International Journal of Scientific Research in Computer Science, Engineering and Information Technology
© 2017 IJSRCSEIT | Volume 2 | Issue 2 | ISSN : 2456-3307

An Efficient Resource Aware Scheduling Algorithm for Mapreduce Clusters

Sharmilarani D, Vinothini K, Ramya V, Shobika R
Department of Computer Science Engineering, Sri Krishna Institute of Technology, Coimbatore, TamilNadu, India

ABSTRACT
MapReduce has become a popular model for data-intensive computation in recent years. The schedulers are critical in enhancing the performance of MapReduce/Hadoop in presence of multiple jobs with different characteristics and performance goals. The propose improve the resource aware scheduling technique for Hadoop map-reduce multiple jobs running that aims to improving resource utilization across multiple virtual machines while observing completion time goals. The propose algorithm influences job profiling information to dynamically adjust the number of slots allocation based on job profile and resource utilization on each machine, as well as workload placement across them, to maximize the resource utilization of the cluster. This single node experimental result show the resource aware scheduling that improves job running time and reduce the resource utilization without introducing stragglers.

Keywords : Hadoop, Map-Reduce, Resource Aware Scheduling Profiling

I. INTRODUCTION

AIS

MapReduce is a processing technique and a program model for distributed computing based on java. The MapReduce algorithm contains two important tasks, namely Map and Reduce. Map takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). Secondly, reduce task, which takes the output from a map as an input and combines those data tuples into a smaller set of tuples. As the sequence of the name MapReduce implies, the reduce task is always performed after the map job. located along coast lines or, when out of range of terrestrial networks, through a growing number of satellites that are fitted with special AIS receivers which are capable of deconflicting a large number of signatures. MapReduce is that it is easy to scale data processing over multiple computing nodes. Under the MapReduce model, the data processing primitives are called mappers and reducers. Decomposing a data processing application into mappers and reducers is sometimes nontrivial. But, once we write an application in the MapReduce form, scaling the application to run over hundreds, thousands, or even tens of thousands of machines in a cluster is merely a configuration change. This simple scalability is what has attracted many programmers to use the MapReduce model.

II. METHODS AND MATERIAL

1. Ease of Use

A. MAP-REDUCE Map stage

The map or mapper's job is to process the input data. Generally, the input data is in the form of file or directory and is stored in the Hadoop file system (HDFS). The input file is passed to the mapper function line by line. The mapper processes the data and creates several small chunks of data. The MapReduce framework operates on key-value pairs, that is, the framework views the input to the job as a set of key-value pairs and produces a set of key-value pair as the output of the job, conceivably of different types.

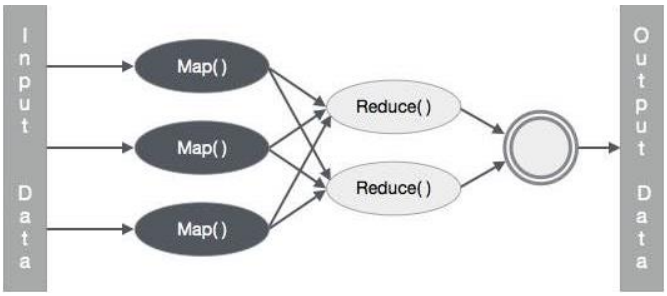

Figure 1. Map-Reduce work flow

The key and value classes have to be serializable by the framework and hence, it is required to implement the Writable interface. Additionally, the key classes have to implement the Writable Comparable interface to facilitate sorting by the framework.

Reduce stage: This stage is the combination of the Shuffle stage and the Reduce stage. The Reducer's job is to process the data that comes from the mapper. After processing, it produces a new set of output, which will be stored in the HDFS.

B. HDFS File System

Hadoop File System was developed using distributed file system design. It is run on commodity hardware. Unlike other distributed systems, HDFS is highly fault tolerant and designed using low-cost hardware.

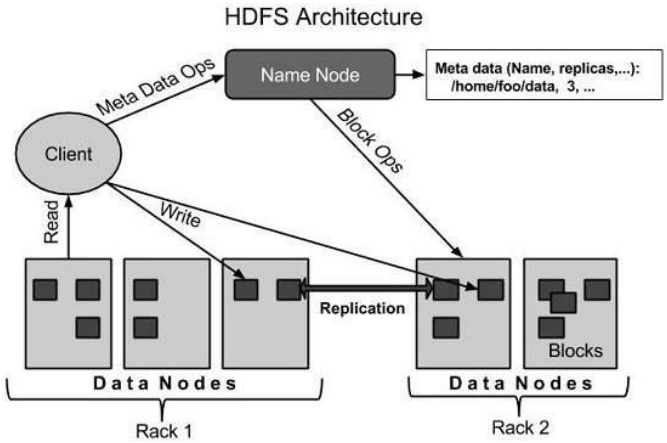

Figure 3. HDFS follows the master-slave architecture

HDFS holds very large amount of data and provides easier access. To store such huge data, the files are stored across multiple machines. These files are stored in redundant fashion to rescue the system from possible data losses in case of failure. HDFS also makes applications available to parallel processing.

C. Hadoop MRv1

Apache Hadoop 1.x and its processing framework, MRv1 (Classic), so that we can get a clear picture of the differences in Apache Hadoop 2.x MRv2 (YARN) in terms of architecture, components, and processing framework. In Hadoop 1, a single Namenode managed the entire namespace for a Hadoop cluster. With HDFS federation, multiple Namenode servers manage namespaces and this allows for horizontal scaling, performance improvements, and multiple namespaces. The implementation of HDFS federation allows existing Namenode configurations to run without changes. For Hadoop administrators, moving to HDFS federation requires formatting Namenodes, updating to use the latest Hadoop cluster software, and adding additional Namenodes to the cluster.

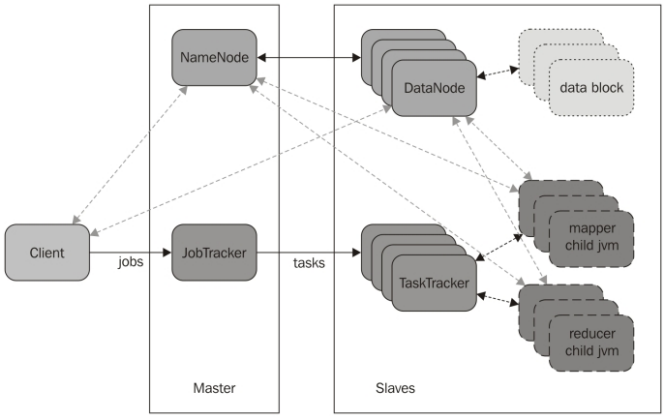

Figure 2. Hadoop MRv1 architecture diagram

Apache Hadoop is a scalable, fault-tolerant distributed system for data storage and processing. The core programming model in Hadoop is MapReduce.

2. Hybrid Job-Driven Scheduling

The cost-efficient for a tenant with a limited budget to establish a virtual MapReduce cluster by renting multiple virtual private servers (VPSs) from a VPS provider. To provide an appropriate scheduling scheme for this type of computing environment, the existing paper a hybrid job-driven scheduling scheme (JoSS for short) are used. JoSS provides not only job level scheduling, but also map-task level scheduling and reduce-task level scheduling. JoSS classifies MapReduce jobs based on job scale and job type and designs an appropriate scheduling policy to schedule each class of jobs. The goal is to improve data locality for both map tasks and reduce tasks, avoid job starvation, and improve job execution performance. Two variations of JoSS are further introduced to

separately achieve a better map-data locality and a faster task assignment.

In order to provide an appropriate scheduling scheme for a tenant to achieve a high map-and-reduce data locality and improve job performance in his/her virtual MapReduce cluster, in this paper we propose a hybrid job-driven scheduling scheme (JoSS for short) by providing scheduling in three levels: job, map task, and reduce task. JoSS classifies MapReduce jobs into either large or small jobs based on each job's input size to the average datacenter scale of the virtual MapReduce cluster, and further classifies small MapReduce jobs into either map-heavy or reduce-heavy based on the ratio between each job reduce-input size and the job's map-input size. Then JoSS uses a particular scheduling policy to schedule each class of jobs such that the corresponding network traffic generated during job execution (especially for inter-datacenter traffic) can be reduced, and the corresponding job performance can be improved. In addition, we propose two variations of JoSS, named JoSS-T and JoSS-J, to guarantee a fast task assignment and to further increase the VPS-locality, respectively.

D. State of the Art JoSS schduling Algorithms

Job Classification: Before introducing the algorithm of JoSS, first describe how JoSS classifies jobs and schedules each class of jobs. Let Sreduce and Smap be the total reduceinput size and the total map-input size of J, respectively. Based on the ratio of Sreduce over Smap, J can be classified into either a reduce heavy job or a map-heavy job. If J satisfies Eq. (1), implying that the network overhead is dominated by J's reduce-input data, then J is classified as a reduce-heavy job (RH job for short). Otherwise, J is classified as a map-heavy job (MH job for short). Note that td is a threshold to determine the classification, td 0:

$$\frac{S_{Reduce}}{S_{map}} > td \quad (1)$$

In fact, $S_{map} = \sum_{i=1}^{m} |B_i|$ where |Bi| is the size of Bi, $S_{reduce} = \sum_{i=1}^{m} ¿B_i \vee \cdot FP_i$ where FPi is the filtering percentage of Bi showing the ratio of $M_i^{'}$ map-output size over $M_i^{'}$ map-input size, FPi ≥ 0.

Scheduling Policies: JoSS utilizes the following three scheduling policies

Policy A: This policy is designed for a small RH job. If J is a small RH job, it would be better that each reducer of J is close to all mappers of J since the reducer can more quickly retrieve its input data from all the mappers. But this also implies that all mappers of J should be close to each other. Hence, policy A works as follows. It first chooses cen_w , which is a datacenter having the least number of unprocessed tasks among all the k datacenters, $cen_w \in cen_1 cen_2 \ldots . cen_k t$ hen it schedules all tasks of J to cen_w by putting J's map tasks and J's reduce tasks at the end of $MQ_{w,0} \wedge RQ_{w,0}$, respectively. In this way, all these tasks can be executed only by the VPSs at $c\,en_w$, and each reducer of J can retrieve its input data from its local datacenter (i.e., reduce-data locality can be improved).

Policy B : This policy is designed for a small MH job. If J is a small MH job, it would be better that each mapper of J is close to its input block, and each reducer of J is close to most mappers of J. Hence, policy B works as follows: It schedules J's map tasks based on the number of unique input blocks of J held by each datacenter. If a datacenter holds more unique blocks of J, more map tasks of J will be scheduled to the VPSs at this datacenter. The purpose is allowing each mapper of J to retrieve its input block from its local datacenter. In addition, to make J's reducers close to most J's mappers, policy B schedules all reduce tasks of J to the datacenter that holds the maximum number of J's unique blocks.

Policy C: This policy is designed for a large job. If J is a large job to a virtual MapReduce cluster, using one datacenter of the cluster to run all map tasks of J might need several rounds to finish these map tasks, implying that job turnaround time will prolong. To prevent this from happening, it is better not to use a single datacenter to run all these map tasks. Hence, as long as J is a large job, JoSS utilizes policy C, which in fact uses the same strategy of policy B to schedule all tasks of J. However, in policy C, all the map tasks scheduled to cen_c will not be put into $MQ_{C,0}$ since $MQ_{C,0}$ is reserved for only small jobs. Instead, these map tasks will be put into a new map-task queue created for cenc. Similarly, the reduce tasks of the large job scheduled to cenc will be put into a new reduce-task queue created for cen_c , rather than $RQ_{c,0}$. The purpose is to separate large jobs and small jobs into different queues and allow JoSS to avoid job starvation.

Selecting the Best Threshold: If J is classified as a RH job, policy A will be used to schedule J. The worst case for J's mappers is that all of them need to retrieve their input blocks from other datacenters. However, because of policy A, J's reducers can completely retrieve their

input from their local datacenters. Hence, the worst-case inter-datacenter traffic for this classification, denoted by TR1, is

$$TR_1 = \sum_{i=1}^{m} |B_i| \qquad (2)$$

On the other hand, if J is classified as a MH job, policy B will be used, which guarantees that all mappers of J can always retrieve their input blocks from their local datacenters. But in the worst case (i.e., all map tasks of J are evenly scheduled to each datacenter because of the even distribution of J's input blocks over all datacenters), J's reducers have to retrieve the $\frac{K+1}{K}$ k of their input from other datacenters where k is the total number of the datacenters comprising the virtual MapReduce cluster. Hence, the worst-case inter datacenter traffic for this classification, denoted by TR_2 , is

$$TR_2 = \frac{k-1}{k} \cdot \sum_{i=1}^{m} |B_i| \cdot FP_J (3)$$

If TR2 > TR1, J should be determined as a RH job (rather than a MH job) since the related worst-case inter datacenter traffic is less. Otherwise, J should be determined as a MH job, rather than a RH job. In fact, TR2 > TR1 can be expressed as $\frac{k-1}{k} \cdot \sum_{i=1}^{m} |B_i| \cdot FP_J > \sum_{i=1}^{m} |B_i|$

which also implies that $\frac{k-1}{k} \quad FP_J > 1$.

$$FP_J > \frac{k}{k-1} (4)$$

the condition to determine a RH job (i.e., FPJ > td we can derive the best value of td, i.e.,

$$td = \frac{k}{k-1} (5)$$

E. Disadvantage of Hybrid Job-Driven Scheduling

JOSS, jobs are still forced to wait significantly when the MapReduce system assigns equal sharing of resources due to dependencies between Map, Shuffle, Sort, Reduce phases. JOSS response time is still longer than necessary due to dependencies between the Map/Shuffle/Sort/Reduce phases. Joss Task assignment problem into a Linear Sum Assignment Problem so as to find the optimal assignment so it will take long time for job assign.

III. RESULTS AND DISCUSSION

1. Proposed Framework

In our framework, each Task Tracker node monitors resources such as CPU utilization, disk channel IO in bytes/s, and the number of page faults per unit time for the memory subsystem. Although we anticipate that other metrics will prove useful, we propose these as the basic three resources that must be tracked at all times to improve the load balancing on cluster machines. In particular, disk channel loading can significantly impact the data loading and writing portion of Map and Reduce tasks, more so than the amount of free space available. Likewise, the inherent opacity of a machine's virtual memory management state means that monitoring page faults and virtual memory-induced disk thrashing is a more useful indicator of machine load than simply tracking free memory. Instead of having a fixed number of available computation slots configured on each Task Tracker node, we compute this number dynamically using the resource metrics obtained from each node. In one possible heuristic, we set the overall resource availability of a machine to be the minimum availability across all resource metrics. In a cluster that is not running at maximum utilization at all times, we expect this to improve job response times significantly as no machine is running tasks in a manner that runs into a resource bottleneck. the fixed maximum number of compute slots per node, viewing it as a resource allocation decision made by the cluster administrators at configuration time. Instead, we decide the order in which free Task Tracker slots are advertised according to their resource availability. As Task Tracker slots become free, they are buffered for some small-time period (say, 2s) and advertised in a block. Task Tracker slots with higher resource availability are presented first for scheduling tasks on. In an environment where even short jobs take a relatively long time to complete, this will present significant performance gains. Instead of scheduling a task onto the next available free slot (which happens to be a relatively resourcedeficient machine at this point), job response time would improve by scheduling it onto a resource-rich machine, even if such a node takes a longer time to become available. Buffering the advertisement of free slots allows for this scheduling allocation. The output of the resource aware algorithm process F. Map /Reduce Programming model MapReduce is a popular programming model for processing large data sets, initially proposed by Google. Now it has been a de facto standard for large scale data processing on the cloud. Hadoop is an open-source java implementation of MapReduce. When a user submits jobs to the Hadoop cluster, Hadoop system breaks each job into multiple map tasks and reduce tasks. Each map task processes (i.e. scans and

records) a data block and produces intermediate results in the form of key-value pairs. Generally, the number of map tasks for a job is determined by input data. There is one map task per data block. The execution time for a map task is determined by the data size of an input block. The reduce tasks consists of shuffle/sort/reduce phases. In the shuffle phase, the reduce tasks fetch the intermediate outputs from each map task. In the sort/reduce phase, the reduce tasks sort intermediate data and then aggregate the intermediate values for each key to produce the final output. The number of reduce tasks for a job is not determined, which depends on the intermediate map outputs.

G. ***Job Completion Time Estimator***

The Job Completion Time Estimator estimates the number of map tasks that should be allocated concurrently (s^j_{req}) to meet the completion time goal of each job. To perform this calculation, it relies on the completion time goal s^j_{goal} , the number of pending map tasks (s^j_{pend}), and the observed average task length. Notice that the scenario we focus on is very dynamic, with jobs entering and leaving the system unpredictably, so the goal of this component is to provide estimates of s^j_{req} that guide resource allocation.

H. Task Scheduler

Task Scheduler is responsible for enforcing the placement decisions, and for moving the system smoothly between a placement decision made in the last cycle to a new decision produced in the most recent cycle. The Task Scheduler schedules tasks according to the placement decision made by the Placement Controller. Whenever a task completes, it is the responsibility of the Task Scheduler to select a new task to execute in the freed slot, by providing a task of the appropriate type from the appropriate job to the given Task Tracker. The placement algorithm generates new placements, but these are not immediately enforced as they may overload the system due to tasks still running from the previous control cycle. The Task Scheduler component takes care of transitioning without overloading any Task Trackers in the system by picking jobs to assign to the Task Tracker that do not exceed its current capacity, sorted by lowest utility first. For instance, a Task Tracker that is running 2 map tasks of job A may have a different assignment for the next cycle, say, 4 map tasks of job B. Instead of starting the new tasks right away while the previous ones are still running, new tasks will only start running as previous tasks complete and enough resources are freed.

I. Job Profiles

The proposed job scheduling technique relies on the use of job profiles containing information about the resource consumption for each job. Profiling is one technique that has been successfully used in the past for MapReduce clusters. Its suitability in these clusters stems from the fact that in most production environments jobs are ran periodically on data corresponding to different time windows. V

2. Resource Aware Scheduling Algorithm

The job given a placement matrix, it defines a utility function that combines the number of map and reduce slots allocated to the job with its completion time goal and job characteristics. Below we provide a description of this function. Given placement matrices $P^M$and P^R, it can define the number of map and reduce slots allocated to a job j

as $s^j_{alloc} = \sum_{tt \in TT} P^M_{j,tt} = \sum_{tt \in TT} P^R_{j,tt}$.

It defines the utility of a job j given a placement $s^j_{pend} \wedge r^j_{pend}$ correspondingly.

$$u_j(p) = u^M_j(P^M) + u^R_j(P^R) \; where \; P = P^M + P^R \quad (6)$$

where u^M_j is a utility function that denotes increasing satisfaction of a job given a placement of map tasks, and u^R_j is a utility function that shows satisfaction of a job given a placement of reduce tasks. The definition of both functions is:

$$u^M_j(P^M) = \begin{cases} \dfrac{s^j_{alloc} - s^j_{req}}{s^j_{pend} - s^j_{req}} s^j_{alloc} \geq s^j_{req} \\ \dfrac{\log(s^j_{alloc})}{\log(s^j_{req})} - 1 \; s^j_{alloc} < s^j_{req} \end{cases} \quad (7)$$

$$u^R_j(P^R) = \frac{\log(r^j_{alloc})}{\log(r^j_{pend})} - 1$$

The intuition behind it is that reduce tasks should start at the earliest possible time, so the shuffle sub-phase of the job (reducers pulling data produced by map tasks) can be fully pipelined with execution of map tasks.

J. Map /Reduce Programming Model

Given an application placement matrix P, a utility value can be calculated for each job in the system. The performance of the system can then be measured as an ordered vector of job utility values, U. The objective of

RAS is to find a new placement P of jobs on Task Trackers that maximizes the global objective of the system, U(P), which is expressed as follows:

$$max\, min_{u_j}(P) \quad (8)$$

$$min\, \Omega_{tt,r} - \sum_{tt} \sum_{j} P_{j,tt} * \Gamma_{j,r} \quad (9)$$

Such that

$$P_{j,tt}\Gamma_{j,r} \leq \Omega_{tt,r} \text{¿}$$

$$\forall_{tt} \forall_{r} \sum_{j} \text{¿}$$

10

The proposed algorithm consists of two major steps: placing reduce tasks and placing map tasks. Reduce tasks are placed first to allow them to be evenly distributed across Task Trackers. By doing this we allow reduce tasks to better multiplex network resources when pulling intermediate data and also enable better storage usage. The placement algorithm distributes reduce tasks evenly across Task Trackers while avoiding collocating any two reduce tasks. If this is not feasible due to the total number of tasks it then gives preference to avoiding collocating reduce tasks from the same job. Recall that in contrast to other existing schedulers, RAS dynamically adjusts the number of map and reduce tasks allocated per Task Tracker while respecting its resource constraints. Notice also that when reduce tasks are placed first, they start running in shuffle phase. demand of resources is directly proportional to the number of map tasks placed for the same job. Therefore, in the absence of map tasks for the same job, a reduce task in shuffle phase only consumes memory. The second step is placing map tasks. This stage of the algorithm is utility-driven and seeks to produce a placement matrix that balances satisfaction across jobs while treating all jobs fairly. This is achieved by maximizing the lowest utility value in the system. This part of the algorithm executes a series of rounds, each of which tries to improve the lowest utility of the system. In each round, the algorithm removes allocated tasks from jobs with the highest utility, and allocates more tasks to the jobs with the lowest utility. For the sake of fairness, a task gets de-allocated only if the utility of its corresponding job remains higher than the lowest utility of any other job in the system. This results in increasing the lowest utility value across

3. Advantage of Traffic Knowledge

Balancing different kinds of workload in Task Tracker to increase the resource utilization of both I/O bound and CPU bound jobs. The resource aware scheduling algorithm that is said to improve the job execution without introducing stragglers. The resource aware scheduling algorithm for this and show that produces up to 18% improvement in resource utilization while allowing jobs to complete up to 1.3 times faster than current Hadoop schedulers. The resource aware priorities can accommodate multiple tasks' lengths, job sizes, and jobs' waiting time in this environment while reducing average response time. The resource aware scheduler based on dynamic priority in order to reduce the delay for variable length concurrent jobs, and relax the order of jobs to maintain data locality.

4. Implementation

Hadoop is a framework written in Java for running applications on large clusters of commodity hardware and incorporates features similar to those of the Google File System (GFS) and of the MapReduce computing paradigm. Hadoop's HDFS is a highly fault-tolerant distributed file system and, like Hadoop in general, designed to be deployed on low-cost hardware. It provides high throughput access to application data and is suitable for applications that have large data sets.

K. Word Count Job Running

The size of intermediate data has a big impact on performance of Hadoop. Three typical built-in benchmark applications in Hadoop are used in these experiments: Word Count without Combiner, Sort and Word-Count with combiner (WCC). These three benchmarks represent different relations between intermediate data and input data. Word Count without combiner, Sort, and Word-Count with combiner represent the cases where the size of intermediate data is larger than, equal to and smaller than the size of input data, respectively.

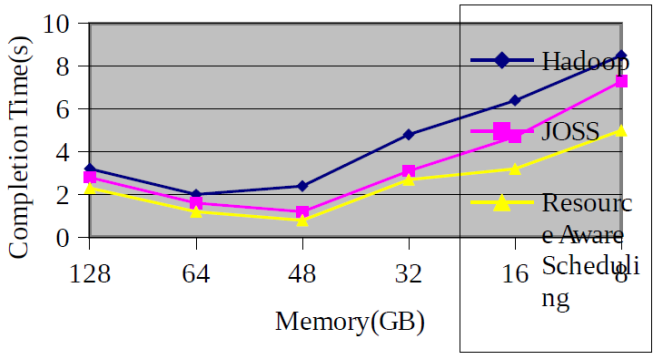

Figure 4. Comparison of job completion Time

IV. CONCLUSION

The proposed resource-aware scheduling technique for MapReduce multi-job workloads that aims at improving resource utilization across machines while observing completion time goals. Job profiling information to dynamically adjust the number of slots on each machine, as well as workload placement across them, to maximize the resource utilization of the cluster. Phase and Resource Information-aware Scheduler for MapReduce clusters that performs resource-aware scheduling at the phase level.

V. REFERENCES

[1] G. Z. Guo, G. Fox, and M. Zhou, "Investigation of data locality in mapreduce," in Proc. 12th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput., May 2012, pp. 419–426.

[2] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in Proc. 5th Eur. Conf. Comput. Syst., Apr. 2010, pp. 265–278.

[3] J. Jin, J. Luo, A. Song, F. Dong, and R. Xiong, "BAR: An efficient data locality driven task scheduling algorithm for cloud computing," in Proc. 11th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput., May 2011, pp. 295–304.

[4] M. Ehsan, and R. Sion, "LiPS: A cost-efficient data and task coscheduler for MapReduce," in Proc. IEEE 27th Int. Symp. Parallel Distrib. Process. Workshops PhD Forum, May 2013, pp. 2230–2233.

[5] J. Park, D. Lee, B. Kim, J. Huh, and S. Maeng, "Locality-aware dynamic VM reconfiguration on MapReduce clouds," in Proc. 21st Int. Symp. High-Perform. Parallel Distrib. Comput., Jun. 2012, pp. 27–36.

[6] X. Bu, J. Rao, and C.-Z. Xu, "Interference and locality-aware task scheduling for Mapreduce applications in virtual clusters," in Proc. 22nd Int. Symp. High-Perform. Parallel Distrib. Comput., Jun. 2013, pp. 227– 238.

[7] C. Tian, H. Zhou, Y. He, and L. Zha, "A dynamic mapreduce scheduler for heterogeneous workloads," in Proc. IEEE 8th Int. Conf. Grid Cooperative Comput., 2009, pp. 218–224.

[8] J. Polo, D. Carrera, Y. Becerra, J. Torres, E. Ayguade, M. Steinder, and I. Whalley, "Performance-driven task co-scheduling for mapreduce environments," in Proc. IEEE Netw. Oper. Manage. Symp., 2010, pp. 373–380.