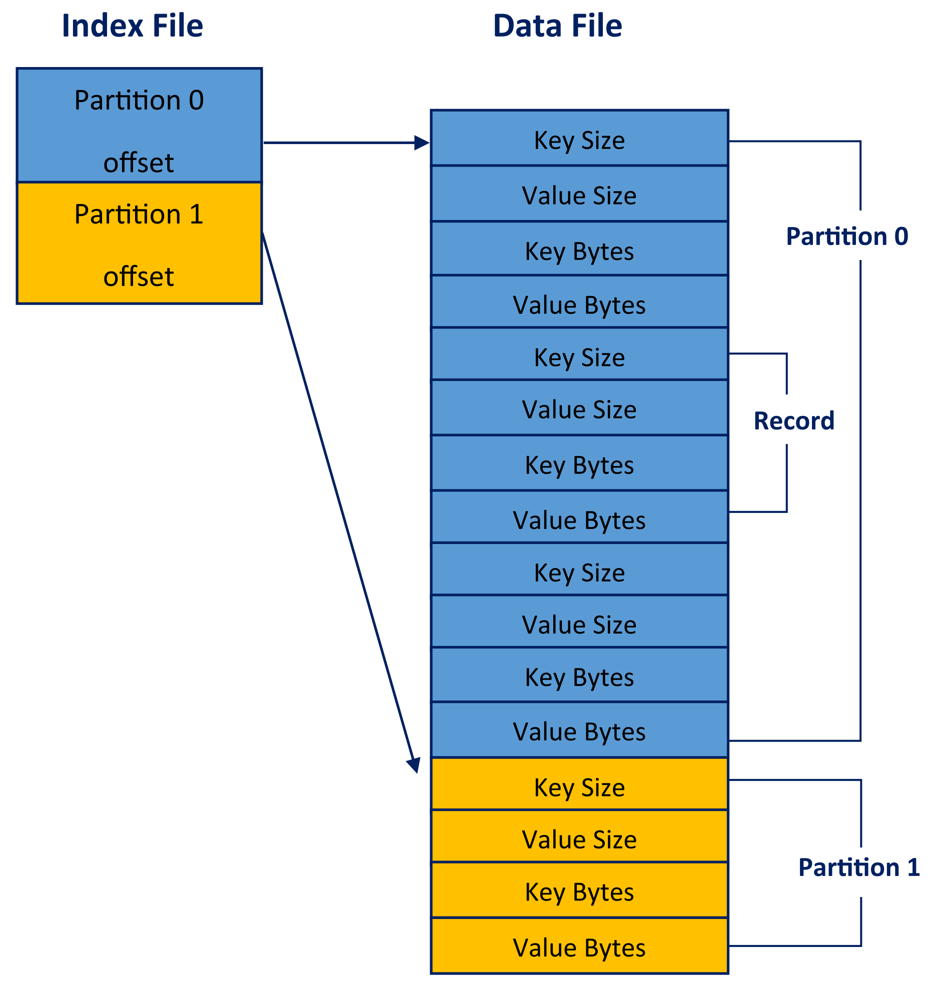


Fig. 4 Map Output File (MOF) Format [37]

indicator of the execution time of its Reduce tasks, we did profiling both on the CPU-intensive and IO-intensive benchmarks to better estimate the Reduce task execution time. This considering is required as in the Reduce stage scheduler, for defining the partition placement on heterogeneous resources, we need the Reduce task execution time information.

Map stage scheduler: Algorithm 1

According to [39], *Map Selectivity* is defined as the Map output compression ratio, i.e. the average number of records output by Map tasks per input record. All Map tasks of a job possess the same *M*ap selectivity [7] i.e., they process the same amount of data and do the same functionality. Then, the only effective criterion on Map task execution time would be the node speed. Moreover, all Map tasks of a particular job have the same execution time on a specific machine. Inspired by the original algorithm proposed by Topcuoglu et al. [40] and with the Map tasks execution time information in Preprocessing phase, our Map stage scheduler places the Map tasks such that the finish time of each task is minimized (see Algorithm 1, line5). After the task assignment, the status of the cores in matrix C_A is updated (line 6).

Algorithm 1 Map Tasks Scheduler

```
1: Inputs: set of Unscheduled Map tasks {UMT}, vector of cores speed $C_S$, $Map\_Matrix$
2: Output: The scheduled Map Tasks on a heterogeneous system.
3: Core available time vector $C_A$ with initial value of zero
4: repeat
5:     Select from {UMT} and assign it to the core which minimizes the Map task execution time according to $Map\_Matrix$.
6:     Update $C_A$
7: until there are unscheduled tasks in the {UMT}
```

In the situations where there is more than one candidate that satisfies the time minimization, we select the resource with less power consumption after task assignment. To do this, it is enough to obtain the power consumption of cluster after assigning the task to the host which has been recognized as "best host (min)" from the previous selection (line 9), and compare it with the power consumption of the host with the same condition (in terms of time minimization) as "new host" (line 10). Then, select the one with a lower value. We call this Algorithm $TMaR^+$, which is an extension of TMaR in terms of power improvement by importing the lines 6–11 in the Map Scheduler (Algorithm 2).

Algorithm 2 $\mathrm{TM}a\mathrm{R}^+$

```
1: If (There is more than one candidate which minimizes the time)
2: Put the candidates into the $P_{list}$
3: /*Choose the one that minimizes the power consumption of the cluster*/
4: CPC (Task to bestHost)= Pow (cluster)+ PAS (bestHost.Task)
5: CPC (Task to newHost)= Pow (cluster)+ PAS (newHost.Task)
6: End
```

CPC function returns the total cluster power consumption, and the Pow function returns the current power usage of a host. However, our system is not DVFS-enabled (CPUs can be operated at different speeds at runtime) and when a task is running on a resource, its execution is completed at full capacity of the resource. But, one can apply the DVFS technology for better power saving, and it can be implemented subject to the platform. Therefore, we define the general linear power model, according to Eq. 2.

$$\mathrm{PAS} = h_{static} + (h_{max} - h_{static}) * U(t) \tag{2}$$

Where *PAS* is the power consumption after task assignment, h_{static} is the static power of host, h_{max} is the maximum host power consumption, and $U(t)$ is the CPU utilization level at time t. Therefore, the resource with the lowest cost in terms of power consumption is prioritized. At line 7, the P_{list} is a list of the proper candidates which are homogeneous in terms of makespan minimization but, heterogeneous in terms of power consumption.

It is worth mentioning that the cluster is more load-balanced compared to the situation where there is no priority metric to select from the proper candidates. We illustrate this problem, according to Fig. 5. Let suppose we have eight homogeneous

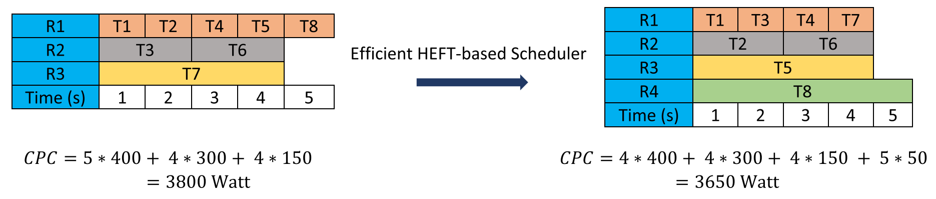

R1
T1
T2
T4
T5
T8
R2
T3
T6
R3
T7
Time (s)
1
2
3
4
5
Efficient HEFT-based Scheduler
R1
T1
T3
T4
T7
R2
T2
T6
R3
T5
R4
T8
Time (s)
1
2
3
4
5
CPC = 5 * 400 + 4 * 300 + 4 * 150 = 3800 Watt
CPC = 4 * 400 + 4 * 300 + 4 * 150 + 5 * 50 = 3650 Watt

Fig. 5 Map Scheduling using *TMaR*$^+$ [37]

Table 1 Example of task execution time on heterogeneous resources

	Resource 1/400 W (s)	Resource 2/300 W (s)	Resource 3/150 W (s)	Resource 4/50 W (s)
T_1	1	2	4	5
T_2	1	2	4	5
T_3	1	2	4	5
T_4	1	2	4	5
T_5	1	2	4	5
T_6	1	2	4	5
T_7	1	2	4	5
T_8	1	2	4	5

tasks and their execution time on the four heterogeneous resources with their power consumption per 1 unit task processing is available in *Map_Matrix*. The *Map_Matrix* looks like the Table 1.

As shown in Fig. 5, when we consider power in the same condition of candidates, the R_4 is turned on as compared to the R_1, it consumes less power for processing $T_8(5 \times 50 \text{ vs } 1 \times 400)$. Also, there is a load balancing between the resources while the makespan remains the same (makespan=5). Therefore, it is true that the objective of the solution is to minimize the makespan of tasks, such implicit improvement in terms of power consumption does not contradict the objective.

Reduce stage scheduler: Algorithm 3

In Hadoop-stock, the shuffle phase will start once the produced Map outputs meet a predefined threshold. The threshold is defined as a percentage of Mappers that have finished their execution. Since TMaR's goal is to define the number of Reducers dynamically subject to the partition size, it schedules Reduce tasks when all Map outputs are produced. A partition size is calculated by aggregating the related sub-partitions scattered on the nodes of the cluster. Therefore, TMaR requires that Reduce tasks are launched on the node that hold the corresponding shuffled sub-partitions. To this end, TMaR breaks the static binding of Reduce tasks in job initialization and provides dynamic Reduce partition binding. TMaR employs Reduce Partition Binding (PRB) approach that assigns partitions to Reduce tasks at the time of dispatching. It determines the number of Reducers based on the hosted partition size and spawns them in run-time to be assigned to the partition. Such binding reduces the network traffic in the Shuffle phase and also guarantees a data local Reduce tasks scheduling.

After determining how Reducers are assigned to a partition, we should first decide on which node the reducers finish time will be minimized. According to Algorithm 3, we sort the partitions in descending order by size (line 3). Next, for each partition (line 4), we calculate the Reducers finish time using PRB algorithm and schedule the Reducers on the node that will return the minimum finish time (line 6) (see Algorithm 4). To achieve this goal, by knowing the Reduce tasks start time and Reduce tasks execution time, we can define the Reduce tasks finish time according to Eq. 3.

$$\text{Reducers Finish Time} = \text{Reducers Start Time} + \text{Reducers Execution Time}. \tag{3}$$

- **Reducers execution time:** We can simply calculate the Reducers execution time related to a partition, since in the Preprocessing stage we have obtained the Reduce tasks execution time on each node of the cluster and maintain the information in *Reduce_Matrix*.
- **Reducers start time:** For calculating Reducers start time, we only need to find the first Reduce start time as all other Reducers will execute on the same node in parallel on the free cores or waited in the resource queue until it becomes free. We calculate Reducers start time according to Eq. 4:

  $$\text{Reducers Start Time} = Max(\text{Resource Available Time, Partition Transfer Time}). \tag{4}$$

  It means that the Reducers start time depends on the maximum time of two factors: (a) Time elapsed to transfer the sub-partitions related to a partition from other nodes to a specific node; (b) Time at which the resource will be available.

  (a) **Partition transfer time:** We calculate transfer time of data between two nodes by Eq. 5:

  $$\text{Transfer Time}_{a,b} = L_{a,b} + \frac{\text{Partition Size}}{|BW|_{a,b}} \tag{5}$$

  where the $L_{a,b}$ is the propagation delay between two resources C_a and C_b. Notably, since for starting the Reducers execution time the total data related to a key-range is required, the maximum time required for transferring all the sub-partitions of a partition to a resource are taken into account.

  (b) **Resource available time:** After all Map outputs are ready and the resources are load-free, we can decide on which resource the Reducers should be scheduled. Each time by placing each partition on a proper resource i.e., its related Reducers finish time, we update the available time of the resources (line 7).

Table 2 Algorithm variables

Variable	Description
Res_{min}	Resource on which the execution time of Reduce Tasks is minimized
R_F	Reduce tasks finish time
R_S	Reduce task start time
R_E	Execution time of Reduce tasks
min	Temporary variable to hold minimum of Reduce tasks finish time related to a partition hosted on a node
N_R	Number of Reduce tasks
N_{Res}	Number of resources of the cluster
R_{split}	System parameter which determines the number of Reducers required for a partition

Algorithm 3 Reduce Tasks Scheduler

```
1: **Inputs**: vector $C_A$ indicating cores available time.
2: **Output**: The scheduled Reduce Tasks on a heterogeneous system.
3: $\{P\}$= Gather partition sizes from all resources, sort them in descending order, and put into the list $\{P\}$.
4: **for** each partition $P_i$ in $\{P\}$ **do**
5:     /*Calculation of $R_F$ using PRB function*/
6:     $(R_F, Res_{min})$= PRB().
7:     Update $C_A(R_F, Res_{min})$.
8: **end for**
```

The variables used in the Algorithms are presented in Table 2.

Performance evaluation

Simulation setup

To achieve an efficient simulation that addresses various scenarios, the choice of a robust simulator is essential. Cloudsim [41] is an event-driven and java-based simulation environment which supports modeling and simulation of different resource provisioning schemes and workload descriptions. CloudSim enables the consideration of MapReduce as well as physical data simulation and modeling of the latency of physical and virtual machines, networks, and data storage devices in a large-scale distributed environment [42]. According to [29] many MapReduce papers have evaluated their work through simulation [22], either using Cloudsim [43, 44] or its derivations CloudsimRT [42], CloudsimEX [45], and CloudsimMR [46]. We developed TMaR by extending the CloudSim and designed completely all the required classes to implement TMaR. The fundamental classes of TMaR are TaskDispatcher, TaskSchedule, KeyValuePair, JobSpec, MapTaskInfo, PartirionInfo, ReduceTaskInfo, and NetworkInfo. We have implemented TMaR using Java (JDK 1.8) on a laptop with Windows 10 Operating system at 2.7 GHz quad core and 16 GB main memory running a 64 bit version of Windows 2018. The efficacy of TMaR, is compared to Hadoop-stock and Hadoop-A [15]. We chose Hadoop-stock since it considers both Map and Reduce tasks scheduling, runs in polynomial time, and has been used as baseline in many related work [14, 18–20, 22, 23].

Algorithm 4 PRB()

```
1: Inputs: matrix $C_B$ to represent network bandwidth of nodes, vector of cores speed $C_S$, and vector $C_A$ indicating
   cores available time, list of partition sizes $P_i$, $Reduce\_Matrix$.
2: Output: A pair of Reduce tasks finish time and the resource on which the partition processing time is minimized
   $(R_F, Res_{min})$.
3: /*Calculate number of Reduce tasks related to this partition*/
4: $N_R = \left\lceil \frac{\text{partition size}}{R_{split}} \right\rceil$
5: $min = \infty$
6: $R_F = 0$
7: for each $N_{Res}$ do
8:     if (there are not enough available cores for all $N_R$ Reducers) then
9:         according to $C_A$ put the extra Reduce tasks in the core queue
10:    end if
11:    /*Calculate execution time of $N_R$ Reducer tasks*/
12:    Execution time of $N_R$ Reducers=⌈(⌉ $\frac{N_R}{\text{number of available cores}}$) (Reduce task execution on the current node according to
       $Reduce\_Matrix$)
13:    Reduce start time=$Max$ (core available time according to $C_A$, received time of all $P_i$ from the cluster nodes
       using $C_B$)
14:    $R_F = R_S + R_E$
15:    if ($R_F < min$) then
16:        min=$R_F$
17:    end if
18: end for
19: $R_F$=min
20: return $(R_F, Res_{min})$
```

Environment and workload description

To evaluate TMaR, since the objective is makespan minimization while considering the network traffic, we consider both the heterogeneity of workload (different jobs) and environment. The jobs are heterogeneous in terms of CPU/IO-intensive and shuffle-light/heavy and the environment is heterogeneous in terms of the processing power of hosts, represented by MIPS. For the simulations in homogeneous and heterogeneous environments, the following settings were adopted respectively: each host in a homogeneous environment is an Intel Xeon@2.4 GHz processor and each host in heterogeneous environments consists of the following Intel Xeon types in a round-robin distribution: 1.2 GHz, 1.7 GHz, 2.4 GHz, 2.7 GHz, 3.6 GHz. We chose the processing power of homogeneous system of 2.4 GHz since it is an average computing power compared to the heterogeneous system resources and results in a fair comparison conditions.

Besides, we practically assess the scalability of TMaR, by different data sizes varying from 1GB to 10GB in three different sizes of environment i.e. small, medium, and large. For small environments, we considered that they have 10 hosts, in medium sized environments, these numbers are 20 hosts and in large environments these values are 30 hosts. In each scenario, we create a YARN environment with the homogeneous and heterogeneous hosts and all hosts were interconnected by a Gigabit Ethernet (125 MBps). Since we focus on CPU utilization in this work, we assume each YARN resource container has unlimited memory space.

According to Eq. 1, the number of Map tasks is determined by the input file size and the HDFS block size, i.e. 128MB for all the scenarios. If the intermediate data size is large, then more data needs to be shuffled from Map tasks to Reduce tasks. We call such jobs *shuffle-heavy*. Shuffle-heavy applications tend to use more networking and IO resources. Therefore, according to [7] we generate the Map intermediate data using

Table 3 Application characteristics

Job	Description	CPU/IO-intensive	Shuffle-light/heavy
Wordcount	Counts the occurrence of each word in the input data	CPU-intensive	Shuffle-heavy
K-means	A clustering analysis algorithm for multi-dimensional numerical samples in data mining	CPU-intensive	Shuffle-light
TeraSort	A popular benchmark to sort one terabyte of randomly distributed data	IO-intensive	Shuffle-heavy
Grep	Counts the number of occurrences of strings matching the target in a text file	IO-intensive	Shuffle-light

Table 4 Scenario description

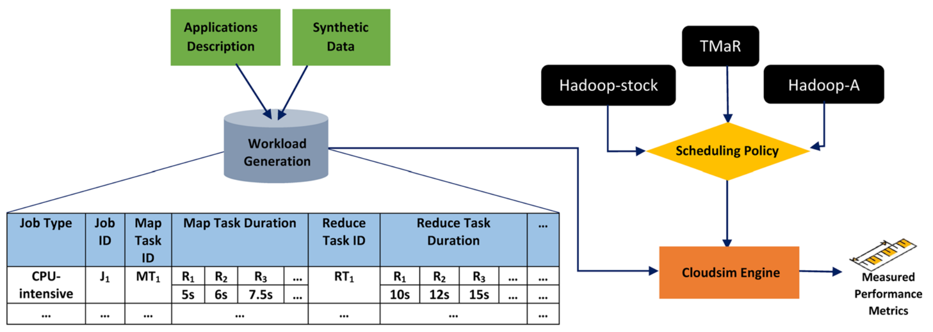
	Workload type	Input data size small, medium, large	# of Map Tasks
Scenario 1	Job is CPU-intensive and the produced intermediate data is high in the range of [30, 300] GB	<1 GB, 3 GB, 5 GB, 10 GB>	<8, 24, 40, 80>
Scenario 2	Job is CPU-intensive and the produced intermediate data is low in the range of [10, 30] GB	<1 GB, 3 GB, 5 GB, 10 GB>	<8, 24, 40, 80>
Scenario 3	Job is IO-intensive and the produced intermediate data is high in the range of [30, 300] GB	<1 GB, 3 GB, 5 GB, 10 GB>	<8, 24, 40, 80>
Scenario 4	Job is IO-intensive and the produced intermediate data is low in the range of [10, 30] GB	<1 GB, 3 GB, 5 GB, 10 GB>	<8, 24, 40, 80>

uniform distribution between *[10, 30]* and *[30–300]* Gigabyte as shuffle-light job and shuffle-heavy job, respectively. Since the Map output data size is an application-specific parameter and also depends to input data size, we define the shuffle degree of jobs based on the *MapSelectivity* (in short, *MS*) that $MS = 2$ and $MS = 0.5$ represents the shuffle-heavy and shuffle-light Map output data respectively. For example, by applying $MS = 2$, for a shuffle-heavy job with *10GB* dataset, the *20GB* intermediate data is generated by *800 Map* tasks. The selected applications exhibit different processing patterns and allow for a detailed analysis on a diverse set of MapReduce workloads. For example, *WordCount* and *TeraSort* are shuffle-heavy while *Grep* and *K-means* have a significantly reduced data size after the Map stage and therefore belong to the shuffle-light category. In addition, *WordCount* and *KMeans* are computation-intensive because their Map phase processing time is orders of magnitude higher than other phases. The benchmark characteristics and scenarios we use in these experiments are summarized in Tables 3 and 4, respectively.

Performance metrics

We measure the following two parameters as evaluation criteria:

1. Makespan: The total elapsed time required to execute the entire MapReduce job is called makespan. The makespan is calculated as follows: $Makespan = Max_{\forall ReduceTask \in N_R}\ \{R_F\}$. Where R_F, the Reduce task finish time is achieved using $R_F = R_S + R_E$ (Eq. 3).
2. Intermediate data processing time: It is the time required for remotely fetch the data produced by Map tasks and process it on the intended node.
3. Power consumption: The total power consumption of cluster when we apply $TMaR^+$.



Job Type	Job ID	Map Task ID	Map Task Duration				Reduce Task ID	Reduce Task Duration				...
CPU-intensive	J_1	MT_1	R_1	R_2	R_3	...	RT_1	R_1	R_2	R_3	...	...
			5s	6s	7.5s	...		10s	12s	15s	...	...
...	...	...	...				...	...				...

Fig. 6 Simulation design

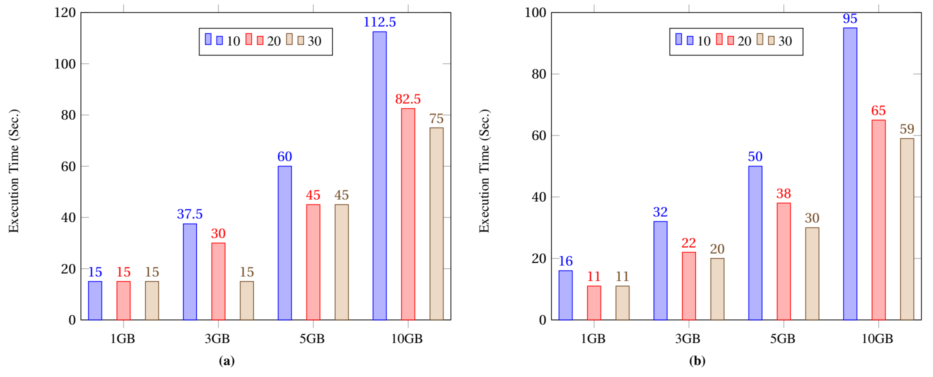


Fig. 7 **a** Wordcount execution time by varying data size in homogeneous environment. **b** Wordcount execution time by varying data size in heterogeneous environment

The overview of our simulation design is illustrated in Fig. 6.

Results analysis

Makespan

We analyze the experiments in two parts from two perspectives to consider the TMaR performance: (i) TMaR is evaluated under different cluster and dataset size in both homogeneous and heterogeneous environments with different kind of jobs (scalability), (ii) TMaR is compared to Hadoop-stock and Hadoop-A in terms of makespan and network traffic.

Part 1: Figures 7, 8, 9, 10 represent the total job execution time for Wordcount, Kmeans, Sort, and Grep in homogeneous and heterogeneous systems, respectively.

(a) *WordCount:* According to the Fig. 7a, TMaR provides less execution time by increasing the number of resources in the Wordcount application for all different input sizes. TMaR also achieves more performance for big amount of input data sizes by increasing the number of resources. The performance gain in large environment compared to small environment with 3 GB and 10 GB input size are equal to **+ 2.5X** and **+ 1.5X**, respectively. We should mention that in homogeneous environment with 1GB dataset, since the number of Map tasks are less than the resources,

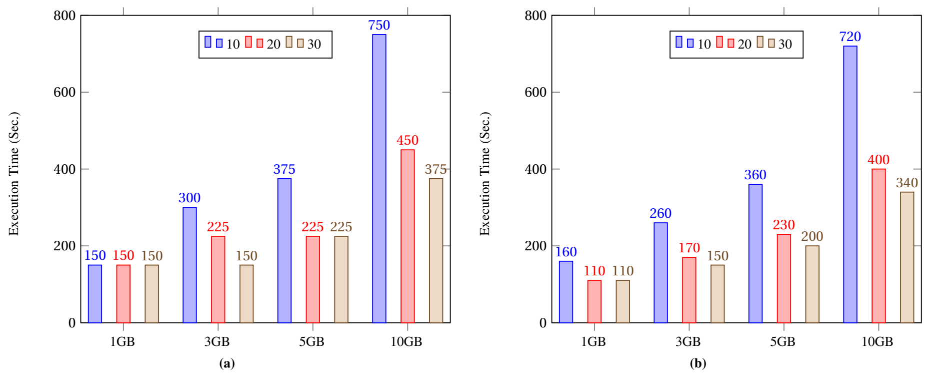


Fig. 8 **a** Kmeans execution time by varying data size in homogeneous environment. **b** Kmeans execution time by varying data size in heterogeneous environment

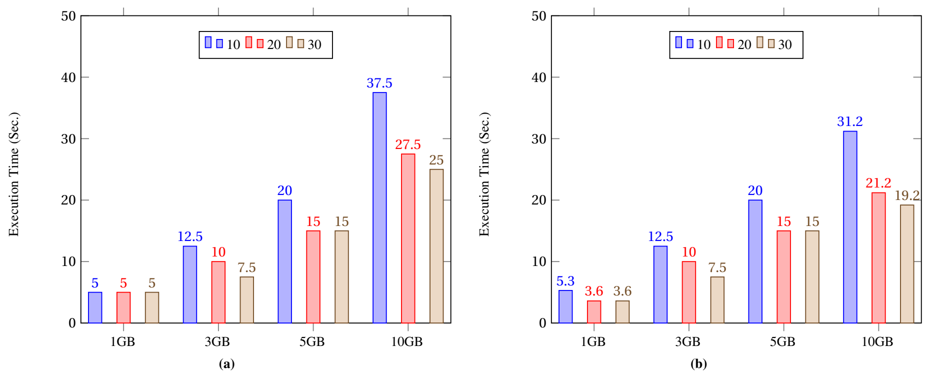


Fig. 9 **a** Sort execution time by varying data size in homogeneous environment. **b** Sort execution time by varying data size in heterogeneous environment

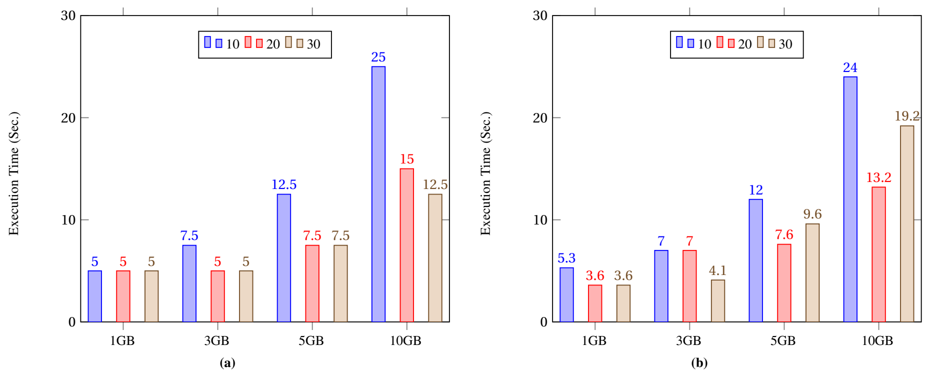


Fig. 10 **a** Grep execution time by varying data size in homogeneous environment. **b** Grep execution time by varying data size in heterogeneous environment

there is no performance in all cluster sizes. In Fig. 7b, the performance gain is more considerable in the heterogeneous environment and makespan is about **+ 1.2X** less compared to homogeneous system. The performance gain is **+1.4X** and **+ 1.6X** in large environment compared to small environment with 1 GB and 10 GB input size, respectively. The reason is that in heterogeneous environment, with greedy behavior of TMaR, the fastest resources are selected in each decision making which makes earlier the finish time of Map tasks and subsequently results in less finish time of Reduce tasks and overall makespan.

(b) *K-means:* K-means is divided into two main phases, the first phase is the iteration phase and the second phase is the clustering phase. In the iteration phase, the performance is a CPU-bound, which means the performance will increase if there is an increase in processing power such as an increase in the number of resources. This is perceptible in Fig. 8a, b with **2X** and **2.11X** performance gain in large environment compared to small environment with 10 GB input size in homogeneous and heterogeneous environment, respectively. However, the performance gain of heterogeneous system compared to homogeneous system is not considerable (about **6%**). The reason is that in the clustering phase of K-means, the performance is IO-bound which means that the performance is limited and bounded by IO communication within a cluster. Since K-means is a shuffle-light job i.e. the produced intermediate partition sizes is small, the network traffic overhead in all cases is almost the same and low in both environments. It therefore, indicates that the slightly higher makespan of K-means in homogeneous environment compared to heterogeneous environment is due to its computational degree.

(c) *Sort:* As shown in Fig. 9a, TMaR provides less execution time by increasing the number of resources in the Sort application for all different input sizes in homogeneous environment. The performance gain in homogeneous large environment compared to small environment with 3 GB and 10 GB input size are equal to **+ 1.7X** and **+ 1.5X**, respectively. Fig. 9b shows that the performance in heterogeneous environment is almost the same with all input data size and only when the input data is large (10GB), it reaches to a better performance, **1.2X** and **1.3X** in small and large environment, respectively, compared to homogeneous environment.

(d) *Grep:* Grep application has the minimum run time among other applications, **+ 6.1X**, **+ 35.4X**, and **+2X** faster compared to Wordcount, K-means, and Sort benchmarks in large heterogeneous environment, respectively (Fig. 10b). The reason is that Grep is an IO-intensive job with light shuffling which based on TMaR, makes smaller number of Reduce tasks for processing the produced partitions and consequently less makespan. Furthermore, the performance gain of heterogeneous environment compared to homogeneous environment with 10GB input size in small, medium, and large environment are equal to **+ 1.2X**, **+ 1.3X**, and **+ 1.2X**, respectively.

Part 2: For comprehensive performance analysis of TMaR, we use the benchmarks including, WordCount, Sort, and Grep and consider the makespan by increasing the number of Map tasks from 160 to 200, 400, 900, 1600, 2200, 2400, and 2800, respectively. We run each simulation ten times and report the average value to show the confidence of the results. The deviation of results where the random intermediate data size is generated is negligible for the jobs Grep and K-means. Besides, for the Wordcount and Sort jobs, the deviation is less than 1%. We also compare *TMaR* with $Hadoop - stock$ and $Hadoop - A$, respectively. To simulate Hadoop-A, we set much higher bandwidth between the nodes and implement the Merge sort algorithm in the shuffle phase. Figure 11a, b show the performance comparison between Hadoop-stock, Hadoop-A, and TMaR where the Y-axis shows the execution time and the X-axis indicates the number of Map tasks.

Figure 11a shows the execution time of multiple tasks using Hadoop-stock, Hadoop-A, and TMaR in a heterogeneous environment where the different sizes of Wordcount jobs are applied. To find a correlation between workload size and the execution time, we chose to present a large volume of tasks. As seen, the execution time is reasonably stable with the increase in the number of tasks to be executed by the schedulers, and the completion time of the overall processing is increased. The Hadoop-stock slightly degrades the performance because it is not resource-aware and does not consider the performance of nodes. Hadoop-stock selects Map tasks based on the data locality and in this case, if the head-on-the-line Map task is not local, it would be placed randomly on one of the resources of the cluster and results in worse time. The Hadoop-A and TMaR exhibit better performance (on average 29%) compared to the Hadoop-stock scheduler. As we can see, TMaR achieves a bit better level of performance compared to Hadoop-A. Although Hadoop-A can accelerate the execution time of shuffle-heavy jobs by its fast shuffling, however, it poses extra delay in shuffling for the building of the priority queue. Besides, Hadoop-A does not consider the performance of resources while scheduling the Map tasks, and since the Wordcount is a CPU-intensive job, it cannot benefit from the higher speed resources. In contrast, TMaR schedules Map tasks by considering the performance of resources which considerably reduces the makespan of Map tasks. Also, TMaR schedules Reduce tasks based on the proposed PRB algorithm, which reduces the network data movement and consequently, the shuffle phase. However, this reduction in network movement and data locality in the side of Reducers, make the makespan of Reduce tasks increases.

According to Fig. 11b, Sort benchmark benefits from both TMaR and Hadoop-A. Compared to Hadoop-stock, TMaR achieves on average 36% higher performance. The

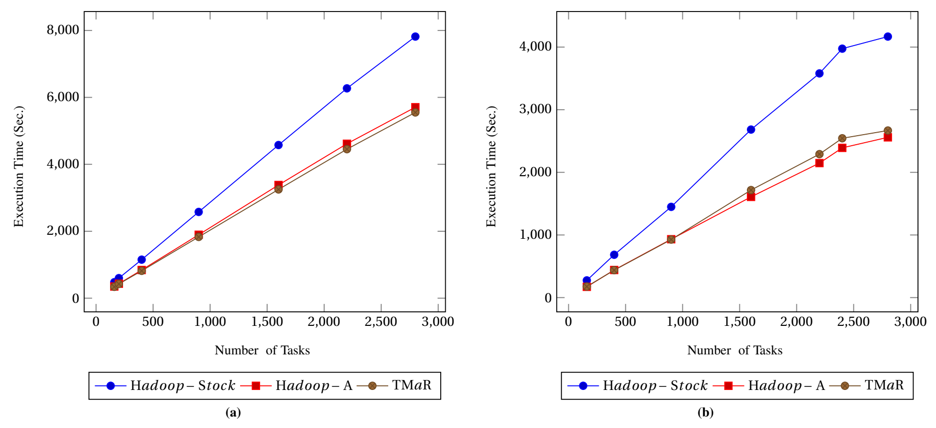


Fig. 11 Execution Time using the **a** Wordcount, **b** Sort

reason is that TMaR considers the placement of Reduce tasks regarding the placement of the Map partitions while in Hadoop-stock, the Reduce tasks are randomly deployed, and the heterogeneity of resources is not taken into account. Compared to Hadoop-A, TMaR achieves less performance (on average 6%) when the number of tasks increases. The reason is that since the Sort job is IO-intensive, it produces an extensive volume data in the middle stage, i.e., network. TMaR can reduce the network traffic by the PRB partition placement but it suffers when the number of required Reducers responsible for processing the partition is large. However, the amount of difference between the two line charts tends to decrease with each step of increasing tasks.

Figure 12a, b shows the detailed performance of each stage in TMaR against Hadoop-stock and Hadoop-A. Hadoop-stock is already good enough at overlapping the communication (shuffle phase) with computation (Map stage) since it follows the slow-start mechanism where shuffling starts when only 5% of Map tasks are completed. However, the shuffle traffic is considerably high due to the significant volume of data that is transferred across the network towards the randomly scheduled Reduce tasks, which consequently results in repetitive merges and disk accesses. After performing the rest of the shuffling (the small grey part which is about 4%), the Reduce phase starts, which takes 30% of the time to complete. In Hadoop-A, several Maps and Reduces are concurrently running on each node to overlap computation and data transfer. The interleaved Map, Shuffle, and Reduce phase forms the major part of the time (90%) by overlaying the Map, shuffle, and Reduce phase using the priority queue and the high-performance network resources. However, despite starting the shuffling along with Map, the performance of Hadoop-A is less than TMaR. This is because of the asynchronous Map and Reduce scheme where TMaR starts the shuffle phase after all Maps are completed. So, here the complete resources are in the disposal to the Map tasks which fastens the Map execution while in Hadoop-stock and Hadoop-A, the Map tasks and Reduce tasks will compete for the resources. In the shuffle phase, TMaR schedules the partitions using the data locality-based partition placement algorithm and mitigates the Map and shuffle stage on average by 27% and 20% compared to Hadoop-stock and Hadoop-A, respectively. The asynchronous Map and Reduce scheme makes a trade-off between improving the data locality along with fair distribution of input data size for Reducers (achieved by PRB algorithm) and

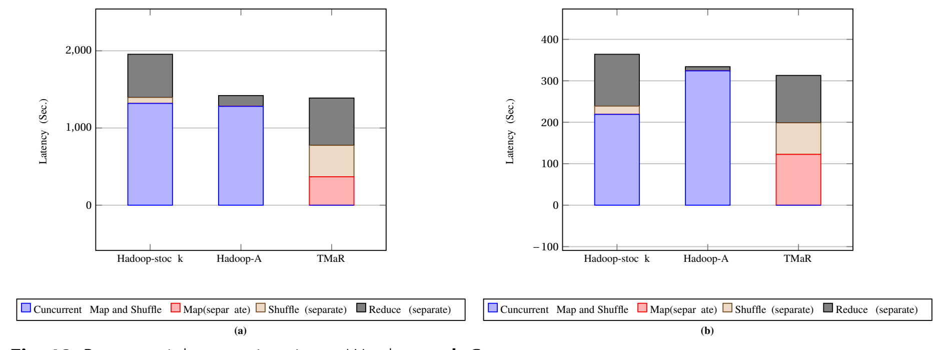


Fig. 12 Per-stage job execution time **a** Wordcount, **b** Grep

concurrent MapReduce, i.e., concurrent execution of Map phase and Reduce phase. These results adequately demonstrate that TMaR is able to efficiently accelerate Wordcount, Sort, and Grep job execution on average 29%, 36%, and 14%, respectively, meanwhile achieves competent scalability for large-scale data processing.

According to [33], we have conducted a complex experiment where four jobs are running with different input sizes in the heterogeneous environment to consider the scheduling behavior of TMaR in presence of multi jobs. The jobs including, one Wordcount job with 20GB, two Sort jobs with 5GB, and one Grep job each with 5 GB input data, respectively. Figure 13a, b, c presents the start time, completion time, and time duration for all scheduled tasks. We can see that TMaR can complete the jobs faster compared to Hadoop-stock and Hadoop-A.

Intermediate data size

Figure 14 plots the results of job completion time of our scheduler and others. For the shuffle-light jobs such as K-means and Grep in which the intermediate data is small, the shuffle delay is negligible. Therefore, to understand the performance of the schedulers under different intermediate data size, we measured the job completion time with intermediate data size ranging from 30 GB to 150 GB using Sort benchmark. Figure 14 shows that the job completion time of all three approaches scales linearly with the intermediate

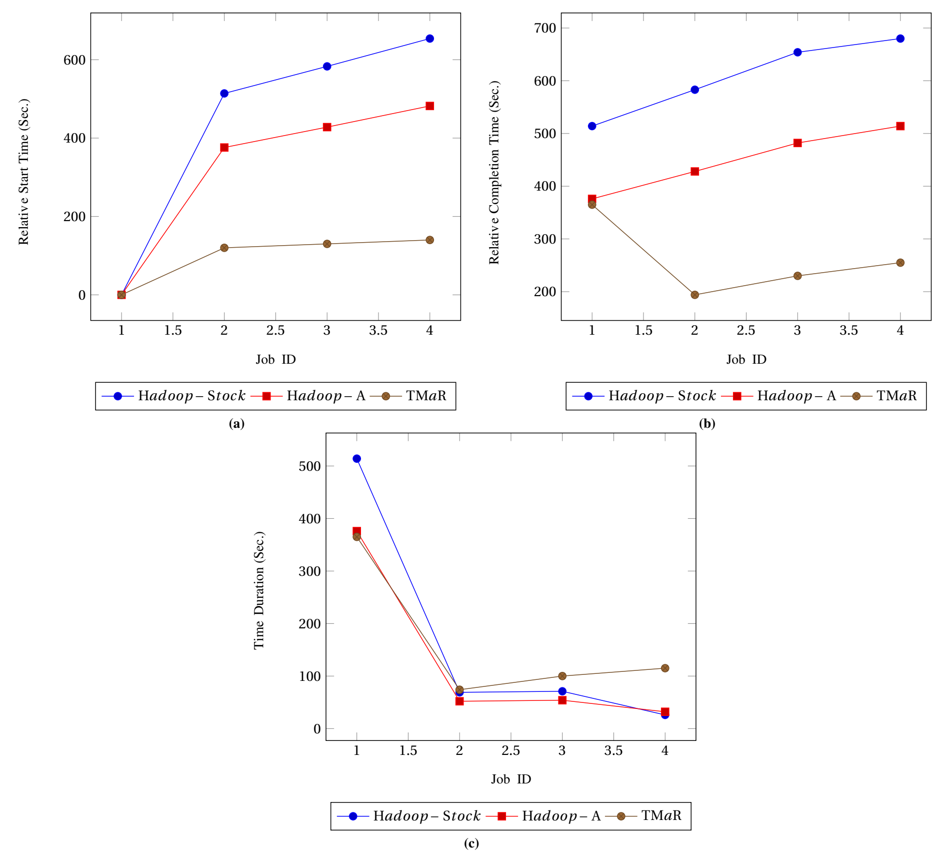


Fig. 13 Runtime for all scheduled tasks **a** Start time, **b** completion time, and **c** time duration

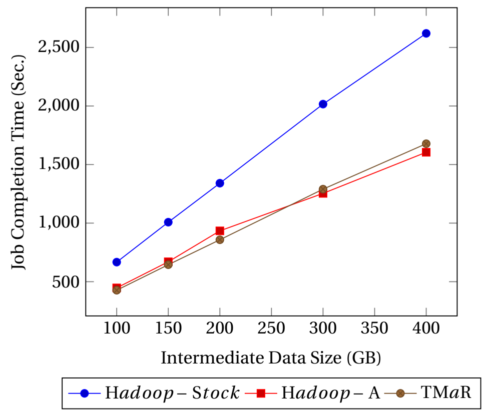


Fig. 14 Impact of intermediate data size on job completion time

Table 5 Comparison between *TMaR* and *TMaR*$^+$ in terms of power consumption

Scenario	*TMaR*/KW	*TMaR*$^+$/KW
Small homo	79.5	79.5
Small hetero	68.35	59.85
Medium homo	88.12	88.12
Medium hetero	85.75	82
Large homo	132.5	132.5
Large hetero	125.5	115

data size. The job completion time of TMaR is consistently (36%) less than Hadoop-stock scheduler, where the Reduce tasks are statically scheduled and result in prolonging the shuffle/network phase. However, if the intermediate data size gets higher than 300GB, the job benefits little more from Hadoop-A compared to TMaR. The time complexity of TMaR is $O(n \times m)$, where n and m represent the number of tasks and resources, respectively.

Power comparison

For measuring the power consumption of cluster, we compare *TMaR* with *TMaR*$^+$ in a small, medium, and large-scale cluster with the homogeneous and heterogeneous resources in terms of processing capacity and power consumption. We consider 10, 25, and 30 resources with the power consumption in range of {350, 300, 250, 150} Watt in a round robin distribution. We generate 5*GB*, 6.5*GB*, and 9.5*GB* input data with 40, 50, and 76 Map tasks, respectively and conduct the experiments with benchmark *Wordcount*.

Table 5 shows the reduction in power consumption by applying *TMaR*$^+$. The results show that *TMaR*$^+$ can improve the power consumption of cluster in all scale of heterogeneous systems. besides, the power consumption in homogeneous environment compared to heterogeneous environment is considerable especially in small scale Hadoop environment. However, since the main objective is makespan minimization, in the homogeneous environment there is any change (improvement) by $TMaR^+$ compared to *TMaR*. The percentage of power improvement in $TMaR^+$ compared to *TMaR* is 12.5%, 5%, and 8% in small, medium, and large heterogeneous cluster, respectively.

Conclusions

In this paper, we presented a two-stage MapReduce task scheduler, named *TMaR* which enhances Hadoop performance in terms of makespan. The primary goal of our scheduler is to reduce the makespan of the overall tasks of MapReduce jobs while considering network traffic in the shuffle phase. By accelerating the Map tasks finish time in Map stage, and the proposed partition placement in shuffling, *TMaR* reduces the Reduce tasks finish time. Since the Reduce tasks are not prescheduled and the number of Reduce tasks is dependent on the size of partitions, this approach mitigates the resource waste. Moreover, in Hadoop, the shuffle time depends on the location of prescheduled Reduce tasks however, in *TMaR*, since the Reduce task-partition binding is dynamically performed at runtime based on the partition placement, the shuffling time is decreased. $TMaR^+$ is an extension of *TMaR* that improves total power consumption of cluster and reduces it up to 12%. *TMaR* is suitable for the dashboard reporting where the independent jobs are specified individually while the final result of all the jobs (tasks makespan) is the key concern. The experimental results demonstrated that *TMaR* improves performance in terms of makespan under different workloads. *TMaR* is power efficient since it selects the resources with lower power consumption while this decision does not contradict with the objective i.e. makespan. *TMaR* is not optimal but it outperforms the Hadoop-stock scheduler and Hadoop-A in terms of makespan and network traffic.

In Hadoop systems, the latency occurs only because of the nature of the MapReduce-based execution, where it produces lots of intermediate data. Thus, much data is exchanged between nodes that cause huge disk IO latency. TMaR has implicitly considered this latency by aggregating partitions that belonged to the Reduce tasks on a specified node, called, Partition-Reducer Binder (PRB). The PRB goal is to reduce the network traffic by preventing the unnecessary data movements between nodes which results in a reduction of disk IO latency. The Apache Spark is yet another batch processing system but it is relatively faster than Hadoop MapReduce since it caches much of the input data on memory by RDD and keeps intermediate data in memory itself and eventually writes the data to disk upon completion. We will implement *TMaR* in Spark, as our future plan, to investigate the disk IO and fault-tolerant factors. Furthermore, to improve the parallelism of Map and Reduce tasks, we intend to estimate earlier the partition sizes in advance by estimating intermediate data using Map selectivity. We also plan to propose a multi-objective optimization model which considers a trade-off between system cost in terms of energy usage and the job completion time.

Acknowledgements
Not applicable.

Authors' contributions
All authors contributed equally to this manuscript. All authors read and approved the final manuscript.

Funding
No funding was received.

Data availability statement
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Author details
[1] Department of Computer Engineering, Science and Research Branch, Islamic Azad University, Tehran, Iran. [2] Department of Computer Science and Communication, KTH Royal Institute of Technology, Stockholm, Sweden. [3] Institute of Research and Development, Duy Tan University, Da Nang 550000, Vietnam. [4] Department of Mathematics, University of Padua, Padua, Italy. [5] Sandia National Laboratories, Albuquerque, NM, USA. [6] Faculty of Information Technology, Duy Tan University, Da Nang 550000, Vietnam. [7] Department of Computer Science, Khazar University, Baku, Azerbaijan.

Received: 27 June 2020 Accepted: 15 September 2020
Published online: 07 October 2020

References

1. Reinsel D, Gantz J, Rydning J (2017) Data age 2025—the evolution of data to life-critical: do not focus on Big Data; focus on the data that is big. IDC White Pap., no. April
2. Irandoost MA, Rahmani AM (2019) Learning automata-based algorithms for MapReduce data skewness handling. J Supercomput 78:6488–6516
3. Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Commun ACM 51(1):107–113
4. Hadoop A (2011) Apache hadoop. http://hadoop.apache.org
5. Wang K, Zhou Q, Guo S, Luo J (2018) Cluster frameworks for efficient scheduling and resource allocation in data center networks: a survey. IEEE Commun Surv Tutor 20(4):3560–3580
6. Al-Fares M, Radhakrishnan S, Raghavan B, Huang N, Vahdat A *et al.* (2010) Hedera: dynamic flow scheduling for data center networks. In: Nsdi, vol. 10
7. Guo Y, Rao J, Cheng D, Zhou X (2016) ishuffle: Improving Hadoop performance with shuffle-on-write. IEEE Trans Parallel Distrib Syst 28(6):1649–1662
8. Pandey V, Saini P (2018) How heterogeneity affects the design of Hadoop MapReduce schedulers: a state-of-the-art survey and challenges. Big Data 6(2):72–95
9. Ghodsi A, Zaharia M, Hindman B, Konwinski A, Shenker S, Stoica I (2011) Dominant resource fairness: fair allocation of multiple resource types. NSDI 11:24–24
10. Grandl R, Chowdhury M, Akella A, Ananthanarayanan G (2016) Altruistic scheduling in multi-resource clusters. In: 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16), pp 65–80
11. Bodík P, Menache I, Chowdhury M, Mani P, Maltz DA, Stoica I (2012) Surviving failures in bandwidth-constrained datacenters. In: Proceedings of the ACM SIGCOMM 2012 conference on applications, technologies, architectures, and protocols for computer communication. ACM, New York, pp 431–442
12. Gao PX, Narayan A, Karandikar S, Carreira J, Han S, Agarwal R, Ratnasamy S, Shenker S (2016) Network requirements for resource disaggregation. In: 12th {USENIX} symposium on operating systems design and implementation ({ OSDI} 16), pp 249–264
13. White T (2012) Hadoop: the definitive guide. O'Reilly Media, Inc., Sebastopol
14. Zaharia M, Borthakur D, Sen Sarma J, Elmeleegy K, Shenker S, Stoica I (2010) Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In: Proceedings of the 5th European conference on computer systems. ACM, New York, pp 265–278
15. Yu W, Wang Y, Que X (2013) Design and evaluation of network-levitated merge for Hadoop acceleration. IEEE Trans Parallel Distrib Syst 25(3):602–611
16. Ahmad F, Lee S, Thottethodi M, Vijaykumar T (2013) MapReduce with communication overlap (maRCO). J Parallel Distrib Compu 73(5):608–620
17. Lin M, Zhang L, Wierman A, Tan J (2013) Joint optimization of overlapping phases in MapReduce. Perform Eval 70(10):720–735
18. Verma A, Cherkasova L, Campbell RH (2013) Orchestrating an ensemble of MapReduce jobs for minimizing their makespan. IEEE Trans Depend Secure Comput 10(5):314–327
19. Zhu Y, Jiang Y, Wu W, Ding L, Teredesai A, Li D, Lee W (2014) Minimizing makespan and total completion time in MapReduce-like systems. In: IEEE INFOCOM 2014-IEEE conference on computer communications. IEEE, New York, pp 2166–2174
20. Jiang Y, Zhou P, Cheng T, Ji M (2019) Optimal online algorithms for MapReduce scheduling on two uniform machines. Optim Lett 37:1663–1676
21. Tian W, Li G, Yang W, Buyya R (2016) Hscheduler: an optimal approach to minimize the makespan of multiple MapReduce jobs. J Supercomput 72(6):2376–2393
22. Jiang Y, Zhu Y, Wu W, Li D (2017) Makespan minimization for MapReduce systems with different servers. Future Gener Comput Syst 67:13–21
23. Hashem IAT, Anuar NB, Marjani M, Gani A, Sangaiah AK, Sakariyah AK (2018) Multi-objective scheduling of MapReduce jobs in big data processing. Multimedia Tools Appl 77(8):9979–9994
24. Braam PJ, Zahir R (2002) Lustre: A scalable, high performance file system. Cluster File Systems, Inc
25. Amazon E (2015) Amazon web services. http://aws.amazon.com/es/ec2/(2012)
26. Selvitopi O, Demirci GV, Turk A, Aykanat C (2019) Locality-aware and load-balanced static task scheduling for MapReduce. Future Gener Comput Syst 90:49–61

27. Yao Y, Gao H, Wang J, Sheng B, Mi N (2019) New scheduling algorithms for improving performance and resource utilization in Hadoop yarn clusters. IEEE Trans Cloud Comput
28. Wang W, Zhu K, Ying L, Tan J, Zhang L (2016) Maptask scheduling in MapReduce with data locality: throughput and heavy-traffic optimality. IEEE/ACM Trans Network (TON) 24(1):190–203
29. Jeyaraj R, Ananthanarayana V, Paul A (2019) MapReduce scheduler to minimize the size of intermediate data in shuffle phase. In: 2019 IEEE/ACIS 18th international conference on computer and information science (ICIS). IEEE, New York, pp 30–34
30. Maleki N, Rahmani AM, Conti M (2019) MapReduce: an infrastructure review and research insight. J Supercomput 75:6934–7002
31. Mustafa S, Sattar K, Shuja J, Sarwar S, Maqsood T, Madani SA, Guizani S (2019) Sla-aware best fit decreasing techniques for workload consolidation in clouds. IEEE Access 7:135256–135267
32. Liaqat M, Naveed A, Ali RL, Shuja J, Ko K-M (2019) Characterizing dynamic load balancing in cloud environments using virtual machine deployment models. IEEE Access 7:145767–145776
33. Nita M-C, Pop F, Voicu C, Dobre C, Xhafa F (2015) Momth: multi-objective scheduling algorithm of many tasks in Hadoop. Cluster Comput 18(3):1011–1024
34. Kalra M, Singh S (2015) A review of metaheuristic scheduling techniques in cloud computing. Egypt Inf J 16(3):275–295
35. Rao S, Ramakrishnan R, Silberstein A, Ovsiannikov M, Reeves D (2012) Sailfish: a framework for large scale data processing. In: Proceedings of the third ACM symposium on cloud computing. ACM, New York, p 4
36. Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, *et al.* (2013) Apache Hadoop yarn: Yet another resource negotiator. Proceedings of the 4th annual symposium on cloud computing. ACM, New York, p 5
37. Condie T, Conway N, Alvaro P, Hellerstein JM, Elmeleegy K, Sears R (2010) MapReduce online. NDSI 10:20
38. Maleki N, Loni M, Daneshtalab M, Conti M, Fotouhi H (2019) Sofa: A spark-oriented fog architecture. In: IECON 2019-45th annual conference of the IEEE industrial electronics Society, vol. 1, IEEE, New York, pp 2792–2799
39. Herodotou H, Babu S (2011) Profiling, what-if analysis, and cost-based optimization of MapReduce programs. Proce VLDB Endow 4(11):1111–1122
40. Topcuoglu H, Hariri S, Wu M-y (2002) Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans Parallel Distrib Syst 13(3):260–274
41. Calheiros RN, Ranjan R, Beloglazov A, De Rose CA, Buyya R (2011) Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw Pract Exp 41(1):23–50
42. Kao Y-C, Chen Y-S (2016) Data-locality-aware MapReduce real-time scheduling framework. J Syst Softw 112:65–77
43. Cai X, Li F, Li P, Ju L, Jia Z (2017) Sla-aware energy-efficient scheduling scheme for Hadoop yarn. J Supercomput 73(8):3526–3546
44. Kathiravelu P, Veiga L (2014) An adaptive distributed simulator for cloud and MapReduce algorithms and architectures. In: 2014 IEEE/ACM 7th international conference on utility and cloud computing. IEEE, New York, pp 79–88
45. Alrokayan M, Dastjerdi AV, Buyya R (2014) Sla-aware provisioning and scheduling of cloud resources for big data analytics. In: 2014 IEEE international conference on cloud computing in emerging markets (CCEM). IEEE, New York, pp 1–8
46. Jung J, Kim H (2012) Mr-cloudsim: Designing and implementing MapReduce computing model on cloudsim. In: 2012 international conference on ICT convergence (ICTC). IEEE, New York, pp 504–509

Publisher's Note