An Alternating Direction Method Approach to Cloud Traffic Management

Chen Feng  
Department of Electrical and Computer Engineering  
University of Toronto  
Toronto, ON, Canada  
cfeng@eecg.toronto.edu

Hong Xu  
Department of Computer Science  
City University of Hong Kong  
Kowloon, Hong Kong  
henry.xu@cityu.edu.hk

Baochun Li  
Department of Electrical and Computer Engineering  
University of Toronto  
Toronto, ON, Canada  
bli@eecg.toronto.edu

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 abstract these real-world problems as a multi-facility resource allocation problem, and develop two distributed optimization algorithms that are amenable to parallel implementation. Our algorithms not only overcome the major difficulties of the standard dual-decomposition method, but also enjoy low computational complexity and low message-passing overhead. We prove the rate of convergence of our algorithms by utilizing several very recent results on alternating direction method of multipliers. As a by-product of our analysis, we consider applications to networking research is still in an early stage. To the best of our knowledge, the work [41–43] represents one of the first such applications. Compared to these previous algorithms, the algorithms proposed in this paper require much weaker technical assumptions to ensure convergence, and at the same time, enjoy much lower computational complexity and message-passing overhead.

We prove the rate of convergence of our algorithms by utilizing several very recent results on ADMM. Although the convergence of ADMM is well known in the literature (see, e.g., [5, 7]), its rate of

Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

1. INTRODUCTION
In this paper, we consider problems of the following form:

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

As we will describe in Sec. 2, optimization problems of this form appear in many cloud traffic management scenarios, including geographically load balancing and backbone traffic engineering. Generically, the problem (1) amounts to allocating resources from n facilities to N users such that the “social welfare” (i.e., utility minus cost) is maximized. 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 only cares about the total amount of allocated resources. We assume that \(g_j(\cdot)\) are convex. The constraint sets \(\{\mathcal{X}\}\) and \(\{\mathcal{Y}\}\) are used to model the additional constraints, which are assumed to be convex sets.

We refer to problem (1) as the multi-facility resource allocation problem. In this paper, we are interested in solutions that are amenable to parallel implementations, since a cloud provider usually has abundant servers for parallel computing. As we will soon see, (1) is inherently a large-scale convex optimization problem, with millions of variables, or even more, for a production cloud. 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 usually requires delicate adjustments of step sizes, leading to slow convergence especially for large-scale problems. Second, dual decomposition generally requires the utility functions \(f_i(\cdot)\) to be strictly concave and the cost functions \(g_j(\cdot)\) to be strictly convex. However, these requirements cannot be met in many problem settings of (1), as demonstrated in Sec. 2.

To overcome these difficulties, we develop new distributed algorithms for the multi-facility resource allocation problem. Our solutions achieve faster convergence under weaker technical assumptions. In particular, our algorithms achieve \(O(1/k)\) rate of convergence for general utility and cost functions (where \(k\) is the number of iterations), and achieve \(O(1/a^k)\) (for some \(a > 1\)) rate of convergence when either the utility functions are strictly concave or the cost functions are strictly convex. More importantly, compared to dual decomposition, our solutions enjoy lower computational complexity and lower message-passing overhead.

Our distributed algorithms are 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 [7]. Although ADMM has been widely applied to areas of statistics, machine learning, and signal processing, its application to networking research is still in an early stage. To the best of our knowledge, the work [41–43] represents one of the first such applications. Compared to these previous algorithms, the algorithms proposed in this paper require much weaker technical assumptions to ensure convergence, and at the same time, enjoy much lower computational complexity and message-passing overhead.

We prove the rate of convergence of our algorithms by utilizing several very recent results on ADMM. Although the convergence of ADMM is well known in the literature (see, e.g., [5, 7]), its rate of
convergence has only been established very recently [12, 21]. These
new results provide a solid theoretical foundation for our analysis.
Based on these results, we are not only able to establish the con-
vergence rates of our algorithms, but also able to give a simple yet
rigorous stopping rule compared to the conventional stopping rule
proposed in [7].

Finally, we present an extensive empirical study on our algo-
rithms. Our simulation results not only confirm our theoretical
analysis, but also highlight some other important advantages of our
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 several cloud traffic management problems as
   instances of the multi-facility resource allocation problem.
2. We develop new distributed algorithms for the multi-facility
   resource allocation problem, which have a number of unique
   advantages compared to dual decomposition and previous al-
   gorithms.
3. We prove convergence rates for our algorithms by using sev-
   eral very recent results. We also provide a simple yet rigor-
   ous stopping rule for our algorithms.
4. We present extensive simulation results, which not only con-
   firm our analysis, but also demonstrate the scalability and
   fault-tolerance of our algorithms.

2. APPLICATIONS TO CLOUD TRAFFIC
MANAGEMENT

Before developing distributed algorithms to the multi-facility re-
source allocation problem, we first give a few examples from the
recent literature in the context of cloud traffic management, where
optimization problems of the form (1) naturally appear. We also
illustrate the large scale of these problems for a production system,
which motivates our quest for efficient distributed algorithms.

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 di-
rect 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 geo-
graphic diversity as seminally pointed out by [37]. This question
has attracted much attention recently [17, 29, 30, 37, 41–43], and is
generally referred to as geographical load balancing.

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, 30, 37, 41–43]. 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 [35]. 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 that
can be fairly easily predicted using machine learning. 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.

\[
\begin{align*}
\text{maximize} & \quad \sum_i f_i(x_i) - \sum_j g_j(y_j) \\
\text{subject to} & \quad \forall i : \sum_j x_{ij} = t_i, x_{ij} \in \mathbb{R}_+^n, \\
& \quad \forall j : y_j = \sum_i x_{ij} \leq c_j,
\end{align*}
\]

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 ar-
guably 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 [27]. The user-perceived latency
largely depends on the end-to-end propagation latency [16, 34],
which can be obtained through active measurements. Let \( l_{ij} \) de-
note the end-to-end propagation latency between user \( i \) and data
center \( j \). The following utility function \( f_i \) has been used in [41, 42]

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

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, 30, 37, 43]

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

Here, \( P_j^{\text{peak}} \) denotes the energy price in terms of $/KWh at data cen-
ter \( j \). 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. Fi-
nally, \( E(y_j) \) 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]:

\[
E(y_j) = c_j P_{\text{idle}} + (P_{\text{peak}} - P_{\text{idle}}) y_j, \tag{7}
\]

\[
\begin{array}{c}
\text{Clients} \\
\hline
\text{Requests} \\
\hline
\text{Datacenters}
\end{array}
\]

Figure 1: A cloud service running on geographically dis-
tributed data centers.

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

\[
\begin{align*}
\text{maximize} & \quad \sum_i f_i(x_i) - \sum_j g_j(y_j) \\
\text{subject to} & \quad \forall i : \sum_j x_{ij} = t_i, x_{ij} \in \mathbb{R}_+^n, \\
& \quad \forall j : y_j = \sum_i x_{ij} \leq c_j,
\end{align*}
\]
where \( P_{\text{idle}} \) is server idle power and \( P_{\text{peak}} \) 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 \( O(10^5) \) [35]. Thus the number of variables \( \{x_{ij}\} \) is \( O(10^6) \). 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 distribution 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, 30].

**Joint Optimization with Batch Workloads.** There are also efforts [29,41,42] 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 \( f_j \left( w_j \right) \) be the utility of these batch workloads. Then the joint optimization can be formulated as follows:

\[
\begin{align*}
\text{maximize} & \quad \sum_i f_i(x_i) + \sum_j f_j(w_j) - \sum_j g_j(y_j) \\
\text{subject to} & \quad \forall i: \sum_j x_{ij} = t_i, \quad x_i \in \mathbb{R}^n; \quad w \in \mathbb{R}^n \\
& \quad \forall j: \quad y_j = \sum_i x_{ij} + w_j \leq c_j.
\end{align*}
\]

The utility function \( f_j(\cdot) \) depends only on \( w_j \) but not on latency, due to its elastic nature. In general, \( f_j(\cdot) \) is an increasing and concave function, such as the log function used in [41,42]. 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 [18,25]. 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 [8,19]. 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 [18,23,25].

#### 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 [13,23,25]) 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:

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

**Figure 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: \{link 1, link 2\} and \{link 2, 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 [23,25] are designed to control the aggregated sending rates of applications across data centers. The aggre-
gated 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 \( y_j = \sum x_{ij} \) is the total traffic on link \( j \). The joint rate control and traffic engineering problem can be formulated as

\[
\text{maximize} \quad \sum_i f_i(A_i^{-1} x_i) - \sum_j g_j(y_j) \\
\text{subject to} \quad \forall i : x_i \in \mathbb{R}_{+}^n, \\
\forall j : y_j = \sum_i x_{ij} \leq c_j,
\]

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 \( f_i(w_i) = \log(\sum p w_{ji}) \), or a more general “rate-fairness” function used for Internet TCP congestion control [33]. 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 will be used in Sec. 3.4. 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. In other words, the topology matrices are easy to store and maintain in practice.

### 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 [23, 25]. Thus the number of application flows is \( O(10^5) \) to \( O(10^6) \). 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 [23,25]) 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 [44]. 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 modeled 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.)

### 3. DISTRIBUTED ALGORITHMS

In this section, we first review some basics of ADMM. We then apply ADMM to design our distributed algorithms.

#### 3.1 Standard ADMM Algorithm

The standard ADMM algorithm solves convex optimization problems in the form

\[
\begin{align*}
\text{minimize} & \quad f(x) + g(y) \\
\text{subject to} & \quad Ax + By = c, \\
& \quad x \in \mathcal{X}, y \in \mathcal{Y},
\end{align*}
\]

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 closed convex subsets of \( \mathbb{R}^n \) and \( \mathbb{R}^m \), respectively. The objective function in (11) is separable over two sets of variables \( x \) and \( y \), which are coupled through a linear equality constraint \( Ax + By = c \).

The augmented Lagrangian [22] for problem (11) is

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

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 (11), and \( L_\rho \) is the sum of \( L_0 \) and a penalty term \( (\rho/2)\|Ax + By - c\|^2 \). Introducing the penalty term leads to improved numerical stability and faster convergence [7].

The standard ADMM algorithm solves problem (11) with the iterations:

\[
\begin{align*}
\underline{x}^{k+1} & := \arg\min_{x \in \mathcal{X}} L_\rho(x, y^k, \lambda^k), \\
\underline{y}^{k+1} & := \arg\min_{y \in \mathcal{Y}} L_\rho(x^{k+1}, y, \lambda^k), \\
\lambda^{k+1} & := \lambda^k + \rho(Ax^{k+1} + By^{k+1} - c),
\end{align*}
\]

where the penalty parameter \( \rho \) is the step size for the update of the dual variable \( \lambda \). Note that the primal variables \( x \) and \( y \) are updated in an alternating fashion, which accounts for the term alternating direction.

The standard ADMM algorithm takes advantage of the separable structure of problem (11) and decomposes (11) over primal variables \( x \) and \( y \). This is particularly useful in applications where the \( x \)-update and \( y \)-update admit simple solutions or can be implemented in a distributed manner.

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

\[
\begin{align*}
\underline{x}^{k+1} & := \arg\min_{x \in \mathcal{X}} \left( f(x) + (\rho/2)\|Ax + By - c + u^k\|^2 \right), \\
\underline{y}^{k+1} & := \arg\min_{y \in \mathcal{Y}} \left( g(y) + (\rho/2)\|Ax^{k+1} + By - c + u^k\|^2 \right), \\
\underline{u}^{k+1} & := u^k + Ax^{k+1} + By^{k+1} - c.
\end{align*}
\]

The optimality and convergence of ADMM can be guaranteed under very mild technical assumptions [5,7]. In practice, it is often...
Then problem (1) can be rewritten as:

\[
\min_{x} f(x) + g(y)
\]
subject to \(Ax = y\)
\(x \in \mathcal{X}, \quad y \in \mathcal{Y},\)

where the matrix \(A = [I, \ldots, I] \) (\(I\) is the \(n \times n\) identity matrix). Clearly, problem (12) is in ADMM form.

Now, the new penalty term is \(A\).

For the simplicity of notations, we let \(x = (x_1^T, \ldots, x_N^T)^T\), \(f(x) = -\sum_{i=1}^N f_i(x_i), y = (y_1, \ldots, y_N)^T\), and \(g(y) = \sum_{j=1}^M g_j(y_j)\). Then problem (1) can be rewritten as:

\[
\begin{align*}
\min_{x} & \quad f(x) + g(y) \\
\text{subject to} & \quad Ax = y \\
& \quad x \in \mathcal{X}, \quad y \in \mathcal{Y},
\end{align*}
\]

for \(\bar{z} \in \mathbb{R}^n\) and then applying (14). Substituting (14) for \(z_i^{k+1}\) in the u-update gives

\[
u_i^{k+1} := \bar{u}_i^k + x_i^{k+1} - z_i^{k+1},
\]

which shows that the dual variables \(u_i^k\) are equal for all the users. Substituting (14) for \(z_i^k\) in the z-update, we obtain the final version of the first algorithm.

**Distributed ADMM Algorithm 1.** Initialize \(\{x_i^0\}, \{z_i^0\}, \{u_i^0\}\). For \(k = 0, 1, \ldots,\) repeat

1. **z-update:** Each facility \(i\) solves the following sub-problem for \(x_i^{k+1}\):

\[
\begin{align*}
\min_{x_i} & \quad -f_i(x_i) + \rho/2\|x_i - x_i^k + \bar{u}_i^k - z_i^{k+1}\|^2 \\
\text{s.t.} & \quad x_i \in \mathcal{X}_i.
\end{align*}
\]

2. **z-update:** Each facility \(j\) solves the following sub-problem for \(z_j^{k+1}\):

\[
\begin{align*}
\min_{z_j} & \quad g_j(N\bar{z}_j) + (N\rho/2)\|z_j - \bar{z}_j^k - u_j^k\|^2 \\
\text{s.t.} & \quad N\bar{z}_j \in \mathcal{Y}_j.
\end{align*}
\]

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

\[
u_j^{k+1} := \bar{u}_j^k + x_j^{k+1} - z_j^{k+1}.
\]

Next, we switch the order of the x-update and z-update in the scaled form of ADMM, obtaining our second algorithm:

\[
z^{k+1} := \arg\min_{z \in \mathcal{Y}} \left(\sum_{i=1}^N \left(\sum_{i=1}^N \|z_i - x_i^k - u_i^k\|^2\right) + \rho/2\|z - \bar{z}\|^2\right)
\]

\[
x_i^{k+1} := \arg\min_{x_i \in \mathcal{X}_i} \left(-f_i(x_i) + \rho/2\|x_i - x_i^k + \bar{u}_i^k - z_i^{k+1}\|^2\right)
\]

\[
u_i^{k+1} := \bar{u}_i^k + x_i^{k+1} - z_i^{k+1}.
\]

Similarly, the second and last steps can be implemented in parallel, and the z-update can be handled in the same way as before. The final version of the second algorithm is as follows.

**Distributed ADMM Algorithm 2.** Initialize \(\{x_i^0\}, \{z_i^0\}, \{u_i^0\}\). For \(k = 0, 1, \ldots,\) repeat

1. **z-update:** Each facility \(j\) solves the following sub-problem for \(z_j^{k+1}\):

\[
\begin{align*}
\min_{z_j} & \quad g_j(N\bar{z}_j) + (N\rho/2)\|z_j - \bar{z}_j^k - u_j^k\|^2 \\
\text{s.t.} & \quad N\bar{z}_j \in \mathcal{Y}_j.
\end{align*}
\]

2. **z-update:** Each facility \(i\) solves the following sub-problem for \(x_i^{k+1}\):

\[
\begin{align*}
\min_{x_i} & \quad -f_i(x_i) + \rho/2\|x_i - x_i^k + \bar{u}_i^k - z_i^{k+1}\|^2 \\
\text{s.t.} & \quad x_i \in \mathcal{X}_i.
\end{align*}
\]

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

\[
u_j^{k+1} := \bar{u}_j^k + x_j^{k+1} - z_j^{k+1}.
\]

Here, we would like to point out that these two algorithms have different strengths and naturally complement each other, which will be made clear in Section 4.
3.3 Parallel Implementation

The distributed nature of the above algorithms allows for efficient parallel implementation in the cloud that has abundant server resources. Here, we briefly discuss several issues pertaining to such implementations. We focus on the first algorithm, since the same efficient parallel implementation in the cloud that has abundant server resources is the same.

We associate each user a type-1 processor, which stores and maintains two states \((x_i^k, x^k - z^k + \bar{u}^k)\). Similarly, we associate each facility a type-2 processor, which stores and maintains \((\bar{u}^k_j, \bar{x}^{k+1}_j)\). At the kth iteration, each type-1 processor solves a small convex problem (in \(n\) variables), and then reports the updated \(x^{k+1}_i\) to facility \(j\). Each facility \(j\) collects \(x^{k+1}_i\) from all type-1 processors, and then computes the average \(\bar{x}^{k+1}_j\). This is called a reduce step in parallel computing [11]. After the reduce step, each type-2 processor solves a single-variable convex problem and updates \(\bar{u}^{k+1}_j\). Then, each type-2 processor sends the value of \(\bar{x}^{k+1}_j - \bar{z}^{k+1}_j + \bar{u}^{k+1}_j\) to all type-1 processors, which is called a broadcast step. In actual implementation, a server can host multiple processors of the same type. For example, one can use one server to host all type-2 processors and a number of other servers to host multiple type-1 processors. This enables us to further reduce the message-passing overhead, since a server hosting type-1 processors can report the local sum to the server hosting type-2 processors, and the server hosting type-2 processors needs only to send one copy of \(\bar{x}^{k+1}_j - \bar{z}^{k+1}_j + \bar{u}^{k+1}_j\) to every other server.

An alternative and perhaps much simpler method to implement Algorithm 1 is based on the MPI Allreduce operation [38], 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) often leads to much better performance. With the help of Allreduce, we only need \(N\) processors of the same type, with each storing and maintaining three states \((x_i^k, \bar{u}^k, \bar{x}^k)\). At the kth 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 \(\bar{x}^{k+1}\). After this Allreduce step, each processor solves \(n\) single-variable convex problems and (redundantly) computes \(\bar{u}^{k+1}\). This method simplifies our implementation and often helps to increase the speed.

For practical implementation, our distributed algorithms can be terminated even before convergence is achieved. This is a feature of ADMM, as it usually finds a reasonably good solution within just tens of iterations [7]. So, an early-braking mechanism can be safely incorporated into our algorithms, making them appealing for a wide range of applications.

3.4 Comparisons with Other Algorithms

Here, we compare our distributed algorithms with other possible algorithms. We begin with the dual-decomposition algorithm for problem (1).

\textbf{Dual Decomposition Algorithm.} Initialize \(\{x^n_i\}, \{y^n_j\}, \{\lambda^n_j\}\).

For \(k = 0, 1, \ldots\), repeat

1. \textit{x-update:} Each user \(i\) solves the following sub-problem for \(x^{k+1}_i\):
   \[
   \begin{align*}
   &\min_{x_i \in \mathcal{X}_i} \quad -f_i(x_i) + (\lambda^k)^T x_i \\
   &\text{s.t.} \quad x_i \in \mathcal{X}_i.
   \end{align*}
   \]

2. \textit{y-update:} Each facility \(j\) solves the following sub-problem for \(y^{k+1}_j\):
   \[
   \begin{align*}
   &\min_{y_j \in \mathcal{Y}_j} \quad g_j(y_j) - \lambda^k y_j \\
   &\text{s.t.} \quad y_j \in \mathcal{Y}_j.
   \end{align*}
   \]

3. \textit{Dual update:} Each facility \(j\) updates \(\lambda^{k+1}_j\):
   \[
   \lambda^{k+1}_j := \lambda^k_j + \rho \left( \sum_{i=1}^N x_{ij}^{k+1} - y^{k+1}_j \right),
   \]
   where \(\rho^k\) is the step-size for the \(k\)th iteration.

At every iteration, each user in the dual decomposition solves an \(n\)-variable convex optimization, and each facility solves a single-variable optimization. Hence, the computational cost of dual decomposition is essentially the same as that of our distributed algorithms for each iteration. The \(x\)-update requires each user to know the value of \(\lambda^k\), which can be achieved through a broadcast step. The dual-update requires each facility to know the sum \(\sum_{i=1}^N x_{ij}^{k+1}\), which can be achieved through a reduce step. Hence, the message-passing overhead of dual decomposition is the same as that of our algorithms at each iteration. Since our algorithms achieve faster convergence, they enjoy lower overall computational complexity and lower message-passing overhead.

On the other hand, dual decomposition usually requires delicate adjustments of step sizes \(\rho^k\), resulting in slow convergence; dual decomposition generally requires the cost functions \(g_j(\cdot)\) to be strictly convex and the utility functions \(f_i(\cdot)\) to be strictly concave. In contrast, our distributed algorithms do not suffer from these two difficulties. As we will show in Sec. 5.4, for solving the geographical load balancing problem (2), dual decomposition does not converge after hundreds of iterations, while our algorithms converge after 50 iterations.

There are some other ADMM-type distributed algorithms in the literature, such as linearized ADMM [21] and multi-block ADMM [20, 24]. 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:

\[
\begin{align*}
   x^{k+1}_i &:= \arg\min_{x_i \in \mathcal{X}_i} \left( -f_i(x_i) + x_i^T g^k + \frac{\rho}{2} \|x_i - x_i^k\|^2 \right) \\
   y^{k+1}_j &:= \arg\min_{y_j \in \mathcal{Y}_j} \left( g_j(y_j) + \frac{\rho}{2} \left( \sum_{i=1}^N x_{ij}^{k+1} - u^k_j \right)^2 \right) \\
   u^{k+1}_j &:= u^k_j + \frac{N}{\rho} \sum_{i=1}^N x_{ij}^{k+1} - y^{k+1}_j,
\end{align*}
\]

where \(g^k = \rho(\sum_{i=1}^N x_i^k - y^k + u^k)\) linearizes the penalty term \((\rho/2)\|x_i - y_i\|^2\), and \((\rho/2)\|x_i - x_i^k\|^2\) is a proximal term. Although the above algorithm admits simple parallel implementation, its convergence requires \(\rho > \rho^N\). When \(N\) is sufficiently large, the \(x\)-update in each iteration just slightly changes \(x_i\) (due to a large \(\rho\)), 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 general convex objective functions. Indeed, a counter-example has just been reported showing the impossibility of convergence for multi-block ADMM [9].

During the final stage of this paper, we noticed a very similar work [40], which also applies the standard two-block ADMM algorithm to solve multi-block convex problems by using auxiliary
variables. Although the algorithms proposed in [40] can solve more general problems, they require the utility functions to be strictly concave and the cost functions to be strictly convex in order to achieve $O(1/k)$ rate of convergence. Such requirements cannot be met in some scenarios. For example, the utility function $f_i(A_i, x_i)$ in backbone traffic engineering is non-strictly concave even if $f_i$ itself is strictly concave, as we discussed before. We also noticed that our first algorithm is in spirit the same as the algorithm proposed in [7, Chapter 7] for solving the sharing problem. The main differences include the stopping rule and the analysis of convergence rates. As we will show in Sec. 4, our stopping rule is based on rigorous analysis, whereas their stopping rule is mainly based on some heuristic principles. Moreover, our analysis of convergence rates in Sec. 4 reveals some potential weakness of Algorithm 1, which motivates us to develop Algorithm 2. In contrast, the need for Algorithm 2 is not recognized in [7], mainly due to the lack of such analysis.

Compared to previous ADMM algorithms developed in the networking context in [41–43], the algorithms proposed here assume weaker technical assumptions to ensure convergence, and have lower computational complexity and lower message-passing overhead. For example, the algorithm in [41] requires strongly convex objective functions and bounded level set in order to achieve convergence (see Theorem 1 in [41]). In contrast, our algorithms converge with arbitrary convex objective functions. Moreover, the previous algorithm [41] needs to solve a large-scale quadratic problem at each iteration, whereas our algorithms only involve small-scale subproblems.

4. CONVERGENCE ANALYSIS

In this section, we study the convergence behavior of our distributed algorithms. Our analysis is based on several very recent results on ADMM, and leads to a simple yet rigorous stopping rule.

4.1 Assumptions

We first present the assumptions based on which our algorithms converge.

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

ASSUMPTION 2. The utility functions $f_i: \mathbb{R}^n \to \mathbb{R}$ are concave, and the cost function $g_j: \mathbb{R} \to \mathbb{R}$ is convex.

ASSUMPTION 3. The constraints $\{X_i\}$ and $\{Y_j\}$ are bounded polyhedral sets.

Assumptions 1 and 2 are rather mild. In particular, the utility functions $f_i(\cdot)$ need not to be strictly concave, and the cost functions $g_j(\cdot)$ need not to be strictly convex. Assumption 3 is satisfied in all of our previous problem settings.

The above three assumptions imply that strong duality holds for problem (1) (see, e.g., Proposition 5.2.1 of [3]). Since a feasible solution of problem (1) is also a feasible solution of problem (13), strong duality also holds for problem (13).

REMARK 1. Assumption 3 is only needed for the proof of strong duality. Hence, Assumption 3 can be replaced by any conditions on $\{X_i\}$ and $\{Y_j\}$ so long as strong duality holds. Such conditions are usually weaker than Assumption 3 (see, e.g., [7]).

4.2 $O(1/k)$ Rate of Convergence

Although the global convergence of ADMM has been extensively studied, only very recently it has been proved that ADMM has an $O(1/k)$ rate of convergence [21]. Based on this result, we are able to establish $O(1/k)$ rate of convergence for our distributed algorithms and provide a simple stopping rule.

Let $\{(x_i^k), (z_i^k)\}$ be a primal optimal solution to problem (13) (in particular, we have $x_i^k = z_i^k$), and $\{\lambda_i^k\}$ be a dual optimal solution. Let $u_i^k = \lambda_i^k / p_i$. Their existence follows from the strong duality theorem. We have the following results.

THEOREM 1. Let $\{(x_i^k), (z_i^k), (a_i^k)\}$ be any sequence generated by the distributed algorithm 1. Let

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

and

$$D^k = \sum_{i=1}^N \left( \|x_{i+1}^k - z_i^k\|^2_2 + \|u_{i+1}^k - u_i^*\|^2_2 \right).$$

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

THEOREM 2. Let $\{(x_i^k), (z_i^k), (a_i^k)\}$ be any sequence generated by the distributed algorithm 2. Let

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

and

$$D^k = \sum_{i=1}^N \left( \|x_{i+1}^k - x_i^k\|^2_2 + \|u_{i+1}^k - u_i^*\|^2_2 \right).$$

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

We now outline the proofs for the above theorems. By symmetry, we need only to prove one of them, say, Theorem 2. For simplicity of notation, we rewrite the problem (13) as

$$\begin{array}{ll}
\text{minimize} & f(x) + h(z) \\
\text{subject to} & x - z = 0 \quad x \in \mathcal{X}, \; z \in \mathcal{Z},
\end{array}$$

where $z = (z_1, \ldots, z_N)^T$, and $h(z) = g(\sum_{i=1}^N z_i)$. Note that $h(\cdot)$ is still a convex function (because affine mappings preserve convexity), and $\mathcal{Z}$ is still a polyhedral set. The scaled form of ADMM for problem (20) (with reversed $x$-update and $z$-update) is

$$\begin{align}
& z^{k+1} := \arg\min_{z \in \mathcal{Z}} \left( h(z) + (\rho/2)\|z - x^k - u^k\|^2_2 \right) \\
& x^{k+1} := \arg\min_{x \in \mathcal{X}} \left( f(x) + (\rho/2)\|x - z^{k+1} + u^k\|^2_2 \right) \\
& u^{k+1} := u^k + x^{k+1} - z^{k+1},
\end{align}$$

which is equivalent to our distributed algorithm 2. Clearly, we have $V^k = \|x^k - x^*\|^2_2 + \|u^k - u^*\|^2_2$, and $D^k = \|x^{k+1} - x^k\|^2_2 + \|u^{k+1} - u^k\|^2_2$. Our proof consists of three steps.

Step 1: $V^k$ is a Lyapunov function. This step can be established by showing the inequality

$$V^{k+1} \leq V^k - D^k.$$

Since $D^k \geq 0$ for all $k$, this states that $V^k$ is indeed a Lyapunov function. The proof of inequality (24) is provided in the Appendix.
Step 2: $D^k$ is non-increasing. This step has been established in [21] through some matrix manipulations. (See the proof of Theorem 4.1 in [21] for details).

Step 3: $D^k$ converges at rate $O(1/k)$. Using (24) and the fact that $D^k$ is non-increasing, we have
\[
V^0 \geq \sum_{t=0}^{k} D^t + V^{k+1}
\]
\[
\geq (k+1)D^k.
\]

Hence, we have $D^k \leq V^0/(k+1)$. This completes the proof of Theorem 2.

Remark 2. The above theorems suggest that the sequence $\{D^k\}$ can be used as a natural stopping rule for our distributed algorithms, which decreases at rate $1/k$. This stopping rule is more rigorous compared to that in [7], since their stopping rule is mainly based on heuristic principles. In particular, their stopping-rule sequence does not enjoy the non-increasing property and may fluctuate over iterations.

Note that our stopping rule can be easily implemented using the MPI Allreduce operation. For example, for a fixed $k$, we have $D^k = \sum_{i=1}^{N} \|z_i^{k+1} - z_i^k\|_2 + N\|\bar{u}^{k+1} - \bar{u}^k\|_2$. Hence, at each iteration, all the processors can perform an Allreduce operation to obtain $\sum_{i=1}^{N} \|z_i^{k+1} - z_i^k\|_2$ and then compute $\|\bar{u}^{k+1} - \bar{u}^k\|_2$ locally. In actual implementation, this Allreduce operation can be combined with the existing Allreduce operation (which is used to obtain $\bar{x}^{k+1}$) so that there is only one such operation for each iteration.

4.3 $O(1/a^k)$ Rate of Convergence

We next prove $O(1/a^k)$ rate of convergence for our distributed algorithms under certain additional assumptions. We need to introduce two definitions.

Definition 1. (Strong Convexity). A function $f: \mathbb{R}^n \to \mathbb{R}$ is strongly convex with constant $\nu > 0$, if $f(x) - \frac{\nu}{2} \|x\|^2$ is convex. A function $f$ is strongly concave if $-f$ is strongly convex.

Definition 2. (Lipschitz Continuous Gradient). A function $f: \mathbb{R}^n \to \mathbb{R}$ has a Lipschitz continuous gradient $\nabla f$ with constant $\kappa > 0$, if for all $x_1, x_2 \in \mathbb{R}^n$,
\[
\|\nabla f(x_1) - \nabla f(x_2)\|_2 \leq \kappa \|x_1 - x_2\|_2.
\]

We have the following results.

Theorem 3. Let $\{\{x_i^0\}, z^0, \bar{u}^0\}$ be any sequence generated by the distributed algorithm 1. Let $V^k$ be the Lyapunov function defined in (16). Assume that the cost functions $g_i(\cdot)$ are strictly convex with Lipschitz continuous gradients. Then starting with any initial point $\{\{x_i^0\}, z^0, \bar{u}^0\}$, there exists some $\delta > 0$ such that $V^k \leq V^0/(1+\delta)^k$ for all $k$.

Remark 3. The above theorems suggest that the distributed algorithm 1 is better suited for the case when only the cost functions are strictly convex, and the algorithm 2 is better suited for the case when only the utility functions are strictly concave. In this sense, our two algorithms have different strengths and complement each other in a natural way, as summarized in the Table 1 below.

In addition, these two theorems help us to choose the step-size $\rho$. In particular, one can show that the step-size $\rho$ can be chosen such that the parameter $\delta$ is maximized.

Table 1: Comparison of two algorithms.

<table>
<thead>
<tr>
<th>case</th>
<th>strictly convex</th>
<th>Lipschitz continuous</th>
<th>recommendation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>none</td>
<td>none</td>
<td>Algorithm 1 or 2</td>
</tr>
<tr>
<td>2</td>
<td>$(g_j)$</td>
<td>$(g_j)$</td>
<td>Algorithm 1</td>
</tr>
<tr>
<td>3</td>
<td>$(-f_i)$</td>
<td>$(f_i)$</td>
<td>Algorithm 2</td>
</tr>
<tr>
<td>4</td>
<td>$(-f_i), (g_j)$</td>
<td>$(f_i), (g_j)$</td>
<td>Algorithm 1 or 2</td>
</tr>
</tbody>
</table>

5. 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 1 in Table 1 since (5) is non-strictly concave and (6) is non-strictly convex. Thus it can be solved using either Distributed ADMM Algorithm 1 or 2. We use Algorithm 1 in all of our simulations. Note that if the objective function exhibits strict convexity, better simulation results can be obtained according to Theorem 3 and 4. 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.

5.1 Setup

We randomly generate each user’s request demand $t_i$, with an average of $9 \times 10^3$. 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 $\sum_j c_j = 1.4x$ the total demand. We use the 2011 annual average day-ahead on peak prices [15] at 10 different local markets as the

---

Note that there is a typo in Table 1.1 of [12], in which $Q \succ 0$ should be $Q \succeq 0$.
power prices $P_i$ for data centers. The servers have peak power $P_{peak} = 200$ W, and consume 50% power at idle. The PUE is 1.5. These numbers represent state-of-the-art data center hardware [14, 37].

We set the penalty parameter $\rho$ 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 $\rho$ can further improve the performance of our algorithms, we confine ourselves to the above 9 choices to demonstrate the practicality.

5.2 Convergence and Scalability

We 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.

Convergence of objectives functions. Figure 3 and 6 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 $N = 10^3$ 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 7 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 sub-linearly, which confirms Theorem 1 for the $O(1/k)$ 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 5 and 8 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 $10^3$ for $N = 10^2$ (or $10^5$ 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 $10^3$. Hence, we conclude that the constraints are well satisfied after 50 iterations in both cases.

5.3 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 $x_i^{k+1} := x_i^k$.

Figure 9–11 plot the convergence with different failure probabilities for $N = 10^2$, and Figure 12–14 for $N = 10^4$. Specifically, Figure 9 and 12 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, and further illustrated by the overlapping curves in Figure 10, 11, 13, and 14 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.

5.4 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 $\rho^k$ is chosen following the commonly accepted diminishing step size rule [6], with $\rho^k = 10^{-7}/\sqrt{k}$.

We plot the trajectory of objective values in Figure 15, and that of primal residuals in Figure 16. 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 16 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.

Figure 15: Objective value. Figure 16: Primal residual. $N = 10^2$. $N = 10^4$.

This phenomenon is due to the oscillation problem [28] 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 $(\rho_1, \rho_2)$.

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.

6. RELATED WORK

6.1 Network Utility Maximization

Network utility maximization (NUM) [4, 39] 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 [26], and has lead to a rich literature on distributed algorithms for network
Figure 3: Objective value. $N = 10^2$.

Figure 4: $D^k$. $N = 10^2$.

Figure 5: Primal residual. $N = 10^2$.

Figure 6: Objective value. $N = 10^4$.

Figure 7: $D^k$. $N = 10^4$.

Figure 8: Primal residual. $N = 10^4$.

Figure 9: Relative errors in objective value. $N = 10^2$.

Figure 10: $D^k$. $N = 10^2$.

Figure 11: Primal residual. $N = 10^2$.

Figure 12: Relative errors in objective value. $N = 10^4$.

Figure 13: $D^k$. $N = 10^4$.

Figure 14: Primal residual. $N = 10^4$. 
rate control [10, 32, 36] and new understandings of existing network protocols [31]. Despite its popularity, dual decomposition suffers from slow convergence, and generally 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. Another advantage is that our algorithms can easily handle multi-path routing, whereas dual decomposition requires non-trivial modifications to address multi-path routing [28].

6.2 ADMM and Its Variations

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 [7]. While the convergence of ADMM is well known in the literature (see, e.g., [5, 7]), its rate of convergence has only been established very recently. [21] proves rate-\(O(1/k)\) of convergence under the most general assumptions. [12] proves rate-\(O(1/a^3)\) of convergence under the assumptions that the objective function is strongly convex and its gradient is Lipschitz continuous in at least one block of variables. These results provide theoretical foundation for our algorithm design and analysis. ADMM has two important variations: linearized ADMM [21] and multi-block ADMM [20, 24]. However, they are not particularly suitable for problem (1), as discussed thoroughly in Section 3.4. In contrast, our ADMM-type algorithms exploit the structure of problem (1), thereby enjoying a number of unique advantages. Our algorithms are in spirit similar to a recent submission [40] and the algorithm proposed in [7, Chapter 7]. Still, our algorithms have clear advantages as discussed in Section 3.4.

6.3 Cloud Traffic Management

Cloud service providers operate two distinct types of WANs: user-facing WANs and backbone WANs [25]. 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, 29, 30, 37, 41–43]. The backbone WAN provides connectivity among data centers for data replication and synchronization. Rate control and multi-path routing [18, 23, 25] 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.

7. 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 developed two distributed algorithms that are amenable to parallel implementation. We have studied the convergence rates of our algorithms under various scenarios. When the utility function is non-strictly concave and the cost function is non-strictly convex, our algorithms achieve \(O(1/k)\) rate of convergence. When the utility function is strictly concave or the cost function is strictly convex, our algorithms achieve \(O(1/a^3)\) rate of convergence. Our analysis also provides a simple yet rigorous stopping rule as well as a guideline on how to choose the step-size \(\rho\).

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 production systems.

8. REFERENCES

if and only if

The proof of Lemma 1 is standard and thus omitted here.

Let $p^k = f(x^k) + h(z^k)$, and $p^*$ denote the optimal value for problem (20). The first inequality is

$$p^* - p^{k+1} \leq (\lambda^*)^T (u^{k+1} - u^k).$$

Proof: Since $(x^k, z^k)$ and $\lambda^*$ is a primal-dual optimal solution pair, by the Saddle Point Theorem, we have

$$f(x^*) + h(z^*) \leq f(x^{k+1}) + h(z^{k+1}) + (\lambda^*)^T (x^{k+1} - z^{k+1}).$$

Using $u^{k+1} = u^k + x^{k+1} - z^{k+1}$, the right-hand side is $p^{k+1} + (\lambda^*)^T (u^{k+1} - u^k)$. This gives (25).

The second inequality is

$$p^{k+1} - p^* \leq -\rho (u^{k+1} - u^k)^T (u^{k+1} - u^k) + \rho (x^* - x^{k+1})^T (x^* - x^{k+1} + u^k).$$

Using $u^{k+1} = x^{k+1} - z^{k+1} + u^k$, we obtain

$$f(x^{k+1}) \leq f(x^*) + \rho (x^* - x^{k+1})^T u^{k+1}.$$  

A similar argument gives

$$h(z^{k+1}) \leq h(z^*) + \rho (z^* - z^{k+1})^T (z^* - z^{k+1} - u^{k+1}).$$

Adding the two inequalities above, we obtain

$$p^{k+1} - p^* \leq \rho (u^{k+1} - u^k)^T u^{k+1} + \rho (x^* - x^{k+1})^T (x^* - x^{k+1} - z^{k+1} + u^k).$$

Note that

$$z^* - z^{k+1} = z^* - z^{k+1} + x^{k+1} - z^{k+1} = z^* - x^{k+1} + u^k.$$  

This gives (26).

Adding (25) and (26), and regrouping terms gives

$$(x^* - x^{k+1})^T (u^{k+1} - u^k) + (z^* - z^{k+1})^T (z^* - z^{k+1} - u^k) + (x^* - x^{k+1})^T (x^* - x^{k+1} - z^{k+1} + u^k).$$

Recall that $x^{k+1}$ minimizes $f(x) + \rho x^T u^{k+1}$ and $z^{k+1}$ minimizes $f(x) + \rho x^T u^k$, we have

$$f(x^{k+1}) - f(x^*) + \rho (x^{k+1} - x^*)^T u^{k+1} \leq 0$$

and

$$f(x^*) - f(x^{k+1}) + \rho (z^* - z^{k+1})^T u^k \leq 0.$$  

Adding the two inequalities above gives

$$(x^{k+1} - x^*)^T (u^{k+1} - u^k) \leq 0.$$  

Substituting this into (29), we have

$$(x^* - x^{k+1})^T (u^{k+1} - u^k) + (z^* - z^{k+1})^T (z^{k+1} - u^k) \geq 0.$$  

Note that

$$\|x^* - x^{k+1}\|^2 = \|(z^* - x^{k+1}) + (x^{k+1} - x^*)\|^2$$

$$= \|x^* - x^{k+1}\|^2 + \|x^{k+1} - x^*\|^2$$

$$+ 2(x^* - x^{k+1})^T (x^{k+1} - x^*).$$

Similarly,

$$\|u^* - u^{k+1}\|^2 = \|u^* - u^{k+1}\|^2 + \|u^{k+1} - u^k\|^2$$

$$+ 2(u^* - u^{k+1})^T (u^{k+1} - u^k).$$

Combining the above three results, we obtain

$$V^k = V^{k+1} + \|x^{k+1} - x^k\|^2 + \|u^{k+1} - u^k\|^2$$

$$+ 2(x^* - x^{k+1})^T (x^{k+1} - x^*)$$

$$+ 2(u^* - u^{k+1})^T (u^{k+1} - u^k)$$

$$\geq V^{k+1} + \|x^{k+1} - x^k\|^2 + \|u^{k+1} - u^k\|^2.$$  

This proves (24).

APPENDIX

Proof of inequality (24)

We need a technical lemma and two simple inequalities.

**Lemma 1.** Let $h_1, h_2 : \mathbb{R}^n \to \mathbb{R}$ be two convex functions. Assume that $h_2$ is differentiable in $\mathbb{R}^n$, and that $X \subseteq \mathbb{R}^n$ is a closed convex set. Then

$$x^* \in \text{argmin}_{x \in X} h_1(x) + h_2(x)$$

if and only if

$$x^* \in \text{argmin}_{x \in X} h_1(x) + x^T \nabla h_2(x^*).$$

The proof of Lemma 1 is standard and thus omitted here.