Center-of-Gravity Reduce Task Scheduling to Lower MapReduce Network Traffic

Mohammad Hammoud, M. Suhail Rehman, and Majd F. Sakr
Carnegie Mellon University in Qatar
Education City, Doha, State of Qatar
Email: {mhhammou, suhailr, msakr}@qatar.cmu.edu

***Abstract*—MapReduce is by far one of the most successful realizations of large-scale data-intensive cloud computing platforms. MapReduce automatically parallelizes computation by running multiple map and/or reduce tasks over distributed data across multiple machines. Hadoop is an open source implementation of MapReduce. When Hadoop schedules reduce tasks, it neither exploits data locality nor addresses partitioning skew present in some MapReduce applications. This might lead to increased cluster network traffic. In this paper we investigate the problems of data locality and partitioning skew in Hadoop. We propose Center-of-Gravity Reduce Scheduler (CoGRS), a locality-aware skew-aware reduce task scheduler for saving MapReduce network traffic. In an attempt to exploit data locality, CoGRS schedules each reduce task at its *center-of-gravity* node, which is computed after considering partitioning skew as well. We implemented CoGRS in Hadoop-0.20.2 and tested it on a private cloud as well as on Amazon EC2. As compared to native Hadoop, our results show that CoGRS minimizes off-rack network traffic by averages of 9.6% and 38.6% on our private cloud and on an Amazon EC2 cluster, respectively. This reflects on job execution times and provides an improvement of up to 23.8%.**

I. Introduction

The exponential increase in the size of data has led to a wide adoption of parallel analytics engines such as MapReduce [3]. MapReduce is becoming a ubiquitous programming model. For instance, Rafique *et al.* employed MapReduce on asymmetric clusters of asymmetric multi-core and general purpose processors [27]. Mars [11] harnessed graphics processors power for MapReduce. Phoenix [28] evaluated MapReduce for multi-core and multiprocessor systems. Other work promoted MapReduce frameworks for dedicated data centers [3], [6], virtual machine clusters [12], [35] and public resource-grids [2].

Compared to traditional programming models, such as message-passing, MapReduce automatically and efficiently parallelizes computation by running multiple map and/or reduce tasks over distributed data across multiple machines. Hadoop [6] is an open source implementation of MapReduce. It relies on its own distributed file system called HDFS (Hadoop Distributed File System), which mimics GFS (Google File System) [5], to partition data into fixed equal-size chunks and distribute them over cluster machines. One of Hadoop's basic principles is: "moving computation towards data is cheaper than moving data towards computation". Consequently, Hadoop attempts to schedule map tasks in the vicinity of input chunks seeking reduced network traffic in an environment characterized by scarcity in network bandwidth.

In contrast to map task scheduling, Hadoop breaks its above basic principle when it schedules reduce tasks. This is mainly because the input to a reduce task is typically the output of many map tasks generated at multiple nodes, while the input to a map task exists at a solo node. With reduce task scheduling, once a slave (or a Task Tracker (TT) in Hadoop's parlance), polls for a reduce task at the master node, (referred to as Job Tracker (JT)), JT assigns TT a reduce task, R, irrespective of TT's network distance locality from R*'s* feeding nodes[1]. This might lead to an excessive data shuffling and performance degradation.

Map and reduce tasks typically consume large amount of data. In our experiments, we observed that the total intermediate output (or total reduce input) size is sometimes equal to the total input size of all map tasks (e.g. sort) or even larger (e.g., 44.2% for K-means)[2]. Similar observations were reached in [10], [13]. For this reason, optimizing the placement of reduce tasks to save network traffic becomes as essential as optimizing the placement of map tasks, which is already well understood and implemented in Hadoop systems.

Existing Hadoop's reduce task scheduler is not only locality unaware, but also *partitioning skew* unaware. As defined in [13], partitioning skew refers to the significant variance in intermediate keys' frequencies and their distribution across different data nodes. In our experiments, we observed partitioning skew to exist within certain Hadoop applications. Partitioning skew causes a shuffle skew where some reduce tasks receive more data than others. The shuffle skew problem can degrade performance because a job might get delayed by a reduce task fetching large input data. The node at which a reduce task is scheduled can highly mitigate the shuffle skew problem. In essence, the reduce task scheduler can determine the pattern of the communication traffic on the network, affect the quantity of shuffled data, and influence the runtime of MapReduce jobs.

Informed by the success and the increasing prevalence of

[1] A feeding node of a reduce task, R, is a node that hosts at least one of R*'s* feeding map tasks.

[2] Our experimentation environment and all our benchmarks are described in Section VI.

MapReduce, this paper explores the locality and the partitioning skew problems present in the current Hadoop implementation and proposes Center-of-Gravity Reduce Scheduler (CoGRS), a locality-aware skew-aware reduce task scheduler for MapReduce. CoGRS attempts to schedule every reduce task, *R*, at its *center-of-gravity* node determined by the network locations of *R's* feeding nodes and the skew in the sizes of *R's* partitions. The network is typically a bottleneck in MapReduce-based systems. By scheduling reducers at their center-of-gravity nodes, we argue for reduced network traffic which can possibly allow more MapReduce jobs to co-exist on the same system. CoGRS controllably avoids scheduling skew, a situation where some nodes receive more reduce tasks than others, and promotes pseudo-asynchronous map and reduce phases. Evaluations show that CoGRS is superior to native Hadoop.

In this work we make the following contributions:

- We describe and motivate the importance of considering data locality and partitioning skew in MapReduce task scheduling. We propose a mechanism that addresses both problems and demonstrate the benefits it offers to MapReduce-based systems.
- We implemented CoGRS in Hadoop 0.20.2 and conducted a set of experiments to evaluate its potential on a private homogenous cluster as well as on a shared heterogeneous cluster at Amazon EC2. On our private cloud, we observed that CoGRS successfully increases node-local data by 34.5% and decreases rack-local and off-rack data by 5.9% and 9.6%, on average, versus native Hadoop. Furthermore, we found that CoGRS increases node-local and rack-local data by 57.9% and 38.6%, respectively, and decreases off-rack data by 38.6%, on average, versus native Hadoop on an Amazon EC2 cluster.

The rest of the paper is organized as follows. A background on Hadoop network topology is given in Section II. Section III discusses how data locality is overlooked in Hadoop. We make the case for partitioning skew in Section IV. CoGRS's design is detailed in Section V. Section VI shows our evaluation methodology and results. In Section VII we provide a summary of prior work. Lastly, we conclude and discuss opportunities for future work in Section VIII.

II. Background

Hadoop assumes a tree-style network topology. Nodes are spread over different racks contained in one or many data centers. A salient point is that the bandwidth between two nodes is dependent on their relative locations in the network topology. For example, nodes that are on the same rack have higher bandwidth between them as opposed to nodes that are off-rack. Rather than measuring bandwidth between two nodes, which might be difficult in practice, Hadoop adopts a simple approach via: (1) representing the bandwidth between two nodes as a measure of distance, (2) assuming the distance from a node to its parent is 1, and (3) calculating the distance between any two nodes by adding up their distances to their closest common ancestor.

III. Data Locality in Hadoop

As described earlier, Hadoop does not consider data locality when scheduling reduce tasks. We define a total network distance of a reduce task, *R* (TND_R), as $\sum_{i=0}^{n} ND_{iR}$ where *n* is the number of partitions that are fed to *R* from *n* feeding nodes and *ND* is the network distance required to shuffle a partition *i* to *R*. Clearly, as TND_R increases, more time is taken to shuffle *R*'s partitions and additional network bandwidth is dissipated. Fig. 1 lists the nodes at which each map task M_i and reduce task R_i of the wordcount benchmark[3] were scheduled by native Hadoop. In this case, every map task is feeding every reduce task. Besides, every map task is scheduled at a distinct node. Nodes 1-7 are encompassed in one rack and the rest in another rack. Hadoop schedules reduce tasks R_0, R_1, and R_2 at nodes 13, 12, and 3, respectively. This results in $\text{TND}_{R_0} = 30$, $\text{TND}_{R_1} = 32$, and $\text{TND}_{R_2} = 34$. If, however, R_1 and R_2 are scheduled at nodes 11 and 8, respectively, this would result in $\text{TND}_{R_1} = 30$ and $\text{TND}_{R_2} = 30$. Hadoop, in its present design, is incapable of making such controlled scheduling decisions.

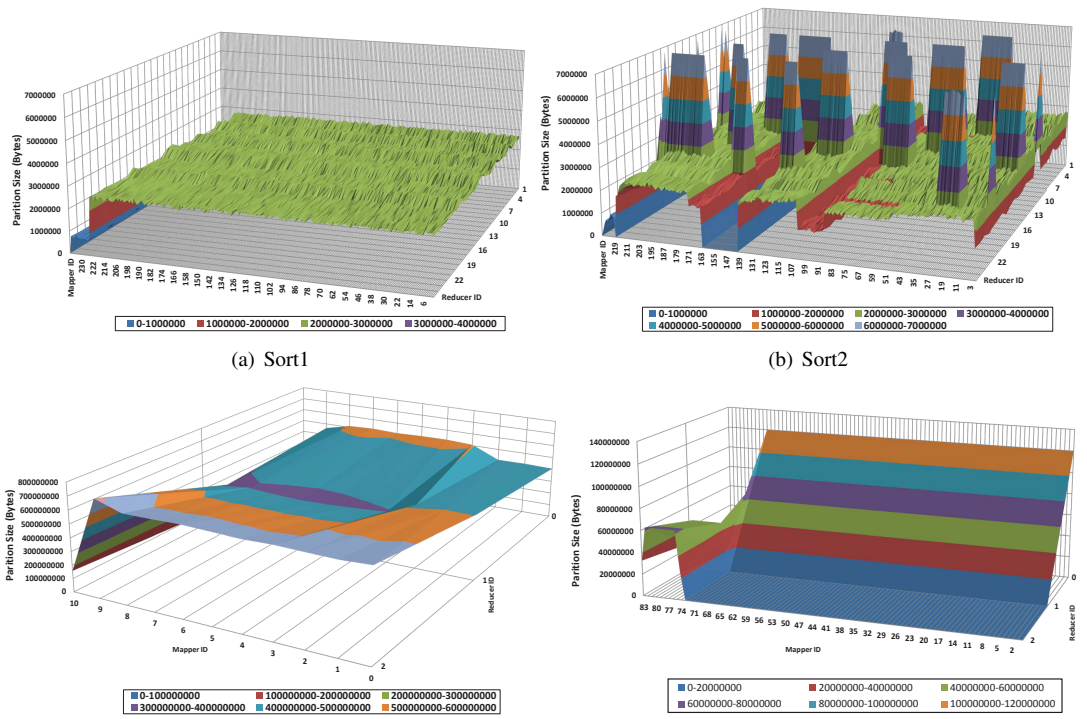
		Nodes													
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
Map Tasks	M0				✓										
	M1											✓			
	M2								✓						
	M3		✓												
	M4						✓								
	M5							✓							
	M6									✓					
	M7										✓				
	M8					✓									
	M9														✓
	M10													✓	
Reduce Tasks	R0													✓	
	R1												✓		
	R2			✓											

Figure 1. The nodes at which native Hadoop scheduled each map task and reduce task of the wordcount benchmark.

IV. Partitioning Skew in MapReduce Workloads

We now demonstrate the partitioning skew in each of our utilized benchmarks. Fig. 2 shows the sizes of partitions delivered by each feeding map task to each reduce task in sort1, sort2, wordcount and *K*-means workloads (see Table II for descriptions on these programs). Sort1 (Fig. 2(a)) utilizes a uniform dataset, hence, it exhibits a partitioning uniformity across reduce tasks, except for some partitions produced

[3]We ran wordcount five times and selected a random run.



(a) Sort1 (b) Sort2

(c) Wordcount (d) *K*-means

Figure 2. The sizes of partitions produced by each feeding map task to each reduce task in sort1, sort2, wordcount, and *K*-means.

by few map tasks. To explain this, the RandomWriter[4] generator, used by sort, generates files (not a single file) across cluster nodes. Each generated file at a node is always broken up into *fixed-size* chunks by HDFS, although the file size might not be a multiple of the chosen chunk size. As such, the content encapsulated in the last chunk of each file may contain less data than the other file chunks. Consequently, the map tasks that will process these smaller chunks will produce less intermediate data than the other map tasks that will process fully filled chunks.

In contrast to sort1, sort2 (Fig. 2(b)) uses a non-uniform dataset and reveals a significant discrepancy among partition sizes of reduce tasks. Clearly, this might lead to a noteworthy shuffle skew. For wordcount (Fig. 2(c)), we observe some skew across reduce tasks, where reduce task 1 receives less data than the other two reduce tasks. Finally, for *K*-means (Fig. 2(d)) we only portray the first job as a representative of the 5 jobs present in *K*-means (details about the workload can be found in Section VI-A). *K*-means exhibits a non-uniformity among partition sizes of reduce task 0 on one hand, and reduce tasks 1 and 2 on the other hand. Reduce task 0 receives most of its input data from most of the map tasks, while reduce tasks 1 and 2 consume most of their data from few map tasks. *K*-means behavior is contingent upon the selection of centroids as well as the adopted clustering mechanism. To conclude, we observe that partitioning skew exists in some MapReduce applications. Hadoop currently does not address such a phenomenon.

[4]RandomWriter is used in Hadoop to generate random numbers, usually for the sort benchmark program.

V. The CoGRS Task Scheduler

This section begins by first motivating the problem at hand and then delving into the core of CoGRS, the center-of-gravity idea. In order to integrate CoGRS into Hadoop, we further discuss the constraint for that (i.e., early shuffle) and suggest a solution (i.e., pseudo-asynchronous map and reduce phases). Lastly, we describe and formally present CoGRS's algorithm.

A. A Motivating Example

Fig. 3 demonstrates a data center with two racks each including 3 nodes. We assume a reduce task *R* with two feeding nodes, Task Tracker 1 (TT1) and Task Tracker 2 (TT2). The goal is to schedule *R* at a requesting Task Tracker. Assuming Task Trackers 1, 2, and 4 (i.e., TT1, TT2, and TT4) poll for a reduce task at the Job Tracker (JT). JT with the native Hadoop scheduler can assign *R* to any of the requesting Task Trackers. If *R* is assigned to TT4, TND_R will evaluate to 8. On the other hand, if *R* is assigned to TT1 or TT2, TND_R will be 2. As discussed earlier, a smaller total network distance is supposed to produce less network traffic and, accordingly, provide better performance.

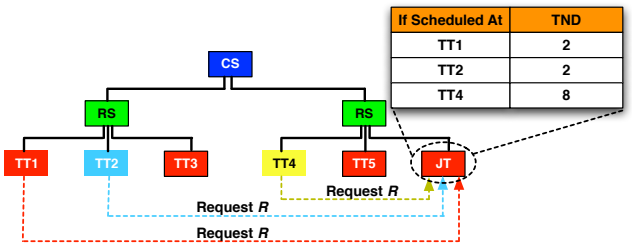


Figure 3. Options for scheduling a reduce task *R* with feeding nodes TT1 and TT2 in a data center with two racks (CS = core switch, RS = rack switch, TT = Task Tracker, and JT = Job Tracker).

Assume now that map tasks at TT1 and TT2 in Fig. 3 generate partitions of sizes 100MB and 20MB, respectively, both of which hash to *R*. If *R* is scheduled at TT2, 100MB of data will be shuffled on the network. On the other hand, if *R* is scheduled at TT1, only 20MB of data will be shuffled. Though both TT1 and TT2 provide the minimal possible TND_R, incorporating partitioning skew with locality places less burden on the network by decreasing the shuffled traffic. Therefore, a requirement emerges for a locality-aware and skew-aware reduce task scheduler capable of scheduling reduce tasks at suitable nodes that minimize total network distances and save MapReduce network traffic.

B. CoGRS Design

1) Center-Of-Gravity (COG) and Weighted Total Network Distance (WTND): To address data locality and partitioning skew, CoGRS attempts to place every reduce task, *R*, at a suitable node that minimizes TND_R. We propose that a suitable node for *R* would be the *center-of-gravity* (COG) node in accordance with the network locations of *R*'s feeding nodes and the *weights* of *R*'s partitions. We define the weight of a partition, *P*, needed by *R*, as the size of *P* divided by the total sizes of all *Ps* needed by *R*. As pointed out in the previous section, it is always desirable to shuffle smaller partitions over larger ones. We accomplish this objective by multiplying every *P's* weight by the network distance required to shuffle *P*. Thus, we introduce a new metric called *Weighted Total Network Distance* (WTND) and define it as follows:

Weighted Total Network Distance (WTND) Per Reduce Task R $= \sum_{i=0}^{n} ND_{iR} \times w_i$

where *n* is the number of partitions needed by *R*, *ND* is the network distance required to shuffle a partition *i* to *R*, and w_i is the weight of a partition *i*. In principle, the COG of *R* (COG_R) is always one of *R's* feeding nodes since it is less expensive to access data locally than to shuffle them over the network (this has been verified empirically as well). Therefore, we designate COG_R to be the feeding node of *R* that provides the *minimum* WTND.

2) Early Shuffle and Pseudo-Asynchronous Map and Reduce Phases: To determine the COG node of a particular reduce task, *R* (i.e., COG_R), we first need to designate the network locations of *R*'s feeding nodes. In essence, COG_R cannot be precisely determined until all the map tasks in *R*'s job commit. This is due to the fact that the last committing map task can still feed *R*. Waiting for all map tasks to commit until we can start computing COG_R (and subsequently trigger reduce task scheduling) might create some side effects on Hadoop's performance. Specifically, default Hadoop starts scheduling reduce tasks before every map task commits (by default after only 5% of map tasks commit). This allows Hadoop to overlap the execution of map tasks with the shuffling of intermediate data and, accordingly, enhance the turnaround times of MapReduce jobs. We refer to this technique as *early shuffle*. We evaluated several benchmarks on native Hadoop by turning early shuffle on and off (see Section VI). We found that Hadoop with early shuffle on outperforms Hadoop with early shuffle off by an average of 5.8%, and by up to 15.9%. This simply indicates the importance of early shuffle on the overall achieved Hadoop performance.

We suggest that to designate the network locations of *R*'s feeding nodes, we might not need to wait for all map tasks of *R*'s job to commit. This has been also suggested and discussed in [9]. That is, we might be capable of determining COG_R without deferring shuffling data to reduce tasks until the map phase is fully done. To describe that, assume, for instance, a map task, *M*, that feeds *only* one reduce task, *R*. In this case, *M* is confirmed to be a feeding map task of *R* after processing only the first input key-value pair and hashing it to *R*. Afterwards, *M* will not hash any data to any other reduce task. Therefore, in such a case we need not wait for *M* to commit before we can determine its consuming reduce task. On the other hand, if *M* feeds multiple reduce tasks, these reduce tasks can appear as consumers of *M* at an early or later period of *M*'s processing time. As *M* makes more progress, the probability for these reduce tasks to appear as consumers of *M* increases. However, a certain probability will remain for *all or some* of these reduce tasks to appear at an early period of *M*'s processing time. We rely on this fact about map tasks and promote computing COG_R after a percentage (not necessarily 5% as in default native Hadoop) of map tasks commit, and then start shuffling *R*'s required partitions. We denote this percentage as α and refer to such an approach as *pseudo-asynchronous* map and reduce phases approach.

Clearly, the value of α is a function of the given application as well as the distribution of the input dataset. In this work, we adopt a static determination of α per application (which we refer to as the *sweet spot* of an application). Locating sweet spots in a synergistic manner is beyond the scope of this paper and has been set as a

main future direction. To this end, we note that although CoGRS might delay the activation of early shuffle, such a delay assists in leveraging enhanced data locality as a result of allowing more accurate COG computations. Thus, CoGRS compensates for the case by seeking greater network bandwidth savings and potentially improving job execution times.

Algorithm 1 CoGRS Algorithm

Input: RT: set of unscheduled reduce tasks
TT: the task tracker requesting a reduce task
Output: A reduce task $R \in RT$ that can be scheduled to run on TT

initialize two sets of *potential* reduce tasks to schedule at TT, $set_{COG} = \Phi$ and $set_{Others} = \Phi$
for every reduce task $R \in RT$ **do**
  calculate center of gravity $COG_R = \min\{WTND_R\}$
  if $TT = COG_R$ **then**
    add R to set_{COG}
  else
    calculate progress score *PS* of $COG_R = \max\{progress\ scores\ of\ reduce\ task\ running\ at\ COG_R\}$
    if COG_R *is currently occupied* && *PS* $< \beta$ **then**
      add R to set_{Others}
    end if
  end if
end for
if set_{COG} is not empty **then**
  calculate most preferring reduce task $R_p = \max\{sizes\ of\ partitions\ held\ by\ COG_R\ for\ preferring\ reduce\ task\}$
  return R_p
else
  if set_{Others} is not empty **then**
    return reduce task $R \in set_{Others}$ whose COG node is the closest to TT
  end if
end if
return null

3) CoGRS Algorithm: We suggest that every reduce task *prefers* its COG node over any other node in the cluster. Hence, when a Task Tracker node (say TT_n) polls for a reduce task, the Job Tracker (JT) checks if any reduce task in the reduce task queue prefers TT_n. If JT finds that there are many reduce tasks who prefer TT_n, it selects the one that prefers TT_n the most. We define the most *preferring reduce task* as the one that consumes the largest input partition at TT_n so as to optimize for network traffic. On the other hand, if JT does not find any preferring reduce task, it can decide to reject TT_n and wait for another potential Task Tracker. If JT decides so, TT_n may stay idle though it has exposed its availability of running reduce tasks. Accordingly, poor cluster utilization might be sustained. In the meantime, some other Task Trackers may successfully receive multiple reduce tasks leading thereby to a scheduling skew. To address this problem, we propose not to reject TT_n and rather assign it a reduce task, *R*, whose COG node is currently *occupied* (i.e., does not have any available reduce slot) so as to avoid delaying *R* and improve cluster utilization (via delegating work to an otherwise idle TT_n). If multiple *R*s are found, we assign TT_n an *R* whose COG node is the *closest* to TT_n among the other found *R*s.

Giving up some locality for the sake of extracting more parallelism and improving cluster utilization should be done meticulously so as to avoid missing any locality opportunity. An opportunity that can be exploited is when an occupied COG node, *N*, will become available *shortly*. In such a case, scheduling a reduce task, *R*, that prefers *N* at a different requesting Task Tracker might not be a good idea. Therefore, we suggest checking the *progress score* of *N* and verify whether it will become available shortly before we delegate its work to any other nearby Task Tracker. Existing Hadoop monitors a task progress using a progress score between 0 and 1. We define the progress score of *N*, as the *maximum* progress score among the progress scores of the reduce tasks that are currently running at *N*. The reduce task that corresponds to the maximum progress score will supposedly finish earlier (and, hence, its slot will become available) than any other reduce task running at *N*, thus its score is selected. We compare the maximum progress score with a threshold β between 0 and 1. If we find the score smaller than β, we schedule *R* at a non-preferred nearby requesting Task Tracker. Otherwise, we do not. As β increases, less locality is achieved because more reduce tasks will be scheduled at non-preferred Task Trackers. In contrary, as β decreases, more locality is leveraged, but less parallelism is extracted. Therefore, in addition to α described in Section V-B2, β also contributes in locating applications' sweet spots. In summary, Algorithm. 1 formally describes CoGRS.

VI. Quantitative Evaluation

A. Methodology

Table I
Cluster configuration parameters

Category	Configuration
Hardware	
Chassis	IBM BladeCenter H
Number of Blades	14
Processors/Blade	2 x 2.5GHz Intel Xeon Quad Core (E5420)
RAM/Blade	8 GB RAM
Storage/Blade	2 x 300 GB SAS
	Defined as 600 GB RAID 0
Virtualization Platform	vSphere 4.1/ESXi 4.1
Software	
VM Parameters	4 vCPU, 4 GB RAM
	1 GB NIC
	60 GB Disk (mounted at /)
	450 GB Disk (mounted at /hadoop)
OS	64-Bit Fedora 13
JVM	Sun/Oracle JDK 1.6, Update 20
Hadoop	Apache Hadoop 0.20.2

We evaluate CoGRS against native Hadoop on our cloud computing infrastructure and on Amazon EC2 [1]. Our infrastructure is comprised of a dedicated 14 physical host IBM BladeCenter H with identical hardware, software and network capabilities. The BladeCenter is configured with the VMware vSphere 4.1 virtualization environment. VMware vSphere 4.1 [32] manages the overall system and VMware ESXi 4.1 runs as the blades' hypervisor. The vSphere system

was configured with a single virtual machine (VM) running on each BladeCenter blade. Each VM is configured with 4 v-CPUs and 4GBs of RAM. The disk storage for each VM is provided via two locally connected 300GB SAS disks. The major system software on each VM is 64-bit Fedora 13 [4], Apache Hadoop 0.20.2 [6] and Sun/Oracle's JDK 1.6 [16], Update 20. Table I summarizes our cloud hardware configuration and software parameters. To employ Hadoop's network topology, blades 1-7 are connected to a 1 gigabit switch, blades 8-14 to another 1 gigabit switch, and the two switches are connected to a third 1 gigabit switch providing a tree-style interconnectivity for all blades. All the switches are physical.

In contrary to our cloud, Amazon does not expose the network locations of EC2 instances required for enabling Hadoop rack-awareness [7]. To tackle this problem, we use the Netperf [22] benchmark and measure point-to-point TCP stream bandwidths between all pairs of EC2 instances in any Hadoop cluster we provision at Amazon. This allows us to estimate the relative locality of instances and arrive at a reasonable inference regarding the rack topology of a cluster. The data obtained is subsequently plugged into Hadoop to make it rack-aware.

Table II
BENCHMARK PROGRAMS

Benchmark	Key Frequency	Data Distribution	Dataset Size	Map Tasks	Reduce Tasks
sort1	Uniform	Uniform	14 GB	238	25
sort2	Non-Uniform	Non-Uniform	13.8 GB	228	25
wordcount	Real Log Files	Real Log Files	11 GB	11	3
K-means	Random	Random	5.2 GB	84	3

To evaluate CoGRS against native Hadoop, we use the sort and the wordcount (with the combiner function being enabled) benchmarks from the Hadoop distribution as well as the *K*-Means clustering workload from Apache Mahout [21], an open-source machine learning library. Sort and wordcount are two main benchmarks used for evaluating Hadoop at Yahoo! [3], [35]. Besides, sort was utilized in the MapReduce Google's paper [3]. *K*-Means is a well-known clustering algorithm for knowledge discovery and data mining [10].

To test CoGRS with various types of datasets, we ran sort over two datasets, one with uniform (the default) and another with non-uniform keys' frequencies and data distribution across nodes. We refer to sort running on a uniform dataset as *sort1*. To produce a non-uniform dataset we followed a similar approach as in [13]. We modified the RandomWriter in Hadoop to obtain a skew in keys' frequencies and data distribution with variances of 267% and 34%, respectively. We refer to sort running on this non-uniform dataset as *sort2*.

The *K*-means clustering workload implements *K*-means, an iterative algorithm that attempts to find *K* similar groups in a given dataset via minimizing a mean squared distance function. *K*-means assumes an initial list of *K* centroids and an input dataset of samples with each sample being represented as a *d*-dimensional vector. We generated a random dataset of 2D data points, selected 4 random centroids, and fixed that for all runs in our experiments. Furthermore, we set 5 iterative jobs similar to [10]. Finally, we applied wordcount to a set of real system log files. Table II illustrates our utilized benchmarks. To account for variances across runs we ran each benchmark 5 times.

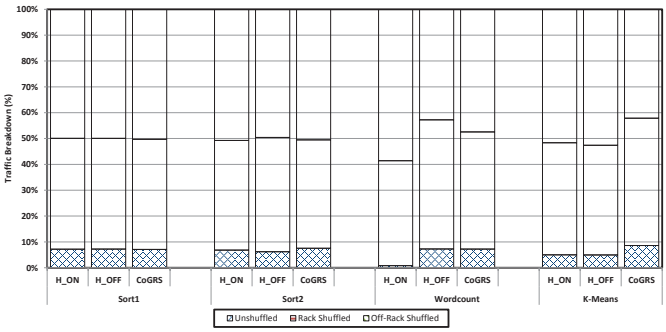
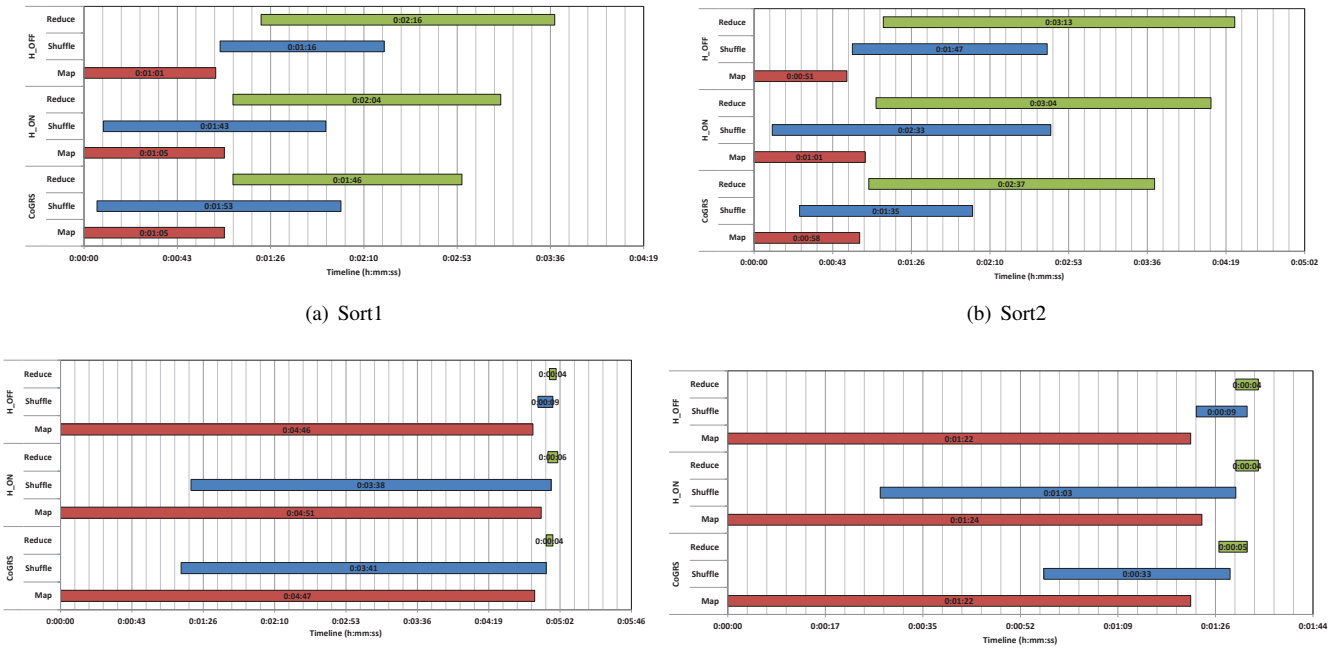


Figure 4. Reduce traffic experienced by Hadoop with early shuffle on (H_ON), Hadoop with early shuffle off (H_OFF), and CoGRS for sort1, sort2, wordcount, and *K*-means benchmarks.

B. Comparison with Native Hadoop

In this section, we run experiments on our private cloud. We evaluate CoGRS against native Hadoop with early shuffle on (H_ON) and off (H_OFF). Because CoGRS attempts to locate a sweet spot for an application by scanning a range of values through which it can activate early shuffle and still achieve good locality, we found it insightful to illustrate Hadoop's behavior with early shuffle being natively enabled (i.e., H_ON) and totally disabled (i.e., H_OFF). In addition, for each of our benchmarks, we conducted a sensitivity study through which we varied α over 7 values (i.e., $\{1.0, 0.9, 0.8, 0.6, 0.4, 0.2, 0.05\}$) and β over 4 (i.e., $\{1.0, 0.8, 0.4, 0.2\}$). In total we got 28 $\{alpha, \beta\}$ configurations. We located sweet spots for wordcount, sort1, sort2, and *K*-means at $\{0.05, 0.4\}$, $\{0.05, 0.2\}$, $\{0.4, 1.0\}$, and $\{0.4, 0.4\}$, respectively. In the following studies, we utilize these located sweet spots, unless otherwise specified.

1) Network Traffic: As CoGRS's main goal is to optimize for shuffled data, we first start by demonstrating the network traffic experienced by CoGRS as well as native Hadoop. Fig. 4 shows a breakdown of network traffic as entailed by H_ON, H_OFF, and CoGRS for all the benchmarks. Since we ran each benchmark 5 times, we depict the best results for each benchmark under each scheme. We modified Hadoop 0.20.2 to instrument reduce data and categorize it into *node-local*, *rack-local*, and *off-rack*. As displayed, H_OFF maximizes node-local and rack-local data by averages of 19.3% and 5.1%, respectively versus H_ON. Furthermore, H_OFF minimizes rack-local data by an average of 6.8%



(a) Sort1

(b) Sort2

(c) Wordcount

(d) *K*-means

Figure 5. Execution Timelines for sort1, sort2, wordcount, and *K*-means (Map = map phase, Shuffle = shuffle stage, and Reduce = reduce stage).

as compared to H_ON. Note that for optimizing network traffic we ought to maximize node-local, maximize/minimize rack-local, and minimize off-rack data. Rack-local data is maximized after only minimizing off-rack data. However, rack-local data is minimized or maximized after minimizing off-rack data and maximizing node-local data.

Overall, H_OFF surpasses H_ON with respect to shuffled data. This can be attributed mainly to the usage of a *resource estimator* by Hadoop. In particular, Hadoop adopts a resource estimator that estimates the input size of each reduce task, R, before R is assigned to a requesting Task Tracker, TT. If Hadoop finds that TT does not have enough space to run R, R is not scheduled at TT. With H_ON, map tasks and reduce tasks share the same system resources. Hence, the likelihood that Hadoop schedules reduce tasks at TTs that demonstrate more available system resources increases. For example, we observed that H_ON schedules every map task of the wordcount benchmark at a distinct node (it has in total 11 map tasks) and 1 of the reduce tasks at a feeding node (it has in total 3 reduce tasks). Conversely, we realized that H_OFF still schedules every map task of wordcount at a distinct node, but the 3 reduce tasks at 3 feeding nodes. As such, H_ON should incur more network traffic.

On the other hand, CoGRS is superior to both, H_ON and H_OFF. As shown in Fig. 4, on average, CoGRS maximizes node-local data by 34.5% and minimizes rack-local and off-rack data by 5.9% and 9.6% versus H_ON. Furthermore, on average, CoGRS maximizes node-local data by 14.3% and minimizes rack-local and off-rack data by 0.5% and 1.6% versus H_OFF. Clearly, this is because CoGRS attempts to schedule each reduce task at its COG node which is the feeding node that provides the minimum WTND (see Section V-B1 for details). For instance, we observed that CoGRS schedules the 11 map tasks of the wordcount benchmark on 11 distinct nodes and the 3 reduce tasks on 3 feeding nodes that provide the minimum WTND. We conclude that CoGRS successfully reduces network traffic.

2) Execution Timelines: In order to understand the behaviors of our applications, we now illustrate in Fig. 5 the overall times taken by the map phase as well as the shuffle and the reduce stages in the reduce phase for each of the benchmarks under CoGRS, H_ON , and H_OFF. We first note that the overall time taken by the shuffle stage of a workload includes *waiting times* as well as *shuffling times*. A shuffling time occurs whenever a data shuffling is going on. A waiting time occurs if no data shuffling is going on. A waiting time is incurred as a result of waiting for at least one partition to be available so that the shuffling process can be restarted. Second, we note that the overall time taken by the reduce stage of a workload also includes *waiting times*, as well as *reducing times*. A reducing time occurs whenever at least one reduce task is reducing its available input. As necessitated by Hadoop, the input to a reduce task does not become available for reduction unless it gets *fully* shuffled and sorted. A waiting time in the reduce stage occurs if no

data reduction is going on. As long as at least one reduce task is reducing its input data, no waiting time is incurred on the overall time taken by the reduce stage. Lastly, we note that the timer in the reduce stage is triggered exactly upon starting the first reduction process in the stage. Hence, it can be seen in Fig. 5 that the reduce bar always overlaps with the shuffle bar.

With that being said, let us proceed by comparing H_ON versus H_OFF. First it can be observed that H_OFF always shortens (by a little) the map phase when compared to H_ON for all benchmarks. This is because the map tasks under H_OFF do not share system resources with the reduce tasks (early shuffle is halted) and, accordingly, they finish faster. Second, H_OFF always shortens the shuffle stage because of avoiding any waiting time as compared to H_ON. Specifically, H_OFF needs not wait for any partition to become available due to the fact that all partitions will be available after the map phase is done. Third, H_OFF might extend (e.g., sort1, sort2, and *K*-means) or shrink (e.g., wordcount) the reduce stage time depending on when a certain reduce input will be available (i.e., waiting times might differ). The actual reducing times between H_ON and H_OFF are always comparable.

By comparing CoGRS against H_ON and H_OFF, we first notice that CoGRS also shortens (by a little) the map phase when compared to H_ON for all benchmarks. This is mainly because CoGRS decreases the pressure on system resources for the fact of exploiting data locality and addressing partitioning skew. Second, for the benchmarks that exhibit partitioning skew (i.e., sort2, wordcount, and *K*-means), CoGRS reduces the response/finish time of the shuffle stage versus H_ON and H_OFF. In essence, as partitioning skew becomes more significant (e.g., sort2 and *K*-means), CoGRS produces better results for being designed to intelligently address such a problem. When the problem disappears (e.g., sort1), CoGRS does not manifest its potential noticeably. Yet, for sort1 CoGRS shows a little worse finish time as compared to H_ON. This can be attributed to the overhead incurred by CoGRS due to constantly attempting to locate COG nodes. Finally, CoGRS might also extend (e.g., *K*-means) or shrink (e.g., sort1, sort2, and wordcount) the reduce stage time depending on when a certain reduce input will be available. The actual reducing times between CoGRS, H_ON and H_OFF are always comparable (as same data is essentially reduced), but the waiting times might differ. In summary, as compared to Hadoop, CoGRS is capable of improving shuffle and reduce times, especially when partitioning skew becomes significant.

3) Overall Performance: Fig. 6 shows the execution time results for all our benchmarks under CoGRS, H_ON, and H_OFF. We display the best, the worst, and the average results for each program (as each was run 5 times). First, when the gain attained by H_ON due to overlapping data shuffling with the map phase, offsets the loss caused by

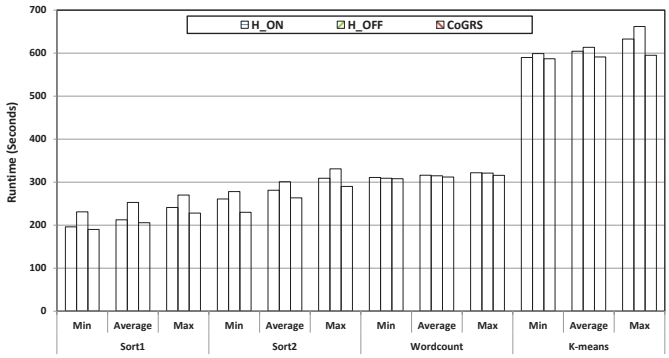


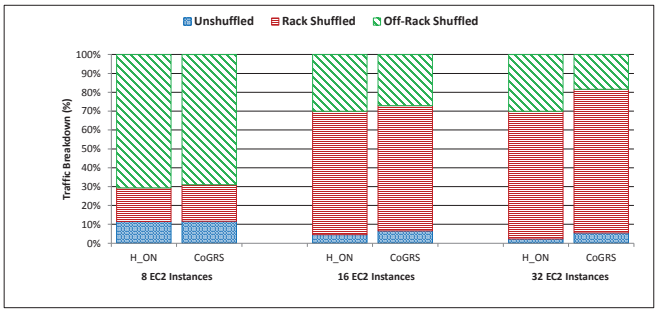
Figure 6. The execution times experienced by H_ON, H_OFF, and CoGRS for sort1, sort2, wordcount, and *K*-means benchmarks (Min, Max, and Average are the best, the worst, and the average-case execution times).

the potential increase in the amount of data shuffled, H_ON surpasses H_OFF (e.g., sort1). Otherwise, H_OFF outperforms H_ON (e.g., wordcount). Second, though CoGRS highly reduces the network traffic of wordcount as compared to H_ON and H_OFF (see Fig. 4), this does not greatly reflect on the overall performance. The reasons for this are as follows: (1) wordcount is a map-bound program (See Fig. 5(c)) while CoGRS mainly targets the reduce phase, (2) the actual shuffle stage time in wordcount (demonstrated by H_OFF. Again, see Fig. 5(c)) is very small in contrast to other phases (or stages) times, hence, an improvement (or degradation) in the shuffle stage time will not mirror visibly on the workload's overall performance, and (3) most of the shuffle stage time is overlapped with the map phase in H_ON, thus, not much room is left for CoGRS to optimize for performance. On average, CoGRS outperforms H_ON by 3.2% and by up to 6.3%, and H_OFF by 8.9% and by up to 12.4%.

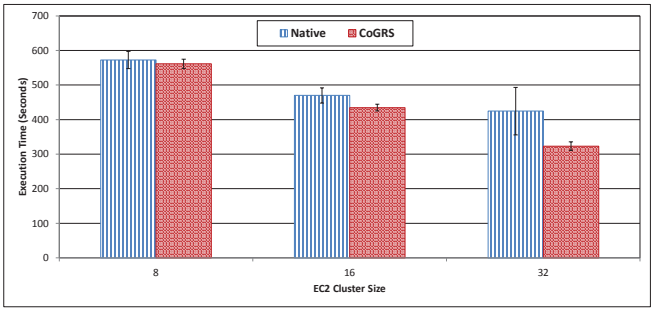
C. CoGRS on Amazon EC2

In this section, we conduct a scalability study of CoGRS and native Hadoop on a shared heterogeneous cloud environment using Amazon EC2. We provisioned three Hadoop clusters with 8, 16, and 32 nodes each (all nodes were configured using the *m1.small* instance) and experimented with native Hadoop as well as CoGRS on each of the clusters. We ran and collected results for one of our benchmarks, sort2, using a 10GB dataset size. For CoGRS, we conducted a sensitivity study and located sweet spots at $\{0.05, 1.0\}$, $\{0.2, 1.0\}$, and $\{0.6, 1.0\}$ on the 8-instance, 16-instance, and 32-instance clusters, respectively. We ran sort2 five times for each CoGRS configuration as well as for native Hadoop.

As the number of nodes increases, the probability for Hadoop to schedule a reduce task far from its center-of-gravity *node* increases. In contrary, CoGRS adopts a more informed strategy and attempts constantly to schedule reduce



(a) Reduce Network Traffic



(b) Execution Time

Figure 7. Reduce traffic and execution times experienced by CoGRS and native Hadoop (H_ON) for sort2 on Amazon EC2 clusters of sizes 8, 16, and 32.

tasks at their center-of-gravity nodes irrespective of the cluster size. Hence, CoGRS is expected to optimize more on cumulative data center network traffic and, consequently, scale better than Hadoop. Fig. 7(a) demonstrates increased savings in network traffic under CoGRS versus native Hadoop as the EC2 cluster is scaled up. Compared to native Hadoop, on average, CoGRS maximizes node-local data by 1%, 32%, and 57.9%, maximizes rack-local data by 7.8%, 1.3%, and 11.1%, and minimizes off-rack data by 2.3%, 10%, and 38.6% with 8-instance, 16-instance, and 32-instance clusters, respectively. As depicted in Fig. 7(b), this translates to 1.9%, 7.4%, and 23.8% average reductions in job execution times under CoGRS versus native Hadoop with 8-instance, 16-instance, and 32-instance clusters, respectively. We conclude that on Amazon EC2, CoGRS successfully reduces network traffic and scales better than native Hadoop. In summary, we found that for our utilized benchmarks, CoGRS always reduces network traffic and improves MapReduce performance, on a dedicated homogenous cluster and on a shared heterogeneous cloud.

VII. Related Work

In this short article, it is not possible to do justice to every related scheme. As such, we only outline few of closely related papers. To start with, we view Hadoop scheduling as a two-level model and classify related work accordingly. At the first level, *job scheduling* (e.g., Hadoop FIFO, Fair and Capacity schedulers) deals with allocating resources to co-located jobs. At the second level, *task scheduling* defines how map and reduce tasks are actually assigned to map and reduce slots (e.g., Hadoop attempts to assign map tasks to map slots in proximity to corresponding splits). In the presence of multiple jobs, a job scheduler is needed, while a task scheduler is required even with only one running job. CoGRS is a task scheduler, and akin to any other task scheduler, is complementary to whichever job scheduler.

Ussop [2] employs MapReduce on public-resource grids and suggests variable-size map tasks. LARTS [9] attempts to collocate reduce tasks with the maximum required data. LATE [35] proposes a scheduling algorithm for speculative tasks robust to heterogeneity. HPMR [30] suggests inspecting input splits in the map phase, and predicts to which reduce task key-value pairs will be partitioned. The expected data are assigned to a map task near the future reduce task. Quincy [15] proposes a novel data flow graph-based framework constructed on Dryad [14]. LEEN [13] reports on the partitioning skew problem and promotes altering Hadoop's existing hash partitioning function in order to alleviate the amount of data shuffled over the network.

While all of the above papers are proposals for task schedulers, the following are examples of job schedulers. CBHS [17] puts an emphasis on meeting jobs' deadlines via scheduling only jobs that can meet estimated deadlines. DP [29] capitalizes on the existing Hadoop FIFO and fair-share schedulers and applies a proportional share resource allocation mechanism. Polo *et al.* [24] build upon the Adaptive scheduler [25] and proposes a scheduler that can make hardware-aware scheduling decisions and minimize jobs' completion times. HFS [34] suggests a job scheduling algorithm based on waiting so as to achieve fairness and data locality (for map tasks only). FLEX [33] extends HFS and optimizes towards a variety of standard scheduling metrics (e.g., deadline-based penalty functions). Phan *et al.* [23] explore the feasibility of enabling real-time scheduling of MapReduce jobs. Lastly, Tian *et al.* [31] suggest the Triple-Queue scheduler that seeks better system utilization via scheduling jobs based on (predicted) resource usage.

To that end, a large body of work has also reported on the partitioning skew problem. For instance, SkewReduce [18] scrutinized skew in feature extraction scientific applications and promotes a new system for expressing feature extraction on Hadoop. Kwon *et al.* [19] describe various causes of skew and suggest some practices for avoiding its negative impact. Ekanayake *et al.* [26] recognized skew in bioinformatics applications and analyzed its influence on scheduling mechanisms. Finally, Lin [20] illustrates the Zipfian distribution of intermediate output and proposes a theoretical model that shows how such distributions can impose a fundamental limit on extracting parallelism.

VIII. Concluding Remarks and Future Directions

In this work, we have observed that the network load is of special concern with MapReduce as a large amount of traffic can be generated during the shuffle phase, potentially causing performance deterioration. We realized that scheduling reduce tasks at their *center-of-gravity* nodes has a positive effect on Hadoop's network traffic as well as performance. Specifically, average reductions of 9.6% and 38.6% of off-rack network traffic have been accomplished on a private cloud and on an Amazon EC2 cluster, respectively. This provided Hadoop a performance improvement of up to 23.8%.

After verifying the promise of CoGRS, we set forth two main future directions. First, our suggested pseudo-asynchronous strategy can be altered to involve dynamic rather than static determination of sweet spots. Finally, we expect CoGRS to play a role in MapReduce for the type of scientific applications examined in [18], [26].

References

[1] Amazon Elastic Compute Cloud, "http://aws.amazon.com/ec2/."

[2] P. C. Chen, Y. L. Su, J. B. Chang, and C. K. Shieh, "Variable-Sized Map and Locality-Aware Reduce on Public-Resource Grids," *GPC,* 2010.

[3] J. Dean and S. Ghemawat, "Mapreduce: Simplified Data Processing On Large Clusters," *OSDI,* 2004.

[4] Fedora, "http://fedoraproject.org/."

[5] S. Ghemawat, H. Gobioff, and S. T. Leung, "The Google File System," *SOSP,* 2003.

[6] Hadoop, "http://hadoop.apache.org/."

[7] Tom White, "Hadoop: The Definitive Guide" *1st Edition, O'REILLY,* 2009.

[8] Hadoop Tutorial, "http://developer.yahoo.com/hadoop/tutorial/."

[9] M. Hammoud and M. Sakr, "Locality-Aware Reduce Task Scheduling for MapReduce," *CloudCom,* 2010.

[10] S. Huang, J. Huang, J. Dai, T. Xie, B. Huang, "The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis," *ICDEW,* 2010.

[11] B. He, W. Fang, Q. Luo, N.K. Govindaraju, T. Wang, "Mars: a MapReduce Framework on Graphics Processors," *PACT,* 2008.

[12] S. Ibrahim, H. Jin, B. Cheng, H. Cao, S. Wu, L. Qi, "CLOUDLET: Towards Mapreduce Implementation on Virtual Machines," *HPDC,* 2009.

[13] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi, "LEEN: Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud," *CloudCom,* 2010.

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," *EuroSys,* 2007.

[15] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair Scheduling for Distributed Computing Clusters," *SOSP,* 2009.

[16] JDK, "http://download.oracle.com/javase/6/docs/."

[17] K. Kc and K. Anyanwu, "Scheduling Hadoop Jobs to Meet Deadlines," *CloudCom,* 2010.

[18] Y. C. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skew-Resistant Parallel Processing of Feature-Extracting Scientific User-Defined Functions," *SOCC,* 2010.

[19] Y. C. Kwon, M. Balazinska, B. Howe, and J. Rolia, "A Study of Skew in MapReduce Applications," *Open Cirrus Summit,* 2011.

[20] J. Lin, "The Curse of Zipf and Limits to Parallelization: A Look at the Stragglers Problem in MapReduce," *LSDS-IR,* 2009.

[21] Mahout Homepage, "http://mahout.apache.org/."

[22] Netperf, "http://www.netperf.org/."

[23] L.T.X. Phan, Z. Zhang, B.T. Loo, and I. Lee, "Real-Time MapReduce Scheduling," *Tech. R No. MS-CIS-10-32, UPenn,* 2010.

[24] J. Polo, D. Carrera, Y. Becerra, V. Beltran, J. Torres, and E. Ayguadé, "Performance Management of Accelerated Mapreduce Workloads in Heterogeneous Clusters," *ICPP,* 2010.

[25] J. Polo, D. Carrera, Y. Becerra, M. Steinder, and I. Whalley, "Performance-Driven Task Co-Scheduling for MapReduce Environments," *NOMS,* 2010.

[26] X. Qiu, J. Ekanayake, S. Beason, T. Gunarathne, G. Fox, R. Barga, and D. Gannon, "Cloud Technologies for Bioinformatics Applications," *MTAGS,* 2009.

[27] M. Rafique, B. Rose, A. Butt, and D. Nikolopoulos, "Supporting MapReduce on Large-Scale Asymmetric Multi-Core Clusters," *SIGOPS Operating Systems Review 43,* 2009.

[28] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis, "MapReduce for Multi-Core and Multiprocessor Systems," *HPCA,* 2007.

[29] T. Sandholm and K. Lai, "Dynamic Proportional Share Scheduling in Hadoop," *Workshop on Job Scheduling Strategies for Parallel Processing,* 2010.

[30] S. Seo, I. Jang, K. Woo, I. Kim, J. Kim, S. Maeng, "HPMR: Prefetching and Pre-Shuffling in Shared MapReduce Computation Environment," *CLUSTER,* 2009.

[31] C. Tian, H. Zhou, Y. He, and L. Zha, "A Dynamic MapReduce Scheduler for Heterogeneous Workloads," *GCC,* 2009.

[32] VMware, "http://www.vmware.com/."

[33] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.L. Wu, and A. Balmin, "FLEX: A Slot Allocation Scheduling Optimizer for MapReduce Workloads," *Middleware,* 2010.

[34] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling," *EuroSys,* 2010.

[35] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, I. Stoica, "Improving Mapreduce Performance in Heterogeneous Environments," *OSDI,* 2008.