An Alternating Direction Method Approach to Cloud Traffic Management

Chen Feng, *Member, IEEE,* Hong Xu, *Member, IEEE,* and Baochun Li, *Fellow, IEEE*

Abstract—In this paper, we introduce a unified framework for studying various cloud traffic management problems, ranging from geographical load balancing to backbone traffic engineering. We first abstract these real-world problems as a multi-facility resource allocation problem, and then present two distributed optimization algorithms by exploiting the special structure of the problem. Our algorithms are inspired by Alternating Direction Method of Multipliers (ADMM), enjoying a number of unique features. Compared to dual decomposition, they converge with non-strictly convex objective functions; compared to other ADMM-type algorithms, they not only achieve faster convergence under weaker assumptions, but also have lower computational complexity and lower message-passing overhead. The simulation results not only confirm these desirable features of our algorithms, but also highlight several additional advantages, such as scalability and fault-tolerance.

Index Terms—cloud traffic management, load balancing, traffic engineering, datacenters, ADMM, distributed optimization

1 Introduction

Cloud services (such as search, social networking, etc.) are often deployed on a geographically distributed infrastructure, i.e., data centers located in different regions. In order to optimize the efficiency of these data centers, how to orchestrate the data transmission, including traffic flowing from users to the infrastructure to access the cloud services, and traffic flowing across these data centers for back-end services, has started to receive an increasing amount of attention. We refer to these problems generally as *cloud traffic management* herein.

In this paper, we introduce a unified framework for studying various cloud traffic management problems, ranging from geographical load balancing to backbone traffic engineering. As we will see in Sec. 2, a large variety of cloud traffic management problems can be abstracted into the following:

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{N} f_i(x_{i1},\ldots,x_{in}) - \sum_{j=1}^{n} g_j(y_j) \qquad (1)\\ \text{subject to} \quad & \forall j : \sum_{i=1}^{N} x_{ij} = y_j \\ & \forall i : x_i = (x_{i1},\ldots,x_{in})^T \in \mathcal{X}_i \subseteq \mathbb{R}^n \\ & \forall j : y_j \in \mathcal{Y}_j \subseteq \mathbb{R}. \end{aligned}$$

Generically, problem (1) amounts to allocating resources from n facilities to N users such that the "social welfare" (i.e., utility minus cost) is maximized. We thus refer to problem (1) as the *multi-facility resource allocation problem*. The utility function $f_i(x_i)$ represents the performance, or the level of satisfaction, of user i when she receives an amount x_{ij} of resources from each facility j, where $x_i = (x_{i1},\ldots,x_{in})^T$. In practice, this performance measure can be in terms of revenue, throughput, or average latency, depending on the problem setup. We assume throughout the paper that $f_i(\cdot)$ are concave. The cost function $g_j(y_j)$ represents the operational expense or congestion cost when facility j allocates an amount y_j of resources to all the users. Note that y_j is the sum of x_{ij} (over i), since each facility often cares about the total amount of allocated resources. We assume that $g_j(\cdot)$ are convex. The constraint sets $\{\mathcal{X}_i\}$ and $\{\mathcal{Y}_j\}$ are used to model the additional constraints, which are assumed to be compact convex sets.

To tackle the multi-facility resource allocation problem, we are particularly interested in solutions that are amenable to *parallel* implementations. There are several reasons. ***First***, for a production cloud, problem (1) is inherently a large-scale convex optimization problem, with millions of variables or even more. A centralized solver is highly inefficient in solving such large-scale problems [10]. Indeed, as we shall show in Sec. 4.2, a state-of-the-art centralized solver cannot solve an instance of problem (1) with a moderate size after 16 hours on a modern server. Moreover, with the upcoming Internet of Things where each home or business may host hundreds of devices, the scale of problem (1) will increase dramatically, making centralized solvers impractical. ***Second***, unlike conventional traffic engineering, cloud traffic management may operate at much finer time-scales (on the order of seconds) [38]. Such a stringent requirement favors a fast and scalable implementation of the underlying optimization algorithm on multi-core CPUs [42]. ***Third***, a cloud provider usually has abundant servers and CPU cores, which can be easily utilized to implement various distributed and parallel solutions.

The standard approach to constructing parallel algorithms is dual decomposition with (sub)gradient methods. However, it suffers from several difficulties for problem (1). First, dual decomposition requires a delicate adjustment of the step-size parameters, which have a strong influence on the convergence rate. Second, dual decomposition requires the utility functions $f_i(\cdot)$ to be strictly concave *and* the cost functions $g_j(\cdot)$ to be strictly convex to achieve convergence. These requirements cannot be met in many problem settings of (1) as we will

- *Chen Feng is with the School of Engineering, University of British Columbia, Kelowna, British Columbia V1V 1V7, Canada. Email: chen.feng@ubc.ca. Hong Xu is with the Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong, China. Email: henry.xu@cityu.edu.hk. Baochun Li is with The Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ontario M5S 3G4, Canada. Email: bli@ece.toronto.edu.*
- *This work was supported in part by the NSERC Discovery Grants (RGPIN-2016-05310), Hong Kong RGC ECS-21201714 and CRF-C7036-15G.*

demonstrate in Sec. 2.

To overcome these difficulties, in a series of our previous work [47–49], we have developed a new decomposition method for problem (1) based on *alternating direction method of multipliers* (ADMM)—a simple yet powerful method that has recently found practical use in many large-scale convex optimization problems [6]. Unlike dual decomposition, an ADMM-based method uses a *single* parameter, which is much easier to tune and often leads to fast convergence. Similar applications of ADMM to radio-access networks and fuel cell generation in geo-distributed cloud have also been proposed in [30] and [52], respectively.

Although ADMM has been widely applied to areas of machine learning and signal processing, its application to networking research is still in an early stage. Despite the success of the above pioneer works, several fundamental questions still remain open. **First**, most prior work on the application of ADMM is somewhat ad-hoc, proposing a number of variations of the classical ADMM algorithm. For example, the work of [47, 48] is based on a multi-block variant of ADMM, and the work of [52] combines multi-block ADMM with Gaussian back substitution. *Can we develop a simple framework that unifies (and hopefully improves) the prior art?* **Second**, prior work often requires the utility functions to be strictly concave or the cost functions to be strictly convex. For instance, our previous algorithms [47, 48] require strictly convex objective functions and bounded level sets. *Can we relax these technical assumptions while still achieving fast convergence?* **Finally**, parallel implementation of ADMM-based algorithms may incur some extra computational costs and message-passing overheads. *How can we reduce these computational costs and message-passing overheads as much as possible?*

At first glance, it seems difficult to develop a unified framework that requires weaker technical assumptions to ensure convergence, and, at the same time, enjoys lower computational complexity and message-passing overhead. We achieve this objective by making use of the following two observations.

1) The multi-facility formulation can be made general enough by introducing "virtual" users. As we will see in Sec. 2, with the help of "virtual" users, problem (1) contains the previous formulations in [47–49, 52] as special cases.
2) The multi-facility problem can be efficiently solved by a distributed algorithm proposed in Chapter 7 of [6], which, however, has been somewhat overlooked by networking researchers.

In this paper, we first demonstrate the usefulness of the (overlooked) distributed algorithm [6, Chapter 7] in the context of cloud traffic management through a convergence analysis. We then develop a new distributed algorithm that complements the aforementioned algorithm with respect to the rate of convergence. Together, these two algorithms not only overcome the shortcomings of dual decomposition, but also achieve faster convergence under weaker technical assumptions and enjoy lower overhead compared to previous ADMM-based algorithms.

Finally, we present an extensive empirical study of these two algorithms. Our simulation results reveal some additional advantages of these algorithms, including their scalability to a large number of users and their fault-tolerance with respect to updating failures.

The main contributions of this paper are as follows:

1) We identify a variety of cloud traffic management problems as instances of the multi-facility resource allocation problem (1).
2) We highlight the usefulness of a known algorithm and develop a new algorithm for problem (1). We show that these two distributed algorithms enjoy a number of unique advantages over dual decomposition and previous ADMM-based algorithms.
3) We present extensive simulation results, which further demonstrate the scalability and fault-tolerance of these algorithms.

2 Applications to Cloud Traffic Management

In this section, we show that a large variety of optimization problems in the context of cloud traffic management are indeed instances of the multi-facility resource allocation problem (1). In particular, we illustrate the inherent large scale of these problems for production systems, and explain why the utility function is non-strictly concave and the cost function is non-strictly convex for some applications.

2.1 Geographical Load Balancing

2.1.1 Background

Cloud services, such as search, social networking, etc., are often deployed on a geographically distributed infrastructure, i.e. data centers located in different regions as shown in Fig. 1, for better performance and reliability. A natural question is then how to direct the workload from users among the set of geo-distributed data centers in order to achieve a desired trade-off between performance and cost, since the energy price exhibits a significant degree of geographical diversity [43]. This question has attracted much attention recently [17, 33, 34, 43, 47–49], and is generally referred to as geographical load balancing.

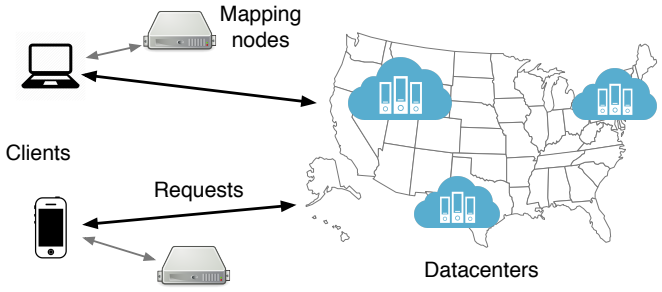

Fig. 1. A cloud service running on geographically distributed data centers.

2.1.2 Basic Model

We now introduce a formulation for the basic geographical load balancing problem, which captures the essential performance-cost trade-off and covers many existing works [17, 34, 43, 47–49]. Here, we define a user to be an group of customers aggregated from a common geographical region sharing a unique IP prefix, as is often done in practice to reduce complexity [40]. We use x_{ij} to denote the amount of workload coming from user i and directed to data center j. We use t_i to denote the total workload of each user. We use $f_i(\cdot)$ to represent the utility of user i, and use $g_j(\cdot)$ to represent the

cost of data center j. These functions can take various forms depending on the scenario as we will elaborate soon.

With these notations, we formulate the basic geographical load balancing problem:

$$\text{maximize} \quad \sum_i f_i(x_i) - \sum_j g_j(y_j) \tag{2}$$

$$\text{subject to} \quad \forall i: \sum_j x_{ij} = t_i, x_i \in \mathbb{R}_+^n, \tag{3}$$

$$\forall j: \ y_j = \sum_i x_{ij} \le c_j, \tag{4}$$

where (3) describes the workload conservation and non-negativity constraint, and (4) is the capacity constraint at data centers. Since the constraint (3) can be rewritten as $\forall i : x_i \in \mathcal{X}_i$, where $\mathcal{X}_i$ is a convex set, problem (2) is an instance of problem (1).

Now, let us consider the utility function $f_i(\cdot)$. Latency is arguably the most important performance metric for most interactive services: A small increase in the user-perceived latency can cause substantial utility loss for the users [29]. The user-perceived latency largely depends on the end-to-end propagation latency [16, 39], which can be obtained through active measurements. Let l_{ij} denote the end-to-end propagation latency between user i and data center j. The following utility function f_i has been used in [47, 48]

$$f_i(x_i) = -qt_i \left(\sum_j x_{ij} l_{ij}/t_i \right)^2. \tag{5}$$

Here, q is the weight factor that captures the relative importance of performance compared to cost in monetary terms. Clearly, the utility function $f_i(\cdot)$ achieves its maximum value when latency is zero. Also, the function $f_i(\cdot)$ depends on the *average latency* $\sum_j x_{ij} l_{ij}/t_i$. For different applications, f_i may depend on other aggregate statistics of the latency, such as the maximum latency or the 99-th percentile latency, which may be modeled after a norm function.

For the cost function $g_j(\cdot)$, many existing works consider the following [17, 34, 43, 49]

$$g_j(y_j) = P_j^E \cdot \text{PUE} \cdot E(y_j). \tag{6}$$

Here, P_j^E denotes the energy price in terms of \$/KWh at data center j. PUE, power usage effectiveness, is the ratio between total infrastructure power and server power. Since total infrastructure power mainly consists of server power and cooling power, PUE is commonly used as a measure of data center energy efficiency. Finally, $E(y_j)$ represents the server power at data center j, which is a function of the total workload y_j and can be obtained empirically. A commonly used server power function is from a measurement study of Google [14]:

$$E(y_j) = c_j P_{\text{idle}} + \left(P_{\text{peak}} - P_{\text{idle}}\right) y_j, \tag{7}$$

where P_{idle} is server idle power and P_{peak} peak power.

2.1.3 Problem Scale

The geographical load balancing problem (2) would be easy to solve, if its scale is small with, say, hundreds of variables. However, for a production cloud, (2) is inherently an extremely large-scale optimization. In practice, the number of users N (unique IP prefixes) is on the order of $\mathcal{O}(10^5)$ [40]. Thus the number of variables $\{x_{ij}\}$ is $\mathcal{O}(10^6)$. The load balancing decision usually needs to be updated on a hourly basis, or even more frequently, as demand varies dynamically. The conventional dual decomposition approach suffers from many performance issues for solving such large-scale problems, as we argued in Sec. 1. Thus we are motivated to consider new distributed optimization algorithms.

2.1.4 Extensions

In this section, we provide some additional extensions of the basic model (2) from the literature to demonstrate its importance and generality.

Minimizing Carbon Footprint. In (2), the monetary cost of energy is modeled. The environmental cost of energy, i.e., the carbon footprint of energy can also be taken into account. Carbon footprint also has geographical diversity due to different sources of electricity generation in different locations [17]. Hence, it can be readily modeled by having an additional carbon cost P_j^C in terms of average carbon emission per KWh in the objective function of (2) following [17, 34].

Joint Optimization with Batch Workloads. There are also efforts [33, 47, 48] that consider the delay-tolerant batch workloads in addition to interactive requests, and the integrated workload management problem. Examples of batch workloads include MapReduce jobs, data mining tasks, etc. Batch workloads provides additional flexibility for geographical load balancing: Since their resource allocation is elastic, when the demand spikes we can allocate more capacity to run interactive workloads by reducing the resources for batch workloads.

To incorporate batch workloads, we introduce n "virtual" users, where user j generates batch workloads running on data center j. Let w_j be the amount of resource used for batch workloads on data center j, and let $\tilde{f}_j(w_j)$ be the utility of these batch workloads. Then the joint optimization can be formulated as follows:

$$\text{maximize} \quad \sum_i f_i(x_i) + \sum_j \tilde{f}_j(w_j) - \sum_j g_j(y_j)$$

$$\text{subject to} \quad \forall i: \sum_j x_{ij} = t_i, \ x_i \in \mathbb{R}_+^n; w \in \mathbb{R}_+^n$$

$$\forall j: \ y_j = \sum_i x_{ij} + w_j \le c_j.$$

The utility function $\tilde{f}_j(\cdot)$ depends only on w_j but not on latency, due to its elastic nature. In general, $\tilde{f}_j(\cdot)$ is an increasing and concave function, such as the log function used in [47, 48]. Clearly, this is still an instance of (1).

2.2 Backbone Traffic Engineering

2.2.1 Background

Large cloud service providers, such as Google and Microsoft, usually interconnect their geo-distributed data centers with a private backbone wide-area networks (WANs). Compared to ISP WANs, data center backbone WANs exhibit unique characteristics [18, 27]. First, they are increasingly taking advantage of the software-defined networking (SDN) architecture, where a logically centralized controller has global knowledge and coordinates all transmissions [7, 19]. SDN paves the way for implementing logically centralized traffic engineering. In addition, the majority of the backbone traffic, such as copying user data to remote data centers and synchronizing large data sets across data centers, is elastic. Thus, since the cloud service provider controls both the applications at the edge and the

routers in the network, in addition to routing, it can perform application rate control, i.e., allocate the aggregated sending rate of each application, according to the current network state. These characteristics open up the opportunity to perform joint rate control and traffic engineering in backbone WANs, which is starting to receive attention in the networking community [18, 24, 27].

2.2.2 Basic Model

We model the backbone WAN as a set $\mathcal{J}$ of interconnecting links. Conceptually, each cloud application generates a *flow* between a source-destination pair of data centers. We index the flows by i, and denote by $\mathcal{I}$ the set of all flows. We assume that each flow can use multiple paths from its source to destination. This is because multi-path routing is relatively easy to implement (e.g., using MPLS [13, 24, 27]) and offers many benefits. For each flow i, we denote by $\mathcal{P}_i$ the set of its available paths and define a *topology matrix* A_i of size $|\mathcal{J}| \times |\mathcal{P}_i|$ as follows:

$$A_i[j,p] = \begin{cases} 1, & \text{if link } j \text{ lies on path } p \\ 0, & \text{otherwise.} \end{cases}$$

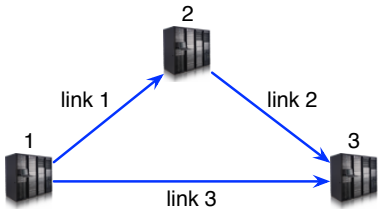

Fig. 2. An illustration of three data centers with 3 links.

For example, consider a network with three data centers and 3 links as illustrated in Fig. 2. A flow (say, flow 1) from data center 1 to data center 3 has two paths: $\{\text{link } 1, \text{link } 2\}$ and $\{\text{link } 3\}$. In this case, $|\mathcal{J}| = 3$, $|\mathcal{P}_1| = 2$, and the topology matrix A_1 is

$$A_1 = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Clearly, the topology matrix A_i provides a mapping from paths to links. Let w_{ip} denote the amount of traffic of flow i on path p, and let x_{ij} denote the amount of traffic of flow i on link j. Then we have $x_i = A_i w_i$, where $w_i = (w_{i1}, \ldots, w_{i|\mathcal{P}_i|})^T$. Since A_i is always full column-rank (otherwise some path must be redundant), A_i has a left-inverse A_i^{-1} such that $w_i = A_i^{-1} x_i$. For instance, a left-inverse of A_1 in the previous example is

$$A_1^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Note that w_i models the rate control decision for each application flow. A flow corresponds to potentially many TCP connections between a particular source-destination pair of data centers, carrying traffic for this particular application. We choose to model rate control at the application flow level because the latest data center backbone architectures [24, 27] are designed to control the aggregated sending rates of applications across data centers. The aggregated rate can be readily apportioned among different connections following some notion of fairness, and rate control can be enforced by adding a shim layer in the servers' operating system and using a per-destination token bucket [2].

We use $f_i(w_i)$ to represent the utility of flow i, and $g_j(y_j)$ to represent the congestion cost of link j, where $y_j = \sum_i x_{ij}$ is the total traffic on link j. The joint rate control and traffic engineering problem can be formulated as

$$\text{maximize} \quad \sum_i f_i(A_i^{-1} x_i) - \sum_j g_j(y_j) \tag{8}$$

$$\text{subject to} \quad \forall i: \; x_i \in \mathbb{R}_+^n, \tag{9}$$

$$\forall j: \; y_j = \sum_i x_{ij} \le c_j, \tag{10}$$

where (9) describes the non-negativity constraint, and (10) says that the total traffic on link j cannot exceed the capacity c_j. Clearly, problem (8) is again an instance of problem (1).

The utility function $f_i(w_i)$ should be concave, such as the log function $f_i(w_i) = \log(\sum_p w_{ip})$, or a more general "rate-fairness" function used for Internet TCP congestion control [37]. It is worth noting that even if $f_i(w_i)$ is strictly concave (with respect to w_i), $f_i(A_i^{-1} x_i)$ is *not* strictly concave (with respect to x_i) in general. This important fact has been used in Sec. 3.4 to demonstrate the advantages of our distributed algorithms. The cost function $g_j(y_j)$ is convex and non-decreasing. For example, the function can be a piece-wise linear function with increasing slopes, which is used in [18].

Finally, note that the topology matrix A_i only depends on the source-destination pair. Hence, for a given source data center, the number of all possible topology matrices is bounded by the number of all other data centers, which is typically less than 30. In other words, the topology matrices are easy to store and maintain in practice. Note also that all the inverse matrices can be computed before the algorithm runs. That is, there is no need to calculate any A_i^{-1} on the fly.

2.2.3 Problem Scale

Similar to the geographical load balancing problem, backbone traffic engineering is also a large-scale optimization problem for a production data center backbone WAN. In practice, a provider runs hundreds to thousands of applications with around ten data centers [24, 27]. Thus the number of application flows is $\mathcal{O}(10^5)$ to $\mathcal{O}(10^6)$. For a WAN with tens of links, we potentially have tens of millions of variables $\{x_{ij}\}$. Compared to geographical load balancing, the traffic engineering decisions need to be updated over a very small time window (say, every 5 or 10 minutes as in [24, 27]) to cope with traffic dynamics. This further motivates us to derive a fast distributed solution.

2.2.4 Extensions

We present some possible extensions of the basic model.

Minimizing Bandwidth Costs. Unlike big players like Google and Microsoft, small cloud providers often rely on ISPs to interconnect their data centers. In this case, bandwidth costs become one of the most important operating expenses. Although many ISPs adopt the 95-percentile charging scheme in reality, the link bandwidth cost is often assumed to be linear with the link traffic, because optimizing a linear cost in each interval can reduce the monthly 95-percentile bill [50]. Hence, the bandwidth cost can be easily incorporated by adding these linear functions to (8).

Incrementally Deployed SDN. Instead of upgrading all routers to be SDN-capable with a daunting bill, cloud providers could deploy SDN incrementally [1]. In such a

scenario, some routers still use standard routing protocols such as OSPF, while other routers have the flexibility to choose the next hop. This scenario can be easily handled by imposing additional constraints on the set $\mathcal{P}_i$ of available paths such that $\mathcal{P}_i$ only contains *admissible* paths. (See Definition 1 in [1] for details.) Clearly, with some routers restricted to standard protocols, the number $|\mathcal{P}_i|$ of available paths for flow i is reduced, resulting in a smaller-scale optimization problem.

3 ADMM-Based Distributed Algorithms

In this section, we will present two ADMM-based distributed algorithms that are well suited for the multi-facility resource allocation problem, with a particular focus on their convergence rates as well as their advantages over other ADMM-based algorithms.

3.1 A Primer on Dual Decomposition and ADMM

We begin with some basics of optimization techniques, including dual decomposition and ADMM, which will be used throughout the paper. We refer our readers to [4, 6, 23] for more details. A recent extension of ADMM to non-convex settings are in [26]

3.1.1 Dual decomposition

Dual decomposition is a standard approach to solving large-scale convex problems, which has been widely used in the networking research. Consider a convex optimization problem with linear constraints:

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & Ax = b \\ & x \in \mathcal{X} \subseteq \mathbb{R}^n \end{aligned}$$

where x is the variable, $f(x)$ is a convex function, and A is a matrix of size $m \times n$. The basic idea behind dual decomposition is to convert an optimization problem with constraints into a new problem without constraints.

We define the *Lagrange* of the above problem as

$$L(x, \lambda) = f(x) + \sum_{i=1}^{m} \lambda_i (A_i x - b_i),$$

where λ_i is called the Lagrange multiplier associated with the ith constraint, and the vector $\lambda = (\lambda_1, \ldots, \lambda_m)$ is called the dual variables. For any given λ, we define the *Lagrange dual function* as the minimum value of the Lagrangian $L(x, \lambda)$ over x:

$$g(\lambda) = \inf_{x \in \mathcal{X}} L(x, \lambda) = \inf_{x \in \mathcal{X}} f(x) + \lambda^T (Ax - b). \qquad (11)$$

The dual function $g(\lambda)$ has several interesting properties. First, it amounts to solving an *unconstrained* optimization problem. Second, it gives lower bounds on the optimal value of the original problem, i.e., $g(\lambda) \leq f(x^*)$, where x^* is an optimal solution to the original problem. This is because x^* is also a feasible solution to the unconstrained problem (11) for any given λ. It turns out that the "best" lower bound matches the optimal value. That is, $g(\lambda^*) = f(x^*)$, where λ^* maximizes $g(\lambda)$. These two properties suggest that, instead of solving the original constrained optimization problem, we can choose to solve a series of unconstrained problems as follows:

$$\begin{aligned} x^{k+1} &:= \operatorname*{argmin}_{x \in \mathcal{X}} L(x, \lambda^k), \\ \lambda^{k+1} &:= \lambda^k + \rho(Ax^{k+1} - b), \end{aligned}$$

where x^0 and λ^0 are some initial values, and the parameter ρ is the step size for the update of the dual variable λ.

The rationale is that $Ax^{k+1} - b$ is a subgradient of $g(\lambda)$ at λ^k. Hence, the above iterations amount to using the subgradient method to maximize $g(\lambda)$. Sometimes, we can vary the step size ρ across different iterations, and we use ρ^k to denote the step-size for the kth iteration.

Applying the above iterations to problem (1), we arrive at the following algorithm:

Algorithm 3.1. Initialize $\{x_i^0\}$, $\{y_j^0\}$, $\{\lambda_j^0\}$. For $k = 0, 1, \ldots,$ repeat

1) **x-update:** Each user i solves the following sub-problem for x_i^{k+1}:

$$\begin{aligned} \min \quad & -f_i(x_i) + (\lambda^k)^T x_i \\ \text{s.t.} \quad & x_i \in \mathcal{X}_i. \end{aligned}$$

2) **y-update:** Each facility j solves the following sub-problem for y_j^{k+1}:

$$\begin{aligned} \min \quad & g_j(y_j) - \lambda_j^k y_j \\ \text{s.t.} \quad & y_j \in \mathcal{Y}_j. \end{aligned}$$

3) **Dual update:** Each facility j updates λ_j^{k+1}:

$$\lambda_j^{k+1} := \lambda_j^k + \rho^k \left(\sum_{i=1}^{N} x_{ij}^{k+1} - y_j^{k+1} \right).$$

The following (very mild) assumption is valid throughout the paper:

Assumption 3.1. The optimal solution set of problem (1) is non-empty, and the optimal value p^* is finite.

It is known that Algorithm 3.1 is convergent under Assumption 3.1 and the assumption that the utility functions $f_i(\cdot)$ are strictly concave and the cost functions $g_j(\cdot)$ are strictly convex [4][1]. However, as we have shown in Sec. 2, for many interesting problems of form (1), either $f_i(\cdot)$ are *non-strictly* concave or $g_j(\cdot)$ are *non-strictly* convex, making conventional dual decomposition unsuitable for such applications.

3.1.2 Alternating direction method of multipliers

Alternating direction method of multipliers (ADMM) is a decomposition method that does not require strict convexity. Originally proposed in the 1970s, ADMM has recently received much research attention and found practical use in many areas, due to its superior empirical performance in solving large-scale convex optimization problems [6]. While the convergence of ADMM is well known in the literature (see, e.g., [4, 6]), its rate of convergence has only been established very recently (see, e.g., [12, 21, 22]).

1. Otherwise, if $f_i(\cdot)$ are non-strictly concave or $g_j(\cdot)$ are non-strictly convex, the primal variable x_{ij} in problem (1) will not converge, leading to the so-called oscillation problem.

ADMM solves convex optimization problems in the form

$$\begin{array}{ll} \text{minimize} & f(x)+g(y) \\ \text{subject to} & Ax+By=c, \\ & x\in\mathcal{X},\ y\in\mathcal{Y}, \end{array} \tag{12}$$

with variables $x\in\mathbb{R}^n$ and $y\in\mathbb{R}^m$, where $f:\mathbb{R}^n\to\mathbb{R}$ and $g:\mathbb{R}^m\to\mathbb{R}$ are convex functions, $A\in\mathbb{R}^{p\times n}$ and $B\in\mathbb{R}^{p\times m}$ are matrices, $\mathcal{X}$ and $\mathcal{Y}$ are nonempty compact convex subsets of $\mathbb{R}^n$ and $\mathbb{R}^m$, respectively. Note that $f(\cdot)$ and/or $g(\cdot)$ are *not* assumed to be strictly convex.

The *augmented Lagrangian* for problem (12) is

$$L_\rho(x,y,\lambda)=f(x)+g(y)+\lambda^T(Ax+By-c) +(\rho/2)\|Ax+By-c\|_2^2,$$

where $\lambda\in\mathbb{R}^p$ is the Lagrange multiplier (or the dual variable) for the equality constraint, and $\rho>0$ is the *penalty parameter*. Clearly, L_0 is the (standard) Lagrangian for (12), and L_ρ is the sum of L_0 and a *penalty term* $(\rho/2)\|Ax+By-c\|_2^2$.

The standard ADMM algorithm solves problem (12) with the iterations:

$$\begin{aligned} x^{k+1}&:=\operatorname*{argmin}_{x\in\mathcal{X}} L_\rho(x,y^k,\lambda^k),\\ y^{k+1}&:=\operatorname*{argmin}_{y\in\mathcal{Y}} L_\rho(x^{k+1},y,\lambda^k),\\ \lambda^{k+1}&:=\lambda^k+\rho(Ax^{k+1}+By^{k+1}-c), \end{aligned}$$

where the penalty parameter ρ can be viewed as the step size for the update of the dual variable λ. Note that the primal variables x and y are updated in an alternating fashion, which accounts for the term *alternating direction*.

To better understand the standard ADMM, we can compare it to the dual-decomposition algorithm for the following optimization problem

$$\begin{array}{ll} \text{minimize} & f(x)+g(y)+\dfrac{\rho}{2}\|Ax+By-c\|_2^2 \\ \text{subject to} & Ax+By=c, \\ & x\in\mathcal{X},\ y\in\mathcal{Y}, \end{array} \tag{13}$$

with the iterations

$$\begin{aligned} \left(x^{k+1},y^{k+1}\right)&:=\operatorname*{argmin}_{x\in\mathcal{X},y\in\mathcal{Y}} L(x,y,\lambda^k),\\ \lambda^{k+1}&:=\lambda^k+\rho(Ax^{k+1}+By^{k+1}-c). \end{aligned}$$

The key difference is that ADMM updates the primal variables x and y alternatively, whereas dual decomposition updates x and y jointly. Intuitively, the alternating update in ADMM reduces the complexity, and the extra term $(\rho/2)\|Ax+By-c\|_2^2$ stabilizes the iterations.

The standard ADMM algorithm has a *scaled form*, which is often more convenient (and will be used in this paper). Introducing $u=(1/\rho)\lambda$ and combining the linear and quadratic terms in the augmented Lagrangian, we can express the ADMM algorithm as

$$\begin{aligned} x^{k+1}&:=\operatorname*{argmin}_{x\in\mathcal{X}}\left(f(x)+(\rho/2)\|Ax+By^k-c+u^k\|_2^2\right),\\ y^{k+1}&:=\operatorname*{argmin}_{y\in\mathcal{Y}}\left(g(y)+(\rho/2)\|Ax^{k+1}+By-c+u^k\|_2^2\right),\\ u^{k+1}&:=u^k+Ax^{k+1}+By^{k+1}-c. \end{aligned}$$

Applying this algorithm to problem (1), we obtain the following algorithm:

Algorithm 3.2. Initialize $\{x_i^0\}$, $\{y_j^0\}$, $\{u_j^0\}$. For $k=0,1,\ldots,$ repeat

1) **x-update:** The users jointly solve the following problem for $\{x_i^{k+1}\}$:

$$\begin{array}{ll} \min & -\sum_{i=1}^N f_i(x_i)+(\rho/2)\|\sum_i x_i-y^k+u^k\|_2^2 \\ \text{s.t.} & \forall i: x_i\in\mathcal{X}_i. \end{array}$$

2) **y-update:** Each facility j solves the following subproblem for y_j^{k+1}:

$$\begin{array}{ll} \min & g_j(y_j)+(\rho/2)\left(\sum_{i=1}^N x_{ij}^{k+1}-y_j+u_j^k\right)^2 \\ \text{s.t.} & y_j\in\mathcal{Y}_j. \end{array}$$

3) **Dual update:** Each facility j updates u_j^{k+1}:

$$u_j^{k+1}:=u_j^k+\sum_{i=1}^N x_{ij}^{k+1}-y_j^{k+1}.$$

It is known that Algorithm 3.2 is convergent under Assumption 3.1. Thus, it solves the oscillation problem of dual decomposition. However, the x-update requires all the users to solve a joint optimization due to the penalty term $(\rho/2)\|\sum_i x_i-y^k+u^k\|_2^2$, which is undesirable for large-scale systems. By contrast, the x-update in dual decomposition can be performed independently by the users.

3.2 Distributed ADMM Algorithms

We now present two distributed ADMM algorithms, as well as their convergence analysis. Both algorithms can be viewed as variants of the standard ADMM algorithm.

3.2.1 An overlooked algorithm

The first algorithm was proposed in [6, Chapter 7], but was somehow overlooked by networking researchers.

Algorithm 3.3. Initialize $\{x_i^0\}$, $\{y_j^0\}$, $\{u_j^0\}$. For $k=0,1,\ldots,$ repeat

1) **x-update:** Each user i solves the following sub-problem for x_i^{k+1}:

$$\begin{array}{ll} \min & -f_i(x_i)+(\rho/2)\|x_i-x_i^k+d^k\|_2^2 \\ \text{s.t.} & x_i\in\mathcal{X}_i, \end{array}$$

where $d^k\triangleq(1/N)\left(u^k+\sum_{i=1}^N x_i^k-y^k\right)$.

2) **y-update:** Each facility j solves the following subproblem for y_j^{k+1}:

$$\begin{array}{ll} \min & g_j(y_j)+(\rho/2N)\left(y_j-\sum_{i=1}^N x_{ij}^{k+1}-u_j^k\right)^2 \\ \text{s.t.} & y_j\in\mathcal{Y}_j. \end{array}$$

3) **Dual update:** Each facility j updates u_j^{k+1}:

$$u_j^{k+1}:=u_j^k+\sum_{i=1}^N x_{ij}^{k+1}-y_j^{k+1}.$$

3.2.2 A new algorithm

The second algorithm switches the order of x-update and y-update, leading to a new set of convergence conditions that complements those for Algorithm 3.3.

Algorithm 3.4. Initialize $\{x_i^0\}$, $\{y_j^0\}$, $\{u_j^0\}$. For $k = 0, 1, \ldots,$ repeat

1) **y-update:** Each facility j solves the following sub-problem for y_j^{k+1}:

$$\begin{aligned} \min \quad & g_j(y_j) + (\rho/2N)\left(y_j - \sum_{i=1}^N x_{ij}^k - u_j^k\right)^2 \\ \text{s.t.} \quad & y_j \in \mathcal{Y}_j. \end{aligned}$$

2) **x-update:** Each user i solves the following sub-problem for x_i^{k+1}:

$$\begin{aligned} \min \quad & -f_i(x_i) + (\rho/2)\|x_i - x_i^k + d^k\|_2^2 \\ \text{s.t.} \quad & x_i \in \mathcal{X}_i, \end{aligned}$$

where $d^k \triangleq (1/N)\left(u^k + \sum_{i=1}^N x_i^k - y^{k+1}\right)$.

3) **Dual update:** Each facility j updates u_j^{k+1}:

$$u_j^{k+1} := u_j^k + \sum_{i=1}^N x_{ij}^{k+1} - y_j^{k+1}.$$

Clearly, Algorithm 3.3 and Algorithm 3.4 enjoy the best of both worlds: independent x-updates by the users and convergence of primal variables. As such, they indeed overcome the shortcomings of dual decomposition and Algorithm 3.2. More interestingly, they are complementary to each other with respect to the rate of convergence, as we will see in later sections.

3.2.3 Connection to the standard ADMM

The connection between Algorithm 3.3 and the standard ADMM was first noted in [6, Chapter 7], which is provided below for completeness. The key idea is to introduce auxiliary variables and make a clever use of the structure of dual variables.

We write $x = (x_1^T, \ldots, x_N^T)^T$, $f(x) = -\sum_{i=1}^N f_i(x_i)$, $y = (y_1, \ldots, y_n)^T$, and $g(y) = \sum_{j=1}^n g_j(y_j)$. Then, problem (1) can be rewritten as:

$$\begin{aligned} \text{minimize} \quad & f(x) + g(y) \\ \text{subject to} \quad & Ax = y \\ & x \in \mathcal{X},\ y \in \mathcal{Y}, \end{aligned} \tag{14}$$

where the matrix $A = [I, \ldots, I]$ (I is the $n \times n$ identity matrix).

Next, we introduce a set of auxiliary variables $z_i = x_i$, and reformulate problem (1) as:

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^N f_i(x_i) - g(\sum_{i=1}^N z_i) \\ \text{subject to} \quad & \forall i : x_i = z_i \\ & \forall i : x_i \in \mathcal{X}_i;\ \sum_{i=1}^N z_i \in \mathcal{Y}. \end{aligned} \tag{15}$$

Applying the scaled form of ADMM to problem (15), we obtain the following iterations:

$$\begin{aligned} x_i^{k+1} &:= \operatorname*{argmin}_{x_i \in \mathcal{X}_i} \left(-f_i(x_i) + (\rho/2)\|x_i - z_i^k + v_i^k\|_2^2\right) \\ z^{k+1} &:= \operatorname*{argmin}_{(\sum_i z_i) \in \mathcal{Y}} \left(g(\sum_{i=1}^N z_i) + (\rho/2)\sum_{i=1}^N \|z_i - x_i^{k+1} - v_i^k\|_2^2\right) \\ v_i^{k+1} &:= v_i^k + x_i^{k+1} - z_i^{k+1}. \end{aligned}$$

We will show that the above iterations are equivalent to Algorithm 3.3. The key observation is that the dual variables v_i^k are equal for all the users, i.e., $\forall i : v_i^k = v^k$.

Let $u^k \triangleq \sum_{i=1}^N v_i^k = Nv^k$ and $y^k \triangleq \sum_{i=1}^N z_i^k$. Then, the dual update can be rewritten as

$$u^{k+1} := u^k + \sum_i x_i^{k+1} - y^{k+1},$$

which is exactly the dual update in Algorithm 3.3.

Substituting $v^k = v^{k-1} + x_i^k - z_i^k$ and

$$v^k = v^{k-1} + (1/N)\left(\sum_i x_i^k - y^k\right)$$

in the x-update gives

$$x_i^{k+1} := \operatorname*{argmin}_{x_i \in \mathcal{X}_i} \left(-f_i(x_i) + (\rho/2)\|x_i - x_i^k + d^k\|_2^2\right),$$

which is exactly the x-update in Algorithm 3.3.

Finally, substituting

$$z_i^{k+1} - x_i^{k+1} - v_i^k = -v^{k+1} = (1/N)\left(y^{k+1} - \sum_i x_i^{k+1} - u^k\right)$$

in the z-update gives

$$y^{k+1} := \operatorname*{argmin}_{y \in \mathcal{Y}} g(y) + (\rho/2N)\|y - \sum_i x_i^{k+1} - u^k\|_2^2,$$

which is precisely the y-update in Algorithm 3.3. Hence, Algorithm 3.3 is indeed a variant of the standard ADMM algorithm.

Similarly, we can show that Algorithm 3.4 is equivalent to the following iterations:

$$\begin{aligned} z^{k+1} &:= \operatorname*{argmin}_{(\sum_i z_i) \in \mathcal{Y}} \left(g(\sum_{i=1}^N z_i) + (\rho/2)\sum_{i=1}^N \|z_i - x_i^k - v_i^k\|_2^2\right) \\ x_i^{k+1} &:= \operatorname*{argmin}_{x_i \in \mathcal{X}_i} \left(-f_i(x_i) + (\rho/2)\|x_i - z_i^{k+1} + v_i^k\|_2^2\right) \\ v_i^{k+1} &:= v_i^k + x_i^{k+1} - z_i^{k+1} \end{aligned}$$

which can be viewed as the scaled form of ADMM with the order of x-update and z-update switched.

3.2.4 Convergence analysis

We characterize the convergence rates of Algorithms 3.3 and 3.4 by making use of several very recent results [12, 21, 22] on ADMM. Note that these rate-of-convergence results are absent in [6, Chapter 7] (which only proves the convergence of Algorithms 3.3).

It turns out that both algorithms have an $\mathcal{O}(1/k)$ rate of convergence for the general case. Moreover, if the cost functions $g_j(\cdot)$ are strictly convex and their gradients $\nabla g_j(\cdot)$ are Lipschitz continuous, Algorithm 3.3 achieves linear convergence, i.e., convergence at rate $\mathcal{O}(1/a^k)$ for some $a > 1$. Similarly, if the utility functions $f_i(\cdot)$ are strictly concave and

TABLE 1
Convergence rates of Algorithms 3.3 and 3.4.

Case	Strictly convex	Lipschitz continuous	Recommended algorithms	Rate
1	none	none	Alg. 3.3 or 3.4	$\mathcal{O}(1/k)$
2	$\{g_j\}$	$\{\nabla g_j\}$	Alg. 3.3	$\mathcal{O}(1/a^k)$
3	$\{-f_i\}$	$\{\nabla f_i\}$	Alg. 3.4	$\mathcal{O}(1/a^k)$
4	$\{-f_i\}$, $\{g_j\}$	$\{\nabla f_i\}$, $\{\nabla g_j\}$	Alg. 3.3 or 3.4	$\mathcal{O}(1/a^k)$

their gradients $\nabla f_i(\cdot)$ are Lipschitz continuous, Algorithm 3.4 achieves linear convergence. Hence, Algorithms 3.3 and 3.4 naturally *complement* each other, as summarized in Table 1.

Let us formally state the rate-of-convergence results presented in Table 1. Without loss of generality, we mostly focus on Algorithm 3.3.

Let $(\{x_i^*\},\{z_i^*\})$ be a primal optimal solution to problem (15) (in particular, we have $x_i^* = z_i^*$), and $\{\lambda_i^*\}$ be a dual optimal solution. (The existence of $\{\lambda_i^*\}$ follows from the strong duality theorem.) Let $v_i^* = \lambda_i^*/\rho$. Then, Propositions 3.1 and 3.2 establish the rate-$\mathcal{O}(1/k)$ and rate-$\mathcal{O}(1/a^k)$ convergences for Algorithm 3.3, respectively.

Proposition 3.1. Let $\{\{x_i^k\}, y^k, u^k\}$ be any sequence generated by Algorithm 3.3. Let $v^k = u^k/N$ and $z_i^k = x_i^k + v^{k-1} - v^k$. Let

$$V^k = \sum_{i=1}^{N} \left(\|z_i^k - z_i^*\|_2^2 + \|v^k - v_i^*\|_2^2 \right), \tag{16}$$

and

$$D^k = \sum_{i=1}^{N} \left(\|z_i^{k+1} - z_i^k\|_2^2 + \|v^{k+1} - v^k\|_2^2 \right). \tag{17}$$

Then starting with any initial point $\{\{x_i^0\}, y^0, u^0\}$, D^k is non-increasing, and $D^k \le V^0/(k+1)$ for all k.

Remark 3.1. Proposition 3.1 has an important implication. It suggests that the sequence $\{D^k\}$ can be used as a natural stopping rule for Algorithm 3.3, which decreases at rate $1/k$. This stopping rule is more rigorous compared to that in [6, Chapter 7], since their stopping rule is based on heuristic principles. For example, their stopping-rule sequence does not have the non-increasing property and may fluctuate over iterations.

Proposition 3.2. Let $\{\{x_i^k\}, y^k, u^k\}$ be any sequence generated by Algorithm 3.3. Let V^k be the Lyapunov function defined in (16). Assume that the cost functions $g_j(\cdot)$ are strictly convex with Lipschitz continuous gradients. Then starting with any initial point $\{\{x_i^0\}, y^0, u^0\}$, there exists some $\delta > 0$ such that $V^k \le V^0/(1+\delta)^k$ for all k.

Remark 3.2. Proposition 3.2 provides a guideline for choosing the penalty parameter ρ. In particular, one can show that the parameter $\delta = \min\{c_9/\rho, c_{11}\rho\}$, where c_9 and c_{11} are given in [12]. Hence, ρ can be chosen such that the parameter δ is maximized.

The proofs of Propositions 3.1 and 3.2 are slight modifications of those presented in [12, 21, 22][2]. Note that both algorithms use a *single* parameter ρ, which is easier to tune than dual decomposition with varied step sizes. This is desirable for practical implementation.

2. We do not provide the proofs here, but could include them upon editor's request.

3.3 Parallel Implementation

Here, we discuss how the above two algorithms can be effectively implemented on parallel processors in a cloud environment. with a particular focus on Algorithm 3.3, since the same discussion applies to Algorithm 3.4.

We associate each user a processor, which stores and maintains two states (x_i^k, d^k). Similarly, we associate each facility a processor, which stores and maintains $(u_j^k, \sum_i x_{ij}^{k+1})$. At the k-th iteration, each user's processor solves a small-scale convex problem (in n variables), and then reports the updated x_{ij}^{k+1} to facility j. Each facility j collects these x_{ij}^{k+1} from all the users, and then computes the sum $\sum_i x_{ij}^{k+1}$. This is called a *reduce* step in parallel computing [11]. After the reduce step, each facility's processor solves a single-variable convex problem for y_j^{k+1} and updates u_j^{k+1}. Then, each facility's processor sends the value of $d_j^{k+1} \triangleq (1/N)\left(u_j^{k+1} + \sum_i x_{ij}^{k+1} - y_j^{k+1}\right)$ to all the users, which is called a *broadcast* step. Hence, each iteration consists of a reduce step and a broadcast step, performing message-passing between users and facilities.

An alternative and perhaps simpler method to implement Algorithm 3.3 is based on the MPI *Allreduce* operation [44], which computes the global sum over all processors and distributes the result to every processor. Although the Allreduce operation can be achieved by a reduce step followed by a broadcast step, an efficient implementation (for example, via butterfly mixing [51]) often leads to much better performance. With the help of Allreduce, we only need N processors, with each storing and maintaining three states $(x_i^k, u^k, \sum_i x_i^k)$. At the k-th iteration, each processor solves a small convex problem and updates x_i^{k+1}. Then, all the processors perform an Allreduce operation so that all of them (redundantly) obtain $\sum_i x_i^{k+1}$. After this Allreduce step, each processor solves n single-variable convex problems and (redundantly) computes u^{k+1}. Clearly, this method simplifies the implementation and can potentially increase the speed.

With respect to the communication overhead, it increases with N, the number of users. Note that the communication overhead per user only depends on the number of iterations, because each user just needs to send one message and receive one message from a facility during each iteration. Fortunately, the number of iterations is insensitive to the system size (as we will see in Sec. 4). Thus, the communication overhead per user is insensitive to the system size as well.

On the other hand, the communication overhead per facility is sensitive to the system size, because each facility needs to wait for the messages from all the users during each iteration. The "slow" users (often called the "stragglers" in the literature) would become the communication bottleneck in each iteration. To mitigate this issue, we propose the following strategy: each facility simply reuses the previous messages from those stragglers rather than waiting for their current messages. In this way, those stragglers will no longer be the bottleneck. This strategy will be formally introduced in our fault-tolerance model in Sec. 4.4. Surprisingly, our experimental results suggest that even if each facility gives up the slowest 10% of users at each iteration and treats them as stragglers, the convergence behaviour of our algorithms is still very close to the case where each facility waits for all the users. This means that 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/TPDS.2017.2658620, IEEE Transactions on Parallel and Distributed Systems

communication overhead per facility can be well controlled as the system size grows.

3.4 Comparisons with Other Algorithms

As we explained before, Algorithms 3.3 and 3.4 successfully overcome the shortcomings of dual decomposition and standard ADMM (i.e., Algorithm 3.2). Here, we will highlight their clear advantages over several other ADMM-based algorithms.

First, compared to our previous ADMM-based algorithms [47–49], Algorithms 3.3 and 3.4 enjoy a number of unique strengths. For example, the algorithm in [49] requires all cost functions $g_j(\cdot)$ to be linear, because otherwise Lemma 1 in [49] no longer holds and so each facility has to solve a large-scale convex optimization (where the number of variables equals to the number of users which can be in the order of 10^5 as explained before). In sharp contrast, Algorithms 3.3 and 3.4 only require the cost functions $g_j(\cdot)$ to be convex, which allows us to consider a much wider range of cloud applications, such as backbone traffic engineering. More importantly, although Algorithms 3.3 and 3.4 have much wider applications, their computational complexity is still lower than the algorithm in [49], because each facility here only needs to solve a *single-variable* convex optimization. In addition, Algorithm 3.3 achieves linear convergence when the cost functions $g_j(\cdot)$ are strictly convex, whereas no such rate-of-convergence results are available in [49]. To sum up, Algorithms 3.3 and 3.4 have wider applications, lower complexity and rate-of-convergence guarantees compared to the algorithm in [49].

Similarly, compared to our previous algorithms [47, 48] (which require strictly convex objective functions and bounded level sets), Algorithms 3.3 and 3.4 achieve faster convergence, even under weaker technical assumptions and with lower computational complexity and message-passing overhead.

There are some other ADMM-type distributed algorithms in the literature, such as linearized ADMM [21] and multi-block ADMM [20, 25]. However, they are not particularly suitable for the multi-facility resource allocation problem (1). For example, applying linearized ADMM to problem (1) gives the following iterations:

$$x_i^{k+1} := \operatorname*{argmin}_{x_i \in \mathcal{X}_i} \left(-f_i(x_i) + x_i^T g^k + (r/2)\|x_i - x_i^k\|_2^2 \right)$$

$$y_j^{k+1} := \operatorname*{argmin}_{y_j \in \mathcal{Y}_j} \left(g_j(y_j) + (\rho/2)(y_j - \sum_{i=1}^{N} x_{ij}^{k+1} - u_j^k)^2 \right)$$

$$u_j^{k+1} := u_j^k + \sum_{i=1}^{N} x_{ij}^{k+1} - y_j^{k+1},$$

where $g^k = \rho(\sum_i x_i^k - y^k + u^k)$ *linearizes* the penalty term $(\rho/2)\|\sum_i x_i - y\|_2^2$, and $(r/2)\|x_i - x_i^k\|_2^2$ is a *proximal term*. Although the above algorithm preserves separability of the problem, its convergence requires $r > \rho N$. When N is sufficiently large, the x-update in each iteration just slightly changes x_i (due to a large r), making the convergence slow. Hence, linearized ADMM is not well suited for large-scale problems.

Multi-block ADMM is another candidate for solving problem (1). However, it generally requires users to solve their subproblems sequentially rather than in parallel. Moreover, it still lacks theoretical convergence guarantees for the general case. Indeed, a counter-example has recently been reported showing the impossibility of convergence of multi-block ADMM for the general case [8, 31].

The algorithms presented in [46] are most similar to ours. Their basic idea is also to apply variants of the standard ADMM algorithm to solve separable convex problems. However, their algorithms require the utility functions to be strictly concave *and* the cost functions to be strictly convex in order to achieve $\mathcal{O}(1/a^k)$ rate of convergence. Such requirements cannot be met in some application scenarios. One such example is backbone traffic engineering, as we will discuss in Sec. 2.

4 Empirical Study

We present our empirical study of the performance of the distributed ADMM algorithms. For this purpose, it suffices to choose one of the two cloud traffic management problems since they are equivalent in nature. We use the geographical load balancing problem (2) with the utility and cost functions (5) and (6) as the concrete context of the performance evaluation. This problem corresponds to the most general case (i.e., case 1 in Table 1), since (5) is non-strictly concave and (6) is non-strictly convex. Thus it can be solved using either Algorithm 3.3 or Algorithm 3.4. We use Algorithm 3.3 in all of our simulations. Note that if the objective function exhibits strict convexity, better simulation results can be obtained according to Proposition 3.2. In other words, we mainly focus on the "worse-case" performance of the algorithms in this section. We plan to make all our simulation codes publicly available after the review cycle.

We implement Algorithm 3.3 in Matlab in a sequential manner to evaluate its convergence behaviour. Note that the convergence time is equal to the product of the number of iterations and the time spent on each iteration, where the number of iterations is independent of specific platforms used for implementation, and the time for each iteration depends heavily on specific implementation platforms. As such, we mostly focus on the platform-independent metric: the number of iterations. It turns out that this metric is *insensitive* to the scale of the system, as we will soon see.

4.1 Setup

We randomly generate each user's request demand t_i, with an average of 9×10^4. We then normalize the workloads to the number of servers, assuming each request requires 10% of a server's CPU. We assume the prediction of request demand is done accurately since prediction error is immaterial to performance of the optimization algorithms. The latency l_{ij} between an arbitrary pair of user and data center is randomly generated between 50 ms and 100 ms.

We set the number of data centers (facilities) $n = 10$. Each data center's capacity c_j is randomly generated so that the total capacity $\sum_j c_j$ is 1.4x the total demand. We use the 2011 annual average day-ahead on peak prices [15] at 10 different local markets as the power prices P_j for data centers. The servers have peak power $P_{\text{peak}} = 200$ W, and consume 50% power at idle. The PUE is 1.5. These numbers represent state-of-the-art data center hardware [14, 43].

We set the penalty parameter ρ of the ADMM algorithm to $\rho = 10^{-3}$ after an empirical sweep of $\rho \in \{10^{-4}, 10^{-3}, \ldots, 10^3, 10^4\}$. Although a more fine-grained search for ρ can further improve the performance of our algorithms, we confine ourselves to the above 9 choices to demonstrate the practicality.

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/TPDS.2017.2658620, IEEE Transactions on Parallel and Distributed Systems

4.2 Failure of Centralized Solvers

First of all, to motivate the need of distributed algorithms, we run experiments with state-of-the-art centralized solvers under the previous setup, and show that they cannot be readily used to solve traffic management problems in a large scale. We use the `quadprog` solver in Matlab 2014b with the `active-set` algorithm. The solver runs on a linux server with an 8-core Intel Xeon E5-2640v2 2.0GHz CPU and 32GB memory. We set `maxIter`, the maximum number of iterations, to 2000 because the default value of 500 is too small for our problems.

We find that, when number of users N is 10^2, the centralized solver takes ~226.5s to terminate; when $N = 10^3$, the centralized solver runs for more than 16 hours without solving the problem; when $N = 10^4$, Matlab reports out of memory due to the huge number of variables and constraints. This clearly shows that it is practically infeasible to resort to a centralized solver to handle large-scale problems. Even for giant companies like Google with a computer say 100 times faster than our server, it takes at least 9.6 minutes to solve the problem with $N = 10^4$, which is still too slow for practical use. In reality N can be even larger which makes things even more difficult. Thus we believe it is necessary to adopt distributed solution algorithms.

4.3 Convergence and Scalability

We now evaluate the convergence of Algorithm 1 under the previous setup. We vary the problem size by changing the number of users $N \in \{10^2, 10^3, 10^4, 10^5\}$ and scaling data center capacities linearly with N. We observe that our algorithm converges quickly after 50 iterations in all cases, *independent* of the problem size.

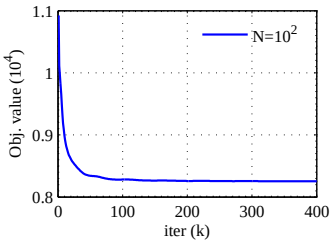

Fig. 3. Objective value. $N = 10^2$.

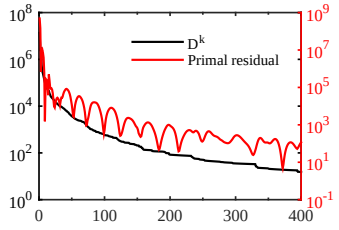

Fig. 4. D^k and primal residual. $N = 10^2$.

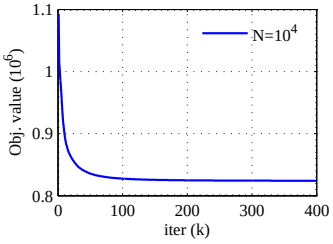

Fig. 5. Objective value. $N = 10^4$.

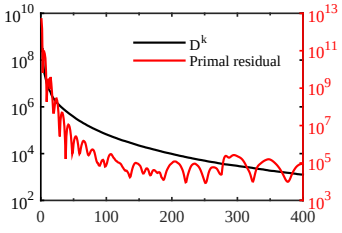

Fig. 6. D^k and primal residual. $N = 10^4$.

Convergence of objective functions. Figure 3 and 5 plot the convergence of objective values for $N = 10^2$ and $N = 10^4$, respectively. Notice that the objective values for $N = 10^4$ are roughly 100 times the corresponding values for $N = 10^2$ at each iteration. This means that our algorithm has excellent scalability, which is very helpful in practice. Since the number of iterations is independent of the problem size, it suggests that our algorithm can solve a large-scale problem with (almost) the same running time by simply scaling the amount of computing resources linearly with the number of users.

Convergence of D^k. Figure 4 and 6 show the trajectory of D^k as defined in (17) for $N = 10^2$ and $N = 10^4$, respectively. We observe that D^k is indeed non-increasing in both cases. Further, the two figures are in log scale, implying that D^k decreases sublinearly, which confirms Proposition 3.1 for the $\mathcal{O}(1/k)$ convergence rate. In addition, one can see that D^k scales linearly with N as expected from its definition. This implies that D^k is an ideal candidate for the stopping rule: the algorithm can be terminated when D^k/N is below a certain threshold.

Convergence of primal residuals. Figure 4 and 6 show the trajectory of the primal residual, which is defined as $\sum_i^N \|x_i - z_i\|_2^2$ here. It reflects how well the constraints $\{x_i = z_i\}$ are satisfied, and is sometimes called the primal feasibility gap. For example, if the primal residual is 10^4 for $N = 10^2$ (or, 10^6 for $N = 10^4$), then on average each $\|x_i - z_i\|$ is around 10, which is already small enough since x_i is in the order of 10^4. Hence, we conclude that the constraints are well satisfied after 50 iterations in both cases.

4.4 Fault-tolerance

We have observed that our algorithms converge fast to the optimal solution for large-scale problems. Yet, because failures are the norm rather than the exception, fault-tolerance is arguably the most important design objective for parallel computing frameworks that involve a large number of servers currently [11]. A parallel algorithm that is inherently robust against failures in the intermediate steps is highly desirable for practical deployment. To investigate the fault-tolerance of our algorithm, we carry out a new set of simulations where each user fails to update x_i^k with a probability p at each iteration (independent of each other). Whenever a failure happens, user i simply reuses its previous solution by setting $x_i^{k+1} := x_i^k$.

This failure model can be used to capture several practical scenarios. For example, if user i experiences a temporary failure at iteration $k+1$, it can simply report its previous value x_i^k to all the facilities. In addition, under this model, a facility doesn't have to wait until it collects the updates from all the users. Instead, it can simply reuse x_i^k for some "slow" users (perhaps due to temporary network congestion) and treat those users as failures. In this way, those slow users are no longer the bottleneck.

Figure 7–9 plot the convergence with different failure probabilities for $N = 10^2$, and Figure 10–12 for $N = 10^4$. Specifically, Figure 7 and 10 plot the relative error in objective value with failures (i.e. OBJ_FAIL/OBJ $-$ 1, where OBJ_FAIL is the objective value with failures, and OBJ is the objective value when every step is solved correctly). We observe that increasing the failure probability from 5% to 10% increases the relative error, causing the solution quality to degrade at the early stage. Yet surprisingly, the impact is very insignificant: The relative error is at most 1.5%, and ceases to 0 after 100 iterations. In fact, after 50 iterations the relative error is only around 0.2% for both problem sizes.

Moreover, failures do not affect the convergence of the algorithm at all. This is indicated by the relative error plots,

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/TPDS.2017.2658620, IEEE Transactions on Parallel and Distributed Systems

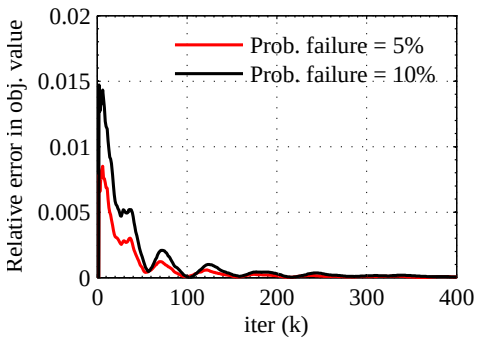

Fig. 7. Relative errors in objective value. $N = 10^2$.

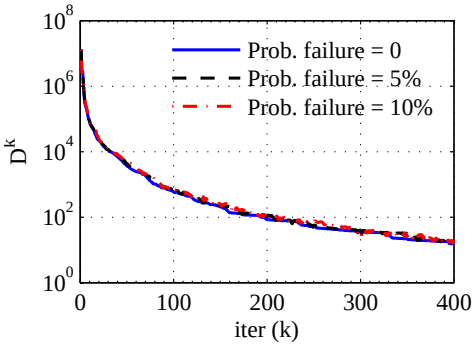

Fig. 8. D^k. $N = 10^2$.

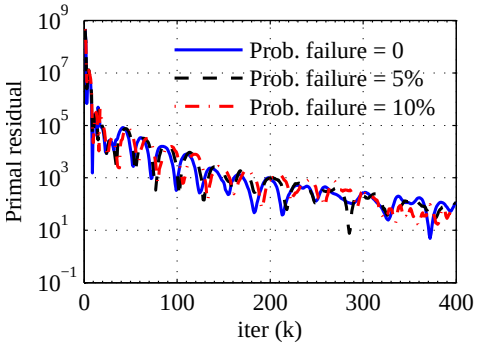

Fig. 9. Primal residual. $N = 10^2$.

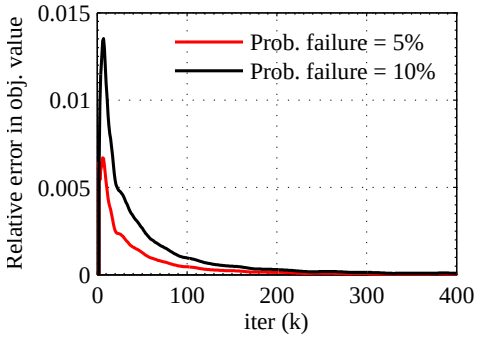

Fig. 10. Relative errors in objective value. $N = 10^4$.

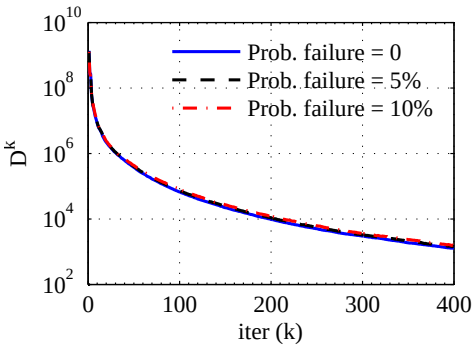

Fig. 11. D^k. $N = 10^4$.

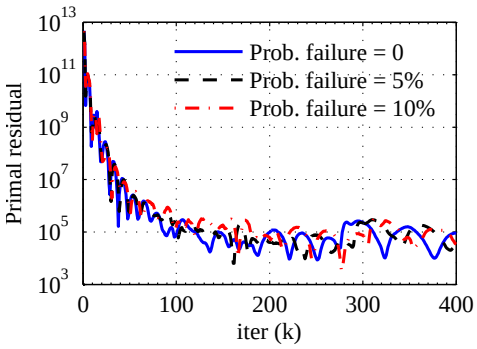

Fig. 12. Primal residual. $N = 10^4$.

and further illustrated by the overlapping curves in Figure 8, 9, 11, and 12 for D^k and primal residual.

Thus, we find that our distributed ADMM algorithms are inherently fault-tolerant, with less than 1% optimality loss and essentially the same convergence speed for up to 10% failure rate. They are robust enough to handle temporary failures that commonly occur in production systems.

4.5 Comparison with Dual Decomposition

We also simulate the conventional dual decomposition approach with subgradient methods as explained in Sec. 3.4 to solve problem (2). The step size ρ^k is chosen following the commonly accepted diminishing step size rule [5], with $\rho^k = 10^{-5}/\sqrt{k}$.

We plot the trajectory of objective values in Figure 13, and that of primal residuals in Figure 14. Compare to Algorithm 1, dual decomposition yields wildly fluctuating results. Though the objective value decreases to the same level as Algorithm 1 after about 200 iterations, the more meaningful primal variables $\{x_i\}$ never converge even after 400 iterations. One can see from Figure 14 that the primal residual does not decrease below 10^7. This implies that the equality constraints $\{x_i = z_i\}$ are not well-satisfied during the entire course, and the primal variables $\{x_i\}$ still violate the capacity constraints after 400 iterations.

This phenomenon is due to the *oscillation problem* [32] when dual decomposition method is applied to non-strictly convex objective functions. To mitigate this problem, one can make the objective function strictly convex by adding a small penalty term, e.g., $\rho_1\|x\|_2^2 + \rho_2\|z\|_2^2$). Nevertheless, we found that the primal variables $\{x_i\}$ still converge very slowly after an extensive trial of different (ρ_1, ρ_2).

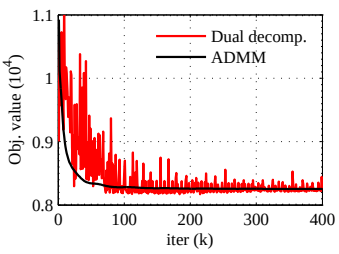

Fig. 13. Objective value. $N = 10^2$.

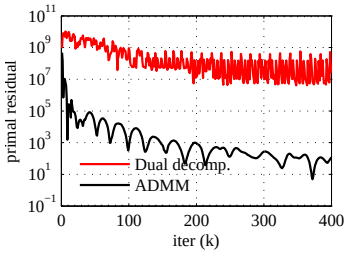

Fig. 14. Primal residual. $N = 10^2$.

To summarize, our simulation results confirm our theoretical analysis, demonstrate fast convergence of our algorithms in various settings, and highlight several additional advantages, especially the scalability and fault-tolerance.

5 Related Work

5.1 Network Utility Maximization

Network utility maximization (NUM) [3, 45] is closely related to our multi-facility resource allocation problem. A standard technique for solving NUM problems is dual decomposition. Dual decomposition was first applied to the NUM problem in [28], and has lead to a rich literature on distributed algorithms for network rate control [9, 36, 41] and new understandings of existing network protocols [35]. Despite its popularity, dual decomposition requires a delicate adjustment of the step-size parameters, which are often difficult to tune. In addition, dual decomposition requires the utility functions to be strictly concave and the cost functions to be strictly convex. Our ADMM-type algorithms overcome these difficulties, achieving faster

convergence under weaker assumptions as discussed in Sec. 3.4 in detail.

5.2 Cloud Traffic Management

Cloud service providers operate two distinct types of WANs: user-facing WANs and backbone WANs [27]. The user-facing WAN connects cloud users and data centers by peering and exchanging traffic with ISPs. Through optimized load balancing, this type of networks can achieve a desired trade-off between performance and cost [17, 33, 34, 43, 47–49]. The backbone WAN provides connectivity among data centers for data replication and synchronization. Rate control and multi-path routing [18, 24, 27] can significantly increase link utilization and reduce operational costs of the network. Previous work developed different optimization methods for each application scenario separately, whereas our work provides a unified framework well suited to a wide range of network scenarios. More importantly, our approach achieves faster convergence than prior art, even under weaker assumptions and with lower computational complexity and message passing overhead, as discussed thoroughly in Section 3.4.

6 Conclusion

In this work, we have introduced a general framework for studying various cloud traffic management problems. We have abstracted these problems as a multi-facility resource allocation problem and presented two distributed algorithms based on ADMM that are amenable to parallel implementation. We have provided the convergence rates of our algorithms under various scenarios. When the utility functions are non-strictly concave and the cost functions are non-strictly convex, our algorithms achieve $\mathcal{O}(1/k)$ rate of convergence. When the utility functions are strictly concave *or* the cost functions are strictly convex, our algorithms achieve $\mathcal{O}(1/a^k)$ rate of convergence.

We have shown that, compared to dual decomposition and other ADMM-type distributed solutions, our algorithms have a number of unique advantages, such as achieving faster convergence under weaker assumptions, and enjoying lower computational complexity and lower message-passing overhead. These advantages are further confirmed by our extensive empirical studies. Moreover, our simulation results demonstrate some additional advantages of our algorithms, including the scalability and fault-tolerance, which we believe are highly desirable for large-scale cloud systems.

References

[1] S. Agarwal, M. Kodialam, and T. V. Lakshman. Traffic engineering in software defined networks. In *Proc. IEEE INFOCOM*, 2013.

[2] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proc. ACM SIGCOMM*, 2011.

[3] D. P. Bertsekas. *Network Optimization: Continuous and Discrete Models*. Athena Scientific, 1998.

[4] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, 1997.

[5] S. Boyd and A. Mutapcic. Subgradient methods. Lecture notes of EE364b, Stanford University, Winter Quarter 2006-2007. http://www.stanford.edu/class/ee364b/notes/subgrad_method_notes.pdf.

[6] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, 2010.

[7] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a routing control platform. In *Proc. USENIX NSDI*, 2005.

[8] C. Chen, B. He, Y. Ye, and X. Yuan. The direct extension of ADMM for multi-block convex minimization problems is not necessarily convergent. *Mathematical Programming*, 155(1):57–59, January 2016.

[9] M. Chiang, S. H. Low, A. R. Calderbank, and J. C. Doyle. Layering as optimization decomposition: A mathematical theory of network architectures. *Proc. IEEE*, 95(1):255–312, January 2007.

[10] E. Danna, S. Mandal, and A. Singh. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering. In *Proc. IEEE INFOCOM*, 2012.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI*, 2004.

[12] W. Deng and W. Yin. On the global and linear convergence of the generalized alternating direction method of multipliers. Technical report, Department of Computational and Applied Mathematics, Rice University, 2012.

[13] A. Elwalid, C. Jin, S. H. Low, and I. Widjaja. Mate: Mpls adaptive traffic engineering. In *Proc. IEEE INFOCOM*, 2001.

[14] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *Proc. ISCA*, 2007.

[15] Federal Energy Regulatory Commission. U.S. electric power markets. http://www.ferc.gov/market-oversight/mkt-electric/overview.asp, 2011.

[16] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and S. Diot. Packet-level traffic measurements from the Sprint IP backbone. *IEEE Netw.*, 17(6):6–16, November 2003.

[17] P. X. Gao, A. R. Curtis, B. Wong, and S. Keshav. It's not easy being green. In *Proc. ACM SIGCOMM*, 2012.

[18] A. Ghosh, S. Ha, E. Crabbe, and J. Rexford. Scalable multi-class traffic management in data center backbone networks. *IEEE J. Sel. Areas Commun.*, 31(12):1–12, December 2013.

[19] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4D approach to network control and management. *ACM SIGCOMM Comput. Commun. Rev.*, 35(5):41–54, October 2005.

[20] D. Han and X. Yuan. A note on the alternating direction method of multipliers. *J. Optim. Theory Appl.*, 155:227–238, 2012.

[21] B. He and X. Yuan. On non-ergodic convergence rate of Douglas-Rachford alternating direction method of multipliers. Technical report, 2012.

[22] B. He and X. Yuan. On the $o(1/n)$ convergence rate of the Douglas-Rachford alternating direction method. *SIAM J. Num. analysis*, 50:700–709, 2012.

[23] M. R. Hestenes. Multiplier and gradient methods. *Journal of Optimization Theory and Applications*, 4(5):303–320, 1969.

[24] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *Proc. ACM SIG-*

COMM, 2013.

[25] M. Hong and Z.-Q. Luo. On the linear convergence of the alternating direction method of multipliers. http://arxiv.org/abs/1208.3922, August 2012.

[26] M. Hong, Z.-Q. Luo, and M. Razavlyayn. Convergence analysis of alternating direction method of multiplies for a family of nonconvex problems. *SIAM J. Optim.*, 26(1):337–364, 2016.

[27] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined WAN. In *Proc. ACM SIGCOMM*, 2013.

[28] F. P. Kelly, A. K. Maulloo, and D. K. H. Tan. Rate control for communication networks: Shadow prices, proportional fairness and stability. *J. Operat. Res. Soc.*, 49(3):237–252, March 1998.

[29] R. Kohavi, R. M. Henne, and D. Sommerfield. Practical guide to controlled experiments on the web: Listen to your customers not to the hippo. In *Proc. ACM SIGKDD*, 2007.

[30] W.-C. Liao, M. Hong, H. Farmanbar, X. Li, Z.-Q. Luo, and H. Zhang. Min flow rate maximization for software defined radio access networks. *IEEE J. Sel. Areas Commun.*, 32(6):1282–1294, June 2014.

[31] T. Lin, S. Ma, and S. Zhang. On the global linear convergence of the admm with multiblock variables. *SIAM J. Optim.*, 25(3):1478–1497, 2015.

[32] X. Lin and N. B. Shroff. Utility maximization for communication networks with multi-path routing. *IEEE Trans. Autom. Control*, 51(5):766–781, May 2006.

[33] Z. Liu, Y. Chen, C. Bash, A. Wierman, D. Gmach, Z. Wang, M. Marwah, and C. Hyser. Renewable and cooling aware workload management for sustainable data centers. In *Proc. ACM Sigmetrics*, 2012.

[34] Z. Liu, M. Lin, A. Wierman, S. H. Low, and L. L. Andrew. Greening geographical load balancing. In *Proc. ACM Sigmetrics*, 2011.

[35] S. H. Low. A duality model of TCP and queue management algorithms. *IEEE/ACM Trans. Netw.*, 11(4):525–536, August 2003.

[36] S. H. Low and D. E. Lapsley. Optimization flow control—I: Basic algorithm and convergence. *IEEE/ACM Trans. Netw.*, 7(6):861–874, December 1999.

[37] J. Mo and J. Walrand. Fair end-to-end window-based congestion control. *IEEE/ACM Trans. Netw.*, 8(5):556–567, October 2000.

[38] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proc. ACM SIGCOMM*, 2016.

[39] S. Narayana, J. W. Jiang, J. Rexford, and M. Chiang. Distributed wide-area traffic management for cloud services. In *Proc. ACM Sigmetrics*, Extended Abstract, 2012.

[40] E. Nygren, R. K. Sitaraman, and J. Sun. The Akamai network: A platform for high-performance Internet applications. *SIGOPS Oper. Syst. Rev.*, 44(3):2–19, August 2010.

[41] D. Palomar and M. Chiang. A tutorial on decomposition methods and distributed network resource allocation. *IEEE J. Sel. Areas Commun.*, 24(8):1439–1451, August 2006.

[42] J. Perry, H. Balakrishnan, and D. Shah. Flowtune: Flowlet control for datacenter networks. In *Proc. USENIX NSDI*, 2017.

[43] A. Qureshi, R. Weber, H. Balakrishnan, J. Guttag, and B. Maggs. Cutting the electricity bill for Internet-scale systems. In *Proc. SIGCOMM*, 2009.

[44] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

[45] R. Srikant. *The Mathematics of Internet Congestion Control*. Birkhäuser, 2004.

[46] X. Wang, M. Hong, S. Ma, and Z.-Q. Luo. Solving multiple-block separable convex minimization problems using two-block alternating direction method of multipliers. Technical report, August 2013.

[47] H. Xu, C. Feng, and B. Li. Temperature aware workload management in geo-distributed datacenters. In *Proc. USENIX ICAC*, 2013.

[48] H. Xu, C. Feng, and B. Li. Temperature aware workload management in geo-distributed datacenters. *IEEE Trans. Parallel Distrib. Syst.*, 26(6):1743–1753, June 2015.

[49] H. Xu and B. Li. Joint request mapping and response routing for geo-distributed cloud services. In *Proc. IEEE INFOCOM*, 2013.

[50] Z. Zhang, M. Zhang, A. Greenberg, Y. C. Hu, R. Mahajan, and B. Christian. Optimizing cost and performance in online service provider networks. In *Proc. USENIX NSDI*, 2010.

[51] H. Zhao and J. Canny. Butterfly mixing: Accelerating incremental-update algorithms on clusters. In *Proc. SIAM Int. Conf. Data Mining*, 2013.

[52] Z. Zhou, F. Liu, B. Li, B. Li, H. Jin, R. Zou, and Z. Liu. Cell generation in geo-distributed cloud services: A quantitative study. In *Proc. IEEE ICDCS*, 2014.