Smart Shuffling in MapReduce: a solution to Balance Network Traffic and Workloads

Wei Shi
Faculty of Business and IT
University of Ontario Institute of Technology
Oshawa, Ontario L1H7K4
Email: wei.shi@uoit.ca

Yang Wang
Shenzhen Inistitute of Advanced Technology
Chinese Academy of Science
Shenzhen, P. R. China
Email: yang.wang1@siat.ac.cn

Jean-Pierre Corriveau
School of Computer Science
Carleton University
Ottawa, Ontario, K1S5B6
Email: jeanpier@scs.carleton.ca

Boqiang Niu
Faculty of Business and IT
University of Ontario Institute of Technology
Oshawa, Ontario L1H7K4
Email: boqiang.niu@uoit.ca

William Lee Croft
School of Computer Science
Carleton University
Ottawa, Ontario, K1S5B6
Email: leecroft@cmail.carleton.ca

Mengfei Peng
Faculty of Business and IT
University of Ontario Institute of Technology
Oshawa, Ontario, L1H7K4
Email: mengfei.peng@uoit.ca

***Abstract*—In the context of Hadoop, recent studies show that the shuffle operation accounts for as much as a third of the completion time of a MapReduce job. Consequently, the shuffle phase constitutes a crucial aspect of the scheduling of such jobs.**

During a shuffle phase, the job scheduler assigns *reduce tasks* to a set of *reduce nodes*. This may require multiple intermediate data items which share a key to be relocated to this new set of reduce nodes. In turn, this could cause a large volume of simultaneous data relocations within the network. Intuitively, a reduce task experiences shorter access latency if its required items are available locally or in close proximity. This, however, may also result in a hotspot in the network due to imbalanced traffic, as well as the imbalance of the workload on different nodes, regardless of their homogeneity.

In this paper, we study data relocation incurred during the shuffle stage in the MapReduce framework. Within an arbitrary network, we aim at a) minimizing the overall network traffic, b) achieving workload balancing, and c) eliminating network hotspots, in order to improve the overall performance. Our contribution consists of the development of a scheduler that satisfies these three goals. We then present an in-depth simulation. Our results show that, for arbitrary network topologies, our Smart Shuffling Scheduler systematically outperforms the CoGRS scheduler in terms of hotspot elimination as well as reduce task load balancing, while ensuring traffic caused by data relocation is low. Not only does our algorithm handle any topology but also its benefits are inversely proportional to the inter-node connectivity of the network topology: the lower this connectivity, the better our algorithm. In particular, for the tree topology commonly used within data centres, our proposed scheduler offers significant improvements over the CoGRS scheduler.

I. Introduction

The International Data Corporation estimates that, by 2020, the digital universe will grow up to 4×10^{22} bytes. Big Data Analytics and clouds are energizing organizations across diverse industries in that they present an enormous opportunity to make these organizations more agile, more efficient and more competitive. Improving the infrastructure that enables analytics through an architecture optimized for big data is a most pressing and challenging concern for computer science researchers.

Hadoop MapReduce is a software framework for easily writing applications which process big data in-parallel on large scale clusters of commodity hardware in a reliable manner. It has become the *de facto* research prototype on which many studies are conducted. The MapReduce framework was first advocated by Google in 2004 as a programming model for its internal massive data processing [8]. Since then it has been widely discussed and accepted as the most popular paradigm for data intensive processing in different contexts, e.g. Halim et al. propose an improvement for the maximum flow algorithm with the help of MapReduce framework. This improvement rests on the fact that increased graph size have greatly comprised the efficient of memory-resident algorithms. Although the new algorithm has quadratic runtime complexity, it is able to show the ability to compute the large real-world graphs in an efficient way [14]. Therefore there are many implementations of this framework in both industry and academia (such as Hadoop [1], Dryad [17], Greenplum [2]), each with its own strengths and weaknesses. We thus use the terminology of the Hadoop community in the rest of this paper, and focus here mostly on the related works built using Hadoop implementations.

A. Background

From an abstract viewpoint, a MapReduce job essentially consists of two sets of tasks: map tasks and reduce tasks, as shown in Figure 1. The executions of both sets of tasks are synchronized into a map stage followed by a reduce stage. In the map stage, the entire dataset is partitioned into smaller chunks in forms of key-value pairs, each chunk being assigned to a map slot for partial computation results. The map stage ends up with a set of intermediate key-value pairs on each map node that may have several map slots available. Each of a key-value pair is further shuffled based on the intermediate keys

978-0-7695-5697-0/15 $31.00 © 2015 IEEE
DOI 10.1109/UCC.2015.18

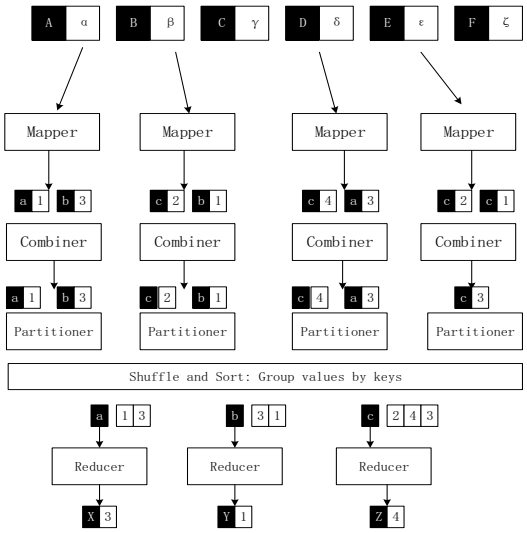

Fig. 1. MapReduce framework.

into a set of scheduled reduce slots on reduce nodes where the received pairs are aggregated to obtain the final results. For an iterative MapReduce job, the final results could be tentative and further partitioned and shuffled into a new set of map slots for the next round of the computation.

Hadoop MapReduce is made up of an execution runtime and a distributed file system. The execution runtime is responsible for job scheduling and execution. It is governed by one master node called *JobTracker* and multiple slave nodes called *TaskTrackers*. The distributed file system, referred to as *HDFS*, is used to store and transfer input and output data of jobs that are being processed across nodes. When a JobTracker receives a submitted job, it first splits the job into a number of map and reduce tasks and then put them into the map queue and the reduce queue, respectively. The existing job schedulers schedule map and reduce tasks separately and distribute the results to TaskTrackers that are on disjoint map and reduce nodes. As with most distributed systems, the performance of the task scheduler greatly affects the execution time of each specific job.

Hadoop MapReduce provides a FIFO-based default scheduler at job level [9], while at task level, it offers developers a TaskScheduler interface to allow customer designed schedulers. By default, each job will use the whole set of nodes in a cluster and execute the jobs in the order of submission. In order to overcome this inadequate strategy and share fairly the cluster nodes among jobs, users over time and data locality [13], Facebook and Yahoo! leveraged the interface to implement Fair Scheduler [3], [26], Capacity Scheduler [4] and Delay Scheduler [29] respectively.

Given the dependencies between different phases of a MapReduce job, MapReduce framework has presented several significant challenges in performance optimization. One of them is the shuffle operation which assumes simultaneous possession of multiple resources to transfer the intermediate results of the map phase to the processors performing the reduce tasks. Recent studies by Chowdhury et al. [7] have shown that the shuffle operation accounts for as much as a third of the completion time of a MapReduce job. As such, the shuffle phase has to be taken into consideration carefully in the scheduling problem.

During a shuffle phase, job scheduler assign reduce tasks to a set of reduce nodes. This may request multiple intermediate data items (with the same key-value) to be relocated to this new set of reduce nodes. This could cause a large volume of data relocation in the network at the same time. Intuitively, a reduce task would experience shorter access latency if required items were placed locally or in its closer proximity, but it can also result in hotspot (i.e., overloaded links) in the network due to imbalanced traffic, as well as the imbalance of the workload on different nodes, regardless of their homogeneity. Therefore, scheduling the reduce tasks to appropriate machine would minimize the total network traffics (globally and locally), balance the workload and hence improve the overall performance. It is well known that all data centres today have very symmetric topologies, yet, having a single solution that can handle various symmetric network topologies without requiring the knowledge of the topology a priori constitutes a major improvement over the existing solutions that are targeting specific topologies.

B. Our Contribution

In this paper, we address this problem by carefully scheduling the reduce tasks in appropriate locations to reduce the network hotspots and balance the node workload while minimizing the overall traffic. We introduce a Smart Shuffling Scheduler in order to satisfy these three goals. Our in-depth simulation results show that, for arbitrary network topologies, our Smart Shuffling Scheduler systematically outperforms the Random and CoGRS scheduler in terms of hotspot elimination as well as reduce task load balancing, while ensuring traffic caused by data relocation is low. Not only does our algorithm handle any topology but also its benefits are inversely proportional to the inter-node connectivity of the network topology: our algorithm performs the best in tree topology that is commonly adopted in data centres.

II. Related Work

Chen et al. [6] consider the problem of jointly scheduling all three phases of the MapReduce process with a view of understanding the theoretical complexity of the joint scheduling. The value of the research is the guaranteed approximation algorithms and several heuristics as well to solve the joint scheduling problem. Although result allow us to gain deep insight of this problem, the model in the shuffle phase is simplified for easy analysis, and not fully practical to manifest the reality.

Many efforts to optimize the shuffle phase are mainly either on data placements [5], [10], [27] or on the reduce task scheduling [15], [24], [25], both with a goal to improve the

data locality whereby reducing the amount of data that needs to be moved in a shuffle phase. However, in the former case, focus is put on MapReduce in heterogeneous environments where the compute nodes are typical virtual machines connected via an overlay network. For example, in [27], Xie et al. address the problem of how to place data across nodes in such a way that each node in a heterogeneous environment has a balanced data processing load. In [10], Eltabakh et. al introduce *CoHadoop*, a lightweight extension of Hadoop that allows applications to control where data are stored, and furthermore, leverage the hints from the applications to co-locate the related files in order to improve the efficiency of the garget network. In both cases as well as algorithm presented in [21], the environment on which the MapReduce framework is deployed is quite similar to ours as the compute nodes in the virtual cluster nodes may be connected in an arbitrary way.

The latter case usually optimize the scheduling of reduce tasks by maximizing the local data accesses. A typical example is the Center-of-Gravity Reduce Scheduler (CoGRS), a locality-aware and skew-aware reduce task scheduler, proposed by Hammoud et al. [15], [16], concerning 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. Although CoGRS can minimize the overall network traffic, it is not always useful to address the hotspot problem on network links and node workload balancing problem due to the heterogeneous network traffic in the shuffle phase, which is specifically the problems that we aim to solve in this paper. Tan et al. [25] also consider the optimization of reduce task scheduling. They extend the idea of CoGRS to their Waiting Scheduler, which couples the progresses of both map and reduce tasks jointly to alleviate starvation, and optimizes the data locality. Significant improvements in job response times is demonstrated through experiments.

Xu et al. [28] study the severity of network hotspots in racks when the network is shared among various MapReduce applications. To deal with this issue, they develop a model to analyze the relationship between job completion time and the assignment of both map and reduce tasks across racks. They further design a network-aware task assignment strategy to shorten the completion time of MapReduce jobs in shared cluster nodes. Our proposed research focus on reducing the network hotspots in shuffle phase while minimizing the network traffic and balance the workload of nodes in clusters. More importantly, we consider the underlying networks in a more general form, instead of a simple tree topology often adopted within a data center.

There are several other related works on network-aware scheduling of MapReduce jobs [5], [18], [23], but none of them focus on reducing network hotspots while minimizing the network traffic while balancing the node workload. For example, Kondikoppa et. al [18] add network awareness in Hadoop to help place the map tasks close to is data splits over federated clusters. Similarly, Arasnal et al. [5] propose several enhancements to data placing algorithms in Hadoop such that the load is distributed across the nodes evenly, a goal also pursued in [27]. In [23], Qin et al. present a heuristic bandwidth-aware task scheduler BASS. In [19], Lai et al. introduce a scheduling algorithms are proposed with a constant approximation bound to balance the server workloads and, at the same time to meet the response time requirements of MapReduce jobs

Unlike aforementioned research, Li et al. [20] adopt a different way to address various computing constraints other than the network bandwidth. To this end, they propose, CAM, a cloud platform that provides an innovative resource scheduler particularly designed for hosting MapReduce applications in the cloud.

Palanisamy *et al.* [22] presents Purlieus, a resource allocation system. The work considers data placement and Virtual Machine placement through three criteria: Job specific locality-awareness, load awareness and Job-specific data replication. However, those criteria are established under many conditions. Such as, the job specific locality-awareness needs typical data as a benchmark or monitors the execution of a job, which is not always easily available. Load awareness has similar problem. Moreover, the performance of the solution rests on the appropriateness of the classification of the Job specific locality-awareness. Unfortunately, according to the definition of Job specific locality-awareness: reduce-input heavy and map-and-reduce-input heavy are often hard to classify among many similar jobs.

In [15], Hammoud et al. suggest to place every reduce task at "centrality" of intermediate data to minimize the network traffics whereby they propose *Center-of-gravity reduce task scheduling* algorithm.

The proposed solution has two unsolved problems. First, the selected target node may not have available reduce slot for the scheduled reducer. Second, the network contention incurred by the data transfers is not considered. Due to the unpredictable locations of the intermediate files as well as their partition skews, it is highly possible for the network traffic to have an uneven distribution, which would introduce hotpots in the network, and thus low the network resource utilization, compromising the benefits of the CoGRS algorithm. An illustrative example is shown in Figure 2 where the map results (only keys are shown) are generated at 7 nodes and among those with key=3 are shuffled to the selected node for the reduce task.

In this example, given the shortest path routing, all key 3s except the one at E are passed over link $(S2, C)$ to node C, which would become a hotspot to degrade the network performance. To remove or mitigate this phenomenon, in spite of extra one hop, selecting the route over switches $S3$ and $S4$ to node C for key 3 at node B would be a better solution.

Tan et al. notice this first problem. The improved algorithms presented in [25] has been used in their *Waiting Scheduling* for reduce task scheduling with data locality. To our best knowledge, no one so far has studied the second problem.

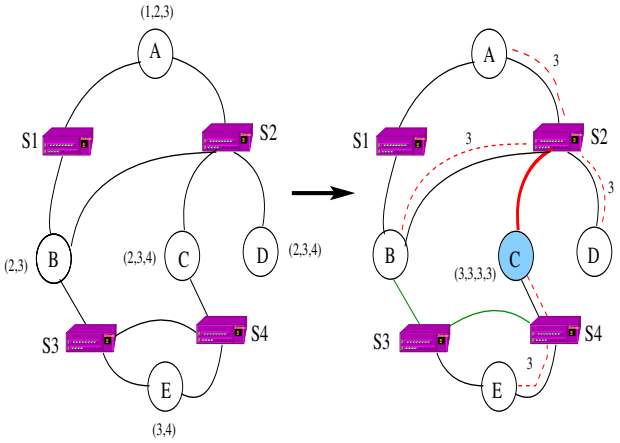

Fig. 2. An example of hotspots in a network. The intermediate key-value pairs are generated by map tasks at 5 nodes that are connected by 4 switches ($S1--S4$), each only showing the keys in parentheses (left sub-graph). The right sub-graph shows how key 3 is shuffled from nodes A, B, C, D and E to the selected reduce task at C according to the shortest path routing (dashed red lines), and how link $(S2, C)$ becomes a hotspot link (bold red line).

III. Model, Assumption and Problem Definition

Let $G(E, V)$ denote an arbitrary network topology, where a link $(u, v) \in E$ represents a network link and $u, v \in V$ represent network nodes. $|E|$=e is the number of edges and $|V| = n$ is the number of nodes in G. A *Task Tracker* (TT) is available on each $u \in V$ sending heartbeats to a central *Job Tracker* J. Let (TT_1, TT_2, $\cdots$, TT_Δ) represent a list of nodes/job trackers that has at least one available slot that can process a Reduce Task.

There are m Reduce Tasks, $RT_1 \cdots RT_m$, each of which is waiting to be processed on its destination node D_{RT_x} ($1 \leqslant x \leqslant m$) during the Reduce phase. In order to perform each such reduce task RT_x, data need to be transferred from a list of feeding nodes to the candidate destination node D_{RT_x} . Let RT_xF_y denote the y^{th} feeding node of RT_x, where $0 < y \leqslant n$. For each such feeding node RT_xF_y of RT_x, let $\omega(RT_xF_y)$ denote its Partition Size, that is the intermediate outputs from a feeding node after the Map phase. Z_x denotes the number of feeding nodes of RT_x, where $1 \leqslant x \leqslant m$.

Let $\psi(RT_xF_y, D_{RT_x})$ and $d(\psi(RT_xF_y, DRT_x))$ denote a shortest path between the y^{th} feeding node of RT_x and its destination node D_{RT_x} $(1 \leqslant x \leqslant m)$ and the length (number of hops) of this path respectively.

We define the *Actual Weight* (AW) of a link $\mathcal{L} = (u, v) \in E$ as the following:

$$W(u,v) = \begin{cases} \sum_{i=1}^{x}\sum_{i=1}^{y} 1 & \text{if } (u,v) \in \psi(RT_xF_y, D_{RT_x}) \\ & (1 \leqslant x < m \text{ and } 0 < y \leqslant n) \\ 0 & \text{otherwise} \end{cases}$$

In order to minimize the network traffic, we always try to send a package of intermediate data from its feeding node to the *destination node* (i.e. the ideal node that will process the reduce task during reduce phase) of its reduce task through a shortest path. We call each of such path an *Intermediate Data Relocation Path*. We say a link $\mathcal{L}$'s actual weight is 1 if $\mathcal{L}$ is on such a shortest path. Obviously, $\mathcal{L}$ may not only be on several intermediate data relocation paths of a specific reduce task, but also on the intermediate data relocation paths of multiple reduce tasks. The actual weight of link $\mathcal{L}$ is the total number of times all intermediate data relocation paths of all reduce tasks that passes through $\mathcal{L}$.

$$\alpha = \sum_{i=1}^{x} Z_x/e$$

α represents an average weight of a link $\mathcal{L}$, that is the total number of intermediate data relocation paths of all reduce tasks in the entire network over the total number of links on these paths.

Similarly, we define the *Actual Workload* (AL) of a node $u \in V$ as follows:

$$Ł(u) = \begin{cases} \sum_{i=1}^{x}\sum_{i=1}^{y} \omega(RT_xF_y) & \text{if } u \text{ is the } D_{RT_x} \text{ of } RT_x \\ & (1 \leqslant x \leqslant m, 1 \leqslant y \leqslant n) \\ 0 & \text{otherwise} \end{cases}$$

Theoretically, node u can be the destination node of several reduce tasks depending on the available reduce slot on it. Hence, we calculate the total size of intermediate data of all reduce tasks that has node u as its destination node. In other words, if we call $Ł(RT_x)$, the sum of the partition sizes of all feeding nodes of each specific reduce task RT_x the *total intermediate data items*, then the actual workload of node u ALu is the sum of total intermediate data items of all reduce tasks that has u as its (candidate) destination node.

$$\beta = \sum_{i=1}^{\Delta} Ł(RT_i)/\Delta$$

β represents the average workload per node, that is the ratio between the total intermediate data items of all reduce tasks in a Job Tracker and Δ, that is the number of nodes that has a reduce task slot available and has sent a request to the Job Tracker.

The scheduler that we propose aim at finding a set of nodes as targets, on which to process reduce tasks in job i that satisfies the following requirements:

Min $(aA * bB * cC)$, where

$$A = \max|W(u,v) - \alpha|$$

$$B = \max|\sum_{x=1}^{m} Ł(RT_x) - \beta|$$

$$C = \sum_{i=1}^{x}\sum_{i=1}^{y} d(\psi(RT_xF_y, D_{RT_x})) * \omega(RT_xF_y)$$

A represents the largest link weight difference, B is used to represent the biggest node load difference and$C$ is used to calculate the overall intermediate data items that need to be relocated from any feeding node to a destination node of all reduce tasks registered in the Job Tracker. We explain the details in the next section.

IV. Algorithm *Smart Shuffling*

A. General Description

Our proposed scheduler aim at suggesting a list of destination nodes, referred to as *Candidate Group* in our algorithm description, each of which process a reduce task of a specific job registered on the Job Tracker. This scheduler can be easily adjusted to the meet the needs of the customers who may either value the node workload balance more than the existence of the network hotspot or the overall intermediate data relocation level, or prefer to eliminate the network hotspot or value the data locality more than the two other facts which focus on reducing the overall job processing time. We use a, b, c, $(0 < a, b, c \leq 1)$ as the weight of:

- A: the largest link weight difference, that is the difference between 1). the actual weight of a link that appears on at least one Intermediate Data Relocation Path of any reduce task and 2). the average weight of a link $\mathcal{L}$ in the network; and
- B: the biggest node load difference, that is the difference between 1). the actual workload of a candidate node, on which there is at least one reduce task that will be scheduled, and 2). the average amount of intermediate data of all reduce tasks being spread to the nodes that has a reduce task slot available and has sent a request to the Job Tracker; and
- C: the overall intermediate data size, that is the sum of all the intermediate data items of all reduce tasks registered in the Job Tracker multiply the distance that each data item needs to travel in order to arrive at its reduce task's destination node.

respectively.

Assuming that we value each of these three aspects: link hotspot elimination, node workload balance and data locality equality, we will choose the set of candidate nodes that returns the minimal values of A, B and C. Changing the coefficients a, b and c will allow us to change the weights of each aspects that changes the overall performance accordingly. Such setup clearly indicates the tradeoffs between these three above-mentioned performance impact factors (see Algorithm 1 for more detail).

In order to maximize the data locality and minimize the data relocation caused during Shuffle stage, we first incorporate the idea proposed by Hammoud et al. [15]. Hammoud et al. pointed out in [15]: in principle and verified empirically as well, the Center-of-Gravity (COG) of a reduce task R (CoGR) is always one of R's feeding nodes since it is less expensive to access data locally than to shuffle them over the network. As the first step of our scheduling algorithm we first calculate a list of *Candidate Node List* for each reduce task RT_i that may be considered as its destination node D_{RT_x} during the Reduce phase. It is important to know the fact that not all COG happen to be a node with available slots to process at least one reduce task. Consequently, the task tracker on such a node would have not sent out a reduce task request to the job tracker. This situation was overlooked by the CoGR scheduler. We choose a node cn as a *Candidate Node* of a specific reduce task RT_i, if:

- cn is a feeding node of reduce task RT_i and cn's task tracker has sent out a reduce task request; or,
- cn's task tracker has sent out a reduce task request and cn is the closest node to one of the feeding nodes of reduce task RT_i' that is not a candidate node.

Algorithm 1 Smart Shuffling Scheduler

1: **procedure** $CanSelection(CandidateGroup(cRt_1, cRt_2....cRt_i))$
2: **for** each candidate group **do**
3: Execute **Procedure** *Link Weight Calculation* and return Max $|LinkWeight - \alpha|$ to A
4: Execute **Procedure** *Node Load Calculation* and return Max $|workload - \beta|$ to B
5: Execute **Procedure** *Overall Intermediate Data Relocation Calculation* and return Sum Data relocation to C
6: **end for**
7: The candidate group $(CN_{1j}, CN_{2j}, CN_{3j}, ..., CN_{ij})$ that returns the Min $(aA * bB * cC)$ is the list of destination nodes for the reduce tasks that needs to be processed.
8: **end procedure**

B. Detailed Description and Simulation Detail

Procedure *Candidate Group Selection* (Algorithm 2) describes how each candidate group is selected.

Algorithm 2 Candidate Group Selection

1: **procedure** $CanSelection(TaskRq, FeedingNodes, ReduceTask)$
2: **for** Every Feeding Node $FN_{ij} \in RT_i$ **do**
3: **if** $FN_{ij} \in$ TaskRq **then**
4: Add FN to Candidate Node List of RT_i
5: **else**
6: Calculate the distance between all nodes in TaskRq with FN_{ij}
7: Add the node with the minimal distance to this Candidate List of RT_i
8: **end if**
9: **end for**
10: **for** Every Candidate Node List **do**
11: Choose one CN_{ij} and added it to Candidate Group
12: **end for**
13: Return Candidate Group
14: **end procedure**

Procedure *Link Weight Calculation* (Algorithm 3) describes how 'A' value is calculated for each selected candidate group. As we have mentioned earlier, shortest path is always used to

send a package of intermediate data from its feeding node to the destination node of its reduce task, in order to minimize the network traffic. In our algorithm, we use Floyd-Warshall algorithm [12] to calculate the route that any intermediate data item travels from a feeding node of a reduce task to its destination node.

Algorithm 3 Link Weight Calculation

procedure $CanSelection$(Candidate Group(...))
  $AW \leftarrow \varnothing$
  $AW(\mathcal{L}_j) \leftarrow \varnothing$
  for each link $\mathcal{L}_j$ **do**
    for each Candidate destination node CN_{ij} in the Candidate Group **do**
      Calculate a set of Intermediate Data Relocation Paths: $\mathcal{P}_i$
      if $\mathcal{L}_j \in \mathcal{P}_i$ **then**
        AW $(\mathcal{L}_j) + +$
      end if
      $\alpha = \sum_{i=1}^{x} Z_x/e$
      if α exists **then**
        $AW \leftarrow |AW(\mathcal{L}_j) - \alpha|$
      end if
    end for
    Return $\max AW$
  end for
end procedure

Procedures *Node Workload Calculation* (Algorithm 4) and *Overall Intermediate Data Relocation Calculation* (Algorithm 5) describe how 'B' and 'C' values are calculated respectively for each selected candidate group.

Algorithm 4 Procedure *Node Load Calculation*

procedure $CanSelection(CandidateGroup(...))$
  $AL(CN_{ij}) \leftarrow \varnothing$ ▷ the actual workload of each candidate destination node CN_{ij}
  $AW \leftarrow \varnothing$
  for each Candidate destination node CN_{ij} in the Candidate Group **do**
    for every reduce task RT_x that has CN_{ij} as its candidate destination node **do**
      $Ł(RT_x) + = Ł(RT_(x-1))$ ▷ Calculate the total intermediate data items of this reduce task
    end for
    x + +
    $\beta = \sum_{x=1}^{\Delta} Ł(RT_x)/\Delta$
    if β exists **then**
      $AW \leftarrow |AL(cRt) - \beta|$
    end if
  end for
  Return max AW
end procedure

Algorithm 5 Overall Intermediate Data Relocation Calculation

procedure $CanSelection(CandidateGroup(...))$
  $PartitionSize(RT_xF_y) \leftarrow \varnothing$ ▷ the partition size of all feeding nodes of Reduce task RT_x
  $DR \leftarrow \varnothing$
  for each Candidate destination node CN_{ij} in the Candidate Group **do**
    for every reduce task RT_x that has CN_{ij} as its candidate destination node **do**
      Calculate Intermediate Data Relocation Path $P_{CN_{ij}}$ between RT_xF_y and CN_{ij} ▷ RT_xF_y is the y^{th} feeding node of RT_x
      if $P_{CN_{ij}}$ is not empty **then**
        DR+ = $P_{CN_{ij}} * PartitionSize(RT_xF_y)$ ▷ calculate the overall intermediate data items that need to be relocation
      end if
    end for
  end for
  Return DR
end procedure

V. Empirical Evaluation

A. Experiment Setup

Due to the inaccessibility of arbitrary topology network with large number of nodes, we use simulation study to verify and validate the proposed scheduler, study its performance and compare the results to *Center-of-gravity reduce task scheduler* (CoGRS) presented in [15] by Hammoud et al. and a *Random Scheduler* (RS). In all our experiment, we use $a = 1; b = 1; c = 1$ as the coefficients of A, B and C assuming all these three aspects are equally important.

First of all, we execute our Smart Shuffling algorithm in arbitrary networks. When generating an arbitrary network topology, we use Erdös-Rényi's model [11], in which p the probability of choosing an edge. In our simulation, we choose $p = 0.05, 0.06, 0.07$ and 0.08. Under this setup, we conducted the following three experiments:

1) E-1: we collect the values of $\min A * B * C$, when a). the value of p, b). the number of reduce tasks and c). a list of feeding nodes of each reduce task are given, in an arbitrary network with node numbers vary from 100 to 400.
2) E-2: we repeat E-1 in other arbitrary networks generated using different p values.
3) E-3: we compare our results collected from E-1 and E-2 to the one of *Random Scheduler*

In these above mentioned experiments, the number of reduce tasks and each of their feeding nodes are randomly generated. We also randomly assign nodes to send the reduce task requests and/or as the feeding node of a randomly chosen reduce task at hand. In the simulation running Random Scheduler, we keep the reduce task and each of their feeding nodes number the same as when we run our Smart Shuffling Scheduler (SSS) in the same networks. In the *Random Scheduler*, we randomly assign nodes as the destination node to

run a reduce task. Results and their analysis are explained in subsection V-B.

We then study the performance of our proposed algorithm in tree networks, that is commonly used within data centres. Furthermore, we compare our results in both arbitrary networks and different tree topology networks against the results of *Random Scheduler* and *Center-of-gravity reduce task scheduler*. Experiment results, comparative analyses are explained in detail in the following subsection.

B. Results Analyses and Comparative Evaluation

1) Overall Performance Comparison SSS VS RS:

a) E-1: : Figure 3 to 6 show the overall performance of both SS and Random schedulers in arbitrary networks with node numbers vary from 100 to 400 and network connectivity factor p vary from 0.05 to 0.08. Results show that:

- regardless the network connectivity, SSS consistently outperform RS;
- not only does SSS handle any topology but also its benefits are inversely proportional to the inter-node connectivity of the network topology: the lower this connectivity, the better our algorithm.

In other words, the hotspot elimination mechanism in SSS shows more advantage in a sparsely connected network than a densely connected network. This is because, for example, in a complete network (i.e. there is a link between any pair of nodes), the length of any intermediate data relocation path is 1. The chance of having these paths overlapping, consequently causing a hotspot is minimal. This conclusion is further supported by the results done in various tree network topologies (see Figure 7, 9 and 11).

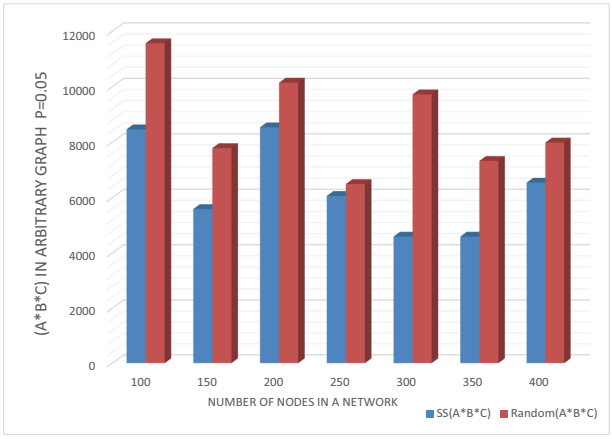

Fig. 3. Overall performance comparison between SS and Random schedulers in arbitrary networks when p=0.05.

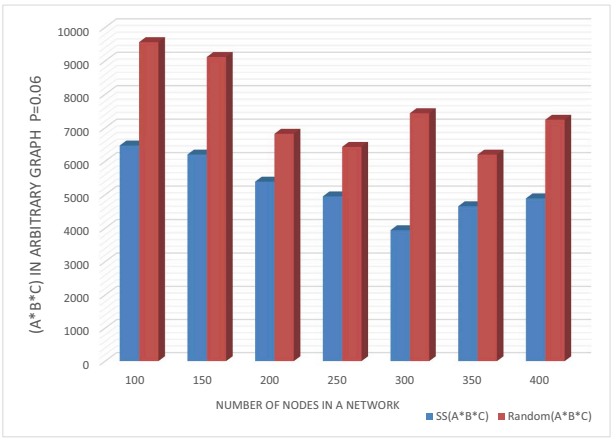

Fig. 4. Overall performance comparison between SS and Random schedulers in arbitrary networks when p=0.06.

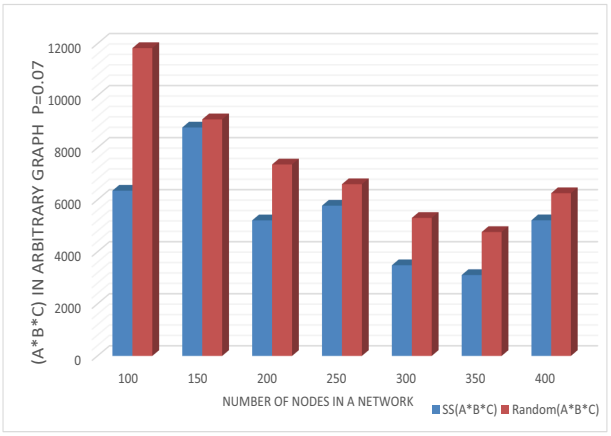

Fig. 5. Overall performance comparison between SS and Random schedulers in arbitrary networks when p=0.07.

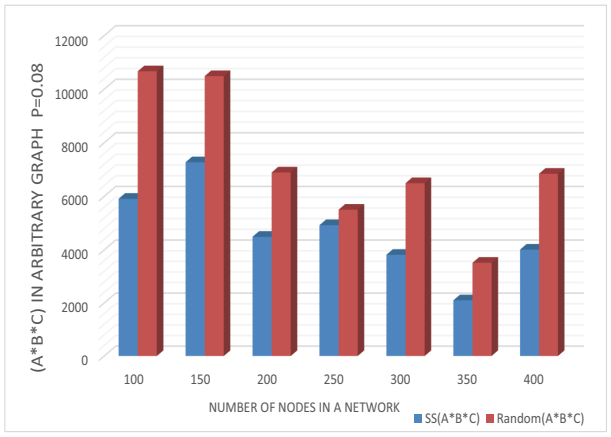

Fig. 6. Overall performance comparison between SS and Random schedulers in arbitrary networks when p=0.08.

2) SSS V.S CoGRS: Figure 7 shows the results of comparing the peak (i.e. the highest) link weight using SSS and CoGRS in various tree network topologies when number of nodes in the network increases from 50 to 300. More specifically, when the same routing algorithm is used, we run SSS and CoGRS and get two sets of destination nodes respectively. We then calculate the peak link weight of using each set of destination nodes. Results show the peak link weight of SSS is consistently lower than the one of CoGRS. Most importantly, the advantage of SSS becomes more significant when the network size scales up.

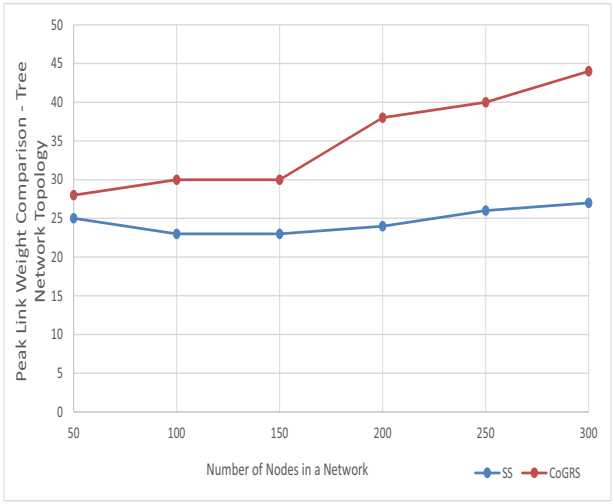

Fig. 7. Peak link weight comparison between SSS and CoGRS in Tree networks.

Figure 8 illustrates the results of similar experiments with different arbitrary topologies. We use a network connectivity factor $p = 0.07$ for generating different arbitrary network topologies. Our intention is to test whether SSS still performs better than CoGRS with respect to peak link weight in rather well-connected networks. We expect that the more connected a network is, the more potential routing routes it offers for data relocation. Thus, the hot link problem should be more severe in sparsely connected network topologies than in more connected ones. This expectation is confirmed by the test results shown in Figure 8. More specifically, peak link weights for rather dense arbitrary networks are less important than those for the tree networks shown in Figure 7. In fact, the more nodes in the network, the less weight the peak link bears. Overall, though not as significant as for trees, the advantage of SSS over CoGRS is still obvious. Thus, we conclude that SSS is clearly stronger in hotspot elimination than CoGRS.

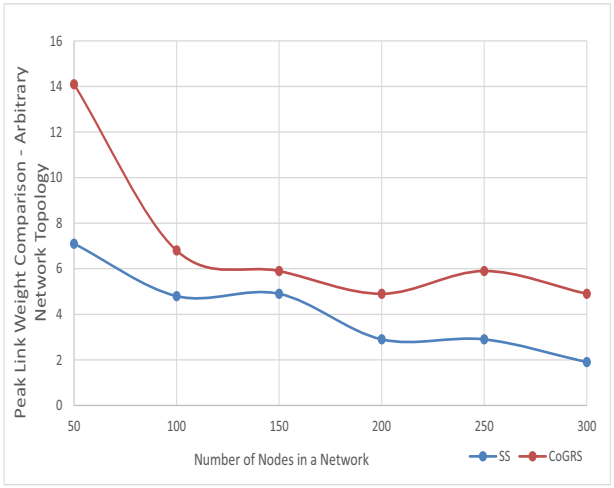

Fig. 8. Peak link weight comparison between SSS and CoGRS in Arbitrary networks.

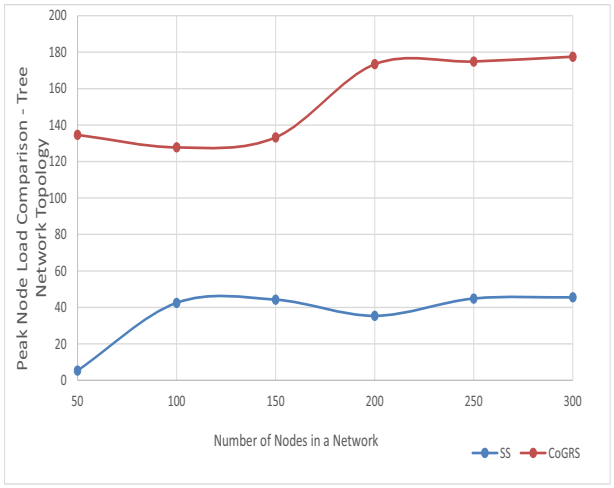

Fig. 9. Peak node load comparison between SSS and CoGRS in Tree networks.

Figure 9 shows the results of comparing the peak (i.e. the highest) node load using SSS and CoGRS in various tree network topologies when number of nodes in the network increases from 50 to 300. Similar to the experiment measuring the peak link weight, we also run SSS and CoGRS and get two sets of destination nodes and calculate the peak node load accordingly. Results show the peak node load of SSS is significantly lower than the one of CoGRS. The results are consistent even when the network size scales up. After carefully study the network setup and experiment results, we noticed that because CoGRS does not consider the workload of nodes, in most cases, CoGRS actually shuffles many reduce tasks to the same node to be processed. This unfortunately increased the workload of some task trackers. On the contrary, due to the workload balancing consideration, SSS spreads

the workload to as many task trackers that has sent out reduce task requests as possible. This greatly contributes to the improvement of overall performance. Similar conclusion can be made in arbitrary network topologies from observing the results illustrated in Figure 10.

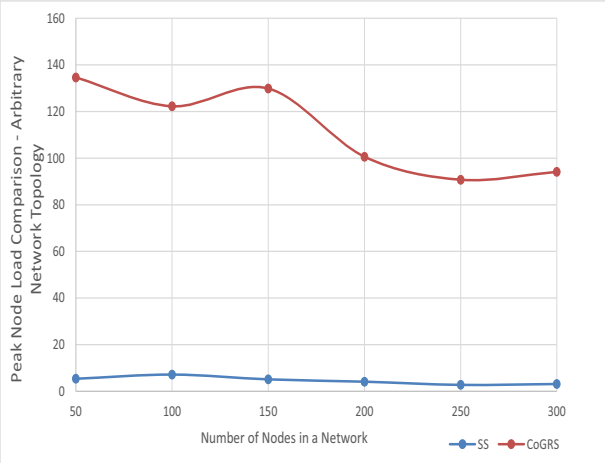

Fig. 10. Peak node load comparison between SSS and CoGRS in Arbitrary networks.

Figure 11 and 12 show the results of comparing the overall intermediate data relocation amount using SSS and CoGRS in various tree and arbitrary network topologies respectively. It is important to know that the goal of CoGRS is to maximize the data locality, which leads to low data relocation. But SSS aims at not only to minimize the data relocation amount, but also maximize the node workload balance as well as reduce the hotspot. As we have mentioned earlier, the tradeoffs between these three aspects are unavoidable. The question that concerns us is whether the link weight and node workload balance gain are at the expense of data locality? The results of this experiment (in both various tree and arbitrary network topologies) show:

- in majority cases (8/12), the overall intermediate data relocation amount caused by both schedulers are very similar;
- the larger network size, the smaller difference in the overall intermediate data relocation amount between SSS and CoGRS;
- when the network size is small (e.g. 50 nodes in both Figure 11, and 12), the overall intermediate data relocation amount cause by SSS is 11% higher comparing to the one of CoGRS.

In our opinion, comparing to the significant advantage on the link weight and node workload balance, this difference is acceptably small.

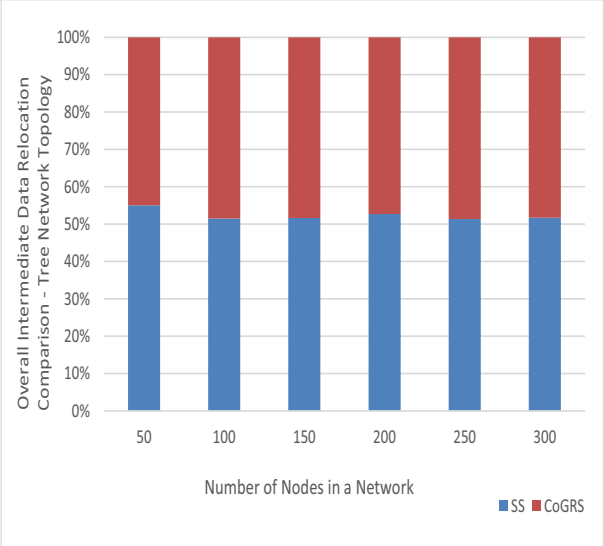

Fig. 11. An example of hotspots in a network.

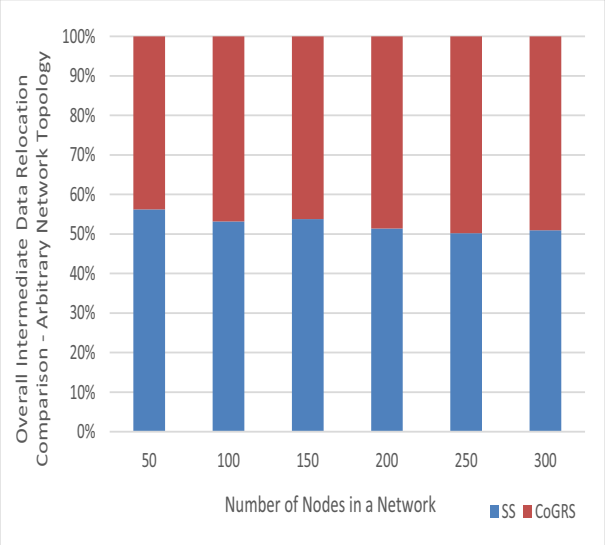

Fig. 12. An example of hotspots in a network.

VI. Conclusion and Future Work

In this paper, we study data relocation incurred during the shuffle stage in the MapReduce framework. We aim at a) minimizing the overall network traffic, b) achieving workload balancing, and c) eliminating network hotspots in a network with arbitrary topologies, in order to improve the overall performance of the MapReduce framework.

We introduce a Smart Shuffling Scheduler in order to satisfy these three goals. Our in-depth simulation results show that, for arbitrary network topologies, our Smart Shuffling Scheduler systematically outperforms the Random and CoGRS

scheduler in terms of hotspot elimination as well as reduce task load balancing, while ensuring traffic caused by data relocation is low. Not only does our algorithm handle any topology but also its benefits are inversely proportional to the inter-node connectivity of the network topology: our algorithm performs the best in tree topology that is commonly adopted in data centres.

ACKNOWLEDGMENT

The authors gratefully acknowledge financial support from the Natural Sciences and Engineering Research Council of Canada (NSERC) under Grant No. 371977-2009 RGPIN.

REFERENCES

[1] Apache Software Foundation. Hadoop, http://hadoop.apache.org/core [Online; accessed Jan-11-2014].

[2] Greenplum HD, http://www.greenplum.com [Online; accessed Jan-11-2014].

[3] Hadoop FairScheduler, http://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html [Online; accessed Jan-11-2014].

[4] Hadoop CapacityScheduler, http://hadoop.apache.org/docs/r0.19.1/capacity_scheduler.html [Online; accessed Jan-11-2014].

[5] ARASANAL, R., AND RUMANI, D. Improving mapreduce performance through complexity and performance based data placement in heterogeneous hadoop clusters. In *Distributed Computing and Internet Technology*, C. Hota and P. Srimani, Eds., vol. 7753 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 115–125.

[6] CHEN, F., GUO, K., LIN, J., AND LA PORTA, T. Intra-cloud lightning: Building cdns in the cloud. In *INFOCOM, 2012 Proceedings IEEE* (March 2012), pp. 433–441.

[7] CHOWDHURY, M., ZAHARIA, M., MA, J., JORDAN, M. I., AND STOICA, I. Managing data transfers in computer clusters with orchestra. *SIGCOMM Comput. Commun. Rev. 41*, 4 (Aug. 2011), 98–109.

[8] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (2004), OSDI'04, pp. 10–10.

[9] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[10] ELTABAKH, M. Y., TIAN, Y., ÖZCAN, F., GEMULLA, R., KRETTEK, A., AND MCPHERSON, J. Cohadoop: Flexible data placement and its exploitation in hadoop. *Proc. VLDB Endow. 4*, 9 (June 2011), 575–585.

[11] ERD, P., ET AL. {On Random Graphs, I}. *Publicationes mathematicae 6* (1959), 290–297.

[12] FLOYD, R. W. Nondeterministic algorithms. *Journal of the ACM (JACM) 14*, 4 (1967), 636–644.

[13] GUO, Z., FOX, G., AND ZHOU, M. Investigation of data locality in mapreduce. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)* (2012), IEEE Computer Society, pp. 419–426.

[14] HALIM, F., YAP, R. H., AND WU, Y. A mapreduce-based maximum-flow algorithm for large small-world network graphs. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on* (2011), IEEE, pp. 192–202.

[15] HAMMOUD, M., REHMAN, M. S., AND SAKR, M. F. Center-of-gravity reduce task scheduling to lower mapreduce network traffic. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing* (2012), CLOUD '12, pp. 49–58.

[16] HAMMOUD, M., AND SAKR, M. F. Locality-aware reduce task scheduling for mapreduce. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on* (2011), IEEE, pp. 570–576.

[17] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (2007), EuroSys '07, pp. 59–72.

[18] KONDIKOPPA, P., CHIU, C.-H., CUI, C., XUE, L., AND PARK, S.-J. Network-aware scheduling of mapreduce framework ondistributed clusters over high speed networks. In *Proceedings of the 2012 workshop on Cloud services, federation, and the 8th open cirrus summit* (2012), FederatedClouds '12, pp. 39–44.

[19] LAI, Z.-R., CHANG, C.-W., LIU, X., KUO, T.-W., AND HSIU, P.-C. Deadline-aware load balancing for mapreduce. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on* (2014), IEEE, pp. 1–10.

[20] LI, M., SUBHRAVETI, D., BUTT, A. R., KHASYMSKI, A., AND SARKAR, P. Cam: A topology aware minimum cost flow based resource manager for mapreduce applications in the cloud. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing* (2012), HPDC '12, ACM, pp. 211–222.

[21] LIN, J., AND SCHATZ, M. Design patterns for efficient graph algorithms in mapreduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs* (2010), ACM, pp. 78–85.

[22] PALANISAMY, B., SINGH, A., LIU, L., AND JAIN, B. Purlieus: locality-aware resource allocation for mapreduce in a cloud. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011), ACM, p. 58.

[23] QIN, P., DAI, B., HUANG, B., AND XU, G. Bandwidth-aware scheduling with sdn in hadoop: A new trend for big data. *CoRR abs/1403.2800* (2014).

[24] TAN, J., MENG, S., MENG, X., AND ZHANG, L. Improving reducetask data locality for sequential mapreduce jobs. In *INFOCOM, 2013 Proceedings IEEE* (2013), IEEE, pp. 1627–1635.

[25] TAN, J., MENG, X., AND ZHANG, L. Coupling task progress for mapreduce resource-aware scheduling. In *INFOCOM, 2013 Proceedings IEEE* (April 2013), pp. 1618–1626.

[26] WOLF, J., RAJAN, D., HILDRUM, K., KHANDEKAR, R., KUMAR, V., PAREKH, S., WU, K.-L., AND BALMIN, A. Flex: A slot allocation scheduling optimizer for mapreduce workloads. In *Middleware 2010*. Springer, 2010, pp. 1–20.

[27] XIE, J., YIN, S., RUAN, X., DING, Z., TIAN, Y., MAJORS, J., MANZANARES, A., AND QIN, X. Improving mapreduce performance through data placement in heterogeneous hadoop clusters. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on* (April 2010), pp. 1–9.

[28] XU, F., LIU, F., ZHU, D., AND JIN, H. Boosting mapreduce with network-aware task assignment. In *Cloud Computing*, V. C. Leung and M. Chen, Eds., vol. 133 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*. 2014, pp. 79–89.

[29] ZAHARIA, M., BORTHAKUR, D., SEN SARMA, J., ELMELEEGY, K., SHENKER, S., AND STOICA, I. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems* (2010), ACM, pp. 265–278.