Joint Optimization of MapReduce Scheduling and Network Policy in Hierarchical Clouds

Donglin Yang
University of North Carolina at Charlotte
dyang33@uncc.edu

Wei Rang
University of North Carolina at Charlotte
wrang@uncc.edu

Dazhao Cheng
University of North Carolina at Charlotte
dazhao.cheng@uncc.edu

ABSTRACT

As MapReduce is becoming increasingly popular in large-scale data analysis, there is a growing need for moving MapReduce into multi-tenant clouds. However, there is an important challenge that the performance of MapReduce applications can be significantly influenced by the time-varying network bandwidth in a shared cluster. Although a few recent studies improve MapReduce performance by dynamic scheduling to reduce the shuffle traffic, most of them do not consider the impact by widely existing hierarchical network architectures in data centers. In this paper, we propose and design a Hierarchical topology (Hit) aware MapReduce scheduler to minimize overall data traffic cost and hence to reduce job execution time. We first formulate the problem as a Topology Aware Assignment (TAA) optimization problem while considering dynamic computing and communication resources in the cloud with hierarchical network architecture. We further develop a synergistic strategy to solve the TAA problem by using the stable matching theory, which ensures the preference of both individual tasks and hosting machines. Finally, we implement the proposed scheduler as a pluggable module on Hadoop YARN and evaluate its performance by testbed experiments and simulations. The experimental results show Hit-scheduler can improve job completion time by 28% and 11% compared to Capacity Scheduler and Probabilistic Network-Aware scheduler, respectively. Our simulations further demonstrate that Hit-scheduler can gain the traffic cost by 38% at most and improve the average shuffle flow traffic time by 32% compared to Capacity scheduler.

CCS CONCEPTS

• **Computer systems organization** → **Distributed system**; • **General and reference** → *Performance*; • **Networks** → Network scalability;

KEYWORDS

Joint Optimization, MapReduce Scheduling, Network Policy, Hierarchical Clouds, Topology Aware Assignment

ICPP 2018, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6510-9/18/08.
https://doi.org/10.1145/3225058.3225099

ACM Reference Format:
Donglin Yang, Wei Rang, and Dazhao Cheng. 2018. Joint Optimization of MapReduce Scheduling and Network Policy in Hierarchical Clouds. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225099

1 INTRODUCTION

While many enterprises and organizations have aggressively deployed highly scalable data-parallel framework likes MapReduce [9] to process petabytes of data every day, a recent trend is to move MapReduce applications from the environment of dedicated clusters to multi-tenant shared clusters, such as Amazon EC2, to improve the cluster utilization. However, there is an important challenge that the performance of MapReduce applications can be significantly influenced by the network bandwidth in a shared cluster. As the network resource is shared among virtual machines hosting various applications or among different computing frameworks, the bandwidth available for MapReduce applications becomes changeable over time.

The data communication traffic in MapReduce is mainly caused by two factors, i.e., remote map access and intermediate data shuffle. Many popular techniques like delay scheduling [29] and flow-based scheduling [28] are designed to place individual Map tasks on the machines or racks where most of their input data is located, which aims to reduce the remote map traffic. However, most intermediate data sets still spread over the cluster randomly in a distributed HDFS (Hadoop Distributed File System). The subsequent job stages (i.e., shuffle) have to transfer intermediate data cross machines or racks, which are often heavily congested by the constrained bandwidth. Prior studies [2, 14] have shown the shuffle traffic mostly dominates the overall performance of MapReduce jobs compared to the remote map traffic. Thus, a few recent studies [2, 30] improve MapReduce performance by dynamic scheduling to reshape or reduce the shuffle traffic. However, most of them do not consider the impact by widely existing complex network architectures in data centers, or just set up a simple model for workload scheduling. Indeed, a large number of researches have payed attention to the areas of MapReduce scheduling and network policy management, respectively. However, dynamic MapReduce task scheduling and network policy optimization have been so far addressed in isolation, which may significantly degrade the overall system performance. On the one hand, studies [15], [27] have focused primarily on exploiting software defined networking and network function virtualization in the area of network policy management. They mainly assume a static allocation of compute resources, which is not true in most real clusters. On the other hand, MapReduce task scheduling has largely

concentrated on the flexible assignment, efficient placement and locality awareness to maximize various cluster resource (e.g., CPU, MeM and network I/O) utilization, and optimize application level SLAs. However, there are very few works on dynamic task assignment in conjunction with dynamic network policy configuration to optimize the cluster wide communication cost.

In this paper, we tackle this challenging problem by jointly optimizing task scheduling and network policy management in the cloud with hierarchical network architecture. We find that the correlation between the network architecture and the task scheduling is very weak under the current popular schedulers. The shuffle-heavy data does not correspond to the low latency route due to two factors. First, the data fetching time of reduce tasks depends on not only distribution of the intermediate data in the cluster but also the limited bandwidth. Second, the bandwidth on the routing path is not static but dynamic, which is also influenced by the assignment of tasks running in the cloud. In this work, we propose and design a Hierarchical topology (Hit) aware MapReduce scheduler to minimize overall data traffic cost and hence to reduce job execution time. More specifically, we make the following technical contributions.

- We formulate the problem as a Topology Aware Assignment (TAA) optimization problem while considering dynamic computing and communication resources in the cloud with hierarchical network architecture. To achieve jointly optimizing network policy and task assignment, we model traffic flow, traffic policy, task assignment, routing path and corresponding cost according to flow-based traffic policy.
- We find that optimizing each shuffle flow to obtain a globally optimal task assignment is NP-Hard. We then develop a synergistic strategy to solve the TAA problem by using the stable matching theory, which ensures the preference of both individual tasks and hosting machines.
- We implement the proposed scheduler as a pluggable module on Hadoop YARN and evaluate its performance by testbed experiments and simulations. The experimental results show Hit-scheduler can improve job completion time by 28% and 11% compared to Capacity Scheduler and Probabilistic Network-Aware scheduler, respectively. Our simulations further demonstrate that Hit-scheduler can gain the traffic cost by 38% at most and improve the average shuffle flow traffic time by 32% compared to Capacity scheduler.

The rest of this paper is organized as follows. Section 2 gives background and motivations on shuffle traffic optimization. Section 3 describes the modeling and formulation of TAA problem. Section 5 gives details on solution design. Section 6 gives details on system implementation. Section 7 presents experimental results. Section 8 reviews related work. Section 9 concludes the paper.

2 BACKGROUND AND MOTIVATION

2.1 Performance Impact due to Data Shuffle

When running a MapReduce job, Map and Reduce tasks are usually scheduled to maximize concurrency (i.e., occupy the entire cluster or as much as possible) in order to improve cluster utilization or achieve load balance. As Reduce tasks have to read the output from the corresponding Map tasks, such all-map-to-all-reduce shuffle operation results in an all-node-to-all-nodes communication,

Table 1: Benchmarks Characterization

Workload Type	Benchmark Proportion and Type
Shuffle-heavy	terasort(5%), index(10%), join(10%), sequence count(10%), adjacency(5%)
Shuffle-medium	inverted-index(10%), term-vector(10%)
Shuffle-light	grep(15%), wordcount(10%), classification(5%), histogram(10%)

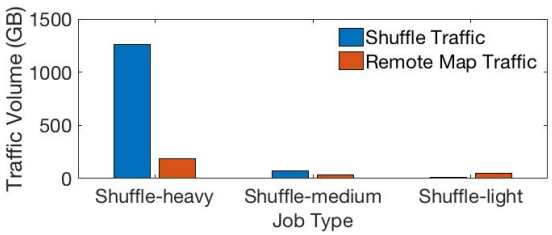


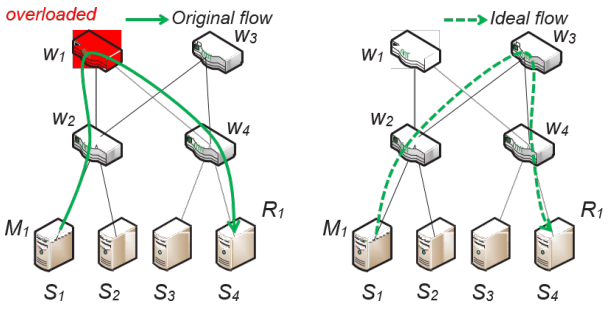
Figure 1: Traffic Volume During Shuffle Phase

which stresses the network bisection bandwidth. In particular, concurrently running multiple shuffle-heavy jobs may significantly increase the pressure of network bandwidth [7]. Although many existing schedulers pay attention to optimize remote map access traffic, most of them are shuffle-unaware and result in high pressure on the network in the cloud. This is due to the fact that Reduce tasks are typically scheduled well before the completed distribution of Map output is known.

To identify the impact of shuffle communication, we analyze a few representative workloads consisting of benchmark drawn from Apache Hadoop release [1]. The benchmarks are characterized as Shuffle-heavy, Shuffle-medium and Shuffle-light, and the percentages of different jobs are shown in Table 1. Figure 1 shows that the actual volumes of the total Shuffle traffic and remote Map traffic for Shuffle-heavy, Shuffle-medium and Shuffle-light jobs, respectively. The result shows the Shuffle data volume of Shuffle-heavy jobs has a significant contribution (>75%) of the total communication traffic, and the contribution of the remote Map traffic is less than 20% of the total communication traffic. This observation demonstrates that the shuffle traffic dominates the overall performance of Shuffle-heavy jobs compared to the remote map traffic. Thus, we focus on optimizing the data shuffle traffic for MapReduce applications.

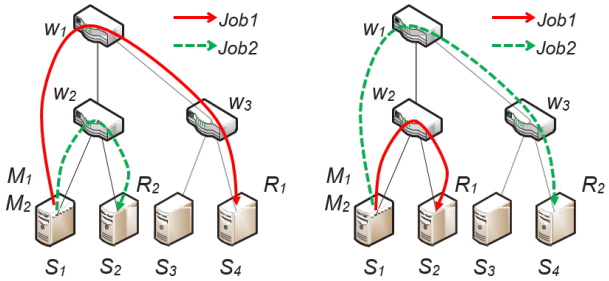
2.2 Challenges due to Hierarchical Networks

Current data centers follow to a great extend a common network architecture, known as the three-tier architecture [1]. At the bottom level (i.e., access tier), each server connects to one (or two) access switch. At the aggregation tier, each access switch connects to one (or two) switches. Finally, each aggregation switch connects with multiple switches at the core tier. Figure 2 shows a 2-layer topology, which is usually rooted at one of the core switches. There are more alternative architectures proposed recently, such as VL2 [12], PortLand [23] and BCube [13]. In contrast, most existing task scheduling policies do not consider the possible impact by these hierarchical



(a) Traffic-unaware flow (b) Traffic-aware flow

Figure 2: Workflow under hierarchical networks.



(a) Topology-unaware (b) Topology-aware

Figure 3: Assigning Job 1 (M_1, R_1) and Job 2 (M_2, R_2) in the cloud.

network architectures, or just set up a simple model for data intensive applications.

These topology unaware strategies may significantly deteriorate the system performance in the hierarchical network architecture. For example, Figure 2 depicts a scenario that the task(R_1) requires to transfer the related data from the task(M_1). As shown in Figure 2(a), the shuffle traffic flow from M_1 task to R_1 task is configured to traverse w_2, w_1 and w_4 sequentially. Since the capacity of each switch is constrained by the processing rate, the overloaded w_1 will lead to the packets of this shuffle traffic flow being rejected. An alternative solution is to optimize the shuffle traffic flow as shown in Figure 2(b) and correspondingly achieves lower network overhead. Furthermore, cloud environment hides the physical topology of the infrastructure, which inhibits optimal scheduling. For instance, the tasks associated with a job may be placed across multiple racks while this information is not typically visible to the application. Apparently, topology invisible solutions will lead to a longer transfer time to shuffle large amounts of intermediate data to Reduce task.

2.3 Case Study

We conducted a case study based on a 5-node Hadoop cluster (i.e., one master nodes and four slave nodes) and ran two jobs (one shuffle-heavy job and one shuffle-light job) with the same input data sizes by using different schedulers. As shown in Figure 3, four slave nodes are connected via the Tree network topology. We set different network latencies between machines by implementing API `DelayFecther()`, which provides a function to mimic the task level data transmission delay between machines. In order to simplify the data shuffle analysis, we configure that each server can host at most two tasks. In the experiment, we submit jobs by using Capacity Scheduler to enqueue and assign the Map and Reduce tasks to different nodes. Here, we assume that the delay caused by one switch equals to 1 (T) and the total delay caused by the network is linearly related to the number of switches the data packets have traversed [22].

After job executions completed, we analyze the related log files and find that the total shuffle data for Job 1 is nearly 34 GB and 10 GB for Job 2. It demonstrates that Job 1 is shuffle heavy and Job 2 is shuffle light workload. In particular, we focus on two selected Map tasks and two selected Reduce tasks as shown in Figure 3. The log files show that Map tasks M_1 and M_2 are assigned to server S_1 while Reduce tasks R_1 and R_2 are assigned to S_4 and S_2, respectively. Based on our assumption, the network delay between machines can be calculated based on the number of switches. For example, the delay between S_1 and S_2 is 3 (T). In the shuffle phase, the overall latency is determined by the size of shuffle data and the network delay. The statics from the log files show that the total shuffle delay cost is 112 (GB*T). In contrast, a better solution is to assign reduce task R_1 to server S_2 and R_2 to S_4 based on the given network architecture. Then the total shuffle delay cost can be reduced to 64 (GB*T) correspondingly, which achieves nearly 42% improvement compared with the original task placement policy. The above observations demonstrate that the shuffle-heavy data transmission does not correspond to the low latency route in the hierarchical network architecture. It may lead to network overhead and degrade the performance while scheduling different types of workloads in the cloud.

3 PROBLEM MODELING

3.1 Formulation

We consider a multi-tier data center network which is typically structured under a multi-root tree topology such as canonical [16]. Let $\mathbb{S} = \{s_1, s_2, s_3...\}$ be the set of servers in the data center and $\mathbb{C} = \{c_1, c_2, c_3...\}$ be the set of containers hosted by the servers. We use r_i to denote the physical resource requirements of c_i, such as memory size, CPU cycles. Accordingly, the available physical resource of s_j is defined as q_j. Hence, we use $\sum_{c_i \in A(s_j)} r_i \leq q_j$ to denote that s_j has sufficient resource to accommodate containers c_i, in which $A(s_j)$ defines the set of containers hosted by s_j, and r_i is the resources requirement of c_i.

Running MapReduce application in the cloud, the shuffle traffic is flow-based. We define the shuffle traffic flow as $\mathbb{F} = \{f_1, f_2, f_3...\}$. For each flow f_i, it has several important properties {size, src, dst}. Correspondingly, $f_{i.src}$ specifies the source container running Map task while $f_{i.dst}$ specifies the destination container running Reduce task, e.g., $f_{i.src} = c_1$ and $f_{i.dst} = c_2$. The data rate of $f_{i.rate}$ is represented by the shuffle data rate from $f_{i.src}$ to $f_{i.dst}$. Then we define a binary variable $x_{ij}(x_{ij}^m, x_{ij}^r)$ to denote whether the j_{th} Map

or Reduce task is assigned to the container c_i. Each container can host at most one Map or Reduce task.

Let $\mathbb{W} = \{w_1, w_2, w_3...\}$ denote the set of all switches in the cloud. Each switch has two properties, {capacity, type}. We define that $w_{i.capacity}$ is the capacity of w_i, and $w_{i.type}$ is the type of the switches. The set of policies for shuffle traffic is defined as $\mathbb{P} = \{p_1, p_2, p_3...\}$. In general, the shuffle traffic flows and policies in the cloud are one-to-one correspondence. For each policy p_i, it also has important properties {list, len, type}, where $p_{i.list}$ is the list of switches that f_i will traverse, e.g. $p_{i.list[0]}$ is the first access switch that the flow will traverse. And $p_{i.len}$ is the size of the switches list. And $p_{i.type}$ is the type of i_{th} switch the flow will traverse, e.g. $p_{i.type[0]}$ is the type of the first switch in the list. Let $P(c_i, c_j)$ be the policies defined for shuffle traffic from container c_i to c_j.

Here, we define that if and only if all the required switches are allocated to p_i with the correct type and order, then the policy p_i is *satisfied.*

$$p_{i.type[j]} == w_{type}, \forall\, w = p_{i.list[j]}, j = 1, ..., p_{i.len}.$$

We denote $R(n_i, n_j)$ as the routing path between nodes (i.e., servers, switches) n_i and n_j. For a shuffle traffic flow f_i, its actual routing path is:

$$\begin{aligned} R_i(f_{i.src}, f_{i.dst}) &= R(f_{i.src}, p_{i.list[0]}) \\ &+ \sum_{j=1}^{p_{i.len-2}} R(p_{i.list[j]}, p_{i.list[j+1]}) \\ &+ R(p_{k.list[p_{k.len}-1]}, f_{i.dst}). \end{aligned} \tag{1}$$

Hence, we define the shuffle cost of all the traffic from containers c_i to c_j as

$$\begin{aligned} C(c_i, c_j) &= \sum_{p_k \in P(c_i, c_j)} \sum_{f_k \in R_k(c_i, c_j)} f_{k.rate} \times c_s \\ &= \sum_{f_k \in R_k(c_i, c_j)} \{C_k(c_i, p_{k.list[0]}) \\ &+ \sum_{j=1}^{p_{i.len-2}} C_k(p_{k.list[j]}, p_{k.list[j+1]}) \\ &+ C_k(p_{k.list[p_{k.len}-1]}, c_j)\}, \end{aligned} \tag{2}$$

in which c_s is the unit cost for corresponding routing path, and $C_k(c_i, p_{k.list[0]})$ is the shuffle traffic cost between c_i and the first access switch for flows which matches p_k, while, similarly, we define that $C_k(p_{k.list[p_{k.len}-1]}, c_j)$ is the shuffle traffic cost between the c_j and corresponding access switch.

4 MINIMIZING DATA SHUFFLE COST

We denote $A(c_i)$ as the server which hosts container c_i. And $A(p_k)$ is the set of switches which are allocated to policy p_k. Give the set of containers $\mathbb{C}$, servers $\mathbb{S}$, policies $\mathbb{P}$ and switches $\mathbb{W}$, we define the Topology Aware Assignment (TAA) of Map and Reduce tasks to minimize the total shuffle traffic cost:

$$\begin{aligned} &min \sum_{c_i \in \mathbb{C}} \sum_{c_j \in \mathbb{C}} C(c_i, c_j) \\ &s.t. A(c_i) \neq 0,\ \forall c_i \in \mathbb{C}; \\ &\sum_{i \in R} x_{ij}^m = 1; \sum_{i \in R} x_{ij}^r = 1; \\ &\sum_{j \in R} x_{ij}^r + \sum_{j \in R} x_{ij}^m = 1; \\ &\sum_{c_i \in A(s_j)} r_i \leq q_j; \\ &\sum_{p_k \in A(w_i)} f_{k.rate} \leq w_{i.capacity}, \forall w_i \in \mathbb{W}; \\ &p_{i.type[j]} == w_{type}, \forall\, w = p_{i.list[j]}, \forall p_i \in \mathbb{P}. \end{aligned} \tag{3}$$

The first constraint ensures that each container is only deployed on one server. The second constraint guarantees that one Map or Reduce task is hosted by one container. The third constraint demonstrates that one container can host one task. The fourth and fifth constraints are the capacity requirement for switches and servers. The sixth constraint requires that all the flow should satisfy the traffic policies.

The above mentioned TAA problem is NP-Hard, we will show that the Multiple Knapsack Problem (MKP) [19] can be reducible to this topology aware task assignment in polynomial time. It has been proven that the decision for MKP is strongly NP-complete.

The multiple knapsack problem [19] is one of the most studied problems in combinatorial optimization, given n items, with each item $1 \leq j \leq n$ having an associated profit p_j and weight w_j. Given a set of K knapsacks with a corresponding c_i capacity for each knapsack, the MKP is to select k disjoint subsets of items such that the total profit due to selected items is maximized. For each knapsack, the total weight of items assigned to it in the subset should be less than its capacity.

Let's perform the mapping from MKP to a special case of our TAA problem: : there a simple cluster containing only two physical servers, i.e., s_1 and s_2, are connected through hierarchical edge switches. For these servers, each of them is configured to host n containers. All these containers form two equal groups A and B. Group A hosts n Map tasks and group B hosts n Reduce task respectively. We assume that all the Reduce tasks only have to retrieve the output from a Map task in group A, resulting in n shuffle traffic flows. To simplify the problem, here we assume that each flow has to traverse one switch which is selected from multiple intermediate switches. Thus, a reasonable solution is to assign all n containers hosting Map tasks to s_1 and all n containers hosting Reduce tasks to s_2, correspondingly. Then, the TAA problem is equivalent to finding an appropriate intermediate switches for each flow. Then, we assume that each flow f_i is an item to be assigned to a knapsack, which is the intermediate switch here. Corresponding, the size of item is $f_{i.size}$ and each knapsack has limited capacity. The profit of assigning f_i to one of intermediate switches is the negative of the shuffle cost defined in the equation 3. The TAA problem becomes finding an optimal allocation of all flows to the corresponding intermediate switches, maximizing the total profit.

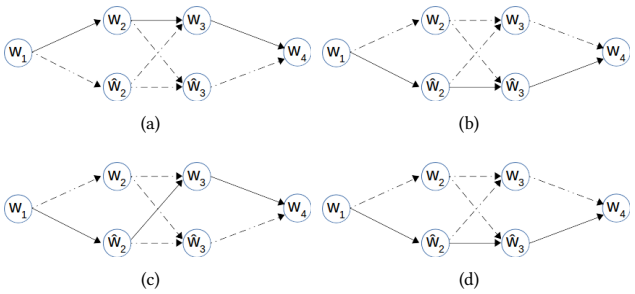


Figure 4: Separated Optimization on Policies

Therefore, the MKP problem is reducible to the TAA problem in polynomial time. The TAA problem is NP-Hard.

5 SOLUTION

5.1 Optimization between Network policy and Task Assignment

5.1.1 Network Policy. When performing the network policy optimization, it will possibly result in rescheduling the switches on the selected path. Here, we denote $p_{k.list[i]} \rightarrow \hat{w}$ as rescheduling the i_{th} switch of p_k to a new switch $\hat{w}$. All the switches having sufficient capacities to handle the shuffle data traffic are denoted as the candidates to be rescheduled as below:

$$\begin{aligned} &S(p_{k.list[i]}) = \{\hat{w} | \hat{w}_{type} == p_{k.type[i]}, \\ &\sum_{p_i \in A(\hat{w})} f_{i.rate} \leq \hat{w}_{capacity} - f_{k.rate}, \\ &\forall \hat{w} \in \mathbb{W} \setminus p_{k.list[i]}\}. \end{aligned} \tag{4}$$

In the beginning, the flow f_k is assigned with required switches based on a random policy p_k. Then we will consider optimization on intermediate switches (e.g. $p_{k.list[i]}$ ∀i=1,2, ... $p_{k.len}$-2) and the end access switches respectively (e.g. $p_{k.list[0]}$ or $p_{k.list[p_{k.len}-1]}$). We start from simplest case that performing optimization of p_k on one switch $p_{k.list[i]}$ on intermediate access switches. Here, we define the *utility* as the shuffle traffic cost reduction gained by optimizing the shuffle traffic policy $p_{k.list[i]} \rightarrow \hat{w}$:

$$\begin{aligned} &U(p_{k.list[i]} \rightarrow \hat{w}) = C_k(p_{k.list[i-1]}, p_{k.list[i]}) \\ &+ C_k(p_{k.list[i]}, p_{k.list[i+1]}) - C_k(p_{k.list[i-1]}, \hat{w}) \\ &- C_k(\hat{w}, p_{k.list[i+1]}). \end{aligned} \tag{5}$$

If the optimization of p_k involves two or more switches along the flow path, we can optimize the policies by rescheduling the corresponding switches one by one. For example, as shown in Figure 4(a) and (b), if the original flow path $w_1 \rightarrow w_2 \rightarrow w_3 \rightarrow w_4$ is rescheduled as $w_1 \rightarrow \hat{w}_2 \rightarrow \hat{w}_3 \rightarrow w_4$, the optimization can be separated into Figure 4(c) and (d). The corresponding *utility* remains the same:

$$U(w_2 \rightarrow \hat{w}_2, w_3 \rightarrow \hat{w}_3) = U(w_2 \rightarrow \hat{w}_2) + U(w_3 \rightarrow \hat{w}_3). \tag{6}$$

The second case is that the optimization of p_k is performed on end access switches, which results in $p_{k.list[0]} \rightarrow \hat{w}$ or $p_{k.list[p_{k.len}-1]} \rightarrow \hat{w}$. The difference between intermediate switches and end access switches is that the associated source or destination container should be considered, because the end access switches communicate with containers directly. The *utility* of optimization on $p_{k.list[0]}$ is shown as below:

$$\begin{aligned} &U(p_{k.list[0]} \rightarrow \hat{w}) = C_k(f_{k.src}, p_{k.list[0]}) \\ &+ C_k(p_{k.list[0]}, p_{k.list[1]}) - C_k(f_{k.src}, \hat{w}) \\ &- C_k(\hat{w}, p_{k.list[1]}). \end{aligned} \tag{7}$$

5.1.2 Task Assignment. In order to assign tasks, we should make sure that there are available containers to host the Map or Reduce tasks. For example, if we want to optimize a task x_{ij} hosted on c_i from currently allocated server $A(c_i)$ to a new server $\hat{s}$, the candidate servers $\hat{s}$ can be characterized by:

$$O(c_i) = \{\hat{s} | (\sum_{c_k \in A(\hat{s})} r_k + r_i \leq \hat{q})\}. \tag{8}$$

Let's consider a container c_i hosting j_{th} map task, where x^m_{ij}=1, is allocated on server s_k initially. The shuffle traffic cost induced by c_i between s_k and the access switches is as below:

$$C_i(s_k) = \sum_{p_k \in P(c_i, *)} C_k(c_i, p_{i.list[0]}). \tag{9}$$

While optimizing the assignment of j_{th} reduce task x^r_{ij}, which is original hosted on the container c_i, the difference between optimization on map tasks is that it only involves the last egress switches. Similarly, the shuffle traffic cost is $\sum_{p_k \in P(*, c_i)} C_k(p_{i.list[p_{k.len-1}]}, c_i)$. Here we define the *utility* brought by rescheduling container c_i hosting x_{ij} from $A(c_i)$ to another server $\hat{s}$ as below:

$$U(A(c_i) \rightarrow \hat{s}) = C_i(A(c_i)) - C_i(\hat{s}). \tag{10}$$

5.1.3 Separable Optimization. For each flow, we can conclude that the optimization of task assignment and network policies are independent with each other. We can optimize them separately and achieve the same total *utilities* as optimizing them together.

$$\begin{aligned} &U(s_1, w_1, w_2 \rightarrow \hat{s_1}, \hat{w_1}, \hat{w_2}) \\ &= U(s_1 \rightarrow \hat{s_1}) + U(w_1 \rightarrow \hat{w_1}) + U(w_2 \rightarrow \hat{w_2}). \end{aligned} \tag{11}$$

From Equation 6 and 11, we can conclude that the rescheduling order of individual switches and containers running corresponding tasks are independent with each other and the total *utilities* remains the same as optimizing all the switches on the routing path together. These observations mean that the optimization of traffic policies and task assignment can be performed independently.

5.2 Separated Optimization Strategy

In the following, we solve the above separable optimization problem based on a typical many-to-one stable matching, i.e., Stable Marriage Problem.

5.2.1 Grading Tasks by Servers. A container, hosting Map or Reduce task, usually has multiple shuffle traffic flows. It is difficult to find the optimal policies for all of these flows. Considering the separable properties observed above, we firstly optimize the traffic policies for each Map and Reduce task pair. We then output a preference matrix for all containers hosting tasks from the servers.

For a flow f_i, it needs to traverse n = $p_{i.len}$ switches. Flows originate from the *source* containers ($f_{i.src}$) hosting Map task x^m_{*j},

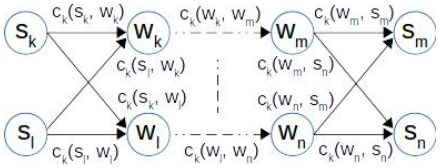


Figure 5: Shuffle Traffic Flow Path

and terminate at the *sink* containers ($f_{i.dst}$) hosting Reduce task x^r_{*k}. The container $f_{i.dst}$ or $f_{i.dst}$ can be possibly allocated to any servers satisfying Equation 8. Similarly, the switches on the shuffle traffic path should be all possible switches defined in Equation 4. The weight of each edge is the corresponding shuffle cost between two connected nodes. Figure 5 is an example traffic flow f_k. It originates from containers hosting Map tasks allocated on server $\{s_i,...,s_j\}$ and then traverses two kind of switches, finally terminates at one of the containers hosting Reduce tasks allocated on $\{s_m,...,s_n\}$. Obviously, the shuffle traffic route which can achieve the highest utility for Equation 5 is the optimal routing path from *source* to *sink*. Here, we propose the *Policy Optimization Algorithm*, which is described in Algorithm 1. It aims to minimize the total shuffle traffic cost via the optimization of policies. The optimal routing path is returned in the line 5, where s_{src} is the server, which is assigned with container hosting Map task, and s_{dst} is the destination server which is assigned with container hosting Reduce task and w_{list} is the list of switches that the flow will traverse.

Based on the algorithm, we obtain an updated M×N preference matrix P for task assignment, where M is the number of servers and N is the number of containers hosting tasks. For each component in the matrix, P(s,c) is the grade while assigning the container c hosting corresponding task on server s. The grades are based on the utility function and will be updated when rescheduling a new routing path.

5.2.2 Grading Servers by Tasks. According to Equation 10, each task hosted by corresponding container also can rank the candidate servers in decreasing order depending on their *utility* values. In the policy optimization, we obtain the grades while assigning containers hosting tasks to different candidate servers through the preference matrix. We can also obtain the ranked order list for each container, i.e. $l_i = \{c_1, c_2, ..., \}$, where c_1 is the most preferable container hosting corresponding task for the i_{th} server.

5.2.3 Stable Matching. The preferences of containers and servers might be inconsistent, which have been proven to be a typical many-to-one stable matching problem (i.e., Stable Marriage Problem [11]). Here, we define a pair (c_i, s_j), which means assigning c_i container hosting task on the s_j server, as a blocking pair, if both c_i and s_j prefer being together to their assignments. A matching is stable if it does not have any blocking pairs. Thus, an unstable matching between containers and servers will always give us opportunities to find the optimal tasks assignment towards minimizing the total shuffle traffic cost. Here, to solve the problem, we apply the modified Gale-Shapley algorithm to address the conflict, which is shown in Algorithm 2. Initially, we should ensure that tasks are hosted by containers. In the beginning, all servers and containers are unmatched. Given a container c_i hosting x_{ij}, it will be first assigned to its most preferred container s_j, which can achieve the highest utility and have not rejected c_i yet. If s_j has sufficient capacity, it will accept c_i. Otherwise, it will sequentially reject less preferable containers, which is assigned to s_j previously. Whenever s_j rejects a container, it will update the *rejected-top* variable, which indicates the list of already rejected container by s_j. Furthermore, the other containers hosting corresponding tasks which are ranked lower than the *rejected-top* will remove this s_j from its *preferred-list* to *black-list*.

Algorithm 2 always outputs a stable matching in $O(M \times N)$, where M is the number of servers and N is the number of containers hosting tasks. Here we denote μ as the matching between containers hosting tasks and servers. We can prove the stability of the matching pairs by contradiction. Suppose that Algorithm 2 produces a matching μ with a blocking pair (c_i, s_j), i.e, there is at least one container $c\prime = A(s_j)$ worse than c_i according to l_j. Because c_i should have been proposed to s_j and rejected by it before. So $c\prime$ should been rejected by s_j, or s_j have been added to *black-list* of $c\prime$. So this matching output contradicts the assumption. Algorithm 2 will always output a stable matching with the complexity of $O(M \times N)$.

Algorithm 1 Policy Optimization Algorithm

Require: $\mathbb{W}, \mathbb{C}, \mathbb{F}, \mathbb{P}$
Ensure: Preference Matrix P(M × N)
1: let F represent the shuffle traffic flows for application
2: **for** $f_k \in F$ **do**
3: apply policy p_k on the flow f_k
4: construct the shuffle traffic flow path
5: return optimal shuffle path (c_{src}, w_{list}, c_{dst})
6: **for** i = 1 to $w_{list.len}$ **do**
7: **if** $p_{k.list[i]} \neq w_{list[i]}$ **then**
8: policy optimization: $p_{k.list[i]} \rightarrow w_{list[i]}$
9: **end if**
10: **end for**
11: P(s_{src}, $f_{i.src}$) ++
12: P(s_{dst}, $f_{i.dst}$) ++
13: update preference matrix P(M × N)
14: **end for**
15: Return P

5.3 Applying Solution in Hadoop

The slave nodes of Hadoop cluster are configurable to concurrently execute up to a particular number of Map (and Reduce) tasks. If the number of Map (or Reduce) task in the job exceeds the number of containers available, then Maps (or Reduces) are first scheduled to execute on all available containers and these Maps (or Reduces) form the first "wave" of tasks, and subsequent tasks form the second, third, and subsequent waves. In this paper, each map and reduce pair form a shuffle traffic flow, in which the container $f_{i.src}$ host map task will transfer its output to container $f_{i.dst}$ host reduce task. We characterize MapReduce task placement problem into two

Algorithm 2 Tasks Assignment Algorithm

Require: Matrix P(M×N), $\mathbb{X}$, $\mathbb{C}$
Ensure: containers-based task assignment A
1: obtain ranked order list l_i, where $s_i \in \mathbb{S}$
2: initialize the blacklist b_k, where $c_k \in \mathbb{C}$
3: initialize $\hat{A} = 0$
4: allocate containers $\mathbb{C}$ for tasks by x_{ik}
5: **while** $\exists\ c_k, A(\hat{c_k}) = 0$ **do**
6: obtain s_j where $argmax_{s \in S(c_k) \setminus b_k}\ \mathrm{U}(A(c_k) \rightarrow s)$
7: $A(\hat{c_k}) = s_j$;
8: **if** $\sum_{\forall c_k \in A(\hat{c_k})} r_k > q_j$ **then**
9: **repeat**
10: $c_m \leftarrow$ last container according to l_k
11: $A(c_m) = 0$
12: *rejected-top* $\leftarrow c_m$
13: **until** $\sum_{\forall c_k \in A(\hat{c_k})} r_k \leq q_j$
14: **end if**
15: **for** $c_n \in l_k$, where $c_n \leq$ *rejected-top* **do**
16: $b_n = b_n \bigcup s_j$
17: **end for**
18: **end while**
19: Perform tasks assignment
20: Update routing path

types: Map and Reduce initial placement and subsequent-wave placement.

5.3.1 Initial-wave Task Scheduling. For the case where both Map and Reduce tasks form the new waves, we should apply the Hit-Scheduler to optimize both $f_{i.src}$ and $f_{i.dst}$, minimizing the total shuffle delay. In these cases, the map and reduce tasks have been not assigned. We assume that they are randomly assigned in the beginning. Each map and reduce pair form a shuffle traffic flow. Because they have to transfer the map task's output to corresponding reduce task through the hierarchical topology. Under this assumption, we use the Hit-Scheduler to make the placement decision.

5.3.2 Subsequent-wave Task Scheduling. For the case multiple Map waves and one reduce wave, where Map tasks occur in multiple waves, while Reduce tasks tend to complete in one waves, we do not need to consider the optimization of the placement of Reduce task in the same reduce wave, but only optimize the placement of Map task. For the objective function, the destination of each shuffle traffic flow $f_{i.dst}$ is static. This problem can be interpreted as finding the optimal $f_{i.src}$. In these cases, we can fix the destination of each flow in TAA scheme, and greedily find the optimal placement of map tasks. In this stage, we should pair the Map tasks that have higher shuffle output with the physical servers which can achieve low delay in network traffic. We choose one of the solution among the all possible placement of Map tasks to achieve the lowest communication delay in that wave. The total algorithm complexity for these cases is $O(n^2)$.

6 IMPLEMENTATION ON HADOOP

In order to implement Hit-scheduler, we split our solution into offline and online phase. In the offline phase, we profile the shuffle data rate for each application and capture the topology architecture configuration in the cluster. In the online phase, we add a new class **mapred.job.topologyaware** to collect the information of task placement, embedding shuffle traffic flow, network architecture and cluster configuration from the offline phase. We modify three mechanisms based on the default Hadoop. We use the modified **DelayFecther**() to express the delay between two servers in the cluster. We develop a new **Hit − ResourceRequest** based on **ResourceRequest** to enable Hit-Scheduler to be aware of the network architecture. We design a new class **Hit − Scheduler** to implement the algorithm of the topology aware task assignment.

6.1 DelayFecther

We add a new function **Sleep**(**Delay**) into the original **Fecther** mechanism to obtain **DelayFecther**. The delay between two machines s_i and s_j is decided by the shuffle cost C(s_i, s_j) and corresponding bandwidth on the path B_{ij} , so the delay for the flow is *Delay* = $\frac{C(s_i,s_j)}{B_{ij}}$. With this component, we can mimic the performance degradation caused by hierarchical network architecture.

6.2 Hit-ResourceRequest

Essentially an application can ask for specific resource requests via the **ApplicationMaster** to satisfy its demand. The scheduler responds to a resource request by granting a container, which satisfies the requirements laid out by the **ApplicationMaster** in the initial **ResourceRequest**. To implement **Hit − ResourceRequest**, we specify *resource-name* as the preferred host for the specific task, which is the hostname of the preferred machine. We update the preferred *resource-name* for each task in **Hit − ResourceRequest** from the class file **mapred.job.topologyaware.taskdict**.

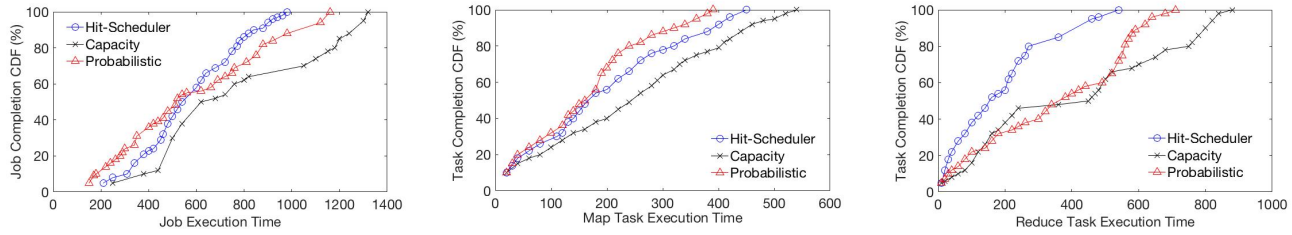
6.3 Hit-Scheduler

For the resource allocation, container is the successful result of the **ResourceManager**, which grants a specific request from a new component **Hit − ResourceRequest**. A Container grants rights to an application to use a specific amount of resources (e.g. memory, CPU) on a preferred host. According to the optimal task assignment via **Hit − ResourceRequest**, we use the **Hit − Scheduler** algorithm to implement our strategy. For each task, we assign resource by calling *getContainer(Hit-ResourceRequest, node)* if the task preferred container matches the current node with available resource. Then the **ApplicationMaster** has to take the preferred container and present it to the **NodeManager** managing the host, on which the container was allocated, to use the resources for launching corresponding tasks.

7 EVALUATION

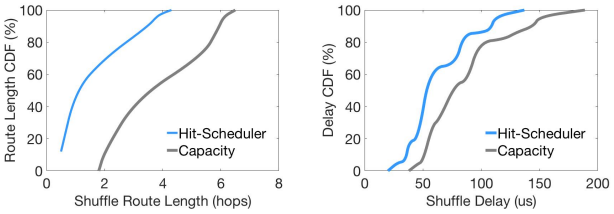
7.1 Testbed and Hierarchical Network

The evaluation testbed consists 9 nodes, each of which is configured with Intel Xeon(R) CPU E5-2630v4@2.20GHz x 20 and 32GB DDR3 memory, running Ubuntu 16.04 LTS operating system. One node serves as the *master* node, and all the 8 nodes serve as *slaves*. The nodes are connected by switches with 16GbE ports. We evaluate the performance of our proposed strategy with Hadoop YARN by using the benchmarks from Purdue MapReduce Benchmarks Suite



(a) (b) (c)

Figure 6: CDF of Job Completion Times, Map and Reduce Task Execution Times under Various Policies.



(a) Average Route Length (b) Average Shuffle Delay

Figure 7: Comparison on Shuffle Traffic Flow

(PUMA) [3]. Based on the size of shuffled intermediate data, we characterize the benchmarks as Shuffle-heavy, Shuffle-medium or Shuffle-light in Table 1. The table also shows the percentage of jobs of each type in the workload.

We use Mininet [8] to create a realistic virtual network, running real kernel, switch and application code for hierarchical topology. We create a network with a tree topology of depth 3 and fanout 8 (i.e. 64 hosts connected to 10 switches), using Open vSwitch switches under the control of the OpenFlow reference controller. Through these controllers, we can manage the switches and optimize the corresponding policies. Under this simulation setup, a collection of containers will communicate with each other through the network. For each shuffle traffic flow, it should originate from the source container which hosts map task, and terminate at the destination container which hosts corresponding reduce task. And we implement a centralized controller to collect all the network information and perform the policy optimization. In order to compare our strategy against other schemes, we have also implemented Capacity and Probabilistic Network-Aware scheduling schemes to assign the tasks, and we use D-ITG [4] applying TCP protocol to measure the average route length and shuffle traffic delay at packet level accurately. To verify the scalability of our proposed Hit-Scheduler under various architectures, we implement three other different network architectures, i.e., FAT-Tree [20], BCube [13] and VL2 [12].

We evaluate the performance of our strategy in terms of the job completion time and the improvement on shuffle flow. To verify Hit-Scheduler algorithm's scalability, we compare our performance under different hierarchical network architectures, network bandwidths, job types and job numbers with Capacity scheduler and Probabilistic Network-aware scheduler [26].

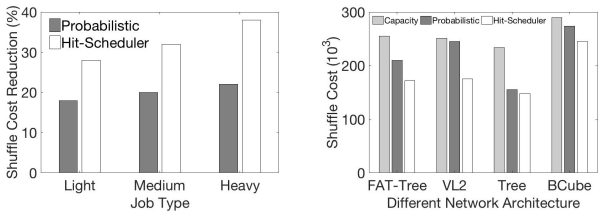
7.2 Improvement on Job Completion Time

Figure 6(a) shows that our strategy saves a significant amount of time to complete MapReduce jobs. Specifically, Hit-Scheduler outperforms Capacity scheduler 28% and Probabilistic Network-Aware scheduling strategy 11% in terms of the job completion time reduction, respectively. The reason is that our strategy takes the hierarchical network architecture into account and jointly optimizes the assignment of map and reduce tasks to reduce the cost on shuffle phase. The results in Figure 6(b) also show that Capacity scheduler performs better in the beginning stage as more resources are allocated to Map tasks for improving the resource utilization. Probabilistic Network-Aware scheduler achieves better performance than Hit-Scheduler during map phase due to the fact that our strategy does not consider the remote access for map input. However, the overall job completion time is better than Probabilistic Network-Aware scheduler, which verifies our assumption that shuffle traffics play a more important role when running MapReduce applications in the cloud. Figure 6(c) further confirms that Hit-Scheduler achieves significant improvement in terms of Reduce task execution times compared to the other strategies.

Figure 7(a) illustrates the effectiveness on reducing the length of the average routing path. It shows that Hit-Scheduler reduces the average route path from 6.5 hops to 4.4 hops compared with Capacity Scheduler, which achieves nearly 30% improvement. It is because Capacity Scheduler is unaware of the network architecture, resulting in longer flow route path which is caused by failing to consider optimizing the traffic in the network. As a result of shorter route path, Figure 7(b) shows that Hit-Scheduler reduces the average shuffle delay from 189 *us* to 131 *us*. Reducing the shuffle traffic flow delay play an important role in improving the job completion time.

7.3 Impact of Network Architecture and Job Characterization

To illustrate the effectiveness of our strategy for different workloads, we compare the shuffle cost reduction gained by Hit-Scheduler with Probabilistic Network Aware Scheduler under the same Tree network architecture. Figure 8(a) shows that for a single job, the reduction on shuffle cost increases to 38% for shuffle-heavy workload, while 21% for Probabilistic Network Aware Scheduler. The results also demonstrate that the improvements for the shuffle-light and



(a) Different Job Types (b) Different Networks

Figure 8: Impact of Network Architecture and Job Characterization

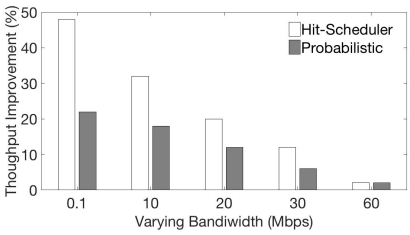


Figure 9: Sensitivity to Network Bandwidth

shuffle-medium datasets are not as apparent as the shuffle-heavy dataset because they require less data shuffle traffic.

Figure 8(b) shows the shuffle traffic cost of the shuffle-heavy workload under four different network architectures. It illustrates that Map-and-Reduce style fits the Tree network architecture very well because it results in less shuffle cost. Under all of these different architectures, Hit-Scheduler outperforms Probabilistic Network Aware Scheduler and Capacity Scheduler about 19% and 32% in terms of the shuffle cost. Though Probabilistic Network Aware Scheduler takes the network topology and bandwidth into consideration, it cannot handle some complex topologies, e.g, VL2 shown in the figure. This is because Probabilistic Network Aware Scheduler assumes that the network cost is static and fixed among all nodes in the cloud and does not take the limited bandwidth into consideration. Compared with Probabilistic Network Aware Scheduler, Hit-Scheduler can effectively support various complex topologies and correspondingly achieves better scalability.

7.4 Impact of Bandwidth and Job Numbers

We further implement a large-scale simulation to evaluate the network policy, where we set the the total number of nodes to be 512, which are connected via Tree network. Figure 9 shows the throughput improvement achieved by Hit-Scheduler and Probabilistic Network Aware scheduler compared to Capacity scheduler under varying bandwidth from 0.1 Mbps to 60 Mbps. It demonstrates that Hit-Scheduler significantly outperforms Probabilistic Network Aware scheduler especially when given the limited bandwidth. The improvement can be nearly 48% while the bandwidth is limited to 0.1Mbps. This is due to the fact that Probabilistic Network Aware scheduler assumes the communication cost between two nodes

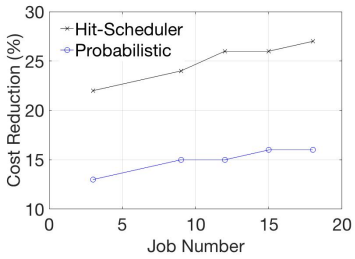


Figure 10: Sensitivity to Job Numbers

is static and the routing path is single-one path, which is simply decided by the number of switches it will traverse. Compared with Probabilistic Network Aware scheduler, there are more opportunities for Hit-Scheduler to improve the throughput with the limited bandwidth. To evaluate the sensitivity of Hit-Scheduler under various job numbers, we investigate the overall cost reduction while varying the number of jobs ranging from 3 to 18. Figure 10 shows that the cost reduction by applying Hit-Scheduler increases significantly with the number of jobs at the beginning stage and then increases slowly when the number is greater than 12. The reason is that parallel running more jobs may provide more opportunities to optimize the network traffic and task assignment to reduce traffic cost. However, as the job number increasing, the cost reduction is not apparent as the beginning stage since the bandwidth may approaching the bottleneck. In comparison, the shuffle cost reduction gain by applying Probabilistic Network Aware scheduler is relatively stable, i.e., around 15% and increases slowly.

8 RELATED WORK

There are many network architectures [15], [27] to improve cluster bisection bandwidth. However, most techniques require specialized hardware or communication protocols. Recent development in SDN enable more flexible policy deployment over the network. *SIMPLE* [25] is an SDN-based policy enforcement scheme to steer traffic in data center according to policy requirements. FlowTags [10] is proposed to leverage SDN's global network visibility and guarantee correctness of policy enforcement. However, they are not fully designed with computation in consideration, and may put the application running in the data center on the risk of policy violation or performance degradation.

A number of researches have been proposed to improve the scheduling for MapReduce jobs. Techniques like delay scheduling [29] and Quincy [17] try to improve the locality of tasks by scheduling them close to their input. Zaharia et al. [30] developed a scheduling algorithm called LATE that tried to improve the response time of short jobs by executing duplicates of some tasks in a heterogeneous system. However, these techniques do not guarantee locality for shuffle stages. Purlieus [24] and CAM [21] achieve locality via synergistic placement of virtual machines and input data. ShuffleWacther [2] and iShuffle [14] try to improve the locality of the shuffle by scheduling both maps and reducers on the same set of racks. iShuffle [14] tries to improve job performance by decoupling shuffle from reduce tasks and optimizing the scheduling of reduce

tasks by automatic balancing workload. However, All But these scheduling schemes do not explicitly take into account the cost caused by network for deciding the placement of tasks, which may lead to excessive latency in shuffling and degrade the performance of job execution.

Recently, Corral [18] couples the placement of data and computation, improving data locality for all stages of a job. Another important category is network scheduling [6] [5], whose idea is to schedule the flows or groups of flows at shared links, based on given task placement to minimize the flow completion time. The limitations are that the source or destination of each flow is independently decided by the task scheduler and not necessarily optimal. A transmission cost-based scheduling method, Probabilistic Network-Aware Scheduler [26] was proposed considering the network topology and link bandwidth. However, they assume that network cost among nodes is static and the bandwidth for shuffle flow is fixed. Actually, the computation running in the cloud and transmission on the network will affect each other. Unlike previous works, we propose a task scheduling scheme taking both the dynamic network policy and computation into account.

9 CONCLUSION

In this paper, we focus on jointly optimizing task scheduling and network policy management in the cloud with hierarchical network architecture. We have proposed and developed a hierarchical topology aware MapReduce scheduler to minimize overall data traffic cost and hence to reduce job execution time. The main technical novelty of Hit-scheduler lies in the integration of dynamic computing and communication resources in hierarchical clouds. As demonstrated by the modeling, optimization and experimental results based on the testbed implementation, Hit-scheduler can improve job completion time by 28% and 11% compared to Capacity Scheduler and Probabilistic Network-Aware scheduler, respectively. Our simulations further demonstrate that Hit-scheduler can gain the traffic cost by 38% at most and improve the average shuffle flow traffic time by 32% compared to Capacity scheduler.

ACKNOWLEDGMENTS

This research was supported by the University of North Carolina at Charlotte.

REFERENCES

[1] Faraz Ahmad, Srimat T Chakradhar, Anand Raghunathan, and TN Vijaykumar. 2012. Tarazu: optimizing mapreduce on heterogeneous clusters. In *Proc. of ACM SIGARCH*.

[2] Faraz Ahmad, Srimat T Chakradhar, Anand Raghunathan, and TN Vijaykumar. 2014. ShuffleWatcher: Shuffle-aware Scheduling in Multi-tenant MapReduce Clusters. In *Proc. of USENIX ATC*.

[3] Faraz Ahmad, Seyong Lee, Mithuna Thottethodi, and TN Vijaykumar. 2012. Puma: Purdue mapreduce benchmarks suite. (2012).

[4] Alessio Botta, Alberto Dainotti, and Antonio Pescapè. 2012. A tool for the generation of realistic network workload for emerging networking scenarios. *Computer Networks* 56, 15 (2012), 3531–3547.

[5] Wei Chen, Jia Rao, and Xiaobo Zhou. 2017. Preemptive, low latency datacenter scheduling via lightweight virtualization. In *Proc. of ATC*. USENIX.

[6] Dazhao Cheng, Yuan Chen, Xiaobo Zhou, Daniel Gmach, and Dejan Milojicic. 2017. Adaptive scheduling of parallel jobs in spark streaming. In *Proc. of INFOCOM*. IEEE.

[7] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. 2013. Leveraging endpoint flexibility in data-intensive clusters. In *Proc. of ACM SIGCOMM*.

[8] Rogério Leão Santos De Oliveira, Ailton Akira Shinoda, Christiane Marie Schweitzer, and Ligia Rodrigues Prete. 2014. Using mininet for emulation and prototyping software-defined networks. In *Proc. of IEEE COLCOM*.

[9] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[10] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C Mogul. 2014. Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags. In *Proc. of USENIX NSDI*.

[11] David Gale and Lloyd S Shapley. 1962. College admissions and the stability of marriage. *The American Mathematical Monthly* 69, 1 (1962), 9–15.

[12] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: a scalable and flexible data center network. In *Proc. of ACM SIGCOMM*.

[13] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. 2009. BCube: a high performance, server-centric network architecture for modular data centers. *In ACM SIGCOMM Computer Communication Review* (2009).

[14] Yanfei Guo, Jia Rao, Dazhao Cheng, and Xiaobo Zhou. 2017. ishuffle: Improving hadoop performance with shuffle-on-write. *In IEEE Transactions on Parallel and Distributed Systems* 28, 6 (2017), 1649–1662.

[15] László Gyarmati and Tuan Anh Trinh. 2010. Scafida: A scale-free network inspired data center architecture. *Proc. of ACM SIGCOMM* (2010).

[16] Americas Headquarters. 2007. Cisco Data Center Infrastructure 2.5 Design Guide. In *Cisco Validated Design I*. Cisco Systems, Inc.

[17] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: fair scheduling for distributed computing clusters. In *Proc. of ACM SIGOPS*.

[18] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. 2015. Network-aware scheduling for data-parallel jobs: Plan when you can. *Proc. of ACM SIGCOMM* (2015).

[19] Hans Kellerer, Ulrich Pferschy, and David Pisinger. 2004. Introduction to NP-Completeness of knapsack problems. In *Knapsack problems*. Springer, 483–493.

[20] Charles E Leiserson. 1985. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE transactions on Computers* 100, 10 (1985), 892–901.

[21] Min Li, Dinesh Subhraveti, Ali R Butt, Aleksandr Khasymski, and Prasenjit Sarkar. 2012. CAM: a topology aware minimum cost flow based resource manager for MapReduce applications in the cloud. In *Proc. of ACM HPDC*.

[22] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. 2010. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *Proc. of IEEE INFOCOM*.

[23] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. 2009. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *Proc. of ACM SIGCOMM*.

[24] Balaji Palanisamy, Aameek Singh, Ling Liu, and Bhushan Jain. 2011. Purlieus: locality-aware resource allocation for MapReduce in a cloud. In *Proc. of IEEE/ACM SC*.

[25] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. 2013. SIMPLE-fying middlebox policy enforcement using SDN. In *Proc. of ACM SIGCOMM*.

[26] Haiying Shen, Ankur Sarker, Lei Yu, and Feng Deng. 2016. Probabilistic network-aware task placement for mapreduce scheduling. In *Proc. of IEEE CLUSTER*.

[27] Liang Tong, Yong Li, and Wei Gao. 2016. A hierarchical edge cloud architecture for mobile computing. In *Proc. of IEEE INFOCOM*.

[28] Abhishek Verma, Brian Cho, Nicolas Zea, Indranil Gupta, and Roy H Campbell. 2013. Breaking the MapReduce stage barrier. *Cluster computing* 16, 1 (2013), 191–206.

[29] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proc. of ACM Eurosys*.

[30] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. 2008. Improving MapReduce performance in heterogeneous environments.. In *Proc. of USENIX OSDI*.