Joint Optimization of MapReduce Scheduling and Network Policy in Hierarchical Data Centers

Donglin Yang, Dazhao Cheng, Wei Rang and Yu Wang

Abstract—As large-scale data analytic becomes norm in various industries, using MapReduce frameworks to analyze ever-increasing volumes of data will keep growing. In turn, this trend drives up the intention to move MapReduce into multi-tenant clouds. However, the application performance of MapReduce can be significantly affected by the time-varying network bandwidth in a shared cluster. Although many 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 testbed 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 reduce the traffic cost by 38% at most and the average shuffle flow traffic time by 32% compared to Capacity scheduler. In this manuscript, we have extended Hit-scheduler to a decentralized heuristic scheme to perform the policy-aware allocation in data center environments. Many existing centralized approximation approaches are too complex and infeasible to implement over a data center, which typically include large amounts of servers, containers, switches and traffic flows. In the extension, we have designed a decentralized heuristic scheme to perform the Policy-Aware Task (PAT) allocation by using existing centralize algorithm to approximately maximize the total gained utility. Finally, the simulation based experimental results show that the proposed PAT policy reduces the communication cost by 33.6% compared with the default scheduler in data centers.

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

1 Introduction

MapReduce is based on an implicit parallel programming model that provides a convenient way to express distributed computations, particularly for large-scale data sets. It is originally designed for petabytes or multi-terabytes of data processing, which has been widely accepted by industries [1]. 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 data center. 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 [2] and flow-based scheduling [3] 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) [4]. 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 [5], [6] have shown the shuffle traffic mostly dominates the overall performance of MapReduce jobs compared to the remote map traffic. Thus, a few recent studies [5], [7] 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 [8], [9] have focused primarily on exploiting software defined networking and network function virtual-

- *D. Yang, D. Cheng, W. Rang are with the Department of Computer Science, University of North Carolina at Charlotte, NC, 28223. E-mail: dyang33, dazhao.cheng, wrang@uncc.edu.*
- *Y. Wang is with the Department of Computer and Information Sciences, Temple University, PA, 19122. E-mail: wangyu@temple.edu.*
- *D. Cheng is the corresponding author.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2961653, IEEE Transactions on Cloud Computing

ization 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. Network management is a complex task but overlooked by research in the field of big data analytics. Providing a balanced services while ensuring high performance is a major challenge when moving MapReduce into the cloud [10]. In data center, packet transmission is managed by a collection of networking policies to ensure performance. Network policies enforce that the packet transmission follows a sequence of specified middleboxes. Traditionally, cloud providers have to carefully craft routing in order to satisfy the policy demand [11]. Recently, to efficiently manage middleboxes, Software-Defined Networking [12] has been proposed to enforce networking policies. SDN abstracts a global view of network so as to grammatically ensure the correctness of packet transmission.

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 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 reduce the traffic cost by 38% at most and reduce the average shuffle flow traffic time by 32% compared to Capacity scheduler.

TABLE 1
Benchmarks Characterization

Shuffle Types	Benchmark Proportion and Type
Heavy	terasort(5%), index(10%), join(10%), sequence count(10%), adjacency(5%)
Medium	inverted-index(10%), term-vector(10%)
Light	grep(15%), wordcount(10%), classification(5%), histogram(10%)

A preliminary version of the paper appeared in [13]. In this manuscript, we have extended Hit-scheduler to a decentralized heuristic scheme to perform the policy-aware allocation in data center environments. To enforce these policies, we use SDN to implement the correct sequence to data packet when scheduling MapReduce tasks. Specifically, policy-aware task allocation problem can be treated as a restricted version of the Generalized Assignment Problem [14], which has been proven APX-hard. Many existing centralized approximation approaches are infeasible to implement over a data center environment, which typically include large amounts of servers, containers, switches and traffic flows. The computation complexities of those algorithms are unacceptable for moving MapReduce applications into data center, especially considering billions of traffic paths [15]. In the extension, we have designed a decentralized heuristic scheme to achieve the policy-aware allocation, approximately maximize the total gained utility, and adhere to policy demand meanwhile. Simulation based experimental results show that the proposed PAT policy reduces the communication cost by 33.6% compared with the default scheduler in data centers.

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 describes the policy-aware scheduling in data centers. Section 7 gives details on system implementation. Section 8 presents experimental results. Section 9 reviews related work. Section 10 concludes the paper.

2 BACKGROUND AND MOTIVATION

2.1 Performance Impact due to Data Shuffle

When running a MapReduce job, all Map and Reduce tasks are typiclly scheduled to maximize concurrency (i.e., occupy the whole 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 data shuffle operation results in an all machines to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2961653, IEEE Transactions on Cloud Computing

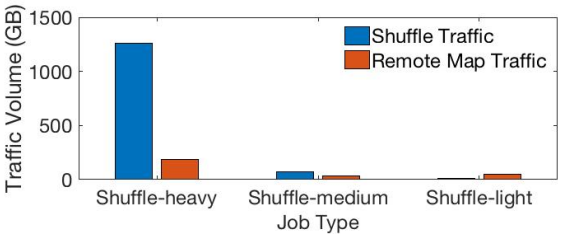
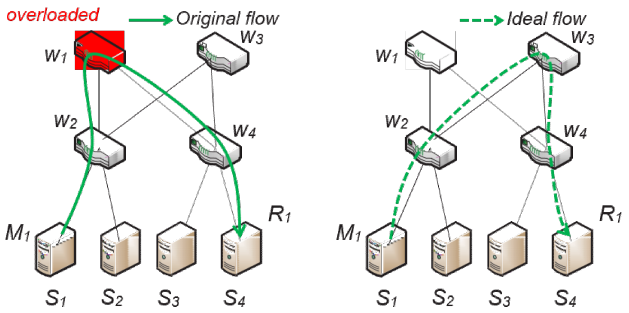


Fig. 1. Traffic Volume During Shuffle Phase



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

Fig. 2. Workflow under hierarchical networks.

all machines communication, which stresses the limited network bandwidth. In particular, concurrently running multiple shuffle-heavy jobs may significantly increase the pressure of network bandwidth [16]. 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 task output is known.

To identify the impact of shuffle communication, we analyze a few representative workloads consisting of benchmark drawn from Apache Hadoop release [17]. The benchmarks are characterized as heavy shuffle, medium shuffle and light shuffle, 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 input traffic for heavy shuffle, medium shuffle and light shuffle jobs, respectively. The result shows the shuffle data volume of heavy shuffle jobs is a significant contributor (>75%) of the total communication traffic, and the contribution of the remote Map input 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

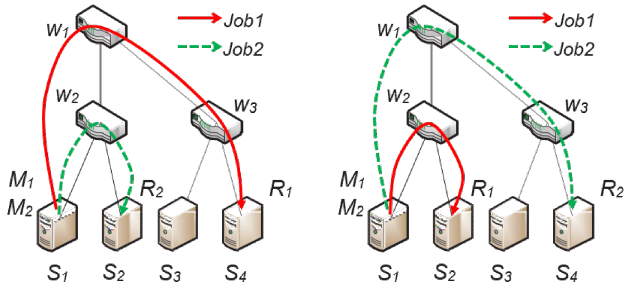
Today's cloud operators greatly extend the popular three-tier network architecture [17] in data centers. At the access tier (i.e., bottom level), each machine connects to one or two access switches. At the aggregation tier, each access switch connects to one (or two) switches. Finally, every aggregation switch is connected 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 [18], PortLand [19] and BCube [20]. In contrast, most existing task scheduling policies do not consider the possible impact by these hierarchical network architectures, or just set up a simple model for data intensive applications. Furthermore, the network is oversubscribed from the bottom layers to the core ones in datacenters. To efficiently avoid network congestion in the core layers, it's better to enforce the data packet to route over edge layers as much as possible [21]. As a result, those virtual machines and containers who exchange data packets intensively should be collocated with the edge layers.

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 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 `DelayFetcher()`, 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].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2961653, IEEE Transactions on Cloud Computing



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

Fig. 3. Assigning Job 1 (M_1, R_1) and Job 2 (M_2, R_2).

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. The data volume to be exchanged between M_2 and R_2 is much larger than another pair of task. Without policy awareness, the default allocation decision would cost precious bandwidth on core routers. 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 and data volume. Then the total required bandwidth on the core router can be reduced. The above observations demonstrate that the shuffle-heavy data transmission does not correspond to the bandwidth-efficient 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. The policy-aware scheduling has to find the optimal allocation while meeting the network bandwidth and policy demand.

3 PROBLEM MODELING

In this section, we formulate the problem by moving MapReduce applications into a multi-tier data center which is typically structured under a multi-root tree topology, such as VL2 [18].

3.1 Formulation

We consider a multi-root tree topology, such as canonical [23], as a typical multi-tier data center network in the formulation. 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.

TABLE 2
Notation Summary.

c_i	the i-th container
s_i	the i-th server
r_i	the resources requirement of i-th container
f_i	the i-th traffic flow
p_i	the policy for the i-th flow
w_i	the i-th middlebox
x_{ij}	the indicator for task
C_k	the shuffle cost for the k-th policy
q_j	the available resource of the j-th server
U	the network utility
A	the allocation of tasks

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 this work, policies are configured through the policy controller (e.g SDN) to govern traffic related to the MapReduce application running in the cloud. 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) to be the server which hosts container c_i and A(p_k) to be 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 (important notation is summarized in Table 2.):

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

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.

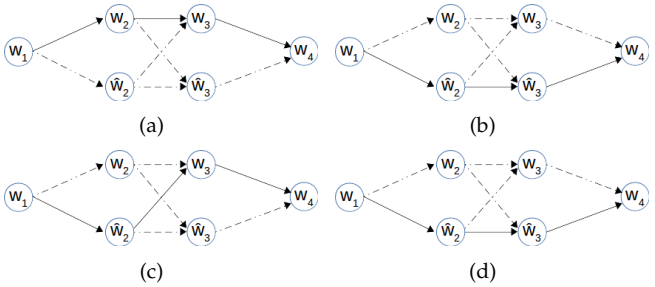


Fig. 4. Separated Optimization on Policies

The above mentioned TAA problem is NP-Hard, we will show that the Multiple Knapsack Problem (MKP) [24] 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 proof can be found in the preliminary version.**

5 Solution

In this section, we introduce a separable scheme to efficiently solve our problem.

5.1 Optimization on Network 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 $w_1 \rightarrow$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2961653, IEEE Transactions on Cloud Computing

$\hat{w}_2 \rightarrow \hat{w}_3 \rightarrow w_4$, 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:

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

Consider a container c_i hosting j_{th} map task, where x_{ij}^m=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_{ij}^r, 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. Please refer to the preliminary study [13] for solving the above separable optimization problem based on a typical many-to-one stable matching, i.e., Stable Marriage Problem.

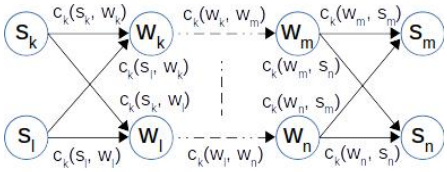


Fig. 5. Shuffle Traffic Flow Path

5.2 Applying Solution in Hadoop

The slave nodes of a Hadoop cluster can be configured to concurrently support up to a certain number of Map or Reduce tasks. If the number of Map or Reduce tasks exceeds the number of available containers, Map/Reduce tasks are first assigned to execute on all free containers. These tasks form the first "wave" of MapReduce 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 types: Map and Reduce initial placement and subsequent-wave placement.

5.2.1 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.

5.2.2 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 wave, 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2961653, IEEE Transactions on Cloud Computing

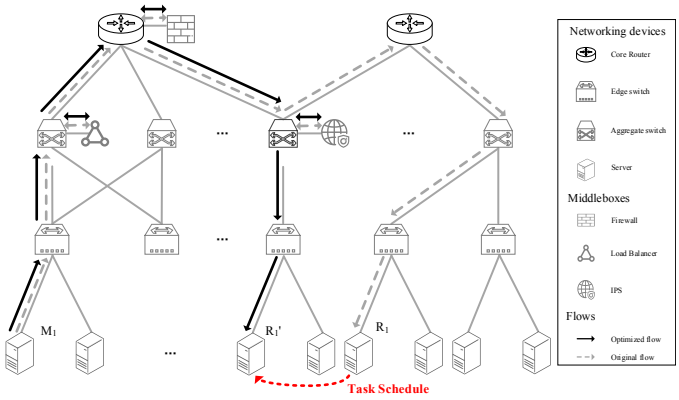


Fig. 6. Data traffic flows traversing different sequences of switches and middleboxes in a data center environment.

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 $\mathcal{O}(n^2)$.

6 Policy-Aware Scheduling in Clouds

In this section, we describe moving MapReduce into DC to demonstrate that a policy-unaware strategy could cause performance degradation.

Cloud data centers offer flexibility and elasticity for users to utilize large amounts of computational and storage resources to satisfy their requirements. In particular, running MapReduce in data centers allows enterprises to efficiently analyze large amounts of data without building their own infrastructures [25]. However, moving MapReduce into Data Centers (DC)s may incur several challenging problems, such as security and load balancing issues, while achieving high application performance from the view of cloud operators. Data-intensive applications like MapReduce over DC networks typically lead complex communication behaviors which are managed by some network policies regarding performance and security [26]. To implement such policies, cloud operators typically apply various network appliances or "control middleboxes", e.g., load balancers, traffic shapers, Intrusion Detection and Prevention Systems [25] in DCs. As a result, such a network environment leads to various new constraints while assigning tasks in DCs. In this section, we extend the proposed Hit Scheduler to a decentralized heuristic scheme to perform the policy-aware allocation in DC environments.

6.1 Motivating Example

It is necessary to propose a policy-aware scheduling in DC environments to avoid the unexpected failure and possible performance degradation. Figure 6 depicts a motivation example by describing a running MapReduce application in DC. The example uses a common network architecture, Fat-Tree [27], in DC, which is composed of a number of network components, such as networks switches and middleboxes. Firewall (FW) will control the coming network traffic based on predetermined security rules and monitor the network environments. Intrusion Prevention Systems (IPS) are configured with rules to monitor networks or systems for malicious activity or policy violations. Any malicious activity or violation will be typically reported to the administrator. These devices will also detect and evaluate the performance of each middlebox. Load Balancers (LB) will forward traffic flows from one host to other hosts, aiming to optimize resources use and maximize throughput. In data center, policies are configured by using a tuple, which consists of the sequence of middleboxes. The following policy is configured to govern how the output from Map function will be traversed to the destination in the cloud when running the application:

- p = {c_1, c_2, 1,001, 1,002, TCP} $\rightarrow$ {*LB*, *FW*, *IPS*}.

For this policy p, container c_2 hosting Reduce task will communicate with the specific server to fetch the Map output results. The traffic flow will first traverse by *LB* for load-balancing, after which it will be checked by *FW* in the core router for security. Before reaching the destination, it will be forwarded to the *IPS* located in the aggregation layer. Consider the allocation of c_2 in the above example application. Without network-aware strategies, c_2 may be hosted by server s_3. A large volume of intermediate data will be exchanged between container c_1 and c_2, which requires precious bandwidth in the core layer. In order to allocate tasks in near distance and keep the traffic in edge layers, c_2 can be scheduled to s_1 so that the containers hosting Map and Reduce task are close to each other, neglecting the requirement of policies. However, this allocation decision will increase the routing length of traffic flow and occupy much more network resources. Because according the policy configurations p, the traffic flows between the containers c_1 and c_2 have to be forwarded through *LB*, *FW* and *IPS* sequentially. The container c_2 hosting Reduce task should be scheduled to the server s_2 so as to reduce the network cost between container c_2 and *IPS* by taking both the network policy configurations and traffic patterns into consideration. Obviously, a policy-aware task scheduling strategy will find an optimal placement decision while meeting the demand of both network bandwidth and networks policies. In the following sections, we will corporate the policy-aware task scheduling into our existing network-aware allocation problem.

6.2 Policy-Aware Task Allocation Problem

In this section, we design a decentralized heuristic solution to perform policy-aware and network-aware task scheduling in the data center.

Given a set of tasks, servers and policies, $m^{in}(c_i)$ is defined as ingress middleboxes of all outgoing traffic flows from container c_i, i.e., $m^{in}(m_i)$ = {m_j| m_j=p_k^{in}, $p_k.src = c_i$}. Similarly, $m^{out}(c_i)$ = {m_j| m_j=p_k^{out}, $p_k.dst =$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2961653, IEEE Transactions on Cloud Computing

$c_i\}$ is defined as egress middlebox of all incoming flows to the destination container c_i. We further define $S(m_k)$ as a set of servers that can retrieve the middlebox m_k from the policy controller. To meet the requirements of policies configurations, we can obtain the feasible servers that a container c_i hosting task can be allocated as following:

$$S(c_i) = \bigcap_{m_k \in m^{in}(c_i) \cup m^{out}(c_i)} S(m_k). \tag{12}$$

$S(v_i)$ will be all the possible destination servers for container c_i, which have enough capacity. Here, we denote a vector R_i to represent the physical resource requirements of container c_i. R_i can have three dimensions such as CPU, RAM and I/O operations. Correspondingly, we denote H_j to represent the amount of resources provisioned by the server s_j. Obviously, $R_i \leq H_i$ can ensure that all the requested resources of s_i can be satisfied to host the allocated container c_i. We also define A to be a possible allocation of task. In particular, $A(c_i)$ will be the server which hosts the container c_i. Correspondingly, $A(s_j)$ will be a set of containers hosted by the server s_j. If we reschedule a container c_i from $A(c_i)$ to a server $\hat{s}$, the possible hosting servers for c_i can be characterized as following:

$$S_i = \{\hat{s} | (\sum_{s_k \in A(\hat{s})} R_k + R_i) \leq H_i, \forall \hat{s} \in S(c_i)/A(c_i)\}. \tag{13}$$

We define the total network cost as $C_i(s_i)$, which is caused by c_i among s_j and $m^{in}(c_i) \cap m^{out}(c_i)$, where s_j = $A(v_i)$

$$C_i(s_i) = \sum_{p_k \in P(c_i,*)} C_k(c_i, p_k^{in}) + \sum_{p_k \in P(*,c_i)} C_k(c_i, p_k^{out}). \tag{14}$$

The improved utility by adjusting $A(v_i) \rightarrow \hat{s}$ can be measured as:

$$U(A(c_i) \rightarrow \hat{s}) = C_i(A(c_i)) - C_i(\hat{s}). \tag{15}$$

The problem can be formulated as maximizing the total utility by optimizing the allocation $\hat{A}$:

$$\begin{aligned} &max\ U_{A \rightarrow \hat{A}} \\ &s.t.\ U_{A \rightarrow \hat{A}} > 0. \end{aligned} \tag{16}$$

The Policy-Aware Task allocation (PAT) problem is NP-Hard, which can be mapped into an APX-hard Generalized Assignment Problem (GAP) [14]. In a cloud data center environment, there are thousands or millions of physical machines, containers, network devices and traffic flows, which makes it extremely hard to solve the problem using existing heuristic algorithms. we will show that the Multiple Knapsack Problem (MKP) [24] can be reducible to the PAT problem in polynomial time by introducing the non-polynomial complexity. The multiple knapsack problem [24] 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.

We start with performing the mapping from MKP to a special case of our PAT problem. Take a scheduling decision A_0 as example, in which the PAT is to find the optimal solution $\hat{A}$ which can maximize the total network utility $U_{A \rightarrow \hat{A}}$. $S\prime$ = $\mathbb{S} \backslash \{s_0\}$ is defined to be a feasible set of servers which can host the container. For each container c_i to be allocated, we assume that the network cost introduced by c_i on all the possible servers are the same, which means $C_i(\hat{s}) = \delta_i, \forall \hat{s} \in S\prime$, where δ_i is a fixed value. Then, we treat each task to be an item with size R_i which will be allocated to a knapsack. The profit of assigning container c_i can be U(A(c_i))$\rightarrow \hat{s}$=$C_i(A(c_i))$-δ-$C_m(A(c_i))$. The PAT problem is be to obtain the optimal allocation of all flows to the corresponding intermediate middleboxes and a feasible allocation of tasks to servers $S\prime$, while maximizing the total network utility in the same time. Therefore, the MKP problem can be reduced to the PAT problem in polynomial time and hence the PAT problem is NP-Hard.

6.3 Policy-Aware Task Allocation Algorithm

Algorithm 1 PAT-task for c_i

```
initialize L= 0;
Optimization(c_i, L)
loop
    inf ← synMsg()
    if inf.type==reject then
        L = L ∪ {inf.update}
        Optimization(c_i, L)
    end if
    if inf.type==accept then
        updateMsg(optimize, inf.update, R_i)
        perform optimization: v_i → s
    end if
end loop
function Optimization(c_i, L)
    s_0 ← A(c_i)
    S_i ← possible candidates;
    X ← arg max_{x∈S_i\L} U(A(c_i) → x)
    if X≠0 and s_0 ∉ X then
        s ← candidate with enough capacity;
        updateMsg(request, s, R_i)
    end if
end function
update networking route.
```

The computation times of using the existing heuristic algorithms [14] to approximate the optimization problem (16) is unacceptable, especially when this problem

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2961653, IEEE Transactions on Cloud Computing

Algorithm 2 PAT-server for s_j

```
1: loop
2:     inf ← synMsg()
3:     if inf.type == request then
4:         c_i = inf.update
5:         R_i = inf.resource
6:         if Σ_{c_k∈A(s_j)} R_k+R_i leq H_j then
7:             updateMsg(accept, c_i)
8:         elseupdateMsg(reject, c_i)
9:         end if
10:    end if
11:    if inf.type == optimize then
12:        if Σ_{c_k∈A(s_j)} R_k+R_i leq H_j then
13:            update resources.
14:        else
15:            upsateMsg(reject, c_i)
16:        end if
17:    end if
18: end loop
```

involves thousands or millions of physical machines, containers, network devices and traffic flows [15]. In this section, we propose a decentralized heuristic algorithm to obtain the optimal allocation decision considering the configurations of policies. Algorithm 1 and 2 show procedures of the decentralized algorithms in terms of tasks and servers respectively. PAT-task is designed for tasks to find the feasible servers which can maximize the utility under constraints of resources and policies. Correspondingly, PAT-server will control the NodeManagers on servers to decide which request can be satisfied considering residual resources. PAT-task and PAT-server will synchronize the allocation information with each other. The API $updateMsg(type, dst, res)$ will update the specific device type and resource information to the destination server. The API $synMsg()$ will synchronize the source allocation decision. The variable $request$ indicates that a task requests the feasible server. Correspondingly, a server can answer the request with an $accept$ or $reject$, while considering both the available capacities and the demand of tasks. If the request is satisfied, the server will confirm the acceptance.

From the view of PAT-task algorithm, it will first try to find the feasible candidates greedily to improve the network utility, as shown in the line 4-7. The function $Optimization()$ is to maximize utility by iterating over all the possible candidates for c_i. To avoid the repeating request for the same servers, we will maintain a blacklist L. The function $Optimization()$ will always pick the server which have the most available resources as candidate for allocation, i.e., line 15-20. If a candidate s accepts c_i's request, then container c_i hosting task will be allocated to s.

Correspondingly, the PAT-server algorithm working on server s_j will always synchronize with PAT-task to obtain requests. PAT-server will answer the request with an $accept$ if this server has enough available capacities to host the task c_i. Or it will return a $reject$ message in the line 15. If this server accepts the request, servers s_j will update its residual resources after considering hosting c_i. The above PAT-task and PAT-server can reduce the network cost and approach a stable state in the end.

Proof. Our preliminary study [13] has shown that the optimization of traffic policies and task assignment can be performed independently. The cost of each task c_i can be obtained by considering the hosting server and related middleboxes in $m^{in}(c_i)$ and $m^{out}(c_i)$. After introducing the policies configurations, the allocations of tasks are still independent. Meanwhile, when algorithms implement optimization, i.e. $A(c_i)\rightarrow s$, the improved utility will be always positive, i.e., $U(A(c_i) \rightarrow s)>0$. This means that the network cost will be always decreased while optimizing the allocation of tasks among feasible servers. The amount of gained utility will vary after each optimization step. These two algorithms 1 and 2 will finally reach a stable state in a number of steps.

7 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 **DelayFetcher**() 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.

7.1 DelayFetcher

We add a new function **Sleep**(**Delay**) into the original **Fecther** mechanism to obtain **DelayFetcher**. 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.

7.2 Hit-ResourceRequest

A MapReduce application could ask specific computing resource requests to satisfy its demand via the **ApplicationMaster**. The task scheduler then gives responses to these resource requests by delivering some containers, which aims to satisfy 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**.

7.3 Hit-Scheduler

The container allocation 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** will take the preferred container and treated it as the **NodeManager**, on which the container was allocated, to use the resources for corresponding tasks.

8 Evaluation

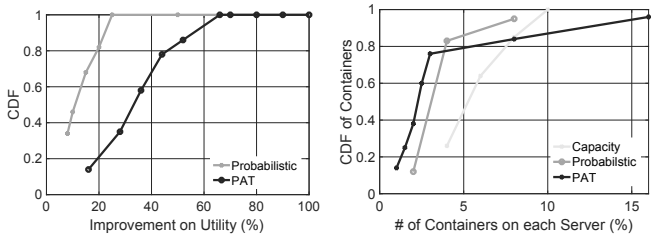
8.1 Testbed and Hierarchical Network

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. More details can be found in the preliminary version of the paper.

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 [28].

8.2 Simulation of DC Setup

We implement our solution and evaluate its performance under a typical DC topology, i.e. Fat-Tree, using ns-3 [29]. In this evaluation setup, containers running MapReduce application will communicate with one or more other containers in the DC network. We also introduce a container manger on each server, which serves as NodeManager in Apache Hadoop Yarn, to manage the allocation of requested containers. The central ResourceManager is responsible for scheduling containers among different servers in the data center. Fat-tree can be extended to any other types of representative networks architectures without loss of generality in our simulated environment. We constrain the amount of CPU and RAM resources of each server to represent the limited capacities of servers.



(a) Improvement on Utility (b) Allocation of Container

Fig. 7. Performance Evaluation of PAT.

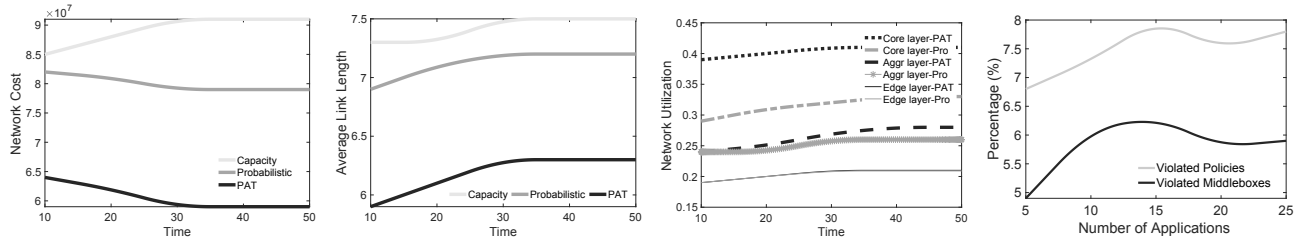
In the experiments, we configure that each server has 16GB RAMs and 8 cores, which can host 8 containers running in parallel if each container is configured with 2GB memory and one core. In network layers, the resource constraint is represented as the demand of all the traversing flows do not exceed the device's capacities. Considering all of these factors, a task allocation decision is only feasible if the hosting server has sufficient computational resources and networking bandwidth, i.e., a feasible server as defined in Equation 13.

In SDN-based implementation, there is a centralized controller that stores policies for routers and middleboxes. It decides whether these devices should accept and transmit data packets. We apply the FlowTags [30] implementation to enforce that data transimission should travel a sequence of middileboxes based on policies. We constraint that all the traffic flows from running MapReduce applications in the cloud have to follow the policy configurations introduced in the section 6. The constraints represent that all the traffic flows have to traverse a number of devices sequentially as configured by the networking policies before the intermediate results reach their destinations [31]. All the traffic flows are configured to traverse 1-3 networking devices in the cloud, including Firwall, IPS and LB. To evaluate the performance and efficiency of our solution, we compare it with Probabilistic Network-aware scheduler [28], which however is a non-policy-aware task scheduling scheme. As for the network cost, we evaluate the impact of networking policies in terms of link length and network utilization. Here, we define the link length as the number of hops that each flow will traverse and network utilization as average usage of bandwidth in each networking layer in a data center topology.

8.3 Improvement on Overall Utility

We first evaluate the improvement on utility in terms of the cumulative distribution function (CDF). Figure 7(a) demonstrates the performance of PAT in term of network cost. We measure the improvement of the individual container's network cost after policy-aware scheduling by comparing the improved utility brought by both Probabilistic Network-aware scheduler and PAT with the baseline using the default Capacity scheduler. Results show that PAT can reduce the network cost by 33.6%,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2961653, IEEE Transactions on Cloud Computing



(a) (b) (c) (d)

Fig. 8. Performance comparison and analysis of PAT and Probabilistic Network-aware scheduler.

while Probabilistic Network-Aware Scheduler can only achieve that by nearly 14.6% on average. It means that 55% of scheduling decisions can reduce the corresponding network cost by almost 45% effectively. We further study intrinsic properties to learn how PAT works to achieve the performance. Figure 7(b) shows the allocations results of containers from the default Capacity scheduler, Probabilistic Network-Aware Scheduler and PAT, respectively. It can be observed that, Capacity scheduler randomly assigns tasks on server wherever there are available capacities, e.g., each server hosts 5-7 containers. Probabilistic Network-Aware Scheduler can cluster containers into smaller groups of servers. However, with policy-aware scheduling, PAT can find more potential improvement, which group nearly 70% of total tasks into 22.8% of servers.

8.4 Performance of PAT on Network Utilization

We further present performance results of PAT and Probabilistic Network-aware scheduler in terms of the overall network cost reduction, average end-to-end link length and network utilization. Figures 8(a) demonstrates that PAT can efficiently reduce the total network cost by 28.2% while Probabilistic Network-aware scheduler only achieves an improvement by 13.6% compared with default capacity scheduler. The reason why PAT has higher improvement is that it offers more space for optimization than Probabilistic Network-aware scheduler with policy constraints. More importantly, as shown in Figure 8(b), by policy-aware scheduling, PAT can significantly shorten the average link length by 15.8%, while Probabilistic Network-aware scheduler reduces it only by 5.42%. These two results imply that the traffic flows can be forwarded to the destinations as soon as possible by reducing the average link length while avoiding the possible network congestion in the same time. PAT scheduling decisions can localize the containers with heavy shuffle data and reduce the link length of the end-to-end flow by optimizing the network communication.

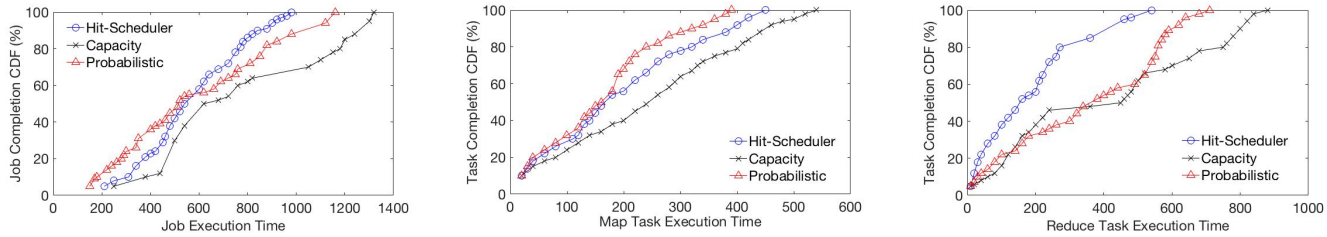
Furthermore, Figure 8(c) illustrates that PAT can improve network utilization in both the core and aggregation layers in the data center by 23.9% and 8.5%, respectively. The corresponding reduction for Probabilistic Network-aware scheduler is only 6.5% and 3.8%, respectively. However, the improvement on network utilization in edge layers is marginal compared with the other two layers. This is because all of these strategies can fully utilize lower-layer links. Improvements on the network utilization in the core and aggregation layers demonstrate that our proposed PAT can effectively create extra headroom to accommodate much more containers by utilizing the network topological resources effectively. We also analyze the cases of policy violations as depicted in our case study. From Figure 8(d), we can see that Probabilistic Network-aware scheduler causes more than 12% of policy violations and 8.4% on average of middleboxes are involved in the violations. Because Probabilistic Network-aware scheduler only considers feasible servers and middleboxes in the network, which will obviously cause potential security vulnerabilities.

8.5 Improvement on Job Completion Time

Figure 9(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 9(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 9(c) further confirms that Hit-Scheduler achieves significant improvement in terms of Reduce task execution times compared to the other strategies.

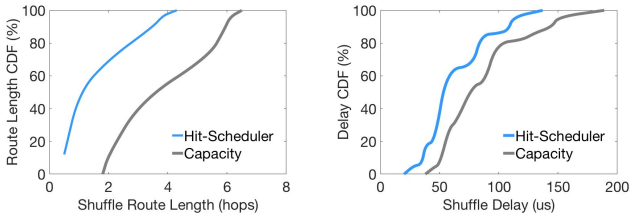
Figure 10(a) illustrates the effectiveness on reducing the length of the average routing path. It shows that Hit-Scheduler reduces the average route path from 6.5 hops to 4.4 hops compared with Capacity Scheduler, which achieves nearly 30% improvement. It is because Capacity Scheduler is unaware of the network architecture, resulting in longer flow route path which is caused by failing

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2019.2961653, IEEE Transactions on Cloud Computing



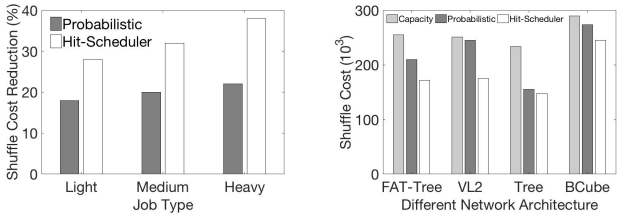
(a) (b) (c)

Fig. 9. Cumulative Distribution of Job Completion Times, Map and Reduce Execution Times under Various Policies.



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

Fig. 10. Performance Comparison Hit vs. Capacity.



(a) Different Job Types (b) Different Networks

Fig. 11. Impact of Network and Job

to consider optimizing the traffic in the network. As a result of shorter route path, Figure 10(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.

8.6 Impact of Network and Workload

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

Figure 11(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

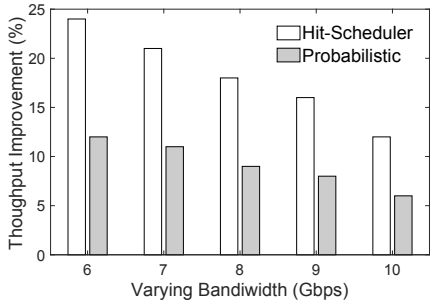


Fig. 12. Sensitivity to Network Bandwidth

can effectively support various complex topologies and correspondingly achieves better scalability.

8.7 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 12 shows the throughput improvement achieved by Hit-Scheduler and Probabilistic Network Aware scheduler compared to Capacity scheduler under varying bandwidth from 6Gbps to 10Gbps [32]. It demonstrates that Hit-Scheduler significantly outperforms Probabilistic Network Aware scheduler especially when given the limited bandwidth. The improvement can be nearly 23.8% while the bandwidth is limited to 6Gbps. This is due to the fact that Probabilistic Network Aware scheduler assumes the communication cost between two nodes 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 opportuni-

ties for Hit-Scheduler to improve the throughput with the limited bandwidth. Please refer to the preliminary study [13] for the sensitivity of Hit-Scheduler under various job numbers.

9 Related Work

There are considerable literatures on computation placement considering network and resource optimization [15]. However, most techniques require specialized hardware or communication protocols. A number of researches have been proposed to improve the scheduling for MapReduce jobs. Zaharia et al. [7] 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. ShuffleWacther [5] and iShuffle [6] try to improve the locality of the shuffle by scheduling both maps and reducers on the same set of racks. However, all 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. A transmission cost-based scheduling method, Probabilistic Network-Aware Scheduler [28] 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.

Recent development in SDN can support much more policy deployment over the data center. *SIMPLE* [33] is an SDN-based policy enforcement scheme to steer traffic in data center according to policy requirements. FlowTags [30] is proposed to leverage SDN's global network visibility and guarantee correctness of policy enforcement. EnforSDN [34] is proposed to decouple the policy resolution layer from the policy enforcement layer so as to provide better flexibility. PGA [35] PGA is proposed to model the behavior of closed middleboxes and ensure their correct behavior in a service chain while minimizing operator interventions. However, they are not fully designed with computation in consideration, and may put the application running in the data center under the risk of violating the policies. However, these approaches did not fully consider the allocation of virtual machines or container, especially the specific big data analytic workloads, which might cause performance degradation.

Multi-tenant Cloud DC environments enable more flexible middlebox deployments over the network. $PACE$ [25] is proposed to support application-wide, in-network policies. However, it only considers both the computation placement and network policies in the one-off scenario. Quokka [36] is proposed to schedule both the traffic flows and middleboxes in the network to reduce the transmission latencies. A programming middlebox [37] based on Clos network design is presented to improve the bandwidth utilization and reduce the network latency by utilizing the global network information. $PLAN$ [38] studies the optimization of policy-aware network resource usage but focus on the VMs management. All of those works do not consider the impact from network policies on task placement.

10 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 reduce the traffic cost by 38% at most and reduce the average shuffle flow traffic time by 32% compared to Capacity scheduler. In this manuscript, we have extended Hit-scheduler to a decentralized heuristic scheme to perform the policy-aware allocation in data center environments. The simulation based experimental results show that the proposed PAT policy reduces the communication cost by 33.6% compared with the default scheduler in data centers.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proc. of ACM Eurosys*, 2010.

[3] A. Verma, B. Cho, N. Zea, I. Gupta, and R. H. Campbell, "Breaking the mapreduce stage barrier," *Cluster computing*, vol. 16, no. 1, pp. 191–206, 2013.

[4] K. Shvachko, H. Kuang, S. Radia, R. Chansler *et al.*, "The hadoop distributed file system." in *Proc. of IEEE MSST*, 2010.

[5] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar, "Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters," in *Proc. of USENIX ATC*, 2014.

[6] Y. Guo, J. Rao, D. Cheng, and X. Zhou, "ishuffle: Improving hadoop performance with shuffle-on-write," *In IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1649–1662, 2017.

[7] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments." in *Proc. of USENIX OSDI*, 2008.

[8] L. Gyarmati and T. A. Trinh, "Scafida: A scale-free network inspired data center architecture," *Proc. of ACM SIGCOMM*, 2010.

[9] L. Tong, Y. Li, and W. Gao, "A hierarchical edge cloud architecture for mobile computing," in *Proc. of IEEE INFOCOM*, 2016.

[10] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The case for evaluating mapreduce performance using workload suites," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*. IEEE, 2011, pp. 390–399.

[11] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Proc. of USENIX NSDI*, 2012.

[12] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: an intellectual history of programmable networks," *Proc. of ACM SIGCOMM*, 2014.

[13] D. Yang, W. Rang, and D. Cheng, "Joint optimization of mapreduce scheduling and network policy in hierarchical clouds," in *Proc. of ICPP*. ACM, 2018.

[14] D. G. Cattrysse and L. N. Van Wassenhove, "A survey of algorithms for the generalized assignment problem," *European journal of operational research*, 1992.

[15] F. P. Tso, K. Oikonomou, E. Kavvadia, and D. P. Pezaros, "Scalable traffic-aware virtual machine management for cloud data centers," in *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*. IEEE, 2014, pp. 238–247.

[16] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging endpoint flexibility in data-intensive clusters," in *Proc. of ACM SIGCOMM*, 2013.

[17] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar, "Tarazu: optimizing mapreduce on heterogeneous clusters," in *Proc. of ACM SIGARCH*, 2012.

[18] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," in *Proc. of ACM SIGCOMM*, 2009.

[19] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: a scalable fault-tolerant layer 2 data center network fabric," in *Proc. of ACM SIGCOMM*, 2009.

[20] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "Bcube: a high performance, server-centric network architecture for modular data centers," *In ACM SIGCOMM Computer Communication Review*, 2009.

[21] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," *Proc. of ACM SIGCOMM*, 2008.

[22] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *Proc. of IEEE INFOCOM*, 2010.

[23] A. Headquarters, "Cisco data center infrastructure 2.5 design guide," in *Cisco Validated Design I*. Cisco Systems, Inc, 2007.

[24] H. Kellerer, U. Pferschy, and D. Pisinger, "Introduction to np-completeness of knapsack problems," in *Knapsack problems*. Springer, 2004, pp. 483–493.

[25] L. E. Li, V. Liaghat, H. Zhao, M. Hajiaghayi, D. Li, G. Wilfong, Y. R. Yang, and C. Guo, "Pace: Policy-aware application cloud embedding," in *Proc. of INFOCOM*. IEEE, 2013.

[26] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella, "Toward software-defined middlebox networking," in *Proc. of Workshop on Hot Topics in Networks*. ACM, 2012.

[27] C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," *IEEE transactions on Computers*, vol. 100, no. 10, pp. 892–901, 1985.

[28] H. Shen, A. Sarker, L. Yu, and F. Deng, "Probabilistic network-aware task placement for mapreduce scheduling," in *Proc. of IEEE CLUSTER*, 2016.

[29] "Ns-3," https://www.nsnam.org//.

[30] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags," in *Proc. of USENIX NSDI*, 2014.

[31] D. A. Joseph, A. Tavakoli, and I. Stoica, "A policy-aware switching layer for data centers," in *SIGCOMM Computer Communication Review*. ACM, 2008.

[32] "Amazon ec2 instance types." https://aws.amazon.com/ec2/instance types/.

[33] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "Simple-fying middlebox policy enforcement using sdn," in *Proc. of ACM SIGCOMM*, 2013.

[34] Y. Ben-Itzhak, K. Barabash, R. Cohen, A. Levin, and E. Raichstein, "Enforsdn: Network policies enforcement with sdn," in *2015 International Symposium on Integrated Network Management*. IEEE.

[35] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang, "Pga: Using graphs to express and automatically reconcile network policies," in *Proc. of ACM SIGCOMM*, 2015.

[36] P. Duan, Q. Li, Y. Jiang, and S.-T. Xia, "Toward latency-aware dynamic middlebox scheduling," in *Proc. of ICCCN*. IEEE, 2015.

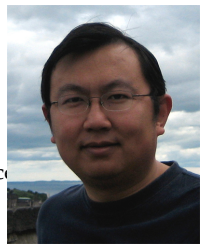
[37] R. Tu, X. Wang, J. Zhao, Y. Yang, L. Shi, and T. Wolf, "Design of a load-balancing middlebox based on sdn for data centers," in *Proc. of INFOCOM Workshops*. IEEE, 2015.

[38] L. Cui, F. P. Tso, D. P. Pezaros, W. Jia, and W. Zhao, "Plan: Joint policy-and network-aware vm management for cloud data centers," *Transactions on Parallel and Distributed Systems*, 2017.

Donglin Yang received his B.S. degree in Electrical Engineering from Sun Yat-sen University, China, in 2016. Currently, he is working towards the Ph.D. degree in Computer Science at the University of North Carolina at Charlotte. His works focus on big data analytics platforms and machine learning computing systems.

Dazhao Cheng received his B.S. and M.S. degrees in Electronic Engineering from Hefei University of Technology in 2006 and the University of Science and Technology of China in 2009, respectively. He received his Ph.D. degree from the University of Colorado, Colorado Springs in 2016. He is currently an Assistant Professor in the Department of Computer Science at the University of North Carolina, Charlotte. His research interests include Big Data and cloud computing. He is a member of the IEEE and ACM.

Wei Rang received his B.S. degree in Computer Science from Shandong Normal University, China in 2013. He obtained his M.S. degree in Computer Science from Southern Illinois University Carbondale in 2017. Currently, he is a Ph.D. student in Computer Science at the University of North Carolina at Charlotte. His research interests mainly focus on Cloud Computing and Parallel Computing.



Yu Wang is a Professor of Computer and Information Sciences, Temple University. He received his Ph.D. degree in Computer Science from Illinois Institute of Technology, his B.Eng. degree and M.Eng. degree in Computer Science from Tsinghua University, China. His research interest includes wireless networks, mobile social networks, smart sensing, and mobile computing. He has published over 150 papers in peer reviewed journals and conferences, with four best paper awards. He is an IEEE Fellow and senior member of the ACM.