Performance Evaluation 96 (2016) 1–11

Contents lists available at ScienceDirect

Performance Evaluation

journal homepage: www.elsevier.com/locate/peva

Data locality in MapReduce: A network perspective

Weina Wang *, Lei Ying

School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ 85281, USA

ARTICLE INFO

Article history:
Received 30 August 2014
Received in revised form 3 December 2015
Accepted 4 December 2015
Available online 17 December 2015

Keywords:
MapReduce
Data locality
Scheduling
Routing
Throughput

ABSTRACT

Data locality, a critical consideration for the performance of task scheduling in MapReduce, has been addressed in the literature by increasing the number of locally processed tasks. In this paper, we view the data locality problem from a network perspective. The key observation is that if we make appropriate use of the network to route the data chunk to the machine where it will be processed in advance, then processing a remote task is the same as processing a local task. However, to benefit from such a strategy, we must (i) balance the tasks assigned to local machines and those assigned to remote machines, and (ii) design the routing algorithm to avoid network congestion. Taking these challenges into consideration, we propose a scheduling/routing algorithm, named the Joint Scheduler, which utilizes both the computing resources and the communication network efficiently. We prove that the Joint Scheduler is throughput optimal; i.e., it supports any load that is supportable by any other algorithm. Simulation results demonstrate that with popularity skew, the Joint Scheduler improves the throughput and delay performance significantly compared to the Hadoop Fair Scheduler with delay scheduling, which is the de facto industry standard.

1. Introduction

The MapReduce framework [1] has been widely deployed in large computing clusters for the growing need of big data analysis. Hadoop is one of the most popular implementations of the MapReduce framework and has been adopted by various organizations.

The MapReduce framework is implemented on top of distributed file systems such as the Google File System (GFS) and the Hadoop Distributed File System (HDFS) [2], which divide large datasets into data chunks and store multiple replicas of each chunk on different machines. MapReduce jobs are submitted to request data processing and each job is divided into a number of *map* tasks and *reduce* tasks. A map task reads one data chunk and processes it to generate intermediate results. Reduce tasks fetch these intermediate results and conduct further computations to get the final results.

Map tasks and reduce tasks are assigned to machines according to a scheduling algorithm. During task scheduling, an important consideration is to place computation near data, i.e., to assign a task on or close to the machine that stores its input data on local disks. This is commonly referred to as the *data locality* problem. The data locality problem is particularly crucial for map tasks since they read data from the distributed file system and map functions are data-parallel. Besides, according to an empirical trace study from a production MapReduce cluster [3], the majority of jobs are map-intensive, and many of them are map-only. Therefore we focus on data locality in map task scheduling algorithms and assume that reduce tasks are not the bottleneck of the job processing or the communication network.

We call a machine a *local machine* for a map task if it has the input data chunk of this task on its local disks, and we call this map task a *local task* on this machine; otherwise, the machine is called a *remote machine* for this map task and this map

* Corresponding author. Tel.: +1 515 509 5697.
E-mail addresses: weina.wang@asu.edu (W. Wang), lei.ying.2@asu.edu (L. Ying).

http://dx.doi.org/10.1016/j.peva.2015.12.002
0166-5316/

task is called a *remote task* on the machine. We also use the term *locality* to refer to the fraction of tasks that are executed on local machines. In most of the existing work, launching a map task on a remote machine is considered to be inefficient, since a remote machine needs to first retrieve the input data from other machines through the communication network before processing it, which introduces an additional delay to the task execution. So local and remote tasks are often modeled with different processing times.

This view of data locality in MapReduce is arguable. If the communication network that connects the machines had infinite capacity and could transfer data instantaneously, then there would be no difference between assigning a task to its local machines or to other machines. Thus the time of processing a remote task depends on the capacity of the communication network and the scheduling algorithm that allocates tasks. If *data can be routed in advance so that machines do not spend time on waiting for input data before executing tasks*, then even though the network capacity is finite, we can still achieve the same throughput as if all tasks are local. Inspired by this intuition, we study the data locality problem from a network perspective beyond just abstracting the effect of the communication network as a "longer processing time". We adopt an approach that explicitly takes account of the structure of the communication network and quantify fundamental limits on the capacity of a MapReduce cluster with network constraints. Then we explore joint scheduling and routing algorithms to fully exploit the system capacity.

To optimize the performance, we face the following challenges: *how to strike the right balance between local and remote tasks*, and *how to route the traffic in the network appropriately to avoid congestion.* Failure to meet these challenges may result in slow data transmission and a waste of computing resources and may even harm the throughput performance. These challenges are more pronounced when data are not ideally uniformly distributed across the cluster [4], in which case placing all the computation near data results in heavy load on some machines while leaving other machines lightly utilized or idle.

In this paper, we first quantify fundamental limits on the capacity of a MapReduce cluster with network constraints by characterizing its capacity region, which consists of arrival rate vectors of tasks for which there exists a scheduling/routing algorithm that stabilizes the system. Then we propose a queueing architecture that enables us to jointly design the scheduling and routing algorithm with the above challenges taken into consideration to achieve throughput optimality, i.e., to stabilize the system for any arrival rate vector strictly within the capacity region. We call our scheduling/routing algorithm the Joint Scheduler. Our contributions are summarized as follows.

- We advocate studying the data locality problem from a network perspective. We propose to route the input data of tasks in advance through the communication network. Joint task scheduling and routing, taking communication network constraints into consideration, should be designed to fully exploit the system capacity.
- We propose a queueing architecture that captures both the data transmission in the communication network and the task execution by machines. This queueing architecture makes accurate status information of the network traffic and the workload of machines available, and thus enables joint scheduling and routing according to the information.
- Based on the proposed queueing architecture, we develop a task scheduling/routing algorithm, named the Joint Scheduler, that uses join the shortest queue policy (with blocking under some conditions) to assign incoming tasks to machines and route tasks in the communication network.
- We characterize the capacity region of a MapReduce cluster with data locality and communication network constraints. Then we prove that the Joint Scheduler is throughput optimal; i.e., it stabilizes the system for any arrival rate vector strictly within the capacity region. Therefore, the Joint Scheduler utilizes the computing resources efficiently by balancing the tasks assigned to local machines and those assigned to remote machines, and exploits the communication network capacity by balancing the traffic load to avoid congestion.
  Note that although the Joint Scheduler is in spirit similar to a MaxWeight/backpressure algorithm [5], the coupling between a task and its input datasets the scheduling/routing problem in a MapReduce cluster apart from a traditional network scheduling problem, in which a packet that arrives at a service node becomes ready for the service immediately.

The rest of this paper is organized as follows. Section 2 discusses the related work in literature. Section 3 introduces the system model. Section 4 presents the design of our Joint Scheduler and Section 5 establishes the throughput optimality. Simulation results are given in Section 6 to compare the Joint Scheduler and the Hadoop Fair Scheduler. The paper is concluded in Section 7.

2. Related work

The data locality problem has been intensively studied in the literature of task scheduling in MapReduce [1,6–9,4,10–19]. The Fair Scheduler in Hadoop is the de facto standard [6], in which the delay scheduling technique is used to improve locality. Quincy [7] uses the amount of data transfer as the measure of locality and makes scheduling decisions by solving a classic min-cost flow problem. Scheduling algorithms with formal theoretical throughput performance guarantee have also been developed [12,15,19]. However, the above existing approaches all use the request-and-wait procedure to obtain input data before processing the task. As we argued, this may lead to underutilization on both the computing resources and the communication network capacity.

Most related to our work is the prefetching idea [8,17]. However, the prefetching schemes developed in these papers are based on heuristics. The performance regarding load balancing and congestion in the network is not analyzed theoretically.

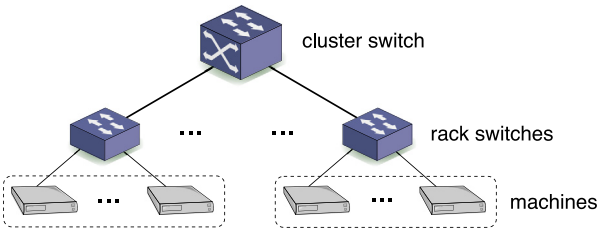


Fig. 1. Hierarchical network architecture.

Although we focus on addressing the data locality problem within the MapReduce framework, we note that approaches from the distributed file system side have also been proposed. The authors of [4,10] exploit the variance in data popularity and access patterns and present algorithms named Scarlett and DARE, respectively, to replicate data chunks based on their data access patterns. These algorithms share the same spirit with ours in that we both make popular data accessible to more machines.

3. System model

We consider a MapReduce computing cluster with M machines, which will be referred to by their indices $1, 2, \ldots, M$. The cluster adopts a multi-level hierarchical network architecture depicted in Fig. 1, where machines are grouped into R racks of machines at the lowest level, and one or more levels of aggregation switches connect the racks. This hierarchical network architecture is commonly used by data centers [20,21,1,6].

Jobs come in stochastically, and when a job comes in, it brings a random number of map tasks, which need to be served by the machines. We assume that data chunks have the same fixed size (e.g., 128 MB), and each data chunk is replicated and placed at three different machines, which is the default configuration of HDFS. Therefore, each task is associated with three local machines. When a task is launched on a remote machine, the machine cannot start processing the task until the necessary data arrives. According to the associated local machines, tasks can be classified into *types* denoted by

$$\vec{L} = (m_{\vec{L}}^1, m_{\vec{L}}^2, m_{\vec{L}}^3),$$

where $m_{\vec{L}}^i$, $i = 1, 2, 3$ are the indices of the three local machines, in increasing order. For example, if the data chunk associated with a task is stored at machines 1, 21 and 23 then $\vec{L} = (1, 21, 23)$. We assume machine $m_{\vec{L}}^1$ is in rack $r_{\vec{L}}^1$, and machine $m_{\vec{L}}^2$ and $m_{\vec{L}}^3$ are in the same rack $r_{\vec{L}}^2$ according to Hadoop's data replication policy [2].

3.1. Arrivals and service

We consider a time-slotted system. Let $A_{\vec{L}}(t)$ denote the number of type $\vec{L}$ tasks arriving at the beginning of time slot t. We assume that $\{A_{\vec{L}}(t),\ t \geq 0\}$ is an i.i.d. sequence with arrival rate $\mathbb{E}[A_{\vec{L}}(t)] = \lambda_{\vec{L}}$ and the second moment $\mathbb{E}[(A_{\vec{L}}(t))^2]$ is finite. We assume that there is a positive probability that there is no arrival at a time slot.

Let $\lambda = (\lambda_{\vec{L}} \colon \vec{L} \in \mathcal{L})$ be the arrival rate vector, where $\mathcal{L}$ is the set of task types with arrival rates greater than zero; i.e., $\mathcal{L} = \{\vec{L} \colon \lambda_{\vec{L}} > 0\}$.

The service/processing time of each task is assumed to follow a geometric distribution with parameter φ, where $0 < \varphi < 1$.

The following notation is used throughout this paper:

- $\mathcal{M}_r$: the set of machines in rack r.
- r_m: the index of the rack that machine m is in.
- For each task type $\vec{L}$, the set of local machines is denoted by
$$\mathcal{M}_{\vec{L}} = \{m_{\vec{L}}^1, m_{\vec{L}}^2, m_{\vec{L}}^3\}.$$
- For each machine m, the set of types such that tasks of these types are local on machine m is denoted by
$$\mathcal{L}_m = \{\vec{L} \in \mathcal{L} \colon m \in \mathcal{M}_{\vec{L}}\}.$$

3.2. Network queueing model

In the considered hierarchical network architecture, a set of machines are mounted within a rack and interconnected by a rack switch. These rack switches have a number of uplink connections to the cluster switch, which can use 1-Gbps or 10-Gbps links. For economic considerations, the design of the network connections usually introduces oversubscription since

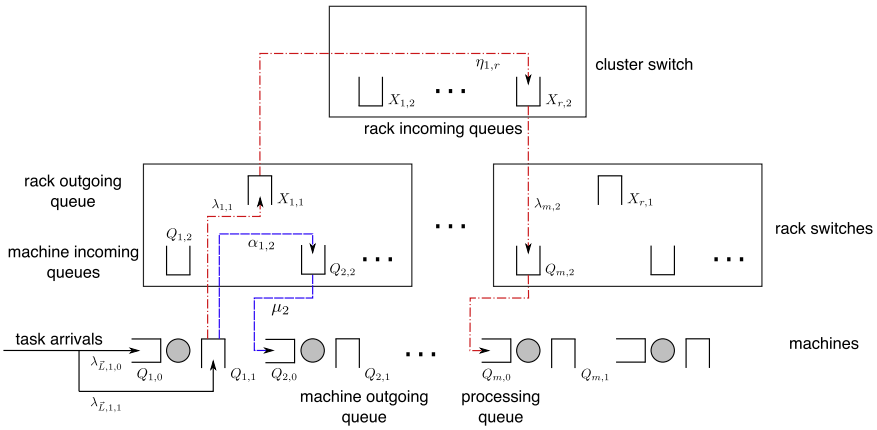


Fig. 2. Network queueing model.

all the interrack traffic needs to go through the cluster switch. For example, in the network that 40 servers in the same rack connect to the rack switch by 1-Gbps ports, rack switches may have between four and eight 1-Gbps uplinks to the cluster switch, corresponding to an oversubscription factor between 5 and 10 for communication across racks [21]. Therefore the cluster-level bandwidth resource is relatively scarce compared with the rack-level.

Data transmission in the network consumes bandwidth. Since each machine connects to the network through the rack switch, the bandwidth between the machine and the rack switch constrains the incoming and outgoing data transmission rates of the machine. When there is a large amount of data that needs to be sent or received by a machine, the unfinished data will be backlogged at some queues. *Let this constraint on the incoming and outgoing data transmission rates of each machine be B_1 data chunks per time slot*. For the interrack traffic, due to oversubscription, the machines in one rack cannot communicate with machines in other racks at their full bandwidth simultaneously. *There are constraints on the overall incoming and outgoing data transmission rates of the rack. Let this constraint be B_2 data chunks per time slot*. This rack bandwidth is shared by machines in the same rack.

Based on the network hierarchy, the communication network in the cluster is modeled as depicted in Fig. 2. For each machine m, $Q_{m,1}$ and $Q_{m,2}$ are the queues for the outgoing traffic and incoming traffic of the machine, respectively. Therefore, at most B_1 data chunks depart from each queue during one time slot. Similarly, for each rack r, $X_{r,1}$ and $X_{r,2}$ are the queues for the outgoing traffic and incoming traffic of the rack, respectively, and at most B_2 data chunks depart from each queue during one time slot. We call $Q_{m,1}$ and $Q_{m,2}$ the *machine outgoing queue* and the *machine incoming queue*, and we call $X_{r,1}$ and $X_{r,2}$ the *rack outgoing queue* and the *rack incoming queue*. The queue $Q_{m,0}$ and other labels will be introduced in Sections 4.1 and 5.2, respectively. Machine m communicates with machine a in the same rack through the path $(Q_{m,1}, Q_{a,2})$. An example is shown in Fig. 2 for $m = 1$, $a = 2$ with corresponding path $(Q_{1,1}, Q_{2,2})$. Machine m communicates with machine b in another rack through the path $(Q_{m,1}, X_{r_m,1}, X_{r_b,2}, Q_{b,2})$, as shown in Fig. 2 for $m = 1$, $b = 20$ with path $(Q_{1,1}, X_{1,1}, X_{r,2}, Q_{20,2})$. Since each map task is associated with a data chunk and the network is used to transfer the data chunks to be processed remotely, B_1 and B_2 can also be viewed as number of tasks per time slot. The phrases transmitting tasks and transmitting data are used interchangeably in the context of communication. The lengths of the queues are counted as the number of tasks that have input data chunks in the queues.

Remark 1. We assume without loss of generality that B_1 is larger than or equal to 1 since otherwise we can rescale the duration of the time slot. The rack switches usually use 1-Gbps uplinks. This transmission rate is usually larger than the data processing rate performed by map functions. So we assume a service rate φ smaller than 1 after rescaling. The rate B_2 is larger than B_1 and smaller than $R \cdot B_1$ due to oversubscription.

4. Map task scheduling/routing

In this section, we present a new algorithm that performs task scheduling and routing jointly. We call this scheduler the Joint Scheduler, which is also referred to as JS in this paper. This algorithm includes two parts: the first part assigns incoming tasks to some machines to serve or to the communication network to transmit as tasks arrive at the cluster; the second part routes the tasks in the communication network.

Before we describe our algorithm in detail, we first further elaborate on our queueing architecture of the cluster. We have derived the queueing model of the communication network in Section 3.2. Now we introduce the architecture of the processing queues in JS. For each machine m, the scheduler maintains a processing queue $Q_{m,0}$ to buffer the tasks assigned to machine m for local processing and the tasks whose data have been transferred to machine m for remote processing. Therefore, the tasks in $Q_{m,0}$ all have the corresponding data on machine m, and thus are ready for the processing.

Algorithm 1 The Joint Scheduler

```
at time slot t
for task in incoming tasks do
    find the task type L⃗
    assign the task to the shortest queue in 𝒬_{L⃗}
for Q in the communication network do
    find the shortest queue D_Q^t in 𝒟(Q)
    if D_Q^t(t) < Q(t) then
        send data to D_Q^t
    else
        block the outgoing traffic from Q
```

4.1. Task scheduling/routing algorithm

As discussed in the introduction, the low efficiency of remote task processing can be ascribed to the underutilization of machines while waiting for input data. Our algorithm addresses data locality by intelligently routing data in advance, which reduces the idling time of machines without causing network congestion.

For each queue Q in the communication network, the set of queues that can receive data directly from Q is called the set of *candidate destinations of Q and is denoted by $\mathcal{D}(Q)$*. These sets represent the connectivity of the system, which is illustrated in Fig. 2. We describe this candidate destination set for each queue as follows.

- For each machine m, the machine outgoing queue $Q_{m,1}$ can send data to the rack outgoing queue and to the machine incoming queues in the same rack, so
$$\mathcal{D}(Q_{m,1}) = \{X_{r_m,1}\} \cup \{Q_{m',2} \mid m' \in \mathcal{M}_{r_m}\}.$$
- The machine incoming queue $Q_{m,2}$ can only send data to the processing queue $Q_{m,0}$, so
$$\mathcal{D}(Q_{m,2}) = \{Q_{m,0}\}.$$
- For each rack r, the rack outgoing queue $X_{r,1}$ can send data to the rack incoming queues of all the racks, so
$$\mathcal{D}(X_{r,1}) = \{X_{r',2} \mid r' = 1, \ldots, R\}.$$
- The rack incoming queue $X_{r,2}$ can send data to the machine incoming queues in its own rack, so
$$\mathcal{D}(X_{r,2}) = \{Q_{m,2} \mid m \in \mathcal{R}_r\}.$$

Now we present the Joint Scheduler as follows. The pseudocode is shown in Algorithm 1 and a toy example illustrating the procedure of the algorithm is shown after.

- **Task assignment for new tasks**. When a type $\vec{L}$ task comes in, the scheduler assigns it to the shortest queue in $\mathcal{Q}_{\vec{L}} = \{Q_{m,i} \mid m \in \mathcal{M}_{\vec{L}},\ i = 0, 1\}$. Note that if a task is assigned to $Q_{m,0}$, it means that the task will be processed at local machine m; if it is assigned to $Q_{m,1}$, it means machine m needs to transmit the data chunk associated with the task to another (remote) machine to process.
- **Routing in the communication network**. For each queue in the communication network, we use a join the shortest queue algorithm with blocking to route tasks: each queue Q in the communication network first finds the shortest queue in the candidate destination set $\mathcal{D}(Q)$, denoted by D_Q^t; then it compares the queue length $D_Q^t(t)$ with $Q(t)$. If $D_Q^t(t) < Q(t)$, Q sends data to D_Q^t; otherwise Q does not send any data. If Q is the machine incoming queue or machine outgoing queue, then by the bandwidth constraint it can send at most B_1 data chunks during each time slot; similarly, if Q is the rack incoming queue or rack outgoing queue, then it can send at most B_2 data chunks during each time slot according to the bandwidth constraint.

A toy example. We use a toy example shown in Fig. 3 to illustrate the procedure of the Joint Scheduler. For the first step, a task of type $\vec{L} = (1, 2, 3)$ arrives at the system. The scheduler assigns it to $Q_{2,1}$, which is the shortest queue in $\mathcal{Q}_{(1,2,3)} = \{Q_{1,0}, Q_{1,1}, Q_{2,0}, Q_{2,1}, Q_{3,0}, Q_{3,1}\}$. For the second step, we use $Q_{1,1}$ and $Q_{2,1}$ to explain. The candidate destination set of $Q_{1,1}$ is

$$\mathcal{D}(Q_{1,1}) = \{Q_{1,2}, Q_{2,2}, X_{1,1}\}.$$

At time slot t, the shortest queue in this set is $X_{1,1}(t) = 2$, and since $Q_{1,1}(t) = 3 > X_{1,1}(t)$, $Q_{1,1}$ sends B_1 data chunks to $X_{1,1}$. For $Q_{2,1}$, the shortest queue in $\mathcal{D}(Q_{2,1})$ is also $X_{1,1}$. However, since $Q_{2,1}(t) = 1 < X_{1,1}(t)$, $Q_{2,1}$ blocks the outgoing traffic and does not send any data at the current time slot.

The Joint Scheduler balances the usage of the computing resources and the network resources by starting routing some of the data chunks as the corresponding tasks arrive. The tasks that have their input data stored on the same machine compete

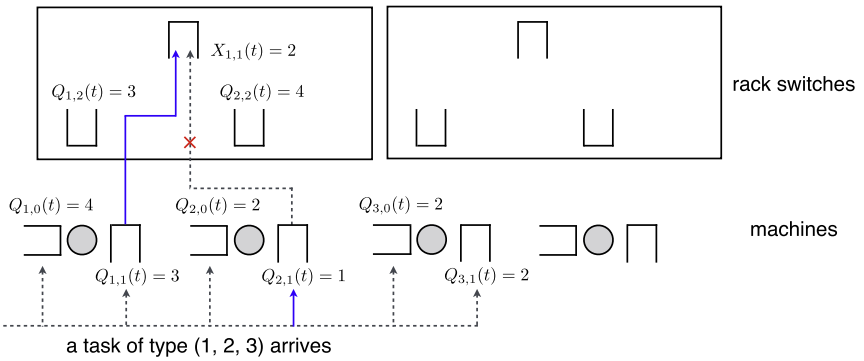


Fig. 3. A toy example illustrating the Joint Scheduler.

for the resources of this local machine. The scheduler assigns some of these tasks to the local machine and spreads other tasks to remote machines in advance. Since these tasks cannot be all launched on the local machine eventually, this foresight of routing reduces the waiting time of remote machines and improves the throughput.

The implementation complexity of the Joint Scheduler in each time slot is determined as follows. In the task assignment step, the scheduler compares six queue lengths for each incoming task. In the routing step, for each queue in the communication network, the scheduler finds the shortest queue in the candidate destination set. The size of a candidate destination set is one of the following values: 1, the number of machines in a rack, the number of racks.

4.2. Queue dynamics

Scheduling decisions are made at the beginning of each time slot t and the service is performed after arrivals. For each queue Q in the system with arrival process A and service process S, the queue dynamics is expressed by the Lindley equation:

$$Q(t+1) = (Q(t) + A(t) - S(t))^+. \tag{1}$$

Next we describe the arrival and service processes for each queue in the system.

At each time slot t, the type $\vec{L}$ tasks that arrive exogenously to the system are assigned to queues in $\mathcal{Q}_{\vec{L}}$, where the number of the arrivals assigned to $Q_{m,i} \in \mathcal{Q}_{\vec{L}}$ with $m \in \mathcal{M}_{\vec{L}}$ and $i \in \{0, 1\}$ is denoted by $A^t_{\vec{L},m,i}(t)$. Other arrivals in the system are internal arrivals, which are the departures of some other queues at the last time slot. Notation for those arrivals is as follows:

- $A_{m,1}$: arrivals coming from $Q_{m,1}$ to $X_{r_m,1}$;
- $A^Q_{m',m}$: arrivals coming from $Q_{m',1}$ to $Q_{m,2}$;
- $A^X_{r',r}$: arrivals coming from $X_{r',1}$ to $X_{r,2}$;
- $A_{m,2}$: arrivals coming from $X_{r_m,2}$ to $Q_{m,2}$;
- $A_{m,0}$: arrivals coming from $Q_{m,2}$ to $Q_{m,0}$.

Under the bandwidth constraints, the service processes of the queues in the communication network are defined as follows:

- For each machine m, the service processes of $Q_{m,1}$ and $Q_{m,2}$ are denoted by $\{S_{m,1}(t),\ t \geq 0\}$ and $\{S_{m,2}(t),\ t \geq 0\}$, respectively, which are defined as

$$S_{m,i}(t) = \begin{cases} B_1 & \text{if } Q_{m,i}(t) > D^t_{Q_{m,i}}(t), \\ 0 & \text{otherwise.} \end{cases}$$

- For each rack r, the service processes of $X_{r,1}$ and $X_{r,2}$ are denoted by $\{S^X_{r,1}(t),\ t \geq 0\}$ and $\{S^X_{r,2}(t),\ t \geq 0\}$, respectively, which are defined as

$$S^X_{r,i}(t) = \begin{cases} B_2 & \text{if } X_{r,i}(t) > D^t_{X_{r,i}}(t), \\ 0 & \text{otherwise.} \end{cases}$$

The service process of each processing queue $Q_{m,0}$ is denoted by $\{S_m(t),\ t \geq 0\}$. If machine m has been working on a task during time slot t, i.e., $Q_{m,0}(t) > 0$, let $S_m(t) = 1$ if the service is completed at the end of time slot t and let $S_m(t) = 0$ otherwise. If machine m is idle during time slot t, i.e., $Q_{m,0}(t) = 0$, let $S_m(t)$ be a Bernoulli random variable with parameter φ that is independent of other random variables. Since the service time of each task is assumed to follow a geometric distribution with parameter φ, the service process $\{S_m(t),\ t \geq 0\}$ is temporally i.i.d. with each $S_m(t)$ being a Bernoulli random variable with parameter φ. We remark that $S_m(t)$ is the potential service since it is also defined for idle queue, instead of the actual service.

Notice that the relations between internal arrivals and service are

$$S_{m,1}(t-1) \geq A_{m,1}(t) + \sum_{m' \in \mathcal{M}_{r_m}} A^{Q}_{m,m'}(t) \tag{2a}$$

$$S^{X}_{r,1}(t-1) \geq \sum_{r'} A^{X}_{r,r'}(t) \tag{2b}$$

$$S^{X}_{r,2}(t-1) \geq \sum_{m \in \mathcal{M}_r} A_{m,2}(t) \tag{2c}$$

$$S_{m,2}(t-1) \geq A_{m,0}(t). \tag{2d}$$

We assemble all the queue lengths into a vector

$$Z = (Q_{m,i}, X_{r,j} \colon m = 1, \ldots, M,\ i = 0, 1, 2,\ r = 1, \ldots, R,\ j = 1, 2).$$

Then under the statistical assumptions we have made and the Joint Scheduler, the queueing process $\{Z(t),\ t \geq 0\}$ is a Markov chain. We assume that the state space $\mathcal{S}$ consists of all the states which can be reached from the zero vector.

Remark 2. This Markov chain $\{Z(t),\ t \geq 0\}$ is irreducible and aperiodic for the following reasons. For any state Z in the state space, since the queue lengths in the system are integers, the Markov chain can reach the zero state from Z within a finite number of time slots when there are a positive number of departures but no arrivals at each time slot. This probability is positive, so Z can reach the zero state and hence the Markov chain is irreducible. We can also see that the transition probability from the zero state to itself is positive, so the Markov chain is also aperiodic.

5. Throughput optimality

The throughput performance of a scheduling/routing algorithm is characterized by its stability region [5], i.e., the set of arrival rate vectors for which this scheduling/routing algorithm stabilizes the system. The system is stable if the number of backlogged tasks does not explode to infinity. Formally, the system is said to be *stable* if the number of backlogged tasks, denoted by $\{\Phi(t),\ t \geq 0\}$, satisfies [22]

$$\lim_{C \to \infty} \lim_{t \to \infty} \Pr(\Phi(t) > C) = 0. \tag{3}$$

5.1. Capacity region

The *capacity region* is defined to be the maximum stability region, i.e., the set of arrival rate vectors for which there exists a scheduling/routing algorithm that stabilizes the system. A scheduling/routing algorithm assigns incoming tasks to machines for processing or for data transmission and routes tasks in the communication network. A scheduling/routing algorithm is said to be *throughput optimal* if its stability region contains the interior of the capacity region; i.e., the scheduling/routing algorithm stabilizes the system for any arrival rate vector strictly within the capacity region [23].

5.2. Characterization of the capacity region

Let $\mathcal{C}$ denote the capacity region of the system. We characterize $\mathcal{C}$ by first considering some necessary conditions for an arrival rate vector λ to be in $\mathcal{C}$.

We say f is a λ-*admissible flow vector*, if an arrival rate vector λ has the following decomposition:

$$\lambda_{\vec{L}} = \sum_{m \in \mathcal{M}_{\vec{L}}} \lambda_{\vec{L},m,0} + \sum_{m \in \mathcal{M}_{\vec{L}}} \lambda_{\vec{L},m,1} \tag{4a}$$

$$\sum_{\vec{L} \in \mathcal{L}_m} \lambda_{\vec{L},m,1} = \lambda_{m,1} + \sum_{m' \in \mathcal{M}_{r_m}} \alpha_{m,m'} \tag{4b}$$

$$\sum_{m \in \mathcal{M}_r} \lambda_{m,1} = \sum_{r'} \eta_{r,r'} \tag{4c}$$

$$\sum_{r' \in \mathcal{R}} \eta_{r',r} = \sum_{m \in \mathcal{M}_r} \lambda_{m,2} \tag{4d}$$

$$\sum_{m' \in \mathcal{M}_{r_m}} \alpha_{m',m} + \lambda_{m,2} = \mu_m, \tag{4e}$$

where f denotes the vector consisting of all the rates in the decomposition, i.e., $f = (\lambda_{\vec{L},m,0}, \lambda_{\vec{L},m,1}, \lambda_{m,1}, \alpha_{m,m'}, \eta_{r,r'}, \lambda_{m,2}, \mu_m)_{(\vec{L} \in \mathcal{L}, m, m' \in \{1,\ldots,M\}, r, r' \in \{1,\ldots,R\})}$. These rates in f can be interpreted as the rates of the following processes, if they exist,

under some scheduling algorithm that stabilizes the system:

- $\lambda_{\bar{L},m,i}$: the rate of $\{A_{\bar{L},m,i}(t),\ t \geq 0\},\ i = 0, 1$;
- $\lambda_{m,i}$: the rate of $\{A_{m,i}(t),\ t \geq 0\},\ i = 1, 2$;
- $\alpha_{m',m}$: the rate of $\{A^{Q}_{m',m}(t),\ t \geq 0\}$;
- $\eta_{r',r}$: the rate of $\{A^{X}_{r',r}(t),\ t \geq 0\}$;
- μ_m: the rate of $\{A_{m,0}(t),\ t \geq 0\}$.

They are labeled in Fig. 2 for an illustration. Let $\mathcal{F}_\lambda$ be the set of all λ-admissible flow vectors.

A flow vector f is said to be *supportable* if the corresponding arrival rate to each queue is less than the potential service rate of that queue; i.e., for each machine m and each rack r,

$$\sum_{\bar{L}\in\mathcal{L}_m} \lambda_{\bar{L},m,1} \leq B_1,\ \sum_{m'\in\mathcal{M}_{rm}} \alpha_{m',m} + \lambda_{m,2} \leq B_1, \tag{5a}$$

$$\sum_{m\in\mathcal{M}_r} \lambda_{m,1} \leq B_2,\ \sum_{r'\in\mathcal{R}} \eta_{r',r} \leq B_2, \tag{5b}$$

$$\sum_{\bar{L}\in\mathcal{L}_m} \lambda_{\bar{L},m,0} + \mu_m \leq \varphi. \tag{5c}$$

Let $\Lambda = \{\lambda \mid$ there exists a $f \in \mathcal{F}_\lambda$ that is supportable$\}$. Then it can be proved that no scheduling/routing algorithm can stabilize the system for an arrival rate vector that is not in Λ. The proof is similar to the proof of Theorem 5.3.1 in [22], which is based on the strict separation theorem and strong law of large numbers. Consider any arrival rate vector that is not in Λ. Rearranging the total departures of the system, we can obtain a supportable flow vector and its corresponding vector in Λ. Then the strict separation theorem gives a gap between this vector and the arrival rate vector, which together with the strong law of large numbers results in queues exploding to infinity. We omit the details of this proof here for conciseness. Therefore, a necessary condition for an arrival rate vector λ to be in the capacity region $\mathcal{C}$ is that it should be in Λ. Thus Λ gives an outer bound of $\mathcal{C}$; i.e., $\mathcal{C} \subseteq \Lambda$.

5.3. Achievability

Theorem 1 (*Throughput Optimality*)**.** *The Joint Scheduler stabilizes the system for any arrival rate vector strictly within Λ. Specifically, let Λ^o denote the interior of Λ and $\mathcal{C}_{JS}$ denote the stability region of the Joint Scheduler. Then Λ characterizes the capacity region in the following sense*

$$\Lambda^o \subseteq \mathcal{C}_{JS} \subseteq \mathcal{C} \subseteq \Lambda.$$

Hence the Joint Scheduler is throughput optimal.

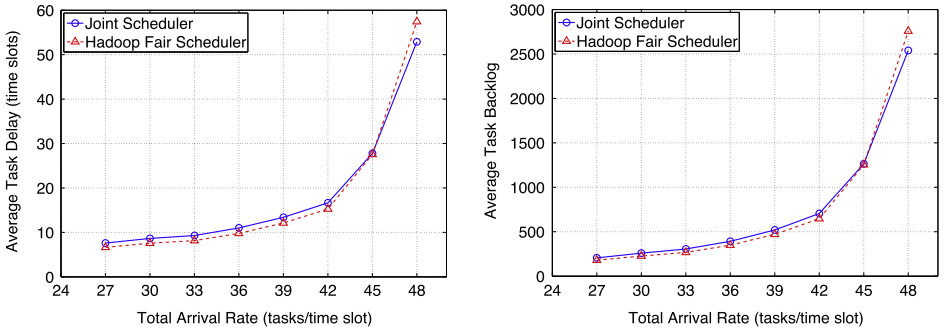
Consider any arrival rate vector strictly within Λ. As pointed out in Remark 2, the Markov chain $\{Z(t),\ t \geq 0\}$ is irreducible and aperiodic. If $\{Z(t),\ t \geq 0\}$ is also positive recurrent, it will converge to its stationary distribution. In this case, the number of backlogged tasks $\Phi(t)$ is the sum of all the queue lengths in $Z(t)$, so $\{\Phi(t),\ t \geq 0\}$ will also converge to a stationary distribution, which implies the stability defined in (3). Therefore, to show that the system is stable, it suffices to show that $\{Z(t),\ t \geq 0\}$ is positive recurrent.

We prove that the Markov chain $\{Z(t),\ t \geq 0\}$ is positive recurrent by considering the following quadratic Lyapunov function:

$$V(Z(t)) = \sum_{m=1}^{M}\sum_{i=0}^{2}(Q_{m,i}(t))^2 + \sum_{r=1}^{R}\sum_{j=1}^{2}(X_{r,j}(t))^2.$$

Then according to an extension of the Foster–Lyapunov theorem [22], it suffices to find a positive integer T such that the T time slot Lyapunov drift is bounded within a finite subset of the state space and negative outside this subset. The details of the proof are provided in our technical report [24].

Note that the proposed queueing architecture plays an important role for the Joint Scheduler to achieve throughput optimality and allows us to use the Lyapunov technique for the proof. In the proposed queueing architecture, a task in the processing queue of a machine has its input data on that machine once it is assigned to that machine. In other words, task assignment and data transmission are coordinated in this queueing architecture. This enables us to use pressure-based scheduling/routing to achieve optimal throughput. In addition, with this queueing architecture, the service times of the tasks in a processing queue are i.i.d. random variables following the same geometric distribution, which facilitates the usage of the Lyapunov technique and simplifies the proof. In general, with other queueing architectures, a task in a queue may have its data at another place, which makes the queue lengths inaccurate information of the load, and thus a pressure-based algorithm and the Lyapunov technique may not be applicable.



(a) Throughput performance. (b) Task delay performance.

Fig. 4. Performance under uniform data access.

6. Simulations

In this section we use simulations to compare the performance of the Joint Scheduler (JS) and the Hadoop Fair Scheduler (HFS) with delay scheduling. We mainly focus on the throughput performance and demonstrate that JS achieves the maximum capacity region while HFS cannot. Even though JS has not been fine-tuned to decrease task delay, the simulation results show delay reduction under JS compared to HFS with moderate to heavy load.

Settings. We simulate a MapReduce computing cluster with two hundred machines organized into ten racks. A distributed database is stored on machines in the cluster, with three replicas of each data chunk, and jobs are submitted to process part of the data. This mimics the scenario that data chunks form a database like the user profile database of Facebook, and each job is some manipulation of the data like searching for some user.

We scale the time slot in the system such that transmitting one data chunk between machines in the same rack takes one time slot; i.e., $B_1 = 1$. For the interrack traffic, we assume an oversubscription factor 4; i.e., $B_2 = 5$. The service rate is set to $\varphi = 0.25$ since typically processing a data chunk is slower than transmitting it. With this processing capability, the total arrival rate λ_Σ of map tasks should be no larger than $200 \times \varphi = 50$. We run the simulations for the two algorithms under several total arrival rates. For each arrival rate we run the simulation for 10^6 time slots and evaluate the performance using results from the last 5×10^4 time slots, during which the system is either unstable or in the steady state. Job sizes are generated following the analysis of workload from [4], which shows that the number of tasks in a job follows a power-law distribution. To make a fair comparison, we let JS prioritize jobs in the same way as HFS. Note that the job-level scheduling does not affect our analysis of the throughput performance.

We consider two different data access patterns for the task arrival processes. The first pattern accesses data uniformly from all the machines, and the second pattern accesses data from half of the racks more frequently than from the other half, which mimics the scenario with popularity skew.

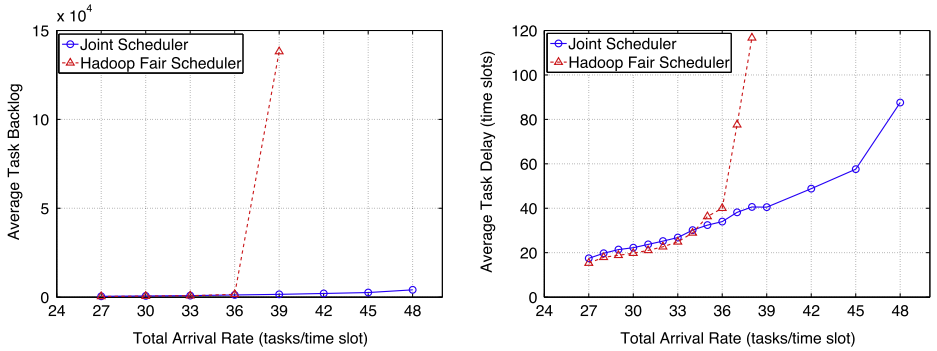
6.1. Uniform data access

HFS performs well under the arrivals with uniform data access patterns since most machines can easily find a local task to serve. In this scenario, JS and HFS perform similarly in throughput performance. Fig. 4(a) shows the average number of backlogged tasks in the last 5×10^4 time slots. The system is stable under both algorithms for all the arrival rates shown in the figure. Fig. 4(b) shows the average task delay in steady state, from which we can see that the task delay performance is very similar (HFS is slightly better than JS).

6.2. Data access with popularity skew

This is the scenario where JS benefits from routing in advance under moderate to heavy load, since many tasks need to be launched on remote machines under HFS in this case. The average number of backlogged tasks in the last 5×10^4 time slots is shown in Fig. 5(a). Under HFS, the average number of backlogged tasks shows a sudden increase at the total arrival rate $\lambda_\Sigma = 39$ tasks per time slot, which indicates that the system is unstable at arrival rates greater than that, while under JS the system remains stable for all the total arrival rates shown in the figure.

The average task delay is shown in Fig. 5(b). When the total arrival rate is small, the task delay performance is still similar (HFS is slightly better than JS). As the arrival rate increases, the task delay performance under HFS becomes worse. The average task delay under HFS becomes larger than that under JS when the arrival rate is greater than 35 tasks per time slot. For arrival rate greater than 39 tasks per time slot, the average task delay under HFS becomes very large (it is already over 2000 for $\lambda_\Sigma = 39$) due to instability. Thus to get a clear comparison figure, we did not show the average task delay under HFS for arrival rate larger than 38 tasks per time slot. Overall, JS significantly reduces the task delay under moderate to heavy load.



(a) Throughput performance. (b) Task delay performance.

Fig. 5. Performance under data access with popularity skew.

7. Conclusions and future work

We addressed the data locality problem in task scheduling by routing data in advance through the communication network in a MapReduce computing cluster. Taking both load balancing and network capacity into consideration, we proposed a scheduling/routing algorithm, named the Joint Scheduler, that maximizes the throughput of the system. The Joint Scheduler can be extended immediately to the case that tasks have different number of local machines and to communication networks with more levels of hierarchy. The Joint Scheduler requires control of the queues on switches. Such control is not available under the current Hadoop implementation, but is possible by using the emerging software-defined networking (SDN) technology [25]. Another limitation of this work is that we focused on the throughput performance of the system and have not put much effort on integrating job-level scheduling. A simple approach is to integrate it in the routing step: when a queue sends data out, it selects data based on the job information of the corresponding tasks. However, more delicate approaches can be explored to achieve different performance goals such as fairness.

With the proposed network perspective of the data locality problem, an exciting direction for future work is to consider more general network architectures. The structure of the connections between machines may not be a hierarchical one, and the bandwidths of different links can be heterogeneous.

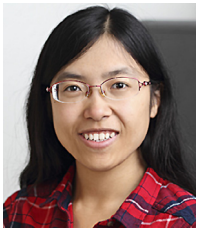
Acknowledgment

This work was supported in part by NSF Grant ECCS-1255425.

References

[1] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, ACM Commun. 51 (1) (2008) 107–113.
[2] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The Hadoop distributed file system, in: IEEE Symp. Mass Storage Systems and Technologies, MSST, Incline Villiage, NV, 2010, pp. 1–10.
[3] S. Kavulya, J. Tan, R. Gandhi, P. Narasimhan, An analysis of traces from a production MapReduce cluster, in: Proc. IEEE/ACM Int. Conf. Cluster, Cloud and Grid Computing, CCGRID, Melbourne, Australia, 2010, pp. 94–103.
[4] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, E. Harris, Scarlett: coping with skewed content popularity in MapReduce clusters, in: Proc. European Conf. Computer Systems, EuroSys, Salzburg, Austria, 2011, pp. 287–300.
[5] L. Tassiulas, A. Ephremides, Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks, IEEE Trans. Automat. Control 4 (1992) 1936–1948.
[6] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, I. Stoica, Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling, in: Proc. European Conf. Computer Systems, EuroSys, Paris, France, 2010, pp. 265–278.
[7] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, A. Goldberg, Quincy: fair scheduling for distributed computing clusters, in: Proc. ACM Symp. Operating Systems Principles, SOSP, Big Sky, MT, 2009, pp. 261–276.
[8] S. Seo, I. Jang, K. Woo, I. Kim, J.-S. Kim, S. Maeng, HPMR: Prefetching and pre-shuffling in shared MapReduce computation environment, in: IEEE Int. Conf. Cluster Computing, CLUSTER, New Orleans, LA, 2009, pp. 1–8.
[9] J. Jin, J. Luo, A. Song, F. Dong, R. Xiong, Bar: An efficient data locality driven task scheduling algorithm for cloud computing, in: Proc. IEEE/ACM Int. Conf. Cluster, Cloud and Grid Computing, CCGRID, Newport Beach, CA, 2011, pp. 295–304.
[10] C. Abad, Y. Lu, R. Campbell, DARE: Adaptive data replication for efficient cluster scheduling, in: IEEE Int. Conf. Cluster Computing, CLUSTER, Austin, TX, 2011, pp. 159–168.
[11] Q. Xie, Y. Lu, Degree-guided map-reduce task assignment with data locality constraint, in: Proc. IEEE Int. Symp. Information Theory, ISIT, Cambridge, MA, 2012, pp. 985–989.
[12] W. Wang, K. Zhu, L. Ying, J. Tan, L. Zhang, Map task scheduling in MapReduce with data locality: throughput and heavy-traffic optimality, in: Proc. IEEE Int. Conf. Computer Communications, INFOCOM, Turin, Italy, 2013, pp. 1609–1617.
[13] J. Tan, X. Meng, L. Zhang, Coupling task progress for MapReduce resource-aware scheduling, in: Proc. IEEE Int. Conf. Computer Communications, INFOCOM, Turin, Italy, 2013, pp. 1618–1626.
[14] X. Bu, J. Rao, C.-Z. Xu, Interference and locality-aware task scheduling for MapReduce applications in virtual clusters, in: Proc. ACM Int. Symp. High-Performance Parallel and Distributed Computing, HPDC, New York City, NY, 2013, pp. 227–238.
[15] W. Wang, K. Zhu, L. Ying, J. Tan, L. Zhang, MapTask scheduling in MapReduce with data locality: Throughput and heavy-traffic optimality, IEEE/ACM Trans. Netw. (2014) in press.

[16] K. Wang, X. Zhou, T. Li, D. Zhao, M. Lang, I. Raicu, Optimizing load balancing and data-locality with data-aware scheduling, in: Proc. IEEE Int. Conf. Big Data, Big Data, Washington DC, 2014, pp. 119–128.
[17] M. Sun, H. Zhuang, X. Zhou, K. Lu, C. Li, HPSO: Prefetching based scheduling to improve data locality for MapReduce clusters, in: Int. Conf. Algorithms and Architectures for Parallel Processing (ICA3PP), in: Lecture Notes in Comput. Sci., vol. 8631, Springer, 2014, pp. 82–95.
[18] W. Wang, M. Barnard, L. Ying, Decentralized scheduling with data locality for data-parallel computation on peer-to-peer networks, in: Proc. Ann. Allerton Conf. Commununication, Control and Computing, Monticello, IL, 2015.
[19] Q. Xie, Y. Lu, Priority algorithm for near-data scheduling: Throughput and heavy-traffic optimality, in: Proc. IEEE Int. Conf. Computer Communications, INFOCOM, Hong Kong, China, 2015, pp. 963–972.
[20] M. Al-Fares, A. Loukissas, A. Vahdat, A scalable, commodity data center network architecture, in: Proc. Ann. ACM SIGCOMM Conf., Seattle, WA, 2008, pp. 63–74.
[21] L. Barroso, U. Hölzle, The datacenter as a computer: An introduction to the design of warehouse-scale machines, Synthesis Lectures on Comput. Architecture 4 (1) (2009) 1–108.
[22] R. Srikant, L. Ying, Communication Networks: An Optimization, Control and Stochastic Networks Perspective, Cambridge Univ. Press, New York, 2014.
[23] M. Andrews, K. Kumaran, K. Ramanan, A. Stolyar, R. Vijayakumar, P. Whiting, Scheduling in a queueing system with asynchronously varying service rates, Probab. Engrg. Inform. Sci. 18 (2) (2004) 191–217.
[24] W. Wang, L. Ying, Data locality in MapReduce: A network perspective, Tech. rep., Arizona State Univ., Tempe, AZ, 2015.
[25] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, OpenFlow: Enabling innovation in campus networks, ACM SIGCOMM Comput. Commun. Rev. 38 (2) (2008) 69–74.



Weina Wang received her B.E. degree in Electronic Engineering from Tsinghua University, Beijing, China, in 2009. She is currently pursuing a Ph.D. degree in the School of Electrical, Computer and Energy Engineering at Arizona State University, Tempe, AZ.

Her research interests include resource allocation in stochastic networks, data privacy and game theory. She won the Joseph A. Barkson Fellowship for the 2015–16 academic year.

Lei Ying (M'08) received his B.E. degree from Tsinghua University, Beijing, China, and his M.S. and Ph.D. in Electrical and Computer Engineering from the University of Illinois at Urbana-Champaign. He currently is an Associate Professor at the School of Electrical, Computer and Energy Engineering at Arizona State University, and an Associate Editor of the IEEE/ACM Transactions on Networking.

His research interest is broadly in the area of stochastic networks, including cloud computing, communication networks and social networks. He is coauthor with R. Srikant of the book Communication Networks: An Optimization, Control and Stochastic Networks Perspective, Cambridge University Press, 2014. The book has been selected as a notable book in the Computing Reviews' 19th Annual Best of Computing list.

He won the Young Investigator Award from the Defense Threat Reduction Agency (DTRA) in 2009 and NSF CAREER Award in 2010. He was the Northrop Grumman Assistant Professor in the Department of Electrical and Computer Engineering at Iowa State University from 2010 to 2012. He received the best paper award at IEEE INFOCOM 2015.