Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2021) 29: 437 – 453
© TÜBİTAK
doi:10.3906/elk-2001-106

A counter based approach for reducer placement with augmented Hadoop rack awareness

Wajahat Hussain MIR[1], **K Hemant Kumar REDDY**[2], **Diptendu SINHA ROY**[1,*]
[1]Department of Computer Science and Engineering, Faculty of Engineering,
National Institute of Technology Meghalaya, Shillong, India
[2]Department of Computer Science and Engineering, Faculty of Engineering,
National Institute of Science and Technology, Berhampur, India

Received: 23.01.2020 • **Accepted/Published Online:** 23.06.2020 • **Final Version:** 27.01.2021

Abstract: As the data-driven paradigm for intelligent systems design is gaining prominence, performance requirements have become very stringent, leading to numerous fine-tuned versions of Hadoop and its MapReduce programming model. However, very few researchers have investigated the effect of intelligent reducer placement on Hadoop's performance. This paper delves into this much ignored reducer placement phase for improving Hadoop's performance and proposes to spawn reduce phase of Hadoop tasks in an asynchronous fashion across nodes in a Hadoop cluster. The main contributions of this paper are: (i) to track when map phase of tasks are completed, (ii) to count the number of maps completed, and finally (iii) assign reducers to Hadoop nodes based on map counts such that run-time data copying is minimized. To this end, this paper presents a novel counter based reducer placement (CBRP) algorithm based on the counter values maintained by JobTracker at the rack and node levels. Experiments conducted demonstrate the merit of the proposed reducer placement with average improvements ranging between 5% and 17% experienced across different benchmarks with both late shuffle and early shuffle.

Key words: MapReduce, HDFS, Hadoop rack awareness, reducer placement

1. Introduction

Unprecedented rise of data used has led to the problem of designing solutions that hitherto databases are incapable of carrying out [1]. Hadoop has emerged as a savior with its scalable and fault-tolerant design and a parallel and distributed computing engine that works in tandem. Hadoop incorporates file storage across a number of machines with replication and it chops a file into a number of blocks, distributing them uniformly across the cluster. MapReduce has been one of the most successful and popular computing engines [2, 3]. A MapReduce program primarily undergoes two functions in course of its execution, namely, map and reduce, each of them having further subphases. Before processing a map task, the data needs to be fetched by a node whether from local or off local node and is called as data read. All the worker nodes in a cluster generally execute the map function simultaneously. During the processing of map phase, intermediate data are stored in buffers with predefined thresholds. A spill is generated once the intermediate data corresponding to the map function exceeds its threshold. A partitioner ensures that multiple copies for every spill are maintained within storage [4]. These copies are gainfully employed for an internal sorting called combiner [4, 5]. Thereafter reduce phase starts with copying all data produced by map function in a copy/shuffle phase. After the culmination of

*Correspondence: khemant.reddy@gmail.com copy/shuffle phase, the data from several maps are sorted. Finally, reducer has to output the desired result. Figure 1 depicts the various phases of a MapReduce operation. The bold arrows represent the interphase data transfers with various actions in different subphases within a single node, incoming dotted arrows represent the computation done by the other nodes in the cluster and outgoing dotted arrows represent the data movement from this node to other nodes in the cluster.

Default Hadoop places data uniformly inside the cluster assuming that all nodes are homogeneous [6]. However, in most Hadoop clusters, nodes are not homogeneous, rather there exist drastic differences among nodes within a cluster as far as their capabilities are concerned [7]. Distributing input data uniformly across such a set of disparate nodes might not produce the intermediate data uniformly across all the nodes. This map output is an intermediate data and hence it has to be passed to a reducer for the final result. The slaves, called TaskTrackers, request to their master, the JobTracker for a reduce task and if the slave has enough resources for the same, then the JobTracker will schedule a reduce task to it; otherwise it will discard the request [8, 9]. Hadoop schedules reduce tasks randomly, ignoring all information related to the node and its computational workload. This may lead to a potential network bottleneck, since the reduce phase may require a lot of shuffled data from the mappers, which may be executed across different nodes in various racks. This may incur severe performance penalty due to run time data transfers [10]. It is obvious in a heterogeneous scenario that some nodes may complete more maps than others; the extent of difference varying depending upon the inherent heterogeneity of the Hadoop cluster.

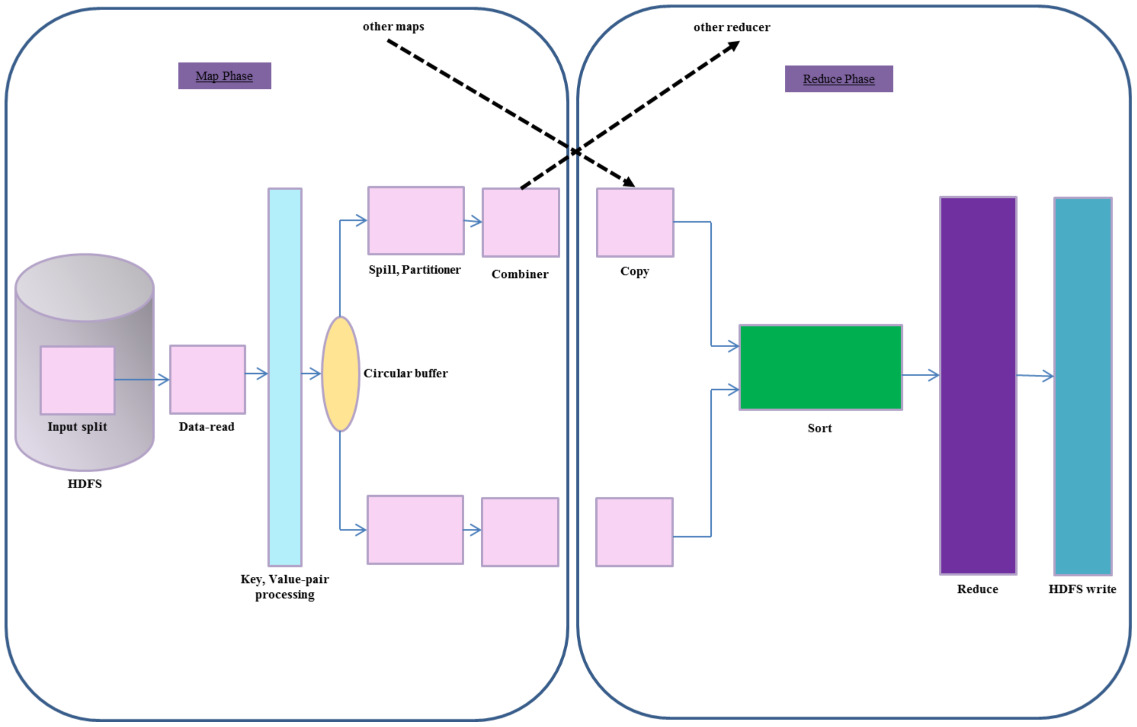


Figure 1. Subphases of map and reduce in a MapReduce application (Map phase is shown for a single data block and the reduce phase takes inputs from other completed map tasks within the cluster).

Majority of research attempts towards Hadoop's performance improvements have focused on optimizing performance of the map phase through effective data placements preceding map operations [9, 11–14], primarily by means of some mechanism of grouping related data block and colocating them within the same cluster node. However, for many applications the amount of intermediate data after a map phase is huge [10]. Trivial implementation of copy | shuffle phase followed by arbitrary reducer decisioning in terms of number of reducers a selecting nodes as reducers are often found inadequate. Some work has been done towards enumerating the number of reducers for a given problem, however proper reducer placement can greatly alleviate real-time data copying across cluster nodes and thus can improve Hadoop's performance.

This paper addresses the problem of reducer placement, which has been relatively less explored. Our proposed methodology attempts to place reducers by keeping track of locality through a couple of counters maintained by JobTracker, namely a rack counter (RC) for every rack in the cluster and a map counter (MC) in every slave node; and thus making informed decisions about reducer placement. Such an arrangement holds the promise of minimizing internode traffic at runtime. Additionally, we have tuned several attributes in Hadoop, such as the number of reducers in the job, number of threads required for copying the map output data. Results presented herein demonstrate that the proposed counter based reducer placement (CBRP) improves the performance from 5% to 17% depending on the nature of job.

The rest of paper organization is described as follows: Section 2 presents an overview of related research works. Section 3 presents our proposed algorithm. Illustrative example to convey the idea along with a mathematical analysis has been presented in Section 4. Section 5 demonstrates the implementations along with experimental results with a thorough evaluation of the results. Finally, the conclusions have been drawn on the results in Section 6.

2. An overview of related research

Hadoop has a Java based file system called Hadoop distributed file system (HDFS) which stores data across a cluster into a number of machines [14, 15]. It follows a policy of write once and read many times and has a master slave architecture [15]. The master is responsible for maintaining the metadata and overall file system [16]. The slaves hold the actual data.

The computation in Hadoop is handled by master and slave daemon called JobTracker and TaskTracker. The JobTracker is responsible for overall life cycle management of a Hadoop job. The TaskTracker does the actual computation. Scheduling of tasks in Hadoop is enticed by a pull strategy rather than a push strategy [8]. The slaves send request for the task scheduling and reducer placement to the JobTracker. It is JobTracker who schedules the tasks to a specified TaskTracker.

As mentioned earlier, there has been a variety of attempts to improve Hadoop's performance by map phase implications. However, there has been very few works that have delved into reduce phase implications and tuning. One of the most fundamental works on reduce phase performance challenge has been studied in [17] wherein the memory challenge in bringing all intermediate data to memory is addressed that may lead to application failure. A method for mitigating such challenges by applications profiling based deduction of optimal number of reducers is also studied. However, the work has not addressed the reducer placement problem which we address in this paper. In that sense, our contribution compliments the finding in [17].

Hammond et al. [8] considered data locality for placing reduce task to that node which holds the largest amount of intermediate data after the map phase. Later the authors improved their algorithms to consider data skew in addition to locality for reducer placement by means of a center of gravity based node and program measures [18]. The work in [17] addressed means for avoiding multiple reducers placed in a node, by means of choosing non preferred TaskTrackers with vicinity. However, [19] completely ignore rack locality in their reducer placement treatment. Additionally, copy | shuffle being the slowest phase, waiting for heartbeat messages, particularly for nonpreferred TaskTrackers becomes another overhead. Li-Yung Ho et al. [19] proposed an algorithm to improve Hadoop's performance through minimizing cross-rack traffic by placing reducers on the basis of dissemination of mappers. The location of reducers was placed in a way that leads to overall minimization of the cross-rack traffic after considering both upward and downward traffic during shuffle. However, the basis of cross-rack traffic minimization is flawed on two grounds. Firstly, a map task (for which map progress unacceptable) may fail and then be executed speculatively with a new set of mappers to new nodes. Secondly, [19] has considered 100% data locality in a map phase which is not pragmatic. Arslan et al. [20] devised a locality and network aware reduce task scheduling for the placement of reducers based on the intermediate data size, bandwidth of link and hop counts required for movement of data from source to reducer. The set of TaskTrackers were partitioned into groups based on a novel cost function enumerated and task is chosen. However, cost function computations can introduce lot of delay in placing reducers because every TaskTracker needs to send heartbeat for reduce task and it will lead to a significant overhead. Shen et al. [21], focused on the task placement (map and reduce) to minimize the completion time of the job. Their idea was to assign tasks based on transmission cost based probabilistic scheduling method, which takes data size and network condition into account while assigning computing slot to a task. However, optimizing map placement would significantly elevate the data transfer phase [22].

Zhao et al. [22] proposed a coflow (parallel flows between two stages like map and reduce) scheduling algorithm using shortest remaining time first and online reducer placement to minimize the average coflow computation time in cloud clusters. The issue with the work is that placement of reducers just on the basis of minimizing completion time might lead to congestion as if most reducers are placed within the vicinity of computed out-put data, thus exacerbating performance [23]. Guo et al. [24] focused on minimizing the coflow completion time (CCT) of applications in interdata center wide area network by optimizing both task placement and coflow routing. The author extended their work in [25], by formulating an optimization problem, proposing an algorithm, describing the theoretical analysis and evaluating the performance by considering both task placement and coflow routing to reduce the CCT. Both these works [24, 25], only focus on the inter DC optimization and not intra DC.

The counter based reducer placement is promising since it accounts for Hadoop's inherent rack locality, addresses multiple reducer placements but avoids long running cost function evaluating bottlenecks. In this paper, besides accounting for counters to tracking number of reducers does not view any overhead. The following section presents a detailed description of the proposed CBRP scheme.

3. Augmenting rack awareness with a counter based reducer placement

Racks in a Hadoop cluster maintain each other's topology related information through its default rack awareness policy. Rack awareness primarily provides increased availability of data blocks and also improve cluster performance by leveraging topology information for collocating map tasks with data blocks required. As mentioned earlier, this paper presents a novel reducer placement scheme by counters maintained by JobTracker for different levels of granularity because job and task level monitoring is done by it.

The Algorithm 1 puts a priority on rack locality over data locality. To track rack locality and data locality we have counters for rack level and node level. These counters count the number of map tasks completed within the rack or a node. When a specified number of maps (early/late shuffle) in a job are completed, the algorithm decides to place a reducer based on the value of rack counter. There are two ways that we know when to start

Algorithm 1: Counter based reducer placement algorithm.

```
public static myreducer(...)
redCount=valueOf ("$MAX\_REDUCER$") ;
Collection <TaskAttemptID >tids = newTaskReport () ;
getRunningTaskAttempts ();
for TaskTid:tids do
    TaskCompletionEvent TidEvnt = newTaskCompletionEvent ($Tid$);
    if TidEvent.is MapTask() && TidEvent.getStatus().equals("SUCEEDED") then
        string resourceURL=TidEventTracket Http ();
        List <String >RackLoc = newCachedDNSToSwitchMapping ().resolve ($resourceURL$);
        String NodeInfo = newTaskAttemptContext ().getNodeID ();
        for ListIteration <String >RackInfo = RackLoc.listIterator() to RackInfo.hasNext () do
            MAP_COUNTER:
            for counter1 = 1 to NumRack do
                if RackInfo.next().equals(RackLabel[counter1]) then
                    Increement RackCounter[counter1]
                    for counter2 = 1 to RackNode[counter1].size do
                        if NodeInfo.equals(NodeLabel[counter1][counter2]) then
                            Increement NodeCounter[counter1]
                        end
                    end
                end
            end
        end
        set alpha =0,sigma=0
        RACK_PRIORITY;
        Sort (RackCounter, RackLabel : Desc)
        Sort (NodeCounter, NodeLabel : Desc)
        GrantReducerRequest (NodeLabel[alpha][sigma])
        if (RackCounter[alpha] − RackCounter[beta]) >beta then
            GrantReducerRequest (NodeLabel[alpha][Increement(sigma)]);
            else
            GrantReducerRequest (NodeLabel[Increement(alpha)][sigma]);
            end
        end
    end
end
```

a reducer. In the obvious case a reducer is started only upon completion of map phase across all tasks. An alternative scheme is often employed where reducers start as soon as a predefined percentage of map tasks are completed. The former scheme is known as late shuffle, whereas the latter scheme is called early shuffle and is preferred for performance reasons. The rack counter values are sorted in a decreasing order and thereafter the rack with highest count will be at the first index position of the sorted list. For this rack, we further sort all constituent nodes based on their map counter values in decreasing order. Thus, we arrive at the node with maximum map counts within rack having highest mapper counters. The way to collect the counter is represented with the label MAP_COUNTER in the algorithm. As a corollary, we can conclude that such a node will have maximum intermediate data after the map phase and hence we place the first reducer there. This is confirmed upon receiving a heartbeat message. The same process continues till all the reducers are placed. However, to avoid a situation where multiple reducers are placed within a rack, a static difference threshold is maintained; beyond which a new reducer is placed in a different rack, despite having a node priority as per the sorted index. This static threshold has been depicted Beta and the calculation of beta is shown with the label RACK_PRIORITY in the algorithm.

The above algorithm is having a polynomial time complexity of the order O (R*M + N*M), where R is number of racks, M being number of nodes in the rack and N being the number of reducers to be placed inside the cluster. Figure 2 depicts the end-to-end control mechanism for the proposed CBRP as a sequence diagram where reducers are placed according to significant computation done. When TaskTracker sends requests to the JobTracker for reduce, it checks whether the computation done by the node and the rack is highest; it grants the request otherwise it defers the request. Thus, the above algorithm places reducer based on improving rack locality and data locality.

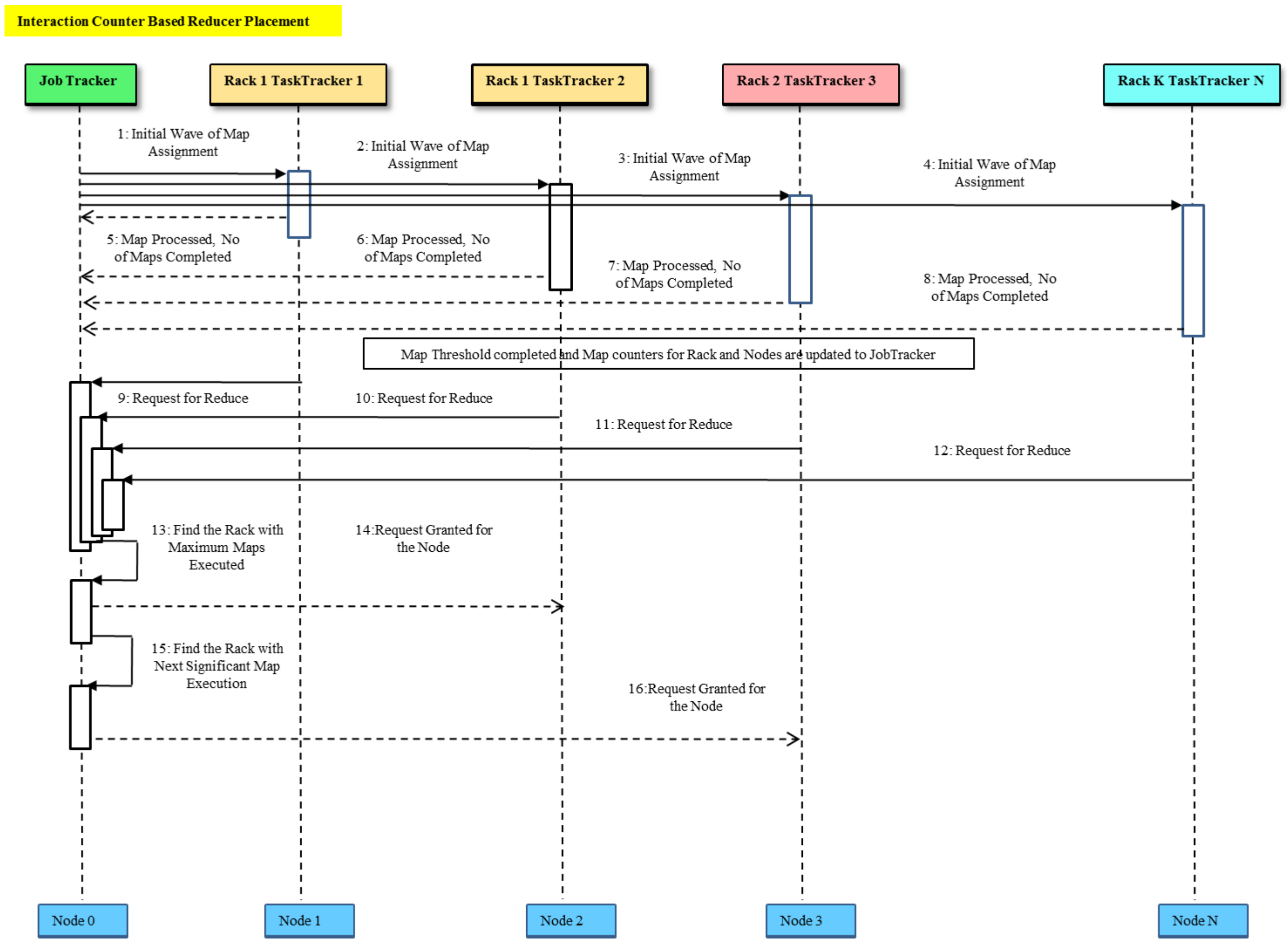

Interaction Counter Based Reducer Placement
Job Tracker
Rack 1 TaskTracker 1
Rack 1 TaskTracker 2
Rack 2 TaskTracker 3
Rack K TaskTracker N
1: Initial Wave of Map Assignment
2: Initial Wave of Map Assignment
3: Initial Wave of Map Assignment
4: Initial Wave of Map Assignment
5: Map Processed, No of Maps Completed
6: Map Processed, No of Maps Completed
7: Map Processed, No of Maps Completed
8: Map Processed, No of Maps Completed
Map Threshold completed and Map counters for Rack and Nodes are updated to JobTracker
9: Request for Reduce
10: Request for Reduce
11: Request for Reduce
12: Request for Reduce
13: Find the Rack with Maximum Maps Executed
14:Request Granted for the Node
15: Find the Rack with Next Significant Map Execution
16:Request Granted for the Node
Node 0
Node 1
Node 2
Node 3
Node N

Figure 2. Sequence diagram for the proposed counter based reducer placement (The TaskTracker sends notifications to the JobTracker for spawning reduce phase after completion of map threshold and the latter grants the same based on the maximum completed map tasks by the former).

4. Illustrative example

Let us consider a scenario of a Hadoop cluster having two racks each containing three machines. At the core switch we have a master node which is implicit. The master node will take care of the metadata storage and the life cycle management of the job i.e. NameNode, Secondary NameNode and JobTracker. The file F is composed of 15 chopped pieces. We assume that the default replication is 3. When the client communicates with the NameNode for placing the data inside the cluster it places it uniformly across the cluster as shown in the Figure 3. Some of the nodes will be receiving either 7 or 8 blocks. The overall picture of the completion of job is that inside rack 1, 6 tasks are completed and inside rack 2, 9 tasks are completed as shown in Figure 3. Moreover, we also can note that DataNode 4 perform the majority computation followed by DataNode 6.

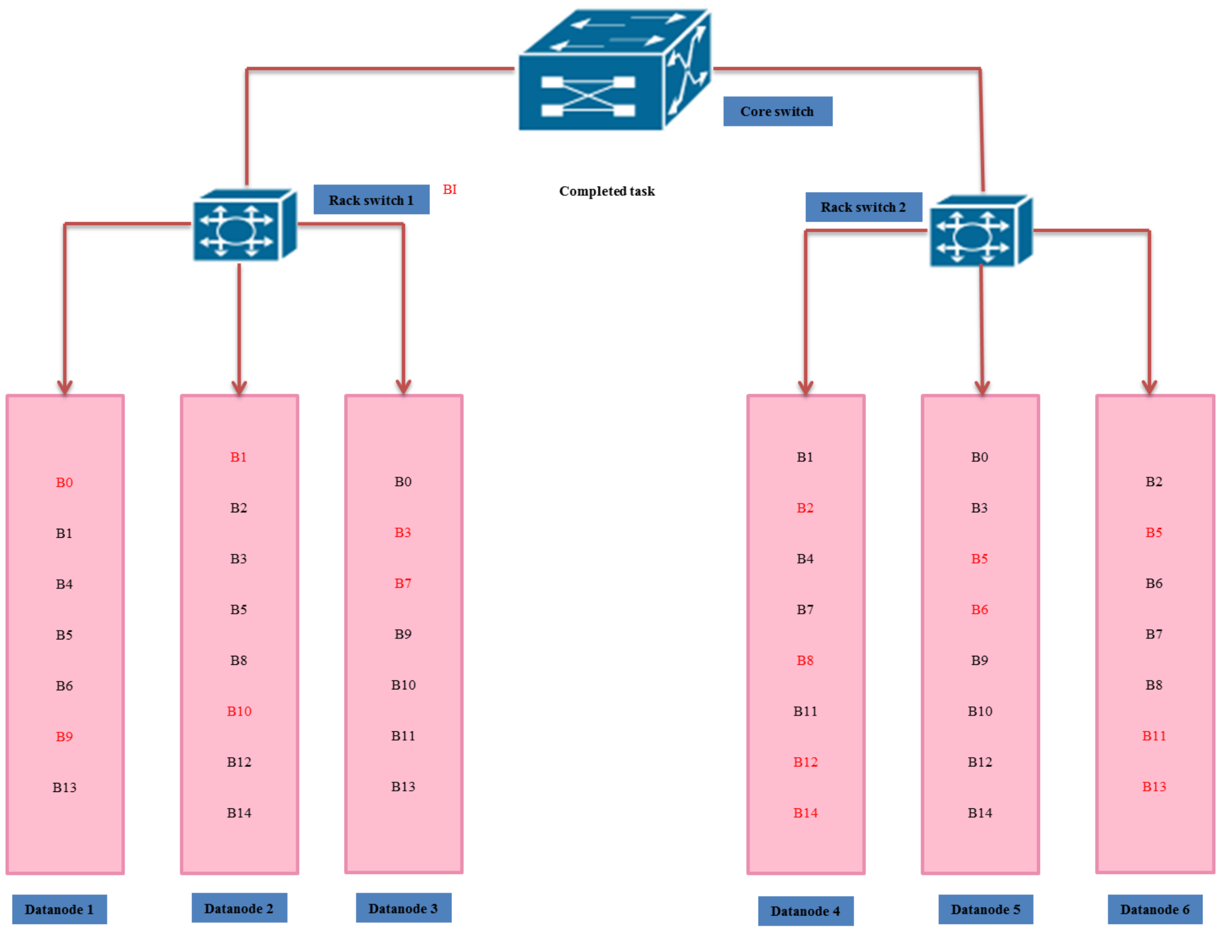


Figure 3. Task and job completion in a Hadoop multinode cluster (Blocks marked as red denote completed tasks in the DataNode).

4.1. Reducer placement cases

This section presents with an example for reducer placement considering both Hadoop locality inherent oblivious and CBRP technique. Hadoop schedules a reduce tasks based on a pull strategy i.e. the request is put forth by slaves through heartbeat. If the slaves have enough resources, then the JobTracker grants that request otherwise it defers request [2, 4]. In this scenario, we consider that the number of reducer used is based on the number of racks. Moreover, we assume that 1 reducer will be scheduled at a node. Thus in this context after culmination of map tasks, all TaskTracker are free and if a node places request for reduce then the request for reduce will be granted for any node. The scheduling of reduce tasks is granted haphazardly in locality inherent oblivious reducer placement and thus the request is placed by nodes 2 and 3. The request for reduce is acquired by these nodes and hence reduce is scheduled at these nodes. These nodes need to fetch an intermediate output data from all the nodes, who are involved in the computation. Thus we need a total of 8 rack local tasks and 18 off local tasks to shuffle as per Figure 4 in case of native Hadoop. This reducer placement is shown in the Figure 4, where the reducers placed on the nodes 2 and 3 describe the Hadoop's locality inherent oblivious placement.

Our CBRP is governed by placing the first reducer onto the rack having highest computation, which is 9 as shown in the Figure 3. Besides, this rack we sort the map counter values of the nodes inside rack, we can infer that node 4 does majority computation. Thus, the first reducer is placed inside rack 2 in node 4. Let us assume that the value of static threshold (beta) be 2 in this case. Now, the next reducer to be placed will be decided on the basis of difference in computation done by rack counters 2 and 1. In this case it evaluates to 3, more than static threshold value. Thus, the next reducer is placed under rack 2 inside the next highest computed node which is node 6 as per Figure 4. Thus, we need 9 rack local tasks and 12 off local tasks to shuffle as per Figure 4 in case of our CBRP approach. Clearly, this explains the merit of our placement by minimizing off local as well as improving rack local shuffle. This reducer placement is shown in the Figure 4, depict our intelligent placement of reducer.

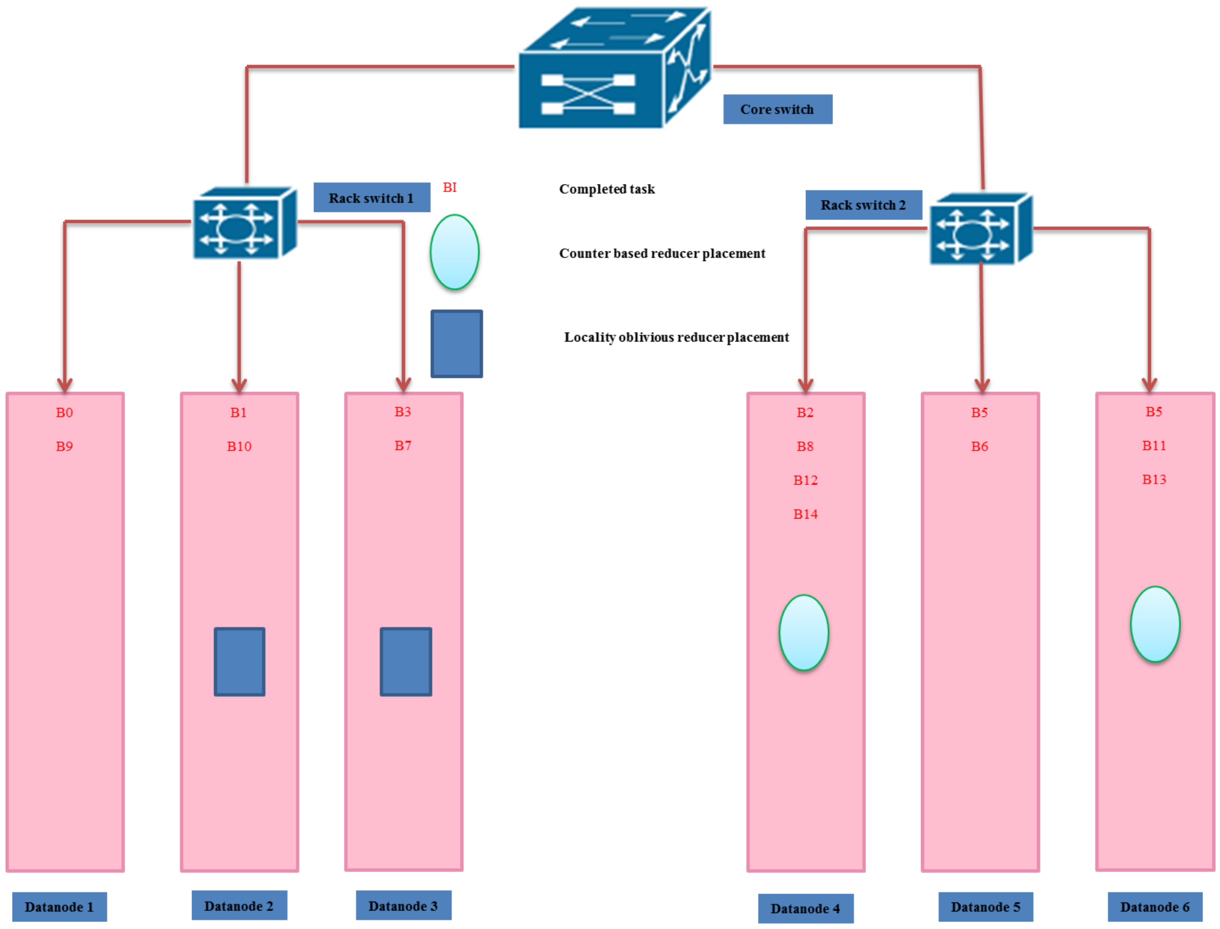


Figure 4. Reducer placement in default Hadoop and counter based placement scheme (Default Hadoop places reducers in a locality oblivious manner, thus the depiction is arbitrary).

4.2. Late shuffle

To get the final picture we have to start the reduce task after completion of all map tasks. This can be done by changing attributes inside the Hadoop. When all the map tasks get completed we find rack counter 2 has more value than rack counter 1. If the reducer is placed inside the rack 1 we have to shuffle the intermediate map output data of 9 map tasks in the rack 1. Since network bandwidth being the scarcest resource so it will take a hefty amount of time to copy the data from rack 2 to rack 1. Now, if the reducer is placed inside the rack 2 we have to shuffle only 6 off local map tasks which will take pretty meager time. Thus it will be pretty beneficial to place reducers inside the rack 2 and improve performance in a prolific way.

4.3. Early shuffle

The above way of starting map tasks after finishing of all maps leads to system slot wastage and increasing job completion time because we have to wait for the last map task until it finishes [26]. If the last map task is slow then it would impact the job performance in a profound way. To minimize overall job completion time, we should start the reduce task during the map phase to overlap both the phases for efficient utilization of resources [27]. Since the copy phase is the first phase of reduce and must be done at an apt time during map phase parallel [28]. Hadoop by default starts reduce task when 5% of map tasks are completed but this is not efficient and leads to increase of map completion time and unwanted wastage of resources [29]. So this attribute must be tuned so that we get an intuitive idea where most of the map tasks will be going to completed. This improvisation is in perfect agreement so as to overlap map with reduce and hence diminishing the completion time. When the tuned attribute value is reached at a particular threshold, we can get the idea whereabouts more maps will be completed [8]. Thus we can place the reducer at an appropriate rack and appropriate node to lower the copying from map to reduce and eventually decrease the load on the network. We have varied the starting time of the reduce task so as to get the optimum value when starting reduce performance will be best.

4.4. A mathematical analysis

Consider a Hadoop cluster with the given specifications as described in Table 1. Let us assume that the rack R_z performs the majority of computation. Further, let the difference of the computation performed by the rack R_z with every rack comes below a particular threshold. Moreover, inside every rack suppose that the p_{i1} node of every rack has the highest computation. Therefore, our work will place reducers across every rack inside every p_{i1} node.

Table 1. Symbol table used in the paper.

Variables	Meaning
k	Number of racks used in the cluster
n	Number of nodes used inside every rack
P_i	Computation done by the rack ‘i’ in the cluster
p_{ij}	Computation done by the machine ‘j’ inside the rack ‘i’
T	Total number of tasks executed inside the cluster
X	Cost of shuffling rack local data inside the machine for placing reducer
Y	Cost of shuffling off local data inside the machine for placing reducer

Total computation done in the cluster is given by

$$= \sum_{i=1}^{k} \sum_{j=1}^{n} p_{ij} \tag{1}$$

Cost incurred in shuffle will only be required for rack local and off local data. Our work CBRP places the first reducer across the first node inside the rack z.

Therefore, the cost of shuffling data for placing reducer inside first machine in a rack R_z is given by

$$= (P_z - p_{z1})\, X + (T - P_z)\, Y \tag{2}$$

Similarly, the cost of shuffling data for placing reducers across all the first nodes of every rack is given by

$$Z_1 = [(P_1 - p_{11}) + (P_2 - p_{22}) + ... + (P_k - p_{k1})]\, X + [(T - P_1) + (T - P_2) + ... + (T - P_k)]\, Y \tag{3}$$

$$Z_1 = \left[T - \sum_{i=1}^{k} p_{i1}\right] X + [KT - T]\, Y \tag{4}$$

After completion of all map tasks, the TaskTracker will be free and it will send the heartbeat for the request of reduce task. Since the scheduling of reduce tasks in Hadoop is done randomly, we can assume that m number of reducers are correctly placed and the rest (k − m = r) reducers get worst placement. The cost of Hadoop's locality oblivious placement inherent for placing reducers across the machines in the cluster is given by

$$\begin{aligned} Z_2 \quad = \quad & \big[(P_1 - p_{11}) + ... + (P_m - p_{m1}) + \big(P_{(m+1)} - p_{(m+1)r}\big) \\ & + ... + \, (P_k - p_{kr})]X + [(T - P_1) + (T - P_2) + ... + (T - P_k)]\, Y \end{aligned} \tag{5}$$

$$Z_2 = \left[T - \sum_{i=1}^{m} p_{i1} - \sum_{i=m+1}^{k} p_{ir}\right] X + [KT - T]\, Y \tag{6}$$

Now, subtracting (6) from (4), we get

$$Z_1 - Z_2 = \left[T - \sum_{i=1}^{k} p_{i1}\right] X + [KT - T]\, Y - \left[T - \sum_{i=1}^{m} p_{i1} - \sum_{i=m+1}^{k} p_{ir}\right] X - [KT - T]\, Y \tag{7}$$

$$Z_1 - Z_2 = \left[T - \sum_{i=1}^{k} p_{i1}\right] X - \left[T - \sum_{i=1}^{m} p_{i1} - \sum_{i=m+1}^{k} p_{ir}\right] X \tag{8}$$

$$= \left[T - \sum_{i=1}^{k} p_{i1} - T + \sum_{i=1}^{m} p_{i1} + \sum_{i=m+1}^{k} p_{ir}\right] X \tag{9}$$

$$= \left[\sum_{i=1}^{m} p_{i1} + \sum_{i=m+1}^{k} p_{ir} - \sum_{i=1}^{k} p_{i1}\right] X \tag{10}$$

$$= \left[\sum_{i=m+1}^{k} p_{ir} - \sum_{i=m+1}^{k} p_{i1}\right] X \tag{11}$$

Since as per (11) the term p_{i1} exceeds the term p_{ir} and hence in totality the value of the equation comes out to be negative. Thus, the proposed CBRP reduces the cost of shuffling data and hence the performance of MapReduce will improve.

5. Experimental evaluation and results

5.1. Configuring Hadoop for performance tuning

There are large number of attributes in Hadoop influencing the job performance [30]. Moreover, we have tuned the attributes directly influencing the reducer. For all and all communication between mappers and reducers we find the memory allocated to a reducer if tuned properly will yield better result. Moreover, increasing the number of threads for data copying during overlapping phase of map and reduce will also impact it in a significant way. The list below shows the attributes which have been tuned to optimize the performance.

- Tasktracker.http.threads
- Mapred.reduce.parallel.copies
- Mapred.reduce.task
- Mapred.reduce.slowstart.completed.maps

5.2. Testbed setup

In order to carry the experiments in the Hadoop cluster, a small test-bed was set-up using HP Proliant Blade server. With the help of its virtualization firmware and Hyper-V manager, 12 nodes were created and segregated them under 3 racks following Hadoop's rack awareness policy oriented configuration. Out of the 12 nodes, one node was configured as the master (NameNode and JobTracker), whereas the other 11 nodes were configured as DataNode and TaskTracker as per Table 2.

All the nodes are equivalent in their hardware and software capability. These nodes have been configured to behave like a multinode Hadoop cluster with 12 nodes. Table 3 presents the configurations of these nodes.

Table 2. Hardware specifications of testbed components.

Node 1	Master node ($NameNode + Resource\ Manager + Application\ Master + Secondary\ NameNode$) With OS ($Ubuntu$ 18.04 *desktop edition*)
Node 2	Slave node ($DataNode + Node\ Manager$) With OS ($Ubuntu$ 18.04 *server edition*)

Table 3. Hardware specifications of testbed components.

Component	Specification
HardDisk	500 GB SATA
RAM	8GB DDR3
Processor OS	Intel (R) Xeon (R) CPU E5-2420 v2 @ 2.20GHz 2.19GHZ Ubuntu 18.04 desktop and server edition
Hadoop Version	2.7.1
Network	1000 Mbps Gigabit Network Switch

5.3. Experiments performed

In order to evaluate the effect of our proposed technique on the performance of Hadoop we have run a set of benchmark like microbenchmarking . These include benchmarks like Sort, WordCount and TeraSort. Sort and WordCount are well-known benchmarks used for evaluation at Yahoo. These set of benchmarks are used because some of them are either map-intensive or reduce-intensive. Besides, Sort was also used by Google to evaluate its computation engine MapReduce [2]. The workloads in case of Sort and WordCount are generated by random workload and RandomTextWriter. WordCount counts the occurrence of words in the input. The Sort benchmark sorts the input. It parses the input which is an identity function and simply outputs key value pairs as output. The input in K-means and TeraSort are generated by random data generator and TeraGen. In TeraSort input data generated by workload TeraGen is sampled and then map/reduce is used to sort the total data generated in order. To take cognizance of the variation of observations each benchmark is run 5 times and the values calculated are based on the average of these observations.

To access the performance improvement of our CBRP technique on MapReduce we have used several performance indicators. These indicators include job execution time, traffic analysis and the value of static threshold (beta). In case of job execution time, the number of maps in the job are varied to note the effects of scaling of maps in the job with the performance indicator. Further the number of maps in evaluation of traffic analysis of benchmarks and calculation of optimal value of static threshold (beta) are kept constant to account for the multitude effects of benchmarks. The performance comparison of our proposed technique is made with the native Hadoop. We have evaluated these results on HiBench a new popular benchmark suite for a mixture of loads like real life applications and synthetic workloads [30].

5.4. Results and discussion

5.4.1. Execution time vs. number of maps

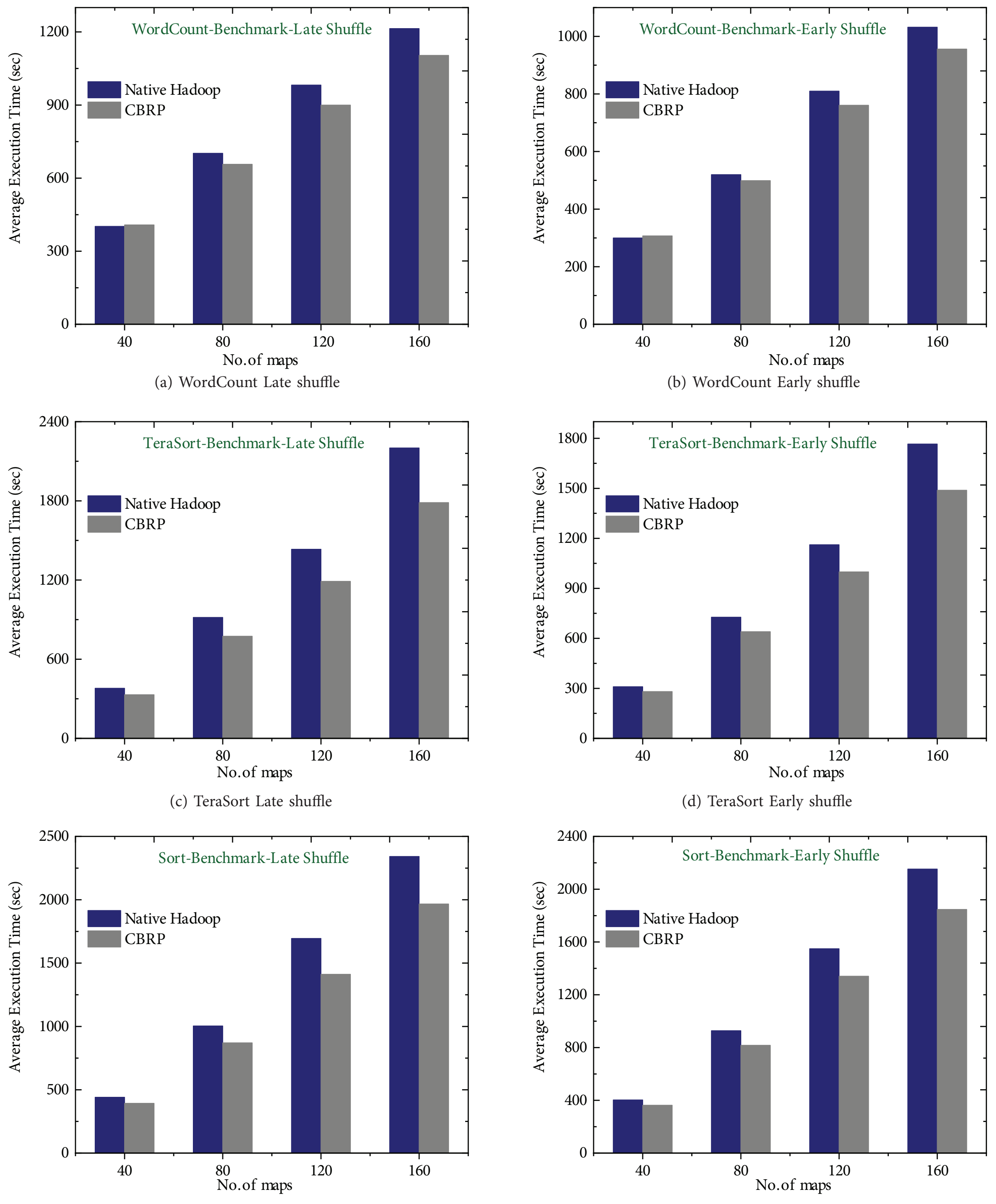
Figures 5a and 5b demonstrate the effect on the completion time of job in late and early shuffle when number of maps is varied in WordCount. As the number of maps is increased in both scenarios, there is a prominent decrease in the completion time of the job in our CBRP. The decrease in completion time in late shuffle is more than early shuffle because the decision on reducer placement is taken after completion of all maps and hence its placement is more precise than early shuffle. There is an unusual behavior noted in both early and late shuffle when number of map tasks are small CBRP performs worse than native Hadoop. This can be attributed to the fact that WordCount generates less intermediate map output data and the proper placement of reducer will incur delay.

Figures 5c and 5d show the effect on the completion time of job in late and early shuffle when number of maps is varied in TeraSort. These graphs portray a clear picture that CBRP improves the completion time to a significant level. As the number of maps in the job is scaled-up, the gain experienced in both late and early shuffle is more showing the proper placement of reducer has a major impact on the performance.

Figures 5e and 5f indicate the effect on the completion time of job in late and early shuffle when number of maps is varied in Sort. Since Sort generates a lot of intermediate traffic and proper placement of reducer will play a pivotal role in decreasing the completion time of job.

To summarize the results in the Figure 5 set , CBRP has a major effect on decreasing the completion time of job. After evaluation of all the benchmarks, we note that CBRP outperforms the native Hadoop in almost in all cases. There is a clear picture in the graphs whether benchmark is more CPU intensive or IO intensive, CBRP plays an important part in minimizing the completion time. The average gain experienced in

WordCount, TeraSort, and Sort, in both late shuffle and early shuffle is about (7%, 5.2%), (17.2%, 14%), and (15.3%, 13.2%), respectively.



(a) WordCount Late shuffle

(b) WordCount Early shuffle

(c) TeraSort Late shuffle

(d) TeraSort Early shuffle

(e) Sort Late shuffle

(f) Sort Early shuffle

Figure 5. Average execution time of default Hadoop and counter based reducer placement scheme as a function of varying number of map tasks [for three benchmarks: WordCount (a- late shuffle, b- early shuffle); TeraSort (c- late shuffle, d- early shuffle); and Sort (e- late shuffle, f- early shuffle)].

5.4.2. Static threshold vs. execution time

Figure 6a demonstrates the effect of the value of static threshold (beta) on the performance of job in late shuffle. It can be inferred from graph, tuning the value of beta in case of WordCount does not have a significant effect in reducing the completion time of job. However, both Sort and TeraSort show a major impact on reducing the completion time as the value of beta is scaled up. This is due to the time required for moving large amount of map intermediate data and henceforth value of beta needs to be set properly.

Figure 6b demonstrates the effect of the value of beta on the completion time of job in early shuffle. In early shuffle map is overlapped with reduce, so the value required in beta is much lesser than the late shuffle. In case of benchmarks such as Sort, TeraSort the variation in completion time is much more as the value of static threshold is tuned properly, although similar trend is not noticed during WordCount because of being a map-intensive benchmark.

To summarize the results in the Figure 6 set our proposed technique plays a prominent role in the reduction of completion time of job. It can further be interpreted that based on the job characteristics beta value needs to be set and not a single value is good for all the benchmarks.

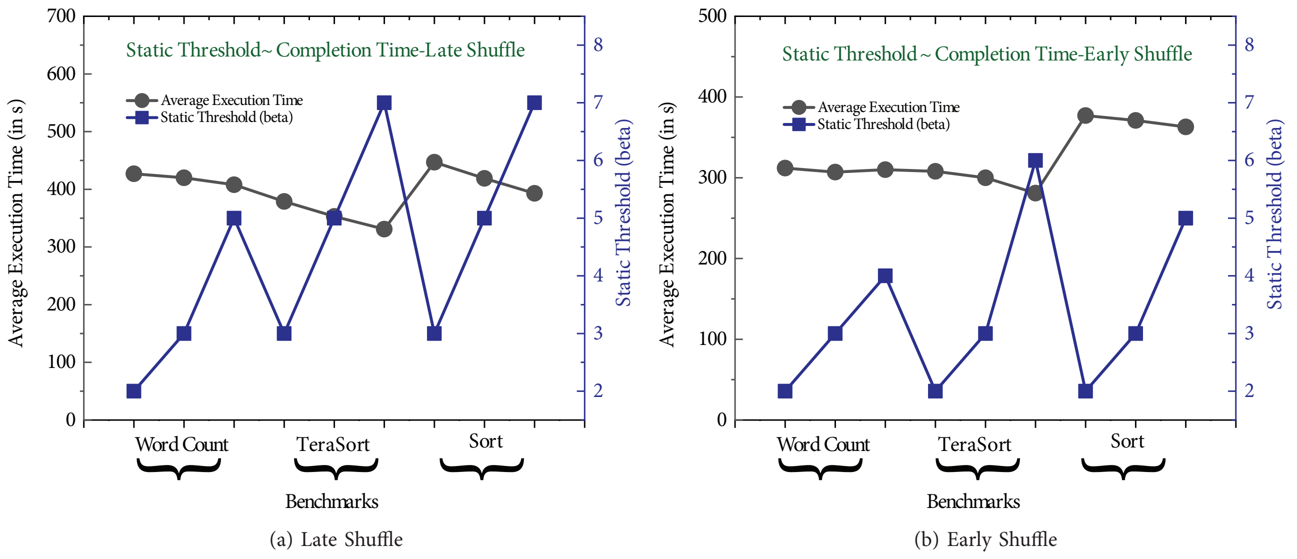


Figure 6. Static threshold value in different benchmarks (y-axis to the left denotes the average execution time of the job, y-axis to the right denotes the static threshold value and x-axis shows different benchmarks).

5.4.3. Traffic analysis of benchmarks

Figure 7a depicts the amount of shuffle required for placement of reducer in case of late shuffle. The traffic is categorized as data-local, rack-local and off-local. The graph depicts in all cases of CBRP there is a major improvement in minimizing the off-local shuffle. This reduction of off-local shuffle automatically improves both data-local as well as rack-local shuffle, hence proving our merit of the proposed technique.

Figure 7b depicts the amount of shuffle required for placement of reducer in case of early shuffle. In early shuffle map is synchronously launched with reduce, hence the improvement in minimizing the off-local shuffle is less. Further based on the graph there is a major improvement in cross-rack shuffle than data-local.

To summarize the results in the Figure 7 set CBRP plays a major effect on reducing cross-rack shuffle.

After running all the set of benchmarks, we see there is a major effect on improvement in both local as well as rack-local shuffle. The performance improvement in benchmarks like Sort, TeraSort is much more as compared to WordCount. This is due to the benchmark characteristics of having high-moderate disk IO. In early shuffle the minimization in off-local shuffle is less as compared to late shuffle. The average gain experienced in off-local shuffle in WordCount, TeraSort, and Sort in both late shuffle and early shuffle is about (6.1%, 6%), (18%, 12%), and (14%, 10%), respectively.

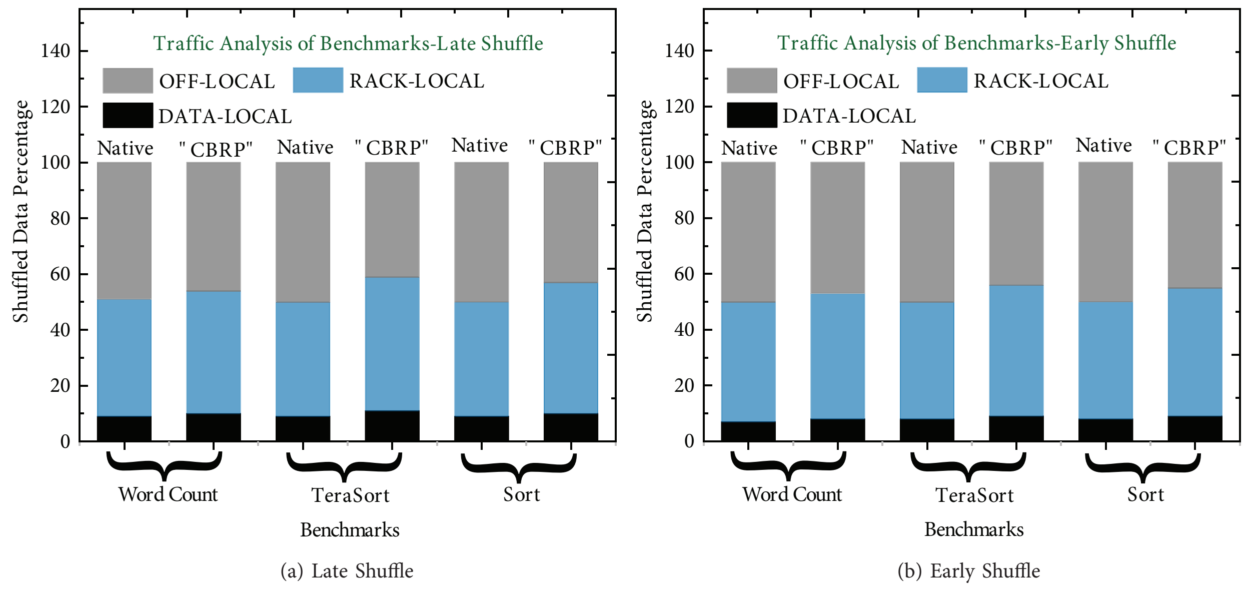


Figure 7. Percentage of shuffle for off local and rack local data for the placement of reduce tasks with different benchmarks in native Hadoop and counter based reducer placement.

6. Conclusion

In this paper, we have demonstrated the effect of locality oblivious reducer placement in MapReduce and have accounted for the performance deteriorations encountered. Our proposed mechanism CBRP that places reducers intelligently based on rack and map counter values. Based on experiments conducted, it was found that our proposed algorithm improves data locality by about 11% to 25% and off local shuffles by about 6% to 18%; thereby reducing completion time of jobs. Improvements amounted between 4% and 17% when compared to native Hadoop's inherent locality oblivious reducer placement. We intend to enrich this research with future works on selecting an optimal number of reducers in an application and reducer placement with concurrently running jobs in a cluster. Also enumerating the optimal value of beta shall be taken up as a future work.

Acknowledgment

The authors acknowledge the support of Mr. Fayaz Ahmad Dar, Lecturer Govt. Girls Higher Secondary School Chanapora, Jammu & Kashmir for providing insights into designing the mathematical analysis. The authors also acknowledge the support of Mr. Subhendu Sekhar Paik & Ms. Kuheli Roy Barman, Research Scholar of National Institute of Technology, Meghalaya & Silchar for there support.

References

[1] Gandomi A, Haider M. Beyond the hype: big data concepts, methods, and analytics. International Journal of Information Management 2015; 35 (2): 137-144.

[2] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Communications of the ACM 2008; 51 (1): 107-113.

[3] Hussain MW, Reddy KH, Roy DS. Resource aware execution of speculated tasks in Hadoop with sdn. International Journal of Advanced Science and Technology 2019; 28 (13): 72-84.

[4] White T. Hadoop: The Definitive Guide. USA: O'Reilly Media, Inc., 2012.

[5] Herodotou H. Hadoop performance models. arXiv 2011; arXiv:1106.0940 [cs.DC].

[6] Zhang X, Wu Y, Zhao C. MrHeter: improving mapreduce performance in heterogeneous environments. Cluster Computing 2016; 19 (4): 1691-1701.

[7] Xiong R, Luo J, Dong F. Optimizing data placement in heterogeneous Hadoop clusters. Cluster Computing 2015; 18 (4): 1465-1480.

[8] Hammoud M, Sakr MF. Locality-aware reduce task scheduling for mapreduce. In: IEEE Third International Conference on Cloud Computing Technology and Science; NW Washington, DC, USA; 2011. pp. 570-576.

[9] Reddy KH, Das H, Roy DS. A data aware scheme for scheduling big data applications with SAVANNA Hadoop. In: Elkhodr M, Hassan QF, Shahrestani S (editors). Networks of the Future. USA: Chapman and Hall/CRC, 2017, pp. 377-392.

[10] Ashu A, Hussain MW, Reddy KH, Roy DS. Intelligent data compression policy for Hadoop performance optimization. In: International Conference on Soft Computing and Pattern Recognition; Hyderabad, India; 2019. pp. 80-89.

[11] Wang J, Shang P, Yin J. Draw: a new data-grouping-aware data placement scheme for data intensive applications with interest locality. In: Springer Cloud Computing for Data-Intensive Applications; New York, NY, USA; 2014. pp. 149-174.

[12] Xiong R, Luo J, Dong F. SLDP: a novel data placement strategy for large-scale heterogeneous Hadoop cluster. In: IEEE Second International Conference on Advanced Cloud and Big Data; Toulouse, France; 2014. pp. 9-17.

[13] Paik SS, Goswami RS, Roy DS, Reddy KH. Intelligent data placement in heterogeneous Hadoop cluster. In: Springer International Conference on Next Generation Computing Technologies; Singapore; 2017. pp. 568-579.

[14] Reddy KH, Roy DS. Dppacs: a novel data partitioning and placement aware computation scheduling scheme for data-intensive cloud applications. The Computer Journal 2015; 59 (1): 64-82.

[15] Shvachko K, Kuang H, Radia S, Chansler R. The Hadoop distributed file system. In: IEEE 2010 26th symposium on mass storage systems and technologies (MSST); NW Washington, DC, USA; 2010. pp. 1-10.

[16] Shafer J, Rixner S, Cox AL. The Hadoop distributed filesystem: balancing portability and performance. In: IEEE 2010 International Symposium on Performance Analysis of Systems & Software (ISPASS); White Plains, NY, USA; 2010. pp. 122-133.

[17] Nabavinejad SM, Goudarzi M, Mozaffari S. The memory challenge in reduce phase of mapreduce applications. IEEE Transactions on Big Data 2016; 2 (4): 380-386.

[18] Hammoud M, Rehman MS, Sakr MF. Center-of-gravity reduce task scheduling to lower mapreduce network traffic. In: IEEE Fifth International Conference on Cloud Computing; Honolulu, USA; 2012. pp. 49-58.

[19] Ho LY, Wu JJ, Liu P. Optimal algorithms for cross-rack communication optimization in mapreduce framework. In: IEEE 4th International Conference on Cloud Computing; NW Washington, DC, USA; 2011. pp. 420-427.

[20] Arslan E, Shekhar M, Kosar T. Locality and network-aware reduce task scheduling for data-intensive applications. In: IEEE Proceedings of the 5th International Workshop on Data-Intensive Computing in the Clouds; New Orleans, LA, USA; 2014. pp. 17-24.

[21] Shen H, Sarker A, Yu L, Deng F. Probabilistic network-aware task placement for mapreduce scheduling. In: IEEE International Conference on Cluster Computing (CLUSTER); Taipei, Taiwan; 2016. pp. 241-250.

[22] Zhao Y, Tian C, Fan J, Guan T, Qiao C. RPC: joint online reducer placement and coflow bandwidth scheduling for clusters. In: IEEE 2018 26th International Conference on Network Protocols (ICNP); Cambridge, UK; 2018. pp. 187-197.

[23] Ananthanarayanan G, Kandula S, Greenberg AG, Stoica I, Lu Y et al. Reining in the outliers in map-reduce clusters using mantri. InOsdi 2010; 10 (1): 24.

[24] Guo Y, Wang Z, Yin X, Shi X, Wu J. Joint optimization of task placement and routing in minimizing inter-DC coflow completion time. In: IEEE 26th International Conference on Computer Communication and Networks (ICCCN); Vancouver, Canada; 2017. pp. 1-2.

[25] Guo Y, Wang Z, Zhang H, Yin X, Shi X et al. Joint optimization of tasks placement and routing to minimize coflow completion time. Journal of Network and Computer Applications 2019; 135: 47-61.

[26] Tang Z, Jiang L, Zhou J, Li K, Li K. A self-adaptive scheduling algorithm for reduce start time. Future Generation Computer Systems 2015; 43: 51-60.

[27] Lin M, Zhang L, Wierman A, Tan J. Joint optimization of overlapping phases in mapreduce. Performance Evaluation 2013; 70 (10) : 720-735.

[28] Zaharia M, Konwinski A, Joseph AD, Katz RH, Stoica I. Improving mapreduce performance in heterogeneous environments. InOsdi 2008; 8 (4): 7.

[29] Chen Q, Liu C, Xiao Z. Improving mapreduce performance using smart speculative execution strategy. IEEE Transactions on Computers 2013; 63 (4) : 954-967.

[30] Huang S, Huang J, Dai J, Xie T, Huang B. The HiBench benchmark suite: characterization of the mapreduce-based data analysis. In: IEEE 26th International Conference on Data Engineering Workshops (ICDEW); New York, NY, USA; 2010. pp. 41-51.